2025 IEEE/ACM International Conference On Computer Aided Design (ICCAD) | 979-8-3315-1560-7/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICCAD66269.2025.11240817

Invited Paper: APS: Open-Source Hardware-Software Co-Design Framework for Agile Processor Specialization

Youwei Xiao, Yuyang Zou, Yansong Xu, Yuhao Luo, Yitian Sun, Chenyun Yin, Ruifan Xu, Renze Chen, Yun Liang*
Peking University, Beijing, China

***Abstract*—APS is an open-source framework for agile hardware-software co-design of domain-specific processors. It provides both hardware synthesis and compiler infrastructure to facilitate the development of instruction extensions (*ISAX*s) for application acceleration. The framework proposes a unified instruction extension interface for seamless integration with diverse RISC-V SoC ecosystems. Based on the unified interface, APS introduces a cross-level architecture description language (CADL) for comprehensive instruction behavior specification, which is translated into a dynamic pipeline architecture through its synthesis flow. Besides, APS's compiler infrastructure introduces a pattern-matching engine for the automated utilization of *ISAX*s in general programs. It also incorporates bitwidth-aware vectorization that leverages operand bitwidth information to reduce the overhead of calling *ISAX*s. We conduct case studies across multiple workloads, including cryptography, machine learning, and digital signal processing. With fewer than 175 lines of *ISAX* description, APS achieves 2.29× to 14.99× speedup for each case study, demonstrating APS's practical productivity and acceleration capability. Overall, APS offers a complete, end-to-end methodology that significantly reduces the development cycle of *ISAX*s, making agile processor specialization practical to the research and open-source hardware communities.**

I. Introduction

The rapid evolution of domain-specific applications demands agile development of Application-Specific Instruction-Set Processors (ASIPs) to maintain competitive performance and efficiency. RISC-V, as an open instruction set architecture (ISA), facilitates the integration of instruction extensions (*ISAX*s) with the general processor and enables processor specialization. RISC-V *ISAX*s have enabled extension and acceleration for diverse domains, including digital signal processing [16], artificial intelligence [5], and cryptography [10]. However, current open-source RISC-V ecosystems impose constraints on agile processor specialization, as they are tied to specific platforms or narrowly scoped design tasks, rather than providing a unified and general framework for hardware-software co-design.

We identify several fundamental challenges in developing domain-specific processors with *ISAX*s. The first challenge is *interface divergence*. Existing System-on-Chip (SoC) frameworks support different instruction extension interfaces, such as the Rocket Custom Coprocessor (RoCC) [6] interface and the Core-V eXtension interface (CV-X-IF) [1], which, although conceptually similar, lack interoperability. This fragmentation forces designers to learn the corresponding interface and implement their *ISAX*s in a compatible, low-level fashion for every target platform, significantly increasing development overhead and limiting design reuse. The second challenge is *ISAX-specific synthesis*. Agile ASIP development requires synthesis features that effectively bridge the gap between high-level algorithmic descriptions and efficient hardware implementations. Current tools fall short in several critical aspects: traditional HLS tools [14], [32] overlook the intricate interactions between *ISAX*s and the base processor system, such as memory system access; existing ASIP solutions, like Longnail [21], lack support for stateful hardware behaviors, including hardware loops and pipelines.

The final challenge is *compiler support*. Hardware agility must be matched by software development for practical utilization of *ISAX*s. Existing open-source ASIP frameworks [22] do not provide the *ISAX*-compliant compiler infrastructure featured by commercial toolchains, including Cadence Tensilica [9], Synopsys ASIP Designer [28], and Codasip Studio [12], preventing developers from conveniently leveraging their custom *ISAX*s. Moreover, the general-purpose register-based RISC-V instruction types put architectural constraints on the operand and result width of *ISAX*s, e.g., at most two 32-bit operands and one 32-bit result for an R-Type instruction. *ISAX*s should pack low-bitwidth arguments and results to leverage the precious register bandwidth. The compiler should automatically transform the original program to utilize such vectorized *ISAX*s, which is not explored by any prior solutions.

To address these challenges, we present *APS*, an open-source hardware-software co-design framework that enables agile research on domain-specific processor specialization. Specifically, we identify the common abstraction for *ISAX* integration and propose the *APS-Itfc* interface to provide seamless portability across different processors and SoC ecosystems, thereby addressing the challenge of interface divergence. *APS-Itfc* ensures compatibility across the Rocket Chip [6]-generated SoCs from the Chipyard [4] framework and the Croc [25] SoC from the PULP platform [23]. For *ISAX*-specific synthesis support, we implement a complete synthesis flow, *APS-Synth*, that translates a cross-level architecture description language, *CADL*, into a register-transfer level (RTL) implementation of the synthesized dynamic pipeline architecture. Our synthesis framework provides rich features, including *ISAX*-processor interaction, stateful hardware behavior, and flexible interoperability between high-level abstractions and low-level custom components. For the compiler infrastructure, we introduce *APSC*, which provides a hybrid pattern matching engine and a bitwidth-aware vectorization feature for automated and efficient utilization of custom *ISAX*s in general applications.

Our contributions are as follows:

- We propose the first open-source[1] hardware-software co-design framework *APS* for agile ASIP specialization.
- We introduce the *APS-Itfc* interface to provide seamless portability across different processor platforms, and propose the *APS-Synth* framework to provide *ISAX*-specific synthesis support.
- We present *APSC*, a compiler framework that enables automated utilization of *ISAX*s in general applications through pattern matching and bitwidth-aware vectorization.

We validate *APS* through comprehensive case studies spanning multiple application domains on two RISC-V SoC platforms. For cryptography, *APS* accelerates the Number Theoretic Transformation (NTT) and the polynomial multiplication in the NTT domain by up to 10.16× and 14.99×; For machine learning, the quantized dot-product *ISAX* accelerates BitNet b1.58 [19]'s *BitLinear* layer by 2.29×; For

*Corresponding author: ericlyun@pku.edu.cn

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

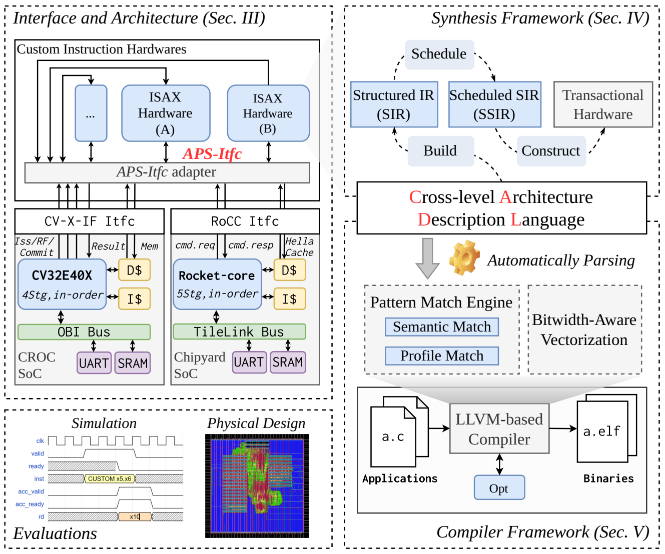


Fig. 1: Overview of the *APS* framework.

digital signal processing (DSP), *APS* accelerates a digital phase-locked loop (DPLL) by up to 8.43×. Each case study only requires the *ISAX* description of fewer than 175 SLOC[2], and *APS* automates the remaining hardware-software co-design tasks in an end-to-end flow.

