Formal Verification of SUBLEQ Microcode implementing the RV32I ISA

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

***Abstract*—The open and royalty free nature as well as the extendable design of the RISC-V Instruction Set Architecture (ISA) has lead to a sprawling ecosystem of RISC-V software and hardware. One of the domains explored by the RISC-V community are processors with minimal area footprints. To reduce the area footprint to the minimum, typically performance is traded for a much more compact design. A promising approach to realizing very small RISC-V processors is to base them on a single instruction, such as SUBLEQ, and using a microcode layer. However, the minimalism of SUBLEQ makes writing correct microcode procedures challenging.**

In this paper, we target the formal verification of SUBLEQ microcode procedures. We present our verification framework and show that we can handle complex SUBLEQ procedures in practical times. In our experiments we consider a set of SUBLEQ procedures which implements the RV32I ISA and passes all official RISC-V compliance tests. However, based on our approach we found 9 intricate bugs in the SUBLEQ procedures.

I. Introduction

The potential of the free and open *Instruction Set Architecture* (ISA) RISC-V [1] has been seen very early in research, the open-source community and a few companies. The same is now also true for big industry players, considering for instance the recent move of Intel in joining the RISC-V foundation and launching an innovation fund which will support chip designs using the RISC-V ISA.

Besides the mentioned openness, major advantages of RISC-V include its modularity, extensibility and the community-driven ecosystem. Due to the implementation-independent specification of the RISC-V ISA, RISC-V processors for targeting many use cases are developed, i.e. ranging from high-performance to micro-controllers which are very important for the IoT domain. In the latter, strict constraints on energy-consumption and area are inevitable. To reduce the area footprint to the minimum, typically performance is traded for a much more compact design.

A very promising approach in this direction, and seen from the *Reduced Instruction Set Computer* (RISC) paradigm an extreme case, is an instruction set with a single instruction only. These are the so-called *One Instruction Set Computers* (OISCs). On the lowest-level examples include bit-manipulating architectures [2] which perform operations such as bit-flipping or copying of individual bits and are therefore complicated and difficult to develop. More practical OISC architectures include transport-triggered architectures that work using only the *MOVE* instruction. They perform arithmetic, control flow, or other operations by writing to special memory-mapped memory locations [3], [4].

An often used OISC instruction from the class of arithmetic-based OISC instructions 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.

Recently, an OISC exploration platform including SUBLEQ microcode procedures implementing the RISC-V ISA (more precisely RV32I) has been presented in [6]. In addition, multiple OISC hardware configurations have been evaluated in terms of different standard software benchmarks. The RISC-V compliance of the microcode procedures was checked by running the official RISC-V test-suite consisting of more than 12,000 individual tests. However, this allows bug-hunting only and not a proof of correctness for the microcode procedure implementing a concrete RISC-V instruction.

In this paper, we present a formal verification framework for SUBLEQ microcode implementing RV32I. Our framework uses Rosette [7], [8], a solver-aided programming language. Rosette extends Racket [9] which is a modern functional programming language in the Lisp and Scheme family and very well suited to implement ideas quickly. Rosette's extensions include language constructs for program synthesis and verification via symbolic variables, and expressing logical constraints on those variables. Finally, these constraints can be solved using *Satisfiability Modulo Theories* (SMT) solvers.

In our approach, we derive a formal execution model for SUBLEQ microcode from a SUBLEQ *Instruction Set Simulator* (ISS) created in Racket. First, we describe the extensions required to make the ISS "formal-verification ready" leveraging Rosette. Then, we present the proposed formal verification framework. In the last step, we create RISC-V specifications for each RV32I instruction. All together this allows automated formal verification, i.e. to prove the correctness of SUBLEQ microcode procedures.

A particular challenge of microcode verification is that many of the microcode procedures implementing the RV32I instructions use loops. This makes formal reasoning hard since these microcode procedures exhibit complex control flow that requires a deep unrolling resulting in a non-trivial search space. We propose two techniques to overcome this challenge: With the first technique we optimize the microcode and with the second we scale down the size of the bit-vectors without sacrificing correctness.

978-1-6654-7332-3/22/$31.00

In the experiments we demonstrate the effectiveness of our verification framework. Overall, we consider 37 SUBLEQ microcode procedures from [6] implementing RV32I. In 9 of the procedures we found intricate bugs that were not known before.

Finally, we provide our formal model, the verification framework and the SUBLEQ microcode as open-source on GitHub[1].

The paper is structured as follows: Section II discusses related work. The preliminaries are given in Section III. The proposed formal verification framework for SUBLEQ microcode is introduced in Section IV. Then, Section V provides the experimental results. Finally, the paper is concluded in Section VI.

