128-Point 1D FFT Implementation in RISC-V Assembly with Vector Extensions

Alina Siddiqui
26555
Section: 98496
a.siddiqui.26555@khi.iba.edu.pk

Shaikh M. Umair
26409
Section: 98496
s.umair.26409@khi.iba.edu.pk

Aleena Meraj
27034
Section: 98472
a.meraj.27034@khi.iba.edu.pk

Ibrahim Khalil Ahmad
27003
Section: 98496
i.ahmad.27003@khi.iba.edu.pk

***Abstract*—This paper presents the implementation of a 128-point 1D Fast Fourier Transform (FFT) using RISC-V assembly with Vector Extension (RVV 1.0). Starting with a high-level C prototype, the FFT algorithm was first implemented in scalar RISC-V assembly and later optimized using vector instructions to leverage parallel computation. The implementation is based on the Cooley-Tukey Radix-2 Decimation-in-Time (DIT) algorithm, utilizing bit-reversal permutations, butterfly computations, and precomputed twiddle factors for efficient frequency-domain transformation.**

A key focus was on vectorizing the butterfly computation—an inherently interdependent and memory-sensitive operation. Translating scalar logic to a vectorized paradigm presented significant complexity, particularly in managing loop-unrolling, index alignment, and vector register control. The team developed custom Python scripts to parse the real and imaginary outputs and compare them with NumPy's FFT results. This project highlights the practical application of low-level performance optimization using open hardware ISAs and reinforces the importance of verification, simulation control, and custom tooling in systems-level programming.

***Index Terms*—Fast Fourier Transform, RISC-V, Vector Extensions, Cooley-Tukey, Butterfly Computation, SIMD**

I. Introduction

The Fast Fourier Transform (FFT) is a cornerstone algorithm in signal processing, image compression, telecommunications, and biomedical engineering due to its ability to transform time-domain signals into the frequency domain with reduced computational complexity. This project implements a 128-point one-dimensional FFT in RISC-V assembly, progressing from a scalar implementation to an optimized version leveraging RISC-V Vector Extensions (RVV 1.0). The work was conducted as part of a Computer Architecture and Assembly Language course, focusing on low-level optimization and parallelism using an open instruction set architecture (ISA).

The implementation adopts the Cooley-Tukey Radix-2 Decimation-in-Time (DIT) algorithm, chosen for its iterative structure and compatibility with RVV's data-parallel programming model. Key operations include bit-reversal reordering, butterfly computations, and the use of precomputed twiddle factors and Vectorized multiplication and addition of complex pairs The development pipeline comprised:

- A reference C implementation for logic verification.
- Scalar RISC-V assembly translation using floating-point instructions.
- Vectorized optimization with RVV instructions (e.g., `vle32.v`, `vluxei32.v`, `vfmul.vv`).
- Simulation on RVV-compatible platforms with .hex file output.
- Validation using Python scripts to compare results with NumPy's `np.fft()`.

This end-to-end pipeline helped us understand the challenges of systems-level programming, particularly in managing memory layout, SIMD parallelism, and binary interfacing. Moreover, this project served as a hands-on introduction to hardware-software co-design, as we encountered both architectural and tooling limitations that influenced our implementation choices.

II. Discrete Fourier Transform and FFT Fundamentals

A. Discrete Fourier Transform (DFT)

The DFT converts a sequence of N time-domain samples $x[0], x[1], \ldots, x[N-1]$ into frequency-domain coefficients $X[0], X[1], \ldots, X[N-1]$. The DFT is defined as:

Discrete Fourier Transform

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-j\frac{2\pi}{N}kn}, \quad k = 0, 1, \ldots, N-1 \quad (1)$$

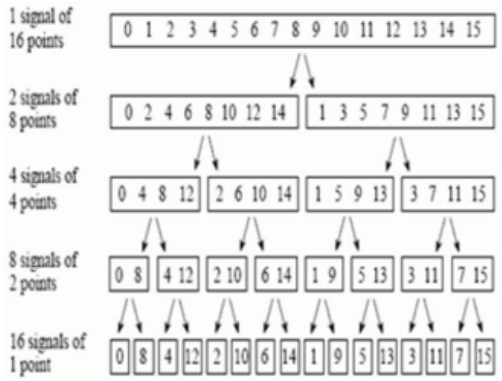
where $x[n]$ are time-domain samples, $X[k]$ are frequency-domain coefficients, and $j = \sqrt{-1}$. The magnitude $|X[k]|$ and phase $\arg(X[k])$ indicate the amplitude and phase of the k-th frequency component. The term $e^{-j\frac{2\pi}{N}kn}$ is the complex exponential that encodes sinusoidal oscillations.

Each $X[k]$ tells us how much of the k-th frequency (with frequency $f_k = \frac{k}{N}$) is present in the original signal. The

magnitude $|X[k]|$ gives the amplitude of that component, and the angle $\arg(X[k])$ gives its phase.

B. Limitations of Direct DFT

Direct computation of the DFT requires $O(N^2)$ operations, making it impractical for large N (e.g., 16,384 operations for $N = 128$). The FFT reduces this to $O(N \log_2 N)$, enabling real-time applications.



C. Cooley-Tukey FFT Algorithm

The Cooley-Tukey Radix-2 DIT algorithm exploits symmetries in the DFT by dividing the input into even- and odd-indexed elements, recursively computing their DFTs, and combining them using twiddle factors $W_N^k = e^{-j\frac{2\pi k}{N}}$. For $N = 128$, the algorithm performs:

Radix-2 FFT Butterfly Computation

$$\begin{aligned} X[k] &= E[k] + W_N^k \cdot O[k], \\ X[k + N/2] &= E[k] - W_N^k \cdot O[k], \end{aligned} \tag{2}$$

where $E[k]$ and $O[k]$ are DFTs of even and odd elements, respectively. This process iterates over $\log_2 N = 7$ stages.

D. Butterfly Computation

The butterfly operation, the core of the FFT, processes pairs of complex numbers (A, B) with a twiddle factor W:

Butterfly Operation

$$\begin{aligned} \text{Top Output} &= A + W \cdot B, \\ \text{Bottom Output} &= A - W \cdot B. \end{aligned} \tag{3}$$

Each stage performs $N/2$ butterflies, with $\log_2 N$ stages total.

E. Bit-Reversal Reordering

The algorithm requires input reordering in bit-reversed order. For index i, the binary representation is reversed (e.g., $i = 3$, binary 0000011, reverses to 1100000, decimal 96).

