# Fast Fourier Transform Algorithm in RISC-V Using Vector Extension

Areesha Fasih 29007, Marium Ali Lakhani 29183, Fatima Faisal 28989, Khadeja Qureshi 27200,
a.fasih.29007@khi.iba.edu.pk, m.lakhani.29183@khi.iba.edu.pk , f.faisal.28989@khi.iba.edu.pk,
k.qureshi.27200@khi.iba.edu.pk

***Abstract*—This project presents the implementation of the Fast Fourier Transform (FFT) and its inverse (IFFT) using RISC-V vector assembly language, optimized for performance on large datasets. The algorithm operates on arrays of 1024 floating-point complex samples and utilizes precomputed twiddle factors for efficient butterfly computations. Key features include a vectorized bit-reversal permutation, stage-wise radix-2 butterfly operations, and optional normalization for inverse transforms. Leveraging the RISC-V Vector Extension (RVV), the design exploits data-level parallelism to accelerate computation. This low-level implementation demonstrates both the feasibility and performance potential of signal processing applications on vector-enabled RISC-V architectures.**

## I. Introduction

IN this project for our Computer Architecture and Assembly Language course, we implemented a high-level algorithm and translated it into low-level RISC-V assembly code using the Vector Extension (RVV). The primary objective was to explore and utilize vector instructions to optimize performance and parallelism. After writing and debugging the RISC-V vector code, we tested it using multiple simulators configured specifically for this task. The report provides a comprehensive explanation of the algorithm, details the translation process from high-level code to RISC-V vector assembly, and presents the results and observations obtained through simulation.

## II. Materials and Methods.

The Fast Fourier Transform algorithm is based on the principles of Fourier analysis, which focuses on expressing signals through their frequency components. The core of the FFT involves leveraging the properties of complex exponentials and the Discrete Fourier Transform to efficiently determine the Fourier transformation of discrete sampled data. The DFT represents a limited sequence of complex values as a combination of sinusoidal functions—sines and cosines—at varying frequencies, making it possible to examine the frequency makeup of a given signal. The FFT improves upon this process by dividing the DFT into smaller, manageable subproblems and utilizing symmetries and periodicity within the Fourier transform. This approach significantly reduces the computational effort compared to evaluating the DFT directly. As a result of this improved efficiency of O(NlogN) from $O(N^2)$, the FFT has become an essential component in a range of scientific and engineering disciplines, supporting fast and effective signal and data analysis for applications in areas such as audio and image processing, scientific computation, and telecommunications.

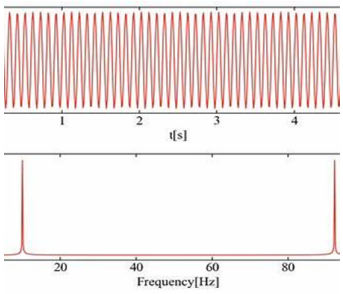


Fig. 1. Illustration of the Fast Fourier Transform (FFT) process, which transforms a time-domain signal into the frequency domain to analyze its spectral content. Cooley-Tukey FFT algorithm transformation of a sine wave from the time domain to its frequency components, highlighting distinct frequency peaks.

We selected the Cooley Tukey variant of the Fast Fourier Transform for our implementation because it is highly compatible with vector instructions, which streamlines the translation process and aligns well with our project requirements. Opting for the iterative version further simplified the overall design and coding effort. Specifically, this approach utilizes the Radix 2 Decimation In Time (DIT) Cooley Tukey FFT algorithm, which is well suited for efficiently calculating the Discrete Fourier Transform and its inverse for signals where the length is a power of two.

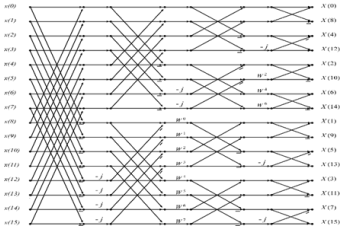

Fig. 2. Radix 2 Fast Fourier Transform (FFT) butterfly computation

The method incorporates several fundamental steps, including the use of bit reversal permutations to reorder the input

sequence as shown in figure three. This is followed by a series of butterfly operations, where pairs of elements are combined to generate the transformed outputs. The butterfly computation is crucial for improving computational efficiency, as it requires fewer multiplicative instructions than a naive approach. Nevertheless, the algorithm still involves complex multiplications when determining the so called roots of unity, since sine and cosine values are essential to calculate these twiddle factors. By leveraging these structures, the Cooley Tukey algorithm achieves both high efficiency and a straightforward translation to vectorized assembly code, making it an ideal choice for our objectives.

