SCAIE-V: An Open-Source SCAlable Interface for ISA Extensions for RISC-V Processors

Mihaela Damian
Technical University Darmstadt
Darmstadt, Germany
damian@esa.tu-darmstadt.de

Julian Oppermann
Technical University Darmstadt
Darmstadt, Germany
oppermann@esa.tu-darmstadt.de

Christoph Spang
Technical University Darmstadt
Darmstadt, Germany
spang@esa.tu-darmstadt.de

Andreas Koch
Technical University Darmstadt
Darmstadt, Germany
koch@esa.tu-darmstadt.de

ABSTRACT

Custom instructions extending a base ISA are often used to increase performance. However, only few cores provide *open interfaces* for integrating such *ISA Extensions (ISAX)*. In addition, the degree to which a core's capabilities are exposed for extension varies wildly between interfaces. Thus, even when using open-source cores, the lack of standardized ISAX interfaces typically causes high engineering effort when implementing or porting ISAXes. We present *SCAIE-V*, a highly portable and feature-rich ISAX interface that supports *custom control flow*, *decoupled execution*, *multi-cycle-instructions*, and *memory transactions*. The cost of the interface itself scales with the complexity of the ISAXes actually used.

KEYWORDS

ISA Extension Interface, Custom Instructions, RISC-V

ACM Reference Format:
Mihaela Damian, Julian Oppermann, Christoph Spang, and Andreas Koch. 2022. SCAIE-V: An Open-Source SCAlable Interface for ISA Extensions for RISC-V Processors. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC) (DAC '22), July 10–14, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3489517.3530432

1 INTRODUCTION

With the emergence of the free and open-source RISC-V ISA, hardware engineers can now pick from an ever-growing list of compatible open-source and commercial processor cores [6, 10]. Depending on the project, it becomes quite practical to switch back and forth between cores [11, 16], to satisfy on evolving requirements. This becomes more difficult though, when the general-purpose RISC-V ISA has been extended with custom instructions (here called ISAX) to achieve specific design goals (e.g., performance, energy etc.).

ISAXes hinder portability in two dimensions. First, custom instructions tailored for a certain application may not be suitable for improving *other* workloads. Second, optimizing an ISAX for a specific processor microarchitecture often leads to lock-in to that specific processor core. As a result, classical ISAXes induce a high engineering effort when being updated, or when being ported between processor cores.

Prior work has already proposed a number of processor-ISAX interfaces. Some of these are tailored for a specific domain, such as vector processing in case of ORCA [18] cores. Others are more generic. E.g., PicoRV32 [5] also provides the PCPI interface for instructions implementing non-standard opcodes. However, PCPI does not support control flow instructions or memory transactions. CV32E40X [12] also supports offloading non-standard opcode instructions via an abstract interface. However, it lacks support for custom control flow ISAXes. While CV32E40X's interface is well suited for multi-cycle accelerators, it does not tailor the resource usage of the interface for the actual ISAXes' requirements.

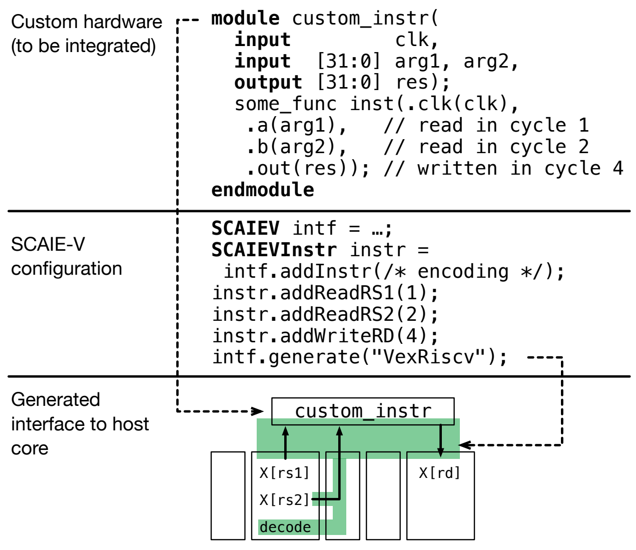
To overcome the drawbacks of the prior mostly static interfaces, which cannot be flexibly matched to the actual ISAXes' needs, we propose SCAIE-V. At the heart of SCAIE-V lies an automatic hardware generator, which integrates custom instructions via an adaptive lightweight interface.

We introduce the workflow of SCAIE-V in Figure 1. A designer provides the hardware implementation underlying a new instruction, and configures SCAIE-V for the required capabilities. For the supported processors, SCAIE-V will then automatically update the core and generate the least-cost interface between the custom logic and the core.