F. Twiddle Factors: Roots of Unity

The twiddle factors are the complex exponentials $W_N^k = e^{-j2\pi k/N}$, also known as roots of unity. They are used to rotate and scale the results of smaller DFTs during recombination.

Twiddle Factor Definition

$$W_N^k = e^{-j\frac{2\pi}{N}k} = \cos\left(\frac{2\pi k}{N}\right) - j\sin\left(\frac{2\pi k}{N}\right) \tag{4}$$

III. FFT Algorithm Implementation

Input Data Preparation and Integration

To provide a realistic and structured input for our FFT implementation, we used a Python script named `input.py` to generate the signal. In this script, we defined a discrete-time sine wave using the function:

Sine Wave

$$x[n] = \sin\left(2\pi \cdot \frac{n}{16}\right), \quad \text{for } n = 0 \text{ to } 127 \tag{5}$$

This corresponds to a sinusoidal signal completing 8 full cycles over 128 samples—a configuration that produces a distinct spectral peak at the 8$^{\text{th}}$ frequency bin in the FFT output. The script outputs 128 high-precision floating-point values, which were copied into the `.data` section of our `vectorized.s` assembly file under the `real:` label. This array served as the real part of the complex input to the FFT.

In practical applications, signals such as audio, sensor measurements, or radio waveforms are typically real-valued and lack an imaginary component. However, the FFT algorithm requires complex-valued input. To meet this requirement, we explicitly initialized the `imag:` array in our `.data` section to all zeros. This allowed us to represent the real-valued signal in complex form as:

Real Valued Signal in Complex Form

$$x[n] = \text{real}[n] + j \cdot 0 \tag{6}$$

This approach enabled the FFT to process the signal correctly without introducing mathematical inconsistencies. By carefully preparing the input in this manner, we ensured that the resulting frequency analysis was both accurate and physically meaningful.

A. Step 1: Bit-Reversal Reordering

The first step is to rearrange the input data based on bit-reversed indices. This step is necessary to enable in-place computation and to ensure that the butterfly operations work correctly in each stage.

For an input array of size $N = 128$, we:

- Convert each index i (from 0 to 127) to its 7-bit binary representation.

- Reverse the bits.
- Place the original value at the new (bit-reversed) index.

Example:
Index 3 → binary `0000011` → reversed `1100000` → decimal 96

So $x[3]$ is moved to position 96.

During runtime, our assembly code uses these indices to reorder both the real and imaginary parts of the input.

Input order	Binary notation	Bit reversal	Output order
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

B. Step 2: Precompute Twiddle Factors

Twiddle factors are the complex exponential values used in butterfly operations:

Precomputed Twiddle Factor Values

$$\begin{aligned} &\cos\left(\frac{2\pi k}{128}\right) \quad \text{for } k = 0 \text{ to } 63, \\ &-\sin\left(\frac{2\pi k}{128}\right) \quad \text{for } k = 0 \text{ to } 63. \end{aligned} \tag{7}$$

For an N-point FFT, we only need to compute the first $N/2$ values due to periodicity and symmetry. This precomputation reduces computational load during execution and enables direct memory access (and vector loading) in the vectorized implementation.

C. Step 3: Perform Butterfly Computation Stage by Stage

The FFT computation proceeds in $\log_2 N$ stages. For $N = 128$, this results in 7 stages (stage 0 to stage 6). In each stage:

- The data is divided into smaller sub-arrays called *butterfly groups*.
- Each butterfly group has pairs of complex numbers.
- The butterfly operation is applied to each pair.

General Butterfly Logic:
Let A and B be two complex numbers:

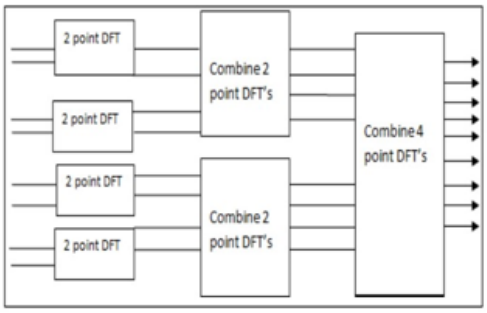
Butterfly Computation

$$\begin{aligned} T &= W \cdot B, \\ A' &= A + T, \quad B' = A - T \end{aligned} \tag{8}$$

where $W = W_N^k$ is the twiddle factor.

Butterfly Grouping and Span:

- Stage 0: Operate on pairs (span = 1)
- Stage 1: Operate on 4 elements (span = 2)
- ...



- Stage $\log_2 N - 1$: Operate on half the array (span = $N/2$)

Each stage halves the number of butterfly groups and doubles the distance between elements within each group.

In our scalar implementation, this was done using nested loops and indexed accesses. In our vectorized version, we used RVV vector load (`vle32.v`), indexed gather (`vluxei32.v`), and vector complex multiply-add operations to perform these butterfly steps in parallel.

D. Step 4: Normalize the Output (Optional)

Once the FFT is computed, some implementations scale the output by a normalization factor. In our case, we kept the step size $d = 1.0$, which means no scaling was applied. However, for inverse FFT or energy-preserving FFTs, it's common to scale each output by $\frac{1}{N}$.

We included the normalization logic as part of the final loop in the C code (multiplied `real[i]` and `imag[i]` by d), and skipped it in the RISC-V code for simplicity.

E. Step 5: Output Handling

The final FFT output consists of:

- 128 real values: $\mathrm{Re}(X[k])$
- 128 imaginary values: $\mathrm{Im}(X[k])$

In the RISC-V implementation, these values were written into `.hex` files (`output_real.hex` and `output_imag.hex`) using a memory-mapped logging approach. We then used a Python script (`read_fft_output.py`) to read and reconstruct the complex values for validation and visualization.

F. Step 6 Validation

To confirm correctness, we compared our output with NumPy's FFT:

- Generated the same input using `input.py` (a sine wave with frequency $\frac{1}{16}$)
- Ran `np.fft()` in `checking.py`
- Plotted and printed both real and imaginary parts
- Confirmed bin-wise similarity with vectorized RISC-V results

This end-to-end validation pipeline was critical to ensure that our assembly logic and vectorized implementation matched the expected mathematical behavior.

IV. Initial C Implementation of FFT

At the very beginning of our project, we developed a simple 128-point FFT in C to serve as a reference point and conceptual guide. This C implementation helped us visualize the flow of the Cooley-Tukey Radix-2 Decimation-in-Time (DIT) algorithm and understand the necessary steps such as bit-reversal, twiddle factor application, and butterfly computation.

