How We Learned to Stop Worrying and Build a RISC-V VP with only one Microcode Instruction*

Lucas Klemmer, Sonja Gurtner, Daniel Große
Institute for Complex Systems, Johannes Kepler Universität Linz, Linz, Österreich
lucas.klemmer@jku.at, sonja.gurtner@gmail.com, daniel.grosse@jku.at

Abstract

In this extended abstract, summarizing [1] and [2], we present *Goldcrest-VP* a *Virtual Prototype* (VP) which serves as an exploration platform for microcoded RISC-V cores leveraging the *One Instruction Set Computer* (OISC) principle. Furthermore, we introduce a formal verification framework for the microcode procedures. Using Goldcrest-VP, we developed SUBLEQ microcode that is fully RISC-V RV32I compliant. We were able to uncover several bugs in the microcode using our formal verification framework.

1 Introduction

The idea to reduce the number of instructions of a *Reduced Instruction Set Computer* (RISC) to the minimum, i.e. to a single instruction, led to the ultimate RISC computer or *One Instruction Set Computer* (OISC) [3, 4]. The careful selection of the instruction makes an OISC Turing-complete, i.e. it can solve any computing problem. Depending on the selected instruction, three OISC types can be distinguished:

1. *Arithmetic-based architectures:* This type uses an arithmetic operation and a conditional jump. A well-known example is to subtract and branch unless positive, abbreviated as `SUBLEQ` [5].
2. *Bit-manipulating architectures:* These machines perform bit operations, like bit-flipping or copying, and pass the control flow either conditionally or unconditionally. Due to their extreme minimalism they are the least practical relevant OISC type [6].
3. *Transport triggered architectures:* While the only available instruction is `MOVE`, arithmetic, control flow, or other operations are available by writing to memory mapped registers; see e.g. [7, 8].

The strengths of OISC machines lie in their extremely small area footprint and their high flexibility wrt. very specific use cases. Thus, OISC machines have been considered in a wide variety of applications, e.g. fault detection [9, 10], cryptography [11], stream processing [12], carbon nanotube computer [13], and (advanced) micro-controllers [14, 8]. However, programming directly in OISC languages is extremely complicated due to the limited features resulting in exceptionally long programs that are hard to debug and maintain. Further, most of these architectures require customized software tooling, in particular from the compiler side. This is due to the fact that typically each OISC machine uses a custom *Instruction Set Architecture* (ISA), thus hindering the emergence of a common OISC ecosystem. Yet, the availability of a mature and widely used hardware and software ecosystem is one of the deciding factors for the success of most technologies. Therefore, we have chosen RISC-V as the outside interface of the VP since it is an open and royalty-free ISA [15] with a lot of traction in industry and research.

An often used OISC instruction is *SUbtract and Branch if Less than or EQual to zero* (SUBLEQ) [5]. While SUBLEQ is not as low-level as the bit-manipulating instructions, it is still challenging to write efficient and correct SUBLEQ procedures thus making it mandatory to very thorougly verify the microcode procedures.

In this extended abstract, we present *Goldcrest-VP*, an exploration platform for microcoded RISC-V cores leveraging the OISC principle and a formal verification framework for the microcode procedures.

1.1 Microcoded RISC-V VP: Goldcrest-VP

Goldcrest-VP [1][1] has been implemented in approximately $1{,}000$ lines of plain C++ code. The current microcode implementation(s) support RV32I, although the *Instruction Set Simulator* (ISS) itself features all requirements to support additional RISC-V extensions. An overview of the architecture of Goldcrest-VP is shown in Figure 1. The ISS core has been structured into four main components: (1) the main memory, (2) the RISC-V interface, (3) the OISC execution unit, and (4) the timing model. In Figure 1 and the rest of this work all components (or data flows) related to RISC-V are colored in blue, and all components (or data flows) related to the OISC microcode are colored in red. In the following sections, we describe all components in more detail.

[1] https://github.com/ics-jku/goldcrest-vp

*This work has partially been supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.