II. Related Work

Formal verification of processors has a very long history. Important work goes back to 1989 published in [10], followed by verification of pipelined control [11], handling of advanced data-path operations [12], improving scalability [13], HW/SW co-verification [14], [15], complete formal verification [16]–[19], verification of security aspects like for instance transient execution attacks [20] or memory protection [21], to just name a few.

The particular focus on microcode verification has come into focus in [22], [23]. The tool MicroFormal uses SAT/SMT techniques for backward compatibility checks and assertion-based verification. Another direction allowing to interconnect RTL and microcode verification by including theorem proving and SAT has been presented in [24]. Recent enhancements of this work has been published in [25]. While our approach share some conceptual similarities with the just mentioned works, none of them targeted the formal verification of SUBLEQ microcode implementing RV32I instructions.

Finally, we have to mention the Serval framework [26]. Serval uses Rosette and provides an extensible infrastructure to create verifiers by lifting interpreters. The lifting principle is somewhat similar to our approach of extending the SUBLEQ ISS for formal verification in Rosette.

III. Preliminaries

A. The SUBLEQ Instruction

Since it is quite intuitive, SUBLEQ is one of the best studied OISC instructions.

The SUBLEQ instruction is a three-operand instruction, performing a combined subtraction and branching operation. The semantic of `SUBLEQ A B C` is as follows:

$$\begin{aligned} r &\leftarrow reg[B] - reg[A] \\ reg[B] &\leftarrow r \\ pc &\leftarrow \begin{cases} pc + C, & \text{if } r \leq 0 \\ pc + 1, & \text{otherwise} \end{cases} \end{aligned}$$

First, the result r is calculated by subtracting the register value in A from the register value in B. The result is written back in register B and a jump is performed depending on the value of r: If r is smaller or equal to 0, C is added to the SUBLEQ program counter, else the SUBLEQ program counter is incremented by 1.

To ease the development of SUBLEQ procedures we implemented a SUBLEQ file format and an assembler that translates assembled SUBLEQ programs to multiple output formats, e.g. Racket lists or Verilog memory dumps. This assembler resolves jump labels to concrete offsets and also creates definitions for the entry points of all SUBLEQ procedures. For the *JAL* RISC-V instruction, we show a SUBLEQ procedure in this format in the following example.

Example 1. *Even though the* `SUBLEQ` *instruction only provides a way to subtract one number from another, complex RISC-V instructions, such as* **JAL***, can be implemented. Listing 1 shows the SUBLEQ procedure for the* JAL *instruction. The* `jal rd, imm` *instruction performs a* Jump And Link *operation, i.e. adding the immediate value* `imm` *to the current program counter and storing the current program counter* $+4$ *in the destination register* `rd`*. At the start of the execution, the* JAL *procedure expects the immediate value* `imm` *to be present in SUBLEQ register* **IMM**[2]*. First, the procedure resets the temporary registers* **TMP0** *and* **TMP1** *as well as the result register* **RSLT** *to* 0 *by subtracting them from themselves (Lines 2-4). Since this is a common operation in SUBLEQ code, it can be abbreviated by writing just one register in a line using our assembler. It is important to reset the registers since they still can contain garbage data from previous procedures which would corrupt the results of the current procedure.*

The procedure first handles computing the link address (RISC-V pc $+4$*). For this, the current RISC-V program counter in* **RVPC** *has to be copied to the* **RSLT** *register. To achieve this, the* **RVPC** *is negated and written into* **TMP0** *by subtracting it from register* **TMP0** *(which contains a* 0 *at this point, Line 5). Next, the pc is negated again and copied into the* **RSLT** *register (Line 6). Register* **NEXT** *contains the constant* -4 *which is subtracted from the* **RSLT** *register to calculate the link address (Line 7).*

What remains now is to add the immediate value containing the jump offset to the RISC-V pc. This is done similarly by first negating the value and subtracting this negated value from the **RVPC** *register (Line 8). Since this is the last operation of this procedure, the jump target is set to* **END** *which stops the SUBLEQ execution (Line 9).*

```
1  @JAL
2    TMP0           ; reset TMP0
3    TMP1           ; reset TMP1
4    RSLT           ; reset RSLT
5    RVPC TMP0      ; -pc => TMP1
6    TMP0 RSLT      ; pc => SRC2
7    NEXT RSLT      ; add link address +4
8    IMM  TMP1      ; negate IMM
9    TMP1 RVPC END  ; PC - -TMP0 link1
```

Listing 1: SUBLEQ implementation of the JAL instruction

[1] https://github.com/ics-jku/riscv-subleq-verification

[2] This will be managed later by the RISC-V interface.

