A Flexible FPGA-based Butterfly Engine for Accelerating Signal Processing and Machine Learning

XUEYUAN LIU, RUILIN WU, and PHILIP H.W. LEONG, The University of Sydney, Australia 2006

Field-programmable gate arrays (FPGAs) have emerged as efficient accelerators for both neural network (NN) inference and digital signal processing (DSP) tasks, particularly on resource-constrained edge devices. While previous NN accelerators based on butterfly operations demonstrate significant acceleration for inference, they are inadequately suited for long sequences and lack support for bit-reversed access patterns, limiting their applicability to variable-length DSP workloads. Based on prior butterfly accelerators, this paper presents flexible butterfly engine (FlexBE), together with a co-designed NN architecture, Butterfly-based Signal Processing Net (BSPNet). The proposed system jointly supports signal pre-processing and butterfly linear (BL)-based NN inference under stringent resource constraints. FlexBE incorporates novel adaptive data switching networks, dynamic access control mechanisms, and an efficient bit-reversal module, enabling runtime reconfiguration of sequence lengths and degrees of parallelism. Implemented on an AMD ZCU104 FPGA running at 300 MHz, FlexBE computes four 32k-point fast Fourier transforms (FFTs) in approximately 15,360 clock cycles. On challenging automatic modulation classification (AMC) datasets, BSPNet achieves accuracy comparable to GPU baselines. For single-batch inference, BSPNet with FlexBE is 2.2 ∼ 3.1× faster than prior butterfly-based accelerators; on the ZCU104, the end-to-end latency achieves speedups of up to 4.92× and 2.89× compared to an Intel Core i9 CPU and an NVIDIA RTX 3090 GPU, respectively.

CCS Concepts: • **Hardware → Digital signal processing**; **Reconfigurable logic and FPGAs**; • **Computing methodologies →** *Machine learning*.

Additional Key Words and Phrases: FPGA, Neural Networks, Butterfly Operations, Signal Processing, Automatic Modulation Classification, Hardware-Software Co-design

ACM Reference Format:

Xueyuan Liu, Ruilin Wu, and Philip H.W. Leong. 2018. A Flexible FPGA-based Butterfly Engine for Accelerating Signal Processing and Machine Learning. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 28 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The interpretation of non-cooperative radio frequency (RF) signals has long been an important topic of research, with results directly applicable to sensing, cognitive radio, and spectrum management fields [18, 34]. As the bandwidth and latency requirements of wireless systems become more demanding, the need for fast and flexible hardware to support processing of RF signals increases [2, 25]. Recently, massive advances in machine learning (ML) techniques, originally developed for computer vision, have been applied to RF signals [24, 32, 35, 36, 39]. However, most of the literature has focused on offline algorithms that maximise accuracy, with little consideration for real-time implementation [36, 37].

Authors' Contact Information: Xueyuan Liu, xueyuan.liu@sydney.edu.au; Ruilin Wu, ruilin.wu@sydney.edu.au; Philip H.W. Leong, philip.leong@sydney.edu.au, The University of Sydney, Sydney, NSW, Australia 2006.

Manuscript submitted to ACM

Table 1. Feature comparison of BE baseline (BE-base) and this proposed FlexBE.

Engine Core	Transform Length Support	Bit-Reversal Support	Max. Achievable Frequency	Routability for Large P_{bu}
BE-base [11]	Limited by hardware parameter P_{bu}	Not supported (Requires external logic)	Lower (Routing-limited)	Impossible
FlexBE (Ours)	Independent of P_{bu}	Integrated efficiently	Higher (Fine-grained pipeline)	Improved

There is hence a strong need for fast hardware accelerators for these scenarios. AMC refers to the problem of identifying the modulation scheme of received radio signals from a specified set of possible modulation techniques and is the representative problem studied in this work. Due to their inherent reconfigurability and energy efficiency, FPGAs provide an excellent platform for implementing radio frequency machine learning (RFML) systems. They enable the tight integration of a software-defined radio (SDR) front-end with signal-processing and deep-learning accelerators on a single chip, which is critical for meeting the stringent latency and bandwidth requirements of wireless systems. Furthermore, many edge RFML applications demand a small area, a minimal memory footprint, and low power consumption—constraints for which FPGA technology is highly favourable. Consequently, researchers have extensively explored strategies to reduce the size, weight, and power (SWaP) of RFML implementations [8, 20, 22, 24, 49, 51]. However, for AMC tasks on edge devices, several significant challenges remain unaddressed:

- Existing FPGA-based AMC accelerators often treat RFML as a standard deep neural network (DNN) inference workload. Consequently, most optimisation efforts focus on the classifier, e.g., model compression [49], quantisation [24, 49], and dataflow acceleration [3, 20], while overlooking the computational patterns of feature extraction.
- Prior studies have focused on classifiers directly adapted from computer vision, such as convolutional neural networks (CNNs) and residual networks. They are usually trained on raw in-phase/quadrature (I/Q) samples, and achieve high intra-dataset accuracy. Unfortunately, this approach fails to achieve robust cross-dataset generalisation for modulation classification scenarios [41–43].

To address the aforementioned problems, we propose a hardware-software co-design framework, BSPNet, that combines cyclostationary features and butterfly neural networks for robust AMC. The key advantage of BSPNet is that both its feature-extraction phase and NN inference phase can be executed on the same butterfly-structured parallel hardware. We further introduce FlexBE, an enhanced butterfly engine featuring architectural improvements that boost performance and reduce hardware overhead compared to the baseline architecture by Fan et al. [11] (hereafter referred to as BE-base). The key differences between the two are summarised in Table 1. The main contributions of this work are:

- We propose FlexBE, a butterfly engine that provides on-the-fly control of transform length and parallelism for executing both FFT-based features and BL-based NN. This eliminates the major bottlenecks in BE-base by introducing new data switches and matching control logic. FlexBE delivers a 2.8 $\sim$ 26.2$\times$ speedup over BE-base with identical configuration; and remains 2.2 $\sim$ 3.1$\times$ faster when BE-base is configured for maximum throughput.
- We introduce BSPNet, a software-hardware co-design framework that enables, for the first time, cyclostationary-based AMCs on resource-constrained FPGAs. By utilising FlexBE, computational and memory costs are reduced.

For single-batch classification on an AMD Xilinx ZCU104, BSPNet achieves a 2.89× speedup compared to an NVIDIA RTX 3090 GPU.

- To facilitate reproducible research, BSPNet and FlexBE are available online and packaged with Python generators for producing synthesizable register-transfer level (RTL) codes. [1].

The remainder of this paper is organised as follows. Section 2 provides background on cyclostationary features in AMC and butterfly architectures. Section 3 details the proposed micro-architecture of our FlexBE engine. Section 4 presents the BSPNet hardware-software co-design framework. Section 5 provides a comprehensive experimental evaluation, and Section 6 concludes the paper.

2 Background and Motivation

2.1 Radio Frequency Machine Learning

There has been increasing interest in utilising FPGAs for RFML because they are commercial, off-the-shelf (COTS) devices that offer the advantages of high performance and ability to integrate an SDR with the ML, resulting in systems with low SWaP. The first RFML implementation was reported by O'Shea et al. [35] in 2016, involving the application of a CNN to synthetically generated I/Q samples for 11 different modulation types in an offline manner. Importantly, the synthetic dataset was published by the authors under the title RADIO.ML.2016.04C to facilitate comparison with other techniques. A later paper in 2018 by the same authors extended the number of classes to 24 modulation types and compared two CNN models (VGG10 and ResNet33), demonstrating excellent accuracy [36]. Scenarios with different radio impairments, as well as transfer learning in which the DNN was trained on synthetic data and tested over-the-air, were also studied. As surveyed in [8, 20], RFML implementations evolved from CPU-based offline processing to GPU-accelerated systems.

Since radio signals have high bandwidth and latency is often crucial for practical applications, interest in applying FPGAs to RFML soon followed. While GPUs offer high parallelism, FPGAs are of particular interest as they are COTS devices that offer high performance, reconfigurability, and the ability to integrate an SDR with the ML pipeline, resulting in systems with low SWaP. The first FPGA implementation utilised a Xilinx RFSoC device to integrate an SDR with the VGG10 CNN [50]. Using ternary weights, this design achieved AMC with a throughput of 488k classifications per second and a latency of 8 μs. Following these, the optimisations of RFML accelerators have focused on several strategies, including quantisation, model optimisation and parallelism. Quantisation, such as using ternary weights [49, 51], hybrid precision schemes [24], or quantisation-aware training [30], reduces the computational requirements by decreasing the precision of multiply-accumulate operations associated with DNNs. Model optimisation aims to reduce the computational load of the DNN itself and includes lightweight CNN models tailored for resource-constrained devices [10]. While effective for resources, some designs report non-trivial accuracy drops [8]. Mapping DNNs to deeply pipelined, layer-parallel data-flow accelerators improves parallelism and throughput/energy, as demonstrated by FINN-style flows in RFML [20], their recent extensions to transformers [3], and streaming CNNs [30]. Other directions involve entirely different computational models, such as neuromorphic systems. For example, Guo et al. [17] proposed an end-to-end RFML heterogeneous streaming architecture using spiking neural network (SNN).

While the vast majority of previous RFML systems have used I/Q as the primary feature, we argue that this is not a robust representation because it is not invariant to scaling and rotation [4]. Moreover, RFML accelerators have

[1] https://github.com/louisinhit/FlexBE_v0

primarily focused on improving DNN performance, with little consideration for system-level issues, such as feature pre-processing.

2.2 Cyclostationary Features for Radio Frequency Machine Learning

To address the generalisation and robustness limitations of models trained on raw I/Q samples, more robust approaches integrate cyclostationary signal processing (CSP) methods. A time series is said to be cyclostationary if its probability distribution varies periodically with time [13, 14]. This property is inherent in man-made communication signals and can be quantified through features like the spectral correlation function (SCF) [12], high-order moments [47], or cyclic cumulants [28, 41, 44]. This combination of CSP with DNNs is one solution for robust AMC. For instance, Mendis et al. [31] computed the SCF for a deep-belief network. More recently, Snoap et al. [41] integrated cyclic cumulants (CCs) with capsule networks [38] to enhance robustness. Snoap et al. [42, 43] extended this by embedding CC estimators directly into novel nonlinear NN layers, achieving high accuracy in complex modulation scenarios. In [42], feature extraction is done by taking the complex input vector, $S = \{s[0], s[1], \ldots, s[N-1]\} \in \mathbb{C}^N$, (denoted by $S = \{s[a]\}$, $a \in \{0, \ldots, N-1\}$) and raising each element to a power (e.g. $S_2 = \{s[a]^2\}, a \in \{0, 1, \ldots, N-1\}$), as shown in the left-hand column of Eq. (1), to obtain the S_2, S_4, S_6, S_8 features. $\mathcal{F}\{\cdot\}$ denotes the N-point FFT, which is applied on the resulting vectors to obtain F_2, F_4, F_6, F_8 in Eq. (1).

$$\begin{aligned} S_2 &= \{s[a]^2\}, & F_2 &= \mathcal{F}\{S_2\} \\ S_4 &= \{s[a]^4\}, & F_4 &= \mathcal{F}\{S_4\} \\ S_6 &= \{s[a]^6\}, & F_6 &= \mathcal{F}\{S_6\} \\ S_8 &= \{s[a]^8\}, & F_8 &= \mathcal{F}\{S_8\} \end{aligned} \tag{1}$$

The resulting features $\{S_2, S_4, S_6, S_8, F_2, F_4, F_6, F_8\}$ are then fed to a downstream DNN (e.g., a capsule network [38]).

From a hardware perspective, this pipeline is heterogeneous: the pre-processing stage comprises a small number of very long transforms (e.g., $4 \times 32{,}768$-point FFTs), whereas the classifier favours many short, highly parallel linear operations. Implementing them as separate accelerators duplicates buffers and increases data movement, which is problematic for edge FPGAs under tight SWaP and timing constraints. Deploying these two computationally disparate modules efficiently on a single, resource-constrained FPGA is a non-trivial co-design problem that current RFML accelerators have not been designed to address. To the best of our knowledge, Wu et al. [52] presents the first FPGA-accelerated AMC system that integrates CSP-based feature extraction with NN classification, and reports state-of-the-art (SOTA) accuracy under an intra-dataset evaluation scenario (i.e., training and testing on the same dataset).

2.3 Butterfly Computation Accelerator as Unifying Architecture

The challenge of deploying CC-based RFML on FPGAs stems from the need to efficiently execute both FFT and DNN workloads, which present distinct computational patterns. A promising direction for unifying these disparate tasks on an FPGA is the butterfly computation accelerator, first proposed for hardware acceleration in [11], designed to execute both BL and FFTs layers using a single, unified computational structure.

