An FFT Accelerator Using Deeply-coupled RISC-V Instruction Set Extension for Arbitrary Number of Points

Shijie Jiang, Yi Zou*, Hao Wang and Wanwan Li
School of Microelectronics, South China University of Technology,
Guangzhou, China
Email: mijiangshijie@mail.scut.edu.cn, zouyi@mail.scut.edu.cn*,
harryw@mail.ustc.edu.cn, mswanwanli@mail.scut.edu.cn

***Abstract*—Fast Fourier Transform (FFT) is one of the most widely used algorithms in digital signal processing (DSP). Many Internet of Things (IoT) devices include dedicated FFT hardware accelerators at the price of both extra hardware resources and power consumption. Limited by constrained resources for IoT applications at the edge, it is favorable as well as crucial to reduce such overhead as much as possible without sacrificing FFT performance. Meanwhile, open-source RISC-V microprocessors have become serious contenders to ARM or x86 alternatives, especially in IoT applications. Recent studies have shown that Application-specific Instruction-set Processors (ASIP) can achieve considerable acceleration performance with low overhead.**

This paper proposes a fine-grained RISC-V Instruction Set Architecture (ISA) extension scheme for efficient FFT operations with 12 new instructions, using four 16-bit multipliers and adders as computing units to minimize overhead. Our extended instructions are deeply coupled with the CPU pipeline, thus significantly reducing the hardware overhead of decoding and executing logic, and theoretically can accelerate the calculation of FFT with an arbitrary number of points. We demonstrate the proposed scheme on a Xilinx PYNQ-Z2 FPGA using the open-source RISC-V NutShell soft IP core, with data loading from either specially designed vector registers (V-mode) or RAM off-the-core (R-mode). The evaluation shows the proposed FFT acceleration scheme achieves a performance gain of 118 times in V-mode and 6.5 times in R-mode respectively, with only 16% power consumption required as compared to the vanilla NutShell RISC-V microprocessor.

***Index Terms*—ISA Extension, FFT Accelerator, ASIP, RISC-V**

I. Introduction

Fast Fourier Transform (FFT) is an efficient implementation of Discrete Fourier Transform (DFT), which plays an important role in digital signal processing (DSP) [1], especially for Internet of Things (IoT) using various types of sensors [2]. FFT generally serves as the very first step in the data processing pipeline to convert the time domain signals to the frequency domain for feature extraction. Note that edge IoT applications are designed to operate 24/7, making a low-cost and low-power yet efficient FFT design one of the key factors to be considered [3]. Furthermore, many existing IoT solutions depend on the built-in hardware-based FFT accelerators or complicated DSP processors to perform FFT operations [4], requiring massive additional hardware resources and power consumption. With constrained resources and compact in size in IoT applications, it is favorable to explore alternatives to achieve efficient FFT operations with as small overhead as possible. On the other hand, RISC-V [5], an open-source instruction set architecture (ISA) alternative to proprietary ARM [6] and x86 [7], [8], has recently emerged as a very promising competitor [5], gaining a wide range of adoption in IoT devices due to its loyalty-free, low-cost, and low-power design. The flexibility and modular design of RISC-V ISA allow extending customized new instructions efficiently.

In this paper, we introduce a fine-grained, deeply coupled RISC-V instruction set extension to the CPU pipeline for edge device acceleration of FFT algorithms. By implementing the FFT extension instruction set on an open-source processor, NutShell, the execution efficiency is improved with reduced power consumption as well as low overhead for additional hardware resources. We summarize our main contributions from this paper as follows below.

- We *propose* a fine-grained RISC-V Instruction Set Architecture (ISA) extension scheme for efficient FFT operations with any number of data points. The proposed FFT instruction set extension is designed to deeply couple with the existing CPU pipeline, thus significantly reducing the hardware overhead of decoding and executing logic.
- We *present* an implementation of the proposed FFT acceleration design using the NutShell open-source RISC-V processor [9], which supports a flexible data loading scheme to handle both small or large-scale problems effectively, i.e., using vector registers (V-mode) or RAM off-the-core (R-mode). We further show that the proposed design achieves industry-grade FFT operations and maintains a minimal footprint of required hardware resources.
- We *demonstrate* a significant performance gain of 118 times using the V-mode, and 6.5 times using the R-mode respectively, with less than 16% power consumption required, as compared to the case using RISC-V directly for FFT operations, making it a very promising approach to RISC-V deployment in IoT applications.

DOI 10.1109/ASAP57973.2023.00036

- We *share* the proposed design of the instruction level FFT acceleration with the open-source RISC-V community through the NutShell project to help grow the RISC-V based solutions to a wider range of adoptions [10] [9].

The paper is structured as follows. In Section II, we discuss related work in this context. Next in Section III, we provide a brief background on RISC-V and FFT, followed by details of the proposed design. In Section IV, we demonstrate the architecture of NutShell-FFT in detail. We describe the lab setup and discuss experimental results in Section V and share our further thoughts based on the analysis of the results. Last, we conclude the paper in Section VI.

