ENSEIRB-MATMECA

Électronique – Systèmes Embarqués

Accelerating an FFT on the CVA6 RISC-V processor

LumiEirb

Étudiants :

Leila FOUNTI
Meryem ELOUASSAI
FatimaZahra EL MEADY
Anass EL KABBAJ

Encadrants :

Camille LEROUX
Mathieux ESCOUTELOUP

27 janvier 2026

Table des matières

Introduction

The growing complexity of embedded and high-performance systems has increased the need for flexible and open processor architectures. In this context, the RISC-V instruction set architecture (ISA) has emerged as a strong alternative to proprietary solutions, enabling both academic research and industrial innovation through its open and extensible design.

Open-source RISC-V processors provide a unique opportunity to study realistic microarchitectures while maintaining compatibility with modern software environments. They allow designers and researchers to explore architectural trade-offs, performance bottlenecks, and hardware–software interactions on platforms that closely resemble industrial systems.

This project is part of an academic effort to explore such architectural optimizations using a real application workload. By focusing on the acceleration of a Fast Fourier Transform (FFT), a widely used algorithm in signal processing and scientific computing, the work illustrates how application-driven analysis can guide meaningful processor-level enhancements.

1 Context

This project is conducted within the framework of the *6th National RISC-V Student Contest (2025–2026)*, organized by Thales in collaboration with the GDR SOC^2 and the CNFM. The contest aims to introduce students to realistic processor architecture design through hands-on experimentation on an industrial-grade open-source RISC-V core.

The target processor of the contest is the **CV32A6**, a 32-bit application-class RISC-V core derived from the CVA6 architecture developed by the OpenHW Group. This processor is widely used in academic and research environments due to its open-source nature, realistic microarchitecture, and support for advanced features such as virtual memory and multiple privilege levels.

The challenge proposed in this edition of the contest is application-oriented and focuses on the execution of a Fast Fourier Transform (FFT) workload written in C. Rather than designing a new algorithm, participants are encouraged to study how a realistic application interacts with the processor microarchitecture, including pipeline behavior, memory accesses, and instruction execution flow.

To support this exploration, the organizers provide a complete development environment, including the CV32A6 RTL design, simulation and synthesis scripts, a Board Support Package (BSP), and a reference FPGA implementation on the Digilent Zybo Z7-20 board. This environment enables both detailed simulation and real-hardware validation.

2 Specifications

2.1 Hardware Platform

The hardware platform used in this project is based on the **CV32A6** processor implemented on a Digilent Zybo Z7-20 FPGA development board. The CV32A6 is a 32-bit, in-order, single-issue RISC-V core featuring a six-stage pipeline and supporting the RV32IM_Zicsr instruction set.

The processor integrates the following execution units :

— an Arithmetic Logic Unit (ALU) for integer computations ;
— a multiplier and divider unit, pipelined over multiple cycles ;
— a Load–Store Unit with separate instruction and data caches, each of size 16 KiB, 4-way associative, with a 64-byte cache line.

The core supports virtual memory through a Memory Management Unit (MMU), Translation Lookaside Buffers (TLBs), and three privilege levels : User, Supervisor, and Machine.

The FPGA implementation targets the Digilent Zybo Z7-20 board, which provides the necessary infrastructure for deployment and debugging. The system uses :

— a JTAG interface for FPGA programming and hardware debugging ;
— a UART interface, implemented through a Pmod USB-UART module, for console output and interaction ;
— Pmod connectors for modular peripheral integration.

The typical operating frequency of the processor on the Zybo Z7-20 board ranges between 100 and 150 MHz, depending on synthesis and timing constraints.

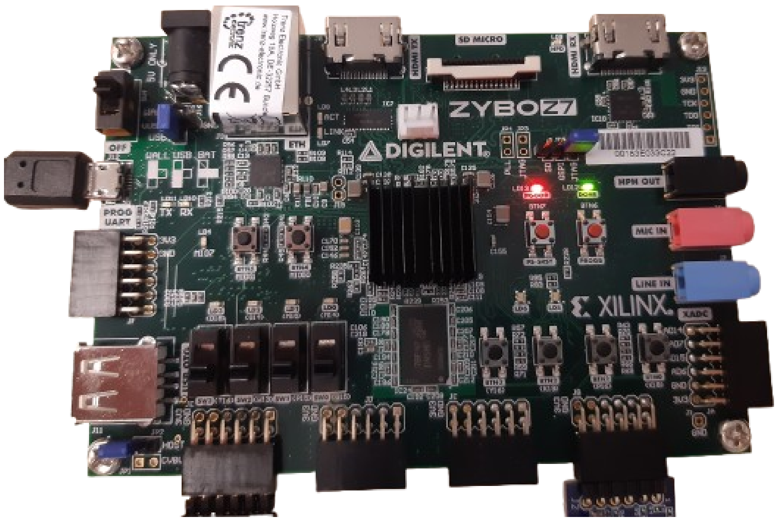

Figure 1 – Digilent Zybo Z7-20 FPGA development board used for CV32A6 deployment.

Figure 2 – JTAG Interface for programming and debugging

Figure 3 – Pmod USB-UART for console output

2.2 Project Objectives and Constraints

The objective of this project is to reduce the execution time of a given **2^n-point FFT** application running on the CV32A6 processor by introducing architectural optimizations.

This objective is illustrated in Figure 4, which highlights the importance of minimizing the number of execution cycles for the FFT workload.

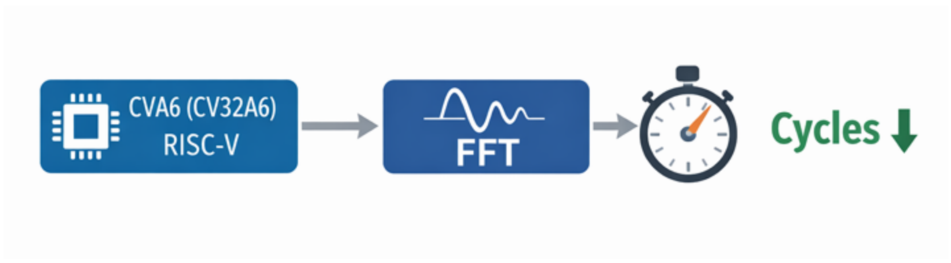

Figure 4 – Illustration of the project goal : reducing the number of cycles to execute the FFT on CV32A6.

