A Cost-Efficient Digital ESN Architecture on FPGA

Victor M. Gan

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

Yang Yi, Chair

Haibo Zeng

Peter M. Athanas

July 6, 2020

Blacksburg, Virginia

Keywords: reservoir computing, echo state network, field-programmable gate array, FPGA design, DSP48, symbol detection, OFDM, wireless communication

Copyright 2020, Victor M. Gan

A Cost-Efficient Digital ESN Architecture on FPGA

Victor M. Gan

(ABSTRACT)

Echo State Network (ESN) is a recently developed machine-learning paradigm whose processing capabilities rely on the dynamical behavior of recurrent neural networks (RNNs). Its performance metrics outperform traditional RNNs in nonlinear system identification and temporal information processing. In this thesis, we design and implement ESNs through Field-programmable gate array (FPGA) and explore their full capacity of digital signal processors (DSPs) to target low-cost and low-power applications. We propose a cost-optimized and scalable ESN architecture on FPGA, which exploits Xilinx DSP48E1 units to cut down the need of configurable logic blocks (CLBs). The proposed work includes a linear combination processor with negligible deployment of CLBs, as well as a high-accuracy non-linear function approximator, both with the help of only 9 DSP units in each neuron. The architecture is verified with the classical NARMA dataset, and a symbol detection task for an orthogonal frequency division multiplexing (OFDM) system on a wireless communication testbed. In the worst-case scenario, our proposed architecture delivers a matching bit error rate (BER) compares to its corresponding software ESN implementation. The performance difference between the hardware and software approach is less than 6.5%. The testbed system is built on a software-defined radio (SDR) platform, showing that our work is capable of processing the real-world data.

A Cost-Efficient Digital ESN Architecture on FPGA

Victor M. Gan

(GENERAL AUDIENCE ABSTRACT)

Machine learning is a study of computer algorithms that evolves itself by learning through experiences. Currently, machine learning thrives as it opens up promising opportunities of solving the problems that is difficult to deal with conventional methods. Echo state network (ESN), a recently developed machine-learning paradigm, has shown extraordinary effectiveness on a wide variety of applications, especially in nonlinear system identification and temporal information processing. Despite the fact, ESN is still computationally expensive on battery-driven and cost-sensitive devices. A fast and power-saving computer for ESN is desperately needed. In this thesis, we design and implement an ESN computational architecture through the field-programmable gate array (FPGA). FPGA allows designers to build highly flexible customized hardware with rapid development time. Our design further explores the full capacity of digital signal processors (DSP) on Xilinx FPGA to target low-cost and low-power applications. The proposed cost-optimized and scalable ESN architecture exploits Xilinx DSP48E1 units to cut down the need of configurable logic blocks (CLBs). The work includes a linear combination processor with negligible deployment of CLBs, and a high-accuracy non-linear function approximator, both with the help of only 9 DSP units in each neuron. The architecture is verified with the classical NARMA dataset, and a symbol detection task for an orthogonal frequency division multiplexing (OFDM) system in a wireless communication testbed. In the worst-case scenario, our proposed architecture delivers a matching bit error rate (BER) compares to its corresponding software ESN implementation. The performance difference between the hardware and software approach is less than 6.5%. The testbed system is

built on a software-defined radio (SDR) platform, showing that our work is capable of processing the real-world data.

Acknowledgments

The completion of the thesis would not have been possible without Dr. Yang Yi's constructive advice during planning and establishing of this research work. I would also like to extend my deepest gratitude to Dr. Haibo Zeng, Dr. Peter Athanas, and Dr. Dong Ha for their inspiring guidance throughout my graduate career. Great thanks are given to Yibin Liang for his generous sharing of knowledge of FPGA, and to Lianjun Li for his technical support on wireless communication. I also cannot leave Virginia Tech without mentioning all my extraordinary labmates in MICS, Hongyu An, Kangjun Bai, Kian Hamedani, Shiya Liu, Qiyuan An, Jiayuan Zhang, Moqi Zhang, Fabiha Nowshin, Jinhua Wang, and Jiayu Li, for their professional assistance and invaluable encouragement in the past two years. Finally, special thanks should also go to my family and my close friends Hao-Hsuan Chang, Ruici Lin, and Ningyuan Liu for their company and supports during the unforeseen quarantine life amid COVID-19.

Contents

List of Figures viii

List of Tables x

1 Introduction 1

1.1 Motivation 1

1.2 Objectives 2

1.3 Thesis Organization 2

2 Background and Literature Review 4

2.1 Echo State Network 4

2.1.1 Fundamental Structure of Echo State Network 4

2.1.2 Training 5

2.1.3 State-of-art ESN Implementations on FPGA 7

2.2 Field Programmable Gate Array 8

2.2.1 FPGA vs. ASIC 8

2.2.2 Configurable Logic Blocks 12

2.2.3 Xilinx DSP48 Units 15

3 ESN FPGA Design **20**

3.1 Numerical System 20

3.2 Reservoir Synapses - Linear Combination Without CLBs 21

3.3 Non-Linear Function Units - Hyperbolic Tangent 23

3.3.1 Piece-wise Linear Approximation 24

3.3.2 Hardware Tanh Implementation 32

4 Experimental Results and Analysis **34**

4.1 NARMA10 Datasets 34

4.1.1 Experiment Results 35

4.2 OFDM Symbol Detection 38

4.2.1 OFDM System 39

4.2.2 ESN-based Symbol Detection 41

4.2.3 Experiment Results 42

5 Conclusions and Future Works **47**

5.1 Conclusions 47

5.2 Future Works 47

5.2.1 Power, Performance, & Cost 48

5.2.2 Expansion of Utility 49

Bibliography **51**

List of Figures

2.1 An illustration of a typical ESN structure with M input nodes, N reservoir neurons, and L output nodes 5

2.2 DSP48E1 architecture 16

3.1 Reservoir neuron architecture 22

3.2 Example DSP configurations for each phase 23

3.3 The zoomed-in absolute error before applying the improvement. 28

3.4 The minimum and maximum local absolute error before improvement 29

3.5 The absolute error comparison before and after the improvement 30

3.6 The architecture of the non-linear function (tanh) unit 33

4.1 NARMA10 performances in NMSE (lower the better) 37

4.2 ESN inferences vs. NARMA 38

4.3 OFDM system procedures 39

4.4 The software-defined radio test bed 42

4.5 Performance of hardware/software ESN-based symbol detection, LMS, Comb and LS. 43

4.6 Analysis of BER variance for ESN symbol detection 44

4.7 HW vs. SW ESN on the symbol detection task in time domain 45

4.8 Relative difference between HW and SW ESN outputs 45

4.9 Example constellation diagrams 46

List of Tables

2.1 Comparison between FPGA and ASIC 11

2.2 Available resources of typical Xilinx FPGA devices 18

3.1 Errors on the interval [0,8] and memory usage of the tanh approximator 31

4.1 Major ESN architecture differences between two applications. 35

4.2 NARMA experimental setup and performances 36

4.3 Analysis of BER variance (%) for ESN symbol detection 44

List of Abbreviations

ALM Adaptive Logic Module

APR Automatic Place & Route

ASIC Application Specific Integrated Circuit

BER Bit Error Rate

BPSK Binary Phase-Shift Keying

CLB Configurable Logic Block

Comb Comb Pilot Interpolation

CP Cyclic Prefix

CPU Central Processing Unit

D-FF D-type Flip-Flop

DFR Delayed Feedback Reservoir

DSP Digital Signal Processor

EDA Electronic Design Automation

ESN Echo State Network

FF Flip-Flop

FFT Fast Fourier Transform

FPGA Field-Programmable Gate Array

HDL Hardware Description Language

IFFT Inverse Fast Fourier Transform

IP Intellectual Property

LAB Logic Array Block

LE Logic Element

LMS Least Mean Square

LOS Line-Of-Sight

LS Least Square

LSM Liquid State Machine

LUT Look-Up Table

MSE Mean Square Error

MUX Multiplexer

NMSE Normalized Mean Square Error

NN Neural Network

OFDM Orthogonal Frequency Division Multiplexing

QPSK Quadrature Phase-Shift Keying

RAM Random Access Memory

RC Reservoir Computing

RLS Recursive Least Squares

RNN Recurrent Neural Network

SDR Software-Defined Radio

SRAM Static Random-Access Memory

UAV Unmanned Aerial Vehicle

Chapter 1

Introduction

1.1 Motivation

Neural network (NN) has shown extraordinary effectiveness on a wide variety of applications, such as image recognition [52, 63], speech processing [26, 48], and wireless communication [73]. The success greatly propels the evolution of hierarchical neural network architecture. Recurrent neural network (RNN), as a sub-category of NN, evolves a loop back structure, creating temporal memory for the network. This advancement profoundly expands the capability of NN in processing time-dependent data. However, RNNs are notoriously difficult to train, in terms of computational complexity.

Reservoir computing (RC), a recently developed machine-learning paradigm whose processing capabilities rely on the dynamical behavior of RNN, greatly simplifies the training procedure in that only the weights in output neurons have to be trained. RC has proved its power efficiency in various applications [54], including pattern classification [45, 47, 57], time series forecasting [31], pattern generation [32], channel equalization [33], etc. Currently, there are three kinds of RC systems, specifically, liquid state machine (LSM) [34], delayed feedback reservoir (DFR) [7], and echo state network (ESN) [30].

There is an urgent need for RC hardware design to provide"fast, less expensive and more energy-efficient" computing platforms [30–32]. In spite of the improved training strategy, the inference

operation of ESN is still costly due to the nature of vector-matrix operations and non-linear functions. Calculation using conventional von Neumann architecture is time and energy consuming, making it unattractive for power/cost-sensitive platforms, such as mobile device and unmanned aerial vehicle (UAV) [31]. Field-programmable gate array (FPGA) provides a good opportunity for the digital hardware designers to take advantage of the parallel essence of ESN. Several FPGA approaches have been made, but none of them have explored the full capacity of DSP blocks to target low-cost/low-power FPGA devices [41–43, 62].

1.2 Objectives

In this thesis, we introduce a cost-optimized, scalable ESN architecture on FPGA which exploits Xilinx DSP unit, DSP48E1. To be more specific, the proposed work includes a linear combination processor with negligible deployment of configurable logic blocks (CLBs), and a high-accuracy non-linear function approximator, both with the help of only 9 DSP units in each neuron. Furthermore, we cut down the need of CLBs in the ESN FPGA implementation, which makes our design ideal for the low-cost Xilinx Artix-7 devices by offering a higher DSP-CLB ratio. Our design is verified by the classical NARMA10 dataset and a symbol detection task on a physical Wi-Fi communication system. The test bed is built on a software-defined radio (SDR) platform, showing that our work is capable of processing the real-world data.

1.3 Thesis Organization

The organization of this thesis are summarized as the following: Chapter 2 introduces the background of ESN, the critical components in an FPGA, a comparison between FPGA and ASIC, a functional review of Xilinx DSP48E1, as well as a literal survey of state-of-the-art ESN implementations

on FPGA; Chapter 3 focuses on our proposed hardware architecture, including a cost-efficient synapse design, and a high-accuracy hardware hyperbolic tangent function with no additional DSP cells required; Chapter 4 demonstrates how we apply the proposed ESN hardware to 1) the classical NARMA dataset and 2) a real-world symbol detection task on GNU software-defined radio platform [40] along with its performances; Chapter 5 summarizes our work and discusses potential future improvements.

Chapter 2

Background and Literature Review

2.1 Echo State Network

2.1.1 Fundamental Structure of Echo State Network

The basic structure of ESN is illustrated in Figure 2.1. The computational model consists of three layers: an input layer takes in data from the user, a reservoir layer evaluates the current state using both a memorized state and current inputs, and an output layer predicts the outcome based on the given state.

Consider a typical ESN structure with M input nodes, N reservoir neurons, and L output nodes, we denote inputs of time step *n* as $\mathbf{u}(n)=(u_1(n), u_2(n), \cdots, n_M(n))$. Reservoir states are written as $\mathbf{x}(n)=(x_1(n), x_2(n), \cdots, x_N(n))$; $\mathbf{y}(n)=(y_1(n), y_2(n), \cdots, y_T(n))$ represents the output predictions.

The calculation of the current state $\mathbf{x}(n)$ and the prediction $\mathbf{y}(n)$ are represented by the following equations:

$$\mathbf{x}(n) = \mathbf{f}\left(\mathbf{W}^{in}\mathbf{u}(n) + \mathbf{W}\mathbf{x}(n-1) + \mathbf{W}^{fb}\mathbf{y}(n-1)\right) \tag{2.1}$$

$$\mathbf{y}(n) = \mathbf{f}^{out}\left(\mathbf{W}^{out}\mathbf{z}(n)\right), \tag{2.2}$$

where extended state $\mathbf{z}(n)=\{\mathbf{x}(n);\mathbf{u}(n);\mathbf{y}(n-1)\}$; $\mathbf{f}=(f_1,f_2,\cdots,f_N)$ are the activation functions used in the reservoir neurons (usually hyperbolic tangent or sigmoid functions), and $\mathbf{f}^{out}=(f_1^{out},f_2^{out},\cdots,f_N^{out})$ are the activation functions used in the output layer (usually linear functions).

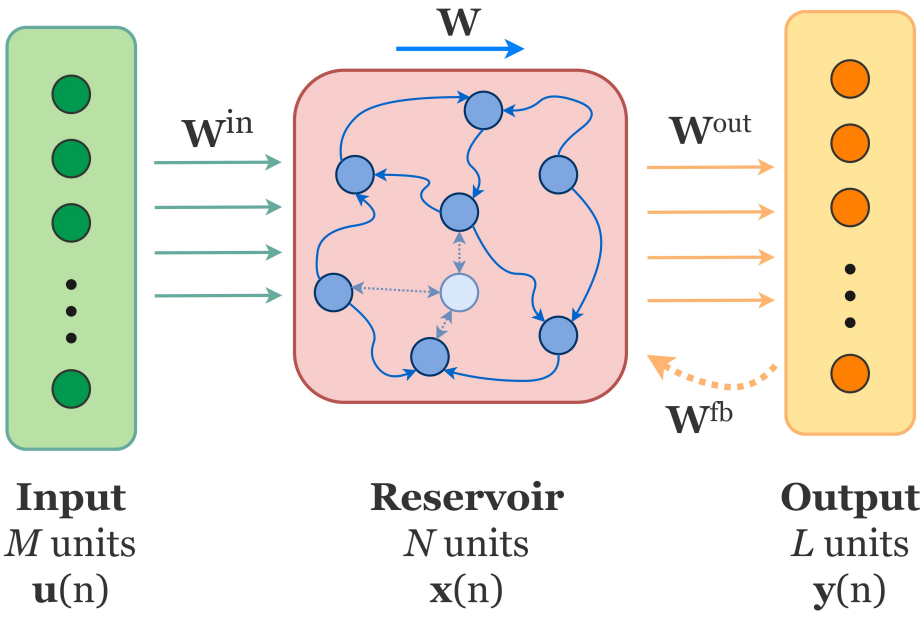

Figure 2.1: An illustration of a typical ESN structure with M input nodes, N reservoir neurons, and L output nodes. $\mathrm{W^{in}}$, with shape $\mathrm{N\times M}$, maps $\mathrm{u(n)}$ into the reservoir neurons; $\mathrm{W^{fb}}$ has the shape $\mathrm{N\times L}$ and brings in the last output $\mathrm{y(n-1)}$; W is the reservoir weight with the form $\mathrm{N\times N}$, which is the critical part to create "echo" (short term memory). $\mathrm{W^{out}}$ has the form $\mathrm{L\times(N+M+L)}$ and is the only weight set to be trained in ESN.