SCAIE-V's offers a number of benefits, with the key contribution being the *reduction in effort to implement and port a wide variety of ISAXes between different cores.*

- Scalability: The hardware cost of SCAIE-V scales with the capabilities required by the ISAXes. Simple instructions carry only a low overhead.
- Portability: SCAIE-V abstracts the individual core's microarchitecture and hardware description language. It supports both pipelined and non-pipelined RISC-V processor cores.
- Flexibility: SCAIE-V supports advanced features such as *custom control flow*, *decoupled execution*, *custom multi-cycle instructions*, and *custom memory and I/O instructions* and optional support for *dynamic scheduling*.

The following section provides details on the differences between SCAIE-V and prior work. Section 3 describes the capabilities of our


Custom hardware (to be integrated)
SCAIE-V configuration
Generated interface to host core
custom_instr
X[rs1]
X[rs2]
decode
X[rd]

```
module custom_instr(
  input         clk,
  input  [31:0] arg1, arg2,
  output [31:0] res);
  some_func inst(.clk(clk),
   .a(arg1),   // read in cycle 1
   .b(arg2),   // read in cycle 2
   .out(res)); // written in cycle 4
endmodule
```

```
SCAIEV intf = …;
SCAIEVInstr instr =
 intf.addInstr(/* encoding */);
instr.addReadRS1(1);
instr.addReadRS2(2);
instr.addWriteRD(4);
intf.generate("VexRiscv");
```

Figure 1: SCAIE-V flow: SCAIE-V extends a base core (green area) to provide/accept a custom hardware module's inputs/outputs at the cycles specified in the configuration.

approach. As the proposed interface aims to support multiple cores, the main characteristics of these cores and challenges are discussed. In Section 4, we evaluate the efficiency of our concept.

2 RELATED WORK

As shown in Figure 2, we differentiate between three hardware accelerator approaches: 1. dedicated accelerators, 2. custom coprocessor ISAXes, 3. in-pipeline ISAXes. These differ in their integration and interfaces. Highly portable dedicated accelerators are often attached via interfaces such as PCI Express, Memory Mapped IO (MMIO), or similar. While portable, these may carry latency penalties, have limited throughput, as well as high area/power costs (PCIe).

Pipeline-integrated or custom on-chip coprocessor-based-ISAXes avoid off-chip latency and throughput limitations. In the following, we differentiate between tightly integrated in-pipeline [13], and coprocessor ISAX approaches [19]. While the prior usually lead to better resource usage, but cause more design effort, the latter scale better, but may come with higher area and timing costs. However, on-chip interfacing between processor cores and tightly integrated ISA-extensions is not portably standardized, yet.

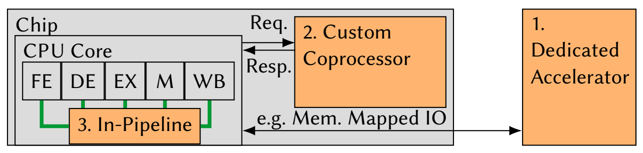


Figure 2: Different Implementation approaches: 1. Dedicated Accelerator, 2. Custom Coprocessor, 3. In-Pipeline. Deep integration can limit inter-core portability. Off-chip designs may affect latency, throughput, and hardware cost. SCAIE-V's connections are highlighted in green.

Codasip [1] offers a closed-source commercial solution for integrating custom instructions. The user can specify a new instruction in a Domain Specific Language (DSL) called CodAL, from which the required tools for an automatic integration of the new instructions [2] are generated. In contrast to most open source interfaces, their approach permits support for custom control-flow. The Andes Copilot closed-source ISAX interface is described to have in-pipeline features and to be compatible with multiple Andes processors [3]. However, we did not find benchmarks, and, due to its closed-source nature, the interface is not available on non-Andes cores.

Multiple open-source co-processor ISAX interfaces exist. Examples include the EAI [19] interface designed for the E203 core, the eXtension Interface [12] of the CV32E40X processor, and the RoCC [14] interface of the Rocket core. In all cases, the communication with the main core is based on a request-response mechanism for offloading the custom instructions, accessing the memory bus, and writing the results.

Although this concept scales well for more complex multi-cycle instructions, the co-processor approach is not sufficiently tightly integrated into a core to also support custom control flows. These can be quite desirable, as they support, e.g., conditional branches depending on ISAX-internal state, or advanced features such as zero-overhead loops. Moreover, with their reliance on request-response protocols, many coprocessor-based designs are too heavyweight for combinational instructions, which have proven sufficient for many practical use-cases.