### KEY STEPS OF THE ALGORITHM

1) **Bit-Reversal Permutation:**
   Before the main FFT computation, the input sequence undergoes a reordering process known as bit-reversal permutation. In this step, the indices of the input array are reversed in binary form, effectively rearranging the data to facilitate in-place computation during subsequent stages.
2) **Radix-2 Decimation-in-Time Butterfly Computations with Twiddle Factors:**
   The FFT algorithm uses a Radix-2 Decimation-in-Time (DIT) approach, which recursively divides the DFT into smaller DFTs of size 2. At each stage, pairs of inputs are combined using butterfly operations, involving addition and subtraction of complex numbers followed by multiplication with precomputed twiddle factors (complex exponentials). This structured decomposition reduces the overall computational complexity from $O(N^2)$ to $O(N \log N)$.
3) **Normalization:**
   After completing all butterfly stages, the output is usually normalized by dividing by the input sequence length, ensuring the FFT and its inverse are properly scaled.

**Advantages:**

- **Computational Efficiency:**
  The Radix-2 DIT FFT reduces the computational complexity from $O(N^2)$ in the native DFT approach to $O(N \log N)$, enabling faster processing of large datasets.
- **In-Place Computation:**
  The algorithm's structure allows for in-place computation, minimizing memory usage by overwriting input data with output results during processing.
- **Hardware Compatibility:**
  Its iterative nature and reliance on simple arithmetic operations make it well-suited for implementation on hardware platforms that support vectorized instructions, enhancing performance in real-time applications.

### FFT, IFFT AND BIT-REVERSAL

The implementation of the Fast Fourier Transform (FFT) and its inverse (IFFT) on the RISC-V architecture is fundamentally based on the core FFT and IFFT equations. The FFT algorithm transforms a signal into its frequency components by recursively breaking down the Discrete Fourier Transform (DFT) into smaller DFTs, thereby reducing computational complexity from $O(N^2)$ to $O(N \log N)$. This decomposition allows for efficient computation through complex additions and multiplications, making it well-suited for RISC-V's parallel processing capabilities.

$$x[k] = \sum_{n=0}^{N-1} x[n] e^{\frac{-j2\pi kn}{N}}$$

Fig. 3. TWiddle factor computation

Similarly, the IFFT is used to reconstruct the original signal from its frequency representation. It involves similar operations to the FFT but with two key differences: the sign of the imaginary part is reversed, and the result is scaled by the reciprocal of the total number of points.

*Bit-Reversal Permutation Example*

In bit-reversal permutation, the index of each element in the input array is represented in binary, and then the bits are reversed to find the new index position.

For example, consider an array of length 8. The indices go from 0 to 7, whose 3-bit binary representations are:

| Decimal Index | Binary (3-bit) | Bit-reversed Binary | New Index (De |
|---|---|---|---|
| 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 |

This reordering allows the FFT algorithm to efficiently compute in-place without extra memory.

## III. CODE

*Data Initialization in Vectorized Code*

The initial section of the vectorized FFT implementation sets up essential data structures and constants:

*.data Section Initialization*

```
.section .data
real:
    .rept 1024
    .float 1.0
```

```
    .endr

imag:
    .rept 1024
    .float 0.0
    .endr
N:          .word 1024
logN:       .word 10
```

- `real[1024]` and `imag[1024]` store the input signal's real and imaginary parts, initialized to 1.0 and 0.0 respectively.
- `W_real[1024]` and `W_imag[1024]` hold precomputed twiddle factors used in butterfly computations.
- `N` (`.word 1024`) specifies the number of points in the FFT, while `const_N` (`.float 1024.0`) is used for normalization.
- `logN` is set to 10, representing $\log_2(1024)$, which determines the number of FFT stages.

