FPGA Reservoir Computing Networks for Dynamic Spectrum Sensing

Osaze Y. Shears

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 Cindy Yi, Chair
JoAnn M. Paul
Cameron D. Patterson

May 5, 2022
Blacksburg, Virginia

Keywords: field-programmable gate array, high-level synthesis, machine learning, reservoir computing, software-defined radio, spectrum sensing, neuromorphic computing

FPGA Reservoir Computing Networks for Dynamic Spectrum Sensing

Osaze Y. Shears

ABSTRACT

The rise of 5G and beyond systems has fuelled research in merging machine learning with wireless communications to achieve cognitive radios. However, the portability and limited power supply of radio frequency devices limits engineers' ability to combine them with powerful predictive models. This hinders the ability to support advanced 5G applications such as device-to-device (D2D) communication and dynamic spectrum sharing (DSS). This challenge has inspired a wave of research in energy efficient machine learning hardware with low computational and area overhead. In particular, hardware implementations of the delayed feedback reservoir (DFR) model show promising results for meeting these constraints while achieving high accuracy in cognitive radio applications. This thesis answers two research questions surrounding the applicability of FPGA DFR systems for DSS. First, can a DFR network implemented on an FPGA run faster and with lower power than a purely software approach? Second, can the system be implemented efficiently on an edge device running at less than 10 watts?

Two systems are proposed that prove FPGA DFRs can achieve these feats: a mixed-signal circuit, followed by a high-level synthesis circuit. The implementations execute up to 58 times faster, and operate at more than 90% lower power than the software models. Furthermore, the lowest recorded average power of 0.130 watts proves that these approaches meet typical edge device constraints. When validated on the NARMA10 benchmark, the systems achieve a normalized error of 0.21 compared to state-of-the-art error values of 0.15. In a DSS task, the systems are able to predict spectrum occupancy with up to 0.87 AUC in high noise, multiple input, multiple output (MIMO) antenna configurations compared to 0.99 AUC in other works. At the end of this thesis, the trade-offs between the approaches are analyzed, and future directions for advancing this study are proposed.

FPGA Reservoir Computing Networks for Dynamic Spectrum Sensing

Osaze Y. Shears

GENERAL AUDIENCE ABSTRACT

The rise of 5G and beyond systems has fuelled research in merging machine learning with wireless communications to achieve cognitive radios. However, the portability and limited power supply of radio frequency devices limits engineers' ability to combine them with powerful predictive models. This hinders the ability to support advanced 5G and internet-of-things (IoT) applications. This challenge has inspired a wave of research in energy efficient machine learning hardware with low computational and area overhead. In particular, hardware implementations of a low complexity neural network model, called the delayed feedback reservoir, show promising results for meeting these constraints while achieving high accuracy in cognitive radio applications. This thesis answers two research questions surrounding the applicability of field-programmable gate array (FPGA) delayed feedback reservoir systems for wireless communication applications. First, can this network implemented on an FPGA run faster and with lower power than a purely software approach? Second, can the network be implemented efficiently on an edge device running at less than 10 watts? Two systems are proposed that prove the FPGA networks can achieve these feats. The systems demonstrate lower power consumption and latency than the software models. Additionally, the systems maintain high accuracy on traditional neural network benchmarks and wireless communications tasks. The second implementation is further demonstrated in a software-defined radio architecture. At the end of this thesis, the trade-offs between the approaches are analyzed, and future directions for advancing this study are proposed.

Acknowledgments

As a first generation college student, the journey I have taken to arrive at this point has been challenging, and I owe all of my accomplishments to the network of family, friends, and colleagues that helped me get here.

My parents, Raynard Shears, Jaquinett Aaron, Roxanne Marr-Shears, and Joel Aaron, have made many sacrifices to allow me to attend college and to learn about what inspires me. It is important to recognize that not everyone has the privileges that I have had when it comes to access to education, a supportive family, and a plethora of others that are often overlooked. My aunts, Sonya Shears and Beverly Shears have also been pivotal in my development thanks to the care they provided to me throughout my life. I would like to acknowledge my fiancée, Gia Ha, who somehow bears to listen to my tangents and crazy ideas without losing her mind. In spite of this, Gia has been the one who has supported me most during this journey with love and compassion, and I cannot thank her enough. I want to extend this thanks to the Ha family as well, who have welcomed me and helped in times of need.

I would like to recognize the host of faculty and peers who helped with my undergraduate studies at George Mason University. Everett-Teejay Brown, who has been my most invaluable advisor, inspires me every day and served as something of a third father to me while at Geoge Mason. Teejay's dedication to helping others and to making the world a more inclusive place has had a wide impact, and motivates me to help others. Additionally, I acknowledge Dr. Jens-Peter Kaps, Dr. Pelin Kurtay, Dr. Craig Lorie, Daniel Pirkl and other members of the George Mason University and BAE Systems communities who have

aided me significantly during my transition to graduate school.

While at Virginia Tech, Dr. Yang Cindy Yi has been an incredible advisor and provided me with funding support, mentorship, and connections to other students and staff. Her leadership over the Multifunctional Integrated Circuits and Systems (MICS) lab has attracted many students who have supported my research at Virginia Tech. I would also like to thank Dr. JoAnn Paul for giving me the opportunity to take many interesting courses and inspiring me with her research, and Dr. Cameron Patterson for connecting me with employment opportunities. Additionally, the Bradley Fellowship provided by the Bradley Department of Electrical and Computer Engineering has been critical as a funding source for my graduate studies. Finally, I'd like to thank Zion Shears, Huy Dang, Armon Adibi, and Jason Pham, among countless others for helping me throughout my journey.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The past decade has been characterized by an unprecedented growth in machine learning. Seemingly every industry has found some way to integrate the various statistical modeling approaches that have emerged from this topic. Consider Tesla, one of the world's largest suppliers of electric vehicles. The company has invested heavily in the development of its Autopilot platform to enhance its vehicles with autonomous driving [1]. Autonomous driving is enabled by sophisticated deep neural network algorithms such as convolutional neural networks (CNNs), which are among the best methods to perform object recognition. This is demonstrated by the ImageNet benchmark results provided by [2]. The ImageNet benchmark, proposed in [3], tasks models with classifying millions of images into one of 1,000 different classes. The highest CNN performance recorded in [2, 4] was 89.2% in 2021. The automotive industry is not the only area where machine learning has been leveraged to create a world changing impact. Natural language processing and machine translation are two additional applications enabled by dedicated machine learning cores in smartphones. This technology enhances the ability for communities across the world to communicate with each other and paves the way for future innovation.

Breakthroughs in computer hardware have significantly contributed to the growth of machine learning applications. The graphics processing unit (GPU), originally designed to accelerate computer graphics rendering, has found new life in machine learning acceleration. In 2009, [5] was one of the first works to demonstrate GPUs' ability to parallelize neural network training. Using NVIDIA's general-purpose GPU programming framework, CUDA, the group found that the same large scale single-instruction multiple data (SIMD) architecture that could be used to improve graphics rendering performance could similarly be used to reduce matrix multiplication latency, which is a frequently used operation in machine learning. GPU acceleration is often used to rapidly train contemporary models. The combination of GPU acceleration paired with the introduction of TensorFlow, a machine learning framework developed by Google in 2016 [6], and Big Data fueled the growth of machine learning research and development. The trends analysis shown in Figure 1.1, conducted using Google Trends, visualizes the relationship between these technologies and search interest in machine learning [7].

Large-scale cloud computing platforms, such as Amazon Web Service (AWS), benefited from the application of GPUs to machine learning hardware. AWS servers equipped with NVIDIA's state of the art A100 Tensor Core GPUs provide efficient training platforms for dense machine learning models [8]. Mobile "edge" devices can leverage powerful cloud computing servers to bring accurate predictive capabilities to consumers. However, cloud computing is affected by communication latency, network bandwidth limits, unstable connections, and low privacy [9]. Moreover, it is impractical to implement cloud servers' specialized processing hardware on mobile devices due to size and energy constraints. According to NVIDIA, the A100 Tensor Core GPU has a typical power consumption of 200 watts [10]. This sharply contrasts the power consumption of a typical cellphone which is on the order of 10 watts.

Alternatively, consumer electronics companies such as Apple and Qualcomm have contributed to the growth of embedded machine learning at the edge. Both companies have produced several system-on-chip (SoC) integrated circuits (ICs) that feature dedicated ma-

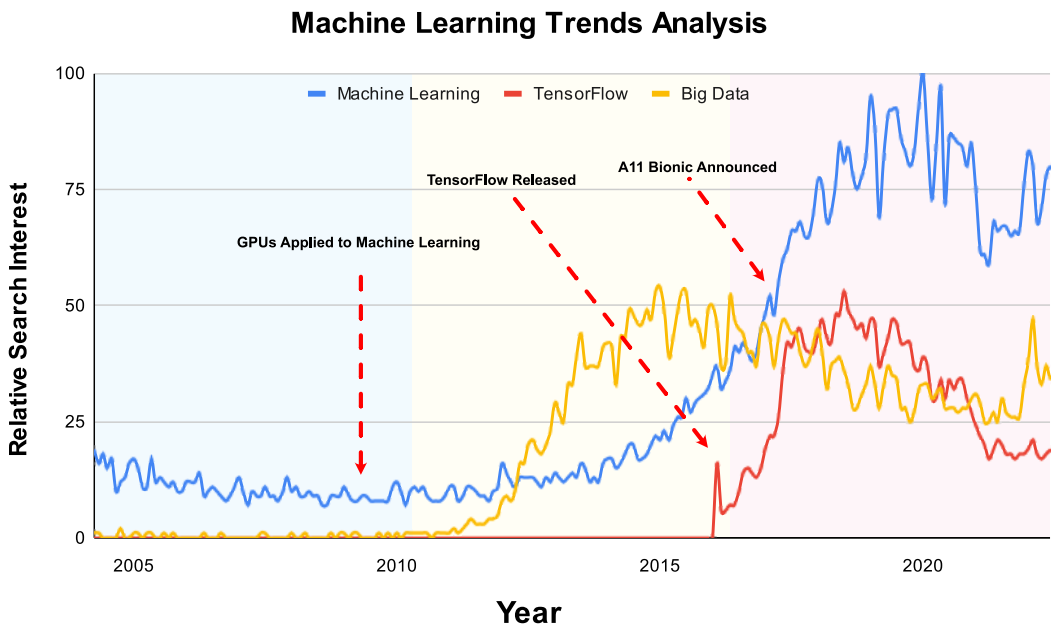

Figure 1.1: The impact of GPUs, TensorFlow and Big Data on machine learning research and development.

chine learning hardware accelerators. For example, the A15 Bionic chip featured in the iPhone 13 contains a 16 core neural engine that detects faces and enhances the quality of consumers' images [11]. This neural engine core contains 14 more cores than the precursor version of the engine announced with the iPhone X in 2017 [12]. Dedicated accelerators inside of smartphones have expanded smartphone capabilities by enabling intelligent consumer features such as natural language processing (NLP), language translation and facial recognition.

One of the most remarkable aspects of machine learning accelerators in smartphones is how little energy and circuit area are required to realize them. Experiments conducted by [13] suggest that the A15 Bionic has an average power consumption of 4.11 watts when tested against the SPECint2017 benchmark. This is nearly two orders of magnitude less than that recorded by NVIDIA's A100 GPU. Maintaining low power consumption is ideal in these types of mobile devices because of the limited energy made available by their batteries.

Energy constraints are critical in edge devices to provide continuous operation to the user for the longest amount of time possible.

This emphasis on power consumption has shifted the focus of machine learning. While first wave of the machine learning was revolutionized by the repurposing of the SIMD GPU architecture, the next wave of machine learning will emphasize simpler models with high energy efficiency. This effort will be supported by the growth of 5G and the internet-of-things (IoT) to create a world of intelligent, interconnected technologies. Dynamic spectrum sharing (DSS), which enables device-to-device (D2D) communication, is one area that can benefit from efficient models. This work hypothesizes that a neural network model known as the delayed feedback reservoir (DFR) can achieve a lower average power consumption, latency and area for spectrum occupancy predictions on FPGAs compared to general-purpose CPU-based neural network approaches.

1.2 Objectives

The objective of this thesis is to compare the average power consumption, latency and area of FPGA and CPU-based DFR implementations for spectrum sensing. This thesis aims to prove that FPGA implementations are ideal for this application compared to CPU-based implementations. The DFR, conceptualized in [14], has shown promising results for enabling low power machine learning in wireless edge devices [15–18]. In this thesis, the mathematical DFR algorithm is translated into a digital hardware implementation for a field-programmable gate array (FPGA). The FPGA is used to verify the DFR's plausibility as a method to support cognitive radio applications in edge devices, such as spectrum sensing. Lastly, the DFR architecture is optimized using high-level synthesis (HLS) design techniques and integrated with a software-defined radio (SDR) architecture to demonstrate its ability for real-time radio frequency (RF) data processing. After an examination of the model's predictive performance and efficiency on two different FPGAs, new directions for further

improvement are suggested. To the best of our understanding, this is the first work to demonstrate a mixed-signal reservoir computing system for spectrum sensing, as well as on a software-defined radio.

1.3 Thesis Organization

This chapters of this thesis are organized in a way that (1) introduces neural network models for time series prediction, (2) details the time series prediction task of interest, namely spectrum sensing, and (3) proposes the novel DFR architectures for FPGAs that can perform spectrum sensing on edge devices. Chapter 2 summarizes reservoir computing for time series prediction and the DFR model. Chapter 3 overviews previous approaches for applying machine learning and reservoir computing methods for wireless edge device machine learning tasks. Chapter 4 discusses FPGA design methodologies for creating hardware neural networks. Chapter 5 overviews the hybrid FPGA-ASIC DFR system developed and demonstrates the system's capabilities. Chapter 6 analyzes HLS techniques that can further optimize the DFR system. Chapter 7 overviews the HLS DFR system integrated with a software-defined radio. Chapter 8 concludes the work presented in this thesis and proposes directions for future research.

Chapter 2

Reservoir Computing

2.1 Recurrent Neural Networks

While on the one hand convolutional neural networks (CNNs) have revolutionized image processing capabilities, shown by their ImageNet classification performance in [4], recurrent neural networks (RNNs), on the other hand, have proven to be invaluable for processing time dependent data. Unlike CNNs, RNNs feature feedback connections between neurons which allow the network output values from previous data samples to influence the output of future samples. Figure 2.1 shows a simple one neuron RNN and an unrolled version of the network. Here, at a single time step t, both the input x_t and the feedback from the neuron h_t are weighed by u and v respectively before being passed through the neuron. The result from the neuron is multiplied by w to produce the network's output prediction. This behavior is equivalent to an unrolled version of the network where the output from the neuron at the first time step, h_0, is used to determine the output of the neuron at the second time step, h_1.

Traditional RNNs are trained using an approach called backpropagation through time (BPTT). Similar to standard backpropagation used for CNNs, BPTT works by using the error of the

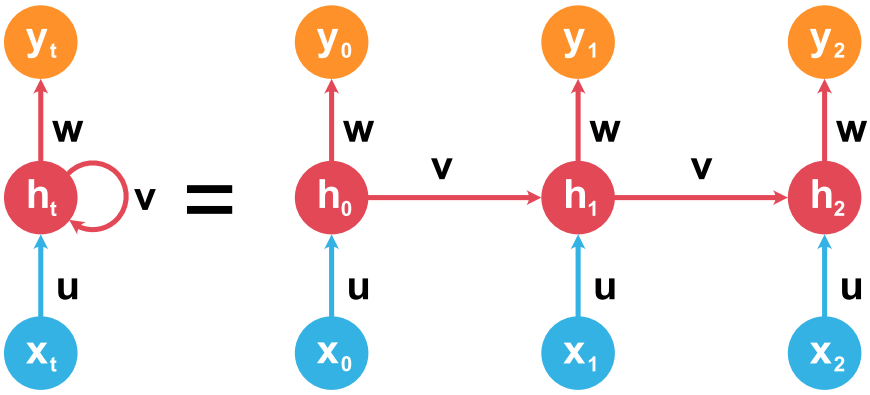

Figure 2.1: Basic architecture for a single neuron, recurrent neural network.

network's prediction to adjust the input, feedback, and output weights. The size of the time window used for BPTT can be adjusted to evaluate the influence of multiple time steps on the network's accuracy. However, many have found that the practice of increasing the time window leads to the exploding gradient and vanishing gradient issues [19].

The exploding gradient issue occurs when a particular feedback weight, v, has backpropagated error values (i.e., the gradients) whose product is greater than 1 for each weight update [19]. When this occurs, the gradient that is used to update the feedback weight will continue to grow exponentially. An exponentially large weight value significantly reduces the accuracy of the network by increasing the output error calculated by the cost function. A similar phenomena occurs when the product of the gradients for a feedback weight is less than 1 for each weight update, resulting in the vanishing gradient issue. In this scenario the feedback weights will not be updated and the output of the cost function will remain unchanged. The exploding and vanishing gradient problems have inspired researchers to study new forms of recurrent neural networks that avoid these issues: long short-term memory (LSTM) and reservoir computing (RC) networks.

LSTM networks, proposed in [19], are composed of memory cells: artificial neurons that retain data and use gates to control the flow of this data. Contemporary LSTM cells use three types of gates: (1) input gates to control when new data is processed by the cell and stored

in memory, (2) output gates to control when output is read from the cell, and (3) forget gates to clear information stored in the cell [20]. When paired with gradient clipping techniques, LSTMs effectively solve the exploding and vanishing gradient issues. LSTMs have achieved state of the art accuracy on time series prediction tasks such as speech recognition [21]. However, hardware implementations of LSTMs are still limited by their complexity since a large number of computationally intensive matrix multiplications are required for inference and BPTT-based training.

2.2 Reservoir Computing Networks

Reservoir computing (RC) networks were proposed as an alternative to traditional RNNs in the early 2000s [22]. The model proposed, called an echo state network (ESN), is composed of a "reservoir" of randomly and sparsely connected neurons. These neurons resemble those found in traditional neural networks with the sum of the weighed inputs being transformed by an activation function. The process by which the neurons are interconnected establishes the "echo state property". This property states that neurons shall be connected in such a way that the output of each neuron echoes throughout the network and establishes a type of short term memory [23, 24]. Training in ESNs only occurs on the output synapses, which avoids the exploding and vanishing gradient issues that were found with BPTT methods. Training can be performed with simpler linear regression algorithms such as ridge regression, ordinary least squares (OLS), and stochastic gradient descent. Jaeger demonstrated that an ESN was able to achieve similar performance to a standard RNN for modeling a moving average time series [22]. He also notes that ESNs may be preferable to other RNN techniques because of their simplified architecture and linear training at the output layer.

