Automatically Retargeting Hardware and Code Generation for RISC-V Custom Instructions

Kari Hepola, Tharaka Ranasinghe Arachchige, Joonas Multanen, and Pekka Jääskeläinen

***Abstract*—Custom instruction (CI) set extensions are beneficial for increasing performance and energy efficiency in a set of target applications. For rapid prototyping of these types of application-specific processors, designers leverage hardware (HW)/software (SW) co-design to create hardware implementations and retarget the compiler using a high-level description of the instruction set extension. Ideally, the architecture description should be flexible enough to support both hardware generation and compiler retargeting from the same description format. The challenge with these methods lies in coupling hardware extensions with the processor core, because using microarchitecture-specific interfaces leads to low design reuse and increased verification effort. To mitigate these challenges, we introduce a HW/SW co-design toolset capable of adapting to a user-defined architecture description that captures the instruction set extension semantics. Based on the architecture description, the toolset can both retarget the compiler and generate co-processors interfacing with the Core-V eXtension interface (CV-X-IF) and Rocket custom co-processor interface (RoCC) protocols that are widely used standard interfaces for RISC-V processors. To demonstrate our methods, we integrate the co-processors with two different variations of CVA6 and Rocket core. The resulting execution time reduction is up to 40% on average, with an area overhead of 8% for the CVA6. For the Rocket core, the execution time reduction is 27% with a 6% area overhead.**

***Index Terms*—Application-specific instruction-set processor (ASIP), co-processors, hardware (HW)/software (SW) co-design, RISC-V.**

I. Introduction

THE recent interest and increase in the amount of open-source processor IPs is expected to facilitate the design of computer systems as hardware and software become more accessible. This trend is linked to the open RISC-V instruction set specification, as it has seen wide adoption in the form of released open-source IPs and a tool ecosystem. Increasing the efficiency of embedded systems requires consideration of domain and application-specific functionalities. This is taken into account in the specification of the RISC-V instruction set by reserving the instruction encoding space for custom instructions (CIs).

Received 10 March 2025; revised 11 June 2025; accepted 30 June 2025. Date of publication 16 July 2025; date of current version 29 September 2025. This work was supported in part by the Academy of Finland under Grant 353199 and in part by the Key Digital Technologies Joint Undertaking (KDT JU) through the Together for RISc-V Technology and ApplicatioNs (TRISTAN) Project under Grant 101095947. *(Corresponding author: Kari Hepola.)*

Kari Hepola, Tharaka Ranasinghe Arachchige, and Joonas Multanen are with the Faculty of Information Technology and Communication Sciences, Tampere University, 33014 Tampere, Finland (e-mail: kari.hepola@tuni.fi).

Pekka Jääskeläinen is with the Faculty of Information Technology and Communication Sciences, Tampere University, 33014 Tampere, Finland, and also with Intel Finland Oy, 02160 Espoo, Finland.

Digital Object Identifier 10.1109/TVLSI.2025.3586902

While tightly coupled, *in-pipeline* implementation of CIs achieves precise control and flexibility in the type of instruction extensions that can be implemented, a generic *interface-based* co-processor or accelerator approach allows separating implementation of the main core and the co-processor. In scenarios where the main core has to be rigorously verified and possibly certified to meet specific standards, it is desirable to do this process only once, as it is time-consuming and costly. In the worst case, adding in-pipeline CI extensions requires reverification and recertification of the main core. In the interface-based approach, the generic interface is added to the main core. This allows using the core modularly as an IP block, allowing separately developed, verified, and certified co-processors to be integrated for domain or application-specific accelerators.

Extending the hardware capabilities with CIs should go hand-in-hand with compiler support for high-level programs. Ideally, the user should not have to modify application source code for the new CIs, but the compiler should be able to automatically retarget to the user-defined architecture description. The benefit of this type of hardware (HW)/software (SW) co-design is that the CIs can be described both to the compiler and hardware generation tools using the same architecture description. In this work, we demonstrate our compiler and hardware generation methods using an architecture description template built on top of directed acyclic graph (DAG) and hardware description language (HDL)-based operation descriptions. The hardware tools can generate co-processors that are compatible with the Core-V eXtension interface (CV-X-IF) [1] and Rocket custom co-processor interface (RoCC) [2] protocols that are two widely adopted standard interfaces for RISC-V processors.

This article proposes open-source tools that perform and contribute the following.

1) Generation of RISC-V CI register transfer level (RTL) based on DAG descriptions and HDL snippets.
2) CV-X-IF and RoCC interface-based integration of the generated CIs.
3) A retargetable RISC-V compiler that can automatically map high-level instructions to the generated CIs based on a user-defined architecture description.

The article extends our preliminary work [3] with the following new contributions: 1) capability to integrate CI accelerators via the RoCC interface; 2) utilization of the ratified version 1.0.0 of the CV-X-IF specification; 3) integration of HDL operation snippets with operation DAGs in co-processor generation; and 4) extended evaluation with a

superscalar-based CVA6 variation and the Rocket core to show adaptability of the proposed methods over a spectrum of different RISC-V microarchitectures.

II. Related Works

This section reviews the existing interface descriptions used to integrate RISC-V CI implementations, tools, and frameworks to generate implementations, and previous works on compilers that can automatically target the described CIs.

A. CI Interfaces

Pico co-processor interface (PCPI) [4] is an interface integrated into the PicoRV32 core. When an unsupported instruction is encountered, it is offloaded via the PCPI interface. If no external core accepts the instruction, an illegal instruction exception is raised after 16 clock cycles. Otherwise, an accepting core signals when it accepts and is finished with the instruction.

TIGRA [5] is a CI interface that tightly integrates into the pipeline stages of RISC-V processors. TIGRA allows adding CI implementations outside of the processor core without adding extra latency, but only supports *R-type* instructions of the RISC-V ISA.

The rocket chip generator specifies RoCC that can be used to offload instructions to external co-processors or accelerators. An RoCC accelerator is instructed by offloading the instruction bits, as well as the source operands and the destination register.

CV-X-IF is a specification for integrating co-processors via a fixed interface. Upon encountering an instruction it cannot decode, the main core offloads it via the co-processor interface. In case the instruction is not accepted by a co-processor, the main core raises an illegal instruction exception.

CCOPI [6] is an interface developed for the VexRiscv core. CIs are registered into the core's decoder and upon encountering one, the core issues it to a custom co-processor.

