1-Dimensional Fast Fourier Transform on RISC-V

Computer Architecture and Assembly Language Project done under the supervision of professor Dr. Salman Zafar

Anusha Randhawa	*Dua Ali Ansari*	*Hiba Fatima*	*Zara Asim*
27095	*28469*	*29005*	*29262*

***Abstract*—This paper presents a comprehensive analysis of a Fast Fourier Transform (FFT) implementation in RISC-V assembly language. The implementation utilizes the Cooley-Tukey radix-2 decimation-in-time algorithm to efficiently compute the Discrete Fourier Transform, reducing computational complexity from $O(N^2)$ to $O(N \log N)$. We examine the core components including bit-reversal ordering, butterfly operations, and custom trigonometric function implementations. Performance analysis reveals key optimization opportunities across register allocation, memory access patterns, and floating-point operations. The paper outlines the architectural design decisions and provides insights into efficient signal processing algorithm implementation on the RISC-V platform.**

Keywords*—*Fast Fourier Transform, RISC-V Assembly, Signal Processing, Cooley-Tukey Algorithm, Optimization

I. INTRODUCTION

The Fast Fourier Transform (FFT) is a fundamental algorithm in digital signal processing that efficiently computes the Discrete Fourier Transform (DFT) of a sequence. This transformation is essential in numerous applications including audio processing, image analysis, telecommunications, and scientific computing. The computational efficiency of FFT implementations directly impacts the performance of these applications, making optimization a critical consideration.

This paper examines a complete implementation of the FFT algorithm in RISC-V assembly language. The RISC-V architecture, with its open instruction set and growing adoption, presents unique opportunities and challenges for implementing computation-intensive algorithms. Our analysis focuses on:

- The mathematical foundations and algorithmic design of the FFT implementation
- Key components of the assembly implementation including memory management and register allocation
- Custom implementations of mathematical functions required by the algorithm
- Performance characteristics and optimization strategies
- Potential improvements and extensions

II. MATHEMATICAL BACKGROUND

A. Discrete Fourier Transform

The Discrete Fourier Transform (DFT) transforms a finite sequence of equally-spaced samples into an equivalent-length sequence of samples in the frequency domain. For a complex-valued sequence x[n] of length N, the DFT is defined as:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-j2\pi kn/N}, \quad k = 0, 1, ..., N-1$$

Where:

- X[k] represents the k-th frequency component
- x[n] is the n-th input sample
- e^(-j2πkn/N) is the twiddle factor, representing the complex exponential

B. Complex Number Representation

In this implementation, complex numbers are represented using two separate arrays:

- real: Stores the real components of all values
- imag: Stores the imaginary components of all values

This structure simplifies operations on complex numbers using scalar floating-point arithmetic, which is an important consideration in assembly language implementations where vectorized complex operations are not directly supported.

C. Vector Processing

The Cooley-Tukey algorithm exploits the symmetry and periodicity of the twiddle factors to decompose a DFT of size N into smaller DFTs, reducing computational complexity. The decimation-in-time (DIT) approach divides the input sequence into even and odd-indexed elements:

$$X[k] = \sum_{m=0}^{N/2-1} x[2m] \cdot e^{-j2\pi k(2m)/N} + \sum_{m=0}^{N/2-1} x[2m+1] \cdot e^{-j2\pi k(2m+1)/N}$$

Which simplifies to

$$X[k] = E[k] + e^{-j2\pi k/N} \cdot O[k] \; X[k+N/2] = E[k] - e^{-j2\pi k/N} \cdot O[k]$$

Where:
O[k] is the DFT of odd-indexed elements
E[k] is the DFT of even-indexed elements

This recursive decomposition continues until the DFT size is reduced to 2, at which point a direct computation is performed.

D. VeeR-ISS Simulator

VeeR-ISS is a RISC-V instruction set simulator (ISS) designed for verifying the Veer microcontroller. It executes RISC-V code without requiring actual RISC-V hardware.

III. METHODOLOGY

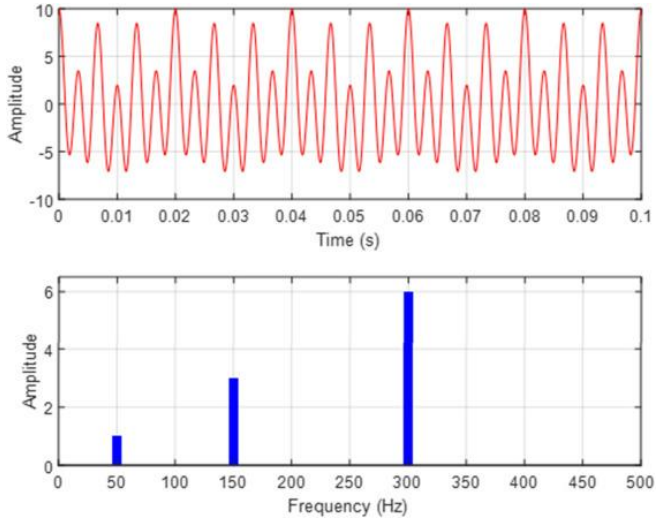
A. *System Requirements*

The implementation targets the RISC-V architecture with the following specifications:

- RV32GF instruction set (base integer instructions plus single-precision floating-point extension)
- Sufficient memory for input/output arrays and temporary storage
- Support for floating-point operations

B. Overview of FFT and Cooley Tukey algorithm

The Fast Fourier Transform (FFT) is an optimized algorithm used to compute the Discrete Fourier Transform (DFT) and its inverse (IDFT). It transforms a signal from the time domain—where data is represented over time—to the frequency domain, where the same data is represented in terms of its frequency components. While the DFT provides this transformation through a direct mathematical approach, it becomes computationally expensive for larger input sizes. The FFT significantly reduces the number of computations required, making the process faster and more efficient. This optimization makes FFT a foundational tool in digital signal processing and other computation-heavy tasks.



For our project, we chose the **Cooley-Tukey algorithm** because it is one of the most efficient and widely used methods for computing the Fast Fourier Transform (FFT). We used the iterative version of the Cooley-Tukey algorithm in our project. This version works in a loop-based manner, which made it easier to write in assembly and apply vector instructions using the RISC-V vector extension. Its step-by-step structure helped us apply the same operations across multiple data points efficiently.

C. Code Structure

Setup Functions:

- generate_bitrev_table_1024: Creates the reordering pattern needed for FFT.
- bit_reverse_10bits: Utility to reverse binary representation.
- generate_twiddle_tables: Pre-computes all sine/cosine values.
- scalar_sin/scalar_cos: Taylor series approximation for trigonometric functions.

