Clay: High-level ASIP Framework for Flexible Microarchitecture-Aware Instruction Customization

Weijie Peng[§†], Youwei Xiao[§†], Yuyang Zou[‡], Zizhang Luo[†], Yun Liang[*†]
[†]*School of Integrated Circuits, Peking University,* [‡]*School of Software and Microelectronics, Peking University*
[*]*Institute of Electronic Design Automation, Peking University*
{weijiepeng, shallwe}@pku.edu.cn, yyzou25@stu.pku.edu.cn, {semiwaker, ericlyun}@pku.edu.cn

2025 IEEE/ACM International Conference On Computer Aided Design (ICCAD) | 979-8-3315-1560-7/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICCAD66269.2025.11240669

***Abstract*—Application-specific instruction-set processors (ASIPs) provide energy-efficient acceleration for embedded systems and IoT devices. The free and open RISC-V ISA promotes open-source ASIP solutions to accelerate diverse application domains. Existing ASIP tools generate hardware and software artifacts from high-level architecture description languages (ADLs), however, they only support the in-pipeline coupling strategy on specific processors. As a result, they suffer from two critical limitations: they restrict instruction extensions to stateless behavior, preventing hardware implementation of efficient control flow like loops, and they impose rigid microarchitectural constraints that limit register file and memory interactions. These restrictions create a fundamental bottleneck in application acceleration and prevent the efficient deployment of custom instructions across different processors.**

We introduce Clay, an open-source high-level ASIP framework that overcomes these limitations. Clay introduces a unified instruction extension interface that abstracts different coupling strategies as microarchitecture-agnostic actions and microarchitectural attributes. Clay ADL (CADL) combines the interface actions and high-level syntax to describe general instruction behavior, which can be stateful. We further propose a microarchitecture-aware synthesis flow that selects the best coupling strategy for each custom instruction and schedules the optimal implementation with microarchitectural attributes modeled as constraints. Our evaluation of diverse workloads demonstrates that Clay delivers substantial performance improvements across two RISC-V processors, our custom `Clay-core` and the open-source `Rocket-core`.

I. Introduction

Compute-intensive applications in modern embedded systems and Internet-of-Things (IoT) devices necessitate efficient hardware acceleration. Application-Specific Instruction-set Processors (ASIPs) [17]–[19], [24] offer a balanced solution by extending existing Instruction Set Architectures (ISAs) with custom instructions that accelerate critical operations while maintaining software compatibility. With its modular design and explicit support for extensions, the free and open RISC-V ISA has become a popular foundation for such customization. Emerging research has leveraged RISC-V to create custom ASIPs for accelerating various domains, such as digital signal processing [15], artificial intelligence [5], [38], cryptography [9], [21], and bioinformatics [22], presenting needs for a unified open-source RISC-V ASIP framework.

Despite growing ASIP adoption, current approaches suffer from significant limitations. Existing ASIP frameworks like Codasip Studio [10] and Longnail [27] only support the specific coupling strategy, *in-pipeline* coupling, which binds functionality of custom instructions to different stages of the main processor pipeline. This approach is efficient for combinational custom instructions, but suffers from two critical constraints that limit their application scope, as shown in Figure 1. First, they cannot support stateful instruction behavior, preventing the implementation of arbitrary control flow as finite-state machines in hardware. While some Architecture Description Languages (ADLs), such as Codasip's CodAL and Longnail's CoreDSL,

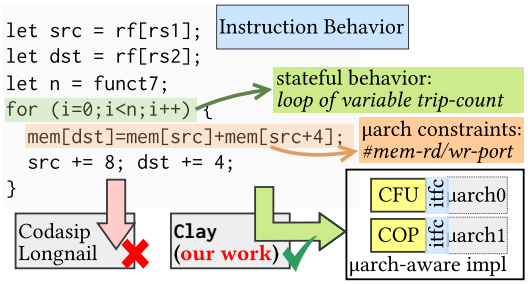


Fig. 1: Clay overcomes the limitations of existing ASIP frameworks.

allow control flow description in instruction behavior, they restrict it to constant trip-counts and conduct loop unrolling before hardware generation to meet the requirements of the in-pipeline coupling. Second, these frameworks impose rigid microarchitectural constraints, such as restricting the number of register and memory operations per custom instruction, allowing for at most two register operands and one memory access due to the limited microarchitectural capabilities provided by the *in-pipeline* coupling strategy. These constraints further restrict the custom instruction behavior. For example, the stream-style instruction behavior in Figure 1 contains a stateful loop with multiple memory accesses. Such stateful and memory-intensive behavior patterns are quite common in diverse domains, including machine learning and digital signal processing, and require manual efforts [14], [28], [29] to be implemented into custom instructions due to the incapability of existing ASIP frameworks [8], [10], [27], [32].

To address these limitations, we present Clay, a novel RISC-V-based high-level ASIP framework that supports multiple coupling strategies, enables stateful custom instructions, and synthesizes implementations that fully exploit the target processor's microarchitectural capabilities. Specifically, Clay introduces a unified instruction extension interface that abstracts both the *in-pipeline* coupling and *coprocessor* coupling into microarchitecture-agnostic actions and underlying microarchitectural attributes. Clay provides a high-level Clay Architecture Description Language (CADL) that combines the microarchitecture-agnostic actions and high-level control flow statements in its syntax, enabling the description of general custom instruction behavior. With CADL, designers can describe custom instructions with arbitrary stateful behaviors and register and memory accesses as exemplified in Figure 1, without microarchitectural constraints imposed by Longnail instructions [27].

For instruction implementation, Clay proposes a microarchitecture-aware synthesis flow that selects the appropriate coupling strategy for each custom instruction and synthesizes efficient hardware implementation concerning target processor features abstracted as microarchitectural attributes in the unified instruction extension interface. Clay formulates scheduling as an integer linear programming (ILP)

[§]These authors contributed equally to this work
[*]Corresponding author

problem, modeling microarchitectural attributes as constraints and generating hardware implementations from the optimal solutions. By doing so, Clay can always pick the appropriate coupling strategy and generate an efficient implementation when a custom instruction is deployed on different RISC-V processors.

The contributions of this paper are:

- An open-source[1] high-level RISC-V ASIP design framework composed of ADL and compiler to implement flexible custom instructions in different coupling strategies.
- A unified instruction extension interface and high-level ADL description, jointly enabling the description of arbitrary stateful instruction behavior without microarchitectural constraints.
- A microarchitecture-aware synthesis methodology that selects the appropriate coupling strategy for each custom instruction and synthesizes efficient hardware with respect to microarchitectural features of the target RISC-V processor.

The experimental results demonstrate that Clay delivers substantial performance improvements—up to 203× on individual kernels and 34× on real-world workloads—across two RISC-V processors with different coupling strategies and microarchitectural configurations: our custom `Clay-core` and the open-source `Rocket-core`. We discuss the effectiveness of Clay's stateful instruction behavior support and microarchitecture-aware synthesis methodology with comprehensive consideration of the performance and hardware overhead.

