UNIVERSITÄT SIEGEN

Department Elektrotechnik und Informatik

Digitale Integrierte Systeme

Master Thesis

im Studiengang Informatik

Implementation of a small scale AI supporting architecture on an FPGA

Student: Ghazanfer Hussain

Matriculation number: 1537279

First examiner: Dr.Ing. Michael Wahl

Second examiner: M.Sc Ferid Mahdi

Submission date: February 2024

Statutory declaration

I affirm that I have written my thesis (in the case of a group thesis, my appropriately marked part of the thesis) independently and that I have not used any sources or aids other than those indicated, and that I have clearly indicated citations.

All passages that are taken from other works in terms of wording or meaning (including translations) have been clearly marked as borrowed in each individual case, with precise indication of the source (including the World Wide Web and other electronic data collections). This also applies to attached drawings, pictorial representations, sketches and the like. I take note that the proven omission of the indication of origin will be considered as attempted deception.

SIEGEN 28 February 2024 | [signature]

Location, Date | Signature

Statutory Declaration

I hereby certify, Ghazanfer Hussain, that I have written this thesis with the topic Implementation of a small scale AI supporting architecture on an FPGA independently and only using the sources and resources specified.

I further declare that this work has not yet been submitted as part of another examination procedure.

Place, Date	Unterschrift
Siegen, Wednesday 28th February, 2024	G Hussain

Contents

1. Introduction

Machine learning has become part of today's contemporary world. There are numerous applications that use machine learning techniques to simplify life for people. Examples of these applications are autonomous driving, health monitoring, speech recognition, motion detection, weather prediction, and so on and so forth. The chatGPT, which was trained on a large dataset, is an excellent illustration of a machine learning and natural language processing model. Given its close relationship to hardware, the concept of employing an edge devices dataset holds great potential. In the automotive sector, much research is underway to identify defects using sensor data to improve performance and make decisions based on the data at hand. It is possible to get off with the system failing altogether because fault detection is closely related to efficiently scheduling maintenance. The car contains a variety of sensor, cameras data types, and the datasets themselves are diverse. A clean dataset is unavoidable since it includes inaccurate and missing values; therefore, data cleaning and analysis are required to make the dataset suitable for machine learning. In addition to detecting potential outliers, machine learning models trained on clean datasets can also assist with failure, performance, and maintenance. Utilizing time and resources effectively is essential for these systems since it directly affects system dependability and availability requirement. According to studies, FPGAs can perform better than software-based deep neural network (DNN) implementations because of their concurrent-friendly paradigm [10].

Implementing the artificial intelligence algorithm on FPGA is the primary objective of this research. A crucial initial step is reading the article's literature review, which is previously researched. Many architectures, including (k-nearest neighbor)KNN, Navie base (NB), and Neural network (NN), are available that employ medical data (EEG, ECG) for patient health monitoring. The other methods are approximation computing, quantom computing, memristor technology, and high-dimensional computing. The neural network was chosen for implementation due to its significant potential for resolving real-world problems. In the modern world, neural networks come into play in a wide range of applications to gain an understanding of datasets and make decisions based on them. Software understanding of neural networks is a necessary step; it establishes the foundation for hardware implementation by explaining how neurons behave, how layers operate ,and the impact of the activation function. Steps for hardware implementation were carried out after acquiring an awareness of software implementation. Since an understanding of Verilog is required for FPGA implementation, hardware and software implementation are entirely distinct dimensions. Hardware implementation was completed using the Zynet package by compiling weight and bias values from TensorFlow software implementation. The MNIST dataset was utilized to generate footprints since it is a large, extensively investigated dataset that is often used for machine learning tasks. To prevent the overfitting

phenomenon, big datasets are always used for machine learning applications. Since the MINST dataset is rather clean, there is not much need for data cleaning. The network is expanded with varying numbers of neurons and layers, and the architecture that provides the highest accuracy was preferred. Following a thorough analysis of footprint production using the MNIST dataset, the 6 D sensor dataset was evaluated. We collected the 6 D sensor data ourselves, thus it is not clean. The setup for gathering the information was quite uncomplicated; for example, a 6 D sensor was attached to Arduino hardware, which was then connected to a computer. The data it collects will be stored in a CSV file, and include timestep, three-dimensional acceleration, three-dimensional gyroscope, and temperature values. As with all sensor data, it is full of illegible and numerous missing values. Therefore, before the dataset is used to train the algorithm, it needs to be cleaned to get rid of missing values. Initially, the network was trained on outlier values, and its performance on test data was unsatisfactory. Following that, the interquartile range (IQR) technique was used to remove outlier values, and the mean value of the specific column was used to fill in any missing data. The network was trained using two distinct architectures, acquired weight and bias information, and then produced a footprint based on these data. The resource usage of the algorithm was estimated using the Zybo board, but it can be altered to any other board. In this way, the goals of determining which algorithms can be implemented in hardware, what resources are required, and how the algorithm is implemented in FPGA were accomplished. In order to produce an effective and adaptable end-edge system, this research strategy blends deep learning approaches with hardware design based on software implementation. It is trivial to construct a footprint based on a dataset with the aid of the methodology presented in this thesis.

The paper is structured as follows: chapter 2 describes several examples of cutting-edge architecture from literature. Subsequently, an table of the previously mentioned architecture is shown. The working environment and technologies required to put the neural network into practice are covered in the chapter 3,4. The hardware design implementation example makes use of the thoroughly investigated MNIST dataset discusssed in chapter 5. Finally, it concentrates on the data analytics of the City Siegen dataset, and generate footprint on 6-D sensor data in chapter 6.

2. State of the Art

2.1. Quantum Dot Cellular Automata(QCA) [8]

Energy-efficient and rapid accelerators are required for applications like computer vision and speech recognition which use Convolutional Neural Networks (CNN) and Deep Neural Networks. The multiply and accumulate(MAC) and Static Random Memory(SRAM) are vital building blocks for AI accelerators. The traditional CMOS technology has drawbacks [2], such as severe process variation, GHz frequency limit, increased leakages, reduced control over the gate, and high power densities. Due to electrostatic force, two electrons fill a diagonal position in a quantum-dot cell, as shown in the picture 1. Coulombic repulsion force is responsible for signal propagation between electrons and adjacent cells that result in QCA wire, as shown in the figure 2(a). The inverter and the majority gate, which are constructed by joining several QCA cells, are of the vital gates for QCA(eq.1), as seen in the picture 2 (b,c).

Figure 1: Quantum Dot Cell Polarisation (a) Binary 1, Polarisation 1, (b) Binary 0, Polarisation -1

$$MV(A, B, C) = AB + BC + AC \tag{1}$$

The gate is a complete set of QCA technology since it can operate with OR or AND functions and have inputs that may be adjusted to 0 or 1. Clocking is necessary to preserve cell logic and to govern the direction of signal transmission. Compared to CMOS, QCA clocking employs reversible logic and has four phases(switch, hold, release, relax(figure3(a)) to conserve energy. In the switching phase, cells have a low potential barrier as they are ready to be polarized, and in the hold phase, the cells have high potential barriers, and can not alter polarization. The cells depolarize during the release phase and remain non-polarized until the subsequent switch. To preserve information only flowing in the forward direction, QCA cells divide into four zones, and each zone phase shifts from the previous zone by 90 degrees. As a result of this hierarchy, when one cell is in the switching phase, the preceding cell is serving as the input while in the hold phase,as represented in the figure3(b). The graphic 4 shows the USE clocking scheme's zones 1-4 and the information flow between them, which is shown by an arrow. The USE clocking method has several benefits, including simplicity in manufacturing, avoidance of

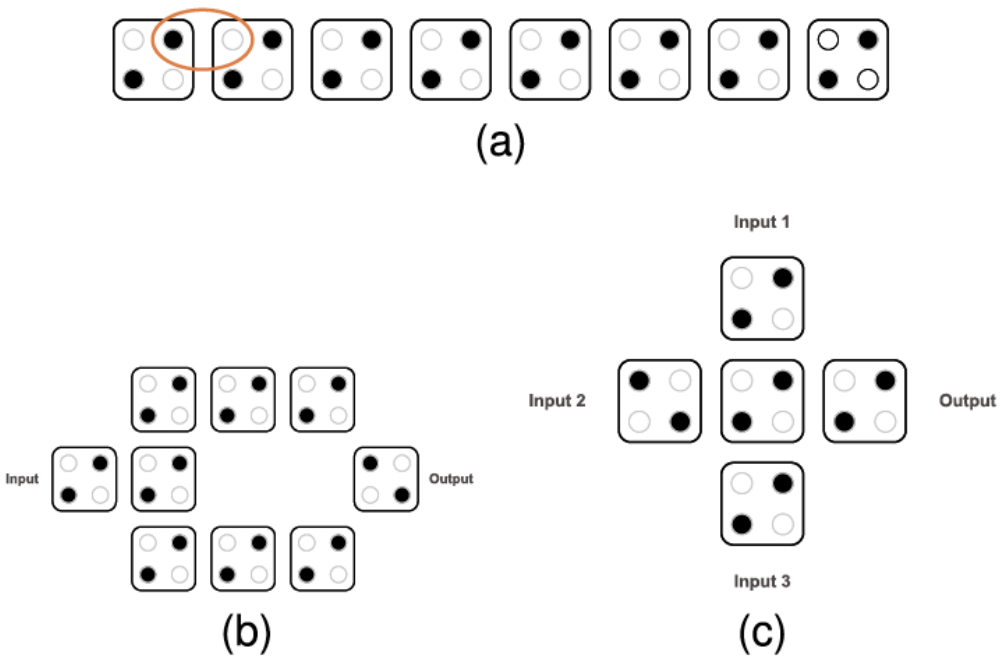


Figure 2: (a) QCA wire, (b) QCA inverter, (c) QCA Majority Gate

thermodynamic problems, short feedback pathways, adaptable routing, and scalable regulation. In the illustration 5(a,b), the QCA wire cross is either coplanar through 45° rotated cells or multilayer. Mutilayer wire crossing helps in reduction of design area, cell count ,and latency benefits due to 3-D fabrication process. The Nand-Nor-Inverter(NNI) universal gate, whose architecture and truth table are shown in the picture 6, has a lesser footprint to enable NAND and NOR operation. It can receive three inputs from both directions or zones and adheres to the USE clocking scheme, which is a set of design standards. The logical function of NNI gate represented by equation 2.

$$NNI(A, B, C) = MV\,(A', B, C') = A'B + BC' + A'C' \tag{2}$$

An n-bit multiplier, a 2nd adder, and an accumulator register are components of the classic MAC unit figure 7(a). Due to the multiplier, the MAC unit uses a massive amount of energy to multiply between input points and accumulate the output over time. QCA-based implementation consumes less energy than CMOS technology. In this technique, the precise synthesis method is used to synthesize the circuit with a blend of NNI and majority gates while taking into consideration the USE clocking scheme. In AI accelerators, static random access memory (SRAM) was utilized to store feature maps and kernels. A 4*4 scalable design is depicted in the picture 7(b). The decoder and multiplexer blocks are used based on S0 and S1, respectively, to address certain memory words for reading or writing (figure7(b)).

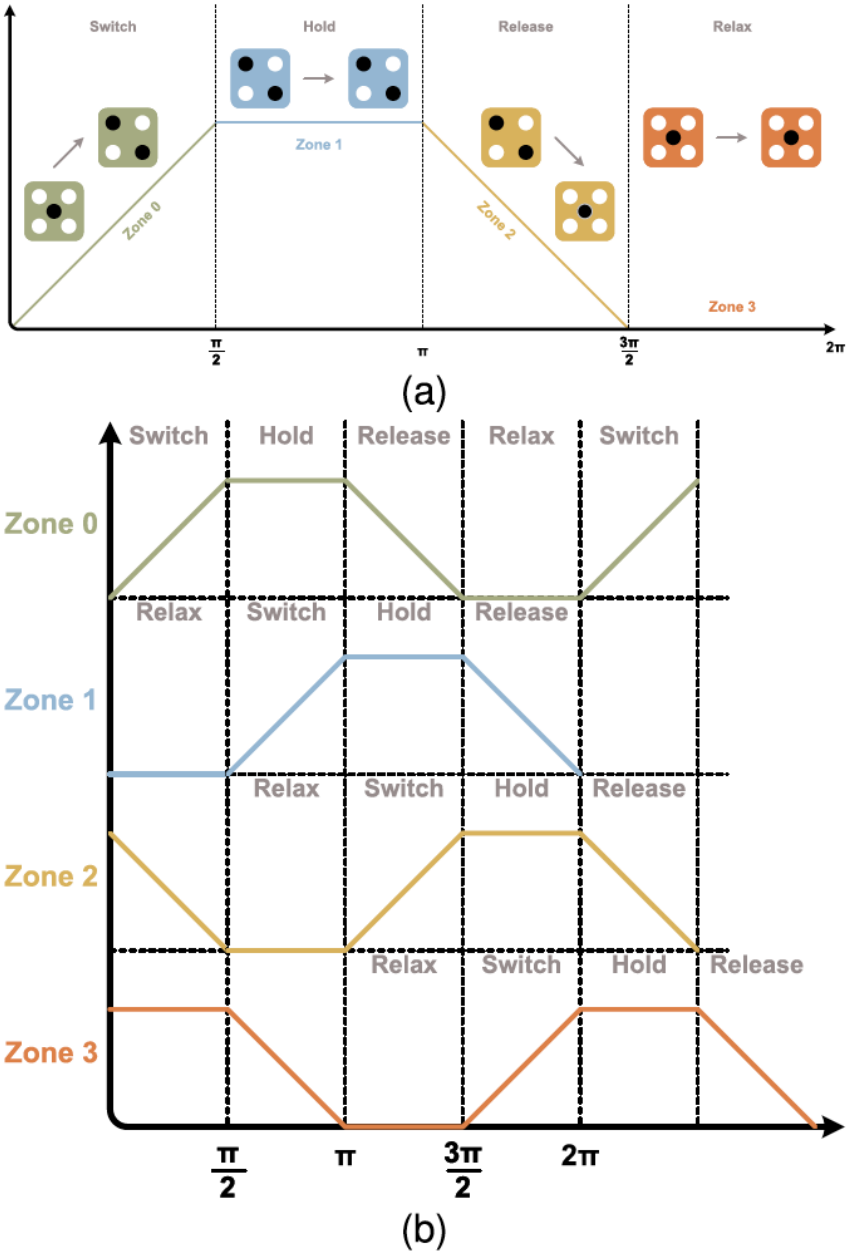


Figure 3: QCA clocking Clock(a) phases, (b) zones

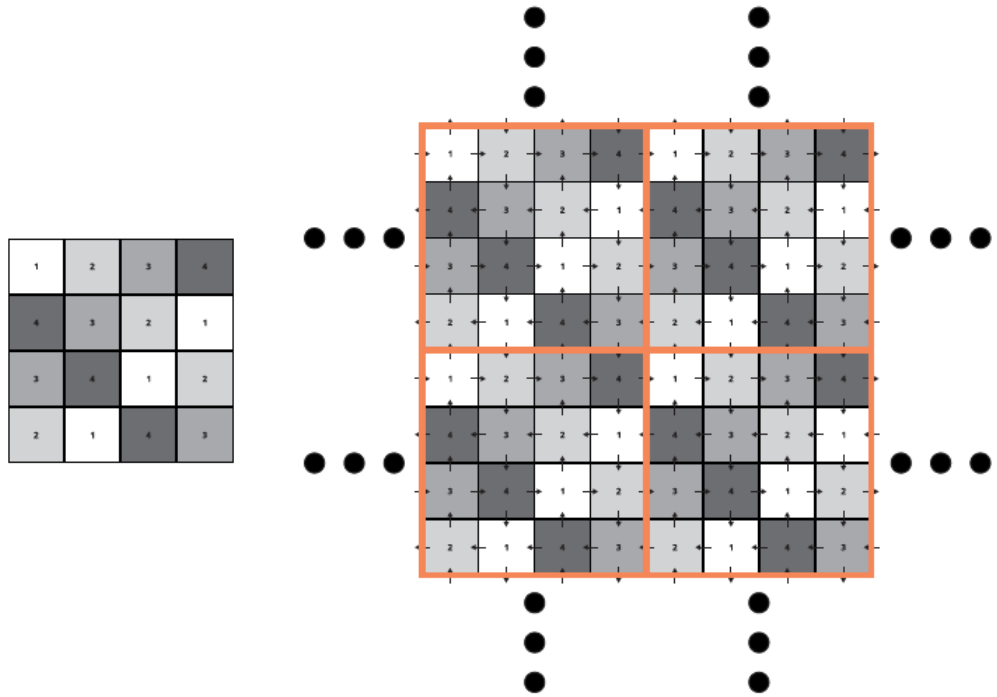


Figure 4: USE clocking scheme

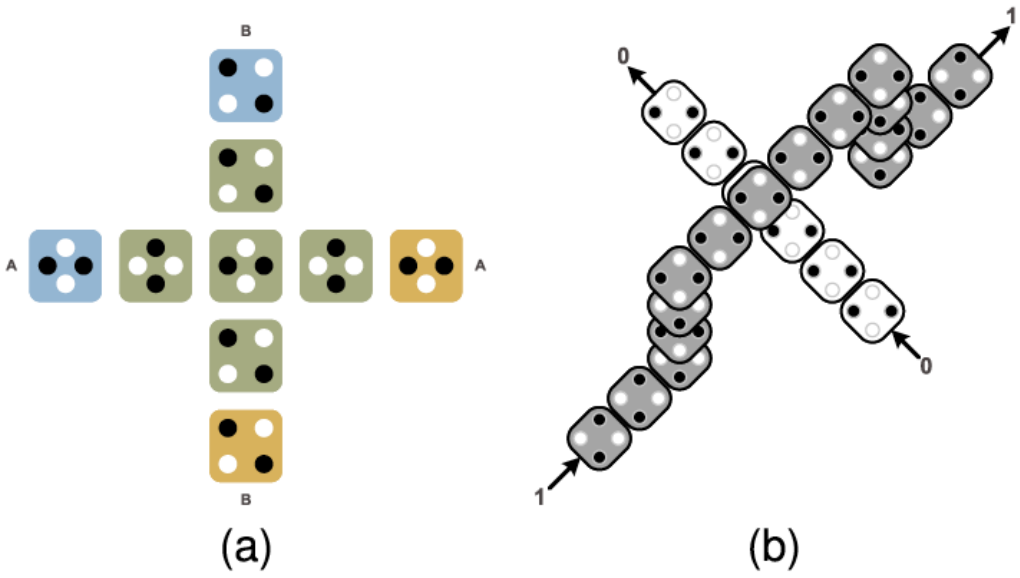
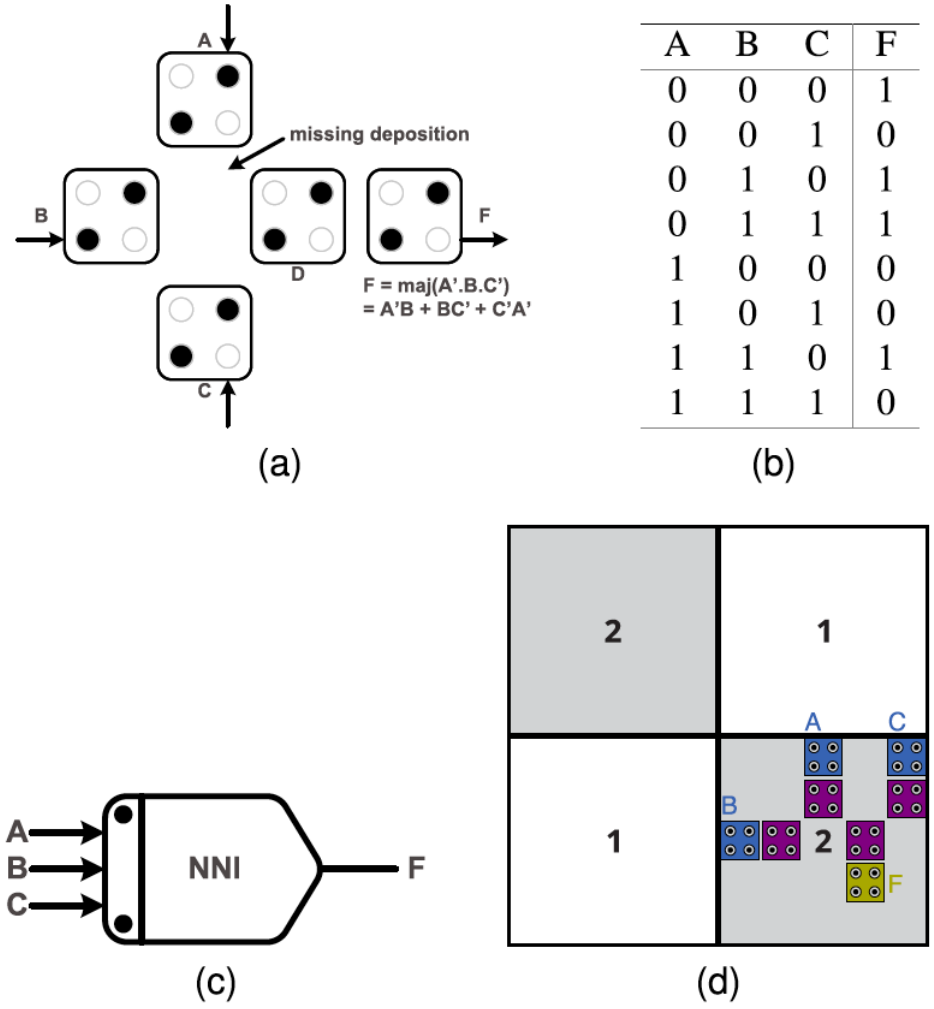


Figure 5: QCA (a) coplanar and (b) multilayer wire crossing



A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Figure 6: Nand-Nor-Inverter(NNI) (a)NNI Gate QCA Layout (b) NNI Gate Truth Table,(c) NNI Gate Symbol, (d)NNI structure in USE Layout

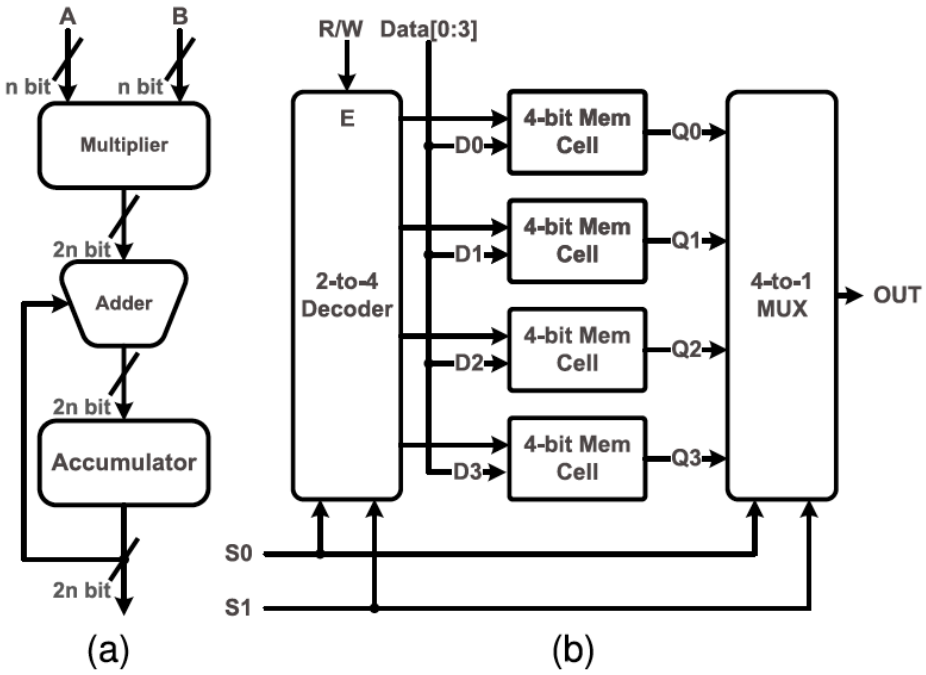


Figure 7: Common AI accelerator blocks (a)MAC unit (b) SRAM unit

The simulation settings used by QCA designer to create design simulation are listed in the table 1. Afterwards,it creates distinct designs by adhering to the clocking scheme, NNI, majority gate, and PIPO principles, which are described in the following section.