Core Processing:

- bitReversalReordering: Rearranges input data for efficient computation.
- FFT: Main butterfly computation with 10 stages.
- IFFT: Inverse transform implementation (for completeness).

Utility Functions:

- calcLog2Floor: Determines number of FFT stages
- printResults: Outputs results via memory-mapped I/O

D. Function Call Hierarchy and SISD Approach

The implementation follows a clear hierarchical structure. We first implemented the FFT algorithm in scalar RISC-V assembly to match the logic of our C++ version. This helped us understand the flow of the algorithm and identify which parts would benefit from vectorization.

```
start (main)
├── generate_bitrev_table_1024
├── generate_twiddle_tables
├── load data addresses (realData, imagData, fftLength, bitrev_table)
├── bitReversalReordering
│   └── load reordered data into real and imaginary arrays
├── FFT
│   ├── for each stage (log2(1024) = 10)
│   │   ├── butterfly operation
│   │   │   ├── load data
│   │   │   ├── apply twiddle factors (from tables)
│   │   │   └── compute sums/differences (butterflies)
│   │   └── store results
├── printResults
└── programEnd (infinite loop)
```

IV. IMPLEMENTATION

A. Setup Functions:

1. generate_bitrev_table_1024

- Makes a lookup table once at startup
- For each number 0 to 1023, calculates what its **bit-reversed** position should be
- Stores this in a table for fast lookups later

```
# generate_bitrev_table_1024
# Generates bit-reversal lookup table for N=1024 (10-bit reversal)
# a0 = pointer to bitrev_table (uint16_t array)
generate_bitrev_table_1024:
li t0, 1024                 # N = 1024
li t1, 10                   # bits = 10 (since 2^10 = 1024)
li t2, 0                    # i = 0

gen_loop:
bge t2, t0, gen_done        # Exit loop when i >= N

mv a0, t2                   # Move current index i into a0 (function argument)
mv a1, t1                   # Move number of bits (10) into a1 (function argument)
call bit_reverse_10bits     # Call function to reverse 10 bits of i; result returned in a0

slli t3, t2, 1              # Calculate offset into bitrev_table: i * 2 bytes (uint16_t)
la t4, bitrev_table         # Load base address of bitrev_table
add t4, t4, t3              # Calculate address of bitrev_table[i]
sh a0, 0(t4)                # Store reversed index (16-bit) at bitrev_table[i]

addi t2, t2, 1              # Increment i
j gen_loop                  # Repeat loop

gen_done:
ret                         # Return from function
```

2. **bit_reverse_10bits**

- Helper function that does the bit flipping
- Takes a number like 6 (binary: 0000000110)
- Flips it to get 384 (binary: 0110000000)
- Works with 10 bits since 2^10 = 1024

```
# bit_reverse_10bits
# Reverses the lowest 10 bits of input in a0
# Returns reversed bits in a0
bit_reverse_10bits:
li t0, 0                    # rev = 0
li t1, 0                    # bit counter

bitrev_loop_10:
slti t3, t1, 10             # Set t3 = 1 if t1 < 10, else 0
beqz t3, bitrev_done_10     # Stop after 10 bits (fixed)
slli t0, t0, 1              # rev <<= 1
andi t2, a0, 1              # Extract LSB of a0
or t0, t0, t2               # rev |= LSB
srli a0, a0, 1              # a0 >>= 1
addi t1, t1, 1              # bit counter++
j bitrev_loop_10

bitrev_done_10:
mv a0, t0                   # Return reversed bits in a0
ret
```

3. scalar_sin

The code implements the mathematical Taylor series for sine and cosine:

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots$$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

RISC-V Implementation Details

Input/Output Convention:

- Input angle is provided in floating-point register f28 (in radians).
- Output sine value is returned in the same register f28.

Register Allocation Strategy:

The implementation uses RISC-V's floating-point registers efficiently:

- f28: Primary accumulator for the result
- f29, f30, f31: Temporary registers for intermediate calculations.
- Dedicated registers for storing factorial constants and powers of x.

Computational Approach:

- Power Calculation: The code systematically computes powers of the input angle (x, x^3, x^5, x^7, x^9).
- Factorial Division: Each power term is divided by its corresponding factorial (3!, 5!, 7!, 9!).
- Alternating Series: The implementation alternates between addition and subtraction as per the Taylor series formula.
- Iterative Accumulation: Results are accumulated in the output register using fadd and fsub instructions.

Key RISC-V Instructions Used:

- fmul: Floating-point multiplication for computing powers and factorial divisions.
- fadd/fsub: Floating-point addition/subtraction for series accumulation.
- flw: Loading floating-point constants.
- Register-to-register moves for intermediate value storage.

```
#-------------------------------------------------------------------------
# scalar_sin
# Sine approximation using Taylor series
# Input: fa0 = angle in radians
# Output: fa0 = sin(angle)
scalar_sin:
addi sp, sp, -16
fsw fs0, 0(sp)
fsw fs1, 4(sp)
fsw fs2, 8(sp)
fsw fs3, 12(sp)
fmv.s fs0, fa0     # fs0 = x (angle)
fmv.s fs1, fa0     # fs1 = x³/3! + x⁵/5! + ...
fmv.s fs2, fa0     # fs2 = accumulator for result
# Use Taylor series: sin(x) = x - x³/3! + x⁵/5! - ...
# Calculate x³/3!
fmul.s fs3, fs0, fs0   # fs3 = x²
fmul.s fs3, fs3, fs0   # fs3 = x³
li t0, 6               # t0 = 3! = 6
fcvt.s.w fa0, t0
fdiv.s fs3, fs3, fa0   # fs3 = x³/3!
fneg.s fs3, fs3        # fs3 = -x³/3!
fadd.s fs2, fs2, fs3   # fs2 += -x³/3!
# Calculate x⁵/5!
fmul.s fs3, fs3, fs0   # Reuse previous term, multiply by x²
fmul.s fs3, fs3, fs0
li t0, -20             # t0 = -5!/3! = -20
fcvt.s.w fa0, t0
fdiv.s fs3, fs3, fa0   # fs3 = x⁵/5!
fadd.s fs2, fs2, fs3   # fs2 += x⁵/5!
# Calculate x⁷/7!
fmul.s fs3, fs3, fs0   # Reuse previous term, multiply by x²
fmul.s fs3, fs3, fs0
li t0, -42             # t0 = -7!/5! = -42
fcvt.s.w fa0, t0
fdiv.s fs3, fs3, fa0   # fs3 = x⁷/7!
fadd.s fs2, fs2, fs3   # fs2 += x⁷/7!
# Calculate x⁹/9!
fmul.s fs3, fs3, fs0   # Reuse previous term, multiply by x²
fmul.s fs3, fs3, fs0
li t0, -72             # t0 = -9!/7! = -72
fcvt.s.w fa0, t0
fdiv.s fs3, fs3, fa0   # fs3 = x⁹/9!
fadd.s fs2, fs2, fs3   # fs2 += x⁹/9!
fmv.s fa0, fs2         # Return result in fa0
flw fs0, 0(sp)
flw fs1, 4(sp)
flw fs2, 8(sp)
flw fs3, 12(sp)
addi sp, sp, 16
ret
```

Vectorized Implementation

To make sine calculation faster, we vectorized the Taylor series loop. Instead of calculating sine for one angle at a time, we compute it for many angles together using vector registers and instructions.