The proposed solution must satisfy the following constraints :

— the sizes of the instruction and data caches must remain unchanged ;

— the operational frequency of the processor must not decrease by more than 20% ;

— compatibility with the RV32IM_Zicsr instruction set must be preserved, although custom extensions are allowed ;

— acceleration must rely on modifications to the CV32A6 core or on a tightly-coupled coprocessor ;

— the FFT output must match the provided reference results on a per-bit basis.

Architectural enhancements may include custom instructions, microarchitectural modifications, or coprocessor integration, provided that all contest requirements are respected.

3 Fast Fourier Transform (FFT)

The Fourier transform is a fundamental tool in signal processing, as it provides a mathematical framework to analyze signals in the frequency domain. Many physical signals, although observed in the time domain, are more naturally described by their spectral content. Applications such as communications, audio and image processing, radar systems, and embedded sensing rely on frequency-domain analysis to identify dominant frequencies, filter noise, or extract meaningful features.

In practical digital systems, signals are represented by a finite number of samples, which motivates the use of the Discrete Fourier Transform (DFT). However, the direct computation of the DFT quickly becomes computationally expensive as the signal size increases. The Fast Fourier Transform (FFT) addresses this limitation by significantly reducing the number of required operations, making real-time spectral analysis feasible even on constrained hardware platforms.

3.1 Principle of the Discrete and Fast Fourier Transform

The Discrete Fourier Transform (DFT) converts a finite sequence of time-domain samples into its frequency-domain representation. For a signal composed of N complex samples $(x_0, x_1, \ldots, x_{N-1})$, the DFT is defined as :

$$X_k = \sum_{n=0}^{N-1} x_n \, e^{-j\frac{2\pi}{N}kn}, \quad k = 0, \ldots, N-1 \tag{1}$$

Each coefficient X_k represents the amplitude and phase of the frequency component $\frac{k}{N}$ present in the signal. The direct computation of the DFT requires $\mathcal{O}(N^2)$ complex operations, which becomes costly for large values of N.

The Fast Fourier Transform (FFT) is an optimized algorithm that reduces this complexity to $\mathcal{O}(N \log_2 N)$ by exploiting symmetries in the complex exponential terms. The most common approach relies on dividing the input signal into smaller sub-signals and recursively combining their transforms.

3.2 Decimation and Butterfly Structure

The Fast Fourier Transform used in this project follows the Cooley–Tukey algorithm, which reduces the computational complexity of the Discrete Fourier Transform by recursively decomposing it into smaller

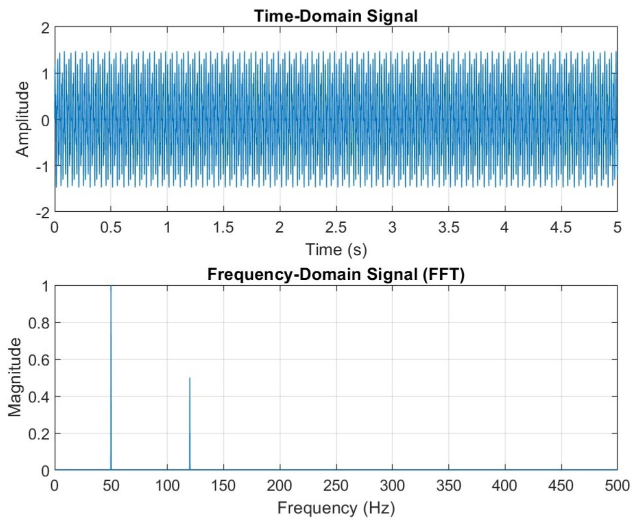

Figure 5 – Transformation from time domain to frequency domain using the FFT

transforms. In the radix-based formulation, the input signal is separated into even-indexed and odd-indexed samples. Two smaller DFTs are then computed and recombined using complex multiplications by precomputed coefficients, commonly referred to as *twiddle factors* :

$$W_N^k = e^{-j\frac{2\pi}{N}k}$$

The recombination step is performed through a basic computational pattern known as the *butterfly*. In the radix-2 case, this operation is expressed as :

$$X_k = E_k + W_N^k \cdot O_k \tag{2}$$
$$X_{k+N/2} = E_k - W_N^k \cdot O_k \tag{3}$$

where E_k and O_k denote the FFT results of the even and odd subsequences, respectively.

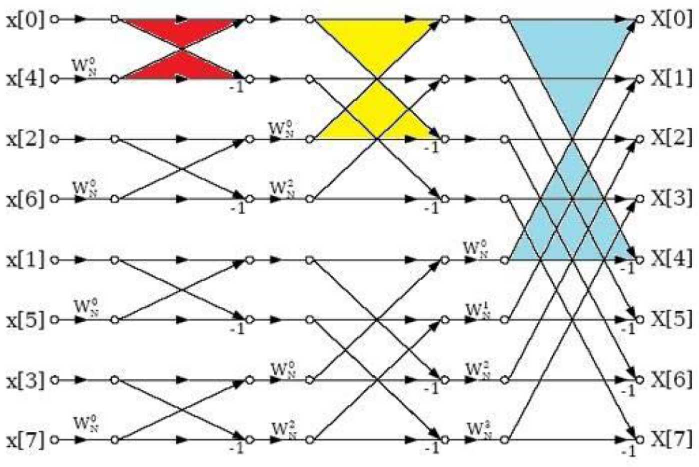

Figure 6 – Butterfly structure for an 8-point FFT

This butterfly structure is applied iteratively across $\log_2(N)$ stages, enabling the complete FFT to be computed with a total complexity of $\mathcal{O}(N \log_2 N)$.