II. Overview

Figure 1 provides an overview of the *APS* framework. *APS-Itfc* (Section III) is a unified *ISAX* interface with portability across diverse RISC-V platforms, including the open-source Chipyard [4] and PULP [23] projects. The *ISAX* hardware implementation is synthesized from high-level behavioral specifications written in the Cross-level Architecture Description Language, *CADL*. The synthesis framework, *APS-Synth* (Section IV), translates *CADL* into an efficient hardware implementation that interfaces with the base processors through the standard *APS-Itfc* interface. On the software side, the *APSC* compiler infrastructure (Section V) automatically generates *ISAX* invocation intrinsics and pattern matchers based on the *CADL* specification, facilitating practical adoption of *ISAX*s. *APSC* also introduces the bitwidth-aware vectorization optimization pass to exploit vectorized *ISAX*s. This end-to-end methodology streamlines the entire development process, from architectural specification to hardware generation and compiler integration, and runs RTL simulation and the physical design flow for comprehensive evaluation.

III. Interface and Architecture

In this section, we propose *APS-Itfc*, a unified interface for *ISAX*'s seamless integration into different RISC-V platforms.

A. *ISAX Interfaces Across RISC-V Platforms*

Leading open-source RISC-V projects have established standardized interfaces for *ISAX*s. The Chipyard framework [4] provides the Rocket Custom Coprocessor (RoCC) interface, which is implemented by Rocket [6] and BOOM [35] cores. Besides, the PULP platform [23] introduces the CORE-V eXtension Interface (CV-X-IF) [1], implemented in cores such as CV32E40X [26] and CVA6 [33].

[2]Significant lines of code excluding comments and blank lines

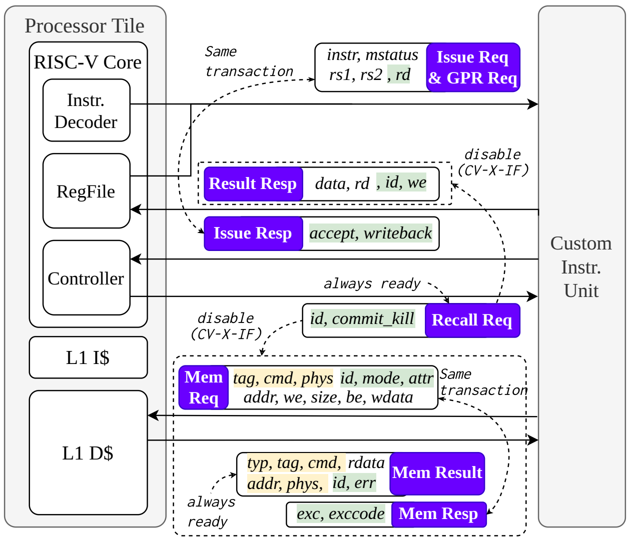


Fig. 2: Common processor-*ISAX* interaction of RoCC and CV-X-IF protocols. Yellow and green highlights denote protocol signals specific to RoCC and CV-X-IF, respectively. Common signals are shown without highlights.

Both RoCC and CV-X-IF adopt decoupled interface designs, separating the custom *ISAX* unit from the processor's main pipeline through a handshake protocol, as depicted in Figure 2. They share similar structures. The processor pipeline manages all *hazards*, ensuring that source operands are ready before dispatch. It then issues the decoded instruction and its source operands to the *ISAX*, which executes the operation—potentially involving memory accesses via the core's load/store unit—and finally writes the result back to the core's register file.

However, despite their architectural similarity, *ISAX*s targeting different interfaces are inherently incompatible without a complete and time-consuming reimplementation, due to differing interface specifications. RoCC prioritizes simplicity for accelerator designers. Specifically, its basic version consists of four primary channels: two pairs of request and response channels for instructions and main memory access. Crucially, RoCC guarantees the commitment of any offloaded *ISAX*s, relieving designers from implementing complex logic to handle instruction recalls. In contrast, CV-X-IF exposes a *commit* channel that explicitly notifies the *ISAX* unit of instruction commitment or recall. This recall mechanism requires any operations with side effects (e.g., memory accesses) to defer execution until confirmation, thereby necessitating a more complex *ISAX* controller. Moreover, CV-X-IF employs compact, bidirectional protocols. For instance, it merges issue requests and responses into a single handshake channel. Different protocol signals in Figure 2 demonstrate the protocol-level divergence.

B. *Unified ISAX Interface Abstraction*

Despite the significant disparities between existing *ISAX* interfaces, we observe that both interfaces share a common set of underlying transactional semantics. This shared semantic foundation enables the definition of a unified abstraction that hides backend-specific details, thereby facilitating *ISAX* portability. In *APS*, we propose the unified *transaction-based ISAX* abstraction, *APS-Itfc*. One *transaction* corresponds to a unidirectional data transfer between the processor and the *ISAX*, controlled by a pair of valid and ready signals. During

TABLE I: Semantics for *APS-Itfc* transactions

Transaction	**Signature**	**Semantics**
❶ Issue Req	()→(instr)	Receives a new instruction from the core to begin execution.
❷ GPR Req	()→(rf[N])	Receives source operands from the core's register file for the current instruction.
❸ Recall Req	()→(killed)	Check the instruction's commit status. A false value for killed guarantees execution.
❹ WrMem Req	(addr, wdata, mask)→()	Initiates a memory write access request to the core's Load/Store Unit.
❺ RdMem Req	(addr)→()	Initiates a memory read access request to the core's Load/Store Unit.
❻ RdMem Resp	()→(rdata)	Receives the result data (rdata) corresponding to a prior memory load request.
❼ Result Resp	(data)→()	Sends the final result to the core for write-back, signaling instruction completion.

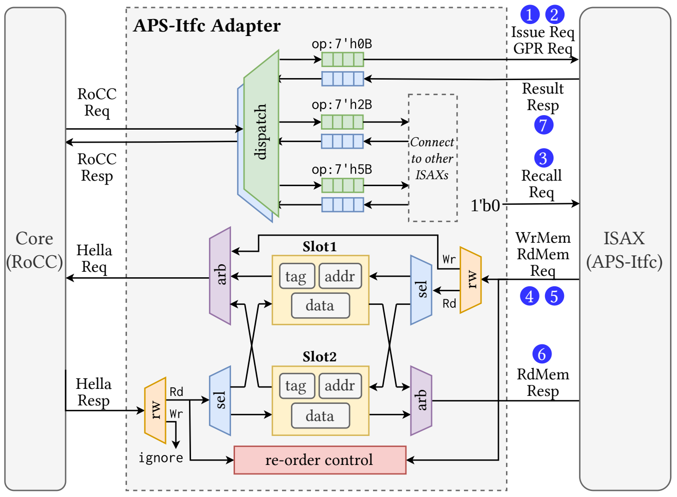


Fig. 3: *APS-Itfc* adapter implementation for RoCC interface.

hardware implementation, these transactions are instantiated as slave interfaces of bypass-capable FIFOs with control logic.

Table I summarizes the semantics for *APS-Itfc* transactions. Over the course of an *ISAX* execution, the processor first offloads it through an issue request ❶. *ISAX* units can then request two operands using the GPR Request ❷. For the Recall Request ❸, it stalls any side-effect operations and interface transactions of the *ISAX* until a not killed promise is received. *ISAX* can optionally request memory write ❹ and read ❺. Only critical signals are exposed in *APS-Itfc* transactions, while other backend-specific signals, such as tag and size, are handled automatically by the *adapter* logic. *ISAX* should wait for a memory response ❻ after sending out a memory request. The *ISAX* must send a Result Response ❼ to write back the result to the register file and notify that the execution has finished. *APS-Itfc* provides such unified and concise transactions to enable agile description of *ISAX*'s interactions with the processor and to support implementation across different backend platforms.