II. Related Work

The hardware implementation of the FFT algorithm is divided into pipeline structure [11] [12] [13] [14] [15] and memory-based structure [16] [17] according to the data flow. Generally, the pipeline structure is divided into parallel processing and serial feedback structures [13] and are encapsulated in the form of an IP core in practical implementation. The pipeline structure is able to handle large amounts of data at higher throughput such as cameras or 4D mmWave sensors but often requires a large number of hardware resources. Compared with the pipeline structure, the memory-based structure has less overhead on hardware resources and power consumption but suffers from low data throughput and high latency, which is often used to deal with scenarios with a relatively small data flow.

Meanwhile, the open-source RISC-V Instruction Set Architecture (ISA) has received a lot of attention from both industry and academia. Its open source license as well as its simple and efficient architecture design allow it to be freely used and modified, which is best suited in low-power embedded application scenarios [5]. In recent years, the scalability of RISC-V has received a lot of attention, in particular in the RISC-V extension instruction set used for Application-specific Instruction-set Processor (ASIP) [18] [19] [20]. Koppelmann et al. introduced advanced bit manipulation instructions (BMIs) extension for RISC-V ISA, and implement them on Berkeley's Rocket CPU, which improved the running efficiency and power consumption of compiling software for various applications [18]. Fritzmann et al. proposed RISQ-V, which is a set of instruction extensions with 29 new instructions and deeply integrated into the CPU pipeline to speed up lattice-based post-quantum cryptography (PQC) [19]. Compared with the original CPU, it has increased the resource area by 1.6 times but has achieved a maximum of 11.4 times acceleration effect and 9.5 times lower energy consumption in the test case. Schuiki et al. propose a stream semantic register (SSR) to improve both utilization and energy efficiency on single-issue processors [20]. SSR is a lightweight RISC-V extension that implicitly encodes memory access as register read/write, eliminating a large amount of load/storage. The experiment proves that SSR extension provides a significant architecture acceleration of 2 to 5 times between different cores, and the core area only increases by 11%. The above studies prove that the deep coupling between RISC-V instruction set extension and CPU pipeline can achieve considerable performance with low overhead. Similarly, RISC-V instruction set extension has also been studied in FFT applications, and [21] and [22] are both involved in this work. Hussain et al. [21] present an ASIP for 64 and 128-point FFT operation by integrating a template-based Coarse-Grain Reconfigurable Array (CGRA). They use RISC-V extension instructions for data loading and logic controlling. The experimental results show that compared with the FFT accelerator in [23], it has improved performance and reduced hardware resource overhead. Hu et al. [22] construct a framework based on the RISC-V processor that can receive data flow graph (DFG)-based descriptions and automatically perform instruction set extensions. They use this method to generate FFT extension instructions with a 2.6x performance improvement.

However, existing solutions that implement FFT in hardware generators require considerable hardware resources and power consumption. With the adoption of RISC-V, it is beneficial to support FFT operations in a more organic manner in RISC-V architecture. Thus using RISC-V instruction extension for an ASIP seems to be a more promising approach. We believe that the FFT instruction set extension design with a deep coupling on the processor can achieve better acceleration performance improvement with little hardware resource overhead, as illustrated in later sections of this paper.

III. Proposed Design

We next show the proposed fine-grained FFT extension instruction set for RISC-V, which achieves controlling the FFT at the instruction level with only 12 new instructions. More precisely, we deeply couple the decoding and execution of the extended FFT instructions into the microprocessor pipeline, maximizing the benefit of the existing RISC-V architecture without introducing any significant complexity. As low as a total of only four multipliers and adders are required as the operating units and fully integrated with the existing RISC-V architecture seamlessly. Another benefit of deeply coupled with the CPU pipeline is that we can precisely control the FFT calculation at the instruction level. Using the proposed FFT extension instructions, users can program the FFT algorithm at the operation level, thus supporting FFT calculation at any point. Before going into details of the proposed design, we provide a primer on the FFT algorithm in the following.

A. Analysis and Abstraction of FFT Operation

1) Principle of FFT algorithm: As an efficient algorithm of DFT [24], FFT plays an irreplaceable role in time-frequency transformation analysis. The operation formula of DFT is

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{nk}, \ 0 \leqslant k \leqslant N-1, \tag{1}$$

where,$W_N^{nk} = e^{-j\frac{2\pi k}{N}n}$. If DFT is split into odd and even terms, the first $\frac{N}{2}$ points can be expressed as,

$$\begin{aligned} X[k] &= \sum_{r=0}^{\frac{N}{2}-1} x\left[2r\right] W_N^{2rk} + \sum_{r=0}^{\frac{N}{2}-1} x\left[2r+1\right] W_N^{(2r+1)k} \\ &= \sum_{r=0}^{\frac{N}{2}-1} x\left[2r\right] W_{\frac{N}{2}}^{rk} + W_N^k \sum_{r=0}^{\frac{N}{2}-1} x\left[2r+1\right] W_{\frac{N}{2}}^{rk} \\ &= A[k] + W_N^k B[k] \ , k = 0, 1, \dots \ , \frac{N}{2} - 1 \end{aligned} \tag{2}$$