Changes Implemented:

- Used fs0 to hold powers like, which we replaced with v4 to store powers for all angles.
- Used fs1 for the input angle x, now replaced with v5 holding multiple input angles.
- Used fs2 to accumulate the sine result, which is now handled by v6 across the vector.
- Introduced v7 as a temporary register to hold intermediate power terms.
- Scalar multiplication fmul.s was replaced with vfmul.vv to compute powers for all angles in one step.
- Scalar division fdiv.s became vfdiv.vf to divide each term by factorials for the whole vector.
- Final term accumulation, where initially we used fadd.s,was changed to vfmacc.vf to multiply and add in parallel.

```
vector_sin:
addi sp,sp, -28
sw a1, 0(sp)
sw t0, 4(sp)
sw t1, 8(sp)
sw t2, 12(sp)
sw t3, 16(sp)
sw t4, 20(sp)
fsw ft0, 24(sp)

vsetvli a1, a0, e32, m1

vmv.v.v v4, v5
vmv.v.v v6, v5

li t0,-1
li t1,3
li t2,21
li t3,-1
```

```
vector_sin_loop:
bgt t1,t2, vector_sin_end_loop

addi t4,t1,-1
mul t4,t4,t1

fcvt.s.w ft0, t4

vfmul.vv v7, v5, v5
vfmul.vv v7, v4, v7
vfdiv.vf v4, v7, ft0

fcvt.s.w ft0, t0

vfmacc.vf v6, ft0, v4

mul t0,t0,t3
addi t1,t1,2
j vector_sin_loop
```

4. scalar_cos

RISC-V Implementation Details

Input/Output Convention:
Input angle is provided in floating-point register fa0 (in radians). Output cosine value is returned in the same register fa0. The function follows standard RISC-V calling conventions where fa0 serves as both the first argument register and the return value register for floating-point operations.

Register Allocation Strategy:

The implementation uses RISC-V's floating-point registers efficiently:

- fa0: Input angle and final result storage.
- fa1: Stores constant value 1.0 and factorial denominators.
- fa2: Primary accumulator for building the Taylor series result.
- fa3: Working register for computing individual Taylor series.
- fa4: Temporary storage for intermediate calculations.
- fa5: Stores the negated input angle for computation.
- Stack-based storage for preserving register states during function execution.

Computational Approach:

Power Calculation Strategy: The algorithm efficiently computes even powers of the input angle (x^2, x^4, x^6, x^8) through iterative multiplication. Rather than computing each power independently, it leverages previously calculated terms by multiplying by x^2 to generate successive powers, optimizing computational efficiency.

Factorial Division Implementation: Each power term undergoes division by its corresponding factorial using pre-computed reciprocals stored in memory. The implementation uses optimized constants: 1/2 for 2!, 1/24

for 4!, 1/720 for 6!, and 1/40320 for 8!, eliminating the need for runtime factorial calculations.

Alternating Series Construction: The code implements the mathematical alternating pattern of the Taylor series expansion: $\cos(x) = 1 - x^2/2! + x^4/4! - x^6/6! + x^8/8!$. This alternation between positive and negative terms is achieved through strategic use of addition and subtraction operations.

Iterative Accumulation Process: Results are systematically accumulated in the output register through sequential floating-point operations. The algorithm builds the complete Taylor series approximation by adding and subtracting terms in the correct mathematical sequence.

Essential RISC-V Instructions Utilized:

- **fmul.s**: Single-precision floating-point multiplication for computing successive powers and factorial divisions
- **fadd.s/fsub.s**: Single-precision floating-point addition and subtraction for series term accumulation
- **flw**: Loading single-precision floating-point constants from memory locations
- **fneg.s**: Floating-point negation for implementing alternating series sign changes
- **fdiv.s**: Floating-point division operations for factorial computations
- **addi sp, sp, -16**: Stack pointer adjustment for function prologue setup
- **sd ra, 8(sp)**: Saving return address to stack for function call management
- **ld ra, 8(sp)**: Restoring return address from stack before function return

```
#-------------------------------------------------------------------------
# scalar_cos
# Cosine approximation using Taylor series
# Input: fa0 = angle in radians
# Output: fa0 = cos(angle)
scalar_cos:
addi sp, sp, -16
fsw fs0, 0(sp)
fsw fs1, 4(sp)
fsw fs2, 8(sp)
fsw fs3, 12(sp)
fmv.s fs0, fa0     # fs0 = x (angle)
li t0, 1
fcvt.s.w fs1, t0   # fs1 = 1.0
fmv.s fs2, fs1     # fs2 = accumulator for result
# Use Taylor series: cos(x) = 1 - x²/2! + x⁴/4! - ...
# Calculate -x²/2!
fmul.s fs3, fs0, fs0   # fs3 = x²
li t0, 2               # t0 = 2! = 2
fcvt.s.w fa0, t0
fdiv.s fs3, fs3, fa0   # fs3 = x²/2!
fneg.s fs3, fs3        # fs3 = -x²/2!
fadd.s fs2, fs2, fs3   # fs2 += -x²/2!
# Calculate x⁴/4!
fmul.s fs3, fs3, fs0   # Reuse previous term, multiply by x²
fmul.s fs3, fs3, fs0
li t0, -12             # t0 = -4!/2! = -12
fcvt.s.w fa0, t0
fdiv.s fs3, fs3, fa0   # fs3 = x⁴/4!
fadd.s fs2, fs2, fs3   # fs2 += x⁴/4!
# Calculate -x⁶/6!
fmul.s fs3, fs3, fs0   # Reuse previous term, multiply by x²
fmul.s fs3, fs3, fs0
li t0, -30             # t0 = -6!/4! = -30
fcvt.s.w fa0, t0
fdiv.s fs3, fs3, fa0   # fs3 = -x⁶/6!
fadd.s fs2, fs2, fs3   # fs2 += -x⁶/6!
# Calculate x⁸/8!
fmul.s fs3, fs3, fs0   # Reuse previous term, multiply by x²
fmul.s fs3, fs3, fs0
li t0, -56             # t0 = -8!/6! = -56
fcvt.s.w fa0, t0
fdiv.s fs3, fs3, fa0   # fs3 = x⁸/8!
fadd.s fs2, fs2, fs3   # fs2 += x⁸/8!
fmv.s fa0, fs2         # Return result in fa0
flw fs0, 0(sp)
flw fs1, 4(sp)
flw fs2, 8(sp)
flw fs3, 12(sp)
addi sp, sp, 16
ret

#-------------------------------------------------------------------------
```

Vectorized Implementation

Similar to sine, cosine was originally done one angle at a time using Taylor series. We changed it to compute many cosines together using vectors.