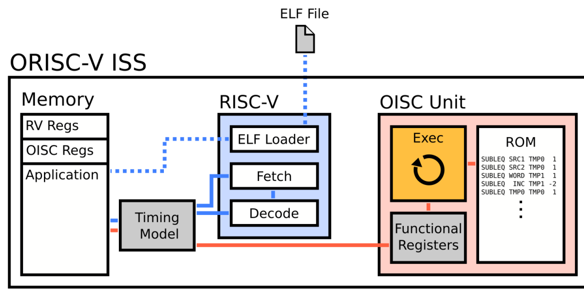

Figure 1 Architecture of the Goldcrest-VP

1.2 RISC-V Interface

To the user, Goldcrest-VP is a RISC-V compliant ISS, completely hiding the microcode layer. Internally, the *RISC-V interface* (center of Figure 1 with blue background) handles loading RISC-V binaries, and fetching plus decoding RISC-V instructions. Before the ISS starts executing a RISC-V binary, it is loaded into the main memory using the ELF loader (see blue dotted line). Then, the ELF loader sets the initial RISC-V program counter and starts decoding the first RISC-V instruction. After the RISC-V interface has decoded an RISC-V instruction, it sets up the OISC execution unit by first putting register values and immediate values of the RISC-V instruction at hand into the OISC registers. It then looks up the address of the microcode implementation for the current RISC-V instruction and hands over the execution to the OISC unit. After the OISC unit finishes its computation, the result, if one is produced, is loaded from the OISC registers and stored back at the RISC-V destination register. Then, the next RISC-V instruction is processed.

1.3 OISC Execution Unit

The OISC execution unit (right side in Figure 1) implements the execution loop of the OISC microcode[2]. During execution, microcode instructions are fetched from a *very small* microcode ROM (less than 1 KB), executed, and then their results are written back to the OISC registers. Moreover, microcode instructions can perform additional operations (e.g. bit-operations) by writing to memory-mapped *functional registers*. Using these registers developers can integrate new operations and evaluate their performance impact. By this, Goldcrest-VP allows a hybrid OISC design space exploration of a mixture of arithmetic-based and transport triggered OISC architectures.

1.4 Memory

Often, OISC architectures are designed without dedicated hardware registers, i.e. all registers are placed in the system's main memory. While this greatly reduces the system's complexity and area requirements, it also has a high performance cost, since accessing memories is highly sequential and slow compared to dedicated registers. The Goldcrest-VP reflects this common feature of OISC architectures by mapping all RISC-V and OISC registers into the main memory[3].

1.5 Timing Model

To enable design space exploration of different architectures (e.g. dedicated registers vs. registers in memory) every access to the memory is routed through the timing model. In Figure 1, the timing model is located between the memory and the RISC-V interface. In the timing model developers can specify the time required to access different registers or memory regions. The timing model adds a delay to the simulation for every read or write operation on the main memory and can be customized by the user. To summarize, the timing model enables early design space exploration thus helping developers deciding on the hardware architecture.

2 Formal Verification Framework

In this section we present the formal verification framework we developed to verify the correctness of the microcode procedures of the Goldcrest-VP [2][4].

2.1 Overview

The scope of the formal verification is the microcode. This means, that our verification starts after the RISC-V Framework placed the correct register values in the OISC Registers and stops after the microcode procedure is done by checking that the microcode produced the correct results.

Our formal verification framework is based on a SUBLEQ ISS written in the *Rosette* [16, 17, 18] programming language. *Rosette* [16, 17, 18] is a framework for designing solver-aided domain-specific languages and an extension of *Racket*. In practice, Rosette's core verification part is a wrapper around the SMT-LIB2 language, therefore most operations can be easily translated to SMT-LIB2. The SUBLEQ ISA is extremely minimal, therefore the Racket ISS consists of just a single function (`step`) that totals about 30 lines of code which implements the SUBLEQ instruction.

The formal verification is performed by symbolically evaluating a microcode procedure and checking that the results always match the results of the specifications of the RISC-V instruction the procedure implements. In the next section we will briefly describe how RISC-V instructions are specified for verification in our framework.

2.2 RISC-V Specifications

To formally capture the specification of each RISC-V instruction we have defined the macro `rv-verify`. This macro generates Racket code which handles the formal verification, verification runtime measurement, and the

[2] As already mentioned we use `SUBLEQ` microcode instructions, but Goldcrest-VP can easily be extended to other OISC instructions.

[3] Except the OISC program counter.

[4] https://github.com/ics-jku/goldcrest-microcode-verification

concrete execution of models, if a model is found. Listing 1 shows how the `rv-verify` macro is used for the `JAL` RISC-V instruction.

```
1  (rv-verify
2   #:name "JAL"
3   #:init-pc JAL-PC
4   #:fuel 20
5   #:microcode microcode
6   #:solver (boolector)
7   #:spec
8     (lambda (res)
9       (and (eq? (list-ref res REG-RVPC)
10                (bvadd val-rvpc val-immi))
11           (eq? (list-ref res REG-RSLT)
12                (bvadd val-rvpc (bv 4 XLEN)))))
13  #:assumptions
14    (lambda (res)
15      (assume (eq? (bv 0 2)
16                   (extract 1 0 (list-ref res 1))))))
```

Listing 1 Full specification of the JAL instruction

Our `rv-verify` macro is responsible for executing the respective SUBLEQ instruction, printing the result after execution, verifying that the specification holds, and providing information about the runtime of the verification. The specification and assumptions have to be passed in a lambda expression as they would otherwise be evaluated immediately. This immediate evaluation is not possible as both refer to register values *after* execution which are not known to the macro at the time of definition. The macro first symbolically executes the instructions starting at the defined program counter, and saves the resulting state of all registers. The selected SMT solver then tries to find concrete values for the defined symbolic constants which would lead to a violation of the specification. If no such counterexample is found, the correctness is proven and the function finally returns OK and the runtime for the verification. Otherwise, it returns a model, which consists of the register values that led to a wrong result after executing the instructions, and prints the incorrect output.