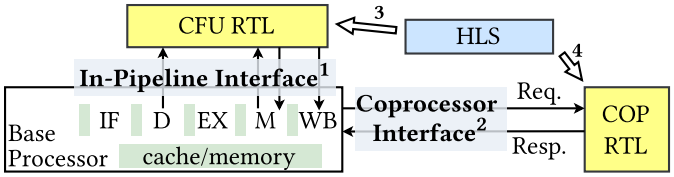
II. Preliminaries

A. Instruction Extension Interface

RISC-V's modular design has revitalized interest in Application-Specific Instruction-set Processors (ASIPs). There are two primary coupling approaches for custom instructions: *in-pipeline* coupling and *coprocessor* coupling, as illustrated in Figure 2. This paper does not discuss off-chip accelerators [20], [23] coupled via Memory Mapped IO (MMIO) or other interfaces due to their high latency and area/power costs. For on-chip custom instructions, in-pipeline coupling reuses the existing control and datapaths of the main processor pipeline, using a very lightweight interface to deliver data between the main processor and the custom function unit across different stages. Coprocessor coupling, on the other hand, adopts the *request-response* interface for offloading and synchronization. Coprocessors have self-contained control and datapaths. These approaches achieve different trade-offs: in-pipeline coupling is lightweight and efficient, especially suitable for combinational custom instructions, while coprocessor coupling is more flexible and scalable to support complex control flows and memory accesses.

Instruction extension interfaces are indispensable for custom instructions of both coupling approaches. Different processor cores typically provide their specific interfaces. For example, Cadence Tensilica [8], Synopsys ASIP Designer [32], Codasip [10], and Andes [4] provide closed-source interfaces that support in-pipeline coupling but only target their processor IPs, while open-source solutions, such as the NICE interface of the Hummingbirdv2 E203 core [25], the Core-V eXtension Interface [26] of the CV32E40X processor, and the RoCC interface of the Rocket core [6], support coprocessor coupling. SCAIE-V [13] provides a portable interface across diverse cores, but it only supports in-pipeline coupling with strict constraints on the instruction behavior, such as allowing only one memory access per instruction, even in its decoupled mode. Clay's instruction extension interface abstracts the microarchitectural details of different cores while supporting both coupling approaches.

[1]https://github.com/pku-liang/clay



[1] [13] [2] [6], [25], [26] [3] [10], [27] [4] [7]

Fig. 2: Instruction extension interface and HLS's role.

B. High-level Synthesis of Custom Instructions

Although instruction extension interfaces provide mechanisms for adding custom instructions, describing the hardware implementation, especially when handling interaction signals with the processor core, remains verbose and error-prone. High-level synthesis (HLS) [12], [37] tools can save such human efforts by automatically generating hardware implementations from behavioral descriptions. For example, Longnail [27] provides a closed-source parameterized HLS flow that generates custom instructions that implement the SCAIE-V interface. In contrast to traditional HLS flows [30] that generate standalone accelerator modules, Longnail additionally models the SCAIE-V operations, such as register file access and memory-stage operations, which have microarchitecture-specific timing constraints, to implement the in-pipeline coupling interface correctly. However, Longnail's applicability is restricted to SCAIE-V's in-pipeline coupling target; for example, it cannot synthesize custom instructions with complex control flows and multiple memory operations. On the other hand, Accelerate-HLS [7] targets a request-response-based coprocessor coupling interface, and can synthesize custom instructions of general control flows and memory operations. However, its heavy interface is inefficient for simple custom instructions, such as pure arithmetic operations. Besides, its synthesis algorithm is not microarchitecture-aware, presuming fixed microarchitectural resources like memory access units, restricting the acceleration potentials. Moreover, it is also closed-source and only supports Bluespec processor IPs.

Clay's HLS flow targets a unified instruction extension interface that supports both coupling approaches. It automatically synthesizes implementations that can either be efficiently in-pipeline coupled or be a coprocessor that implements general control flows and memory accesses, according to the specified instruction behavior and the target processor's microarchitectural capabilities. Clay is the first open-source HLS-based ASIP framework that supports microarchitecture-aware instruction synthesis targeting flexible coupling strategies on RISC-V processors.

III. Methodologies

A. Overview

Figure 3 presents the overview of Clay framework. At the center is the Clay instruction extension interface, which abstracts the interaction between the custom instruction and the base processor as *actions* for different coupling strategies in a unified manner. Each action is associated with a set of microarchitectural *attributes*, which describe the timing and resource constraints of the action provided by the target processor. The Clay ADL (CADL) serves as the frontend of Clay, providing a high-level language to describe custom instructions in a microarchitecture-agnostic manner. For every custom instruction, the CADL describes its encoding and behavior. In addition to computation operations, instructions in CADL also contain *interface operations* that correspond to the *actions* in the Clay instruction extension interface, describing how the custom instruction interacts with the base processor. CADL also provides control flow statements to describe general stateful behavior. Clay's synthesis flow takes CADL

TABLE I: Clay instruction extension interface. Each Clay ADL (CADL) operation is associated with one or a pair of interface actions. Every action has arguments and return values. There are two types of microarchitectural attributes: timing and resource.

CADL Operation	µarch-Agnostic Action	Description	Additional Microarchitectural Attributes
`value=rf[idx]`	`RegRdReq(idx)->(token)`	Send request to read the value of GPR/CSR indexed by `idx`	Latency. Stage range. Maximum number of RF reads per cycle/instruction.
	`RegRdResp(token)->(value)`	Receive the value of GPR/CSR from the read request `token`	Stage range. Maximum number of RF reads per cycle/instruction.
`rf[idx]=value`	`RegWrReq(idx,value)`	Send request to write the value of GPR/CSR indexed by `idx`	Latency. Stage range. Maximum number of RF writes per cycle/instruction.
`value=mem[addr]`	`MemRdReq(addr)->(token)`	Send request to read the value of memory at `addr`	Latency. Stage range. Maximum number of memory reads per cycle/instruction.
	`MemRdResp(token)->(value)`	Receive the value of memory from the read request `token`	Stage range. Maximum number of memory reads per cycle/instruction.
`mem[addr]=value`	`MemWrReq(addr,value)`	Send request to write the value of memory at `addr`	Latency. Stage range. Maximum number of memory writes per cycle/instruction.
`updatePC(pc,cond)`	`PCUpdate(pc,cond)`	Send request to update the PC with `pc` if `cond` holds	Only available for in-pipeline integration (at most one per instruction).

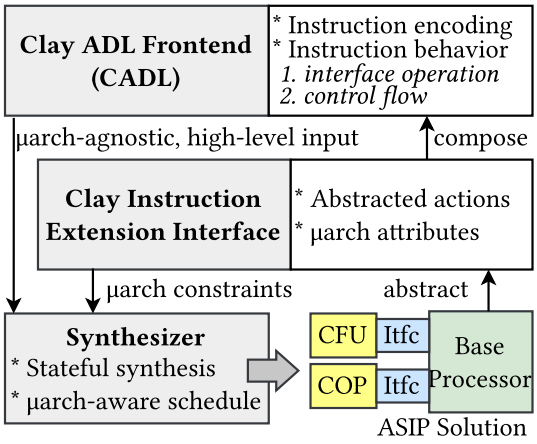


Fig. 3: Overview of the Clay framework

description as inputs and implements each custom instruction in an automatically selected coupling strategy, producing the complete ASIP design. The synthesizer not only generates finite-state machines (FSMs) for stateful instructions, but also conducts microarchitecture-aware scheduling by modeling the microarchitectural *attributes* as constraints, maximizing the instruction's performance towards the target RISC-V processor.