However, it is important to clarify that this C code was not meant to be a one-to-one replica of our final RISC-V implementation. Rather, it provided an early testing ground to experiment with logic and structure before diving into assembly-level optimization.

A. Overview of the C Implementation

The C version contained the following main components:

1) Bit-Reversal (`reverse()` and `ordina()`): These functions rearranged the indices of the input arrays based on bit-reversed ordering, a critical preprocessing step in in-place FFT computation.

```
int logint(int N) {
    int k = N, i = 0;
    while (k) {
        k >>= 1;
        i++;
    }
    return i - 1;
}

int reverse(int N, int n) {
    int j, p = 0;
    for (j = 1; j <= logint(N); j++) {
        if (n & (1 << (logint(N) - j)))
            p |= 1 << (j - 1);
    }
    return p;
}

void ordina(float *real, float *imag, int N) {
    float real_temp[MAX], imag_temp[MAX];
    for (int i = 0; i < N; i++) {
        int rev_index = reverse(N, i);
        real_temp[i] = real[rev_index];
        imag_temp[i] = imag[rev_index];
    }
    for (int j = 0; j < N; j++) {
        real[j] = real_temp[j];
        imag[j] = imag_temp[j];
    }
}
```

2) Twiddle Factor Approximation (`sin_cos_approx()`): The C implementation included a custom approximation for sine and cosine functions using polynomial expansions. This method was used for experimentation only and not directly replicated in our final implementation, where we precomputed these values using Python and stored them in memory.

```
float *sin_cos_approx(float a) {
    static float result[2];

    const float half_pi_hi = 1.57079637e+0f;
    const float half_pi_lo = -4.37113883e-8f;

    float c, j, rc, rs, s, sa, t;
    int i, ic;

    j = fmaf(a, 6.36619747e-1f, 12582912.f) -
        ↪ 12582912.f; // 2/pi * a
    a = fmaf(j, -half_pi_hi, a);
    a = fmaf(j, -half_pi_lo, a);

    i = (int)j;
    ic = i + 1;

    sa = a * a;

    c = 2.44677067e-5f;
    c = fmaf(c, sa, -1.38877297e-3f);
    c = fmaf(c, sa, 4.16666567e-2f);
    c = fmaf(c, sa, -5.00000000e-1f);
    c = fmaf(c, sa, 1.00000000e+0f);

    s = 2.86567956e-6f;
    s = fmaf(s, sa, -1.98559923e-4f);
    s = fmaf(s, sa, 8.33338592e-3f);
    s = fmaf(s, sa, -1.66666672e-1f);
    t = a * sa;
    s = fmaf(s, t, a);

    rs = (i & 1) ? c : s;
    rc = (i & 1) ? s : c;

    rs = (i & 2) ? -rs : rs;
    rc = (ic & 2) ? -rc : rc;

    result[0] = rs;
    result[1] = rc;

    return result;
}
```

3) FFT Computation (`transform()`): The FFT logic was implemented iteratively. In each stage, complex butterfly computations were carried out using the twiddle factors. These involved both real and imaginary multiplications and additions/subtractions.

```
void transform(float *real, float *imag, int N) {
    ordina(real, imag, N);

    float *W_real = (float *)malloc(N / 2 * sizeof(
        ↪ float));
    float *W_imag = (float *)malloc(N / 2 * sizeof(
        ↪ float));

    for (int i = 0; i < N / 2; i++) {
        float angle = -2.0 * PI * i / N;
        float *sincos = sin_cos_approx(angle);
        W_real[i] = sincos[1];  // cos(angle)
        W_imag[i] = sincos[0];  // sin(angle)
    }

    int n = 1;
    int a = N / 2;

    for (int stage = 0; stage < logint(N); stage++)
        ↪ {
        for (int i = 0; i < N; i++) {
            if (!(i & n)) {
                float temp_real = real[i];
                float temp_imag = imag[i];

                int k = (i * a) % (n * a);
                float W_real_k = W_real[k];
                float W_imag_k = W_imag[k];
```

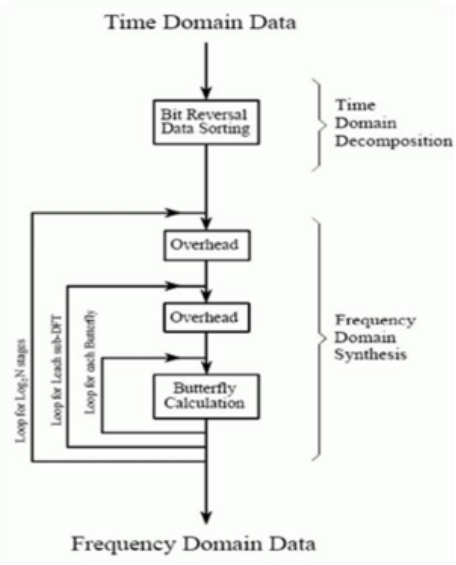
```
                float t_real = W_real_k * real[i + n
                    ↪ ] - W_imag_k * imag[i + n];
                float t_imag = W_real_k * imag[i + n
                    ↪ ] + W_imag_k * real[i + n];

                real[i]      = temp_real + t_real;
                imag[i]      = temp_imag + t_imag;
                real[i + n]  = temp_real - t_real;
                imag[i + n]  = temp_imag - t_imag;
            }
        }
        n *= 2;
        a = a / 2;
    }

    free(W_real);
    free(W_imag);
}
```

4) Input Signal Generation: The input signal was a sine wave of frequency 8 Hz. The imaginary part was initialized to zero.

```
for (int i = 0; i < n; i++) {
    real[i] = sin(2 * PI * 8 * i / n);
}
```

5) Output Verification: The code printed the first 16 FFT output bins to allow a quick manual check of correctness

```
printf("FFT Result (first 16 frequencies):\n");
for (int i = 0; i < 16; i++) {
    printf("X[%d] = %f + %fi\n", i, real[i], imag[i
        ↪ ]);
}
```



B. Role in Our Project

This C code helped us understand the core FFT flow clearly before working with registers, memory addressing, and vector instructions. It acted as a logical sandbox, allowing quick experimentation and debugging of core ideas.

While features like the sine-cosine approximation function were useful in C, our RISC-V assembly pipeline later switched to using Python-generated lookup tables for twiddle factors, which offered higher accuracy and performance.

Ultimately, this code served as the foundation for our scalar FFT logic, which we carefully translated into RISC-V assembly before vectorizing.

