PARAMETERIZABLE DESIGN ON CONVOLUTIONAL NEURAL NETWORKS WITH CHISEL HARDWARE CONSTRUCTION LANGUAGE

by

Mukesh Chowdary Madineni, B.E.

THESIS

Presented to the Faculty of

The University of Houston-Clear Lake

In Partial Fulfillment

Of the Requirements

For the Degree

MASTER OF SCIENCE

in Computer Engineering

THE UNIVERSITY OF HOUSTON-CLEAR LAKE

MAY, 2023

PARAMETERIZABLE DESIGN ON CONVOLUTIONAL NEURAL NETWORKS WITH CHISEL HARDWARE CONSTRUCTION LANGUAGE

by

Mukesh Chowdary Madineni

APPROVED BY

Xiaokun Yang, PhD, Chair

Hakduran Koc, PhD, Committee Member

Ishaq Unwala, PhD, Committee Member

APPROVED/RECEIVED BY THE COLLEGE OF SCIENCE AND ENGINEERING:

David Garrison, PhD, Associate Dean

Miguel A. Gonzalez, PhD, Dean

Dedication

I dedicate this dissertation to my friends and my parents. Without their encourage, understanding, and most of all love, the completion of this work would not have been possible.

Acknowledgments

I would like to acknowledge the faculty and staff in the Computer Engineering department at the University of Houston – Clear Lake. I would like to acknowledge my thesis committee members for inspiring and helping me. I would like to thank Dr. Xiaokun Yang for his support and knowledge. Without his insight my thesis work would have been a very difficult journey. I also would like to thank Dr. Hakduran Koc for mentoring me throughout my computer engineering academic career and by helping me prioritize my thesis work. I would like to specially thank Mario Vega who was my research lab mate for providing his research work related to floating-point implementation and sharing his knowledge. I feel very lucky that I am at a school that has outstanding professors who have helped me grow academically and professionally. Finally, I would like to acknowledge my computer engineering friends who have helped me with my thesis related and academic work.

ABSTRACT

PARAMETERIZABLE DESIGN ON CONVOLUTIONAL NEURAL NETWORKS WITH CHISEL HARDWARE CONSTRUCTION LANGUAGE

Mukesh Chowdary Madineni

University of Houston-Clear Lake, 2023

Thesis Chair: Xiaokun Yang, PhD

This thesis presents a parameterizable design generator on convolutional neural networks (CNNs) using Chisel hardware construction language (HCL). Chisel HCL is an open-source embedded domain-specific language (created and maintained by University of California, Berkeley) that inherits the object-oriented feature of Scala for constructing hardware. By parameterizing structural designs such as the streaming width, pooling layer type, and floating-point precision, multiple register-transfer level (RTL) implementations can be created to meet various accuracy and hardware cost requirements. The HCL design can generate the RTL implementations with Verilog, which is synthesizable and implementable on FPGAs (field-programmable gate arrays). The evaluation is based on generated RTL designs including 16-bit, 32-bit, 64-bit, and 128-bit implementations on FPGAs. The experimental results show

that the 32-bit design achieves optimal hardware performance when setting the same weights for estimating the quality of results, FPGA slice count, and power dissipation. Although the focus is on CNNs, the approach can be extended to other neural network models for efficient RTL designs.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 Background

Neural networks (NNs) have become an extensively used technique for image classification, speech processing, digit recognition, and many more [29, 36, 35]. In the era of high-performance computing, leveraging the design complexity, power dissipation, and quality of results is one of the big challenges for the hardware implementation of complex NNs. Field-programmable gate arrays (FPGAs) are a popular choice for hardware acceleration due to their parallelism and power efficiency, but their limited extensibility between projects and design specifications is a challenge.

Additionally, most existing FPGA implementations are based on software-hardware co-design platforms, where the processors and FPGAs are on the same chip for the execution of controllers and data processing [8, 9, 10, 38]. With a wide range of applications of CNNs in image classification and detection, they can be implemented on an Electronic Control Unit (ECU) for autonomous driving assistance in an automobile application [31, 37, 32, 33, 34]. For instance, the authors in [32] discuss the application of 3D image detection for an autonomous vehicle. This 3D image analysis can be handled through CNNs by applying multiple filters on the same frame for different feature extraction. The survey in [34] highlights the acceptable accuracy and detailed spatial point extraction of CNNs for a distributed automation system. In order to provide a configurable hardware core to complex NNs, a parameterizable register-transfer level (RTL) design, which can be compatible with reference EDA tool flows and work across both FPGA and ASIC implementations, is necessary.

Another research direction for implementing NNs on FPGAs is based on generating Verilog Hardware Description Language (HDL) with HDL generators, particularly

for reusable intellectual properties (IPs) such as arithmetic operators and standard design modules including wrappers and interfaces [11]. These generators are often created using script languages like Perl/tcl or high-level synthesis tools. However, a major drawback of this approach is the lack of robust libraries supporting the generators, which can make the generator design more challenging than coding in Verilog. Additionally, using such generators without considering hardware-related descriptions like timing and performance constraints can result in timing violations and performance issues in the generated HDL code.

To leverage programming productivity and high-performance hardware implementations, using Hardware Construction Languages (HCL), such as Chisel, is one of the RTL design options. Chisel HCL is an open-source, embedded domain-specific language that uses object-oriented features from Scala to construct hardware at a higher abstraction level, including hardware information like signal width and timing. Chisel can generate RTL implementations with Verilog, which is synthesizable and can be implemented on both ASICs and FPGAs. However, previous works on HCL-related designs were limited to complex mathematical algorithms[15], lacking the scalability feature necessary for designing parameterizable NNs.

Under this context, this thesis investigates the use of Chisel HCL for designing parameterizable convolutional neural networks (CNNs), building on prior work on the HCL-HDL design flow [4]. A case study is presented, demonstrating the design of a configurable binary design library that includes fundamental arithmetic circuits like full-adders, full-subtractors, binary multipliers, shifters, and many more. Experimental results show that the proposed design methodology achieves the same accuracy as Verilog HDL implementations, while also estimating the hardware cost in terms of slice count, power consumption, and maximum clock frequency. Using the design

library a parameterizable CNN which is open to both ASIC and FPGA-based design tools is constructed. Specifically, below are the main contributions of this thesis:

- This thesis presents a Verilog RTL generator for CNNs that can be easily customized by parameterizing the precision of submodules and the structural designs of layers, such as the streaming width of convolution and fully-connected layers, as well as the max/mean pooling layers.
- This thesis conducts a design flow from a Chisel HCL description to a Verilog HDL design, and lastly to the final hardware cost, implementation, and evaluation. Experimental results show that the 32-bit design achieves the optimal normalized performance when considering error percentage, FPGA slice count, and energy dissipation with equal weights.
- Our proposed streaming design achieves an accuracy of 98.39% with lower resource cost and latency compared to prior works. By employing an iterative design structure, it can further reduce resource cost in terms of look-up tables (LUTs) and registers at the expense of more clock cycles. The case study can be extended to the design of other complex NNs as well as multi-layer CNN designs.

1.2 Structure Of The Thesis

The remainder of this thesis is organized as follows: we first review the relevant related works to the application of implementing CNNs on FPGAs, and Chapter 2 presents the background on CNN. In Chapter 3, Chisel HCL is introduced along with the design flow of the implementation methodology. Later the thesis goes on to explain modules fixed-point and floating-point submodules used in the implementation of CNN in Chapters 4 & 5. The proposed parameterizable design of the CNN and

implementation are discussed in Chapter 6. In Chapter 7, the FPGA design performance is evaluated in terms of slice count and power dissipation. Finally, Chapter 8 concludes this thesis.

1.3 Related Works

Numerous works have focused on the hardware design of CNNs, primarily addressing challenges such as reducing computational complexity and energy dissipation[16, 18]. For example, real-world applications that involve complex computation require a hardware accelerator with floating-point (FP) operations. In order to reduce the hardware cost, the accuracy can be traded off by employing fixed-point implementations for calculation [16, 39]. As an example in [18], an 8-bit and 16-bit fixed-point design is carried out to demonstrate a lower accuracy implementation with less hardware resource cost compared with the FP implementations. Another immediate option for low-cost design is to simplify the design structures of the CNN models. For example, in reference [14] authors implemented a Super Skinny CNN (SS-CNN) with 39,541 parameters and three layers in addition to the input and output layer. The overall latency of the design on a Cyclone IVE FPGA is about 2.2 seconds. Additionally, hardware design engines have been widely employed as accelerators for complex computations of neural networks. For example, a latency of 3.58 ms and 3.2 ms with an accuracy of 98.64% and 96% were achieved in two high-level synthesis designs on LeNet-5 CNN, presented in [12] and [13], respectively.

The arithmetic logic units are the fundamental building blocks of system-level circuits and design applications. A modern real-time application requires very powerful and complex arithmetic operators in its design, from a basic microprocessor to image/video processing unit to complex neural networks [30].

To balance the quality of the CNN results with hardware resource utilization, the RTL designs are based on the FP operators provided by the FPGA tools. For example, the FP adders and multipliers from AMD Vivado are used to construct multi-layer perceptron (MLP) NNs in references [26, 25, 40]. By offering different levels of parallelism in the design engine, five MLP NNs were presented as case studies. The implementations are based on the provided Vivado IPs and not the RTL programming, therefore, the implementations are not synthesizable to be an ASIC. Second, all five design structures are based on single-precision modules due to the limited configurations provided by AMD Vivado. Another example of a configurable design was demonstrated in [23]. The RTL design on a CNN has reconfigurable convolution, pooling, and fully connected modules. The system was built using reconfigurable IP cores and deployed on the Intel Cyclone10 FPGA platform. Experimental results showed that the implementation achieved a latency of 17.6 us with an accuracy of 97.57%.

From the hardware designer's perspective, the fundamental design neurons in each layer can be constructed with multiple FP operators such as adders and multipliers. Furthermore, the CNN can be integrated with multiple neurons and network layers. In the preliminary results in [4], the binary design library was proposed as a case study of the design flow from Chisel HCL to Verilog HDL to the final FPGA development and evaluation.

Further, the comparative study on designing with Chisel HCL against Verilog HDL was carried out in [28]. A N-bit fixed priority arbiter was designed in Chisel HCL and Verilog to compare the timing, power, and area of the designed module. The Chisel implementation required 250 lines of Chisel source code whereas the Verilog implementation required 400 lines. This article also showed that the Chisel implemented design used less hardware resources compared to the Verilog design. Using

the Chisel platform, additionally it is able to make the design library open to ASIC simulators and synthesis tools like Synopsis VCS and Design Compiler, and Cadence NC and Genus Synthesis Solution.

In this thesis, the extended work focuses on the parameterizable design of CNNs with the binary and FP design library. The proposed approach supports a series of pipelined CNN engine designs that can be applied to a wide variety of problems. Highly parameterized digital circuit generators allow developers to re-use and customize their implementations for different design specifications, thus cutting design cycles and complexity.

CHAPTER II

FUNDAMENTAL THEOREM OF CNN

This section discusses the fundamental theorem of CNNs, including the number of hidden layers and activation functions used for constructing the network.

2.1 CNN Structure

The mathematical model for a basic CNN typically includes a multi-layer convolution layer, a pooling layer, a fully-connected layer, and a soft-max layer, as illustrated in Fig. 2.1. The output of each layer is referred to as the output feature map, which serves as the input feature map for the subsequent layer, creating inter-operational characteristics. The number of convolutional and pooling layers can be adjusted and arranged in the network depending on the implementation algorithms and application requirements.

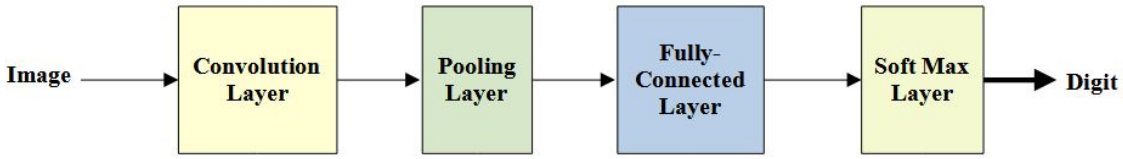

Figure 2.1: *CNN Structure.*

2.2 Convolutional Layer

The convolutional layer performs feature extraction from the input matrices. The calculation of the convolution result involves the idea of both local receptive field and shared weights and bias.

2.2.0.1 Local Receptive Fields

The convolution layer input is considered a 10×10 pixel intensity, which is fed in as input matrices. Instead of entering all the input pixels, the connections are made in small, partial regions of the input matrix. For example, a 3×3 region; corresponding to 9-input pixels is connected to a small region of the input neurons. So, for a particular neuron, the connections are shown in Fig. 2.2. On the right-hand side of the image, the highlighted element is the output neuron corresponding to the selected partial region on the left-hand side of the image.