B. Unified Instruction Extension Interface

Clay instruction extension interface abstracts different coupling strategies of any target processor into microarchitecture-agnostic *actions* and microarchitectural *attributes* on the actions, as listed in Table I. The actions provide the primitive operations that custom instructions can perform to interact with the base processor without considering microarchitectural details. The interface allows each custom instruction to access more registers without being restricted to the *rs1*, *rs2*, and *rd* fields in the RISC-V instruction format, eliminating fixed constraints. This flexibility is essential for the CADL frontend to describe arbitrary instruction behavior. The microarchitectural attributes, on the other hand, provide necessary information for the synthesizer to select the appropriate coupling strategy and generate microarchitecture-aware implementation to optimize the performance and hardware overhead of the custom instruction.

To represent processor behaviors that can be combinational, multi-cycle, or latency-insensitive across different coupling strategies and processor implementations, the Clay instruction extension interface adopts a *request-response* mechanism: both the action `RegRdReq` and `MemRdReq` are request actions that initiate the operation and return a `token` as a handle to retrieve the result later; the action `RegRdResp` and `MemRdResp` are the response actions that consume the token and get the result value. For other actions without returning values, such as `RegWrReq`, `MemWrReq`, and `PCUpdate`, we omit the response action for simplicity.

Table I presents two types of microarchitectural attributes for different actions. The *timing* attributes describe the required latency or the legal pipeline stages for the action. In contrast, the *resource* attributes describe the maximum number of a specific type of actions that can be executed concurrently. The response actions do not have the execution latency attribute, since their latency is counted in the corresponding request action. The `PCUpdate` action has a special attribute that restricts it to *in-pipeline* coupling only, with a maximum of one `PCUpdate` allowed per instruction. These attributes are microarchitecture-specific and not exposed to CADL.

C. High-level Architecture Description Language

To describe instructions at a high level to ease the instruction customization task, we propose Clay Architecture Description Language (CADL). CADL describes custom instructions' encoding, states (custom registers or scratchpad memory), and behavior. We focus on CADL's behavior description. In addition to general computation operations, CADL supports *interface operations* to describe the interaction with the base processor and *control flow* statements to describe general stateful behavior.

1) Interface operations without microarchitectural constraints: CADL provides a set of interface operations to describe custom instructions' register file access, memory access, and PC update behavior, all of which are straightforwardly mapped to the corresponding one or pair of actions in the Clay instruction extension interface, as listed in Table I. Since the interface actions are microarchitecture-agnostic, the CADL description is also microarchitecture-agnostic, allowing describing custom instructions without considering microarchitectural constraints and targeting different coupling strategies and processors. Furthermore, CADL omits the *request-response* mechanism for a more brief description. The rationale is that CADL only describes the high-level *untimed* behavior of instructions, and all the operations can be abstracted as instantly executed, without the need for splitting into request and response.

2) Stateful behavior description: CADL supports general control flow statements, including `if`, `for`, and `while`, to describe the stateful behavior of custom instructions. CADL does not require the trip count of the `for`-loop to be known at synthesis time for forced loop

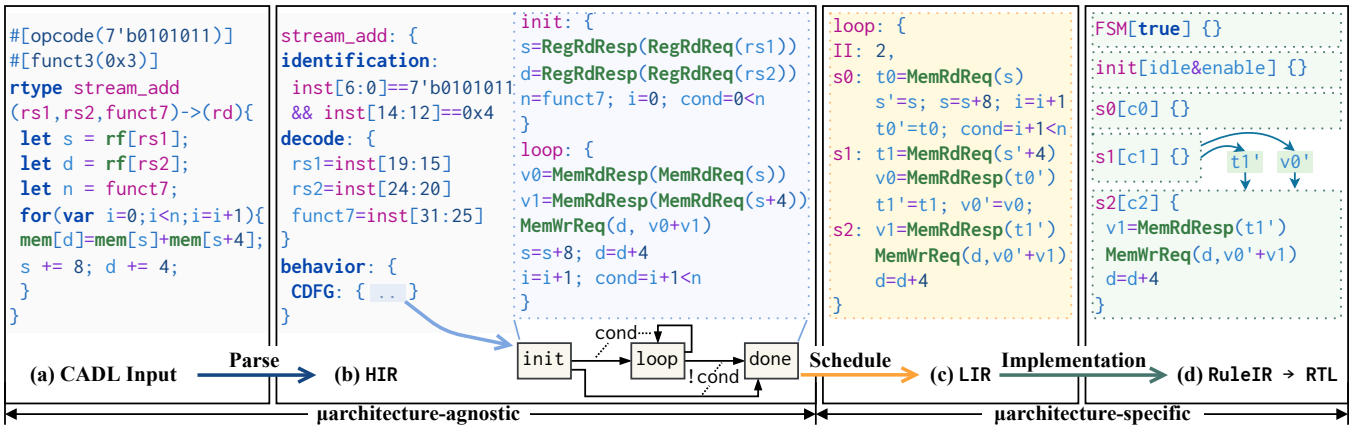


Fig. 4: Code transformation from CADL input to multi-level IRs through parsing and synthesis. Data types are omitted for brievity.

Algorithm 1: Coupling strategy selection and scheduling

Data: custom instruction set C, coupling strategies I
Result: schedule set S

```
1  Function Synthesis (C, I):
2      for custom instruction ci ∈ C do
3          for coupling strategy cs ∈ I do
4              get μarch attributes A from cs;
5              if μarchSanityCheck (ci, A) then
6                  schedule s ← Schedule (ci, A);
7                  if schedule s is feasible then
8                      S.insert(s);
9                      break // explore next ci
10     return S
```

unrolling. Instead, a CADL `for`-loop can take a decoded variable or even a Control and Status Register (CSR) value as the loop bound. This is essential for Clay to describe stateful custom instructions implemented as a coprocessor, which can be configured through the instruction encoding or CSR and runs along with the main processor pipeline.

Combining the microarchitecture-agnostic interface operations and the general control flow statements, CADL is capable of describing arbitrary instruction behavior. Figure 4a presents an example CADL description for the custom instruction *stream_add*. For encoding, the instruction is specified to be an `R-Type` RISC-V instruction, with `opcode` and `funct3` used for identification. It uses the decoded value of fields `rs1`, `rs2`, and `funct7` in the behavior description, where `func7` determines the `for`-loop's bound, and `rs1` and `rs2` specify the register indices of the initial memory address of the source and destination streams, respectively.

The CADL description is parsed and transformed into a high-level intermediate representation (`HIR`) for the Clay compiler to further synthesize. Figure 4b presents the `HIR` for the *stream_add* instruction, where the identification condition and the decoding logic are explicitly described, and the behavior is specified as a control and data flow graph (CDFG). In the CDFG, CADL's interface operations are transformed into actions in the Clay instruction extension interface. Program transformations such as if-conversion [2] and user-directed loop unrolling are conducted on the `HIR` to prepare for the synthesis.