2.1.1. QCA MAC unit

The half-adder figure 9 and the full-adder figure 10 are crucial parts of the Vedic Multiplier because they allow it to sum up partial products in parallel since it is more efficient than other approaches. The Vedic multiplier has an 8-bit output and two 4-bit inputs, A and B. The graphic 8 depicts a Vedic multiplier; it has an OR gate, three 4-bit adders, and four 2-bit Vedic multipliers. The 4*4 Vedic multiplier has 5829 cells, 13.27 μm^2 of area, and 19 clock cycles of delay. The full adder figure 10 has a delay of 1 clock cycle, 78 cells, and an area of 0.09 μm^2, whereas the half adder figure 9 design has a wait time of 1 clock cycle, USE clocking technique, 41 cells, and an area 0.06 μm^2. The two half-adders and the four majority gates make up the 2*2 vedic multiplier figure 11 (a). The 2*2 vedic multiplier in the figure 11 (b) has 440 cells, a 1.03 μm^2 total area, and a 4-clock cycle delay. Equations 3 and 4 are used to create the half adder boolean, while equations 5 and 6 are utilized to create the full adder boolean.

$$\text{Cout } = A \cdot B = MV(A, B, 0) \tag{3}$$

$$\text{Sum } = A \oplus B = NNI(\text{ Cout }, MV(A, B, 1), 1) \tag{4}$$

$$\text{Cout } = AB + BC + AC = MV(A, B, C) \tag{5}$$

$$\text{Sum } = A \oplus B \oplus C = NNI(\text{ Cout }, A, NNI(B, A, C)) \tag{6}$$

To do the accumulator operations, an 8-bit parallel in parallel out (PIPO) register constructed of D flip-flops is utilized, as shown in the figure 13 (a). The D flip-flop is created with the assistance of the level-to-edge converter circuit 13 (f) and D-latch 13 (d). The PIPO register is constructed using the D flip-flop, as seen in the picture 13 (a). The first design 13 (b), which has 932 cells and a 2.30 μm^2, 4.75 clock cycle delay, employs typical input, and output delay matching to synchronize the arrival of direct and feedback input. The last D flip-flop on the left side is reached by the signal from the right side after traveling 15 clock zones; input D7 is delayed by the same amount. In a similar manner, output delays must match for all output to be produced simultaneously. The D-latch QCA layout design uses 20 cells, area 0.02 μm^2, and 2 clock zones delay.
The second design 13 (c) eliminates aside with the necessity for input/output delay due to the periodic nature of the clock signal, 561 cells, 0.87 μm^2 of area, and 0.75 clock cycles of latency. It guarantees that the signal is stored in its original form at each flip-flop by giving CLK signal that is twice as large as QCA in the clock period by delaying the CLK signal by two clock cycles between flip-flops. This is the flaw in the design; for example, if an n-bit register is used, the CLK signal must travel n*2 QCA clock cycles to reach the last flip-flop. At startup, it requires 16 clock cycles of initialization before it can operate properly. The multiplier and adder in the system have 19 and 8-clock cycle delays, respectively, giving the design more than adequate time for a startup.

2.1.2. QCA SRAM

The decoder is switched off when the signal is 0 and turned on when the signal is 1. The design in the picture 12 is a 2-to-4 decoder with USE clock zones, and it has 163 cells, 0.20 μm^2 of area, and 1.25 clock cycles of delay. The 4-bit memory cells that have been suggested are parallel in parallel out (PIPO) registers, in which the memory cells are D-latch. The R/W signals of the latches are the output of the decoder figure 13. The 4*4 SRAM design is illustrated in the diagram 14; it has 4293 cells, a surface area of 6.81 μm^2, and a 15-clock cycle delay. One

Parameter	Value
Cell size	18 * 18 nm^2
Dot diameter	5.00 nm
Distance between cells	2 nm
Number of samples	12,800
Convergence tolerance	0.001
Radius of effect	65 nm
Relative permittivity	12.9*
Clock high	$9.8E-22$ J
Clock low	$3.8E-23$ J
Clock amplitude factor	2
Layer separation	11.500 nm
Maximum iterations per sample	100

* Relative permittivity of Gallium arsenide (GaAs)

Table 1: Approximation Simulink Parameters

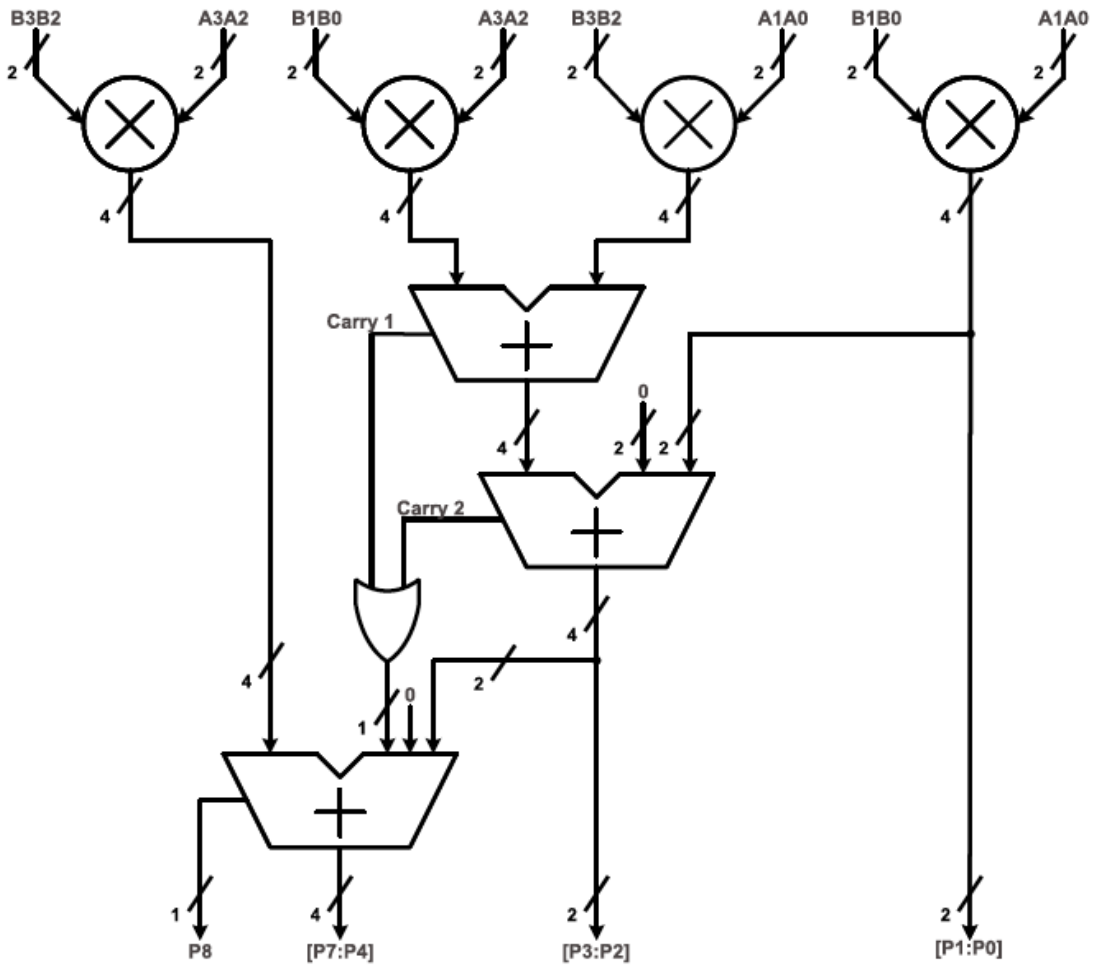


Figure 8: 4-bit Vedic Multiplier

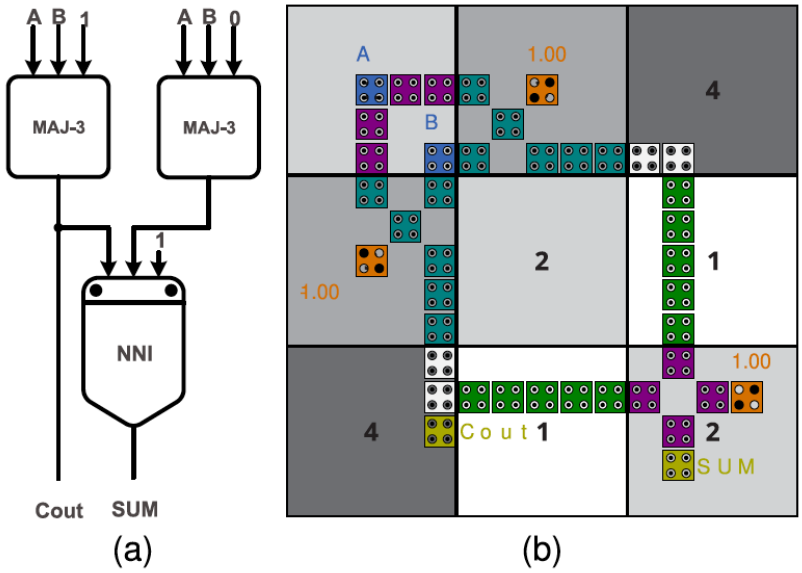


Figure 9: (a) Half-Adder Block diagram, (b) QCA layout of half-adder design

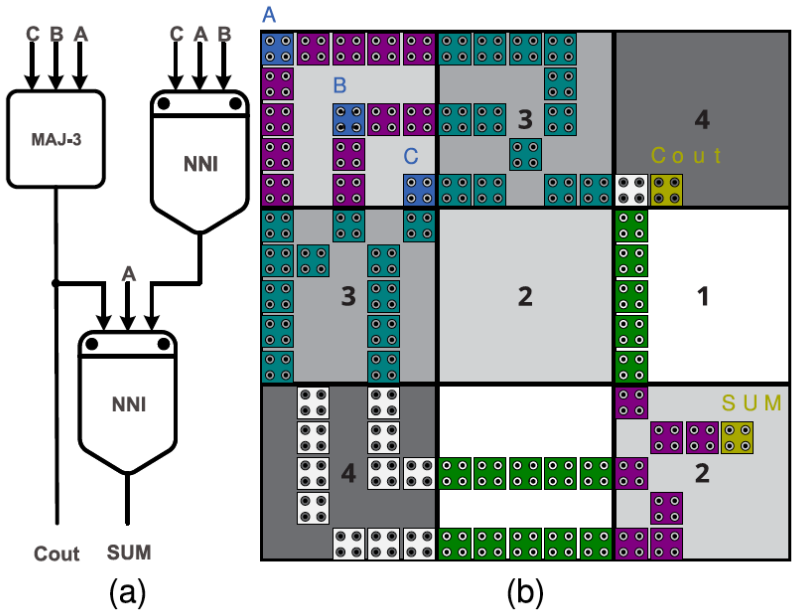


Figure 10: (a) Full-Adder Block diagram, (b) QCA layout of Full-adder design

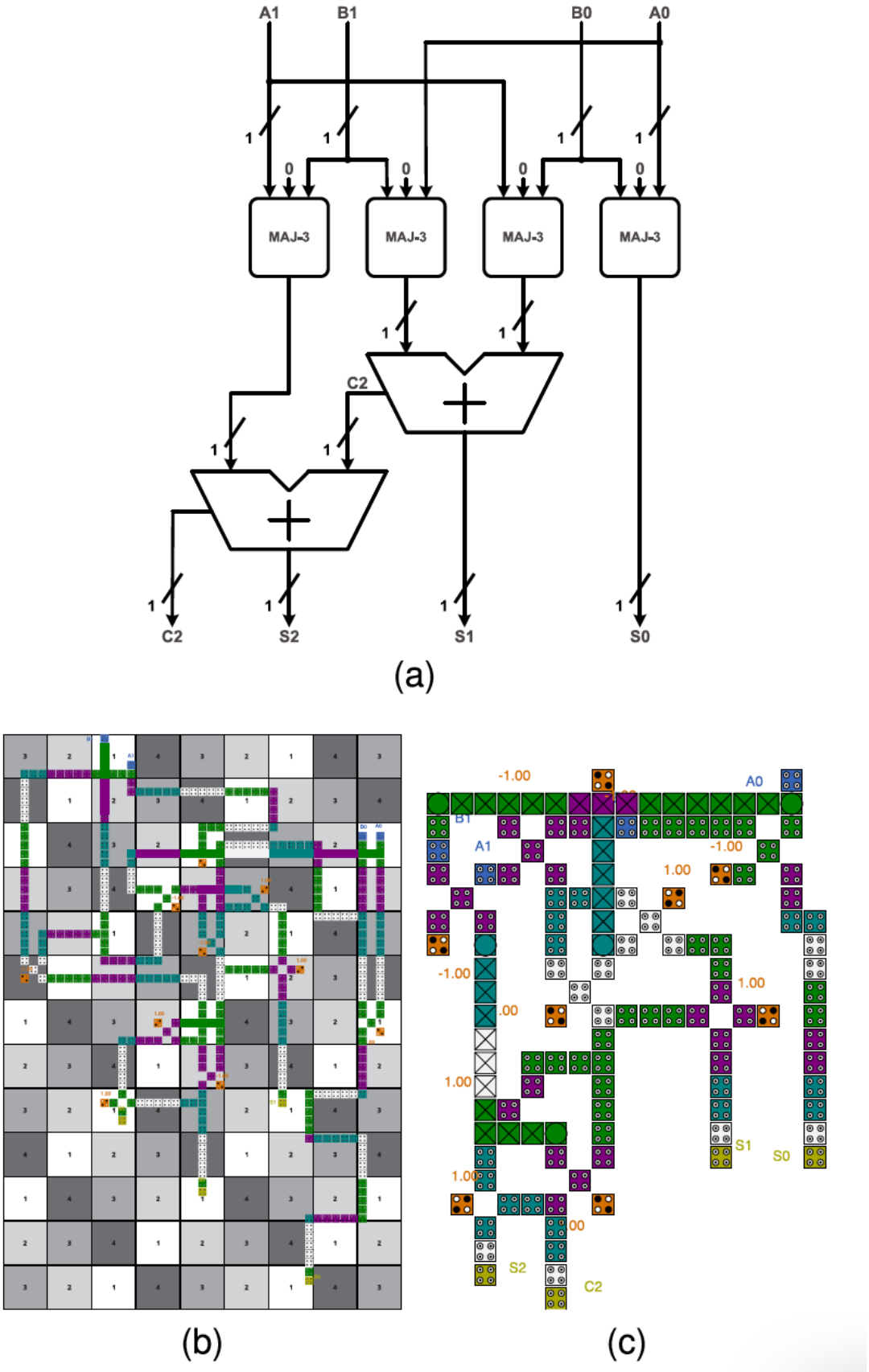


Figure 11: (a) 2 * 2 Vedic Multiplier Block diagram, (b) QCA layout of 2 * 2 Vedic Multiplier design, and (c) QCA layout of 2 * 2 Vedic Multiplier design without USE.

half-adder from figure 9(b) and 7 full adders from figure 10 (b) were used to create an 8-bit half-adder with 2262 cells, an area of 5.09 μm^2, and a delay of 8 clock cycles with help of the ripple-carry design.

2.2. Approximate Computing [7]

The idea of approximate computing architecture is astonishingly trivial for energy-efficient and high-performance design. It focuses on a possible inaccurate result rather than the possible correct result. One particular example is a search engine, where multiple specific query results are possible instead of the exact one. The paper [7] used approximate computing for Digital signal processing(DSP) and AI accelerators. Figure 15 shows the architecture methodology in detail. The first step is to achieve bit level accuracy by utilizing different arithmetic formats, e.g, floating/fixed point, block floating point, and for QAM demodulation application-specific algorithms, e.g, Log-Likelihood-Ratio(LLR), Maximum Likelihood or Winograd for convolution. Secondly, combine with approximate arithmetic units to check accuracy with realistic datasets or/and versatile input distributions. If the application-specific accuracy constraints are satisfied, hardware development for FPGA/ASIC can start. Otherwise, a re-design of the software models with different arithmetic/algorithms needs to be done. In case of hardware side accelerators do not meet the design goal, such as decreased throughput or resource overutilization, result in either an alternative design proposal or back to the initial software level.
The Sobel edge detector and FIR filters are built using the Synopsys Design compiler, TSMC's (Taiwan Semiconductor Manufacturing Company) 65 nm standard-cell library, and the approximation design approach. While the FIR filter is commonly used in digital signal processing, the Sobel edge detector can be employed to find edges in images. Correct Edge Detected (CED) is the accuracy metric for Sobel with 16-bit images, and Mean Relative Error (MRE) is the accuracy metric for the FIR filter with 32,16-bit coefficients. The diagram 16 illustrates hardware improvements for ASIC accelerators with accuracy metrics for several high-radix multipliers (k = 6/8/10). The filter offers gains of 34 percent in energy, 11 percent in delay, and 35 percent in the area at the cost of modest errors (MRE 0.41-3.6 percent). Even with a high radix multiplier (k = 10, CED 99.87 percent) and significant spikes in energy, area, and critical path delay of 55, 46, and 8 percent, respectively, the edge detector is unaffected by errors.

The Quadrature Amplitude Modulation (QAM) arhitecture implemented in Xilinx Vivado on Zynq Ultrascale + ZCU106. Bit Error Rate (BER) is an accuracy metric for demodulation, together with 64-QAM keying signals, 32-bit floating, and 16-bit fixed point arithmetic. The results of 64-QAM circuits are summarised in the tablular figure 17; the BER values range from 10-1 to 10-4 depending on the signal-to-noise ratio (SNR) of the transmitted signal. When compared to floating-point, the resource utilization with approximate design is 1.12 times faster,