B. Rosette

Rosette [7], [8], [27] 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. In particular Rosette is typed at runtime and, critically for this work, does not support unbounded loops but only loops with a fixed bound (like the bit length). Those loops are simply unfolded before querying an underlying SMT solver. Currently, Rosette supports the three solvers Z3 [28], CVC4 [29], and Boolector [30]. Rosette translates assertions and assumptions to constraints and passes them to the SMT solver which is able to identify counterexamples or prove that the assertions hold.

IV. SUBLEQ Formal Verification Framework

This section introduces the formal microcode verification framework. First, Section IV-A describes the verification setting in terms of executing SUBLEQ microcode for RISC-V instructions. Second, Section IV-B introduces the RISC-V Interface which acts as an abstraction layer above the hardware. Section IV-C introduces the ISS which serves as the base for our formal model. Section IV-D describes the extensions to the ISS which lead to the formal model of the framework. Section IV-E shows how RISC-V instructions can be specified in our framework. Finally, Section IV-F presents techniques that reduce the verification runtime.

A. Verification Setting

For the verification of SUBLEQ procedures we make some assumptions about the hardware on which the microcode runs. First, we target an RV32I configuration with 32-bit word size and support for the basic I extension of RISC-V (integer instructions). Second, the hardware on which the microcode runs handles some RISC-V specific operations such as decoding RISC-V instructions and passing the correct arguments to the SUBLEQ procedures. We assume that the hardware implementing these operations, which is called *RISC-V Interface*, and which is presented in Section IV-B, is correct. Hence, we are only verifying the SUBLEQ microcode procedures themselves. Consequently, our specifications assume that the correct arguments are present in the SUBLEQ registers and the SUBLEQ program counter is set to the start of the correct SUBLEQ procedure. The verification of a RISC-V Interface implementation remains for future work.

B. The RISC-V Interface as an Abstraction Layer

The *RISC-V Interface* handles RISC-V related operations such as decoding instructions or extending immediate values. By this, an abstraction layer is provided that handles operations that would be very hard to implement purely in SUBLEQ microcode. Additionally, the RISC-V Interface makes SUBLEQ code portable between different hardware implementations.

The SUBLEQ instructions can only access 16 microcode registers (which will be presented in Section IV-C2) and can not access the RISC-V registers or the main memory. This is instead done by the RISC-V Interface in the execution loop shown in Algorithm 1. First, the RISC-V Interface fetches and decodes a RISC-V instruction and places the instruction arguments in the SUBLEQ registers. Then, the SUBLEQ program counter is set to point to the SUBLEQ procedure implementing the decoded RISC-V instruction and the control is handed over to the SUBLEQ procedure. After the SUBLEQ procedure signals termination, the RISC-V Interface reads the result from the SUBLEQ registers and writes it to the correct RISC-V destination register. Finally, load and store operations are executed (if necessary) and the loop starts over with the next RISC-V instruction.

Algorithm 1 Microcoded RISC-V Execution Cycle

```
loop
    instr ← mem[rvpc]
    decode(instr)
    subleq-regs ← decoded-arguments
    while not subleq-done do
        run subleq
    riscv-regs[decoded-rd] ← subleq-result
    if isLoad(instr) then
        riscv-regs ← mem[result]
    else if isStore(instr) then
        mem[result] ← riscv-regs
```

C. SUBLEQ ISS

Our formal verification framework is based on a SUBLEQ ISS written in the Racket programming language. The SUBLEQ ISA is extremely minimal, therefore the Racket ISS consists of just a single function (`step`, Listing 2) that totals about 30 lines of code which implements the SUBLEQ specification from Section III-A. Please note, that Listing 2 also contains some of the extensions to derive the formal model which we describe below in Section IV-D. These extensions are not part of the ISS and are highlighted using red color.