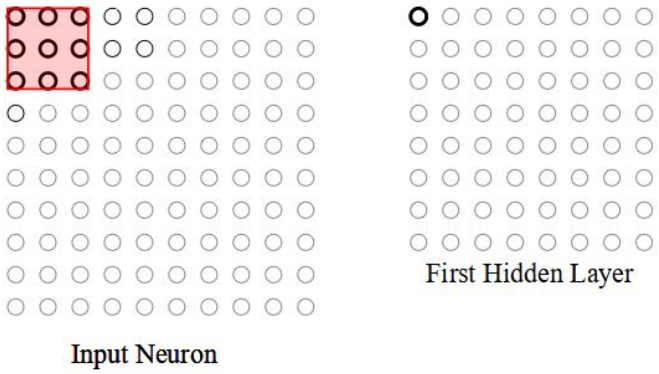

Figure 2.2: *Local Receptive Field in the Top-left Corner to Connect to First Hidden Neuron.*

The region in the input pixel image to which the neuron is connected is called the local receptive field. The neuron learns an overall bias of the field. The local receptive field is slid across the entire input image, corresponding to having a different neuron in the hidden layer. Fig. 2.3 shows that the local receptive field is moved along the input image by one pixel at a time. Stride length decides the shift amount of the local receptive field in the input image pixel matrix.

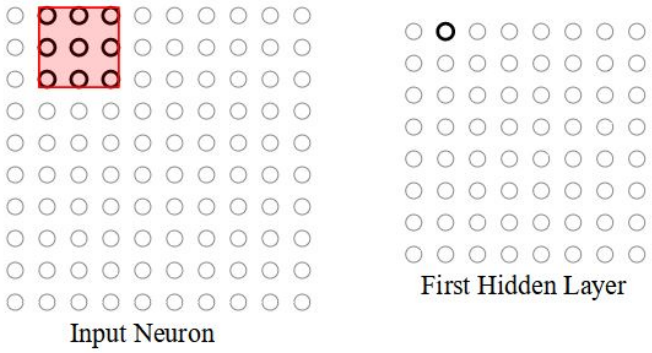

Figure 2.3: *Local Receptive Field is Slid Over by One Pixel to the Right (i.e., by One Neuron), to Connect to a Second Hidden Neuron.*

If the input is 10×10 and the local receptive field is 3×3, then 8×8 neurons will be generated in the hidden layer. This is because the local receptive field can be moved seven neurons across (or downwards) before reaching the edge of the input image. More generally, for an input of size $A \times A$ and a local receptive field of size $B \times B$, the output matrix size can be calculated using an equation

$$output = rounddown\left(\frac{A-B}{stride}\right) + 1 \tag{2.1}$$

where the round-down function is used to round the result to the nearest lowest integer. For example, the above input matrix A is 10×10, B is 3×3, and stride length is one. So, the output matrix size will be 8×8.

2.2.0.2 Shared weights and biases

The shared weights and biases determine the value of the output neuron in the convolution layer. The exact weights and biases will be used by the entire matrices to obtain 8×8 hidden neurons. In other words, the output for the j^{th}, k^{th} hidden neuron can be expressed as:

$$output_{j,k} = \sigma(b_{j,k} + \sum_{l=0}^{2}\sum_{m=0}^{2} w_{l,m} a_{j+l,k+m}). \tag{2.2}$$

where σ represents the neural activation function, such as the Rectified Linear Unit (ReLU) activation function [23]. The indexes j and k range from 0 to 2 for a 3×3 local receptive field. $b_{j,k}$ represents the shared value for the bias, $w_{l,m}$ denotes a 3×3 array of shared weights, and $a_{x,y}$ denotes the input activation at position x,y. The equation indicates that all the neurons in the hidden layer detect the same feature. It is generally called the feature map from the input layer to the hidden layer.

2.3 Pooling Layer

The pooling operation can be carried out after the convolution layer, which prepares a condensed feature map. Each unit in the pooling layer may summarize a region of 2×2 neurons from its previous layer. The operation used in summarizing the output is max-pooling or mean-pooling.

In the max-pooling, the pooling region outputs the maximum activation in the 2×2 input region, as illustrated in Fig. 2.4. On the left-hand side, it shows the output neurons of the previous layer. The maximum value in the selected region is processed as an output of the max-pooling operation. In contrast, in mean-pooling, the region outputs the average of the four neurons in the selected region. The selection of max or mean pool operation is parameterizable in our proposed work. Practically, the max pool operation extracts the brightest pixel in the regions, whereas the mean pool operation spreads the brightness of pixels in the given area.

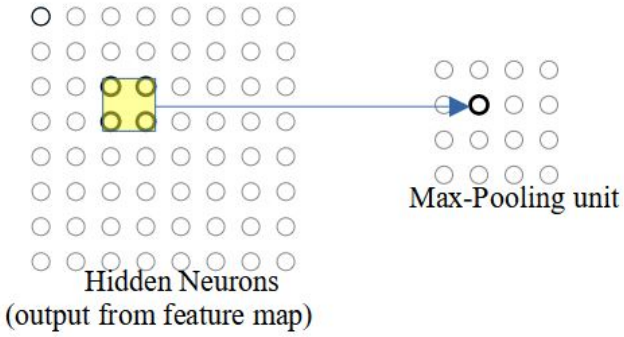

Figure 2.4: *Max Pooling of* 2×2 *on the Output Feature Map.*

As shown in this example, the input to the pooling layer is 8×8 neurons and the stride is two, therefore, using Equation 2.1 the size of the output matrix is 4×4.

2.4 Fully Connected Layer

After the convolutional and pooling layers, the input matrix is converted into a suitable form for CNN. In what follows, the matrix is further flattened into a column vector. The linear output is fed into a feed-forward neural network. In fully connected layers, every input is connected to every output by a learnable weight. Fig. 2.5 shows the operation of a fully connected layer after flattening the output from the pooling layer. In the fully-connected layer, each neuron performs a multiplication-addition operation with the kernel weights and bias values, which is similar to the execution in the convolution operation.

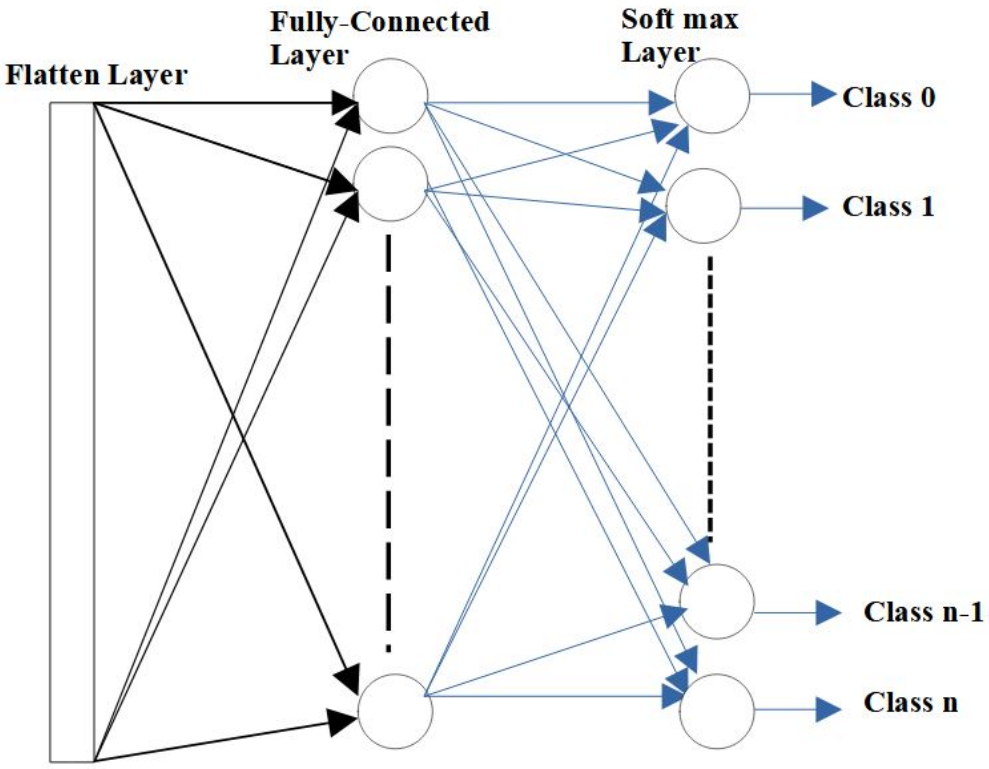

Figure 2.5: *Fully−Connected Layer Operation.*

2.5 Soft-max Layer

In the soft-max layer, the output corresponds to the highest value in the output neurons from the fully connected layer. In the application of digit recognition, as an example, this layer is used to find the highest percentage of the classified results. This method of distinguishing between dominating and low-level features in an image is called the soft-max classification technique.

CHAPTER III

CHISEL HARDWARE CONSTRUCTION LANGUAGE

3.1 Introduction to Chisel HCL

Chisel HCL is a hardware design language/generator created by University of California, Berkeley that supports advanced hardware design by using highly parameterized generators [27]. It provides circuit generation and reuse of design components for both ASIC and FPGA-based digital circuit designs. Chisel adds hardware construction primitives to Scala embedded programming language, allowing designers to construct parameterizable circuit generators that produce synthesizable Verilog HDL. The use of Chisel HCL in designing modules has been restricted to complex mathematical algorithms leaving behind the scalability feature to design a parameterizable module.

First, the Chisel HCL is employed to build the parameterizable design libraries including fundamental FP operations and fixed point operations [4] like adders, multipliers, dividers, reciprocals, square roots, exponential functions, etc. By parameterizing the precisions and latency over clock cycles, the FP submodules needed for constructing the CNN will be generated with Verilog HDL.

3.2 Development Environment and Chisel HCL set up

The Chisel HCL is based on the Scala language, which itself is based on the Java language. For this project, I will be using the IntelliJ IDE from JetBrains. The IntelliJ IDE was designed primarily for Java coding, but we can install a Scala plug-in to be able to make Scala projects and import the Chisel HCL libraries.

To install Chisel libraries, go to the settings menu and go to the plugins tab. Search for the Scala plugin and install as shown in Fig. 3.2.

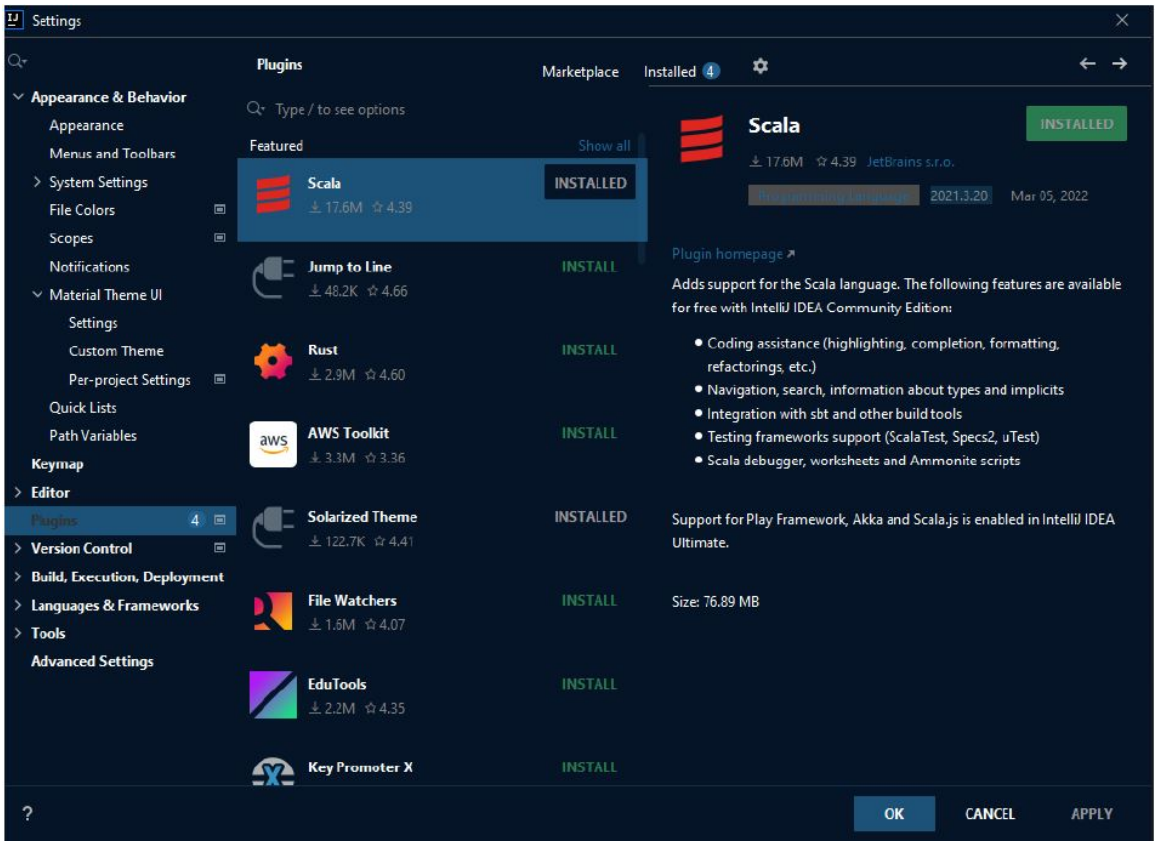

Figure 3.1: *Chisel Libraries.*

After installing the Scala plugin, create a new Scala Project. In the directory you will find a build.sbt file. Open the file and add the following lines to get all the libraries for Chisel.

```
lazy val root = (project in file("."))
  .settings(

    scalaVersion := "2.13.7",
    addCompilerPlugin("edu.berkeley.cs" % "chisel3-plugin" % "3.5.0" cross
CrossVersion.full),
    libraryDependencies += "edu.berkeley.cs" %% "chiseltest" % "0.5.0",
    libraryDependencies += "edu.berkeley.cs" %% "chisel3" % "3.5.0",
    // We also recommend using chiseltest for writing unit tests
    libraryDependencies += "edu.berkeley.cs" %% "chiseltest" % "0.5.0" % "test",
    name := "ChiselParameterizedDesigns"
  )
```