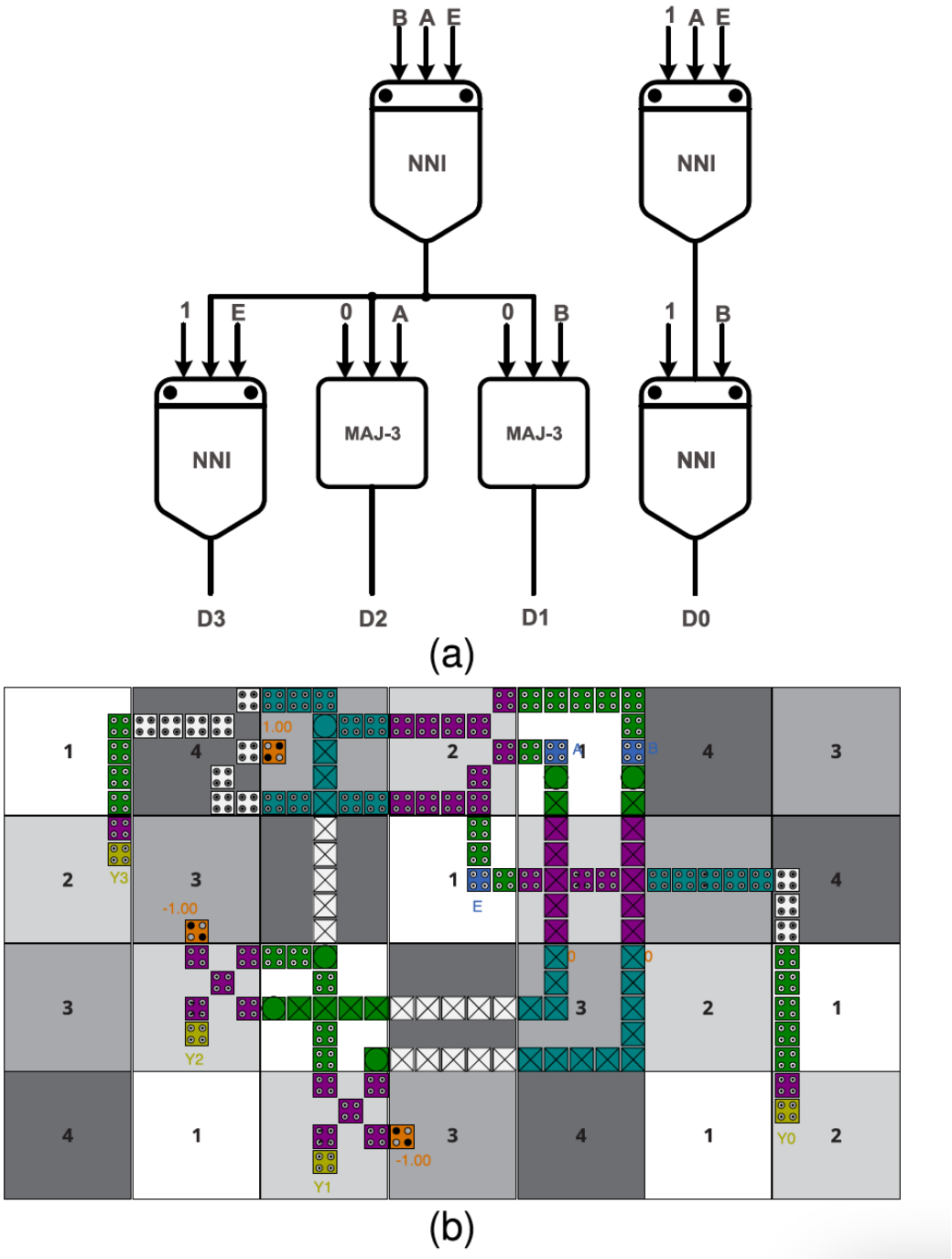


Figure 12: (a) 2-to-4 Decoder Block Diagram (b) 2-to-4 Decoder Layout

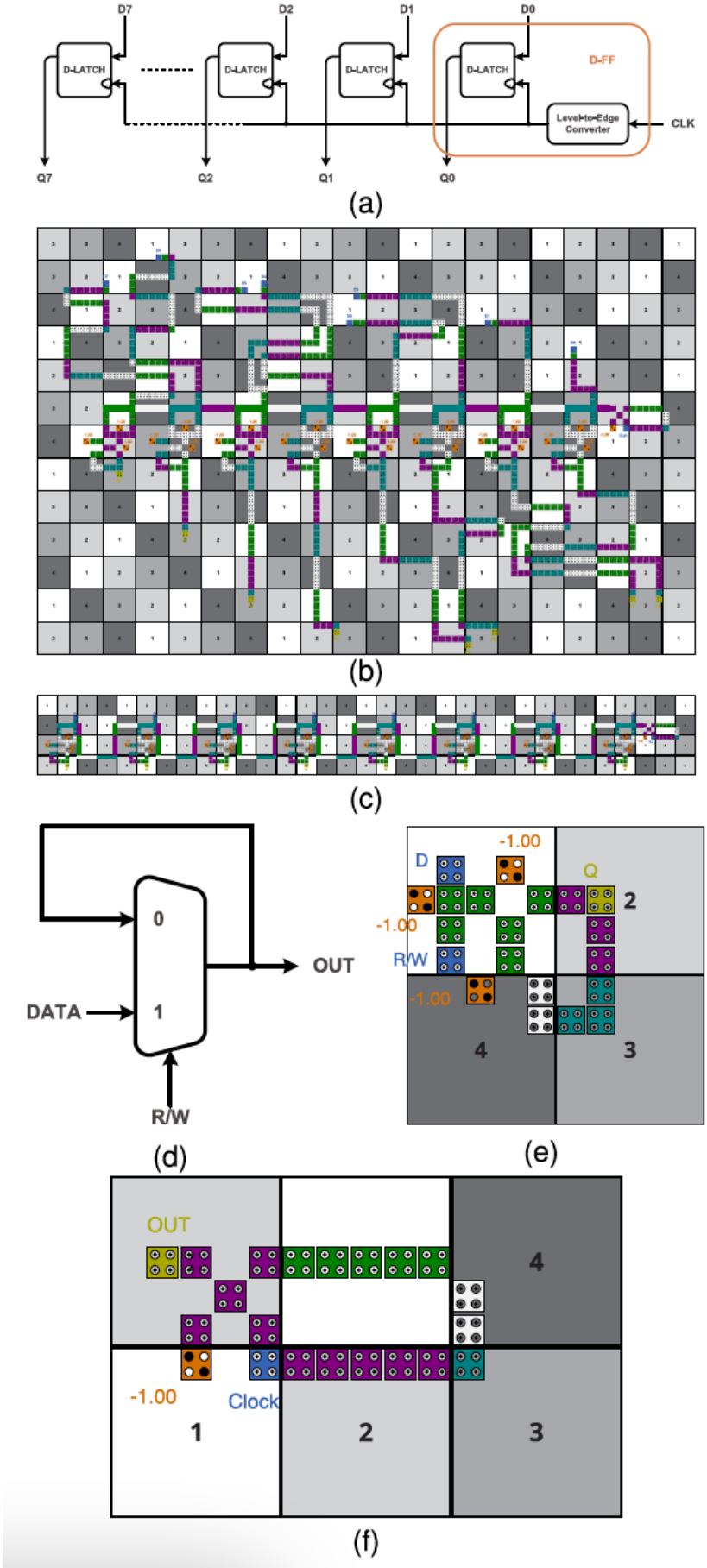


Figure 13: (a)PIPO block diagram (b)First 8 bit Register layout (c) Second 8-bit register layout (d) D-Latch Block diagram (e)Proposed D-Latch layout (f) Level to Edge converter

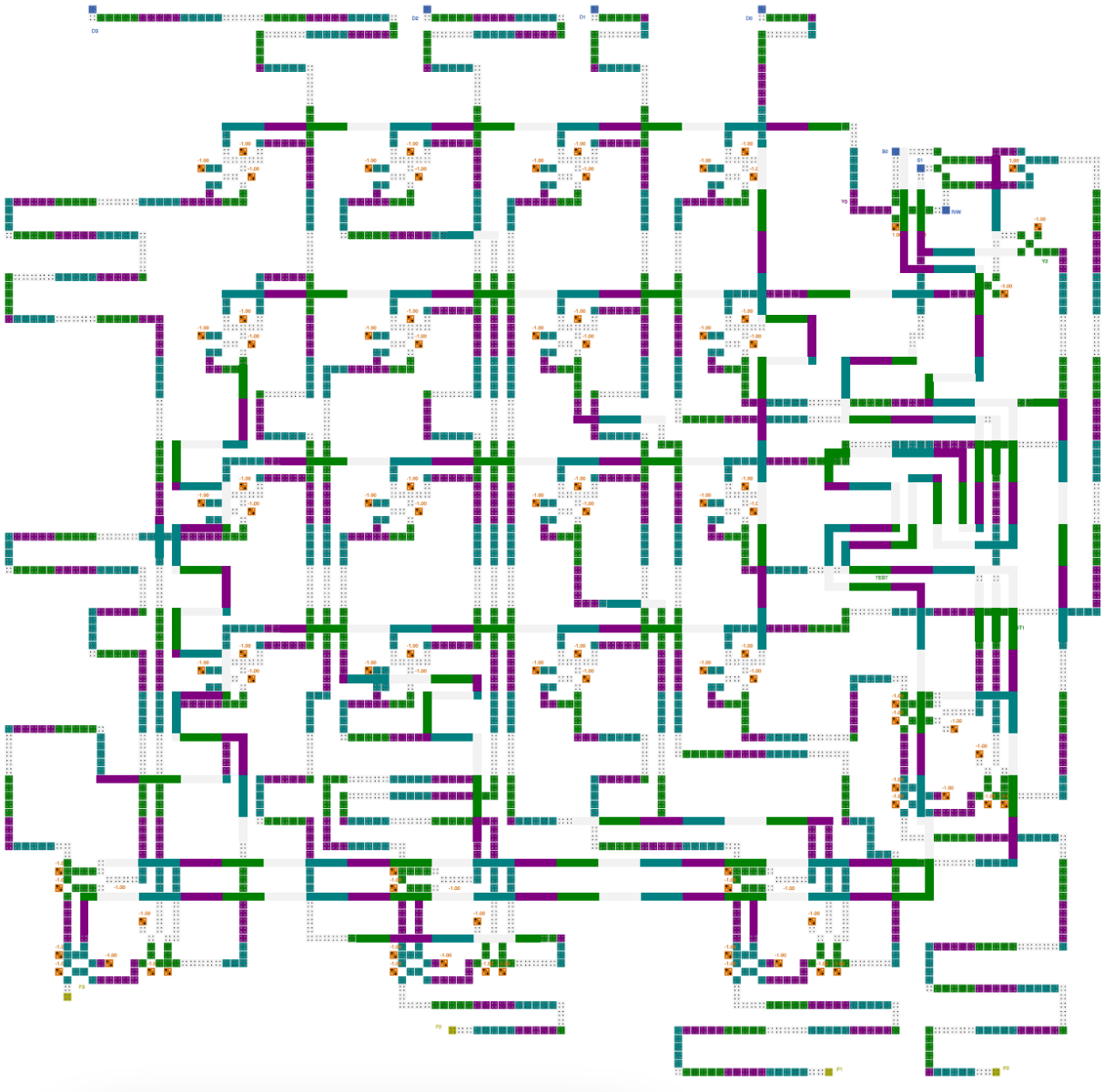


Figure 14: 4*4 SRAM

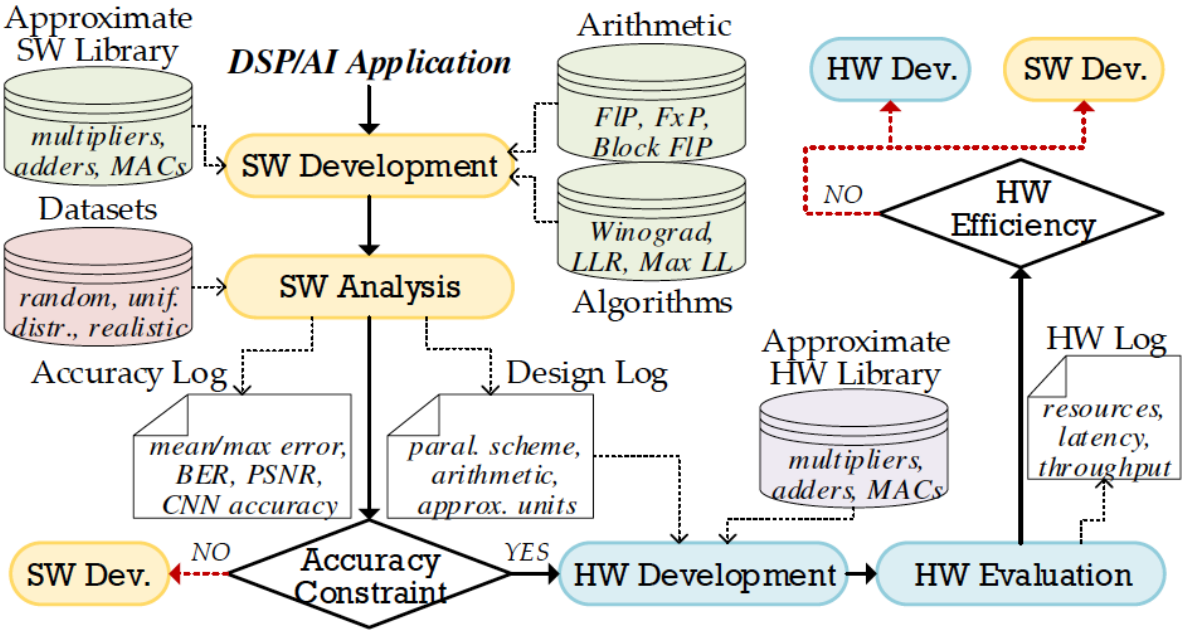


Figure 15: Approximate Computing Methadology

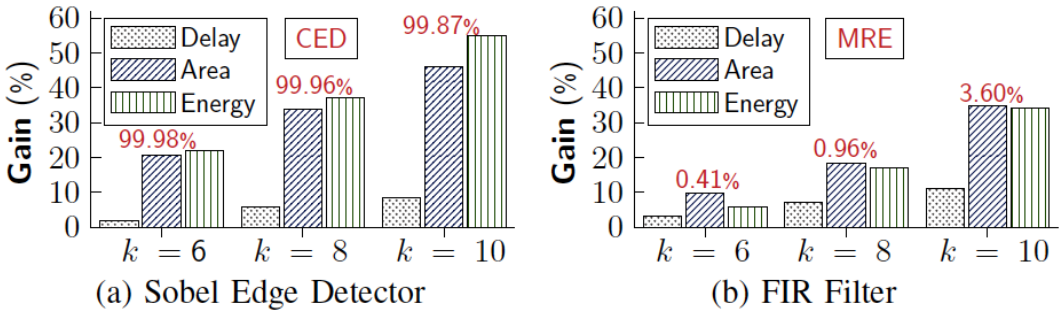


Figure 16: Approximate ASIC based accelerator

reduces LUT by 65 percent, and significantly outperforms fixed-point with approximation (k = 6).

Arithmetic Config.	Paral.	LUT	MHz	Msamples/s	BER
Accurate Fl. Point	64	46%	286	286	10^{-1} – 10^{-4}
Accurate Fx. Point	64	24%	312	312	10^{-1} – 10^{-4}
Fx. Point ($k = 6$)	64	16%	321	321	10^{-1} – 10^{-4}

– BER is for SNR/symbol values 0–14dB.

Figure 17: Approximate 64-QAM circuits on Zynq ZCU106

Convolutional neural network (CNN) architecture on Zynq 7020 FPGA has 4 convolutional layers and 2 fully connected layers (132K parameter) developed in Xilinx Vivado. The inference is implemented on 16-bit floating and fixed point arithmetic, and the training phase is carried out via Tensorflow with an accuracy of 96.8 percent for ship images 128*128. The results of CNN when used with accurate and approximative convolution engines in balancing resources to fit within the device are shown in the table 18. Higher parallelization is apparent in block floating point with k = 8 (2nd row), which offers 4 * Frame Per Second (FPS) but 100 DSP use. When using Winograd (4th row), you may obtain a similar throughput with a large 42 percent decrease in DSP and a 4 percent increase in LUT. When Winograd is used, the growth in fixed point arithmetic resources can be described as growing by up to 38 percent with a 13 percent reduction when k = 8. Only a 0.1–0.2 percent accuracy decrease is visible across all approximation CNN architectures.

Arithmetic Config.	Paral.	LUT	DSP	RAMB	MHz	FPS
Accurate Fl. Point	8	37%	91%	78%	125	182
BFl. Point ($k = 8$)	32	65%	100%	95%	125	730
W. BFl. Point ($k = 8$)	8×4	69%	59%	95%	124	724
Accurate Fx. Point	32	69%	58%	95%	118	689
Fx. Point ($k = 8$)	32	60%	54%	95%	124	724
W. Fx. Point ($k = 8$)	8×4	43%	58%	95%	112	654

– accuracy loss is 0.1–0.2% among configurations.

Figure 18: Approximate CNN circuits on Zynq-7020

2.3. Adder Convolution neural network (AdderNet)[11]

Convolution neural network (CNN) networks consume intensive energy due to their computationally intensive multiplication operations. These operations are necessary to achieve

significant accuracy. In an adder convolutional neural network(AdderNet)[11], original convolution replaces multiplication with adder kernel-only additions. Low-bit quantization algorithm with int8/int16 quantization consequence high performance consuming less energy resource. It is vital to consider addition operations are lightweight compared to multiplication operations. Normally, there are five convolution kernels. Figure 19 (b) CNN, the shift operation kernel (c), the analog memristor kernel (d), the XNOR logic operation kernel (e), and the adder kernel (f). The convolutional layer uses a sequence of convolution kernels (filters) to identify local combinations of features by comparing the similarity of the filters and inputs. To measure the similarity between input feature and convolution filter vector-based multiplication (cross-correlation) used in the CNN model as in eq 7. In AdderNet, to measure similarity by calculating the absolute difference between filters and inputs. The negative sign changes the negative L1 distance to positive with eq. 8.

$$\mathrm{S}\left(\mathrm{F}_{\text{in}}, W\right) = \mathrm{F}_{\text{in}} \cdot W \tag{7}$$

$$\mathrm{S}\left(\mathrm{F}_{\text{in}}, W\right) = -\left|\mathrm{F}_{\text{in}} - W\right| \tag{8}$$

The memristor network figure 20 is a component that can be used in neural networks because electrical pulses can modulate conductance. When a substance conducts electricity well, it has a high conductance and a low resistance. It is crutical to take conductance into account when pulse applied. Due to the phenomena known as the iconic effect, the conductance decreases as the memristor's pulse count rises. The iconic effect is connected to the movement of ions when pulses are applied, cause material within the memristor to be rearranged and alter the conductance behavior figure 20(a). The conductance value may be changed from the lowest to the maximum, much like weight parameters in a network. The conductance variation is inevitable, and the conductance level is limited to only 4-6 bits. The memristor's construction comprises a one-transistor-one-memristor (1T1R), a transistor individually controlled memristor utilizing write-line (WL), and select-Line (SL). The weight parameter value can be positive or negative; however, conductance in the memristor is always positive. Due to the usage of bit-level operations and analog circuits, memristors consume the least energy; however, give the worst network performance. The chip area is also a concern in the memristor network due to large digital-to-analog and analog-to-digital converters.

The AdderNet kernel to calculate the distance of weight and feature contains one-Comparator-one-Adder(1C1A) or two Adders(2D). Figure 21 shows the logical circuits of the AdderNet network. The lower power consumption is a result of adder-operation and comparator-operation in AdderNet. The quantization technique helps in compression to represent lower bit values of weights and features. To achieve high performance and low power consumption 8 or 16-bit quantization schemes have been used in AdderNet. Continuous values can be represented as finite discrete values with the use of quantization. In terms of representation, 8-bit implies 2^8 = 256, and 16-bit means 2^{16} = 65536, with 16-bit undoubtedly providing better precision due to

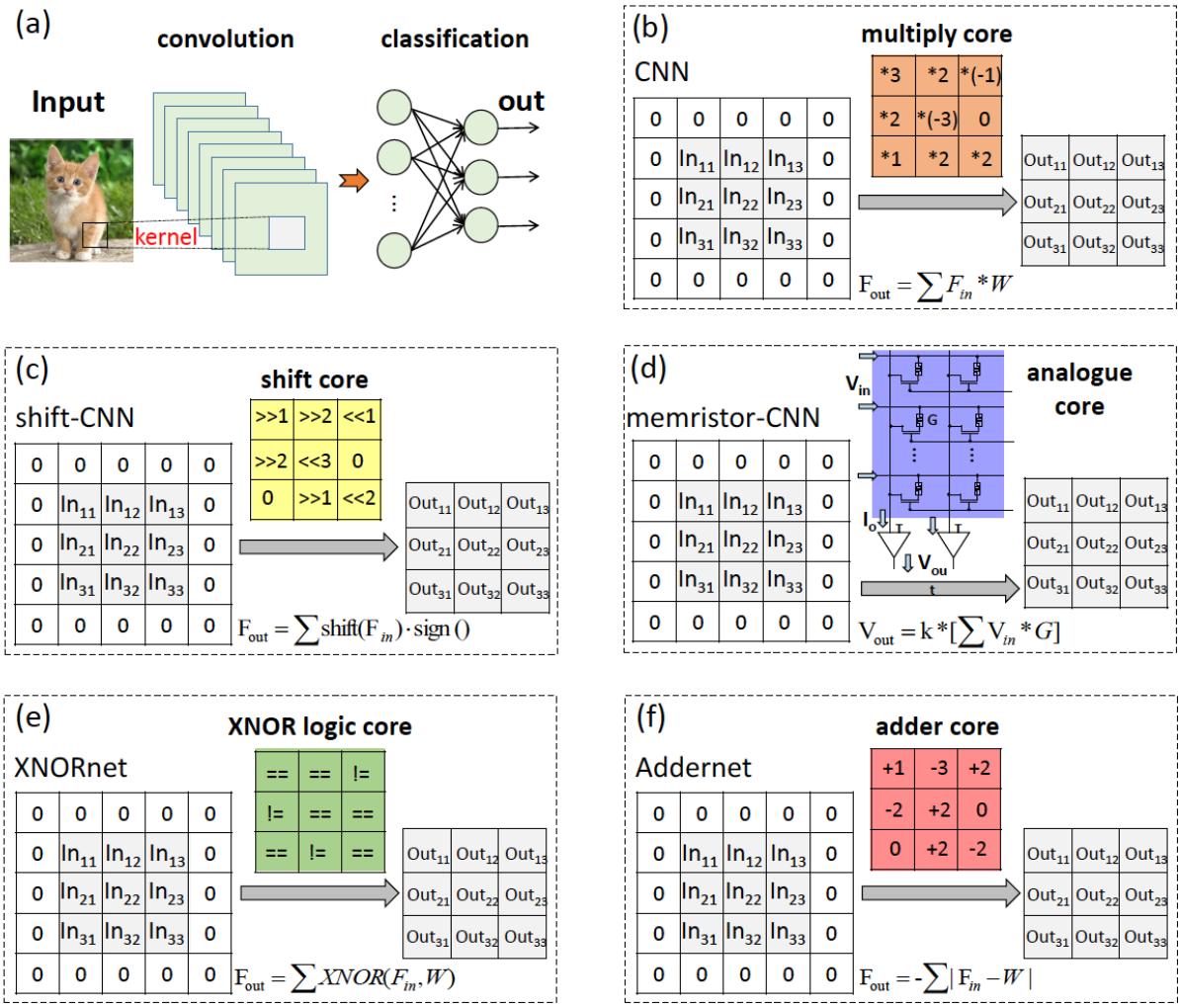


Figure 19: Five Types of Convolution Kernel

more number of bits.