APS-Itfc adapter implementation: Adapter logic is necessary for hardware implementation of *APS-Itfc* towards the target platform. Figure 3 illustrates the adapter logic from *APS-Itfc* to RoCC. Each *ISAX* instance maintains dedicated instruction and return queues, indexed by opcode. Offloaded instructions are dispatched to the corresponding queue, and a shared arbiter selects one response per cycle for the core pipeline. To preserve the ordering of memory read responses, we employ a two-slot buffer under the control of a simple reorder controller that tracks request and response status, ensuring in-sequence delivery. Write responses are omitted for simplicity. The complete adapter logic requires only 425 SLOC for RoCC, demonstrating the ease of adapting *APS-Itfc* to a specific RISC-V platform like Rocket Chip [6]. CV-X-IF needs extra 338 SLOC for the adaptation due to its control logic complexity.

C. Support Multiple Open-Source RISC-V SoC Platforms

At the processor side, we select two representative open-source RISC-V cores to demonstrate the versatility of our approach. We support the Rocket core, a 5-stage in-order RISC-V core that implements the RoCC protocol, and the CV32E40X, a 4-stage in-order RISC-V core that implements the CV-X-IF protocol. To construct complete SoC designs, we leverage two prominent open-source SoC generator frameworks: Chipyard for Rocket and Croc for CV32E40X. Specifically, the Rocket core is integrated into the Rocket Chip generator to be compatible with the Chipyard ecosystem. For the CV32E40X, while Croc, an extensible SystemVerilog-based RISC-V microcontroller platform towards education, offers limited configurability, we extend it with parameterizable instruction and data caches, enabling robust deployment of real-world workloads.

IV. Hardware Synthesis Framework

Agile processor specialization requires synthesis support that translates high-level descriptions of *ISAX* behaviors into hardware implementations to boost design productivity. We propose the *APS-Synth* framework to synthesize the cross-level architectural description language (*CADL*). It features *APS-Itfc* support, flexible interoperability with low-level implementations, and dynamic pipeline generation.

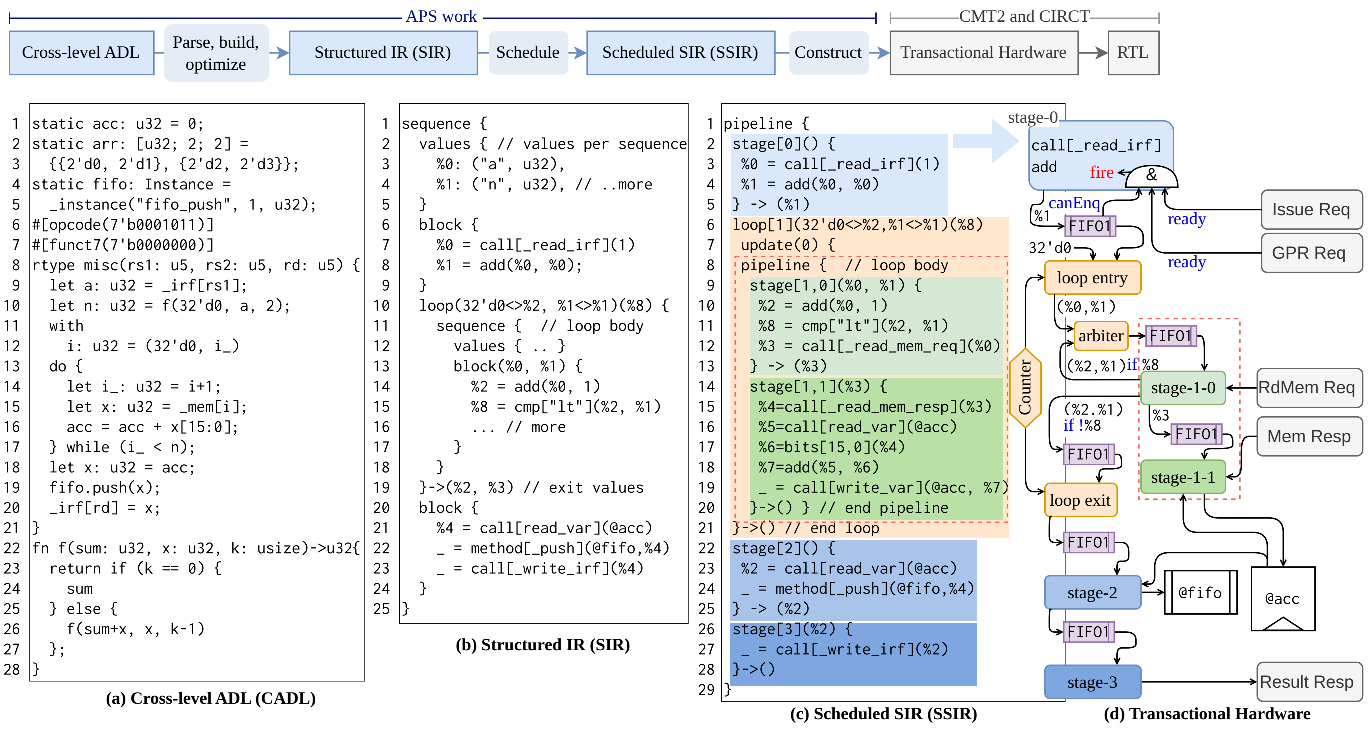
A. Cross-level Architectural Description Language

CADL not only describes the high-level behavior of *ISAX*s, supporting stateful control flow structures, but also provides direct access to both the unified *APS-Itfc* for processor-*ISAX* interaction and the custom low-level hardware components. Figure 4a shows an example of *CADL*. It comprises three parts:

Static instance definition: Lines 1-5 define three static instances: `acc` is a scalar variable, `arr` is an array variable, and `fifo` is a custom FIFO component. The static variables, either scalar or array, are built-in instances in *CADL*. Each variable corresponds to a register of the specified data type; for example, an array variable is implemented as a single register holding flattened data. In addition to the built-in instances, *CADL* provides a general mechanism to define custom components. For example, `fifo` at lines 4-5 instantiates a `fifo_push` module of depth 1 and element data type `u32`. *CADL* can instantiate any low-level hardware modules, either programmed in the CMT2 [30], [31] framework as transactional modules or imported as Verilog/SystemVerilog/FIRRTL [17] blackboxes. This provides the basis for *CADL*'s flexible interoperability with any low-level designs.

Compilation-time function: Lines 22-28 define a *function* named `f`, which calculates `x*k+sum` in a recursive manner. All functions in *CADL* are interpreted and inlined during compilation. This feature enables metaprogramming, reducing description complexity.

ISAX description: Lines 6-21 describe the *ISAX* named `misc`. Lines 6-8 describe the `misc`'s encoding, a R-Type instruction with the specified opcode and the func7 field. Line 9 reads the integer register file through the `_irf` syntax and assigns the result to a local variable named `a`. Line 10 calls the function `f` to calculate `n=a*2`. Lines 11-17 present a loop structure, describing stateful hardware control behavior. Line 12 following the `with` keyword specifies a loop-carried variable `i`, whose initial value is `32'd0` and is updated by the `i_` value inside the loop body (line 14). Lines 14-16 describe



```
 1 static acc: u32 = 0;
 2 static arr: [u32; 2; 2] =
 3   {{2'd0, 2'd1}, {2'd2, 2'd3}};
 4 static fifo: Instance =
 5   _instance("fifo_push", 1, u32);
 6 #[opcode(7'b0001011)]
 7 #[funct7(7'b0000000)]
 8 rtype misc(rs1: u5, rs2: u5, rd: u5) {
 9   let a: u32 = _irf[rs1];
10   let n: u32 = f(32'd0, a, 2);
11   with
12     i: u32 = (32'd0, i_)
13   do {
14     let i_: u32 = i+1;
15     let x: u32 = _mem[i];
16     acc = acc + x[15:0];
17   } while (i_ < n);
18   let x: u32 = acc;
19   fifo.push(x);
20   _irf[rd] = x;
21 }
22 fn f(sum: u32, x: u32, k: usize)->u32{
23   return if (k == 0) {
24     sum
25   } else {
26     f(sum+x, x, k-1)
27   };
28 }
```