Figure 3.2: *Build file to include Chisel Libraries.*

3.3 Chisel Design Flow

In this section, it presents a configurable and reusable binary design library that is developed with Chisel HCL. The IntelliJ software from JetBrains is employed as the developing environment with Scala plugins and Chisel HCL libraries. For example, Fig. 3.3 shows the design flow using the proposed Chisel library, from the Chisel design and verification, then the generated Verilog HDL. After the HDL generation, the traditional FPGA design procedure including the synthesis, layout, and final implementation and evaluation can be conducted.

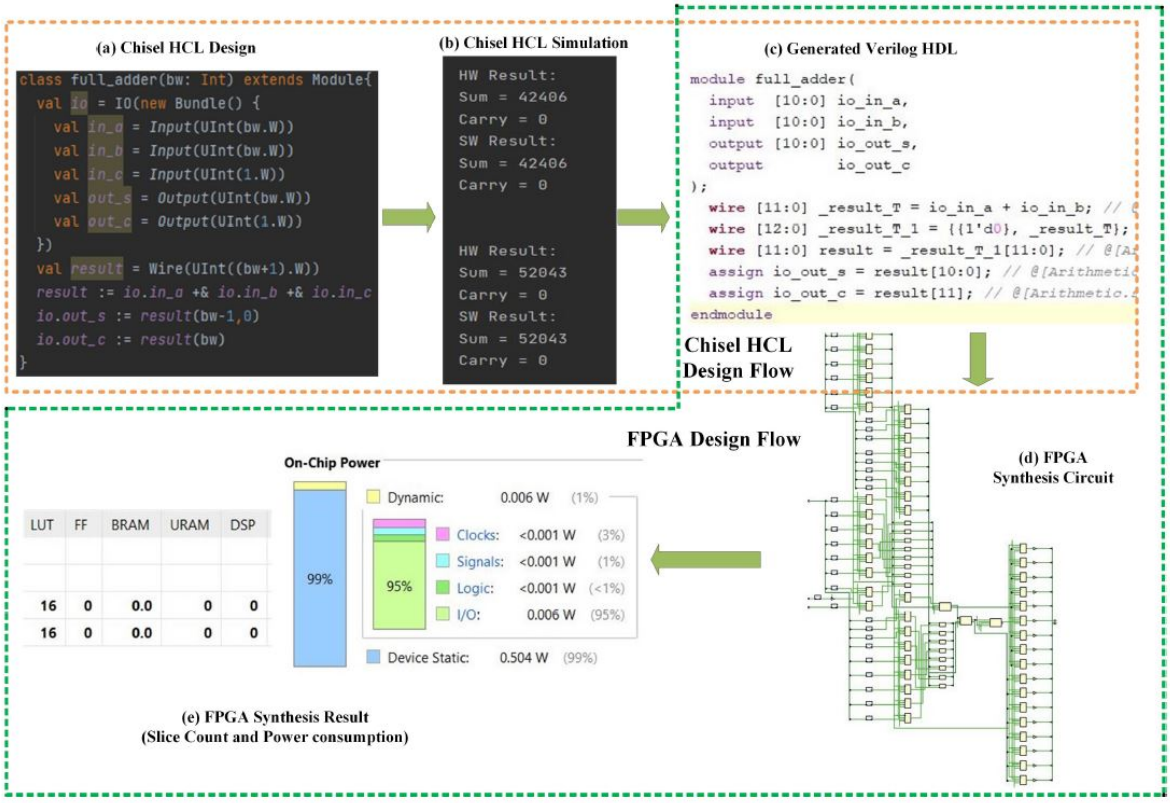

Figure 3.3: *A Design Example between Chisel HCL to Verilog HDL Design Flow.*

Specifically, Fig. 3.3(a) shows the example of a Chisel HCL design on a full adder. In what follows, the binary design library is imported to a Scala project. After

importing the library, a Scala object file is used to run the design module that is present in the main package. Fig. 3.3(b) shows the simulation result of the Chisel HCL design after running the test script file by passing the input data. This result proves the functionality of the design.

After declaring the Chisel design module, we then use Scala to call the Chisel compiler to translate Chisel designed module into Verilog HDL. This elaboration process requires passing bit width as a parameter to generate a synthesizable Verilog for the design module. The generated Verilog HDL is written into a new Verilog file which is added into a Vivado project as shown in Fig. 3.3(c).

In Vivado, the FPGA design flow can be carried out including synthesis and implementation. The synthesis circuit can be shown in Fig. 3.3(d). After Synthesis, the final performance estimation is conducted. Fig. 3.3(e) shows the results of FPGA slice count and power analysis after the implementation is successfully completed.

3.4 Design Approach

The neural network design which is discussed in this thesis is not intended to perform the training procedure. The training will be done through existing software models which will generate the lists of optimized inputs, weights, and biases. Through the Scala/Chisel language, we can take the list of weights and biases and turn them into ROMs which can be accessed by the hardware model. However, when it comes to testing the neural network design, we don't need to have optimized weights and biases because we can just generate random sets of inputs, weights, and biases, and compare the results from the hardware model with a software golden model.

For the design of the parameterizable neural network, we also need access to Floating-Point arithmetic circuit designs that allows us to compute all the compu-

tations involved with the convolution calculations. Since the Chisel language allows scalability, Fixed-Point, and Floating-Point arithmetic modules were designed for an adjustable precision level of the IEEE-754 numbers, where precision of the design can be selected.

CHAPTER IV

FIXED POINT COMPONENTS

This section presents the implementations of six fundamental designs of binary arithmetic circuits, including full-adder, full-subtractor, multiplier, shifter, 2's complement operator, and leading bit detector modules.

4.1 Full-adder module design

Full-adder is a basic arithmetic module used in mathematical computation. This module is built on the logic of binary addition. Based on the requirement, the bit width for input numbers and the output sum is selected and the Verilog code is generated. It takes a single clock cycle to compute output sum and carry for the selected bit width. The Fig. 4.1 shows the synthesis schematic with the RTL analysis for a 16-bit full-adder. The maximum clock frequency that can be achieved for a 16-bit module is 604.96 MHz and the total on-chip power is 510 mW.

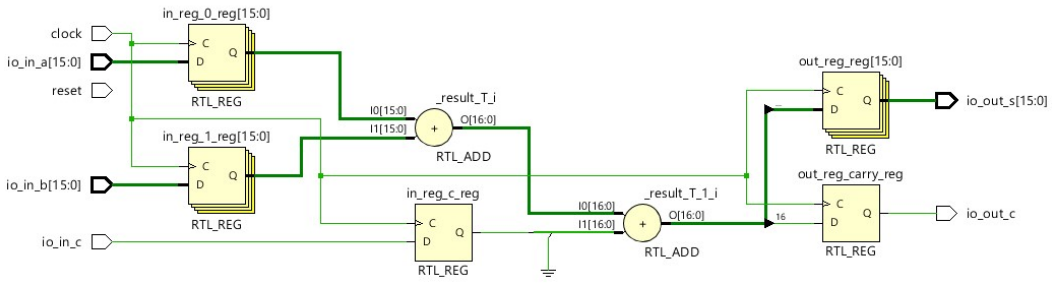

Figure 4.1: *16-bit full-adder synthesis result.*

4.2 Full-subtractor module design

Similar to the full-adder module, the full subtractor is developed on the logic of binary subtraction. This module takes two numbers as input and computes the borrow out and difference as the output. This module takes a single clock cycle to

compute the output. The bit width for the module can be selected based on the requirement. The Fig. 4.2 shows the synthesis schematic with the RTL analysis for a 16-bit full-subtractor. The maximum clock frequency that can be obtained from the 16-bit module is 868.8 MHz and the total on-chip power is 510 mW.

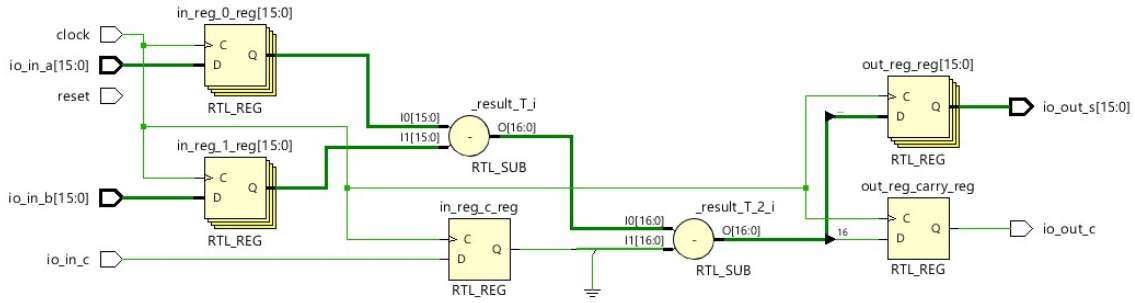

Figure 4.2: *16-bit full-subtractor synthesis result.*

4.3 Multiplier module design

Similar to the multiplication of two decimal numbers, the binary multiplier follows the same method for computing a product result of the two binary numbers. The bit width for the product result is double the size of the input numbers. The product of two binary numbers is computed in a single clock cycle. The advantage of using a fixed-point multiplier is that it can be built using look-up-tables, saving DSPs resources on the FPGA. The Fig. 4.3 shows the synthesis schematic with the RTL analysis for a 16-bit binary multiplier. The maximum clock frequency that can be obtained from the 16-bit module is 148.65 MHz and the total on-chip power is 514 mW.

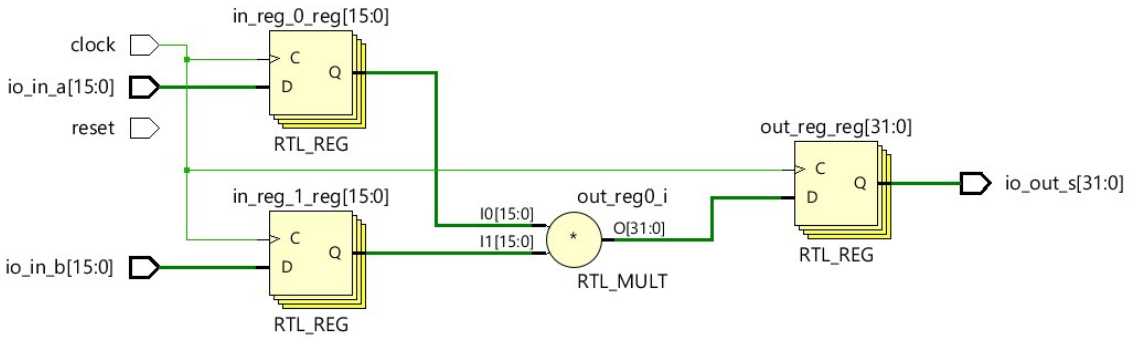

Figure 4.3: *16-bit multiplier synthesis result.*

4.4 Shifter module design

The shifter module shifts the input number by a specified number of bit positions to the right or left. The input to this module is the number to be shifted, the number of bit positions, and, the shift left or right. The output bit width is considered to have the same bit width of the input number that is being shifted.The Fig. 4.4 shows the synthesis schematic with the RTL analysis for a 16-bit shifter. The maximum clock frequency that can be obtained from the 16-bit module is 393.54 MHz and the total on-chip power is 508 mW.