1) Step Function: The `(step pc regs code)` function fetches one SUBLEQ instruction at position `pc` from the microcode in `code` (Line 4). Please note, that Rackets `(list-ref xs i)` function returns the `i'th` element from list `xs`. Next, the *src1*, *src2*, and *jump* components of the instruction are extracted from the previously fetched SUBLEQ instruction (Lines 5-9). With the register addresses decoded, the values are read from the registers addressed by *src1*, and *src2* (Lines 10-15). At this point, the values are sign extended to $XLEN + 1$ bits to overcome overflow errors which can impact the correctness of the SUBLEQ procedures [6]. The constant XLEN specifies the register bit-width of all registers and arithmetic operations. We verify RV32I instructions therefore XLEN is 32. However, we later show how XLEN can be adjusted to decrease the verification runtime of long-running SUBLEQ procedures. The main part of the SUBLEQ instruction, the subtraction, is implemented in Line 16. Next, the result is written back into the registers (Lines 18-22). The step function is implemented as a pure function without

side effects. Therefore, a new list of registers `new-regs` is constructed which is identical to the list `regs` except in the place *src2* which holds the result of the subtraction. Racket's `for/list` function implements a *for each* loop with the addition that this function returns a list containing the results of all loop iterations. Subsequently, the next program counter is calculated in Lines 24-27. The `bvsle` function performs a signed less-or-equal operation on the bitvector arguments. Finally, the `step` function is called recursively with the previously computed new pc and register values (Line 30). However, if the end of the procedure is reached, which is indicated by the EXIT constant in the jump offset (Line 29), the resulting register values in `new-regs` are returned (Line 31) and the SUBELQ execution stops.

```
1  (define (step fuel pc regs code)
2    (cond [(= fuel 0) "unrolling depth too small"]
3          [else
4           (define instr (list-ref-bv code pc))
5           (define src1 (extract 15 12 instr))
6           (define src2 (extract 11 8 instr))
7           (define jump
8             (sign-extend (extract 7 0 instr)
9                          (bitvector 9)))
10          (define val1
11            (sign-extend (list-ref-bv regs src1)
12                         (bitvector (+ 1 XLEN))))
13          (define val2
14            (sign-extend (list-ref-bv regs src2)
15                         (bitvector (+ 1 XLEN))))
16          (define res (bvsub val2 val1))
17
18          (define new-regs
19            (for/list [(index (range 16))]
20              (if (bveq (integer->bitvector index
                           (bitvector 4)) src2)
21                  (extract (- XLEN 1) 0 res)
22                  (list-ref regs index))))
23
24          (define new-pc
25            (if (bvsle res (bv 0 (+ 1 XLEN)))
26                (bvadd pc jump)
27                (bvadd pc (bv 1 9))))
28
29          (if (bveq jump EXIT)
30              new-regs
31              (step (- fuel 1) new-pc new-regs code))]))
```

Listing 2: Step function of the verification ready model

2) SUBLEQ Registers: SUBLEQ instructions can access 16 SUBLEQ registers. Table I lists all 16 SUBLEQ registers which contain the arguments to the procedure as well as space for temporary variables and some constant values required by the SUBLEQ procedures. The registers can be grouped into four categories: (1) operands for passing the RISC-V operands, (2) temporary registers, (3) constants that store frequently needed values, such as -1 (in register **INC**) to increment by one, or -4 (in register **NEXT**) to increment the RISC-V program counter, and (4) functional registers for additional hardware support of specific operations. Our microcode uses no special hardware functions, so these registers serve as two additional temporary registers.

D. Formal SUBLEQ Execution Model

This section describes the ISS extensions which we introduced to obtain the formal model. These extensions include the usage of symbolic input values (Section IV-D1), assumptions on the input values (Section IV-D2), and the inclusion of a maximum unroll depth of the model (Section IV-D3).

TABLE I: The 16 SUBLEQ registers visible to the microcode

Operands	Temporary	Constants	Functional
SRC1	TMP0	ONE	FUNC0/TMP6
SRC2/RSLT	TMP1	TWO	FUNC1/TMP7
IMM	TMP2	WORD	
	TMP3	INC	
	TMP4	NEXT	
	TMP5		

1) Symbolic Input: The first extension to the ISS concerns the input values which are the initial register contents. While the registers contain concrete values in an ISS use case, they must contain symbolic values in a formal setting. Listing 3 showcases how symbolic registers can be created using Rosette's `define-symbolic` function. In our model, the register file is implemented as a list of registers. Therefore, the symbolic variables are created with the `define-symbolic` function (Lines 1-2) before they are inserted into a list (Lines 4-5).

```
1  (define-symbolic
2    val-tmp0 val-rvpc ... val-tmp7 (bitvector XLEN))
3
4  (define init-regs
5    (list val-tmp0 val-rvpc ... val-tmp7))
```