Figure 2.2 visualizes the organization of layers and neurons in a typical ESN. In this image, portions of a discretized input time series $u(n)$ are multiplied by input weights w_{in} and fed into the ESN's neurons. The neurons process the weighed input data, along with data from

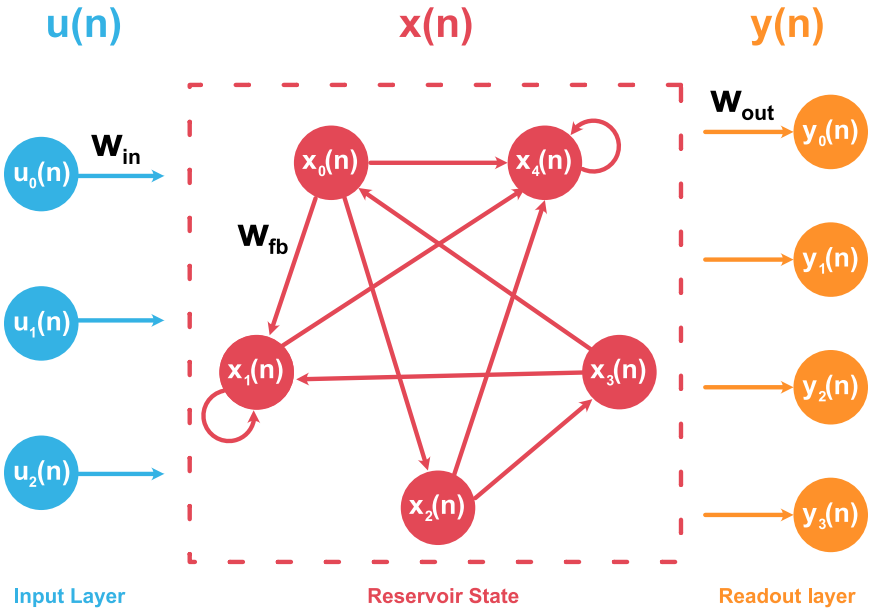

Figure 2.2: Basic architecture for a five neuron echo state network.

feedback connections multiplied by feedback weights w_{fb}. Neurons with connections to the output layer have their output weighed by w_{out} and accumulated to model or classify the time series.

Similar to ESNs, liquid state machines (LSMs) are reservoir computing networks that feature randomized internal connections and exclusive training at the output layer. LSMs differ from ESNs by leveraging spiking neurons as their basic processing elements as opposed to artificial neurons [25]. Spiking neurons work by accumulating a sum, called a membrane potential, of binary spikes that arrive from their synapses to produce an output [26]. These neurons only produce an output after a certain threshold has been reached, and the dynamics for how the values inside of these neurons can vary depending on the model.

While the ESN and LSM models introduced nearly two decades ago provide techniques to model sequential data with significantly less complex, they have not been used as widely as LSTM RNNs. One reason for this may be due to a lack of industry exploration on practical applications for reservoir computing. However, these models have the potential to revolutionize time series prediction. ESNs and LSMs have been demonstrated in several

applications, including spoken word detection, human activity recognition (HAR), and waveform classification [23, 27]. Some authors have also demonstrated their ability to be modeled and optimized using custom integrated circuit designs [28, 29]. The idea of implementing simplified RC models in hardware has been one of the major drivers in the research of an even more minimalistic model known as the delayed feedback reservoir.

2.3 Delayed Feedback Reservoir

The delayed feedback reservoir (DFR) is a type of reservoir computing network that features neurons arranged in a ring topology [14]. DFR networks have three primary features: an input layer where input data gets masked and sent to the reservoir, a reservoir layer where input gets passed through a non-linear transformation and delayed by N time steps, and an output layer where each of the reservoir neurons has their data weighed and accumulated to form an output prediction. The DFR simplifies the ESN architecture by utilizing only one non-linear activation function at the first neuron, while the remaining neurons simply delay the output value. The reservoir neurons are arranged in a simple ring architecture which further reduces complexity. The advantage of the DFR architecture over the ESN is its ability to be more easily realized in computer hardware as demonstrated by [14, 15, 17, 23, 27, 30]. Since the ESN features a large number of non-linear nodes, its hardware implementation would utilize significantly more energy and area compared to the minimal DFR model. Additionally, some works have shown that the DFR can accomplish nearly the same accuracy as the ESN on time series benchmarks such as NARMA10 [14, 30].

The DFR can be conceptualized as a type of folded ESN. Figure 2.3 helps to show this by picturing a simple DFR architecture, and the unrolled version of the architecture. In the first stage of the DFR, the masking stage, each input sample $u(t)$ is multiplied by an $N \times 1$ matrix of scalar values M. Once the input is masked, now $J(t)$, each of the N subsamples is sent to the DFR's non-linear node which is the first node of the reservoir. The

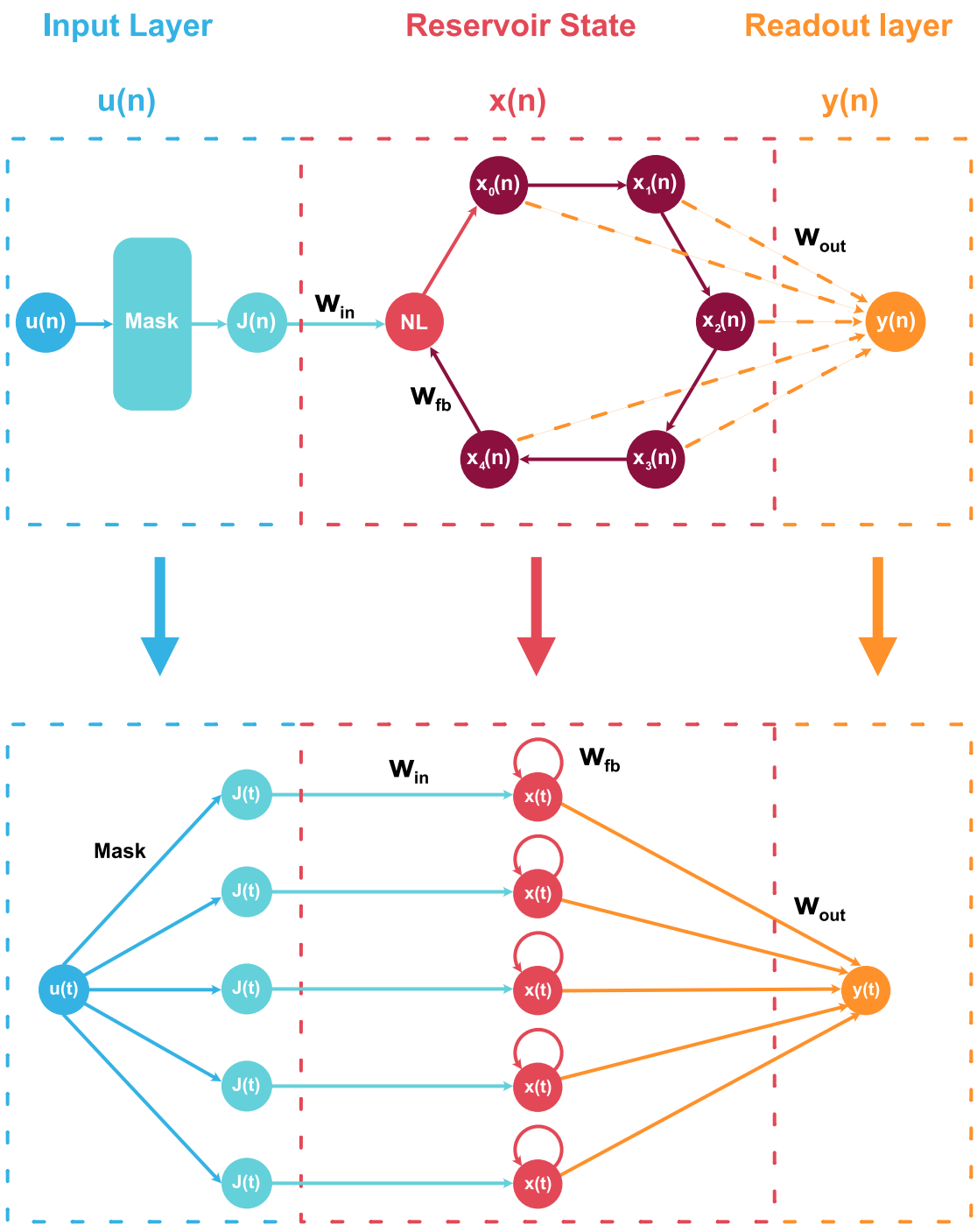

Figure 2.3: Basic architecture for a five neuron delayed feedback reservoir network.

DFR's non-linear node transforms the weighted sum of the input $J(t)$ and the time-delayed reservoir output $x(t-\tau)$ based on a non-linear transformation function $f(x)$. The output of the transformation function is then sent to the chain of N virtual nodes that delays the feedback by τ. Equation 2.1 below models the behavior of the DFR's reservoir stage. Here, the input gain γ and the feedback scale η are used to adjust the influence of the new input and feedback on the reservoir state.

$$x(t) = f[\gamma \cdot J(t) + \eta \cdot x(t-\tau)], \tag{2.1}$$

During the readout stage, an output prediction $\hat{y}(t)$ is generated by evaluating the weighted sum of the values in each of the virtual reservoir nodes. Equation 2.2 models the behavior of the DFR's readout stage. In this equation, w_i is the weight of virtual node i, τ is the feedback delay, and N is the number of virtual nodes.

$$\hat{y}(t) = \sum_{i=1}^{N} w_i \cdot x(t - \frac{\tau}{N}(N-i)), \tag{2.2}$$

In order to train the DFR to accurately model a given time series, linear regression can be applied at the readout layer. Equation 2.3 shows the ridge regression approach taken to generate a matrix of weights according to the previous reservoir states for each input.

$$\mathbf{w} = \frac{\mathbf{y} \cdot \mathbf{X}}{\mathbf{X} \cdot \mathbf{X^T} + \lambda \mathbf{I}}, \tag{2.3}$$

In this equation, $\mathbf{y}$ is an M length vector of expected outputs, $\mathbf{X}$ is a matrix of N reservoir node values for each of the M inputs, $\mathbf{I}$ is the identity matrix, and λ is the regularization coefficient.

The delayed feedback reservoir pioneered the idea of performing predictive tasks using neural network models with few non-linear transformations and recurrent connections. Future experiments inspired by the Appeltant's work, such as the folded-in-time neural network

proposed in [31], would demonstrate the capability of compacted neural networks on sophisticated image classification benchmarks such as MNIST, CIFAR-10 and SVHN.

Chapter 3

Reservoir Computing for Wireless Communications

3.1 Device-to-Device Communications

Cisco's annual internet report indicates that nearly 30 billion devices will be connected to the internet by 2023: an increase of 5 billion devices from 2021's estimate [32]. The company suggests that the growing number of devices communicating directly, without the need for human intervention, will attribute to this rapid growth. Machine-to-machine (M2M) connections enable internet of things (IoT) devices to share data necessary for automation. For example, a user's internet-connected car may transmit its position to a smart home controlling control the state of lights, air conditioning and security systems. Such an increase in data being transmitted over the air will continue to impact the bandwidth and latency of cellular communications.

This poses a challenge for emerging 5G networks. 5G networks are characterized by massive data transmission, strict latency requirements, and high device diversity compared to previous generations [33]. To fulfill these requirements, researchers have proposed device-

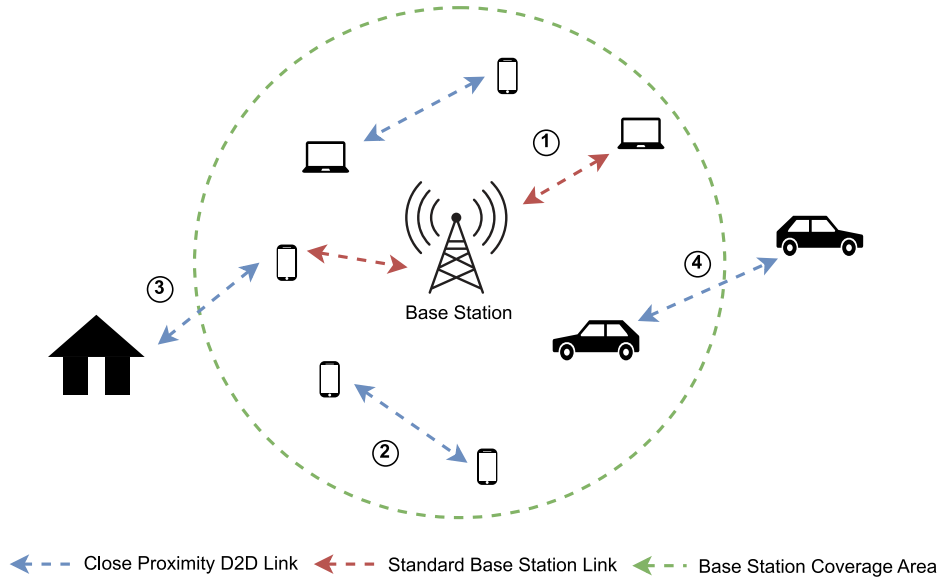

Figure 3.1: Sample use cases for device-to-device (D2D) communications.

to-device (D2D) links that circumvent communication through intermediary cellular base stations, and instead transmit data directly between the devices [34, 35]. These D2D links take advantage of the spatial locality between devices to achieve faster and energy efficient communication. D2D links are enabled by intelligent *spatial spectrum sensing* capabilities within a transmitting device [36]. By analyzing the cellular spectrum for idle subcarrier frequencies, a transmitting device can attempt to send data to a receiver without explicit time and frequency allocations [37]. Figure 3.1 depicts several example use cases for D2D enabled technologies. In this image, ① depicts a standard cellular link, ② depicts simple D2D transactions between in-coverage mobile phones, ③ depicts a scenario where D2D links can extend the base station coverage to out-of-range devices, and ④ depicts transactions between devices both in and out of the coverage area.

D2D communication requires devices to analyze the RF spectrum before attempting to communicate directly with other devices. This process, know as spectrum sensing, is an important step to avoid interfering with other users operating within the same frequency channel. Spectrum sensing is one of many cognitive radio applications where machine learn-

ing algorithms are finding new applications.

3.2 Spectrum Sensing

The number of connections that can be made over cellular frequency bands is limited by the amount of spectrum available to users. Spectrum sharing is an approach that aims to multiplex spectrum allocations by allowing secondary users to transmit data over a pre-allocated channel when it is not being actively used by its primary users. The secondary users are able to analyze the channel activity using machine learning spectrum sensing algorithms that learn the access patterns of other users over time to avoid interfering with their transmissions, while also providing efficient data transmissions of their own.

Traditionally, energy detection algorithms have been used for determining the availability of spectrum frequencies. In such approaches, the energy of a received signal is simply compared to a specific threshold to decide if it is safe to transmit over a wireless channel. In recent years, machine learning (ML) has been proposed as a more sophisticated method for realizing distributed spectrum analysis in embedded systems [38]. ML algorithms demonstrate the ability to learn temporal patterns in the signal's energy variations over time to dynamically predict whether or not a transmitter can send data over the channel. [39] provides a survey of the potential applications of machine learning techniques for wireless communication, including dynamic frequency allocation, cooperative spectrum sensing, and channel estimation. [40] overviews a joint recurrent neural network (RNN) and reinforcement learning approach which has been demonstrated by previous works to be well suited for practical spectrum sensing.

[17] demonstrates a reservoir computing approach where a delayed feedback reservoir (DFR) can be used to predict the occupancy of a wireless channel. This approach was noted to require low training complexity, few hardware resources, and minimal energy consumption, which enables it to be realized in energy-constrained wireless edge devices. Figure 3.2 illus-

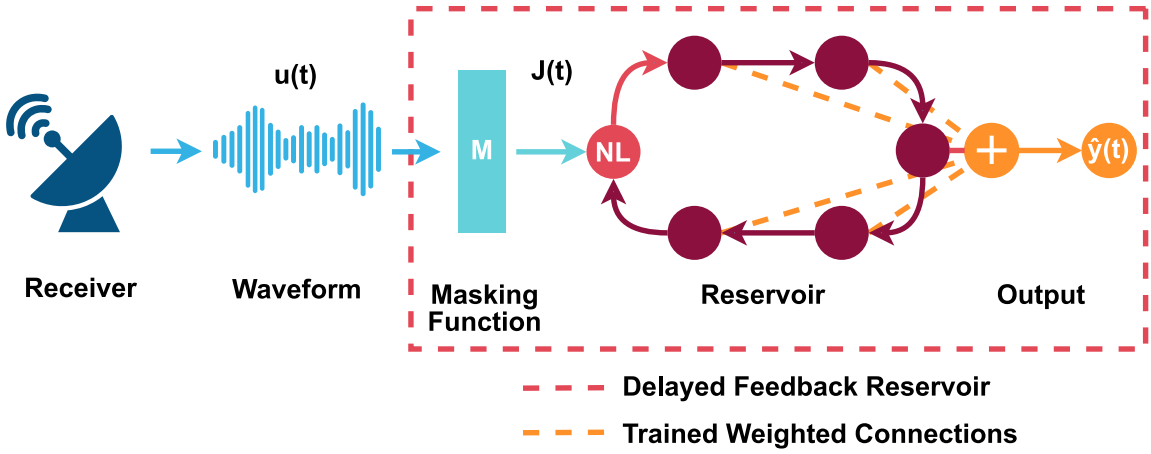

Figure 3.2: Execution of a delayed feedback reservoir (DFR) network used for spectrum sensing.

trates how the DFR can be interfaced with an RF transceiver to perform spectrum sensing.

Typically the energy observed at the antennas of an RF receiver is used to determine whether or not a subcarrier frequency is being occupied. During this process, in-phase and quadrature (IQ) samples are first read from the RF interface of a receiver and converted into digital representations using an analog-to-digital converter. Next, the measured energy of the RF spectrum for the sample is calculated using the amplitude equation:

$$E = \sqrt{I^2 + Q^2} \tag{3.1}$$

The resulting energy is finally processed by the spectrum sensing algorithm to determine whether or not the channel is occupied.