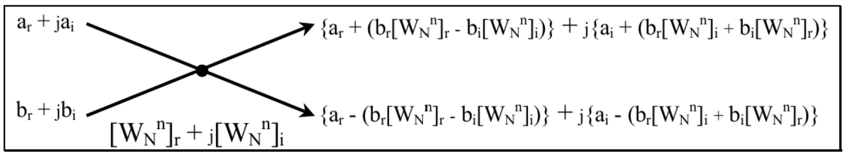
$a_r + ja_i$
$b_r + jb_i$
$[W_N^n]_r + j[W_N^n]_i$
$\{a_r + (b_r[W_N^n]_r - b_i[W_N^n]_i)\} + j\{a_i + (b_r[W_N^n]_i + b_i[W_N^n]_r)\}$
$\{a_r - (b_r[W_N^n]_r - b_i[W_N^n]_i)\} + j\{a_i - (b_r[W_N^n]_i + b_i[W_N^n]_r)\}$

Figure 7 – Elementary radix-2 butterfly

3.3 FFT implementation used in the project

The FFT implementation used in this project is based on the KISS FFT library provided by Thales. The algorithm operates on a fixed FFT size of $N = 512$ and follows a mixed-radix decomposition that relies mainly on radix-4 butterfly operations, with a final radix-2 stage.

The computation is organized as a recursive process. At each recursion level, the input signal is split into smaller sub-sequences whose sizes are defined by a predefined factorization of N. Once the smallest transforms are reached, the partial results are recombined using butterfly operations. In this implementation, only radix-4 and radix-2 butterflies are effectively used.

All twiddle factors required by the FFT are precomputed and stored in memory. They are represented using fixed-point arithmetic, where real and imaginary parts are encoded as signed integers with a fixed number of fractional bits. This choice avoids floating-point operations and is well suited for embedded and hardware-oriented environments.

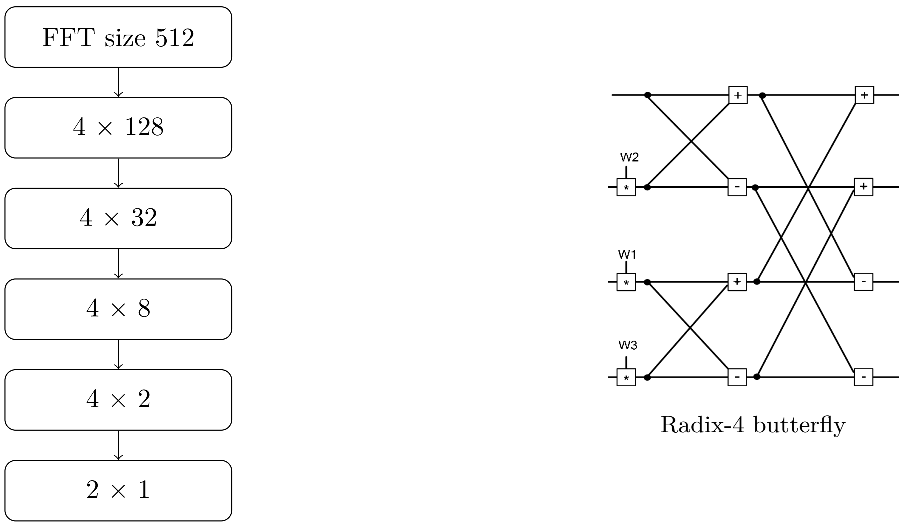

Figure 8 – Recursive decomposition of a 512-point FFT using radix-4 and radix-2 stages

The overall structure of the algorithm combines recursion, precomputed twiddle factors, and butterfly-based recombination to efficiently compute the FFT of a 512-point complex signal.

4 CVA6 Processor Architecture

4.1 Design Rationale and Role in the System

CVA6 is an open-source application-class RISC-V processor core designed to provide a realistic yet accessible microarchitecture. Unlike minimal embedded cores, CVA6 incorporates features such as a multi-stage pipeline, cache hierarchy, and optional virtual memory support, while remaining simpler than out-of-order processors. This balance makes it particularly suitable for academic projects that aim to study performance behavior and hardware–software interactions.

In this project, CVA6 serves as the reference execution platform for evaluating architectural optimizations applied to a compute-intensive workload. The selected configuration is the 32-bit variant, CV32A6, which provides sufficient computational capabilities while limiting architectural complexity. This choice allows a focused analysis of instruction-level behavior, memory accesses, and custom hardware extensions without introducing additional constraints related to 64-bit addressing or large data paths.

4.2 Instruction Set and Software Interaction

The CV32A6 core implements the RV32IM instruction set along with the Zicsr extension. This configuration ensures compatibility with standard RISC-V toolchains while providing hardware support for multiplication and division, which are heavily used in numerical algorithms such as the FFT.

CSR support plays a central role in system-level execution. It enables precise exception handling, privilege management, and access to performance counters. From a software perspective, this allows the execution of low-level runtime code and facilitates fine-grained performance analysis, which is essential when evaluating the impact of architectural optimizations.

4.3 Pipeline Structure and Execution Model

CVA6 relies on a six-stage, in-order pipeline composed of program counter generation, instruction fetch, decode, issue, execute, and commit stages. This organization allows a higher operating frequency than shorter pipelines, while preserving a deterministic execution model.

The in-order nature of the pipeline implies that execution speed is directly affected by instruction count, data dependencies, and control-flow complexity. As a consequence, workloads involving frequent function calls, branching, or sequences of dependent arithmetic operations may experience performance limitations. These characteristics are particularly relevant for FFT computations, which involve deeply nested loops and repetitive arithmetic patterns.

The explicit commit stage ensures precise exceptions by delaying architectural state updates until instruction completion is guaranteed. This design choice simplifies exception handling and enables safe integration of custom execution units through interfaces such as CVXIF.