2.1.2 Training

The major reason that ESN is so popular is due to the simplicity of training compared with traditional RNNs. Only output weights $\mathbf{W}^{out}$ need training in order to match the application. ESN is commonly trained by supervised learning methods, given a set of training input $\mathbf{U}^{training}\in\mathbb{R}^{M\times T}$ and desired output $\mathbf{Y}^{label}\in\mathbb{R}^{L\times T}$ with a training length T time steps. A good approach tries to find $\mathbf{W}^{out}$ such that the mean square error (MSE) between $\mathbf{Y}^{trained}$ and $\mathbf{Y}^{label}$ is minimized,

where $\mathbf{Y}^{trained} \in \mathbb{R}^{L \times T}$ is the inferential output of ESN. The MSE can be defined as equation (2.3):

$$MSE = \frac{1}{L}\left(\mathbf{Y}^{label} - \mathbf{Y}^{trained}\right)^{T}\left(\mathbf{Y}^{label} - \mathbf{Y}^{trained}\right) \tag{2.3}$$

Regard an ESN with linear functions applied as output activation functions and with no $\mathbf{W}^{fb}$. $\mathbf{Y}^{trained}$ can be depicted by equation (2.4):

$$\mathbf{Y}^{trained} = \mathbf{W}^{out}\mathbf{Z}, \tag{2.4}$$

where $\mathbf{Z} \in \mathbb{R}^{(N+M) \times T}$ is the row-wise concatenation of $\{\mathbf{X}; \mathbf{U}^{training}\}$, and $\mathbf{X}$ is derived by equation (2.1), running through the whole training time steps T.

The training then becomes a simple linear regression problem. The well-known normal equation method shown as equation (2.5) is often used to solve such a problem:

$$\mathbf{W}^{out} = \mathbf{Y}^{label}\mathbf{Z}^{T}\left(\mathbf{Z}\mathbf{Z}^{T}\right)^{-1} \tag{2.5}$$

To mitigate the impact of overfitting and penalize extreme large $\mathbf{W}^{out}$, Tikhonov regularization [44] is commonly added to (2.5). The training of ESN then can be rewritten as:

$$\mathbf{W}^{out} = \mathbf{Y}^{label}\mathbf{Z}^{T}\left(\mathbf{Z}\mathbf{Z}^{T} + \beta\mathbf{I}\right)^{-1} \tag{2.6}$$

where β is the regularization factor and $\mathbf{I}$ is an identity matrix.

2.1.3 State-of-art ESN Implementations on FPGA

FPGA is an ideal platform to build acceleration hardware of artificial neural networks (ANNs), since developers can largely exploit its parallel nature. For the same reason, application-specific integrated circuits (ASICs) are options to researchers too. Previously, cutting-edge ASIC implementation using delay-based RC has been realized. Bai at el. design and fabricate a pure analog delay-based RC integrated circuit on the CMOS 130-nm technology node, verified by NARMA dataset [13]. A series of related research work is conducted in the following publication: [12, 14–17, 39]. However, ASIC design of RC consumes longer development time and cost much more on experienced personnel and testing equipment, compared with the FPGA design.

FPGAs, on the other hand, provides a great balance between power, performance, area, and flexibility. Several FPGA implementations of ESN have been investigated. Yi et al. introduce the use of FPGA to reservoir computing with on-board training capability [62]. Liao et al. realize pattern recognition and sinusoidal wave generation with on-board batch training on a 65-nm Intel Stratix III high-performance FPGA [41–43]. Antonik et al. show the potential of applying recursive least squares (RLS) online training [23] on FPGA when solving the channel equalization problem [6]. Another group of research on ESN hardware tries to improve the efficiency using stochastic bitstream neurons [11]. Signals in stochastic neurons are represented by a series of bitstream, where the information is encoded as the probability of "1"s. With stochastic neurons, complex arithmetic such as multiplication and non-linear functions can be implemented with reduced hardware. The idea is first introduced to reservoir computing in [57], and a proof-of-concept implementation is presented in [4, 5]. Furthermore, Huang et al. propose a scalable ESN hardware generator [27]. The use of high-level synthesis reasonably reduces the development time. However, none of the above fully utilizes the advantage of DSP blocks on contemporary FPGAs. To our best knowledge, the work presented in this thesis is for the first time exploiting the full capacity of

Xilinx DSP48 units to minimize the cost of the physical RC.

2.2 Field Programmable Gate Array

Field Programmable Gate Arrays (FPGAs) are semiconductor devices whose function can be programmed to mimic a wide variety of digital circuit designs. A common FPGA device contains configurable logic blocks, I/Os, and fabric routing resources. More advanced devices often comes with dedicated arithmetic units and built-in soft-cores/intellectual properties (IPs). A FPGA designer usually writes hardware description languages (HDLs), e.g., Verilog or VHDL, to "sketch" a desired digital circuit system. Contemporary FPGA development environments, e.g., Intel Quartus Prime [20] and Xilinx Vivado [61], read the HDL code and synthesize them into cells and nets delivered by the target device. Such technique is referred to as "inference". Next, the tool automatically decide the physical location of each cell and route them together using available wiring resources. This step is also known as automatic place & route (APR). Finally, the tool generates a bit-stream file that can be sent to the FPGA through a device programmer. Then the designated function can be run on the programmed FPGA chip. In the following sections, we compare the major differences between FPGA and ASIC, and two key components of FPGAs that regularly impact a designer's decision.

2.2.1 FPGA vs. ASIC

ASIC, or application specific integrated circuit, is another popular option of realizing high-speed operations. It's extremely high efficiency has attracted the industry for decades. Similar to FPGA, digital circuit designers can freely implement arithmetic units, customized logic, memory, and even analog modules in an ASIC. The comparison between FPGA and ASIC can start from four

major aspects: reconfigurability, cost, time to market, and efficiency. These four considerations affects our decision on which platform is more suitable for building a dedicated ESN hardware. A comparison is summerised in Table 2.1.

Reconfigurability

The production procedure of ASICs requires masking, which contains the design pattern of the circuit of each production step. Once the mask is created and the pattern is etched onto the wafer, it is almost impossible to change it. Redesign of the mask is not only time consuming, but costly. On the other hand, FPGA offers phenomenally great flexibility. The circuit functionality can be entirely changed even after a previous version has been burned on the chip. The upgrade can also be done in a reasonable amount time, with nearly no cost.

Cost

Two major costs are involved on both side: development cost and production cost. For development, FPGA costs apparently lower than ASIC. The former one usually only require one evaluation board, a set of EDA tools provided by chip vendors, and a small team of personnel; while the latter one may demand dedicated testing equipment, a more complex EDA tool set, and a larger team with longer development cycle.

The production cost of FPGA and ASIC is consist of very distinct elements. The manufacturing steps of ASIC intrinsically bring in higher non-recurring engineering (NRE) costs, including wafers, masks, and equipment. This could be up in tens of thousand to millions of USD, whereas with high enough volume, its cost of each die could be in cents. In contrast, FPGAs are sold with zero or negligible NRE, where the per die price is higher. Generally speaking, the production cost makes

ASIC shines in high volume, while FPGA plays a better role in low to medium volume.

Time to Market

Development time is a significant factor of the adoption of each technology. With both approaches, engineers write hardware description language, such as Verilog HDL [28] or VHDL[2] for the front-end design. Modern FPGA EDA tool chains usually can handle the rest of the procedures, including synthesis, automatic place and route (APR), generation of bit-stream, and chip programming. ASIC EDA tools are capable of synthesis and APR as well, however, back-end ASIC design involves more electrical challenges. The unskippable additional probing, testing, and packaging stages take substantial amount of time too. In fact, FPGA are well-known for its better time-to-market that it is often used for fast prototyping of digital circuits.

Efficiency

The better operational speed and energy efficiency are often one of the crucial criteria why ASIC is chosen in the industry. The top speed of Xilinx 7-series FPGA runs at around 50MHz to few hundreds of MHz, while an ARM A9 CPU made in ASIC can easily work at 800MHz to few GHz. This is due to the programming mechanism of FPGAs, whose routing is achieved by programmable switches and CLBs are mostly made by static random-access memory (SRAM). For the same reason, excessive devices in FPGA and longer routing distances take away more enerygy, making it not as efficient as ASIC approaches.

Best Fit of ESN Hardware

When the target application requires extremely high speed, or when the power consumption has to be reduced at all cost, ASIC performs better because of its magnificent efficiency. Also,

Table 2.1: Comparison between FPGA and ASIC

Criteria	FPGA	ASIC
Reconfigurability	yes	no, or costly
Language for development	Verilog HDL or VHDL	
Front-end design flow	synthesis, simulation, place, & route	
Additional back-end steps	programming & testing	manufacturing, packaging, & testing
Time to market	short to medium	long
Product cycle	less restriction	long
Development cost	low to medium	high
Best production volume	low to medium	high volume
Clock speed	medium	high
Energy Efficiency	medium	high
Common applications	fast prototyping designs require constant updates	high-efficiency circuit high-volume product

when a high volume of production is expected, ASIC might be the superior option in terms of cost. However, ESN, as well as most machine learning methods, evolve rapidly in this era. We are under a time when algorithms advance faster than hardware. The reprogrammability nature of FPGA grant the opportunity to bridge the gap. Furthermore, ESN itself is sensitive to multiple hyperparameters, such as the number of reservoir neurons and whether applying output feedback to the reservoir layer. Users may find one architecture suits the application better after a more thorough training or a change of the input scenario. Building the circuit on an FPGA is the only way for upgrading, while there is nearly no chance or too costly on ASICs. Last but not least, the lower cost for development and the more compact time to market really makes FPGA stand out against the ASIC path. We argue that FPGA works best in most cases for specialized ESN hardware. In the coming sections, we will discuss the critical components in Xilinx FPGAs which we exploit to improve its efficiency.

2.2.2 Configurable Logic Blocks

Configurable logic blocks (CLBs) are the elemental units implementing combinational and sequential logics in Xilinx FPGAs. As suggested by its name, FPGA engineers can configure the function of CLBs, in order to realize custom logics. In FPGA terminologies, it is also referred to as logic array block (LAB) or logic element (LE). On Intel FPGA devices, they have a similar unit named as adaptive logic module (ALM), coming with similar purposes but very different architectures. [21]

Xilinx CLBs provide extraordinary flexibility and high performance for designers. Each CLB consists of four slices, and each slice contains:

- Four logic-function generators (6-input look-up tables)
- Eight storage elements (flip-flops or latches)

- Wide multiplexers
- High-speed carry logic for arithmetic functions

Look-Up Tables

The look-up table (LUT) in a CLB is the key component making it "configurable". It can work as any arbitrarily defined 6-input boolean function, or two arbitrarily defined 5-input boolean function.

Storage Elements

In the eight storage elements in each slice, four of them can be configured as either D-type flip-flops (D-FFs) or latches. When working as D-FFs, they can be accessed by the LUTs directly, performing registered functional output. The D-FF is driven by clock, clock enable, and optionally set/reset input signals, creating versatile register options.

Multiplexers

With the help of function generators and built-in multiplexers (MUXs), a 7-series CLB can perform the following functions:

- 4:1 multiplexers using one LUT (4 MUXs per slice)
- 8:1 multiplexers using two LUTs (2 MUXs per slice)
- 16:1 multiplexers using four LUTs (1 MUX per slice)

Carry Logic

The lookahead carry logic helps perform fast addition and subtraction in a slice. It is specifically reserved for small (less than 16-bit) adder designs. Each slice contains a CIN pin, a COUT pin, as well as a 4-bit carry logic. The CIN and COUT pin of adjacent slices are wired physically for a shorter carry delay.

Distributed RAM

One interesting application of CLBs is the distributed RAM[1]. The CLBs can be configured as a various type of memory cells. The following is the list of configurations [58]:

- 32 × 1-bit Single-Port RAM
- 32 × 1-bit Dual-Port RAM
- 32 × 2-bit Quad-Port RAM
- 32 × 6-bit Simple Dual-Port RAM
- 64 × 1-bit Single-Port RAM
- 64 × 1-bit Dual-Port RAM
- 64 × 1-bit Quad-Port RAM
- 64 × 3-bit Simple Dual-Port RAM
- 128 × 1-bit Single-Port RAM
- 128 × 1-bit Dual-Port RAM
- 256 × 1-bit Single-Port RAM

The write operation of distributed RAM is synchronous, while the read operation can be either synchronous with flip-flops or asynchronous without one. The freedom of configurations largely increases the on-chip memory capacity besides dedicated block memories.

[1] CLBs in Xilinx 7-series consist of two types of slice, SLICEM and SLICEL. The distributed RAM function is only available in SLICEMs. A typical SLICEL:SLICEM ratio is around 2:1. For detailed numbers of each slice, please consult [58].

The Cost of CLB

With aforementioned basic components and functions, combined with fabric wiring resources, designers can almost freely implement any logics on FPGA, including our goal, an ESN accelerator. However, the utilization of CLBs is highly related to the "cost" in FPGA design. Using less CLBs indicates a broader choice of low-cost devices, introducing higher possibility of reducing cost in large-volume production. This thesis provides a solution deploying the DSP units in Xilinx FPGAs. The DSP unit is fabricated with dedicated circuit, introducing high-speed and low-power arithmetic capability. Unlike CLBs, the DSP unit is not fully configurable. Some of its ports are hidden from custom wiring. We must understand the in-depth architecture of it to unlock its full power, which will be discussed in the next section.

2.2.3 Xilinx DSP48 Units

The basic computation of equation (2.1) and (2.2) is matrix multiplication and accumulation, which can be efficiently realized with the Xilinx DSP48 units. DSP48 is a dedicated circuit designed for arithmetic data path in Xilinx FPGAs, which is widely supported in Xilinx's product family. It provides higher performance and lower power consumption for multiplication and accumulation, compared to using CLBs.

In this section, as well as in our proposed ESN architectural design, DSP48E1 [59] is mainly discussed and optimized for, specifically, for the following reasons:

- DSP48E1 is widely supported in Xilinx's 7-series product family since announced in 2010.
- The multiplier width is improved from 18×18 in the previous version, DSP48A1, to 25×18. This is critical since in certain circumstances, multiplicands with up to 20 bits are required in calculating equation (2.1).