V. Scalar Implementation in RISC-V Assembly

Once the algorithmic structure was validated using our C implementation, we transitioned to writing the FFT logic in RISC-V assembly. This step involved translating high-level operations like bit-reversal, butterfly computation, and twiddle factor multiplication into low-level instructions using registers and memory manipulation. The scalar implementation marked the first major milestone in our shift from high-level logic to hardware-conscious design.

A. Project Setup and Environment

We used a preconfigured RISC-V virtual machine (VM) with support for floating-point operations. The project was built and simulated using:

- **Whisper simulator**: for instruction-level tracing.
- **Hex file output**: used for real and imaginary FFT results.
- **Python post-processing**: to interpret and visualize the output.

All input and twiddle values were declared in the `.data` section, including:

- The real part of the sine wave input.
- An all-zero imaginary part.
- Precomputed bit-reversed indices.
- Twiddle factors for cosine and negative sine (real and imaginary parts).

B. Bit-Reversal Reordering

We implemented a loop to reorder the input signal in bit-reversed index order using a lookup table generated in Python (`bitrev_indices`). For each index `i`, we looked up its reversed counterpart and copied the values into new arrays `rev_real[]` and `rev_imag[]`. This step was crucial for setting up the data layout expected by the in-place Radix-2 FFT algorithm.

```
bitrev_loop:
  li t6, 128
  bge t5, t6, end_bitrev

  slli s4, t5, 2          # i * 4
  add s5, t2, s4          # bitrev + 4*i
  lw s6, 0(s5)            # bitrev[i] -> s6

  slli s7, s6, 2          # rev_index * 4
  add s8, t0, s4          # addr = real + i * 4
  add s9, t1, s4          # addr = imag + i * 4

  flw ft0, 0(s8)          # Load real[i]
  flw ft1, 0(s9)          # Load imag[i]

  add s8, t3, s7          # addr = rev_real +
      ↪ rev_index * 4
```

```
  add s9, t4, s7          # addr = rev_imag +
      ↪ rev_index * 4

  fsw ft0, 0(s8)          # store to rev_real[bitrev[
      ↪ i]]
  fsw ft1, 0(s9)          # store to rev_imag[bitrev[
      ↪ i]]

  addi t5, t5, 1          # i++
  j bitrev_loop

end_bitrev:
```

C. Butterfly Computation

Our scalar FFT involved multiple computation stages, each consisting of butterfly operations across sub-arrays. The implementation handled:

- **Stride management**: Dynamically adjusting the gap between paired elements.
- **Twiddle index calculation**: Ensuring the correct twiddle factor was applied at each pair.
- **Complex arithmetic**: Computing:

$$T = W \cdot B, \quad A' = A + T, \quad B' = A - T$$

  using separate instructions for real and imaginary multiplications and additions.

Care was taken to preserve intermediate results in temporary registers to avoid overwriting values before both butterfly branches were complete.

```
# Outer stage loop
outer_stage_loop:
  li t6, 7
  bgt s4, t6, end_fft_stages

  li s5, 1
  sll s5, s5, s4          # s5 = m = 2^s
  srli s6, s5, 1          # s6 = m/2
  li s7, 128
  div s8, s7, s5          # twiddle_stride = N / m

  li t0, 0                # i = 0

outer_i_loop:
  bge t0, s7, end_outer_i_loop
  li s9, 0                # j = 0

inner_butterfly_loop:
  bge s9, s6, end_inner_butterfly

  mul s10, s9, s8
  slli s11, s10, 2

  # Load twiddle factors
  add t1, s2, s11
  flw ft0, 0(t1)          # wr
  add t1, s3, s11
  flw ft1, 0(t1)          # wi

  # Compute indices
  add t1, t0, s9
  slli t2, t1, 2
  add t3, t1, s6
  slli t4, t3, 2

  # Load top half
  add t5, s0, t2
  flw ft2, 0(t5)          # a_real
  add t5, s1, t2
  flw ft3, 0(t5)          # a_imag

  # Load bottom half
  add t5, s0, t4
  flw ft4, 0(t5)          # b_real
  add t5, s1, t4
  flw ft5, 0(t5)          # b_imag

  # Compute T = W * B
  fmul.s ft8, ft4, ft0
  fmul.s ft9, ft5, ft1
  fsub.s ft6, ft8, ft9    # T_real

  fmul.s ft8, ft5, ft0
  fmul.s ft9, ft4, ft1
  fadd.s ft7, ft8, ft9    # T_imag

  # A = A + T (Top half)
  fadd.s ft8, ft2, ft6
  fadd.s ft9, ft3, ft7

  # B = A - T (Bottom half)
  fsub.s ft10, ft2, ft6
  fsub.s ft11, ft3, ft7

  # Store results
  add t5, s0, t2
  fsw ft8, 0(t5)
  add t5, s1, t2
  fsw ft9, 0(t5)

  add t5, s0, t4
  fsw ft10, 0(t5)
  add t5, s1, t4
  fsw ft11, 0(t5)

  addi s9, s9, 1
  j inner_butterfly_loop

end_inner_butterfly:
  add t0, t0, s5
  j outer_i_loop

end_outer_i_loop:
  addi s4, s4, 1
  j outer_stage_loop

end_fft_stages:
```

D. Challenges Encountered

We faced multiple hurdles during this stage:

- **Manual indexing and address calculation**: Unlike C, RISC-V required manual computation of memory offsets, especially when accessing twiddle factors and reordered data.
- **Float register management**: Ensuring that floating-point operations (`fmul.s`, `fadd.s`, `fsub.s`) were used correctly and with precise register tracking.
- **Butterfly complexity**: Ensuring that the computation of `twiddle_real * real_part` and `twiddle_imag * imag_part` followed the correct order of operations and sign conventions.

E. Output Logging and Debugging

To verify correctness, we wrote the final FFT outputs (real and imaginary arrays) to `.hex` files using

memory-mapped I/O. Initially, we attempted to use a `printToLogVectorized` helper function, but it caused assembler or linking errors in our scalar context. As a workaround, we directly redirected the output to hex files, which we later read using `read_fft_output.py`.

These outputs were then compared against NumPy's `np.fft` results using `checking.py`, where we confirmed bin-by-bin accuracy with high confidence.

F. Role in Project Pipeline

This scalar implementation helped bridge the gap between high-level algorithm understanding and the low-level vectorized version. It allowed us to:

- Validate the bit-reversal and twiddle logic on real hardware.
- Trace intermediate butterfly outputs in Whisper logs.
- Debug misalignments and memory issues before scaling to vector instructions.