The challenge in spectrum sensing occurs when there is noise and interference present that affects the expected energy measurement. Figure 3.3 demonstrates this phenomena with different levels of Additive White Gaussian Noise (AWGN). Here, four constellation diagrams show the in-phase and quadrature components of four quadrature phase shift keying (QPSK) symbols. As the amplitude of AWGN increases, the ability to determine if the spectrum is

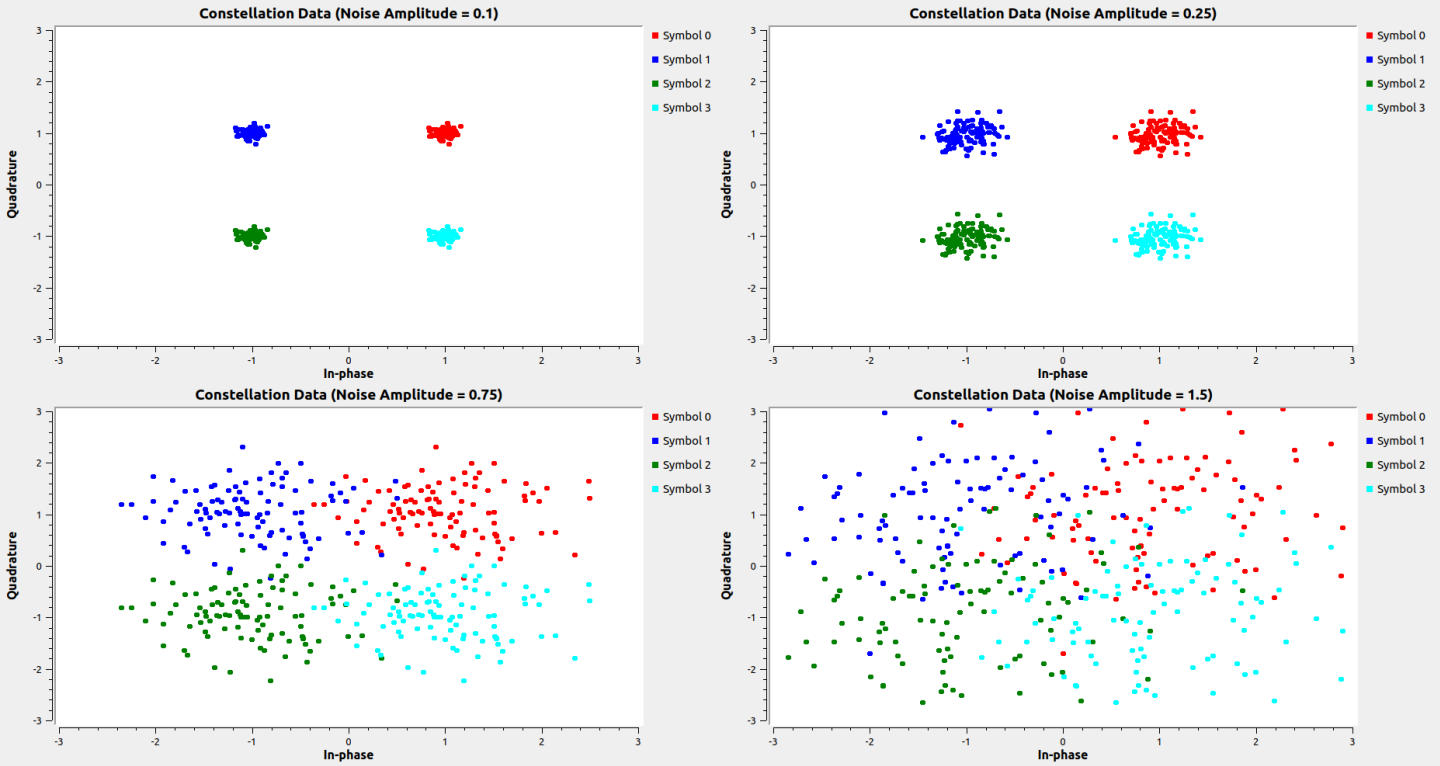

Figure 3.3: A visualization of the impact of noise on the ability to detect symbols from in-phase and quadrature data.

being utilized becomes increasingly difficult. This is where static energy detection methods would fail due to high variations in the energy calculated by Equation 3.1. On the other hand, dynamic spectrum sensing algorithms use previous spectrum occupancy behaviors to determine the probability of spectrum occupancy. [41] has recorded sets of spectrum occupancy data at various locations throughout Germany and the Netherlands. This data can be used to recreate sets of RF samples that model realistic spectrum occupancy activity.

3.3 Software-Defined Radio

Software-defined radios (SDRs) are often used to test cognitive radio tasks such as spectrum sensing [42–44]. SDRs have historically been a popular way to implement flexible RF

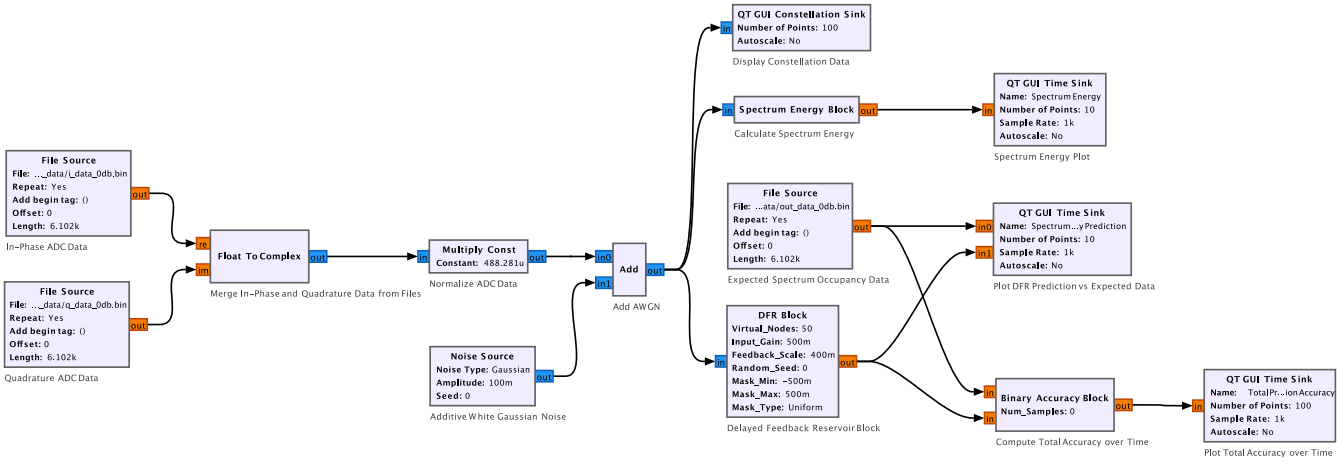

Figure 3.4: A GNU Radio Companion flowgraph featuring a DFR block for spectrum sensing.

functions using general purpose computing hardware [45]. The hardware within an SDR (1) acquires analog RF signals, (2) performs filtering on the signals, (3) converts the signals into sets of digital samples using an ADC, and (4) sends the samples to a computer. The computer uses software programs to perform additional filtering, demodulation, and processing on the samples for the end user. While the digital signal processing performed by an SDR may lack the speed of dedicated hardware, a primary advantage of this approach is the reconfigurability allowed through the use of different programs. An SDR can initially be configured to interpret a frequency modulated (FM) waveform at one frequency, then modified to demodulate QPSK symbols on a different frequency.

GNU Radio Companion is a popular SDR software programming framework [46]. The tool runs Python and C++ code that can perform filtering and modulation on RF samples for a variety of different SDR platforms. In GNU Radio, users create *flowgraphs* to configure the digital signal processing performed on the samples. Flowgraphs can be modified to include specialized cognitive radio blocks. Figure 3.4 shows a GNU Radio flowgraph configured to perform spectrum sensing using a custom delayed feedback reservoir block. This demonstrates one approach for testing the DFR model analyzed in this work.

Chapter 4

Field-Programmable Gate Array Development

4.1 FPGA Motivation

When determining the best approach to optimize the inference and training speed of neural network algorithms, it is common for practitioners to turn to hardware. While software optimizations allow these algorithms to be compiled and executed on general purpose processors (e.g., ARM, Intel), hardware optimizations tend to provide significant performance boosts in all aspects of machine learning. This is especially true for matrix multiplication which frequently involves hundreds to thousands of multiply and accumulate (MAC) operations per second [5]. Graphics processing units (GPUs) have been an incredibly popular and accessible approach for accelerating neural networks. These devices feature vast arrays of simple arithmetic logic units (ALUs) that can process data concurrently. The Compute Unified Device Architecture (CUDA) is a device programming library that enables developers to process data in parallel on NVIDIA GPUs. Popular machine learning frameworks like TensorFlow and PyTorch leverage CUDA to allow programmers to efficiently create machine learning models and accelerate training and testing performance [6, 47].

While GPUs have greatly benefited machine learning, many find that the performance of these models can be improved with hardware designed and fabricated intentionally for the purpose of machine learning. Apple's M1 SoC, which has gained worldwide recognition due to its radical improvements in energy efficiency and performance, features a neural engine built specifically to accelerate machine learning [48]. These neural engines, sometimes called neural processing units (NPUs), consist of dedicated hardware to perform machine learning tasks with high efficiency. Application-specific integrated circuit (ASIC) design approaches for machine learning have been used by other companies such as Google and NVIDIA, and provide the best way to achieve efficient machine learning [10, 49]. ASIC-based accelerators have significantly lower execution times and consume much less power than general purpose computing approaches, with growing research in analog computing methods further highlighting this advantage [50, 51]. However, these approaches can be expensive if designers determine that changes need to be made to the circuit. This is due to the high monetary and time cost of designing, fabricating, and testing the ASICs. Typically this process takes up to one or more years for small designs, and even longer for intricate SoC designs.

Field-programmable gate arrays (FPGAs) strike a balance between general purpose and accessible CPUs/GPUs and custom ASICs. FPGAs offer hardware designers a platform to rapidly prototype and deploy new ML accelerators, while attaining lower power consumption compared to CPU and GPU alternatives [52, 53]. Furthermore, these devices avoid the high costs associated with ASIC fabrication and testing.

4.2 FPGA Architecture

At the core of all Xilinx FPGAs are configurable logic blocks (CLBs), also referred to as logic array blocks (LABs) by Intel [54, 55]. As shown in Figure 4.1, these circuits are composed of lookup tables, flip-flops and multiplexers that can be configured to reproduce the behavior of

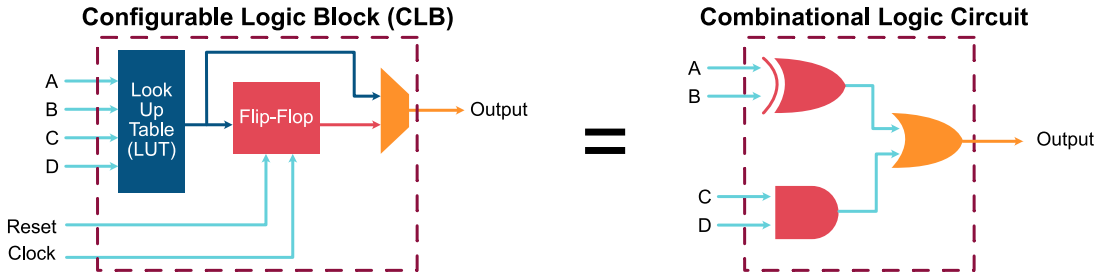

Figure 4.1: A simplified schematic for a configurable logic block (CLB) and its equivalent combinational logic circuit.

other digital logic circuits. The output values and interconnects between CLBs are managed by a configuration engine, which uses a set of binary instructions provided by a developer to configure the FPGA. These binary instructions are stored in the form of a bitstream file, which is shifted into the device using a serial interface such as JTAG. Bitstream files are generated based on synthesized netlists created using hardware description languages (HDL), such as Verilog or VHDL.

In addition to CLBs, FPGA packages are often designed to include a plethora of other components to improve their utility. Digital signal processors (DSPs), a staple component in modern FPGA architectures, provide complex arithmetic capabilities in a small area [55,56]. FPGA design tools often substitute logic that uses computationally complex arithmetic with DSPs that are able to perform the operations faster and with less energy than CLBs. Additionally, embedded RAMs offer dense on-chip volatile memory that is essential when a design needs to reference an array of values [55, 56]. Intel FPGAs also feature elements called memory logic array blocks (MLABs) that can use LUTs to form RAMs.

In recent years, FPGA vendors have sought to provide even more functionality to their FPGA devices. Xilinx's Zynq-7000 exemplifies this by featuring a programmable logic system coupled with an embedded ARM microcontroller [56]. The processing system inside of the Zynq-7000 XC7Z020 contains a dual core ARM Cortex-A9 multiprocessor with UART, I2C, USB and Ethernet controllers. These features enable hardware-software co-design where

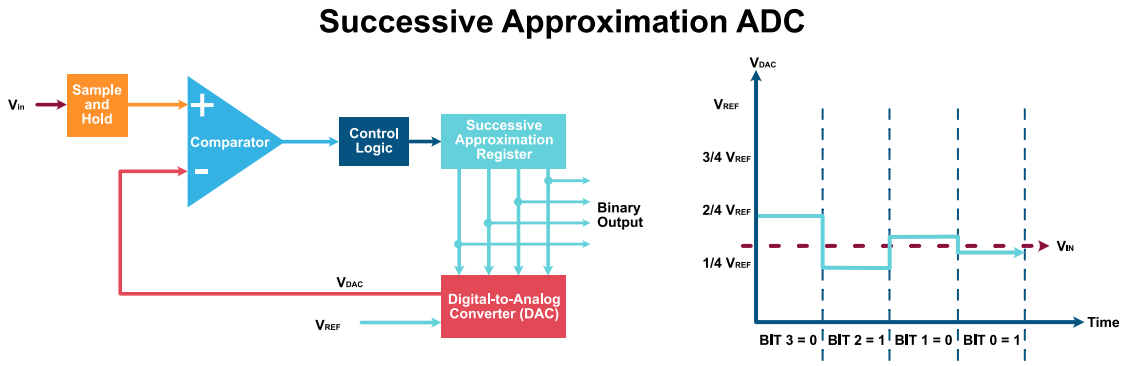

Figure 4.2: A successive approximation register (SAR) analog-to-digital converter (ADC).

designers can develop software that is very closely interfaced with custom programmable logic, allowing for a variety of different accelerator applications to be tested.

Analog mixed signal (AMS) design is also capable with modern FPGAs thanks to their inclusion of embedded analog-to-digital converters (ADCs). The Zynq-7000 package includes a 12-bit successive approximation register (SAR) ADC with a maximum sampling rate of 1 million samples per second (MSPS) [56, 57]. The SAR ADC works by using a DAC to first approximate the input analog voltage, and then providing the binary representation of the analog signal to the digital logic [58].

4.3 Hybrid FPGA-ASIC Machine Learning Systems

Research on hybrid integrated circuits that leverage both FPGA and ASIC technology in a combined ML system is limited, likely due to the effort it requires to merge the two mediums. [59] is one of the few works that accomplishes this by featuring an FPGA-ASIC system that performs real-time object detection with data-intensive 1080HD video at 60 frames/second. The impressive aspect of such a project is that their implementation consumes a mere 45.3mW of power, making it highly capable of being deployed into power-constrained unmanned aerial vehicles (UAVs). [60] is even more interesting as it merges a Stratix 10 FPGA

with multiple TensorRAM ASIC chiplets to improve both energy efficiency and latency of RNN workloads compared to GPU approaches. In this structure, the FPGA is used as a central controller and sends data to high-speed matrix-vector multiplication hardware within the chiplets. BrainScaleS, a wafer-scale neuromorphic computing system, interfaces 48 Xilinx Kintex-7 FPGAs with 384 analog neural network ASICs [61]. The FPGAs in BrainScaleS configure the system and allow for the spike data to be fed into the wafers, while the ASICs model the dynamics of continuous time spiking neurons.

The state-of-the-art systems in [59–61] demonstrate the potential of using hybrid FPGA-ASIC architectures to achieve low energy and low latency in ML tasks. In this work, an energy-efficient FPGA-ASIC hybrid DFR system is proposed which, along with [15, 27], is one of the few approaches that utilizes heterogeneous hardware platforms to accelerate RC applications.

Chapter 5

Hybrid FPGA-ASIC Delayed Feedback Reservoir

5.1 Proposed Design

In this work, a 16-bit, fixed-point DFR accelerator was developed [18]. The system is built using a Zynq-7000 XC7Z020 SoC, which interfaces with a custom 180nm CMOS chip. A programming interface is provided via the SoC's embedded dual-core ARM Cortex-A9 processor. The controller, reservoir, and readout layer for the DFR are instantiated in the SoC's programmable logic (PL) fabric. Data that enters the reservoir is sent to the analog ASIC, which models a Mackey-Glass (MG) activation function. Figure 5.1 illustrates the connections between the components of the introduced system.

5.1.1 FPGA Architecture

Inference on the testing samples is performed using the FPGA's PL, which contains a physical version of the DFR described in Chapter 2. The accelerator is interfaced with the embedded processor using an advanced extensible interface (AXI) interconnect that can be used to

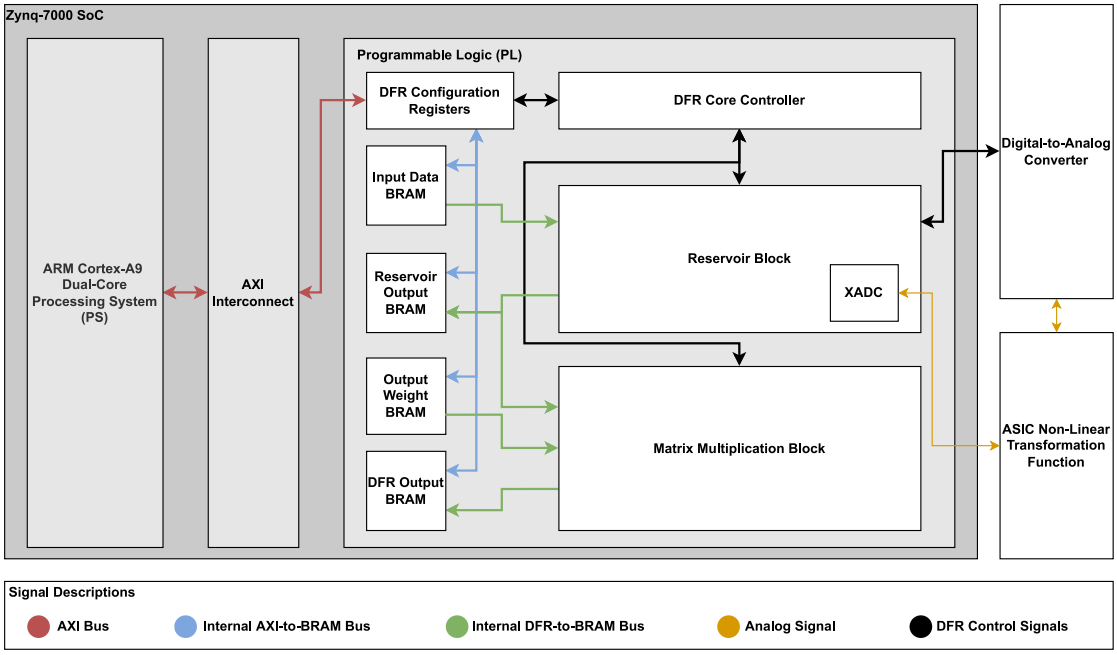

Figure 5.1: Overview of the hybrid delayed feedback reservoir (DFR) system architecture.

access the system's registers and internal memories. Ten configuration registers control the system, monitor its status, and specify the number of samples used for initialization and testing. Four internal dual-port memories are used to store the masked input samples, reservoir node values, output weights, and predicted outputs.