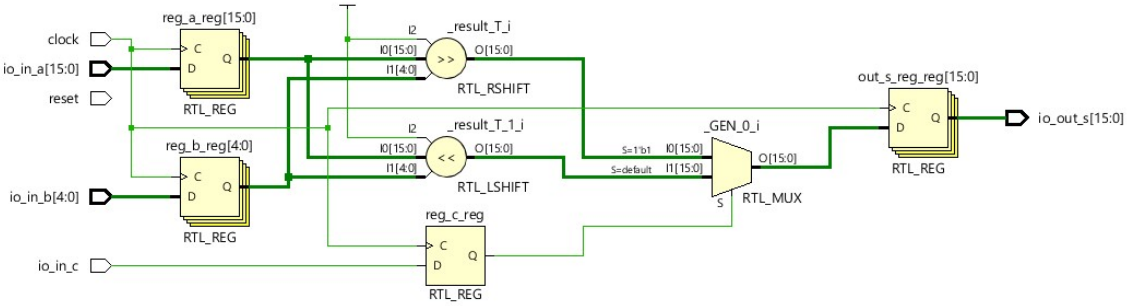

Figure 4.4: *16-bit shifter synthesis result.*

4.5 Two's complement module design

Two's complement of a number is used to store the negative value of a number. This module involves functions such as bit flipping and bit-wise addition. The Fig. 4.5 shows the synthesis schematic with the RTL analysis for a 16-bit two's compliment design module.The maximum clock frequency that can be obtained from a 16-bit module is 708.71 MHz and the total on-chip power is 510 mW.

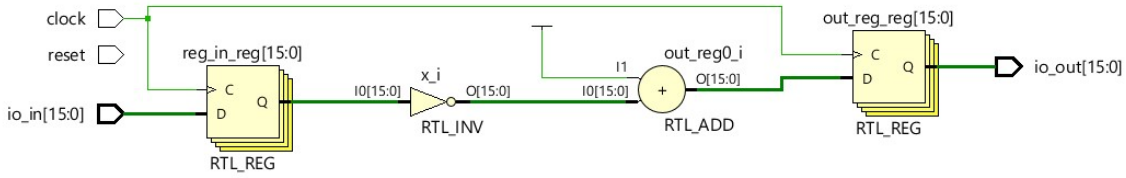

Figure 4.5: *16-bit two's compliment synthesis result.*

4.6 Leading one detector module design

The leading one bit detector is designed to return the bit position of the most significant bit in the number. It is designed to use in a floating-point arithmetic operation. The specified bit widths are 11, 24, 53, and, 113. The Fig. 4.6 shows the synthesis schematic with the RTL analysis for a 11-bit leading one detector module. The maximum clock frequency that can be obtained from the 11-bit module is 896.86 MHz and the total on-chip power is 505 mW.

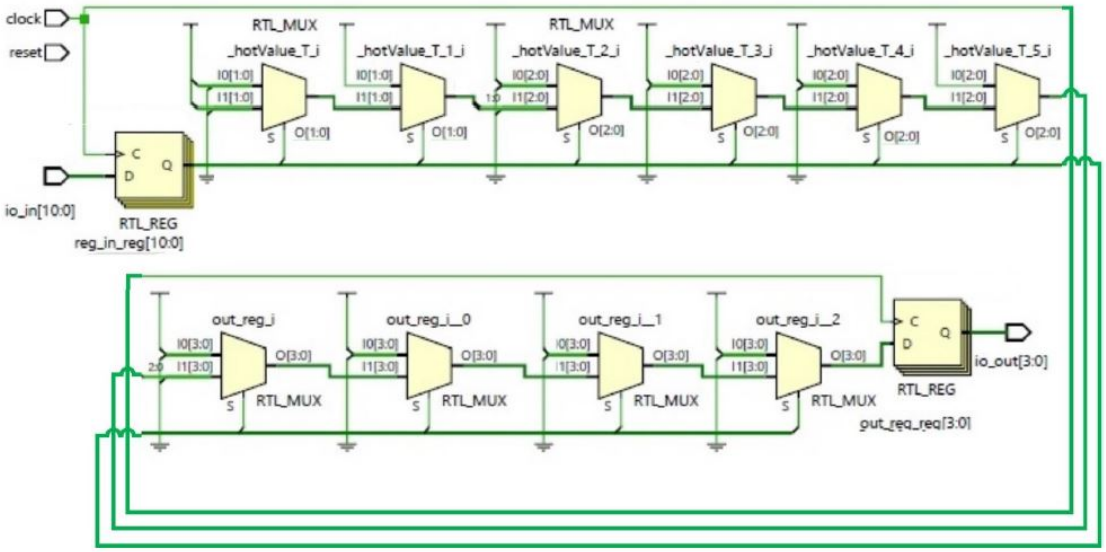

Figure 4.6: *11-bit Leading one detector synthesis result.*

CHAPTER V

FLOATING POINT COMPONENTS

This chapter presents the implementations of fundamental designs of floating point arithmetic circuits, including adder, multiplier, multiply and accumulate (MAC), and accumulator modules. The floating-point arithmetic modules are designed using parameterizable binary arithmetic modules discussed in the previous chapter. Most arithmetic operation involves a fixed-point operation to compute the floating-point result.

The IEEE 754 format consists of three main parts: sign, exponent, and mantissa as shown in Fig. 5.1 for a 32-bit floating point number. The sign indicates if the number is positive or negative and it is always represented by the msb. The exponent part of the number indicates the number of left shifts or right shifts that were used to normalize the binary representation of the floating-point number, but this is also includes the bias associated with the IEEE 754 precision level. The mantissa of the number is simply the fractional part of the normalized floating-point number. An important thing to mention is that the bit width for the exponent, bias, and mantissa will vary for different precision levels, but the concept will remain the same. For 32-bit IEEE 754 format, the exponent is 8 bits, mantissa is 23 bits, and bias is 127. The IEEE floating-point number has the value as shown in equation 5.1.

$$num = -1^{sign} * 1.mantissa * 2^{(exponent-bias)} \tag{5.1}$$

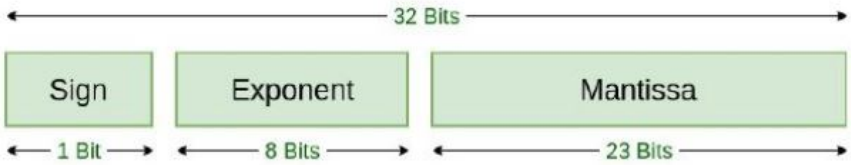

Figure 5.1: *Single Precision IEEE 754 Floating-Point Standard.*

5.1 Floating Point Adder Design

This circuit is designed to perform the addition of two IEEE 754 floating-point numbers. Since the IEEE 754 format has different precision representations, we have designed the FP adder module to have adjustable precision levels, which is specified through a parameter. Currently, the design supports 16,32,64, and 128-bit precision. The input to this module should be a number represented in IEEE 754 format, and the output sum is calculated. The Fig. 5.2 shows the schematic view of the floating-point adder after synthesizing the generated RTL code.

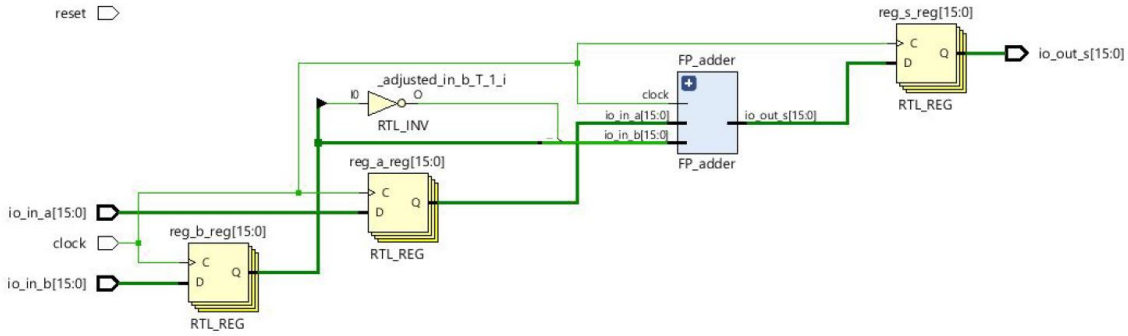

Figure 5.2: *16-bit floating point adder synthesis result.*

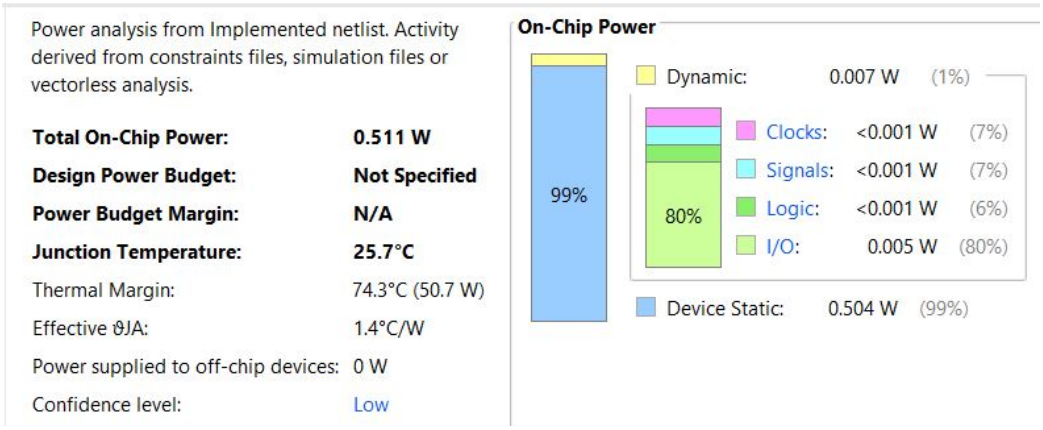

Figure 5.3: *16-bit floating point adder power analysis.*

After implementation, the power analysis is conducted as the static and dynamic on-chip power consumption on the FPGA board. The power implementation estimate of the FP adder is shown in Fig. 5.3. The total on-chip power estimation is 0.511 W. The on-chip power desitricution for static power is 1% while the dynamic power is the remaining 99%.

5.2 Floating Point Multiply Design

Similar to the multiplication of two decimal numbers, the floating point multiplier follows the same method for computing a product result of the two floating point numbers. This circuit has been designed to perform the multiplication operation on IEEE 754 floating-point numbers. Like the FP-adder described previously, this design also comes with adjustable precision. The circuit of floating point multiplier involves fulladder, full-subtractor, and two's compliement. The Fig. 5.4 shows the schematic of 16-bit floating-point multiplier

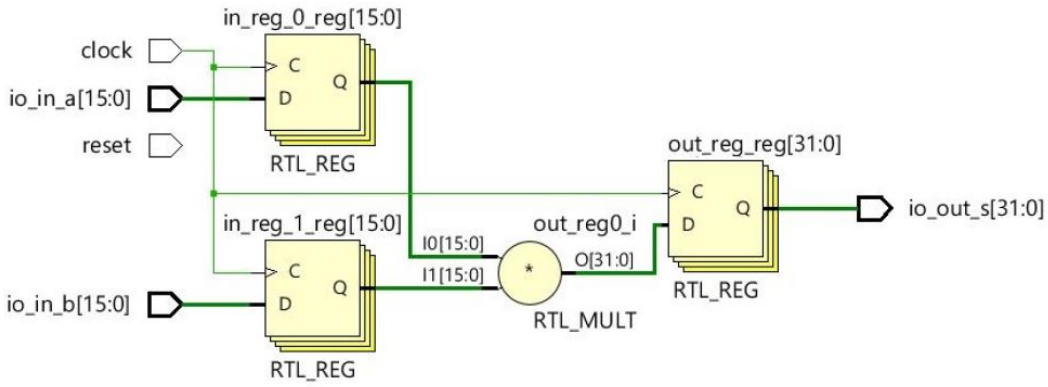

Figure 5.4: *16-bit floating point multiplier synthesis result.*

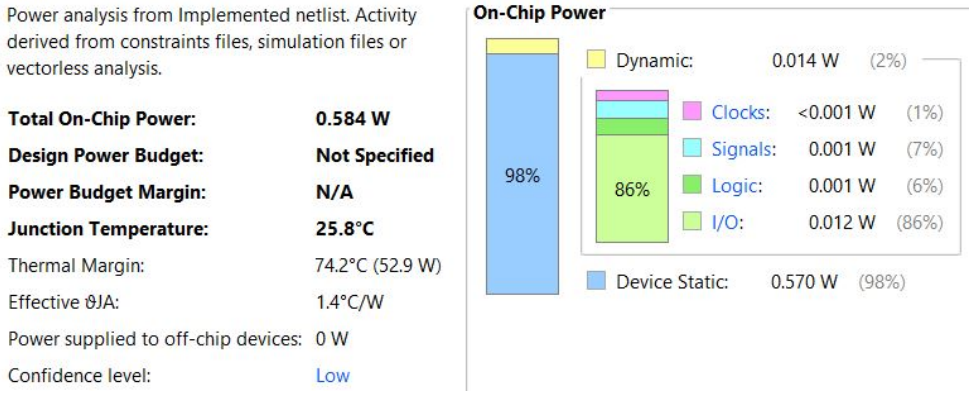

Figure 5.5: *16-bit Floating Point Multiplier Power Analysis.*

In Fig. 5.5, we can analyze the power implementation estimate. The total onchip power estimation is 0.583 W. The on-chip power distribution for static power is 2% while the dynamic power is the remaining 98%.