In the nuclei instruction co-unit extension (NICE) [7] interface, the main core decodes instructions and marks them as NICE instructions when appropriate.

SCAIE-V [8] is an interface for incorporating in-pipeline instruction set extensions. It is intended as a generic interface, which was shown by implementing CIs to four different RISC-V cores.

B. Interface-Based Accelerator Generation

This section reviews tools that can be used to generate interface-based accelerators or co-processors. Individual implementations of CI accelerators are not included in this review, as the focus is on generator tools.

The rocket chip generator [2] toolset is able to integrate accelerators using the RoCC interface. CIs of the RoCC co-processor are described in Chisel [9].

Egert [10] described FRANCIS-V, a tool to generate CV-X-IF-based accelerators and co-processors. The tool assumes that the processor core has an existing co-processor interface, i.e., CV-X-IF. The tool generates a wrapper for user-defined custom operations, which are defined as HDL. The tool is not openly available.

Longnail [11] is a high-level synthesis (HLS) toolset to generate RISC-V CIs based on coreDSL descriptions. The tool generates RTL for the CIs and an SCAIE-V interface wrapper. The tool is not openly available.

SCAIE-V [8] is an IF description and a tool that tightly couples the CI implementations with the target core. Whereas other generator tools reviewed here assume that the interface exists on the target core, SCAIE-V modifies the core depending on the user-defined CI types, such as control transfer operations.

Table I presents a summary of the generator tools. Unlike the previously proposed tools, our toolset can adapt to multiple co-processor interfaces by generating both CV-X-IF and RoCC-compatible accelerators. It also includes a retargetable compiler, which automatically maps high-level code to the generated CI implementations.

C. Compilation With CIs

Codasip Studio [12] incorporated a retargetable LLVM-based compiler. This allows operations to be mapped to the custom hardware. Similarly, the Andes tools and application-specific instruction-set processor (ASIP) designer [13] feature a retargetable compiler. All three are commercial tools.

Seal5 [14] is a tool that patches LLVM to support RISC-V CIs. Operation descriptions are created with coreDSL and transformed into LLVM TableGen patterns, allowing them to be mapped from high-level source code. Subsequently, the patches are added to the RISC-V LLVM backend to achieve retargeting to CIs after the recompilation and installation of the LLVM toolchain, whereas this work proposes a compiler that can modularly generate and dynamically load new targets without requiring recompiling or installing the toolchain itself.

III. OpenASIP

OpenASIP [15], [16] is an open-source HW/SW co-design toolset for ASIPs. It uses an architecture definition file to describe the processor's instruction set, which in turn is used as input to the compiler and hardware tools to achieve retargetable compilation and automatic hardware generation. OpenASIP supports generating RISC-V processors with user-defined CI set extensions, but the custom hardware is tightly coupled to the microarchitecture-specific interface and cannot be directly used with other RISC-V processors. Another limitation is with the RISC-V compiler, which achieves retargetability via loading a custom LLVM pass for processing inline assembly blocks during the pre-emit stage of the code generation flow. While this approach achieves fast compile times for custom architectures, it does not support automatic instruction selection due to the lack of custom backend generation.

A. Operation Descriptions

Description of user-defined operation semantics is important for hardware generation, and, on the other hand, for achieving automatic compiler retargetability. OpenASIP has a modular

TABLE I
COMPARISON OF RISC-V TOOLS THAT GENERATE INTERFACE-BASED CI EXTENSIONS

	Rocket Chip Generator [2]	**SCAIE-V** [8]	**FRANCIS-V** [10]	**Longnail** [11]	**This work**
Year	2016	2022	2023	2024	2025
Availability	open-source	open-source	N/A	N/A	open-source
Supported IF	RoCC	SCAIE-V	CV-X-IF	SCAIE-V	CV-X-IF, RoCC
Language	Chisel	N/A	HDL	CoreDSL	DAG and/or HDL
IF RTL generation	✓	✓	✓	✓	✓
Operation RTL generation	✓	×	×	✓	✓
Multi-cycle operations	✓	✓	×	✓	✓
Compiler Support	×	×	×	×	✓

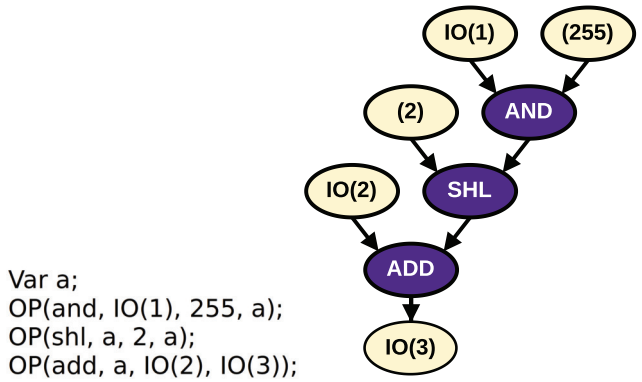


Fig. 1. (a) Code description and (b) DAG of a CI.

```
op3 = 0;
for (i = 0; i < 8; i++) begin
  op1_i = op1[4*i +: 4];
  op2_i = op2[4*i +: 4];
  mul_i = op1_i * op2_i;
  op3 = op3 + mul_i;
end
```

Fig. 2. Verilog snippet for an int4 dot product operation.

architecture definition that lists supported instructions, whose semantics are described in separate databases for high reuse. In the scope of the toolset, two approaches exist for adding new operations: DAGs and hardware HDL snippets. Both of these operation description approaches are used to describe dataflow, since they lack functionality for directly manipulating the processor state, such as the program counter. On the other hand, the co-processor interfaces target mainly dataflow acceleration, which can be described with OpenASIP's operation descriptions.

1) Directed Acyclic Graphs: DAGs describe the operation semantics as a chain of basic operations. These operation nodes and their edges that describe dependencies form an acyclic graph structure. An example of this type of graph is illustrated in Fig. 1 that describes a CI combining and shift-and-add operations. Using this type of description has the benefit of easy semantic extraction.

2) HDL Snippets: While DAGs are an efficient way of describing the semantics, they lack the flexibility of describing complex instruction structures. HDL snippets allow the user to describe the operation semantics directly with HDLs, such as Verilog and VHDL, for more fine-grained control. These types of descriptions make it challenging to automatically extract the semantic features from the description, which is why its use is targeted for hardware generation. Fig. 2 illustrates an example of an int4 dot product instruction whose semantics are described directly with Verilog. Predefined signal names, such as *op{1,2,3,n}*, are used to interface with the instruction operands.