Listing 3: Initialization of the symbolic registers

2) Constant Assumption: Since all registers are symbolic variables we must make sure that the variables for the constants (e.g. **INC** or **NEXT**) always contain the correct values. Therefore, we use the Rosette's `assume` function to force the register values to take the constant values at the very start of the model before the first call to *step*. The `assume` function reduces the search space by dropping all variable assignments that violate the assumptions. Listing 4 shows the call to Rosette's `assume` function to initialize the correct constants in the initial register values. `REG-ONE`, `REG-WORD`, `REG-INC`, and `REG-NEXT` are definitions of the internal register numbers for the registers **ONE**, **WORD**, **INC**, and **NEXT** respectively. The assumption condition is a conjunction of multiple equivalence checks (via Racket's **eq?** comparison function) each testing if the register holds the correct value. Since we use Rosette on problems with bounded arithmetic, the values are specified using the `(bv v w)` function. This function constructs a bitvector representation of value `v` using `w` bits.

```
1  (assume
2   (and
3    (eq? (list-ref init-regs REG-ONE)  (bv 1 XLEN))
4    (eq? (list-ref init-regs REG-WORD) (bv (- XLEN 1)
         XLEN))
5    (eq? (list-ref init-regs REG-INC)  (bv -1 XLEN))
6    (eq? (list-ref init-regs REG-NEXT) (bv -4 XLEN))))
```

Listing 4: The constant registers are set using the Rosettes assume function

3) Unroll depth: During formal verification the model is unrolled depending on the number of executed SUBLEQ

instructions. Rosette does not support unbounded loops and unbounded recursion, therefore we have to explicitly specify the maximum number of unrollings. This unroll depth is one of the major factors for the verification runtime. Therefore, minimizing the unroll depth is of utmost importance. Since we know the worst case run-times of all SUBLEQ procedures an upper bound for the unroll depth can be determined. We integrate this unroll depth into the model by providing an additional argument *fuel* (Line 1 in Listing 2). In other words, this argument specifies how many SUBLEQ instructions can still be executed until the result of the specified RISC-V instruction is placed in the **RSLT** register. At each recursive call to `step` the fuel argument is decremented. If the initial fuel parameter was chosen too small, the procedure terminates with a respective error message (Line 2 in Listing 2) since the SUBLEQ microcode procedure did not meet the specification of the RISC-V instruction at hand. Now, the model is ready to be used for verification with Rosette. The last remaining piece to a full verification setup are the RISC-V specifications which we will describe in the following section.

E. 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 concrete execution of models, if a model is found. Listing 5 shows how the `rv-verify` macro is used for the `JAL` RISC-V instruction. Before we discuss the concrete definitions for `JAL`, we explain the general meaning of the 7 parameters used in the macro and the general task of the macro in the following.

1) `rv-verify` parameters:

Name The name of the RISC-V instruction which is implemented by the SUBLEQ microcode procedure.

PC The microcode program counter for fetching the first SUBLEQ instruction of the RISC-V instruction at hand.

Fuel The maximum number of unrollings.

Microcode The microcode containing all SUBLEQ instructions.

Solver The SMT solver used during verification.

Spec The specification of the RISC-V instruction which defines the functional behavior that must hold on all valid inputs. For branching operations, the only specification is the correct adaption of the program counter (pc+offset in case of a jump, pc+4 otherwise). For other instructions, it ensures that the result of a calculation is correct and stored in the right register. The registers for pc and the result are the SUBLEQ registers RVPC and RSLT, respectively. For both the position in the memory is known and therefore we can access their content via the respective named index (see Line 9 and Line 11).

Assumptions Assumptions that restrict valid inputs for some instructions. For example, the shift amount for shifting instructions is limited to 5 bits (≤ 31).

2) `rv-verify` general task: 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 (Line 8-16). 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.