(a) Cross-level ADL (CADL)

```
 1 sequence {
 2   values { // values per sequence
 3     %0: ("a", u32),
 4     %1: ("n", u32), // ..more
 5   }
 6   block {
 7     %0 = call[_read_irf](1)
 8     %1 = add(%0, %0);
 9   }
10   loop(32'd0<>%2, %1<>%1)(%8) {
11     sequence {  // loop body
12       values { .. }
13       block(%0, %1) {
14         %2 = add(%0, 1)
15         %8 = cmp["lt"](%2, %1)
16         ... // more
17       }
18     }
19   }->(%2, %3) // exit values
20   block {
21     %4 = call[read_var](@acc)
22     _ = method[_push](@fifo,%4)
23     _ = call[_write_irf](%4)
24   }
25 }
```

(b) Structured IR (SIR)

```
 1 pipeline {
 2  stage[0]() {
 3   %0 = call[_read_irf](1)
 4   %1 = add(%0, %0)
 5  } -> (%1)
 6  loop[1](32'd0<>%2,%1<>%1)(%8)
 7   update(0) {
 8   pipeline {  // loop body
 9    stage[1,0](%0, %1) {
10     %2 = add(%0, 1)
11     %8 = cmp["lt"](%2, %1)
12     %3 = call[_read_mem_req](%0)
13    } -> (%3)
14    stage[1,1](%3) {
15     %4=call[_read_mem_resp](%3)
16     %5=call[read_var](@acc)
17     %6=bits[15,0](%4)
18     %7=add(%5, %6)
19     _ = call[write_var](@acc, %7)
20    }->() } // end pipeline
21  }->() // end loop
22  stage[2]() {
23   %2 = call[read_var](@acc)
24   _ = method[_push](@fifo,%4)
25  } -> (%2)
26  stage[3](%2) {
27   _ = call[_write_irf](%2)
28  }->()
29 }
```

(c) Scheduled SIR (SSIR) **(d) Transactional Hardware**

Fig. 4: APS Synthesis Framework.

the loop body, where line 15 reads the processor memory through the `_mem` syntax and line 16 accumulates the result to the `acc` static variable. Line 17 describes the continuation condition of the loop. Lines 18-19 show how *CADL* accesses or updates the general custom instance: it calls the method `push` of the `fifo` instance to push an element. *CADL*'s *ISAX* description provides rich features from high-level behavior to low-level module access, enabling comprehensive *ISAX* accelerator design for diverse customization requirements.

B. Intermediate Representations

To synthesize *CADL* into a hardware implementation, we propose and implement two intermediate representations: Structured IR (SIR) and Scheduled Structured IR (SSIR). Figure 4b and Figure 4c present examples of each, respectively.

SIR: directly corresponds to the *CADL* description, providing key structures to describe *ISAX*s' high-level behavior, including `sequence`, `block`, and `loop`. SIR facilitates program analysis and transformations, such as *type inference* and *function interpretation*. A `sequence` comprises a list of `block`s and `loop`s, with a table to store the values living in the `sequence`. In Figure 4b, the outermost `sequence` (lines 1-25) includes two `block`s (lines 6-9 and lines 20-24) and one `loop` (lines 10-19), with a `value` scope (lines 2-5) holding the value table. The high-level syntax in *CADL* is lowered to primitive operations. For example, reading `_irf` is transformed into `call(_read_irf)` at line 7, and the method call `fifo.push` is transformed into the `method` primitive at line 22. The `loop` structure specifies the loop-carried variables within the first parentheses and the condition value in the second. At line 10, there are two loop-carried variables: the `i` as described in the *CADL*, and the `n`, which is passed into the loop body to calculate the condition. Both initial values and update values are passed into the loop body sequence as the arguments of the first `block` (line 13). The condition value for the loop is `%8`, calculated at line 15. The updated values from the final iteration of the loop are returned to the outer sequence (`%2` and `%3` at line 19) for subsequent usage.

SSIR: represents the scheduled results of the *ISAX* behavior. Without losing generality, SSIR adopts the `pipeline`-based representation with `stage`s as the scheduling units. Figure 4c shows a scheduling solution for SIR in Figure 4b. The outermost `pipeline` structure, mapped from a `sequence`, includes four stages, whose *stage id* is shown in the brackets. Notably, `loop[1]` is treated as a stage whose execution spans multiple cycles. Besides, every `loop` specifies an *update stage*, indicating which stage prepares for the execution of the next iteration. In Figure 4c, `loop[1]`'s *update stage* is 0, which means the next iteration will start after `stage[1,0]`, indicating the initial interval (II) of 1. Every `stage` inside the loop body `pipeline` is prefixed with the loop's *ID* in their *stage IDs*. In addition, the *value passing* between the neighbor stages is encoded in SSIR. For example, `stage[2]` passes the value `%2` to `stage[3]`. SSIR provides the appropriate abstraction for representing scheduling solutions, estimating performance, and preparing hardware generation.

C. Synthesis Flow

Figure 4 presents the synthesis flow from the *CADL* to the final RTL implementation. The *CADL* is first parsed to build SIR, on which we conduct type inference, function interpretation, and general optimizations such as dead code elimination. Then, the SIR is scheduled into SSIR, which is further used to construct transactional hardware in a dynamic pipeline architecture. The transactional hardware is constructed on the CMT2 [30] framework, which translates the design into RTL description through the CIRCT [11] project.

ISAX-specific Scheduling: We implement a SDC [13]-based scheduling infrastructure to schedule the *ISAX* behavior. In addition to the general scheduling constraints, such as data dependency constraints and clock period constraints, we consider more *ISAX*-specific constraints: operation position constraints, which specifies the required scheduling stage of certain operations, for example, `call(_read_irf)` must be scheduled at the first stage since the register file values are passed into an *ISAX* at its trigger; latency-insensitivity constraints, which specifies that two latency-insensitive operations cannot be scheduled at the same stage, avoiding potential *deadlocks* caused by execution misalignment. Both constraints are encoded as difference constraints in the SDC formulation. For loop scheduling, we currently adopt the normal SDC scheduling on the loop body sequence, without modulo considerations [34].

Dynamic Pipeline Construction: We construct a dynamic pipeline architecture in the *transactional* manner from the SSIR. Every stage constructs a transaction, which executes its included state update actions when its *fire* logic holds. For example, the `stage[0]` in Figure 4c constructs the transaction `stage-0` in Figure 4d, whose actions include sending an *ISAX* issue request, sending a GPR access request, and enqueueing the summed value into a FIFO for the next stage to use. The first two actions correspond to *APS-Itfc* transactions in Table I, and they are *ready* only when the processor executes the *ISAX*. The *fire* logic of `stage-0` requires that the called *APS-Itfc* transactions are ready and the output can be enqueued to the non-full FIFO. Besides, the loop structure, such as `loop[1]` in Figure 4c, is also implemented as transactions, including a loop entry and a loop exit. The loop entry transaction gets the initial values prepared by the previous stage and pushes them into the starting FIFO of the loop body pipeline. The loop update stage, such as `stage[1,0]` in Figure 4c, pushes the prepared update values for loop-carried variables to either the starting FIFO of the loop body pipeline if the condition is met, or to the FIFO leading the loop exit transaction otherwise. The loop is equipped with a *counter*, indicating the number of in-flight iterations. The loop entry and the loop exit can only execute when the counter is zero, indicating the end of the loop structure. The dynamic pipeline architecture features strong latency insensitivity tolerance. Besides, our architecture supports general loop pipelining even for nested loop structures. Moreover, *arbiters* are necessary for components to be updated by multiple transactions, such as the leading FIFO of a loop body pipeline and the static instances (`fifo` and `acc`). They are automatically inserted by the CMT2 compiler during RTL generation.