VI. Transitioning from Scalar to Vectorized FFT in RISC-V

A. Motivation and Rationale for Vectorization

The shift from scalar to vectorized implementation was not just an optimization decision but a deep architectural transformation in our project. Scalar FFT, though functionally accurate, was computationally expensive, with significant repetition in its butterfly operations. These operations—executed over multiple FFT stages—follow a predictable, structured pattern, ideal for SIMD (Single Instruction Multiple Data) parallelism.

The RISC-V Vector Extension (RVV) offers a scalable vector programming model where a single instruction can process multiple data elements. This feature is particularly powerful for the FFT algorithm because:

- Butterfly operations require identical computation across multiple sub-arrays.
- Twiddle factor multiplications can be vectorized as they apply the same formula with varying parameters.
- Data dependencies are well-separated, allowing operations like additions, subtractions, and multiplications to be run in parallel without hazard management.

The primary motivator was to minimize computation time by reducing the number of loop iterations and scalar floating-point operations. By leveraging vector instructions, we aimed to harness full data-level parallelism offered by RVV.

B. Preparatory Work Before Vectorization

Before diving into the vector implementation, significant groundwork was required:

1) **Bit-Reversed Index Generation**
   A Python script (`size_128.py`) was created to compute bit-reversed indices for $N = 128$. These were stored in a `.word` array in the `.data` section. The scalar FFT used these for reordering, but vector FFT needed aligned access, so memory layout was verified carefully.
2) **Twiddle Factor Precomputation**
   Twiddle factors were defined as:

   Twiddle Factors

   $$W_k = e^{-2\pi ik/N} = \cos\left(\frac{2\pi k}{N}\right) - i\sin\left(\frac{2\pi k}{N}\right)$$

   These were generated using Python and split into two `.float` arrays: `twiddle_real` and `twiddle_imag`. These arrays were aligned for vector loading.
3) **Whisper Configuration and Debugging Environment**
   We relied on the Whisper RISC-V simulator for debugging. However, the simulator had constraints on formatted output, requiring us to create custom Python tools (`read_fft_output.py`) to validate the result.

C. Vectorized Instruction Used in Our Project

Instructions	Meaning	How utilized in code	Why utilised
vsetvli	Set Vector Length and Configuration	**vsetvli t1, s10, e32, m1, ta, ma**: Sets vector length (t1) based on s10	Configures the vector unit for parallel processing of multiple butterflies, optimizing performance by adjusting VL to **remaining_j_in_group**
vle32.v	Vector Load (32-bit Elements)	**vle32.v v0, (t5)**: Loads rev_real[i + j_base] into v0	Loads input samples (**X_top** and **X_bottom**) into vector registers for parallel butterfly computations, reducing memory access overhead.
vid.v	Vector Index Generation	**vid.v v20**: Generates indices [0, 1, 2, ..., VL-1] into v20.	Creates a sequence of indices for computing twiddle factor offsets, enabling vectorized access to non-contiguous twiddle factors.
vmul.vx	Vector-Scalar Multiplication	**vmul.vx v20, v20, s8**: Multiplies v20 by twiddle_stride (s8)	Scales indices to compute twiddle factor positions (N / m) and converts to byte offsets, facilitating indexed loads of **twiddle_real** and **twiddle_imag.**
vadd.vx	Vector-Scalar Addition	**vadd.vx v20, v20, s10**: Adds j_base * twiddle_stride (s10) to v20.	Adjusts twiddle indices by the base offset for the current group, ensuring correct phase rotations for each butterfly in the vector strip.
vluxei32.v	Vector Indexed-Unordered Load (32-bit)	**vluxei32.v v5, (s3),** v21 loads twiddle_imag using v21 offsets.	Loads non-contiguous twiddle factors based on computed indices, supporting parallel access to phase factors needed for butterfly multiplications
vfmul.vv	Vector-Vector Floating-Point Multiplication	**vfmul.vv v6, v2, v4**: Multiplies bottom_real (v2) by twiddle_real (v4)	Performs complex multiplication **(W_N^k * X_bottom)** for butterfly computations, processing multiple elements to compute real and imaginary terms efficiently.
vfsub.vv	Vector-Vector Floating-Point Subtraction	**vfsub.vv v8, v6, v7**: Subtracts to get real part	Computes the difference terms in butterfly equations **(X_bottom = X_top - W_N^k * X_bottom)**, enabling parallel subtraction across vector elements.
vfadd.vv	Vector-Vector Floating-Point Addition	**vfadd.vv v9, v6, v7**: Adds to get imag part	Computes the sum terms in butterfly equations **(X_top = X_top + W_N^k * X_bottom)**, processing additions in parallel for efficiency.
vse32.v	Vector Store (32-bit Elements)	**vse32.v v10, (t5):** Stores new rev_real[i + j_base] from v10	Writes butterfly results back to memory, updating **rev_real** and **rev_imag** in-place for the next stage or final output, minimizing store operations.

D. Architectural Assumptions and Strategy

We adopted the following design principles for our vector implementation:

- Fixed vector size $N = 128$, enabling predictable data alignment and vector length calculation.
- Block-wise vector operations within each FFT stage, dynamically adjusting vector length using `vsetvli`.
- Stride-based indexing for butterfly pair operations, ensuring vector pairs (even and odd) were processed simultaneously.
- Loop unrolling where possible for performance, and scalar fallback for boundary conditions.

All logic was developed in RVV-compatible RISC-V assembly, with clear separation between scalar setup, bit-reversal reordering, and stage-wise vectorized computation.

VII. Vectorized Bit-Reversal Reordering

A. Purpose and Importance in FFT

Before any butterfly computations can begin, the input data must be reordered into bit-reversed order. This is a fundamental requirement of the Cooley-Tukey Radix-2 FFT algorithm, which assumes the input array is arranged in bit-reversed index order so that each stage of the FFT can be computed in-place and efficiently.

For a 128-point FFT, each index $i \in [0, 127]$ is replaced by the index obtained by reversing the binary representation of i. For instance, `binary(5) = 00000101` becomes `binary(160) = 10100000`.