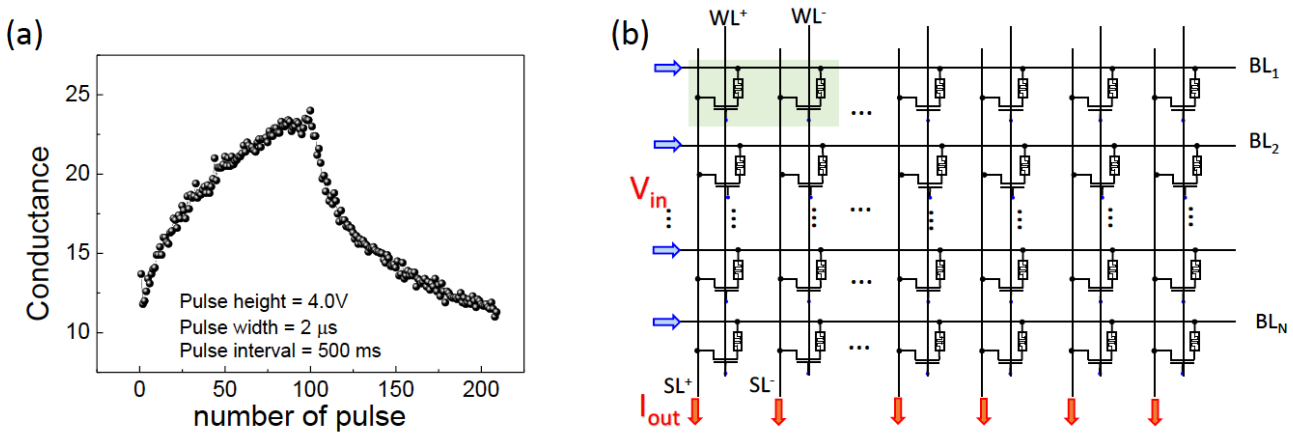


Figure 20: (a) number of the pulse (b) Structure of memristor Network

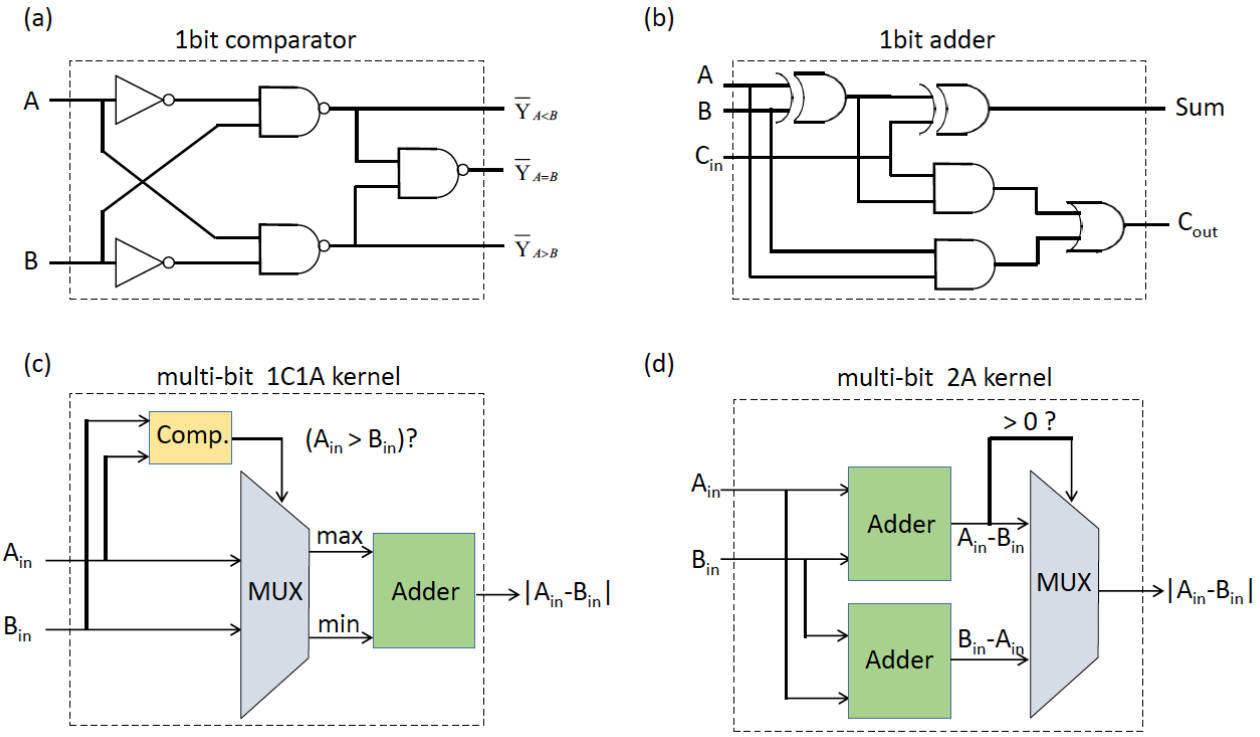


Figure 21: Logical Circuit of AdderNet Convolution kernel

The structure of an FPGA accelerator consists of 4 parts, a data storage unit, an Input/output port, a data path control module, a parallel kernel operation unit, and a pooling, BN unit. Figure 22 (a) shows the input channel P_{in}total sum of the adder tree output channel P_{out}, and the data width is DW. The CNN kernel only has one multiplier, but the adderNet kernel has two adders and one multiplexer. However, adderNet is more efficient than CNN due to its operations. Figure 22 (b)represents a general-purpose accelerator on an MPSoC board with a combination of the Processing system(PS), and Programmable logic(PL). The Advanced extensible Interface

(AXI) controls data transfer between PS and PL. The challenging task is data movement from the main memory(DRAM) to the computation part, which takes a substantial energy consumption.

The LeNet-5 on Zynq-7020 has been implemented in Zynq with no input/output. All computation and weights are stored on board. The LeNet5 is a conventional convolution neural network designed for handwritten recognition tasks. It lays the foundation to state-of-art CNN architecture. Figure 23(a) depicts the structure of LeNet5 as follows, first convolution layers with 6 filters followed by average pooling, then 16 filters convolution followed by average pooling layers, a fully connected layer, and an output layer. The result shows a significant drop in energy consumption and logic resource utilization figure23 (b,c) compared to figure 22(c1,c2,c3,d1,d2,d3) where we need to transfer data by bus.

2.4. Processing in Emerging Memory(PIEM) [3]

Traditionally, processing data required sensor systems that could gather analog data and transform it into digital data. The performance set of adders and logic gates on digital data typically consumes energy and needs a sizable amount of chip area. The alternative approach is using memristors [3](non-volatile memory) arranged in an array configuration. Since the memristor can remember the conductance state after power is disconnected, it could be utilized in neural network implementation. Moreover, altering the conductance value to low-value yields low-energy MAC calculations. The setup of the Processing in the Emerging Memory (PIEM) array is depicted in the image 24.The current depends on the total memristor conductance and the reference voltage from the associated row. The PIEM array might be used as memory or a reconfigurable logic array of a sensor to do high-performance and low-energy computing in edge platforms (mobile phones, drones, wearable devices, etc.) figure 25. Due to the fact that a fully integrated MAC near sensor only functions in the analog domain, there would be no need for sophisticated data converters for data processing.

2.5. Vector Symbolic Architecture (VSA) or Hyper Dimensional Computing[5]

Although the Von Neumann architecture was developed to compute numbers for applications requiring precise responses, there is now a greater demand for computational elements with low power requirements, tiny size, and stochastic behavior. Deep neural networks still have a limited capacity for symbolic computing. In symbolic representation, calculations are made using exact mathematical relationships and algebraic rules, whereas in numerical computing, calculations are made using numerical approximations. Vector-based symbolic architecture(VSA), on the

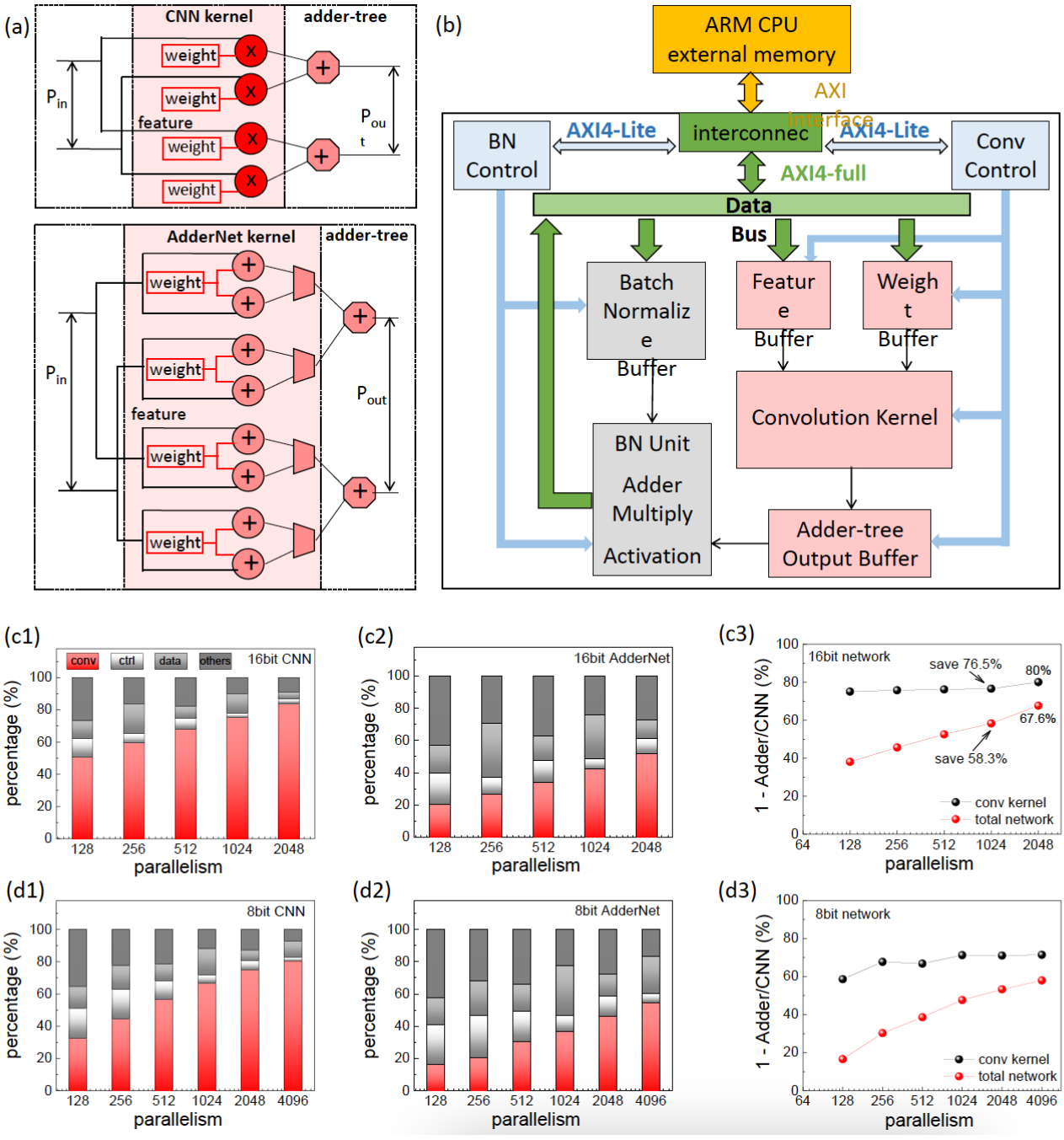


Figure 22: (a) Detailed structure of the FPGA accelerator, (b) Parallel computation on the convolution kernel,(c1,c3)Effect on parallelism with 16-bit AdderNet and 16 bit CNN,(d1-d3) 8-bit comparison of results AdderNet and CNN

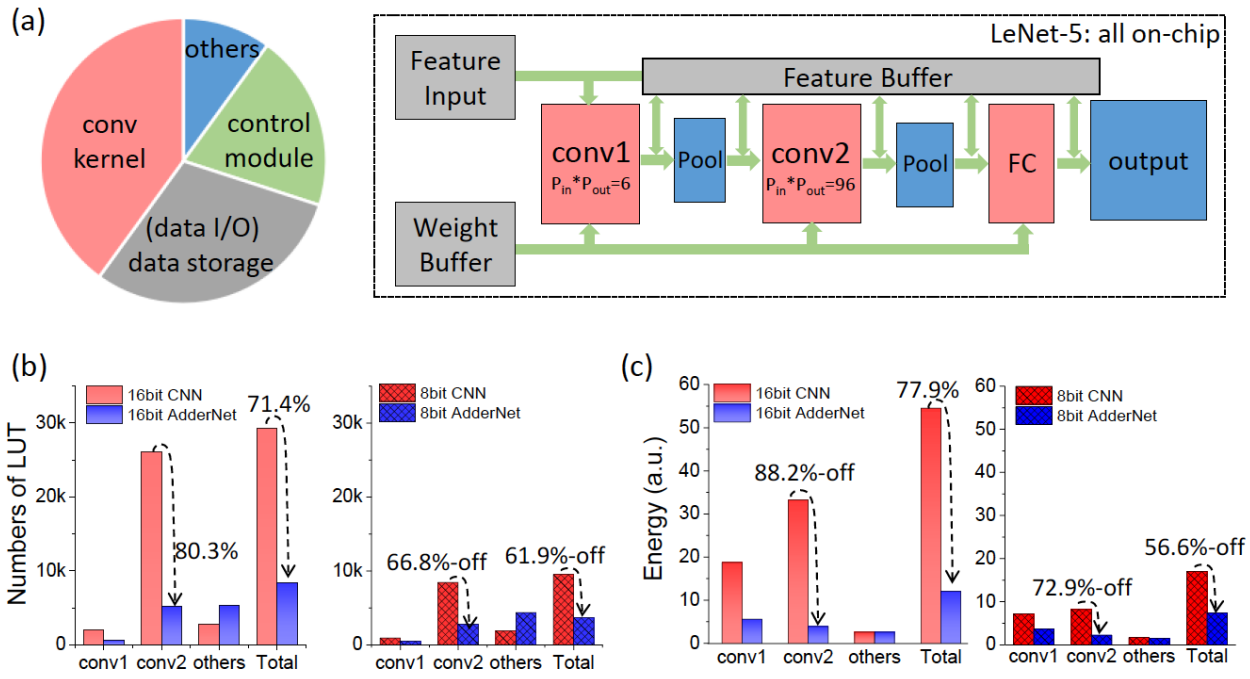


Figure 23: LeNet5 Archtitecture

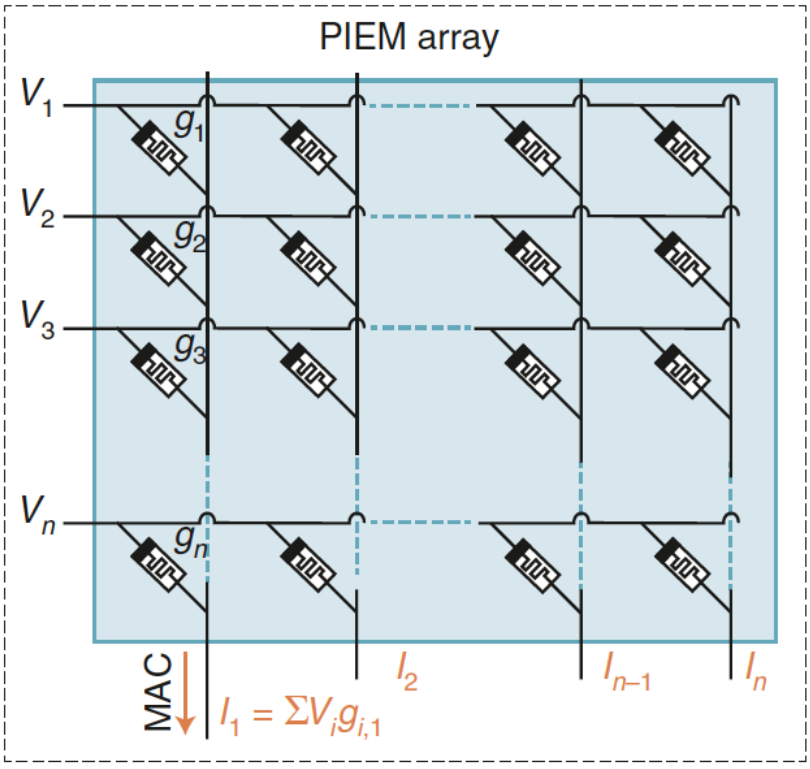


Figure 24: PIEM array

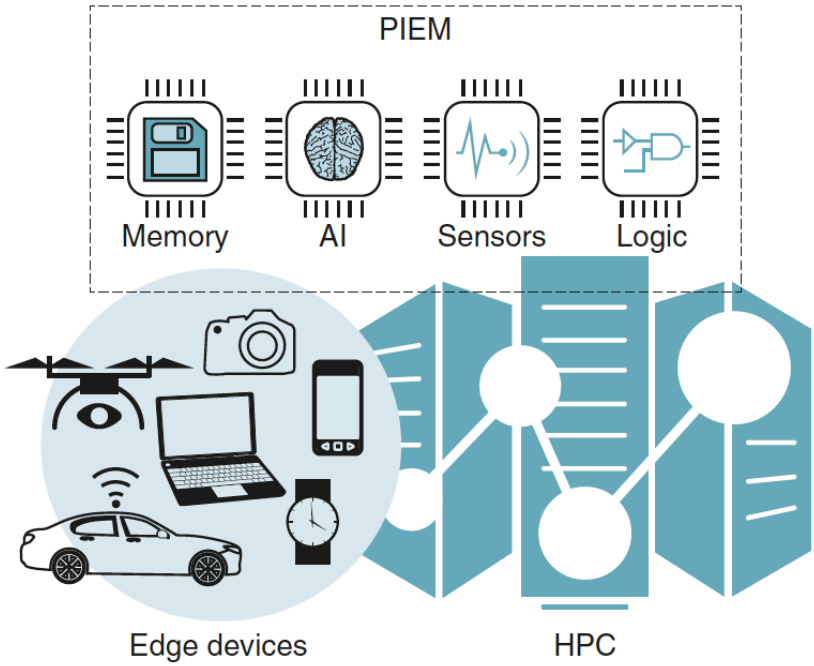


Figure 25: PIEM Applications

other hand, offers several advantages over classical architecture. VSA is able to solve AI issues and can operate on both symbolic and numeric methods by using distributed representation. Distributed representation refers to the presentation of data as a vector with each element representing a distinct aspect. In neural network hidden layers can be presented as vectors ,models for natural language processing, and in graph embedding nodes and edges can be represented as vector. The brain served as inspiration for VSA, which has two interfaces: one for computation and algorithm, the other for implementation and representation.
Using three operations—bundling, binding, and permitting(to protect) the VSA employs the multiply-add permute paradigm. Bunding:+ incorporates two vectors to produce an output vector that is identical to the input vector. Binding $\otimes$ integrates different vectors into one, and the resulting vector is dissimilar from the input vector, but somehow it allows for an approximate retrieval of the input vector. Roles and fillers figure 26 cannot be mixed because of permutation Π since the binding is associative and commutative. The VSA has a capacity of thousands of dimensions which is a difficult task for other algorithms.

"What is the dollar of Mexico?"[4] is a classic illustration of vector symbolic architecture. Using the name of the country, its capital, and its currency, we can distinguish between these data. Since each item in vector computing is given a high-dimensional vector according to the standard methodology, in this example, "NAME" is a random vector NAM, "USA" is a random vector USA, "Capital City" is a random vector CAP, "Washington DC" is a random vector WDC, and so on. Constructing a single high-dimensional vector with all the data given in the equation 9 is trivial and contains all the information.

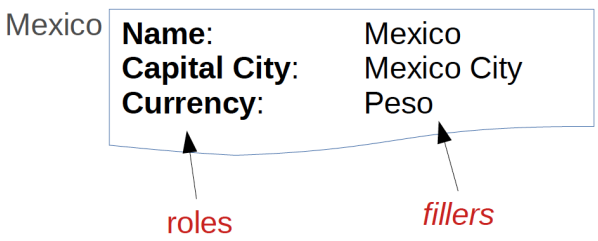


Figure 26: Role and Filler

$$F = (NAM * USA + CAP * WDC + CUR^{*}DOL) * (NAM * MEX + CAP * MCX + CUR * PES) \tag{9}$$

$$\mathrm{F} * \mathrm{DOL} \sim \mathrm{PES} \tag{10}$$

2.6. Application specific hardware[12]

2.6.1. Stress Detection

The algorithms K-nearest neighbor (KNN) and Support Vector Machine (SVM) are utilized to extract features from data. The ECG data comprised, acceleration, breathing rate, and SpO2. To handle multiple supporting vectors, the SVM classification figure 27 is built on a parallel pipelined design, whereas the KNN classification figure 28 uses a semi-parallel architecture to accommodate a distinct set of data and features. To identify K samples with the shortest distance, a sorting block is utilised, and ROM is used for training data.

2.6.2. Intelligent Hearing Aid

Converting an input signal from the time domain to the frequency domain can be done by utilizing Fast Fourier Transform (FFT). A convolution neural network(CNN) collects features to create masks, remove noise, and enhance the quality of speech sounds. Both neural networks and FFT employ the PE array for various operations figure 29. A butterfly unit and a dual-MAC unit make up each PE. The letters "A," "B," "C," and "D" represent the ALU, PE control, CORDIC, and data registers, respectively, in each PE module. The needed nonlinear functions are calculated through CORDIC. CNN mode employs a dual MAC unit to compute input data with weights from two separate filters and a butterfly unit to collect partial sums. In the FFT mode, a dual MAC unit and a butterfly unit serve to compute input with twiddle factor in real and imaginary components and butterfly operation, respectively.