The PULPissimo platform [15] also provides a solution for integrating custom instructions. The user can add accelerators via the MMIO-based Hardware Processing Engine protocol. However, communicating through this interface requires additional clock cycles and is expensive especially for simple instructions.

At first glance, it would seem promising to employ existing interfaces designed for verification, tracing, debugging and short-latency attack responses [7–9, 17] to attach ISAXes. However, aside from the debugging interface, these cannot modify registers, and thus are not suitable for ISAXes. While the debugging interfaces could modify registers, their out-of-core nature would induce frequent pipeline stalls, resulting in high performance overheads.

Some cores that we currently support in SCAIE-V already bring their own ISAX interface. However, for the ORCA core, this interface is mainly designed for vector processing applications [4]. That of the PicoRV32 [5] core allows neither control flow instructions nor memory accesses. SCAIE-V overcomes all of these limitations and, as we demonstrate, is portable across multiple cores.

With its tighter integration into the logic of the main core, SCAIE-V supports not only rapid transfers to the register file, memory, and I/O devices, but also allows custom instructions to affect the control flow. Moreover, when possible, it saves hardware costs by *re-using* existing control- and datapaths of the main pipeline. Finally, SCAIE-V is portable to cores with significantly different microarchitectures (e.g., pipelined vs. FSM-based).

3 SCAIE-V FEATURES

3.1 Portability to Different RISC-V Cores

To demonstrate portability, we implemented SCAIE-V on four very different processor microarchitectures. Table 1 gives an overview of the configurations we chose for the four cores. The pipelined ones also require a handling of hazards, ideally by reusing existing bypassing/forwarding paths. E.g., ORCA has paths from the write-back to the decode stage, as well as from the output of the ALU to the input of the execution stage. SCAIE-V abstracts these variations behind its external ISAX interface.

Table 1: Overview of core characteristics.

Core	Pipeline Stages	Stage handling taken branches	Memory op stage	Memory interface	Design Language
Piccolo	3	1	2	AXI	Bluespec
VexRiscv	4	3	3	AHB	SpinalHDL
ORCA	5	5	4	AXI	VHDL
PicoRV 32	FSM, not pipelined	1st FSM state; addr. given by prev. instr.	3rd FSM state	native interface	Verilog

3.2 Simple ISAX-Core Interface

To enable a wide range of applications, SCAIE-V offers multiple register and memory operations, as shown in Table 2.

The interface has mandatory, as well as optional signals, to adapt to the requirements of the actual ISAX. For example, to support memory accesses, the interface requires a data signal, an optional valid flag, and an address bus. If the valid signal is not used, the memory transfers always occur when the corresponding instruction reaches the pipeline stage responsible for memory operations (Table 1). The memory accesses either reuse the existing address generation logic in the core (usually X[rs1] plus offset), or optionally instantiate a dedicated address bus to allow the ISAXes to perform their own address computations (e.g., skip/stride accesses).

Similar to memory accesses, the interface for register writes can also have optional valid and address (register number) signals. The valid signal can be employed to easily realize predicated updates. This is helpful when the custom instruction does not always want to write its result, but rather commits its output depending on an ISAX-internal condition.

For ISAXes holding internal state, SCAIE-V can additionally provide control signals from the core, such as forwarding the stall and flush signals of the main pipeline. These are essential for the correct

Table 2: Operations supported by SCAIE-V.

Operation	Semantics
RdRS1/2	Reads from the register file.
WrRD*	Writes to register file at the regno provided in the instruction field, or by the user.
RdPC	Reads the current program counter
WrPC*	Updates the program counter.
Rd/WrMem*	Starts memory accesses. Reuses core's address computation logic, or optionally employs a custom address bus.
RdInstr	Read current instruction word.
RdIValid_*X*	Status bit informing the custom logic that a certain pipeline stage currently contains an instruction of type *X*.

*Optionally predicated by a valid bit.

synchronization between the pipeline and the ISAX hardware. Furthermore, they prevent invalid updates of the internal state in case of flushed ISAXes. SCAIE-V also allows an ISAX to stall the pipeline, e.g., when transferring data to/from an ISAX-local memory or I/O interface.