5.3 Floating Point Multiply and Sum Design

This circuit was designed to multiply and sum up a large series of multiplication results. It is parameterizable in the sense that we can specify the number of multiplications that the circuit can perform and sum up.

Suppose we want a circuit that performs N multiplications and sums up all the results. This means we will need to instantiate N FP-multipliers and N-1 FP-adders. The general structure of a N-FP multiply and add is similar to diagram shown in Fig. 5.6

When we have to sum up an odd number of results, the summation is a little tricky since the FP adders have two inputs, so we have to hold off on one addition until the end of the computation.

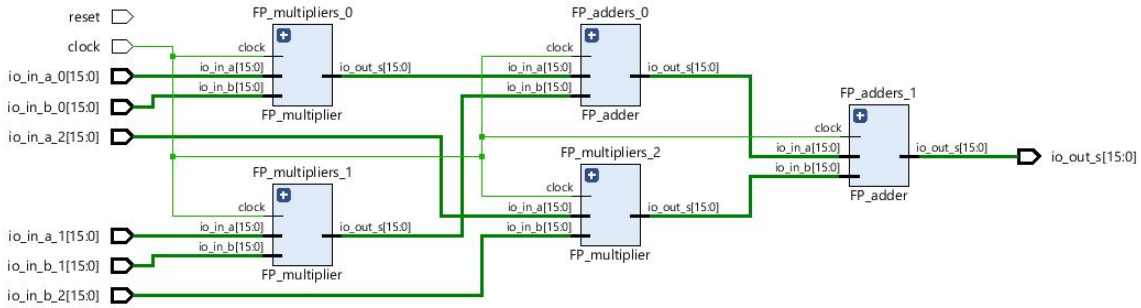

Figure 5.6: *General structure of N-FP Multiply-Add.*

5.4 Floating Point Accumulator

This circuit was designed to accumulate a sequence of inputs within a certain amount of clock cycles. The number, N, of accumulations that the circuit will perform must be specified through a parameter. The N accumulations will be performed within N clock cycles using N FP-adders and N registers. The circuit itself only has one input, so the input value will have to change every clock cycle in order to accumulate different values together. The structure of the circuit is shown in the Fig. 5.7.

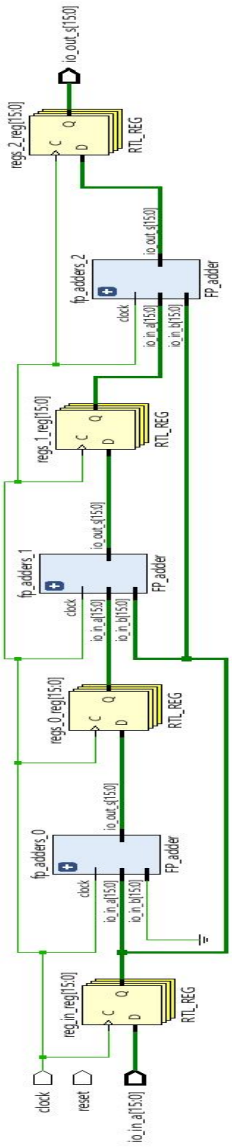

Figure 5.7: *General structure of N-FP Accumulator.*

CHAPTER VI

DESIGN OF NETWORK

6.1 Design on Parameterizable CNNs

This section discusses the idea of designing a parameterizable CNN with Chisel HCL. The IntelliJ software from JetBrains is employed as the developing tool using Scala plugins and Chisel HCL libraries. By parameterizing the proposed generator, the RTL design on the CNNs can be constructed. The design functionality is verified by a direct test from Chisel HCL, and the corner test cases are tested through RTL verification using Siemens ModelSim. After verification, the experimental results including the slice count, latency, and power cost are estimated with AMD Vivado.

6.2 Convolution Module Design

The convolution layer is the most computationally complex part of the CNN, which is used to extract features of the input matrices using kernel filters or convolution kernels. The specific goal is to convolve the input matrices with a kernel filter, weigh the summation of the convolution results, and then obtain the output feature map of this layer after processing it through an activation function.

To implement Equation 2.2 with code, four layers of embedded "for loop" are needed. Here, pseudo-codes as algorithm 1 are adopted to output 64 neurons from the convolution layer. An element-wise product between each element of the kernel and the input matrix is calculated at each location of the matrix and summed to obtain the final value in the corresponding position of the feature map. The convolution operation involves two-dimensional multiplication-addition calculations. An element-wise product between each element of the kernel filter and the input matrix

is calculated at each location of the matrix and summed to obtain the final value in the corresponding position of the output feature map.

Algorithm 1 Convolution layer computation

Input: $MatA - Input\ data$ of $image\ pixel$;
Input: $MatB - Kernel\ map\ array$
Output: $Conv\_out - Output\ feature\ map\ array$

procedure #1 - PERIODICALLY TRAVERSE THE ROWS OF INPUT:
  for $(i = 0; i \leq MatARowSize - 2; i = i + 1)$ **do**
    $count = 0$
    procedure #2 - PERIODICALLY TRAVERSE THE COLUMNS OF INPUT:
      for $(j = 0; j \leq MatARowSize - 2; j = j + 1)$ **do**
        procedure #3 - PERIODICALLY TRAVERSE THE ROWS OF KERNEL MAP:
          for $(n = 0; n \leq MatBRowSize; n = n + 1)$ **do**
            procedure #4 - PERIODICALLY TRAVERSE THE COLUMNS OF KERNEL MAP:
              for $(m = 0; m \leq MatBRowSize; m = m + 1)$ **do**
                $output = MatA[i * stridelength + j] * MatB[n + m]$
                $conv\_out = (output < 0)?\ 0 : output$

6.2.0.1 Streaming Design on Convolution Module

For the hardware implementation of the convolution layer, various methods have been proposed. As an example, [23] presented a conventional method by implementing convolution hardware with a 3×3 kernel circuit. The implementation made full use of parallelism for computing convolution results. The convolution window sliding was realized, and a convolution computation circuit of efficient parallel pipeline operation was formed. The downside of such implementations is that the design structures have to employ a fixed length of convolution.

In this thesis, therefore, a pipelined structure and reconfigurable convolution module is presented. By parameterizing the streaming width of the design, different convolution modules with Verilog HDL can be generated. As a case study, Fig. 6.1 shows

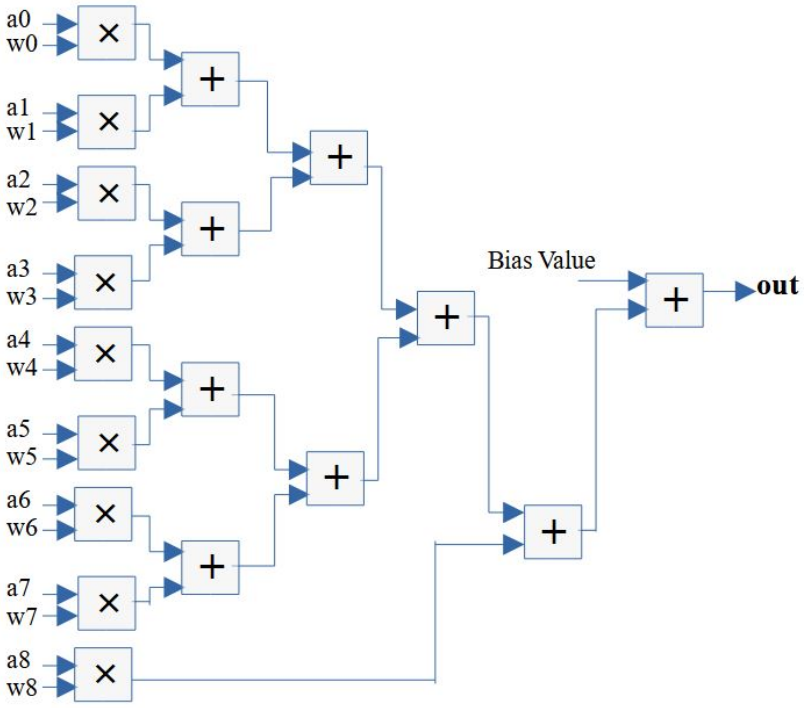

Figure 6.1: *Streaming Design Structure of a Convolutional Layer Neuron.*

the design module which takes a stream of nine inputs (3×3) for every clock cycle and calculates the convolution result within seven clock cycles, during which the following stream of inputs is taken. Specifically, the nine input data (denoted as "a0-a8") and weights (denoted as "w0-w8") are fed into the design engine in parallel, and then each FP operator including the FP multiplier and FP adder takes one clock cycle for the computation. As a result, the design structure utilizes nine FP multipliers and nine FP adders.

As the timing diagram shown in Fig. 6.2, in the first and second clock cycles two consecutive groups of inputs (denoted as "A0" and "A1") are fed into the engine, which is specified by an asserted "ready" signal to indicate the valid data input. Notice that "A0" and "A1" are two groups of inputs, and each of them includes all the nine inputs "a0" to "a8" in parallel. Their corresponding weights (denoted as

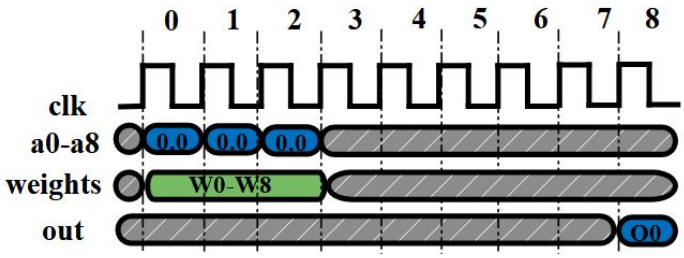

Figure 6.2: *Timing Diagram of Streaming Design of a Convolutional Layer Neuron.*

"W0" and "W1") are read out from a ROM and the multiplications ($A0 \times W0$ or $A1 \times W1$) are performed in the same clock cycles. In the following clock cycles two to six, the products will be pushed out and then cascading summed up together. Adding the bias takes one more clock cycle to push out the final groups of output (denoted as "O0" and "O1"), which are indicated by the asserted "vld" signal in the seventh and eighth clock cycles.

6.2.0.2 Parameterizable Streaming Width on Convolution Module

As mentioned earlier, the proposed work allows for the parameterization of the streaming width N, enabling the generation of Verilog designs with different resource costs and latencies. Generally, the latency of the streaming design structure can be calculated as follows:

$$Latency\_Streaming = 1 + 1 + roundup(\log_2 N) + 1 \tag{6.1}$$

where one cycle is needed for valid data fed into the engine, followed by an additional one cycle for the multiplications in parallel, followed by the logarithm result for the clock cycles needed by the cascading additions, and finally one clock cycle for the addition with the bias. The $roundup(x)$ functions round the x up to the integer if $\log_2 N$ is a fraction. For example, $N = 9$ so that $roundup(\log_2 9) = 4$ and the final

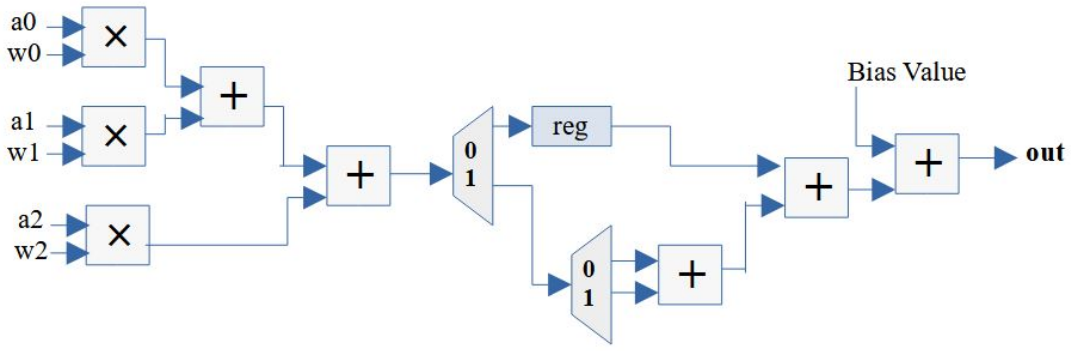

Figure 6.3: *Iterative Design Structure of a Convolutional Layer Neuron.*

estimated result is seven clock cycles. Similarly, the number of FP operators can be approximately estimated as N FP multipliers and N FP adders for the streaming design.

