Longnail: High-Level Synthesis of Portable Custom Instruction Set Extensions for RISC-V Processors from Descriptions in the Open-Source CoreDSL Language

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

Brindusa Mihaela Damian-Kosterhon
damian@esa.tu-darmstadt.de
Technical University of Darmstadt
Germany

Florian Meisel
meisel@esa.tu-darmstadt.de
Technical University of Darmstadt
Germany

Tammo Mürmann
muermann@esa.tu-darmstadt.de
Technical University of Darmstadt
Germany

Eyck Jentzsch
eyck@minres.com
MINRES Technologies GmbH
Germany

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

Abstract

In the RISC-V ecosystem, custom instruction set architecture extensions (ISAX) are an energy-efficient and cost-effective way to accelerate modern embedded workloads. However, exploring different combinations of base cores and ISAXes for a specific application requires automation and a level of portability across microarchitectures that is not provided by existing approaches.

To that end, we present an end-to-end flow for ISAX specification, generation, and integration into a number of host cores having a range of different microarchitectures. For ISAX specification, we propose CoreDSL, a novel behavioral architecture description language that is concise, easy to learn, and open source. Hardware generation is handled by Longnail, a domain-specific high-level synthesis tool that compiles CoreDSL specifications into hardware modules compatible with the recently introduced SCAIE-V extension interface, which we rely on for automatic integration into the host cores.

We demonstrate our tooling by generating ISAXes using a mix of features, including complex multi-cycle computations, memory accesses, branch instructions, custom registers, and decoupled execution across four embedded cores and evaluate the quality of results on a 22nm ASIC process.

ACM Reference Format:
Julian Oppermann, Brindusa Mihaela Damian-Kosterhon, Florian Meisel, Tammo Mürmann, Eyck Jentzsch, and Andreas Koch. 2024. Longnail: High-Level Synthesis of Portable Custom Instruction Set Extensions for RISC-V Processors from Descriptions in the Open-Source CoreDSL Language. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3620666.3651375

1 Introduction

Modern embedded systems and Internet-of-Things devices are expected to support increasingly complex applications from domains such as digital signal processing, artificial intelligence, and post-quantum cryptography, which require more computational power than embedded general-purpose processor cores can typically deliver. Hence, *energy-efficient* and *cost-effective* acceleration of the underlying algorithms is required. In between the two extremes of the design space, i.e. using more (or more capable) processor cores (often less energy-efficient) and designing a custom standalone accelerator (often not cost-effective), customizing an existing core into an application-specific instruction set processor (ASIP) is a proven solution to reach these goals.

ASIP design usually builds on an existing base instruction set architecture (ISA), which is then augmented with additional instructions and architectural state elements (e.g. registers) specific to the target domain. The benefit of such ISA eXtensions (ISAX) is that the additional design effort and silicon area are spent only on accelerating the *performance-critical* parts of the computation, while the rest of the core remains unchanged and thus compatible with the existing software ecosystem.

Architecture description languages (ADL) [43] for ASIPs as well as the corresponding toolchains to generate the hard- and software artifacts that make up a development ecosystem have been researched extensively in the past decades, and resulted in multiple commercial offerings available today [44]. However, the topic has recently grown in relevance

```
import "RV32I.core_desc"

InstructionSet X_DOTP extends RV32I {
  instructions {
    DOTP {
      encoding: 7'd0 :: rs2[4:0] :: rs1[4:0] ::
                3'd0 ::  rd[4:0] :: 7'b0001011;
      behavior: {
        signed<32> res = 0;
        for (int i = 0; i < 32; i += 8) {
          signed<16> prod = (signed) X[rs1][i+7:i] *
                            (signed) X[rs2][i+7:i];
          res += prod;
        }
        X[rd] = (unsigned) res;
} } } }
```

Figure 1. 4x8bit dot-product ISAX in CoreDSL, the input language for our high-level synthesis flow. The base ISA description `RV32I` (not shown here) declares the standard RISC-V register field `X` with 32 elements of type `unsigned<32>`.

again through the introduction of the free and open RISC-V ISA with its modular design. The RISC-V specification [54] defines multiple lean base instruction sets and includes first-class support for standardized and application-specific extensions. RISC-V has received widespread adoption in the industry and in academia, resulting in a diverse choice of proposed microarchitectures and extensions.

While the RISC-V ecosystem provides an ideal environment for the ISAX approach, we observe that current solutions for actually implementing an extension are highly microarchitecture-dependent for all but the simplest custom instructions. This is unfortunate for two reasons: Firstly, ISAX design is not *accessible* to application domain experts unfamiliar with computer architecture and digital design. Secondly, the hardware underlying an ISAX is traditionally not *portable* across base core microarchitectures. Thus, the developer is locked into the specific core, or core family, supported by the chosen ASIP toolchain, or faces the high development effort of *manually* integrating an ISAX into one or more existing cores.

Our novel idea to tackle both issues simultaneously is to design and implement *parameterized high-level synthesis* (HLS) algorithms that rely on *bidirectional interaction with a vendor-neutral abstraction* of a core's microarchitecture to generate ISAX hardware from easily usable high-level descriptions, and then automatically integrate them into the target core using an adaptive scalable interface. Our approach supports a wide range of ISA extensions, i.e. not just simple R-type, but also control, memory, and decoupled execution, expressed in a new *user-friendly ISAX description language*.

We present our contributions as follows:

- In Section 2 we introduce *CoreDSL*, an easily accessible behavioral ADL that carefully extends a familiar C-like syntax with arbitrary-precision integer types and bit-level operations. Figure 1 shows a complete SIMD dot-product ISAX as an example.
- In Section 3, we detail how we leverage the recently introduced *SCAIE-V* interface generator [23] as a vendor-neutral and HLS-friendly abstraction for different base processor microarchitectures and configurations. SCAIE-V tailors the processor integration precisely to the needs of the ISAX to be implemented. It also provides the HLS flow with metadata in the form of a "virtual datasheet" for the base processor, in order for the synthesis algorithms to perform core-specific optimizations when generating the ISAX hardware.
- In Section 4, we present *Longnail*, our custom, portable high-level synthesis flow from CoreDSL to register-transfer level SystemVerilog. Longnail processes SCAIE-V's core metadata to create tailored, SCAIE-V-compatible hardware for the ISAX.

We demonstrate a working end-to-end flow and evaluate the practical benefits of our approach in Section 5 by compiling eight ISAXes to hardware, and then automatically integrating them into four open-source RISC-V cores. We review related work in Section 6 and outline future directions in Section 7, before concluding in Section 8.

2 CoreDSL

In this section, we introduce the open-source CoreDSL language, which is the entry point to our proposed ISAX design toolflow.

2.1 Goals and requirements

Early on, discussions with potential users and domain experts yielded the following requirements for an input language that would be accessible to non-experts and still enable powerful and portable ISAX design:

- *User-friendly syntax*: We identified embedded software engineers as the primary target audience for our toolflow. In their daily work, C is the predominant high-level language, but engineers often resort to assembly programming as well to maximize the efficiency of embedded software. ISA manuals usually describe the instruction behavior in imperative pseudocode. Hence, we argue that our input language should be syntactically familiar, or at least obvious, to this group of people.
- *Suitable for ISA description*: The language should be tailored to the task of describing an ISA. This includes offering a concise syntax for the architectural state, instruction metadata, and typical behavior such as bit-level manipulations and arithmetic operations with narrow data types.
- *High-level language*: Portability mandates that the language is fully behavioral, but its semantics should still enable efficient hardware synthesis.

- *Powerful*: Automatically synthesized ISAXes have the potential to move beyond simple custom instructions. For this, our input language should be able to express longer-running behavior that executes *decoupled* from the main pipeline, as well as custom-control flows and memory accesses.
- *Open source*: Lastly, the language should be free and open, in order to serve as the single source-of-truth between other components (e.g. compiler, simulator, etc.) of a vendor-independent RISC-V-based ecosystem.

None of the existing ADLs meet all of these requirements (see Section 6). Hence, our first contribution is *CoreDSL*, which we present here. More information and sample front-ends are available on GitHub[1].

2.2 Syntax and structure

```
desc:     import* (instrSet | coreDef)+;
import:   'import' STRING ';';
instrSet: 'InstructionSet' ID ('extends' ID)? isa;
coreDef:  'Core' ID ('provides' ID (',' ID)*)? isa;
isa:      '{' (state | instrs | always | funcs)* '}';
state:    'architectural_state' '{' stmt* '}';
instrs:   'instructions' '{' instr '}';
instr:    ID '{' 'encoding:' enc 'behavior:' stmt '}';
enc:      (INT | ID '[' INT ':' INT ']')
          ('::' INT | ID '[' INT ':' INT ']')* ';';
always:   'always' '{' ID '{' stmt '}' '}';
funcs:    'functions' '{' /*function definitions*/ '}';
stmt:     /* C-inspired statements and declarations */;
```

Figure 2. Simplified grammar for the top-level structure of a CoreDSL description file. `INT`, `STRING` and `ID` denote integer/string literals and identifiers, respectively.

The top-level structure of a CoreDSL description is shown in Figure 2. *Instruction sets* (**`instrSet`**) and *core definitions* (**`coreDef`**) are the main concepts in our model. Instruction sets are intended to provide reusable and composable (through inheritance) definitions of ISAs, whereas core definitions represent the ISA supported by a concrete core, providing one or more previously defined instruction sets. Together with the *import* concept (**`import`**), these mechanisms provide the necessary modularity to handle the growing number of RISC-V standard extensions. An ISA definition (**`isa`**) is split into multiple, optional sections. The *architectural state section* (**`state`**) contains declarations for *registers* and *address spaces*, which use the usual C syntax with the **`register`** and **`extern`** keyword. Additionally, declarations without storage-class specifiers are interpreted as *parameters*. As part of an elaboration phase, parameter assignments are evaluated in the context of each core definition.

