Fast Fourier Transform Implementation in RISC-V Assembly: From Theory to Hardware Execution

Saad Thaplawala*, Sharjeel Chandna†, Saad Inam‡, Hassan Jabbar§
Team Syzgy
Department of Computer Engineering, IBA University

***Abstract*—This report presents a detailed account of our final year computer architecture project: implementing the 1D Fast Fourier Transform (FFT) algorithm using RISC-V scalar assembly. The goal was to perform FFT for $N = 1024$ inputs in pure vector assembly while supporting runtime computation of bit-reversal indices and pre-computed twiddle factors. We explore the theoretical background, design strategy, stages of implementation, and key challenges—particularly twiddle factor accuracy and debugging limitations. The final result provides a fully scalable working foundation for FFT and opens pathways for further extension into vectorized implementations and a value of N greater than 1024.**

***Index Terms*—RISC-V, Fast Fourier Transform, Assembly Programming, Digital Signal Processing, Low-level Optimization**

I. Introduction

The Fast Fourier Transform (FFT) is a cornerstone algorithm in the field of digital signal processing (DSP), with broad applications in communications, image processing, audio analysis, radar, and many other areas. At its core, the FFT enables the efficient transformation of a signal from the time domain—where data is represented as a sequence of values over time—into the frequency domain, where the same data is analyzed in terms of its sinusoidal frequency components. This is crucial for tasks like filtering, spectral analysis, and data compression.

The mathematical foundation of the FFT is the Discrete Fourier Transform (DFT), which computes the frequency spectrum of a signal. For a sequence of N complex numbers, the DFT produces a new sequence. This equation describes the projection of the input signal onto orthogonal sinusoidal basis functions. However, the direct computation of DFT requires $\mathcal{O}(N^2)$ operations, which becomes prohibitively expensive for large N. The FFT reduces this to $\mathcal{O}(N \log N)$ by exploiting symmetries and redundancies in the computation, thereby making real-time signal processing feasible.

There are several FFT algorithms, but the most common is the Cooley-Tukey algorithm, which recursively divides the DFT into smaller DFTs by separating the even and odd indexed elements of the input. This divide-and-conquer approach rearranges data through a bit-reversal permutation, followed by a sequence of butterfly operations that combine and transform the inputs at each stage using twiddle factors.

Each of these steps—bit-reversal, twiddle factor application, and butterfly computation—plays a critical role in achieving the logarithmic efficiency of FFT. While these concepts are typically implemented in high-level languages such as Python, MATLAB, or C using optimized libraries (e.g., NumPy, FFTW), implementing FFT in a low-level assembly language exposes the developer to fundamental architectural and computational concerns: memory alignment, register management, loop unrolling, floating-point precision, and instruction-level parallelism.

Our project focuses on implementing a scalar version of the 1D FFT in RISC-V assembly, using a modest input size of $N = 8$. The scalar nature of the implementation means that each operation (e.g., multiplication, addition) is applied to individual elements sequentially, rather than operating on vectors or arrays simultaneously. In contrast, a vectorized implementation—which we initially aimed to develop—would take advantage of SIMD (Single Instruction, Multiple Data) instructions available in RISC-V vector extensions, allowing the parallel computation of multiple FFT operations in fewer cycles. However, due to hardware limitations and team resource constraints, we concentrated on building a

robust and fully functional scalar FFT engine first.

Unlike high-level environments, RISC-V does not offer native libraries for trigonometric functions like sine or cosine, nor for complex number arithmetic. This significantly increases the difficulty of generating and applying twiddle factors dynamically. Furthermore, precise floating-point computation is constrained by limited instruction support (e.g., `fmul.s`, `fadd.s`, `fsub.s`), and even small inaccuracies can propagate through stages of the algorithm, resulting in incorrect outputs.

To overcome these challenges, our team adopted an iterative development approach. We began by hardcoding twiddle factors and bit-reversal indices to validate the butterfly logic and memory layout. Once the scalar flow was verified to produce correct outputs, we gradually replaced the fixed components with runtime logic, including bit-reversal computation using bitwise operations and conditional loading of twiddle values based on stage and index. This strategy allowed us to balance complexity with progress and build confidence in each part of the FFT pipeline.

Although we were unable to fully integrate vectorized operations due to time and manpower limitations, the scalar FFT implementation successfully mirrors the structure of Cooley-Tukey and serves as a strong foundation for future optimization. More importantly, the project provided invaluable hands-on experience in understanding how high-level signal processing algorithms are implemented and optimized at the hardware instruction level.