These constants are crucial for guiding the control flow and arithmetic operations during vectorized FFT processing.

*.text Section with `_start` Entry Point*

```
.section .text
.globl _start

_start:
    la a0, real
    la a1, imag
    li a2, 1024
    la a3, W_real
    la a4, W_imag
    lw t0, logN
    lw t1, direction
    call vbit_reverse
    call vfft_ifft_transform
    call vifft_normalize
    call log_output
    li a7, 93
    ecall
```

The code segment begins by declaring the program's entry point `_start` and setting up the RISC-V registers. Registers `a0` through `a4` are loaded with pointers to the input real and imaginary arrays, the input size (1024), and the precomputed twiddle factors (`W_real` and `W_imag`). Registers `t0` and `t1` hold the logarithm of the input size and the transform direction (FFT or IFFT), respectively. The program then calls four main functions in sequence: `vbit_reverse` for bit-reversal permutation, `vfft_ifft_transform` for performing the core FFT/IFFT butterfly computations, `vifft_normalize` to normalize the output for the inverse transform, and `log_output` to handle any output logging or display. Finally, the program exits cleanly via the `ecall` instruction.

*Bit Reversal*

```
vbit_reverse:
    li t0, 0
    li t1, 1024
    li t2, 16
vbit_loop:
    bge t0, t1, vbit_done

    vsetvli t3, t2, e32, m1

    vid.v v0
    vadd.vx v0, v0, t0  # v0 = [i..i+15]

    vmv.v.i v1, 0

    vsrl.vi v2, v0, 0
    vand.vi v2, v2, 1
    vsll.vi v2, v2, 9
    vor.vv v1, v1, v2

    # bit 1 -> shift 8
    vsrl.vi v2, v0, 1
    vand.vi v2, v2, 1
    vsll.vi v2, v2, 8
    vor.vv v1, v1, v2

     # Bit 2
    vsrl.vi v2, v0, 2
    vand.vi v2, v2, 1
    vsll.vi v2, v2, 7
    vor.vv v1, v1, v2

    ......

    # bit 9 -> shift 0
    vsrl.vi v2, v0, 9
    vand.vi v2, v2, 1
    vor.vv v1, v1, v2

    # Create mask where i < reversed
    vmslt.vv v3, v0, v1

    vsll.vi v4, v0, 2  # i*4
    vsll.vi v5, v1, 2  # rev_i*4
    vmv.v.v v0, v3

    vloxei32.v v6, (a0), v4  # real[i]
    vloxei32.v v7, (a1), v4  # imag[i]
    vloxei32.v v8, (a0), v5  # real[rev_i]
    vloxei32.v v9, (a1), v5  # imag[rev_i]

    vmerge.vim v10, v8, v6, v0.t # new_real[i]
    vmerge.vvm v11, v9, v7, v3  # new_imag[i]
    vmerge.vvm v12, v6, v8, v3  # new_real[rev_i]
    vmerge.vvm v13, v7, v9, v3  # new_imag[rev_i]

    vsoxei32.v v10, (a0), v4
    vsoxei32.v v11, (a1), v4
    vsoxei32.v v12, (a0), v5
    vsoxei32.v v13, (a1), v5
```

```
    addi t0, t0, 16
    j vbit_loop

vbit_done:
    ret
```

The `vbit_reverse` function performs bit reversal using RISC-V vector instructions to process 16 elements simultaneously. Initially, the vector length is configured with `vsetvli` to operate on 32-bit elements in vectors of length 16.

The vector of indices starting at `i` is generated using `vid.v`, representing the data positions to be processed in parallel. Bit reversal is executed by isolating each bit using `vsrl.vi` (vector shift right immediate) and `vand.vi` (vector bitwise AND immediate), then shifting it to its reversed position using `vsll.vi` (vector shift left immediate). Partial results are combined through `vor.vv` (vector bitwise OR vector).

To identify which elements need swapping, the comparison mask is created with `vmslt.vv`, which sets bits where the original index is less than the reversed index. Data is loaded via indexed vector loads `vloxei32.v` using computed byte offsets.