The *instructions* section (**`instrs`**) groups instruction definitions together. An *instruction* (**`instr`**) has a name, an *encoding* specifier (**`enc`**), and a *behavior* comprised of C-like statements (see Section 2.4).

[1]https://github.com/Minres/CoreDSL

The *always* section (**`always`**) holds the definitions of *always*-blocks, which are part of our novel decoupled execution model introduced in Section 2.5. Syntactically, an *always*-block resembles an instruction definition *without* an encoding, as it operates continuously, *independently* of the fetched instruction stream.

Lastly, the *functions* section allows the definition of reusable helper functions. These follow the well-known C syntax, which we do not repeat here for brevity.

2.3 Type system

The CoreDSL type system is built around signed and unsigned integers with arbitrary bitwidths w in two's complement representation, for which we borrow the C++ template syntax: **`signed<w>`** and **`unsigned<w>`**. We provide type aliases for common widths, such as **`int`** a shorthand for **`signed<32>`**. The language *strictly* enforces that precision or sign information is never lost *implicitly* on assignments, which we consider a key precaution against user oversights ("off-by-one" or "fencepost" errors) when dealing with small integer types. Consider the declarations **`unsigned<4>`** `u4`, **`unsigned<5>`** `u5` and **`signed<4>`** `s4`, for example. In our type system, the assignments `u4 = u5` (discarding the most significant bit) and `u4 = s4` (discarding sign information) are forbidden. In addition, all arithmetic operators are bitwidth-aware, meaning that they allow differently-typed operands and produce a result that is wide enough to capture all possible values without over/underflow. For example, the addition of `u5` and `s4` yields a result of type **`signed<7>`**. Narrowing is, of course, supported, but must be explicitly requested with a C-style cast such as `u4 = (`**`unsigned<5>`**`) (u5 + s4)`.

For integer literals, we support the common C syntax (`42`, `0xcafe`), and assign the unsigned type with the minimal width required to represent the value. We borrow Verilog's literal syntax (`6'd42, 3'b111`) to give users more precise control over the desired type, e.g. in the encoding specifier.

While CoreDSL is well-suited to express *instructions* for standard or custom floating-point (FP) formats, based on user feedback, we currently do not see a need to support FP arithmetic *inside* the language. As a further refinement, we plan to support C-inspired structure- and union types in the future.

2.4 Behavior

CoreDSL supports C's arithmetic, shift, bitwise, logical, comparison and conditional operators, adapted to our bitwidth-aware type system. Additionally, we introduce a concatenation operator (`::`), and extend the array-subscript operator to be applicable to scalar values (then returning a single bit) as well as to accept *ranges* (`[from:to]`). `from` and `to` either have to be compile-time constants, or reference the same variable with a constant offset. Applying the array-subscript operator to an address space with a range returns the concatenation of multiple elements. For example, let `MEM` represent the byte-addressable standard address space, then the expression `MEM[addr+3:addr]` encodes a 32-bit little-endian load from memory.

Our tooling currently supports **`if`**-statements, **`for`**-loops and function calls, each with the usual C-style syntax. Note that CoreDSL only behaviorally describes **`for`**-loops. The language leaves it open to the synthesis tool whether these are later unrolled or are executed in an iterating sequential manner. As a convenience to users, we plan to offer full support for other loop constructs and switch statements in the future.

2.5 Decoupled execution

The automatic synthesis of *simple* combinational or short multi-cycle custom instructions that execute within the host core's pipeline is well-supported in the already available ASIP ecosystems. However, more complex ISAXes, e.g. whose instructions have latencies *longer* than the core's pipeline, or that are supposed to execute in parallel to the base pipeline in a decoupled manner, require manual design changes for integration. Such ISAXes can be used to judiciously realize specific higher-performance operations, even when continuing to use simple, MCU-class base cores (e.g., single-issue, in-order) for regular processing.

To motivate the capabilities of CoreDSL, we discuss two examples of such more complex ISAXes. Previous work looked at integrating a CORDIC accelerator into a VexRiscv core [48]. In that work, the instruction needed eight clock cycles to commit its result, which is longer than the original base core's pipeline. It would be advantageous to run this instruction *decoupled* without stalling the base pipeline. However, in this approach, additional attention must be paid to the handling of data hazards. Furthermore, when the execution finishes, the decoupled instruction may write its result only if it does not conflict with the current instructions in the base pipeline. Such a conflict could happen, e.g., during an ongoing load instruction, waiting for data from the memory. Another example of a complex ISAX is adding a mechanism for Zero-Overhead Loops (ZOL). They are supported, for example, by the Ri5cy processor [27], and speed-up code execution significantly for small loops with a single loop counter that is not modified inside the loop body. An additional out-of-pipeline functional unit (FU) holds the addresses of the beginning and the end of the loop, as well as the iteration's number. When the instruction fetch reaches the end of the loop, the ZOL forcibly resets the PC to the beginning of the loop, and updates the iteration's number. When realized as an ISAX, which is easily possible in our system, this FU runs in *parallel* to the core's pipeline. Yet, its requirements differ slightly from the previous example. Compared to CORDIC, ZOL continuously evaluates the current PC *every clock cycle*, regardless of which instruction is currently in the pipeline. Additionally, it does not require a data hazard mechanism, but must be able to update the PC in any clock cycle. These two examples highlight the two main scenarios for which the automatic synthesis and base core integration of FU-like hardware is possible and beneficial:

- Behavior that is executed repeatedly *after* each instruction fetch, as in the ZOL example. Other examples of such continuously running operations could include I/O behavior, which, e.g., could regularly add incoming data to a ring buffer, to be retrieved later by the base core through explicitly issued custom instructions.
- Instructions with longer run-times that may run decoupled and commit their results at a later point in time, as in the CORDIC example. An implementation can choose to support either architecturally visible latencies, requiring, e.g., static compiler scheduling combined with a custom register allocator for correct operation, or dynamic hazard resolution in hardware, using a tailored and lightweight scoreboard-like mechanism.

In both cases, we want to describe behavior that executes *decoupled* from the main pipeline or state machine[2].

We propose two novel constructs, *always* and *spawn*, to describe these scenarios in CoreDSL. The canonical example for an *always*-block is an ISAX providing ZOL functionality to the core, as shown in Figure 3. Wrapping parts of an instruction's behavior in a *spawn*-block, such as the computation of a square root as outlined in Figure 4, is the canonical use-case for the second scenario. Once the execution of such an instruction reaches the *spawn*-block, other instructions are allowed to *overtake* the decoupled instruction in the base pipeline, effectively providing a limited form of out-of-order execution (more accurately: lightweight out-of-order commit/writeback) even for simple in-order base cores.

3 Core microarchitecture abstraction

3.1 Abstraction model

Prior work [23] showed that popular open-source RISC-V cores can differ significantly in their microarchitecture, pipeline (or FSM) configuration, and implementation details. Still, our HLS generated ISAXes must be compatible with different base cores and for this we require an abstraction model. While some cores offer a bespoke extension interface, no portable solution has emerged yet. Here, we leverage the approach recently proposed by Damian et al. [23]: Instead of trying to establish a single static ISA extension interface, they designed an *adaptive* interface *generator*, which internally has all of the knowledge required to make tailored extensions to any supported core in one simple-to-use tool.

On the ISAX side, SCAIE-V offers a well-defined synthesis target for our HLS flow: Commonly-needed actions in

[2]Our tooling also supports the integration of ISAXes into non-pipelined base cores.

Table 1. SCAIE-V sub-interface operations for a 32-bit host core. SCAIE-V creates individual sub-interfaces for each custom register on demand. *AW* denotes the register's address width, $\lceil \log_2(\text{num. elements}) \rceil$, and *DW* its data width.

Sub-interface	Operands	Results	Description
RdInstr	-	i32	Read the full instruction word.
RdRS1, RdRS2	-	i32	Read the value of the GPR indicated by the rs1 or rs2 encoding field.
RdCustReg	i*AW* index, i1 pred	i*DW*	Read the value of a custom register at the given index.
RdPC	-	i32	Read the program counter.
RdMem	i32 address, i1 pred	i32	Load a word from main memory.
WrRD	i32 value, i1 pred	-	Write a value to the GPR indicated by the rd encoding field.
WrCustReg.addr	i*AW* index	-	Submit an index for a write to a custom register.
WrCustReg.data	i*DW* value, i1 pred	-	Write a value to a custom register at the previously submitted index.
WrPC	i32 newPC, i1 pred	-	Write the program counter.
WrMem	i32 address, i32 value, i1 pred	-	Store a word to the core's main memory.
RdIValid_*s*	-	i1	Query whether an instruction is currently executing in stage *s*.
RdStall_*s*, RdFlush_*s*	-	i1	Query whether stage *s* is stalled or being flushed.
WrStall_*s*, WrFlush_*s*	i1 pred	-	Stall stage *s* or flush stages zero to *s*.

```
InstructionSet zol extends RV32I {
  architectural_state {
    register unsigned<32> START_PC, END_PC, COUNT;
  }
  instructions {
    setup_zol {
      encoding: uimmL[11:0] :: uimmS[4:0] :: 3'b101
              :: 5'b00000 :: 7'b0001011;
      behavior: {
        START_PC = (unsigned<32>) (PC + 4);
        END_PC   =
            (unsigned<32>) (PC + (uimmS :: 1'b0));
        COUNT = uimmL;
  } } }
  always {
    zol {
      // program counter (`PC`) defined in RV32I
      if (COUNT != 0 && END_PC == PC) {
        PC = START_PC;
        --COUNT;
} } } }
```

Figure 3. Excerpt from an ISAX that provides zero-overhead loop functionality via custom registers, a dedicated setup instruction, and an *always*-block.

```
InstructionSet sqrt extends RV32I {
  instructions {
    sqrt {
      encoding: 12'd0 :: rs[4:0] :: 3'b000 :: rd[4:0]
              :: 7'b0101011;
      behavior: {
        unsigned<32> arg = X[rs], res = 0;
        spawn {
          for (int i = 0; i < 32; ++i) {/* CORDIC */}
          X[rd] = res;
} } } } }
```

Figure 4. Excerpt from an ISAX that computes the square root of a register operand using an iterative CORDIC algorithm. The operand is retrieved *in-order* with the fetched instruction. Wrapping the long-running part of the computation in a *spawn*-block allows the core to execute independent operations in *parallel* to the CORDIC iterations.

