An Exploration Platform for Microcoded RISC-V Cores leveraging the One Instruction Set Computer Principle

Lucas Klemmer Daniel Große
Institute for Complex Systems, Johannes Kepler University Linz, Austria
{ lucas.klemmer,daniel.grosse }@jku.at

***Abstract*—In this work, we present an exploration platform for microcoded RISC-V cores leveraging the *One Instruction Set Computer* (OISC) principle. Following the industry-proven virtual prototyping approach, we have realized our exploration platform by implementing an extensible and configurable *Instruction Set Simulator* (ISS). The developed ORISCV-ISS combines the advanced ecosystem of RISC-V with the ultimate minimalism of OISCs. ORISCV-ISS serves as development platform for both, hardware architecture and microcode procedures, and provides the basis for early design space exploration.**

Using ORISCV-ISS, we developed `SUBLEQ` microcode that is fully RISC-V compliant and ready to be run on real hardware. We evaluate how multiple hardware configurations and OISC extensions affect the performance, providing key information to balance between area savings and system performance.

I. 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) [1], [2]. 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` [3].
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 [4].
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. [5], [6].

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 [7], [8], cryptography [9], stream processing [10], carbon nanotube computer [11], and (advanced) micro-controllers [12], [6]. However, programming directly in the 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.

In modern industry-proven design flows *Virtual Prototypes* (VPs) are created to enable the parallel development of hardware and software in combination with design space exploration [13], [14], [15], [16]. A VP is essentially an executable abstract model of the entire hardware platform, and from the software perspective a VP mimics the real hardware. For enabling the execution of programs by the VP, an *Instruction Set Simulator* (ISS) has to be created for the processor. We follow this approach and take advantage of the ISS principle, and by this significantly enhance the design flow of OISCs.

Contribution: In this paper, we present ORISCV-ISS, an exploration platform for microcoded RISC-V cores leveraging the OISC principle. We have chosen RISC-V since it is an open and royalty-free ISA [17]. In recent years, RISC-V gained a lot of attraction in academia and industry and started to become a game changer for embedded systems. Our developed ORISCV-ISS[1] allows to combine the advanced ecosystem of RISC-V with the ultimate minimalism of OISCs. Through this combination, we mitigate the common OISC drawbacks and investigate a new approach to implementing the RISC-V ISA.

ORISCV-ISS serves as development platform for both, hardware architecture and microcode procedures, and provides the basis for early design space exploration. This is demonstrated by utilizing the `SUBLEQ` OISC architecture to serve as the microcode basis: From the software side ORISCV-ISS accepts RISC-V code, i.e. it behaves like an RV32I compliant core[2], however ORISCV-ISS decodes each RISC-V instruction and selects the defined `SUBLEQ` microcode procedure to be executed on the OISC execution unit. To enable design space exploration we have integrated a timing model into ORISCV-ISS. Hence, we can evaluate the effect of various memory configurations (which define the location of the different registers, such as RISC-V registers, OISC registers, and functional registers) on the performance. At the same time, ORISCV-ISS enables the parallel development and exploration of microcode procedures wrt. alternatives in the hardware support of the chosen OISC instruction. Altogether, ORISCV-ISS provides key information to balance between area savings and system performance in an early design stage while leveraging the benefits of the RISC-V ecosystem.

This paper is structured as follows: Section II discusses related work. Thereafter, the preliminaries are provided in Section III. Section IV introduces the architecture of the proposed ORISCV-ISS, whereas Section V presents the microcode layer and considers the RISC-V compliance. In Section VI, we present the evaluation of ORISCV-ISS for a large set of benchmarks and configurations. Finally, the paper is concluded in Section VII.

II. Related Work

We are not aware of any work considering the OISC principle and the RISC-V ISA. However, there are two related approaches, which we discuss in the following.

[1]ORISCV-ISS is available open-source on GitHub: https://github.com/ics-jku/riscv-subleq-iss.

[2]ORISCV-ISS passes all offical RISC-V compliance tests.

Emulating instructions from another ISA was first proposed in [8], where the authors complemented a MIPS processor with a tiny OISC coprocessor to detect and mitigate hardware faults. To "activate" the coprocessor, the authors extended the MIPS ISA with a `SUBLEQ` instruction and corresponding utility functions. This enabled them to instrument MIPS binaries with `SUBLEQ` instructions emulating subsequent MIPS instructions and react in case of a detected fault. In comparison to our proposed ORISCV-ISS architecture, the goal of [8] is fault detection, and in addition a modified compiler toolchain is required, thus dramatically increasing the entry barrier.