Conditional swaps are efficiently performed using `vmerge.vim` and `vmerge.vvm`, which merge vector elements based on the mask, enabling in-place swaps without branching. The updated values are stored back with `vsoxei32.v`.

This vectorized method exploits data-level parallelism, substantially accelerating the bit reversal step crucial to the FFT algorithm.

*Vectorized bit reversal*

```
vbit_reverse:
    li t0, 0           # i = 0
    li t1, 1024        # N = 1024
    li t2, 16          # vector length

vbit_loop:
    bge t0, t1, vbit_done

    # Set vector length
    vsetvli t3, t2, e32, m1

    # Generate vector indices [i..i+15]
    vid.v v0
    vadd.vx v0, v0, t0  # v0 = [i..i+15]

    # Compute bit reversal (10 bits)
    # Initialize reversed = 0
    vmv.v.i v1, 0

    # Unrolled bit reversal (10 bits)
    # bit 0 -> shift 9
    vsrl.vi v2, v0, 0
    vand.vi v2, v2, 1
    vsll.vi v2, v2, 9
    vor.vv v1, v1, v2

    # bit 1 -> shift 8
    vsrl.vi v2, v0, 1
    vand.vi v2, v2, 1
    vsll.vi v2, v2, 8
    vor.vv v1, v1, v2

     # Bit 2
    vsrl.vi v2, v0, 2
    vand.vi v2, v2, 1
    vsll.vi v2, v2, 7
    vor.vv v1, v1, v2

    .......

    # bit 9 -> shift 0
    vsrl.vi v2, v0, 9
    vand.vi v2, v2, 1
    vor.vv v1, v1, v2

    # Create mask where i < reversed
    vmslt.vv v3, v0, v1


    # Compute byte offsets
    vsll.vi v4, v0, 2  # i*4
    vsll.vi v5, v1, 2  # rev_i*4
    vmv.v.v v0, v3

    vloxei32.v v6, (a0), v4  # real[i]
    vloxei32.v v7, (a1), v4  # imag[i]
    vloxei32.v v8, (a0), v5  # real[rev_i]
    vloxei32.v v9, (a1), v5  # imag[rev_i]

    vmerge.vim v10, v8, v6, v0.t
    vmerge.vvm v11, v9, v7, v3
    vmerge.vvm v12, v6, v8, v3
    vmerge.vvm v13, v7, v9, v3

    vsoxei32.v v10, (a0), v4
    vsoxei32.v v11, (a1), v4
    vsoxei32.v v12, (a0), v5
    vsoxei32.v v13, (a1), v5

    addi t0, t0, 16
    j vbit_loop

vbit_done:
    ret
```

**Vectorized Bit Reversal Routine**

This routine performs bit reversal on indices of a vector of length 1024 using RISC-V vector instructions. The main idea is:

- Initialize the loop counter and vector length (16 elements per iteration).
- For each vector segment, generate consecutive indices `[i .. i+15]`.

- Compute the bit-reversed index for each element by extracting each bit of the 10-bit index and reversing their positions.
- Create a mask to identify indices where the original index is less than its bit-reversed counterpart, ensuring swaps happen only once.
- Load the complex data (real and imaginary parts) at both the original and reversed indices.
- Conditionally swap the values using vector mask operations.
- Repeat until all indices are processed.

This approach exploits SIMD parallelism in the vector unit to perform bit reversal efficiently, which is crucial for FFT implementations.