ISAXes, such as reading a register value, or writing the program counter to initiate a jump, are represented by *sub-interface operations*. For each such sub-interface, SCAIE-V provides a *virtual datasheet* to characterize the base core's microarchitecture. This specifies the *latency* and the temporal availability of a sub-interface, abstracted as *earliest* and *latest* time steps relative to a time step 0 representing the instruction fetch stage of the core. Each sub-interface can only be used once per instruction in an ISAX, an exception to this rule being the flush and stall signals, which may be instantiated in each stage. Our Longnail (Section 4) tool interprets this virtual datasheet for its own scheduling decisions in order to generate extensions that are compatible with and optimized for the specific target core. Table 1 summarizes the available sub-interfaces and their signatures.

Using the original SCAIE-V tool as a baseline, we extend its capabilities here to support CoreDSL features in an HLS-friendly manner. One such feature is the added support for SCAIE-V-managed custom registers, holding ISAX-internal state. For these, SCAIE-V provides timing information on the custom read- and write sub-interfaces, and receives size, element type and usage information from a HLS tool. This information is used to automatically instantiate new storage elements that are accessed in a similar manner as the general-purpose register file. Custom register files are accessed with an index that is explicitly computed inside an instruction's behavior, rather than being extracted from the fields in the instruction's encoding.

3.2 Execution modes

We support three execution modes for instructions, and a dedicated mode for implementing *always*-blocks.

- the *in-pipeline* mode: the computation is scheduled so that all interface operations are used during their native availability in the base core's stages.
- the *tightly-coupled* mode: it leverages SCAIE-V's tightly-coupled strategy for multi-cycle instructions that have longer run-times, and can not commit their result within the pipeline's length. SCAIE-V can then insert logic to *stall* the base core until the instruction finishes the computation.
- the *decoupled* mode: it leverages SCAIE-V's decoupled variants of the interfaces, which trigger the generation of the required scoreboarding logic to ensure correct hazard-free execution without compiler support.
- the *always* mode, which allows the continuous execution of operations *independently* of the fetched instruction stream. This is discussed in more detail below.

While the in-pipeline and *always* modes are available for all sub-interfaces, the other mechanisms may be used only for the WrRD, RdMem, or WrMem sub-interfaces.

The *in-pipeline* mode is the preferred choice for short-running instructions, which then execute as if they were part of the base core's pipeline. For this scenario, we further extended the SCAIE-V tool to be more flexible for HLS algorithms. The original tool required results to be provided by the ISAX *exactly* in the time step corresponding to the write-back stage of the core. However, a key advantage of automatic HLS tools stems from the optimization algorithms used within the scheduler. Also allowing results in earlier stages, as needed, gives these algorithms more leeway to find a globally "better" solution, according to the cost function used for optimization. Thus, SCAIE-V has been extended to support this feature. A similar issue arises in case of memory transfers. In our updated version, accesses that happen later than the core's memory stage, but earlier than the write-back stage, are now allowed and implemented in a decoupled manner, making such schedules also feasible.

Longer-running instructions can most easily be supported by the *tightly-coupled* mode. It incurs only a negligible hardware overhead, however the host core's resources are idle during the ISAX execution.

The *decoupled* mode is only employed if explicitly requested by a user through a *spawn*-block (in the CoreDSL ISAX description), as it requires additional hardware resources for the automatically created register data hazard handling that conditionally stalls subsequent issue of dependent instructions. Moreover, this mode incurs a runtime overhead, because the execution in the core pipeline must be stalled for one cycle to avoid possible write-back conflicts with the hardware resources of the base core. Yet, this mode is able to easily add some parallel processing even to simple in-order single-issue base cores.

To further simplify the connection with HLS tools, we also modified SCAIE-V to be able to consistently construct the same set of signals for each sub-interface, regardless of the execution mode it is invoked from.

The *always* mode resembles a dedicated functional unit, which may be set-up (configured) by other instructions, and runs continuously in parallel to the main pipeline. However, their requests to update the core's state are not tied to a fetched instruction, since they can overwrite the core's state at any time. Such write accesses are triggered by a mandatory valid bit, and are not subject to hazard handling to avoid stalling and enable performance-critical scenarios.

Apart from the *always* mode, all other execution modes specified above must ensure hazard handling. Because portability and thus abstraction of microarchitectural details is a key goal of this work, we hide these issues from the HLS tool. For this, we build upon the native SCAIE-V hazard handling mechanism [23]. One enhancement we realized here is allowing earlier result writes and, as a result, earlier pipeline forwarding, leading to less stalling. Moreover, the general-purpose register file's hazard handling concepts are also applied for the new ISAX internal state elements. This way, the HLS tool can remain unaware of the hardware semantics of the ISAX internal state or general-purpose register file.

3.3 Arbitration

Multiple HLS-generated instruction modules may simultaneously request a state update, requiring us to provide arbitration in that case. We extended the initial SCAIE-V tool to multiplex incoming payloads from different instructions, based on the current opcode processed in the pipeline. Thus, an HLS tool may independently generate hardware modules for multiple instructions, without having to worry about multiplexing their interfaces. If multiple ISAXes want to write their result in the same clock cycle, a static arbitration priority is chosen by SCAIE-V that always ensures a deterministic order between the involved ISAXes.

4 Longnail

Now that we have covered the input language and the synthesis target of our approach, we can discuss the actual specialized HLS flow. We realized a lean, domain-specific high-level synthesis tool called *Longnail* to automatically generate hardware modules targeting SCAIE-V and the base core (pipelined or non-pipelined) as described in the previous section. The rationale here is that handling the base core-specific timing constraints for interfacing must be considered in many HLS algorithms, in particular the scheduler, which is often neither exposed nor sufficiently extensible in existing general-purpose HLS tools.

Longnail builds on top of MLIR [38] and CIRCT [14]. MLIR is a framework to build domain-specific compilers, and allows the definition of, and lowering between, multiple intermediate representations that capture different abstraction levels.

The *operation* is the basic entity in MLIR. An operation can carry *regions*. Regions are comprised of *blocks*, which contain other (nested) operations. Together, a set of related operations, *types* and *attributes* form a *dialect*. A core principle in MLIR is to define orthogonal dialects that compose well with other dialects and provide reusable transformation passes. The CIRCT project aims to be an incubator for next-generation hardware compilers. At its core, it provides reusable abstractions for RTL hardware descriptions as MLIR dialects, and a production-quality SystemVerilog export facility. In Longnail, we also leverage CIRCT's static scheduling infrastructure [19], to which we heavily contributed, for defining and solving the scheduling problem required for the ISAX SCAIE-V synthesis target.

4.1 ISAX representations and lowerings

The following sections give a brief overview of the different representations an ISAX description takes on throughout the Longnail flow. The existing RISC-V `ADDI` ("add immediate") RV32I instruction is used as a simple running example, illustrated at four abstraction levels in Figure 5.

(a)+(b) High-level input description The CoreDSL frontend leverages Xtext [2] to parse ISAX descriptions into an abstract syntax tree (AST) and performs contextual analysis to elaborate the type and parameters used in the input. The resulting decorated AST is then traversed to emit the ISAX into a mix of four MLIR dialects at the same abstraction level. The `coredsl` dialect is original to Longnail and contains eponymous operations to model instructions, *always*-blocks, registers and address spaces. Instructions and *always*-blocks carry a region for their behavior, expressed in terms of other MLIR operations. In addition, instructions contain an encoding specification. Registers and address spaces are accessed uniformly with a pair of get and set operations. The dialect also contains additional arithmetic operations such as concatenation and bit-range extraction, which are not available in the corresponding upstream dialects yet. The `coredsl.end` operation is the default terminator for behavior regions. The dialect defines the `coredsl.spawn` operation as an alternative terminator. The operation also carries a region to model the behavior in the *spawn*-block.

The `hwarith` dialect [17] is part of CIRCT and models bitwidth-aware arithmetic on signed and unsigned integer types without over-/underflows. It captures CoreDSL's type system and operators perfectly.

In addition, Longnail supports a subset of the `scf` and `func` dialects [45, 47], i.e. the MLIR standard dialects for structured control flow and function definitions and calls. In particular, we handle branches, loops with known trip counts, and non-recursive functions.

(c) Data-flow graph The next IR inside Longnail serves as input to the actual HLS algorithms, making a set of flat control-data-flow graphs (CDFGs) the obvious choice. To that end, we define the `lil` dialect, short for "Longnail Intermediate Language", which serves two main purposes: First, it provides simple container operations to represent the input instructions as a set of graphs, leveraging MLIR's first-class support for regions with graph semantics. Secondly, `lil` makes the SCAIE-V sub-interfaces, such as RdRS1 or WrRD, explicit operations in the IR, and in consequence, subject to be scheduled alongside the rest of an instruction's behavior.

Each `coredsl.instruction` and `coredsl.always` is lowered to a `lil.graph`. We pattern-match standard register accesses using specific bitranges in the instruction word to the corresponding SCAIE-V interfaces. For example, reading the core's standard register field `@X` at bits 19 to 15 of the instruction word maps to a `lil.read_rs1` operation.

`coredsl.spawn` operations are flattened into the surrounding instruction graph. We mark sub-interface operations used within the *spawn*-block with an attribute to preserve their provenance.