The butterfly computation originates from the work of Dao et al. [6], who introduced a parametrisation of linear transforms using butterfly factorisations. This allows dense, fully connected layers in NNs to be replaced with sparse, FFT-like butterfly operations that use trained weights rather than traditional twiddle factors. This duality is highly

advantageous from a hardware perspective, as it enables a single computational core to be designed for executing both standard FFTs and these specialised BL layers.

Fan et al. [11] leveraged this principle in their FABNet accelerator, which utilises a unified core—referred to in this work as the BE-base—for this purpose. However, existing implementations such as FABNet were designed for text and computer vision workloads, whose data characteristics differ substantially from those of RF signals. When applied to RFML, such architectures do not flexibly accommodate the wide range of sequence lengths encountered in practice, which can span from very short (e.g., tens) to extremely long (tens of thousands). Additionally, in radix-2 Cooley–Tukey FFTs, decimation-in-time (DIT) produces bit-reversed outputs while decimation-in-frequency (DIF) expects bit-reversed inputs; thus, natural-order I/O requires an explicit bit-reversal stage or reorder buffer [9, 54]. Although FPGA-based parallel streaming bit-reversal can reorder N items over P lanes using commutators and banked memories with conflict-free scheduling—achieving the N/P cycle bandwidth bound [5, 15, 16, 26] — These methods cannot be directly applied due to incompatibility with the addressing scheme in BE-base memory. In summary, BE-base lacks built-in support for efficient bit reversal.

To address these specific challenges, we propose FlexBE. Our architecture is designed to maintain the unified philosophy of the butterfly computation accelerator while being explicitly enhanced for the long-sequence and dynamic-length cases. FlexBE improves routability for large N, reduces latency, and enables low-overhead runtime control over both sequence length and parallelism. The design of this flexible architecture is detailed in the following section.

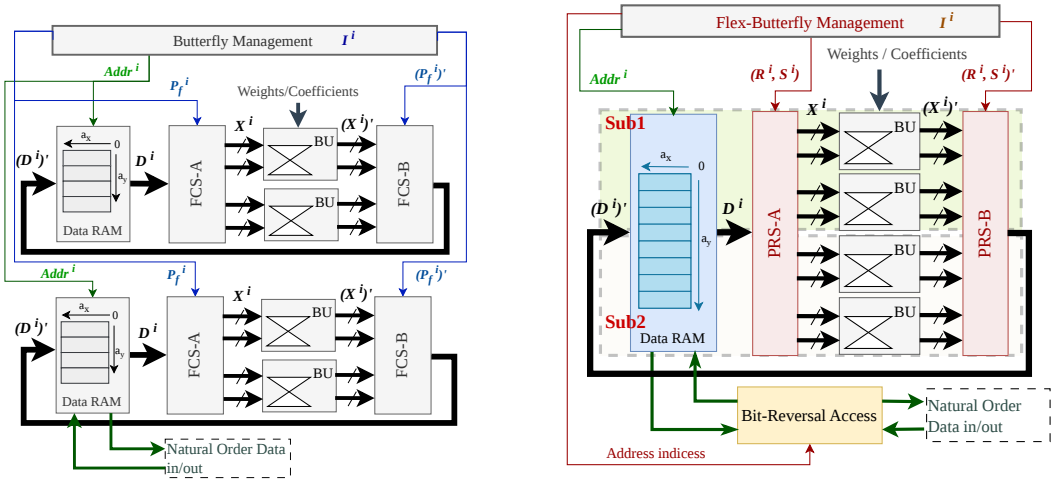
3 Flexible Butterfly Engine Architecture

The primary novelty of our accelerator lies in its data path, where we introduce a data-switching structure with integrated bit-reversal. In addition, we provide fine-grained control logic for flexible, runtime-programmable read/write behaviours. Figure 1a illustrates the BE-base data-flow, while the highlighted modules in Figure 1b denote our modifications. Here, P_{be} is the parallelism of butterfly engines (BEs), P_{bu} is the number of butterfly units (BUs) in one BE. P_{sub} is a specific parameter for FlexBE. Fully connected switch (FCS) and permute-rotate switch (PRS) are $2P_{bu}$ to $2P_{bu}$ data-switching modules: Module-A handles read reordering, and Module-B performs write recovery. All important symbols are listed in Table 2.

The i-th iteration of an N-point transform is described by Eq. (2) with the following steps:

$$\begin{aligned}
\text{Read data vector from } 2P_{bu} \text{ parallel banks:} &\quad \mathbf{D}^i \leftarrow \mathrm{MEM}(\mathrm{Addr}^i) \\
\text{Permute data via FCS (matrix } \mathbf{P}_f^i\text{):} &\quad \mathbf{X}^i \leftarrow \mathbf{P}_f^i \times \mathbf{D}^i \\
\text{Compute } P_{bu} \text{ radix-2 butterflies:} &\quad (\mathbf{X}^i)' \leftarrow \mathbf{B}(\mathbf{X}^i, \mathrm{Coeff}^i) \\
\text{Restore data via inverse permutation:} &\quad (\mathbf{D}^i)' \leftarrow (\mathbf{P}_f^i)^{-1} \times (\mathbf{X}^i)' \\
\text{Write result back to memory:} &\quad \mathrm{MEM}(\mathrm{Addr}^i) \leftarrow (\mathbf{D}^i)'
\end{aligned} \tag{2}$$

A challenge in the design of a butterfly processor is managing data RAM access to prevent bank conflicts [40], a classic and well-studied problem in parallel FFT processor design [15, 23]. To solve this, BE-base employs a specialised "shift-down" storage scheme. This strategy implements the well-established "skewed storage" or "linear address transformation" techniques [21, 48], originally developed to guarantee conflict-free memory access in dedicated FFT hardware. In this



(a) BE-base main datapath, $P_{be} = 2, P_{bu} = 2$.

(b) FlexBE main datapath, $P_{be} = 1, P_{bu} = 4, P_{sub} = 2$.

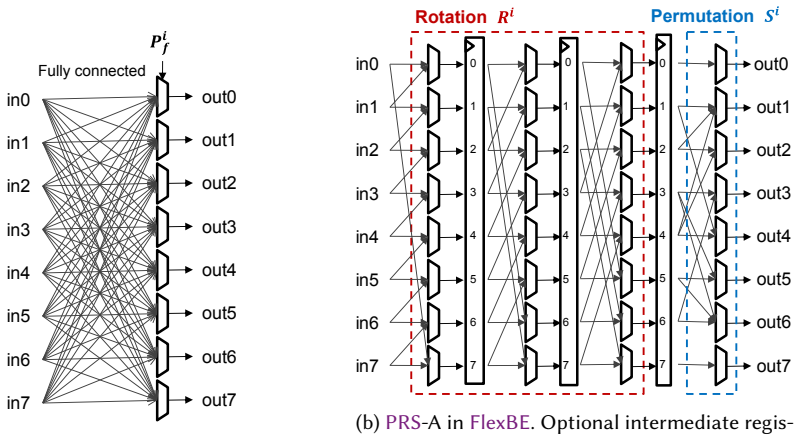
Fig. 1. Butterfly engine datapaths. Thick lines represent data paths, while thin lines denote addresses and control indices. Both designs contain a cycle-by-cycle management module, a RAM array for in-flight data storage, two read/write switching modules, and P_{bu} radix-2 BUs.

Table 2. Table of Symbols

Symbol	Meaning
$i \in [0, \frac{N}{2P_{bu}} \times n)$	Iterations (cycle number)
$n = \log_2 N$	Number of butterfly stages
$m = \log_2(2P_{bu})$	Permutation address bit-width
$\mathbf{I}^i \in \mathbb{N}^{2P_{bu}\times 1}$	Butterfly indices
$I = \text{n'b}\{b_{n-1}b_{n-2}...b_0\}$	One index and its binary representation
$\text{Addr}^i \in \mathbb{N}^{2P_{bu}\times 1}$	Accessed data address for data RAM
$\mathbf{D}^i \in \mathbb{R}^{2P_{bu}\times 1}$	Data out from data RAM
$\mathbf{P}_f^i \in \{0,1\}^{2P_{bu}\times 2P_{bu}}$	Permutation bit array in FCS
$\mathbf{X}^i \in \mathbb{R}^{2P_{bu}\times 1}$	Data after reorder
$\mathbf{B}(\cdot)$	Radix-2 butterfly operation

"shift-down", element with index I is stored in a_y-th RAM bank at depth a_x (as given in Figure 1):

$$\begin{aligned} a_x &= I >> m \\ a_y &= \text{bsm}(I) = [(\text{popcount}(I >> m) + (I \bmod 2^m)) \bmod 2^m] \end{aligned} \tag{3}$$



(a) FCS in BE-base.

(b) PRS-A in FlexBE. Optional intermediate registers are for timing optimisation.

Fig. 2. Comparison with example of 8-to-8 switching.

where bsm(I) is a bit-sum and modulo operation, and function popcount($\cdot$) counts the number of 1s in the binary representation of the input. According to [11], $\mathbf{I}^i$ is generated by a specific circuit, and a decode circuit produces Addr^i and $\mathbf{P}_f^i$. Their associations are:

$$\mathbf{P}_f^i[j,k] = \begin{cases} 1, & \text{if } k = \text{bsm}(\mathbf{I}^i[j]), \\ 0, & \text{otherwise,} \end{cases}$$

for $0 \le j,k < 2P_{bu}$. And the access address is computed as $\text{Addr}^i = \mathbf{P}_f^i \times (\mathbf{I}^i >> m)$.

Although these effective strategies guarantee conflict-free memory access, the BE-base architecture remains constrained by three fundamental limitations:

(1) *Minimum transform-length constraint.* In BE-base, each engine inherently processes $2P_{bu}$ values in parallel. Hence, a transform whose length is shorter than $2P_{bu}$ cannot fully utilise the datapath. To accommodate such inputs, they must be zero-padded to the minimum length of $2P_{bu}$, resulting in inefficiency.
(2) *Lack of integrated bit-reversal support.* The existing BE-base design assumes natural-order processing. It lacks an efficient internal mechanism to handle bit-reversed data ordering during both the read and write-back phases, which is critical for standard FFT computations, particularly when considering the "shift-down" storage scheme in the engines.
(3) *Quadratic scaling of the fully connected switch.* The FCS in BE-base acts as a general $2P_{bu}$-to-$2P_{bu}$ permutation network. Because each output port can select any input port, the number of switching connections grows quadratically with the number of ports. Its interconnect complexity can therefore be approximated as $C_{\text{fcs}} \propto$ data-width $\times (2P_{bu})^2$. As P_{bu} scales up, this dense, crossbar-like structure imposes severe logic and routing overhead, ultimately degrading scalability and hindering timing closure.

Our work addresses these deficiencies by introducing building blocks described in the following subsections.

3.1 Permute-Rotate Switching

The main novelty of this part is the introduction of PRS permutation structure to replace the FCS crossbar. In BE-base, the permutation matrix $\mathbf{P}_f^i$ changes each cycle i and encompasses all possible permutation orders. Consequently, the interconnect complexity of FCS follows: $C_{\text{fcs}} = \Theta(P_{bu}^2)$. This is implemented in hardware via programmable combinational logic, composed of $2P_{bu}$ instances of $2P_{bu}$-to-1 multiplexers (MUXs), as shown in Figure 2a.

Due to the properties of the butterfly transform, we observe that there exists a decomposition such that $\mathbf{P}_f^i = \mathbf{P}_s^i \times \mathbf{P}_r^i$. Here, $\mathbf{P}_r^i$ is a circular shift matrix, and $\mathbf{P}_s^i$ represents $\log_2(2P_{bu})$ specific cases out of $\frac{N}{2P_{bu}}$ kinds of possible $\mathbf{P}_f^i$ matrices. This decomposition implies that the FCS can be realised as a cascade of a barrel shifter for $\mathbf{P}_r^i$ and a simpler fixed permutation unit for $\mathbf{P}_s^i$. A structural illustration of the PRS placed in front of the BUs is shown in Figure 2b. Here, $\mathbf{P}_s^i$ and $\mathbf{P}_r^i$ can be described as:

$$\mathbf{P}_s^i[j,k] = \begin{cases} 1, & \text{if } k = [\text{bsm}(\mathbf{I}^i[j]) - \text{bsm}(\mathbf{I}^i[0])] \bmod 2^m, \\ 0, & \text{otherwise,} \end{cases}$$

$$\mathbf{P}_r^i[j,k] = \begin{cases} 1, & \text{if } k = \left(j + \text{bsm}(\mathbf{I}^i[0])\right) \bmod 2^m, \\ 0, & \text{otherwise,} \end{cases}$$

for $0 \le j,k < 2P_{bu}$. As shown in Eq. (2), the first (read) and second (write) PRS exhibit opposite behaviours. Since all permutation matrices $\mathbf{P}_r^i$, $\mathbf{P}_s^i$, and $\mathbf{P}_f^i$ are orthogonal, each inverse equals its transpose: $(\mathbf{P}_f^i)^{-1} = (\mathbf{P}_f^i)^T = (\mathbf{P}_r^i)^T \times (\mathbf{P}_s^i)^T$, the former PRS performs rotation followed by permutation, while the latter PRS first permutes and then rotates. As depicted in Figure 2, the barrel shift part requires $2mP_{bu}$ 2-to-1 MUXs, getting rid of large $2P_{bu}$-to-1 MUXs, while the subset switch part uses only m switching states with a total of $(m+1) \times 2^{m-1}$ connections. In total, the interconnect complexity of PRS is: $C_{\text{prs}} = \Theta(m \cdot P_{bu})$.