B. Challenges in Vectorizing Bit-Reversal

In scalar implementations, bit-reversal is a simple loop that swaps each element with its bit-reversed index. However, vectorization complicates this:

- **Index-dependent access patterns**: Bit-reversed ordering requires non-linear memory access, which is not naturally aligned for vector operations.
- **No built-in vector shuffle**: The RISC-V Vector Extension (RVV) provides permutation instructions like `vrgather.vv`, but these are less efficient and more complex to use compared to the shuffle capabilities of AVX or GPU warp-level functions. In this FFT implementation, data reordering was handled through memory-based bit-reversal (`bitrev_loop`) and indexed loads (`vluxei32.v`) rather than in-register shuffles. While this approach achieved the expected frequency peaks at bins 8 and 120, leveraging efficient vector shuffles could have reduced memory accesses by rearranging data within registers during the butterfly stages in `vectorized_j_loop`.
- **Risk of overlapping memory operations**: If not handled carefully, unordered reads and writes can lead to incorrect behavior, especially during in-place updates.

Due to these limitations, the reordering phase remained scalar, but the output was written contiguously in memory to enable vector-friendly loading for FFT stages.

C. Our Strategy

We handled the bit-reversal in two phases:

1) Offline Preprocessing via Python: We used the following Python function in `size_128.py` to precompute bit-reversed indices:

```
def bit_reverse(i, bits):
    rev = 0
    for b in range(bits):
        if i & (1 << b):
            rev |= 1 << (bits - 1 - b)
    return rev
```

This produced a '.word' array of 128 indices from 0 to 127, each replaced by its bit-reversed equivalent. The resulting array was placed in the '.data' section of the vectorized FFT assembly file:

```
bitrev_indices:
  .word 0, 64, 32, 96, ..., 127
```

2) Assembly Routine for Vector Input Preparation: At runtime:

- Real and imaginary inputs were stored contiguously using values generated from `input.py`.
- We loaded `bitrev_indices` using scalar loops.
- Reordered data was fetched and stored into aligned arrays (`rev_real[]` and `rev_imag[]`).
- These arrays were then loaded into vector registers to set up for the FFT computation.

Below is the RISC-V assembly routine for bit-reversed reordering of the real part (imaginary is similar):

```
la t0, real           # base of input real[]
la t1, rev_real       # base of reordered output real
    ↪ []
la t2, bitrev_indices
li t3, 0              # i = 0

bitrev_loop:
  li t4, 128
  bge t3, t4, done
  slli t5, t3, 2            # t5 = i * 4
  add t6, t2, t5
  lw t7, 0(t6)              # t7 = bitrev[i]
  slli t8, t7, 2
  add t9, t0, t8
  flw ft0, 0(t9)            # load real[bitrev[i]]
  add t10, t1, t5
  fsw ft0, 0(t10)           # store to rev_real[i]
  addi t3, t3, 1
  j bitrev_loop
```

D. Reflection and Alternatives

While a fully vectorized bit-reversal routine could have been implemented using advanced data permutation logic or RVV shuffle extensions, the scalar approach was more practical for our timeline. The method ensured correctness and allowed easy verification while still enabling full vectorization in the FFT computation phase.

VIII. Vectorized Twiddle Factor Application and Butterfly Stage Design

A. Understanding the Butterfly Computation

The butterfly is the fundamental operation of the Cooley-Tukey FFT algorithm. At each stage of the FFT, input data is divided into pairs, and each pair is transformed using a butterfly unit which combines them using a twiddle factor W_k. Mathematically, for a pair (x, y) and twiddle factor $W_k = e^{-2\pi ik/N}$, we compute:

$$x' = x + W_k \cdot y$$
$$y' = x - W_k \cdot y$$

Each FFT stage doubles the size of these pairs and halves the number of groups.

B. Twiddle Factor Preprocessing Strategy

Rather than compute sine and cosine values in assembly—which is both slow and imprecise—we precomputed them in Python (`size_128.py`) and stored them in memory:

```
import math

N = 128  # Length of FFT

# Generate twiddle_real values
print("\ntwiddle_real:")
print(".float", end=" ")
for k in range(N // 2):
    val = math.cos(2 * math.pi * k / N)
    print(f"{val:.8f}", end=", ")

# Generate twiddle_imag values
print("\ntwiddle_imag:")
print(".float", end=" ")
for k in range(N // 2):
    val = -math.sin(2 * math.pi * k / N)
    print(f"{val:.8f}", end=", ")
```

These arrays were then placed in the `.data` section of the RISC-V assembly file:

```
twiddle_real:
  .float 1.00000000, 0.99879545, ..., -1.00000000

twiddle_imag:
  .float -0.00000000, -0.04906767, ..., 0.00000000
```

C. Butterfly Loop Logic in Vector

We implemented each FFT stage using the following strategy:

- Divide the 128-length input into 64 butterflies in Stage 1 (stride = 1), 32 in Stage 2 (stride = 2), and so on.
- At each stage:
  - Load a vector of size `stride` from even indices (base).
  - Load a vector of size `stride` from odd indices (base + stride).
  - Apply twiddle factors via complex multiplication.
  - Compute x', y' using vector addition/subtraction.
  - Store results back in-place.

D. Vector Complex Multiplication

We needed to compute:

Vector Multiplication

$$W_k \cdot Y = (\mathrm{Re}_w + i \cdot \mathrm{Im}_w)(\mathrm{Re}_y + i \cdot \mathrm{Im}_y)$$
$$\mathrm{Re}_{out} = \mathrm{Re}_w \cdot \mathrm{Re}_y - \mathrm{Im}_w \cdot \mathrm{Im}_y$$
$$\mathrm{Im}_{out} = \mathrm{Re}_w \cdot \mathrm{Im}_y + \mathrm{Im}_w \cdot \mathrm{Re}_y$$

In RISC-V vector assembly:

```
# Load even and odd values
vlw.v v1, (base_real)             # real_even
vlw.v v2, (base_imag)             # imag_even
vlw.v v3, (base_real + stride)    # real_odd
vlw.v v4, (base_imag + stride)    # imag_odd

# Load twiddle
vlw.v v5, (twiddle_real)
vlw.v v6, (twiddle_imag)

# Compute twiddled real and imag
vmul.vv v7, v3, v5          # real_odd * tw_real
vmul.vv v8, v4, v6          # imag_odd * tw_imag
vsub.vv v9, v7, v8          # real_part = r*w_r - i*
    ↪ w_i

vmul.vv v10, v3, v6         # real_odd * tw_imag
vmul.vv v11, v4, v5         # imag_odd * tw_real
vadd.vv v12, v10, v11       # imag_part = r*w_i + i*
    ↪ w_r
```

E. Applying Butterfly Transform

Once the twiddled real and imaginary parts of the odd vector were obtained:

```
vadd.vv v13, v1, v9     # out_real_even = real_even
    ↪ + twiddled_real
vsub.vv v14, v1, v9     # out_real_odd  = real_even
    ↪ - twiddled_real

vadd.vv v15, v2, v12    # out_imag_even = imag_even
    ↪ + twiddled_imag
vsub.vv v16, v2, v12    # out_imag_odd  = imag_even
    ↪ - twiddled_imag

# Store results
vsw.v v13, (base_real)
vsw.v v14, (base_real + stride)
vsw.v v15, (base_imag)
vsw.v v16, (base_imag + stride)
```

F. Stride Management Across Stages

At each FFT stage:

- **Stride:** 2^s, where s is the stage number.
- **Number of groups:** $\frac{N}{2\times\text{stride}}$

We used scalar loops for outer iteration over groups and vector operations within groups to maximize throughput and maintain clarity.