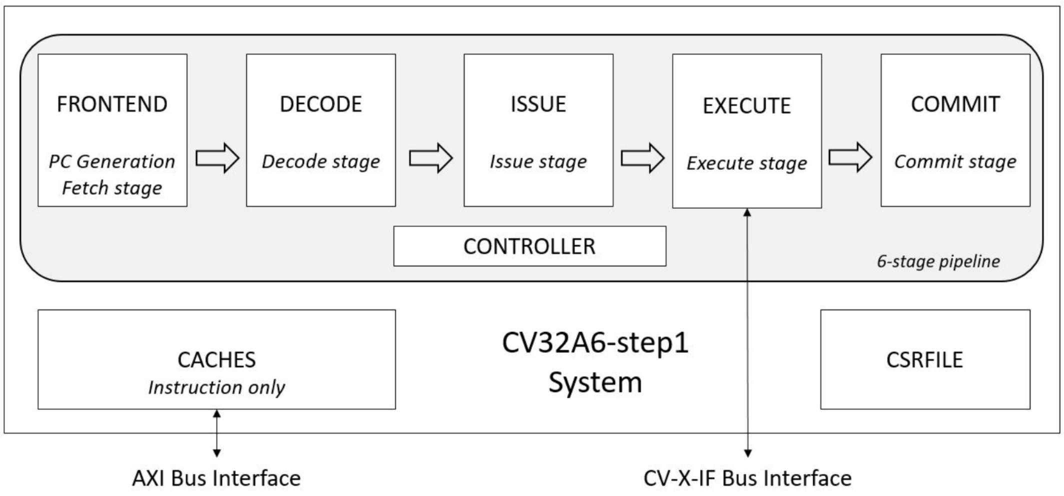

Figure 9 – Instruction progression through the six-stage CVA6 pipeline

4.4 Execution Units and Arithmetic Behavior

The execution stage integrates several functional units, including an arithmetic logic unit, a multiplier/divider, and a load–store unit. While the presence of a hardware multiplier reduces the cost of scalar multiplications, complex arithmetic operations still require multiple dependent instructions, increasing execution latency and register pressure.

The load–store unit handles all memory accesses and enforces ordering constraints with respect to the cache hierarchy. In an in-order pipeline, memory access latency directly impacts instruction throughput, making efficient data representation and reduced memory traffic important considerations for performance optimization.

4.5 Memory Hierarchy and Translation

CVA6 implements separate level-1 instruction and data caches, allowing instruction fetches and data accesses to proceed in parallel. This Harvard-style organization reduces structural hazards and improves average throughput. Cache parameters such as size and associativity are configurable, enabling architectural trade-offs between performance and resource usage.

An optional SV32 memory management unit provides virtual-to-physical address translation using page-based mapping. While virtual memory support is not directly required for FFT execution, it contributes to a realistic execution environment and influences memory access behavior through address translation and TLB activity.

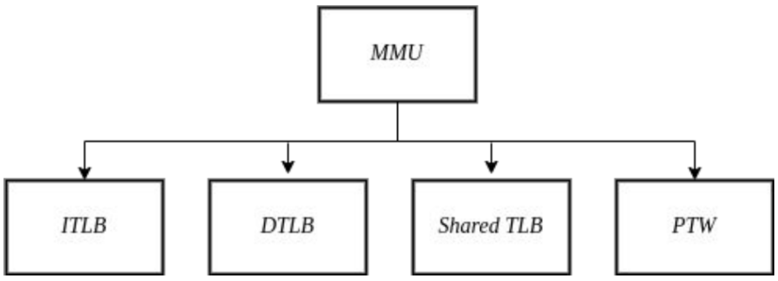

Figure 10 – CVA6 memory subsystem

4.6 Memory Protection and Architectural Constraints

Physical Memory Protection (PMP) enables hardware-enforced access control to memory regions. Although PMP is not directly involved in performance optimization, it imposes architectural constraints that must be respected when introducing custom extensions or coprocessors. Its presence highlights the need to maintain compatibility with the core's protection mechanisms during system modifications.

4.7 Architectural Implications for Optimization

Overall, the CVA6 architecture exhibits characteristics that strongly influence optimization strategies. The in-order pipeline emphasizes the importance of reducing instruction count and minimizing control-flow overhead. The cost of complex arithmetic motivates the use of custom instructions to collapse frequently executed computation patterns.

5 CVXIF : Custom Extension Interface

5.1 Motivation and Design Principles

CVXIF (CORE-V eXtension Interface) is a standardized interface designed to extend a RISC-V processor with custom hardware functionality without modifying the core pipeline. In traditional processor extensions, adding new instructions often requires changes to the decoder, execution stages, and sometimes even the register file, which increases design complexity and validation effort. CVXIF addresses this issue by providing a clean separation between the processor core and external coprocessors.

In the context of this project, CVXIF plays a central role by enabling application-specific acceleration while preserving the integrity of the CVA6 microarchitecture. Rather than modifying the core execution pipeline, computationally intensive operations are offloaded to a dedicated coprocessor through a well-defined interface. This approach allows architectural experimentation while maintaining compatibility with the original processor design.

5.2 General Operating Principle

From the processor's perspective, a CVXIF instruction behaves like a regular arithmetic instruction. When the CVA6 decoder encounters an instruction that does not belong to its native instruction set but uses a reserved custom opcode, the instruction is forwarded to the CVXIF interface instead of being executed internally.

The processor remains in control of instruction scheduling and retirement. The coprocessor only executes the operation itself and returns the result when requested. This division of responsibilities ensures that speculation, exception handling, and architectural state updates remain managed by the core, while the coprocessor focuses exclusively on computation.

5.3 Integration within the CVA6 Pipeline

CVXIF is tightly integrated into the CVA6 pipeline while remaining logically external to the core. The interaction follows a structured sequence that mirrors the lifetime of an instruction in the pipeline.

During the issue stage, the processor checks whether the coprocessor is able to accept the instruction. If accepted, the source operands are transmitted directly from the register file to the coprocessor. The instruction then proceeds through the pipeline like a standard instruction, without stalling unrelated operations.

At commit time, the processor confirms that the instruction is no longer speculative. Only after this confirmation is the coprocessor allowed to finalize the operation and make the result visible. This mechanism guarantees precise exceptions and preserves the architectural correctness of the system.