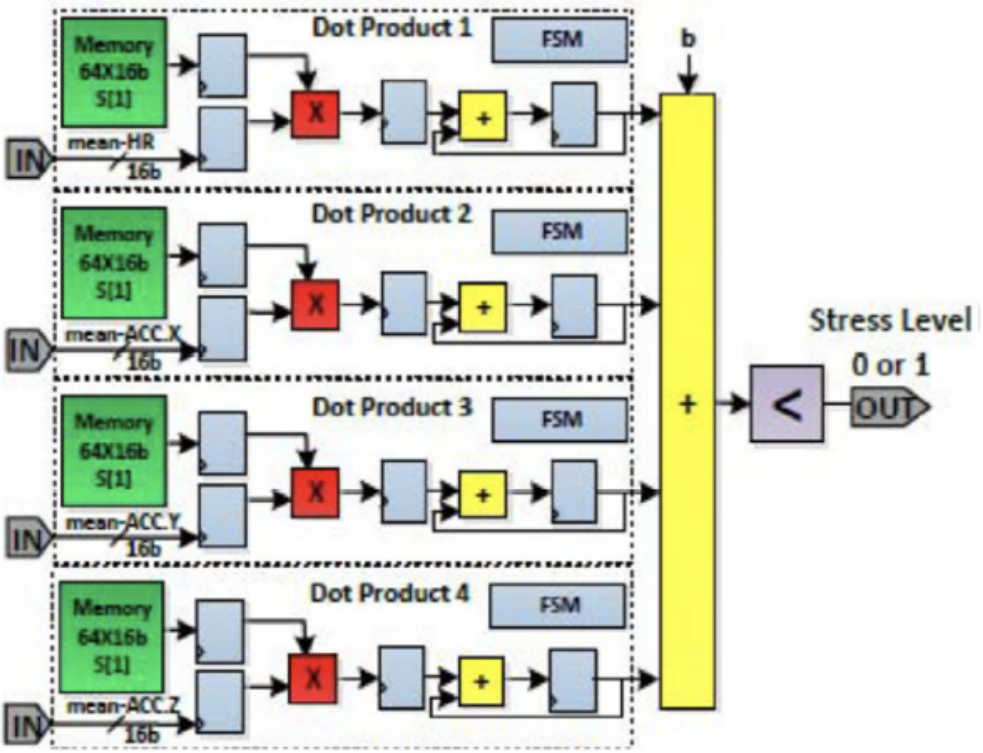


Figure 27: SVM classification processor

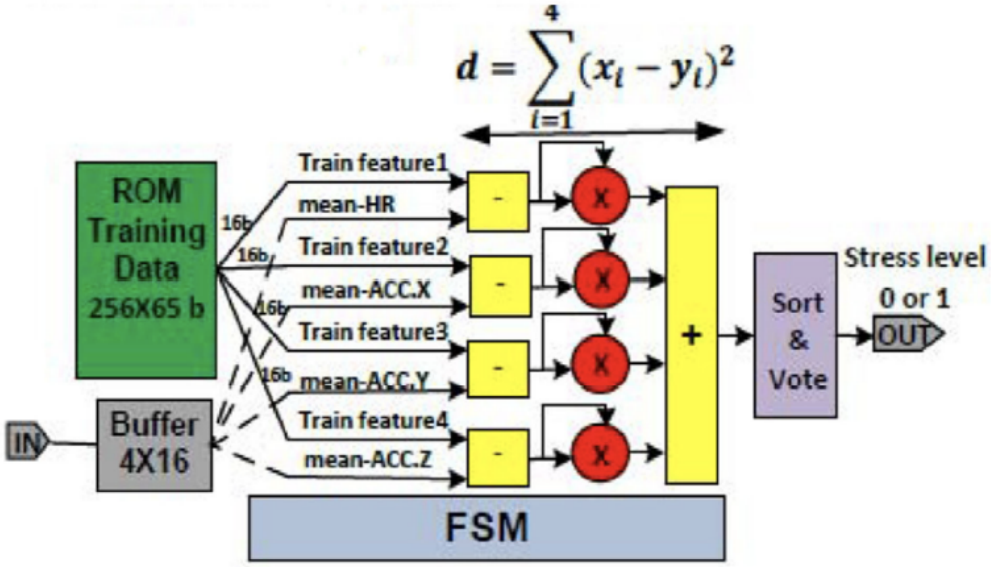


Figure 28: KNN classification processor

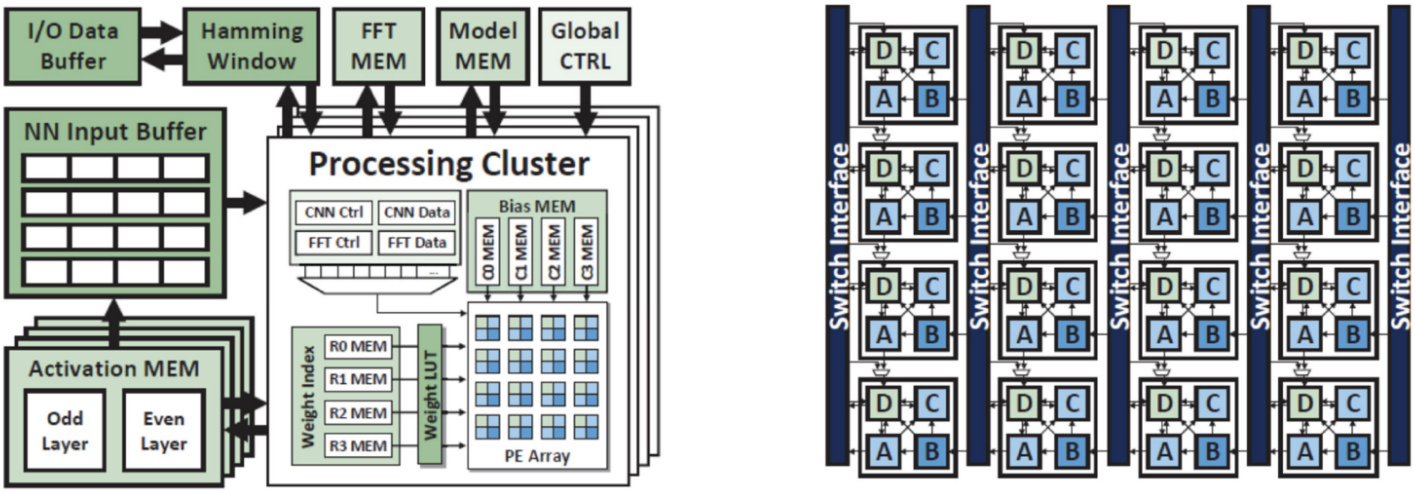


Figure 29: Hearing Aid Architecture

2.6.3. Gesture Recognition

The system design for gesture recognition is based on a wearable device and emphasizes classification accuracy alongside minimal battery consumption. To keep track of muscle activity, it employs electromyography(EMG). It entails placing electrodes right on the skin to monitor and track electrical activity generated by contracting muscles. Figure 30 depicts this particular system architecture. Figure 31 depicts Cerebro ASIC architecture in detail. The Cerebro ASIC, which has 8 differential data collecting channels that are multiplied for data sampling with a frequency sampling of 1 KHz, is utilized for data collection and interface with MCU through SPI. Each channel includes an active RC low-pass(LP) filter after an instrumentation amplifier (IA) with variable gain. It consists of an ADC to detect internal temperature and a Bluetooth module to communicate the results of gesture recognition. It also has a low-impedance patient ground (PGND) to select input common mode.

2.6.4. Feature Extraction from Electrocardiography (ECG) in Mobile Application

ECG sensors used to solely capture raw data in the past, which resulted in a rise in power consumption and a delay in data transmission. ECG sensors with built-in ECG classification processors or " smart ECG sensors " are now popular. These sensors utilize AI algorithms to fulfill the demanding performance and classification accuracy requirements. The picture 32 depicts a two-stage architecture for extracting characteristics from the ECG data, with Discrete Wavelet transform (DWT) identifying QRS/P/T waves and multivariate autoregressive (MAR) analyzing each heartbeat. Applying DWT to readings from the electrocardiogram (ECG) facilitated the identification of QRS, P, and T waves based on unique frequency wavelengths.

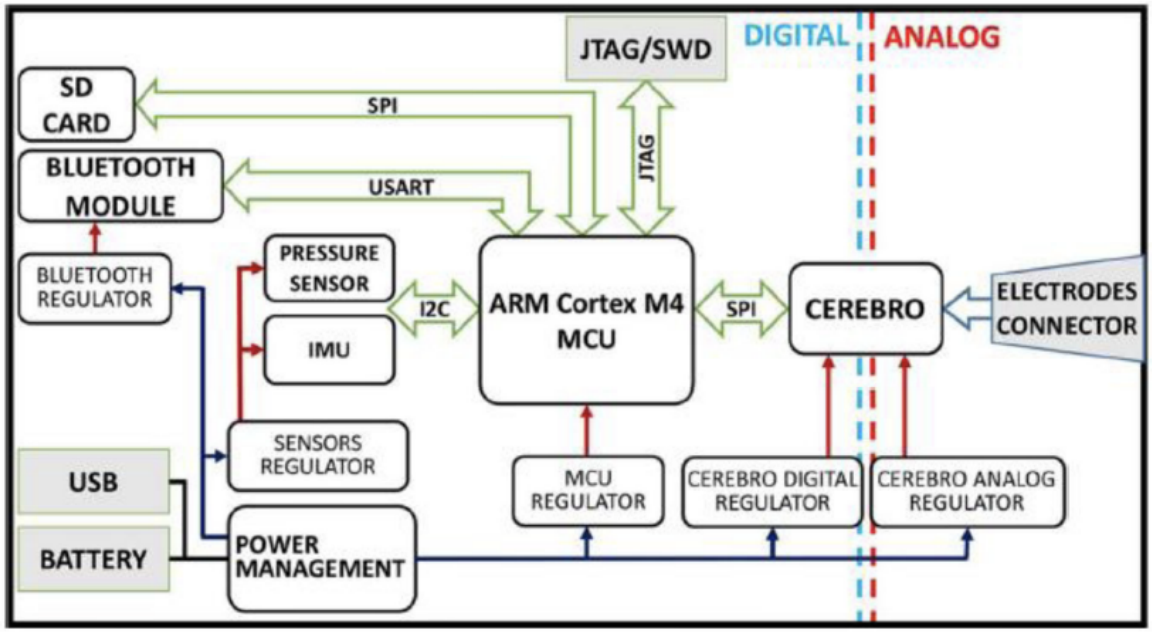


Figure 30: Gesture Recognition Architecture

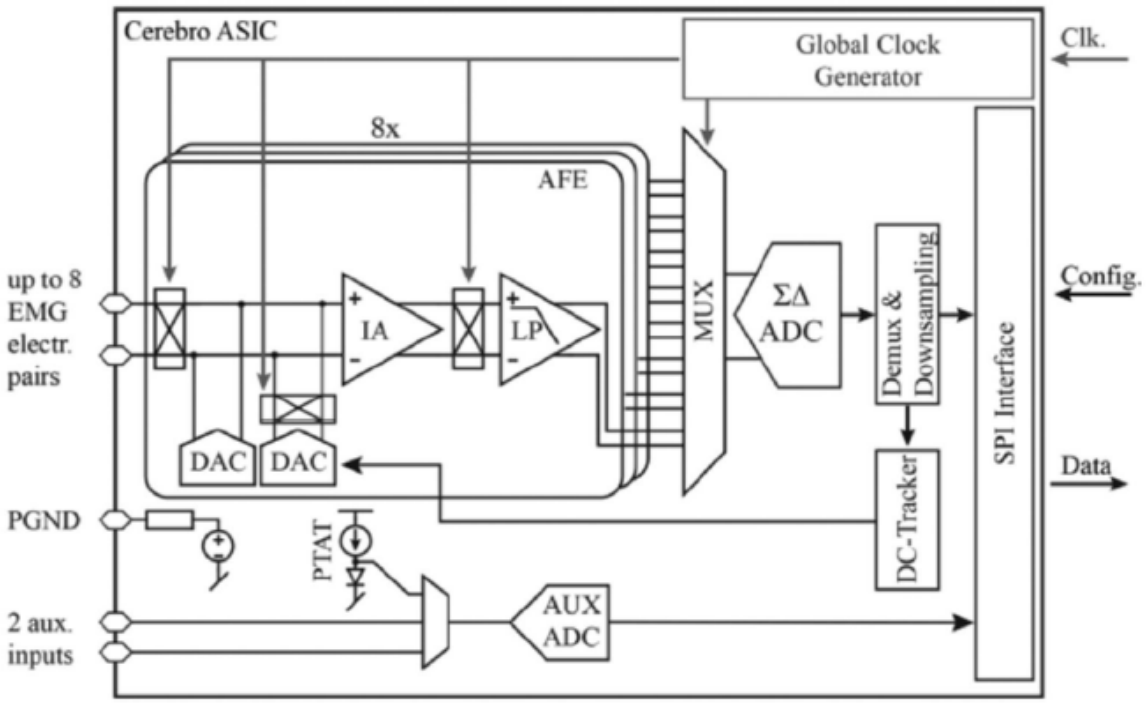


Figure 31: System Architecture of Cerebro ASIC

In order to entertain a diverse application by reusing hardware switchable classification engine built to switch between SVM and maximum likelihood classification (MLC) figure 33.

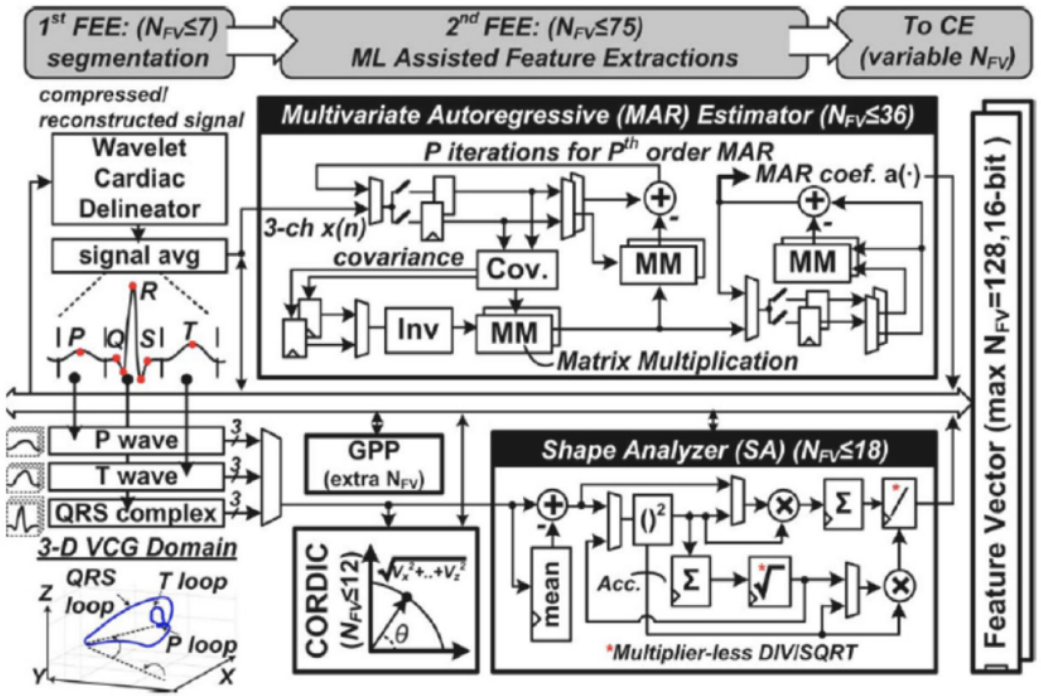


Figure 32: Two stage Processing Engine

2.6.5. Arrhythmia Detection and Biometric Authentication

ECG signal and fingerprint authentication are effective methods for wearable device figure 34, which are becoming more and more ubiquitous. The image 35 demonstrates a simplified neural network technique for user authentication and detecting the QRS complex of an ECG signal. The QRS complex is a crucial part of the ECG signal and offers significant details on the electrical activity of the heart. The design implemented using an FPGA, and testing on a single subject yields notable accuracy. It includes stages like Finite-impulse-response (FIR) Filtering, outlier discovery/removal, R-peak detection, NN-base feature extraction, and cosine similarity. Outlier detection and R-detection are used to identify aberrant ECG rhythms and abnormal ECG shapes.

2.6.6. Seizure Detection

Accuracy is crucial for ensuring safety precautions and warning those nearby. For nonlinear and isolated signals like electroencephalography(EEG), the Hilbert transform (HT) is an effec-

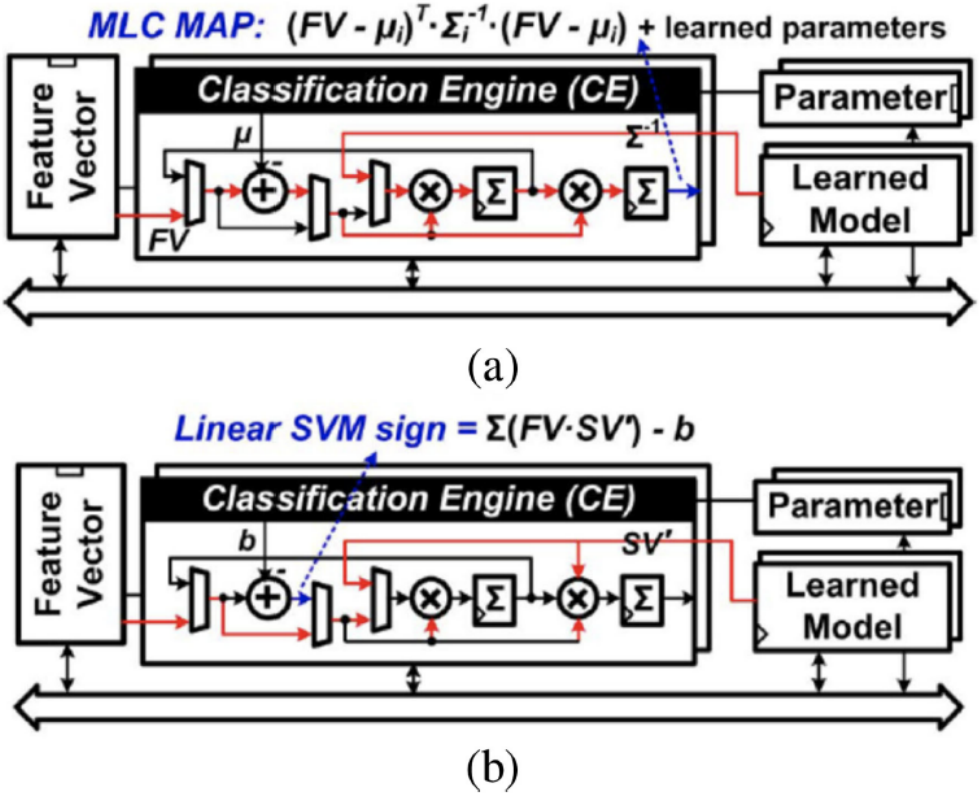


Figure 33: Switchable Classification with hardware reuse (a) MLC classification mode (b) SVM classification mode

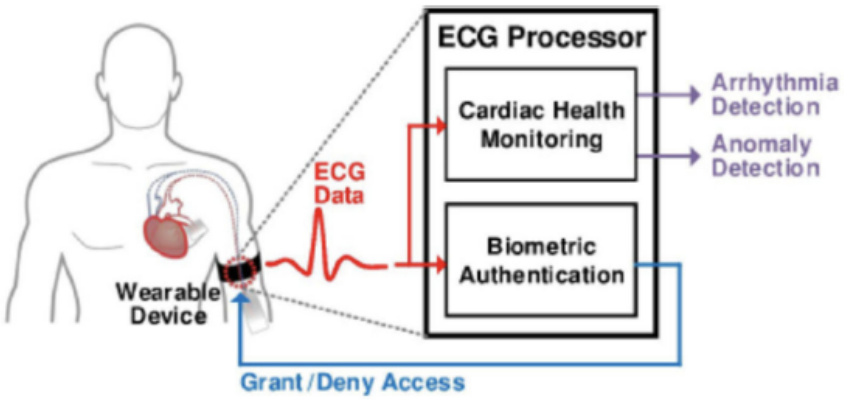


Figure 34: Wearable Device for Biometric and Cardiac Monitoring

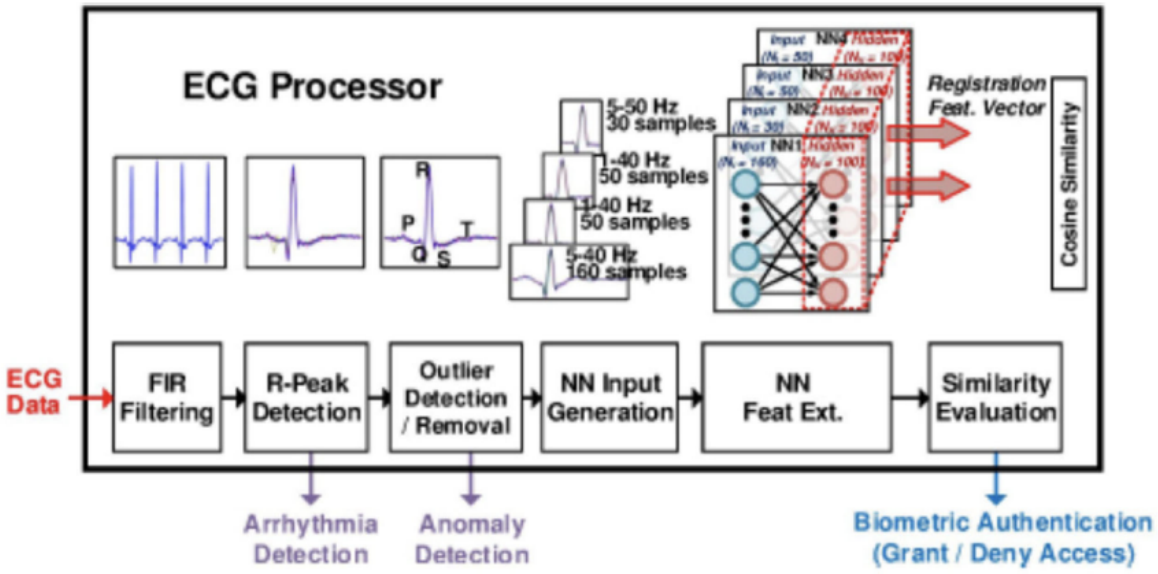


Figure 35: Flow of ECG processor

tive approach for feature extraction. EEG is a technique for recording an electrogram while observing electrical brain activity. Mean power frequency aids in reducing hardware resource requirements and speeding up processing. Using a multi-layer perceptron (MLP) neural network, EEG data can be classified as normal or abnormal situations. Figure 36 shows the system architecture for the design, along with the FIR filter utilized for HT feature extraction. There are many widely used hardware classifiers for seizure detection, notably k-nearest neighbor (KKN), logical regression (LR), Naive Bayes (NB), and Support Vector Machine (SVM). SVM engine uses pipeline architecture to improve performance for dot product. In LR and NB algorithms, serial architecture is incorporated to minimize area as shown in graphic 37.

Figure 36: Seizure detection Architecture

As opposed to more traditional techniques like SVM and KNN, the neural network has considerable promise in accuracy, performance, and classification in the seizure detection. In this design figure 38, a bit-serial-based architecture data processing unit (DPU) is adopted for neural network computation, and low power and cost are prioritized above high performance. The ALU included a multiplier for bit-serial processing, a DPU for vector arrangement that can be controlled by the state machine, and Wmem is an SRAM to store the weight of the neural network.