D. Microarchitecture-aware Synthesis

The Clay compiler further conducts synthesis to select the best coupling strategy and generate microarchitecture-aware low-level IR (`LIR`). The synthesis algorithm is presented in algorithm 1. In addition to a set of custom instructions to synthesize, denoted as C, the synthesizer also takes the candidate coupling strategies I as input. The possible coupling strategies are *in-pipeline*, *coprocessor*, or both, depending on the target processor's supporting situation. For every specific coupling strategy cs, it specifies the required microarchitectural attributes, denoted as A, on interface actions, as shown in Table I. These attributes will be essential in the scheduling algorithm to produce microarchitecture-aware instruction implementation.

In algorithm 1, the synthesizer iterates over all the custom instructions (line 2) and tries the coupling strategies one by one in a priority order for every instruction (line 3). Typically, the synthesizer will first try the *in-pipeline* strategy if the processor supports it, as it is more area-efficient, and will fall back to the *coprocessor* strategy if the *in-pipeline* is infeasible. For every coupling strategy, the synthesizer extracts the microarchitectural attributes (line 4) and conducts a fast sanity check (`μarchSanityCheck`, line 5) to ensure the instruction is possible to be scheduled on the chosen coupling strategy. There are two rules for the sanity check: for *in-pipeline* coupling, the CDFG of the instruction cannot have multiple basic blocks, since the instruction implementation should reuse the pipeline stages' control logic, which does not support the complex control flow; for *coprocessor* coupling, the action `PCUpdate` should not be used in the instruction, since the coprocessor is far away from the PC update logic. If the sanity check passes, the synthesizer will try scheduling the custom instruction on the coupling strategy (`Schedule`, line 6).

1) Synthesizing stateful behavior: For custom instructions whose CDFG contains multiple basic blocks, indicating the instruction has stateful behavior, the synthesizer will schedule the basic blocks individually and generate a finite-state machine (FSM) to implement the control flow represented by transitions among the basic blocks. Here we introduce how the synthesizer schedules each basic block. For *coprocessor* coupling, we consider the *modulo scheduling* for innermost loop body basic blocks with an incrementally increasing initial interval (II), and normal scheduling without II consideration for other basic blocks. For *in-pipeline* coupling, we only consider the normal scheduling without II consideration. We formulate both scheduling problems as a unified integer linear programming (ILP) problem with the variables and constants defined in Table II, as well as objectives and constraints listed in Table III, generally according

TABLE II: Variables and constants of ILP formulation

Variable	**Description**	
$x_{s,i}$	whether operation i is scheduled at stage s?	
s_i	the scheduled stage of operation i	
l_i	the lifetime of operation i's result	
Constants	**In-pipeline-specific**	**Coprocessor-specific**
$d_{i,j}$	whether operation i and j have data dependency	
p	/	whether modulo scheduling?
$dist_{i,j}$	/	inter-iteration dependency distance
S μ	processor pipeline stages	scheduling stages
L_i μ	latency of operation i	
R_i μ	legal stage range	/
c_r μ	microarchitectural resource constraints	
$u_{i,r}$ μ	whether operation i uses resource r?	

μ **indicates microarchitecture-specific constants**.

TABLE III: Objectives and constraints of ILP formulation

Name	**Expression**
Objective	minimize $\quad \alpha M + \beta \sum_{i \in V} w_i l_i$
Dependency Constraints	$s_j + p \times dist_{i,j} \times II \geq s_i + L_i$
Resource Constraints	$\sum_{kt \in S} \sum_{i \in V} u_{i,r} x_{kt,i} \leq c_r \quad (\text{if } p)$ $\sum_{i \in V} u_{i,r} x_{s,i} \leq c_r \quad (\text{if not } p)$
Lifetime Constraints	$s_j + p \times dist_{i,j} \times II - s_i - L_i \leq l_i$
Interval Constraints	$s_i \in R_i$
Schedule Once	$\sum_{s \in S} x_{s,i} = 1$
Overall Latency	$M \geq s_i + L_i$

to their classic formulations. The rationale of selecting the ILP formulation is two-fold: first, the problem scale is manageable since the overall chip area limits the number of operations in a custom instruction, and modern ILP solvers such as Gurobi [16] can efficiently find the optimal solution; and second, microarchitecture features are modeled as resource constraints and ILP-based scheduling can discover the optimal resource-constrained solutions, more suitable for our design goal than other scheduling approaches [11], [39].

2) Modeling microarchitecture in scheduling: Clay's scheduling algorithm tries to find a legal schedule of the best performance and area efficiency when targeting a specific coupling strategy and microarchitecture. The key is to model the microarchitectural information in the scheduling formulation. We therefore convert both the *timing* and *resource* microarchitectural attributes from the Clay instruction extension interface (Table I) into microarchitecture-specific constants shown in Table II. The latency attributes of actions are converted into the latency constants L_i of the operation i. The *in-pipeline*-specific stage range attributes, specifying the legal processor pipeline stages for an action to be scheduled, are converted into the stage range constants R_i. And the resource attributes are converted into the resource limit constants c_r, and the usage constant $u_{i,r}$ is set high to indicate the corresponding operation i's usage of resource r. A special case in microarchitecture modeling is the handling of *response* actions (such as `MemRdResp` in Figure 4b), which lack latency attributes as shown in Table I. We model them as L_{resp}=0, and their scheduled stage will be determined considering the latency of the corresponding request action as well as the data dependency between the request and response actions.

The microarchitecture-specific constants have an essential impact

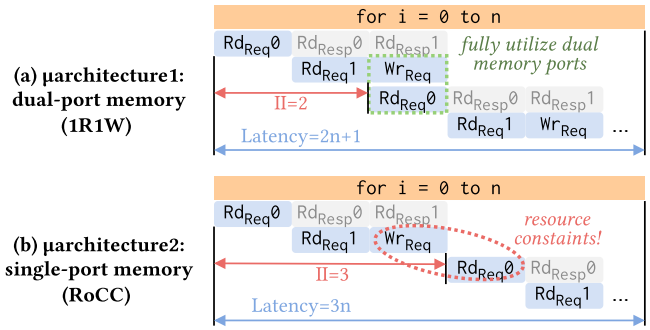


Fig. 5: Microarchitecture-aware scheduling of *stream_add*

on the ILP problem by participating in the constraints, as shown in Table III. Specifically, the latency constant L_i is involved in the dependency constraints, indicating that the operation j cannot start before the operation i finishes. The stage range constant R_i is used in the interval constraints, which forces the scheduled stage of operation i to be within the range. The resource constraints are also affected by the microarchitecture-specific constants, as the resource limit constant c_r specifies the maximum number of microarchitectural resources that can be used per cycle.