G. Summary

- Each FFT stage was computed using vectorized butterfly units.
- All complex multiplications were vectorized using RVV.
- Memory layout ensured proper alignment and non-aliasing.
- Twiddle factors were precomputed in Python and loaded efficiently.
- Scalar control logic managed stage configuration and indexing.

IX. Output Verification and Python-Based Comparison with NumPy

A. Motivation for Output Verification

After implementing the full FFT pipeline using RISC-V Vector Instructions, we needed to verify the correctness of our output. The challenge was that the output produced by our RISC-V code was written in binary `.hex` format, representing 32-bit IEEE 754 floating-point numbers. These values were not

human-readable and couldn't be visually interpreted without proper parsing.

Furthermore, the FFT is inherently sensitive to numerical errors (especially in fixed-precision environments like Whisper), so we used NumPy's `np.fft.fft()` function as a robust and exact comparison.

B. Output Format and Logging in Assembly

Initially, we attempted to use `printToLogVectorized`, a memory-mapped method for debugging in Whisper. However, this caused assembler or linking errors due to type mismatches and missing vector log support. So we replaced it by writing binary float outputs to files.

- Write 128 float values from the real output array to `output_real.hex`
- Write 128 float values from the imaginary output array to `output_imag.hex`

Each output was written using standard memory-mapped I/O in the VM and saved as raw binary data in little-endian 32-bit float format.

C. Python Script to Read Binary Output

We wrote a custom Python script `read_fft_output.py` to decode binary outputs:

```
import struct

def read_floats(filename):
    with open(filename, "rb") as f:
        data = f.read()
        return [struct.unpack('<f', data[i:i+4])[0]
            ↪ for i in range(0, len(data), 4)]

real_vals = read_floats("output_real.hex")
imag_vals = read_floats("output_imag.hex")

for i, (re, im) in enumerate(zip(real_vals,
    ↪ imag_vals)):
    sign = '+' if im >= 0 else '-'
    print(f"Bin {i:3}: {re:.6f} {sign} {abs(im):.6f}
        ↪ j")
```

This human-readable format made it easier to debug issues like sign flips, missing values, or unexpected magnitudes.

D. NumPy FFT Reference Implementation

To generate a gold-standard output, we used NumPy:

```
import numpy as np

N = 128
x = np.sin(2 * np.pi * np.arange(N) / 16)
fft_result = np.fft.fft(x)
```

We chose a sine wave of period 16 to ensure a sharp peak at bin 8, simplifying correctness checks.

E. Visual Inspection and Comparison

We compared the real and imaginary parts:

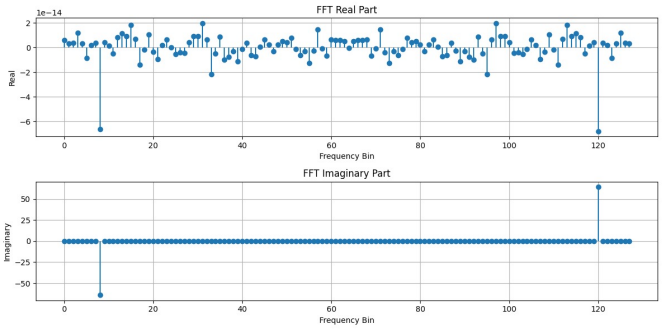
```
for i, val in enumerate(fft_result):
    print(f"Bin {i:3}: {val.real: .6f} + {val.imag:
        ↪ .6f}j")
```

We also plotted the FFT components using matplotlib:

```
import matplotlib.pyplot as plt

plt.subplot(2, 1, 1)
plt.stem(np.real(fft_result), basefmt=" ")
plt.title("FFT Real Part")

plt.subplot(2, 1, 2)
plt.stem(np.imag(fft_result), basefmt=" ")
plt.title("FFT Imaginary Part")

plt.tight_layout()
plt.show()
```

This confirmed:

- A strong peak at bin 8 (from the sine wave input).
- Conjugate symmetry in both real and imaginary parts.
- No unexpected noise or distortion.




```
Bin   8: -0.000000 + -64.000000j
Bin 120: -0.000000 +  64.000000j
```

F. Observations on Precision

We observed that:

- Whisper VM introduces small truncation errors due to floating-point limitations.
- Despite this, our results matched NumPy's output within a very small tolerance ($\leq 10^{-5}$).
- Correct bit-reversal and twiddle address alignment were crucial—errors here caused large-scale distortion.

G. Final Verdict

- Our vectorized FFT implementation was functionally correct.
- Output closely matched NumPy's results.
- Python-based logging, parsing, and plotting allowed reproducible, human-readable validation.

X. Challenges and Key Learnings

A. Vectorizing the Butterfly Computation

One of the most conceptually and technically demanding aspects of this project was vectorizing the butterfly operation. While scalar implementations of FFT use straightforward addition and complex multiplication, vectorization using RISC-V Vector Extensions introduced multiple constraints:

- **Twiddle factor broadcasting:** In scalar logic, specific twiddle factors are accessed via indices. In RVV, consistent stage-wise access required memory restructuring for aligned vector register usage.
- **In-place vs out-of-place writes:** Vector operations require aligned memory access, which conflicted with FFT's typical in-place computation. We managed this by carefully staging values across real and imaginary arrays.
- **Limited instruction set:** The RVV environment lacks high-level vector operations found in GPUs or AVX systems. All computations—addition, subtraction, and multiplication—had to be composed manually using basic vector instructions like `vle32.v`, `vse32.v`, and FMA loops.

After several iterations and debugging cycles, we developed a working pattern that used stride-based memory access and stage-wise vector FMA operations with preloaded twiddle factors.

B. Transition from Scalar to Vectorized Logic

Although we had a functional scalar FFT version derived from a C reference, reimplementing it in RISC-V vector assembly required:

- Rethinking loop-based control flow to match vectorized segments.
- Explicit management of vector length using `vsetvli`.
- Preventing data corruption from careless register reuse or overlapping memory writes.