Additionally, as P_{bu} increases, additional pipeline registers may be required to meet timing constraints. The proposed PRS naturally accommodates such pipelining due to its staged architecture, as shown in Figure 2b. The approximate register cost to fully pipeline the PRS scales as $2P_{bu} \times m \times$ data_width. In contrast, pipelining an FCS necessitates decomposing each large combinational MUX into a multi-level tree of smaller MUXs. For instance, a $2P_{bu}$-to-1 switch is typically implemented as an m-level binary tree composed of

$$\frac{2P_{bu}}{2} + \frac{2P_{bu}}{4} + \cdots + 1 = 2P_{bu} - 1 \tag{4}$$

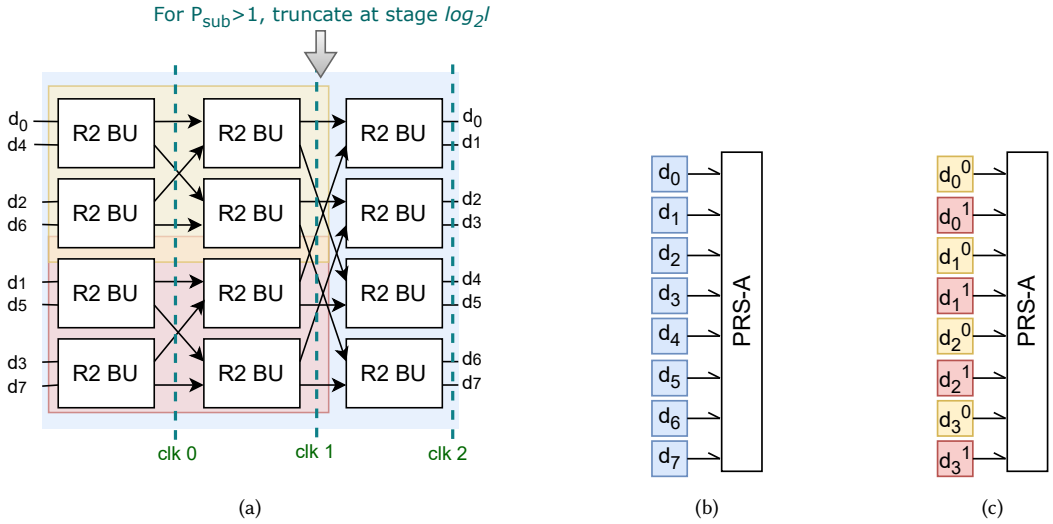
2-to-1 multiplexers per output port. Across all $2P_{bu}$ output ports, this culminates in a total of $2P_{bu}(2P_{bu}-1)$ multiplexers. Consequently, to fully pipeline the FCS structure, the required register count increases to approximately $2P_{bu} \times (2P_{bu} - 1) \times$ data_width. Therefore, pipelining the FCS can reduce logic depth, but it does not remove the underlying all-to-all routing and quadratic scaling costs.

Algorithm 1 outlines the procedure for generating the PRS control parameters for each iteration i: the rotation amount R^i that configures the barrel shifter and the permutation state S^i that drives the subset switch. The construction of the index sequence $\mathbf{I}^i$ follows the methodology of Slade [40].

Algorithm 1: Generation of PRS control signals

Input: Transform size N, number of BUs P_{bu}
Output: Rotation factor R^i and permutation state S^i

```
// Butterfly stage index
for k ← 0 to n − 1 do
    base ← 0
    // Iterations required to complete stage k
    for j ← 0 to N/(2P_bu) − 1 do
        // Global iteration index
        i ← k · N/(2P_bu) + j
        ii ← rotate_{n−1}(base, k + 1)
        I^i[0] ← rotate_n(2 · ii, k)
        // Rotation factor for the barrel shifter
        R^i ← bsm(I^i[0])
        // Permutation state for the subset switch
        if (n − m) ≤ k ≤ (n − 2) then
            S^i ← n − k − 1
        else
            S^i ← 0
        base ← base + P_bu
// rotate_n(p, q) denotes a circular left rotation of the n-bit word p by q bit positions
```



(a) (b) (c)

Fig. 3. An 8-point butterfly transform is used to perform two 4-point transforms, $P_{bu} = 4$. (a) Unfolded dataflow. (b) Case $l \geq 2P_{bu}$, $P_{sub} = 1$, no reordering required. (c) Case $l = 4$, $P_{sub} = 2$, reordering is required.

Algorithm 2: Sub-parallelism control for variable-length butterfly transforms

Input: N, P_{bu}, P_{sub}, l, data vectors $\mathbf{D}_0, \ldots, \mathbf{D}_{P_{sub}-1}$
Output: Data for computation $\mathbf{D}^i$ and butterfly state

```
// Initialisation of effective transform length
if l < 2P_bu then
    N ← 2P_bu
    P_sub ← 2P_bu / l
else
    N ← l
    P_sub ← 1
n ← log_2 N
// Prepare data for butterfly computation according to Eq. (5)
D^i ← Reorder(D_0, ..., D_{P_sub-1})
// Butterfly starts
for k ← 0 to n − 1 do
    if k = n − log_2(P_sub) then
        // Butterfly ends
        break
    base ← 0
    ...                    // Standard butterfly computation under PRS control (Algorithm 1)
```

3.2 Variable-Length and Sub-Parallel Butterfly Transform

As previously discussed, for a given parallelism P_{bu}, the BE-base architecture is incapable of processing transform lengths smaller than $2P_{bu}$. This inherent constraint significantly limits design flexibility. For instance, classic CNN architectures based on butterfly convolutions [6, 7] employ butterfly transforms ranging from 16 to 1024 points.

To address this restriction, we introduce the concept of sub-parallelism, denoted as P_{sub}. Let l denote the length of the actual input vectors and N denote the computational length within the BEs. We define P_{sub} such that if $l < 2P_{bu}$, then $N = 2P_{bu} = P_{sub} \times l$; otherwise, $N = l$ (assuming all parameters are powers of two). Intuitively, P_{sub} corresponds to the number of transforms of length l that a single P_{bu}-parallel engine can process concurrently. For example, with $P_{bu} = 16$, a single FlexBE can simultaneously perform four 8-point transforms, yielding $P_{sub} = 4$.

Figure 3a illustrates this concept, showing how two 4-point transforms (highlighted in yellow and red) are mapped onto a single 8-point datapath (blue region). Here, $d_{0\sim7}$ represents the data sequence after reordering by PRS-A. It is important to note that the operation of the PRS remains invariant to P_{sub}. By interleaving multiple short sequences to emulate a larger input vector, the hardware can process them in parallel, with the final results obtained by intercepting intermediate stages at the appropriate depth. Algorithm 2 outlines the procedure for a single FlexBE to process multiple parallel input streams, where the ellipsis represents the aforementioned butterfly control logic. Each short data vector is denoted as $\mathbf{D}_s = [d_0^s, d_1^s, \ldots, d_{l-1}^s]$, where $s \in [0, P_{sub} - 1]$. Rather than directly concatenating these vectors, they must be interleaved prior to entering PRS-A, following the mapping defined in Eq. (5). Illustrative examples of this reordering scheme are provided in Figure 3b and Figure 3c.

$$\text{Reorder}(\mathbf{D}_0, \ldots, \mathbf{D}_{P_{sub}-1}) = [d_0^0, d_0^1, d_0^2, \ldots, d_0^{P_{sub}-1}, \ \ldots, \ d_{l-1}^0, d_{l-1}^1, \ldots, d_{l-1}^{P_{sub}-1}] \tag{5}$$

3.3 Bit-Reversal Control in Flexible Butterfly Engine

To achieve the theoretical minimum latency while saving fabric resources, we integrate the bit-reversal operation directly into the write-back path from the FlexBE local RAM to the global buffer, as illustrated in Figure 4. Table 3 summarises the notation and address mappings. Both memories are organised as $P = 2P_{bu} = 2^m$ parallel lanes, each with depth $\frac{N}{P}$, where $N = 2^n$ is the total data length stored. Here, we focus on the long-transform regime $N \geq P^2$ (equivalently, $n \geq 2m$), which is typical of the targeted long-point FFT workloads and ensures that the non-overlapping bit-field decomposition below is well-defined.

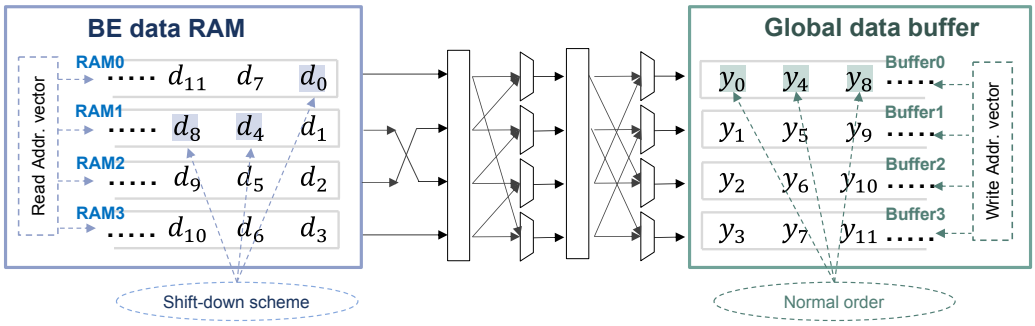


Fig. 4. Bit-reversal write-back datapath. The central network implements a source-lane-indexed destination-bank mapping.

Table 3. Address mapping between FlexBE RAM and global buffer, where $j = \text{bit-rev}_n(k)$.

Memory	Data	a_x mapping	a_y mapping
FlexBE RAM	d_j, $0 \leq j \leq N-1$	$j >> m$	$\text{bsm}(j)$
Global buffer	y_k, $0 \leq k \leq N-1$	$k >> m$	$(k \bmod 2^m)$

3.3.1 Algorithm Design. The write-back preserves the data value, so the table implies $y_k = d_j$ for $j = \text{bit-rev}_n(k)$. We synthesise the per-cycle index vectors with three objectives:

(1) Realise the global bit-reversal permutation;
(2) Avoid read and write bank conflicts in both memories;
(3) Sustain the bandwidth-limited lower bound of $\frac{N}{P}$ cycles.

We first express the FlexBE RAM index $j \in [0, N)$ in bit-sliced form:

$$j \;=\; [\, H\ (m\text{-bit}) \mid Q\ (q\text{-bit}) \mid R\ (r\text{-bit}) \mid M\ (m\text{-bit})\,], \tag{6}$$

where $q = \min\{m,\ n-2m\}$ and $r = n-2m-q \geq 0$. Thus, the most significant m bits are H, the middle field is $[Q \mid R]$ of width q+r, and the least significant m bits are M. The corresponding bit-reversed index is $k = \text{bit-rev}_n(j)$, where $\text{bit-rev}_n(\cdot)$ reverses the n-bit binary representation of an index, e.g., with $n = 3$ bits, the index $j = 1\ (001_2)$ maps to $k = 4$ (100_2).

Algorithm 3 constructs the per-cycle index vectors $(\boldsymbol{j}, \boldsymbol{k})$ such that each cycle transfers P words without bank conflicts: the inner loop over lanes $a = 0, \ldots, P-1$ guarantees a full sweep of FlexBE RAM banks per cycle; the rotation

Algorithm 3: Bit-reversal index control in FlexBE

Input: $N = 2^n$, $P = 2^m$, q, r
Output: Per-cycle index vectors $(\boldsymbol{j}, \boldsymbol{k}) \in \mathbb{N}^{P\times 1}$

```
// Outer loop over rotation offsets
for u ← 0 to 2^m − 1 do
    // Enumerate all combinations of middle bits
    for rr ← 0 to 2^r − 1 do
        for qq ← 0 to 2^q − 1 do
            for a ← 0 to P − 1 do
                b ← (a + u) & (P − 1)
                // Construct the high (m + q + r)-bit field [b | qq | rr]
                high ← (b << (q + r)) | (qq << r) | rr
                x ← (a − popcount(high)) & (P − 1)
                j[a] ← (high << m) | x
                k[a] ← bit-rev_n(j[a])
```

parameter $u \in [0, P)$ enforces a full sweep of buffer banks on the same cycle; and the middle loops over qq and rr, enumerating all middle-bit combinations to ensure full coverage of the index space.