In [18], the authors also complement a MIPS processor with an OISC coprocessor but with the goal of reducing the size of the main processor by shifting area intensive instructions to the OISC core. While the authors utilize a similar "microcode" architecture, they provide a hardware platform generated with *High Level Synthesis* (HLS). ORISCV-ISS is positioned much earlier in the design process focusing on guiding the development of concrete hardware implementations by providing early insight through execution metrics.

III. Preliminaries

A. OISCs

In this section, we briefly review the OISC principle. For further information on OISC architectures we refer the reader to [3], [4]. OISCs are the ultimate form of RISC computers relying on only one single instruction for all computations. Of all OISC instructions, `SUBLEQ` has found the widest adoption due to its relative efficiency and simplicity in programming. We also use `SUBLEQ` as a basis for ORISCV-ISS and therefore introduce it in more detail now.

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 OISC program counter, else the OISC program counter is incremented by 1.

In the following example we demonstrate how the `SUBLEQ` instruction can be used to implement the *addition* operation. Please note that there is no opcode for the `SUBLEQ` instruction in the following example since it is the only available instruction and therefore the opcode is redundant. Thus, the instruction is represented by the arguments A, B and C only.

Example 1. *Even though the* `SUBLEQ` *instruction only provides a way to subtract one number from another we can easily perform an* **addition** *by (i) negating the 1st operand and (ii) subtracting the resulting value from the 2nd operand. In Listing 1, we want to calculate* `SRC1` + `SRC2`*. For this, we first negate* `SRC1` *in Line 1 by subtracting it from register* `TMP0`*, which we assume is* 0*. Also, we set the branching target to* 1 *such that no matter the result, the next executed instruction will be the following one. Then, we subtract the negated value in* `TMP0` *from* `SRC2`*, thus calculating* `SRC2` *-* (*-*`SRC1`) *in Line 2. Finally, in Line 3 we reset* `TMP0` *by calculating* `TMP0` *-* `TMP0` *which will always result in* 0*.*

```
1  SRC1 TMP0 1          # TMP0 = -SRC1
2  TMP0 SRC2 1          # SRC2 = SRC2 - (-SRC1)
3  TMP0 TMP0 1          # TMP0 = 0
```

Listing 1: Integer addition

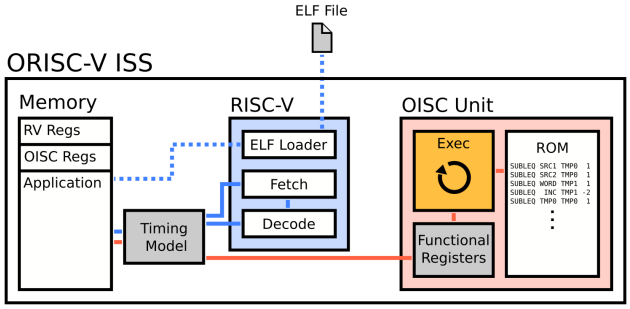

Fig. 1: Architecture of the ORISCV-ISS

B. RISC-V

The RISC-V ISA emerged from UC Berkeley and the ISA standard is maintained by the non-profit RISC-V foundation. The RISC-V ISA consists of a mandatory base integer instruction set and various optional extensions. In this work, we consider the configuration with 32 bit registers which is denoted as RV32I. Each instruction has at most two source registers RS1 and RS2, and one destination register RD. For example the shift left logical `sll t0, t1, t2` which takes the content of `t1`, shifts it by the content of `t2` and stores this result into `t0`. Another example is the add immediate `addi t0, t1, i` instruction which performs the addition of the source registers `t1` and the constant `i` and stores the result in `t0`. A comprehensive description of the RISC-V ISA is available in [17].

IV. ORISCV-ISS Architecture

In this section, we introduce our proposed ORISCV-ISS. First, we give an overview about the ORISCV-ISS and discuss the architecture (Section IV-A). Then, in the following sections, we detail the main components.

A. ORISCV-ISS *Overview and Architecture*

ORISCV-ISS has been implemented in approximately $1,000$ lines of plain C++ code. The current microcode implementation(s) support RV32I, although the ISS itself features all requirements to support additional RISC-V extensions. An overview of the architecture of ORISCV-ISS 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.