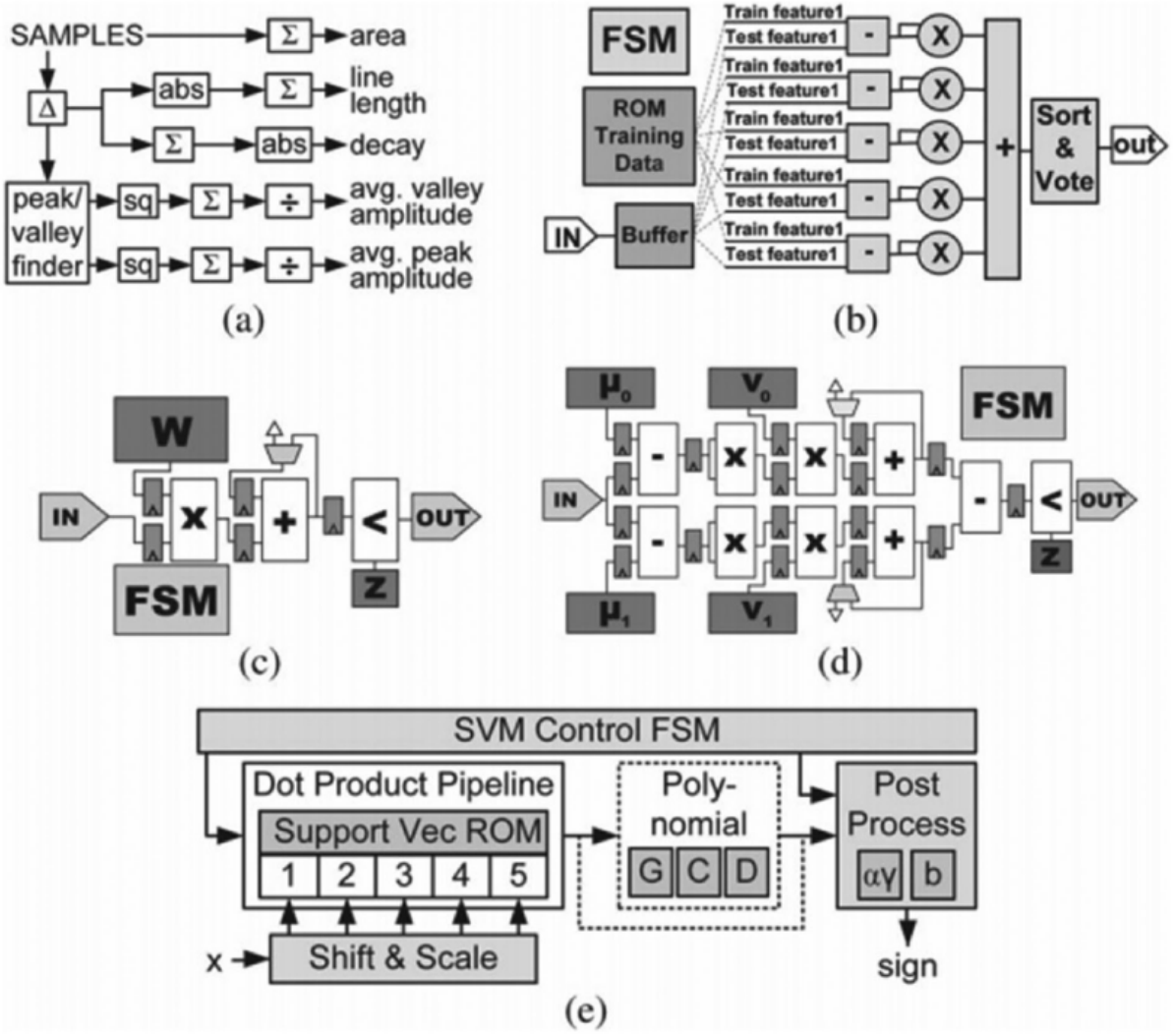
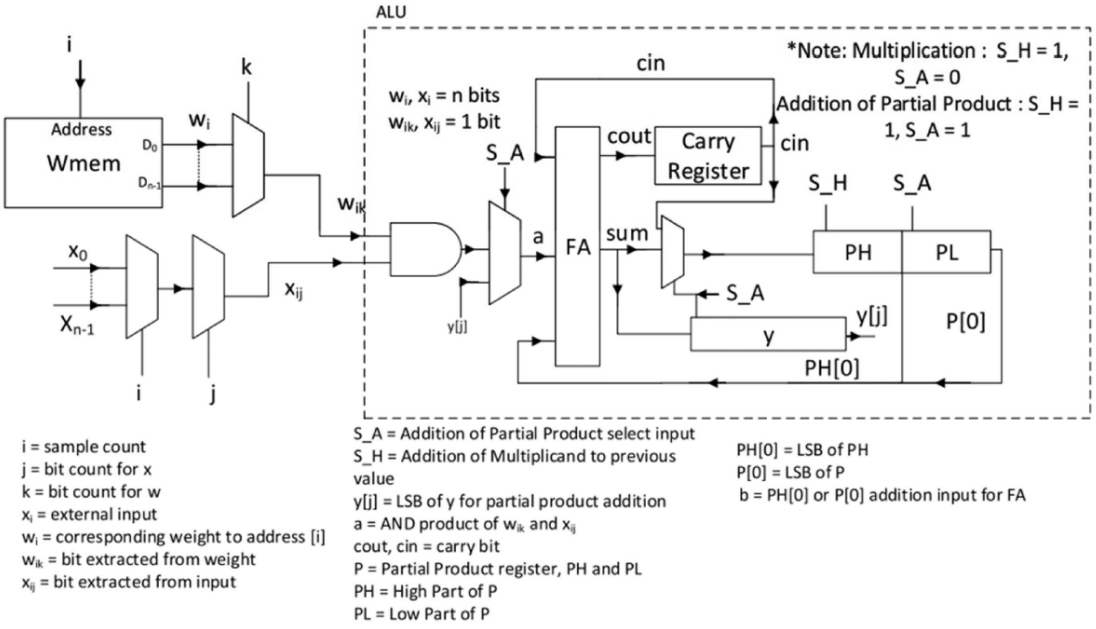


Figure 37: Hardware Classifiers and feature extraction a)Feature Extraction,b)KNN,(c)LR,(d)NB, and (e)SVM



bit	LE Cost	
	Cyclone V	Stratix IV
6	20	20
8	24	28
12	28	33

i = sample count
j = bit count for x
k = bit count for w
x_i = external input
w_i = corresponding weight to address [i]
w_{ik} = bit extracted from weight
x_{ij} = bit extracted from input

S_A = Addition of Partial Product select input
S_H = Addition of Multiplicand to previous value
y[j] = LSB of y for partial product addition
a = AND product of w_{ik} and x_{ij}
cout, cin = carry bit
P = Partial Product register, PH and PL
PH = High Part of P
PL = Low Part of P

PH[0] = LSB of PH
P[0] = LSB of P
b = PH[0] or P[0] addition input for FA

Figure 38: Data Processing Unit

2.7. Summary

Quantum Dot Cellular Automata (QCA), is a potential solution for building energy-efficient and rapid accelerators needed for AI applications like computer vision and speech recognition. The traditional CMOS technology has the drawbacks, such as an abrupt increase in cost, GHz frequency limit, and static power consumption. QCA technology employs electrostatic force and Coulombic repulsion force to propagate signals between electrons and adjacent cells, resulting in QCA wires, inverters, and majority gates. QCA clocking uses reversible logic and four phases to conserve energy and maintain cell logic, enabling information flow in the forward direction. The USE clocking scheme is a set of design standards that ensure simplicity in manufacturing, scalability, and adaptable routing. The Nand-Nor-Inverter (NNI) universal gate is introduced, which adheres to the USE clocking scheme and has a smaller footprint to enable NAND and NOR operations.

QCA Designer 2.0.3 was used to develop the QCA design simulation, and the results were displayed. QCA Mac unit introduces an 8-bit PIPO register using D flip-flops and presents two designs. The first design focuses on input/output delay matching, while the second design eliminates the need for such delays using the periodic clock signal. Each design has specific characteristics in terms of cell count, area, and latency. The second design requires initialization cycles at startup but the PIPO register has enough time to start. QCA SRAM includes details about a 2-to-4 decoder, PIPO registers used as memory cells, a 4*4 SRAM design, and the utilization of half-adders and full adders to create an 8-bit half-adder. The specifications such as cell count, area, and delay are mentioned for each component.

Approximate computing is a promising approach to achieve energy-efficient and high-performance design in various applications, including digital signal processing and artificial intelligence. The main idea behind this approach is to focus on the possible inaccurate result instead of the exact one. This can be achieved by combining different arithmetic formats with approximate arithmetic units. The accuracy constraints for specific applications can be satisfied by checking the accuracy with realistic datasets and versatile input distributions. In case the accelerators do not meet the design goal, an alternative design proposal or going back to the initial software level can be considered.
The alternative approach to processing data using memristors arranged in an array configuration, which can be utilized in neural network implementations for low-energy MAC calculations. The Processing in the Emerging Memory (PIEM) array is described in detail, where MAC operation is the output current of the sum of the memristor's conductance and the reference voltage. The PIEM array can be used for high-performance and low-energy computing in edge platforms. A fully integrated MAC near the sensor would eliminate the need for sophisticated data converters for data processing. The energy consumption issue with convolutional neural networks (CNNs) is due to their computationally intensive multiplication operations.
AdderNet is introduced as an alternative, which replaces the original convolution with adder kernel-only additions, resulting in lower energy consumption. The other components that can be used in neural networks, such as the analog memristor network and XNOR logic operation kernel. The AdderNet kernel uses bit-level operations and analog circuits, resulting in lower power consumption. The LeNet-5 on Zynq-7020, where all computation and weights are stored on board, resulted in a significant drop in energy consumption and logic resource utilization.
Vector symbolic architecture (VSA) is a computational approach inspired by the brain that can operate on both symbolic and numeric methods. VSA uses three operations: bundling, binding, and permuting to perform computations. It offers several advantages over classical architectures, such as low power requirements and the ability to use distributed representation to solve AI issues. VSA can be used in various applications, such as natural language processing and image recognition.
The various techniques and architectures used for stress detection, intelligent hearing aids, gesture recognition, ECG feature extraction, arrhythmia detection, biometric authentication, and seizure detection. The techniques employed include K-nearest neighbor, Support Vector Machine, Fast Fourier Transform, convolution neural network, Hilbert transform, and neural networks. Different hardware architectures are utilized for different applications, such as parallel pipelined design for SVM, semi-parallel architecture for KNN, and bit-serial-based architecture for neural networks, to prioritize low power and cost over high performance.
The table 2 shows how much hardware space each design requires as well as how many resources are used and how much power is consumed.

Table 2: Comparison of different Architecture resources

Architecture	Number of Gates	Applications	Power	Platform and Resource
MAC (4*4 Vedic Multiplier)	NNI gates,majority gate, USE clocking scheme, 5829 cells, two 4 bit inputs,8 bit result, multiplier (four 2-bit Vedic multipliers,three 4 bit adder, and OR gate)	-	9.43$e^{-}3$ eV per cycle	QCA Designer 2.0.3 13.27μm^2
4*4 SRAM	NNI gates,majority gate,USE clocking scheme, 4293 cells,PIPO register, and multiplexer	-	-	QCA Designer 2.0.3 6.81 μm^2
64-Quadrature Amplitude Modulation(QAM)	For demodulation 64-QAM keying signals, 32-bit floating, 16-bit fixed-point arithmetic, Bit Error Rate(BER) as accuracy metric	Approximate Computing, signal filtering, telecommunication	-	Xilinx Vivado Zynq Ultrascale ZCU106
Convolutional Neural Network(CNN)	4 convolution and 2 fully-connected layers(132K parameters)	Approximate Computing, image processing,	-	Zynq-7020 FPGA
8-bit or 16-bit Adder Convolution network(AdderNet)	2 adders or one-Comparator-one-Adder(1C1A) ,one Multiplixer	Image Classification algorithms(ResNet,VGG)	1.34 W	FPGA on board Xilinx Zynq UltraScale+MPSoC ZCU104
8-bit or 16-bit Convolution network	One multiplier, Serial-Shift-Register for shift operation, one multiplexer	Image Classification algorithms(ResNet,VGG)	2.57 W	FPGA on board Xilinx Zynq UltraScale+MPSoC ZCU104

Table 2: Comparison of different Architecture resources

Architecture	Number of Gates	Applications	Power	Platform and Resource
Processing in Emerging Memory(PIEM)	Memristor based crossbar array	Edge Devices(mobile phones, drones)	64.4mW	54*108 memristor crossbar 180-nm CMOS technology
Vector Symbolic Architecture	Addition and Multiplication	Information retrieval ,Vector models for Natural Language Processing(NLP),Robotics	-	-
Support Vector Machines (SVM)	Approx. 4 Input,16 accumulator,4 Memory Unit,4 Multiplier,4 adder,one adder,one Comparetor ,one output	Stress Detection	39mW @1V	ASIC $0.17mm^2$@65nm
K-nearest neighbor (KNN)	Approx. ROM Training Data ,one input buffer 4*16,4 accumulator, 4 multiplier, one adder, one sort and vote,one output	Stress Detection	77mW @1V	ASIC $0.3mm^2$ @65nm

Table 2: Comparison of different Architecture resources

Architecture	Number of Gates	Applications	Power	Platform and Resource
Neural Network(NN)	Approx. I/O Data Buffer, Hamming Window, FFT MEM, Model MEM, Global CTRL ,Processing Cluster(Bias MEM,(CNN Ctrl,FFT Ctrl,CNN Data,FFT Data) ,One Multiplixer,Weight Index,Weight LUT,PE Array),NN Input Buffer,Activation MEM,	Hearing Aid	2.17mW@0.6-0.9V	ASIC $4.2mm^2$ @40nm
Support Vector Machine(SVM)	CEREBRO,ARM MCU,Sensors	EMG based Gesture Recognition	29.7mW	Microcontroller Unit (MCU)

Table 2: Comparison of different Architecture resources

Architecture	Number of Gates	Applications	Power	Platform and Resource
Support Vector Machine(SVM)	Segmentation (average signal, heart signal P,T,QTS complex wave separation), ML Assisted Feature Extractions(shape Analyzer, Multivariate Autoregressive(MAR) Estimator) Classification Engine(CE)(Multilabel Classification(1 adder,2 multipliers, 2 accumulators),SVM classification(1 adder, 2 multipliers,2 accumulators) , Feature Vector	ECG Arrhythmia	48.6 μW @0.5V	ASIC 4.99mm^2 @90nm
Neural Network	Input ECG data ,FIR Filtering,R-Peak Detection,Outlier Detection/Removal, NN Input Generation, NN Feature Extraction, Similarity Evaluation	ECG Arrhythmia and Biometric Authentication	Arrhythmia 0.83 μW @0.51 V , Authentication 1.06 μW @0.55V	ASIC 5.88 mm^2 @65nm

Table 2: Comparison of different Architecture resources

Architecture	Number of Gates	Applications	Power	Platform and Resource
KNN,LR,NB, SVM	Feature Extraction(ECG SAMPLES,total sum, Peak/Valley finder, square root, sum, division, average valley amplitude, average peak amplitude), KNN(Input data, buffer,test feature, ROM training data,train feature, 5 accumulator, 5 multiplication, one adder, sort and vote, output value), Logistic Regression(input ECG data,one multiplier, one adder, one adder, one comparator, one output), Naive Bayes(input ECG signal,4 multiplier,2 adder, 3 subtract, one comparator, one output, SVM,ECG data, Shift and Scale, Dot product Pipeline),Support Vector(ROM, Polynomial, Post Process)	Epilepsy Seizure Detection	37nW @1V	ASIC 0.008 mm^2 @65 nm

Table 2: Comparison of different Architecture resources

Architecture	Number of Gates	Applications	Power	Platform and Resource
Neural Network	ECG data signal, HT Features Extraction, MPF Feature Selection,MLP Classifier(lethal or normal)	Epilepsy Seizure Detection	159.7 mW	FPGA FF:357, 4LUT:5355

3. Tools and working environment

3.1. Tensorflow [1]

An open-source framework called TensorFlow is used to train neural networks using machine learning methods. It offers thorough documentation examples for specifying network layers, activation functions, layer numbers, and various optimization samples, among other things. Installing TensorFlow and Python would be both of the primary requirements, specifically, for the system. Python is essential for working with and developing machine learning models. To determine trainable parameter,performance and power requirements, a neural network was trained on two distinct datasets: the mnist dataset and the 6-D sensor dataset.

3.2. Zynq [10]

Verilog and VHDL for FPGA programming can be automated with the assistance of the Zynq framework. The structure it uses is comparable to TensorFlow for defining neural network layers and activation functions. For the experiment, the Zybo board xc7z020clg400-1 figure 41 was utilized, but it can synthesized and implemented in any target FPGA. Assume you need to program a different FPGA board. In such a case, specify the board number, and the package will generate hardware components corresponding to that specific board. To program the Avenet Zedboard XC7z020clg484-1, for example, one has to initialize the board number in the object's argument zynet.makeXilinxProject("project-name", "board-number"). Understanding the neural network algorithm's underlying working is crucial. The neural network shown in the image 39 is fully connected, which indicates that each neuron is entirely coupled to the one before it. The first layer is the input layer, the output layer is the last layer, and the layers in between are referred to as the hidden layer. A hidden layer comprises neurons, and the quantity of neurons in each layer can be changed as needed. A neural network can include a variety of hidden levels, in this study, the network is built using different hidden layers.

3.3. MNIST [6] Dataset

To study the hardware requirements for small scale AI/ML, a well known data set was used. The MNIST[6] data set, a widely used data set for machine learning tasks, is composed of 10,000 test images and 60,000 handwritten images for training, and every digit from 0 to 9 is saved as a 28 by 28 pixel greyscale image. Considering all of the sensor inaccuracy, the MNIST data set is rather clean in comparison to the 6D sensor data. The size of the mnist dataset is significant

for machine learning tasks since a large number of training samples are needed for machine learning. The MNIST [6] dataset figure 40 was used for the experiment, and as each image contains 28*28 pixels, the first input layer comprises a total of 784 pixels(e.g 28*28=784).

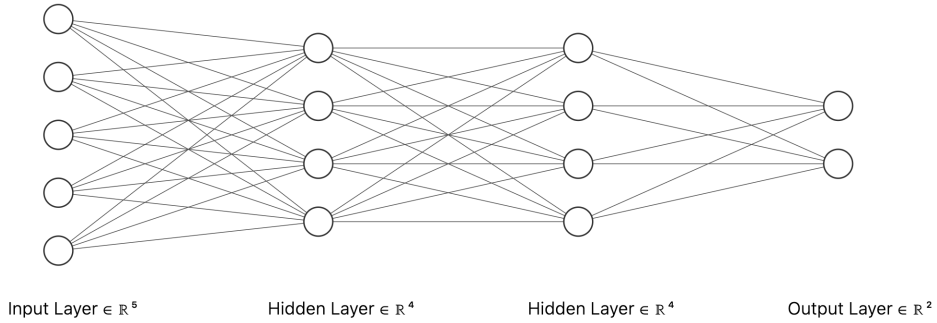


Figure 39: Neural Network

Figure 40: MNIST dataset[13]

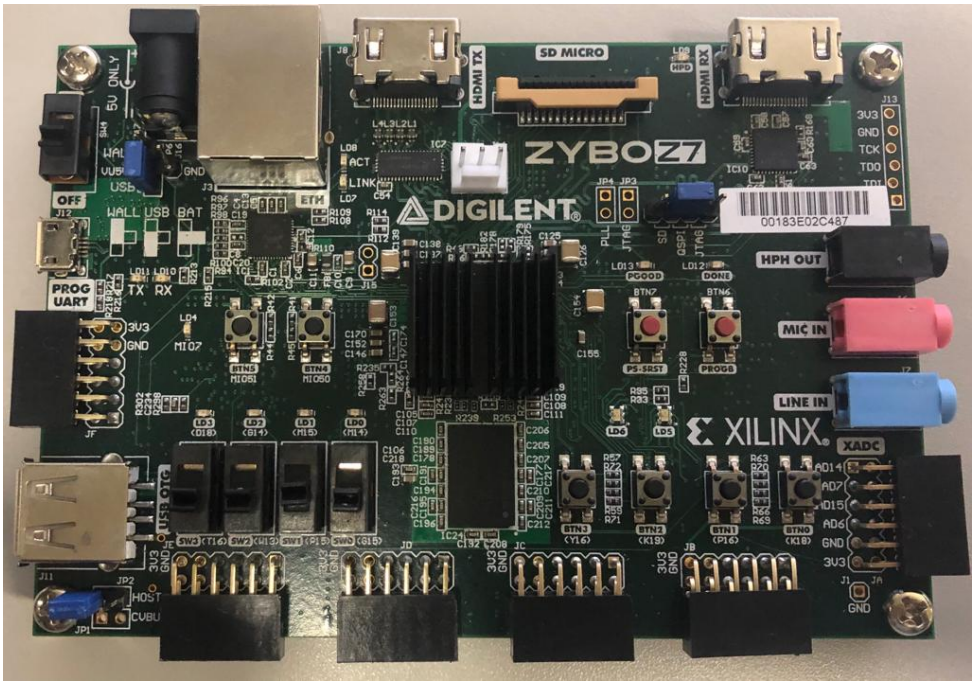


Figure 41: Zybo Z7

4. Methadology

The layer denotes the total number of hidden layers the neural network will have during training, and each layer comprises of several neurons. Depending on the type of network structure required, different numbers of layers and neurons might be included. The following describes the system architecture, layers and neurons function.

4.1. System Architecture

The data that goes into the neural network values are normalized to 1 to -1 and the numbers are represented in fixed point representation. The fixed point is adjustable since we can choose how many bits go into the integer and fractional portions of the representation. For instance, the four-bit integer and four-bit fraction portions of number 10.2342, as indicated in the illustration 42, would be 1010.0011. The more bits utilized for the fractional part, the higher the precision; however, it uses a lot of power, and you need enough bits to represent the integer part. There is always a fractional element in all numbers, and the bias and weight values can be either positive or negative. The construction of digital circuits to calculate activation functions is a heavy resource job, making it impractical to represent hundreds of neurons. To calculate the activation function, we precalculate sigmoid $(1/1 + e^{-x})$ values of x for different values of x. We then acquire an awareness of the range of x and represent it as a signed magnitude representation. After that, we save these values as ROM, commonly known as a Lookup Table

(LUT), a common software and digital design technique. A precalculated value, either a sigmoid or a relu activation, will be returned by the LUT when we provide it with the address bus x. To be clear, a lookup table is a digital design approach rather than the fundamental building component of an FPGA.

The block design figure 44 describes the system's general layout and how its parts interconnect together. Each neuron layer's weights and biases are kept in a separate folder as a memory initialization file. The crucial aspect is that, in order to improve time performance and utilization of resources, all inputs must be sent sequentially and one at a time. All neurons are connected to the same bus, which carries the weight values, and weight values can be sent to a specific neuron by adjusting the layer and neuron numbers. The Zynq processing system's GP0 (general purpose 0) interface is linked to the axi-lite interface, which is used to configure the output of the network. The axi-lite interface can access a variety of Zynet register files, and it can read the address from the output register to access the output of every neuron in the output layer. The first read relates to the output of the first neuron, and to access subsequent data, the output neuron number is automatically increased with each read. After gaining access to the weight and bias values from the trained network, layer number, and neuron number registers write those values into the weight and bias register. The status register provides the network status for valid output data and interrupt signals associated with the bit in the status register. For setting up a soft reset for the network control register in use.

Integer Part (10):

- Decimal 10 in binary: 1010

Fractional Part (0.2342):

- Multiply the fractional part by 2 and record the integer part of the result:
  - $0.2342 \times 2 = 0.4684$ (integer part: 0)
  - $0.4684 \times 2 = 0.9368$ (integer part: 0)
  - $0.9368 \times 2 = 1.8736$ (integer part: 1)
  - $0.8736 \times 2 = 1.7472$ (integer part: 1)
  - $0.7472 \times 2 = 1.4944$ (integer part: 1)
- The fractional part in binary is 0011.

Combine Integer and Fractional Parts:

- Integer part: 1010
- Fractional part: 0011
- The fixed-point representation is 1010.0011.

Figure 42: Fixed point representation of 4 bit integer portion and 4 bit fractional portion

4.2. Neuron

The figure 43 shows the working principle of neuron and layer structure, and how it is connected to the next layer. The equation 11 illustrates how a network operates; where w stands for weight and x for the network's input. The weights are stored in weight memory and initialised as ROM followed by multiplied and accumulated (MAC). The input value x is multiplied by the corresponding weight value w, and the result is saved in the variable sum. This process is repeated until all input has been multiplied by weight and add bias value to it. At the end of this multiplication, we need to apply activation function. Without the activation function, the network would have been a collection of linear layers layered on top of one another, making it impossible for it to learn from the variety of input data. The weight and bias values are created using a tensorflow-based implementation in software and hardware these values saved as a memory initialization file. It is crucial to keep in mind that weights are shared on the common bus while number allocated to distinguish between layers and neurons. The number distinguishes between distinct layers of neurons, whereas the layer and neuron number registers are responsible for proper ordering of layer and neuron number.