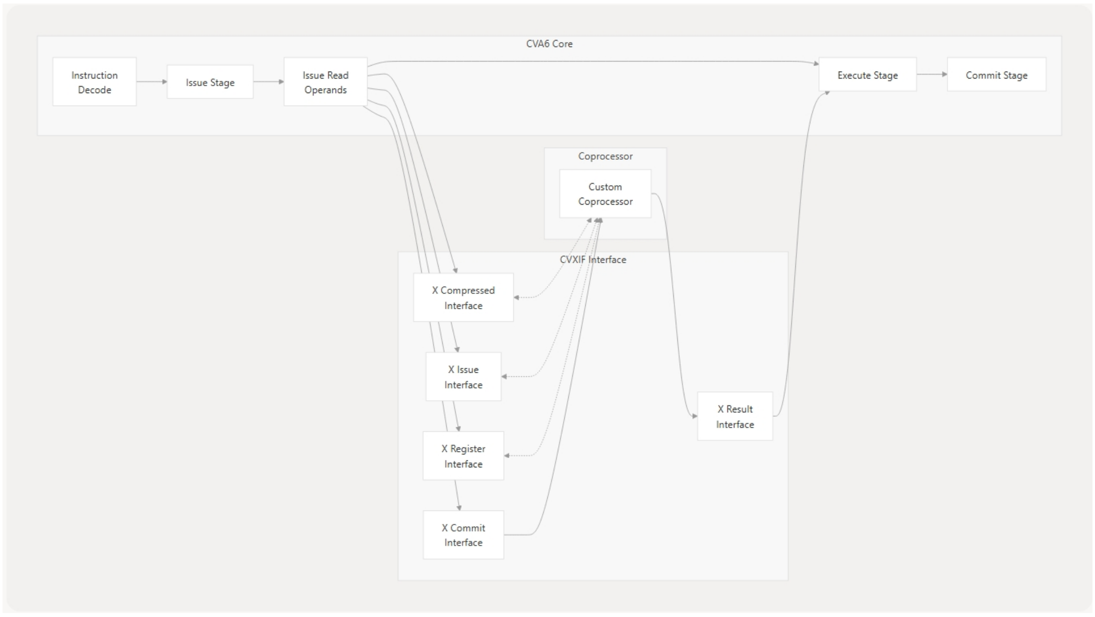

Figure 11 – Integration of CVXIF instructions within the CVA6 pipeline

5.4 CVXIF Interfaces and Data Flow

The CVXIF mechanism relies on a small number of well-defined interfaces that structure communication between the core and the coprocessor. These interfaces cover instruction issue, operand transfer, instruction validation, and result write-back.

Source operands are read directly from the CVA6 register file and forwarded to the coprocessor, avoiding unnecessary memory accesses. Once the coprocessor has completed execution, the result is sent back to the core and written into the destination register as if it were produced by a native instruction.

Importantly, CVXIF does not allow the coprocessor to alter control flow or access memory directly in the CVA6 implementation. This restriction simplifies verification and ensures that program execution remains fully controlled by the processor core.

5.5 Software Transparency and Custom Instructions

One of the key advantages of CVXIF is its transparency from a software perspective. Custom instructions can be inserted into C code using inline assembly without modifying the compiler, assembler, or toolchain. The processor automatically redirects unsupported instructions to the coprocessor through the CVXIF interface.

This approach allows rapid prototyping of custom instructions and simplifies validation. Software remains largely portable and readable, while performance-critical kernels can selectively exploit hardware acceleration when available.

5.6 Relevance for FFT Acceleration

The FFT workload used in this project is dominated by repetitive complex arithmetic operations and butterfly computations. When executed purely in software, these operations translate into long instruction sequences with multiple data dependencies, which are inefficient on an in-order pipeline such as CVA6.

CVXIF provides a natural solution to this limitation by allowing these recurring computation patterns to be collapsed into single custom instructions executed in hardware. By offloading complex arithmetic and butterfly operations to a coprocessor, the number of executed instructions is reduced, data movement is simplified, and overall execution speed is improved without altering the processor core.

5.7 Summary

CVXIF enables a flexible and low-intrusion approach to extending the CVA6 processor with application-specific hardware acceleration. By decoupling computation from control and preserving the core pipeline structure, it provides an effective framework for hardware–software co-design. In this project, CVXIF serves as the key mechanism that enables targeted acceleration of FFT operations while maintaining architectural correctness and software compatibility.

6 Development and Validation Workflow

To ensure a structured, reproducible, and reliable development process, a well-defined workflow was established and consistently followed throughout all experiments and tests conducted in this project. This workflow was applied systematically whenever new features, fixes, or optimizations were introduced. The adopted workflow is summarized as follows :

— **SystemVerilog Design Analysis (Vivado)** : Analysis of the RTL architecture using Vivado, including the CVA6 core, the CVXIF interface, and the coprocessor integration. This step focuses on understanding signal interactions, module dependencies, and architectural constraints.

- **RTL Compilation, Simulation, and Debug (QuestaSim)** : RTL sources are compiled to verify syntactic correctness and interface consistency, followed by execution of RTL simulations using QuestaSim to validate the baseline design. Simulation waveforms and logs are analyzed to observe system behavior and ensure correct communication between the processor core and the coprocessor through the CVXIF interface. When functional or integration issues are identified, targeted fixes or patches are applied, followed by recompilation and resimulation until a stable and validated baseline is achieved.

- **KISS FFT C Code Adaptation** : To prepare data formats compatible with custom coprocessor instructions. This step aims to reduce interface overhead and enable efficient interaction between the software and hardware components.

- **FPGA Implementation and Hardware Validation** : Synthesis and implementation of the design on the Zybo FPGA board, followed by hardware validation and result analysis to confirm functional correctness and evaluate the impact of the proposed optimizations.

7 Initial Code Analysis

At the beginning of the challenge, we followed the development and validation workflow described in the previous section in order to establish a reliable and well-understood baseline. The objective of this initial phase was to analyze the provided sources, validate the reference implementation, and identify potential optimization opportunities before introducing any modifications.

First, the codebase and reference sources available on the official GitHub repositories were analyzed. This included the study of key software and hardware components such as the `kiss_fft.c` file for the FFT implementation, the coprocessor RTL modules (e.g., `copro_alu.sv`), and the CVXIF-related definitions (notably `cvxif_instr_pkg.sv`). This analysis allowed us to understand the role of each component, their interactions, and the overall data flow.