B. RISC-V Interface

To the user, ORISCV-ISS 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.

C. OISC Execution Unit

The OISC execution unit (right side in Figure 1) implements the execution loop of the OISC microcode[3]. 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, ORISCV-ISS allows a hybrid OISC design space exploration of a mixture of arithmetic-based and transport triggered OISC architectures.

D. 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 ORISCV-ISS reflects this common feature of OISC architectures by mapping all RISC-V and OISC registers into the main memory[4]. However, ORISCV-ISS allows to easily model access times on the registers via a timing model. By this developers can estimate the impact of the system architecture on the run time without modification of the microcode. We will use this feature intensively in the experiments in Section VI.

E. Timing Model

To enable design space exploration of different architectures (e.g. dedicated registers vs. registers in memory) every access to the memory (except from the ELF loader since it runs before simulation time) 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. This way, even though all registers (RISC-V and OISC) are contained in the main memory, the timing model enables the evaluation of execution metrics when running RISC-V software. For example, the added delay can be significantly reduced for reads or writes to the OISC registers if they are implemented in dedicated registers. Naturally, these dedicated registers incur significant area costs when implemented in hardware. To summarize, the timing model enables early design space exploration thus helping developers deciding on the hardware architecture.

In the next section, the microcode layer of ORISCV-ISS is introduced and the RISC-V compliance is shown.

[3]As already mentioned we use `SUBLEQ` microcode instructions, but ORISCV-ISS can easily be extended to other OISC instructions.

[4]Except the OISC program counter.

TABLE I: 16 OISC registers visible to microcode

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

V. ORISCV-ISS Microcode Layer and RISC-V Compliance

In this section, we introduce the microcode layer of ORISCV-ISS. This includes in particular the registers visible to the microcode as well as the microcode instruction format (Section V-A). Next, we show a concrete microcode example for an RISC-V instruction (Section V-B). Finally, we show the RISC-V compliance of ORISCV-ISS (Section V-C).

A. Registers and Microcode Instruction Format

Since compactness is a major goal during the design of microcode, we adapt the instruction format of `SUBLEQ A B C`: In comparison to other OISC architectures, the operands *A* and *B* do not have to address the full main memory, but only a limited number of registers. Since in other works applications are directly compiled to OISC instructions they must be able to address the complete memory. In contrast in this work, only a few registers are sufficient to implement the RISC-V instructions. Limiting the *A* and *B* operands of the `SUBLEQ` instruction to a few registers only enables us to reduce the instruction length, thus saving a precious amount of microcode memory. Table I lists the registers available to the microcode.

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. Usually, the `SUBLEQ A B C` instruction sets the OISC program counter to the value of operand *C*, if the resulting value *r* of the `SUBLEQ` computation is smaller or equal to 0. This allows to perform long-range jumps, which is crucial for programming applications directly in the OISC language. In the proposed microcode however, only local jumps of a few instructions are needed enabling us to change the global jump target to a pc-relative jump. This again allows to reduce the instruction length saving a lot of space. We provide a concrete example demonstrating the considerations.

Example 2. *For this example we assume a main memory of* $16\ kB$. *Fully addressing this memory requires* 14 *bits, therefore one instruction has a length of* $3 * 14 = 42$ *bits.*[5] *With the above introduced enhancements and the microcode registers of Table I, we only need* 4 *bits to address every register. Additionally, we assume that no microcode jump is more than a generous* 128 *instructions long, requiring* 8 *additional bits for the signed jump offset operand. In total, this results in a* $4+4+8 = 16$ *bit long microcode instruction, thus reducing the length by* 60%.

B. `SUBLEQ` Microcode implementing RV32I

For the implementation of the RV32I instructions as `SUBLEQ` microcode procedures, we required no additional hardware support.

[5]We also assume, that the program resides in the main memory as in most OISC implementations.

In the following, we show exemplary microcode procedures for the RISC-V immediate addition `addi` and shift left logical `sll`, respectively.