When launched, the system goes through three states: (1) reservoir initialization, (2) reservoir emulation, and (3) output evaluation. In the first two states, 16-bit input values are read into the reservoir from the input memory. To calculate the output of the single nonlinear neuron, each new input is added to the scaled output of the last virtual neuron and sent to the ASIC via an external 16-bit digital-to-analog converter (DAC) with a reference voltage of 2.5V. The DAC's output voltage is updated according to this sum, which causes a change in the ASIC's output voltage. The ASIC's output is read back into the FPGA using an embedded 12-bit analog-to-digital converter (ADC) with a reference voltage of 1V. The 12-bit ADC value is then stored in the first virtual neuron of the chain of nodes, and the cycle continues for subsequent inputs. A diagram illustrating the datapath of this operation

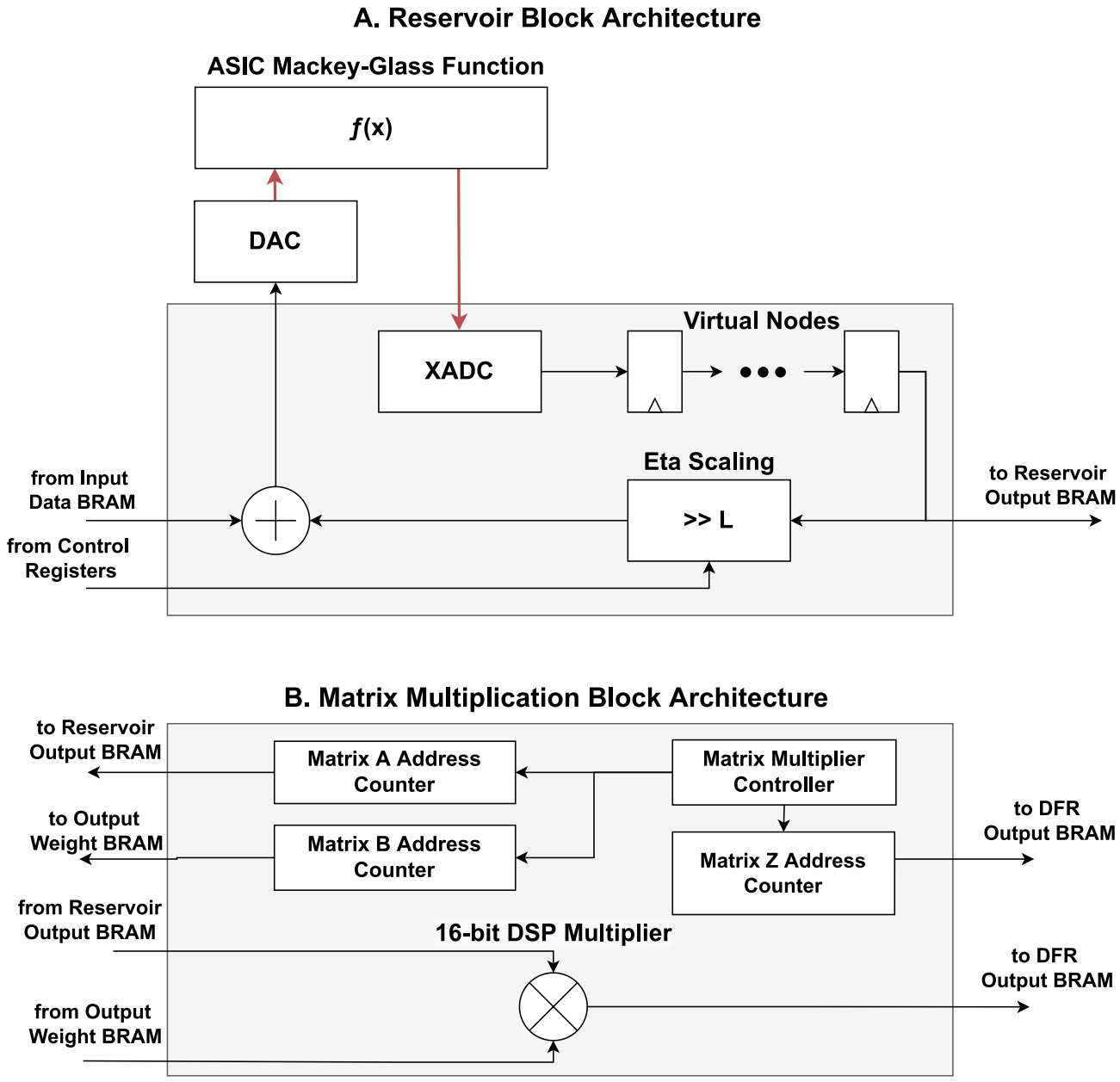

Figure 5.2: Hardware architecture for the (A) reservoir and (B) matrix multiplication blocks.

is shown in Figure 5.2A.

In the output evaluation state, each set of N virtual neuron states corresponding to the test inputs is multiplied by the output weight matrix to determine the output predictions. The matrix multiplication block is composed of several hardware counters to indicate the matrix data being read from the BRAM memories. A state-machine controller manages the values of the hardware counters to accurately perform the matrix multiplication function. Three DSP48E1 blocks are used to realize the 16-bit multiplication between the reservoir outputs and the output weights. This method of performing time-multiplexed operations is an optimization technique used in other works such as [28]. The matrix multiplication block's datapath is shown in Figure 5.2B.

Xilinx Vivado was used to synthesize the register transfer language (RTL) code and export the implemented bitstream file. The RTL was designed using SystemVerilog[1] and includes several generalized parameters to specify the DFR properties before synthesis (e.g., number of virtual nodes, input bit width). The logic utilization for the hardware implementation of the DFR accelerator is shown in Table 5.1. In this implementation, BRAMs are used over external DDR3 memories to simplify memory access operations and to improve read and write times. The bitstream was packaged with a PetaLinux image, which was used as the boot image for the Zynq SoC.

The reported dynamic power of the FPGA DFR hardware is 130 milliwatts when operating at a clock rate of 10MHz. With these settings, the system is able to process approximately 1625 samples per second. Table 5.2 below shows the power and latency measurements for the software DFR run on an AMD Ryzen 7 1700 CPU [62], the DFR from [17], and this work's DFR. The power dissipation of a single core of a the AMD Ryzen 7 CPU is approximated as 8.125 watts since the entire eight core processor has a power dissipation of 65 watts. The proposed DFR demonstrates a clear power improvement compared to the software model executed on the AMD processor. However, the latency decreases by 0.311 milliseconds. The

[1]SystemVerilog code for this project is available at https://github.com/oshears/hybrid_dfr_system.

Table 5.1: Zynq-7000 XC7Z020 Post-Implementation Logic Utilization

Logic Type	**Elements Used**	**Utilization Percentage**
Slices	1319	9.92%
Slice LUTs	2328	4.37%
Slice Registers	1934	1.82%
DSPs	3	1.36%
Block RAM Tiles	118	84.29%

Table 5.2: Comparison of DFR Implementations on FPGA and CPU

Implementation	**Metric**	**Measure**
Software DFR	Average Power	8.125 W (65 W)
Hamedani DFR [17]	Average Power	0.199 W
This Work	**Average Power**	**0.130 W**
Software DFR	Sample Latency	0.304 ms
Hamedani DFR [17]	Sample Latency	302 ms
This Work	**Sample Latency**	**0.615 ms**

latency of the mixed signal DFR can likely be improved by increasing the clock speed of the circuit to 20 MHz, causing the FPGA DFR to match the CPU DFR's latency. Compared to [17]'s implementation of the DFR, this system demonstrates a 34.7% power reduction, and a latency improvement of more than 90%.

5.1.2 ASIC Mackey-Glass Function

Activation functions allow neural networks to model non-linear data. Recent studies show that the Mackey-Glass (MG) function is the ideal activation function for DFR networks due to its natural nonlinear behavior and delay properties [14,15,23,30]. The nonlinear behavior of the MG function is modeled by the differential equation from [16], which can be depicted as:

$$\frac{dx}{dt} = \frac{ax(t-\tau)}{1+x(t-\tau)^{\xi}} - bx(t), \tag{5.1}$$

where a and b are scaling parameters, and ξ is a nonlinear exponent. Additionally, the MG function has a straightforward analog circuit implementation with an adjustable nonlinear exponent [14, 30]. [30, 63] further show that the natural delay property of the MG function makes it more capable of mapping temporal data over traditional sigmoid and hyperbolic tangent functions used for artificial neurons.

In this work, the 180nm CMOS analog circuit model of the MG function developed in [63] was used as a portion of the proposed mixed-signal DFR. Figure 5.3 demonstrates sample output voltage readings of the MG function collected from the fabricated ASIC prototype for all 2^{16} DAC voltage settings.

From Figure 5.3, it can be observed that the measured nonlinear characteristic resemble the ideal MG function from [14] with a scaling parameter and nonlinear exponent of $a = 1$ and $\xi = 16$, respectively. The measured power consumption of 24.55μW and silicon area of $372\mu\text{m}^2$ demonstrate the capability of realizing an efficient nonlinear transformation in silicon, potentially reducing computational resources and improving energy efficiency.

5.1.3 Software Training

To obtain the masked input data and readout layer weights for the FPGA-ASIC system, a pre-trained DFR model was developed in Python using the NumPy library[2]. As described

[2]Python code for this project is available at https://github.com/oshears/hybrid_dfr_system

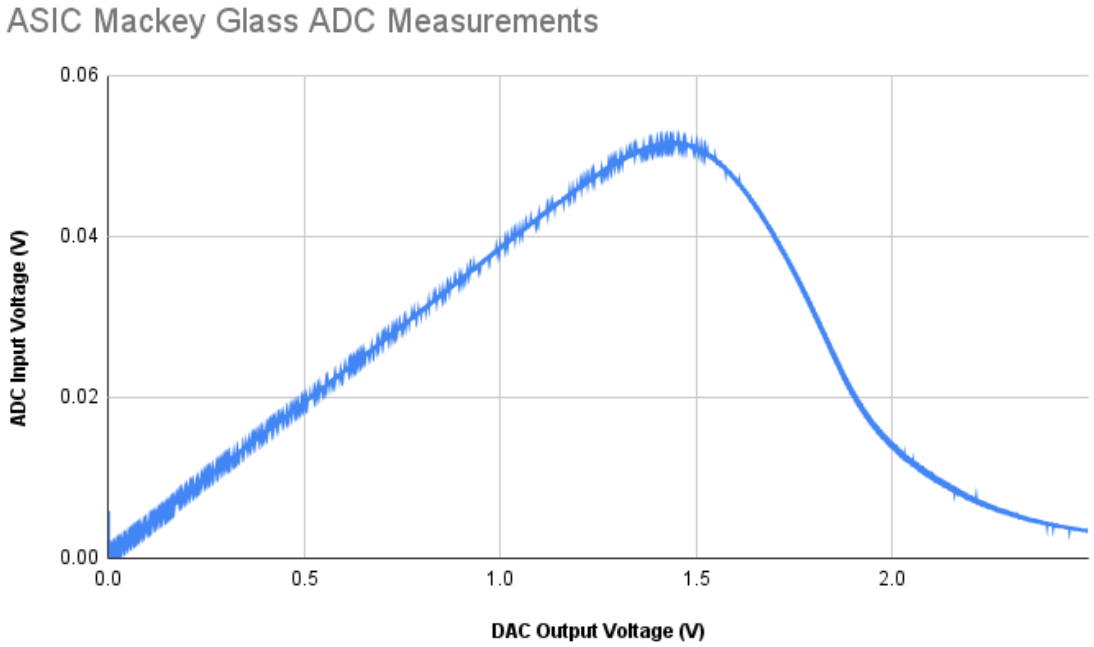

Figure 5.3: Measured Analog ASIC Mackey-Glass (MG) activation function.

in Chapter 2, all inputs from the dataset are masked by an $N \times 1$ matrix. Each element in the mask matrix is a random integer from the set $[0, 2^{16} - 1]$. Once the masked dataset is obtained, the reservoir is initialized with a specified number of samples dictated by the application. After the reservoir is initialized, the M training samples are then provided. For each group of N samples, the entire reservoir state (*i.e.*, the values of the N virtual neurons) are recorded to be used for training the output weights. Lastly, the output weights are obtained using Equation 2.3. The resulting masked input and output weights are loaded into the PL from the embedded processor via AXI. The hyperparameters of the model used in both the software and FPGA implementations are noted in Table 5.3. Note that the value for the feedback scale, η, was implemented as a tunable parameter based on the tested application.

Table 5.3: Delayed Feedback Reservoir Hyperparameters

Hyperparameter Name	**Symbol**	**Value**
Input Gain	γ	1
Feedback Scale	η	-
Feedback Delay	τ	100
Virtual Nodes	N	100
Regularization Constant	λ	10^8
Mask Matrix Range	$\mathbf{M}$	$[0, 2^{16} - 1]$

5.2 Experimental Results and Analysis

The hardware used in this combined FPGA-ASIC implementation is visualized in Figure 5.4. In this image, the Zynq-7000 SoC shown by ① is located on the FPGA development board (ZedBoard). The Zedboard's PMOD pins shown by ② are interfaced with an external DAC found on the ASIC's board. The ASIC's I/O interface on the board, shown by ③, receives data from a voltage divider circuit at ④ and sends the data back to the FPGA. Lastly, a serial console displaying programmable PetaLinux environment from the Zynq-7000 SoC is shown by ⑤.

To evaluate the predictive performance of the system, it was tested it against two applications: the NARMA10 benchmark, and a spectrum sensing task. For NARMA10, The inference accuracy of the system was measured using the normalized root mean squared error (NRMSE) equation:

$$NRMSE = \frac{||y_i - \hat{y}_i||}{||y_i||}, \tag{5.2}$$

This metric was chosen because it has been extensively employed for evaluating the accuracy

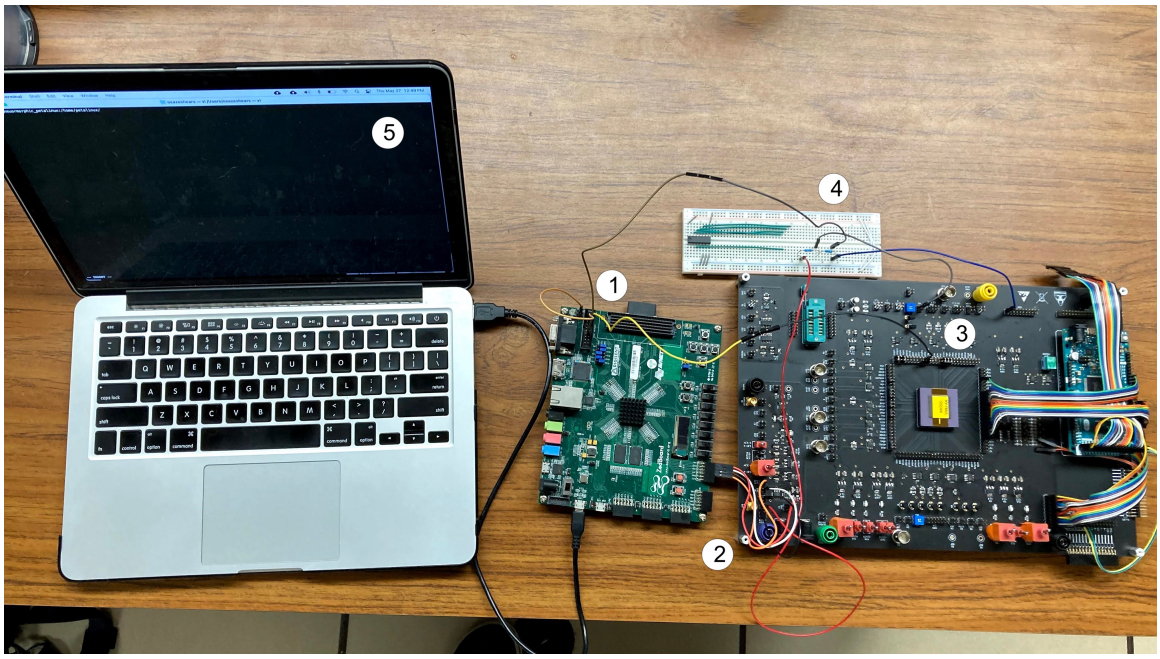

Figure 5.4: Hardware setup of the proposed FPGA-ASIC DFR system.

of DFR models [14, 30, 64]. Receiver operating characteristic (ROC) curves were used to evaluate the system's accuracy for spectrum sensing. Due to significant signal variations when reading the output of the ASIC-based MG function shown in Figure 5.3, the outputs of the activation function for each of the 2^{16} input voltage levels were recorded and used to evaluate the system's accuracy.