V. Software Compiler Framework

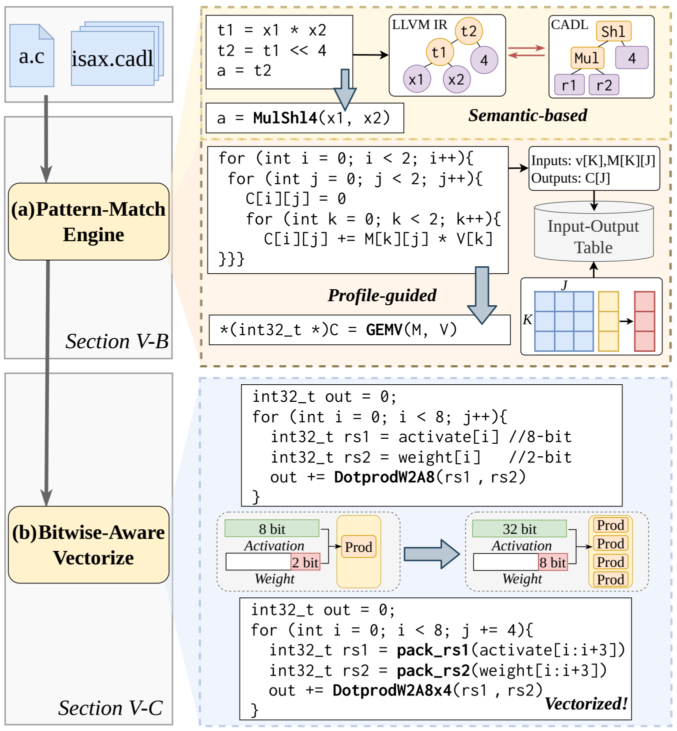
In this section, we propose *APSC*, an integrated compilation framework that enables applications to adopt *ISAX*s for acceleration automatically.

A. Compiler infrastructure

The overall compilation flow is depicted in Figure 5. It begins with a Clang-based frontend [18] that translates C source code into LLVM IR. *APSC* then automatically generates intrinsic-like C wrappers from *CADL* specifications, enabling transparent invocation of custom *ISAX*s. Building upon this foundation, we introduce two key *APSC* features to specialize the compilation for domain-specific *ISAX*s. First, *APSC* employs a hybrid pattern-matching engine comprising semantic-based matching followed by profile-guided matching (Section V-B). Second, *APSC* introduces a bitwidth-aware vectorization pass to exploit parallelism within architectural constraints (Section V-C).

B. Pattern Matching Engine

While compiler intrinsics provide a convenient interface for calling *ISAX*s, developers still need to manually modify their programs to include them, which severely increases the learning curve and brings



```
t1 = x1 * x2
t2 = t1 << 4
a = t2
```

```
a = MulShl4(x1, x2)
```

```
for (int i = 0; i < 2; i++){
 for (int j = 0; j < 2; j++){
   C[i][j] = 0
   for (int k = 0; k < 2; k++){
     C[i][j] += M[k][j] * V[k]
}}}
```

```
*(int32_t *)C = GEMV(M, V)
```

```
int32_t out = 0;
for (int i = 0; i < 8; j++){
  int32_t rs1 = activate[i] //8-bit
  int32_t rs2 = weight[i]   //2-bit
  out += DotprodW2A8(rs1 , rs2)
}
```

```
int32_t out = 0;
for (int i = 0; i < 8; j += 4){
  int32_t rs1 = pack_rs1(activate[i:i+3])
  int32_t rs2 = pack_rs2(weight[i:i+3])
  out += DotprodW2A8x4(rs1 , rs2)
}
```

Fig. 5: APS Compiler workflow.

more performance tuning challenges. To automate this process, *APSC* incorporates a pattern-matching engine.

APSC automatically parses the SIR emitted by *APS-Synth* to construct semantic-based matching functions. Leveraging LLVM's built-in pattern-matching infrastructure, these functions can efficiently identify and substitute semantically equivalent instruction sequences during the transformation phase. To address complex control flow constructs, such as nested loops, *APSC* further employs profile-guided matching. Specifically, we implement an LLVM IR interpreter to simulate the program execution. For each program region (e.g., a basic block or a loop body), *APSC* analyzes use-def chains to identify the region's input and output variables. In particular, instructions that write to memory are treated as outputs because of their side effects. The interpreter dynamically compares the observed input-output behavior of this region with representative input-output pairs collected offline for each *ISAX*. Once input-output matching is confirmed, the compiler automatically rewrites the matched region to invoke the corresponding *ISAX* intrinsic, enabling automated adoption of *ISAX*s with complex behavior. Since the profile-guided comparison cannot guarantee the full equivalence between the general program region and the *ISAX* behavior, *APSC* reports the matching and rewriting status and allows the users to revert any unexpected actions.

As illustrated in Figure 5a, *APSC* first extracts the behavioral semantics of the *ISAX*s through offline semantic parsing and automatically generates matching functions capable of detecting equivalent instruction sequences, allowing the replacement of these patterns with dedicated *ISAX* calls, such as `MulShl4` in Figure 5a. Beyond this, the compiler further analyzes loop regions, identifying arrays `v` and `M` as inputs and `C` as the output, and uses profiled input-output data to verify functional equivalence with the custom `GEMV` *ISAX*. Upon a successful match, the compiler rewrites the loop to directly invoke the `GEMV` intrinsic, replacing the original computation.

By employing the pattern-matching engine, *APSC* effectively han-

Algorithm 1: Bitwidth-Aware Vectorization

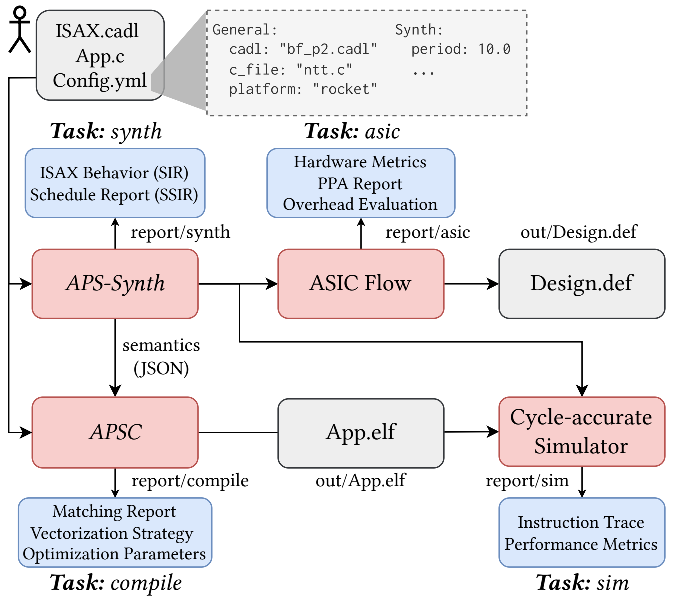
```
   Input: LLVM IR function $F$, custom ISAX information $\mathcal{C}$
   Output: Vectorized function $F'$
1  for each region $R$ in $F$ do
2      $seeds \leftarrow$ DetectSeeds($R$, $\mathcal{C}$);
3      if CanVectorize($seeds$, $\mathcal{C}$) then
4          $vecIntrinsic \leftarrow$ Pack($seeds$);
5          if not BitwidthFits($seeds$) then
6              $config \leftarrow$ GenConfig($excess\ inputs$);
7              Insert $config$ before $seeds$;
8          end
9          Replace $seeds$ with $vecIntrinsic$;
10     end
11 end
12 return $F'$
```

dles both control-free instruction sequences and complex control-flow regions, eliminating the need for manual intrinsic insertion for scalable adoption of *ISAX*s.