3.3.2 Mapping to Hardware. Based on Algorithm 3, we obtain the index pair $(\boldsymbol{j}, \boldsymbol{k})$ for each clock cycle. Let $\mathbf{A}_x^{\text{be}}, \mathbf{A}_y^{\text{be}} \in \mathbb{N}^{P\times 1}$ denote the read depth address vector and the RAM index vector for the FlexBE RAM banks, respectively. These vectors are determined from the mapping defined in Table 3 using $(\boldsymbol{j}, \boldsymbol{k})$. For the read-side memory, we have:

$$\mathbf{A}_y^{\text{be}}[a] = \text{bsm}(\boldsymbol{j}[a]) = a, \qquad \mathbf{A}_x^{\text{be}}[a] = \boldsymbol{j}[a] \gg m.$$

Let $\mathbf{D}$ denote the vector of values retrieved from the RAM banks. Following the relations established above, this is expressed as:

$$\mathbf{D}[a] = \text{RAM}_a\big[\boldsymbol{j}[a] \gg m\big].$$

On the write side, the destination bank of $\mathbf{D}[a]$ is given by the lower m bits of $\boldsymbol{k}[a]$. These bits are the m-bit reversal of the upper m bits of $\boldsymbol{j}[a]$. From Algorithm 3, the latter equal $b = (a + u) \bmod P$; moreover, $\mathbf{A}_y^{\text{be}}[a] = a$. Hence, if $\mathbf{A}_y^{\text{gb}}[a]$ denotes the destination-bank label of the word read from source lane a, then

$$\mathbf{A}_y^{\text{gb}}[a] = \text{bit-rev}_m\big((\mathbf{A}_y^{\text{be}}[a] + u) \bmod P\big) = \text{bit-rev}_m\big((a + u) \bmod P\big). \tag{7}$$

To expose the correspondence between Eq. (7) and the hardware, define the element-wise mapping $\mathcal{B}_m(\mathbf{v})[a] = \text{bit-rev}_m(\mathbf{v}[a])$ and the cyclic re-indexing $\rho_u(\mathbf{v})[a] = \mathbf{v}[(a + u) \bmod P]$. Since $\mathbf{A}_y^{\text{be}} = [0, 1, \ldots, P-1]^T$, the complete destination-bank vector can be written as

$$\mathbf{A}_y^{\text{gb}} = \underbrace{\rho_u}_{\text{barrel shifter}} \Big(\underbrace{\mathcal{B}_m(\mathbf{A}_y^{\text{be}})}_{\text{permutation MUXs}} \Big). \tag{8}$$

Here, ρ_u rotates the entries of the bank-label vector; it does not bitwise-rotate an individual label. Thus, the fixed bit-reversal permutation produces the base bank-label pattern, and the barrel shifter cyclically re-indexes this pattern

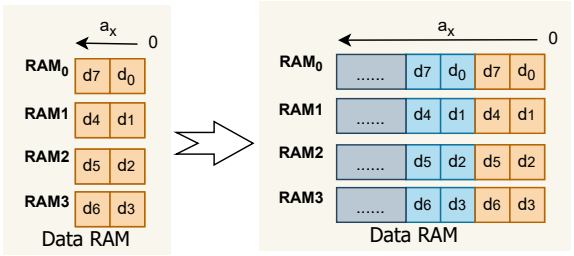


Fig. 5. Improved RAM utilisation and data alignment. Compared to BE-base, FlexBE enables multiple datasets to reside concurrently in the internal RAM banks.

according to u. The corresponding data movement is the source-indexed scatter operation

$$\mathbf{D}^{\mathrm{gb}}\left[\mathbf{A}_y^{\mathrm{gb}}[a]\right] = \mathbf{D}[a], \qquad 0 \le a < P, \tag{9}$$

which removes any ambiguity about the routing direction. An illustrative example for $P = 4$ is presented in Figure 4. While the derivation above details the implementation methodology for $\mathbf{A}_y^{\mathrm{be}}$ and $\mathbf{A}_y^{\mathrm{gb}}$, we adopt a direct approach for generating $\mathbf{A}_x^{\mathrm{be}}$ and $\mathbf{A}_x^{\mathrm{gb}}$. The access patterns for each cycle are pre-stored in ROM, utilising a simple counter to control the emission of the address vectors. Therefore, both interfaces remain conflict-free, and the bit-reversal operation completes in exactly $\frac{N}{P}$ cycles.

3.4 Optimisation and Modelling

This subsection discusses the optimisation that improves local RAM utilisation and reduces dependency on external buffering. Because FPGA block memories are allocated with fixed granularities, an engine (e.g., with $P_{bu} = 2$) requires at least four RAM banks regardless of the processed transform length N. Consequently, a configuration that stores only a single short or moderate-length sequence per acceleration step leaves a large fraction of the available RAM depth unused. To maximise the utilisation of limited on-chip resources, deeper memory configurations are preferred [19, 56].

As illustrated in Figure 5, BE-base accommodates only one length-N sequence in the local RAM array during an acceleration step. If multiple sequences must be processed, additional global or external buffers are inevitably required to stage data between the host interface and the engines. In contrast, the enhanced management module in FlexBE packs P_N independent sequences, each of length N, into the same banked RAM array. This architectural feature ensures the full utilisation of the engine RAMs and eliminates reliance on separate, external staging buffers. It is therefore highly advantageous under tight BRAM constraints, as it minimises the total memory footprint without compromising computational throughput.

For a grouped acceleration step processing P_N sequences of length N, the total number of butterfly-computation cycles can be estimated as:

$$\text{cycles} \approx \frac{P_N\, N}{2P_{sub}P_{be}P_{bu}} \times \log_2 \frac{N}{P_{sub}}. \tag{10}$$

4 Application: Automatic Modulation Classification and Butterfly-based Signal Processing Net

To assess the algorithmic and hardware performance of FlexBE on RFML tasks that require highly variable transform lengths, we introduce BSPNet, a software-hardware co-designed model that leverages FlexBE as its core accelerator. We validate the proposed architecture on the challenging `CSPB.ML.2018` [45] and `CSPB.ML.2022` [46] benchmarks for

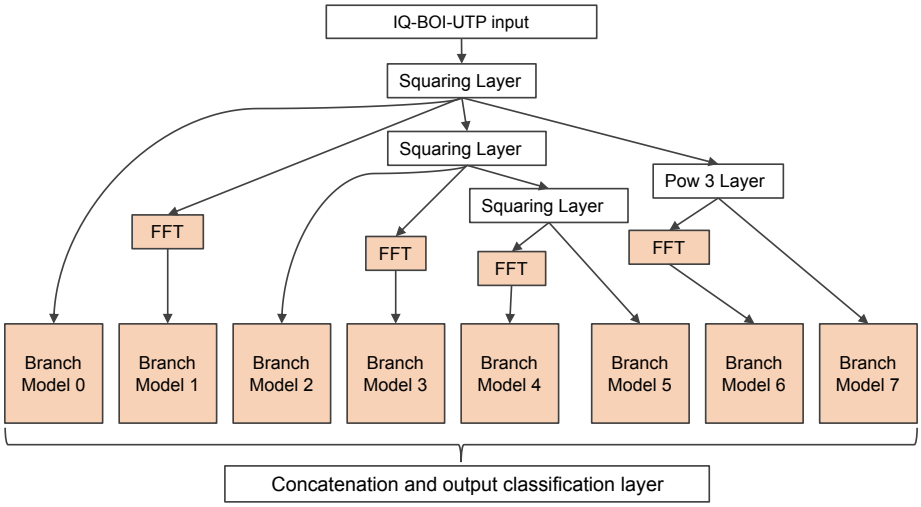


Fig. 6. BSPNet software model, adapted from CAP family. The highlighted regions indicate co-design components.

long-signal modulation classification. These datasets are particularly suitable for evaluation as they provide diverse signal parameters and extensive sample coverage, representing realistic AMC deployment scenarios.

4.1 Software Model

At the software level, BSPNet is adapted from the SOTA CAP-based network family [41–43], which exhibits superior robustness and generalisation in AMC applications. The complexity of the CAP architecture is dominated by two primary components: (i) the feature extraction layer, which computes Eq. (1) to generate features $F_{0\sim7}$ using four 32, 768-point FFTs, and (ii) eight independent classifiers, $\text{CNN}_{0\sim7}$, that process $F_{0\sim7}$ in parallel. The fundamental challenge lies in the heterogeneous computational structures of long-point FFTs and highly parallel CNNs, which complicates their deployment on resource-constrained FPGA platforms. To address this limitation, BSPNet replaces the CNN-intensive blocks with butterfly-linear layers, enabling them to share hardware resources with the FFT pre-processing stage. This co-design methodology reduces on-chip overhead while preserving model accuracy.

The topology of BSPNet is illustrated in Figure 6, with its module specifications detailed in Table 4. The butterfly-mode column indicates whether the coefficients in BUs are FFT twiddle factors or BL weights. Here, N is the input signal length, d_{in} is the input dimension of the feature-expansion transform, and d_m is the expanded model dimension. $L = N/d_{in}/8$ or $L = N/d_{in}/4$ is the input feature length before multilayer perceptron (MLP) blocks, which is reduced by a max-pooling operation. Furthermore, ReLU activation functions are integrated within each MLP block to introduce non-linearity. The parameter values for all evaluated configurations are listed explicitly in Table 7.

4.2 Hardware Architecture

Figure 7 illustrates the overall architecture of the BSPNet accelerator. It interfaces with external direct memory access (DMA) engines [57] through standard AXIS protocols, enabling efficient data streaming and straightforward system integration.

4.2.1 Workflow. The complete data processing flow operates as follows:

Table 4. BSPNet module breakdown.

Module Name		Butterfly Mode	Input Shape	Twiddle/Weights Shape	Bfly-Trans Length
CC Extraction		fft	$[4, N]$	$[2, 2, \frac{N}{2}, \log_2(N)]$	N
Branch Model 0~7	Feature Expansion	lin	$[\frac{N}{d_{in}}, d_{in}]$	$[2, 2, \frac{d_{in}}{2}, \log_2(d_{in}), \frac{d_m}{d_{in}}]$	d_{in}
	MLP-0-0	lin	$[L, d_m]$	$[2, 2, \frac{d_m}{2}, \log_2(d_m)]$	d_m
	MLP-0-1	lin	$[L, d_m]$	$[2, 2, \frac{d_m}{2}, \log_2(d_m)]$	d_m
	NormPool-0*	–	–	–	–
	MLP-1-0	lin	$[L/8, d_m]$	$[2, 2, \frac{d_m}{2}, \log_2(d_m)]$	d_m
	MLP-1-1	lin	$[L/8, d_m]$	$[2, 2, \frac{d_m}{2}, \log_2(d_m)]$	d_m
	NormPool-1	–	–	–	–
	MLP-2-0	lin	$[L/64, d_m]$	$[2, 2, \frac{d_m}{2}, \log_2(d_m)]$	d_m
	MLP-2-1	lin	$[L/64, d_m]$	$[2, 2, \frac{d_m}{2}, \log_2(d_m)]$	d_m
	NormPool-2	–	–	–	–

* The NormPool layer is defined as Layer Normalisation → Shortcut Addition → ReLU → Max Pooling. This component does not contribute much to the overall computational complexity.

- **Ingress** Incoming samples are first written to the global buffer.
- **Dispatch to BE array** When the BE arrays are ready, the buffer streams data into them.
- **After each round computation done** Outputs from BE array are steered to one of: the max pooling / shortcut addition / activation module; the BE array; the bit-reversal unit; the global buffer; or the AXIS egress.
- **Egress** Upon completion of inference, results are streamed back to the DMA; the pipeline then returns to the receive state to await the next data segment.

Each functional module is well-encapsulated. The top-level task workflow control module coordinates inter-module communication and data movement using built-in counters and finite-state machines. It aggregates status signals from each block, determines the current acceleration phase, and issues the required control signals: MUX selection signals that enable functional modules. As demonstrated in the timing diagram of Figure 8, this architecture ensures continuous utilisation of all engines throughout the processing, thereby maximising hardware efficiency and computational throughput.

4.2.2 Parameters. The complete hardware implementation is realised using RTL (Verilog HDL). Key architectural parameters, i.e. P_{be}, P_{bu}, `data_width` specifications, and the number of BU pipeline stages, directly determine the hardware resource overhead and achievable clock frequencies. These parameters can be customised according to specific application requirements and target platform constraints. They are declared through Verilog `parameter` syntax, which are fixed after instantiation. In contrast, the runtime-reconfigurable parameters are port-driven using Verilog `port list`, and can be dynamically updated by external modules or a host interface. The core latches these settings at frame boundaries, enabling reconfiguration without re-synthesis. In this BSPNet framework, they are dynamically controlled in the state machines in the top control module, as summarised in Table 5.