Following this code analysis, initial compilations were performed to generate the RTL of the CVA6-based system and to verify that the baseline design could be correctly elaborated. The CVXIF interface was then enabled to allow communication with the provided coprocessor.

RTL simulations were subsequently carried out using **QuestaSim**. Due to the complexity of the design and the large number of available signals, the analysis focused primarily on the ALU and coprocessor-related signals, as well as on the execution timeline of base instructions. This instruction-level observation made it possible to identify performance-critical operations and provided a reference behavior used to guide the subsequent optimization strategy.

8 Proposed and Implemented Optimizations

Based on the baseline and instruction-level analysis, we identified several optimization directions, including the acceleration of elementary FFT operations, the reduction of CPU–coprocessor interface overhead, and improvements in data representation, with the objective of reducing execution latency while maintaining compatibility with the CVA6 architecture.

8.1 Integration of Custom CVXIF Instructions

Custom instructions are introduced through the **CVXIF (Custom eXtension Interface)** to offload computation-intensive tasks from the processor core to a dedicated coprocessor. This mechanism enables extensions to the RISC-V instruction set without modifying the core pipeline, while preserving compatibility with the existing architecture. Through CVXIF, custom instructions are issued by the processor and executed by the coprocessor, with proper handling of synchronization, register access, and write-back. This approach provides a flexible and scalable solution for hardware acceleration of application-specific operations.

8.2 Acceleration of Elementary FFT Operations

FFT computations involve a large number of elementary complex arithmetic operations, which are sequentially executed by the processor in the baseline implementation, resulting in high instruction counts and increased execution time.

To reduce this overhead, elementary complex operations are targeted for hardware acceleration using custom instructions :

— complex addition (CADD),
— complex subtraction (CSUB),
— complex multiplication (CMUL).

Executing these operations directly in hardware allows a full complex computation to be performed using a single instruction, significantly reducing the number of processor instructions per FFT stage.

8.3 Use of Inline Assembly for Custom Instructions

Inline assembly using the `.insn` directive is used to invoke custom CVXIF instructions from C code without modifying the compiler toolchain. This approach enables direct insertion of custom instruction encodings while preserving compatibility with standard RISC-V compilation flows, allowing rapid prototyping and validation of custom instructions.

8.4 Packed Representation of Complex Numbers

During the initial phase, complex arithmetic operations relied on a separate representation for real and imaginary parts, which simplified validation but introduced additional interface overhead. To address this limitation, a packed data representation was introduced, where a complex number is stored in a single 32-bit register containing two signed 16-bit fixed-point components. This representation reduces load/store operations and simplifies operand passing to custom complex instructions.

8.5 Summary and Outlook

By combining custom CVXIF instructions, acceleration of elementary FFT operations, inline assembly integration, and a packed complex data representation, an efficient hardware–software co-design approach is established. These enhancements provide a solid foundation for further acceleration of FFT processing and the introduction of more details of the custom instructions in subsequent stages of the project.

9 Accomplished work

9.1 Baseline Performance

As a reference point for performance evaluation, we analyzed first the default architecture configuration without any custom instructions or hardware acceleration. In this baseline setup, all computations are executed using the standard CVA6 pipeline and the existing software implementation.

Under these conditions, the execution of the target workload required approximately **169 000 cycles**. This result serves as the baseline performance metric against which subsequent optimizations and custom instruction integrations are evaluated.

9.2 Complex Multiplication Instruction

As a first optimization step, a custom instruction for complex multiplication was considered to accelerate one of the most frequently executed operations in FFT computations and to evaluate the impact of instruction-level acceleration.

To accelerate this operation, a custom CVXIF instruction `CMUL_I16` was introduced. It performs a full complex multiplication in hardware on packed Q15 fixed-point complex numbers, with the real and imaginary parts stored in a single 32-bit register.

An excerpt of the hardware implementation is shown in Listing 1, which illustrates the unpacking of operands, Q30 intermediate multiplication, rounding, and packing of the final Q15 result.

```
cvxif_instr_pkg::CMUL_I16: begin
  // Unpack operands: rs1 -> {ai, ar}, rs2 -> {bi, br}
  ar = registers_i[1][15:0];   // real part of rs1
  ai = registers_i[1][31:16];  // imag part of rs1
  br = registers_i[0][15:0];   // real part of rs2
  bi = registers_i[0][31:16];  // imag part of rs2

  // Perform Q30 multiplication
  real_q30 = ar * br - ai * bi;
  imag_q30 = ar * bi + ai * br;

  // Convert to Q15 with rounding
  real_q15 = (real_q30 + (1 <<< 14)) >>> 15;
  imag_q15 = (imag_q30 + (1 <<< 14)) >>> 15;

  // Pack result
  result_n = {imag_q15, real_q15};
  valid_n  = 1'b1;
  rd_n     = rd_i;
  we_n     = 1'b1;
end
```

Listing 1 – CVXIF complex multiplication instruction (CMUL_I16)

From a functional standpoint, this instruction replaces multiple integer operations and register moves with a single hardware execution. The FFT workload executes in approximately **152 000 cycles**, which represents a reduction of about **10%** compared to the baseline of 169 000 cycles.

9.3 Iterative Code Structure

The original KISS FFT implementation relies on a recursive function, `kf_work`, to traverse the FFT factorization tree and apply radix-specific butterfly operations. While this formulation is algorithmically clear, it introduces non-negligible overhead on the CVA6 processor due to repeated function calls, stack manipulation, and limited visibility at the instruction level.

In order to gain finer control over the execution flow and to better isolate performance-critical operations, the recursive structure was replaced by an explicit iterative formulation. The recursion is emulated using a manually managed software stack, where each stack frame represents the state of a former recursive call (output pointer, input pointer, strides, radix, and stage index).