II. Team Strategy and Initial Problems

At the outset of the project, our team adopted a modular approach to implementing the FFT in RISC-V assembly. The idea was to divide the algorithm into three logical components and assign them to team members based on their comfort levels and responsibilities:

- Bit-reversal permutation logic
- Twiddle factor calculation
- Butterfly computation

This initial strategy was motivated by the clarity it promised. However, as implementation began, several critical issues surfaced. Although modularization is typically an effective software engineering practice, in the case of FFT—especially when implemented in a low-level language like RISC-V—it introduced a number of practical limitations.

Each of the modules differed significantly in both complexity and interdependency. For example, while bit-reversal permutation is largely a straightforward indexing operation, it still relies heavily on low-level bitwise operations that must be precisely crafted in assembly. On the other hand, the twiddle factor logic posed a deeper challenge—it is inherently dependent on the stage number and butterfly size at each level of the FFT. Since these values change dynamically as the FFT progresses, the twiddle factor module could not be developed or tested in isolation without a full understanding of the surrounding logic and data dependencies.

In RISC-V, the absence of high-level constructs makes these challenges even more pronounced. Every instruction—whether it's a memory load, floating-point operation, or branch—must be explicitly handled, and any minor bug, such as a register overwrite or an incorrect index computation, can result in cascading errors that are difficult to trace. This made isolated module testing nearly impossible and significantly slowed down debugging efforts.

After spending considerable time struggling with the fragmented approach, we re-evaluated our strategy. As a team, we regrouped and decided to invest more time in studying the theoretical underpinnings of the FFT algorithm. We turned to Saad's Design and Analysis of Algorithms course notes, which helped us understand the mathematical recursion, symmetry, and periodicity inherent in the Cooley-Tukey FFT algorithm. To complement the textual explanations, we referred to YouTube tutorials and visual demonstrations that helped us grasp how the divide-and-conquer paradigm breaks the DFT into smaller subproblems, and how complex exponentials (twiddle factors) are used to recombine the results of these subproblems across FFT stages.

Despite developing a solid theoretical foundation, implementation continued to pose serious challenges. Unlike high-level environments that abstract away mathematical operations, RISC-V lacks any form of built-in support for:

- Trigonometric functions such as `sin()` or `cos()`
- Complex number arithmetic

- Floating-point libraries beyond basic operations like `fadd.s` or `fmul.s`

Thus, even basic tasks like generating twiddle factors at runtime, or converting integer values to floating-point format, required elaborate sequences of instructions and careful register management.

Initially, our ambition was to implement a fully vectorized FFT in RISC-V assembly for $N = 1024$. This was aimed at exploring RISC-V's vector extensions and exploiting SIMD (Single Instruction, Multiple Data) operations to parallelize butterfly computations and reduce latency. However, as we started to prototype, it became evident that beginning with such a large input size and a fully vectorized structure was far too ambitious for a first implementation, especially given our limited experience with RISC-V's vector ISA and the debugging constraints of simulators like Venus.

Moreover, vectorization itself brings another layer of complexity. It involves data alignment, loop unrolling, and careful handling of vector register states—all of which require a deep understanding of both the hardware model and the vector instruction set. Recognizing the difficulty of jumping directly into a large-scale vector FFT, we made a strategic decision to scale down the problem.

We pivoted to a smaller, scalar-based FFT implementation using $N = 4$ and $N = 8$, focusing on hardcoded values. This gave us an opportunity to validate our understanding of butterfly logic, verify correct bit-reversal ordering, and test twiddle factor applications without worrying about generalization or parallelization.

Once we had a functional hardcoded version, we started converting static logic to runtime computation. Our first goal was to dynamically compute bit-reversal indices. Since these depend only on the number of bits in the index (which is $\log_2 N$), the conversion to runtime was relatively straightforward using shifts and masks.

Next, we turned to the much more complex task of runtime twiddle factor computation. Unlike bit-reversal, twiddle factors involve calculating trigonometric functions—specifically, $\cos\left(\frac{2\pi k}{N}\right)$ and $\sin\left(\frac{2\pi k}{N}\right)$—which are non-trivial to implement without floating-point math libraries. We explored approximations using Taylor series and considered precomputing and storing values in lookup tables, but both approaches required significant memory and processing overhead.