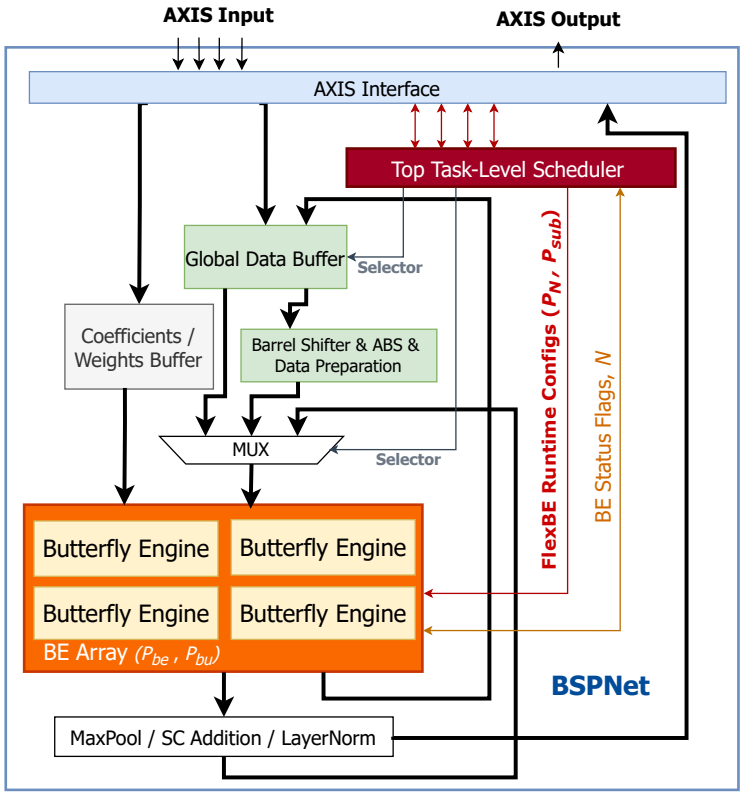


Fig. 7. Structure of the BSPNet intellectual property (IP) core. External I/O uses Advanced eXtensible Interface 4-Stream (AXIS).

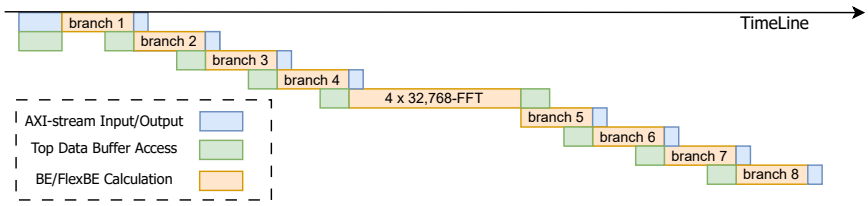


Fig. 8. Task-level pipeline behaviour of BSPNet on FPGA.

Table 5. Run-time reconfigurable parameters

BE-base	FlexBE
N	N, P_{sub}, P_N

Table 6. Specifications of the evaluated hardware platforms.

Platform	Tech. Node	Development Tools
Intel Core i9-9900KF (2019)	14 nm	PyTorch 2.1.0 / Python 3.12
NVIDIA RTX 3090 (2020)	8 nm	PyTorch 2.1.0 / CUDA 12.1
AMD Zynq ZCU104 (2018) [55]	16 nm	Vivado 2024.2 / Verilog

5 Results and Evaluation

5.1 Experimental Setup

5.1.1 Benchmarks. To demonstrate the effectiveness of FlexBE, we deploy the BSPNet model and benchmark its performance against several baselines on the platforms detailed in Table 6. The BSPNet model is first trained on GPUs to establish a software performance baseline. The evaluation focuses on both intra-dataset accuracy and cross-dataset generalisation, utilising datasets with highly diverse signal parameters and distributions. We evaluate four experimental scenarios: training on `CSPB.ML.2018` and testing on the `2018` and `2022` datasets; and training on `CSPB.ML.2022` and testing on the `2018` and `2022` datasets. For each dataset, we employ a split of 80,000 samples for training and 32,000 for testing and validation. Each sample consists of 32,768 complex-valued points representing one of eight signal classes. To align with the target hardware implementation, all numerical values are quantised to a 16-bit fixed-point (`fxp16`) format. The model is trained using the `AdamW` optimiser [29] with a learning rate of 0.0003.

5.1.2 FPGA Configurations. On ZCU104 FPGA, we compare our FlexBE implementation with two differently configured BE-bases baselines. This setup allows us to isolate and quantify the advantages of architectural improvements. To ensure a fair comparison, all configurations were designed within the same BSPNet framework shown in Figure 7, while also considering the programmable logic (PL) resource constraints. All designs maintain the same total number of BUs deployed on-chip. The three configurations are defined as follows:

- **BE-base-1**: A baseline utilising the BE-base core configured with $P_{bu} = 4$, which is inherited from FABNet. According to the available hardware resources, $P_{be} = 16$.
- **BE-base-2**: Second baseline configured with $P_{be} = 4$ and $P_{bu} = 16$. To maintain data alignment for computations with transform lengths $N < 2P_{bu}$, zero-padding is applied to the input.
- **FlexBE**: Our proposed design configured with $P_{be} = 4$ and $P_{bu} = 16$. This configuration enables a direct, fair comparison of the core engine's efficiency and flexibility with the baselines.

The dynamic network parameters used for these configurations are detailed in Table 4.

5.1.3 GPU and CPU Baselines. To provide a broader performance context, we implemented BSPNet on both CPU and GPU platforms. For the FFT stage, we utilised floating-point (`fp`) data types, as `torch.fft` does not support fixed-point types. For the classifier, the BL layers were replaced with standard `nn.Linear` layers because the existing Butterfly-Torch implementation[2] is slower on both platforms. The configuration details are:

- **CPU Baseline:** Official half-precision (`fp16`) FFT support in PyTorch is currently restricted to CUDA, so the best option was `fp32`. After evaluating multiple format combinations, the end-to-end {`fp32`-FFT, `fp32`-NN} configuration delivered the highest performance.

[2] https://github.com/HazyResearch/butterfly

Table 7. Various Configurations for BSPNet

Parameters	cfg-1	cfg-2	cfg-3	cfg-4	cfg-5	cfg-6	cfg-7	cfg-8
d_{in}	8	8	4	4	8	8	16	32
d_m	64	64	32	32	32	32	32	32
L	512	512	1024	1024	512	1024	1024	1024
# of MLP Blocks	3	2	3	3	3	3	3	3
Bit-reversal	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes

Table 8. Accuracy comparison between different models.

Model	Train on CSPB.ML.2018		Train on CSPB.ML.2022	
	Test on 2018	Test on 2022	Test on 2018	Test on 2022
CAP-based (fp) [42]	89.8%	78.2%	77.1%	86.7%
BSPNet-cfg-3 (fp)	84.4%	77.4%	81.9%	86.1%
BSPNet-cfg-6 (fp)	84.0%	77.3%	82.5%	86.8%
BSPNet-cfg-6 (fxp16)	82.3%	78.1%	81.7%	84.5%

- **GPU Baseline:** PyTorch's CUDA backend efficiently supports both fp16 and BFloat16 (bf16) formats. We compared the {fp16-FFT, fp16-NN} and {fp16-FFT, bf16-NN} configurations and observed that {fp16-FFT, fp16-NN} yielded the highest performance.

5.2 Design-Space Exploration and Accuracy Validation

To explore the trade-off between hardware performance and classification accuracy, we first conducted a design space exploration (DSE) by evaluating various BSPNet network configurations. As summarised in Table 7, we varied key parameters including input dimension (d_{in}), model dimension (d_m), feature length (L), and the presence of bit-reversal logic. Following the architecture definition, we validate the efficacy of a fxp16 quantisation scheme for hardware implementation. We adopt the Q1.15 format for all butterfly computations to maintain high precision within the $[-1.0, 1.0)$ dynamic range [33]. To ensure fidelity, we employed a bit-accurate software simulation that models the hardware datapath where multiply-accumulate operations are internally accumulated at higher precision (i.e., 32-bit) to prevent overflow, followed by convergent rounding and saturation logic. We compared the performance of the quantised BSPNet model with that of its floating-point counterparts. The validation showed a modest, scenario-dependent impact on quantisation. Finally, we benchmark the verified quantised model against the baseline CAP-based model. The accuracy results from the four experimental scenarios are presented in Table 8. It is worth noting that while Cyclo-AMC [52] reports the highest intra-dataset accuracy of 93.8% on CSPB.ML.2022, it does not report the results of CSPB.ML.2018; therefore, it is not included in Table 8. These results confirm the proposed model's effectiveness in both intra-dataset performance and cross-dataset generalisation.

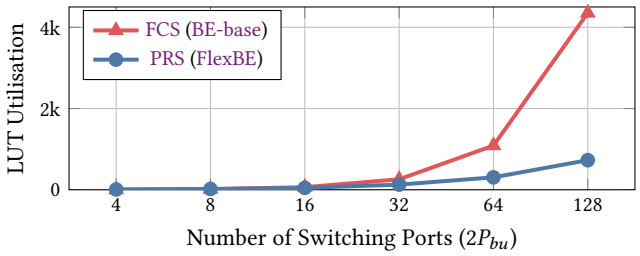


Fig. 9. Comparison of LUTs required for 1-bit switching with different numbers of ports.

5.3 Micro-architectural Evaluation

This subsection conducts microarchitecture analysis of the components that differentiate FlexBE from BE-bases. Our analysis focuses on the data-switching and addressing logic, which represent the dominant contributions in computation engines. Unlike the BUs and RAM arrays, which rely predominantly on DSPs and BRAMs, respectively, this evaluation concentrates exclusively on LUTs resource consumption. All reported results are derived from Vivado post-implementation reports.

5.3.1 PRS vs. FCS. Figure 9 provides a comparative analysis of the LUT utilisation between the baseline FCS and the proposed PRS. The PRS consistently demonstrates superior resource efficiency, consuming significantly fewer LUTs. Notably, this advantage becomes increasingly pronounced as the number of switching ports scales up. This reduction significantly alleviates overall resource pressure, preserving LUTs for other critical logic functions. Moreover, the PRS architecture offers lower logic depth than the FCS. This characteristic minimises propagation delay and facilitates timing closure on paths that are typically critical.

5.3.2 Comparison of BE Management Logic. The data switch requires a dynamic index generation module to control its routing configuration on a cycle-by-cycle basis [11]. BE-base accomplishes this using a resource-intensive, priority-encoder-based reverse lookup mechanism. This approach exhaustively searches for the mapping from a value back to its corresponding index within a large combinational block. Conversely, FlexBE eliminates this bottleneck. It leverages Algorithm 1 to efficiently compute and generate the (R_i, S_i) index pairs for the PRS, resulting in significantly lighter control logic. Figure 10 quantifies this improvement, comparing the precise resource utilisation (LUTs and FFs) of the logic modules for both BE-base and FlexBE.

5.4 Evaluation on FABNet Workloads

For a direct comparison with BE-base on the same FABNet benchmark, we assess FlexBE using the Transformer-oriented workloads introduced in [11]. To avoid the need for NN retraining, we target the same VCU128 platform, `fp16` arithmetic, and an operating frequency of 200 MHz. For this implementation, we replace the BE-base core and its associated control logic with FlexBE and the corresponding control logic. Furthermore, we follow the identical methodology to estimate latency[3].

According to the reports in [11], the original BE-base implementation on the VCU128 is configured with $P_{be} \times P_{bu} = 120 \times 4 = 480$ total BUs. As detailed in Table 9, it is bottlenecked primarily by logic and routing resources rather than

[3]Latencies are estimated using the publicly released performance simulator from BE-base: https://github.com/os-hxfan/Butterfly_Acc/tree/master/hardware/npu_design/simulator

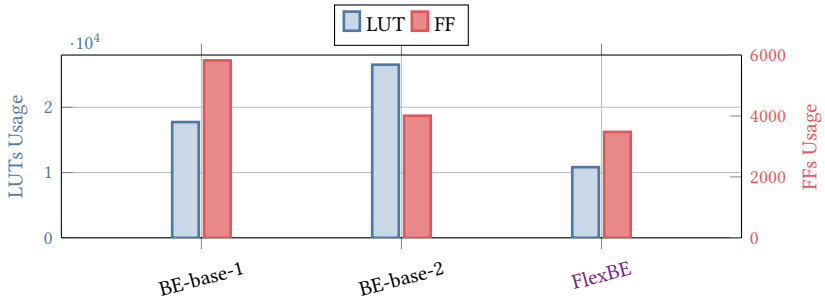


Fig. 10. Resource utilisation at the highest achievable frequency for the core management logic in BE-base-1, BE-base-2, and FlexBE. LUTs and FFs are shown using the left and right y-axes, respectively.