Listing 2 shows an excerpt of the original recursive implementation. The key characteristics are the recursive descent into smaller FFTs and the final recombination using radix-dependent butterfly functions.

```
if (m == 1) {
    do {
        *Fout = *f;
        f += fstride * in_stride;
    } while (++Fout != Fout_end);
} else {
    do {
        kf_work(Fout, f, fstride * p, in_stride, factors, st);
```

```c
        f += fstride * in_stride;
    } while ((Fout += m) != Fout_end);
}

switch (p) {
    case 2: kf_bfly2(Fout, fstride, st, m); break;
    case 3: kf_bfly3(Fout, fstride, st, m); break;
    case 4: kf_bfly4(Fout, fstride, st, m); break;
    case 5: kf_bfly5(Fout, fstride, st, m); break;
    default: kf_bfly_generic(Fout, fstride, st, m, p); break;
}
```

Listing 2 – Excerpt of the recursive `kf_work` function

The iterative version replaces this recursion with an explicit loop and a stack of frames. Each iteration corresponds either to the descent into a sub-FFT or to the recombination step once all child FFTs have been processed. The structure of the computation is preserved, but the control flow becomes fully explicit.

```c
while (sp > 0) {
    kf_frame_t *fr = &stack[sp - 1];

    if (fr->stage == 0) {
        if (fr->m == 1) {
            /* terminal copy */
            ...
            fr->stage = 1;
            continue;
        }

        if (fr->idx < fr->p) {
            /* push sub-FFT */
            ...
            continue;
        }
        fr->stage = 1;
    }

    /* butterfly recombination */
    switch (fr->p) {
        case 2: kf_bfly2(fr->Fout, fr->fstride, st, fr->m); break;
        ...
    }
    sp--;
}
```

Listing 3 – Excerpt of the iterative FFT implementation

From a performance standpoint, this transformation resulted in a slowdown compared to the recursive baseline. When combined with the previously introduced complex multiplication custom instruction, the iterative version requires approximately **174 000 cycles** to execute the FFT workload. This corresponds to an increase of about **8.8%** compared to the baseline execution without any architectural modification.

This degradation is mainly due to the additional loop control, stack management, and conditional branching introduced by the iterative structure, which increase the instruction count and stress the in-order CVA6 pipeline.

Nevertheless, this cost was accepted as a deliberate trade-off. The iterative structure exposes the FFT execution phases more clearly and makes butterfly operations and complex arithmetic kernels explicit and easily identifiable. This significantly simplified instruction-level analysis and enabled precise targeting of additional operations for hardware acceleration through custom CVXIF instructions.

9.4 Complex Addition/Subtraction Instruction

In FFT computations, complex additions and subtractions are ubiquitous, particularly in butterfly recombination stages. In the baseline implementation, each complex addition or subtraction is decomposed into multiple scalar integer operations, increasing instruction count and register pressure.

To reduce this overhead, two custom CVXIF instructions were introduced to perform complex addition and subtraction directly in hardware, operating on packed fixed-point complex numbers. Each complex number is represented in a single 32-bit register, with the real and imaginary parts stored as signed 16-bit values in Q15 format.

The custom instructions `CADD_I16` and `CSUB_I16` unpack the operands, perform the arithmetic on both components in parallel, and repack the result into a single register. An excerpt of the hardware implementation is shown in Listing 4.

```
cvxif_instr_pkg::CADD_I16: begin
  // Unpack: {imag, real}
  xr = registers_i[0][15:0];
  xi = registers_i[0][31:16];
  wr = registers_i[1][15:0];
  wi = registers_i[1][31:16];

  real_q15 = xr + wr;
  imag_q15 = xi + wi;

  result_n = {imag_q15, real_q15};
  valid_n  = 1'b1;
  we_n     = 1'b1;
end

cvxif_instr_pkg::CSUB_I16: begin
  // Unpack: {imag, real}
  xr = registers_i[0][15:0];
  xi = registers_i[0][31:16];
  wr = registers_i[1][15:0];
  wi = registers_i[1][31:16];

  real_q15 = xr - wr;
  imag_q15 = xi - wi;

  result_n = {imag_q15, real_q15};
  valid_n  = 1'b1;
  we_n     = 1'b1;
end
```

Listing 4 – Excerpt of CVXIF complex addition and subtraction instructions

From a functional perspective, these instructions directly replace multiple integer additions, subtractions, and register moves with a single operation. From an architectural standpoint, they reduce instruction count and simplify the interaction between software and the coprocessor by relying on a compact data representation.

In terms of performance, the integration of complex addition and subtraction custom instructions results in a modest improvement. The FFT workload executes in approximately **164 000 cycles**, corresponding to **2.9% fewer cycles than the baseline** configuration.

9.5 Radix-2 Butterfly Instruction

While complex addition and subtraction reduce arithmetic overhead, the dominant computation in an FFT remains the radix-2 butterfly, which combines a complex multiplication with multiple complex additions and subtractions. In the baseline implementation, this operation is decomposed into several scalar instructions, resulting in a high instruction count and repeated data movement.

To address this, a custom CVXIF radix-2 butterfly instruction, `BFLY2_I16`, was introduced. This instruction operates on two packed complex inputs and a complex twiddle factor, all represented in Q15 fixed-point format. The butterfly computes the intermediate product between the second input and the twiddle factor, followed by the sum and difference with the first input to produce the two butterfly outputs.

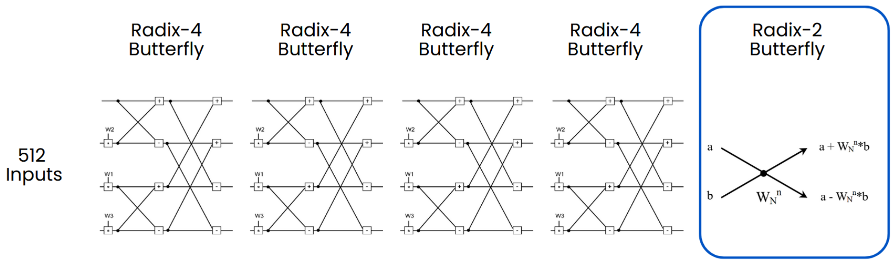

Figure 12 – Mixed-radix decomposition of a 512-point FFT.