Computations in the `hwarith` dialect are mapped using an upstream conversion pass to CIRCT's `comb` dialect [15], which represents typical combinational operators in hardware description languages. Local registers are expressed in terms of the operations and types in the `memref` dialect [46].

Branches are lowered to conditional data-flow by using multiplexers to select values at control-flow merge points. In addition, SCAIE-V interface operations that change the architectural state optionally carry a predicate.

CoreDSL loops and function calls are supported in the input description if they have compile-time known loop bounds, and are unrolled respectively inlined with pre-HLS upstream utilities.

(d) Register-transfer level CIRCT's RTL dialects `hw`, `comb`, `seq` and `sv` [15, 16, 18, 20] provide abstractions for hardware modules and instances, combinational and sequential logic, and syntactic constructs specific to the SystemVerilog language. The framework also provides an export pipeline to emit correct, idiomatic and well-formatted SystemVerilog from the MLIR representation. Longnail uses these facilities to capture the synthesized microarchitecture, which we present in more detail below in Section 4.5.

4.2 Modeling of the scheduling problem

In order to synthesize a synchronous data path from an untimed behavior specification, high-level synthesis flows must *schedule* the operations in the computation, *allocate* operator instances to execute them, and *bind* operations to operators [24]. Currently, Longnail constructs fully spatial data paths, hence allocation and binding are trivial, and we focus here on handling the interface timing constraints (Section 3) from a core's virtual datasheet in a scheduling problem. In the future, Longnail is intended to also perform resource sharing, as we already took care in its design to enable leveraging prior work [49] for extending its initial spatial-only approach.

```
architectural_state {
  register unsigned<32> X[32];
}
instructions {
  ADDI {
    encoding: imm[11:0] :: rs1[4:0] ::
              3'b000 :: rd[4:0] :: 7'b0010011;
    behavior:
      X[rd] = (unsigned<32>)(X[rs1] + (signed) imm);
  }
}
```

(a) ISAX description (CoreDSL)

```
coredsl.register core_x @X[32] : ui32
coredsl.instruction @ADDI(
    %imm : ui12, %rs1 : ui5,
    "000", %rd : ui5, "0010011") {
  %0 = coredsl.get @X[%rs1 : ui5] : ui32
  %1 = coredsl.cast %imm : ui12 to si12
  %2 = hwarith.add %0, %1 : (ui32, si12) -> si34
  %3 = coredsl.cast %2 : si34 to ui32
  coredsl.set @X[%rd : ui5] = %3 : ui32
  coredsl.end
}
```

(b) High-level instruction description (MLIR, `coredsl` and `hwarith` dialects)

```
lil.graph "ADDI"
    mask "-----------------000-----0010011" {
  %0 = lil.instr_word
  %1 = comb.extract %0 from 20 : (i32) -> i12
  %2 = lil.read_rs1
  %3 = comb.extract %0 from 31 : (i32) -> i1
  %4 = comb.replicate %3 : (i1) -> i20
  %5 = comb.concat %4, %1 : i20, i12
  %6 = comb.add %2, %5 : i32
  lil.write_rd %6
  lil.sink
}
```

(c) Data-flow graph IR (MLIR, `lil` and `comb` dialects)

```
module ADDI(
  input          clk, rst,
  input  [31:0] instr_word_2,
                arg1_2,
  input          stall_in_2,
  output [31:0] res_3_data);
  reg [31:0] pipe_2;
  always_ff @(posedge clk)
    pipe_2 <= stall_in_2 ? pipe_2 : arg1_2 +
      {{20{instr_word_2[31]}}, instr_word_2[31:20]};
  assign res_3_data = pipe_2;
endmodule
```

(d) Register-transfer level (SystemVerilog)

Figure 5. Different representations of an "add-immediate" instruction in the Longnail HLS flow. The CoreDSL description (a) is translated to MLIR dialects of a similar abstraction level (b), and lowered to a CDFG-like representation (c). After performing high-level synthesis, a hardware architecture is constructed and emitted as SystemVerilog (d) through CIRCT.

CIRCT's static scheduling infrastructure [19] is built around an extensible problem model, which we adopt here. In its terminology, *problems* are comprised of *operations*, *operator types*, and *dependences*. Operations and dependences are the vertices and edges in a dependence graph and represent the computation to be scheduled, whereas the operator types encode the characteristics of the target hardware we want to schedule for. In the extensible problem model, these components are always the same, and concrete problem definitions differ in the *properties*, *input constraints* (checking that a problem instance is valid) and *solution constraints* (verifying that a computed solution is feasible).

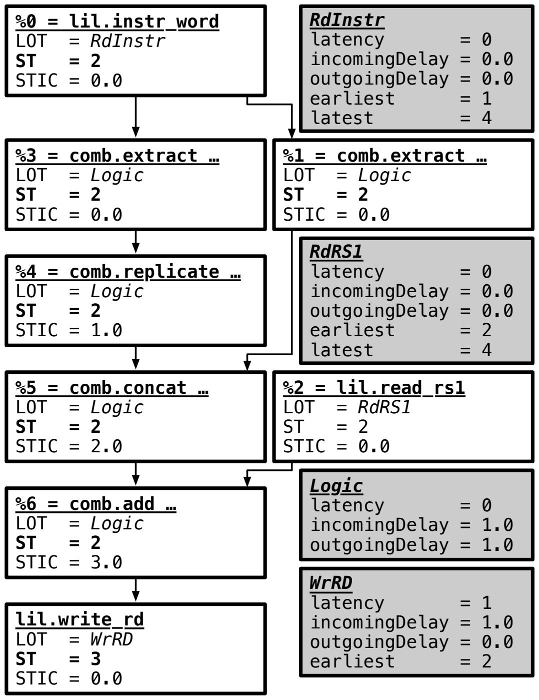
We define the *LongnailProblem* as an extension of CIRCT's *ChainingProblem*, which itself is based on the hierarchy's root *Problem*. Table 2 shows the formal definition in terms of CIRCT's infrastructure. For space reasons, we also introduce abbreviations such as LOT for the "linked operator type" to keep the formulas compact. Intuitively, the *LongnailProblem* is an acyclic scheduling problem without (currently) operator sharing. Operations are associated with an operator type, whose units have a fixed latency and incoming and outgoing propagation delays (incomingDelay, outgoingDelay) to model operator chaining during scheduling. The parts relevant to scheduling for integration into a processor core are the operator type properties earliest and latest, which directly map to the sub-interface constraints provided by SCAIE-V in the virtual data sheet. For the operator types that represent the WrRD, RdMem and WrMem sub-interfaces, we set latest $= \infty$. This allows the sub-interface operations to be scheduled later than their designated pipeline stages and thus unlocks the use of the tightly-coupled or decoupled variants of the sub-interface. Non-interface operator types use default values of earliest $= 0$ and latest $= \infty$ to allow any start time. A solution for the *LongnailProblem* is valid if operations execute after their operands are available and if the computed start time for an operation is between its linked operator type's earliest and latest time.

Figure 6 shows the constructed and solved *LongnailProblem* for the running example, targeting a host core that provides access to the instruction word in stages $1 \ldots 4$ and to the standard register file in stages $2 \ldots 4$. The modeling of the scheduling problem naturally allows operator types with zero latency. In order to prevent that long chains of operations using these operators are all scheduled to the same time step and in consequence are evaluated combinationally, we use CIRCT's chaining support to break overly long chains by distributing their elements over multiple time steps. For simplicity, we currently assume uniform delays and area for logic and non-combinational sub-interface operations, however we plan to leverage an actual target-specific technology library, providing real hardware delays and areas, in the future.

Table 2. Modeling of the Longnail scheduling problem in CIRCT's infrastructure

	Properties		
Problem	Operation	Operator type	Solution constraints
Problem	linkedOperatorType (LOT), startTime (ST)	latency	$\forall(i \rightarrow j) \in$ dependences : i.ST + i.LOT.latency $\leq j$.ST
ChainingProblem	startTimeInCycle (STIC)	incomingDelay, outgoingDelay	$\forall(i \rightarrow j) \in$ dependences : i.LOT.latency = 0 $\land$ i.ST = j.ST $\implies$ i.STIC + i.LOT.outgoingDelay $\leq j$.STIC i.LOT.latency > 0 $\land$ i.ST + i.LOT.latency = j.ST $\implies$ i.LOT.outgoingDelay $\leq j$.STIC
LongnailProblem		earliest, latest	$\forall i \in$ operations : i.LOT.earliest $\leq i$.ST $\leq i$.LOT.latest

Input and solutions constraints that check the presence of properties and perform basic sanity checks are omitted here for brevity.



```
%0 = lil.instr_word
LOT  = RdInstr
ST   = 2
STIC = 0.0

RdInstr
latency       = 0
incomingDelay = 0.0
outgoingDelay = 0.0
earliest      = 1
latest        = 4

%3 = comb.extract …
LOT  = Logic
ST   = 2
STIC = 0.0

%1 = comb.extract …
LOT  = Logic
ST   = 2
STIC = 0.0

%4 = comb.replicate …
LOT  = Logic
ST   = 2
STIC = 1.0

RdRS1
latency       = 0
incomingDelay = 0.0
outgoingDelay = 0.0
earliest      = 2
latest        = 4

%5 = comb.concat …
LOT  = Logic
ST   = 2
STIC = 2.0

%2 = lil.read_rs1
LOT  = RdRS1
ST   = 2
STIC = 0.0

%6 = comb.add …
LOT  = Logic
ST   = 2
STIC = 3.0

Logic
latency       = 0
incomingDelay = 1.0
outgoingDelay = 1.0

lil.write_rd
LOT  = WrRD
ST   = 3
STIC = 0.0

WrRD
latency       = 1
incomingDelay = 1.0
outgoingDelay = 0.0
earliest      = 2
```

Figure 6. Instance of the *LongnailProblem*, comprised of operations (white background), dependences (arrows) and operator types (grey background), corresponding to Figure 5c. The instance is scheduled to meet a maximum cycle time of 3.5 ns, pushing the `lil.write_rd` operation to start time 3.