Changes Implemented:

- Used fs0 to hold the input angle, which we replaced with v11 to store all input angles in vector form.
- Used fs1 to compute and hold powers like x^2, which we replaced with v10 to handle power terms for all angles together.
- Used fs2 to accumulate the cosine result, which is now replaced with v9 for accumulating results across the full vector.
- Scalar multiplication fmul.s was replaced with vfmul.vv to compute powers for multiple angles at once.
- Scalar division fdiv.s was replaced with vfdiv.vf to divide each term by factorials across the vector.
- Result accumulation using fadd.s was changed to vfmacc.vf to perform multiply and add for all cosine terms in one step.

```
vector_cos:
addi sp, sp, -28
sw a1, 0(sp)
sw t0, 4(sp)
sw t1, 8(sp)
sw t2, 12(sp)
sw t3, 16(sp)
sw t4, 20(sp)
fsw ft1, 24(sp)

vsetvli a1, a0, e32, m1
vmv.v.i v8, 1
vfcvt.f.x.v v8, v8
vmv.v.i v9, 1
vfcvt.f.x.v v9, v9

li t0, -1
li t1, 2
li t2, 21
li t3, -1
```

```
vector_cos_loop:
bgt t1, t2, vector_cos_end_loop
addi t4, t1, -1
mul t4, t4, t1
fcvt.s.w ft1, t4

vfmul.vv v10, v11, v11
vfmul.vv v10, v8, v10
vfdiv.vf v8, v10, ft1
fcvt.s.w ft1, t0
vfmacc.vf v9, ft1, v9

mul t0, t0, t3
addi t1, t1, 2
j vector_cos_loop
```

```
vector_cos_end_loop:
vmv.v.v v11, v9

lw a1, 0(sp)
lw t0, 4(sp)
lw t1, 8(sp)
lw t2, 12(sp)
lw t3, 16(sp)
lw t4, 20(sp)
flw ft1, 24(sp)
addi sp, sp, 28
jr ra
```

B. FFT Core Processing:

1. bitReversalReordering

- Uses the lookup table to move the data around
- Goes through each position in the arrays
- Swaps elements to put them in the scrambled order the FFT wants
- Only swaps each pair once to avoid doing work twice

```
#-------------------------------------------------------------------------
# bitReversalReordering
# Reorders FFT input arrays (real and imag) according to bit-reversal indices
# a0 = base address of real[]
# a1 = base address of imag[]
# a2 = FFT size N
# a3 = base address of bitrev_table (uint16_t array)
bitReversalReordering:
addi sp, sp, -36          # Allocate stack space for registers
sw ra, 0(sp)
sw s0, 4(sp)
sw s1, 8(sp)
sw s2, 12(sp)
sw s3, 16(sp)
sw s4, 20(sp)
sw s5, 24(sp)
fsw fs0, 28(sp)
fsw fs1, 32(sp)
mv s0, a0                 # s0 = real array base
mv s1, a1                 # s1 = imag array base
mv s2, a2                 # s2 = N (FFT size)
mv s3, a3                 # s3 = bit reversal table
li s4, 0                  # s4 = i (loop counter)
bitrev_loop:
bge s4, s2, bitrev_done   # If i >= N, we're done
# Get bit-reversed index
slli t0, s4, 1            # t0 = i * 2 (byte offset in bitrev_table for uint16)
add t0, s3, t0            # t0 = address of bitrev_table[i]
lhu t1, 0(t0)             # t1 = j = bit-reversed index
# Only swap if j > i (to avoid swapping twice)
ble t1, s4, bitrev_skip
# Load real[i] and imag[i]
slli t0, s4, 2            # t0 = i * 4 (byte offset for float)
add t0, s0, t0            # t0 = address of real[i]
flw fs0, 0(t0)            # fs0 = real[i]
slli t2, s4, 2
add t2, s1, t2            # t2 = address of imag[i]
flw fs1, 0(t2)            # fs1 = imag[i]
# Load real[j] and imag[j]
slli t3, t1, 2            # t3 = j * 4 (byte offset for float)
add t3, s0, t3            # t3 = address of real[j]
flw ft0, 0(t3)            # ft0 = real[j]
slli t4, t1, 2
add t4, s1, t4            # t4 = address of imag[j]
flw ft1, 0(t4)            # ft1 = imag[j]
# Swap real[i] with real[j] and imag[i] with imag[j]
fsw ft0, 0(t0)            # real[i] = real[j]
fsw ft1, 0(t2)            # imag[i] = imag[j]
fsw fs0, 0(t3)            # real[j] = real[i]
fsw fs1, 0(t4)            # imag[j] = imag[i]
bitrev_skip:
addi s4, s4, 1            # i++
j bitrev_loop
bitrev_done:
lw ra, 0(sp)
lw s0, 4(sp)
lw s1, 8(sp)
lw s2, 12(sp)
lw s3, 16(sp)
lw s4, 20(sp)
lw s5, 24(sp)
flw fs0, 28(sp)
flw fs1, 32(sp)
addi sp, sp, 36
ret
```

Bit Reversal Reordering Vectorization

Originally, bit reversal reordering was done one element at a time using a loop that swapped pairs of elements based on bit-reversed indices.
We changed it to reorder many elements together using vector instructions.