3) `rv-verify` for `JAL`: Listing 5 shows the specification for SUBLEQ procedure of the RISC-V instruction `JAL`. As it is a J-Type instruction, the program counter needs to be adjusted by the offset to the instruction that should be executed next (RVPC=RVPC+IMM), and the previous program counter gets incremented by 4 and written into the destination register RSLT. For this specification we assume in Line 13 that the `JAL` instruction gets a valid value for the program count, namely 2-bit aligned (lowest two bits are 0). All registers contain bitvectors; therefore the behavior of the addition is defined with `bvadd`, one of Rosette's bitvector operations. The equality of the content of the RSLT register and the desired result is checked using the `eq?` predicate, which returns true if two values or their contents are equal.

```
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 5: Full specification of the JAL instruction

F. Optimization Techniques

A large part of the RV32I instructions can be formally verified with our framework. By using a formal approach, we were able to identify microcode bugs which were not uncovered by testing and the RISC-V architectural test suite. However, some procedures showed runtimes of much more

than 12 hours which we set as time limit. For these procedures we had to introduce special optimizations to make them complete in under 12 hours.

1) Microcode Optimization: The procedures with long runtimes are exclusively procedures that have internal loops that lead to very deep unroll depths. Especially the number of instructions inside the internal loops of microcode procedures contributes to the required unroll depth. Minimizing the number of instructions inside those internal loops is especially effective since they are multiplied by the XLEN and thus even one instruction less reduces the unroll depth by up to 32. Therefore, as a first measure to reducing the verification runtime, we spend significant effort in optimizing the procedures. For some instructions, we were able to remove more than 10 instructions which, assuming $XLEN = 32$, reduces the required unroll-depth already by ≥ 320.

2) Register Bit-Width: All procedures that can not be verified given our time limit share the property that their unrolling depth depends on the bit-width of the registers. For example, logical operations such as *XOR* have to perform multiple SUBLEQ instructions in a loop for each register bit. Therefore, decreasing the bit-width (XLEN) of the formal model directly reduces the required unroll depth. By this, we were able to verify the *AND*, *OR*, and *XOR* instructions and their immediate counterparts.

We argue that in the case of these logical instructions all bits are independent. Therefore, instead of proving correctness for 32 bits we can safely scale down the size of the bitvectors.

V. Verification Results

In this section, we present the results for formal verification of SUBLEQ procedures implementing RV32I. Our experiments have been carried out on an Intel Core i7-10700 with 64 GB of main memory. For SMT-solving we used Boolector [30] with the timeout set to 12 hours (43,200 seconds).

In the following, we evaluate the verification runtime of all microcode procedures (Section V-A), describe one found bug in detail (Section V-B), and analyze how the verification runtime scales with increasing word lengths (Section V-C).

A. Overall Results

Table II shows the verification results for all RISC-V instructions. Column **Instruction** provides the name of the RISC-V instruction that is implemented by the corresponding SUBLEQ procedure. Please note that one SUBLEQ procedure can implement multiple RISC-V instructions, e.g. the microcode for ADD and ADDI is the same as the only difference is that the second source is either the RISC-V register rs2 or the immediate `imm` which is handled by the RISC-V Interface by making the respective value in both cases available in the SUBLEQ register **`IMM`**. Column **#SUBLEQ Instrs** gives the number of SUBLEQ instructions needed to implement the RISC-V instruction. Then, Column **Unrolling depth** reports how often we had to unroll the `step` function for the proof.

TABLE II: Verification results for all RISC-V instructions

Instruction	#SUBLEQ Instrs	Unroll depth	Result	Runtime (s)
LB	4	4	Pass	0.02
LH	4	4	Pass	0.02
LW	4	4	Pass	0.02
LBU	4	4	Pass	0.02
LHU	4	4	Pass	0.02
SB	4	4	Pass	0.02
SH	4	4	Pass	0.02
SW	4	4	Pass	0.02
LUI	4	4	Pass	0.02
ADD	4	4	Pass	0.02
ADDI	4	4	Pass	0.02
SUB	6	6	Pass	0.04