4.3 Scheduler

The constraints imposed by the *LongnailProblem* map trivially to the integer linear program (ILP) shown in Figure 7. Note that we do not handle the concerns of operator chaining directly. Rather, we rely on utilities in CIRCT to compute special dependencies, referred to as chainBreakers below, to break up long combinational chains.

The ILP uses the following decision variables: For each operation i, t_i models its start time. For each dependence

$$\textbf{minimize} \quad \sum_{i \in \text{operations}} t_i \;+\; \sum_{i \rightarrow j \in \text{dependences}} l_{ij} \tag{obj}$$

subject to

$$\forall i \rightarrow j \in \text{dependences}:$$

$$t_i + i.\text{LOT.latency} \leq t_j \tag{C1}$$

$$l_{ij} \geq t_j - t_i \tag{C2}$$

$$\forall i \in \text{operations}:$$

$$i.\text{LOT.earliest} \leq t_i \leq i.\text{LOT.latest} \tag{C3}$$

$$t_i, l_i \in \mathbb{N}_0 \tag{C4}$$

$$\forall i \rightarrow j \in \text{chainBreakers}:$$

$$t_i + i.\text{LOT.latency} + 1 \leq t_j \tag{C5}$$

Figure 7. ILP formulation to schedule the *LongnailProblem*

edge $i \rightarrow j$, l_{ij} represents the lifetime of the intermediate result. All variables are bound to be non-negative integers by the domain constraints (C4).

Our multi-criteria objective (obj) is to minimize the sum of all *start times* (minimizing the overall latency) and *lifetimes* (saving registers in the ISAX module).

The precedence constraints (C1) ensure that operations start after their operands are available. Constraints (C2) bind the lifetime variables to the desired values. The interface constraints are reflected in (C3). Lastly, constraints (C5) ensure that the endpoints of a chain-breaking edge are separated by at least one time step.

We solve this ILP with the Cbc solver from the COIN-OR project [41], via the OR-Tools library [30]. After computing an optimal solution according to the objective (obj), we transfer the values of the t_i-variables to the startTime property of the problem instance. The startTimeInCycle property is computed afterwards by a utility function in CIRCT.

The concrete sub-interface variant is selected after scheduling, based on the metadata in the "virtual datasheet" which

SCAIE-V provides to Longnail. If the corresponding operation's start time is within the base core's constraint for the used interface, the in-pipeline version is used. Otherwise, if the operation is inside a *spawn*-block, then the decoupled version is used, else we the use tightly-coupled version.

4.4 *always*-blocks

As both `coredsl.instruction`s and `coredsl.always` operations are lowered to `lil.graphs`, the construction of the scheduling problem is similar. The main difference is that all interface constraints are set to stage 0. Solving the scheduling problem hence merely checks that the behavior can be executed in a single clock cycle.

4.5 Hardware generation

After successful scheduling, the actual hardware generation is straightforward. For each `lil`-graph, Longnail constructs an individual hardware module in which the graph's interface operations become input/output ports. This includes ports to access non-constant custom registers, which are instantiated and managed by SCAIE-V. The numerical suffixes indicate in which stage the interface is active, as depicted in Figure 5d. Note that constant registers are internalized into the ISAX module and subject to MLIR's usual canonicalization patterns.

Recall that the remaining ISAX behavior is already lowered to `comb` operations, which are just moved over to the new `hw` dialect's `hw.module` operation. Stallable pipeline registers for intermediate results are inserted into the data path where needed.

A peculiarity of the ISAX-setting is that Longnail does not need to infer a controller circuit for the data path. Rather, the *SCAIE-V-generated logic* keeps track of the progress of custom instructions in the pipeline, and commits their results at the appropriate time.

```
1  - {register: COUNT,  width: 32, elements: 1}
2  # ... other new ISAX-internal state elements
3  - instruction: setup_zol
4    mask: "-----------------101000000001011"
5    schedule:
6      - {interface: RdPC, stage: 1}
7      - {interface: WrCOUNT.addr, stage: 1}
8      - {interface: WrCOUNT.data, stage: 1, has valid: 1}
9    # ...other scheduled operations for the instruction
10 - always: zol
11   schedule:
12     - {interface: RdPC, stage: 0}
13     - {interface: WrPC, stage: 0, has valid: 1}
14     - {interface: RdCOUNT, stage: 0}
15     - {interface: WrCOUNT.addr, stage: 0}
16     - {interface: WrCOUNT.data, stage: 0, has valid: 1}
17   # ...other scheduled operations for the always-block
```

Figure 8. Excerpt from the SCAIE-V configuration file for the ZOL-ISAX from Figure 3, demonstrating the instantiation and the use of a custom counter register in an instruction and an *always*-block.

4.6 Longnail↔SCAIE-V metadata exchange

A key piece of our tool flow, shown in Figure 9, is the exchange of metadata between SCAIE-V and Longnail, for which we are using simple YAML-based file formats to encode the necessary information.

When beginning the HLS, Longnail reads the target core's virtual datasheet from a SCAIE-V-provided file to feed the correct interface timing constraints to the scheduler. The virtual data sheet shows the earliest and latest times (cycles) a specific SCAIE-V interface operation may be scheduled in (cf. Section 4.2), as well as the latency of the operation. In the corresponding box in Figure 9, we show an excerpt from our modeling of the 5-stage VexRiscv core.

After HLS, Longnail outputs the RTL description of the ISAX hardware as SystemVerilog and a configuration file for use by SCAIE-V. This contains the instruction encoding, requested ISAX-internal state elements, and the computed interface schedule. The emitted configuration file for our simple running example is included in Figure 9. In Figure 8, we show an excerpt of the SCAIE-V configuration file for the ZOL ISAX presented in Section 2.5 to explain the features in more detail.

The first line requests SCAIE-V to provide a 32-bit wide register called `COUNT`, which holds the ZOL-internal loop counter. Other state elements, e.g., for the starting and ending address of the loop, have been omitted for brevity.

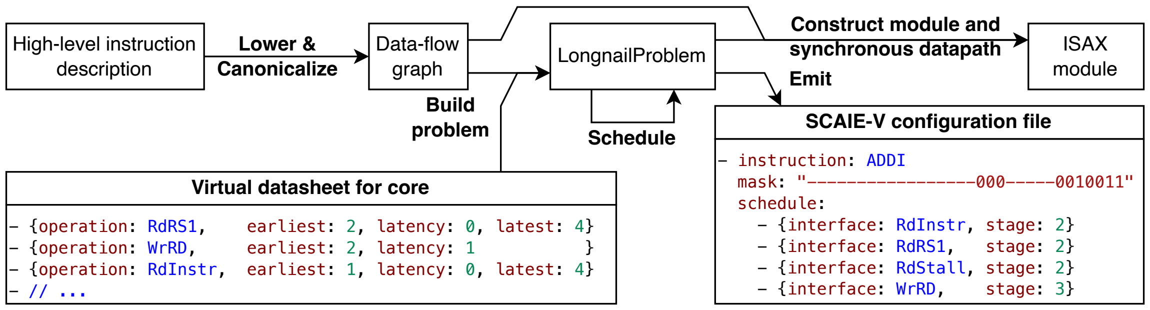
Then, two functionalities are defined. One in the form of a new instruction `setup_zol`, for which the instruction encoding given by the original CoreDSL description is provided (Lines 3. . . 4), the second one in the form of an *always*-block (Line 10) that will be executing continuously in the background.

Each of these functionalities must indicate which specific sub-interfaces are required and in which stages. For reading and updating the new `COUNT` state element, the RdCOUNT and WrCOUNT.addr/.data sub-interfaces are used. For write accesses, the result may be written in later stages.

To enable the generation of hazard handling logic, SCAIE-V needs to determine the earliest access time to a state element. The tooling derives this from the earliest write access to the `addr` port of a state interface, as the address port will always need to be driven first.

Note that, due to `COUNT` having just a *single* element, the WrCOUNT.addr entries shown in Line 7 and 15 is just used to consistently provide the hazard-handling mechanism with stage information, and will not actually result in an address port being created in hardware.

The underlying ZOL functionality continuously monitors and potentially updates the PC at any time, independently of the instructions fetched, and is thus realized as an *always*-block, requesting SCAIE-V to generate interfaces to read/write both the internal loop counter as well as the PC in the base core.



High-level instruction description
Lower & Canonicalize
Data-flow graph
Build problem
LongnailProblem
Schedule
Construct module and synchronous datapath
Emit
ISAX module
Virtual datasheet for core
SCAIE-V configuration file

```
- {operation: RdRS1,    earliest: 2, latency: 0, latest: 4}
- {operation: WrRD,     earliest: 2, latency: 1           }
- {operation: RdInstr,  earliest: 1, latency: 0, latest: 4}
- // ...
```

```
- instruction: ADDI
  mask: "-----------------000-----0010011"
  schedule:
    - {interface: RdInstr, stage: 2}
    - {interface: RdRS1,   stage: 2}
    - {interface: RdStall, stage: 2}
    - {interface: WrRD,    stage: 3}
```

Figure 9. The Longnail flow, showing the YAML-based formats to exchange metadata with SCAIE-V. The depicted datasheet corresponds to the 5-stage version of the VexRiscv core, and was used to schedule the **ADDI** instruction from Figure 5a, resulting in the exported SCAIE-V configuration file.

The `has valid` property indicates that a sub-interface carries an explicit valid signal, which is mandatory for making state updates from an *always*-block.