In order to reduce the number of FP operator utilization, the iterative design can be extended with less streaming width compared to the number of input data. For example, the number of data inputs is nine and the streaming width of the design is three. So one group of data should be divided into three data frames and then fed into the engine over three clock cycles. The design structure is shown in Fig. 6.3, including a register to delay one more clock cycle on the first data frame and two FP adders for accumulating the three summations from the three data frames. Finally, an FP adder is needed to add the bias to push out the final result. Notice that for the multiplication-addition design, it only takes three FP multipliers and two FP adders that can be reused by three data frames over multiple clock cycles. The latency of the iterative design can be calculated as

$$Latency\_Iterative = SW + roundup(\log_2 SW) + SW + 1 \tag{6.2}$$

where SW represents the parameter of the streaming width. The first SW indicates the clock cycles for feeding in the entire data group, the second SW shows

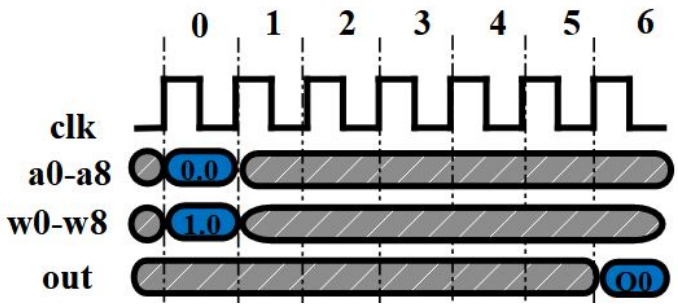

Figure 6.4: *Timing Diagram of Iterative Design of a Convolutional Layer Neuron*

the latency for accumulating the results from all the data frames, a final clock cycle is needed for the bias addition, and the $roundup(\log_2 SW)$ function is used for calculating the clock cycles for the cascading summation.

A specific example is shown in Fig. 6.4 including two groups of data. The first group is divided into three data frames denoted as "A00, A01, and A02" and the second group is composed of three data frames denoted as "A10, A11, and A12". Likewise, the corresponding weights are denoted as "W00, W01, W02" for the first group data, and "W10, W11, and W12" are for the second group data. For this example, $SW = 3$ so that the latency can be calculated as $3+roundup(\log_2 3)+3+1 = 9$ clock cycles to push out the final output, where three cycles are needed for valid data fed into the engine, two cycles for each multiplication-addition result, additional three cycles for accumulation of the three multiplication-addition results, and the final clock cycle for the bias value addition. Notice that the output can be pushed out over every three clock cycles in the pipeline since fewer resources are utilized for the convolution module.

6.3 Pooling Module

A pooling module provides a down-sampling operation that reduces the in-plane dimensions of the feature map. Unlike the convolutional layer, the pooling layers

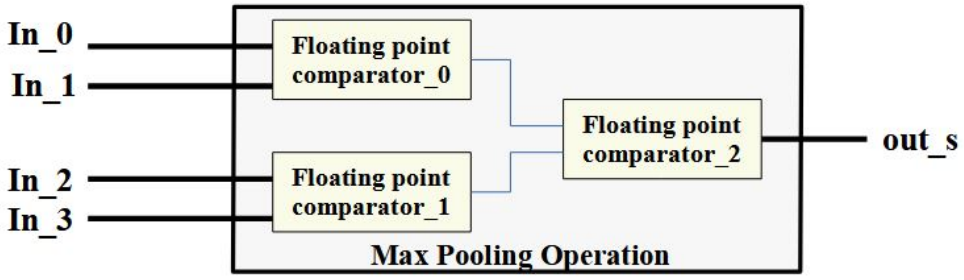

Figure 6.5: *Visual Representation of the Max Pooling using FP Comparators.*

do not have any learnable parameters; instead, filter size, stride, and padding are hyperparameters in the pooling operations. The construction of the pooling layer can be parameterized using either max pool or mean pool modules, which are implemented in this subsection.

6.3.0.1 Max Pool Module

The most popular form of the pooling operation is the max pool, which fetches patches from the feature maps, outputs the maximum value in the filter size in each patch, and discards the rest of the values in the patch. The general filter size used in a CNN is 2×2 with a stride of two.

Fig. 6.5 shows the structure of the max-pooling computation for a 2×2 pooling kernel. Using the FP operators, the final maximum value can be obtained by comparing all four elements. In the design of the pooling operation, the pseudo-codes adopted are shown as algorithm. 2. Implementing the max-pooling operation requires four layers of embedded "for loops". In this example, pseudo-codes are adopted to output 16 neurons from the pooling layer.

Algorithm 2 Max pooling computation

Input: $pool\_in\_buff[size\_A][size\_B]$: $Input\ feature\ map\ array$
Output: Max_out : $max\_out\_buff[size\_out]$ $Output\ feature\ map\ array$

procedure #1 - Periodically traverse the rows of input:
    $initialize : s(0) = p(0);$
    for $(i = 0; i \leq MatARowSize/2; i = i + 1)$ **do**
        $count = 0$
        procedure #2 - Periodically traverse the columns of input:
            for $(j = 0; j \leq MatARowSize/2; j = j + 1)$ **do**
                procedure #3 - Periodically traverse the rows of kernel map:
                    for $(n = 0; n \leq 2; n = n + 1)$ **do**
                        procedure #4 - Periodically traverse the columns of kernel map:
                            for $(m = 0; m \leq 2; m = m + 1)$ **do**
                                tmp1=pool_in_buff[i*stride + n];
                                tmp2=pool_in_buff[j*stride + m];
                                tmp3=pool_in_buff[i*stride + 1 + n];
                                tmp4=pool_in_buff[j*stride + 1 + m];
                                max1=(tmp1¿tmp2)?tmp1:tmp2;
                                max2=(tmp3¿tmp4)?tmp3:tmp4;
                                max=(max1¿max2)?max1:max2;
                                index = i*stride+j*stride+m+8*(in_matA_size/2+8);
                                max_out_buff[index]=max;

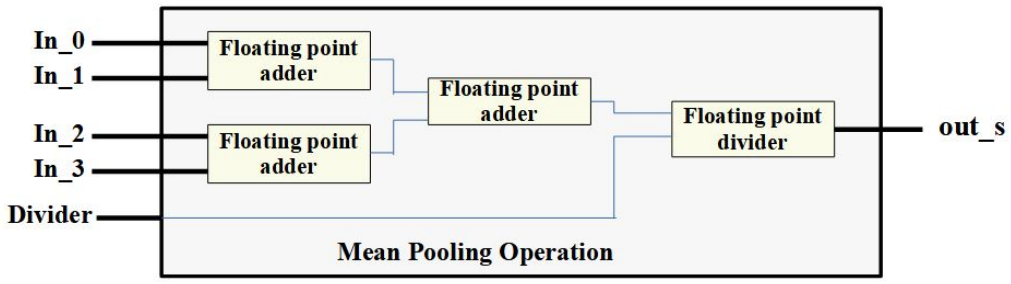

Figure 6.6: *Visual Representation of the Mean Pooling using FP Adders and Divider.*

6.3.0.2 Mean Pool Module

The mean pool technique is similar to the max pooling, where data elements are fetched from feature maps, and the output is the mean of all the elements in the fetched patch. The general filter size used in mean pooling is 2×2 with a stride of two. Fig. 6.6 shows the implementation of the mean pooling operation using three FP adders and one FP divider for a 2×2 patch.

In the design of the pooling operation, the pseudo-code adopted is shown as algorithm. 3. Implementing the mean pooling operation requires four layers of embedded "for loops". Similar to the algorithm shown in the design of the max pool module, pseudo-codes are adopted to output 16 neurons from the pooling layer.

From the computational design perspective, the key difference between the max pool layer and the mean pool layer is that the mean pooling operation requires an additional FP divider in the calculation of the patch mean. The downside of using mean pooling is that the pixel intensity is averaged across the region, whereas in max pooling, only the maximum intensity value is retained. Both max pool and mean pool layers can be parameterized and generated by our proposed work, providing flexibility for constructing complex neural networks.

Algorithm 3 Mean pooling computation

Input: $pool\_in\_buff[size\_A][size\_B]$: $Input\ feature\ map\ array$
Output: $Mean\_out$: $mean\_out\_buff[size\_out]$ $Output\ feature\ map\ array$

1: **procedure** #1 - Periodically traverse the rows of input:
2: $initialize : s(0) = p(0);$
3: **for** $(i = 0; i \leq MatARowSize/2; i = i + 1)$ **do**
4: $count = 0$
5: **procedure** #2 - Periodically traverse the columns of input:
6: **for** $(j = 0; j \leq MatARowSize/2; j = j + 1)$ **do**
7: **procedure** #3 - Periodically traverse the rows of kernel map:
8: **for** $(n = 0; n \leq 2; n = n + 1)$ **do**
9: **procedure** #4 - Periodically traverse the columns of kernel map:
10: **for** $(m = 0; m \leq 2; m = m + 1)$ **do**
11: tmp1=pool_in_buff[i*stride + n];
12: tmp2=pool_in_buff[j*stride + m];
13: tmp3=pool_in_buff[i*stride + 1 + n];
14: tmp4=pool_in_buff[j*stride + 1 + m];
15: avg=(tmp1+tmp2+tmp3+tmp4)/4;
16: index = i*stride+j*stride+m+8*(in_matA_size+8);
17: mean_out_buff[index]=avg;

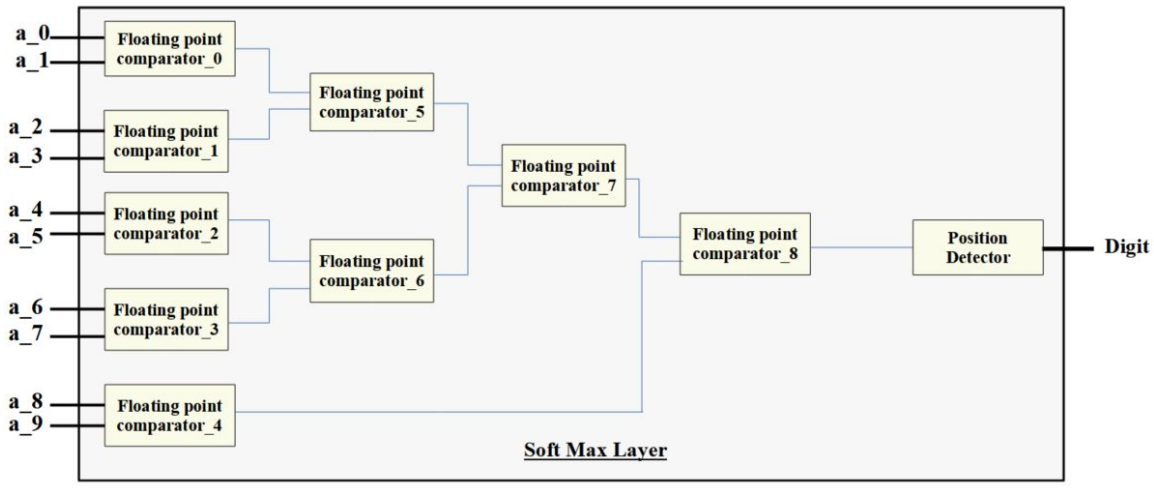

Figure 6.7: *Visual Representation of the Soft-Max Layer using FP Comparators.*

6.4 Fully Connected Module and Soft Max Module

After the final pooling operation, the outputs from the feature maps are flattened into a one-dimensional array of numbers and connected to one or more fully connected layers, each of which has learnable weights that enable multiplication-addition operations between every input and output. The clock cycle operations for these layers are similar to those shown in Fig. 6.1. The implementation algorithm for the fully connected layer is similar that shown in algorithm. 1, but with different weight and bias values.

The softmax module uses FP comparators to determine the maximum value among the ten output neurons. The design circuit shown in Fig. 6.7 is then used to identify the position of the neuron with the highest value.