3.3 From Simple Combinational to Complex Multi-Cycle and Decoupled Instructions

Our proposed interface targets a wide range of applications, from simple combinational custom instructions, to complex, multi-cycle accelerators. When integrating solely combinational instructions, only the decoder and the data hazard unit of the core are extended. However, the control of multi-cycle functional units is more complex. One possible approach would be stalling the main pipeline until the functional unit commits its results to the register file. Although SCAIE-V can be configured to use this approach, this strategy is inefficient, as an ISAX could halt the main pipeline for many cycles. Instead, SCAIE-V can be configured to allow ISAXes to be *decoupled* and run *in parallel* to the main pipeline, while the other stages continue their normal execution. Although this has a higher hardware cost, it can achieve better performance. Following the basic SCAIE-V paradigm, the cost for this more powerful functionality is only paid if the feature is actively used and implemented by our generator. Note that SCAIE-V permits a highly efficient implementation of such decoupled ISAXes. E.g., the area required for the decoupled zero-overhead loop mechanism shown in Figure 5(c) gets lost in the "optimization noise" of the FPGA mapping flow.

Enabling decoupled instructions implies a more complex control unit for handling data hazards. To track Read After Write (RAW) and Write After Write (WAW) hazards, we implement a "not-ready" bit for destination registers. Whenever a new multi-cycle instruction is issued, this bit is set for the corresponding destination register. The core's pipeline is stalled whenever it tries to execute an instruction that wants to read or write to an address for which the "not-ready" bit is set. This bit is cleared when the corresponding result is committed, or the ISAX finishes.

Decoupled instructions also require special care when jumps or branches are executed, as the ISAXes' underlying hardware must be synchronized with the main pipeline in that case. To this end, SCAIE-V forwards the flush control signal from the main pipeline to the hardware of the custom instruction. But simply connecting the core's flush signal to the ISAXes' hardware would lead to erroneous behaviour. When a branch is taken, the prior stages of the core are *always* flushed. But the hardware underlying the decoupled ISAX should only be flushed if that ISAX was issued *after* the branch instruction. To solve this problem, SCAIE-V *shifts* an "active" status for each decoupled ISAX along the pipeline. If a flush occurs for a pipeline stage where such an "active" bit is set, the corresponding ISAX hardware is aborted (will not commit any results).

If decoupled instructions are used (and only then!), the SCAIE-V generator automatically creates hardware logic for two synchronization instructions. `disaxfence` stalls the pipeline until all decoupled ISAXes have completed, `disaxkill` terminates all currently executing ISAXes. It is up to the hardware block realizing the ISAX to react to the kill signal appropriately, e.g., completing the current bus-cycle before resetting an internal control FSM back to `idle`.

These synchronization capabilities are required for constructs such as a zero-overhead loop mechanism, which runs in parallel to the main pipeline and is no longer valid when the execution leaves the scope of the current function (e.g., by an early return), thus must be explicitly terminated.

To simplify the logic for the control mechanisms discussed above, decoupled multi-cycle instructions are restricted to be issued in a single, specific pipeline stage. This also avoids the case where such an ISAX would be launched in a stage *after* than where the corresponding `disaxkill` functionality is realized, which would erroneously allow the decoupled ISAX to continue executing even after its intended demise.

A similar problem arises when handling data hazards. If an instruction starts and sets its "not-ready" bit in a later stage, the control logic must ensure that this does not cause any data hazards for instructions earlier in the pipeline. By always starting multi-cycle instructions in a fixed pipeline stage, namely as soon as register operands, or bypassed/forwarded values are available, the data hazard logic can be simplified.

3.4 Meta-Data Support and Efficient Intra-ISAX Scheduling

The supported cores have very different microarchitectures and offer various configurations. SCAIE-V supports multiple operations, as presented in Section 3.2, and each of them may occur in different clock cycles, depending on the underlying core. One possible approach for solving this issue would be by implementing a handshake for each interface signal. However, this would add hardware and delay overhead. An alternative would be restricting the interface just to a certain pipeline stage, for example the execution stage. In this scenario, the custom instruction would receive all operands, and would be allowed to request memory accesses or jumps, only in this stage. Clearly, this would be a significant limitation for the custom extensions. For example, in case of zero-overhead (hardware) loops or jumps, both of which are operations that could/must update the program counter in earlier stages, such a constraint would add unnecessary flushes and cause a run-time penalty.

This is where the second key component of SCAIE-V comes into play, beyond the interface/integration generator described so far. SCAIE-V can also provide core-dependent meta-data describing the intra-instruction timing (e.g., pipelines stages), which specifies when the core's state may be read or written. Respectively, it indicates in which clock cycles the custom instruction is allowed to conduct one of the operations shown in Table 2. For example, in case of the ORCA core, the functional unit can read the instruction fields already in the second pipeline stage, request memory access in the fourth stage, and update the register file in the fifth state. Program counter updates can be performed in *any* of these stages, with SCAIE-V auto-generating the required flush-handling logic.