Similarly, the last $\frac{N}{2}$ points can be expressed as,

$$X\left[k + \frac{N}{2}\right] = A[k] - W_N^k B[k] \ , k = 0, 1, \dots \ , \frac{N}{2} - 1. \tag{3}$$

It can be seen that the value of the last $\frac{N}{2}$ points can be completely determined through the intermediate process of calculating the first $\frac{N}{2}$ points. If we continue odd even decomposition of $A[k]$ and $B[k]$ until they become 2-point DFT, avoiding repeated calculations.

2) FFT calculation structure characteristics: Using the 8-point FFT operation structure diagram as an example in Fig. 1, we analyze and abstract FFT calculation characteristics.

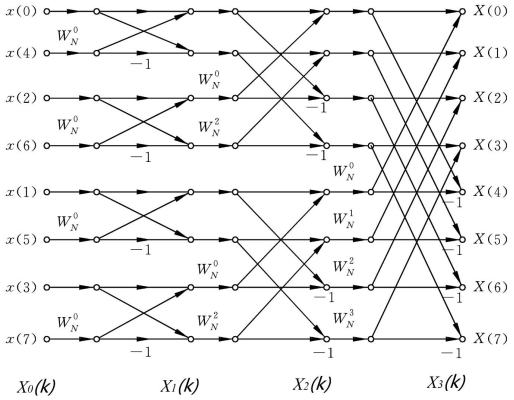
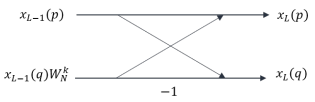


Fig. 1: 8-points FFT Operation Diagram

a) Layered calculation: In the radix-2 FFT algorithm, if the number of FFT operation points is N and there are L-layer operations, they meet the requirements $L = log_2 N$. The number of operation layers $m = 0, 1, ..., L-1$. As shown in Fig1, 8-point FFT has three layers.

b) Butterfly processing element: The main computing structure of FFT consists of several butterfly computing units. The schematic diagram of butterfly computing units is shown in 2. We can see that the butterfly processing element first performs the complex multiplication of the corresponding twiddle factor, and then the operation of butterfly addition. Therefore, we can divide the traditional butterfly operation unit into two calculation instructions, the `complex-mul` instruction and the `butterfly-add` instruction. We split the function of the butterfly processing element into two instructions because if we just use one combined instruction, the clock cycle will be greatly prolonged, impacting the performance dramatically.



$$\begin{aligned} x_L(p) &= x_{L-1}(p) + W_N^k x_{L-1}(q) \\ x_L(q) &= x_{L-1}(p) - W_N^k x_{L-1}(q). \end{aligned} \tag{4}$$

Fig. 2: Illustration of the butterfly processing element

c) Sequence transposition: In FFT operation, the input data needs to do the sequence transposition first, so that the output data is sequential. Taking 8-point FFT as an example, the binary codes corresponding to $X(0) \sim X(7)$ are $X(000)$, $X(001)$, $X(010)$, $X(011)$, $X(100)$, $X(101)$, $X(110)$, and $X(111)$. Correspondingly, the exchanged sequence of binary codes are $X(000)$, $X(100)$, $X(010)$, $X(110)$, $X(001)$, $X(101)$, $X(011)$, and $X(111)$, which is just the bit-flip of the previous sequence. We introduce the `shuffle` instruction to do sequence transposition.

B. Design of FFT Extended Instruction

We use the extended instruction opcode reserved by RISC-V ISA to design our FFT extension instruction to avoid any conflict with standard RISC-V extension instructions at the compilation level. We design FFT operation instructions (`complex-mul`, `butterfly-add`, and `shuffle`) for the FFT algorithm. Moreover, we design 128-bit wide vector registers for these special FFT operation instructions, to achieve the single-data multi-data (SIMD) parallel acceleration. We use a 32-bit fixed-point format, which means the data is a 32-bit complex number with 16 bits each for the real and imaginary parts and one vector register can hold up to four data. We also design vector register instructions for data load/storage. Our vector register instruction support memory access and immediate value load, making the FFT extension instructions versatile and scalable to support other DSP-related algorithms at low cost.

C. FFT Operation Instructions

a) `shuffle`: The `shuffle` instruction is used to exchange the input sequence so that the output sequence is sequential. For vector registers, a total of four data can be stored. According to the requirements of the FFT algorithm, we have designed two sequence exchange methods as shown in Fig. 3, which are represented by `inst` [27:25].