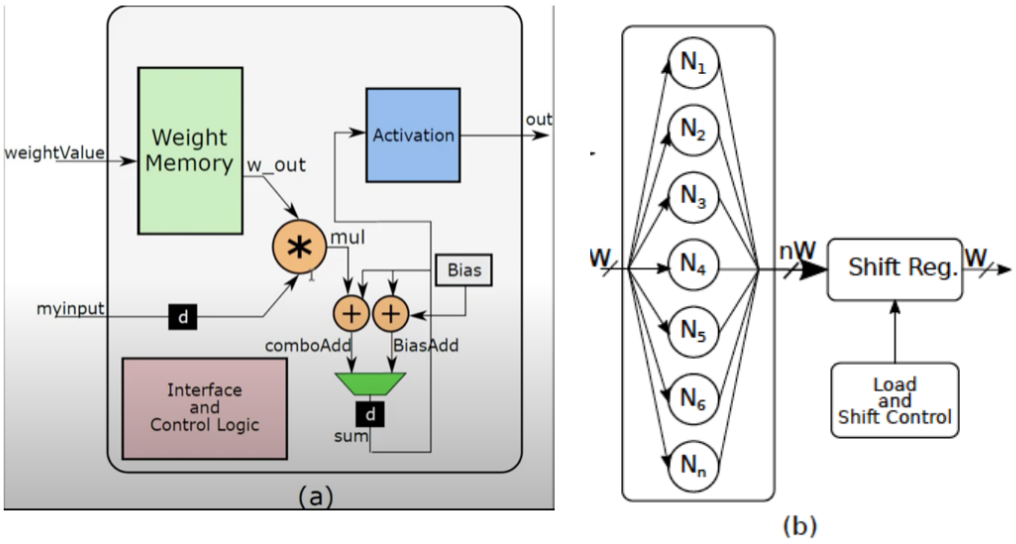


Figure 43: (a)Neuron Architecture (b) Layer Architecture

$$y = wx + b \tag{11}$$

4.3. Layer

In fully connected network requires the connection of every neuron from the previous layer. Data from each layer is initially kept in the shift register between layers, and it is transferred to the following layer once every clock cycle, as shown in Figure 43(b). While data initialization is handled by the AXI-Lite interface, communication between various architectural components such as memory, zynet, zynq processing system is managed by the AXI-stream interface.

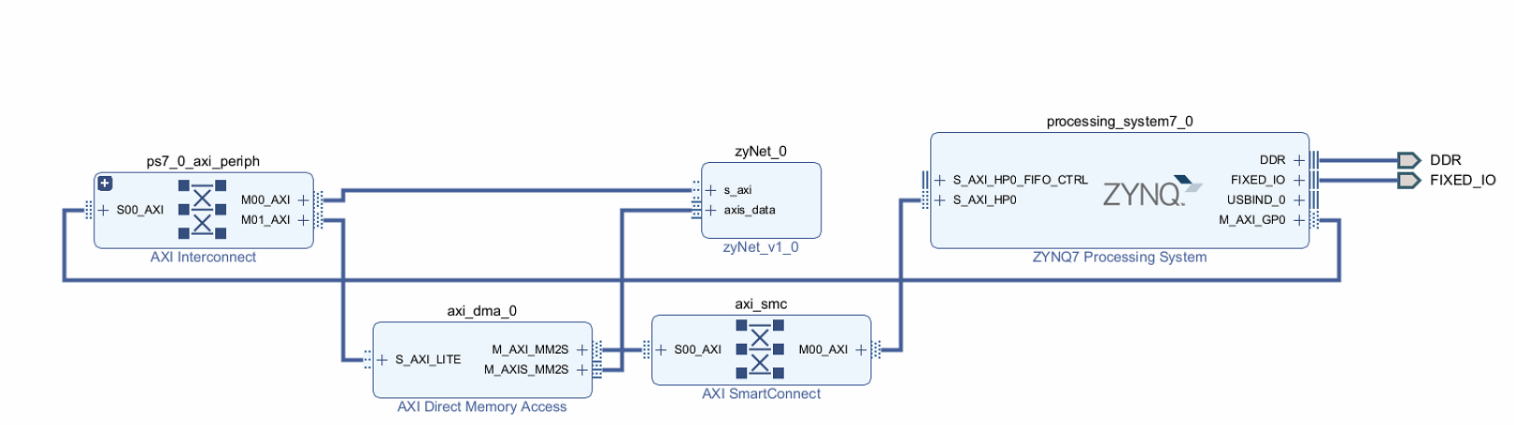


Figure 44: System Design

5. Implementation of Neural Network with MNIST dataset

5.1. Neural Network 0

The graphic figure 45 depicts the hardware design on an FPGA board; the appendix C has more information.

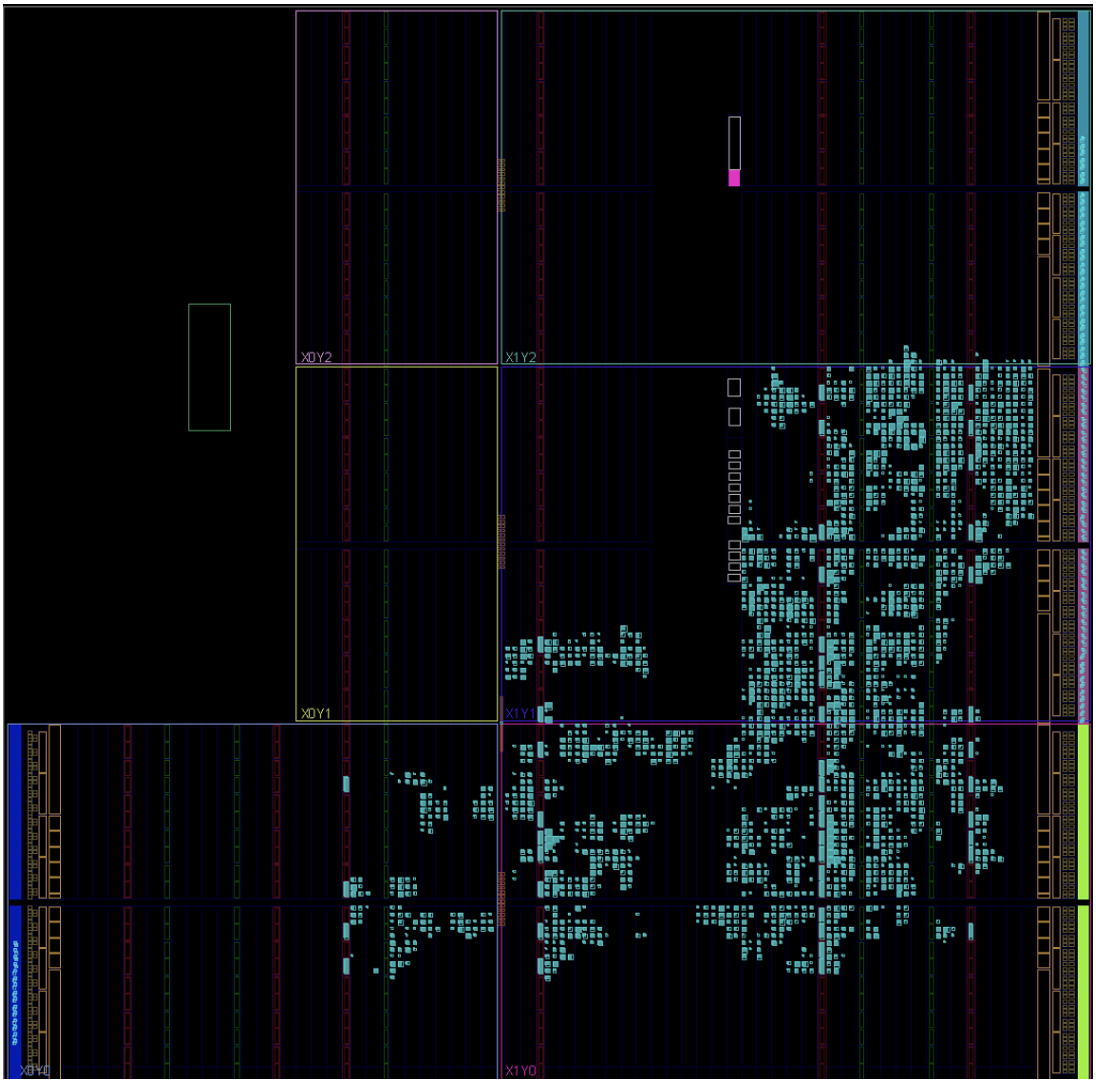


Figure 45: Hardware mapping of network 0 with Mnist dataset

5.2. Neural Network 1

The graphic figure 46 depicts the hardware design on an FPGA board; the appendix D has more information.

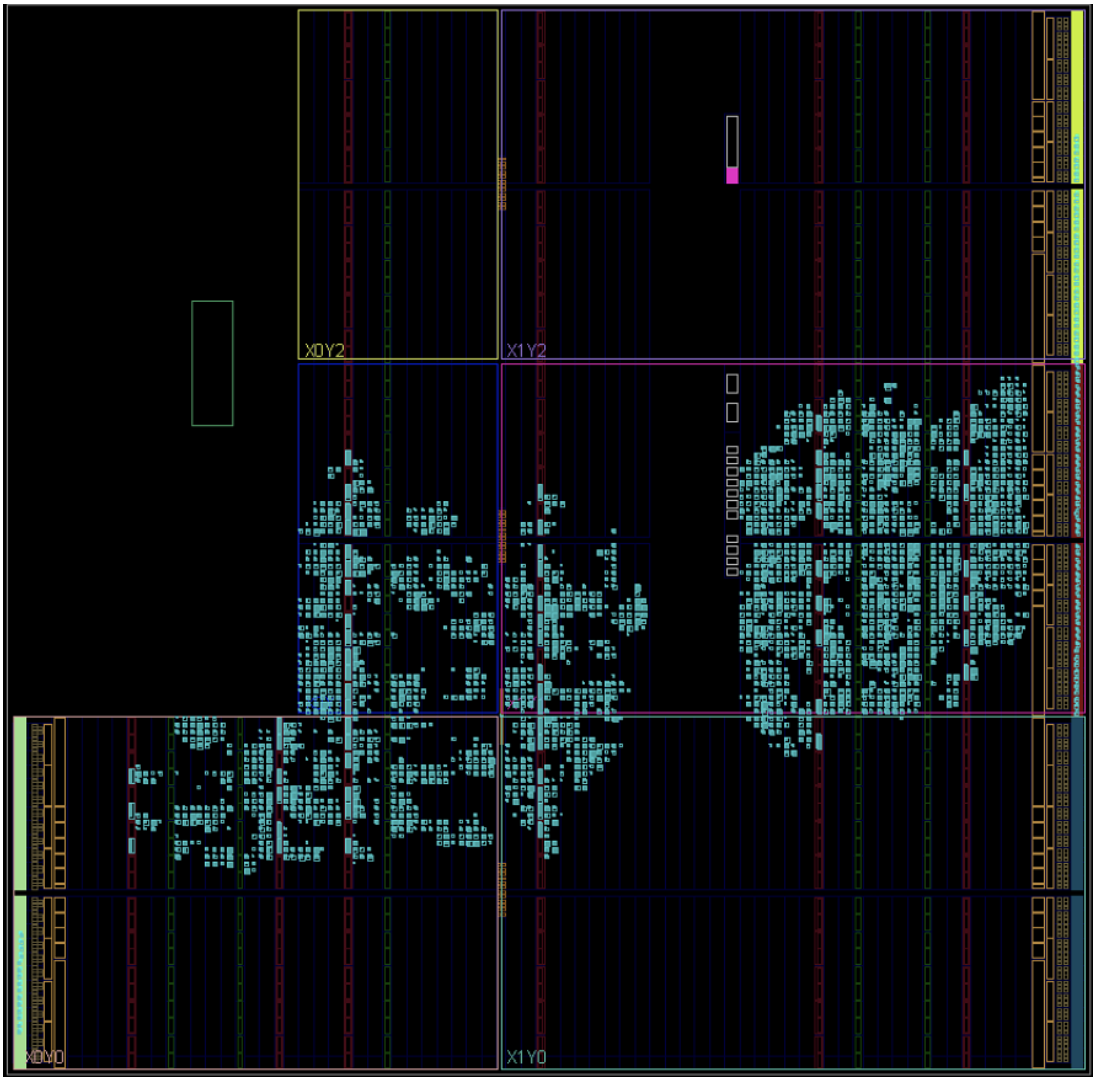


Figure 46: Hardware mapping of network 1 with Mnist dataset

5.3. Neural Network 2

The graphic figure 47 depicts the hardware design on an FPGA board; the appendix E has more information.

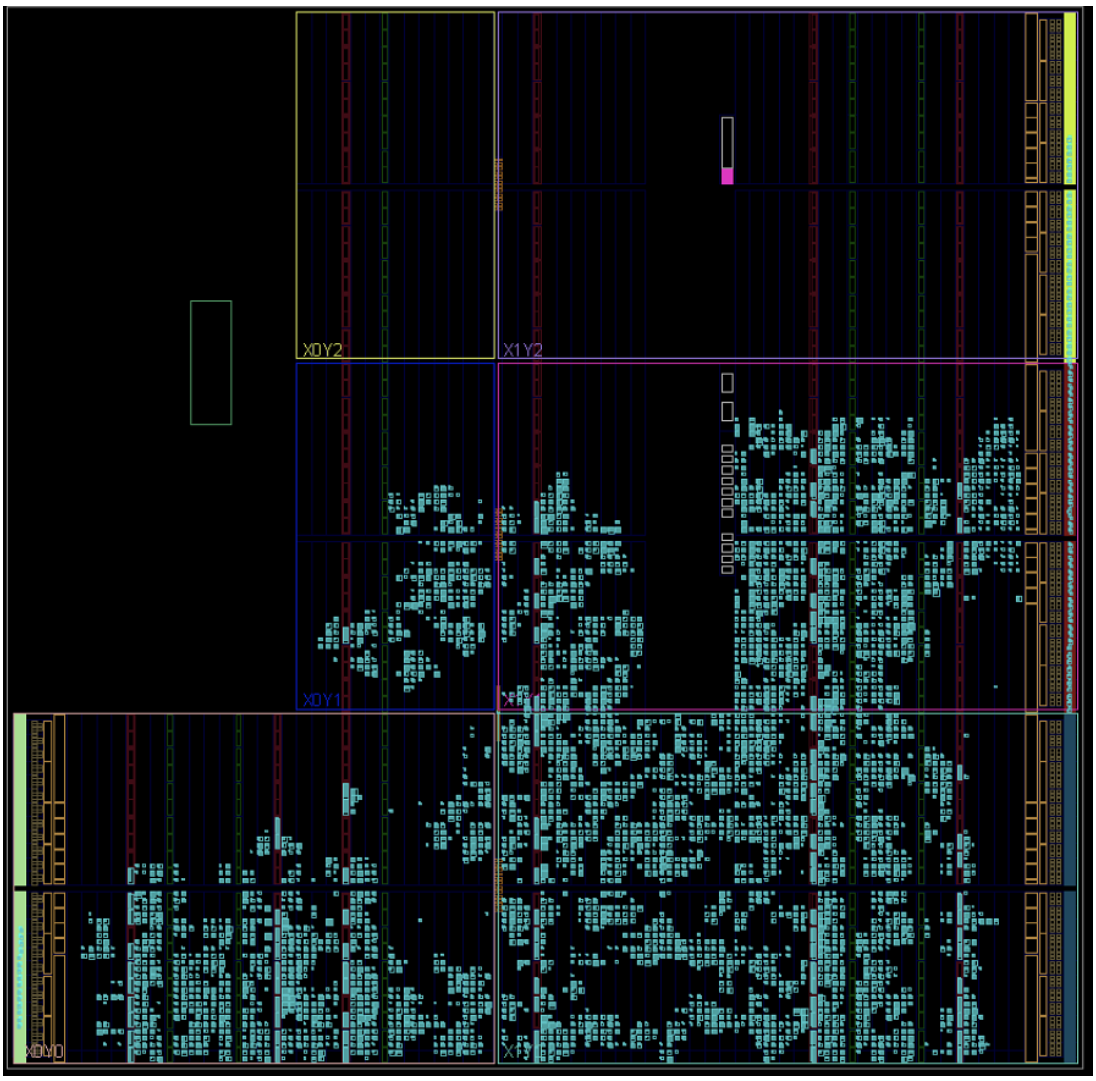


Figure 47: Hardware mapping of network 2 with Mnist dataset

5.4. Neural Network 3

5.4.1. TensorFlow Neural Network Implementation

The figure 48 follows is a tensorflow-based implementation of a deep neural network; the network's prerequisite libraries are included in its initial few lines. After that, it downloads a collection of 60,000 handwritten MNIST digit images and separates them into training sets

and test pictures. This separation of the training and testing sets ensures that the network generalises to unseen datasets, as it would otherwise overfit to the training dataset. Data normalization is required for feature transformation to ensure that data has a similar scale. Additionally, this stops some features from being dominant over others during learning of the machine learning model. The network itself has six layers, five of which are hidden layers and made up of 50, 50, 50, 50, and 10 neurons with sigmoid activation. The last layer consists of 10 neurons that suggest which neuron is giving the maximum value. The model is trained on 25 epochs with adam optimizer, and sparse categorical cross entropy loss. The final couple of lines of the script gather the weight and bias values from the second layer to the last layer—not from the first layer because it does not contain weight and bias value—and store them in a text file. The weight and biases files are obtained using a tensor flow methodology, while it's probable that there are further methods [9]. We require these weights and biases for hardware implementation, thus we obtain these values from software implementation. The trained neural network's weights and biases will be applied in the following Zynq implementation.

5.4.2. Zynet Neural Network Implementation

The tensorflow-based implementation figure 48 of neural networks and the Zynet implementation figure 49 of deep neural networks are comparable. Installing a necessary Zynet module is the first step, followed by defining the layer structure. The network comprises six layers; the first five hidden layers are made up of 50, 50, 50, 50, and 10 neurons with sigmoid activation function, while the final layer is made up of 10 neurons with hardmax activation. The hard max activation function evaluates the output of each neuron and determines which neuron is providing the highest value. Since TensorFlow-based and Zynet are completely linked networks, which that density parameter represents, every neuron in the previous layer is entirely connected to the subsequent layer. For hardware implementation, we needed the weight and bias values from the Tensorflow implementation, hence we already saved them in the.txt file. We need to generate hardware on the weight and bias value, we provide the file as input to the generateArray method. The compile function requires three more parameters since hardware implementation offers bit-level precision.
Zynet specifies the number of bits used for the integer part of input and weight values using fixed point representation, inputIntSize. WeightIntSize specifies bit as the integer portion; the other bits are fractional portions. The depth of the activation function, which corresponds to the sigmoid size, is implemented as the LUT specified depth 10, which are 1024 values, and when depth is 5, values are 32 (e.g $2^{address\text{-}bits}$).. The increase in sigmoid depth are directly proportional to accuracy but inversely proportional to resource consumption. The zynet.model() method creates a deep neural network object, and model.add() adds a new layer to the network. The method zynet.makeXilinxProject() generates Xilinx project with deep neural network(DNN) as

```
import tensorflow as tf
import json
import time
mnist = tf.keras.datasets.mnist
(x_train,y_train),(x_test,y_test) = mnist.load_data()
x_train=tf.keras.utils.normalize(x_train,axis=1)
x_test =tf.keras.utils.normalize(x_test,axis=1)



mdl = tf.keras.models.Sequential()

mdl.add(tf.keras.layers.Flatten())
mdl.add(tf.keras.layers.Dense(50,activation=tf.nn.sigmoid))
mdl.add(tf.keras.layers.Dense(50,activation=tf.nn.sigmoid))
mdl.add(tf.keras.layers.Dense(50,activation=tf.nn.sigmoid))
mdl.add(tf.keras.layers.Dense(50,activation=tf.nn.sigmoid))
mdl.add(tf.keras.layers.Dense(10,activation=tf.nn.sigmoid))
mdl.add(tf.keras.layers.Dense(10,activation=tf.nn.sigmoid))
mdl.compile(optimizer ='adam',
loss='sparse_categorical_crossentropy',
metrics = ['accuracy'])
start_time = time.time()
mdl.fit(x_train,y_train,epochs=25)
(val_loss,val_accuracy) = mdl.evaluate(x_test,y_test)
end_time = time.time()

inference_time = end_time - start_time
print(f"Inference time: {inference_time} seconds")

num_params=mdl.count_params()
print(f"Number of parameters:{num_params}" )
weightLst =[]
biasLst =[]

for i in range (1,len(mdl.layers)):
    weights = mdl.layers[i].get_weights()[0]
    weightLst.append((weights.T).tolist())
    bias =  [[float(b)] for b in mdl.layers[i].get_weights()[1]]
    biasLst.append(bias)
data ={"weights":weightLst,"biases":biasLst}
f = open('weightsandbiases.txt',"w")
json.dump(data,f)
f.close()
```

Figure 48: Tensorflow Neural network

top module. It takes two parameters; first parameter is project name, second is FPGA board which you need to program which is Z-board, and zynet.makeIP() package the Deep neural network(DNN) in IP-XACT format. The zynet.makeSystem() makes block design for a specified IP block, IO peripherals, AXI interface, Zynq processor system, and DMA controller. When we run neural networks using the given parameter figure 49, the hardware footprint is depicted in figure 50.

```
from zynet import zynet
from zynet import utils
import numpy as np
def MnistZynet(dataWidth,sigmoidSize,weightIntSize,inputIntSize):


    mdl = zynet.model()

    mdl.add(zynet.layer("flatten",784))
    mdl.add(zynet.layer("Dense",50,"sigmoid"))
    mdl.add(zynet.layer("Dense",50,"sigmoid"))
    mdl.add(zynet.layer("Dense",50,"sigmoid"))
    mdl.add(zynet.layer("Dense",50,"sigmoid"))
    mdl.add(zynet.layer("Dense",10,"sigmoid"))
    mdl.add(zynet.layer("Dense",10,"hardmax"))


    weightArray =utils.genWeightArray('WeigntsAndBiases.txt')
    biasArray = utils.genBiasArray('WeigntsAndBiases.txt')

    mdl.compile(pretrained ='Yes',
    weights =weightArray,
    biases=biasArray,
    dataWidth=dataWidth,
    weightIntSize=weightIntSize,
    inputIntSize=inputIntSize,
    sigmoidSize=sigmoidSize
    )

    zynet.makeXilinxProject('ai','xc7z020clg400-1')
    zynet.makeIP('ai')
    zynet.makeSystem('ai','block_design')
if __name__=="__main__":
    MnistZynet(dataWidth=8,sigmoidSize=10,weightIntSize=3,inputIntSize=1)
```