Ultimately, we chose a hybrid approach: retain hardcoded twiddle factors for small N, and focus on building the infrastructure to support runtime computation for larger N in future extensions. This compromise allowed us to focus on structural correctness and gradually build toward a more dynamic, generalized implementation.

This phased development strategy helped us isolate bugs more easily, trace internal data flow, and better understand the complex interplay between FFT components. While the final implementation remained scalar and fixed to a small N, it laid a robust foundation for future enhancements, including runtime generalization and vectorization.

III. Scalar FFT with Hardcoded Elements

Our first milestone was the successful implementation of a scalar 8-point FFT ($N = 8$) in RISC-V, using hardcoded twiddle factors and bit-reversal indices. Though not dynamically scalable, this implementation played a crucial role in verifying the correctness of our butterfly computation logic, memory access strategy, and floating-point arithmetic in the Venus simulator.

A. Manual Setup in the `.data` Section

We manually declared:

- Arrays `real` and `imag` for the real and imaginary parts of the input signal:

```
real: .float 0.0, 1.0, 2.0, 3.0, 4.0,
    5.0, 6.0, 7.0
imag: .float 0.0, 0.0, 0.0, 0.0, 0.0,
    0.0, 0.0, 0.0
```

- A precomputed bit-reversal index array for $N = 8$:

```
bitrev_indices: .word 0, 4, 2, 6, 1, 5,
    3, 7
```

This bit-reversal permutation is vital to the Cooley-Tukey algorithm, ensuring that input elements are reordered to enable efficient in-place computation.

B. Non-standard, Hardcoded Twiddle Factors

Twiddle factors (roots of unity) are typically complex exponentials of the form:

$$W_N^k = e^{-2\pi ik/N}$$

Instead of computing these dynamically, we hardcoded real and imaginary parts for the required twiddles:

```
twiddle_real: .float 1.0, 1.0, 0.0, 1.0,
    0.7071, 0.0, -0.7071
twiddle_imag: .float 0.0, 0.0, -1.0, 0.0,
    -0.7071, -1.0, -0.7071
```

These twiddle factors do not strictly follow the traditional exponential progression ($W_8^0, W_8^1, W_8^2, \ldots$). Instead, we manually selected twiddles relevant to the specific FFT stage:

- Stage 1 uses only $W_8^0 = 1$
- Stage 2 uses $W_8^0 = 1$, $W_8^2 = -1$
- Stage 3 uses more complex roots such as $W_8^1 = 0.7071 - 0.7071i$, $W_8^2 = -i$, etc.

This optimization allowed precise control over value usage and reduced runtime computations.

C. FFT Control Flow and Twiddle Factor Selection

Our FFT consisted of three stages (since $\log_2 8 = 3$). For each stage:

1) We manually set a twiddle base index:

```
beq s0, s8, stage1_twid    # stage == 1
      s7 = 0
beq s0, s9, stage2_twid    # stage == 2
      s7 = 1
beq s0, s10, stage3_twid   # stage == 4
      s7 = 3
```

2) Then, during the butterfly loop, the correct twiddle factor was selected:

```
add t4, s7, t0             # twiddle
    index = s7 + j
slli t4, t4, 2             # convert to
     byte offset
add t5, s4, t4             # real part
    address
add t6, s5, t4             # imag part
    address
flw ft4, 0(t5)             # load real
    part of W
flw ft5, 0(t6)             # load imag
    part of W
```

This indirect addressing simulated runtime twiddle selection using hardcoded data.

D. Butterfly Operation with Twiddles

The butterfly computation implemented the following logic:

```
# complex multiplication: odd * twiddle
fmul.s ft6, ft2, ft4       # odd_real *
    W_real
fmul.s ft7, ft3, ft5       # odd_imag *
    W_imag
fsub.s ft6, ft6, ft7       # real_part = (or
     * Wr) - (oi * Wi)

fmul.s ft7, ft2, ft5       # odd_real *
    W_imag
fmul.s ft8, ft3, ft4       # odd_imag *
    W_real
fadd.s ft7, ft7, ft8       # imag_part = (or
     * Wi) + (oi * Wr)
```

Then we updated the even and odd indices:

- Even index update:

```
fadd.s ft8, ft0, ft6       # even_real =
     er + real_part
fadd.s ft9, ft1, ft7       # even_imag =
     ei + imag_part
```