Table 9. Resource utilisation and latency comparison on VCU128. Both implementations use the floating-point-16 Vivado IP core and operate at 200 MHz. Execution times for FABNet tasks are estimated using the publicly released performance model from [11], assuming an efficiency factor of 0.85.

Category	Metric	BE-base (P_{bu} = 4, P_{be} = 120)	FlexBE (P_{bu} = 8, P_{be} = 76)	Speedup
Utilisation	LUTs	79.3%	75.2%	–
	FFs	63.2%	44.0%	–
	BRAMs	48.5%	60.8%	–
	DSPs	31.9%	61.1%	–
FABNet Latency (ms)	Base-128	2.69704	2.27689	1.185×
	Base-256	3.87614	3.41534	1.135×
	Base-512	6.23435	5.12301	1.217×
	Base-1024	10.95078	9.10758	1.202×
	Large-128	5.39407	4.55379	1.185×
	Large-256	7.75228	6.83068	1.135×
	Large-512	12.46871	10.24602	1.217×
	Large-1024	21.90155	18.21515	1.202×

DSP capacity, with a spatial footprint occupying 99.95% of the available Configurable Logic Block (CLB) sites. In contrast, the lower switching and control complexity of FlexBE enables a larger BU array placed on the same PL fabric. For this comparison, FlexBE is scaled to $P_{be} \times P_{bu} = 76 \times 8 = 608$ total BUs. The implemented design achieves a CLB occupancy of 99.37%. The high CLB occupancy in both designs indicates that further increasing the number of BUs is challenging.

As shown in Table 9, FlexBE results in a 1.14× ∼ 1.22× speedup across the evaluated FABNet inferences. This indicates that the benefit of FlexBE is not limited to the RFML case study. For broader NN workloads, such as FABNet, which require high parallelism rather than long sequence processing, FlexBE permits a larger number of BUs to be incorporated, yielding consistent computation-core speedups and showcasing the generality and scalability of the proposed architectural enhancements.

5.5 Algorithm and Hardware Co-design

To achieve an optimal balance between hardware efficiency and algorithmic accuracy, we adopt a co-design approach that jointly optimises the BSPNet model and its FPGA acceleration kernel. This subsection evaluates the BSPNet implementations on ZCU104 using these three BEs. We first present their Vivado reports, and subsequently, we explore

Table 10. Reports for standalone BE arrays on ZCU104 (Data format `fxp16`)

Solutions	LUTs	FFs	BRAM[1†]	BRAM[2‡]	DSP	F_{max}
BE-base-1	61, 280	76, 480	320	224	640	210 MHz
BE-base-2	140, 953	74, 046	256	128	640	115 MHz
FlexBE	61, 574	72, 372	256	128	642	310 MHz

†Utilisation under `cfg-1~4`. ‡Utilisation under `cfg-5~8`. Unmarked metrics indicate consistent utilisation across all configurations.

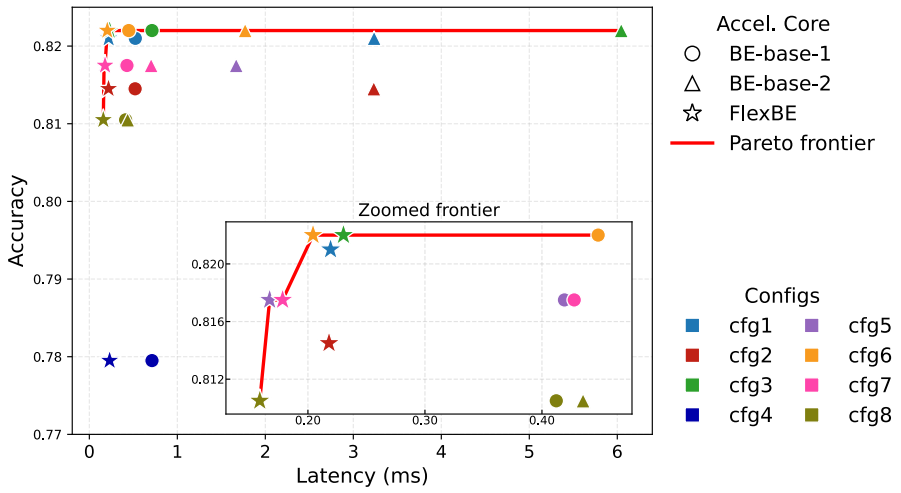


Fig. 11. Latency–accuracy trade-offs for models with different configurations. The Pareto frontier is evaluated with models trained on `CSPB.ML.2022` and tested on both `CSPB.ML.2022` and `CSPB.ML.2018`.

various BSPNet model configurations, estimate their latencies based on the timing reports, and analyse the accuracy-latency trade-offs via a Pareto plot. This analysis demonstrates the flexibility and better performance of FlexBE, while justifying our software-level configuration choices for hardware-aware optimisation.

Table 10 reports standalone BE-array results, excluding other BSPNet-level modules. Resource and F_{max} values are taken from routable implementations where the configuration fits the device; entries whose BRAM demand exceeds device capacity are reported as synthesis-level capacity estimates. Different model configurations primarily impact the BRAM overhead, as detailed in the table. Designs with `cfg-1~4` require larger buffering depths, resulting in higher BRAM consumption compared to `cfg-5~8`. The ZCU104 provides 312 BRAM blocks; hence, the 320-BRAM requirement of BE-base-1 under `cfg-1~4` is not physically implementable on this device. The results also show that, with the same number of BUs deployed on the PL, FlexBE consistently consumes comparable or fewer resources (LUTs, FFs, BRAM) than its BE-base counterparts, while achieving a significantly higher F_{max}. This efficiency stems from FlexBE's enhanced routability, which enables the flexible insertion of custom pipeline stages through additional registers within the BE structure, reducing timing bottlenecks and improving overall hardware performance.

To identify the optimal trade-off between performance and accuracy, we conduct a DSE by evaluating different BSPNet network configurations. These configurations (as summarised in Table 7) vary the key parameters listed in Table 4. For this analysis, the models are trained on `CSPB.ML.2022` dataset, and the average inference accuracy on both

CSPB.ML.2022 and CSPB.ML.2018 is used as the performance indicator for the Pareto plot. The application-level impact of this co-design is illustrated in Figure 11, which plots the accuracy-latency trade-off for all eight software configurations across the three hardware back-ends. Latency is calculated as the number of clock cycles required by BE operations (Eq. (10)) divided by the core's implemented $F_{\max}$ from Table 10. The results clearly demonstrate that FlexBE-based solutions dominate the Pareto frontier, offering superior accuracy–latency trade-offs compared to the baselines. This underscores the flexibility of our proposed BE, which adapts efficiently to diverse model configurations while delivering optimal hardware performance (e.g., lower resource usage and higher $F_{\max}$). On the software side, we strategically select configurations that align with FlexBE's strengths, such as using bit-reversal to improve accuracy without excessive hardware overhead. Notably, cfg-4 (which omits bit-reversal) exhibits significant accuracy degradation, serving as an ablation study that validates the dedicated bit-reversal module discussed in Section 3.3.

Based on the entire DSE, we selected the cfg-6 configuration as our system-level optimum, as it provides the best trade-off between inference accuracy and low latency. As detailed in Table 8, the finalised cfg-6 model remains competitive with the GPU-based CAP baseline in accuracy and generalisation, although individual scenarios exhibit differences. This result demonstrates that the hardware-oriented model retains comparable average performance while enabling efficient FPGA execution.

5.6 End-to-End Performance and Comparison

5.6.1 On-Board Implementation. Figure 12 illustrates the system-level datapath, incorporating the BSPNet-IP detailed in Figure 7. We refer to the BSPNet configured with cfg-6 as BSP-Flex. Its computational core comprises four FlexBE arrays ($P_{be} = 4$), each containing 16 BUs ($P_{bu} = 16$). On the processing system (PS) side, a Jupyter Notebook within the PYNQ framework [53] orchestrates the DMA engines to manage data transfers with the PL. To maximise input bandwidth, I/Q signals are transferred to the accelerator using the maximum number of AXIS DMA engines that can operate in parallel. As shown in Figure 6, the highlighted components are implemented on the PL. The accelerator sequentially returns the output features from the eight branches, while the final processing steps are performed on the PS. Given the minimal volume of return data, one single stream-to-memory-mapped (S2MM) DMA channel is sufficient for the write-back path. Figure 13 shows the device and the final layouts. We applied physical constraints to avoid the cross-die placement of the FlexBEs, thereby improving timing closure. Table 11 presents the post-implementation resource utilisation of the fully integrated BSP-Flex system. It should be noted that the "BE Processor" entry in Table 11 encompasses not only the standalone FlexBE core detailed in Table 10, but also the auxiliary NN processing logic, such as ReLU activations, shortcut additions, and max-pooling operations. This expanded reporting scope directly accounts for the higher overheads observed in Table 11. Due to the integration of system-level peripherals and routing overheads, the final global operating frequency is adjusted to 300 MHz, slightly lower than the standalone FlexBE peak of 310 MHz reported in Table 10.

5.6.2 Comparison with CPUs and GPUs. Figure 14 presents the inference speedup of our ZCU104 design and the RTX 3090, normalised to the Intel Core i9 CPU baseline. Here, the batch size refers to the number of samples sent to the hardware platform per transaction. On the FPGA, latency is measured as the time difference between the dma.sendchannel.transfer() and dma.recvchannel.transfer() calls [4]. This measurement captures the end-to-end round-trip latency, including all data-transfer path latency in Figure 12. For a fair comparison, the CPU and GPU

[4] https://github.com/mariodruiz/PYNQ_tutorials/blob/main/ila/notebooks/3.2.dma_working.ipynb

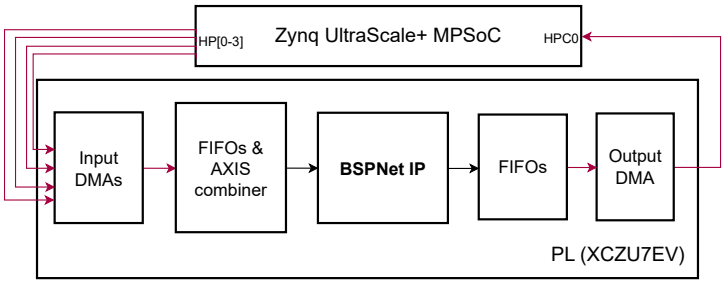
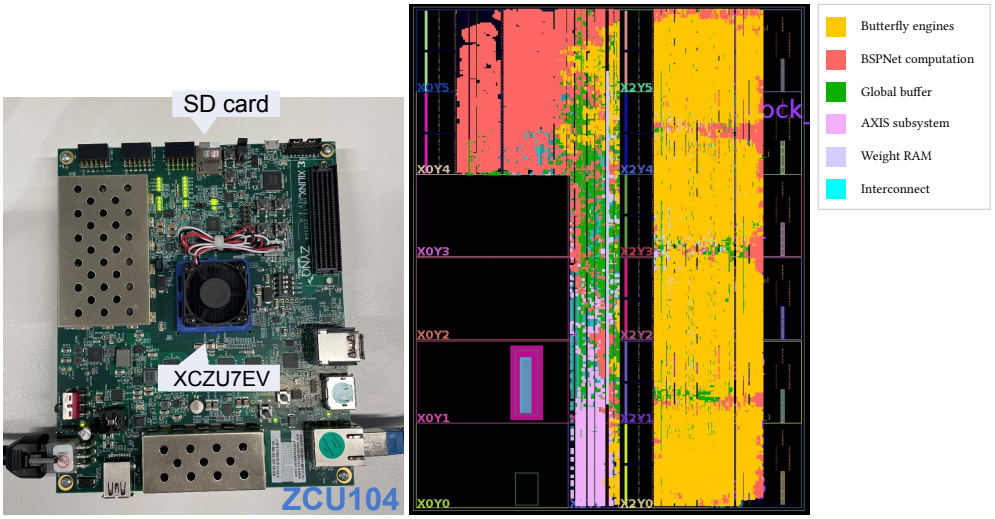


Fig. 12. Top-level block design of the system, showing the PS-PL integration. The coloured paths represent the high-latency data transfers, which add overhead compared to the on-chip PL processing.



(a) Photograph of ZCU104. (b) Implementation layout on XCZU7EV.

Fig. 13. Hardware and physical floorplan.

benchmarks are measured after an initial warm-up phase to exclude initialisation overheads. All reported values are obtained by running each benchmark multiple times and reporting the average.

As shown in Figure 14, the ZCU104 implementation provides the largest relative advantage in the small-batch regime. For a single input sample, `BSP-Flex` achieves a 4.92× speedup over the CPU baseline, whereas the RTX 3090 is 1.70×. As the batch size increases, `BSP-Flex` maintains a robust 4.05×–7.25× speedup over the CPU. However, the GPU speedup increases more rapidly with batch size, reaching 8.26× at batch size 10 and thereby exceeding the FPGA result. This trend is consistent with the throughput-oriented nature of GPU architectures. The RTX 3090 follows the SIMT execution model and relies on massive thread-level parallelism to amortise kernel launch overhead and hide memory-access latency [27]. Therefore, its efficiency improves substantially as the batch size grows. By contrast, `BSP-Flex` is optimised for low-latency stream processing, and is therefore most advantageous for real-time inference scenarios where single-sample or small-batch latency is the dominant design objective.