- The latest version DSP48E2 [60] is functionally equivalent and backwards compatible with DSP48E1. The major difference is the further extended bit length of inputs.

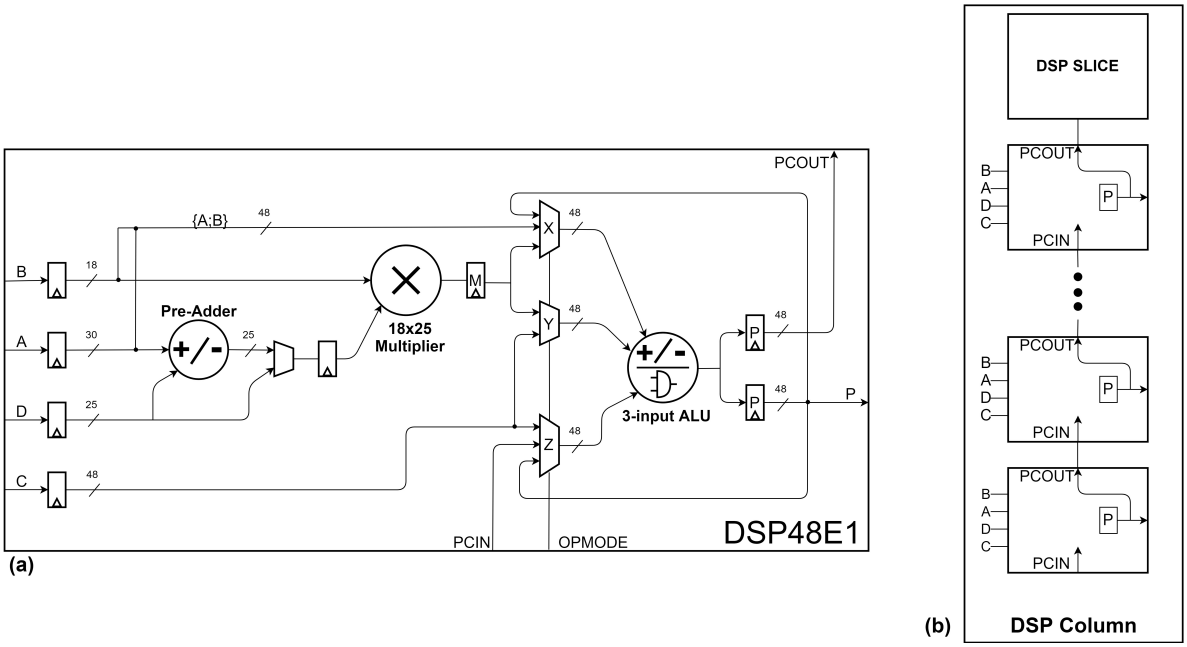

Figure 2.2: DSP48E1 architecture. (a) The simplified architecture of DSP48E1. Our design primarily makes use of the multiplier, the 3-input ALU, and PCIN/PCOUT inter-DSP connection. Unrelated paths and units are hidden in this graph. (b) DSP slices are cascaded and linked with PCIN/PCOUT ports.

Architecture

Figure 2.2 illustrates the simplified architecture of DSP48E1. It consists of three major arithmetic units - one 2-input multiplier, one 2-input power-saving pre-adder before the multiplier, and one 3-input ALU at the end of the cell[2].

Four inputs are brought into one cell at the same time. Each input drives a series of configurable registers before fed into the next stage. A proper setting of the registers can be handful in pipelining

[2]Additionally, DSP48E1 has a built-in pattern recognizer, which extends its application by including more "flags" available to designers. However, it it not essential in building an ESN accelerator. Please check the DSP48E1 user guide [59] for the very detailed information.

and increasing clock speed. The pre-adder is prestigiously useful for filter designs. Depending on the structure, Ahmed Akif demonstrates that building finite impulse response (FIR) filters with the help of pre-adders can reduce either latency or the demand of CLBs [3].

The DSP48E1 multiplier accepts 18-bit $\times$ 25-bit inputs with two's complement number capability. The multiplication result is then optionally stored in register M. Before the number goes into the next ALU, a stage of MUXs is deployed to control the three inputs of the arithmetic unit. In the end, the final number is stored in register P as the output, which can finally be accessed by fabric routing resources.

Operations

The three MUXs (X, Y, and Z), along with the input register setting, define the functionality of the DSP cell. By careful planning, users can almost freely determine the arithmetic equation of P[3]. The general equation of output P can be expressed as:

$$P = C \pm (B \times (A \pm D)) \tag{2.7}$$

One advanced feature that can be exploited is the PCIN/PCOUT inter-DSP connection. These two ports cannot be accessed by custom logics, but they are wired with high-speed connections between cells, as shown in Figure 2.2 (b). Also, DSP48E1 provides a 48-bit [A:B] concatenation bus as an option of inputs of the final stage ALU. By carefully selecting OPMODE, one can configure the operation of DSP48E1 into 3-input adders as equation (2.8a) & (2.8b):

[3]When multiplication is activated, product M occupies both MUXs X and Y. This fact is due to the multiplier design, which register M actually holds two partial products of the multiplier, taking two of the three inputs of the ALU.

$$P = PCIN + C + P \tag{2.8a}$$

$$P = [A: B] + C + P \tag{2.8b}$$

This feature is greatly useful when building the compressor in synapses. In section 3.2, we will further describe how we utilize the advanced features to improve cost-efficiency.

Table 2.2: Available resources of typical Xilinx FPGA devices

		CLB				
Family	Device	Slices	Max Distributed RAM (Kb)	DSP Slices	CLB-DSP Ratio	Block RAM (Kb)
Spartan-7	XC7S100	16,000	1,100	160	100	4,320
Artix-7	XC7A200T	33,650	2,888	740	45	13,140
Kintex-7	XC7K325T	50,950	4,000	840	60	16,020
Virtex-7	XC7VX485T	75,900	8,175	2,800	27	37,080
Kintex UltraScale	KU040	60,600	7,100	1,920	31	21,000
Virtex UltraScale	VU095	134,400	4,800	768	175	60,800

*We list the devices whose native Xilinx evaluation boards are available on the market.

The Adoption of DSP48E1

Naturally, designers tend to avoid placing multipliers to minimize cost in digital circuit design. However, it is not true when working on a FPGA, where DSP blocks are pre-planned and already included inside the chip. In contrast, utilizing DSP blocks largely reduce the need of CLBs, making

the design compatible with low-cost FPGAs. Xilinx Artix-7 family devices are good examples falling into this category, whose DSP-to-CLB ratio is greater than their 7-series brothers. Table 2.2 lists the resources available of typical Xilinx FPGAs.

Chapter 3

ESN FPGA Design

3.1 Numerical System

In hardware arithmetic, dealing with fractional numbers can be challenging. There are two major ways to represent fractional numbers, floating-point and fixed-point expression. The first one usually follows the IEEE 754 standard [1] with either single-precision (32 bits) or double-precision (64 bits) format. They are widely used in general purpose computers due to a wide representative value range and a better precision. However, the down-side of floating-point expression is also obvious. It takes more storage space for each number and it is more complex to build the arithmetic unit with floating-points. This approach is often not desirable for application-dedicated hardware, such as our work.

In our implementation, unless otherwise stated, all fractional numbers are stored in signed fixed-point interpretation. We denote the numbers using $< l, f >$ expression, where l represents the total bit length and f represents the number of fractional bits reserved. Negative numbers are depicted by the 2's complements of their absolute value. The maximum precision loss is 2^{-f}, and the value range can be declared as following:

$$\left[-2^{l-f-1},\ +2^{l-f-1}-2^{-f}\right]$$

A conversion between the nominal value v_{nom} and representative value v_{rep} can be written in equation (3.1)

$$v_{rep} = \begin{cases} v_{nom} \times 2^{-f}, & \text{when } v_{MSB} = 0 \\ -\overline{v_{nom}} \times 2^{-f}, & \text{when } v_{MSB} = 1 \end{cases}, \tag{3.1}$$

where $\overline{v_{nom}}$ is the 2's complement of v_{nom}, and v_{MSB} is the most significant bit of the stored number.

3.2 Reservoir Synapses - Linear Combination Without CLBs

Synapses in reservoir computing take the analogy from neural science, where a synapse connects between neuron cells and takes in information. In ESN, reservoir synapses receive data from both input and reservoir layer, applying pre-defined weights to the data accepted. Data here indicate the extended states $\mathbf{z}(n)$ and the process is mathematically modeled by equation (2.8a) as the linear combination before the sum-of-product is fed into non-linear functions $\mathbf{f}$.

The two fundamental operations in a linear combination are multiplication and accumulation. Consider an ESN without feedback connections and $\mathbf{W}^{fb}$. Each neuron involves (M+N) multiplications and (M+N-1) additions in each time step. A naïve implementation may try to instantiate (M+N) DSP cells as the multiply-accumulator followed by $2^{\lceil \log_2 (M+N) \rceil} - 1$ fabricated adders as the compressor tree, where $\lceil \cdot \rceil$ represents the ceiling function. Nonetheless, such design is unscalable and not suitable for a larger neural network. Moreover, building the compressor using CLBs consumes more power and execution time than adopting dedicated DSP blocks [59].

In our work, a reservoir synapse using DSP blocks almost eliminates the need of CLBs. In fact,

we configure individual DSP unit such that it explores the full function introduced in equation (2.7) & (2.8). The same DSP block is reused in the compressing stage and as well as the non-linear function calculation. Figure 3.1 demonstrates the architecture of the entire reservoir neuron.

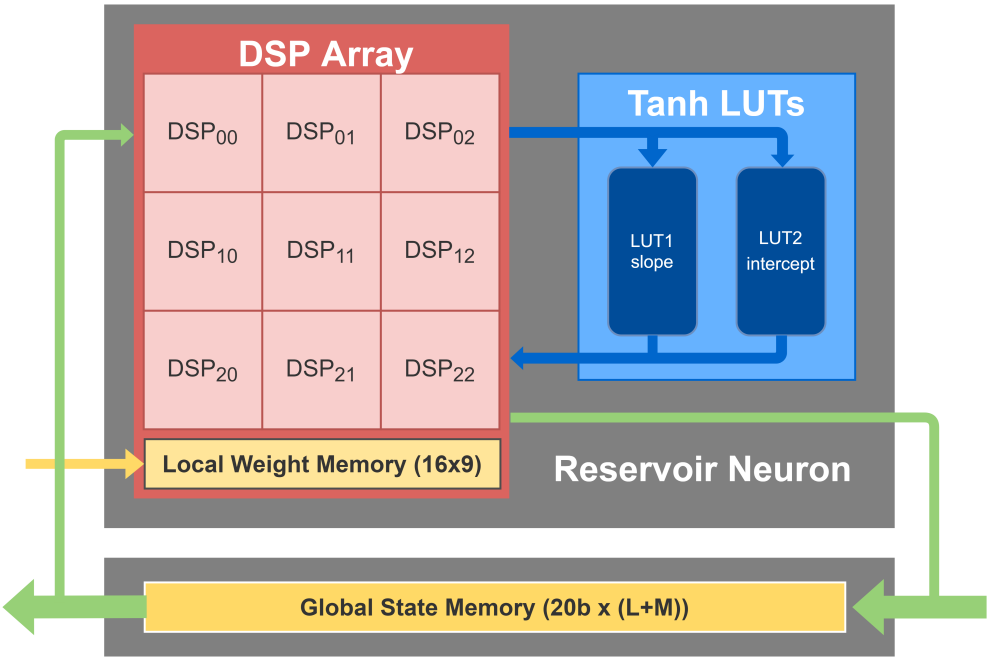

Figure 3.1: Reservoir neuron architecture. The whole neuron consists of 9 DSP48E1 slices, a local weight memory (16 bits $\times$ 9 words), and two look-up tables for the approximative coefficients of tanh.

Each neuron has only 9 fixed DSP blocks. A local weight memory is introduced to store the i_{th} reservoir weights and input weights. The extended state $\mathbf{z}(n)$ is brought in by a bus connected to the global state memory. As shown in Figure 3.2, we then divide the computation of equation (2.1) into three phases:

- **Multiplication and Accumulation (MACC):** 9 groups of weight and state are multiplied and accumulated every clock cycle. Partial sum-of-products (SoPs) are saved inside P registers.
- **Compression I:** With 3 DSP blocks as a group, the 9 SoPs are compressed into three. For

example: $\mathbf{P}_{10} = \mathbf{P}_{00} + \mathbf{P}_{10} + \mathbf{P}_{20}$.

- **Compression II:** Finally, the 3 SoPs are compressed into one. We can express this step as $\mathbf{P}_{12} = \mathbf{P}_{10} + \mathbf{P}_{11} + \mathbf{P}_{12}$. The final result $x_i(n)$ is store in the register $\mathbf{P}_{12}$.

The example configurations of each DSP are given in Figure 3.2:

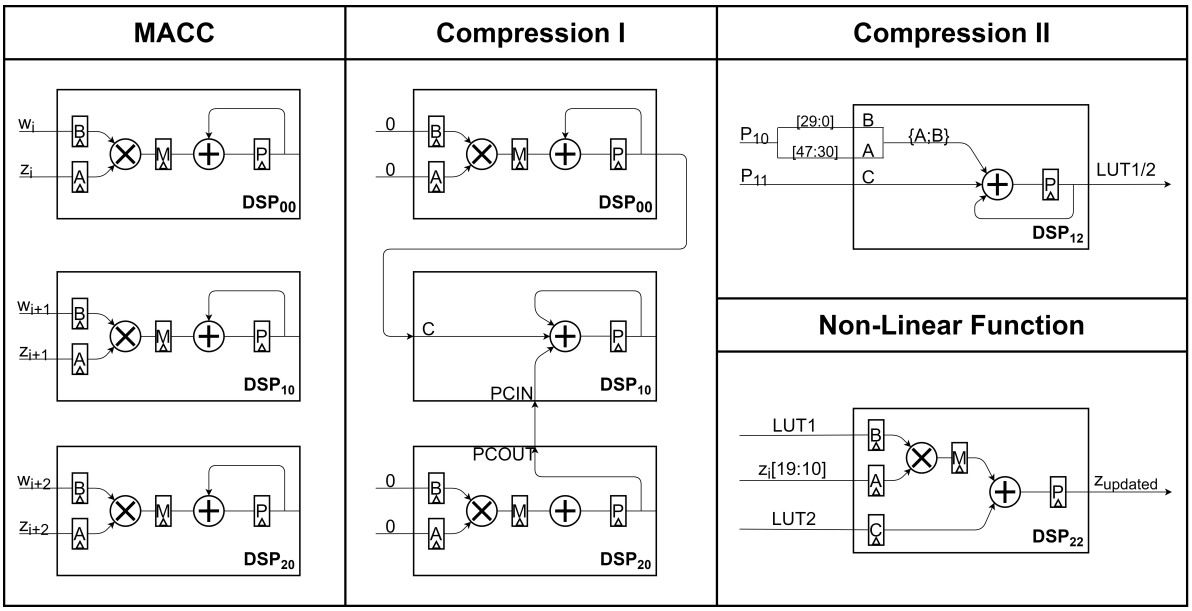

Figure 3.2: Example DSP configurations for each phase. In MACC stage, all DSPs have the same setup. In Compression I, 3 DSPs are combined as a group. The inputs of upper and lower DSPs are tied to 0 to prevent washing out of P registers. The three partial SoPs are cached in P_{10}, P_{11}, & P_{12}. In Compression II, DSP_{12} is assigned to compute the final linear combination result. Last but not least, the result is sent to non-linear function LUTs, and DSP_{22} is arranged to close up the calculation.