Example 3. *Consider Listing 1. Before the microcode procedure realizing the RISC-V instruction* `addi t0, t1, 22` *is started*[6]*, the RISC-V interface copies the required operands from RISC-V register* `t1` *and the immediate value* 22 *into the OISC registers* `SRC2` *and* `IMM`*, respectively. Then, in Line 1 the* `22` *in* `IMM` *is subtracted from OISC register* `TMP0` *and the result is stored back in* `TMP0`*. Since* `TMP0` *is guaranteed to contain a zero at the start of the microcode procedure, OISC register* `TMP0` *now holds the value* `-22`*. Next,* `TMP0` *is subtracted from* `SRC2` *which finally performs the addition via a doubled negation. At the end of the procedure the temporary register* `TMP0` *is cleared (Line 3), and the RISC-V program counter is incremented (Line 4). Finally, the RISC-V interface reads the result from* `SRC2` *and copies it into the destination register* `t0`*.*

```
   addi t0, t1, 22
   # RISC-V Interface: reg[SRC2] = reg[t1]
   # RISC-V Interface: reg[IMM] = 22
1    IMM  TMP0    1     # TMP0 = -22
2    TMP0 SRC2    1     # SRC2 = SRC2 - (-22)
3    TMP0 TMP0    1     # TMP0 = 0
4    NEXT   PC  END     # RV-PC = RV-PC + 4
   # RISC-V Interface: reg[t0] = reg[SRC2]
```

Listing 2: Microcode procedure for RISC-V addi

Example 4. *Typical* `SUBLEQ` *shift operations make use of the fact that a single left shift of* x *is equal to* $x + x$ *and simply repeating this operation depending on the shift amount. Listing 3 contains the microcode procedure that implements the* `sll` *RISC-V instruction. Before the execution of a* `sll` *instruction, the RISC-V interface places the shift amount in register* `SRC1` *and the value to be shifted in* `SRC2`*. First, in Line 1 the shift amount is negated and stored into register* `TMP0`*. Then, the shift loop body is entered in which a left shift of one bit is performed by adding the current shift value in* `SRC2` *to itself in Line 2-3. At the end of the shift loop body (Line 4), the temporary register* `TMP1`*, which stored the negated shift value, is cleared. In Line 5, we increment the negated shift amount in register* `TMP0` *and jump back to the beginning of the shift loop body if the result is smaller or equal to* 0*. Note that in* `SUBLEQ` *microcode loops, counters are typically counting from negative to positive since this can easily be implemented using the* `SUBLEQ` *instruction. If the loop is left, the used temporary registers are cleared in Line 6-7 and the RISC-V program counter is incremented in Line 8. Finally, the RISC-V interface reads the result of the shift operation from* `SRC2` *and writes it into the destination register* `t1`*.*

```
   sll t0, t1, t2     RISC-V shift left logical
   # RISC-V Interface: reg[SRC1] = reg[t2]
   # RISC-V Interface: reg[SRC2] = reg[t1]
1    SRC1 TMP0 2       # TMP0 = -SRC1
2    SRC2 TMP1 1       # TMP1 = -SRC2
3    TMP1 SRC2 1       # SRC2 = SRC2 - (-TMP1)
4    TMP1 TMP1 1       # TMP1 = 0
5    INC  TMP0 -3      # TMP0 = TMP0 + 1
6    TMP0 TMP0 1       # TMP0 = 0
7    TMP1 TMP1 1       # TMP1 = 0
8    NEXT PC   END     # RV-PC = RV-PC + 4
   # RISC-V Interface: reg[t0] = reg[SRC2]
```

Listing 3: Microcode procedure for RISC-V sll

[6]We use a concrete immediate value of 22 in the example for ease of understanding.

In the next section we show the RISC-V compliance of the developed microcode.

C. RISC-V Compliance of Microcode

The *RISC-V Architectural Tests*[7] from the RISC-V foundation provide a standardized testing infrastructure for checking that a RISC-V implementation (ISS or hardware) meets the RISC-V ISA standard [19]. The architectural test-suite is split up in multiple tests for each RISC-V configuration. To check that ORISCV-ISS correctly implements the RV32I configuration, we created a new target architecture for our ISS and compiled the RV32I test-suite. Overall, the RV32I base instruction set contains 37 unprivileged instructions. Our microcode implementation passes the complete RV32I architectural tests for each of the 37 instructions. In sum, the architectural test-suite consists of $12,603$ individual test cases.