In the targeted FFT configuration, the transform operates on 512 input samples. In the software implementation, this corresponds to a mixed-radix decomposition consisting of four radix-4 stages followed by a single radix-2 stage, since $512 = 4 \times 4 \times 4 \times 4 \times 2$. Figure 12 illustrates this decomposition. The radix-2 stage, highlighted in the figure, represents the final recombination step and is the focus of the hardware optimization presented in this work.

Due to the CVXIF interface returning a single register result, only one of the two butterfly outputs is written back directly. The second output is stored internally and retrieved using a dedicated instruction, `GETY1_I16`. This split preserves interface compatibility while still allowing both results to be efficiently produced by the hardware. An excerpt of the implementation is shown in Listing 5.

```
cvxif_instr_pkg::BFLY2_I16: begin
  // unpack x0 (rs1)
  x0r = registers_i[0][15:0];
  x0i = registers_i[0][31:16];

  // unpack x1 (rs2)
  x1r = registers_i[1][15:0];
  x1i = registers_i[1][31:16];

  // unpack twiddle factor (shadow register)
  twr = shadow_tw_q[15:0];
  twi = shadow_tw_q[31:16];

  // complex multiplication: t = x1 * tw
  p1 = x1r * twr;
  p2 = x1i * twi;
  p3 = x1r * twi;
  p4 = x1i * twr;

  tr = (p1 - p2) >>> 15;
  ti = (p3 + p4) >>> 15;

  // butterfly outputs
  y0r = x0r + tr;  y0i = x0i + ti;
  y1r = x0r - tr;  y1i = x0i - ti;

  result_n     = {y0i, y0r};
  shadow_y1_n = {y1i, y1r};

  valid_n = 1'b1;
  we_n    = 1'b1;
  rd_n    = rd_i;
```

```
end

cvxif_instr_pkg::GETY1_I16: begin
  result_n = shadow_y1_q;

  valid_n  = 1'b1;
  we_n     = 1'b1;
  rd_n     = rd_i;
end
```

Listing 5 – Excerpt of CVXIF radix-2 butterfly instruction

By collapsing an entire radix-2 butterfly into a small number of custom instructions, this approach significantly reduces instruction count and register traffic compared to the baseline software implementation. Although the CVXIF interface necessitates an additional instruction to retrieve the second output, the overall computational pattern remains substantially more efficient.

In terms of performance, the radix-2 butterfly acceleration yields a more pronounced improvement than isolated complex arithmetic. The FFT workload executes in approximately **158k cycles**, corresponding to a **6.5% reduction in cycle count compared to the baseline**.

9.6 Planned Additions

9.6.1 Radix-4 Butterfly Instruction

While the radix-2 butterfly acceleration provides a first step toward reducing FFT execution time, the majority of the computation in the 512-point FFT is performed using radix-4 butterflies.

Introducing a dedicated radix-4 butterfly instruction would allow a larger portion of the FFT to be executed using custom hardware primitives. Given the higher frequency of radix-4 operations compared to radix-2, accelerating these stages is expected to result in a more substantial reduction in cycle count than further optimizing the final radix-2 stage alone.

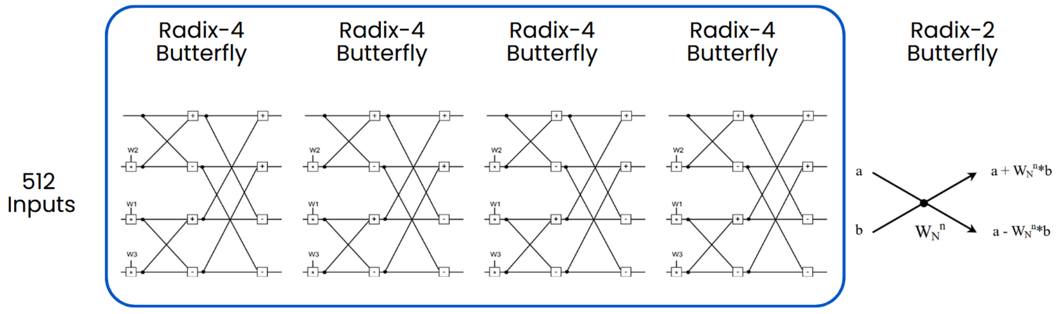

Figure 13 – Mixed-radix decomposition of a 512-point FFT.

9.6.2 Memory and Register Usage Optimization

Beyond arithmetic acceleration, further performance improvements may be achievable through a detailed analysis of memory access patterns and register utilization. Although custom instructions reduce instruction count, a significant portion of execution time can still be attributed to data movement between memory, registers, and the coprocessor interface.

A systematic study of how FFT data is loaded, stored, and reused across stages could reveal opportunities to reduce redundant memory accesses or improve register reuse. Potential optimizations include reorganizing data layouts, minimizing register spills, or leveraging shadow registers more effectively to retain intermediate results across butterfly operations.

10 Conclusion

This project explored instruction-level acceleration of a 512-point FFT on the CVA6 RISC-V processor using a tightly-coupled coprocessor via the CVXIF interface. Through systematic analysis of the baseline code and identification of performance-critical operations, several custom instructions were introduced to offload complex arithmetic and butterfly operations to dedicated hardware. The primary achievements can be summarized as follows :

Table 1 – Summary of implemented FFT optimizations and their impact on execution cycles

Optimization	**Cycles**	**Reduction vs. Baseline**
Baseline	169k	—
Complex multiplication	152k	10%
Complex addition/subtraction	164k	2.9%
Radix-2 butterfly	158k	6.5%

These results demonstrate that even modest custom instruction enhancements can lead to meaningful reductions in FFT execution time, particularly when targeting operations that dominate instruction count, such as complex arithmetic and butterfly recombinations.

Future work will focus on extending these techniques to radix-4 butterflies, which are more prevalent in the 512-point FFT, and on optimizing memory and register usage to further reduce execution cycles. Together, these directions have the potential to substantially accelerate FFT workloads on embedded RISC-V processors while maintaining compatibility and low-overhead hardware integration.

Table des figures