This meta-data is essential when generating the ISAX hardware and internally scheduling its operations according to Table 2. It can be interpreted both by human designers as well as automatic High Level Synthesis (HLS) systems for creating hardware blocks optimally scheduled for each target processor.

For operations reading the core's state, SCAIE-V specifies not only the *first* cycle in which they may be scheduled, but also the

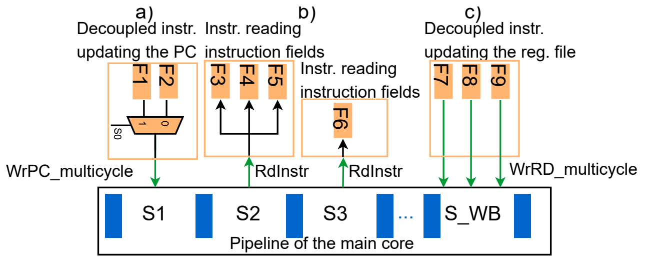

a)
Decoupled instr. updating the PC
F1
F2
1
0
s0
WrPC_multicycle
b)
Instr. reading instruction fields
F3
F4
F5
RdInstr
Instr. reading instruction fields
F6
RdInstr
c)
Decoupled instr. updating the reg. file
F7
F8
F9
WrRD_multicycle
S1
S2
S3
...
S_WB
Pipeline of the main core

Figure 3: Shared and exclusive interfaces, generated depending on operation type and pipeline stage.

last cycle when a state can be requested before it gets overwritten. This, too, is helpful to save hardware resources via an optimized schedule. For example, if an ISAX requires a field of the original instruction word in a later stage, SCAIE-V will provide this data exactly to that stage. In most cores, the relevant instruction fields are shifted to these later stages anyways, which enables the *re-use* of this part of the core's pipeline by SCAIE-V. This hardware re-use is currently supported for register file operands, instruction fields, and the program counter.

In order to automatically generate the ISAX-specific integrations, SCAIE-V must be informed not only which logical operations (Table 2) are required by each custom instruction, but also in which cycles they are to be performed. The generator will then create separate physical interfaces for the operations at the desired pipeline stages. Only a single physical interface will be generated at each stage, even if *multiple* instructions require the data in the same clock cycle (e.g., F3,F4,F5 in Figure 3.b). If other instructions need the *same* information in *later* stages (e.g., F6), the corresponding physical interface logic will also be generated in those stages (but again, at most once-per-stage). This rule does not apply to interfaces that serve *multi-cycle decoupled* instructions. When such instructions need to write their result to the register file, or access the memory bus, a dedicated interface is generated for each of the decoupled ISAXes (e.g., F7,F8,F9 in Figure 3.c). Multiple instructions running in parallel could return their results in the *same* clock cycle. In this case, the results are buffered and then committed sequentially in program-order to the register file. In this case, the pipeline is stalled until all results are written. For all decoupled instructions that may modify the program counter, only a single interface is created (e.g., F1,F2 in Figure 3.a). If multiple such instructions want to update the program counter at the same time, a priority mechanism (arbiter) is required, which must be defined in the ISAX hardware blocks. Note that such a priority mechanism is *not* required when multiple decoupled instructions want to commit their result in the same clock cycle to the *register file*. The decoupled instructions will have been issued in the first place only if they write to *different* registers. Otherwise, the Data Hazard (DH) handling logic generated by SCAIE-V would have prevented them to execute in parallel, due to detecting possible WAW hazards.

4 EVALUATION

4.1 Implemented Custom Instructions

In order to evaluate the SCAIE-V interface, the following custom instructions were implemented and tested on all supported cores:

- `hardware (zero-overhead) loop`: This custom instruction requires the program counter and two operands from the register file. These set the start and end addresses of a software loop, as well as the number of iterations. The information is stored in ISAX-internal registers, and the hardware block controlling the loop is decoupled to act in parallel to the main pipeline. As long as the loop is still active, it updates the program counter with the loop's *start address* each time the fetcher reaches the *end address* of the loop.
- `auto-increment load/store`: This functional unit encompasses three instructions: load, store and setup. Its main role is to generate a regular access pattern for memory transfers. One instruction sets-up the start address and the increment (stride) value. This data is stored in ISAX-internal registers. Two more instructions load or store data from the memory using the address given in the internal register, which is incremented after each transfer.
- `sincos`: These are two separate ISAXes, realized by the same hardware block. Depending on an instruction bit, this unit computes either the sine or the cosine of the input operand. It implements the well-known Cordic algorithm, and executes as a decoupled multi-cycle instruction.
- `SBOX`: As part of the Advanced Encryption Standard (AES) algorithm, this instruction performs the SBOX computation.
- `indirect jump`: This instruction updates the program counter based on an address read from the memory location given by `rs1` plus offset.