TABLE II
ROCC INTERFACE SIGNALS. DIRECTIONS ARE REPORTED FROM THE CORE'S POINT OF VIEW

Signal	**Direction**	**Description**
Command Interface		
valid	out	Valid command from CPU
instr	out	Instruction bits
rs1_bits	out	rs1 operand data
rs2_bits	out	rs2 operand data
ready	in	Accelerator accepts command
Response Interface		
ready	out	CPU accepts response
valid	in	Response valid
rd_bits	in	Result register index
resp_bits	in	Result bits

IV. CO-PROCESSOR INTERFACES

Co-processor or accelerator interfaces specify a standardized method for coupling application-specific IP with processor cores. Two widely used accelerator interfaces for RISC-V are RoCC and CV-X-IF.

A. RoCC

The RoCC interface is used to tightly couple accelerators via a direct signal interface with processor cores. This interface is implemented as part of the Rocket Chip Generator [2] with existing implementations for the Rocket and BOOM [17] cores. The software interface to the accelerator is implemented in the form of CIs by offloading instructions encoded with custom opcode fields to the accelerator.

The hardware interface, listed in Table II at its core uses a simple ready-valid protocol. The main interface is divided into a command and response interface, where first the core

issues instructions via the command interface. This is done by asserting valid with the decoded register operands and the instruction bits. After the accelerator has executed the command, it returns the optional result operand with the register index via the respond interface. In addition to this basic functionality of offloading instructions, RoCC has support for its own accelerator memory interface and floating point unit interface for executing floating point operations in the processor core.

B. CV-X-IF

CV-X-IF is another standard interface for coupling accelerators with processor cores. CV-X-IF functions by offloading instructions that it does not recognize as legal instructions to the accelerator. The hardware interface functionality, however, is more complex than with RoCC, which allows a higher degree of flexibility in the offloading of instructions with support for out-of-order execution.

Table III lists the subinterfaces and signals of the CV-X-IF. For all interfaces, an instruction and a hart ID are issued to follow the offloading of instructions, which makes it possible to implement out-of-order capabilities. The core initiates offloading via the issue interface when it decodes an unrecognized instruction. It does this by asserting valid and passing the instruction bits to the accelerator that can accept the instruction if it is implemented by the accelerator. If the accelerator does not recognize the instruction, it can deassert the accept signal with the ready assertion, which will lead the processor core to resume exception routines. Otherwise, the accelerator will decode the instruction and assert accept and set the response signals to correct values.

The register interface is used to transfer the decoded register operands to the accelerator. The commit interface signals the accelerator whether the issued instruction is to be committed. This is done by deasserting the *commit_kill* signal. The CV-X-IF standard also allows to kill instructions in batches, by squashing all instructions that were issued after the killed instructions in the pipeline. The result interface is used by the accelerator to write back the result operand to the register file after it has received the commit information, which can also be done in batches.

TABLE III
CV-X-IF SIGNALS. DIRECTIONS ARE REPORTED FROM THE CORE'S POINT OF VIEW

Signal		Description
Issue Interface		
issue_valid	out	Issue request valid
issue_ready	in	Issue request ready
instr	out	Offloaded instruction
hartid	out	Thread id
id	out	Id of the offloaded instruction
accept	in	Signals whether coprocessor accepts the instruction
writeback	in	Signals whether the instruction performs a writeback
register_read	in	States whether the instruction requires specific registers to be read
Register Interface		
register_valid	out	Signals that the CPU provides register operands
register_ready	in	Register request ready
hartid	out	Thread id
id	out	Id of the offloaded instruction
rs	out	Register operand values
rs_valid	out	Validity of register operands
Commit Interface		
commit_valid	out	Commit request valid. CPU has valid commit information
hartid	out	Thread id
id	out	Id of the offloaded instruction
commit_kill	out	Signals whether the instruction is squashed or commited
Result Interface		
result_valid	in	Signals that the coprocessor has a valid result
result_ready	out	Signals that the CPU accepts the result
hartid	in	Thread id
id	in	Id of the offloaded instruction
data	in	Output data
rd	in	Output register index
we	in	Signals a writeback

V. COMPILER

A retargetable compiler is a key tool for efficient utilization of CIs, as otherwise application developers must rely on methods such as inline assembly, which reduces code portability. We build our retargetable compiler framework on top of the LLVM [18] toolchain, which is a highly modular compiler framework that uses an intermediate representation for performing target-independent code optimization. The benefit of this structure lies in its expandability of the compiler frontend and backend for different languages and target architectures. LLVM supports RISC-V as one of its targets, which is the backend of interest in the scope of this work.

LLVM utilizes a domain-specific language, *TableGen*, for describing the target architectures. This includes programming interface-specific features such as instruction semantics and programmer-visible registers. The TableGen descriptions can be automatically lowered into C++ source code that describes instruction selection and register allocation methods, with the goal of increase in code reuse between different targets.

A. Automatic Retargeting

Traditionally, new targets are added by tightly coupling their descriptors and static source code to the compiler source tree and compiling them together with the entire toolchain. While this is sufficient for fixed architectures, it quickly becomes a bottleneck when incorporating user-defined CI-set extensions.