3.3 Non-Linear Function Units - Hyperbolic Tangent

In software, users can precisely approach hyperbolic tangent (tanh) using Cody-Waite polynomial [19], with high precision floating-point numbers, whereas, our work use fixed-point number system and the polynomial approach is surely too expensive.

The required accuracy for tanh hardware approximation may differ from one application to another. The difference majorly stems from the dynamic range of output weights, due to the nature of the applications. Take our two experiments for example, in 10 individual test runs, the teacher labels from NARMA10 dataset are no smaller than 0.2 and no larger than 0.8, yielding the trained output weights in the range of (-200, 200). In contrary, in the symbol detection task, some output weights can go up to $\pm$100,000. The large weight numbers can magnify any small accuracy/precision loss in the reservoir states $\mathbf{x}$ (the output of tanh), and thus mess up the final prediction. The bit-length of $\mathbf{x}$, as well as the tanh approximator has to be carefully designed.

3.3.1 Piece-wise Linear Approximation

To compensate the large weights while reducing resources, we apply a 1_{st}-order piece-wise linear approximation to tanh. Let $f(s) = \tanh(s), |s| < 16$, where s is the 36-bit compressed sum-of-product from the linear combination stage; $|\cdot|$ denotes the absolute value. The method tries to find the approximated function $\hat{f}(s)$, such that:

$$\begin{aligned} x = \hat{f}(s) &= \textit{slope} \times \Delta s + \textit{intercept} \\ \Delta s &= s - round(s), \end{aligned} \tag{3.2}$$

where $round(s)$ is the rounded s to a certain precision; $slope$ and $intercept$ are two sets of constants such that $\Delta f(s) = \hat{f}(s) - f(s)$ is minimized.

This work employs two look-up tables (LUTs) to store $slope$ and $intercept$, respectively.

Number Formats

Dynamic range, precision, and accuracy are three key factors for constructing the approximator. Thanks to the characteristics of tanh, the neuron output $\mathbf{x}$ is always within the range of [-1, 1], meaning that we can grant all the bits to the fractional part except for the first sign bit. Precision loss is introduced by the fixed-point number system. We can calculate the appropriate fractional bit length, as long as the precision loss is less than the application requirement. For example, given the output weights from the symbol detection task (Table 4.1), 2×10^{-6} is a safe precision setting. A reasonable number of fractional bits can be calculated by:

$$\text{\#fractional bits} = \left\lceil -\log_2 \left(2 \times 10^{-6}\right) \right\rceil = 19, \tag{3.3}$$

where $\lceil \cdot \rceil$ denotes the ceiling function. Therefore, $\mathbf{x}$ is safely applied with a $< 20, 19 >$ format.

Accuracy, on the other hand, is a relative complex problem, since it is impacted by how the *slope* and *intercept* tables are generated, as well as the number format of each table entry. Further experiments and analysis are needed, which we will discuss more in the following subsections.

The Generation of *slope* and *intercept*

Before working on the table, we first need to define the interval $I = [G, H]$, where $G < H$ and all input of the approximator lies within. Due to the symmetry nature of tanh function, the two tables only need to serve the positive half of input s. Moreover, when $s > 8$, $tanh(s)$ is very close to 1 (error < 10^{-6}). Combining the above two reasons, we set $I = [G, H] = [0, 8]$.

Next, let *a* denotes the address bit-length of the tables. This divides the interval I into 2^a sections

by entry points. The i_{th} entry point is denoted as $s_i = 0 + (8 - 0) \times i \times 2^{-a}$. The *intercept* is then derived by:

$$intercept_i = \tanh(s_i);$$

each entry of *slope* is further extended as:

$$slope_i = \frac{(intercept_{i+1} - intercept_i)}{8 \times 2^{-a}}.$$

Analysis of The First Approach

An analysis is performed to verify if the first approach is optimal. Four metrics are useful to evaluate the effectiveness of the approximation. We follow the definitions in [18] & [56]. The metrics are defined as follows.

For function $f : I = [G, H] \rightarrow \mathbf{R}$, the approximation $\hat{f} : I \rightarrow \mathbf{R}$ has the absolute error of $f(s) - \hat{f}(s)$, where there are K uniformly sampled s in interval I. (Please note that the "absolute" here intend to distinguish itself from the "relative" errors. It has nothing to do with the mathematical absolute value.) That is,

$$AverageAbs.Error = \frac{\sum_{i=0}^{K-1} |f(s_i) - \hat{f}(s_i)|}{K}, \tag{3.4}$$

where $s_i = G + i \times \delta$, and $\delta = (H - G)/K$. Analogously, we define the maximum absolute error as:

$$MaximumAbs.Error = \max_{s \in I} |f(s_i) - \hat{f}(s_i)|. \tag{3.5}$$

Next, we have the average relative error defined as:

$$AverageRel.Error = \frac{\sum_{i=0}^{K-1} \left| \frac{f(s_i)-\hat{f}(s_i)}{f(s_i)} \right|}{K}, \tag{3.6}$$

and the maximum relative error is defined as:

$$MaximumRel.Error = \max_{s \in I} \left| \frac{f(s_i) - \hat{f}(s_i)}{f(s_i)} \right|. \tag{3.7}$$

For hyperbolic tangent function, relative errors always approach infinity when s is close to 0, since $\tanh 0$ is 0. Thus, the following paragraph will mainly discuss absolute errors.

Figure 3.3 shows that the error fluctuates between intervals. Zooming in on a single sub-interval, we find the error goes from nearly 0 to a local maximum, and goes down again, forming a convex and parabola-like trend in the graph. It makes sense that the error are closer to 0 on both ends, since they are calculated by the tanh function exactly. The error only arises from the precision loss of fixed-point numbers. But the convex shape, along with the all positive error, suggest that this method can be improved.

Improved *intercept* Table

In Figure 3.4, the whole interval [0,8] is divided into 2^{10} sub-intervals. We extract the maximum and minimum absolute error in each sub-interval and plot them in the graph. The figure further confirms our hypothesis that all errors are positive. That is, it is possible to decrease the absolute error $f(s) - \hat{f}(s)$, by lifting the approximation $\hat{f}(s)$ slightly. The idea is to offset downward the positive, parabolic error curve, such that the local minimum decreases from nearly 0 to a negative number, and the local maximum is reduced to |local minimum|. This modification potentially

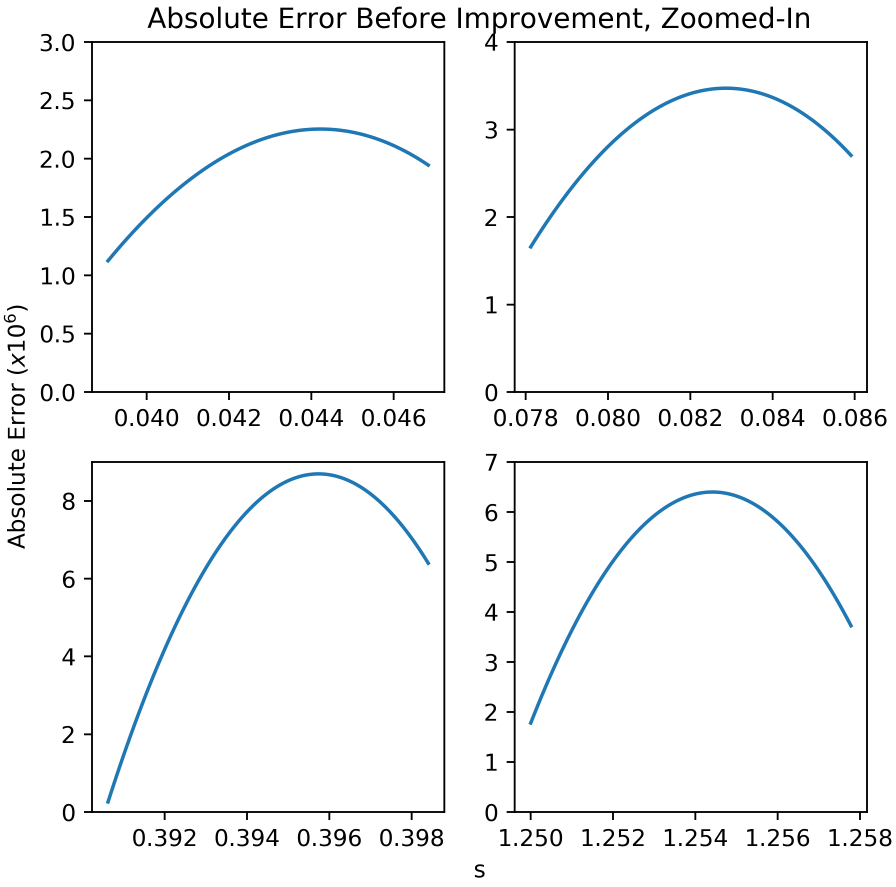

Figure 3.3: The zoomed-in absolute error before applying the improvement. In this graph, each table has 2^{10} entries; *slope* has the format $< 10, 10 >$; *intercept* is stored with $< 19, 19 >$ numbers; δs consumes 8 bits.

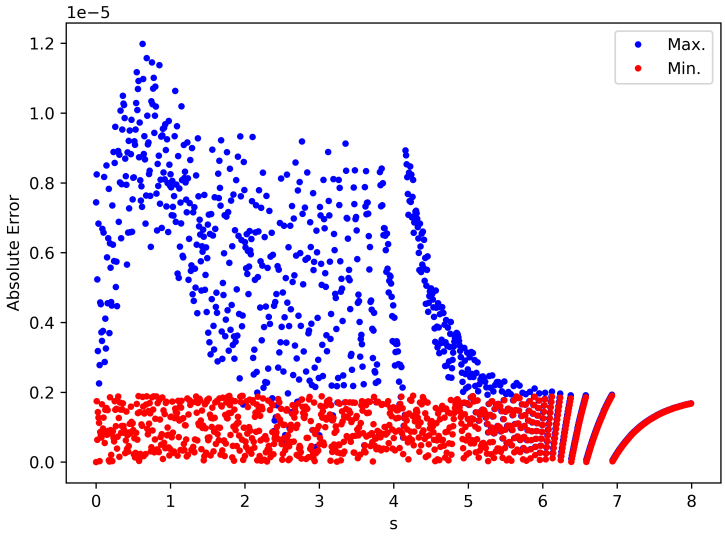

Figure 3.4: The minimum and maximum local absolute error before improvement. The experiment follows the same setting in Figure 3.3.

moves the error curve "closer" to the y-axis, decreasing the average and maximum absolute error.

The improved *intercept* table is updated by the following steps:

1) Calculate *offset* $= \frac{1}{2}(Max.Abs.Err. + Min.Abs.Err.)$ in each sub-interval.

2) Add the *offset* back to each *intercept* entry.

Figure 3.5 demonstrate the absolute error before and after the advancement. The maximum absolute error within [0,8] declines from 12×10^{-6} to around 6×10^{-6}, which is nearly half of the original value. Table 3.1 lists the experimental result of the tanh approximation with various configurations. The results displays that the improved *intercept* table effectively mitigates approximation errors.

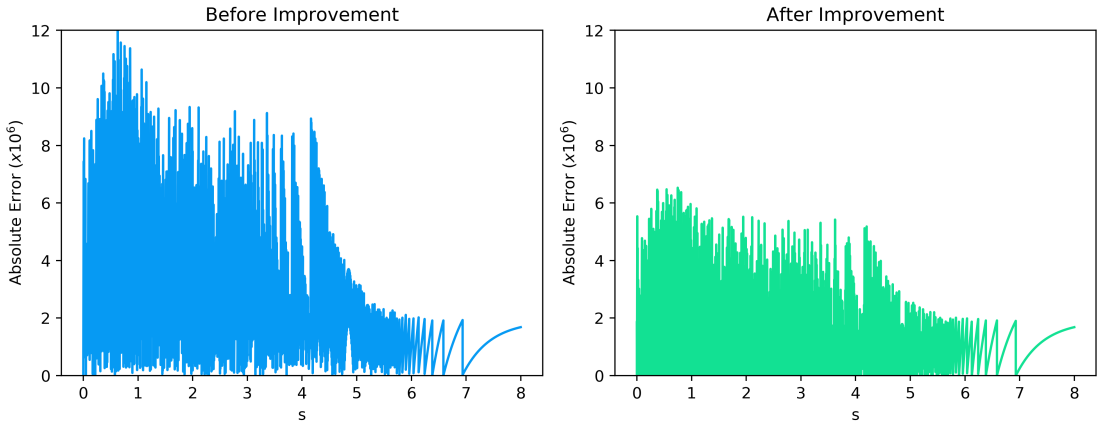

Figure 3.5: The absolute error comparison before and after the improvement. The absolute error using the improved *intercept* table is about half of our first approach. The number formats follows the same setting in Figure 3.3.

Let us look deeper into the data. Unsurprisingly, larger LUT entries and wider LUT instances yield better results. Using 8-bit LUT address (256 entries) with the improved method, generates 7×10^{-6} to 5×10^{-5} average absolute error, while the maximum error is about one magnitude higher (5×10^{-5} to 5×10^{-4}). By widen the LUT address to 10 bits (1024 entries), the maximum and average error can go down to 7.6×10^{-6} and 1.6×10^{-6}, respectively, almost approaching the limit of <20,19> output numbers[1]. But the cost of memory usage of 10-bit-entry LUTs is four times higher than using 8-bit-entry ones. It is truly a performance-cost trade-off that a designer should consider. Table 3.1 provides a handy reference for building a 1-st order piecewise-linear tanh approximation.

[1]The LSB of a <20,19> number represents 2^{-19}, which is about 2×10^{-6}. The maximum error 7.6×10^{-6} indicates that only the last two bits are off.