D. SUBLEQ Correctness and Fixed Word Lengths

During the development of ORISCV-ISS as well as the OISC microcode procedures for each RISC-V instruction we also looked into the available OISC literature. While the idea of mapping a certain operation to `SUBLEQ` microcode is pretty simple, corner cases in particular wrt. to overflows can be problematic.

```
1   SRC1 TMP0 2         # 0 - (-4) = -4
2   TMP1 TMP1 END       # correct
3   TMP1 TMP1 -5        # wrong
```

Listing 4: Overflow error leading to wrong branch taken

In the following we briefly discuss this challenge which, to the best of our knowledge, is not described in the OISC literature. For example, consider the `SUBLEQ` program in Listing 4 and assume that all `SUBLEQ` instructions operate on 3-bit values using two's complement: `SRC1` holds the 3-bit value -4 and `TMP0` holds the 3-bit value 0. In 3-bit two's complement, -4 and 3 are the smallest and largest possible values, respectively. Therefore, Line 1 leads to an erroneous branch since $0 - (-4) = 4$, which cannot be represented using 3 bit only, and thus the branch should not be taken. However, in 3-bit signed arithmetic using two's complements $0 - (-4) = -4$ which is $<= 0$ and thus the branch in Line 1 two instructions ahead is taken.

We solve this problem in ORISCV-ISS by computing the results using "enough" bits. Since the ISS is at the VP level, we cast the 32-bit values before the subtraction to 64-bit, perform the comparison on this 64-bit result and then cast the result back to 32-bit before storing the result in a register. Therefore, our developed microcode shows the correct behavior. In a hardware implementation (using VHDL or Verilog) it is important that the OISC instruction can compute exact results in a mathematical sense.

VI. Evaluation

The performance of a processor depends not only on the raw number of instructions it can execute in a given time, but also to a large extend on other components such as the memory layout or the ISA. Since OISC architectures primarily target low-performance embedded devices, balancing between performance and chip area is of great importance. In this section, we demonstrate how ORISCV-ISS helps to find this balance.

We start from an ISS configuration in which only the `SUBLEQ` instruction is available and all registers are mapped into the main memory. This represents our base case for maximum area efficiency

[7]formerly known as RISC-V Compliance Tests

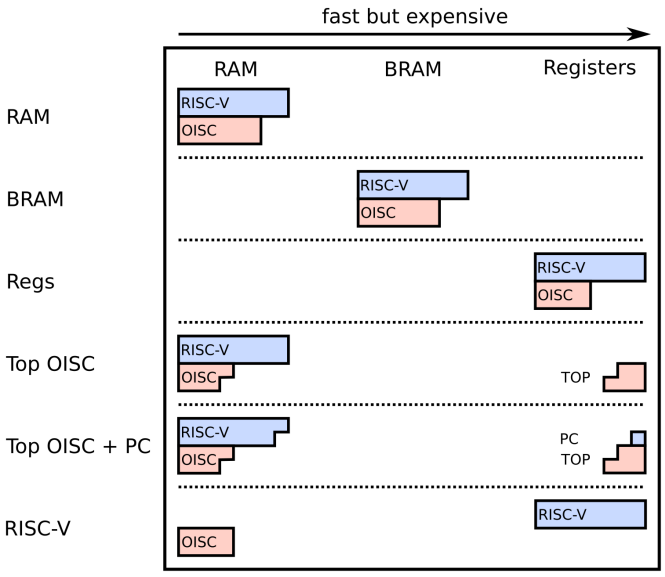

Fig. 2: Memory configs for RISC-V and OISC registers

from which we explore possible improvements and their impact on the system performance (Section VI-A). To achieve this goal we tune and explore two design characteristics: (1) the memory configuration, and (2) extensions to the OISC ISA. First, we shift the memory mapped registers towards dedicated registers for improved access times (Section VI-B). Then, we analyze the effect of an extended `SUBLEQ` architecture with specialized hardware for the most computationally intensive microcode procedures (Section VI-C).

A. Basic Timing Models

In this evaluation, we employed three basic memory configurations. Figure 2 visualizes these basic configurations as well as three additional configurations which we will describe in Section VI-B. The base assumption is that the execution of one `SUBLEQ` instruction takes 1 cycle.