C. Bitwidth-Aware Vectorization

The *APSC* compiler infrastructure enables *ISAX*-specific compiler optimizations. Here we present *bitwidth-aware vectorization*, which enables the simultaneous execution of multiple data operations, significantly boosting computational throughput. While traditional vectorization techniques [20], [24] are effective for standard element width, they overlook the subword-level register-bitwidth utilization problem, which is especially obvious for *ISAX* scenarios. Consider the BitNet [19] dot product shown in Figure 5c, which computes an 8-element dot product with 8-bit activations and 2-bit weights. Each invocation occupies only 10 bits within a 32-bit register, resulting in the underutilized register bitwidth. To improve utilization, *APSC* detects a corresponding custom *ISAX* `DotprodW2A8x4` and packs four consecutive activations and weights into two operands, enhancing register bitwidth usage. Furthermore, when the packed bitwidth surpasses the architectural limit, e.g., at most two 32-bit operands and one 32-bit result for an R-Type *ISAX*, *APSC* needs to insert auxiliary memory operations to accommodate the overflow data automatically.

To systematically harness these optimization opportunities under bitwidth constraints, *APSC* analyzes the program to identify scalar custom *ISAX*s that can be promoted to existing SIMD-style vectorized forms. Algorithm 1 outlines the algorithm in detail. It begins by iterating over all regions in the function F. For each region, it applies `DetectSeeds` (line 2) to detect a set of individual calls to scalar *ISAX*s that can potentially be replaced with a SIMD *ISAX*, referred to as *seeds*. Next, the algorithm checks whether the selected *seeds* can be co-scheduled without introducing dependencies (line 3). If so, the inputs are bit-level concatenated via the `Pack` (line 4) function, which inserts necessary operations on the input values in the general program to prepare packed operands for the *ISAX*s. Subsequently, `BitwidthFits` (line 5) checks whether the packed total bitwidths fit within architectural constraints. For the case that the total bitwidth exceeds the register bandwidth, Algorithm 1 calls `GenConfig` (line 6) to allocate stack memory for the overflow variables and construct a *config ISAX*, which is an auxiliary *ISAX* preparing the initial values of the target *ISAX*'s static registers from the given memory address range. The call to the *config ISAX* is inserted before the *seeds* as a preprocessing step of the vectorized *ISAX* execution (line 7). Finally, the identified *seeds* are replaced with the vectorized *ISAX* (line 9), improving performance without violating the original functionality. Overall, *APSC* balances register constraints and performance by consolidating low-bitwidth operands and introducing auxiliary config *ISAX*s for overflow handling, thus enhancing bitwidth utilization and execution efficiency.



```
General:                  Synth:
  cadl: "bf_p2.cadl"        period: 10.0
  c_file: "ntt.c"           ...
  platform: "rocket"
```

Fig. 6: *APS* framework workflow

VI. Experiment Evaluation

In this section, we describe the toolchain workflow of *APS* and evaluate it by three case studies across different application domains, including post-quantum cryptography, machine learning, and digital signal processing.

A. Tool Demonstration

As shown in Figure 6, the workflow starts with a *CADL* design, C-based application source code, and a configuration file (YAML), which allows users to specify custom instructions, design constraints, and target hardware platforms. *APS* users can either run modular tasks for individual hardware/software design processes or run a single `build-all` command to trigger the end-to-end design flow. Specifically, the `synth` task launches *APS-Synth*, which synthesizes the *CADL* into SystemVerilog implementation at the given timing constraints, while exporting *CADL* into JSON-formatted semantics for the compiler infrastructure to use. A schedule report is generated, enabling designers to analyze pipeline timing, latency, and design trade-offs. The SoC is then automatically assembled using the respective SoC generator. The `compile` task invokes *APSC* to compile the user's C program, perform *ISAX* matching according to its semantics, and apply optimizations like vectorization. A concise report summarizes matching details, vectorization patterns, and inserted memory operations, allowing designers to trace how *ISAX*s are adopted in their source code.

For evaluation, the `sim` task runs a cycle-accurate RTL simulator for both baseline binary without triggering any *ISAX* units and binary compiled by *APSC* to exploit *ISAX*s, validating correctness and reporting performance gains. Detailed performance metrics, including cycle counts, instruction numbers, and CPI, are automatically recorded, allowing developers to identify bottlenecks and quantify the performance impact of their *ISAX* design. When the design meets users' performance expectation, the `asic` tasks can be used to launch the ASIC backend flow, generating the final layout and reporting power, performance, and area metrics. These results are then compared with baseline processors to quantify the benefits and the overhead introduced by *ISAX*s. This analysis enables developers to evaluate trade-offs and improve their CADL design to achieve a better balance between performance gains and hardware overheads.

In summary, APS delivers an automated flow from *CADL* and application program to *ISAX* hardware synthesis, software integration, performance evaluation, and ASIC implementation. Each design task provides key artifacts and reports, enabling rapid feedback during the agile development of specialized *ISAX*s.

B. Evaluation Setup

In our evaluation, cycle information is collected by cycle-accurate hardware simulation using Verilator v5.034 [27], using testbenches provided by Chipyard [4] and Croc [25]. Physical implementation metrics are derived through complete ASIC logic synthesis and physical placement, using Yosys 0.50 [29] and OpenROAD v2.0-16235 [3], respectively. We update original flow scripts to match our customized RTL design, including resizing chip area, inserting SRAM macros, and porting to the sg13g2 process. These ASIC evaluations are conducted at the SoC level, including instruction and data caches. For the compiler infrastructure, we use LLVM-18.0.1 [18] to implement our algorithms. We evaluate two baseline SoCs, both running general C programs without any *ISAX*s. The first is a Rocket tile featuring a Rocket core with a 16kB and 4KB direct-mapped instruction and data caches, respectively. Our ASIC flow yields a physical design operating at 160.7MHz with a total area of 1.35mm^2. The second is the Croc system, which integrates a CV32E40X core with 8KB and 4KB 2-way set-associative instruction and data caches, respectively. Its physical implementation achieves a frequency of 69.2MHz and occupies 5.74mm^2.

C. Accelerating Post-Quantum Cryptography Workloads

CRYSTALS-KYBER [8] has been standardized as the key encapsulation mechanism (KEM) in the domain of Post-Quantum Cryptography (PQC). CRYSTALS-KYBER is defined over the polynomial ring $\mathbb{Z}_q[x]/\langle x^N+1\rangle$, where N=256 and q=3329, and the polynomial multiplication is its computational bottleneck. Number Theoretic Transform (NTT) [2] is widely adopted to accelerate CRYSTALS-KYBER. The $O(n^2)$ complexity of the large-degree polynomial multiplication on the ring is significantly reduced to $O(nlogn)$ of point-wise multiplication (PWM) in the NTT domain. However, NTT and PWM still suffer from long latency due to the butterfly kernel and the 1-degree polynomial multiplication. They occupy 76% and 90% of the execution latency, respectively. The butterfly kernel in NTT is defined as:

$$f'_{j+\text{len}} \leftarrow \left(f_j - f_{j+\text{len}} \cdot \zeta_i\right) \pmod q,$$
$$f'_j \leftarrow \left(f_j + f_{j+\text{len}} \cdot \zeta_i\right) \pmod q.$$

where ζ_i is the precomputed rotation factor. The computation of degree-1 polynomial multiplication in PWM is given by:

$$h_{2i} \leftarrow \left(f_{2i}g_{2i} + \zeta_{2i+1}\, f_{2i+1}g_{2i+1}\right)\pmod q,$$
$$h_{2i+1} \leftarrow \left(f_{2i}g_{2i+1} + f_{2i+1}g_{2i}\right)\pmod q.$$

Implementation: To reduce the overall latency of NTT and PWM, we implement two butterfly *ISAX*s, the scalar version Butterfly and the 2-way parallel version Butterflyx2, and a Karatsuba *ISAX* for fast polynomial multiplication. The *ISAX*s are described with only 175 SLOC in *CADL*. *APSC* automatically utilizes the *ISAX*s in the original program via pattern matching.