Table 3.1: Errors on the interval [0,8] and memory usage of the tanh approximator

LUT Addr.	*intercept* Format	*slope* Format	δs	Absolute Error		Absolute Error (Improved Method)		Memory Usage (Kb)
				Avg.	Max.	Avg.	Max.	
8 bits	<15,15>	<8,8>	6 bits	7.467e-5	5.294e-4	4.904e-5	4.530e-4	5.75
			8 bits	5.270e-5	2.145e-4	2.752e-5	1.229e-4	
		<10,10>	6 bits	6.448e-5	4.694e-4	4.558e-5	4.201e-4	6.25
			8 bits	4.193e-5	1.608e-4	2.346e-5	9.304e-5	
	<19,19>	<8,8>	6 bits	5.098e-5	4.798e-4	3.369e-5	4.223e-4	6.75
			8 bits	3.851e-5	1.574e-4	1.308e-5	7.924e-5	
			10 bits	2.870-e5	1.578e-4	1.308e-5	7.961e-5	
		<10,10>	6 bits	3.911e-5	4.186e-4	2.876e-5	3.844e-4	7.25
			8 bits	1.642e-5	1.084e-4	7.248e-6	5.502e-5	
			10 bits	1.650e-5	1.090e-4	7.243e-6	5.526e-5	
10 bits	<19,19>	<8,8>	6 bits	1.168e-5	1.228e-4	8.094e-6	1.086e-4	27
			8 bits	6.131e-6	3.354e-5	2.852e-6	1.758e-5	
			10 bits	6.190e-6	3.389e-5	2.851e-6	1.758e-5	
		<10,10>	6 bits	9.001e-6	1.002e-4	7.237e-6	9.720e-5	29
			8 bits	3.355e-6	1.368e-5	1.610e-6	7.602e-6	
			10 bits	3.378e-6	1.386e-5	1.611e-6	7.602e-6	

3.3.2 Hardware Tanh Implementation

Here we take the most precise approximation setting in Table 3.1, where each LUT has a 10-bit address; *intercept* is 19-bit wide, holding <19,19> numbers; *slope* is 10-bit wide, holding <10,10> numbers; δs has 8 bits available.

Figure 3.6 depicts the overall architecture of the non-linear function block. Before the linear combination sum-of-product s is sent to the tanh unit, the module first derive the absolute value of it and omit the most significant bit (MSB) (sign bit), as well as redundant least significant bits (LSBs), truncating it down to a 10-bit number.

For convenience, we use s^C to denote the $< 10, 10 >$ converted positive number in the rest of this section.

$$s^C = |s|_{34:25} \tag{3.8}$$

In our design, each LUT takes the whole 10 bits of s^C as input. Now the approximation can be expressed as:

$$\hat{f}(s) = \begin{cases} -\hat{f}\left(|s|\right) & , s < 0 \\ 1 - 2^{-19} & , s \geq 8 \\ \left(slope(s^C) \times |s|_{24:18} + intercept(s^C)\right) \times 2^{-19} & , 0 \leq s \leq 8 \end{cases} \tag{3.9}$$

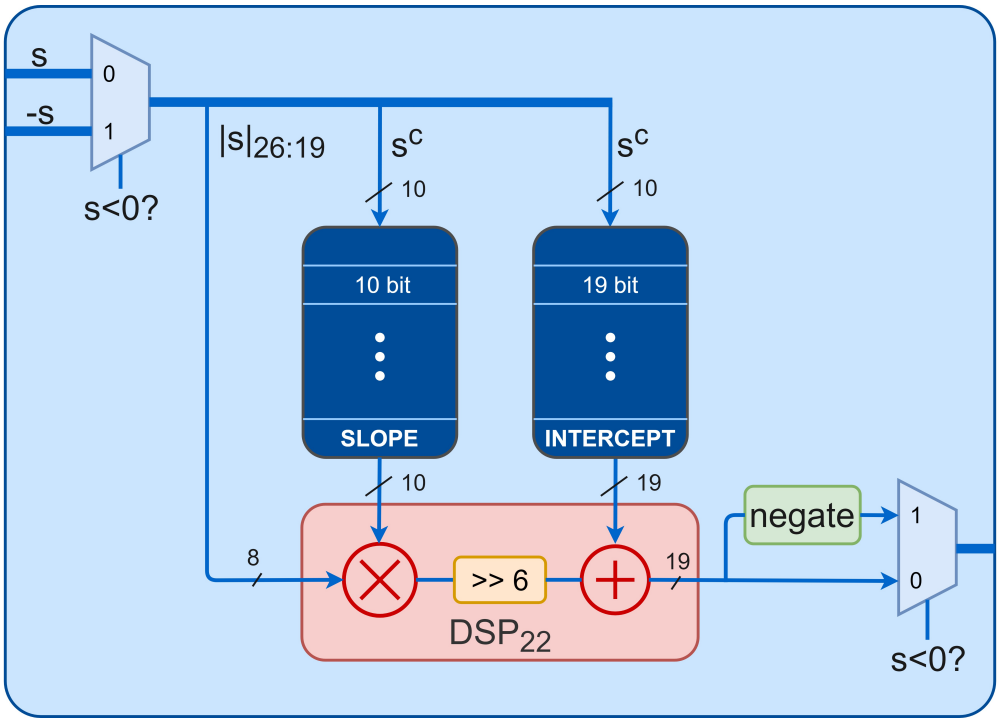

Figure 3.6: The architecture of the non-linear function (tanh) unit.

As shown in Figure 3.6, the multiplication and addition is dispatched back the DSP array in each neuron, more specifically, DSP_{22}, to eliminate the need of additional arithmetic units. The only part that uses CLBs are simple MUXs, concatenations, and a 2's complement generator. Furthermore, two LUTs are comprised of two-port block memories, that is, one Tanh core can serve two reservoir neurons simultaneously, making our design more efficient.

Chapter 4

Experimental Results and Analysis

To verify our work, the hardware ESN architecture is tested on two distinct tasks. Two major versions of ESN are implemented, targeting two different applications. The first one is tested and optimized for the classical nonlinear autoregressive moving average (NARMA) dataset [49]; The second is fine tuned for a a real-world symbol detection task in an orthogonal frequency division multiplexing (OFDM) system [40]. The major differences between these two applications in terms of hardware architecture are summarized in table 4.1. In short, the symbol detection task requires more accurate approximation of the non-linear function, which is explained in more details in section 3.3.

All experiments are conducted using post-synthesis timing simulation with Xilinx Vivado 2018.3. The environment targets Virtex-7 VC707 FPGA evaluation board. The initial reservoir weights and output weights are trained on a Ubuntu server using Python 3.7 and a modified pyESN [36] framework.

4.1 NARMA10 Datasets

First introduced in [10], the NARMA system is widely used for testing RNN performances. We follow the 10_{th}-order parameter setup in [49] to build the testing vectors. Modeling such system is challenging since the nonlinearity and the relative long-term memory are being stressed. The

Table 4.1: Major ESN architecture differences between two applications.

	NARMA10 Dataset	Wireless Symbol Detection
# of input nodes	1	4
# of reservoir neurons	20/50/100	16
# of output neurons	1	2
Range of inputs	normalized to $(-1, 1)$	
Range of training labels[1]	$(0.1, 0.8)$	(-8, 8)
Range of learned output weights[1]	(-20, 20)	(-50,000, 50,000)
Sustainable tanh accuracy loss	$\lvert 4 \times 10^{-3} \rvert$	$\lvert 4 \times 10^{-5} \rvert$
Tanh output bit length	16 bits	20 bits

system can be described as:

$$y(t+1) = 0.3y(t) + 0.05y(t)\sum_{i=0}^{9} y(t-1) + 1.5y(t-9)y(t) + 0.1 \tag{4.1}$$

4.1.1 Experiment Results

The ESN is trained with three different settings. Small-size (20-neuron) to midium-size (100-neuron) ESNs are tested to demonstrate the scalability and versatility of the architecture. Table 4.2 lists the experimental setup of three implementations using the same proposed architecture. All three sizes of ESN are achieved by only 20 physical neurons in the circuit. That is, the same smaller size ESN core is capable of processing larger size ESN data, with the help of proper peripheral circuits. For example, updating 100 neurons using 20 physical neurons can be realized by calculating each 20 neuron outputs at a time. This is a design trade-off between speed and resources. In this case, five iterations are taken to finish up one time step of the ESN. Circuit

[1]The range of training labels and learned output weights may vary depending on random seeds in NARMA dataset and real test data in the wireless communication.

designers can choose the optimal number of physical neurons for the target application and device. Our solution uses a temporary cache to store the partial calculated states. The global state memory will then updates its value from the cache once all 100 neuron outputs are calculated.

To carry out the best performance, the length of the training and testing sequence are varied. Distinct regularization factors in equation (2.6) are selected to mitigate over-fitting. Detailed numbers are also summarized in Table 4.2.

Table 4.2: NARMA experimental setup and performances

	20 Neurons	50 Neurons	100 Neurons
# of physical neurons		20	
Training sequence (steps)	1000	2000	8000
Regularization factor	0	10^{-8}	2×10^{-7}
Testing sequence (steps)	200	1000	1000
Training NMSE	0.136	0.142	0.203
Testing NMSE	0.246	0.132	0.103
Testing NMES (HW)	0.228	0.141	0.126

We test the performance of ESN in both software and circuit simulation using the metric, normalized mean square error (NMSE):

$$\begin{aligned} NMSE &= \frac{\frac{1}{n}\sum_{t=1}^{n}\left|y^{test}(t) - y^{label}(t)\right|^2}{\frac{1}{n-1}\sum_{t=1}^{n}\left|y^{label}(t) - E\left(y^{label}(t)\right)\right|^2} \\ E\left(y^{label}(t)\right) &= \frac{1}{L}\sum_{i=0}^{L-1} y_i^{label}(t), \end{aligned} \tag{4.2}$$

where $y^{test}(t)$ is the ESN output given the test set in time step t; y^{label} is the ground truth; and $|\cdot|$ denotes the Euclidean norm. NMSE gives an idea of how well the prediction follows the actual

NARMA system. The smaller the number, the better the system is inferred.

Figure 4.1 shows that our design models NARMA10 system genuinely well, almost as good as using the Python model. In the case of 20 neurons, the outcome from the hardware circuit performs even slightly better than the Python inference. We believe this is due to the small output deviation between the hardware and software intrinsically introduced by the fixed-point numerical system. The hardware model tries to follow the software prediction as precise as possible, but the small deviation may fall closer to the actual NARMA output. A slightly better NMSE could possibly be obtained. This situation does not appear in the case of 50 and 100 neurons, where software and hardware ESNs both perform excellently, leaving smaller gap for the deviation to "luckly" drive NMSE better.

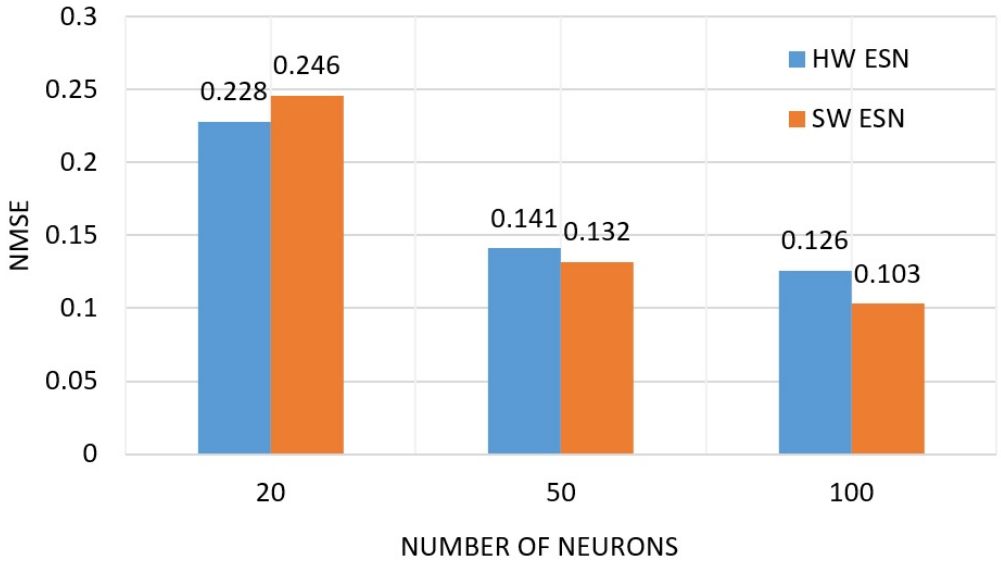

Figure 4.1: NARMA10 performances in NMSE (lower the better).

Figure 4.2 further visualizes the performance of our hardware implementation. The dotted green line indicates the actual NARMA system output; the blue and orange line represents the ESN inferences using software and hardware, respectively. As shown in the graph, the pattern of

ESN outputs follow ground truth NARMA pretty well. Furthermore, hardware ESN produce almost identical numbers as software ESN. This demonstrates the performance capability of the architecture.

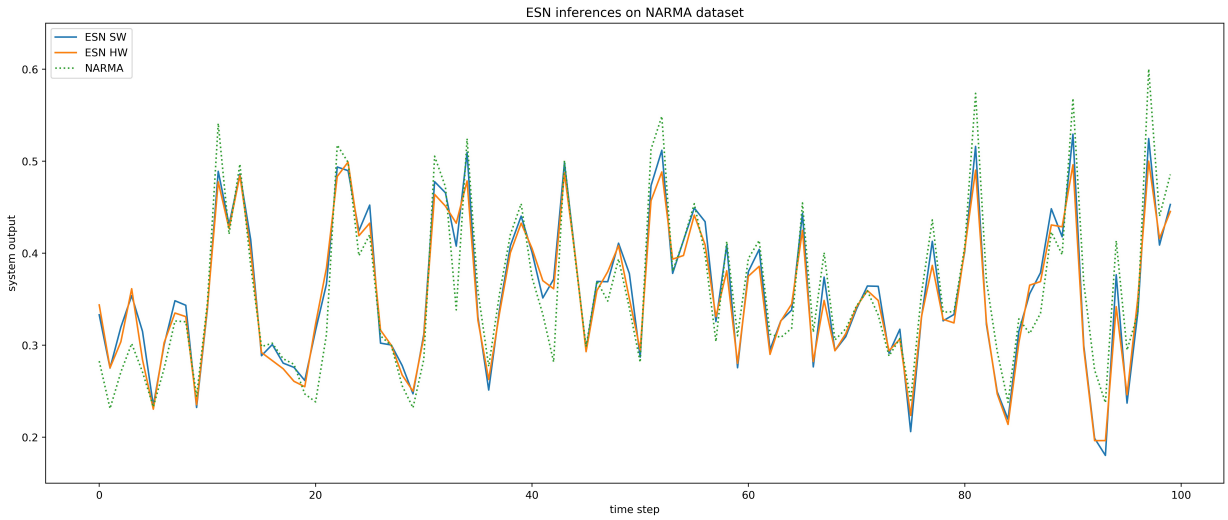

Figure 4.2: ESN inferences vs. NARMA. This graph shows the partial output of the 100-neuron ESNs.

4.2 OFDM Symbol Detection

In wireless communications systems, the training overhead is extremely costly and limited. ESN, due to its fast convergence property, becomes a perfect candidate for wireless communications related applications. In this section, we show the ESN-based symbol detection application in orthogonal frequency division multiplexing (OFDM) wireless communications systems introduced by Lianjun Li et al. [40]. More related works are conducted in [25, 46, 50, 51, 53, 71, 72].