5.2.1 NARMA10 Benchmark

The tenth-order nonlinear autoregressive moving average (NARMA10) benchmark was first introduced as a method to evaluate RNN performance in [65]. This was primarily due to its ten step time dependency, which makes it harder for an RNN to learn. The inputs of the dataset are composed of uniformly distributed random numbers from the set [0, 0.5], while the outputs are determined by the equation:

Table 5.4: Model Accuracy for NARMA10

Model	γ	η	τ	θ	**N**	**Testing NRMSE**
[14]	0.05	0.5	80	0.2	400	0.15
[64]	0.05	0.75	40	0.2	200	0.17
This Work	1	0.5	100	1	100	0.21

$$y(k+1) = 0.3y(k-1) + 0.05y(k)[\sum_{i=0}^{9} y(k-i)] + 1.5u(k-9)u(k) + 0.1, \tag{5.3}$$

where $u(t)$ represents an input at timestep k. A total of 100 samples were used for reservoir initialization, 5900 for training, and 4000 for testing. The feedback scale, η was configured as 0.5 for this application. As shown in Table 5.4, the proposed DFR achieved an NRMSE of 0.21 for the test samples, demonstrating a difference of 0.06 from the best performing DFR on this benchmark from [14]. This difference in accuracy is due to the selected parameter configurations, which were modeled after the DFR in [30] and reduced hardware complexity. Here 100 virtual nodes are used with a separation factor θ of 1, compared to [14]'s 400 virtual nodes with a separation factor of 0.2. Furthermore, the input gain γ and feedback scale η are set as 1 and 0.5, respectively, as opposed to [14]'s optimal values of approximately 0.1 and 0.4. The predictive performance of the DFR in modeling this time series is visualized in Figure 5.5.

5.2.2 Spectrum Sensing Benchmark

In this experiment, three different MIMO antenna array configurations were tested, in addition to three signal-to-noise ratios (SNRs) for the additive white Gaussian noise. The feedback scale, η was configured as 0.0625 for this application. A total of 20 samples were used for reservoir initialization, 980 were used for training, and 5082 were used for testing.

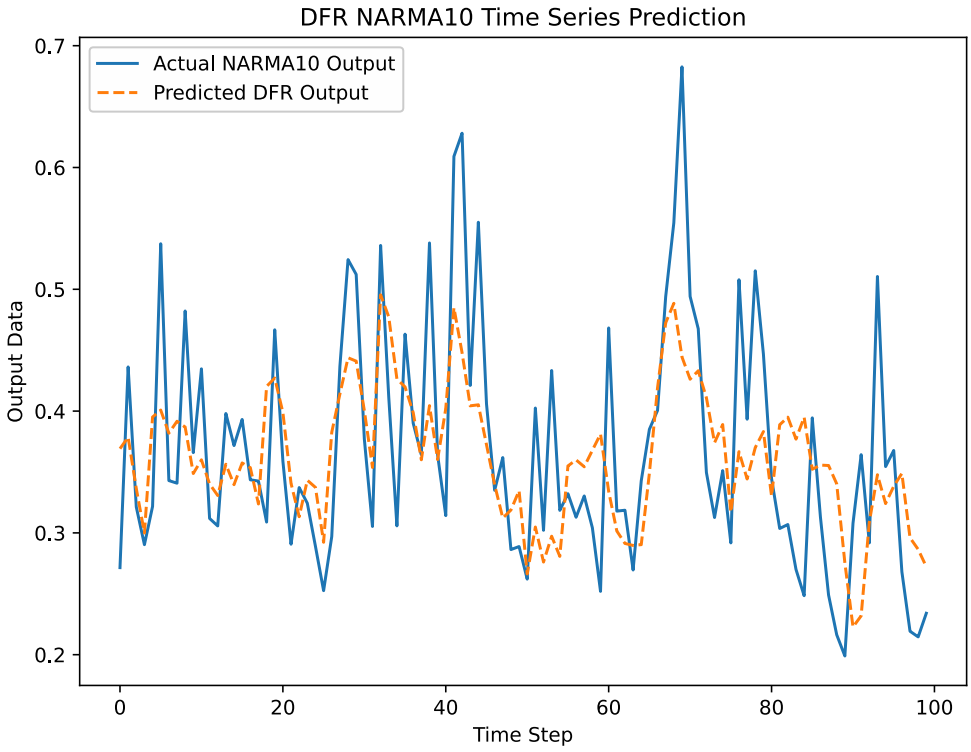

Figure 5.5: A graph showing the DFR's predictive performance on NARMA10.

Since spectrum sensing is a binary classification task requiring the model to determine if the spectrum is free or occupied, the accuracy was measured using receiver operating characteristic (ROC) curves. ROC curves show the relationship between the false positive rate and true positive rate for varying thresholds that are used in the model's decision making process. The false positive and true positive rates are calculated according to the Equations 5.4 and 5.5 below.

$$FPR = \frac{FP}{TN + FP} \tag{5.4}$$

$$TPR = \frac{TP}{TP + FN} \tag{5.5}$$

In these equations, for a given threshold, FP represents the number of false positives, FN represents the number of false negatives, TP represents the number of true positives, and TN represents the number of true negatives. When the area under the ROC curve (AUC) of a model is 1, it indicates that the model is able to accurately classify samples in the data set. Table 5.5 shows the accuracy of the hybrid DFR system in predicting the availability of the spectrum for the tested configurations. To better visualize the binary classification performance, the ROC curves are plotted in Figure 5.6.

As shown by both Table 5.5 and Figure 5.6, when given ideal conditions for performing the spectrum sensing task (*i.e.*, a noise level of -10db with 6 antennas), the system is able to achieve an AUC of 0.999. When compared with traditional energy detection based approaches, such as the square law combining (SLC) technique discussed in [17], the model performs better than SLC in low antenna configurations with high noise settings. With a SNR setting of -20dB, the SLC used in [17] achieves an AUC of 0.11 and 0.7 for the 2-antenna and 4-antenna settings, compared to the proposed model's performance of 0.6 and and 0.76. Additionally, the DFR developed in [17] achieves an AUC of 0.99 for the 6-antenna and -20dB SNR configuration, compared to the proposed model's AUC of 0.87. [14] uses a 2D heat map to help visualize the model's accuracy for various input weight and feedback weight

Table 5.5: Model Accuracy for Spectrum Sensing

Model	SNR	Antenna Count	AUC
Hamedani SLC [17]	-20dB	2	0.11
Hamedani SLC [17]	-20dB	4	0.7
Hamedani SLC [17]	**-20dB**	**6**	**0.97**
Hamedani DFR [17]	-20dB	2	0.76
Hamedani DFR [17]	-20dB	4	0.95
Hamedani DFR [17]	**-20dB**	**6**	**0.99**
This Work	-20dB	2	0.603
This Work	-20dB	4	0.764
This Work	**-20dB**	**6**	**0.871**

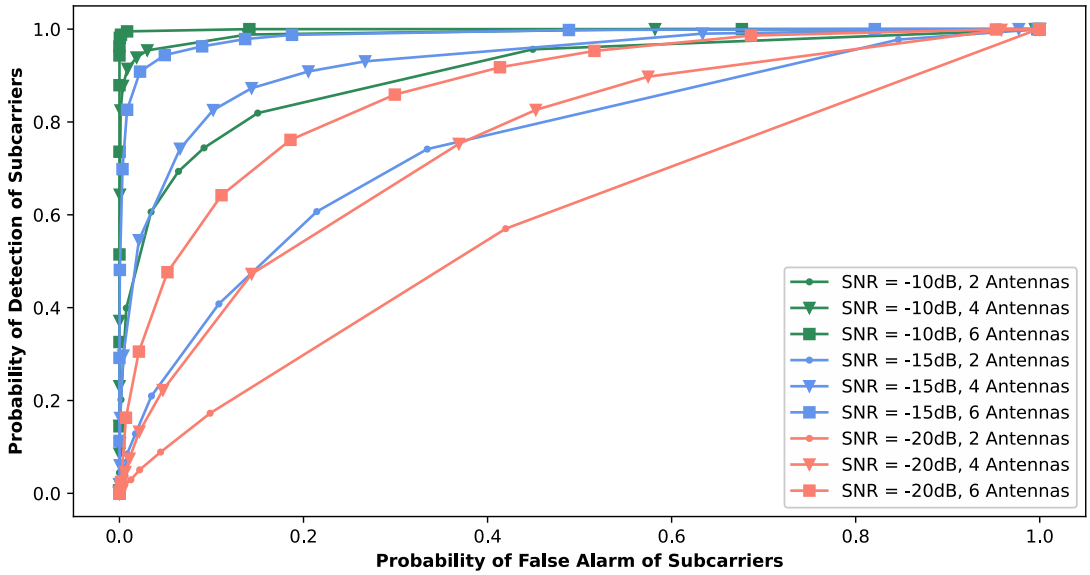

Figure 5.6: ROC curves for the spectrum sensing configurations.

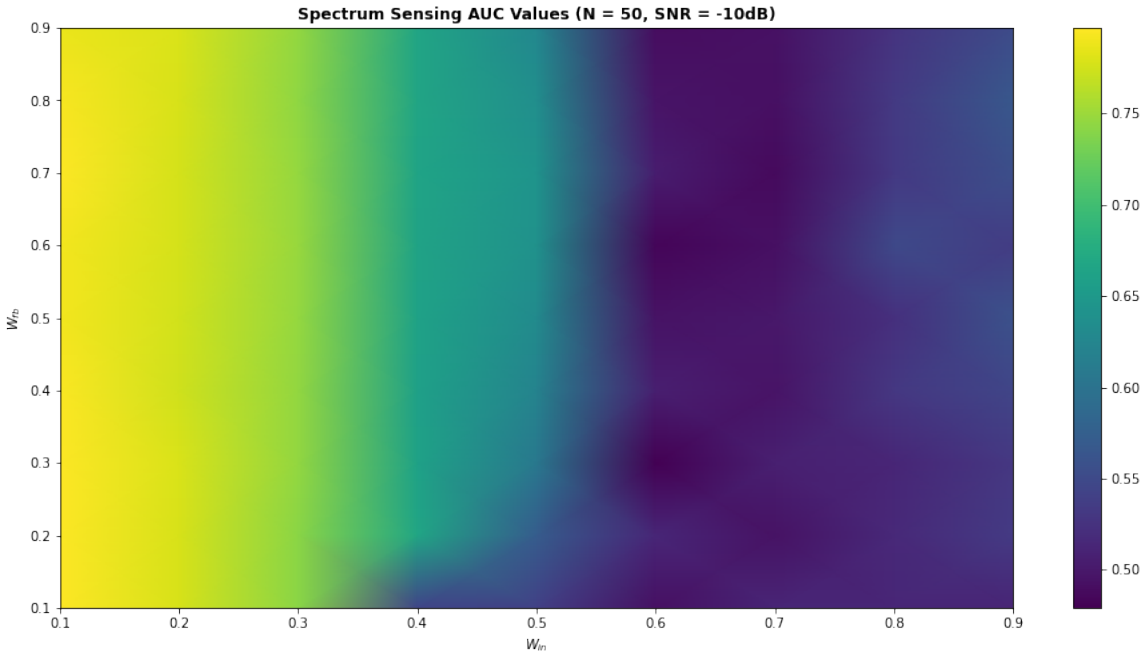

Figure 5.7: Effect of Input and Feedback Weight on Spectrum Sensing AUC.

configurations. Figure 5.7 visualizes the effect of tuning the input and feedback weight values on the model's accuracy with a configuration of one receiving antenna, 50 nodes and a -10dB SNR. For this scenario, the input weight is significantly tied to the AUC since low values for W_{in}, correspond to higher AUC measurements, while the feedback weight W_{fb} has little effect on the AUC.

5.2.3 System Limitations

While the proposed system demonstrates an effective way of realizing energy-efficient spectrum sensing in a small area, there are several limitations that hinder the applicability of the system. First, there is a large amount of noise that is experienced while reading the analog output from the Mackey-Glass ASIC circuit. This noise is visualized in Figure 5.3 which shows jittering in the output voltage, compared to a smooth, continuous curve. This results in large prediction accuracy variations since the output values that were used to train

the network in software are often misaligned with the output values used to evaluate the network. The XADC's averaging capabilities described in [57] can be leveraged to reduce this noise by averaging the input voltage of up to 256 samples. However, this would reduce the system's energy efficiency and inference speed.

Another limitation of the system is the ability to implement it on an actual RF device. Currently the design uses two separate IC packages: one containing the Zynq-7000 SoC, and another containing the Mackey-Glass circuit. Since the two implementations are in separate packages, the system can not be as easily demonstrated on a small device such as a cellphone. An ASIC implementation of the system that combines the two circuits into one package would solve this issue, but would also introduce more design and verification costs. Furthermore, this implementation makes it more challenging to test the circuit against real time wireless communications applications due to the lack of a dedicated RF interface.

The follow sections of this thesis propose an alternative approach to realizing the DFR on an low-profile edge device. The approach leverages a circuit board with built-in RF capabilities to stream data directly to the DFR for real-time spectrum sensing. The approach uses high-level synthesis design tools to further optimize the DFR and to provide increased design customization.

Chapter 6

High-Level Synthesis

6.1 High-Level Synthesis Overview

Traditionally, FPGA designs are created using a hardware description language (HDL) that can be synthesized into a gate-level representation, called a netlist, that can be implemented on the FPGA. Testbenches consist of non-synthesizable HDL code that can be used to simulate and verify the functionality of the designed hardware. However, the process of using HDL to create and verify an FPGA design can be inaccessible, time consuming, and prone to errors. To avoid these issues, a growing number of designers have begun to adopt a new approach known as high-level synthesis.

High-level synthesis (HLS) seeks to reduce the design and verification time of custom hardware by directly translating C and C++ code directly into an HDL representation [66]. This provides a more accessible approach to creating custom intellectual property (IP) cores by letting the designer focus on the algorithmic implementation while the HLS tool will create the hardware implementation. Figure 6.1 illustrates the idea behind this process. Furthermore, HLS streamlines the verification process by allowing a software testbench written in C++ to test both the software and hardware versions of the IP.

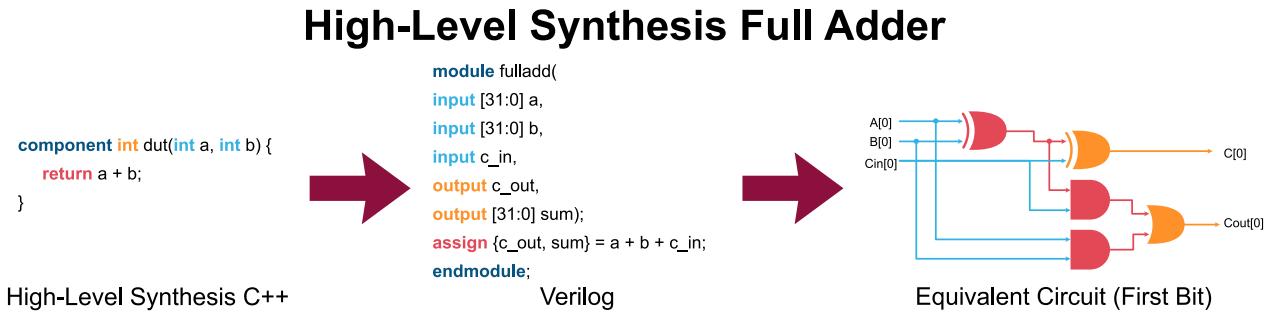

Figure 6.1: An example of C++ code for a full adder compiled using high-level synthesis to generate the equivalent Verilog code and circuit schematic.

Several recent works have demonstrated the capability of using HLS to model neural networks in hardware. [67] uses HLS to develop a graph convolutional network and evaluates the ability of this hardware to cluster data across graph data sets. [68] shows how HLS can be used to optimize the clock speed, throughput and scalability of hardware convolutional neural networks (CNNs). [69] proposes a custom HLS library for CNNs that extends the functionality of Xilinx Vivado HLS.

As demonstrated in these works, a number of optimization techniques are made available by HLS that can improve the area, latency, and power consumption of generated IP. These optimizations include arbitrary precision data types, loop pipelining, loop unrolling, and memory instantiation.

6.2 Arbitrary Precision Data Types

Arbitrary precision data types in HLS allow a designer to specify the number of bits used to represent a numeric value [70]. Typically in software programs, an integer is represented by 32 bits, however, HLS allows designers to have direct control over the number of bits used. This can potentially reduce the area of a design since smaller registers and fewer components will be required. When considering more sophisticated applications such as

machine learning, care should be taken when modifying the number of bits used since a model's accuracy is often directly proportional to the number of bits used. The reduced double exampled overviewed in [71] summarizes the benefits of using arbitrary precision floating point data types. In this example, a 30% reduction in area is observed by using a 56-bit floating point representation, in addition to a 15% reduction in latency.

6.3 Loop Unrolling

Loop unrolling is the process of taking a software loop and executing each of the loop iterations in parallel [70]. For example, a loop that modifies five samples of independent data over the course of five clock cycles can modify all of the data in a single clock cycle if unrolled five times. This idea is visualized in Figure 6.2. It should be noted that while loop unrolling may significantly increase computational performance through parallelism, it can increase the amount of logic used by a factor of the unrolling amount. This behavior may not be ideal when designing IP for devices with high area constraints.

The Intel HLS guide explores the effect of unrolling in the resource sharing filter example [71]. Here, a finite-impulse response (FIR) filter is implemented with and without the unroll directive. Data from simulations of these two versions of the filter show that the rolled (resource-shared) version of the loop has an average latency of 801 cycles, compared to the unrolled version's latency of 719 cycles. This produces a speed up of 11%. However, the unrolled FIR filter uses 20 times as many ALUTs and 30 times as many DSPs to achieve this speedup.