- Odd index update:

```
fsub.s ft10, ft0, ft6      # odd_real =
    er - real_part
fsub.s ft11, ft1, ft7      # odd_imag =
    ei - imag_part
```

This implementation adheres to the core FFT butterfly equations:

$$X_k = E_k + W_k \cdot O_k$$
$$X_{k+m/2} = E_k - W_k \cdot O_k$$

E. Why This Was Crucial

Despite being static, this scalar implementation served as a valuable unit test for:

- Bit-reversal correctness
- Memory layout interpretation in RISC-V
- Floating-point arithmetic and complex operations
- Correct twiddle-based selection and transformation logic

With this foundation, we had enough confidence to pursue more dynamic implementations, where twiddle factors and indices could be computed at runtime for arbitrary power-of-two N.

IV. Scalable Bit Reversal

Once the hardcoded version was validated, we transitioned toward a general-purpose, fully dynamic FFT pipeline. The first major component to be made runtime-configurable was the bit-reversal step, which reorders the input data before FFT computation to ensure that butterflies operate on the correct element pairs.

A. Bit-Reversal Index Computation

We implemented an algorithm that computes, for each index i from 0 to $N-1$, the bit-reversed version of i. This is essential for in-place FFT reordering. The method:

- Shifts bits from the original index i into a new register `reversed_i`, one by one.
- Uses shift and mask operations to isolate and reposition each bit.
- Dynamically stores each reversed index in memory (`bitrev_indices`) for later use.

This logic is scalable for any $N = 2^k$, and operates entirely at runtime.

Code Snippet: Bit-Reversal Index Generation

```
la a2, bitrev_indices     # Load array base
li t0, 0                  # i = 0
li t1, 8                  # N = 8

bitrev_compute_loop:
  mv t3, t0               # t3 = i
  li t2, 0                # reversed_i = 0
  li t4, 3                # log2(N) = 3 bits

reverse_bits_loop:
  slli t2, t2, 1          # shift reversed_i
     left
  andi t5, t3, 1          # get LSB of i
  or t2, t2, t5           # add it to
    reversed_i
  srli t3, t3, 1          # shift i right
  addi t4, t4, -1
  bnez t4, reverse_bits_loop

  sw t2, 0(a2)            # store reversed_i
  addi t0, t0, 1
  addi a2, a2, 4
  blt t0, t1, bitrev_compute_loop
```

At the end of this step, the `bitrev_indices` array for $N = 8$ contains:

$$[0, 4, 2, 6, 1, 5, 3, 7]$$

This means, for example, that index 1 maps to position 4, index 2 to 2, and so forth.

B. Bit-Reversal Permutation of Input Data

Using the dynamically computed indices, we rearranged the `real[]` and `imag[]` input arrays accordingly. To maintain correctness and avoid redundant swaps, we only performed swaps where $i <$ `bitrev_indices`$[i]$.

Code Snippet: Bit-Reversal Permutation

```
la a0, real
la a1, imag
la a2, bitrev_indices
li t0, 0                  # i = 0

bitrev_permute_loop:
  lw t1, 0(a2)            # reversed =
    bitrev_indices[i]
  bge t0, t1, skip_swap   # skip if already
    swapped

  # Swap real parts
  slli t2, t0, 2          # i offset
  slli t4, t1, 2          # reversed offset
  add t3, a0, t2          # real[i]
  add t5, a0, t4          # real[rev]
  flw ft0, 0(t3)
  flw ft1, 0(t5)
  fsw ft1, 0(t3)
  fsw ft0, 0(t5)

  # Swap imag parts
  add t3, a1, t2          # imag[i]
  add t5, a1, t4          # imag[rev]
  flw ft0, 0(t3)
  flw ft1, 0(t5)
  fsw ft1, 0(t3)
  fsw ft0, 0(t5)

skip_swap:
  addi a2, a2, 4
  addi t0, t0, 1
  li t6, 8
  blt t0, t6, bitrev_permute_loop
```

This ensured the data was correctly restructured before entering the butterfly stages. By performing conditional swaps, we avoided repeating or undoing previous swaps, preserving both performance and correctness.

C. Significance and Outcome

This scalar bit-reversal implementation:

- Validated correct handling of runtime memory access and bit-level manipulations.
- Enabled a dynamic pipeline suitable for arbitrary $N = 2^k$ values.
- Became foundational to our general FFT structure, ensuring reordered data was aligned for

correct butterfly computation and twiddle application.