Example of microarchitecture-aware scheduling: Figure 5 presents the scheduling solutions for the *stream_add* custom instruction described in Figure 4b, under two different microarchitecture configurations. Since the instruction behavior is stateful, the synthesizer only tries the *coprocessor* coupling strategy and conducts modulo scheduling for the loop body basic block. Figure 5 only presents the scheduling situation of memory access operations. For the dual-port memory configuration in Figure 5a, the ILP formulation includes two resource constraint constants c_{rd} and c_{wr}, both of which are 1, indicating there are two independent ports, one for read and the other for write, available at the same cycle. The ILP solution schedules the memory write operation in the current iteration and the first memory read operation in the next iteration in the same cycle, leading to the optimal II of 2 and the minimal latency of $2n + 1$ for the stream length n. For the single-port configuration in Figure 5b (which reflects the RoCC coprocessor interface detailed in Section IV-A), the ILP formulation includes just one resource constraint constant c_{rdwr} set to 1, with both memory read and write operations using this resource as indicated by the u constants. The ILP solver only finds the feasible solution with II of 3 due to the resource constraint, and the latency becomes $3n$. The difference in II and latency demonstrates the microarchitecture-awareness of our scheduling algorithm, which tries to find the optimal legal schedule for different microarchitectures. The schedule results are represented as `LIR`. Figure 4c presents the `LIR` for the *stream_add* custom instruction targeting the microarchitecture configuration in Figure 5a.

E. Hardware Implementation

The Clay compiler generates the hardware implementation from the synthesis results in the `LIR` format. The implementation process comprises two steps: *behavior implementation*, constructing hardware to implement the custom instruction behavior, and *interface implementation*, building the interface logic to connect the custom instruction hardware to the base processor.

1) Behavior implementation: In `LIR`, Clay's synthesis has scheduled each basic block into a set of stages according to the selected coupling strategy and the target microarchitecture. To implement the scheduled behavior, we first translate each scheduled basic block into a set of rules, where each rule corresponds to a scheduling stage and

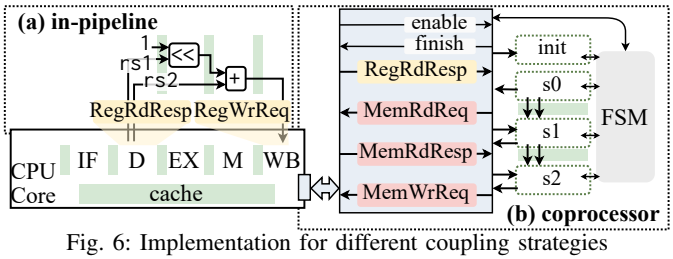


Fig. 6: Implementation for different coupling strategies

contains the operations scheduled to that state, composing the `RuleIR` in Figure 4d. Data dependencies exist between operations in different rules, such as variables `t1'` and `v0'`, which are produced in rule `s1` and consumed in rule `s2`. The Clay compiler instantiates necessary pipeline registers to deliver data between rules.

Implementing stateful behavior: Since the *coprocessor* coupling strategy allows custom instructions to have stateful behavior, the Clay compiler must generate finite-state machines (FSMs) to implement the control flow represented by transitions among the basic blocks. As shown in Figure 4d, each rule is associated with a *guard* condition, specifying the FSM state at which the rule is executed. The state transition logic is implemented in the generated rule `FSM`, whose guard is always `true` and calculates the next state based on the current state and the transition condition specified in the instruction's CDFG.

2) Interface implementation: Rules in the `RuleIR` contain operations that correspond to the *interface actions*, such as `MemRdResp` and `MemWrReq` in the `s2` rule in Figure 4d. These interface actions exchange data between the custom instruction and the base processor, whose implementation differs for different coupling strategies to provide the necessary hardware functionality.

a) In-pipeline interface: Scheduling stages represented as rules in the `RuleIR` format are bound to the processor pipeline stages. Since the microarchitectural attributes on the interface actions are considered in the scheduling problem formulation, the rules must satisfy the *interval constraints*, guaranteeing that each interface action must be bound to the correct processor pipeline stage. Therefore, implementing an in-pipeline interface is straightforward: connecting wires that the main processor exposes for interface actions with the wires that the custom instruction requires for using the interface actions. For example, Figure 6a illustrates the implementation of a custom instruction calculating `(rs1<<1)+rs2` that is in-pipeline coupled. The processor pipeline specifies that the action `RegRdResp` is provided at the decode (`D`) stage and the action `RegWrReq` is only provided at the write-back (`WB`) stage. The custom instruction is scheduled across three clock cycles, retrieving the operand value from the decode stage and returning the result value to the write-back stage, with pipeline registers instantiated to provide the correct timing, as illustrated in Figure 6a.

b) Coprocessor interface: When a custom instruction is implemented as a coprocessor, it implements the *request-response* mechanism to conduct the interface actions to exchange data with the main processor. Figure 6b presents the *stream_add* custom instruction implemented as a coprocessor. When its rule `s2` is executed according to the FSM, the custom instruction will retrieve a `MemRdResp` response from the main processor and send the computation result as a `MemWrReq` request to the main processor, according to Figure 4d. The custom instruction hardware and the main processor must implement the same coprocessor interface to achieve coordinated functionality. For example, when Clay synthesizes the *stream_add* custom instruction to the `Rocket-core`, both the custom instruction hardware and the `Rocket-core` adopt the RoCC interface to achieve the *request-response* mechanism. The major challenge for custom instruction hardware in implementing a coprocessor interface is that it must provide the capability to tolerate unpredictable response latencies. For example, in Figure 4d, the rule `s2` must wait for the `MemRdResp` response to execute the remaining operations, whose arrival timing depends on the cache behavior when targeting the `Rocket-core`. We implement a latency-insensitive stalling mechanism in the FSM of the custom instruction hardware to wait for required responses.

3) RTL generation: The Clay compiler translates rules in `RuleIR`, as shown in Figure 4d, to the description in the rule-based hardware description language `cmt2` [35], [36]. The stalling mechanism is also implemented in `cmt2` to provide latency insensitivity for rules' execution. Finally, SystemVerilog descriptions of the custom instructions are generated through `cmt2`'s elaboration and compilation flow.

IV. Evaluation

A. Evaluation Setup

1) Base processors: We evaluate Clay on two RISC-V processors, in-house `Clay-core` and open-source `Rocket-core`, which support different coupling strategies and have diverse microarchitecture configurations. `Clay-core` is a 5-stage fully-bypassed pipelined processor based on RV32I ISA. `Clay-core` implements a cacheless design with physically separated memory load/store channels, completing every memory operation in one clock cycle. `Rocket-core`, generated by the Chipyard [3] framework, implements the RV32IMAC ISA with a 5-stage fully-bypassed pipeline architecture. It features the built-in multiplier function unit (FU) and cache hierarchy, where the data cache employs a shared memory channel for both read/write requests, which is latency-insensitive. `Clay-core` supports both *in-pipeline* and *coprocessor* coupling, while `Rocket-core` only supports *coprocessor* coupling through the RoCC interface.