These ISAXes were selected in order to cover most of the SCAIE-V capabilities, and evaluate their impact on the core.

- `hardware loop`: a *decoupled control-flow* instruction that runs in *parallel* to the main pipeline and updates the program counter.
- `auto-increment load/store`: custom *memory* accesses.
- `sincos`: *decoupled multi-cycle* instructions that run in *parallel* to the main core, and commit into the register file.
- `SBOX`: a simple *combinational* instruction.
- `indirect jump`: a *control-flow* jump executed when the instruction is in a *later* pipeline stage.

4.2 Integration of Custom Instructions

The custom instructions were integrated with SCAIE-V in all four different cores. However, for brevity, only the integration flow for the ORCA core with its five `IF-D1-D2-M&EX-WB` stages is discussed here in closer detail. The instructions described in Section 4.1 require different signals in different pipeline stages, which is shown in Figure 4. The user would inform SCAIE-V which of the operations listed in Table 2 are required for each custom instruction and in which stage. Furthermore, the encoding of the new instructions must be provided. Our tool will then automatically update the core, and generate the customized SCAIE-V interface for the selected ISAXes. This means that the logic of the core is extended to recognise the new instructions, flush the pipeline in case of a jump, set the memory address in case of an auto-increment load/store, or update the register file when required. Moreover, the core-specific mechanism for handling data hazards is extended. Logic is also added to support the `sincos` and `hardware loop` *decoupled* instructions, as described in Section 3.3. For instructions with *internal state*, the flush and stall control signals of the main pipeline are forwarded to the custom logic. These are just some of the adaptations which are performed automatically by SCAIE-V, simplifying the integration considerably over the traditional manual approach.

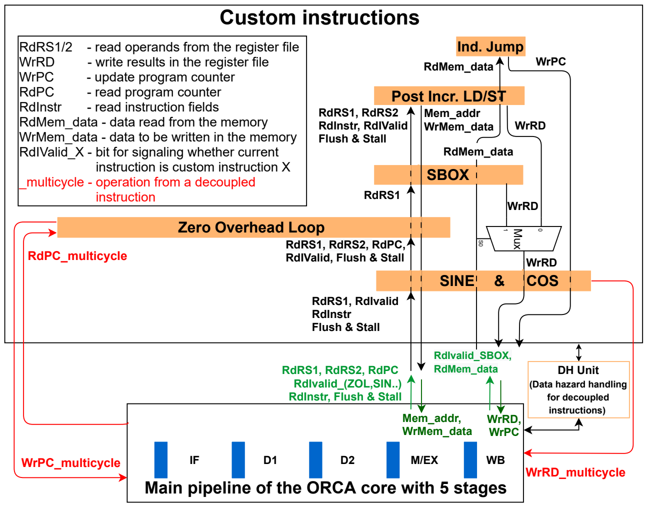

Custom instructions
RdRS1/2 - read operands from the register file
WrRD - write results in the register file
WrPC - update program counter
RdPC - read program counter
RdInstr - read instruction fields
RdMem_data - data read from the memory
WrMem_data - data to be written in the memory
RdIValid_X - bit for signaling whether current instruction is custom instruction X
_multicycle - operation from a decoupled instruction
Ind. Jump
RdMem_data
WrPC
Post Incr. LD/ST
RdRS1, RdRS2 RdInstr, RdIValid Flush & Stall
Mem_addr WrMem_data
WrRD
RdMem_data
SBOX
RdRS1
WrRD
Zero Overhead Loop
Mux
RdPC_multicycle
RdRS1, RdRS2, RdPC, RdIValid, Flush & Stall
WrRD
SINE & COS
RdRS1, Rdlvalid Rdlnstr Flush & Stall
Rdlvalid_SBOX, RdMem_data
RdRS1, RdRS2, RdPC Rdlvalid_(ZOL,SIN..) Rdlnstr, Flush & Stall
DH Unit (Data hazard handling for decoupled instructions)
Mem_addr, WrMem_data
WrRD, WrPC
WrPC_multicycle
IF
D1
D2
M/EX
WB
WrRD_multicycle
Main pipeline of the ORCA core with 5 stages

Figure 4: Interfaces required for the examined ISAXes. The width of the ISAX boxes indicates the affected stages.