5 Evaluation

5.1 Benchmark ISAXes

To the best of our knowledge, no established set of benchmark ISAXes to evaluate ASIP design tools has been proposed. To that end, we define the ISAXes in Table 3 to demonstrate the capabilities of our tooling, such as main memory access (**autoinc**, **ijmp**), definition of custom registers (**autoinc**, **zol**), read/write access to the program counter (**ijmp**, **zol**), definition of ROMs (**sbox**, **sparkle**), support for long-running instructions (**sqrt_tightly**, **sqrt_decoupled**) and decoupled execution (**zol**). We also include the **dotprod** instruction from Figure 1. Of course, all ISAXes use basic custom instruction features such as accessing the standard register file and immediate operands. In addition, we also evaluate the combination **autoinc+zol** for the case study in Section 5.5.

5.2 Host cores

With ORCA [3], Piccolo [5], PicoRV32 [4] and VexRiscv [1], we evaluate a selection of open-source embedded cores that represent the diversity in the RISC-V ecosystem well. ORCA and VexRiscv contain 5-stage pipelines, Piccolo uses a 3-stage pipeline, and PicoRV32 is a non-pipelined core that uses an FSM for sequencing. Our proposed abstraction in terms of earliest and latest stages to use the available sub-interfaces (cf. Section 3) allows us to target the cores uniformly from Longnail. The baseline application-specific integrated circuit (ASIC) results in Table 4 also show how the different microarchitectures impact the cores' area requirements and achievable operating frequencies. A more detailed comparison of these cores can be found in [23, 33].

5.3 Evaluation setup

We use the synthesis and place-and-route steps of a commercial ASIC reference flow, targeting a 22nm technology node, to compare the unmodified cores to the ISAX-extended variants (including any SCAIE-V-generated interface logic). The target frequency is set to or above the reachable frequency of each base core to obtain stable results for area. If the worst setup slack of a configuration is negative, we factor it in to the frequency results. We instantiated SRAM macros for the Piccolo core's cache tag and data memories, but substract their area from the reported data for comparability, as the other cores are configured without any caches.

We verified the functional correctness of the extended cores by performing RTL simulation of the execution of handwritten assembler programs.

5.4 Results

Table 4 shows our ASIC results for the various core and ISAX combinations. Note that we expect a moderate increase in chip area because we *add* logic to the core to selectively accelerate specific workloads. Regarding the maximum achievable frequency, we have to consider that some variations are also due to the inherent randomness of the synthesis and ASIC place-and-route heuristics. We obtain frequency variations below 10% for **autoinc**, **ijmp**, **sbox** and **zol**. The latter result is notable in particular, as zero-overhead loops are usually implemented as deeply integrated functional units rather than using an ISA extension mechanism.

We observe frequency regressions when integrating the **dotprod** and **sparkle** ISAXes into the ORCA core. Here, register operands are available in stage 3, and the result write-back is already expected in the following stage. For handling hazards, the core has a forwarding path from the last stage to stage 3. Therefore, any operations scheduled in the last stage will be included in the data forwarding mechanism, leading to a longer critical path. The synthesis tool also tries to reach better timing results by duplicating logic, causing higher area usage. In the future, we will prevent this situation by supplying more precise physical delays to Longnail's scheduling algorithm.

Table 3. ISAXes used in the evaluation to demonstrate the capabilities of our proposed flow

ISAX	Description	Demonstrates
`autoinc`	Auto-incrementing load / store instructions and setup, using a custom register to track the current address	Custom register and main memory access
`dotp`	4x8bit dot product (Figure 1)	Use of loop and bit ranges to concisely describe SIMD behavior
`ijmp`	Read next PC from memory	PC and main memory access
`sbox`	Lookup from AES S-Box	Constant custom register
`sparkle`	Lightweight post-quantum cryptography [13]	R-type instructions, bit manipulations, helper functions
`sqrt_tightly`	CORDIC-based fix-point square root	Loop unrolling, use of tightly-coupled interfaces
`sqrt_decoupled`	CORDIC-based fix-point square root	*spawn*-block, use of decoupled interfaces
`zol`	Zero-overhead loop inspired by PULP extensions [27]. Loop bounds and counter modeled as custom registers.	PC and custom register access in *always*-block.

Table 4. ASIC results for area and frequency overheads of ISAX when integrated into base cores

	ORCA		Piccolo		PicoRV32		VexRiscv (5 stage)	
	Area	Freq.	Area	Freq.	Area	Freq.	Area	Freq.
Base core, area excluding any caches	6,612 μm^2	996 MHz	26,098 μm^2	420 MHz	4,745 μm^2	1,278 MHz	9,052 μm^2	701 MHz
`autoinc`	+ 20 %	- 6 %	+ 3 %	- 9 %	+ 23 %	+ 0 %	+ 12 %	+ 2 %
`dotprod`	+ 23 %	- 14 %	+ 4 %	+ 0 %	+ 21 %	- 2 %	+ 21 %	+ 2 %
`ijmp`	+ 2 %	- 3 %	+ 7 %	+ 3 %	+ 7 %	+ 2 %	+ 12 %	+ 0 %
`sbox`	+ 7 %	- 2 %	+ 0 %	+ 3 %	+ 6 %	+ 2 %	+ 8 %	- 1 %
`sparkle`	+ 85 %	- 24 %	+ 2 %	- 1 %	+ 46 %	+ 0 %	+ 45 %	- 2 %
`sqrt_tightly`	+ 80 %	- 32 %	+ 22 %	- 15 %	+ 100 %	- 5 %	+ 43 %	- 8 %
`sqrt_decoupled`	+ 56 %	- 5 %	+ 10 %	+ 3 %	+ 111 %	- 7 %	+ 47 %	+ 6 %
... without data-hazard handling	+ 46 %	- 6 %	+ 10 %	+ 3 %	+ 96 %	- 2 %	+ 40 %	+ 4 %
`zol`	+ 7 %	- 2 %	+ 13 %	+ 4 %	+ 10 %	- 1 %	+ 14 %	- 3 %
`autinc+zol`	+ 29 %	- 6 %	+ 3 %	+ 2 %	+ 32 %	- 1 %	+ 16 %	+ 5 %

The square root ISAXes share the same behavior specification, i.e. 32 unrolled iterations of a CORDIC-based approach, resulting in the largest extensions in our evaluation. Longnail distributes the computation across 10 pipeline stages, which is longer than any of our host cores can accommodate, necessitating either the tightly-coupled or decoupled execution mode.

The tightly-coupled mode usually saves area, as we can see for VexRiscv and PicoRV32. Yet, in case of ORCA and Piccolo, the downstream ASIC synthesis have to put more effort to achieve timing closure within the ISAX module, using more area in order to satisfy the timing constraints. For completeness, we performed a supporting experiment where we manually added an additional pipeline stage in the ISAX for returning the result. This simplifies timing closure significantly and reduces the ISAX area overhead considerably. As discussed before, this inefficiency will be solved in future work with a technology library that enables the scheduler to make better-informed decisions about where to insert pipeline register to break long combinational paths.

As an additional experiment, we also deactivated the automatic data-hazard handling in SCAIE-V, while using the decoupled mode. This configuration reduces area further, but would require either compiler support to handle data hazards, or very careful manual use of the ISAXes as intrinsics.

5.5 ISAX performance benefits

While the focus of this paper is the synthesis and integration of ISAXes and not the potential performance gains of specific custom instructions, even a simple example, namely adding the elements of an n-element integer array held in memory, already shows typical benefits. When simulated, the baseline VexRiscv version requires $18n + 50$ cycles. The version using our sample **`autoinc`** and **`zol`** ISAXes takes $11n + 50$ cycles. Considering the $f_{\max}$ on the 22nm ASIC, we observe that the core's frequency is practically unaffected by the extension. Hence, the 16% additional chip area enables a >60% speed-up in this simple example.

In a more complex application of performing ML inference on audio signals (see Section 5.6) , we have observed four ISAXes, including **`zol`**, leading to overall gains of 2.15x in wall-clock performance and 30% power savings.

5.6 Impact of proposed technology

CoreDSL was created by some of the authors in context of the *Scale4Edge* project [26], a federal German research project encompassing 23 partners from industry and academia. In the project, the language was not just used for the high-level hardware synthesis we demonstrate here with Longnail, but also for, e.g., verification by automated generation of extended instruction-set simulators [36, 37], the generation of software compilers targeting the ISA-extended base cores [34], and as the source for extracting design properties for later formal verification in a commercial tool.

Furthermore, industry application engineers (non-hardware designers) were able to use the language to successfully formulate ISAXes for the automotive signal processing application described in Section 5.5, which were automatically integrated by SCAIE-V into a VexRiscv [48] base core and successfully taped-out as an SoC in 22nm [10], yielding the performance benefits discussed in the prior section. The performance of the hardware created by Longnail from these CoreDSL descriptions matches or exceeds that of their manually designed implementations which were used on the fabricated chip.

6 Related work

Our proposed flow aims to provide accessible ISA extensions that are portable across a variety of RISC-V microarchitectures. As such, it solves a similar problem to the commercially available ASIP offerings from EDA and processor vendors as well as outside of the RISC-V ecosystem.

Cadence maintains the "Tensilica Instruction Extension" (TIE) language [51, 53] to allow the customization of their "Tensilica Xtensa" line of cores [29]. TIE is a behavioral ADL which uses Verilog-inspired syntax to describe the functionality of custom instructions. Users can access standard registers and memory, use custom instruction formats to define immediate operands, and introduce custom registers of arbitrary size and width. In addition, the Tensilica tools allow defining VLIW execution (i.e. one instruction controls multiple functional units at once) and access to external components via wire, queue and memory interfaces. The behavior of custom instructions must fit into the core pipeline, whereas in our approach, we can flexibly trade-off between in-pipeline and decoupled execution.

Synopsys offers the "ARC Processor Extension" (APEX) technology for the eponymous processor family. From the sparse publicly available information about APEX it can be learned that the behavior specification is based on Verilog, and the tools support multi-cycle instructions as well as the definition of additional conditional checks [11].