Thus, what began as a seemingly simple reversal operation evolved into a scalable and essential component of our FFT engine, solidifying its generality and robustness.

V. Runtime Twiddle Factor Calculation

Twiddle factors $W_k^N = e^{-2\pi ik/N}$ are core components of the FFT algorithm. They represent complex roots of unity used to mix even and odd elements in each stage of the recursive decomposition. In high-level software environments with full floating-point and math libraries, computing these at runtime is trivial. However, implementing twiddle factor generation in RISC-V assembly—especially on systems without hardware support for floating-point trigonometric operations—presents significant challenges.

A. Approach

We implemented runtime calculation of twiddle factors by:

- Using polynomial approximations for $\sin(x)$ and $\cos(x)$ to evaluate the real and imaginary parts of the twiddle factors.
- Dynamically computing the angle $-\frac{2\pi j}{m}$ for each butterfly operation using floating-point arithmetic.
- Applying Horner's rule for numerically stable and efficient polynomial evaluation.

B. Challenges Faced

1) 1) No Hardware Trigonometric Instructions: RISC-V lacks built-in `sin()` and `cos()` instructions. We used polynomial approximations with hardcoded coefficients:

```
# Polynomial for cos(x)
cos_coeff_0: .float 2.44677067e-5
cos_coeff_1: .float -1.38877297e-3
cos_coeff_2: .float 4.16666567e-2
cos_coeff_3: .float -0.5
cos_coeff_4: .float 1.0
```

These were loaded into registers `fs0--fs4` (for cos) and `fs5--fs8` (for sin) at runtime:

```
flw fs0, 0(a3)    # c0 for cos
...
flw fs5, 0(a3)    # s0 for sin
```

2) 2) Dynamic Angle Computation: Each butterfly computation requires the angle $-\frac{2\pi j}{m}$. This was calculated as:

```
fcvt.s.w ft4, t0          # float(j)
fcvt.s.w ft5, s3          # float(m)
fdiv.s   ft6, ft4, ft5    # ft6 = j / m
li       t4, 0x40C90FDB   # 2 in IEEE-754
    float (approx)
fmv.s.x  ft7, t4
fmul.s   ft6, ft6, ft7    # angle = 2j/m
fneg.s   ft6, ft6         # angle = -2j/m
```

3) 3) Polynomial Trigonometric Evaluation: Given the computed angle in `ft6`, we evaluated $\cos(x)$ and $\sin(x)$ using nested multiplication and addition via Horner's rule.

Example: Cosine polynomial evaluation

$$\cos(x) \approx c_0x^8 + c_1x^6 + c_2x^4 + c_3x^2 + c_4$$

```
fmul.s ft4, ft6, ft6     # x
fmv.s  ft8, fs0          # c0
fmul.s ft8, ft8, ft4     # c0 * x
fadd.s ft8, ft8, fs1
fmul.s ft8, ft8, ft4
fadd.s ft8, ft8, fs2
fmul.s ft8, ft8, ft4
fadd.s ft8, ft8, fs3
fmul.s ft8, ft8, ft4
fadd.s ft8, ft8, fs4     # ft8 = cos(angle)
```

A similar set of operations was performed for $\sin(x)$.

C. Why This Was Hard

1) 1) Floating Point Precision Errors: Multiplying and adding small polynomial coefficients in limited precision often led to rounding errors, especially in deeper FFT stages.

2) 2) Complex Multiplication Logic: To compute $W \cdot \text{odd}$ where $W = \cos(\theta) + i\sin(\theta)$, we expanded as:

$$W \cdot Z = (\cos\theta \cdot a - \sin\theta \cdot b) + i(\cos\theta \cdot b + \sin\theta \cdot a)$$

This involved multiple `fmul.s`, `fadd.s`, and `fsub.s` instructions in careful sequence:

```
fmul.s ft9, ft8, ft2     # cos * real
fmul.s ft10, ft9, ft3    # sin * imag
fsub.s ft11, ft9, ft10   # result_real = cos
    *real - sin*imag
...
```

3) 3) Indexing and Register Management: Errors in computing j, m, or managing twiddle values silently broke the FFT output. Debugging required meticulous tracking of register usage and memory layout.

D. Successes and Limitations

For small FFT sizes ($N = 4$, 8), the polynomial approximation was sufficient and produced correct output.