*Vectorized FFT/IFFT Transform*

```
vfft_ifft_transform:
    li t2, 1
    li t3, 1

fft_stage_loop:
    bgt t3, t0, fft_done
    slli t4, t2, 1
    srli t5, t4, 1
    li t6, 0
    li a2, 1024
    div x19, a2, t4

group_loop:
    bge t6, x19, stage_done
    mv a5, t5
    vsetvli a6, a5, e32, m1
    mul x20, t6, t4      # j*m
    vid.v v0

    vadd.vx v0, v0, x20  # i = j*m + k
    vadd.vx v1, v0, t5   # ip = i + m/2
    vsll.vi v2, v0, 2    # i*4
    vsll.vi v3, v1, 2    # ip*4

    vloxei32.v v4, (a0), v2  # real[i]
    vloxei32.v v5, (a1), v2  # imag[i]
    vloxei32.v v6, (a0), v3  # real[ip]
    vloxei32.v v7, (a1), v3  # imag[ip]

    vsub.vx v8, v0, x20
    li t0, 1024
    div t0, t0, t4
    vmul.vx v8, v8, t0
    vsll.vi v9, v8, 2

    vloxei32.v v10, (a3), v9
    vloxei32.v v11, (a4), v9

    li t0, -1
    bne t1, t0, no_conj
    vfneg.v v11, v11

no_conj:
    vfmul.vv v12, v10, v6
    vfmul.vv v13, v11, v7
    vfsub.vv v14, v12, v13

    vfmul.vv v15, v10, v7
    vfmul.vv v16, v11, v6
    vfadd.vv v17, v15, v16

    vfsub.vv v18, v4, v14
    vfsub.vv v19, v5, v17
    vfadd.vv v4, v4, v14
    vfadd.vv v5, v5, v17

    vsoxei32.v v4, (a0), v2
    vsoxei32.v v5, (a1), v2
    vsoxei32.v v18, (a0), v3
    vsoxei32.v v19, (a1), v3

    addi t6, t6, 1
    j group_loop

stage_done:
    addi t3, t3, 1
    mv t2, t4
    j fft_stage_loop

fft_done:
    ret
```

**Vectorized FFT/IFFT Transform:** This RISC-V assembly code implements a vectorized Fast Fourier Transform (FFT) and Inverse FFT (IFFT) using vector instructions to process multiple data points simultaneously for high performance. The algorithm proceeds in stages, where at each stage the butterfly size doubles ($m = 2^{stage}$).

Key points: - `vsetvli` sets vector length dynamically based on current butterfly size. - Index vectors are generated (`vid.v`, `vadd.vx`) for data element access. - Data and twiddle factors are loaded using indexed vector loads (`vloxei32.v`). - For IFFT, twiddle factors are conjugated by negating the imaginary part. - Butterfly computations use vector floating-point multiply and add/subtract to combine elements. - Results are stored back with indexed vector stores (`vsoxei32.v`). - The loops iterate over FFT stages and groups within stages until the full transform is computed.

This vectorized approach significantly speeds up the FFT by leveraging parallelism at the hardware level.

**Vectorized IFFT normalization**

```
vifft_normalize:
    lw t1, .direction
    li t2, -1
    bne t1, t2, normalize_done

    la a0, real
```

```
    la a1, imag

    li t3, 1024
    li t4, 16
    vsetvli t5, t4, e32, m1

    li t6, 0x3a800000
    fmv.w.x ft0, t6

normalize_loop:
    beqz t3, normalize_done

    vle32.v v0, (a0)
    vle32.v v1, (a1)

    vfmul.vf v0, v0, ft0
    vfmul.vf v1, v1, ft0

    vse32.v v0, (a0)
    vse32.v v1, (a1)

    addi a0, a0, 64
    addi a1, a1, 64
    addi t3, t3, -16
    j normalize_loop
normalize_done:
    ret
```

**Vectorized IFFT normalization:** This code performs normalization of the IFFT output by dividing all real and imaginary values by $N = 1024$. It first checks if the operation is an IFFT (direction = -1). If so, it loads vectors of 16 floats at a time, multiplies each element by the constant divisor (1/1024 = 0.0009765625), and stores the results back. The use of vector instructions (`vle32.v`, `vfmul.vf`, `vse32.v`) allows parallel processing of multiple elements, optimizing performance. The loop continues until all elements are normalized.

*Print Function*

```
log_output:

    la a0, real
    la a1, imag

    li t0, 1024
    li t1, 0

log_loop:
    bge t1, t0, log_done

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

    fmv.x.s a0, ft0
    fmv.s.x ft0, a0
    call printToLog

    fmv.x.s a0, ft1
    fmv.s.x ft1, a0
    call printToLog

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

log_done:
    ret

printToLog:
    li a7, 42
    ecall
    ret
```

**Print Function:** This function logs (prints) all the output values stored in the `real` and `imag` arrays, typically after an FFT or IFFT operation. It loops over all 1024 complex values and prints their real and imaginary parts using a syscall via `printToLog`.