2) Custom instructions for evaluation: We implement a set of custom instructions across diverse application domains for evaluation. Table IV presents that the custom instructions require different CADL features for description. Among them, *simd* and *complex_mul* are combinational computations with bit-select and concatenation operations, *sbox* queries a constant lookup-table, *autoinc* contains a custom register and conducts memory access, *crc* and *cordic* require multiple loop iterations where loop unrolling is optional by user directives, and *gemm2x2* and *stream_add* include more memory access operations than that conventional processor pipelines' memory stage provides, requiring *coprocessor* coupling and microarchitecture-aware synthesis. All custom instructions follow the standard RISC-V R-type instruction encoding format. Table IV also presents the lines of code (LOC) required to describe each custom instruction in CADL. With the exception of *sbox*, which requires 256 lines for its constant lookup table, all other custom instructions can be described within 30 lines, demonstrating the productivity of high-level CADL. Moreover, Clay produces RTL design for every evaluated custom instruction within one second, demonstrating the efficiency of the ILP-based synthesis flow.

3) Methodology: In addition to the CADL description, we also program the custom instructions' behavior as C functions, which are compiled using GCC with the -O2 compilation flag enabled. We conduct cycle-accurate simulation with Verilator 5.034 [33] for both base processors to quantify the execution cycles with and without the custom instruction acceleration. All cycle data is collected after cache warm-up. We also collect the timing and area data with Clay-synthesized custom instruction integrated. We employ the OpenROAD open-source ASIC flow with Yosys 0.46 [34] and OpenROAD v2.0-15774 [1] for logic synthesis and place-and-route, targeting the

TABLE IV: Custom instructions for evaluation.

Custom instruction	Behavior description	CADL features required	CADL LOC
simd	SIMD instructions for vector add	Bit-select and concatenation, register access	10
complex_mul	Complex multiply using four parallel multiplier	Arithmetic operations, bit-select and concatenation, register access	11
sbox	S-box for AES encryption/decryption	Constant lookup-table, register access	262
autoinc	Auto-increment addressing for load/store instructions	Custom register, memory access, register access	7
crc	CRC hash algorithm	Control flow (`for`), bitwise operations, register access	10
cordic	Fixed-point sin/cos/arctan implemented with CORDIC algorithm	Control flow (`if`, `for`), signed arithmetic, shift operations, register access	26
gemm2x2	2x2 matrix multiplication from and to memory	Arithmetic operations, memory access, register access	21
stream_add	Streaming pairwise reduction `a[i]=b[i*2]+b[i*2+1]`	Control flow (`for`), memory access, register access	11

TABLE V: Performance and hardware overheads of custom instructions across two RISC-V processors: `Clay-core` (ClayC) and `Rocket-core` (Rocket). For coupling strategies, *ip* denotes *in-pipeline* coupling and *cop* denotes *coprocessor* coupling.

Custom instruction		#Cycle (original)		#Cycle (optimized)		%Cycle (reduction)		Speedup		Period		Area		Coupling	
		ClayC	Rocket	ClayC	Rocket	ClayC	Rocket	ClayC	Rocket	ClayC	Rocket	ClayC	Rocket	ClayC	Rocket
Base		/	/	/	/	/	/	/	/	1.16ns	1.68ns	20137μm^2	50941μm^2	/	/
simd		20	18	1	6	-95.0%	-66.6%	20.0×	2.8×	-0.2%	+5.7%	+2.5%	+10.6%	*ip*	*cop*
complex_mul		283	38	1	4	-99.6%	-89.0%	203.2×	9.3×	+39.3%	+2.5%	+19.2%	+23.7%	*ip*	*cop*
sbox		4	8	1	6	-75.0%	-25.0%	4.0×	1.3×	-0.3%	+2.5%	+5.7%	+7.7%	*ip*	*cop*
autoinc		72	301	14	116	-80.5%	-61.0%	5.1×	2.5×	+0.4%	+4.4%	+0.3%	+14.5%	*ip*	*cop*
crc	comb	73	61	1	6	-98.6%	-90.0%	72.9×	9.4×	+0.2%	+8.7%	+2.5%	+8.5%	*ip*	*cop*
	iterative	73	61	8	24	-89.0%	-60.0%	9.1×	2.5×	+0.6%	+3.2%	+3.7%	+6.3%	*cop*	*cop*
cordic	comb	125	85	1	4	-99.2%	-95.0%	30.9×	7.5×	+305.0%	+182.6%	+39.8%	+33.6%	*ip*	*cop*
	multicycle	125	85	9	16	-92.8%	-81.0%	13.6×	5.3×	+1.8%	+0.7%	+62.6%	+38.3%	*cop*	*cop*
	iterative	125	85	9	16	-92.8%	-81.0%	13.7×	5.2×	+1.7%	+1.8%	+16.8%	+23.1%	*cop*	*cop*
gemm2x2		305	61	12	31	-96.0%	-49.0%	14.6×	1.5×	+73.5%	+32.8%	+44.5%	+45.8%	*cop*	*cop*
stream_add		71	101	18	46	-74.6%	-54.0%	3.9×	2.1×	+0.5%	+3.0%	+12.1%	+21.1%	*cop*	*cop*

Nangate45 process. The target frequency is configured above the attainable frequency, with the final frequency being calculated by subtracting the worst negative slack. For timing and area evaluation, we removed caches from `Rocket-core` to ensure fair comparison.

B. Impacts of custom instructions

Table V presents the impacts of custom instructions on the performance and hardware overheads. The *simd* custom instruction reduces the execution cycles by 95% and 66%, achieves 20.0× and 2.8× overall speedup, increases the clock period by -0.2% and 5.7%, and increases the area by 2.5% and 10.6%, on `Clay-core` and `Rocket-core`, respectively. For *complex_mul*, the custom instruction on `Clay-core` achieves a much higher speedup of 203.2× by reducing 99.6% execution cycles, but it also leads to 39.3% clock cycle period increase, while on `Rocket-core`, the new instruction achieves a moderate 9.3× speedup and causes a more minor 2.5% period increase. This occurs because `Rocket-core` already features a built-in multiplier FU with an original execution cycle of 38 (significantly lower than `Clay-core`'s 283), while the newly added multipliers substantially degrade `Clay-core`'s frequency. For *sbox* and *autoinc* custom instructions, Clay saves 75.0%-80.5% and 25.0%-61.0% execution cycles, achieves 4.0×-5.1× and 1.3×-2.5× overall speedup, increases period by -0.3%-0.4% and 2.5%-4.4%, and uses 0.3%-5.7% and 7.7%-14.5% more area, for `Clay-core` and `Rocket-core`, respectively. For all the remaining custom instructions, Clay also achieves significant speedup, ranging from 3.9× to 72.9× for `Clay-core` and from 1.5× to 9.4× for `Rocket-core`. These results demonstrate that Clay introduces substantial performance gains across RISC-V cores.