4.3 Hardware overhead of the SCAIE-V Interface

For evaluating the impact of the SCAIE-V interface, we implemented the resulting designs on a Xilinx Zynq 7020 FPGA using an out-of-context flow. All instructions presented in Section 4.1 were implemented on the VexRiscv, Piccolo, PicoRV32 and ORCA cores. Figure 5(a) shows the resource usage and Figure 5(b) the clock frequencies for the core extended with SCAIE-V, but not considering the hardware underlying the ISAXes themselves. For these first two graphs, we consider just two scenarios: The original *unaugmented* core as a baseline, and one where *all* of the ISAXes discussed above were added to each of the four cores. The latter scenario is broken down into two subcases: One including automatically generated dynamic Data Hazard (DH) handling capabilities for the ISAX, and one without that relies on correct latency-aware static scheduling of instructions by the programmer or compiler.

As an example, for the VexRiscv core, adding all ISAXes without a DH unit causes SCAIE-V to have an area overhead of 11% of LUTs over the baseline, while adding the DH unit increases this to 26%. For the larger Piccolo core, these overheads drop to 5% and 7%, respectively. The clock frequency of VexRiscv drops by 7% when using SCAIE-V with dynamic hazard handling, and by just 2% for the static approach.

For space reasons, we can individually examine SCAIE-V overheads for each ISAX just for a single core. Figure 5(c) shows the area requirements for the SCAIE-V interface (again: *not* the underlying ISAX hardware itself) on a per-ISAX basis for VexRiscv with static scheduling. As can be seen, SCAIE-V is lightweight, even a decoupled multi-cycle instruction such as `sincos` requires less than 5% of extra LUTs to integrate in the core. Also, the generator is able to reuse hardware effectively, as the hardware overhead of SCAIE-V configured for all ISAXes (rightmost columns) is *less* than the sum of the overheads of the ISAX-individual SCAIE-V interfaces.

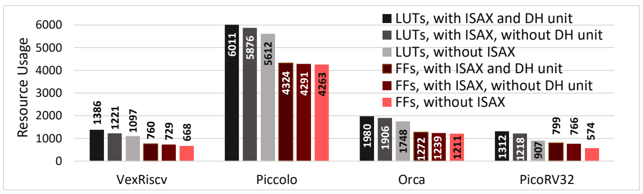
PicoRV32 already provides an interface (PCPI) for integrating custom instructions. As PCPI is more limited than SCAIE-V, we could not compare the resource usage overhead for control flow or memory instructions. However, for the `SBOX` instruction, our approach requires around 35 *fewer* FFs compared to a PCPI-based implementation. Taking into consideration that the unmodified PicoRV32 has just under 600 FFs, these are considerable savings. Also, when using the PCPI-integrated `SBOX` ISAX, the core's $f_{\max}$ drops from 217 MHz down to 204 MHz, while the SCAIE-V integrated `SBOX` does not slow down the core at all.

5 CONCLUSION

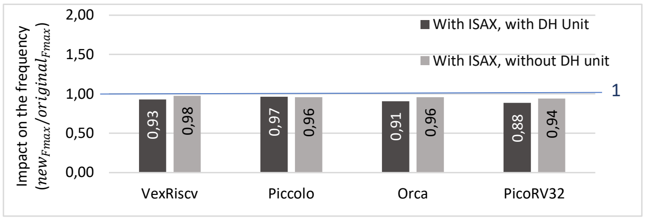
In this work, we demonstrated the capabilities of SCAIE-V, a lightweight interface for adding custom instructions to RISC-V cores, which is portable across very different microarchitectures. The hardware integration layer is automatically generated and tailored to the needs of the actual ISAXes employed, aiming for area-efficient heavily reused hardware resources. The approach is highly flexible, supporting both dynamic as well as static hazard handling approaches.

In future work, we will examine extending SCAIE-V to more complex processors, including dual-issue and out-of-order micro architectures.

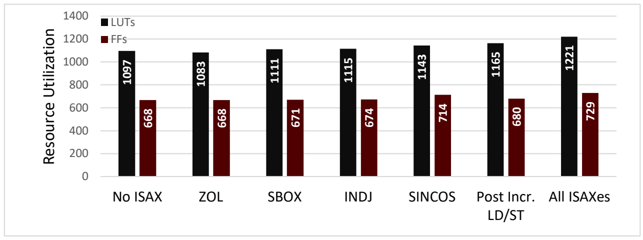
SCAIE-V is an open source tool. The source code is available at: `https://github.com/esa-tu-darmstadt/SCAIE-V`.