6.4 Loop Pipelining

Loop pipelining takes a different approach to optimizing loop performance [70]. Pipelining is a well studied technique that has been used to overlap instruction execution in general

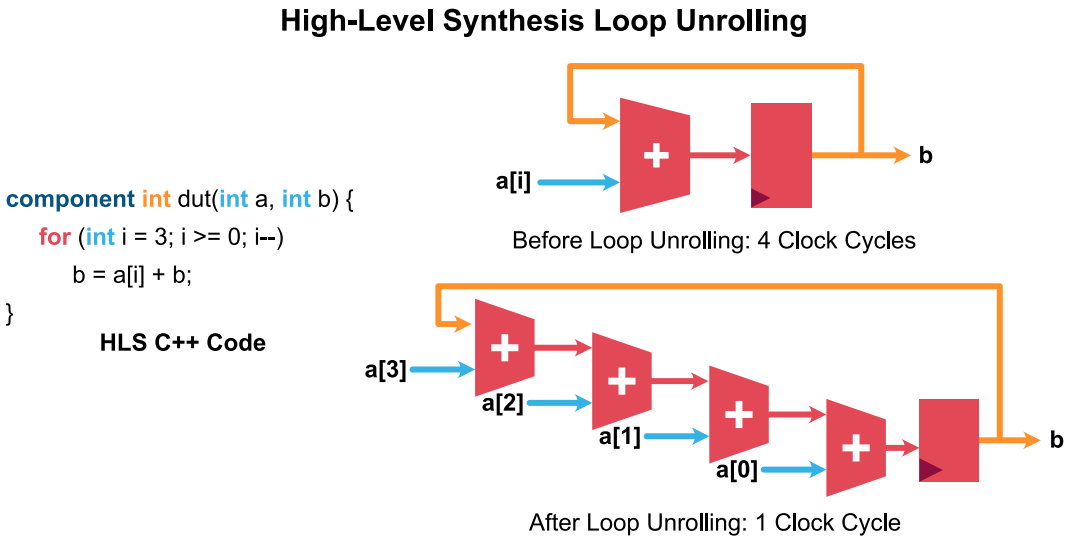

Figure 6.2: An example of unrolling a loop in C++ using Intel HLS Compiler.

purpose CPU architectures. This results in a reduction in execution time and usually an increase in clock frequency. Instead of duplicating the loop's logic as shown with unrolling, pipelining allows different stages of the loop to be broken down into overlapping sub-steps. Consider a program that reads data from an array, multiplies the data, and stores the result in a new array. Using loop pipelining optimizations would allow a second value to be read from the array, while the first value is being multiplied. By default Intel HLS compiler attempts to pipeline the operations in loops. The amount of time required to start a new loop in the pipeline is referred to as the initiation interval (II) [72].

6.5 Memory Instantiation

HLS also supports the ability to instantiate memory interfaces and components [70]. When a component is configured with a memory address pointer as an argument, the HLS tool translates this into a memory interface on the generated IP. During logic verification, the IP will send memory read and write requests through this generated interface. Memory can

also be created directly inside of a HLS IP by declaring an array. The static identifier allows the memory to retain its contents each time the component function is invoked.

Chapter 7

Delayed Feedback Reservoir on a Software Defined Radio

7.1 High-Level Synthesis DFR Core

To demonstrate the capabilities of the delayed feedback reservoir (DFR) for cognitive radio applications such as spectrum sensing, this effort aimed to train a DFR network and synthesize it in the hardware of a software-defined radio. Similar to the experiments from Chapter 5, the DFR model was first constructed in Python[1] to train it to accurately predict spectrum occupancy according to the fabricated data set. As discussed in Chapter 3, spectrum occupancy measurements were obtained from RWTH Aachen University's spectrum occupancy data set. The portion of this data set that was referenced contains 6102 orthogonal frequency-division multiplexing (OFDM) frames across 40 subcarriers. Whether data was transmitted over a given frame is indicated by a 1 or a 0 in the data set. Based on the spectrum occupancy behavior for one of these 40 subcarriers, 6102 random quadrature phase-shift keying (QPSK) symbols were generated. Each symbol corresponding to a point

[1]Python, HDL, and C++ code for this project is available at https://github.com/oshears/bladerf_dfr_accelerator.

Table 7.1: Delayed Feedback Reservoir Parameters

Parameter	**Value**
Mask Range	[-0.5,0.5]
Reservoir Nodes	50
Input Gain γ	0.5
Feedback Scale η	0.4
Floating Point Exponent Precision	8 bits
Floating Point Mantissa Precision	17 bits
Input Data Resolution	16 bits

in the sequence of OFDM frames where the spectrum was not busy was cleared from the frame. Once the array of symbols were generated, random additive white Gaussian noise (AWGN) was added to each symbol. The in-phase and quadrature (I/Q) components of each symbol were then translated into signed 16-bit binary numbers to represent the output of the SDR's analog-to-digital converters (ADC).

The parameters of the DFR implemented in Python are shown in Table 7.1. At each frame the DFR was presented with the generated I/Q components. The energy of a given frame was calculated using Equation 3.1. This energy measurement was then fed into the input layer of the DFR where it was masked and sent to the single non-linear node. Once the outputs of all training samples were calculated, ridge regression was used to adjust the DFR's output weights. The optimal configurations for the mask range, number of reservoir nodes, input gain, and feedback scale were determined after simulating several variations of the DFR and evaluating its performance for this task.

After verifying the DFR in Python, a C++ model of the network was developed using the Intel HLS Compiler libraries [73]. The floating point precision of the DFR implemented in C++ is shown in Table 7.1. The precision was chosen to minimize the number of resources required by the DFR hardware implementation, while sustaining accuracy. Once the C++

Table 7.2: Delayed Feedback Reservoir Logic Utilization

Logic Element	**Utilization**	**Utilization Percentage**
Adaptive Logic Modules (ALMs)	9,149	50%
Block Memory Bits	118,212	58%
Digital Signal Processors (DSPs)	48	73%

model was verified, the HLS tool was used to create Verilog hardware description language (HDL) code corresponding to the software model. Pseudocode for the HLS C++ implementation DFR is provided in Appendix A of this document. The logic utilization of the DFR IP core for the Cyclone V 5CEBA4F23C8 FPGA, shown in Table 7.2, was obtained after synthesizing the core in Quartus Prime.

Intel HLS Compiler provides further information about the adaptive look-up table (ALUT) utilization for each line of the C++ code. Each line of code was categorized into the relevant DFR functions. Figure 7.1 illustrates the ALUT usage for each of these DFR functions. From this figure, it is shown that the Mackey-Glass activation function uses more than half of the ALUTs in the system. The maximum clock rate and latency measurements were additionally obtained from the Intel HLS Compiler report. The system can operate at a maximum clock rate of 194.40 MHz and has a latency of 374 cycles. This corresponds to an inference time of $1.92\mu s$, or approximately 520 thousand samples per second. The latency of the circuit can be reduced to by using additional HLS parallelism techniques in the C++ algorithm, which are explored in the following sections.

The power estimates were obtained using Intel Quartus' Power Analyzer Tool. With an I/O toggle rate of 12.5%, the system has an average total power dissipation of 2.23W. The power estimates generated from this tool indicate that the power consumption of the DFR circuit falls within the same power consumption range of a cell phone that is actively transmitting and receiving data (on the order of 1 watt to 6 watts). Table 7.3 compares the power consumption and latency of the HLS DFR to the previous examples explored in

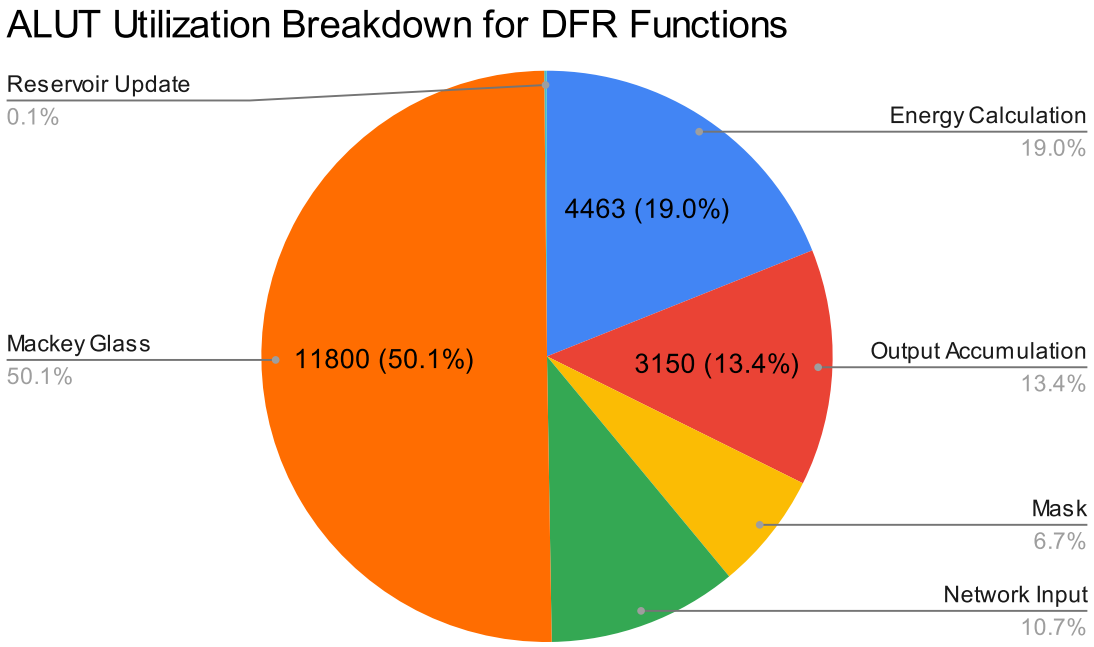

Figure 7.1: HLS ALUT Utilization for DFR Functions.

Table 5.2 of Chapter 5. As shown, the HLS implementation uses significantly more power than other hardware implementations, however, the latency is also greatly reduced. The power consumption of this circuit can likely be reduced if it is optimized to use fixed point arithmetic and a lookup table (LUT) activation function.

The last step in the development of this DFR IP core was verifying the hardware's functionality in a simulation and in real-time on the software radio. The simulation DFR model was verified using the test bench generated by the Intel HLS tool. To test the circuit in real-time, the IP was integrated into the bladeRF 2.0's FPGA architecture, then implemented and programmed into the software radio's FPGA.

Table 7.3: Comparison of DFR Implementations on FPGA and CPU with HLS Results

Implementation	**Metric**	**Measure**
Software DFR	Average Power	8.125 W (65 W)
Hamedani DFR [17]	Average Power	0.199 W
Mixed-Signal DFR	Average Power	0.130 W
HLS DFR	**Average Power**	**2.23 W**
Software DFR	Sample Latency	0.304 ms
Hamedani DFR [17]	Sample Latency	302 ms
Mixed-Signal DFR	Sample Latency	0.615 ms
HLS DFR	**Sample Latency**	**0.00192 ms**

7.2 BladeRF Software-Defined Radio

7.2.1 BladeRF Board Design

The bladeRF 2.0 micro, shown in Figure 7.2, is a mid-tier software defined radio developed by Nuand. The device is capable of transmitting and receiving RF data over two transmit and two receive antennas.

At the heart of this device is an Intel Altera Cyclone V FPGA, indicated by ① in Figure 7.2. The FPGA acts as a bridge between AD9361 RFIC which receives and transmits RF data, and the host computer which performs digital signal processing on the in-phase and quadrature (I/Q) samples. The FPGA enables further configurability of this system by allowing a designer to implement custom logic in the FPGA architecture. Thus, a designer can create custom hardware units to perform filtering, modulation and cognitive radio tasks directly in the hardware. This is advantageous to designers because hardware implementations of these algorithms are likely to run faster than software implementations, in addition to consuming

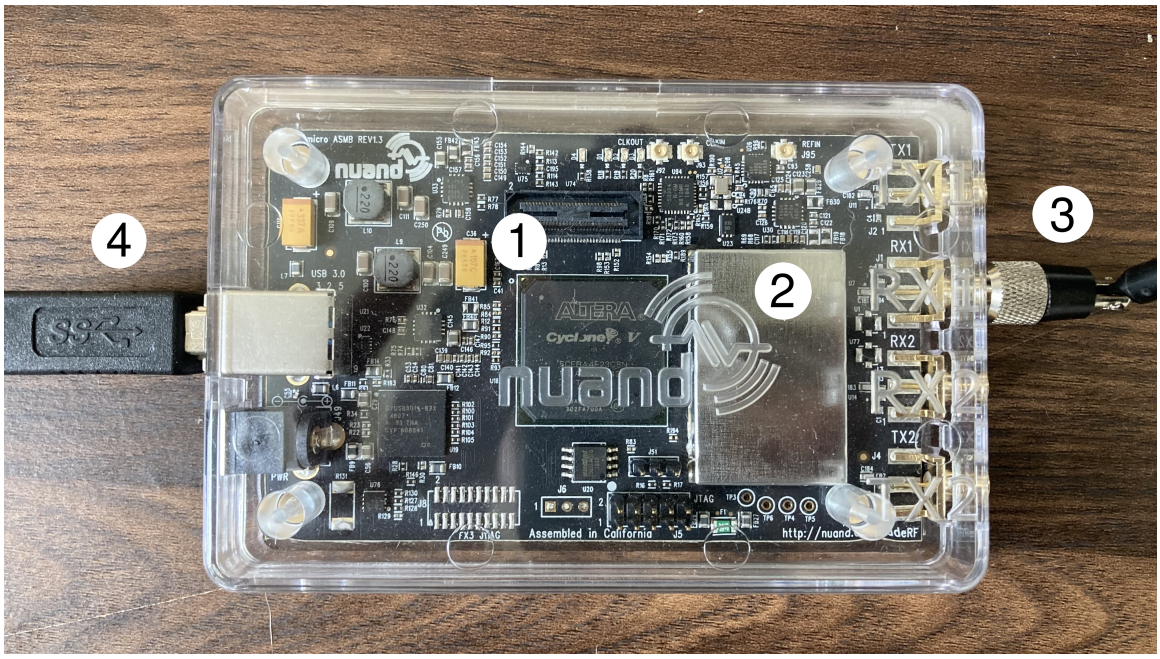

Figure 7.2: bladeRF 2.0 Micro Software-Defined Radio

less power.

The Cypress FX3 USB 3.0 peripheral controller manages transactions between the SDR and the host computer at the USB 3.0 interface shown by ④ in Figure 7.2. The host sends packetized data requests to the SDR to configure the frequency, voltage controlled oscillator (VCO), and gain settings. These data packets are translated from the USB protocol to UART and then interpreted by the FPGA which performs the hardware updates. The controller also reads the stream of I/Q sample data over the general programmable interface (GPIF) bus, and sends these to the host computer via the USB interface.

The last major component is the Analog Devices AD9361 RF transceiver, indicated by ② in Figure 7.2. The transceiver is interfaced with four SMA antenna interfaces, shown by ③, and supports multiple-input, multiple-output antenna configurations. The device is configured by the FPGA using the SPI interface which connects directly to its internal registers. The TX and RX interfaces carry the IQ samples between the transceiver and the FPGA. The SDR attributes, acquired from Nuand's bladeRF web page [74], are summarized in Table

Table 7.4: bladeRF 2.0 micro SDR Attributes

Attribute	**Capability**
Frequency Range	47 MHz - 6 GHz
Sampling Rate	61.44 MHz
RX/TX Antennas	2/2
ADC/DAC Resolution	12 Bits

7.4.

The challenge in developing custom hardware to run on the programmable logic of this device is the difficulty in learning and adapting custom hardware intellectual property (IP) to the bladeRF's FPGA architecture. Nuand has provided documentation to explain the software and hardware features of their device [75]. However, designers must still spend time dissecting the extensive list of VHDL and Verilog source code that is used to program the FPGA in order to adequately integrate their own IP. The following sections will breakdown the bladeRF's FPGA architecture to provide readers with an understanding of the functional units in this SDR system.

7.2.2 BladeRF FPGA Architecture

The bladeRF's bridge logic between the AD9361 RFIC and the host computer is implemented on the Intel Altera Cyclone V FPGA (5CEBA4F23C8). In this architecture, UART packets are received from the external FX3 USB controller and processed by a Nios II soft processor system. If the packets contain configuration data, the soft processor translates these packets into configuration commands that get sent to the external power monitor, local oscillator, or RFIC. The processor system is able to translate this data using several internal IP cores that create I2C and SPI transactions. Otherwise if the packets contain data requests, the processor instructs its internal AD9361 interface controller to read data from the RFIC and

Table 7.5: bladeRF 2.0 micro A4 FPGA Logic Utilization

Logic Element	**Utilization**	**Utilization Percentage**
Adaptive Logic Modules (ALMs)	6,280	34%
Package Pins	173	77%
Block Memory Bits	1,824,256	58%
Digital Signal Processors (DSPs)	8	12%
Phase Locked Loops (PLLs)	3	75%

send it to the receive first-in first-out queue (RX FIFO). The RX FIFO acts as a buffer that synchronizes the rate at which the host is requesting RX samples to the rate that the RFIC is making them available. During its transition through the RX FIFO, the signed 12-bit in-phase and quadrature samples from the RFIC ADCs are sign-extended to 16 bits and concatenated to make 32-bit samples. These samples are sent to the host computer using the FX3 GPIF bridge module. Table 7.5 shows the logic utilization of the unmodified FPGA architecture.

Figure 7.3 illustrates the FPGA architecture observed in the top level bladeRF HDL file, along with the proposed modifications to embed the DFR core and test memory. The bladeRF Python package, documented in [75], is used to read the received samples from the bladeRF device. In addition to reading and writing data to the device, the package can also be used to program the FPGA with a custom bitstream file (.rbf).