1) Benefits of stateful custom instructions: As shown in Table V, implementing large custom instructions, such as *cordic*, in the combinational manner with the computation loop fully unrolled, will seriously worsen the processors' timing. Although the combinational implementation of the *cordic* instruction reduces the execution cycles by 99.2% and 95.0% and achieves 30.9× and 7.5× speedup on `Clay-core` and `Rocket-core`, it increases the clock cycle period by 305.0% and 182.6% on the two base processors, respectively. Even though the high-level synthesis technique can schedule the unrolled computation into multiple stages to ease the timing issue, the multi-cycle implementation cannot fully reuse the computation resources, causing 62.6% and 38.3% area overheads on the two base processors. These challenges remain unavoidable in prior ASIP frameworks [8], [10], [27]. Clay, on the other hand, supports stateful custom instructions to solve the dilemma. The iterative implementation of the *cordic* custom instruction as *coprocessor* without unrolling the loop also avoids the timing issue, while introducing only 16.8% and 23.1% area overheads on the two base processors, both of which are lower than those of the multi-cycle implementation. However, iterative implementation does not always yield superior solutions. For the *crc* custom instruction, the iterative implementation achieves much lower speedup (9.1× and 2.5×) than the loop-unrolled combinational implementation (72.9× and 9.4×), while causing similar timing and area overheads. This is because *crc*'s bitwise operations have low

delay and can be scheduled into a single stage without degrading processor frequency. Clay provides designers with the loop-unrolling directive to easily switch between different implementations for the best solution. Moreover, Clay also supports the *stream_add* custom instruction, which contains a `for`-loop that cannot be fully unrolled due to the variable loop bound, as shown in Figure 4a. Prior ASIP frameworks cannot implement this stateful custom instruction, while Clay synthesizes it into a coprocessor, achieving 3.9× and 2.1× overall speedup on `Clay-core` and `Rocket-core`, respectively.

2) Benefits of microarchitecture-aware synthesis: Table V presents that Clay selects the *in-pipeline* coupling strategy for implementing the four combinational custom instructions (*simd*, *complex_mul*, *sbox*, and *autoinc*) on `Clay-core`, since the *in-pipeline* coupling is more lightweight and hardware efficient. The experimental results show that the *in-pipeline* coupling on `Clay-core` achieves lower clock period overheads on the custom instructions except *complex_mul*, and uses less area overheads on all four custom instructions, compared against the *coprocessor* coupling on `Rocket-core`, even though `Rocket-core` has a larger base clock period and area. These results demonstrate that Clay selects more efficient coupling strategies.

For the *gemm2x2* and *stream_add* custom instructions, Clay implements them as *coprocessors* on both `Clay-core` and `Rocket-core`, since the two RISC-V processor pipelines cannot provide enough memory access channels to couple the custom instructions in the *in-pipeline* manner. For *gemm2x2*, although Clay's implementation achieves a much higher speedup (14.6×) on `Clay-core` than the speedup (1.5×) in `Rocket-core`, the advantage mainly comes from the fact that `Rocket-core` already has a built-in multiplier and its original execution cycle is much lower than that of `Clay-core`. Experiments on the *stream_add* custom instruction can better demonstrate the effectiveness of Clay's microarchitecture-aware scheduling introduced in Section III-D. `Clay-core` and `Rocket-core` have different microarchitecture configurations for memory access. `Clay-core`'s memory bus has the independent load and store channels, corresponding to Figure 5a, while `Rocket-core` uses the TileLink [31]-like HellaCache [6] memory interface, which has only one memory channel for load and store operations to share, corresponding to Figure 5b. Thereby, Clay's microarchitecture-aware scheduling produces implementations with II of 2 and 3 for `Clay-core` and `Rocket-core`, respectively, fully leveraging the microarchitectural resources. The experiments in Table V are conducted with the stream length of 8, and the execution cycles of `Clay-core` and `Rocket-core` with the custom instruction are 18 and 46, presenting the practical II of 2.25 and 5.8, respectively. `Clay-core`'s practical II is close to Clay's scheduled II of 2; however, `Rocket-core`'s practical II is much larger than the scheduled II of 3. The reason for the degraded performance is that `Rocket-core`'s cache has a minimal response latency of 2, which will trigger the stalling mechanism of Clay-synthesized coprocessor to wait for the response and worsen the performance. On the other hand, the better performance on `Clay-core` demonstrates that Clay's microarchitecture-aware scheduling can fully leverage the microarchitectural resources to exploit the acceleration potentials of custom instructions.

C. Accelerating real-world workloads

We also evaluate Clay using real-world workloads from computer vision and digital signal processing domains. To accelerate these workloads, multiple custom instructions are added and work jointly. We evaluate two representative workloads: (1) edge detection of colored images (*EdgeDet*) and (2) carrier frequency offset estimation (*CFOEst*). We customize 4 instructions, *rgba2gray*, *3×3 sobel*, *3×3 erode*, and *3×3 dialate* for *EdgeDet*, and 3 instructions, *complex_mean*, *complex_mul* and *cordic* for *CFOEst*. The custom instructions for the two workloads require CADL's control flow statements for microarchitecture-agnostic stateful description. It takes 87 LOC and 50 LOC in total to describe the two sets of custom instructions, respectively. The experiments are conducted on `Clay-core` according to the methodology described in Section IV-A3.

TABLE VI: Results of accelerating real-world workloads

Workload	#Cycle		#Instr		Speedup	Period	Area
	Orig.	Opt.	Orig.	Opt.			
EdgeDet	1173718	294249	1095685	17	4.0×	-0.1%	+39.0%
CFOEst	1523	36	1113	9	33.9×	+24.7%	+47.9%

Table VI presents the experimental results. With the custom instructions, the optimized program only requires 17 instructions to execute the *EdgeDet* workload, greatly mitigating processor frontend pressures such as instruction cache misses. Every custom instruction includes nested `for`-loop control flow and multiple memory access operations. Clay's microarchitecture-aware synthesis fully utilizes `Clay-core`'s memory channels and achieves the best II of 1 for all four custom instructions. Besides, the synthesized coprocessors have specific FSMs for the loop control, avoiding branch misprediction during the workload execution. For the *CFOEst* workload, the custom instructions reduce the instruction count from 1113 to 9, achieving a 33.9× speedup. These results demonstrate that Clay can significantly accelerate real-world workloads with flexible microarchitecture-aware instruction customization.

For timing and area overhead, the frequency for *EdgeDet* remains almost unchanged, and the area is increased by 39%, according to Table VI. The area overheads are spent on the custom registers introduced to buffer image tiles inside the coprocessors. *CFOEst*'s custom instructions cause a 24.7% increase in clock period and 47.85% in area. The relatively large timing overhead is due to the critical paths inside the newly added multipliers, and the increased area comes from the multipliers (19.4%), the CORDIC module (16.8%), and the custom registers for buffering (11.6%). Since the `Clay-core` base processor is a very small core without on-chip SRAM, the 47.9% area overhead is moderate and acceptable, especially considering the significant speedup achieved. These results highlight that Clay's features are fully exploited to significantly accelerate real-world workloads without introducing unacceptable hardware overheads.

V. Conclusion

This paper presents Clay, a high-level ASIP framework that enables accessible RISC-V-based ASIP customization. Clay delivers a unified instruction extension interface that supports diverse coupling strategies while enabling both stateful instruction behavior description and microarchitecture-aware instruction synthesis. Experimental evaluations on two RISC-V processors demonstrate significant performance gains—up to 203× on individual kernels and 34× on real-world workloads—while maintaining reasonable hardware overhead.

Acknowledgment

This work was supported in part by the National Science Foundation of China (Grant No. T2325001) and the National Key Research and Development Program of China (Grant No. 2022YFB4500401).