(a) Resource usage with and without the SCAIE-V interface



(b) Clock Frequency with and without SCAIE-V



(c) Per-ISAX overhead of SCAIE-V for VexRiscv

Figure 5: Evaluation results on Xilinx Zynq 7020 FPGA

Acknowledgment: This work has been developed in the ZuSE project Scale4Edge. Scale4Edge is funded by the German Ministry of Education and Research (BMBF) (reference numbers: 16ME0122K-16ME0140+16ME0465). The authors are responsible for the content of this publication. Parts of the work have also been supported by BMBF and the Hessian Ministry of Higher Education, Research, Science and the Arts (HMWK) within their joint support of the National Research Center for Applied Cybersecurity *ATHENE*. The authors would also like to thank MINRES Technologies, a commercial provider of functional-safety aware RISC-V cores supporting the SCAIE-V interface, for the fruitful collaboration.

REFERENCES

[1] Codasip. 2021. Codasip. https://codasip.com/. Online; accessed Sept. 2021.
[2] Codasip. 2021. Codasip Tools. https://design-reuse.com/articles/46237/extending-risc-v-isa-with-a-custom-instruction-set-extension.html Online; accessed Sept. 2021.
[3] Div. 2021. Andes Custom extension. http://www.andestech.com/en/products-solutions/andes-custom-extension/
[4] Div. 2021. ORCA Lightweight Vector Extensions. https://riscv.org/wp-content/uploads/2016/07/Tue1515_orca_lve.pdf. Online; accessed Sept. 2021.
[5] Div. 2021. PicoRV32. https://github.com/cliffordwolf/picorv32. Online; accessed Sept. 2021.
[6] Div. 2021. RISC-V Cores. https://github.com/riscv/riscv-cores-list
[7] Div. 2021. RISC-V Debug Spec. https://github.com/riscv/riscv-debug-spec
[8] Div. 2021. RISC-V Formal Verification Framework. https://github.com/SymbioticEDA/riscv-formal
[9] Div. 2021. RISC-V Trace Specification. https://github.com/riscv/riscv-trace-spec
[10] RISC-V Foundation. 2021. RISC-V: The Free and Open RISC Instruction Set Architecture. https://riscv.org/
[11] Carsten Heinz, Yannick Lavan, Jaco Hofmann, and Andreas Koch. 2019. A Catalog and In-Hardware Evaluation of Open-Source Drop-In Compatible RISC-V Softcore Processors. In *IEEE Proc. International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE.
[12] OpenHW Group. 2021. eXtension Interface. https://cv32e40x-user-manual.readthedocs.io/en/latest/x_ext.html. Online; accessed Sept. 2021.
[13] Ross Porter, Sam Morgan, and Morteza Biglari-Abhari. 2019. Extending a Soft-Core RISC-V Processor to Accelerate CNN Inference. In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*. 694–697. https://doi.org/10.1109/CSCI49370.2019.00130
[14] Suleyman Savas, Zain Ul-Abdin, and Tomas Nordström. 2018. Designing Domain-Specific Heterogeneous Architectures from Dataflow Programs. *Computers* 7 (04 2018), 27. https://doi.org/10.3390/computers7020027
[15] Pasquale Davide Schiavone, Davide Rossi, Antonio Pullini, Alfio Di Mauro, Francesco Conti, and Luca Benini. 2018. Quentin: an Ultra-Low-Power PULPissimo SoC in 22nm FDX. In *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*. 1–3. https://doi.org/10.1109/S3S.2018.8640145
[16] Christoph Spang, Florian Meisel, and Andreas Koch. 2021. RT-LIFE: Portable RISC-V Interface for Real-Time Lightweight Security Enforcement. In *Intl. Conf. on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS)*. Springer International Publishing.
[17] Christoph Spang, Florian Meisel, and Andreas Koch. 2021. RT-LIFE: Portable RISC-V Interface for Real-Time Lightweight Security Enforcement. In *Intl. Conf. on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS)*. Springer International Publishing.
[18] VectorBlox Computing Inc. 2021. FPGA-Optimized lightweight Vector Extensions for VectorBlox ORCA. https://riscv.org/wp-content/uploads/2016/07/Tue1515_VectorBlox_ORCA_with_LVE.pdf. Online; accessed Nov. 2021.
[19] Ning Wu, Tao Jiang, Lei Zhang, Fang Zhou, and Fen Ge. 2020. A Reconfigurable Convolutional Neural Network-Accelerated Coprocessor Based on RISC-V Instruction Set. *Electronics* 9, 6 (2020). https://doi.org/10.3390/electronics9061005