Table 11. Hierarchical resource utilisation of the complete BSP-Flex design at 300 MHz on ZCU104.

	LUT	LUTRAM	FF	BRAM	URAM	DSP
AXIS I/O Subsystem	12k (7.97%)	2k (17.08%)	24k (13.28%)	12.5 (4.66%)	0	0
BSPNet Top-level Control	15k (9.53%)	0	25k (13.59%)	0	0	0
Global Buffer	4k (2.64%)	0	4k (2.34%)	128 (47.67%)	0	0
Magnitude Computation	41k (26.66%)	0.8k (6.26%)	31k (16.86%)	0	0	96 (11.05%)
Weight RAM	1k (0.76%)	0	1k (0.61%)	0	64 (100.00%)	1 (0.12%)
BE Processor	80k (52.44%)	9k (76.65%)	98k (53.31%)	128 (47.67%)	0	772 (88.84%)
Total	152,553	12,278	183,826	268.5	64	869
Available	230,400	101,760	460,800	312	96	1,728

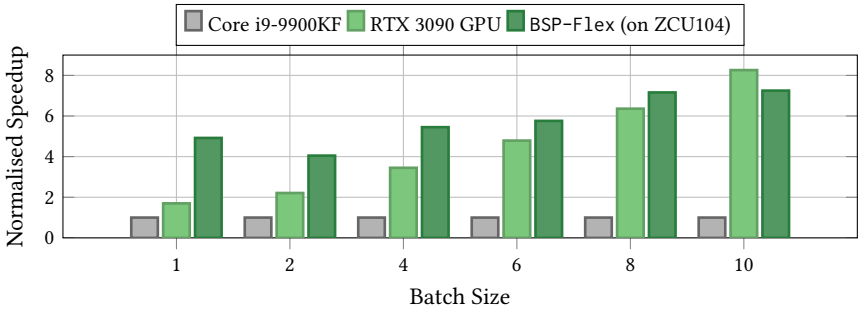
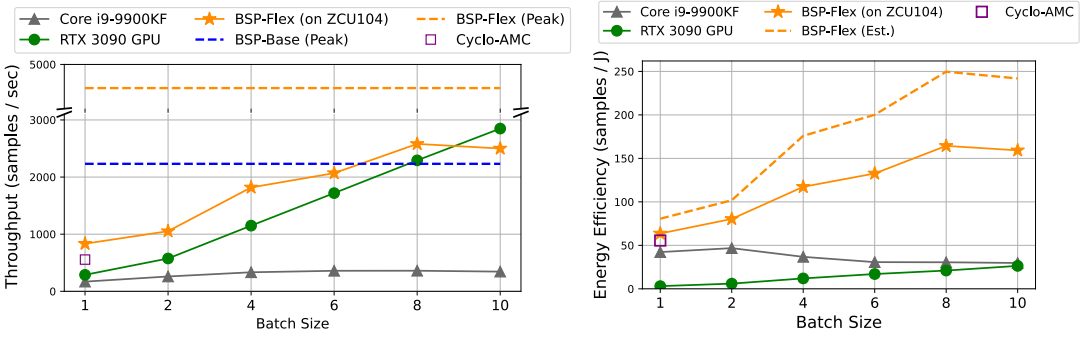


Fig. 14. End-to-end inference speedup of the ZCU104 and RTX 3090, normalised to the Intel Core i9 CPU performance, across different batch sizes.



(a) Inference throughput.

(b) Energy efficiency.

Fig. 15. Comparison results. All solid lines are based on the real measured values. Dashed lines in (a) indicate the theoretical peak throughput; in (b), it shows the tool-estimated power.

5.6.3 Comparison with other FPGA Designs. To evaluate the effectiveness of BSP-Flex, we compare its throughput against a BSPNet implementation based on BE-base-1 (denoted as BSP-Base) and an existing SOTA Cyclo-AMC accelerator [52]. Here, all designs utilise the CSPB.ML dataset family to establish a consistent baseline for comparison. The

results are presented in Figure 15a. Solid lines represent the measured on-board real throughput, which includes all system-level overheads, such as AXIS I/O and PS-PL interactions. Dashed lines indicate the theoretical peak throughput of the accelerators, estimated assuming infinite AXIS bandwidth.

As batch size increases, the latency overhead from PS-PL interactions is effectively amortised, allowing the measured throughput to converge toward the theoretical peak. The key advantage of our architecture is highlighted in the single-batch scenario, which is critical for low-latency applications. Here, our `BSP-Flex` solution achieves the highest end-to-end throughput, outperforming all other tested FPGA implementations. It is worth noting the difference in hardware platforms for this comparison. The `Cyclo-AMC` accelerator is implemented on an AMD Versal VEK280, a modern and more powerful platform featuring AIE-ML (AI Engine-Machine Learning) that is significantly less resource-constrained than our target ZCU104. Achieving superior single-batch throughput on older hardware highlights the efficiency and lightweight nature of the FlexBE datapath. Our `BSP-Flex` architecture demonstrates superior applicability and competitiveness by balancing high-performance, low-latency inference with the robust, general-purpose feature handling of the butterfly computation model.

5.6.4 Energy Efficiency. We further evaluate energy efficiency in terms of samples per Joule (samples/J), computed as throughput divided by power. For the proposed `BSP-Flex` on the ZCU104, we report two sets of efficiency figures to distinguish between theoretical estimates and board-level measured performance. The solid orange line in Figure 15b reports the measured board-level efficiency obtained through the PYNQ framework's `DataRecorder`, where the power is recorded from the ZCU104 12 V input rail using `rails['12V'].power` [5]. This measurement reflects the board-level input power and therefore includes not only the accelerator implemented in the PL, but also the PS, on-board memories and interfaces, voltage-regulator losses, and other board-level components powered through the ZCU104 power tree. The throughput is computed using the same methodology described earlier, and each configuration is executed multiple times with the average value reported. The dashed orange line illustrates the projected efficiency obtained using the AMD Vivado post-implementation total on-chip power estimate of 10.4 W. Unlike the 12 V rail measurement, this estimate reflects the power consumed internally by the implemented device logic under Vivado's activity and power-model assumptions. It is used as a device-level projection of the accelerator's energy-efficiency potential and does not include board-level regulator losses or other off-chip components.

For the CPU baseline, power is measured using the `PowerTOP` utility during steady-state inference. For the GPU baseline, power is measured using `nvidia-smi`, which reports the NVML device-level GPU power rather than full host-system power. Finally, the energy efficiency of the `Cyclo-AMC` baseline is estimated using the AMD Power Design Manager tool [1].

As shown in Figure 15b, `BSP-Flex` achieves substantially higher energy efficiency than the CPU and GPU baselines across all batch sizes. In the single-batch case, the Vivado-based estimate yields 80.62 samples/J, exceeding the estimated efficiency of `Cyclo-AMC`, which is approximately 55 samples/J. Under measured board-level power on the ZCU104, the single-batch efficiency of `BSP-Flex` is around 63 samples/J, lower than the Vivado-based estimate because the 12 V rail measurement includes non-accelerator board-level power components. As the batch size increases, the energy efficiency scales effectively, reaching an on-board peak of over 160 samples/J. The Vivado-based estimate, which more closely reflects the implemented FPGA on-chip power, highlights the accelerator's potential, peaking at approximately 250 samples/J. These results demonstrate the suitability of `BSP-Flex` for power-sensitive edge applications and show that the marginal accuracy trade-offs discussed in Section 5.5 lead to a competitive efficiency–accuracy operating point.

[5] Xilinx ZCU104 PMBus tutorial: https://github.com/Xilinx/PYNQ/blob/master/boards/ZCU104/notebooks/common/zcu104_pmbus.ipynb.

6 Conclusion

In this work, we proposed a novel butterfly-based algorithmic scheduling strategy tailored for efficient FPGA acceleration. By leveraging the specific properties of butterfly operations, we developed the `BSP-Flex` architecture, which significantly optimises data-paths and resource utilisation. Furthermore, the co-designed BSPNet model is explicitly structured to exploit the underlying hardware architecture, enabling the efficient deployment of hybrid tasks that combine CSP and NN on FPGA platforms. Experimental results demonstrate that our solution outperforms existing SOTA implementations on both general-purpose processors and advanced FPGA devices, achieving higher energy efficiency and lower hardware overhead. More importantly, while AMC serves as a representative case study in this work, the proposed framework is applicable well beyond this domain. For instance, the evaluation on standard FABNet workloads reveals that FlexBE also delivers execution speedups and improved resource efficiency across broader NN inference tasks. Consequently, it offers a scalable, viable solution for deploying signal processing and ML tasks on resource-constrained devices. Future work will investigate advanced quantisation techniques, such as Learned Step Size Quantisation, to further reduce the accuracy loss observed when moving from floating-point to 16-bit fixed-point arithmetic.

Acknowledgments

This work was supported by the AMD University Program (AUP) and the University of Sydney Nano Institute. The authors acknowledge the use of `Google Gemini` for assistance with academic language editing and sentence refinement. The authors have verified the generated text and take full responsibility for the content of this work.

References

[1] AMD Xilinx Inc. 2025. Power Design Manager (PDM). https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/power-design-manager.html. Accessed: Dec. 2025.

[2] Jeffrey G. Andrews, Stefano Buzzi, Wan Choi, Stephen V. Hanly, Angel Lozano, Anthony C. K. Soong, and Jianzhong Charlie Zhang. 2014. What Will 5G Be? *IEEE Journal on Selected Areas in Communications* 32, 6 (2014), 1065–1082. doi:10.1109/JSAC.2014.2328098

[3] Christoph Berganski, Felix Jentzsch, Marco Platzner, Max Kuhmichel, and Heiner Giefers. 2024. FINN-T: Compiling Custom Dataflow Accelerators for Quantized Transformers. In *2024 International Conference on Field Programmable Technology (ICFPT)*. IEEE, Sydney, Australia, 01–10. doi:10.1109/ICFPT64416.2024.11113391

[4] Rudrasis Chakraborty, Yifei Xing, and Stella X. Yu. 2022. SurReal: Complex-Valued Learning as Principled Transformations on a Scaling and Rotation Manifold. *IEEE Transactions on Neural Networks and Learning Systems* 33, 3 (2022), 940–951. doi:10.1109/TNNLS.2020.3030565

[5] Sau-Gee Chen, Shen-Jui Huang, Mario Garrido, and Shyh-Jye Jou. 2014. Continuous-flow Parallel Bit-Reversal Circuit for MDF and MDC FFT Architectures. *IEEE Transactions on Circuits and Systems I: Regular Papers* 61, 10 (2014), 2869–2877. doi:10.1109/TCSI.2014.2327271

[6] Tri Dao, Albert Gu, Matthew Eichhorn, Atri Rudra, and Christopher Re. 2019. Learning Fast Algorithms for Linear Transforms Using Butterfly Factorizations. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, Long Beach, California, USA, 1517–1527. https://proceedings.mlr.press/v97/dao19a.html

[7] Tri Dao, Nimit S. Sohoni, Albert Gu, Matthew Eichhorn, Amit Blonder, Megan Leszczynski, Atri Rudra, and Christopher Ré. 2020. Kaleidoscope: An Efficient, Learnable Representation for All Structured Linear Maps. In *Proceedings of the 8th International Conference on Learning Representations*. OpenReview.net, Addis Ababa, Ethiopia. https://openreview.net/forum?id=BkgrBgSYDS

[8] H. den Boer, R.W.D. Muller, S. Wong, and V. Voogt. 2021. FPGA-based Deep Learning Accelerator for RF Applications. In *MILCOM 2021 - 2021 IEEE Military Communications Conference (MILCOM)*. IEEE, San Diego, CA, USA, 751–756. doi:10.1109/MILCOM52596.2021.9652891

[9] Konguvel Elango and Kannan Muniandi. 2023. A novel digital logic for bit reversal and address generations in FFT computations. *Wireless Personal Communications* 128, 3 (2023), 1827–1838.