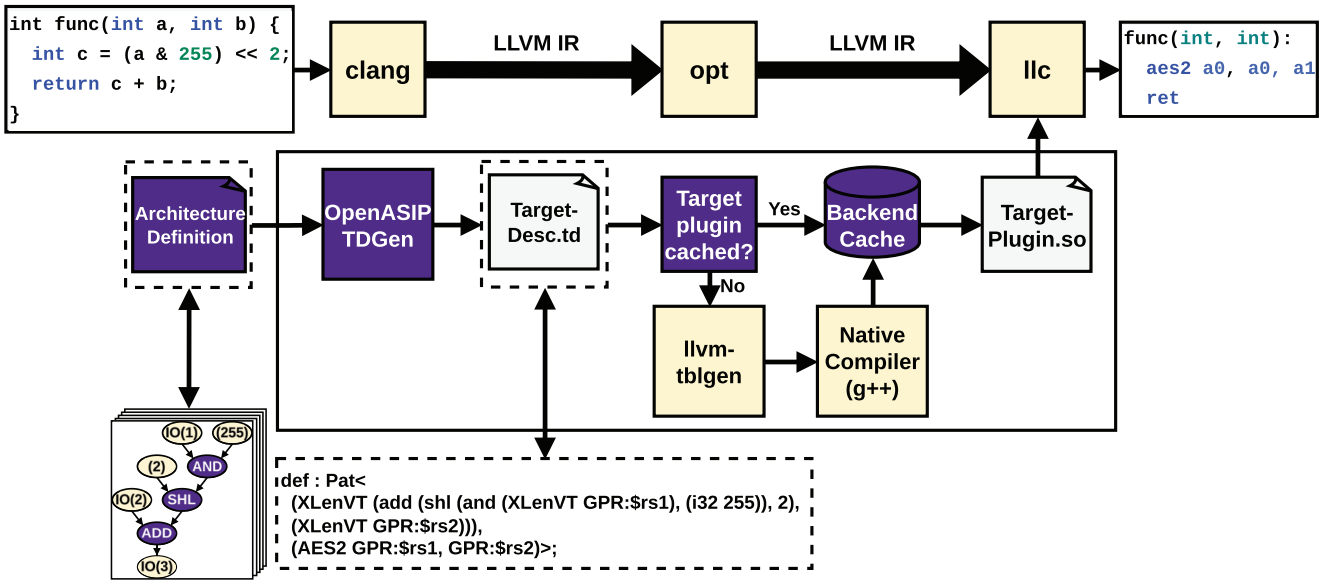


Fig. 3. RISC-V compiler achieves retargetability by generating extensions to the LLVM RISC-V target descriptor based on the architecture definition and compiling it into a dynamic library. This target plugin is dynamically loaded with llc to automatically select user-defined instructions during compilation.

In this work, we eliminate this issue by extending LLVM's backend driver, llc, to support dynamic loading and registering of targets. This makes it possible to decouple targets from the compiler toolchain and compile them out-of-tree into a dynamic library. Dynamic loading also enables the possibility of caching the targets for fast retargeting over multiple custom architectures of interest.

Fig. 3 illustrates the compiler structure. As input, the compiler is given a program written in C and an architecture definition file that describes the instruction set. The program is lowered into an intermediate representation using clang [19], after which target-independent optimizations are performed with opt. After this stage, the intermediate representation can be lowered into target-specific machine code.

Our compiler implementation generates target descriptor extensions to the upstream RISC-V target based on the architecture definition. First, we add new instruction definitions for all user-defined CIs. We extend OpenASIP with a tool, *RISCVTDGen*, for transforming the OpenASIP DAG-based instruction descriptions to the LLVM DAG format. Compilation of target modules is costly, ranging from tens of seconds to multiple minutes, depending on the host machine. The isolated compilation of the target module, however, is significantly faster than a rebuild of the entire compiler framework. In addition, target module decoupling enables an efficient caching mechanism, since the target modules have a relatively small size (~8 MB), we can easily store the unique modules on the host system, which would not be possible if the targets were tightly coupled to the compiler framework. The caching mechanism inspects hash values of the generated target descriptor extensions. In case of a hit, the compiler driver can fetch the dynamic library from the cache and load it with llc. Otherwise, TableGen is used to generate the backend source code from the target descriptors, which is subsequently compiled into a dynamic library with the static source code.

```
#define _OA_RV_DPI4(i1, i2, o1) do {
int __oa_op_output_1 = (int)0;
asm ("dpi4 \%0, \%1, \%2":
  "=r"( __oa_op_output_1):
  "r"((int)(i1)),
  "r"((int)(i2)));
o1 = __oa_op_output_1;
} while(0)
```

Fig. 4. Automatically generated intrinsic definition that can be invoked from C code.

B. Intrinsics Generation

As explained in Section V-A, the retargetable compiler is only able to extract instruction semantics from the operation DAGs. Not all instructions, however, can be described as DAGs. For instance, code patterns with loops typically fall in this category if their representation cannot be simplified. Another issue is the use of complex semantics, which can cause the heuristic-based instruction selectors to fail to recognize the pattern, especially in large basic blocks. Intrablock instructions that operate over basic block boundaries are also challenging for automatic instruction selection [20]. Aside from the instruction selection capabilities, it is difficult to achieve automatic retargeting for data types not natively supported by the compiler, such as subbyte types popular in resource-constrained machine learning applications.

Due to these challenges, it is beneficial to implement more fine-grained control over the code generation to enable high quality of results with complex CIs at the cost of code portability. We do this by automatically generating C-headers of the possible instruction *intrinsics* based on the architecture definition when the compiler is invoked, which allows an easier interfacing with the CIs compared to writing inline assembly manually. Fig. 4 describes the structure of the generated macro function, which wraps the inline assembly block. During the LLVM backend generation, we add definitions for all instructions even if the pattern is not available, which makes

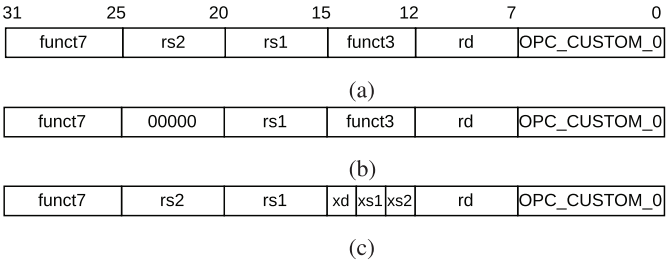


Fig. 5. Different CI formats used with the co-processors. (a) R-format custom template used with CV-X-IF. (b) Unary custom template used with CV-X-IF. (c) RoCC custom format that explicitly encodes the used register operands in the funct3 field.

it possible to use inline assembly directly without adding custom lowering passes as done previously with OpenASIP's RISC-V compiler.

C. Instruction Formats

The RISC-V instruction set specification allows user-defined CIs by reserving custom opcode fields. Since the CV-X-IF requires to use the standardized bit field positions for the register operands in the instruction format, we utilize the R-format to describe our CIs, as illustrated in Fig. 5. To preserve opcode field encodings, we use the same variation for the unary operations with the second input operand encoded to the zero register. The RoCC interface specifies a standardized format for the co-processors where the used register operands are encoded in the instruction. We can leverage unary instructions similar to the CV-X-IF by encoding the unused register operand to zero and modifying the xs2 encoding field. Since the custom opcode field reserves 7 bits from the instruction word, we do not use compressed formats due to the limited encoding space.