2.3 Verification Results

We were able to prove the correctness of SUBLEQ microcode procedures for the majority of the RV32I instructions (28 out of 37). However, for 9 RISC-V instructions the corresponding SUBLEQ procedures were buggy.

Most instructions could be proven in a matter of seconds, however for some of the more complex instructions (i.e. and, xor, ...) we had to apply some tricks to bring down the verification time. These tricks included first further optimizing the procedures and scaling the bitwidth of the microcode registers.

3 Conclusion

In this extended abstract, we presented Goldcrest-VP which serves as a development platform for both, hardware architecture and microcode procedures, and provides the basis for early design space exploration. Additionally, we presented a formal verification framework for the Goldcrest-VP microcode. Our VP is a fully compliant implementation of the RV32I ISA, whose instructions are executed in microcode using the OISC paradigm. But still, even tough the VP passes all RISC-V architectural tests we found some bugs in the microcode with the formal verification framework.

4 Literature

[1] L. Klemmer and D. Große, "An exploration platform for microcoded RISC-V cores leveraging the one instruction set computer principle," in *ISVLSI*, 2022, pp. 38–43.

[2] L. Klemmer, S. Gurtner, and D. Große, "Formal verification of SUBLEQ microcode implementing the RV32I ISA," in *FDL*, 2022, pp. 1–8.

[3] F. Mavaddat and B. Parhami, "URISC: The ultimate reduced instruction set computer," *IJEEE*, vol. 25, no. 4, pp. 327–334, 1988.

[4] D. W. Jones, "The ultimate RISC," *SIGARCH Computer Architecture News*, vol. 16, no. 3, p. 48–55, Jun. 1988.

[5] O. Mazonka and A. Kolodin, "A simple multi-processor computer based on subleq," in *arXiv:1106.2593*, 2011.

[6] O. Mazonka, "Bit copying - the ultimate computational simplicity," in *arXiv:0907.2173.2593*, 2009.

[7] P. Jaaskelainen, A. Tervo, G. P. Vaya, T. Viitanen, N. Behmann, J. Takala, and H. Blume, "Transport-triggered soft cores," in *IPDPS*, 2018, pp. 83–90.

[8] M. Crepaldi, A. Merello, and M. Di Salvo, "A multi-one instruction set computer for microcontroller applications," *IEEE Access*, vol. 9, pp. 113 454–113 474, 2021.

[9] S. Ananthanarayan, S. Garg, and H. D. Patel, "Low cost permanent fault detection using ultra-reduced instruction set co-processors," in *DATE*, 2013, p. 933–938.

[10] A. Rajendiran, S. Ananthanarayanan, H. D. Patel, M. V. Tripunitara, and S. Garg, "Reliable computing with ultra-reduced instruction set co-processors," in *DAC*, 2012, p. 697–702.

[11] K. S. Dharshana, K. Balasubramanian, and M. Arun, "Encrypted computation on a one instruction set architecture," in *ICCPCT*, 2016, pp. 1–6.

[12] M. Yokota, K. Saso, and Y. Hara-Azumi, "One-instruction set computer-based multicore processors for energy-efficient streaming data processing," in *RSP*, 2017, p. 71–77.

[13] M. M. Shulaker, G. Hills, N. Patil, H. Wei, H.-Y. Chen, H.-S. P. Wong, and S. Mitra, "Carbon nanotube computer," *Nature*, vol. 501, no. 7468, pp. 526–530, 2013.

[14] Maxim Integrated, "Maxq microcontroller family," https://www.maximintegrated.com/en/design/technical-documents/userguides-and-manuals/5/5618.html, 2013.

[15] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2019.

[16] "Rosette guide," https://docs.racket-lang.org/rosette-guide/index.html, Accessed: 2022-05-20.

[17] E. Torlak and R. Bodík, "A lightweight symbolic virtual machine for solver-aided host languages," in *PLDI*, 2014, pp. 530–541.

[18] ——, "Growing solver-aided languages with Rosette," in *SPLASH*, 2013, pp. 135–152.