Reading or writing to registers or the main memory adds a delay depending on the specific memory configuration. The first configuration, **RAM**, maps all RISC-V and OISC registers into the main memory. Since reading and writing to the main memory is slow, we add a delay of 4 additional cycles to each memory access. In the second configuration, **BRAM** all RISC-V and OISC registers are implemented in larger on-chip memories like *Block RAMs* (BRAMs), but accessing the main memory for application data still requires 4 additional cycles. These memories are faster than the main memory, and thus, only add a delay of 1 for each read or write access. In the third configuration, **Regs**, we placed all RISC-V and OISC registers into dedicated registers. Reading and writing to these registers is fast and highly parallel, so no additional delay is added to the `SUBLEQ` execution time. The remaining 3 configs in Figure 2 are described later when the context is clear.

B. Evaluation of Memory Configurations

In this section, we present a comparison of 6 memory configurations in terms of the number of cycles needed when executing software. We have taken several well-known benchmarks, compiled them to RV32I, and executed them on ORISCV-ISS in the respective memory configuration. Table II provides the results. The first column gives the name of the benchmark. The next three columns report the required microcode cycles according to the three *Basic Memory Configurations*. As can be seen in Table II, implementing all registers in **BRAM** reduces the number of cycles on average by 67.82% wrt. the baseline. In comparison, shifting all RISC-V and OISC registers from the RAM to dedicated registers (**Regs**) cuts the required cycles more than 90%.

We now extended our design space exploration using ORISCV-ISS as follows: We wanted to find a configuration with minimal additional area costs but a significant cycle reduction. For this, we evaluated the **Top OISC** configuration, in which all registers are kept in the main memory except `SRC1`, `SRC2`, `TMP0`, and `TMP1` from the OISC unit. With just these four registers we already observed on average a reduction of 48.38% wrt. the baseline. In configuration **Top OISC + PC**, also the RISC-V program counter is implemented in a dedicated register since it is one of the most used RISC-V registers. By simply adding the RISC-V PC to the dedicated registers, we were able to additionally reduce the average number of cycles by nearly 10%. Finally, we evaluated a last configuration in which only the **RISC-V** registers are implemented in dedicated registers. Even though this configuration contains all 32 RISC-V registers and the RISC-V PC, the measured cycle reduction amounted to only 18.78%. This shows, that the performance improvement highly depends on the selected registers. This was expected, since every RISC-V instruction requires multiple OISC instructions and thus the OISC registers are used much more frequently, whereas the RISC-V registers are only used at the beginning and end of the execution of one RISC-V instruction.

C. Evaluation of `SUBLEQ` Extensions

Not every microcode procedure has the same complexity. For example, only 4 `SUBLEQ` instructions and cycles are required to execute the `addi` or `add` instructions. In contrast, all instructions that require bit-level access are much more complex. In our base microcode implementation the `and` as well as `andi` microcode procedure is 42 `SUBLEQ` instructions long. This procedure however loops depending on the word width of the processor (in case of RISC-V RV32I this is $XLEN$; 32) resulting in hundreds of microcode cycles for a single RISC-V instruction. This clearly shows that the performance depends on the to be executed instructions. However, this also creates opportunities to tailor the hardware to the specific application which we demonstrate in the following.

We measure how many microcode cycles it takes on average to execute one RISC-V instruction using the base microcode implementation. From there, we extend the ISS and the microcode using the functional registers to reduce the complexity of the most demanding microcode procedures.

Table III compares the average microcode cycles per RISC-V instruction for two microcode implementations. We can see, that the **Base** implementation without any hardware support normally requires around 10 to 14 microcode cycles per RISC-V instruction. Of all benchmarks only the `bit insert` benchmark makes heavy use of bit-level operations, whereas the other benchmarks mostly perform word-level arithmetic operations. This is clearly visible in the benchmark results where the average cycles of the `bit insert` benchmark shoot up to 24.5 cycles per RISC-V instruction for the **Base** implementation.

In order to reduce the overhead for this benchmark, we added hardware support for the `and`, `or`, and `xor` RISC-V instructions[8] via the functional registers. The **BitOps** microcode implementation extends the **Base** configuration but replaces the costly bit operations

[8] and also their immediate variants.