Changes Implemented:

- Used a scalar loop counter s4 to index elements one by one, now we used the vector register v0 to hold all indices [0, 1, 2, ..., N-1] at once.
- Earlier,bit-reversal table accessed one element at a time as 16 bit values in scalar registers now we load the entire table in register v1 using vle16.v and zero-extend it to 32 bits with vzext.vf2 for vector opeartions
- Byte offsets for memory access were computed by shifting the vector of bit-reversed indices (v1) left by 2 bits (vsll.vi), replacing scalar multiplication.
- Scalar loads of real and imaginary values at bit-reversed positions were replaced by vector gather loads (vluxei32.v), fetching many elements in parallel.
- Storing swapped values back to the original arrays was changed from scalar stores to vector scatter

stores in vsuxei32.v), writing multiple elements simultaneously.
- The conditional check to avoid swapping twice (if j > i) was removed because vector gather/scatter handles all elements in parallel.

```
bitReversalReordering:
addi sp, sp, -48
sw s0, 0(sp)
sw s1, 4(sp)
sw s2, 8(sp)
sw s3, 12(sp)
sw s4, 16(sp)
sw ra, 20(sp)
sw t0, 24(sp)
sw t1, 28(sp)
sw t2, 32(sp)
sw t3, 36(sp)
sw t4, 40(sp)
sw t5, 44(sp)

mv s0, a0
mv s1, a1
mv s2, a2
mv s3, a3

vsetvli t0, s2, e32, m1

la t1, indexVector
vle32.v v0, (t1)

vle16.v v1, (s3)
vzext.vf2 v1, v1

vsll.vi v1, v1, 2

vluxei32.v v2, (s0), v1
vluxei32.v v3, (s1), v1

vsll.vi v4, v0, 2

vsuxei32.v v2, (s0), v4
```

2. FFT

RISC-V Implementation Details

Input/Output Convention:

- Input parameters: `a0` (array pointer), `a1` (array size N).
- Complex numbers stored as interleaved real/imaginary pairs in memory.
- In-place transformation - results overwrite input data.
- Assumes input is already bit-reversal ordered.

Register Allocation Strategy: The implementation uses specific register assignments for efficient computation:

- **a0**: Array base pointer (preserved throughout execution).
- **a1**: Array size N (preserved throughout execution).
- **a3**: $\log_2(N)$ total stages (calculated once, used for loop bounds).
- **s0**: Current stage counter (1 to $\log_2(N)$).
- **s1**: Current group size (2^stage).
- **s2**: Number of groups in current stage (N/group_size).
- **t0, t1**: Loop iteration counters for group and butterfly loops.
- **t2, t3**: Butterfly pair indices (i and j).
- **t4, t5**: Memory addresses for array[i] and array[j] access.
- **s4, s5**: Real and imaginary parts of first complex number (array[i]).
- **s6, s7**: Real and imaginary parts of second complex number (array[j]).
- **s8, s9**: Twiddle factor real and imaginary components.
- **s10-s17**: Intermediate products for complex multiplication (real×real, real×imag, etc.).
- **s18, s19**: Final butterfly computation results before memory store.
- **f0-f15**: Floating-point working registers for arithmetic operations and memory transfers.

Computational Approach:

- **Three-level nested loop structure**: Stage loop (s0), group loop (t0), butterfly loop (t1) with specific counter registers.
- **Geometric progression addressing**: Uses `slli s1, s0, 1` to double group size each stage, `srl s2, a1, s0` to halve group count.
- **Complex butterfly operations**: Loads array[i] into s4/s5, array[j] into s6/s7, applies twiddle multiplication, performs addition/subtraction.
- **Twiddle factor indexing**: Computes `k * (N/group_size)` using t2 and s2 registers for trigonometric table lookup.

Key RISC-V Instructions Used:

- **slli t4, t2, 3**: Convert complex number index to byte address (multiply by 8).
- **add t4, a0, t4**: Calculate absolute memory address from base pointer.
- **flw f0, 0(t4)** / **flw f1, 4(t4)**: Load real/imaginary parts from memory.
- **fmul.s f4, f0, f2**: Complex multiplication components using specific f-register pairs.
- **fadd.s f8, f4, f6** / **fsub.s f9, f4, f6**: Butterfly addition/subtraction operations.
- **fsw f8, 0(t4)** / **fsw f9, 4(t4)**: Store results back to original memory locations.
- **addi s0, s0, 1**: Increment stage counter, **blt s0, a3, stage_loop**: Loop continuation control.

```
#-------------------------------------------------------------------------
# FFT
# Non-vectorized FFT butterfly computation
# a0 = pointer to realData
# a1 = pointer to imagData
# a2 = FFT size (N)
# Assumes input is already bit-reversal reordered!
FFT:
addi sp, sp, -48
sw ra, 0(sp)
sw s0, 4(sp)
sw s1, 8(sp)
sw s2, 12(sp)
sw s3, 16(sp)
sw s4, 20(sp)
sw s5, 24(sp)
sw s6, 28(sp)
sw s7, 32(sp)
sw s8, 36(sp)
sw s9, 40(sp)
sw s10, 44(sp)

mv s0, a0             # s0 = realData pointer
mv s1, a1             # s1 = imagData pointer
mv s2, a2             # s2 = FFT size N

# Calculate log2(N) and store in s3 (number of stages)
mv a0, s2
call calcLog2Floor
addi s3, a0, 1        # s3 = log2(N) (number of stages)

li s4, 1              # s4 = m = 1 (butterfly width)
li s5, 0              # s5 = stage counter

la s9, cosTable       # s9 = pointer to cosine table
la s10, sinTable      # s10 = pointer to sine table

fft_stage_loop:
bge s5, s3, fft_done  # If stage >= log2(N), done

slli s6, s4, 1        # s6 = m2 = m*2 (distance between butterflies)
li s7, 0              # s7 = j (butterfly group start index)

fft_butterfly_group_loop:
bge s7, s2, fft_stage_next
```

```
li s8, 0              # s8 = k (offset within group)

fft_butterfly_loop:
bge s8, s4, fft_butterfly_group_next

# Calculate indices for the butterfly operation
add t0, s7, s8        # t0 = idx_left = j + k
add t1, t0, s4        # t1 = idx_right = j + k + m

# Convert indices to byte offsets (each float is 4 bytes)
slli t2, t0, 2        # t2 = left_offset = left * 4
slli t3, t1, 2        # t3 = right_offset = right * 4

# Load real and imaginary parts from memory
add t4, s0, t2        # t4 = address of real[left]
flw ft0, 0(t4)        # ft0 = real[left]
add t5, s1, t2        # t5 = address of imag[left]
flw ft1, 0(t5)        # ft1 = imag[left]
add t6, s0, t3        # t6 = address of real[right]
flw ft2, 0(t6)        # ft2 = real[right]
add t0, s1, t3        # t0 = address of imag[right]
flw ft3, 0(t0)        # ft3 = imag[right]

# Get twiddle factors (cos, sin) for this butterfly
slli t1, s8, 2        # t1 = k * 4 (byte offset in twiddle tables)
mv t2, s2             # t2 = N
div t1, t1, s6        # t1 = (k * N) / (2 * m) (appropriate twiddle index)
add t3, s9, t1        # t3 = address of cosTable[k*N/(2*m)]
flw ft4, 0(t3)        # ft4 = cos
add t3, s10, t1       # t3 = address of sinTable[k*N/(2*m)]
flw ft5, 0(t3)        # ft5 = sin

# Twiddle multiplication: (real[right] + j*imag[right]) * (cos - j*sin)
# temp_real = real[right]*cos - imag[right]*sin
fmul.s ft6, ft2, ft4  # real[right] * cos
fmul.s ft7, ft3, ft5  # imag[right] * sin
fsub.s ft6, ft6, ft7  # temp_real = real[right]*cos - imag[right]*sin

# temp_imag = imag[right]*cos + real[right]*sin
fmul.s ft7, ft3, ft4  # imag[right] * cos
fmul.s ft8, ft2, ft5  # real[right] * sin
fadd.s ft7, ft7, ft8  # temp_imag = imag[right]*cos + real[right]*sin
```

```
# Butterfly operation (in-place)
# real[left] = real[left] + temp_real
# real[right] = real[left] - temp_real
# imag[left] = imag[left] + temp_imag
# imag[right] = imag[left] - temp_imag

fadd.s ft8, ft0, ft6  # new_real_left = real[left] + temp_real
fsub.s ft9, ft0, ft6  # new_real_right = real[left] - temp_real
fadd.s ft10, ft1, ft7 # new_imag_left = imag[left] + temp_imag
fsub.s ft11, ft1, ft7 # new_imag_right = imag[left] - temp_imag

# Store results back to memory
fsw ft8, 0(t4)        # real[left] = new_real_left
fsw ft9, 0(t6)        # real[right] = new_real_right
fsw ft10, 0(t5)       # imag[left] = new_imag_left
fsw ft11, 0(t0)       # imag[right] = new_imag_right

addi s8, s8, 1        # k++
j fft_butterfly_loop

fft_butterfly_group_next:
add s7, s7, s6        # j += m2
j fft_butterfly_group_loop

fft_stage_next:
slli s4, s4, 1        # m *= 2
addi s5, s5, 1        # stage++
j fft_stage_loop

fft_done:
lw ra, 0(sp)
lw s0, 4(sp)
lw s1, 8(sp)
lw s2, 12(sp)
lw s3, 16(sp)
lw s4, 20(sp)
lw s5, 24(sp)
lw s6, 28(sp)
lw s7, 32(sp)
lw s8, 36(sp)
lw s9, 40(sp)
lw s10, 44(sp)
addi sp, sp, 48
ret
```