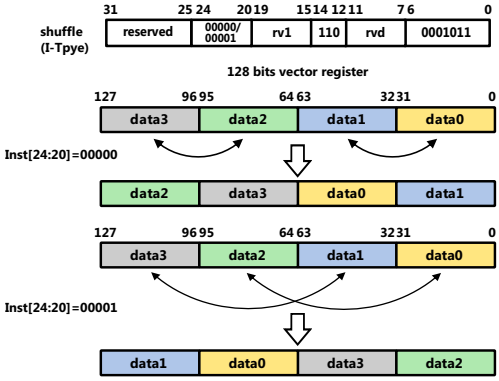


Fig. 3: The `shuffle` Instruction

b) complex-mul: The `complex-mul` instruction is designed to multiply the twiddle and the corresponding data in rv1 and write the result back to rvd as shown in Fig. 4. According to the operation mode of FFT and the data storage mode of the vector register, there are three different forms of multiplication. In the instruction design, we use `inst` [14:12] to specify the level of operation in Fig. 4. `inst` [24:20] indicates the number of levels and groups of the current FFT operation, which is used to index the correct twiddle factor.

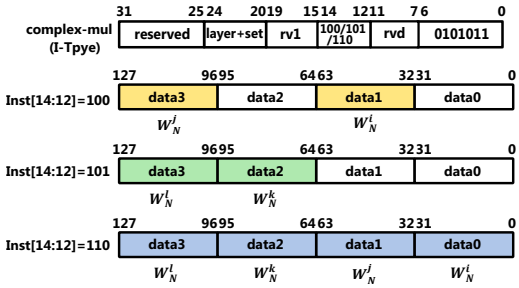


Fig. 4: The `complex-mul` Instruction

c) butterfly-add: The function of the `butterfly-add` instruction is to calculate the cross addition and subtraction of the corresponding data after the `complex-mul` instruction. According to the needs of the FFT algorithm, we have designed two operation modes, indicated by `inst` [27:25] as shown in Fig. 5. In I-type decoding mode, the source operand is only rv1. At this time, the data of rv1 needs to be calculated and written back to rvd. In the R-type decoding mode, the source operands are rv1 and rv2. After the corresponding operation, we directly write the operation result of rv1 back to rvd, and store the operation result of rv2 in the buffer. In this case, the butterfly-add instruction needs to be followed by the write buffer instruction to write the data of rv2 back to the corresponding rvd as shown in Fig. 5.

d) write-buffer: During our operation, there may be situations where the operation results of the two source operands need to be written back to the original registers. To this end, we design a register result buffer to store the register data that has not been written back. After the operation instruction, we use the write-buffer instruction and specify the correct rvd to write the cached data back to the corresponding destination register.

D. Vector Register Instruction

We have designed two ways of data loading for vector registers, including both immediate value and dedicated memory access, supporting loading 16-bit immediate value to the specified location of the vector register.

1) Dedicated vector registers. The FFT data is loaded into the vector register in advance, and we can directly use the register data for FFT calculation. This method supports up to 128-points high-speed FFT operations;
2) External DRAM is used as a data storage module. The DRAM is connected to the microprocessor core through the AXI bus. Before we perform each operation, we need to use the memory access instruction to load the data of the corresponding address into the vector register. This method can support the FFT operation at any point, but the performance is worse than the first one due to the overhead of accessing memory for each operation.

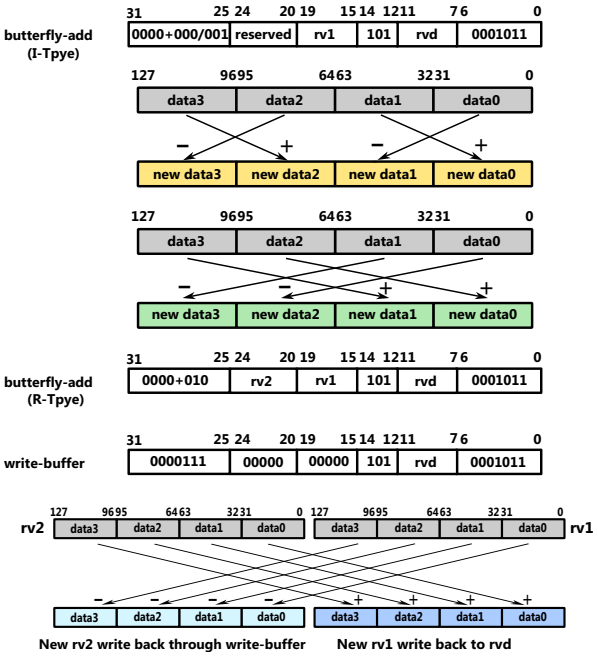


Fig. 5: The `butterfly-add` Instruction

a) li2vec: We load the 16-bit immediate data into the vector register through the instruction, `inst`[12,15] represents the different positions of the vector register. The design of `li2vec` instruction is shown Fig. 6.