TABLE II: Required microcode cycles for different memory configurations

	Basic Memory Configurations			Extra		
Benchmark	RAM	BRAM	Regs	Top OISC	Top OISC+PC	RISC-V
bit insert [20]	56,474	17,714	4,794	36,122	33,762	51,990
dijkstra [21]	753,261	241,062	70,329	378,093	311,569	620,341
heapsort [22]	96,21,348	3,100,233	926,528	4,836,100	4,029,276	7,990,048
bubblesort	2.11×10^8	6.80×10^7	2.05×10^7	9.49×10^7	7.35×10^7	1.67×10^8
ref [3]	2.23×10^8	7.26×10^7	2.23×10^7	1.10×10^8	8.30×10^7	1.73×10^8
ackermann [22]	4.90×10^8	1.58×10^8	4.78×10^7	2.53×10^8	1.96×10^8	3.84×10^8
sieve [22]	4.29×10^9	1.40×10^9	4.33×10^8	2.19×10^9	1.67×10^9	3.26×10^9
Avg. Reduction	Baseline	67.82%	90.43%	48.38%	58.00%	18.78%

TABLE III: Avg. cycles per RISC-V Instr.

Benchmark	Base	BitOps
bit insert	24.5	12.3
dijkstra	12.9	12.3
heapsort	13.9	13.8
bubblesort	11.5	11.4
ref1	10.1	10.1
ackermann	11.3	11.3
sieve	10.0	10.0
Average	13.8	11.6

with microcode procedures that target the functional registers. As a consequence, with this implementation the number of cycles is now only half the number of cycles for the `bit insert` benchmark (first row in Table III). The other benchmarks are mostly unaffected by the new microcode implementation since they do not contain many calls to the bit-level instructions.

Additionally, the size of the **BitOps** microcode implementation with only 259 instructions is 25% smaller then the **Base** microcode implementation. This reduction is possible since the complex procedures handling the bit operations in the base implementation are replaced with simple writes to the special registers.

VII. Conclusions

In this paper, we have proposed ORISCV-ISS which serves as a development platform for both, hardware architecture and microcode procedures, and provides the basis for early design space exploration. Our ISS is a fully compliant implementation of the RV32I ISA, whose instructions are executed in microcode using the OISC paradigm. ORISCV-ISS consists of only 1,000 lines of code. It can be easily adapted to explore various architectures (e.g. reflecting different memory configurations) or custom OISC ISA extensions. We evaluated several ISS configurations and measured their impact on the system. By carefully selecting registers and ISA extensions we were able to improve the performance significantly.

For future work we will enhance ORISCV-ISS by integrating advanced timing models (e.g. [23], [24]) and we will leverage new verification approaches (e.g. [25]).

Acknowledgments

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

References

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

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

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

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

[13] T. De Schutter, *Better Software. Faster!: Best Practices in Virtual Prototyping*. Synopsys Press, March 2014.

[14] O. Bringmann, W. Ecker, A. Gerstlauer, A. Goyal, D. Mueller-Gritschneder, P. Sasidharan, and S. Singh, "The next generation of virtual prototyping: Ultra-fast yet accurate simulation of hw/sw systems," in *DATE*, 2015, pp. 1698–1707.

[15] A. Charif, G. Busnot, R. Mameesh, T. Sassolas, and N. Ventroux, "Fast virtual prototyping for embedded computing systems design and exploration," in *RAPIDO Workshop*, 2019, pp. 3:1–3:8.

[16] V. Herdt, D. Große, and R. Drechsler, *Enhanced Virtual Prototyping: Featuring RISC-V Case Studies*. Springer, 2020.

[17] 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.

[18] T. Ahmed, N. Sakamoto, J. Anderson, and Y. Hara-Azumi, "Synthesizable-from-c embedded processor based on MIPS-ISA and OISC," in *EUC*, 2015, pp. 114–123.

[19] "RISC-V architecture test sig," https://github.com/riscv-non-isa/riscv-arch-test, 2021.

[20] Hansraj Das, "Algorithm repository," https://github.com/hansrajdas/algorithms, 2018.

[21] R. Abishek, "Dijkstra algorithm implementation," https://gist.github.com/rajabishek/2c75052a674fd15e2170, 2015.

[22] Microsoft, "Microsoft test-suite," https://github.com/microsoft/test-suite, 2007.

[23] K. Grüttner, P. A. Hartmann, T. Fandrey, K. Hylla, D. Lorenz, S. Stattelmann, B. Sander, O. Bringmann, W. Nebel, and W. Rosenstiel, "An ESL timing & power estimation and simulation framework for heterogeneous SoCs," in *SAMOS*, 2014, pp. 181–190.

[24] V. Herdt, D. Große, and R. Drechsler, "Fast and accurate performance evaluation for RISC-V using virtual prototypes," in *DATE*, 2020, pp. 618–621.

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