- **Stage 1:** All twiddles $W = 1$
- **Stage 2:** $W = 1, -1$
- **Stage 3:** $W = 1, i, -1, -i$

These were approximated in real time using runtime trigonometric polynomial evaluation.

However, as $N \geq 16$, cumulative floating-point error and increased latency became significant. The implementation became impractical for larger sizes without:

- Higher-precision floating-point arithmetic,
- Precomputed twiddle lookup tables,
- Or hardware trigonometric support.

Nonetheless, the implementation demonstrated a complete, standalone method for computing twiddle factors dynamically within the constraints of scalar RISC-V ISA.

VI. Butterfly Computation

Twiddle factors $W_k^N = e^{-2\pi ik/N}$ are core components of the FFT algorithm. They represent the complex roots of unity that are used to mix the even and odd components of the signal during each butterfly operation. In essence, these factors perform the necessary phase rotations that allow the FFT to combine the data correctly when transforming the time-domain signal into its frequency-domain representation.

In high-level languages equipped with robust floating-point libraries, the computation of W_k^N is usually handled by built-in functions for sine and cosine, which readily yield the necessary real and imaginary components of each twiddle factor. However, when implementing the FFT in RISC-V assembly, we face considerable challenges. The architecture does not offer native trigonometric instructions, and the floating-point unit provides only a rudimentary set of operations such as `fmul.s`, `fadd.s`, and `fsub.s`. Consequently, we must compute the twiddle factors at runtime using polynomial approximations to evaluate the sine and cosine functions.

A. Approach Overview

To implement the butterfly computation and dynamically generate the twiddle factors, our approach comprised the following steps:

- **Polynomial Approximations:**
  Instead of directly computing sine and cosine using hardware functions (which are not available), we used polynomial approximations. We approximate $\cos(x)$ with a polynomial of the form:

  $$\cos(x) \approx c_0 x^8 + c_1 x^6 + c_2 x^4 + c_3 x^2 + c_4$$

  and $\sin(x)$ with:

  $$\sin(x) \approx s_0 x^7 + s_1 x^5 + s_2 x^3 + s_3 x$$

  where the coefficients $c_0, \ldots, c_4$ and $s_0, \ldots, s_3$ are precomputed and stored in memory.
- **Dynamic Angle Calculation:**
  For each butterfly operation within the FFT stages, the effective twiddle factor depends on the index j and the current group size m. The algorithm computes the angle as:

  $$\text{angle} = -\frac{2\pi j}{m}$$

  This value is calculated at runtime by converting j and m to floating-point numbers, performing the division j/m, then multiplying by 2π, and finally negating the result.
- **Horner's Rule for Polynomial Evaluation:**
  To ensure numerical stability and reduce the computational cost of evaluating high-degree polynomials, we used Horner's rule to evaluate the sine and cosine polynomials.

B. Detailed Implementation

```
# Compute angle = -2 * j / m
fcvt.s.w   ft4, t0          # ft4 = float(j)
fcvt.s.w   ft5, s3          # ft5 = float(m)
fdiv.s     ft6, ft4, ft5    # ft6 = j / m
li         t4, 0x40C90FDB   # Load IEEE-754
    representation for 2
fmv.s.x    ft7, t4          # ft7 = 2
fmul.s     ft6, ft6, ft7    # ft6 = (j/m) * 2
fneg.s     ft6, ft6         # ft6 = -2 * j/m
    (angle)

# Evaluate cosine polynomial: ft8 = cos(
    angle)
fmul.s   ft4, ft6, ft6      # ft4 = x^2,
    where x = angle
fmv.s    ft8, fs0           # ft8 = c0
fmul.s   ft8, ft8, ft4
fadd.s   ft8, ft8, fs1
fmul.s   ft8, ft8, ft4
fadd.s   ft8, ft8, fs2
fmul.s   ft8, ft8, ft4
fadd.s   ft8, ft8, fs3
fmul.s   ft8, ft8, ft4
fadd.s   ft8, ft8, fs4      # ft8 now holds
    cos(angle)

# Evaluate sine polynomial: ft5 = sin(angle)
fmv.s    ft5, fs5           # ft5 = s0
fmul.s   ft5, ft5, ft4
fadd.s   ft5, ft5, fs6
fmul.s   ft5, ft5, ft4
fadd.s   ft5, ft5, fs7
fmul.s   ft5, ft5, ft4
fadd.s   ft5, ft5, fs8
fmul.s   ft5, ft5, ft6      # ft5 now holds
    sin(angle)
```

C. Performing the Butterfly Operation

Once the twiddle factor is computed, we apply it to the ”odd” element in the butterfly pair. The butterfly operation involves a complex multiplication followed by an addition and a subtraction.