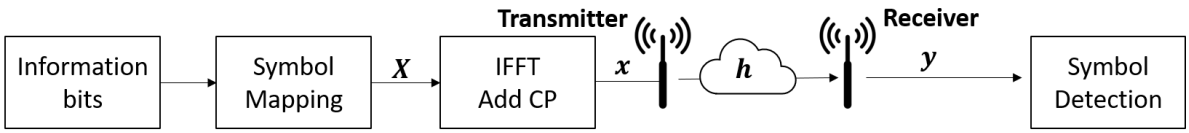

Figure 4.3: OFDM system procedures.

4.2.1 OFDM System

In this section, we briefly introduce the working mechanism of an OFDM wireless communications systems, which benefits understanding the symbol detection problem discussed later. In OFDM systems, data are packaged and transmitted in the unit of "OFDM frame". Each OFDM frame is comprised of "OFDM symbols". The transmitting-receiving procedure is depicted in Figure 4.3. At the transmitter side, the i_{th} OFDM symbol in frequency domain is denoted as:

$$\mathbf{X}_i \triangleq [X_i(0), \cdots, X_i(k), \cdots, X_i(N_{sc}-1)]^T \in \mathbb{C}^{N_{sc}}, \tag{4.3}$$

where $\cdot^T$ is the matrix transpose operator; N_{sc} represents the total number of sub-carriers; $X_i(k)$ is the major information symbol carried on sub-carrier k, randomly chosen from a defined modulation table. For instance, $\{+1, -1\}$ is the table of binary phase-shift keying (BPSK) modulation, while $\{+1+j, -1+j, +1-j, -1-j\}$ is the table of quadrature phase-shift keying (QPSK) modulation.

Afterward, $\mathbf{X}_i$ is converted to time domain via inverse fast Fourier transform (IFFT). The last N_{cp} samples of the time domain signal is then copied and attached to the beginning of the signal as a cyclic prefix (CP). The i_{th} transmitted OFDM symbol in time domain can be expressed as:

$$\mathbf{x}_i \triangleq [x_i(0), \cdots, x_i(n), \cdots, x_i(N_{cp}+N_{sc}-1)]^T \in \mathbb{C}^{(N_{cp}+N_{sc})}, \tag{4.4}$$

where $x_i(n)$ is the nth sample of the ith OFDM symbol in time domain. Note that $\mathbf{X}_i$ can be reversely obtained by removing the CP of $\mathbf{x}_i$ followed by a fast Fourier transform (FFT). The time domain OFDM frame is a series of OFDM symbols, expressed as:

$$\mathbf{x} \triangleq [(\mathbf{x}_0)^T, \cdots, (\mathbf{x}_i)^T, \cdots, (\mathbf{x}_{N-1})^T]^T \in \mathbb{C}^{N(N_{cp}+N_{sc})} \tag{4.5}$$

where N is the total number of OFDM symbols in a frame. The frame $\mathbf{x}$ is then transmitted to the receiver over the wireless channel. The received time domain OFDM frame $\mathbf{y}$ can be denoted as:

$$\mathbf{y} = f(\mathbf{x} \circledast \mathbf{h}) + \sigma, \tag{4.6}$$

where f is the function represents the non-linear effects of the transmitting-receiving procedure; $\circledast$ is the convolution operator; σ is the Gaussian noise; $\mathbf{h} = [h_0, h_1, \cdots, h_{N_{cp}}]^T \in \mathbb{C}^{(N_{cp}+1)}$ indicates the wireless channel. At the receiver side, the i_{th} received time domain OFDM symbol can be denoted similarly as at the transmitter:

$$\mathbf{y}_i \triangleq [y_i(0), \cdots, y_i(n), \cdots, y_i(N_{cp} + N_{sc} - 1)]^T \in \mathbb{C}^{(N_{cp}+N_{sc})} \tag{4.7}$$

The main objective of a symbol detection task is recovering the transmitted OFDM symbol $\mathbf{X}_i$ from received time domain information $\mathbf{y}_i$. Usually, training overhead is pre-defined in OFDM systems to accommodate symbol detection. For example, in Wi-Fi system [29], the first four OFDM symbols are arranged as the training sequence known by both side. In the next subsection, we will discuss how the training overhead can be utilized in ESN-based symbol detection.

4.2.2 ESN-based Symbol Detection

Conventional symbol detection methods are model-based, they often first evaluate the wireless channel $\mathbf{h}$, and then apply various signal processing techniques to recover the transmitted symbol. However, due to the non-linearity of the channel, $\mathbf{x}$ and $\mathbf{y}$ is difficult to model. Conventional methods often suffer from model mismatch. On top of that, the accuracy of symbol detection highly depends on the channel estimation, hence suffers from estimation mismatch as well.

On the other hand, ESN-based symbol detection is a learning-based method which does not require explicit channel estimation. It is designed to learn the direct mapping from $\mathbf{y}$ to $\mathbf{x}$ without the interference of estimation mismatch. The nature of ESN solves aforementioned issues of conventional methods. Furthermore, the ESN-based symbol detection only utilizes the training overhead in existing OFDM systems. Unlike other learning-based symbol detection methods whose frame definition has to be modified, this method can be directly applied to OFDM systems with less effort. The training and testing procedure of ESN-based symbol detection is described as below.

The ESN is trained over the first four OFDM symbols in an OFDM frame. An input-label tuple is used to express the training data set:

$$\Phi \triangleq \{\mathbf{I}, \mathbf{D}\} = \{[\mathbf{y_1}, \mathbf{y_2}, \mathbf{y_3}, \mathbf{y_4}], [\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}, \mathbf{x_4}]\} \tag{4.8}$$

Upon completion of the training , the rest of received OFDM symbols $\mathbf{y}_i (4 < i < N)$ are sent into ESN to recover the transmitted symbols.

4.2.3 Experiment Results

The performance of symbol detection using ESN, as well as three conventional methods, i.e., least mean square (LMS), least square (LS), and Comb pilot interpolation (Comb) [24], are evaluated on a software defined radio platform (Figure 4.4), built with GNU Radio and two USRP N210 transceivers. This means the experiment data is acquired on a real system but not from simulations.

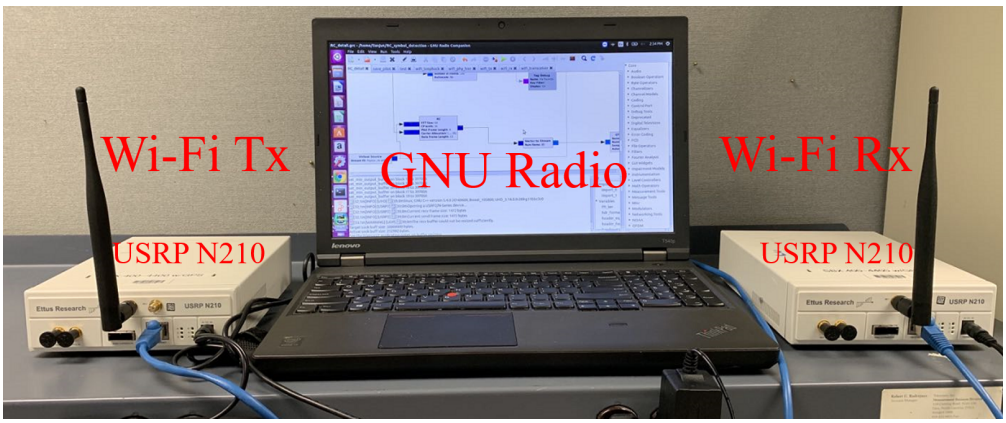

Figure 4.4: The software-defined radio test bed. © IEEE. Reprinted, with permission, from Lianjun Li et al., 2020 29th Wireless and Optical Communications Conference (WOCC), p.1-6 (2020) [40].

The experiment includes three scenarios. The first scenario is line-of-sight with near distance (LOS_Near), where the transmitter and receiver have a 10-inch direct LOS path. The second scenario is LOS with further distance (LOS_Far), where the distance between transceivers is increased to 10 feet. In addition, a 30dB attenuator is added on the receiver to further decrease the signal power. The last scenario is non-line-of-sight (NLOS), where no direct paths occur between the transmitter and receiver. The distance between the transceiver is set to 5 feet, and a 30dB attenuator is deployed on the receiver as well. The bit error rate (BER) of symbol detection task using the five methods (namely, LMS, Comb, LS, software ESN, and the hardware ESN) under these 3 scenarios are shown in Figure 4.5.

As suggested in Figure 4.5, ESN-based methods achieve the best performance amongst all. We

argue that this is due to the efficient learning capability of ESN. A direct mapping from the input to the output can be inferred without exploring channel estimation as required by other three conventional methods. The hardware ESN generates slightly worse result than software ESN, nevertheless, it still dominates the other three algorithms. This demonstrates that our work offers a great opportunity of solving the real-world wireless communication challenge.

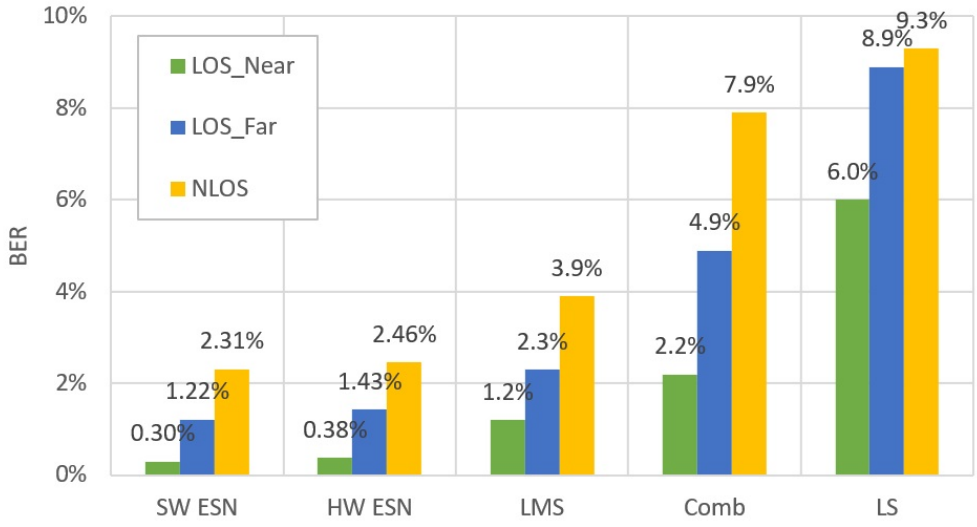

Figure 4.5: Performance of hardware/software ESN-based symbol detection, LMS, Comb and LS.

To ensure the proposed data is not merely a special case, the experiment is repeated 20 to 28 times, depending on the data length collected under each scenario. Table 4.3 and Figure 4.6 summarizes the analysis of variance using software (SW) and hardware (HW) ESN symbol detector. With the LOS_Far and NLOS setup, both shows one data point having around 15% BER, indicating a strongly interfered packet. Excluding the two data points, Figure 4.6 shows a clear trend which BER steps up from LOS_Near to LOS_Far to NLOS. The standard deviation of BER also increases when the transmission condition becomes harder. It is expected due to the longer physical distance and the added attenuator. The low standard deviation suggests that except under the very rare worst conditions, the ESN-based method is stable and produces consistent results.

Table 4.3: Analysis of BER variance (%) for ESN symbol detection

Setup	# of data	Mean	Medium	Standard deviation
LOS_Near SW	28	0.30	0.05	0.67
LOS_Near HW	28	0.39	0.06	0.77
LOS_Far SW	23	1.23	0.10	3.13
LOS_Far HW	23	1.43	0.19	3.83
NLOS SW	21	2.31	0.33	3.77
NLOS HW	21	2.47	0.36	3.95

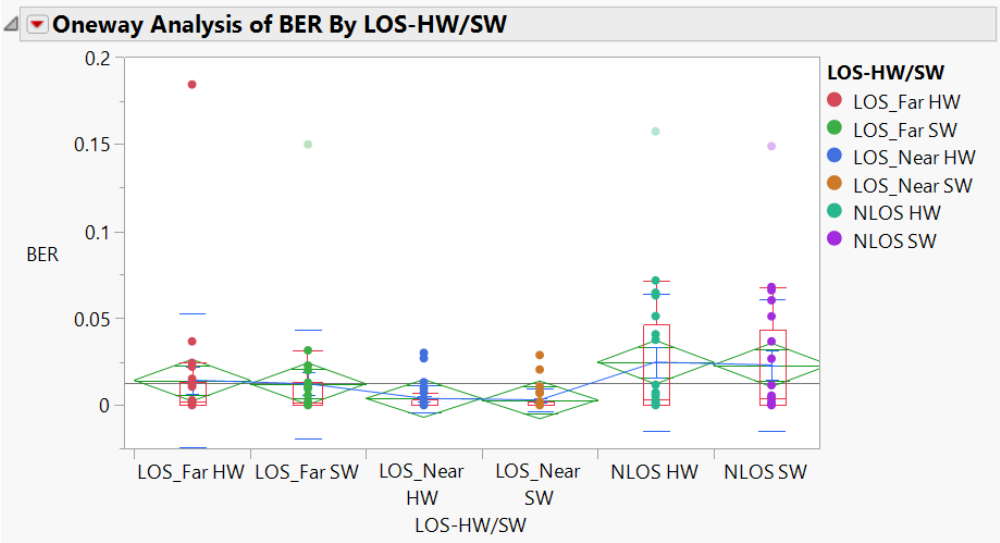

Figure 4.6: Analysis of BER variance for ESN symbol detection.

To verify the performance of the hardware architecture, Figure 4.7 & 4.8 provide the direct visualization of the ESN output. The first one illustrates the actual raw output of both software and hardware ESN. The data is divided into two parts since there are two output nodes in the network, representing the real part and imaginary part of the number, respectively. The following

one depicts the relative difference between the two implementations. As suggested in the graphs, our ESN hardware predictor follows the software version tightly in time domain.

