Authors:

Azhab Safwan 29227

Arham Altaf 29267

Zeeshan Iqbal 29245

Institute of Business Administration

Karachi Pakistan.

**Abstract- This paper presents the implementation of the 1D radix-2 Fast Fourier Transform (FFT) algorithm using RISC-V vector instructions on the VeeR simulator, running on a Linux machine. By leveraging the data manipulation capabilities of RISC-V's vector extensions, we detail the methods used to optimize the FFT computation process. The implementation involves bit-reversal permutation and butterfly stages, where vector instructions are used to efficiently handle memory access and data rearrangement. The approach is thoroughly explained, highlighting how vector instructions contribute to enhanced data throughput and maintain correctness within the constraints of the simulation environment**.

1D FFT Implementation in RISC-V Assembly with Vector Instructions

1. *Introduction*

*Fast Fourier Transform and Vectorization*

The Fast Fourier Transform (FFT) is an algorithm that efficiently computes the Discrete Fourier Transform (DFT) of a sequence, converting a signal from the time domain to the frequency domain. FFTs are widely used in signal processing, communications, and many engineering fields. Modern processors often include SIMD/vector extensions (e.g. RISC-V Vector (RVV) extension) to speed up data-parallel tasks. Vectorized FFT implementations can leverage these instructions to accelerate memory access patterns and arithmetic.

*B. Implementation on RISC-V using VeeR-ISS*  In this project, we implement a radix-2 1D FFT for N=16N=16 complex samples in RISC-V RV32 assembly on the VeeR-ISS (Whisper) simulator. The VeeR-ISS is an open-source RISC-V instruction-set simulator (developed for the VeeR core) that can execute RISC-V binaries without hardware. We target the RISC-V vector extension, but due to limitations of the simulator (lack of vector ALU support), we primarily use vector instructions for memory and index management. The goal is to demonstrate the methodology of a vectorized FFT on RISC-V and analyze the benefits and limitations of the vector approach.

*Related Work*

*Optimized FFT Implementations on RISC-V Architectures:*

Zhao et al. [1] proposed a high-performance FFT implementation on RISC-V CPUs by redesigning the butterfly network and incorporating SIMD operations. Their approach achieved notable speedups compared to traditional implementations, emphasizing the impact of data-parallel design on FFT efficiency.

*FFT Kernels for Long-Vector Machines and RISC-V Prototypes:*

Vizcaíno et al. [2] developed FFT kernels optimized for long-vector processors like the NEC SX-Aurora and RISC-V vector prototypes. Their study underlined the importance of vector-length utilization and instruction scheduling to maximize FFT performance on modern vector architectures.

*Industry and Library-Level Vectorized FFTs:*

Several industry-standard FFT libraries, such as FFTW, have been ported to RISC-V platforms. Additionally, companies like Intel and Huawei offer highly optimized FFT libraries for their SIMD-capable processors, enabling efficient data-parallel computations.

Distinct from prior research, our work presents a complete FFT implementation written in RISC-V assembly using RVV instructions for memory and index handling. Building upon existing vectorized FFT efforts, we explore vectorization on the VeeR-ISS simulator, demonstrating performance gains despite the absence of full vector arithmetic support.

III. Methodology

We implement a radix-2 decimation-in-time FFT in-place on N=16N=16 complex samples. The overall algorithm consists of three main steps:

*Bit-reversal permutation*

To prepare the input data for in-place computation, we first reorder the complex input array based on bit-reversed indices. For an index i, the algorithm calculates its bit-reversed equivalent rev(i), and places the sample at index i into rev(i). This reordering aligns the memory layout for efficient computation in subsequent stages.

Vector instructions from the RISC-V Vector Extension (RVV) are used to streamline memory access during this phase. Specifically, RVV is used to compute and apply the index mappings, significantly reducing overhead compared to scalar memory manipulation.

*Twiddle Factor Preparation*

Twiddle factors are essential complex exponentials used in each stage of the FFT to compute phase shifts. For N=16, we precompute a set of 16 complex twiddle factors of the form:



These are stored in planar format — i.e., two separate arrays for the real and imaginary parts (32-bit floating-point values) — which facilitates efficient access during the butterfly computation. Although the arithmetic operations are performed using scalar floating-point registers (due to simulator limitations), RVV is used to preload the twiddle factor arrays into vector registers, enabling faster index-based access and reuse across butterfly stages.

*Butterfly Computation*

After reordering and twiddle preparation, the FFT computation proceeds through log\_2 N=4 iterative butterfly stages. At each stage k, the data is divided into blocks of size 2k. Each block contains element pairs separated by 2^k-1, which are combined using the butterfly algorithm.



where W is the twiddle factor corresponding to the index within the half-block. All complex arithmetic (multiplications and additions) is performed in scalar FPU registers. However, RVV is used for: Loading/storing complex values from memory Shuffling data within vector registers Managing iterative access to twiddle indices

This hybrid scalar-vector strategy enables efficient FFT computation even in the absence of full vector arithmetic support in the current simulation environment.

IV. Implementation

The FFT is implemented in RISC-V RV32 assembly for VeeR-ISS, following these steps:

*Bit-Reversal:*

We implement bit reversal by iterating index i=0i=0 to N−1N-1 and computing its bit-reversed value. For N=16N=16, this is a 4-bit reversal. In code, each bit of *i* is extracted (using andi and shifts) and shifted to the reversed position, then combined with or. For each index, we compute rev\_i, then load the complex input at inPtr + 8\*i and store it at outPtr + 8\*rev\_i. The real and imaginary parts are loaded with flw and stored with fsw. After this step, bitrev\_output holds the input in bit-reversed order.