7.3 Experimental Results and Analysis

7.3.1 Spectrum Sensing Benchmark

The performance of the DFR in modeling the spectrum occupancy data was evaluated in a similar manner to that described in Chapter 5. Receiver operating characteristic (ROC)

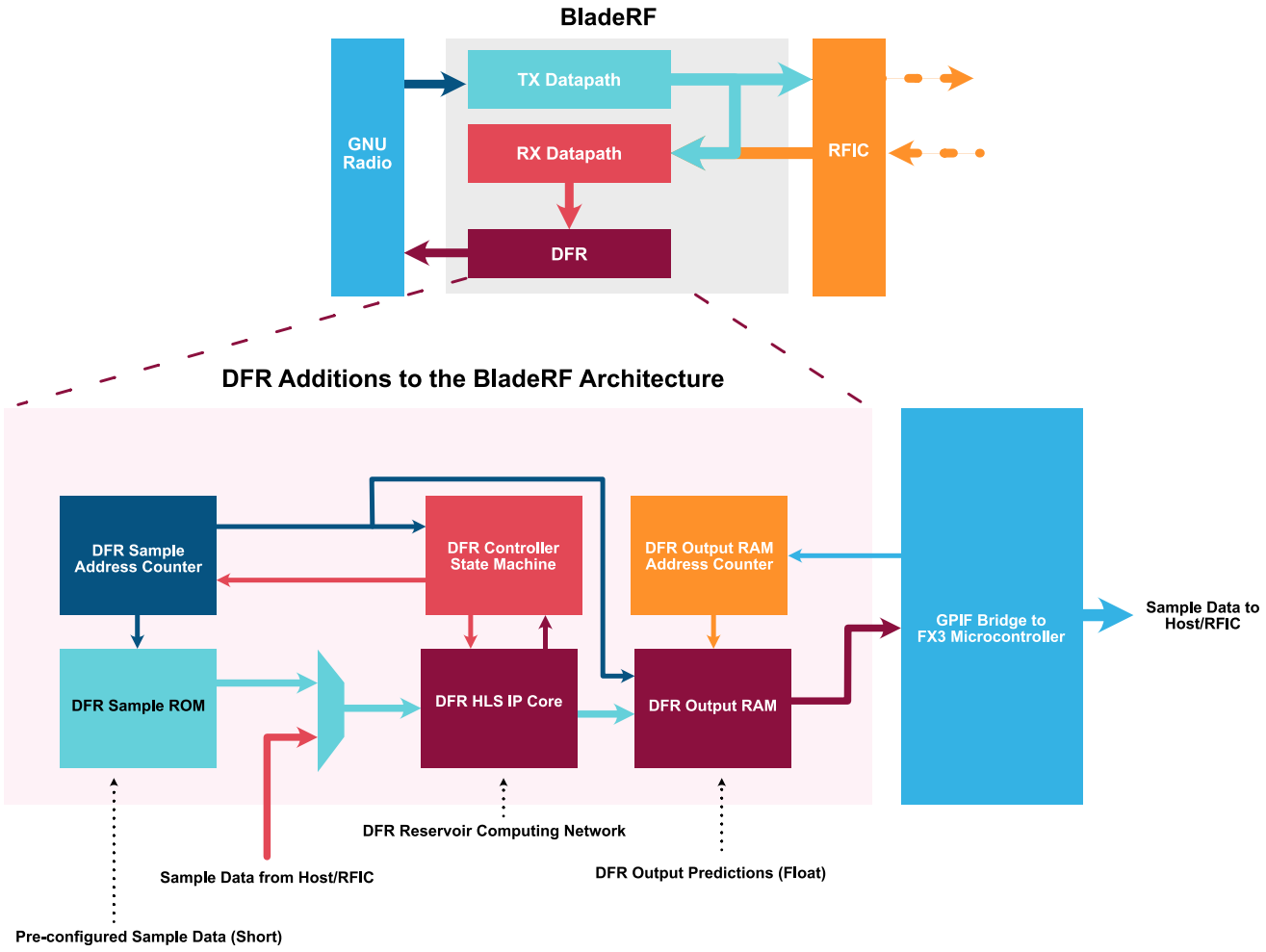

Figure 7.3: bladeRF FPGA Architecture with the HLS DFR Core.

Table 7.6: Spectrum Sensing HLS DFR Performance

SNR	**Area Under the Curve (AUC)**
-10dB	0.797
-15dB	0.737
-20dB	0.708
-30dB	0.688

curves were created to show the relationship between the false positive rates and true positive rates for varying thresholds in the model's decision making process. The false positive and true positive rates were calculated according to equations 5.4 and 5.5. Table 7.6 shows the model's area under the curve (AUC) measurements across four noise levels for a single RX antenna configuration.

To evaluate the hardware version of the spectrum sensing DFR core, the bladeRF's top level HDL file was modified to include the DFR IP core, in addition to a sample memory to test the core's real-time performance on the initial data set. Instructions for modifying the FPGA image are provided on the bladeRF's GitHub repository Wiki page [75]. All generated HDL and modifications to the bladeRF's FPGA image can be found in the GitHub repository for this project[1].

7.3.2 High-Level Synthesis Optimizations

A number of HLS optimizations were explored to observe their effect on the generated DFR system. First, the impact of the floating point precision on the logic element utilization and accuracy were evaluated. Figure 7.4 shows the relationship between precision and utilization for the HLS DFR. Beyond the 46-bit floating point precision, a 97% increase in the memory logic array block (MLAB) utilization is observed, followed by a 128% increase in DSP uti-

[1]Python, HDL, and C++ code for this project is available at https://github.com/oshears/bladerf_dfr_accelerator.

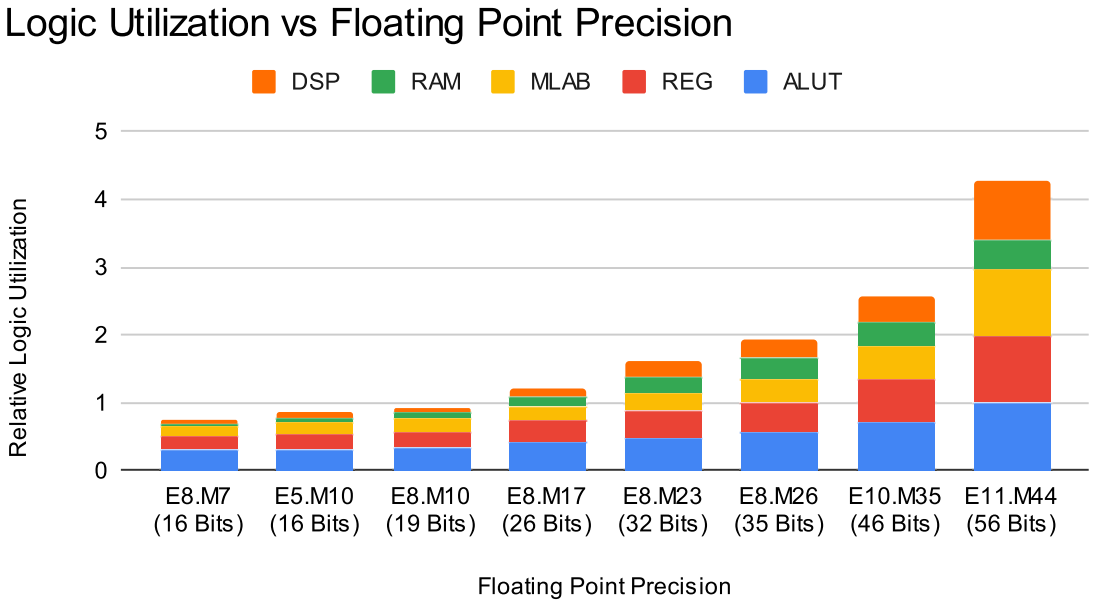

Figure 7.4: HLS Floating Point Precision Utilization.

lization. When considered with regard to the accuracy of the system, little improvement to the AUC was observed after 26 bits, and therefore, indicates that this is the optimal value for the area and accuracy trade-off. Alternatively the accuracy of the system decreases by 30% when the precision is reduced to 16 bits.

When the floating point precision increases from 48 bits to 56 bits, the DSP utilization increases by more than 2×. Figure 7.5 explores this increase by reporting the number of DSPs required for each floating point arithmetic operation. The multiplication and power operations largely contribute to this increase in DSP utilization. Intel's DSP blocks support multiplication of two 32-bit floating point numbers [76]. For 16-bit and 26-bit multiplications, only a single DSP is required. However, when the precision increases to 35 bits, more DSPs and ALUT logic must be introduced to perform the operation. A similar result is observed for the number of memory logic array blocks (MLABs).

The impact of loop unrolling on the system's latency and power was also analyzed. Figure 7.6 shows the trade-off between three levels of loop unrolling on the inference time and power

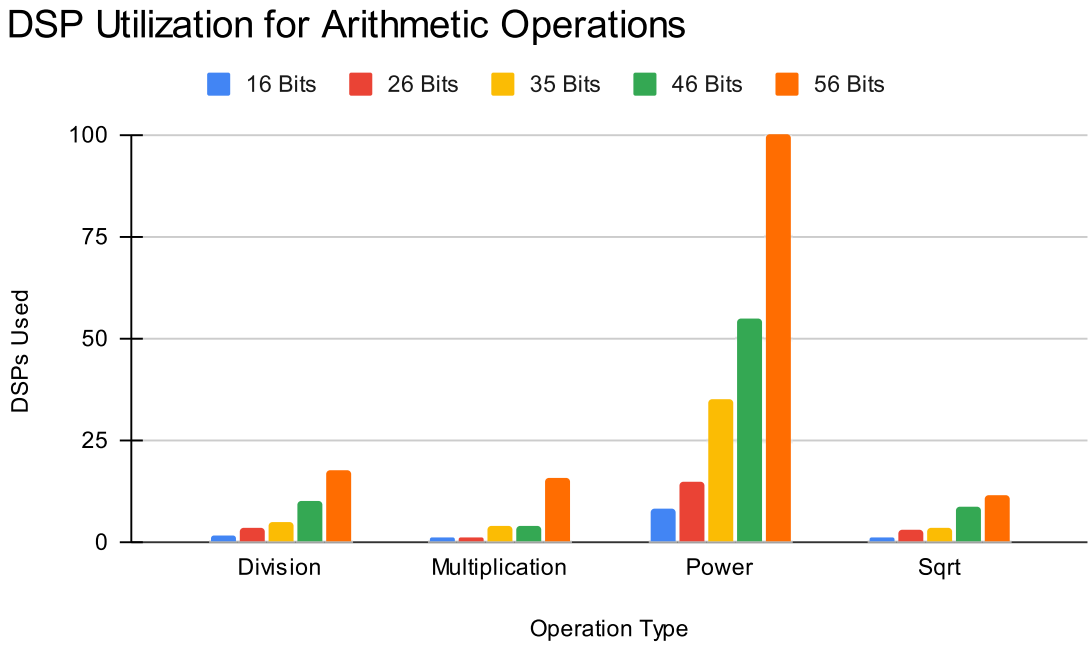

Figure 7.5: DSP Utilization for Floating Point Arithmetic Operations.

of the system. As shown, when the feedback loop of the system is unrolled 3 times compared to 1 time, the power consumption doubles while the inference time reduces only by 3%. This indicates that with the tested implementation, loop unrolling has little benefit for reducing latency.

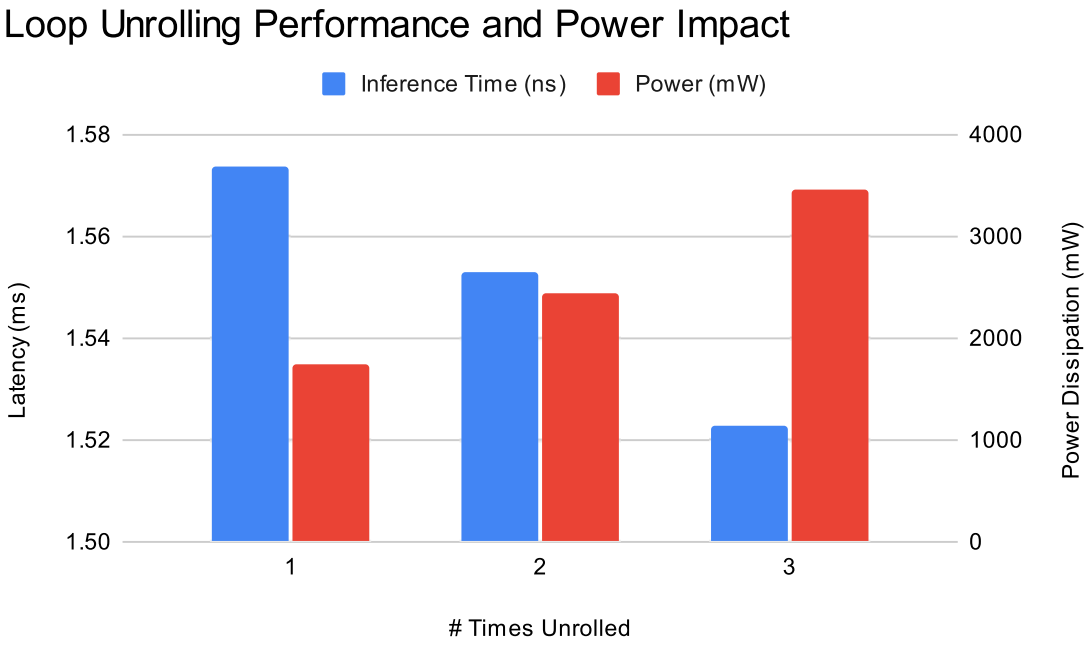

Figure 7.6: HLS Loop Unrolling Power and Latency Impact.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

This thesis hypothesized that FPGA implementations of delayed feedback reservoir circuits would achieve lower power consumption and latency compared to CPU-based approaches. The mixed-signal DFR has an average power of 0.130 watts and a latency of 0.615 milliseconds. The HLS DFR has an average power of 2.23 watts and a latency of 1.92 nanoseconds. Compared to the CPU estimates of 8.125 watts and 0.304 milliseconds, these approaches show that FPGA implementations of the DFR are faster and require less energy than software models. By performing this analysis, the plausibility of using these systems for edge device spectrum sensing is demonstrated. As noted, edge devices have a power dissipation on the order of 10 watts. Both circuits operate within this range, while still being able to accurately model the NARMA10 benchmark and spectrum occupancy datasets.

The first implementation leveraged digital and analog circuits to optimize the energy efficiency of the reservoir computing system. This mixed-signal DFR used a digital circuit to store the reservoir states and to perform the readout layer evaluation, and an analog circuit to serve as the non-linear transformation function. The major benefits of this approach

include its simplified hardware architecture thanks to the utilization of a low area Mackey-Glass function, and high energy efficiency because of fewer required FPGA logic elements. However, this mixed-signal approach has limitations that hinder its applicability. First, the output voltage variations observed from the analog activation function significantly affect the predictive performance of the system. This is likely caused by either the analog circuit attempting to settle on an output value, or by inaccurate voltages being generated by the digital-to-analog converter. The averaging feature of the analog-to-digital converter can be used to reduce these variations in the output voltage readings, but this will induce additional latency in the design and thus affect the inference time. Furthermore, the conversion time and energy required to translate values between the analog and digital domains affects the practicality of this system. Additionally, in its current state, the system cannot be effectively demonstrated in a real-time spectrum sensing application.

Alternatively, the high-level synthesis (HLS) DFR provides a practical way to implement and test the reservoir computing network against different applications. Accessible C++ code can be quickly developed to generate a component and to verify its functionality. Due to the HLS implementation being purely digital, it can be easily integrated in contemporary software-defined radio (SDR) architectures that use FPGAs, such as Nuand's bladeRF 2.0 and Ettus Research's USRP devices. Furthermore, it achieved the lowest sample latency for the spectrum sensing task. However, this HLS approach is limited by the large amount of logic elements required to implement the Mackey-Glass activation function on the FPGA. The high utilization of the activation function constrains the amount of additional logic that can be added to the SDR's FPGA architecture. Furthermore, this high utilization causes the power dissipation to be the highest among the other FPGA approaches. A lookup table (LUT) could be used to help combat this high utilization by replacing dynamically generated values with those stored in on-chip memories. Overall, the experiments conducted here provide a foundation for future implementations of reservoir computing systems in digital hardware for cognitive radio applications.

8.2 Future Work

8.2.1 Advanced Applications

Future work building off of this research could explore other real-time wireless communications, audio processing, and image processing applications. [29] demonstrates the ability of an echo state network on an FPGA to perform wireless symbol detection. [28] shows that similar ESNs can be optimized using on-the-fly binarized weight generation techniques, and applied to applications such as spoken digit recognition (TIDIGITS) and human activity recognition (HAR). [27] shows that DFRs can also be effectively applied to identify specific emitters and communication protocols used in RF settings. They also demonstrate the ability to train the DFR during runtime, often referred to as online learning, which is essential for high accuracy in real world applications. Lastly, [31] shows that single neuron systems inspired by the DFR have the potential to accurately classify images from the MNIST and CIFAR10 data sets.

8.2.2 Spiking DFR

The DFR's energy efficiency and area utilization could be further improved through the use of neuromorphic computing. Neuromorphic computing systems are composed of spiking neuron models, instead of artificial neuron models. As briefly described in Chapter 2, spiking neurons receive data in the form of spikes. Each spike corresponds to a weight specified by the synapse it travels across. As a neuron receives spikes, its membrane potential increments towards a threshold value. When the membrane potential exceeds this threshold value, the neuron emits a spike which gets transmitted to all other neurons connected to it.

SNNs are believed to have more potential than traditional ANNs in several aspects. The energy efficiency of an SNN can outperform that of an ANN depending on the technique used to encode the input data into spikes. The reason for this is because an input neuron

will not necessarily produce a spike at each timestep (data sparsity), reducing the amount of energy consumed. SNNs also do not inherently feature any large-scale matrix multiplication. Each neuron is only required to add the weight values of the synapses that had spikes to its current membrane potential, and to check if the potential exceeds a threshold value. Thus, SNN neurons can be realized with only adders and comparators, which combined have a lower area and power consumption than the multipliers and adders required by a traditional ANN hardware implementations [77]. SNNs may also be more capable of learning patterns in time series and real-time problems due to their temporal properties, and their biologically accurate representation may allow them to better accelerate neuroscience research.

Several works have shown the ability of the DFR to be used effectively with spiking neurons in analog hardware and in software [78–84]. Extensions of the work proposed in this thesis could explore digital hardware implementations of spiking DFRs and demonstrate the ability to perform online learning with techniques such as spike-timing dependent plasticity (STDP).

Bibliography

[1] "Autopilot," 2022. [Online]. Available: https://www.tesla.com/autopilot

[2] "Image classification on imagenet," 2022. [Online]. Available: https://paperswithcode.com/sota/image-classification-on-imagenet

[3] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, Dec 2015. [Online]. Available: https://doi.org/10.1007/s11263-015-0816-y

[4] A. Brock, S. De, S. L. Smith, and K. Simonyan, "High-performance large-scale image recognition without normalization," in *Proceedings of the 38th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 18–24 Jul 2021, pp. 1059–1071. [Online]. Available: https://proceedings.mlr.press/v139/brock21a.html

[5] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors," in *Proceedings of the 26th Annual International Conference on Machine Learning*, ser. ICML '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 873–880. [Online]. Available: https://doi.org/10.1145/1553374.1553486

[6] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "{TensorFlow}: A system for {Large-Scale} machine learn-

ing," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[7] "Google trends," 2022. [Online]. Available: https://trends.google.com/trends/

[8] "Aws and nvidia," 2022. [Online]. Available: https://aws.amazon.com/nvidia/

[9] X. Wang, Y. Han, V. C. M. Leung, D. Niyato, X. Yan, and X. Chen, "Convergence of edge computing and deep learning: A comprehensive survey," *IEEE Communications Surveys Tutorials*, vol. 22, no. 2, pp. 869–904, 2020.