AUIPC	8	8	Pass	0.02
JAL	8	8	Pass	0.03
JALR	16	16	Fail	0.03
BEQ	11	11	Pass	2.76
BNE	11	11	Pass	2.30
BLT	6	6	Pass	3.79
BGE	5	5	Pass	4.35
BLTU	20	485	Fail	1,163.00
BGEU	28	617	Fail	1.26
SLT	6	6	Pass	2.07
SLTI	6	6	Pass	2.07
SLTU	29	618	Fail	165.26
SLTIU	29	618	Fail	165.26
XOR*	26	314	Pass	15,183.00
XORI*	26	314	Pass	15,183.00
OR*	25	297	Pass	13,107.00
ORI*	25	297	Pass	13,107.00
AND†	24	234	Pass	41,352.00
ANDI†	24	234	Pass	41,352.00
SLL	8	132	Pass	280.00
SLLI	8	132	Pass	280.00
SRL	23	550	Fail	14.84
SRLI	23	550	Fail	14.84
SRA	28	555	Fail	14.84
SRAI	28	555	Fail	14.84

* XLEN=17 † XLEN=15

Some SUBLEQ procedures are loop-free (e.g. ADDI or BLT) therefore the model has to be unrolled exactly as many times as the number of SUBELQ instructions in the corresponding microcode procedure. Other instructions are more complex and must be implemented using loops in the SUBLEQ procedure (e.g. XORI or SLL). For these procedures the number of unrollings must be individually calculated. For example, the **XORI** instruction consists of 7 initial instructions that are always executed just once followed by a loop consisting of 18 instructions. The loop is executed for each bit of the arguments (i.e. XLEN times). After the loop, a single instruction which increments the RISC-V program counter is always executed. Overall, the maximum unroll-depth for $XLEN = 32$ is therefore $8+18*XLEN = 8*18*32 = 584$.

The verification result is provided in Column **Result** and the runtime in seconds is reported in Column **Runtime**. Finally, instructions for which we had to scale down the size of the

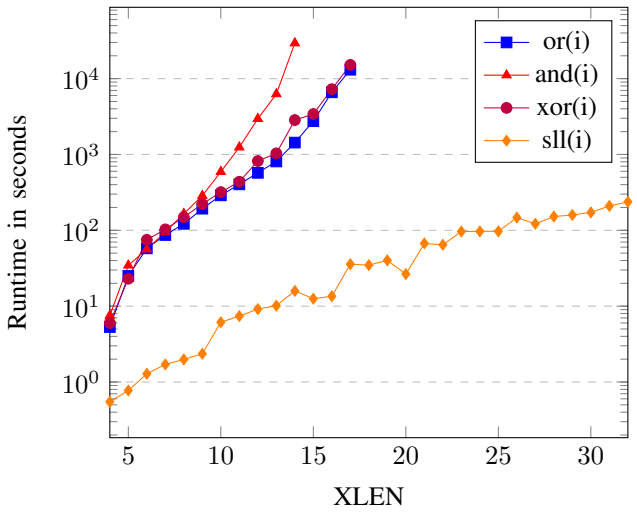

Fig. 1: Verification runtime with increasing XLEN

bitvectors to get practical runtimes are marked by $\star$ and †. For example, since operations on single bits (e.g. logical and) are hard to implement using SUBLEQ instructions only, we had to reduce the bit-width for the *AND(I)*, *OR(I)*, and *XOR(I)* procedures. This reduced the required depth for unrolling significantly.

Overall, as can be seen 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.

B. Example Bug

One of the bugs we found using our framework concerns the correctness of the *JALR* instruction. This instruction adds an immediate to the source register, then has to clear the LSB of the sum, and then jumps to the resulting address. However, our SUBLEQ microcode procedure for *JALR* does not set the LSB to zero and thus invalid program counter values can be reached. The problem occurs when the sum of the source register and the immediate value is odd, which is possible since *JALR* uses the I immediate type and the source register can also hold odd values.

C. Effect of Word Lengths

Fig. 1 shows the runtimes for verifying the SUBLEQ procedures implementing *AND(I)*, *OR(I)*, *XOR(I)*, and *SLL* for various word lengths (i.e. XLEN values). As can be seen, *AND(I)*, *OR(I)*, and *XOR(I)* show exponential runtimes with increasing XLEN. Starting from around $XLEN = 10$ the runtime is increasing significantly for these two microcode procedures with *AND(I)* reaching runtimes of multiple hours from $XLEN = 13$ on. *SLL* on the other hand scales much better with the full $XLEN = 32$ verification taking only 237 seconds. Please note, the runtime differences between the logical and instruction *AND(I)* and the other two instructions *XOR(I)* and *OR(I)*. Despite nearly identical unroll depths the runtime of *AND(I)* scales much worse than *XOR(I)* and *OR(I)*.

VI. Conclusion

In this paper, we presented a verification framework for SUBLEQ microcode procedures implementing the RV32I ISA; the microcode procedures together with an OISC exploration platform have been introduced in [6]. We created our formal model for SUBLEQ procedures in Rosette, a solver-aided programming language. This allowed us to extend an already existing ISS for SUBLEQ procedures with constraints capturing (a) the execution of a single SUBLEQ instruction symbolically and (b) the formalization of the RISC-V instruction specifications in terms of lambda-based macros. Moreover, we presented two techniques to tackle the verification complexity of long-running microcode procedures. Our methodology uncovered multiple microcode bugs which were not found by official RISC-V compliance tests.

In future work, we plan to investigate different SMT solvers as well as advanced techniques such as predicate abstraction. Moreover, we are working on an FPGA-core which implements the RISC-V interface and runs the verified SUBLEQ microcode. For the full verification of the core we will leverage novel approaches such as [31] and we will analyze different metrics using [32], [33].