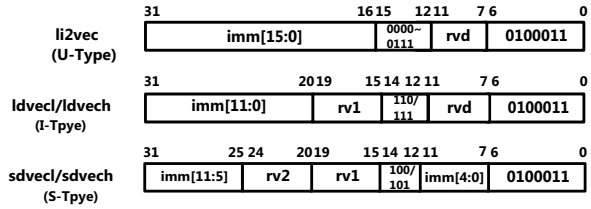


Fig. 6: The memory access related Instruction

b) ldvech/ldvecl/sdvech/sdvecl: We also design memory access instructions to load data. In order to make a fair performance comparison with the NutShell microprocessor, we deliberately kept the original memory access bus width of 64 bits. Therefore, for the 128-bit vector register, we need two data loading and storage instructions, `ldvec` and `sdvec`, respectively. `ldvecl` and `sdvecl` correspond to `rvec` [0,63], and `ldvech` and `sdvech` correspond to `rvec` [64,127].

IV. The Architecture of NutShell-FFT

We select an open-source RISC-V microprocessor, NutShell, to implement the FFT extension instruction set. NutShell is a modular design using the Chisel language and based on the RISC-V RV64 instruction set [9]. NutShell supports parameterized configuration and currently, the verified core

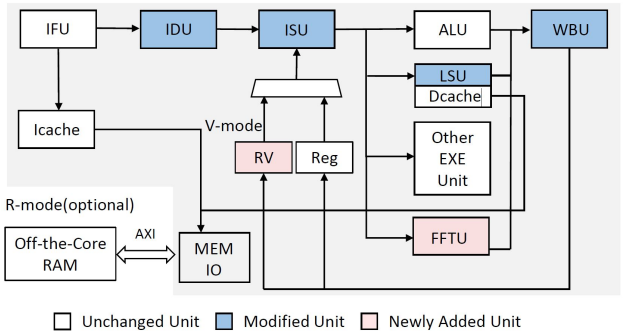


Fig. 7: NutShell-FFT [10], the FFT Extension based on NutShell [9].

configuration sets include embedded core, sequential core, and out-of-order core. We use the NutShell sequential core for development, tightly coupling our FFT extension instructions to the NutShell microprocessor as shown in Fig. 7, and make full use of NutShell's pipeline decoding logic and function unit, minimizing additional hardware resource overhead. We also did not modify the bus width of external memory access, to be able to compare the performance of the RV64 instruction set more fairly.

A. Modifications to the NutShell Microprocessor

1) 128-bit vector register: Since FFT is a computation-intensive algorithm, there is a strong demand for making it as parallel as possible. Therefore, we design a SIMD vectorized extension register to support data-parallel computing. We choose a 32-bit complex data type for FFT data storage with 16-bit real and imaginary parts. Therefore, the vector register is able to store up to 4 data for FFT calculation.

2) FFTU: We add the FFTU module in the execution stage of the pipeline as an independent module and use SIMD technology to execute the FFT operation instruction. There are four multipliers capable of executing `complex-mul` in parallel, as well as four adders used for `butterfly-add`.

3) Decoding logic of pipeline: a) Identify FFT extension instructions at the decoding level; select vector registers, and general-purpose registers through multiplexers; then access data; b) Identify the FFT extension instruction at the execution level; execute the corresponding calculation in the FFTU module; c) Modify the logic of the memory access module to load and store data from general-purpose registers or vector registers through multiplexers. d) Modified the register data path to 128-bit, ensuring that the vector registers and general registers can be loaded in one clock cycle.

B. Architectures of NutShell-FFT

1) Vector register Only (V-mode): In V-mode, all data is loaded into the vector register for execution, and the operation efficiency is high. But due to the limitation of on-chip storage space, the number of operation points is small. As the maximum number of registers in the RISC-V instruction set is 5 bits supporting 32 registers, so V-mode supports up to 128 points for FFT;

2) RAM off-the-core (R-mode) : In R-mode, the data is stored in DRAM off-the-core, and each FFT operation requires a memory access operation first. In theory, it can support FFT operations at any point. We have tested FFT calculations from 256 points to 128K points, which are more practical, but due to the characteristics of the FFT algorithm, the data loading time will be increased, and the performance has a certain loss compared with the former, but it has performance advantages compared with vanilla NutShell.

V. Experiment and Result Comparison

A. Experimental Method

We use the Xilinx PYNQ-Z2 FPGA development board as the hardware environment for realizing the proposed FFT accelerator. Based on the open source NutShell RISC-V, we build the *NutShell-FFT* SoC project using Xilinx Vivado 2019.01 [25]. We next port the project to the PYNQ-Z2 development board for onboard verification.