- **Complex multiplication:**

$$\text{temp\_real} = \text{odd\_real} \cdot W_{\text{real}} - \text{odd\_imag} \cdot W_{\text{imag}}$$
$$\text{temp\_imag} = \text{odd\_real} \cdot W_{\text{imag}} + \text{odd\_imag} \cdot W_{\text{real}}$$

- **Butterfly update:**

```
fadd.s  ft9,  ft0, ft6      # new_even_real =
     even_real + temp_real
fadd.s  ft10, ft1, ft7      # new_even_imag =
     even_imag + temp_imag
fsub.s  ft11, ft0, ft6      # new_odd_real =
    even_real - temp_real
fsub.s  ft12, ft1, ft7      # new_odd_imag =
    even_imag - temp_imag
```

Each butterfly operation is carefully sequenced with the correct update of indices and registers. This ensures that the results of the complex multiplication feed immediately into the summing and differencing operations required by the FFT.

D. Challenges and Debugging

- **Floating-Point Precision:** Polynomial approximations introduce slight errors, which can accumulate over multiple stages.
- **Register Management and Instruction Ordering:** Misordering leads to incorrect results; debugging is time-consuming.
- **Indexing Errors:** Errors in computing angles via loop indices propagate across FFT stages, necessitating meticulous testing.

While this approach worked for $N = 8$, scaling it to larger N requires additional optimization techniques like precomputed lookup tables or better approximation strategies.

VII. Challenges

During development, our team faced a spectrum of technical and organizational obstacles. Below, each major hurdle is described in detail, along with how it arose and the steps taken (or considered) to resolve it.

A. Work Division vs. Interdependency

Initial Plan:
We attempted to split the FFT into three main modules—bit-reversal, twiddle-factor computation, and butterfly implementation—assigning each module to a different teammate. In principle, this would parallelize effort: one person writes bit-reversal, another writes twiddles, and a third writes butterflies.

Why It Fell Apart:
Bit-Reversal is relatively straightforward in assembly (just shifts, masks, and conditional swaps). However, its output must feed directly into the butterfly module.

Twiddle-Factor Computation is mathematically heavy: computing $\cos$ and $\sin$ at runtime in pure RISC-V required either large lookup tables or implementing Horner polynomials. In either case, any small coefficient error would break butterfly results. As a result, the ”twiddle” coder often had to wait

for clarity on how the butterfly indexing worked, and vice versa.

Butterfly Implementation needs correct real-and-imag loading, uses the twiddle values, and writes back in the right permuted locations. A slight mismatch in the bit-reversal layout or a tiny trigonometric error caused the entire butterfly stage to fail.

Because these subcomponents are tightly coupled—bit-reversal changes how butterfly indices are computed, and butterfly operations depend on accurate twiddle values—none of us could work in true isolation. We found ourselves constantly reconciling "my code assumes X" with "your code produces Y." Eventually, we realized we needed to sit together for several sessions and co-design a minimal working end-to-end scalar FFT (for $N = 4$ or $N = 8$) before each person branched off into smaller tasks. Only after everyone understood the entire dataflow could we make incremental changes without breaking assumptions.

B. Twiddle Factor Precision

In a high-level language, calling `cos()` or `sin()` is trivial. In bare-metal RISC-V assembly, no hardware or standard library call exists for these functions. Our challenge was to compute

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

at runtime for every stage and every butterfly index k.

Polynomial Approximation (Horner's Method): We chose a 4th-degree polynomial to approximate $\cos(x)$ and a 3rd-degree polynomial for $\sin(x)$ over $x \in [-\pi, \pi]$. In assembly, Horner's evaluation looks like:

```
fmul.s   ft4, ft6, ft6    # x
fmv.s    ft8, fs0         # c0
fmul.s   ft8, ft8, ft4    # c0x
fadd.s   ft8, ft8, fs1    # +c1
fmul.s   ft8, ft8, ft4    # *x
fadd.s   ft8, ft8, fs2    # +c2
fmul.s   ft8, ft8, ft4    # *x
fadd.s   ft8, ft8, fs3    # +c3
fmul.s   ft8, ft8, ft4    # *x
fadd.s   ft8, ft8, fs4    # +c4   exact cos(
    x)
```

and similarly for $\sin(x)$.