6.5 System Construction and Static Analysis

In this section, the system-level integration and construction are further discussed. This project demonstrates the validation of the proposed parameterizable designs on

Algorithm 4 Soft max layer computation

Input: $soft\_in\_buff[10] : Input\ feature\ map\ array$
Output: $Digit : Digit\_buff : Output\ feature\ map\ array$

```
1: procedure #1 - Periodically traverse the rows of input:
2:     initialize : s(0) = p(0);
3:     for (i = 0; i ≤10; i = i + 1) do
4:         if soft_in_buff[i] ≤ max then
5:             max = soft_in_buff[i]
6:             Digit = i
```

CNNs. The methodology can be extended into different complex neural networks by constructing neurons with parameterizable submodules such as FP adders and multipliers, and further integrating the neurons into multi-layer networks.

6.6 System Design Construction

In order to construct different networks with different precision, a configurable design on the CNN is demonstrated in this section. Specifically, the network architecture can be parameterized and generated to provide different design precision including half-word, word, double-word, and quad-word. Additionally, this thesis presents a parameterizable pooling design for choosing max-pool or min-pool operation in the generated CNN architecture. Finally, the streaming structure and iterative design of the convolution and fully-connected layers are further presented. Based on the size of the local receptive field, the input parameter to the convolution is decided. Similarly, the parameters for the size of the register map are calculated upon the local receptive field and the number of computational outputs.

We follow a layered-based design style, generating Verilog HDL code for individual layers based on the number of hidden layers required for a particular application. As a case study, we employ the FP design library to build the CNN [4]. After parameterizing the precision of each FP operator and finding the design structure of

each layer, the network can be constructed into a sequential system including register maps between different layers of data streams or feature maps. Fig. 6.8 shows the register maps generated for various data widths during the generation of RTL designs.

Specifically, the proposed network takes 10×10 as input to the convolution layer, which generates 64 neurons as the first result of the convolution operation using a 3×3 kernel filter. The convolution layer uses the ReLU activation function, defined as $y = max(x, 0)$, and the window slide step is one. Using our proposed work, the streaming width of the convolutional layer is parameterizable so that convolution operation may take multiple clock cycles to calculate the results of the output neuron. As an example shown in this figure, the streaming design of nine inputs is configured to the implementation on the convolution layer and the results are stored in the register map before being fed into the subsequent pooling layer.

In what follows, the 64 neurons are further downsized through the pooling operation, resulting in 16 neurons using a 2×2 region map for performing the max pooling operation. The pooling layer is configurable depending on the design specifications and by default, uses max pooling with a window slide step of two. The 16-output neurons from the pooling layer are connected to the inputs of the fully connected layer. A minimum of 10-output neurons are required for the final data classification. Hence, we take $16 \times 10 = 160$ weights and ten bias values for calculating a fully-connected layer. The final classification result is the corresponding number of neurons with the highest value output.

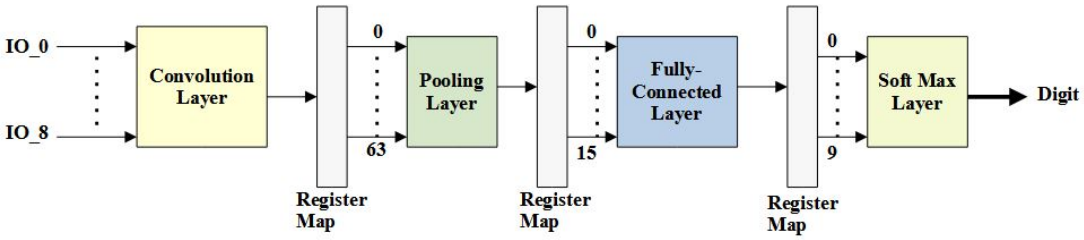

Figure 6.8: *System Construction on CNNs.*

6.7 Static Analysis of Hardware Cost

This subsection analyzes the hardware cost of different parameterizable designs presented in this thesis, as shown in Table 6.1. In the second row, the convolutional layer design is parameterized with a streaming width of nine, and the max pooling module is selected when generating the Verilog code. The implementation with a streaming width of nine requires 45 FP multipliers (FP_MUL), 45 FP adders (FP_ADD), and 57 FP comparators (FP_COMP), and the simulation takes 100 clock cycles to classify the 10×10 inputs. Using the streaming width of three, the design in the third row shows a reduction in the utilization of FP modules but takes $2\times$ clock cycles for data classification.

Compared with the max pool operation, the mean pool operation only utilizes nine FP comparators but much more FP adders. Similar to the difference between streaming widths of nine and three for the max pool operation, the mean pool module with a streaming width of nine takes much more resource cost in terms of FP multipliers and FP adders but spends fewer clock cycles compared with that of a streaming width of three.

Between the max and mean pool design, the mean pool operation requires FP dividers (FP_DIV) to calculate the average of the selected region, as well as more FP adders to sum the inputs. Since the resource cost of the FP adders and dividers is

Layer	FP_MUL	FP_ADD	FP_DIV	FP_COMP	Latency
SW=9, Max	45	45	0	57	100
SW=3, Max	39	41	0	57	200
SW=9, Mean	45	93	16	9	103
SW=3, Mean	39	89	16	9	203

Table 6.1: *Static Analysis of Different Design Structures*

much higher than that of the FP compactors, the mean pool design will spend more LUTs and registers than the design of the max pool structure. Therefore, the default configuration is the max pool for our proposed work. The latency difference of the additional three clock cycles between the max and mean pool operation-based design is due to the additional three FP adder modules in the mean pool-based design.

CHAPTER VII

EXPERIMENTAL RESULTS

The static analysis in the previous section shows the difference between design structures. In this section, the practical hardware performance is further estimated with AMD Vivado in terms of quality of results, slice count, and energy cost. Vivado 2019.2 is applied as the synthesis tool with the FPGA target device Zynq UltraScale+ MPSoCs xczu19eg-ffvb1517-3-e.

7.1 Experiment Design

The experimental setup is based on random FP input numbers fed as input to the convolutional layer. These random inputs are generated by scala built-in random functions during the generation of RTL code. The hardware results after the layer are displayed on the IntelliJ IDE. In our case study, we have considered our test matrix size to be 10×10 which generated a 100 random pixel intensity, and a 3×3 local receptive field which is hard coded onto the circuit. The outputs of each layer are captured and compared to the golden results generated by software implemented CNN using Scala language. For accuracy, multiple test results were captured for different inputs and compared with the software results.

7.2 Resource cost on FPGA

After synthesis with Vivado, the hardware utilization for different designs with different precision is shown in Table 7.1. It can be observed that the higher resource utilization comes from the network with a higher bit width. Specifically, the 128-bit design uses more than four million LUTs and about 27 thousand registers. However, the 16-bit design only takes 9,808 LUTs and 3,250 registers, which is much less than higher precision implementations. The design with less precision usually obtains less

Precision	CLB LUTs	CLB Registers	CARRY8	DSPs	Accuracy
16-bit	9,808	3,250	228	57	97.07
32-bit	25,848	2,677	816	114	98.39
64-bit	86,802	5,436	2,943	513	99.25
128-bit	4,24,723	26,955	42,369	1,959	99.76

Table 7.1: *Resource Utilization of Different Precision Designs*

accuracy when classifying images or videos. As an example shown in the last column, the 128-bit design can achieve the highest accuracy and the 16-bit design obtains an accuracy with about a two percent reduction.

It concludes that a large number of resources can be saved by using low-precision designs with a slight reduction in the quality of the results. Notice that the comparison in this case study is to feed in the random matrix and monitor the output between different designs. For estimating the accuracy in real applications such as digit recognition, more error percentages would be involved by using low-precision implementations.

The number of clock cycles required for computation remains constant in spite of the increase in precision due to parallelism. The generated RTL uses a significant number of hardware resources to achieve high parallelism. The circuit consists of multiple instantiations of modules which also increases energy consumption as discussed in the next section.

7.3 Energy Consumption on FPGA

In what follows, the energy dissipation is summarized in Table. 7.2. As a result, the total on-chip energy for the 16-bit design is 0.30 mJ, including 0.28 mJ dynamic energy and 0.02 mJ static energy. As with the analysis of resource utilization, the energy dissipation for higher precision designs is significantly higher than the im-

	16-bit	32-bit	64-bit	128-bit
Energy (mJ)	0.28	1.044	2.462	7.867

Table 7.2: *Energy Consumption of Different Precision Designs*

plementations with lower precision. The highest energy cost is 7.867 mJ which is obtained by the 128-bit design.

7.4 Hardware Cost Analysis

Our proposed work can be used to evaluate the FPGA cost in combined error percent-area-energy with a simple equation

$$cost = (P \times x) + (S \times y) + E \times (1 - x - y). \tag{7.1}$$

, where P, S, and E represent the normalized values of error percentage, slice count, and energy consumption, respectively. Additionally, x, y, and $(1 - x - y)$ represent the weights of the three design specifications. Specifically, x and y are between 0 and 1.0, thus the summation of all three weights would be 1.0. By configuring the three design specifications x, y, and $1 - x - y$, the parameterizable design can regulate the design features and target one of the design performances. For example, setting x and y as $1/3$ will lead to equal weighting, and $y = 0$ would target a design constraint with a low slice count. In Fig. 7.1, the normalized error-area-energy design cost is summarized. When setting $y = 1.0$ depicted in Fig. 7.1(a), it can be observed that the 16-bit design uses the lowest hardware resource compared to other designs with higher precision. The normalized hardware cost for 16-bit and 32-bit designs are similar, and the 128-bit design spends around 26× FPGA resources when compared to the 32-bit implementation.

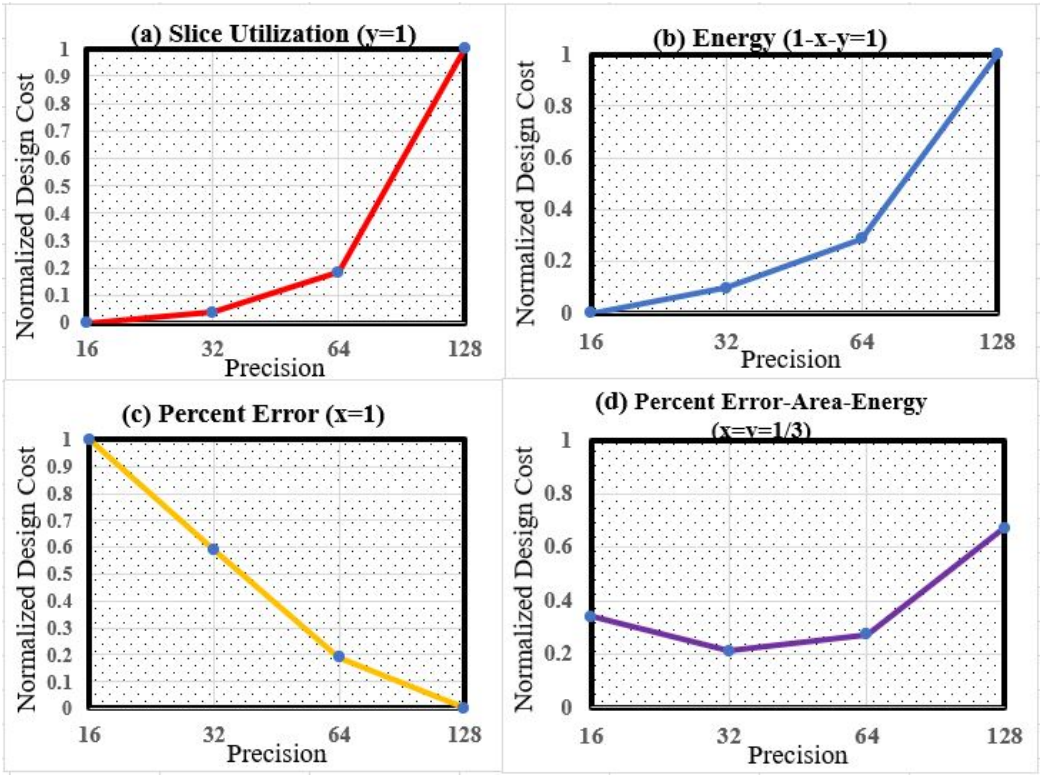

Figure 7.1: *Normalized Accuracy-Area-Energy Consumption.*

7.5 Comparison of related work

As an implementation with minimum error-slice-energy cost within all the generated designs, the 32-bit design is applied to compare with other prior works in this section. As shown in Table 7.3, the hardware cost is summarized in terms of LUTs, FFs, and DSPs. In the last column, the accuracy is further compared between different works.

The number of LUTs and DSPs used in our case study is less than a half compared to [18] and [19] because of the pipelined structure and three hidden layers used to build the network architecture. The number of convolutional layers and hidden layers used for implementing CNNs in articles [18] and [19] are much larger compared to our proposed work. This addition of layers in the network increases the accuracy but utilizes more hardware resources. In contrast, the register count of our design is slightly more than [18] because of the use of a register map to store the values of hidden neurons, which increases the parallelism to execute the computation.