Algorithm 1: Replace FFT-related calculation with FFT extension instructions in a standard FFT algorithm

Def `FFT(X)`**:**
  n = length of (X); **set** $y = [0] * n$
  if $n == 1$ **then**
    return $y = X$
  end
  else
    Use `ldvec` or `li2vec` for data preparation:
    $X_e = [x_0, x_2, \cdots, x_{n-2}]$
    $X_o = [x_1, x_3, \cdots, x_{n-1}]$
    $y_e = \texttt{FFT}(X_e)$; $y_o = \texttt{FFT}(X_o)$; $\omega = e^{\frac{2\pi i}{n}}$
    for j *in* $range(\frac{n}{2})$ **do**
      $y[j] = y_e[j] + \omega^j y_o[j]$
      $y[j + \frac{n}{2}] = y_e[j] - \omega^j y_o[j]$

      Use FFT extension instructions in the test case:
      `complex-mul` $y_o[j]$, $\omega^j \Rightarrow y_o^{'}[j]$
      `butterfly-add` $y_o^{'}[j]$, $y_e[j]$
      $\Rightarrow y[j]$, $y[j + \frac{n}{2}]$
    end
    return y
    Use `sdvec` for data output.
  end
return

As a fair performance evaluation as well as a comparative study, we design the experiment to conduct a performance comparison of FFT running in the proposed NutShell-FFT against FFT running in the vanilla NutShell. We generate test cases and benchmarks as follows.

Since the NutShell currently does not support floating-point calculations, we first choose a standard FFT algorithm as shown in Algorithm 1 and use an open-source library, libfixmath [26] with fixed-point FFT calculations to run on the vanilla NutShell as a benchmark. Libfixmath is a platform-independent fixed-point math library aimed at performing fixed-point math on platforms that lack floating-point support. We use libfixmath to realize the standard FFT algorithm with the 32-bit fix-point data format and the resulting benchmark after cross-compilation is the RV64IM.

Next, We replace the FFT-related calculation in the standard FFT algorithm with our FFT extension instruction as shown

	FFT Size	256	512	1K	2K	4K	8K	16K	32K	64K	128K
NutShell(vanilla)	time	6.28ms	16.64ms	34.29ms	77.80ms	168.85ms	369.76ms	613.58ms	1.47s	3.22s	6.76s
	energy	2.39mJ	6.34mJ	13.07mJ	29.64mJ	64.33mJ	140.88mJ	233.77mJ	561.73mJ	1.23J	2.58J
NutShell-FFT	time	0.79ms	2.17ms	5.20ms	11.60ms	25.43ms	55.27ms	90.90ms	219.40ms	482.68ms	1.01s
	energy	329.4nJ	905.8nJ	2.17mJ	4.85mJ	10.63mJ	23.10mJ	38.00mJ	91.71mJ	201.76mJ	421.86mJ

TABLE I: Execution efficiency comparison for vanilla NutShell and NutShell-FFT (R-mode) for FFT size ranging from 256 to 128K.

in Algorithm 1 to generate the test cases. Due to the lack of compiler support for our FFT extension instructions, we create an extension instruction generator to generate FFT extension instructions in an embedded C program using inline assembly. We can count the running time of this standard FFT program with or without FFT extension instructions. In this way, we make sure that the performance difference between the benchmarks and test cases is entirely caused by our FFT extension instructions.

B. Results Analysis

We compare the proposed FFT extension instruction set for RISC-V against the plain vanilla RISC-V mainly in two aspects, namely the efficiency, and cost. We use the testing program running time as the metric for execution efficiency evaluation. In addition, we use FPGA hardware resources and energy consumption as metrics for cost evaluation.

1) Execution efficiency comparison: We conduct tests over various scales of input data ranging from 128 points to up to 128K data points. This allows us to evaluate the proposed scheme for a wide range of different IoT sensing applications that may face various amounts of incoming data, as shown in Table I below. We use two metrics to evaluate the efficiency, i.e., execution time and energy used for such execution.

Table I indicates that the speed-up as compared to the vanilla NutShell microprocessor is significant, ranging from 6.5x on large-scale data to as large as 118x on small-scale data. We also evaluate as execution energy efficiency given in Table I, which describes the energy cost of the given data size.

2) Cost evaluation: On the other hand, we also study the hardware resource investment on the FPGA development board for the proposed FFT acceleration scheme. Table II gives a side-by-side comparison of the vanilla NutShell RISC-V microprocessor, NutShell with FFT extension in V-mode, NutShell with FFT extension in R-mode, as well as what we think most relevant work [21] [23]. As we can see from Table II, the price for achieving FFT acceleration at the RISC-V instruction level is maintained with $\leq$ 6.6% power increase (V-mode), much lower than [21] [23].

	LUT	FF	BRAMs	DSP	Power
NutShell (vanilla)	16997	18754	27.5	16	381mW
NutShell-FFT (V-mode)	21697	20781	35.5	34	406mW
NutShell-FFT (R-mode)	22489	20919	51.5	34	418mW
[21]	36083	14429	-	112	1025mW
[23]	40277	19313	-	128	986mW

TABLE II: Area and power consumption cost on FPGA.