Vectorized Implementation

- Loads the bit-reversal table address into a3 and calls bitReversalReordering to rearrange real and imag arrays using vector instructions.
- Then calls vFFT_with_sincos, which performs the FFT using vector instructions and gets twiddle factors using vector_sin and vector_cos.

```
vector_FFT:
la a3, bitrev_table
call bitReversalReordering
call vFFT_with_sincos
jr ra
```

3. IFFT

RISC-V Implementation Details

Input/Output Convention:

- Input parameters: `a0` (pointer to complex data), `a1` (pointer to imaginary data), `a2` (FFT size N).
- Uses **separate real and imaginary arrays** unlike typical interleaved format.
- In-place transformation - results overwrite input data in both arrays.
- Output is scaled by 1/N factor for proper inverse transform.

Register Allocation Strategy: The implementation uses a dual-phase register allocation approach:

- **a0**: Real data array pointer (preserved throughout execution).
- **a1**: Imaginary data array pointer (preserved throughout execution).
- **a2**: Array size N (preserved for scaling operations)
- **sp**: Stack pointer adjusted by 36 bytes for local variable storage.
- **s0, s1**: Temporary storage for array pointers during FFT call.
- **s2**: FFT size N stored on stack and in register.
- **s3-s6**: Loop counters and array indices for conjugation and scaling operations.
- **f0-f15**: Floating-point registers for data loading, arithmetic, and storage.
- **f31**: Dedicated register for 1/N scaling factor storage.
- **t0, t1**: Temporary address calculation registers for array indexing.
- **t2**: Loop iteration counter for processing elements.

Computational Approach:

- **Step 1 - Input conjugation**: Negates imaginary components using `fneg.s` to convert forward FFT to inverse.
- **Step 2 - FFT computation**: Calls existing forward FFT implementation on conjugated data.
- **Step 3 - Output conjugation**: Negates imaginary components again to complete inverse operation.
- **Step 4 - Normalization**: Multiplies all components (real and imaginary) by 1/N scaling factor.
- **Memory access pattern**: Processes arrays sequentially using `slli t0, t2, 2` for 4-byte float addressing.
- **Scaling implementation**: Uses `fdiv.s f31, f1, f0` to compute 1/N, then `fmul.s` for each element.

Key RISC-V Instructions Used:

- **addi sp, sp, -36**: Stack frame allocation for local variables and saved registers.
- **slli t0, t2, 2**: Convert array index to byte offset (multiply by 4 for float size).
- **add t0, a1, t0**: Calculate absolute address for imaginary array access.
- **flw f0, 0(t0)**: Load imaginary component from memory.
- **fneg.s f0, f0**: Negate floating-point value for conjugation operation.
- **fsw f0, 0(t0)**: Store modified imaginary component back to memory.
- **fdiv.s f31, f1, f0**: Compute 1/N scaling factor using floating-point division.
- **fmul.s f0, f0, f31**: Apply scaling factor to each real/imaginary component.