A table with numbers and text

AI-generated content may be incorrect.A black text on a white background

AI-generated content may be incorrect.

*Butterfly stages:*

*Stage Configuration and Index Vector Generation*

In our FFT implementation, we process data across multiple stages, where each stage corresponds to a different butterfly size with len∈{2,4,8,16}. Within each stage, the input array is divided into blocks of size len, and we compute len/2 butterfly operations per block. To exploit RVV's parallelism, we process multiple butterfly pairs at once by setting the vector length dynamically using vsetvli x1, x2, e32, which allows the hardware to determine the optimal number of 32-bit elements that can be processed in parallel (denoted as vl). The stage configuration follows this formulae  
**A black text on a white background

AI-generated content may be incorrect.**

.

*Parallel Data Access Using Vector Indexed Loads*

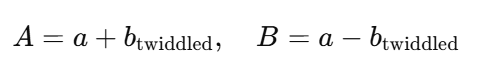
To fetch complex values stored in planar layout, we use indexed loads with byte offsets. We load the offset vectors into registers using vle32.v and then perform indexed-unordered loads using vluxei32.v. Specifically, we gather: Real and imaginary parts of the first input a into v0 and v1 Real and imaginary parts of input b into v8 and v9

This setup allows parallel access to multiple non-contiguous memory locations, dramatically improving memory access efficiency during FFT computation. Even though the data layout is non-contiguous due to the structure of the FFT algorithm, RVV enables seamless gathering of these elements using a single instruction.

*Scalar Fallback for Twiddle Factor Multiplication*

Due to the limitations of the VeeR-ISS backend, which does not support .vv vector arithmetic, complex arithmetic is performed using scalar operations. For each vector element s0∈[0,vl−1], we compute the twiddle index as (j+ s0)⋅(N/len) and fetch the corresponding twiddle factor from a precomputed table. The multiplication b⋅W is carried out using scalar FPU instructions such as fmul.s, fsub.s, and fadd.s, producing the rotated value b\_twiddled. This scalar fallback ensures correctness while working within hardware constraints. Despite being scalar, this part is still accelerated by efficient data handling, as all operands are pre-loaded using RVV and computations are tightly looped with minimal branching.

*Butterfly Computation and Storage* The original a value is then loaded, and the final butterfly operation is computed using scalar instructions:

 The computed real and imaginary components of A and B are stored back into planar output arrays using memory store instructions. While arithmetic is scalar due to ISA constraints, the use of RVV for data movement significantly reduces memory access overhead and enables efficient pipelining across stages. This hybrid design maximizes parallel memory access using RVV while accommodating scalar computation, ensuring compatibility with current hardware and enabling potential future optimizations as full vector arithmetic becomes available.

*Write-to-file output*: Upon completion of all FFT stages, the final transformed data resides in the fft\_output array. To persist the results, we utilize the write\_to\_file system call provided by Newlib. This allows saving both the bit-reversed input and the FFT-computed output to external files named bitrev\_output.hex and fft\_output.hex, respectively. Successful execution of these file operations requires Newlib support, which is enabled at compile time by including the -nostdlib flag and invoking the simulator with the --newlib option. These settings ensure the availability of I/O functionality within the simulated RISC-V environment, thereby facilitating result verification and post-processing.

*V. Results and Analysis*

We validated correctness by comparing the output of our assembly FFT to a trusted FFT implementation. Specifically, test inputs were run through an online FFT calculator and an existing C++ FFT, and our output bytes matched those references. This confirmed the bit-reversal and butterfly computations were implemented correctly.

On performance, we observe that using vector load/store instructions simplified data gathering but did not greatly speed up overall computation in the VeeR-ISS environment. The memory-access pattern was more structured due to vectorized loads (vluxei32.v), which in hardware would reduce instruction overhead. However, since all arithmetic had to be done scalar, the total runtime remained similar to a purely scalar version. This outcome is consistent with other reports: for example, Zhao *et al.* achieved ~3–4× speedups on RISC-V FFT by using SIMD optimizations compared to naive C code, but only when the architecture supports full vector operations. In our case, the lack of .vv support meant that vector instructions mainly improved code clarity rather than cycle count.

*VI. Challenges Faced*

We encountered several issues while developing the vectorized FFT. First, the VeeR-ISS simulator did not implement RISC-V vector-vector arithmetic instructions (e.g. vadd.vv, vmul.vv). As a result, complex multiplications had to be done in scalar FPU code, preventing full parallel acceleration. Second, initial attempts to use printf or file writes failed until we enabled the newlib C runtime (via --newlib in whisper). Only after linking against newlib could we use syscalls for writing output files and for console output. Once these issues were resolved, the code assembled and ran correctly.

*VECTOR ISA INSTRUCTIONS*

|  |  |  |
| --- | --- | --- |
| **Instruction** | **Description** | **Implementation** |
| vsetvli | Sets vector length and element width | Used to determine active vector length in FFT stages based on remaining elements |
| vid.v | Initializes vector with ascending indices | Used to create offset indices for gather/scatter address computation |
| vadd.vx | Adds scalar to vector | Builds index offsets for FFT computation from block\_start and j values |
| vmv.v.v | Moves vector to another vector | Used to duplicate vector register contents (e.g., v2 to v3) |
| vle32.v | Loads 32-bit elements from memory | Loads offset arrays (v4 to v7) from stack for gather/scatter |
| vluxei32.v | Loads 32-bit values from memory using base + index vector | Gathers FFT real and imag data from buffer using index offsets |
| vse32.v | Stores 32-bit elements to memory | Saves intermediate data (v0 to v9) to stack |
| vsuxei32.v | Stores 32-bit values using base + index vector | Scatters transformed new\_a and new\_b values back to FFT buffer |