VI. Hardware Generation

The hardware description of the co-processor is generated by implementing the operations into a custom function unit. Subsequently, the generated function unit is integrated with the appropriate interface by instantiating it in a wrapper that implements the protocol-specific control logic, while the function unit itself implements the datapath and register pipeline of the CIs. Fig. 6 depicts the generation flow from the designer's perspective. An architecture description with the custom function unit is used as input to the hardware generation tool, which creates the hardware description using the selected co-processor interface.

A. RoCC Generator

The command and response interfaces are the main channels for the communication between the co-processor and host core, as illustrated in Fig. 7(a). The wrapper connects the register data from the command interface to the input operands of the function unit and the output of the function unit to the destination register data of the response interface. In addition, the wrapper extracts the operation encoding from the interface and connects it to the function unit's instruction decoder that

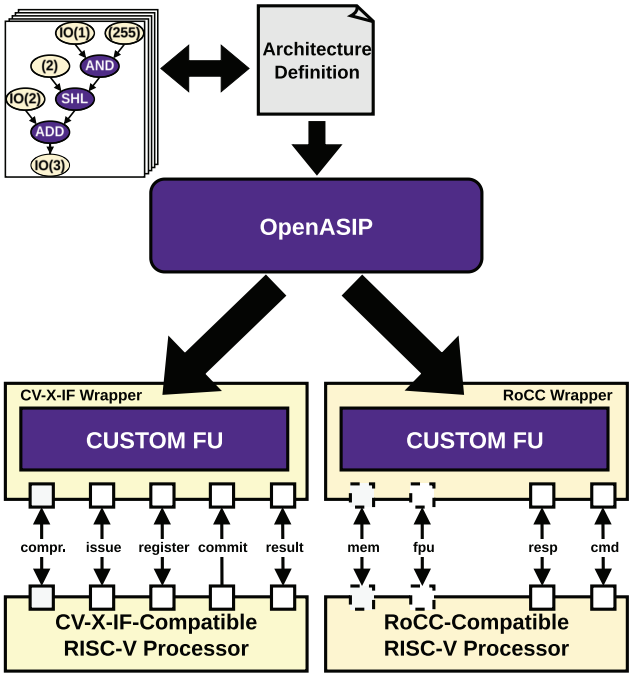


Fig. 6. Custom function unit implementing user-defined instructions is generated based on the architecture definition. The function unit can be embedded into a predesigned wrapper for integrating it through either the CV-X-IF or RoCC interface.

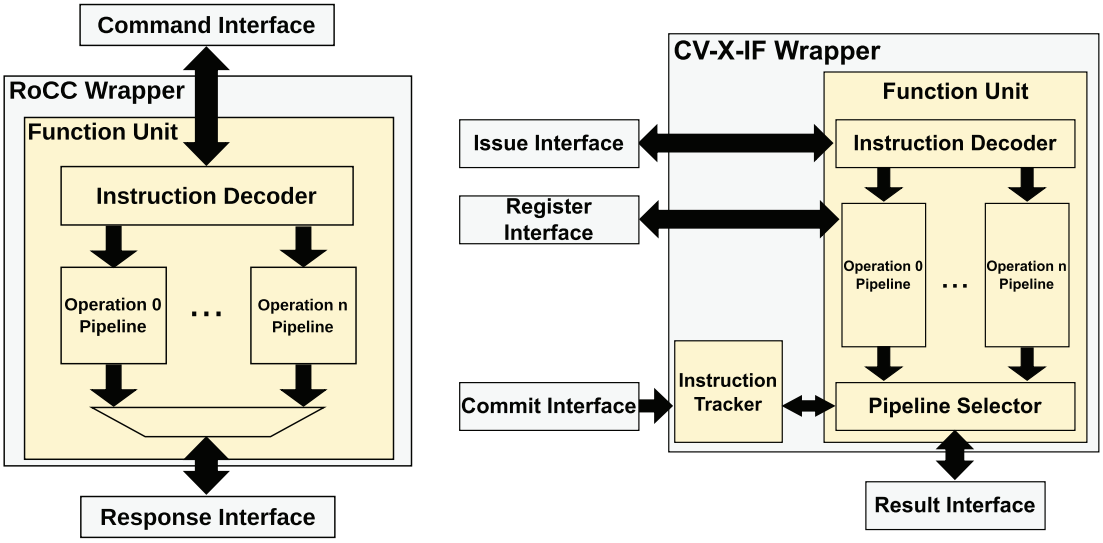
controls the operation pipelines. We extend the function unit pipeline with a valid signal that is propagated into the function unit output with the result operand to signal valid results to the host core based on the pipeline latency of the operation.

We integrate the co-processor to the Chipyard Chisel configuration as a closed-box SystemVerilog component. In this phase, we define the co-processor's opcode encoding, which in turn generates the routing logic from the processor pipeline to the co-processor for instructions matching the encoding. When the processor system is generated to SystemVerilog, the custom co-processor is automatically instantiated through the RoCC interface as part of the Chipyard processor tile.

B. CV-X-IF Generator

CV-X-IF is a more advanced protocol compared with the RoCC interface, which subsequently requires more complex control logic for the co-processor, whose structure is illustrated in Fig. 7(b). The issue interface is connected to the instruction decoding mechanism of the function unit. The instructions offloaded through the issue interface are accepted based on the decoding information and the availability of the resources within the function unit, which is implemented as part of the wrapper's control logic. The accepted instructions are appended with an issue ID to the operation pipeline, which is derived by combining the hart ID and the instruction ID, to keep track of the instructions being executed.

The register operands from the register interface are connected with the operand data inputs of the function unit. Currently, the co-processor supports only the `X_ISSUE_REGISTER_SPLIT = 0` configuration, which conveys the register operands are available at the same cycle as



(a) (b)

Fig. 7. Implementation of the generated (a) RoCC and (b) CV-X-IF co-processors. Due to the explicit commit signaling and out-of-order completion, the CV-X-IF co-processors require an instruction tracker for storing information of the pending instructions.

the issue transaction. The function unit output is connected to the result interface.

Since the CV-X interface enables the possibility to signal the commit and squash information to the co-processor, we add an instruction tracker component to keep track of the instructions being executed by the co-processor. The instruction tracker connects to the commit interface for extracting commit and squash information together with the instruction IDs. At the last stage of the instruction execution, the function unit uses the information stored in the tracker to write back the results associated with the committed issue IDs to the result interface.