In [20], the design uses a block floating point method of implementation, which consumes higher hardware costs. In paper [21], the authors have designed the CNN structure based on the sigmoid activation function, whereas, in this thesis, the design is based on the ReLU activation function which is computationally less complex and hence uses fewer hardware resources.

While the LUTs in [22] and [23] are less than half when compared to our proposed work, [22] uses resource multiplexing, wherein the modules are reused for different operations cycles, which tends to higher latency. In [22], as an example, digit recognition takes 68,139 clock cycles. In reference [23], the implementation does not follow resource multiplexing, however, it takes much more registers to buffer data between different layers, and more DSPs are needed for the arithmetic operations.

Comparison	LUTs	FFs	DSPs	Accuracy(%)	Quantization Strategy
[18]	55,466	2,493	1645	99.17	32-bit floating
[19]	186,251	205,704	2240	-	32-bit floating
[20]	231,761	14,091	1,027	-	8-bit Block floating point
[21]	34K	-	-	89	8-bit fixed
[22]	10,208	8,204	17	97.3	16-bit fixed
[23]	12,588	48,765	274	97.57	18-bit fixed
Our case study	25,848	2,677	114	98.39	32-bit floating

Table 7.3: *Comparision with Prior Works*

The accuracy shown in the fifth column demonstrates that our proposed 32-bit design achieves higher accuracy than that of the implementations with lower precision, including the 8-bit fixed number implementation in [20], 16-bit fixed design in [22], and the 18-bit design in [23]. The design with more resource cost in [18] achieves higher accuracy compared with that of our proposed architecture, which obtains an overall accuracy of 98.39%.

CHAPTER VIII

CONCLUSIONS AND FUTURE WORK

In this chapter, we summarize our contributions presented in this thesis. We then discuss the possible directions for our future research work.

8.1 Summary

This thesis presents a parameterizable design for CNNs that can accommodate various design structures and precision levels. Our proposed approach is scalable, allowing for the creation of RTL designs that meet different design specifications. The resulting Verilog code is both synthesizable and implemented in an FPGA demonstration. As a case study, a three-hidden-layer design structure is ultimately implemented and evaluated on an FPGA, achieving an overall accuracy of 98.39% for 32-bit FP precision. Experimental results demonstrate that our design has lower hardware costs than many existing approaches while still achieving reliable accuracy.

8.2 Future Work

Parameterizablity is one of the main advantages and motivations of using the Chisel-based design. In this thesis, a CNN network is implemented with designs on the floating-point fundamental arithmetic modules to show the validity of the research work. New ways of performing a variety of computing functions were expected to advance to the next generation of computer architectures and approaches. Future works will be focusing on parameterizing the network for handwritten digit recognition on FPGA. FPGAs offer a means to produce customized accelerators that can be reprogrammed in the field as needed for diverse applications. Chisel can also be developed to for verification purpose replacing Verilog HDL and SystemVerilog.

BIBLIOGRAPHY

[1] O. Choudhari, M. Chopade, S. Chopde, S. Dabhadkar, and V. Ingale, Hardware Accelerator: Implementation of CNN on FPGA for Digit Recognition. *2020 24th International Symposium on VLSI Design and Test (VDAT)*, 2020, PP.1-6.

[2] F. U. D. Farrukh, T. Xie, C. Zhang, and Z. Wang, Optimization for Efficient Hardware Implementation of CNN on FPGA. *2018 IEEE International Conference on Integrated Circuits, Technologies and Applications (ICTA)*, 2018, PP. 88-89.

[3] W. Xie, C. Zhang, Y. Zhang, C. Hu, H. Jiang, and Z. Wang, An Energy-Efficient FPGA-Based Embedded System for CNN Application. *Proceedings of the 14th IEEE International Conference on Electron Devices and Solid State Circuits (EDSSC)*, 2018, PP. 1-2.

[4] V. Mario, M. C. Madineni, B. Garrett, X. Yang and H. Xu, Case Studies of Configurable Binary Design Library on FPGA. *2022 International Symposium on Measurement and Control in Robotics (ISMCR), Houston, TX, USA,* 2022, pp. 1-5, doi: 10.1109/ISMCR56534.2022.9950580.

[5] P. Molchanov, A. Mallya, S. Tyree, I. Frosio, and J. Kautz, Importance Estimation for Neural Network Pruning. *In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, PP. 11264–11272.

[6] C. Son, S. Park, J. Lee, and J. Paik, Deep Learning-based Number Detection and Recognition for Gas Meter Reading.*IEIE Trans Smart Process Comput.*, 2019, Vol. 8, No. 5, PP. 367–372.

[7] K. Guo, S. Zeng, J. Yu, Y. Wang, and Huazhong Yang, A Survey of FPGA-Based Neural Network Inference Accelerator. *ACM Trans. Recon€g. Technol. Syst.*, 2017, Vol. 9, No. 4, Article 11.

[8] D. Gschwend, ZynqNet: An FPGA-accelerated Embedded Convolutional Neural Network, 2020, arXiv:2005.06892.

[9] C. Gao, S. Braun, I. Kiselev, J. Anumula, T. Delbruck, and S. Liu, Real-time Speech Recognition for IoT Purpose using a Delta Recurrent Neural Network accelerator. *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2019, PP. 1–5.

[10] K. Vaca, A. Gajjar, and X. Yang, Real-time Automatic Music Transcription (AMT) with Zync FPGA. *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2019, PP. 378–384.

[11] Q. Li, X. Zhang, J. Xiong, W. Hwu, and D. Chen, Implementing neural machine translation with bidirectional GRU and attention mechanism on FPGAs using HLS. *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, 2019, PP. 693–698.

[12] M. Cho and Y. Kim, Implementation of Data-optimized FPGA-based Accelerator for Convolutional Neural Network. *International Conference on Electronics, Information, and Communication (ICEIC)*, 2020, PP. 1–2

[13] H. Madadum and Y. Becerikli, FPGA-based Optimized Convolutional Neural Network Framework for Handwritten Digit Recognition. *1st International Informatics and Software Engineering Conference (UBMYK)*, 2019, PP. 1–6

[14] J. Si, E. Yfantis, and S. L. Harris, A SS-CNN on an FPGA for Handwritten Digit Recognition. *2019 IEEE 10th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, 2019, PP. 0088-0093.

[15] V. M. Milovanović and M. L. Petrović, A Highly Parametrizable Chisel HCL Generator of Single-Path Delay Feedback FFT Processors. *2019 IEEE 31st International Conference on Microelectronics (MIEL)*, 2019, PP. 247-250.

[16] M. Imani, D. Peroni and T. Rosing, CFPU: Configurable Floating Point Multiplier for Energy-efficient Computing. *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2017, PP. 1-6.

[17] J. Bachrach, et al., Chisel: Constructing Hardware in a Scala Embedded Language. *49th ACM/IEEE Design Automation Conference (DAC)*, 2012, PP. 1212-1221.

[18] Z. Li et al., Laius: An 8-Bit Fixed-Point CNN Hardware Inference Engine. *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*, 2017, PP. 143-150.

[19] C. Zhang et al., Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks, *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2015, PP. 161-170.

[20] X. Lian, Z. Liu, Z. Song, J. Dai, W. Zhou and X. Ji, High-Performance FPGA-Based CNN Accelerator With Block-Floating-Point Arithmetic. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 27(8)*, 2019, PP. 1874–1885.

[21] J. Si and S. L. Harris, Handwritten Digit Recognition System on an FPGA. *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*, 2018, PP. 402-407.

[22] Yan, F., Zhang, Z., Liu, Y., and Liu, J. Design of Convolutional Neural Network Processor Based on FPGA Resource Multiplexing Architecture. *Sensors 2022*, 2019, No. 22, PP. 59-67.

[23] R. Xiao, J. Shi and C. Zhang, FPGA Implementation of CNN for Handwritten Digit Recognition. *2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, 2020; PP. 1128-1133.

[24] L. Hui, Convolutional Neural Network Research and FPGA Implementation for Handwritten Digit Recognition. *Xi'an Shiyou University: Xi'an, China*, 2021

[25] A. L. Reed, X. Yang and S. Sha, Lightweight Neural Network Architectures for Resource-Limited Devices. *2022 23rd International Symposium on Quality Electronic Design (ISQED)*, 2022, PP. 1-7.

[26] I. Westby, X. Yang , T. Liu, and H. Xu, Exploring FPGA Acceleration on a Multi-Layer Perceptron Neural Network for Digit Recognition. *The Journal of Supercomputing (JSC)*, 2021, PP. 1-18.

[27] Chisel/FIRRTL Hardware Compiler Framework, [Online]. *Available: https://www.chisel-lang.org/*

[28] P. Lennon and R. Gahan, "A Comparative Study of Chisel for FPGA Design," *2018 29th Irish Signals and Systems Conference (ISSC 2018)*, PP. 1-6, 2018. doi: 10.1109/ISSC.2018.8585292.

[29] H. He, X. Yang , L. Wu, and Y. Feng, Iterated Dilated Convolutional Neural Networks for Word Segmentation. *Neural Network World (NNW)*, 2020, Vol. 30, No. 5, PP. 333-346.

[30] L. Di Tucci, D. Conficconi, A. Comodi, S. Hofmeyr, D. Donofrio, and M. D. Santambrogio, A Parallel, Energy Efficient Hardware Architecture for the merAligner on FPGA Using Chisel HCL. *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, PP. 214-217.

[31] K. Abdollah, M. Dabbaghjamanesh, T. Jin, W. Su, and M. Roustaei. An evolutionary deep learning-based anomaly detection model for securing vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 2020, Vol. 22, No. 7 PP. 4478-4486.

[32] S. Ramin, A. Sahba, and F. Sahba. Using a combination of LiDAR, RADAR, and image data for 3D object detection in autonomous vehicles. *In 2020 11th IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, 2020, PP. 0427-0431.

[33] D. Morteza, A. Moeini, and A. Kavousi-Fard. Reinforcement learning-based load forecasting of electric vehicle charging station using Q-learning technique. *IEEE Transactions on Industrial Informatics*, 2020, Vol. 17, No. 6, PP. 4229-4237.

[34] J., Mina, A. Kavousi-Fard, M. Dabbaghjamanesh, and M. Karimi. A Survey on Deep Learning Role in Distribution Automation System: A New Collaborative Learning-to-Learning (L2L) Concept., *IEEE Access*, 2022.

[35] F. Vasta, et. al. Reproductive Outcomes and Fertility Preservation Strategies in Women with Malignant Ovarian Germ Cell Tumors after Fertility Sparing Surgery. *Biomedicines,* 2020, Vol. 8, Issue 12.

[36] A. Vimercati, et. al. Ultrasonic assessment of cesarean section scar to vesicovaginal fold distance: an instrument to estimate pre-labor uterine rupture risk. *The Journal of Maternal-Fetal & Neonatal Medicine*, 2020, Vol. 35, Issue 22, PP. 4370-4374.

[37] S. Amin, R. Sahba, P. Rad, and M. Jamshidi. Optimized IoT based decision making for autonomous vehicles in intersections. *In 2019 IEEE 10th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, 2019, PP. 0203-0206.

[38] K. Vaca and et al. An Open Real-Time Audio Processing Platform on Zync FPGA. *Intl. Symposium on Measurement and Control in Robotics (ISMCR 2019)*, 2019, PP. D1-2-1-D1-2-6.

[39] Y. Zhang and et al. A Case Study On Approximate FPGA Design With an Open-Source Image Processing Platform. *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2019)*, 2019, PP. 372-377.

[40] I. Westby and et al. A Design on Multilayer Perceptron (MLP) Neural Network for Digit Recognition. *Advances in Artificial Intelligence and Applied Cognitive Computing. Transactions on Computational Science and Computational Intelligence*, 2020.

VITA

Mukesh Chowdary Madineni

2019	B.E., Electronics and Communication Engineering People's Education Society Institute of Technology Bengaluru, India
2023	M.S., Computer Engineering University of Houston-Clear Lake Houston, TX

PUBLICATIONS

M. C. Madineni, V. Mario and X. Yang. 2023. "Parameterizable Design on Convolutional Neural Networks Using Chisel Hardware Construction Language," Micromachines 14, no. 3: 531. https://doi.org/10.3390/mi14030531.

V. Mario, **M. C. Madineni**, B. Garrett, X. Yang and H. Xu, "Case Studies of Configurable Binary Design Library on FPGA," 2022 International Symposium on Measurement and Control in Robotics (ISMCR), Houston, TX, USA, 2022, pp. 1-5, doi: 10.1109/ISMCR56534.2022.9950580.