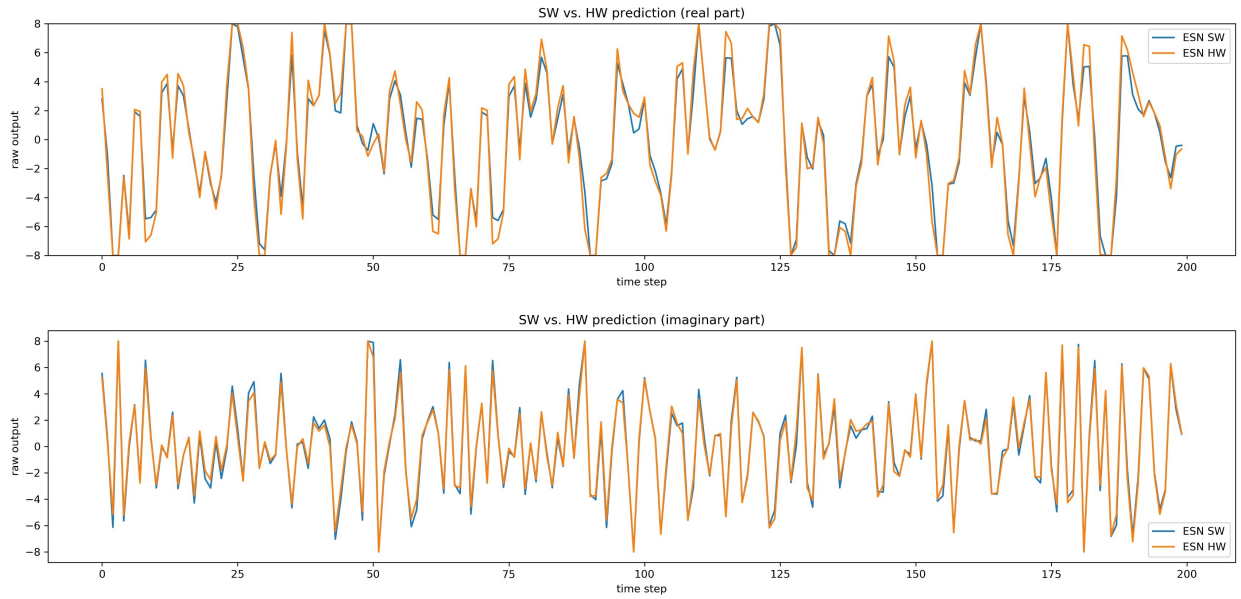

Figure 4.7: HW vs. SW ESN on the symbol detection task in time domain.

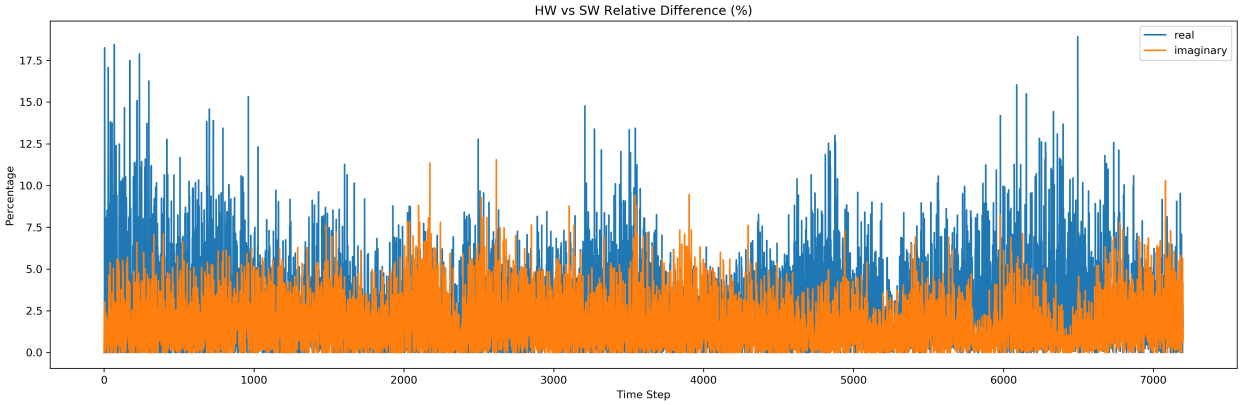

Figure 4.8: Relative difference between HW and SW ESN outputs.

In the frequency domain, Figure 4.9 demonstrates a comparison between the constellation diagram of the predicted symbol detection results. The upper diagrams are generated by the software ESN

approach, while the bottom two are yielded by the corresponding hardware implementation. Figure 4.9 (a) represents an easy data point where both SW and HW ESN predicts nearly perfect results. Figure 4.9 (b) describes a worse data point whose inferred symbols scatter in a larger area, resulting a BER at around 3.5%. As shown in the figure, our hardware ESN design produces very similar output as the more precise software version, even after the data is transferred from the time domain back to the frequency domain. This reinforces that the proposed architecture is capable of processing complex information, such as real-world communication data.

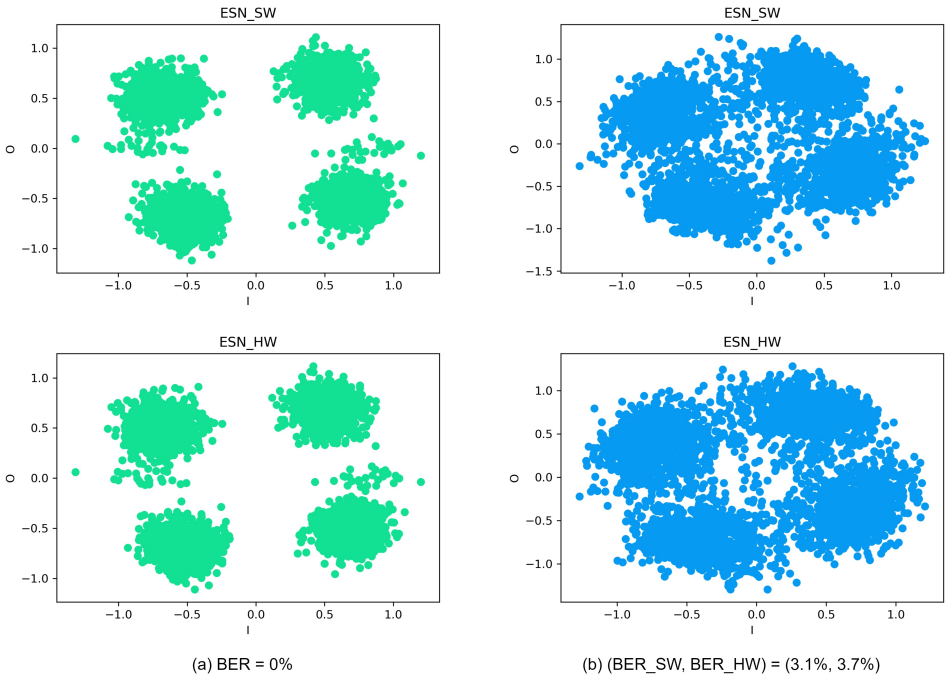

Figure 4.9: Example constellation diagrams. The constellation diagrams of the prediction result using software and hardward ESN show very similar pattern with slight variation. (a) represents a good data point where the BER for both HW and SW are zero. (b) depicts a worse data where the BERs are around 3.5%.

Chapter 5

Conclusions and Future Works

5.1 Conclusions

In this thesis, we propose a cost-efficient ESN architecture on FPGA, which exploits the advanced feature of Xilinx DSP48E1 DSP unit. The proposed work includes a linear combination processor with negligible deployment of CLBs, and a high-accuracy non-linear function approximator, through the help of only 9 DSP units in each neuron. This architecture is especially suitable for low-cost Xilinx Artix-7 family devices, which offers a higher DSP-CLB ratio. It also adapts to all Xilinx devices with DSP48E1 & DSP48E2 units. The ESN FPGA implementation is verified by NARMA 10 dataset, showing the FPGA implementation of ESN could reach comparable performances as the software-based ESN with various neural configurations. In addition, the work is tested on an OFDM symbol detection task, whose data is gathered from a physical SDR platform. The experiment result demonstrates that the FPGA based ESN outperforms the conventional channel equalization methods in a wide-variety of environments.

5.2 Future Works

There are several improvements that can be extended to the ESN architecture. In terms of evaluating a digital design work, there are three main aspects we care about, power, performance, and cost. The following subsection 5.2.1 will cover the future works that could enhance the three key metrics.

On the other hand, one can expand the utility of the ESN hardware by introducing it to the existing eco-system, which will be discussed in subsection 5.2.2

5.2.1 Power, Performance, & Cost

The current design can process 9 input data in the multiply-accumulation stage in each neuron. This is fairly fast, yet not all 9 multiplications are necessary. Almost 20-60% of the reservoir weights in matrices are sparse, indicating that applying these weights always give a zero and thus can be skipped. Since the reservoir weight matrix is given prior to the start of the computation, we can add a table storing the sparsity information, telling the DSP units to avoid the redundant weights and leap to useful data. This advancement can potentially cut-down the execution time and save memory space.

A large use of the block-memory stems from the two look-up-tables in the activation function core. When the trained output weights are large, users may have no choice but increase the size of the tables to remain accuracy. This problem can be solved if the models are pre-trained with hardware-friendly activation functions, e.g., hard tanh and K-Tanh[37]. If the model can achieve a comparable performance compared to using the conventional hyperbolic tangent, the hardware approximation may be largely simplified.

In addition, power consumption, due to the arising number of battery driven devices, can be decisive when choosing the applicable implementation. Human brain can easily contains 10^{12} neurons connected through 10^{15} synapses [22], while powered by only 25 Watts [35]. Spiking neural network (SNN), mimicking the mechanism of biological neural system, has shown a great potential in reducing power expenditure of artificial neurual networks [55]. There is a possibility

adopting spikes with temporal encoding scheme [14, 38, 64–70] in the ESN architecture to further mitigate the need of multipliers and power consumption.

5.2.2 Expansion of Utility

ESN can work well in unmanned aerial vehicles [31]. It is also effective for dealing with the symbol detection problem in the physical layer of wireless communication, as demonstrated in section 4.2. However, the ESN computational hardware cannot work solely in a practical scene.

Several pre-processing can be included, such as weight initialization and the on-board learning ability. At the beginning of the ESN environment setup, the internal weights are chosen randomly with regularization. This could be approached by implementing a chain of 16-bit linear feedback shift registers to generate a set of random numbers. The next step is hard but worth exploring, since generating $\mathbf{W}$ requires calculating the maximum eigenvalue of the matrix. The good news is, we can simply apply a smaller user-defined spectral radius to avoid complex computation. This is doable in most cases, since ESN is tolerant when the maximum eigenvalue is only slightly larger than 1 [32, 44]. Furthermore, the initialization of the weights is only required once upon powering-on. Processing speed would not be the highest priority. Instead of placing a huge parallel block, we could utilize iterations to find the maximum eigenvalue if necessary.

On-board training capability can also be extremely useful when combined with on-line training algorithms. This is especially true for applications that may require model re-training on-the-fly, such as wireless communication. A popular on-line learning method is the recursive least square (RLS) method [23], which has been proven to work well on FPGA in [6] & [42].

Finally, an easy solution to dock with the existing eco-system is critical for developers. The current design has a register interface for parameters and configurations. The work can be extended to

support standard module-to-module interfaces, such as AMBA AXI4[8] and AXI4-Stream[9]. A possible approach could be using AXI4/AXI4-lite for configuration and using AXI4-Stream for input data flows. The support of standard protocols could provide simpler accessibility for ARM soft-cores, which are widely embedded in Xilinx Zync-7000 System-on-Chips (SoCs). A soft-core is easier to program and flexible, thus making the ESN more versatile and adaptable in diverse scenarios.

Bibliography

[1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, August 2008. doi: 10.1109/IEEESTD.2008.4610935.

[2] IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, pages 1–640, January 2009. doi: 10.1109/IEEESTD.2009.4772740.

[3] Ahmed Akif. *FIR Filter Features on FPGA*. PhD thesis, 2018.

[4] M. L. Alomar, V. Canals, V. Martínez-Moll, and J. L. Rosselló. Low-cost hardware implementation of Reservoir Computers. In *2014 24th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–5, September 2014. doi: 10.1109/PATMOS.2014.6951899.

[5] Miquel L. Alomar, Vincent Canals, Nicolas Perez-Mora, Víctor Martínez-Moll, and Josep L. Rosselló. FPGA-Based Stochastic Echo State Networks for Time-Series Forecasting. https://www.hindawi.com/journals/cin/2016/3917892/, December 2015. ISSN 1687-5265.

[6] Piotr Antonik, Anteo Smerieri, François Duport, Marc Haelterman, and Serge Massar. FPGA Implementation of Reservoir Computing with Online Learning. In *24th Belgian-Dutch Conference on Machine Learning (BENELEARN)*, 2015.

[7] L. Appeltant, M. C. Soriano, G. Van der Sande, J. Danckaert, S. Massar, J. Dambre, B. Schrauwen, C. R. Mirasso, and I. Fischer. Information processing using a single dynamical node as complex system. *Nature Communications*, 2(1):468, September 2011. ISSN 2041-1723. doi: 10.1038/ncomms1476.

[8] Arm. AMBA AXI and ACE Protocol Specification: AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite, 2003.

[9] Arm. AMBA 4: AXI4-Stream Protocol Specification, 2010.

[10] A.F. Atiya and A.G. Parlos. New results on recurrent network training: Unifying the algorithms and accelerating convergence. *IEEE Transactions on Neural Networks*, 11(3):697–709, May 2000. ISSN 1941-0093. doi: 10.1109/72.846741.

[11] S.L. Bade and B.L. Hutchings. FPGA-based stochastic neural networks-implementation. In *Proceedings of IEEE Workshop on FPGA's for Custom Computing Machines*, pages 189–198, April 1994. doi: 10.1109/FPGA.1994.315612.

[12] Kangjun Bai and Yang Yi Bradley. A path to energy-efficient spiking delayed feedback reservoir computing system for brain-inspired neuromorphic processors. In *2018 19th International Symposium on Quality Electronic Design (ISQED)*, pages 322–328, March 2018. doi: 10.1109/ISQED.2018.8357307.

[13] Kangjun Bai and Yang Yi. DFR: An Energy-efficient Analog Delay Feedback Reservoir Computing System for Brain-inspired Computing. *ACM Journal on Emerging Technologies in Computing Systems*, 14(4):45:1–45:22, December 2018. ISSN 1550-4832. doi: 10.1145/3264659.

[14] Kangjun Bai, Jialing Li, Kian Hamedani, and Yang Yi. Enabling An New Era of Brain-inspired Computing: Energy-efficient Spiking Neural Network with Ring Topology. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2018. doi: 10.1109/DAC.2018.8465938.

[15] Kangjun Bai, Qiyuan An, and Yang Yi. Deep-DFR: A Memristive Deep Delayed Feedback Reservoir Computing System with Hybrid Neural Network Topology. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2019.

[16] Kangjun Bai, Qiyuan An, Lingjia Liu, and Yang Yi. A Training-Efficient Hybrid-Structured Deep Neural Network With Reconfigurable Memristive Synapses. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(1):62–75, January 2020. ISSN 1557-9999. doi: 10.1109/TVLSI.2019.2942267.

[17] Kangjun Bai, Yang Yi, Zhou Zhou, Shashank Jere, and Lingjia Liu. Moving Toward Intelligence: Detecting Symbols on 5G Systems Through Deep Echo State Network. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 10(2):253–263, June 2020. ISSN 2156-3365. doi: 10.1109/JETCAS.2020.2992238.

[18] Mariusz Bajger and Amos Omondi. Low-error, High-speed Approximation of the Sigmoid Function for Large FPGA Implementations. *Journal of Signal Processing Systems*, 52(2):137–151, August 2008. ISSN 1939-8115. doi: 10.1007/s11265-007-0140-z.