Key instructions:

- `la a0, real`, `la a1, imag`: Load addresses of real and imaginary arrays.
- `flw ft0, 0(a0)`: Load 32-bit float from memory into floating-point register.
- `fmv.x.s a0, ft0`: Move float value from `ft0` to integer register `a0` (needed for syscall).
- `call printToLog`: Calls the custom print routine, which uses `ecall` with service number `42` to print a float.
- `addi a0, a0, 4`, `addi a1, a1, 4`: Move to next float in arrays (4 bytes per float).
- `bge t1, t0,` `log`$_done$ : $Ends loop after printing all 1024 values.$

This loop ensures all real and imaginary values are printed in sequence, which is useful for debugging or validating transform outputs.

## IV. Summary of Vectorized FFT Algorithm using RISC-V Vector Extension

The FFT implementation utilizes the RISC-V Vector Extension (RVV) to achieve high throughput on large datasets. Input arrays `real[1024]` and `imag[1024]` store the signal in time-domain, and `W_real`, `W_imag` store the twiddle factors.

**Key stages of the algorithm include**:

1) **Bit-Reversal Permutation:** Vector instructions such as `vid.v`, `vsrl.vi`, `vsll.vi`, and `vor.vv` are used to compute bit-reversed indices. Swapping is done via `vloxei32.v`, `vsoxei32.v`, and `vmerge.vvm`.
2) **Butterfly Computation:** The FFT stages use `vfmul.vv`, `vfsub.vv`, and `vfadd.vv` for vectorized complex multiplication and addition.
3) **IFFT Normalization:** Each component is scaled using `vfmul.vf` to divide by the total number of points.

**Crucial vector instructions used:**

- `vsetvli` – configure vector length and type
- `vid.v` – generate index vector
- `vloxei32.v`, `vsoxei32.v` – gather/scatter with computed offsets
- `vfmul.vv`, `vfadd.vv`, `vfsub.vv` – floating point butterfly operations
- `vmerge.vvm`, `vmslt.vv` – conditionally merge/swapping elements

## V. Problems Encountered

The very first issue we faced during this project was the downloading of the Virtual Machine/Ubuntu environment. Two of our teammates faced a tough challenge in downloading the virtual machine as it would load up to 100% and then throw a "lack of space" error, which was incredibly frustrating for all of us. Thankfully, one of our TAs stepped in and helped us resolve the issue.

Secondly, when we started understanding the logic to implement this project, we realized that we are still beginners in the RISC-V Vector ISA part, which turned out to be the most challenging part of all.

Thirdly, when we implemented the non-vectorized code after the C program, it would get stuck in infinite loops where we couldn't figure out the cause, and the output would also consume a significant amount of memory on our laptops.

Despite these challenges, through the constant support of each other, we were able to complete this project and learned a great deal of new things.

However, we are still unable to get the correct output, for reasons we don't yet understand, which has been a source of ongoing frustration.

## VI. Experience

Although this project is from our CAAL course, it felt like a real-world implementation. Before starting, we had no idea that FFT or an algorithm like that even existed, but this pushed us to research and build our code, documentation, and understanding on our own. Working with the RISC-V Vector ISA, which we were only introduced to through our instructor's video, was especially challenging; we had to learn both Vector ISA and SIMD concepts from the ground up. Given the lack of resources available over the internet that we realized during our project, it feels rewarding to now be part of something that can support others on the same journey.


## Acknowledgements

We are thankful to our instructor, Dr. Salman Zaffar for introducing us to the RISC-V Vector ISA and providing valuable guidance throughout the course.

We would also like to express our sincere gratitude to our teaching assistant, for their timely assistance in resolving technical issues related to the Virtual Machine setup. Their support was crucial in enabling us to proceed with the project.

Additionally, we appreciate the collaboration and support of our teammates, whose collective effort and determination helped us overcome numerous challenges.

Finally, we acknowledge the availability of online resources and communities that helped us deepen our understanding of the concepts involved.