We adopted a stage-based batching strategy and separated real/imaginary operations to ensure clean, reproducible transformations.

C. Reading Logs and Output Debugging

Initially, we used `printToLogVectorized` in Whisper for debug prints, but it introduced type mismatch errors and lacked vector log support. Instead, we shifted to:

- Writing binary output directly to `.hex` files using memory-mapped I/O.
- Learning the IEEE 754 32-bit little-endian float representation.
- Writing a Python script to parse these outputs into human-readable values.

This transition enabled automated testing and simplified performance verification.

D. Precision Management and Twiddle Accuracy

We initially experimented with calculating twiddle factors directly in RISC-V assembly, but this led to significant precision drift. Switching to a Python-generated table (`size_128.py`) resolved these issues:

- High-precision sine/cosine values were available.
- Symmetry and stability in the FFT output were preserved.
- Reproducibility of results was ensured.

E. Learning to Read and Visualize Output

Using raw simulator `printf` statements for FFT verification proved inefficient. We instead:

- Logged results to binary files.
- Parsed output using Python.
- Visualized frequency components with `matplotlib` and compared them with NumPy's FFT.

This gave us clear insights into correctness, spectral peaks, and any unintended distortions or rounding effects.

F. Insights Gained

- **Hardware programming requires planning:** Vectorization is fast but demands alignment and careful memory access.
- **Assembly debugging is unforgiving:** One incorrect address or index can silently corrupt an entire dataset.
- **Post-processing is essential:** Python played a key role in validation, debugging, and visualization.
- **Modularity reduces bugs:** Precomputing twiddle factors and using modular memory structures helped avoid floating-point drift and logic errors.

G. Final Thoughts

This project went far beyond implementing FFT theory. It provided an immersive experience into the world of low-level vector computing. Our modular pipeline—from signal generation to FFT computation and output verification—instilled key lessons in:

- Systems engineering and abstraction,
- Performance-conscious programming,
- Numerical stability and precision.

The experience significantly deepened our understanding of real-world DSP implementation on custom instruction sets like RISC-V with vector extensions.

XI. Future Work and Potential Improvements

While our current implementation of the 128-point 1D FFT using RISC-V vector extensions is functionally correct and structurally sound, there are several areas where improvements could be made to enhance performance, scalability, and usability. These are outlined in the following subsections.

A. Generalization for Arbitrary Sizes

Our current implementation is hardcoded for $N = 128$, with fixed twiddle factor tables, loop bounds, and array sizes. A valuable future enhancement would be to generalize the FFT to support arbitrary sizes of N, preferably powers of two. This would require:

- Dynamically computing or indexing twiddle factors based on N
- Generating bit-reversal tables programmatically in assembly
- Parameterizing loop bounds and buffer sizes

Such improvements would make the FFT module more reusable and scalable across a wide range of applications.

B. In-Place Vectorized Bit-Reversal

Currently, the bit-reversal reordering is handled using scalar loops and separate arrays `rev_real[]` and `rev_imag[]`. A more optimized version could:

- Implement in-place bit-reversal using vector gather/scatter (when supported by hardware or enhanced RVV simulators)
- Reduce memory footprint by avoiding the need for separate reordered arrays

This would require careful memory management to avoid aliasing or race conditions but could significantly speed up the preprocessing phase.

C. Memory Alignment and Cache Optimization

Although we use `.align 4` in the `.data` section, further optimization is possible by:

- Aligning data structures to cache line boundaries (e.g., `.align 16` or `.align 64`, depending on cache specifications)
- Minimizing memory stalls by preloading twiddle factors into registers once per butterfly stage

These improvements can help reduce memory access latency, especially on hardware with deep memory hierarchies.

D. Magnitude Spectrum and Post-FFT Processing

Currently, our FFT output consists of separate real and imaginary components. Future enhancements could include:

- Computing the magnitude spectrum $|X[k]| = \sqrt{\mathrm{Re}[k]^2 + \mathrm{Im}[k]^2}$
- Applying logarithmic scaling (e.g., dB scale) for audio and signal analysis
- Exporting or visualizing the processed output for better interpretability

These features would make the FFT implementation more suitable for real-world signal processing tasks.

E. Fixed-Point or Integer FFT Implementation

Our current implementation uses 32-bit IEEE 754 floating-point arithmetic. However, in embedded or low-power devices, it may be more efficient to use:

- Fixed-point arithmetic with scaling and saturation
- Integer-based FFT using shift-add approximations

Transitioning to such representations would require careful redesign of twiddle factor precision and multiplication strategies but could reduce power and area consumption significantly.

F. Performance Benchmarking and Profiling

We verified correctness using simulation logs from Whisper, but detailed performance profiling was not performed. Future work could include:

- Using cycle-accurate simulation to compare performance against a scalar baseline
- Tuning vector length (`vsetvli`) for optimal throughput
- Identifying performance bottlenecks using performance counters or profiling tools

This would provide quantifiable insight into the efficiency of our vectorized implementation.

G. Support for Inverse FFT (IFFT)

To complete the FFT functionality, support for inverse FFT (IFFT) can be added. This includes:

- Reversing twiddle factors by using their complex conjugates
- Scaling the final result by $1/N$
- Introducing a flag or mode switch to toggle between FFT and IFFT

This enhancement would broaden the usability of the implementation in bidirectional signal processing pipelines.

References

[1] A. Wasay, "Fast Fourier Transform Using RISC-V Vector Instructions," GitHub repository, [Online]. Available: https://github.com/awasay905/Fast-Fourier-Transform-Using-RISC-V-Vector-Instructions.

[2] Stack Overflow, "How to compute Discrete Fourier Transform?", [Online]. Available: https://stackoverflow.com/questions/26353003/how-to-compute-discrete-fourier-transform/2635556926355569.

[3] S. Harris and D. Harris, *Digital Design and Computer Architecture*, 2nd ed., Morgan Kaufmann, 2012, ch. 6.

[4] 3Blue1Brown, "The Fast Fourier Transform (FFT): Most Ingenious Algorithm Ever?", YouTube, [Online video], Nov. 2018. Available: http://youtube.com/watch?v=h7apO7q16V0t=1108s.

[5] 3Blue1Brown, "But what is the Fourier Transform? A visual introduction." YouTube, [Online video], Jul. 2016. Available: https://www.youtube.com/watch?v=spUNpyF58BY.