Furthermore, to enable the possibility to have multiple operations with different latencies, the last stage of execution pipelines within the function unit is connected to a round-robin selector to select a result of an issue, which is signaled as committed by the commit interface. Ultimately, the committed results should have to be stored until the host is ready to accept the write back. However, to minimize the complexity, the operation pipeline is stalled until the host is ready to accept the result, and the selector keeps selecting the committed available result in the next pipeline, if there are multiple operations in the function unit.

VII. Evaluation

We utilize the *crc*, *aha-compress*, and *nettle-aes* applications from the BEEBS [21] benchmark suite and 4-bit 128×128 matrix multiplication kernel to demonstrate the toolset. Using subbyte datatypes in these types of workloads are interesting for running machine learning models on resource-constrained platforms, such as edge devices [22], [23]. The BEEBS applications were chosen based on their characteristics for instruction set customization, since these programs have long arithmetic chains that are suitable for targeting application-specific instructions.

TABLE IV
Features of the Evaluated Processors

	CVA6	Rocket
Stages	6	5
Pipeline	in-order issue with OoO execute	in-order
Issue width	single-issue | dual-issue	single-issue
Instruction set	RV32	RV64
Extensions	c, m, zba zbb, zbs, zbc	c, m

A. Evaluated Processors

We evaluate the co-processor generation flow using three different RISC-V processors, whose features are listed in Table IV. We integrate our CV-X-IF co-processor generation flow with the CVA6 [24] (cv32a65x) that is an application-class RISC-V core with out-of-order execute implementing the RV32IMC variation of RISC-V ISA with the zba, zbb, zbs, and zbc standard extensions. CVA6 implements the ratified version 1.0.0 of the CV-X-IF specification. We use both the single- and dual-issue variations of the CVA6 core to show that our methods can benefit a wide spectrum of microarchitectures. To demonstrate the RoCC co-processor generation flow, we use the Chipyard system-on-chip (SoC) generation framework [25]. Chipyard utilizes Chisel to integrate and describe different IP components, which allows creating complex SoC implementations. In this work, we utilize the framework to integrate RoCC-compatible co-processors with the Rocket core as a part of a Chipyard processor tile.

B. CIs

Fig. 8 lists the added CIs. These include bit manipulation chains that have a small hardware overhead but can lead to significant performance gains when accelerated. For the matrix multiplication kernel, we add a packed SIMD int4 dot product

$$\textbf{aes0}: \quad ((in \gg 6) \wedge 1020) + in2$$
$$\textbf{aes1}: \quad ((in \gg 14) \wedge 1020) + in2$$
$$\textbf{aes2}: \quad ((in \wedge 255) \ll 2) + in2$$
$$\textbf{aha0}: \quad x \leftarrow in \oplus (in \ll 1)$$
$$x \leftarrow x \oplus (x \ll 2)$$
$$x \leftarrow x \oplus (x \ll 4)$$
$$x \leftarrow x \oplus (x \ll 8)$$
$$out \leftarrow x \oplus (x \ll 16)$$
$$\textbf{dpi4}: \quad \sum_{i=0}^{7} in_i \cdot in2_i$$
$$\textbf{crc0}: \begin{cases} (in \ll 1) \oplus 4129, & (in \wedge 0x8000) \neq 0 \\ in \ll 1, & (in \wedge 0x8000) = 0 \end{cases}$$

Fig. 8. Included CIs.

instruction that is capable of executing an eight-element dot product in two clock cycles. For all instructions except the subbyte dot product, we are able to describe the instructions directly as DAGs, which adds support for automatic instruction selection.

For the CVA6 co-processor, we add an extra register to the output to remove the pipeline selector logic from the commit path. This makes the execution latency three for the dot product operation and two for all other operations. The out-of-order capabilities of the CVA6 can, however, compensate for this delay. For the Rocket core, we can meet sufficient timing without adding an extra register to the output, which reduces the execution latency by one for all instructions.

The CIs are connected to a 32-bit datapath logic. Even though the Rocket core is a 64-bit processor, the applications, apart from the matrix multiplication kernel, target 32-bit data types, which makes it desirable to match the co-processor datapath to this width. The dot product instruction could support a wider packed SIMD width for the Rocket core, but we choose a 4×8 element length to allow comparison with the CV-X-IF co-processor. Increasing the SIMD length would increase the hardware overhead by additional logic and also require deeper pipelining to avoid timing bottlenecks.

C. Retargetability

The compiler, described in Section V, is able to automatically select all of the DAG-based instructions automatically. To achieve the best level of instruction selection quality, we disabled loop unrolling for these programs. The challenge with this optimization lies in the expansion of the loop body size, which makes it less likely for the instruction selector to match the pattern from the generated code, especially for long instruction patterns. For the dot product instruction, we add a built-in intrinsic call that replaces the inner loop that iterates over the 4-bit elements of the matrix.

D. Synthesis

We utilize Cadence Genus and the open-source ASAP7 [26] process design kit to analyze timing and area utilization of the processor implementations with and without the custom co-processors. To more accurately observe the overhead of the co-processor integration on the core's datapath, we synthesize the cache memory elements as closed-box instances to avoid their significant influence on the timing and area results. Each processor configuration is synthesized at its maximum clock frequency, determined through iterative synthesis until timing constraints are met.

TABLE V
SYNTHESIS RESULTS OF THE PROCESSORS

	Superscalar			
	CVA6	CVA6	Rocket	
Area (μm^2)	7197*| 7665†	8596*| 9261†	3762*| 3991†	
Delay (ps)	726*| 779†	780*| 785†	510*| 519†	

* Without the extension.
† With the extension.

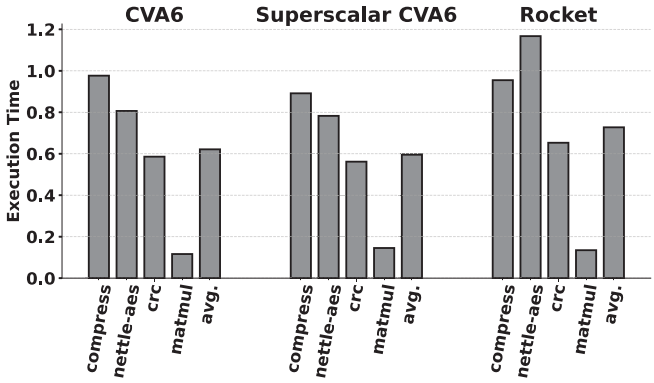


Fig. 9. Execution time reduction for each benchmark and processor over the base instruction set.