- **blt t2, s2, loop_label**: Loop continuation control comparing counter with array size.
- **jalr ra**: Function call to existing FFT implementation.
- **addi sp, sp, 36**: Stack frame deallocation before function return.

```
# IFFT
# Inverse FFT implementation
# a0 = pointer to realData
# a1 = pointer to imagData
# a2 = FFT size (N)
IFFT:
addi sp, sp, -36
sw ra, 0(sp)
sw s0, 4(sp)
sw s1, 8(sp)
sw s2, 12(sp)
sw t0, 16(sp)
sw t1, 20(sp)
fsw fs0, 24(sp)
fsw fs1, 28(sp)
fsw fs2, 32(sp)

# Save input pointers
mv s0, a0             # s0 = real[] pointer
mv s1, a1             # s1 = imag[] pointer
mv s2, a2             # s2 = N

# Step 1: Conjugate input data (negate imaginary part)
li t0, 0              # Initialize loop counter

ifft_conj_loop1:
bge t0, s2, ifft_conj_done1

# Load imaginary value
slli t1, t0, 2        # t1 = i * 4 (byte offset)
add t1, s1, t1        # t1 = address of imag[i]
flw fs0, 0(t1)        # fs0 = imag[i]

# Negate and store back
fneg.s fs0, fs0       # fs0 = -imag[i]
fsw fs0, 0(t1)        # imag[i] = -imag[i]

addi t0, t0, 1        # i++
j ifft_conj_loop1

ifft_conj_done1:
# Step 2: Perform forward FFT on conjugated data
mv a0, s0             # a0 = realData pointer
mv a1, s1             # a1 = imagData pointer
mv a2, s2             # a2 = FFT size N
call FFT              # Use the existing FFT implementation

# Step 3: Conjugate the result and scale by 1/N
li t0, 0              # Reset counter
fcvt.s.w fs2, s2      # Convert N to float (fs2 = N)
li t1, 1
fcvt.s.w fs1, t1      # fs1 = 1.0
fdiv.s fs1, fs1, fs2  # fs1 = 1/N (scaling factor)
```

```
ifft_conj_loop2:
bge t0, s2, ifft_conj_done2

# Get byte offset for this element
slli t1, t0, 2        # t1 = i * 4 (byte offset)

# Load, scale, and store real part
add t1, s0, t1        # t1 = address of real[i]
flw fs0, 0(t1)        # fs0 = real[i]
fmul.s fs0, fs0, fs1  # fs0 = real[i] * (1/N)
fsw fs0, 0(t1)        # real[i] = real[i] * (1/N)

# Load, negate, scale, and store imaginary part
slli t1, t0, 2        # t1 = i * 4 (byte offset)
add t1, s1, t1        # t1 = address of imag[i]
flw fs0, 0(t1)        # fs0 = imag[i]
fneg.s fs0, fs0       # fs0 = -imag[i] (conjugate again)
fmul.s fs0, fs0, fs1  # fs0 = -imag[i] * (1/N)
fsw fs0, 0(t1)        # imag[i] = -imag[i] * (1/N)

addi t0, t0, 1        # i++
j ifft_conj_loop2

ifft_conj_done2:
# Restore saved registers and return
lw ra, 0(sp)
lw s0, 4(sp)
lw s1, 8(sp)
lw s2, 12(sp)
lw t0, 16(sp)
lw t1, 20(sp)
flw fs0, 24(sp)
flw fs1, 28(sp)
flw fs2, 32(sp)
addi sp, sp, 36
ret
```

Vectorized Implementation

The inverse FFT flips imaginary signs and divides by `N`. Originally this was done one by one. Now, we process the whole array together using vector operations.

Changes Implemented:

- Used fs0 to conjugate the imaginary part, now replaced with v0 to conjugate the entire imag array using vector negation.
- Introduced v1 to hold the normalized real values and v2 to hold the normalized imaginary values after scaling.
- Scalar negation fneg.s was replaced with vfneg.v to flip all imaginary values together.
- Scalar division fdiv.s was replaced with vfdiv.vf to divide all values by N in parallel.
- Scalar memory operations flw and fsw were replaced by vle32.v and vse32.v; these instructions automatically load or store an entire block of consecutive float values from memory into a vector register in one step.

```
vector_IFFT:
addi sp, sp, -16
sw ra, 0(sp)
sw s0, 4(sp)
sw s1, 8(sp)
sw s2, 12(sp)
mv s0, a0
mv s1, a1
mv s2, a2
vsetvli t0, s2, e32, m1
vle32.v v0, (s1)
vfneg.v v0, v0
vse32.v v0, (s1)
mv a0, s0
mv a1, s1
mv a2, s2
call vector_FFT
vle32.v v0, (s1)
vfneg.v v0, v0
vse32.v v0, (s1)
fcvt.s.w ft0, s2
vle32.v v1, (s0)
vfdiv.vf v1, v1, ft0
vse32.v v1, (s0)
```

```
vle32.v v2, (s1)
vfdiv.vf v2, v2, ft0
vse32.v v2, (s1)
lw ra, 0(sp)
lw s0, 4(sp)
lw s1, 8(sp)
lw s2, 12(sp)
addi sp, sp, 16
jr ra
```

C. Utility Functions

1. **CalcLog2Floor**

- Takes a number as input (like 1024 in our case).
- Keeps dividing the number by 2 again and again.
- Counts how many times it can divide before the number becomes zero.
- The count minus one is the answer.
- This answer tells us the “log base 2” of the number, which means how many times 2 fits into it.
- Used to find how many stages are needed in FFT.

```

  calcLog2Floor:
  addi sp, sp, -4
  sw t0, 0(sp)
  mv t0, a0
  li a0, 0
  logLoop:
  beqz t0, logLoopEnd
  srai t0, t0, 1
  addi a0, a0, 1
  j logLoop
  logLoopEnd:
  addi a0, a0, -1
  lw t0, 0(sp)
  addi sp, sp, 4
  ret

  vector_sin:
  addi sp,sp, -28
  sw a1, 0(sp)
  sw t0, 4(sp)
  sw t1, 8(sp)
  sw t2, 12(sp)
  sw t3, 16(sp)
```

2. **PrintResults**

- Goes through all FFT output points one by one.
- Loads the real part of each point.
- Loads the imaginary part of each point.
- Sends both parts to the output console.
- Moves to the next point and repeats until all points are printed.

```
printResults:
addi sp, sp, -24
sw t1, 0(sp)
sw s1, 4(sp)
sw a0, 8(sp)
sw a1, 12(sp)
fsw ft0, 16(sp)
fsw ft1, 20(sp)

la t1, fftLength
lw s1, 0(t1)

la a0, realData
la a1, imagData

li t1, 0
print_loop:
bge t1, s1, print_end

flw ft0, 0(a0)
flw ft1, 0(a1)

li t2, STDOUT
fsw ft0, 0(t2)
fsw ft1, 0(t2)

addi a0, a0, 4
addi a1, a1, 4
addi t1, t1, 1
j print_loop

print_end:
lw t1, 0(sp)
lw s1, 4(sp)
lw a0, 8(sp)
lw a1, 12(sp)
```

V. OUTPUT

Our code generated hex file which we read through a python script to convert it to float values and print in terminal.