Acknowledgments

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

References

[1] 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.
[2] O. Mazonka, "Bit copying - the ultimate computational simplicity," in *arXiv:0907.2173.2593*, 2009.
[3] 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.
[4] 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.
[5] O. Mazonka and A. Kolodin, "A simple multi-processor computer based on subleq," in *arXiv:1106.2593*, 2011.
[6] L. Klemmer and D. Große, "An exploration platform for microcoded RISC-V cores leveraging the one instruction set computer principle," in *ISVLSI*, 2022.
[7] E. Torlak and R. Bodík, "A lightweight symbolic virtual machine for solver-aided host languages," in *PLDI*, 2014, pp. 530–541.
[8] ——, "Growing solver-aided languages with Rosette," in *SPLASH*, 2013, pp. 135–152.
[9] "Racket language," https://racket-lang.org, Accessed: 2022-05-01.
[10] W. A. Hunt, "Microprocessor design verification," *J. Autom. Reason.*, vol. 5, no. 4, pp. 429–460, 1989.
[11] J. R. Burch and D. L. Dill, "Automatic verification of pipelined microprocessor control," in *Computer Aided Verification*, 1994, pp. 68–80.
[12] R. Kaivola and K. R. Kohatsu, "Proof engineering in the large: formal verification of pentium 4 floating-point divider," *STTT*, vol. 4, no. 3, pp. 323–334, 2003.
[13] R. Kaivola, "Formal verification of pentium 4 components with symbolic simulation and inductive invariants," in *CAV*, 2005, p. 170–184.

[14] D. Große, U. Kühne, and R. Drechsler, "Hw/sw co-verification of embedded systems using bounded model checking," in *GLSVLSI*, 2006, pp. 43–48.

[15] M. D. Nguyen, M. Wedler, D. Stoffel, and W. Kunz, "Formal hardware/-software co-verification by interval property checking with abstraction," in *DAC*, 2011, pp. 510–515.

[16] J. Bormann, S. Beyer, A. Maggiore, M. Siegel, S. Skalberg, T. Blackmore, and F. Bruno, "Complete formal verification of Tricore2 and other processors," in *DVCon*, 2007.

[17] D. Große, U. Kühne, and R. Drechsler, "Analyzing functional coverage in bounded model checking," *TCAD*, vol. 27, no. 7, pp. 1305–1314, Jul. 2008.

[18] F. Haedicke, D. Große, and R. Drechsler, "A guiding coverage metric for formal verification," in *DATE*, 2012, pp. 617–622.

[19] K. Devarajegowda, M. R. Fadiheh, E. Singh, C. W. Barrett, S. Mitra, W. Ecker, D. Stoffel, and W. Kunz, "Gap-free processor verification by s^2qed and property generation," in *DATE*, 2020, pp. 526–531.

[20] M. R. Fadiheh, J. Müller, R. Brinkmann, S. Mitra, D. Stoffel, and W. Kunz, "A formal approach for detecting vulnerabilities to transient execution attacks in out-of-order processors," in *DAC*, 2020, pp. 1–6.

[21] D. Gao and T. Melham, "End-to-end formal verification of a risc-v processor extended with capability pointers," in *FMCAD*, 2021, pp. 24–33.

[22] T. Arons, E. Elster, L. Fix, S. Mador-Haim, M. Mishaeli, J. Shalev, E. Singerman, A. Tiemeyer, M. Y. Vardi, and L. D. Zuck, "Formal verification of backward compatibility of microcode," in *CAV*, 2005, pp. 185–198.

[23] A. Franzén, A. Cimatti, A. Nadel, R. Sebastiani, and J. Shalev, "Applying SMT in symbolic execution of microcode," in *FMCAD*, 2010, pp. 121–128.

[24] J. Davis, A. Slobodová, and S. Swords, "Microcode verification - another piece of the microprocessor verification puzzle," in *ITP*, 2014, pp. 1–16.

[25] S. Goel, A. Slobodová, R. Sumners, and S. Swords, "Verifying x86 instruction implementations," in *CPP*, 2020, pp. 47–60.

[26] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang, "Scaling symbolic evaluation for automated verification of systems code with serval," in *SOSP*, 2019, p. 225–242.

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

[28] L. M. de Moura and N. S. Bjørner, "Z3: an efficient SMT solver," in *TACAS*, 2008, pp. 337–340.

[29] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *CAV*, 2011, pp. 171–177.

[30] A. Niemetz, M. Preiner, C. Wolf, and A. Biere, "Btor2, BtorMC and Boolector 3.0," in *CAV*, 2018, pp. 587–595.

[31] L. Klemmer and D. Große, "EPEX: processor verification by equivalent program execution," in *GLSVLSI*, 2021, pp. 33–38.

[32] ——, "WAL: a novel waveform analysis language for advanced design understanding and debugging," in *ASP-DAC*, 2022, pp. 358–364.

[33] ——, "Late breaking results: Waveform-based performance analysis of RISC-V processors," in *DAC*, 2022.