[10] "Nvidia a100," 2022. [Online]. Available: https://www.nvidia.com/en-us/data-center/a100/

[11] "iphone 13 pro specs," 2022. [Online]. Available: https://www.apple.com/iphone-13-pro/specs/

[12] "The future is here: iphone x," 2022. [Online]. Available: https://www.apple.com/newsroom/2017/09/the-future-is-here-iphone-x/

[13] A. Frumusanu, "The apple a15 soc performance review: Faster & more efficient," *AnandTech*, 2021. [Online]. Available: https://www.anandtech.com/show/16983/the-apple-a15-soc-performance-review-faster-more-efficient/2

[14] 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*, vol. 2, no. 1, p. 468, Sep 2011. [Online]. Available: https://doi.org/10.1038/ncomms1476

[15] M. C. Soriano, S. Ortín, L. Keuninckx, L. Appeltant, J. Danckaert, L. Pesquera, and G. van der Sande, "Delay-based reservoir computing: Noise effects in a combined analog and digital implementation," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 26, no. 2, pp. 388–393, 2015.

[16] J. Li, K. Bai, L. Liu, and Y. 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)*, 2018, pp. 308–313.

[17] K. Hamedani, L. Liu, S. Liu, H. He, and Y. 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*, vol. 34, no. 02, pp. 1292–1299, Apr. 2020. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/5484

[18] O. Shears, K. Bai, L. Liu, and Y. Yi, "A hybrid fpga-asic delayed feedback reservoir system to enable spectrum sensing/sharing for low power iot devices iccad special session paper," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021, pp. 1–9.

[19] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 11 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[20] Y. Yu, X. Si, C. Hu, and J. Zhang, "A Review of Recurrent Neural Networks: LSTM Cells and Network Architectures," *Neural Computation*, vol. 31, no. 7, pp. 1235–1270, 07 2019. [Online]. Available: https://doi.org/10.1162/neco_a_01199

[21] A. Graves, N. Jaitly, and A.-r. Mohamed, "Hybrid speech recognition with deep bidirectional lstm," in *2013 IEEE Workshop on Automatic Speech Recognition and Understanding*, 2013, pp. 273–278.

[22] H. Jaeger, "Adaptive nonlinear system identification with echo state networks," in *Advances in Neural Information Processing Systems*, S. Becker, S. Thrun, and K. Obermayer, Eds., vol. 15. MIT Press, 2002. [Online]. Available: https://proceedings.neurips.cc/paper/2002/file/426f990b332ef8193a61cc90516c1245-Paper.pdf

[23] G. Tanaka, T. Yamane, J. B. Héroux, R. Nakane, N. Kanazawa, S. Takeda, H. Numata, D. Nakano, and A. Hirose, "Recent advances in physical reservoir computing: A review," *Neural Networks*, vol. 115, pp. 100–123, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0893608019300784

[24] M. Lukoševičius, *A Practical Guide to Applying Echo State Networks*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 659–686. [Online]. Available: https://doi.org/10.1007/978-3-642-35289-8_36

[25] M. Lukoševičius and H. Jaeger, "Reservoir computing approaches to recurrent neural network training," *Computer Science Review*, vol. 3, no. 3, pp. 127–149, 2009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574013709000173

[26] M. Bouvier, A. Valentian, T. Mesquida, F. Rummens, M. Reyboz, E. Vianello, and E. Beigne, "Spiking neural networks hardware implementations and challenges: A survey," *J. Emerg. Technol. Comput. Syst.*, vol. 15, no. 2, apr 2019. [Online]. Available: https://doi.org/10.1145/3304103

[27] S. Kokalj-Filipovic, P. Toliver, W. Johnson, and R. Miller, "Reservoir based edge training on rf data to deliver intelligent and efficient iot spectrum sensors," in *2021 IEEE International Conference on Smart Computing (SMARTCOMP)*, 2021, pp. 17–24.

[28] S. Gupta, S. Chakraborty, and C. S. Thakur, "Neuromorphic time-multiplexed reservoir computing with on-the-fly weight generation for edge devices," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–10, 2021.

[29] V. M. Gan, Y. Liang, L. Li, L. Liu, and Y. Yi, "A cost-efficient digital esn architecture on fpga for ofdm symbol detection," *J. Emerg. Technol. Comput. Syst.*, vol. 17, no. 4, jun 2021. [Online]. Available: https://doi.org/10.1145/3440017

[30] K. Bai and Y. Yi, "Dfr: An energy-efficient analog delay feedback reservoir computing system for brain-inspired computing," *J. Emerg. Technol. Comput. Syst.*, vol. 14, no. 4, dec 2018. [Online]. Available: https://doi.org/10.1145/3264659

[31] F. Stelzer, A. Röhm, R. Vicente, I. Fischer, and S. Yanchuk, "Deep neural networks using a single neuron: folded-in-time architecture using feedback-modulated delay loops," *Nature Communications*, vol. 12, no. 1, p. 5164, Aug 2021. [Online]. Available: https://doi.org/10.1038/s41467-021-25427-4

[32] Cisco Systems Inc., "Cisco annual internet report (2018–2023) white paper," Cisco Systems, Inc., San Jose, CA, USA, Tech. Rep. 1, March 2020. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html

[33] N. Hassan, K.-L. A. Yau, and C. Wu, "Edge computing in 5g: A review," *IEEE Access*, vol. 7, pp. 127 276–127 289, 2019.

[34] M. N. Tehrani, M. Uysal, and H. Yanikomeroglu, "Device-to-device communication in 5g cellular networks: challenges, solutions, and future directions," *IEEE Communications Magazine*, vol. 52, no. 5, pp. 86–92, 2014.

[35] R. Atat, L. Liu, H. Chen, J. Wu, H. Li, and Y. Yi, "Enabling cyber-physical communication in 5g cellular networks: Challenges, spatial spectrum sensing, and cyber-security," *IET Cyber-Physical Systems: Theory & Applications*, vol. 2, no. 1, pp. 49–54, 2017.

[36] H. Chen and L. Liu, "Resource allocation for sensing-based device-to-device (d2d) networks," in *2015 49th Asilomar Conference on Signals, Systems and Computers*, 2015, pp. 1058–1062.

[37] X. Lin, J. G. Andrews, and A. Ghosh, "Spectrum sharing for device-to-device communication in cellular networks," *IEEE Transactions on Wireless Communications*, vol. 13, no. 12, pp. 6727–6740, 2014.

[38] J. Jagannath, N. Polosky, A. Jagannath, F. Restuccia, and T. Melodia, "Machine learning for wireless communications in the internet of things: A comprehensive survey," *Ad Hoc Networks*, vol. 93, p. 101913, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1570870519300812

[39] M. E. Morocho-Cayamcela, H. Lee, and W. Lim, "Machine learning for 5g/b5g mobile and wireless communications: Potential, limitations, and future directions," *IEEE Access*, vol. 7, pp. 137 184–137 206, 2019.

[40] K. Cohen, *Machine Learning for Spectrum Access and Sharing*. John Wiley & Sons, Ltd, 2020, ch. 1, pp. 1–25. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119562306.ch1

[41] "Rwth aachen university static spectrum occupancy measurement campaign," 2009. [Online]. Available: https://download.mobnets.rwth-aachen.de/

[42] P. Smith, A. Luong, S. Sarkar, H. Singh, A. Singh, N. Patwari, S. K. Kasera, and K. Derr, "A novel software defined radio for practical, mobile crowd-sourced spectrum sensing," *IEEE Transactions on Mobile Computing*, pp. 1–1, 2021.

[43] S. Kapoor, S. Rao, and G. Singh, "Opportunistic spectrum sensing by employing matched filter in cognitive radio network," in *2011 International Conference on Communication Systems and Network Technologies*, 2011, pp. 580–583.

[44] S. AGHABEIKI, C. HALLET, N. E.-R. NOUTEHOU, N. RASSEM, I. ADJALI, and M. BEN MABROUK, "Machine-learning-based spectrum sensing enhancement for software-defined radio applications," in *2021 IEEE Cognitive Communications for Aerospace Applications Workshop (CCAAW)*, 2021, pp. 1–6.

[45] M. Sadiku and C. Akujuobi, "Software-defined radio: a brief overview," *IEEE Potentials*, vol. 23, no. 4, pp. 14–15, 2004.

[46] "About gnu radio," 2022. [Online]. Available: https://www.gnuradio.org/about/

[47] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library,"

in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[48] A. Inc., "Apple unleashes m1," *Apple.com*, 2020. [Online]. Available: https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/

[49] N. Jouppi, C. Young, N. Patil, and D. Patterson, "Motivation for and evaluation of the first tensor processing unit," *IEEE Micro*, vol. 38, no. 3, pp. 10–19, 2018.

[50] P. Jawandhiya, "Hardware design for machine learning," *Int. J. Artif. Intell. Appl*, vol. 9, no. 1, pp. 63–84, 2018.

[51] K. Berggren, Q. Xia, K. K. Likharev, D. B. Strukov, H. Jiang, T. Mikolajick, D. Querlioz, M. Salinga, J. R. Erickson, S. Pi *et al.*, "Roadmap on emerging hardware and technology for machine learning," *Nanotechnology*, vol. 32, no. 1, p. 012002, 2020.

[52] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, "Survey and benchmarking of machine learning accelerators," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, 2019, pp. 1–9.

[53] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "[dl] a survey of fpga-based neural network inference accelerators," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 12, no. 1, mar 2019. [Online]. Available: https://doi.org/10.1145/3289185

[54] Xilinx Inc., *7 Series FPGAs Configurable Logic Block User Guide*, Xilinx Inc., San Jose, CA, 2016.

[55] Altera Inc., *Cyclone V Device Handbook Volume 1: Device Interfaces and Integration*, Altera Inc., San Jose, CA, 2018.

[56] Xilinx Inc., *Zynq-7000 SoC Technical Reference Manual*, Xilinx Inc., San Jose, CA, 2021.

[57] ——, *7 Series FPGAs and Zynq-7000 SoC XADC Dual 12-Bit 1 MSPS Analog-to-Digital Converter User Guide*, Xilinx Inc., San Jose, CA, 2018.

[58] T. Matsuura, "Recent progress on cmos successive approximation adcs," *IEEJ Transactions on Electrical and Electronic Engineering*, vol. 11, no. 5, pp. 535–548, 2016.

[59] A. Suleiman and V. Sze, "An energy-efficient hardware implementation of hog-based object detection at 1080hd 60 fps with multi-scale support," *Journal of Signal Processing Systems*, vol. 84, no. 3, pp. 325–337, Sep 2016. [Online]. Available: https://doi.org/10.1007/s11265-015-1080-7

[60] E. Nurvitadhi, D. Kwon, A. Jafari, A. Boutros, J. Sim, P. Tomson, H. Sumbul, G. Chen, P. Knag, R. Kumar, R. Krishnamurthy, S. Gribok, B. Pasca, M. Langhammer, D. Marr, and A. Dasu, "Why compete when you can work together: Fpga-asic integration for persistent rnns," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 199–207.

[61] S. Schmitt, J. Klähn, G. Bellec, A. Grübl, M. Güttler, A. Hartel, S. Hartmann, D. Husmann, K. Husmann, S. Jeltsch, V. Karasenko, M. Kleider, C. Koke, A. Kononov, C. Mauch, E. Müller, P. Müller, J. Partzsch, M. A. Petrovici, S. Schiefer, S. Scholze, V. Thanasoulis, B. Vogginger, R. Legenstein, W. Maass, C. Mayr, R. Schüffny, J. Schemmel, and K. Meier, "Neuromorphic hardware in the loop: Training a deep spiking network on the brainscales wafer-scale system," in *2017 International Joint Conference on Neural Networks (IJCNN)*, 2017, pp. 2227–2234.

[62] "Amd ryzen 7 1700 processor," 2022. [Online]. Available: https://www.amd.com/en/product/1216

[63] K. Bai, L. Liu, and Y. Yi, "Spatial-temporal hybrid neural network with computing-in-memory architecture," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 7, pp. 2850–2862, 2021.

[64] S. Ortín and L. Pesquera, "Reservoir computing with an ensemble of time-delay reservoirs," *Cognitive Computation*, vol. 9, no. 3, pp. 327–336, Jun 2017. [Online]. Available: https://doi.org/10.1007/s12559-017-9463-7

[65] A. Atiya and A. Parlos, "New results on recurrent network training: unifying the algorithms and accelerating convergence," *IEEE Transactions on Neural Networks*, vol. 11, no. 3, pp. 697–709, 2000.

[66] M. Sussmann and T. Hill, "Intel hls compiler: Fast design, coding, and hardware," Intel Corporation, Tech. Rep., 2017. [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html

[67] Y. C. Lin, B. Zhang, and V. Prasanna, "Gcn inference acceleration using high-level synthesis," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, 2021, pp. 1–6.

[68] A. Misra, C. He, and V. Kindratenko, "Efficient hw and sw interface design for convolutional neural networks using high-level synthesis and tensorflow," in *2021 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, 2021, pp. 1–8.

[69] L. Kalms, P. A. Rad, M. Ali, A. Iskander, and D. Göhringer, "A parametrizable high-level synthesis library for accelerating neural networks on fpgas," *Journal of Signal Processing Systems*, vol. 93, no. 5, pp. 513–529, May 2021. [Online]. Available: https://doi.org/10.1007/s11265-021-01651-5

[70] Intel Corporation, *Intel HLS Compiler Pro Edition Reference Manual*, Intel Corporation, Santa Clara, CA, 2021. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683349/21-4/pro-edition-reference-manual.html

[71] ——, *Intel High Level Synthesis Compiler Pro Edition: Getting Started Guide*, Intel Corporation, Santa Clara, CA, 2021. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683680/21-4/pro-edition-getting-started-guide.html

[72] ——, *Intel High Level Synthesis Compiler Pro Edition: Best Practices Guide*, Intel Corporation, Santa Clara, CA, 2021. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683152/22-1/pro-edition-best-practices-guide.html

[73] "Intel high level synthesis compiler," 2022. [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html

[74] "bladerf 2.0 micro xa4," 2022. [Online]. Available: https://www.nuand.com/product/bladerf-xa4/

[75] "bladerf source," 2022. [Online]. Available: https://github.com/Nuand/bladeRF

[76] Intel Corporation, *Intel Stratix 10 Variable Precision DSP Blocks User Guide*, Intel Corporation, Santa Clara, CA, 2018. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/archives/ug-s10-dsp-18-1.pdf

[77] M. Bouvier, A. Valentian, T. Mesquida, F. Rummens, M. Reyboz, E. Vianello, and E. Beigne, "Spiking neural networks hardware implementations and challenges: A survey," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 15, no. 2, apr 2019.

[78] K. Hamedani, L. Liu, S. Hu, J. Ashdown, J. Wu, and Y. Yi, "Detecting dynamic attacks in smart grids using reservoir computing: A spiking delayed feedback reservoir based approach," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 4, no. 3, pp. 253–264, 2020.

[79] K. Hamedani, L. Liu, S. Liu, H. He, and Y. 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*, vol. 34, no. 02, pp. 1292–1299, Apr. 2020. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/5484

[80] K. Hamedani, L. Liu, R. Atat, J. Wu, and Y. Yi, "Reservoir computing meets smart grids: Attack detection using delayed feedback networks," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 2, pp. 734–743, 2018.

[81] J. Li, C. Zhao, K. Hamedani, and Y. Yi, "Analog hardware implementation of spike-based delayed feedback reservoir computing system," in *2017 International Joint Conference on Neural Networks (IJCNN)*, 2017, pp. 3439–3446.

[82] K. Hamedani, L. Liu, and Y. Yi, "Energy efficient mimo-ofdm spectrum sensing using deep stacked spiking delayed feedback reservoir computing," *IEEE Transactions on Green Communications and Networking*, vol. 5, no. 1, pp. 484–496, 2021.

[83] K. Bai and Y. Y. 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)*, 2018, pp. 322–328.

[84] K. Bai, J. Li, K. Hamedani, and Y. 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)*, 2018, pp. 1–6.

Appendix A

DFR High-Level Synthesis Source Code

Here an overview of the HLS code used to synthesize the delayed feedback reservoir network is provided. In Intel HLS Compiler, the "component" keyword is used to indicate C++ that should be transformed into a synthesizable intellectual property (IP) core. Also shown in this example is the usage of the "unroll" directive to expand the single loop into two parallelized versions. Other HLS constructs used in this example include: (1) the initialize on reset keyword to indicate a synthesizable memory that is reset to a default value when the reset is asserted; (2) the stable argument keyword to indicate input values to the IP will remain fixed; and (3) the custom delayed feedback reservoir floating point type to control the quantization level of the data.

Listing A.1: C++ Code for HLS DFR Core

```
#include "HLS/hls.h"
#include "HLS/math.h"
#include "HLS/hls_float.h"
#include "HLS/hls_float_math.h"
```

```
using DFR_FP = hls_float<8, 17, fp_config::FP_Round::RNE>;

component hls_stall_free_return DFR_FP dfr(hls_stable_argument short i_data,
    hls_stable_argument short q_data) {

    // persistent reservoir
    static DFR_FP reservoir[N] hls_init_on_reset;

    // track output
    DFR_FP dfr_out = 0;

    // calculate energy
    DFR_FP i_data_scaled = DFR_FP(i_data) / DFR_FP(MAX_ADC_SIGNED);
    DFR_FP q_data_scaled = DFR_FP(q_data) / DFR_FP(MAX_ADC_SIGNED);
    DFR_FP sample = ihc_sqrt(i_data_scaled * i_data_scaled + q_data_scaled *
    q_data_scaled);

    // loop through each masked input subsample
    #pragma unroll 2
    for(unsigned node_idx = 0; node_idx < N; node_idx++){
        // calculate next node value based on current subsample
        DFR_FP masked_sample_i = MASK[node_idx] * sample;
        DFR_FP mg_in = dfr_gamma * masked_sample_i + dfr_eta * reservoir[
    LAST_NODE - node_idx];

        // Mackey Glass Equation
        DFR_FP mg_power = ihc_pow(mg_b * mg_in,mg_p);
        DFR_FP mg_denominator = mg_a + mg_c * mg_power;
        DFR_FP mg_numerator = mg_C * mg_in;
        DFR_FP mg_out = mg_numerator / mg_denominator;

        // update reservoir
        reservoir[LAST_NODE - node_idx] = mg_out;
```

```
        // calculate output
        dfr_out += W[LAST_NODE - node_idx] * mg_out;
    }

    return dfr_out;
}
```