3) Comparative study: We summarize in Table III main performance characteristics. To the best of our knowledge and in our opinion, this work serves as the best solution for combining several key factors, i.e., high performance, low cost, and native RISC-V ecosystem for seamless integration. Our energy consumption is reduced by 9.5% and 1.2%, respectively compared with [21] and [23]. We furthermore compare our approach with a more recent study in [22]. Due to lack of limited experimental data from [22], we use the the same processor without FFT extension instructions as the baseline for comparison, where [22] achieves a 2.6x FFT acceleration at the cost of 27% hardware resource increase. As for our approach, it achieves a 6.5x FFT acceleration at the cost of 32.3% increase in LUT and 11.5% in FF. As a coarse measurement, this implies that this proposed work has a $\frac{6.5}{(32.3+11.5)} / \frac{2.6}{27} \approx$ 1.5x better acceleration efficiency than [22] in terms of hardware cost.

Testing Object	NutShell	NutShell-FFT	[21]	[23]
Time	0.97ms	8.2us	3.59us	3.42us
Power	381mW	406mW	1025mW	986mW
Frequency	60MHz	60MHz	160MHz	167MHz
Energy	0.37mJ	3.33uJ	3.68uJ	3.37uJ
Native to RISC-V	-	Yes	Yes	No

TABLE III: 128-bit FFT performance comparison

4) Error analysis: Without loss of generality, we use the *mean relative error* (MRE) to evaluate the correctness of the FFT calculation using the proposed extension instructions. We define the correct FFT sequence as $X = (x(1), x(2), ..., x(n))$, the sequence calculated from FFT extension instruction as $\hat{X} = (\hat{x}(1), \hat{x}(2), ..., \hat{x}(n))$, and the relative error sequence is $\varepsilon = \{\varepsilon(k)\}_1^n = \{(x(k) - \hat{x}(k))\}_1^n$. Correspondingly, the MRE is given by Equation (5) as

$$\overline{\Delta} = \frac{1}{n}\sum_{k=1}^{n}\left|\frac{\varepsilon(k)}{x(k)}\right| = \frac{1}{n}\sum_{k=1}^{n}\left|\frac{x(k)-\hat{x}(k)}{x(k)}\right|. \quad (5)$$

Since our data is a 16-bit fixed-point number for both the real part and the imaginary part, we use the Q8.4 data format with 8 bits in integer and 4 bits decimal. Table IV shows the MRE evaluation for the proposed FFT acceleration ranging from 256 points to 16K points. The error in the proposed FFT operation is mostly contributed by the error caused by precision loss from data storage mode in fixed-point number multiplication. As Table IV shows, for IoT applications commonly using 2K-8K FFT points, the MRE is as low as 0.03% and for all supported FFT points from 128 to 16K, The MREs are lower than 0.05%, which indicates that the proposed FFT acceleration design not only meets but actually exceeds actual application in practical deployments.

FFT points	512	1K	2K	4K	8K	16K
MRE(%)	0.012	0. 029	0.028	0.034	0.037	0.044

TABLE IV: FFT MRE Analysis

5) Programming interface: As a general practice, related tool-chain for developers need updating to allow software applications to take advantage of the newly introduced instructions in the proposed approach. At the assembly level, proper definition of newly added instructions and related

vector registers needs to be added, as well as corresponding extension to compilers, such as gcc. However, since we are leveraging the instruction set extension capability native to RISC-V ISA, the effort is minimal and we work with the NutShell RISC-V open source community on this regard.

VI. Conclusion

This paper proposes a fine-grained RISC-V instruction set extension scheme to enable efficient FFT acceleration at the instruction level for arbitrary number of points. We illustrate the design philosophy of such acceleration through the instruction set extension capability native to the RISC-V ISA. The experimental study shows that the proposed acceleration scheme is able to achieve significant efficiency in both execution and energy cost per FFT calculation for various payload sizes, with a low cost in hardware resources and a negligible increase in power consumption footprint. It is our belief that, with more and more RISC-V adoptions in the field of IoT, the proposed approach potentially serves as a promising solution to edge sensing applications for FFT acceleration with minimal cost and seamless RISC-V ecosystem integration.

Meanwhile, we note that there exist several areas for future improvement in the prototype implementation. First, increasing the memory bandwidth of the NutShell microprocessor would allow only one instruction to load or store data from/to vector registers. Second, using the out-of-order version of NutShell, we can leverage the idle FFTU to execute FFT instructions during memory access by multi-transmit technology, thus further reducing the total cycles required for FFT operations. Last, we are working with the NutShell RISC-V open source community to help add proper programming support to allow developers to use the proposed FFT acceleration instructions.

References

[1] P. Duhamel and M. Vetterli, "Fast fourier transforms: a tutorial review and a state of the art," *Signal Processing*, vol. 19, no. 4, pp. 259–299, 1990.

[2] G. Lammel, "The future of mems sensors in our connected world," in *2015 28th IEEE Intl. Conf. on micro electro mechanical systems (MEMS)*. IEEE, 2015, pp. 61–64.

[3] V. Shnayder, M. Hempstead, B.-r. Chen, G. W. Allen, and M. Welsh, "Simulating the power consumption of large-scale sensor network applications," in *Proc. of the 2nd Intl. Conf. on Embedded networked sensor systems*, 2004, pp. 188–200.