Codasip have defined the "CodAL" language [7, 22] to model extensions to their in-house RISC-V processor models. Similar to CoreDSL, CodAL is a behavioral language which uses a C-inspired syntax with support for the usual control constructs and extends it with convenience operators for bit-level operations such as a concatenation. Codasip's tools support multi-cycle instructions [52] and even the definition of zero-overhead loop facilities [21], however it is unknown how the latter is expressed at a language level.

A common theme in these systems is that a) they use a proprietary design entry, and b) provide portability across multiple core, but are always limited to the vendors' own processor families. In contrast, CoreDSL has a publicly available specification and reference implementation, and Longnail can target any core that is supported by SCAIE-V, which is microarchitecture- and language-agnostic.

Synopsys' ASIP Designer is a representative of the class of tools that tackle the design of the *entire* processor. While such an approach is inherently more powerful than an extension mechanism, we argue that it is far less accessible due to the complexity of understanding the inner workings of a given microarchitecture. The proprietary nML language [50] used as input to ASIP Designer describes the hardware at a lower abstraction level and necessarily include structural aspects of the design, such as manually distributing logic across pipeline stages.

The open-source project OpenSoC Architect [35, 40] comprises tools to develop processor cores from scratch with a RISC-like ISA. The structural aspects of the design are described in YAML files, whereas instructions are defined in the "Stonecutter" language [39]. Stonecutter uses a C-like syntax and provides arbitrary-bitwidth integer types, though, in contrast to CoreDSL, does not protect the programmer from losing information due to overflows and implicit casts. The tooling is build on top of LLVM for compiler passes and implements high-level synthesis to a Chisel program.

Lastly, Andes [8] offers an interface generator for their RISC-V cores that supports background instructions, similar to SCAIE-V's decoupled interfaces. Here, the designer is required to provide an implementation of the custom instructions in a HDL, as Andes do not define a high-level design entry for their tooling.

With CORE-V-XIF [6], RoCC [9], PCPI [4] and Tigra [32], the RISC-V community has produced several *open* extension interfaces. These interfaces have been used to implement ISA extensions manually [12, 25, 28], but to the best of our knowledge, no HLS tool targeting a processor core interface has been proposed.

A recent draft specification for *composable extensions* (CX) [31] is motivated by the observation that the current RISC-V extension mechanisms are mostly incompatible with each other, hence hindering the reuse of ISAXes. The specification proposes a generic ISA extension to control composable execution units (CXU), a runtime library for CXU discovery and multiplexing, as well as a logic interface (CXU-LI) to connect CPUs and CXUs. Per the current draft, CXes conform to a fixed interface and are limited to only access integer registers and maintain an isolated state.

In contrast, our *automation-driven* approach enables a deeper and more flexible integration into the host core. This includes support for control-flow ISAXes that modify the program counter, even with advanced features such as zero-overhead loops, as well as shared-state between ISAXes. While the CXU-LI defines four feature levels that are similar to the different execution modes (cf. Section 3.2) supported by SCAIE-V, our approach *automatically* selects the appropriate mode based on user input and the characteristics of the technology library. To the best of our knowledge, no CX-based high-level design flow akin to our combination of CoreDSL and Longnail has been implemented and evaluated. Also, through CoreDSL, we already support reuse and composition of ISAXes at the source-code level. Once CX matures, it will be interesting to see how the two approaches could synergize, e.g., adding CX-like mechanisms for ISAX discovery to SCAIE-V.

7 Outlook

While the focus of this work has been on the automated extension of *microcontroller-class* cores (MCUs), we also aim to apply our approach to more powerful processors.

Current research already has initial prototypes of the SCAIE-V / Longnail flow working on so-called "application-class" cores, such as the OpenHWGroup CVA5 (ex-SFU *Taiga*) [42] and CVA6 (ex-ETHZ *Ariane*) [55]. While still being in-order, single-issue processors, these cores have far more sophisticated pipelines, including branch predictors and parallel execution units for out-of-order execution.

From this prototype flow, we can already observe that the relative cost of SCAIE-V integration *decreases*, as the area of these base cores is generally much larger than that of the MCUs discussed here. On the other hand, the effort to provide the SCAIE-V interface automation *increases*, as it has to deal with more complex machinery (e.g., branch predictors, in-pipeline scoreboard, commit handling, in-order exceptions) in the base core.

After advancing the technology to application-class cores, we are already planning to scale-up to fully Out-of-Order, superscalar microarchitectures. We expect the trends we observed for the application-class cores to continue for these more powerful processors. While these also typically comprise modular execution units, additional considerations will be required for SCAIE-V to ensure correct ordering of *memory requests* and any *custom control flow* issued from within a reordered instruction. For *custom state elements*, an integration with the Out-of-Order core's scheduler will be required for hazard handling. *always*-blocks such as in zero-overhead loops will need to be interfaced with the front-end pipeline stages for PC manipulation, but will be able to leverage the existing mechanisms to detect and handle a misspeculation. All *ISAX-internal state updates* need to be under control of the OoO commit-mechanisms to ensure correct speculative execution and precise exceptions. In order to allow *memory-access ISAX* to profit from any store-to-load forwarding performed in the LSU, SCAIE-V will need to interface with the underlying mechanisms (e.g., the LSQ).

On one hand, these intricacies will complicate the core-specific integration of SCAIE-V. On the other, some of the functionality SCAIE-V currently provides by itself (e.g., hazard handling by private scoreboard, scheduling of writing-back multiple results) can be offloaded to the mechanisms that will be present anyway in an OoO base core.

Above the processor abstraction, though, we expect the ISAX / Longnail-facing SCAIE-V interface to remain stable, allowing portability both of existing ISAX hardware, as well as acting as an HLS target for new functionality.

In addition to extending the processor support, we started to add resource sharing capabilities to our HLS flow to address the large area increases that were observed in some of our evaluation experiments. To this end, we plan to share resources, both within instructions itself and across instruction boundaries, to make extensions with similar functionality (such as packed SIMD) even more economical. Our infrastructure, especially the scheduling phase, is already set-up to leverage advanced integrated optimization techniques [49]. Since area minimization and performance metrics, such as instruction latency, are often conflicting optimization goals, automated design space exploration will be implemented to provide multiple trade-off points. In contrast to the current spatial-only approach, Longnail will then also infer ISAX-local controller circuits to handle resource multiplexing for the shared data paths, instead of just relying on SCAIE-V for sequencing.

8 Conclusion

We presented CoreDSL, a new behavioral architecture description language, and Longnail, a custom high-level synthesis flow. Combined with the previously proposed SCAIE-V interface generator, CoreDSL and Longnail make the design and implementation of instruction set extensions *accessible* to non-hardware experts and *portable* across a variety of RISC-V processor cores without vendor lock-in. Our tooling supports the definition of custom instructions with arbitrarily complex behavior (including memory access and branching capabilities), custom registers with automatic data hazard resolution, and decoupled execution.

9 Acknowledgments

Our work benefited greatly from the CIRCT project and the interaction with its enthusiastic community. We thank our reviewers and shepherd for their valuable feedback and guidance. We acknowledge the financial support by the German Federal Ministry of Education and Research in the projects "Scale4Edge" (grants: 16ME0122K-140, 16ME0465, 16ME0900, 16ME0901) and "MANNHEIM-FlexKI" (grant: 01IS22086A-L).

References

[1] A FPGA friendly 32 bit RISC-V CPU implementation. https://github.com/SpinalHDL/VexRiscv.

[2] Language engineering for everyone. https://eclipse.dev/Xtext/.

[3] Mirror of the now discontinued ORCA RISC-V processor from VectorBlox. https://github.com/cahz/orca.

[4] PicoRV32 - A Size-Optimized RISC-V CPU. https://github.com/YosysHQ/picorv32.

[5] RISC-V CPU, simple 3-stage pipeline, for low-end applications. https://github.com/bluespec/Piccolo.

[6] The OpenHW Group CORE-V-XIF interface. https://docs.openhwgroup.org/projects/openhw-group-core-v-xif/en/latest/.

[7] Hela Belhadj Amor, Carolynn Bernier, and Zdenek Prikryl. A RISC-V ISA extension for ultra-low power iot wireless signal processing. *IEEE Trans. Computers*, 71(4):766–778, 2022.

[8] Andes. Andes custom extension. http://www.andestech.com/en/products-solutions/andes-custom-extension/.

[9] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Palmer Dabbelt, John R. Hauser, Adam M. Izraelevitz, Sagar Karandikar, Benjamin Keller, Donggyu Kim, Jack Koenig, Krste Asanovi ´c, Rimas Avizienis, Palmer Dabbelt, Benjamin Keller, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moretó, Albert J. Ou, David A. Patterson, Brian C. Richards, Colin Schmidt, Stephen Twigg, Huy D. Vo, and Andrew Waterman. The rocket chip generator. 2016.

[10] Paul Palomero Bernardo, Patrick Schmid, Oliver Bringmann, Mohammed Iftekhar, Babak Sadiye, Wolfgang Müller, Andreas Koch, Eyck Jentzsch, Axel Sauer, Ingo Feldner, and Wolfgang Ecker. A scalable risc-v hardware platform for intelligent sensor processing. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2024*, 2024.

[11] Abhishek Bit. 64-Bit Custom Math ISA in Configurable 32-Bit RISC Processor. In Amit Kumar, Marcin Paprzycki, and Vinit Kumar Gunjan, editors, *ICDSMLA 2019*, volume 601, pages 564–575. Springer Singapore, Singapore, 2020. Series Title: Lecture Notes in Electrical Engineering.

[12] Matheus Cavalcante, Domenic Wüthrich, Matteo Perotti, Samuel Riedel, and Luca Benini. Spatz: A compact vector processing unit for high-performance and energy-efficient shared-l1 clusters. In *2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2022.