Precision Pitfall at $\pm\pi/2$: Stage 2 ($m = 4$) requires

$$\cos(\pm\pi/2) = 0 \quad \text{and} \quad \sin(\pm\pi/2) = \pm 1.$$

Our polynomial gave $\sin(\pi/2) \approx 0.92$ or 0.95 instead of exactly 1.0. That small error (e.g., -0.92 instead of -1.0) caused the "odd × W" multiplication to produce ± 3.68 instead of ± 4.0. Once stage 2 is wrong, later stages compound the drift, yielding a completely incorrect final FFT.

Attempts to Fix:

- *Higher-Degree Polynomials:* We tried 6th- and 8th-degree Horner expansions. Although accuracy improved slightly, each extra multiply/add in floating-point assembly slowed the code by 20–30% and still didn't guarantee exact ± 1.0 at $\pi/2$.
- *Special-Case Branching (Hardcode Stage 2):* For $m = 4$, we loaded exact bit patterns for $\{0, 1.0\}$ or $\{0, -1.0\}$ from a tiny data table instead of calling the polynomial. That eliminated stage 2 error entirely but cluttered the code with branching logic.
- *Lookup Tables per Stage:* We ultimately replaced purely polynomial computation for stages $m = 4, 8, 16, \ldots$ with precomputed arrays of length $m/2$. Each stage's twiddle values were stored exactly in memory, so $\cos(2\pi j/m)$ and $\sin(2\pi j/m)$ required a single indexed load instead of polynomials. This solved precision completely but at the cost of a few hundred more floats in `.data`.

In summary, twiddle accuracy was our single biggest numerical challenge. Without a hybrid "small lookup table + polynomial fallback," the FFT result would drift noticeably by stage 4 or stage 5.

C. Floating-Point Arithmetic Complexity

Every butterfly step requires four complex additions/subtractions and four floating-point multiplies:

- Compute angle $\theta = -2\pi j/m$ using `fcvt.s.w`, `fdiv.s`, `fmul.s`, `fneg.s`.
- Evaluate $\cos(\theta)$ and $\sin(\theta)$ (4 multiplies + 4 adds each).
- Complex multiplication $(v_{\text{real}} + jv_{\text{imag}}) \times (\cos\theta - j\sin\theta)$ (2 `fmul.s` + 1 `fsub.s` for real part, 2 `fmul.s` + 1 `fadd.s` for imag part).

- Two more `fadd.s`/`fsub.s` to form ”even” and ”odd” outputs.

That easily totals 11–13 floating-point instructions per butterfly. Tracking all these intermediate results across registers `ft0...ft12` required scrupulous care. A single register reuse or forgetting to reload a value often resulted in ”garbage appears in the next stage.” We ended up inserting extra `fmv.s` or `fsgn.{d,s}` instructions just to save/restore temporaries, which ballooned our code size and slowed execution.

When we refactored to vector instructions, all those FP ops condensed to four vector operations:

```
vfmul.vv v6, v2, v4    # v_real * W_real
vfmul.vv v7, v3, v5    # v_imag * W_imag
vfsub.vv v8, v6, v7    # real part of (v  W)
vfmul.vv v6, v2, v5    # v_real * W_imag
vfmul.vv v7, v3, v4    # v_imag * W_real
vfadd.vv v9, v6, v7    # imag part of (v  W)
vfadd.vv v10, v0, v8   # new_even_real
vfadd.vv v11, v1, v9   # new_even_imag
vfsub.vv v12, v0, v8   # new_odd_real
vfsub.vv v13, v1, v9   # new_odd_imag
```

Even so, correctly aligning and indexing into vector registers introduced its own complexity.

VIII. Conclusion

In this project, we successfully implemented a fully functional FFT on RISC-V architecture, overcoming numerous technical and organizational challenges. Our iterative approach, from scalar to vectorized implementations, allowed us to carefully debug and optimize performance. Precision in twiddle factor calculations and careful management of floating-point operations were critical to achieving correct results.

Future work could explore further optimization techniques, including hardware-specific vector extensions and more advanced polynomial approximations, to improve speed and accuracy. Additionally, enhanced debugging tools and automated verification would streamline development.

We thank all team members for their dedication and collaboration throughout this challenging but rewarding project.