Table V lists the synthesis results of the processors. With the single-issue CVA6, the timing overhead of the co-processor integration is approximately 7% due to increased issue logic. The overhead of this could be, however, mitigated with deeper pipelining. With the superscalar variation, the timing overhead is only 1%. The rocket core also experiences only a 2% increase in critical path length, as the co-processor logic is not directly on the critical path. The area overhead is 7% and 8% with the CVA6 processors and 6% with the Rocket.

E. Performance

The relative execution time for the co-processor-accelerated processors over the benchmark set is illustrated in Fig. 9. An interesting observation is that the execution time is increased in the nettle-aes benchmark with the Rocket core when using the instruction set extension. This is because the Rocket core has a long offloading latency to the co-processor that exceeds the latency of the three suboperations of the CI when executed natively on the processor's datapath. Because of this, the co-processor interface implementation of the Rocket is not well-suited for small CIs. For the CVA6 processors, however, the execution time of nettle-aes is reduced approximately 20% due to the faster execution and increased code density.

It should be noted that in all programs, apart from the matrix multiplication kernel, where the inner accelerated loop has a high amount of instruction-level parallelism, the

TABLE VI
CODE DENSITY OVER THE APPLICATION SET COMPARED WITH THE BASE INSTRUCTION SET

	compress	nettle-aes	crc	matmul
Code Size	0.84	0.94	0.92	0.93
Fetched Instruction Bytes*	0.88	0.83	0.54	0.09

* Values are extracted from L1 instruction cache accesses based on the single-issue CVA6 simulations.

superscalar variation of CVA6 benefits more from instruction set customization. The reasoning for this is twofold, as we observe from the synthesis results, the timing overhead of the co-processor is smaller. More interestingly, the instruction extensions utilize serial operation chains that cannot be accelerated with multiissue capabilities. These code patterns can form bottlenecks in the program; on the other hand, the execution time of other program phases can be reduced by instruction-level parallelism. The matrix multiplication kernel benefits the most from the co-processor integration. With the CVA6 processors, the execution time of the matrix multiplication kernel is reduced by 88% and 85%, and 87% with the Rocket core. The average reductions in execution time over all four benchmarks are 38%, 40%, and 27%.

F. Code Density

CI set extensions improve both the static and dynamic code density of the target workloads. Table VI lists the reduction in code size of the user application and the reduction in L1 instruction cache accesses for different workloads. These results follow the same trend as the performance improvements. Since we target program hot spots, the improvement in static code density is smaller than in the dynamic code density. On average, the code size is reduced by 9%, while the amount of fetched instruction bits is reduced by 41%. These results highlight the potential of customization also in reducing the system-level requirements through higher code density, leading to more efficient instruction streams.

The standard compressed instruction set extension has an effect on the amount of code density improvement. Most notably, in nettle-aes, depending on the allocated registers, the suboperations can be compressed, which limits the acquired code density benefit when implementing the arithmetic chain as a CI.

VIII. FUTURE WORK

In the future, it would be beneficial to research methods for fast retargetable instruction set simulation, which would enable a faster prototyping speed. Similar to the compiler work, we could extend existing simulation frameworks, such as qemu [27] or gem5 [28] that already support simulation of standard RISC-V processors. The integration of these tools would be implemented with OpenASIP's operation set abstraction layer, which contains simulation models for user-defined custom operations. OpenASIP has been used to generate SIMD exposed datapath processors [29]. Leveraging this DAG-based flow for RISC-V packed SIMD instructions would further increase the performance and efficiency gain of the customization framework. In addition, packed SIMD operations can be directly used with the existing co-processor interfaces, since they use the scalar register file and do not require extending the data interfaces. Utilizing the RISC-V V extension vector scheme would require extensions to the standard interfaces, since they need to support varying element lengths that are not directly encoded in the instruction. This would require connecting the co-processors with the vector state registers. Moving toward this direction would enable to couple high-performance accelerators to RISC-V processors through a standard interface utilizing very wide vector lengths.

IX. CONCLUSION

This article introduced methods for achieving retargetable compilation and co-processor generation utilizing CV-X-IF and RoCC interfaces using the same architecture description. The benefit of generating CI set extensions with standard co-processor interfaces is that the hardware implementations can be directly coupled with a large spectrum of different compatible RISC-V microarchitectures. In addition, this reduces the verification effort, since no changes are required to the existing datapath of compatible processor cores. We demonstrated the toolset for the Rocket and two CVA6 processor variations. As target applications, we utilized three BEEBS applications and an int4 matrix multiplication kernel. The results show that the compiler is able to automatically select the CIs when the semantics are described in the architecture description. Utilization of the instruction set extension reduced execution time by 27% on average for the Rocket core, with a 6% area overhead. For the CVA6 processors, the execution time was reduced 38% on average for the single-issue and 40% for the superscalar variation, with a 7% and 8% area overhead.

REFERENCES

[1] OpenHW Group. *Core-V EXtension Interface*. Accessed: Feb. 2025. [Online]. Available: https://github.com/openhwgroup/core-v-xif

[2] K. Asanović et al., "The rocket chip generator," Dept. Electrical Engineering and Computer Science, Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2016-17, 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[3] K. Hepola, T. R. Arachchige, J. Multanen, and P. Jääskeläinen, "Fully automatic compiler retargeting and CV-X-IF hardware interface generation for RISC-V custom instructions," in *Proc. IEEE Nordic Circuits Syst. Conf. (NorCAS)*, Oct. 2024, pp. 1–7.

[4] (2015). *PicoRV32—A Size-Optimized RISC-V CPU: Pico Co-Processor Interface (PCPI)*. Accessed: Feb. 2025. [Online]. Available: https://github.com/YosysHQ/picorv32#pico-co-processor-interface-pcpi

[5] B. Green, D. Todd, J. C. Calhoun, and M. C. Smith, "TIGRA: A tightly integrated generic RISC-V accelerator interface," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, Sep. 2021, pp. 779–782.