[13] Hao Cheng, Johann Großschädl, Ben Marshall, Dan Page, and Thinh Hung Pham. RISC-V instruction set extensions for lightweight symmetric cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(1):193–237, 2023.

[14] CIRCT. Circuit IR Compilers and Tools. https://circt.llvm.org.

[15] CIRCT. 'comb' Dialect. https://circt.llvm.org/docs/Dialects/Comb/.

[16] CIRCT. 'hw' Dialect. https://circt.llvm.org/docs/Dialects/HW/.

[17] CIRCT. 'hwarith' Dialect. https://circt.llvm.org/docs/Dialects/HWArith/.

[18] CIRCT. 'seq' Dialect. https://circt.llvm.org/docs/Dialects/Seq/.

[19] CIRCT. Static scheduling infrastructure. https://circt.llvm.org/docs/Scheduling/.

[20] CIRCT. 'sv' Dialect. https://circt.llvm.org/docs/Dialects/SV/.

[21] Codasip. Mythic case study. https://codasip.com/wp-content/uploads/2021/03/Codasip_case-study_Mythic.pdf.

[22] Codasip. What is CodAL? https://codasip.com/2021/02/26/what-is-codal/.

[23] Mihaela Damian, Julian Oppermann, Christoph Spang, and Andreas Koch. SCAIE-V: an open-source scalable interface for ISA extensions for RISC-V processors. In Rob Oshana, editor, *DAC '22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 - 14, 2022*, pages 169–174. ACM, 2022.

[24] Giovanni De Micheli. *Synthesis and optimization of digital circuits*. McGraw-Hill series in electrical and computer engineering. McGraw-Hill, New York, 1994.

[25] Colin Drewes. Rethinking CPU-FPGA Interfaces: A Reconfigurable RISC-V Co-Processor. https://colindrewes.com/projects/GPCP.pdf.

[26] Wolfgang Ecker, Peer Adelt, Wolfgang Müller, Reinhold Heckmann, Milos Krstic, Vladimir Herdt, Rolf Drechsler, Gerhard Angst, Ralf Wimmer, Andreas Mauderer, Rafael Stahl, Karsten Emrich, Daniel Mueller-Gritschneder, Bernd Becker, Philipp Scholl, Eyck Jentzsch, Jan Schlamelcher, Kim Grüttner, Paul Palomero Bernardo, Oliver Bringmann, Mihaela Damian, Julian Oppermann, Andreas Koch, Jörg Bormann, Johannes Partzsch, Christian Mayr, and Wolfgang Kunz. The scale4edge RISC-V ecosystem. In Cristiana Bolchini, Ingrid Verbauwhede, and Ioana Vatajelu, editors, *2022 Design, Automation & Test in Europe Conference & Exhibition, DATE 2022, Antwerp, Belgium, March 14-23, 2022*, pages 808–813. IEEE, 2022.

[27] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gürkaynak, and Luca Benini. Near-threshold RISC-V core with DSP extensions for scalable iot endpoint devices. *IEEE Trans. Very Large Scale Integr. Syst.*, 25(10):2700–2713, 2017.

[28] Tiago Gomes, Pedro Sousa, Miguel Silva, Mongkol Ekpanyapong, and Sandro Pinto. Fac-v: An fpga-based aes coprocessor for risc-v. *Journal of Low Power Electronics and Applications*, 12(4), 2022.

[29] Ricardo E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, 2000.

[30] Google. Or-tools. https://developers.google.com/optimization.

[31] Jan Gray. Draft Proposed RISC-V Composable Custom Extensions Specification, v. 0.92.231111. https://github.com/grayresearch/CX/blob/main/spec/spec.pdf, 2023.

[32] Brad Green, Dillon Todd, Jon C. Calhoun, and Melissa C. Smith. Tigra: A tightly integrated generic risc-v accelerator interface. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 779–782, 2021.

[33] Carsten Heinz, Yannick Lavan, Jaco A. Hofmann, and Andreas Koch. A catalog and in-hardware evaluation of open-source drop-in compatible RISC-V softcore processors. In David Andrews, René Cumplido, Claudia Feregrino, and Marco Platzner, editors, *2019 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2019, Cancun, Mexico, December 9-11, 2019*, pages 1–8. IEEE, 2019.

[34] Bewoayia Kebianyor Jan Schlamelcher, Thomas Goodfellow and Kim Grüttner. Extending clang/llvm with custom instruction using tablegen – an experience report. In *2024 Methods and Description Languages for Modelling and Verification of Circuits and Systems (MBMV)*. VDE/IEEE, 2024.

[35] Ryan Kabrick, John D. Leidel, and David Donofrio. Toward HDL extensions for rapid AI/ML accelerator generation. In *2021 IEEE High Performance Extreme Computing Conference, HPEC 2021, Waltham, MA, USA, September 20-24, 2021*, pages 1–6. IEEE, 2021.

[36] Johannes Kappes, Robert Kunzelmann, Karsten Emrich, Conrad Foik, Daniel Mueller-Gritschneder, and Wolfgang Ecker. Effective processor model generation from instruction set simulator to hardware design. In *2023 IEEE Nordic Circuits and Systems Conference (NorCAS)*. IEEE, 2023.

[37] Stanislaw Kaushanski and Eyck Jentzsch. Automated cross-level verification flow of a highly configurable risc-v core family with custom instructions. In *RISC-V Summit Europe*, 2023.

[38] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques A. Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: scaling compiler infrastructure for domain specific computation. In Jae W. Lee, Mary Lou Soffa, and Ayal Zaks, editors, *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, pages 2–14. IEEE, 2021.

[39] John D. Leidel, David Donofrio, and Frank Conlon. Stonecutter: a very high level instruction set design language. In Maurizio Palesi, Gianluca Palermo, Catherine Graves, and Eishi Arima, editors, *Proceedings of the 17th ACM International Conference on Computing Frontiers, CF 2020, Catania, Sicily, Italy, May 11-13, 2020*, pages 233–236. ACM, 2020.

[40] John D. Leidel, Ryan Kabrick, and David Donofrio. Toward an automated hardware pipelining LLVM pass infrastructure. In *7th IEEE/ACM Workshop on the LLVM Compiler Infrastructure in HPC, LLVM-HPC@SC 2021, St. Louis, MO, USA, November 14, 2021*, pages 39–49. IEEE, 2021.

[41] Robin Lougee-Heimer. The common optimization interface for operations research: Promoting open-source software in the operations research community. *IBM Journal of Research and Development*, 47(1):57–66, 2003.

[42] Eric Matthews and Lesley Shannon. TAIGA: A new RISC-V soft-processor framework enabling high performance CPU architectural features. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, 2017.

[43] Prabhat Mishra and Nikil Dutt. Chapter 1 - Introduction to Architecture Description Languages. In Prabhat Mishra and Nikil Dutt, editors, *Processor Description Languages*, volume 1 of *Systems on Silicon*, pages 1–12. Morgan Kaufmann, Burlington, 2008. ISSN: 18759661.

[44] Prabhat Mishra and Aviral Shrivastava. Chapter 2 - ADL-driven Methodologies for Design Automation of Embedded Processors. In Prabhat Mishra and Nikil Dutt, editors, *Processor Description Languages*, volume 1 of *Systems on Silicon*, pages 13–33. Morgan Kaufmann, Burlington, 2008. ISSN: 18759661.

[45] MLIR. 'func' Dialect. https://mlir.llvm.org/docs/Dialects/Func/.

[46] MLIR. 'memref' Dialect. https://mlir.llvm.org/docs/Dialects/MemRef/.

[47] MLIR. 'scf' Dialect. https://mlir.llvm.org/docs/Dialects/SCFDialect/.

[48] Khai-Duy Nguyen, Tuan-Kiet Dang, Trong-Thuc Hoang, Quynh Nguyen Quang Nhu, and Cong-Kha Pham. A cordic-based trigonometric hardware accelerator with custom instruction in 32-bit RISC-V system-on-chip. In *IEEE Hot Chips 33 Symposium, HCS 2021, Palo Alto, CA, USA, August 22-24, 2021*, pages 1–13. IEEE, 2021.

[49] Julian Oppermann, Lukas Sommer, Lukas Weber, Melanie Reuter-Oppermann, Andreas Koch, and Oliver Sinnen. Skycastle: A resource-aware multi-loop scheduler for high-level synthesis. In *International Conference on Field-Programmable Technology, FPT 2019, Tianjin, China, December 9-13, 2019*, pages 36–44. IEEE, 2019.

[50] Johan Van Praet, Dirk Lanneer, Werner Geurts, and Gert Goossens. Chapter 4 - nML: A Structural Processor Modeling Language for Retargetable Compilation and ASIP Design. In Prabhat Mishra and Nikil Dutt, editors, *Processor Description Languages*, volume 1 of *Systems on Silicon*, pages 65–93. Morgan Kaufmann, Burlington, 2008. ISSN: 18759661.

[51] Himanshu A. Sanghavi and Nupur B. Andrews. Chapter 8 - TIE: An ADL for Designing Application-specific Instruction Set Extensions. In Prabhat Mishra and Nikil Dutt, editors, *Processor Description Languages*, volume 1 of *Systems on Silicon*, pages 183–216. Morgan Kaufmann, Burlington, 2008. ISSN: 18759661.

[52] Alexey Shchekin and Ettore Giliberti. Compact CORDIC accelerator implementation for embedded RISC-V core. In *RISC-V Summit Europe 2023*.

[53] Cadence Design Systems. TIE Language – The Fast Path to High-Performance Embedded SoC Processing. https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/ip/tensilica-ip/tip-tie-wp.pdf, 2016.

[54] Andrew Waterman and Krste Asanovic. The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA. https://riscv.org/technical/specifications, 2019.

[55] 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 Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, Nov 2019.