[10] A. Emad, H. Mohamed, A. Farid, M. Hassan, R. Sayed, H. Aboushady, and H. Mostafa. 2021. Deep Learning Modulation Recognition for RF Spectrum Monitoring. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, Daegu, South Korea, 1–5. doi:10.1109/ISCAS51556.2021.9401658

[11] Hongxiang Fan, Thomas Chau, Stylianos I. Venieris, Royson Lee, Alexandros Kouris, Wayne Luk, Nicholas D. Lane, and Mohamed S. Abdelfattah. 2022. Adaptable Butterfly Accelerator for Attention-based NNs via Hardware and Algorithm Co-design. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, Chicago, Illinois, USA, 599–615. doi:10.1109/MICRO56248.2022.00050

[12] W. Gardner. 1987. Spectral Correlation of Modulated Signals: Part I - Analog Modulation. *IEEE Transactions on Communications* 35, 6 (1987), 584–594. doi:10.1109/TCOM.1987.1096820

[13] W.A. Gardner and C.M. Spooner. 1993. Detection and source location of weak cyclostationary signals: simplifications of the maximum-likelihood receiver. *IEEE Transactions on Communications* 41, 6 (1993), 905–916. doi:10.1109/26.231913

[14] William A. Gardner. 1994. *Cyclostationarity in Communications and Signal Processing*. IEEE Press, New York, NY, USA. A foundational reference on cyclostationary theory for signal processing applications.

[15] Mario Garrido. 2022. A survey on pipelined FFT hardware architectures. *Journal of Signal Processing Systems* 94, 11 (2022), 1345–1364.

[16] Mario Garrido, J. Grajal, M. A. Sanchez, and Oscar Gustafsson. 2013. Pipelined Radix-2^k Feedforward FFT Architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21, 1 (2013), 23–32. doi:10.1109/TVLSI.2011.2178275

[17] Wenzhe Guo, Kuilian Yang, Haralampos-G. Stratigopoulos, Hassan Aboushady, and Khaled Nabil Salama. 2024. An End-To-End Neuromorphic Radio Classification System With an Efficient Sigma-Delta-Based Spike Encoding Scheme. *IEEE Transactions on Artificial Intelligence* 5, 4 (2024), 1869–1881. doi:10.1109/TAI.2023.3306334

[18] S. Haykin. 2005. Cognitive radio: brain-empowered wireless communications. *IEEE Journal on Selected Areas in Communications* 23, 2 (2005), 201–220. doi:10.1109/JSAC.2004.839380

[19] Intel Corporation. 2024. *Intel® FPGA IP: Internal Memory (RAM and ROM) User Guide*. Intel Corporation. https://www.intel.com/programmable/technical-pdfs/654378.pdf Accessed: 2025-06-08.

[20] Felix Jentzsch, Yaman Umuroglu, Alessandro Pappalardo, Michaela Blott, and Marco Platzner. 2022. RadioML Meets FINN: Enabling Future RF Applications With FPGA Streaming Architectures. *IEEE Micro* 42, 6 (2022), 125–133. doi:10.1109/MM.2022.3202091

[21] L.G. Johnson. 1992. Conflict free memory addressing for dedicated FFT hardware. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 39, 5 (1992), 312–316. doi:10.1109/82.142032

[22] Kuchul Jung, Jongseok Woo, and Saibal Mukhopadhyay. 2022. An On-chip Accelerator with Hybrid Machine Learning for Modulation Classification of Radio Frequency Signals. In *2022 IEEE/MTT-S International Microwave Symposium - IMS 2022*. IEEE, Denver, Colorado, USA, 487–490. doi:10.1109/IMS37962.2022.9865378

[23] Zeynep Kaya, Mario Garrido, and Jarmo Takala. 2023. Memory-Based FFT Architecture With Optimized Number of Multiplexers and Memory Usage. *IEEE Transactions on Circuits and Systems II: Express Briefs* 70, 8 (2023), 3084–3088. doi:10.1109/TCSII.2023.3245823

[24] Satish Kumar, Rajarshi Mahapatra, and Anurag Singh. 2022. Automatic Modulation Recognition: An FPGA Implementation. *IEEE Communications Letters* 26, 9 (2022), 2062–2066. doi:10.1109/LCOMM.2022.3184771

[25] Matti Latva-Aho, Kari Leppänen, et al. 2019. *Key drivers and research challenges for 6G ubiquitous wireless intelligence*. Technical Report. University of Oulu, 6G Flagship.

[26] Weijun Li, Feng Yu, and Zhenguo Ma. 2016. Efficient Circuit for Parallel Bit Reversal. *IEEE Transactions on Circuits and Systems II: Express Briefs* 63, 4 (2016), 381–385. doi:10.1109/TCSII.2015.2504943

[27] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. 2008. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro* 28, 2 (2008), 39–55. doi:10.1109/MM.2008.31

[28] Xueyuan Liu, Carol Jingyi Li, Craig T. Jin, and Philip H. W. Leong. 2022. Wireless Signal Representation Techniques for Automatic Modulation Classification. *IEEE Access* 10 (2022), 84166–84187. doi:10.1109/ACCESS.2022.3197224

[29] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *International Conference on Learning Representations*. OpenReview.net, New Orleans, Louisiana, USA. https://openreview.net/forum?id=Bkg6RiCqY7

[30] Andrew Maclellan, Louise H. Crockett, and Robert W. Stewart. 2025. RFSoC Modulation Classification With Streaming CNN: Data Set Generation & Quantized-Aware Training. *IEEE Open Journal of Circuits and Systems* 6 (2025), 38–49. doi:10.1109/OJCAS.2024.3509627

[31] Gihan J. Mendis, Jin Wei, and Arjuna Madanayake. 2016. Deep learning-based automated modulation classification for cognitive radio. In *2016 IEEE International Conference on Communication Systems (ICCS)*. IEEE, Shenzhen, China, 1–6. doi:10.1109/ICCS.2016.7833571

[32] Fan Meng, Peng Chen, Lenan Wu, and Xianbin Wang. 2018. Automatic modulation classification: A deep learning enabled approach. *IEEE Transactions on Vehicular Technology* 67, 11 (2018), 10760–10772. doi:10.1109/TVT.2018.2868698

[33] Uwe Meyer-Baese. 2007. *Digital Signal Processing with Field Programmable Gate Arrays* (3rd ed.). Springer, Berlin, Heidelberg. See Chapter 2 for detailed discussions on fixed-point arithmetic and Q-formats.

[34] J. Mitola and G.Q. Maguire. 1999. Cognitive radio: making software radios more personal. *IEEE Personal Communications* 6, 4 (1999), 13–18. doi:10.1109/98.788210

[35] Timothy J O'Shea, Johnathan Corgan, and T Charles Clancy. 2016. Convolutional radio modulation recognition networks. In *Engineering Applications of Neural Networks: 17th International Conference, EANN 2016, Aberdeen, UK, September 2-5, 2016, Proceedings 17*. Springer, Springer, Aberdeen, UK, 213–226.

[36] Timothy James O'Shea, Tamoghna Roy, and T. Charles Clancy. 2018. Over-the-Air Deep Learning Based Radio Signal Classification. *IEEE Journal of Selected Topics in Signal Processing* 12, 1 (2018), 168–179. doi:10.1109/JSTSP.2018.2797022

[37] Francesco Restuccia, Salvatore D'Oro, Amani Al-Shawabka, Mauro Belgiovine, Luca Angioloni, Stratis Ioannidis, Kaushik Chowdhury, and Tommaso Melodia. 2019. DeepRadioID: Real-Time Channel-Resilient Optimization of Deep Learning-based Radio Fingerprinting Algorithms. In *Proceedings of the Twentieth ACM International Symposium on Mobile Ad Hoc Networking and Computing* (Catania, Italy) *(Mobihoc '19)*. Association for Computing Machinery, New York, NY, USA, 51–60. doi:10.1145/3323679.3326503

[38] Sara Sabour, Nicholas Frosst, and Geoffrey E. Hinton. 2017. Dynamic routing between capsules. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 3859–3869.

[39] Stefan Scholl. 2019. Classification of Radio Signals and HF Transmission Modes with Deep Learning. arXiv:1906.04459 [eess.SP] https://arxiv.org/abs/1906.04459

[40] George Slade. 2013. The Fast Fourier Transform in Hardware: A Tutorial Based on an FPGA Implementation. Online tutorial / manuscript.

[41] John A. Snoap, James A. Latshaw, Dimitrie C. Popescu, and Chad M. Spooner. 2022. Robust Classification of Digitally Modulated Signals Using Capsule Networks and Cyclic Cumulant Features. In *MILCOM 2022 - 2022 IEEE Military Communications Conference (MILCOM)*. IEEE, Rockville, MD, USA, 298–303. doi:10.1109/MILCOM55135.2022.10017507

[42] John A. Snoap, Dimitrie C. Popescu, and Chad M. Spooner. 2023. Novel Nonlinear Neural-Network Layers for High Performance and Generalization in Modulation-Recognition Applications. In *MILCOM 2023 - 2023 IEEE Military Communications Conference (MILCOM)*. IEEE, Boston, MA, USA, 562–567. doi:10.1109/MILCOM58377.2023.10356331

[43] John A. Snoap, Dimitrie C. Popescu, and Chad M. Spooner. 2024. Deep-Learning-Based Classifier With Custom Feature-Extraction Layers for Digitally Modulated Signals. *IEEE Transactions on Broadcasting* 70, 3 (2024), 763–773. doi:10.1109/TBC.2024.3391056

[44] C.M. Spooner. 1995. Classification of co-channel communication signals using cyclic cumulants. In *Conference Record of The Twenty-Ninth Asilomar Conference on Signals, Systems and Computers*, Vol. 1. IEEE, Pacific Grove, CA, USA, 531–536 vol.1. doi:10.1109/ACSSC.1995.540605

[45] Chad M. Spooner. 2019. Dataset for the Machine-Learning Challenge [CSPB.ML.2018]. https://cyclostationary.blog/2019/02/15/data-set-for-the-machine-learning-challenge/ Accessed: 16 May 2022.

[46] Chad M. Spooner. 2022. Shifted Dataset for the Machine-Learning Challenge: How Well Does a Modulation-Recognition DNN Generalize? [Dataset CSPB.ML.2022]. https://cyclostationary.blog/2022/01/23/shifted-dataset-for-the-machine-learning-challenge-how-well-does-a-modulation-recognition-dnn-generalize-100th-csp-blog-post/ Accessed: 16 May 2022.

[47] A. Swami and B.M. Sadler. 2000. Hierarchical digital modulation classification using cumulants. *IEEE Transactions on Communications* 48, 3 (2000), 416–429. doi:10.1109/26.837045

[48] J.H. Takala, T.S. Jarvinen, and H.T. Sorokin. 2003. Conflict-free parallel memory access scheme for FFT processors. In *Proceedings of the 2003 International Symposium on Circuits and Systems, 2003. ISCAS '03.*, Vol. 4. IEEE, Bangkok, Thailand, IV–IV. doi:10.1109/ISCAS.2003.1205957

[49] Stephen Tridgell, David Boland, Philip H.W. Leong, Ryan Kastner, Alireza Khodamoradi, and Siddhartha. 2020. Real-time Automatic Modulation Classification using RFSoC. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, New Orleans, LA, USA, 82–89. doi:10.1109/IPDPSW50202.2020.00021

[50] Stephen Tridgell, David Boland, Philip H.W. Leong, and Siddhartha Siddhartha. 2019. Real-Time Automatic Modulation Classification. In *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, Tianjin, China, 299–302. doi:10.1109/ICFPT47387.2019.00052

[51] Jongseok Woo, Kuchul Jung, and Saibal Mukhopadhyay. 2024. Efficient Hardware Design of DNN for RF Signal Modulation Recognition Employing Ternary Weights. *IEEE Access* 12 (2024), 80165–80175. doi:10.1109/ACCESS.2024.3409180

[52] Ruilin Wu, Carol Jingyi Li, Wei Zhang, Xueyuan Liu, and Philip H.W. Leong. 2025. Cyclo-AMC: Automatic Modulation Classification on Versal utilising Cyclostationary Features. In *2025 International Conference on Field Programmable Technology (ICFPT)*. IEEE, Shanghai, China, 91–100. doi:10.1109/ICFPT67023.2025.00021

[53] Xilinx. 2018. PYNQ Documentation: Jupyter Notebooks. https://pynq.readthedocs.io/en/v2.3/jupyter_notebooks.html.

[54] Xilinx. 2022. Fast Fourier Transform v9.1: LogiCORE IP Product Guide (PG109). https://www.xilinx.com/support/documents/ip_documentation/xfft/v9_1/pg109-xfft.pdf Version 9.1, May 4, 2022. Accessed 2025-09-02.

[55] Xilinx Inc. 2019. *ZCU104 Board User Guide (UG1267)*. Xilinx Inc. https://docs.amd.com/v/u/en-US/ug1267-zcu104-eval-bd

[56] Xilinx Inc. 2023. *7 Series FPGAs Memory Resources User Guide*. Xilinx Inc. https://docs.xilinx.com/v/u/en-US/ug473_7Series_Memory_Resources Accessed: 2025-06-08.

[57] Xilinx Inc. 2023. *AXI Direct Memory Access (AXI DMA) v7.1 Product Guide*. Xilinx Inc. https://www.amd.com/content/dam/xilinx/support/documents/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf Accessed: 2025-06-13.

Received Dec 2025