TABLE II: Evaluation results of NTT and PWM *ISAX*s

Workload	*ISAX*	Croc		Rocket Tile	
		#Cycles	**Speedup**	**#Cycles**	**Speedup**
NTT	-	76,955	-	112,328	-
	Butterfly	23,052	3.34×	21,796	5.15×
	Butterflyx2	12,323	6.24×	11,053	10.16×
		Area	**Freq.**	**Area**	**Freq.**
	Butterfly	+5.44%	-0.35%	+17.67%	-2.58%
	Butterflyx2	+5.02%	-6.92%	+20.00%	-7.57%
PWM		**#Cycles**	**Speedup**	**#Cycles**	**Speedup**
	-	38,677	-	56,842	-
	Karatsuba	3,808	10.16×	3,793	14.99×
		Area	**Freq.**	**Area**	**Freq.**
	Karatsuba	+5.37%	-0.35%	+19.37%	-4.83%

Results: Table II shows the latency and speedup of NTT and PWM implemented with *ISAX*s in *APS*. The Butterfly *ISAX* achieves 3.34× and 5.15× speedup on the targeted two platforms, as the *ISAX*s adopts the efficient Barrett modular multiplication [7] and the datapath is fully-pipelined by *APS-Synth*. *APSC* automates vectorization for the Butterfly *ISAX* but only achieves 3.88× speedup on the Rocket tile, which is even worse than the scalar version. This performance degradation is primarily caused by additional memory access latency introduced by Config calls inserted during automated vectorization. To address this, we manually inserted Butterflyx2 *ISAX* to implement more compact 2-paralleled vectorization. Butterflyx2 yields speedups of up to 6.24× and 10.16× on the two platforms, nearly doubling the gains over the Butterfly *ISAX*. The Karatsuba *ISAX* achieves 10.16× and 14.99× speedup over the PWM baseline, which is attributed to the customised fast multiplication datapath described in CADL and synthesised in *APS*. The ASIC flow analysis reveals that the Butterfly *ISAX* and the Butterflyx2 *ISAX* lead to 5.44% and 5.02% more area on the Croc platform, respectively, while causing a frequency decrease by 0.35% and 6.92%. On the Rocket tile, the area overheads rise to 17.67% and 20.00%. These higher overheads are attributed to two factors: the base area of the Rocket tile is smaller than that of the Croc, and the Rocket tile has a higher frequency target, which requires more pipeline stages of the *ISAX* implementation, leading to more area overheads. For frequency, the Butterfly and the Butterflyx2 *ISAX* decrease the Rocket tile frequency by 2.58% and 7.57%, respectively. The Karatsuba *ISAX* incurs area overheads of 5.37% and 19.37%, and 0.35% and 4.83% frequency degradation. Both the area and frequency overheads are attributed to hardware multipliers.

D. Accelerating Quantized Large Language Model Workloads

1-bit Large Language Models (LLMs) represent a recent advancement in AI that focuses on extreme efficiency, where at most two bits are used to represent each weight in the model, as opposed to the standard 16-bit or 32-bit floating-point numbers. A prominent example of 1-bit LLMs is Microsoft's BitNet b1.58 [19], which uses ternary representation for each weight: {-1, 0, +1}. Despite the promising efficiency gains of 1-bit LLMs, their practical performance benefits are currently constrained by the lack of specialized computational units on general-purpose processors for operations involving quantized parameters. For instance, the dot product operation of 8-bit inputs and 2-bit weights typically relies on inefficient 8-bit multiplications or look-up table (LUT) methods.

Implementation: In *APS*, we design the dot product of 8-bit inputs and 2-bit weights as a SIMD *ISAX*, named dotprodW2A8x4, which is the core computation within BitNet's *BitLinear* layers [19].

TABLE III: Evaluation results of 1-bit LLM *ISAX*

Workload	*ISAX*	Croc		Rocket Tile	
		#Cycles	**Speedup**	**#Cycles**	**Speedup**
DotProd	-	35,146	-	12,014	-
	dotprodW2A8x4	14,006	2.51×	2,461	4.88×
BitLinear	-	789,966	-	336,531	-
	dotprodW2A8x4	388,586	2.03×	146,939	2.29×
BitNet	-	≈3.48e10	-	≈3.48e10	-
	dotprodW2A8x4	≈2.19e10	1.59×	≈2.05e10	1.70×
		Area	**Freq.**	**Area**	**Freq.**
	dotprodW2A8x4	+1.19%	-2.36%	+3.36%	+1.30%

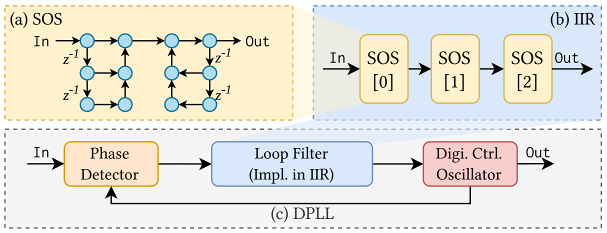


Fig. 7: Structure of Digital Phase-Locked Loop (DPLL): (a) A SOS block as the building unit of the filter; (b) An IIR filter composed of cascaded SOS blocks; (c) A DPLL incorporating the IIR filter.

Specifically, this instruction computes the dot product of four 8-bit inputs packed into a word and four 2-bit weights packed into a byte, which is supported by *APSC* through the generated pattern matcher and the bitwidth-aware vectorization pass.

Results: Table III shows the evaluation results of the proposed *ISAX* for 1-bit LLMs. We evaluate the standalone dot product and the *BitLinear* layer on two platforms: Croc and Rocket tile. For a single dot-product kernel, the *ISAX* achieves a speedup of 2.51× and 4.88× over the baseline on Croc and Rocket tile, respectively. For a complete *BitLinear* layer, the *ISAX* achieves 2.03× and 2.29× speedup on the two platforms. In this case study, *APSC* can automatically utilize the proposed *ISAX* in the C implementation of *BitLinear* using the profile-based pattern match engine, thereby achieving acceleration with minimal hardware-software design efforts. The ASIC flow reports that the dotprodW2A8x4 *ISAX* leads to 1.19% and 3.36% more area on the two platforms, respectively, while causing a frequency decrease by 2.36% and an increase by 1.30%. These results demonstrate that the hardware overhead of the *ISAX* is fully acceptable in practice. Furthermore, we estimate the speedup potential of the proposed *ISAX* on the complete BitNet model. Profiling the BitNet-b1.58-2B-4T configuration reveals that the BitLinear layers account for approximately 73% of the total inference time. Based on this observation, we project that the end-to-end BitNet inference can be accelerated by 1.59x and 1.70x on the two platforms, respectively.

E. Accelerating Digital Signal Processing Workloads

IIR (Infinite Impulse Response) filters are widely employed in digital signal processing for their computational efficiency and compact implementation. Higher-order IIR filters are typically decomposed into cascades of second-order sections (SOS), which enhance numerical stability and simplify fixed-point implementations. Our implementation builds upon the open-source *liquid-dsp* [15] library.