REFERENCES

[1] T. Ajayi, V. A. Chhabria, M. Fogaça, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim, J. Lee, U. Mallappa, M. Neseem, G. Pradipta, S. Reda, M. Saligane, S. S. Sapatnekar, C. Sechen, M. Shalan, W. Swartz, L. Wang, Z. Wang, M. Woo, and B. Xu, "Toward an Open-Source Digital Flow: First Learnings from the OpenROAD Project," in *Proceedings of the 56th Annual Design Automation Conference 2019*, Jun. 2019.

[2] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1983.

[3] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić, "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs," *IEEE Micro*, 2020.

[4] Andes Technology, "Andes Custom Extension™," publication Title: Andes Technology. [Online]. Available: https://www.andestech.com/en/products-solutions/andes-custom-extension/

[5] G. Armeniakos, A. Maras, S. Xydis, and D. Soudris, "Mixed-precision Neural Networks on RISC-V Cores: ISA extensions for Multi-Pumped Soft SIMD Operations," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, Apr. 2025.

[6] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, and others, "The Rocket Chip Generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016. [Online]. Available: https://aspire.eecs.berkeley.edu/wp/wp-content/uploads/2016/04/Tech-Report-The-Rocket-Chip-Generator-Beamer.pdf

[7] Bluespec Inc, "Accelerate-HLS." [Online]. Available: https://info.bluespec.com/acceleratehls

[8] Cadence Design Systems, Inc, "Cadence Tensilica Offerings," publication Title: Cadence Tensilica Offerings. [Online]. Available: https://www.cadence.com/en_US/home/tools/silicon-solutions/compute-ip/technologies.html

[9] H. Cheng, G. Fotiadis, J. Großschädl, D. Page, T. H. Pham, and P. Y. A. Ryan, "RISC-V Instruction Set Extensions for Multi-Precision Integer Arithmetic: A Case Study on Post-Quantum Key Exchange Using CSIDH-512," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, Jun. 2024.

[10] Codasip, "Codasip Studio," publication Title: Codasip. [Online]. Available: https://codasip.com/products/codasip-studio/

[11] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on SDC formulation," in *Proceedings of the 43rd annual Design Automation Conference*, 2006.

[12] P. Coussy and A. Morawiec, Eds., *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer, 2008. [Online]. Available: http://link.springer.com/10.1007/978-1-4020-8588-8

[13] M. Damian, J. Oppermann, C. Spang, and A. Koch, "SCAIE-V: an open-source SCAlable interface for ISA extensions for RISC-V processors," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, Jul. 2022.

[14] J. M. Domingos, N. Neves, N. Roma, and P. Tomás, "Unlimited vector extension with data streaming support," in *Proceedings of the 48th Annual International Symposium on Computer Architecture*. IEEE Press, 2021.

[15] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Oct. 2017.

[16] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," publication Title: Gurobi Optimization. [Online]. Available: https://www.gurobi.com/

[17] A. Hoffmann, O. Schliebusch, A. Nohl, G. Braun, O. Wahlen, and H. Meyr, "A methodology for the design of application specific instruction set processors (ASIP) using the machine description language LISA," in *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No.01CH37281)*, 2001.

[18] P. Ienne and R. Leupers, *Customizable Embedded Processors: Design Technologies and Applications*. Elsevier, Aug. 2006.

[19] M. Jain, M. Balakrishnan, and A. Kumar, "ASIP design methodologies: survey and issues," in *VLSI Design 2001. Fourteenth International Conference on VLSI Design*, Jan. 2001.

[20] L. Jia, Z. Luo, L. Lu, and Y. Liang, "TensorLib: A spatial accelerator generation framework for tensor algebra," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 865–870, ISSN: 0738-100X. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9586329

[21] H. Li, N. Mentens, and S. Picek, "A scalable SIMD RISC-V based processor with customized vector extensions for CRYSTALS-kyber," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, Aug. 2022.

[22] K. Liyanage, H. Gamaarachchi, H. Saadat, T. Li, H. Samarakoon, and S. Parameswaran, "Accelerating Chaining in Genomic Analysis Using RISC- V Custom Instructions," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Mar. 2024.

[23] Z. Luo, L. Lu, S. Zheng, J. Yin, J. Cong, J. Yin, and Y. Liang, "Rubick: A synthesis framework for spatial architectures via dataflow decomposition," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10247743

[24] P. Mishra and A. Shrivastava, "ADL-driven Methodologies for Design Automation of Embedded Processors," in *Processor Description Languages*, P. Mishra and N. Dutt, Eds., Jan. 2008.

[25] Nuclei System Technology, "NICE (Nuclei Instruction Co-unit Extension)." [Online]. Available: https://doc.nucleisys.com/hbirdv2/core/core.html#nice

[26] OpenHW Group, "Core-V eXtension interface (CV-X-IF)," Apr. 2025. [Online]. Available: https://github.com/openhwgroup/core-v-xif

[27] 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 *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, Apr. 2024.

[28] P. Scheffler, L. Colagrande, and L. Benini, "SARIS: Accelerating Stencil Computations on Energy-Efficient RISC-V Compute Clusters with Indirect Stream Registers," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024.

[29] F. Schuiki, F. Zaruba, T. Hoefler, and L. Benini, "Stream semantic registers: A lightweight risc-v isa extension achieving full compute utilization in single-issue cores," *IEEE Trans. Comput.*, vol. 70, no. 2, p. 212–227, Feb. 2021. [Online]. Available: https://doi.org/10.1109/TC.2020.2987314

[30] Siemens, "Catapult High-Level Synthesis Tools," publication Title: Siemens Digital Industries Software. [Online]. Available: https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/hls/

[31] SiFive Inc., "SiFive TileLink Specication Version 1.8.1." [Online]. Available: https://starfivetech.com/uploads/tilelink_spec_1.8.1.pdf

[32] Synopsys, Inc, "Synopsys ASIP Designer," publication Title: Synopsys ASIP Designer. [Online]. Available: https://www.synopsys.com/dw/ipdir.php?ds=asip-designer

[33] Veripool, "Verilator," Apr. 2025. [Online]. Available: https://www.veripool.org/verilator/

[34] C. Wolf, "Yosys open synthesis suite," 2016. [Online]. Available: https://yosyshq.net/yosys/

[35] Y. Xiao, Z. Luo, and Y. Liang, "cmt2: Rule-Based Hardware Description in Rust with Temporal Semantics," in *5th Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE'25)*, 2025.

[36] Y. Xiao, Z. Luo, K. Zhou, and Y. Liang, "Cement: Streamlining FPGA Hardware Design with Cycle-Deterministic eHDL and Synthesis," in *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Apr. 2024.

[37] R. Xu, Y. Xiao, J. Luo, and Y. Liang, "HECTOR: A multi-level intermediate representation for hardware synthesis methodologies," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '22. Association for Computing Machinery, pp. 1–9. [Online]. Available: https://dl.acm.org/doi/10.1145/3508352.3549370

[38] E.-Y. Yang, T. Jia, D. Brooks, and G.-Y. Wei, "FlexACC: A Programmable Accelerator with Application-Specific ISA for Flexible Deep Neural Network Inference," in *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Jul. 2021.

[39] Z. Zhang and B. Liu, "SDC-based modulo scheduling for pipeline synthesis," in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov. 2013.