[6] *CCOPI: Implementing a Custom Coprocessor Interface for VexRiscv*. Accessed: Feb. 2025. [Online]. Available: https://github.com/jens-na/VexRiscv-CCOPI/blob/master/paper/ccopi_paper.pdf

[7] *Nuclei ISA SPEC*. Accessed: Feb. 2025. [Online]. Available: https://doc.nucleisys.com/hbirdv2/core/core.html

[8] M. Damian, J. Oppermann, C. Spang, and A. Koch, "SCAIE-V: An open-source SCAlable interface for ISA extensions for RISC-V processors," in *Proc. 59th ACM/IEEE Design Autom. Conf.*, Jul. 2022, pp. 169–174, doi: 10.1145/3489517.3530432.

[9] J. Bachrach et al., "Chisel: Constructing hardware in a scala embedded language," in *Proc. DAC Design Autom. Conf.*, Jun. 2012, pp. 1212–1221.

[10] F. Egert, "FRANCIS-V: Framework for integrating custom instructions into RISC-V systems," M.S. thesis, Inst. Comput. Technol., Technische Universität Wien, Vienna, Austria, 2023.
[11] J. Oppermann, B. M. Damian-Kosterhon, F. Meisel, T. Mürmann, E. Jentzsch, and A. Koch, "Longnail: High-level synthesis of portable custom instruction set extensions for RISC-V processors from descriptions in the open-source CoreDSL language," in *Proc. 29th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst., Volume 3*, Apr. 2024, pp. 591–606.
[12] *Codasip Studio*. Accessed: Feb. 2025. [Online]. Available: https://codasip.com/products/codasip-studio/
[13] *ASIP Designer: Application-Specific Processor Design Made Easy*. Accessed: Feb. 2025. [Online]. Available: https://www.synopsys.com/dw/doc.php/ds/cc/asip-brochure.pdf
[14] P. Van Kempen, M. Salmen, D. Mueller-Gritschneder, and U. Schlichtmann, "Seal5: Semi-automated LLVM support for RISC-V ISA extensions including autovectorization," in *Proc. 27th Euromicro Conf. Digit. Syst. Design (DSD)*, Aug. 2024, pp. 335–342.
[15] P. Jääskeläinen, T. Viitanen, J. Takala, and H. Berg, "HW/SW co-design toolset for customization of exposed datapath processors," in *Computing Platforms for Software-Defined Radio*. Cham, Switzerland: Springer, 2017, doi: 10.1007/978-3-319-49679-58.
[16] K. Hepola, J. Multanen, and P. Jääskeläinen, "OpenASIP 2.0: Co-design toolset for RISC-V application-specific instruction-set processors," in *Proc. IEEE 33rd Int. Conf. Appl.-Specific Syst., Archit. Processors (ASAP)*, Jul. 2022, pp. 161–165.
[17] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "SonicBOOM: The 3rd generation Berkeley out-of-order machine," in *Proc. 4th Workshop Comput. Archit. Res. RISC-V*, vol. 5, 2020, pp. 1–7.
[18] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Gener. Optim.*, 2004, pp. 75–86.
[19] C. Lattner, "LLVM and clang: Next generation compiler technology," in *Proc. BSD Conf.*, vol. 5, 2008, pp. 1–33.
[20] G. H. Blindell, *Instruction Selection: Principles, Methods, and Applications*. Cham, Switzerland: Springer, 2016.
[21] J. Pallister, S. Hollis, and J. Bennett, "BEEBS: Open benchmarks for energy measurements on embedded platforms," 2013, *arXiv:1308.5174*.
[22] J. Yu, A. Lukefahr, R. Das, and S. Mahlke, "TF-net: Deploying sub-byte deep neural networks on microcontrollers," *ACM Trans. Embedded Comput. Syst.*, vol. 18, no. 5s, pp. 1–21, Oct. 2019.
[23] A. Garofalo, M. Rusci, F. Conti, D. Rossi, and L. Benini, "PULP-NN: A computing library for quantized neural network inference at the edge on RISC-V based parallel ultra low power clusters," in *Proc. 26th IEEE Int. Conf. Electron., Circuits Syst. (ICECS)*, Nov. 2019, pp. 33–36.
[24] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a Linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI technology," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 11, pp. 2629–2640, Nov. 2019.
[25] A. Amid et al., "Chipyard: Integrated design, simulation, and implementation framework for custom SoCs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, Jul. 2020.
[26] L. T. Clark et al., "ASAP7: A 7-nm finFET predictive process design kit," *Microelectron. J.*, vol. 53, pp. 105–115, Jul. 2016.
[27] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. USENIX Annu. Tech. Conf.*, 2005, pp. 5510–5555.
[28] N. Binkert et al., "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, May 2011.
[29] K. Tervo, S. Malik, T. Leppänen, and P. Jääskeläinen, "TTA-SIMD soft core processors," in *Proc. 30th Int. Conf. Field-Programmable Log. Appl. (FPL)*, Aug. 2020, pp. 79–84.

Kari Hepola received the M.Sc. degree in electrical engineering from Tampere University, Tampere, Finland, in 2022, where he is currently working toward the Ph.D. degree.

His current research interests include methods for increasing the energy efficiency and flexibility of software-programmable accelerators and the application of machine learning in their optimization.

Tharaka Ranasinghe Arachchige received the M.Sc. degree in computer sciences and electrical engineering from Tampere University, Tampere, Finland, in 2024.

His research interests include hardware design and wireless communication.

Joonas Multanen received the M.Sc. degree in electrical engineering from Tampere University of Technology, Tampere, Finland, in 2015, and the Ph.D. degree from Tampere University (TAU), Tampere, in 2021.

He is currently the Staff Scientist of the Faculty of Information Technology and Communication Sciences, TAU. His research interests include energy-efficient computer architectures.

Pekka Jääskeläinen is currently a Professor with the Customized Parallel Computing Research Group, Tampere University, Tampere, Finland. He also works as a Principal Engineer with Intel Finland Oy, Espoo, Finland. He has been contributing to heterogeneous platform customization and programming topics since 2000. In addition to his academic publication activities, he maintains or actively contributes to multiple heterogeneous computing-related open source projects, such as OpenASIP, Portable Computing Language, and chipStar. He is interested in methods and tools to reduce the engineering effort in the design and programming of diverse heterogeneous platforms, and more generally, in hardware and compiler techniques to reduce the energy consumption of programmable processors.