[4] TI, "C6000 High Performance Multicore DSPs," https://www.ti.com/sc/docs/products/dsp/c6000, 2022, [Online; accessed 12-September-2022].

[5] R.-V. O. S. Foundationl, "RISC-V International," https://riscv.org/, 2022, [Online; accessed 12-September-2022].

[6] ARM, "Advanced RISC Machine (ARM) ISA," https://www.arm.com, 2022, [Online; accessed 12-September-2022].

[7] Intel, "Instruction Set Architecture," https://www.intel.com/content/www/us/en/developer/tools/isa-extensions/overview.html, 2022, [Online; accessed 12-September-2022].

[8] AMD, "Instruction Set Architecture (ISA) Documents," https://developer.amd.com/resources/developer-guides-manuals/, 2022, [Online; accessed 12-September-2022].

[9] OSCPU, https://github.com/OSCPU/NutShell, 2022, [Online; accessed 12-September-2022].

[10] Anonymous, "Anonymous Title," https://github.com/FredericJiang/NUTSHELL_FFT, 2022, [Anonymous; accessed 12-September-2022].

[11] S. He and M. Torkelson, "Design and implementation of a 1024-point pipeline fft processor," in *Proc. of the IEEE 1998 Custom Integrated Circuits Conf. (Cat. No. 98CH36143)*. IEEE, 1998, pp. 131–134.

[12] Y.-W. Lin, H.-Y. Liu, and C.-Y. Lee, "A 1-gs/s fft/ifft processor for uwb applications," *IEEE Journal of solid-state circuits*, vol. 40, no. 8, pp. 1726–1735, 2005.

[13] S. He and M. Torkelson, "A new approach to pipeline fft processor," in *Proceedings of International Conference on Parallel Processing*. IEEE, 1996, pp. 766–770.

[14] B. M. Baas, "A low-power, high-performance, 1024-point fft processor," *IEEE J. Solid-State Circuits*, vol. 34, no. 3, pp. 380–387, 1999.

[15] S. He and M. Torkelson, "Designing pipeline fft processor for ofdm (de) modulation," in *1998 URSI Intl. Symp. on Signals, Systems, and Electronics*. IEEE, 1998, pp. 257–262.

[16] X. Chen, Y. Lei, Z. Lu, and S. Chen, "A variable-size fft hardware accelerator based on matrix transposition," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 10, pp. 1953–1966, 2018.

[17] H. Cılasun, S. Resch, Z. I. Chowdhury, E. Olson, M. Zabihi, Z. Zhao, T. Peterson, J.-P. Wang, S. S. Sapatnekar, and U. Karpuzcu, "Crafft: High resolution fft accelerator in spintronic computational ram," in *2020 57th ACM/IEEE Design Automation Conf. (DAC)*. IEEE, 2020, pp. 1–6.

[18] B. Koppelmann, P. Adelt, W. Mueller, and C. Scheytt, "Risc-v extensions for bit manipulation instructions," in *2019 29th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. IEEE, 2019, pp. 41–48.

[19] T. Fritzmann, G. Sigl, and J. Sepúlveda, "Risq-v: Tightly coupled risc-v accelerators for post-quantum cryptography," *IACR Trans. Cryptographic Hardware and Embedded Systems*, pp. 239–280, 2020.

[20] F. Schuiki, F. Zaruba, T. Hoefler, and L. Benini, "Stream semantic registers: A lightweight risc-v isa extension achieving full compute utilization in single-issue cores," *IEEE Trans. Computers*, vol. 70, no. 2, pp. 212–227, 2020.

[21] W. Hussain, X. Chen, G. Ascheid, and J. Nurmi, "A reconfigurable application-specific instruction-set processor for fast fourier transform processing," in *2013 IEEE 24th Intl. Conf. on Application-Specific Systems, Architectures and Processors*. IEEE, 2013, pp. 339–345.

[22] B. Hu, Y. Chen, and X. Zeng, "An agile instruction set extension method based on the risc-v processor," in *2021 IEEE 4th Intl. Conf. on Electronics Technology (ICET)*. IEEE, 2021, pp. 342–346.

[23] W. Hussain, F. Garzia, T. Ahonen, and J. Nurmi, "Designing fast fourier transform accelerators for orthogonal frequency-division multiplexing systems," *J. Signal Processing Sys.*, vol. 69, no. 2, pp. 161–171, 2012.

[24] A. V. Oppenheim, J. R. Buck, and R. W. Schafer, *Discrete-time Signal Processing. Vol. 2*. Upper Saddle River, NJ: Prentice Hall, 2001.

[25] Xilinx, https://www.xilinx.com/support/download.html, 2022, [Online; accessed 12-September-2022].

[26] P. Aimonen, https://code.google.com/archive/p/libfixmath/, 2012, [Online; accessed 12-September-2022].