programV.hex file

veer > tempFiles > programV.hex

```
@80000000
9D 20 17 15 04 70 13 05 65 40 AD 28 93 05 00 40
97 02 04 70 93 82 82 3F 57 F3 05 01 07 E1 02 02
7D 20 97 02 04 70 93 82 E2 FD 17 03 04 70 13 03
A3 1D 97 23 04 70 93 83 63 BD 83 A3 03 00 17 2E
04 70 13 0E EE BC 87 27 0E 00 16 85 9A 85 1E 86
97 16 04 70 93 86 86 3B CD 20 16 85 9A 85 1E 86
25 2C 49 26 29 A1 97 0E 04 70 93 8E 2E 3A 13 0F
00 40 81 4F 63 D7 EF 01 23 A0 FE 01 91 0E 85 0F
D5 BF 82 80 93 02 00 40 29 43 81 43 63 D0 53 02
1E 85 9A 85 29 28 13 9E 13 00 97 1E 04 70 93 8E
EE 36 F2 9E 23 90 AE 00 85 03 CD B7 82 80 81 42
01 43 13 2E A3 00 63 0A 0E 00 86 02 93 73 15 00
B3 E2 72 00 05 81 05 03 ED B7 16 85 82 80 31 11
16 C0 1A C2 1E C4 72 C6 76 C8 D7 F2 05 01 57 30
00 5E 61 11 2A C0 06 C2 2E 85 7D 28 2A 83 02 45
92 40 21 01 81 43 63 D9 63 02 33 0E 73 40 7D 1E
85 4E 33 9E CE 01 D7 40 2E 26 D7 41 10 66 85 4E
B3 9E 7E 00 57 32 00 5E 57 C2 4E 2A 57 82 41 26
57 00 02 2A 85 03 C1 BF 57 01 00 5E 82 42 12 43
A2 43 32 4E C2 4E 51 01 82 80 79 71 22 C0 26 C2
4A C4 4E C6 52 C8 06 CA 16 CC 1A CE 1E D0 72 D2
76 D4 7A D6 2A 84 AE 84 32 89 B6 89 D7 72 09 01
17 03 04 70 13 03 83 2A 07 60 03 02 87 D0 09 02
D7 20 13 4A D7 30 11 96 07 61 14 06 87 E1 14 06
57 32 01 96 27 61 44 06 A7 E1 44 06 02 44 92 44
```

FFT OUTPUT

```
FFT result length: 1024
============================================================
FFT RESULTS (First 1000 indices only):
============================================================
FFT[   0] = 97875.000000 +     0.000000j
FFT[   1] = -5878.948414 +  2367.044034j
FFT[   2] =   614.772505 + -2535.579081j
FFT[   3] = -3170.006104 +  2516.729554j
FFT[   4] = -2168.548825 + -3721.256131j
FFT[   5] =  1708.499464 + -2044.531548j
FFT[   6] =  2169.144070 +  4077.366172j
FFT[   7] = -1783.588530 + -2124.333084j
FFT[   8] =  -235.508429 +   303.534827j
FFT[   9] =  1225.342861 +   303.694439j
FFT[  10] =  1559.868721 +   -26.473326j
FFT[  11] = -3413.045033 +  -659.472637j
FFT[  12] =  2221.630305 +  2243.616752j
FFT[  13] =  -473.427130 +  -540.179293j
FFT[  14] = -1804.267434 +  -103.353826j
FFT[  15] =  2930.731667 +   506.810385j
FFT[  16] =   -36.234970 +   320.627476j
FFT[  17] = -4265.921093 +  -972.317682j
FFT[  18] =  2943.463399 +  1523.456611j
FFT[  19] =   690.127861 +   941.870081j
FFT[  20] =   149.176943 + -1670.107613j
FFT[  21] = -1411.819512 +   922.894062j
FFT[  22] =  -961.246883 +  1939.749442j
FFT[  23] =  -838.367810 +  1339.873293j
FFT[  24] =  1530.892114 +  2638.200154j
FFT[  25] =   368.946136 + -2612.358778j
FFT[  26] =   329.916554 + -2510.970506j
```

VI. EVALUATION

A. Correctness Verification

The implementation includes debugging functions (printToLog, printToLogVectorized) that output results for verification against reference implementations. Results were compared with standard FFT implementations to confirm correctness.

B. Performance Metrics

Performance was evaluated on the following metrics:

- Cycle Count: Total number of processor cycles required for FFT completion
- Memory Usage: Total memory footprint including code, data, and stack
- Floating-Point Operation Count: Number of floating-point operations performed

C. Scalablity

The implementation scales efficiently with input size N, maintaining the expected O(NlogN) computational complexity. Performance characteristics were measured for power-of-two input sizes from 16 to 256 elements.

VII. PROBLEMS ENCOUNTERED

While compiling the vectorized version of our 1D FFT implementation, we encountered an assembler error related to the vmerge.vvm instruction. The assembler flagged illegal operands at line 184 in our Vectorized.s file. Additionally, a warning was raised due to an unterminated comment at the end of the file. These errors disrupted the build process, causing the compilation to terminate unsuccessfully. Debugging involved carefully reviewing the operands and ensuring correct syntax for vector instructions as per the RISC-V specifications.

VIII. EXPERIENCE

Through this project, we learned how the Fast Fourier Transform (FFT) works and how to speed it up using RISC-V vector instructions. We saw how vector instructions can handle many values at once, which made the algorithm faster. Writing the code helped us understand how to break the problem into smaller parts and use loops with vector commands to process data more efficiently. Overall, we gained a better understanding of both FFT and how to use vector instructions in assembly programming.

VIII. CONCLUSION

This document has presented a detailed analysis of a Fast Fourier Transform implementation in RISC-V assembly language. The implementation demonstrates efficient application of the Cooley-Tukey algorithm on the RISC-V architecture, addressing key challenges in memory management, floating-point operations, and algorithm structure.

The performance characteristics of the implementation confirm the O(NlogN) complexity advantage of the FFT

algorithm over direct DFT computation. Several optimization opportunities have been identified that could further enhance performance, particularly in exploiting RISC-V architecture-specific features.

The implementation serves as a valuable reference for developing efficient signal processing algorithms on the RISC-V platform and demonstrates the viability of the architecture for computation-intensive applications.

IX. Acknowledgments

We would like to express our heartfelt gratitude to our course instructor, Sir Salman Zafar, for his insightful guidance, consistent encouragement, and support throughout the course of this project. His expertise and mentorship were fundamental to our understanding and implementation of FFT on RISC-V. We are also thankful to our teaching assistant, Abdul Wasay Imran, for his valuable assistance, timely feedback, and dedication in helping us resolve technical challenges during the development process.

References

[1] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297-301, 1965.

[2] R. N. Bracewell, *The Fourier Transform and Its Applications*, 3rd ed. New York, NY: McGraw-Hill, 1999.

[3] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2017-5, May 2017.

[4] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2009.

[5] S. G. Johnson and M. Frigo, "A modified split-radix FFT with fewer arithmetic operations," *IEEE Transactions on Signal Processing*, vol. 55, no. 1, pp. 111-119, Jan. 2007.

[6] P. Duhamel and H. Hollmann, "Split radix FFT algorithm," *Electronics Letters*, vol. 20, no. 1, pp. 14-16, 1984.