[19] William James Cody. *Software Manual for the Elementary Functions*. Prentice-Hall, Inc., USA, 1980. ISBN 978-0-13-822064-8.

[20] Intel Corporation. Intel Quartus Prime Software User Guides. https://www.intel.com/content/www/us/en/programmable/products/design-software/fpga-design/quartus-prime/user-guides.html.

[21] Intel Corporation. Intel Stratix 10 Logic Array Blocks and Adaptive Logic Modules User Guide, April 2020.

[22] Egidio D'Angelo, Sergio Solinas, Jesus Garrido, Claudia Casellato, Alessandra Pedrocchi, Jonathan Mapelli, Daniela Gandolfi, and Francesca Prestori. Realistic modeling of neurons and networks: Towards brain simulation. *Functional Neurology*, 28(3):153–166, 2013 -10- 17. ISSN 0393-5264.

[23] Behrouz Farhang-Boroujeny. *Adaptive Filters: Theory and Applications*. John Wiley & Sons, 2013.

[24] Joseph A. Fernandez, Daniel D. Stancil, and Fan Bai. Dynamic channel equalization for IEEE 802.11p waveforms in the vehicle-to-vehicle channel. In *2010 48th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 542–551, September 2010. doi: 10.1109/ALLERTON.2010.5706954.

[25] Kian Hamedani, Lingjia Liu, Shiya Liu, Haibo He, and Yang Yi. Deep Spiking Delayed Feedback Reservoirs and Its Application in Spectrum Sensing of MIMO-OFDM Dynamic Spectrum Sharing. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(02):1292–1299, April 2020. ISSN 2374-3468. doi: 10.1609/aaai.v34i02.5484.

[26] Geoffrey Hinton, Li Deng, Dong Yu, George E. Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N. Sainath, and Brian Kingsbury. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine*, 29(6):82–97, November 2012. ISSN 1558-0792. doi: 10.1109/MSP.2012.2205597.

[27] Nan-Sheng Huang, Jan-Matthias Braun, Jørgen Christian Larsen, and Poramate Manoonpong. A scalable Echo State Networks hardware generator for embedded systems using high-level synthesis. In *2019 8th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–6, June 2019. doi: 10.1109/MECO.2019.8760065.

[28] IEEE. IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pages 1–590, April 2006. doi: 10.1109/IEEESTD.2006.99495.

[29] IEEE. Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment: Enhanced Broadcast Service (Std 802.11-2012), 2012.

[30] Herbert Jaeger. The “echo state” approach to analysing and training recurrent neural networks-with an erratum note. *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report*, 148(34):13, 2001.

[31] Herbert Jaeger. Adaptive nonlinear system identification with Echo state networks. In *Proceedings of the 15th International Conference on Neural Information Processing Systems*, NIPS’02, pages 609–616, Cambridge, MA, USA, January 2002. MIT Press.

[32] Herbert Jaeger. A tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the "echo state network" approach. 2005.

[33] Herbert Jaeger and Harald Haas. Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, 2004. doi: 10.1126/science.1091277.

[34] Azarakhsh Jalalvand, Glenn Van Wallendael, and Rik Van De Walle. Real-Time Reservoir Computing Network-Based Systems for Detection Tasks on Visual Contents. In *2015 7th International Conference on Computational Intelligence, Communication Systems and Networks*, pages 146–151, June 2015. doi: 10.1109/CICSyN.2015.35.

[35] Eric R. Kandel, James H. Schwartz, Thomas M. Jessell, Steven A. Siegelbaum, and A. J. Hudspeth, editors. *Principles of Neural Science, Fifth Edition*. McGraw-Hill Education / Medical, New York, 5th edition edition, October 2012. ISBN 978-0-07-139011-8.

[36] Clemens Korndörfer. pyESN.

[37] Abhisek Kundu, Alex Heinecke, Dhiraj Kalamkar, Sudarshan Srinivasan, Eric C. Qin, Naveen K. Mellempudi, Dipankar Das, Kunal Banerjee, Bharat Kaul, and Pradeep Dubey. K-TanH: Efficient TanH For Deep Learning. *arXiv:1909.07729 [cs, stat]*, June 2020.

[38] Jialing Li, Chenyuan Zhao, Kian Hamedani, and Yang Yi. Analog hardware implementation of spike-based delayed feedback reservoir computing system. In *2017 International Joint*

Conference on Neural Networks (IJCNN), pages 3439–3446, May 2017. doi: 10.1109/IJCNN.2017.7966288.

[39] Jialing Li, Kangjun Bai, Lingjia Liu, and Yang Yi. A deep learning based approach for analog hardware implementation of delayed feedback reservoir computing system. In *2018 19th International Symposium on Quality Electronic Design (ISQED)*, pages 308–313, March 2018. doi: 10.1109/ISQED.2018.8357305.

[40] Lianjun Li, Lingjia Liu, Jianzhong Charlie Zhang, Jonathan D. Ashdown, and Yang Yi. Reservoir Computing Meets Wi-Fi in Software Radios: Neural Network-based Symbol Detection using Training Sequences and Pilots. In *2020 29th Wireless and Optical Communications Conference (WOCC)*, pages 1–6, May 2020. doi: 10.1109/WOCC48579.2020.9114937.

[41] Yong-bo Liao, Hong-mei Li, and Wen-chang Li. The Application of FPGA Based Real-Time Processing ESN in Pattern Recognition and Waveform Generation. *DEStech Transactions on Engineering and Technology Research*, 0(ameme), 2016. ISSN 2475-885X. doi: 10.12783/dtetr/ameme2016/5753.

[42] Yongbo Liao, Hongmei Li, Yalan Shen, and Wenchang Li. An FPGA Based Real Time Reservoir Computing System for Neuromorphic Processors. In *2018 3rd Asia-Pacific Conference on Intelligent Robot Systems (ACIRS)*, pages 82–86, July 2018. doi: 10.1109/ACIRS.2018.8467252.

[43] Yongbo Liao, Hongmei Li, and Zongbo Wang. FPGA Based Real-Time Processing Architecture for Recurrent Neural Network. In Fatos Xhafa, Srikanta Patnaik, and Albert Y. Zomaya, editors, *Advances in Intelligent Systems and Interactive Applications*, Advances in Intelligent Systems and Computing, pages 705–709, Cham, 2018. Springer International Publishing. ISBN 978-3-319-69096-4. doi: 10.1007/978-3-319-69096-4_99.

[44] Mantas Lukoševičius. A Practical Guide to Applying Echo State Networks. In Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller, editors, *Neural Networks: Tricks of the Trade: Second Edition*, Lecture Notes in Computer Science, pages 659–686. Springer, Berlin, Heidelberg, 2012. ISBN 978-3-642-35289-8. doi: 10.1007/978-3-642-35289-8_36.

[45] Wolfgang Maass, Thomas Natschläger, and Henry Markram. Real-Time Computing Without Stable States: A New Framework for Neural Computation Based on Perturbations. *Neural Computation*, 14(11):2531–2560, November 2002. ISSN 0899-7667. doi: 10.1162/089976602760407955.

[46] Somayeh Susanna Mosleh, Lingjia Liu, Cenk Sahin, Yahong Rosa Zheng, and Yang Yi. Brain-Inspired Wireless Communications: Where Reservoir Computing Meets MIMO-OFDM. *IEEE Transactions on Neural Networks and Learning Systems*, 29(10):4694–4708, October 2018. ISSN 2162-2388. doi: 10.1109/TNNLS.2017.2766162.

[47] Y. Paquot, F. Duport, A. Smerieri, J. Dambre, B. Schrauwen, M. Haelterman, and S. Massar. Optoelectronic Reservoir Computing. *Scientific Reports*, 2(1):287, February 2012. ISSN 2045-2322. doi: 10.1038/srep00287.

[48] Vishal Passricha and Rajesh Kumar Aggarwal. Convolutional Neural Networks for Raw Speech Recognition. *From Natural to Artificial Intelligence - Algorithms and Applications*, December 2018. doi: 10.5772/intechopen.80026.

[49] Ali Rodan and Peter Tino. Minimum Complexity Echo State Network. *IEEE Transactions on Neural Networks*, 22(1):131–144, January 2011. ISSN 1941-0093. doi: 10.1109/TNN.2010.2089641.

[50] Rubayet Shafin, Lingjia Liu, Jonathan Ashdown, John Matyjas, Michael Medley, Bryant Wysocki, and Yang Yi. Realizing Green Symbol Detection via Reservoir Computing: An

Energy-Efficiency Perspective. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–6, May 2018. doi: 10.1109/ICC.2018.8422425.

[51] Rubayet Shafin, Lingjia Liu, Vikram Chandrasekhar, Hao Chen, Jeffrey Reed, and Jianzhong Charlie Zhang. Artificial Intelligence-Enabled Cellular Networks: A Critical Path to Beyond-5G and 6G. *IEEE Wireless Communications*, 27(2):212–217, April 2020. ISSN 1558-0687. doi: 10.1109/MWC.001.1900323.

[52] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv:1409.1556 [cs]*, April 2015.

[53] Hao Song, Jianan Bai, Yang Yi, Jinsong Wu, and Lingjia Liu. Artificial Intelligence Enabled Internet of Things: Network Architecture and Spectrum Access. *IEEE Computational Intelligence Magazine*, 15(1):44–51, February 2020. ISSN 1556-6048. doi: 10.1109/MCI.2019.2954643.

[54] Gouhei Tanaka, Toshiyuki Yamane, Jean Benoit Héroux, Ryosho Nakane, Naoki Kanazawa, Seiji Takeda, Hidetoshi Numata, Daiju Nakano, and Akira Hirose. Recent advances in physical reservoir computing: A review. *Neural Networks*, 115:100–123, July 2019. ISSN 0893-6080. doi: 10.1016/j.neunet.2019.03.005.

[55] Amirhossein Tavanaei, Masoud Ghodrati, Saeed Reza Kheradpisheh, Timothée Masquelier, and Anthony Maida. Deep learning in spiking neural networks. *Neural Networks*, 111:47–63, March 2019. ISSN 0893-6080. doi: 10.1016/j.neunet.2018.12.002.

[56] S. Vassiliadis, Ming Zhang, and J.G. Delgado-Frias. Elementary function generators for neural-network emulators. *IEEE Transactions on Neural Networks*, 11(6):1438–1449, November 2000. ISSN 1941-0093. doi: 10.1109/72.883475.

[57] David Verstraeten, Benjamin Schrauwen, and Dirk Stroobandt. Reservoir computing with

stochastic bitstream neurons. In *Proceedings of the 16th Annual Prorisc Workshop*, pages 454–459, 2005.

[58] Xilinx. 7 Series FPGAs Configurable Logic Block User Guide (UG474), September 2016.

[59] Xilinx. 7 Series DSP48E1 Slice User Guide (UG479), March 2018.

[60] Xilinx. UltraScale Architecture DSP Slice User Guide (UG579), 2019.

[61] Xilinx. Vivado Design Suite User Guide - Getting Started, October 2019.

[62] Yang Yi, Yongbo Liao, Bin Wang, Xin Fu, Fangyang Shen, Hongyan Hou, and Lingjia Liu. FPGA based spike-time dependent encoder and reservoir design in neuromorphic computing processors. *Microprocessors and Microsystems*, 46:175–183, October 2016. ISSN 0141-9331. doi: 10.1016/j.micpro.2016.03.009.

[63] Kyongsik Yun, Alexander Huyen, and Thomas Lu. Deep Neural Networks for Pattern Recognition. *arXiv:1809.09645 [cs]*, September 2018.

[64] Chenyuan Zhao, Jialing Li, and Yang Yi. Making neural encoding robust and energy efficient: An advanced analog temporal encoder for brain-inspired computing systems. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–6, November 2016. doi: 10.1145/2966986.2967052.

[65] Chenyuan Zhao, Bryant T. Wysocki, Clare D. Thiem, Nathan R. McDonald, Jialing Li, Lingjia Liu, and Yang Yi. Energy Efficient Spiking Temporal Encoder Design for Neuromorphic Computing Systems. *IEEE Transactions on Multi-Scale Computing Systems*, 2(4):265–276, October 2016. ISSN 2332-7766. doi: 10.1109/TMSCS.2016.2607164.

[66] Chenyuan Zhao, Jialing Li, Hongyu An, and Yang Yi. Energy efficient analog spiking temporal encoder with verification and recovery scheme for neuromorphic computing systems. In *2017*

18th International Symposium on Quality Electronic Design (ISQED), pages 138–143, March 2017. doi: 10.1109/ISQED.2017.7918306.

[67] Chenyuan Zhao, Yang Yi, Jialing Li, Xin Fu, and Lingjia Liu. Interspike-Interval-Based Analog Spike-Time-Dependent Encoder for Neuromorphic Processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(8):2193–2205, August 2017. ISSN 1557-9999. doi: 10.1109/TVLSI.2017.2683260.

[68] Chenyuan Zhao, Kian Hamedani, Jialing Li, and Yang Yi. Analog Spike-Timing-Dependent Resistive Crossbar Design for Brain Inspired Computing. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 8(1):38–50, March 2018. ISSN 2156-3365. doi: 10.1109/JETCAS.2017.2765892.

[69] Chenyuan Zhao, Qiyuan An, Kangjun Bai, Bryant Wysocki, Clare Thiem, Lingjia Liu, and Yang Yi. Energy Efficient Temporal Spatial Information Processing Circuits Based on STDP and Spike Iteration. *IEEE Transactions on Circuits and Systems II: Express Briefs*, pages 1–1, 2019. ISSN 1558-3791. doi: 10.1109/TCSII.2019.2945690.

[70] Chenyuan Zhao, Lingjia Liu, and Yang Yi. Design and Analysis of Real Time Spiking Neural Network Decoder for Neuromorphic Chips. In *Proceedings of the International Conference on Neuromorphic Systems*, ICONS '19, pages 1–4, Knoxville, TN, USA, July 2019. Association for Computing Machinery. ISBN 978-1-4503-7680-8. doi: 10.1145/3354265.3354280.

[71] Zhou Zhou, Lingjia Liu, Vikram Chandrasekhar, Jianzhong Zhang, and Yang Yi. Deep Reservoir Computing Meets 5G MIMO-OFDM Systems in Symbol Detection. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(01):1266–1273, April 2020. ISSN 2374-3468. doi: 10.1609/aaai.v34i01.5481.

[72] Zhou Zhou, Lingjia Liu, and Hao-Hsuan Chang. Learning for Detection: MIMO-OFDM

Symbol Detection Through Downlink Pilots. *IEEE Transactions on Wireless Communications*, 19(6):3712–3726, June 2020. ISSN 1558-2248. doi: 10.1109/TWC.2020.2976004.

[73] Zhou Zhou, Lingjia Liu, Shashank Jere, Jianzhong, Zhang, and Yang Yi. RCNet: Incorporating Structural Information into Deep RNN for MIMO-OFDM Symbol Detection with Limited Training. *arXiv:2003.06923 [cs, eess]*, March 2020.