Implementation: We first design an SOS *ISAX* to accelerate a single SOS stage of the IIR filter. This *ISAX* operates on 16-bit fixed-point input samples, updates the internal filter state using customized registers, and eliminates repeated memory accesses per stage. However, executing multiple SOS stages by calling the SOS *ISAX* multiple times still incurs software control overheads. Thereby, we introduce a second *ISAX*, named IIR, that processes an entire SOS cascade in a pipelined manner. The IIR *ISAX* supports up to four SOS stages and leverages zero-overhead hardware loop to maximize throughput. The SOS *ISAX* optimizes latency by avoiding redundant loads and stores, and the IIR *ISAX* further enhances throughput by eliminating software loop control and redundant *ISAX* calling. We evaluate the implementation using a digital phase-locked loop (DPLL) as a representative workload (see Figure 7), where the IIR filter acts as a loop filter.

TABLE IV: Evaluation results of SOS/IIR *ISAX*s

Workload	*ISAX*	Croc		Rocket Tile	
		#Cycles	**Speedup**	**#Cycles**	**Speedup**
DPLL	-	15599	-	21706	-
	SOS	7048	2.21×	6923	3.14×
	IIR	2829	5.51×	3598	6.03×
		Area	**Freq.**	**Area**	**Freq.**
	SOS	+9.87%	-2.75%	+44.10%	-6.77%
	IIR	+9.89%	+1.98%	+44.62%	-3.92%

Results: Table IV presents the cycle-accurate latency and the speedup over the software baseline. On Croc, the SOS *ISAX* achieves a 2.21× speedup, while the IIR *ISAX* shows even greater performance with a 5.51× speedup. On Rocket tile, their speedups are 2.78× and 5.18×, respectively, presenting consistent gains across architectures. These results demonstrate that the *APS ISAX*s substantially improves the throughput of the DSP application. When comparing the performance gains of the SOS *ISAX* and the IIR *ISAX*, the IIR *ISAX* achieves 2.49× higher throughput than SOS on Croc and 1.92× on Rocket tile, showing more performance improvement. The reason is that *APS-Synth* effectively schedules the arithmetic operations in the chained IIR *ISAX* behavior, synthesizing a deep pipeline and the hardware loop logic, which eliminates the software *ISAX* calling and loop overhead of the SOS *ISAX*. The area overhead results present platform-dependent characteristics. While both *ISAX* implementations show an area increase of less than 9.9% on Croc, they rise to over 44% on Rocket tile. The reasons for the huge area overhead disparity are explained in subsection VI-C. For frequency, SOS *ISAX* causes a 2.75% decrease on Croc and 6.77% on Rocket tile, while the IIR *ISAX* even improves frequency by 1.98% on Croc and incurs only 3.92% reduction on Rocket tile.

VII. Conclusion

APS provides an open-source hardware-software co-design framework that streamlines the development of domain-specific processors. It aligns heterogeneous processor architectures through a unified *ISAX* interface abstraction, introduces a *ISAX*-specific synthesis flow for the cross-level architectural description language, and provides the compiler infrastructure that automates pattern matching and bitwidth-aware vectorization. Case studies across three application domains, including post-quantum cryptography, machine learning, and digital signal processing, comprehensively demonstrate *APS*'s acceleration capabilities and productivity in practice. The integrated *APS* framework establishes the foundation and provides convenience for future research on RISC-V *ISAX* design and compiler optimization.

Acknowledgment

This work was supported in part by the National Science Foundation of China (Grant No. T2325001).

References

[1] "The openhw group core-v-xif interface." [Online]. Available: https://docs.openhwgroup.org/projects/openhw-group-core-v-xif/en/latest/

[2] R. C. Agarwal and C. S. Burrus, "Number theoretic transforms to implement fast digital convolution," *Proceedings of the IEEE*, vol. 63, no. 4, pp. 550–560, 2005.

[3] 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.

[4] 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*, vol. 40, no. 4, pp. 10–21, 2020.

[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 *et al.*, "The rocket chip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, vol. 4, pp. 6–2, 2016.

[7] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," in *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 1986, pp. 311–323.

[8] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehle, " CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM ," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. Los Alamitos, CA, USA: IEEE Computer Society, Apr. 2018, pp. 353–367. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/EuroSP.2018.00032

[9] 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

[10] 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.

[11] CIRCT community, "CIRCT," 2025. [Online]. Available: https://circt.llvm.org/

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

[13] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on SDC formulation," in *Proceedings of the 43rd annual Design Automation Conference*, ser. DAC '06. New York, NY, USA: Association for Computing Machinery, 2006, pp. 433–438. [Online]. Available: https://dl.acm.org/doi/10.1145/1146909.1147025

[14] F. Ferrandi, V. G. Castellana, S. Curzel, P. Fezzardi, M. Fiorito, M. Lattuada, M. Minutoli, C. Pilato, and A. Tumeo, "Invited: Bambu: An Open-Source Research Framework for the High-Level Synthesis of Complex Applications," in *Proceedings of the 58th Annual ACM/IEEE Design Automation Conference*. IEEE Press, 2022, pp. 1327–1330. [Online]. Available: https://doi.org/10.1109/DAC18074.2021.9586110

[15] J. D. Gaeddert *et al.*, "liquid-dsp." [Online]. Available: https://github.com/jgaeddert/liquid-dsp

[16] 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.

[17] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach, "Reusability is firrtl ground: hardware construction languages, compiler frameworks, and transformations," in *Proceedings of the 36th International Conference on Computer-Aided Design*, ser. ICCAD '17. IEEE Press, 2017, p. 209–216.

[18] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, ser. CGO '04. USA: IEEE Computer Society, 2004, p. 75.

[19] S. Ma, H. Wang, S. Huang, X. Zhang, Y. Hu, T. Song, Y. Xia, and F. Wei, "Bitnet b1.58 2b4t technical report," 2025. [Online]. Available: https://arxiv.org/abs/2504.12285

[20] D. Nuzman, I. Rosen, and A. Zaks, "Auto-vectorization of interleaved data for simd," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 132–143. [Online]. Available: https://doi.org/10.1145/1133981.1133997

[21] 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*. ACM, 2024, pp. 591–606. [Online]. Available: https://dl.acm.org/doi/10.1145/3620666.3651375

[22] W. Peng, Y. Xiao, Y. Zou, Z. Luo, and Y. Liang, "Clay: High-level asip framework for flexible microarchitecture-aware instruction customization," in *Proceedings of the 44st IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '25, 2025.

[23] P. platform, "Pulp platform: Open hardware, the way it should be!" 2025. [Online]. Available: https://pulp-platform.org/index.html

[24] I. Rosen, D. Nuzman, and A. Zaks, "Loop-aware slp in gcc," pp. 131–142, 01 2007.

[25] P. Sauter, T. Benz, P. Scheffler, H. Pochert, L. Wüthrich, M. Povišer, B. Muheim, F. K. Gürkaynak, and L. Benini, "Croc: An end-to-end open-source extensible risc-v mcu platform to democratize silicon," 2025. [Online]. Available: https://arxiv.org/abs/2502.05090

[26] P. D. Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand, and L. Benini, "Slow and steady wins the race? a comparison of ultra-low-power risc-v cores for internet-of-things applications," in *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. IEEE, 2017, pp. 1–8.

[27] W. Snyder, P. Wasson, D. Galbi, and et al, "Verilator." [Online]. Available: https://github.com/verilator/verilator

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

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

[30] 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.

[31] 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*, ser. FPGA '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 211–222. [Online]. Available: https://doi.org/10.1145/3626202.3637561

[32] 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. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3508352.3549370

[33] 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*, vol. 27, no. 11, pp. 2629–2640, Nov 2019.

[34] 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, pp. 211–218, iSSN: 1558-2434. [Online]. Available: https://ieeexplore.ieee.org/document/6691121

[35] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "Sonicboom: The 3rd generation berkeley out-of-order machine," May 2020.