Figure 49: Zynet Neural network

5.4.3. Vivado Project

Following the execution of the Zynet implementation figure 49, a project with the name "ai" will be created in Vivado, and this is demonstrated figure 51 in the project's content. The script assumes that Vivado is in your computer's files because it would produce an error message otherwise. The number of layers in software implementation matching hardware implementation is another crucial factor. If the count of neurons in each layer varies, in the

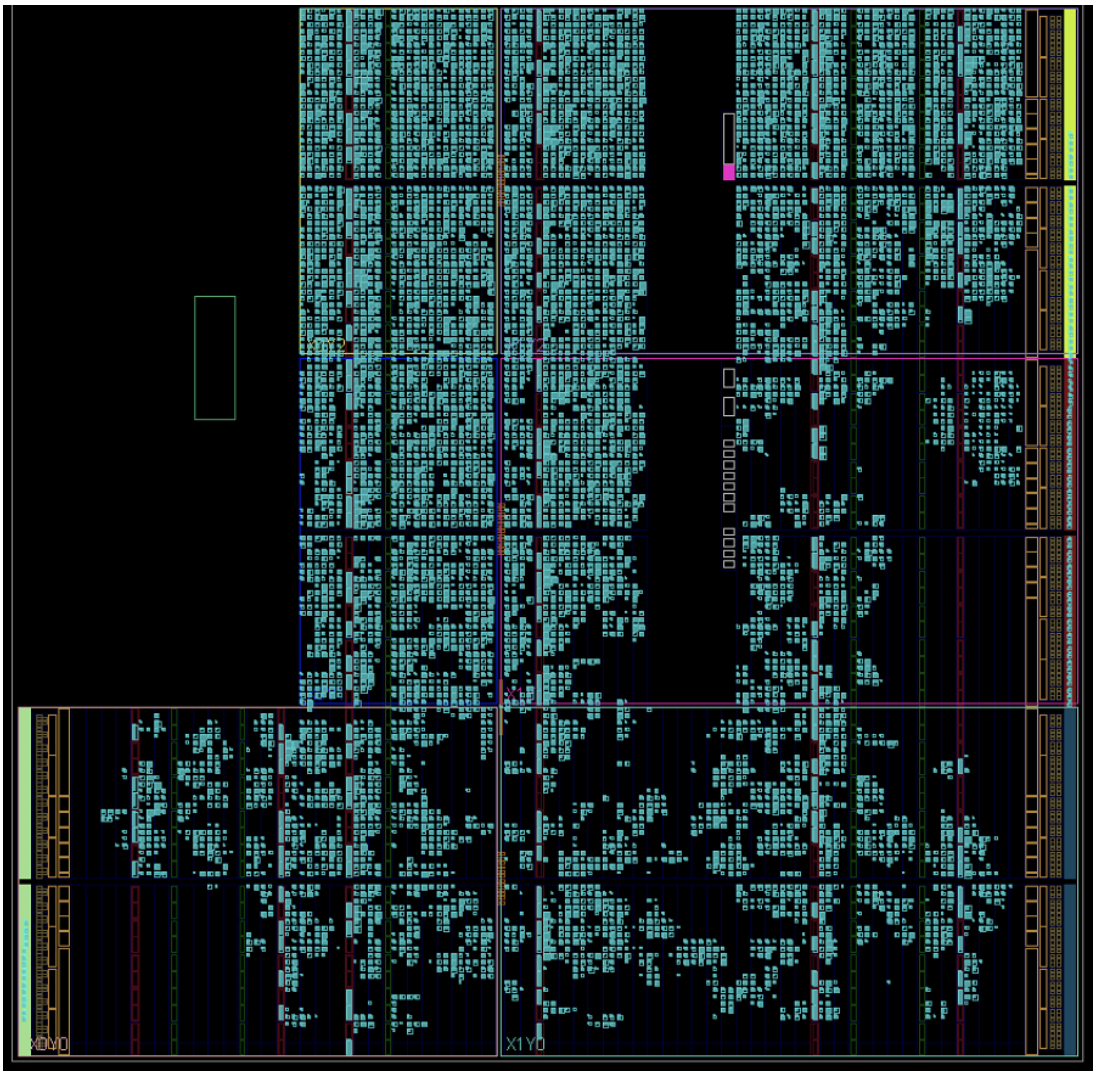


Figure 50: Hardware mapping of network 3 with Mnist dataset

hardware and software implementation, the output will be flawed. We require the weights and bias values of each layer for hardware fabrication, which is the reason why it will lack such values for that specific layer.

The first file include.v contains necessary information; including the neural network's number of layers, neurons per layer, if the model has been trained, the kind of activation, sigmoid size, and data width. Second, the zyNet module is divided into a number of smaller files. For instance, the file axi-lite-wrapper.v contains representation for communication between layers. Many registers allow neurons and other system components to communicate effectively. In the software implementation of Zynet, we have defined 6 layers, each of which corresponds to a specified number of layers and neurons in each layer figure 49. Each of the first four levels in this example contains 50 neurons, as described by our definition of the first four layers. Multiply and accumulate (MAC) 43 (a) steps are implemented in each neuron file; these include weight initialization as ROM, multiplication, and adding bias. As there are ten neurons in the final layer but only one output will be the outcome, the module maxFinder.v compares the output of one neuron with the other neuron values with help of output register.

5.5. Resource utilization when using MNIST dataset

The table 3 displays the number of resources used when neural networks are trained on MNIST datasets. The MNIST dataset was used to train four neural networks, and the table 3 shows the overall resource usage for each network. Network 3 consumes the most power (193.353 W), has 47520 trainable parameters, and has a 93 percent accuracy rate. The neural network 0 consumes the least amount of power, has 16020 trainable parameters, and achieves 92 percent accuracy. With a maximum accuracy of 99 percent, the Neural Network 1 utilises 79.8 W power and uses the following resources: LUT 7189, FF 3799, BRAM 30 and, BUFG (Global buffer) 1. For detecting 100 test photos, the highest prediction accuracy is 99 percent; the total prediction time is 1 minute and 7 seconds. The anticipated time for one photograph when using the Zybo board is approximately 0.67 seconds.

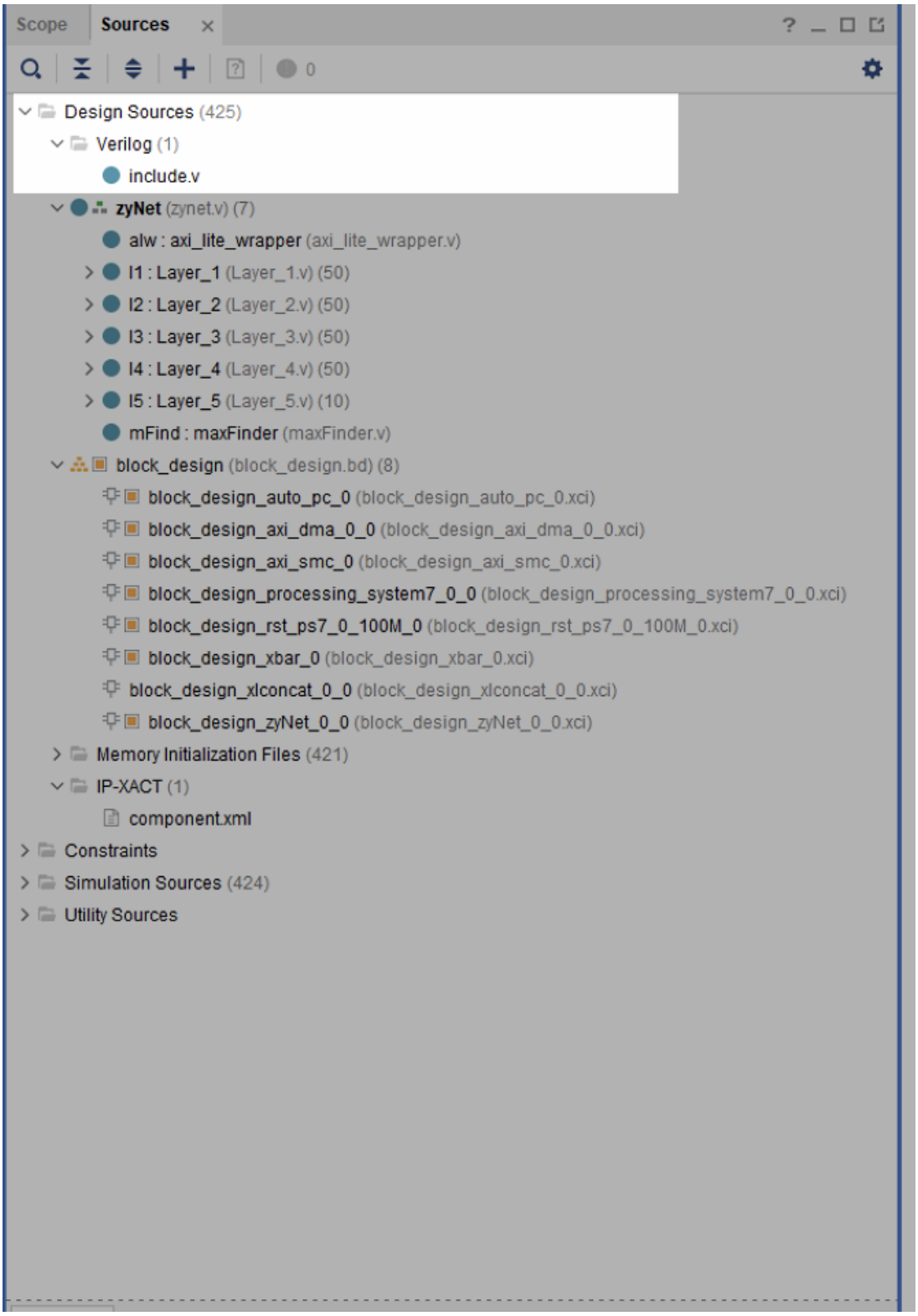


Figure 51: Vivado Project

Table 3: Resources utilization Mnist dataset

Architecture	Dataset	Number of parameters	Applications	Power	Platform and Surface Area	Accuracy
Neural Network 0	MNIST	16020	Computer Vision	60.418 W	Zynq-7000,FPGA LUT 6111,FF 3275,BRAM 22.50, IO 97, BUFG 1	92 percent
Neural Network 1	MNIST	24490	Computer Vision	79.8 W	Zynq-7000,FPGA LUT 7186,FF 3799,BRAM 30, IO 97, BUFG 1	99 percent
Neural Network 2	MNIST	33460	Computer Vision	107.51 W	Zynq-7000, FPGA LUT 11938,FF 6079,BRAM 44.5, IO 97, BUFG 1	98 percent
Neural Network 3	MNIST	47520	Computer Vision	193.353 W	Zynq-7000,FPGA LUT 24711,FF 12727, BRAM 77, IO 97, BUFG 1	93 percent

6. Implementation of Neural Network with 6-D Sensor dataset for city siegen

6.1. 6-D Sensor Dataset

To make driving enjoyable and reliable, the automotive industry uses a multitude of sensors,cameras to gather data for various components of the car. These databases can be used for a variety of things, such as tracking the health of vehicles, giving warning signals to avoid collisions, providing users with feedback about their surroundings, or enabling autonomous driving. The temperature value, the gyroscope movement in three dimensions, and the acceleration in three dimensions make up the 6-D sensor, which measures data in six dimensions.

6.2. Data acquisition and machine learning analysis

The data on the city of Siegen was gathered for the experiment using an Arduino board. The data-collecting arrangement is shown in the figure 52, where the 6 D sensor is attached to an Arduino board that is directly connected to a computer. Ultimately, we will have values in CSV format, which contain temperature, timestamp value, three-dimensional gyroscope, and three-dimensional acceleration. It is crucial to remember that the sensor was directly connected to a computer for value measurement rather than being enclosed within the vehicle. It is significant to remember that this dataset includes outlier values inside this range as well. It is inevitable to apply data analytics to fill in the missing values to prepare the dataset so that machine learning algorithms use it because it contains some missing or unreadable values.
The dataset can be utilized for anomaly detection by machine learning algorithms once it is available after filling in missing values. The machine learning method uses the dataset to identify anomalies or abnormal behavior. The math plot package is useful for visualizing data in two dimensions. The initial thing is to see how the dataset appears in two or three dimensions. To determine which data points might be candidates for outliers by accounting for the dataset maximum and minimum ranges and seeing how each value changes. All of the data points are displayed in two dimensions in the figure 53. As can be seen, the dataset range is rather abrupt; numbers that fall outside of the typical range may be considered outlier values. The figure 54 displays the acceleration value in the x-dimension; further y-z dimensional values are in the appendix F. The gyroscope value in the X-direction is shown in the figure 55, and the appendix G has the remaining values in the Y-Z direction. The neural network model will only identify a small number of anomalous values when trained using the abnormal value dataset. An illustration of identifying anomalies by training on a dataset including abrupt values is

presented in the figure 56. Only a few of them are detected by it; numerous false values are still there but algorithm can not identify them.

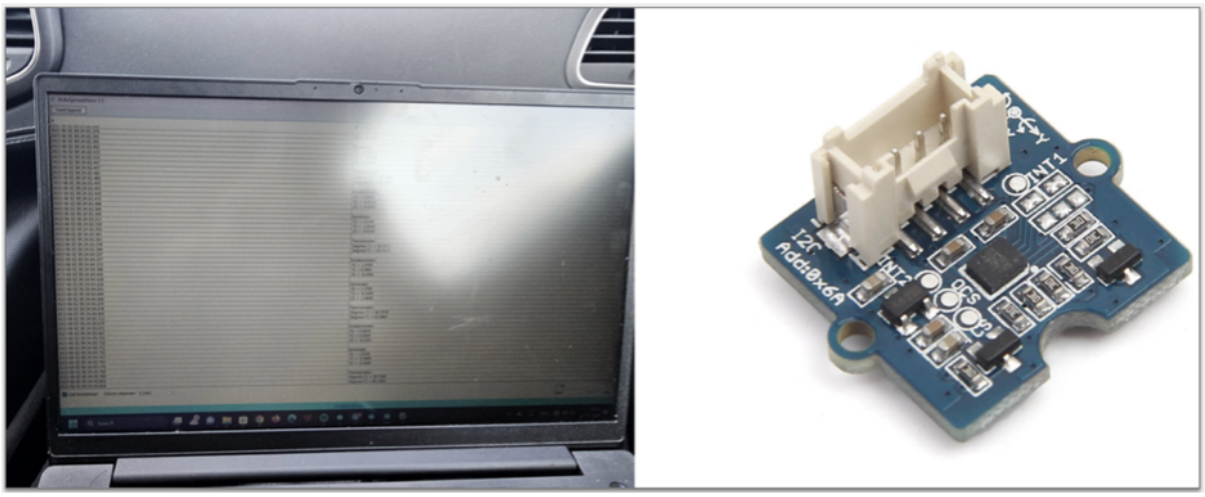


Figure 52: Data acquisition setup

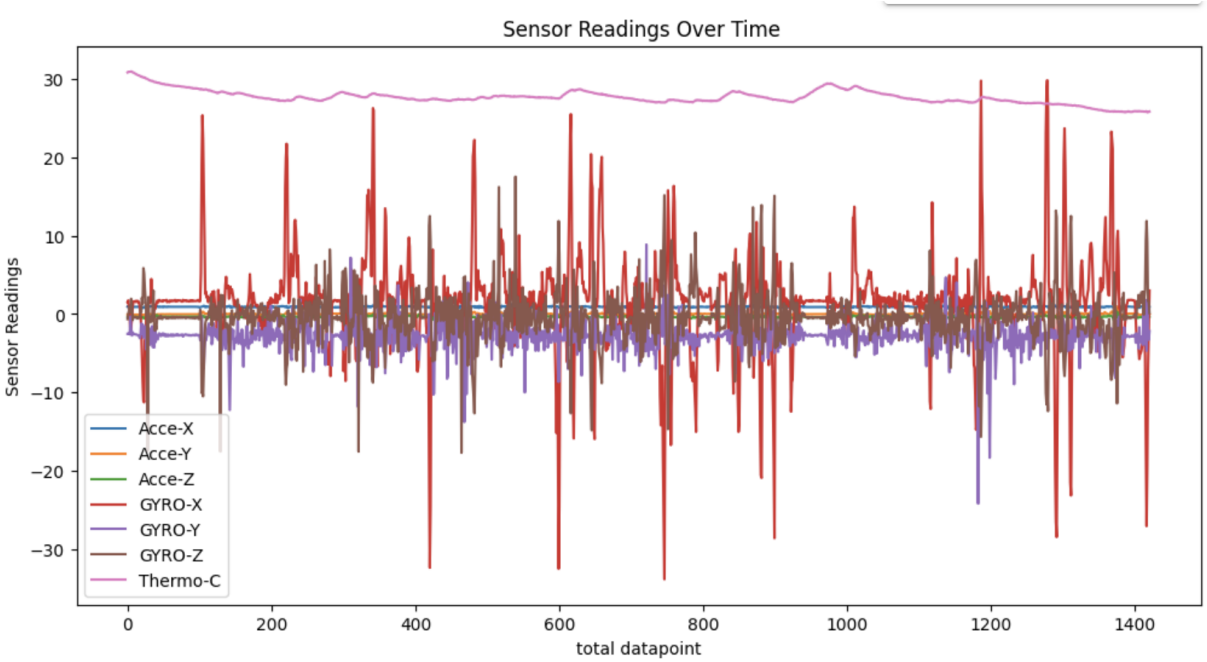


Figure 53: 6-D sensor Dataset for city Siegen

6.3. Analysis of 6-D Sensor Dataset

Numerous statistical techniques are available for examining the distribution of data and determining the most variable areas. The interquartile range (IQR) is one statistical method for determining data that is widely distributed. Four equal portions, known as quartiles, comprise

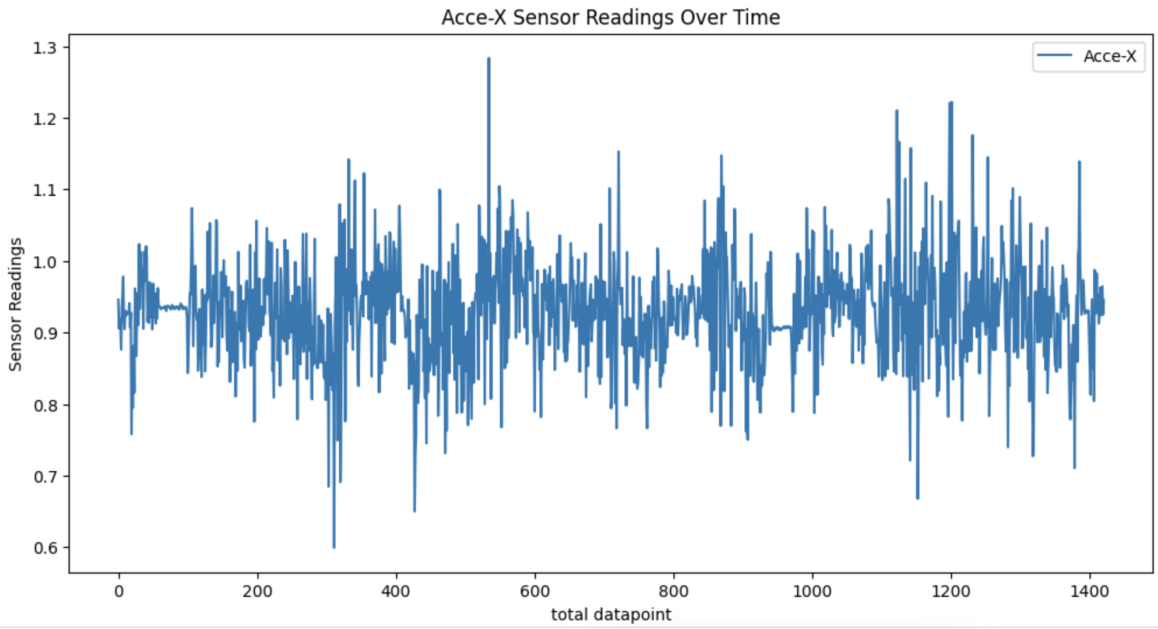


Figure 54: Acceleration in x direction with outlier

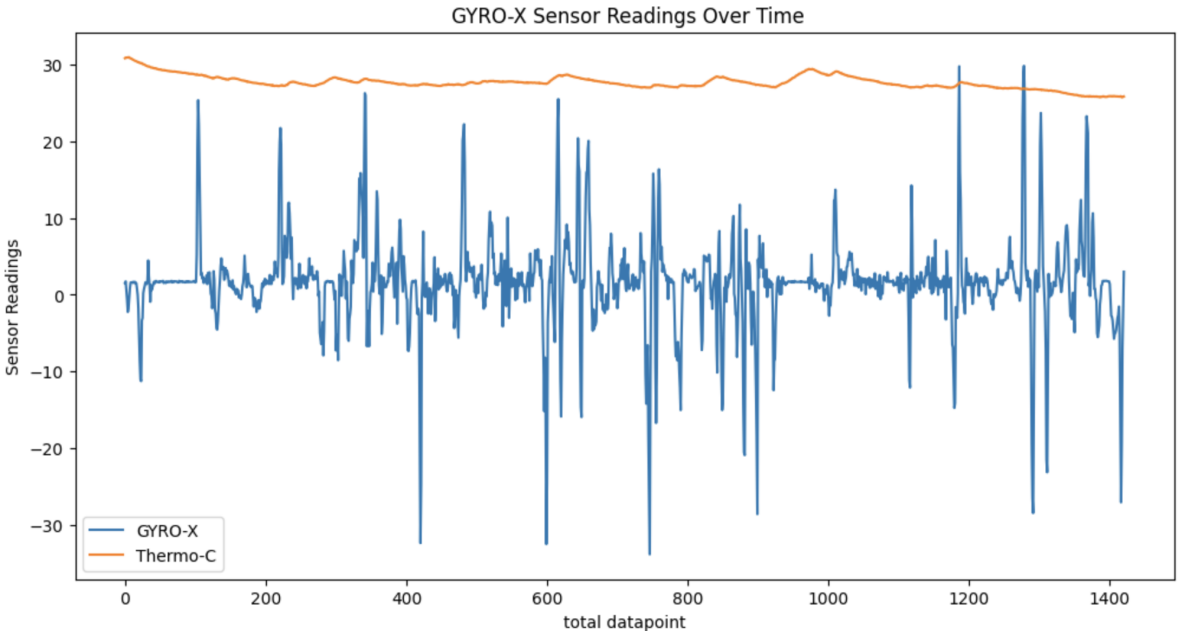


Figure 55: Gyroscope in x direction with outlier

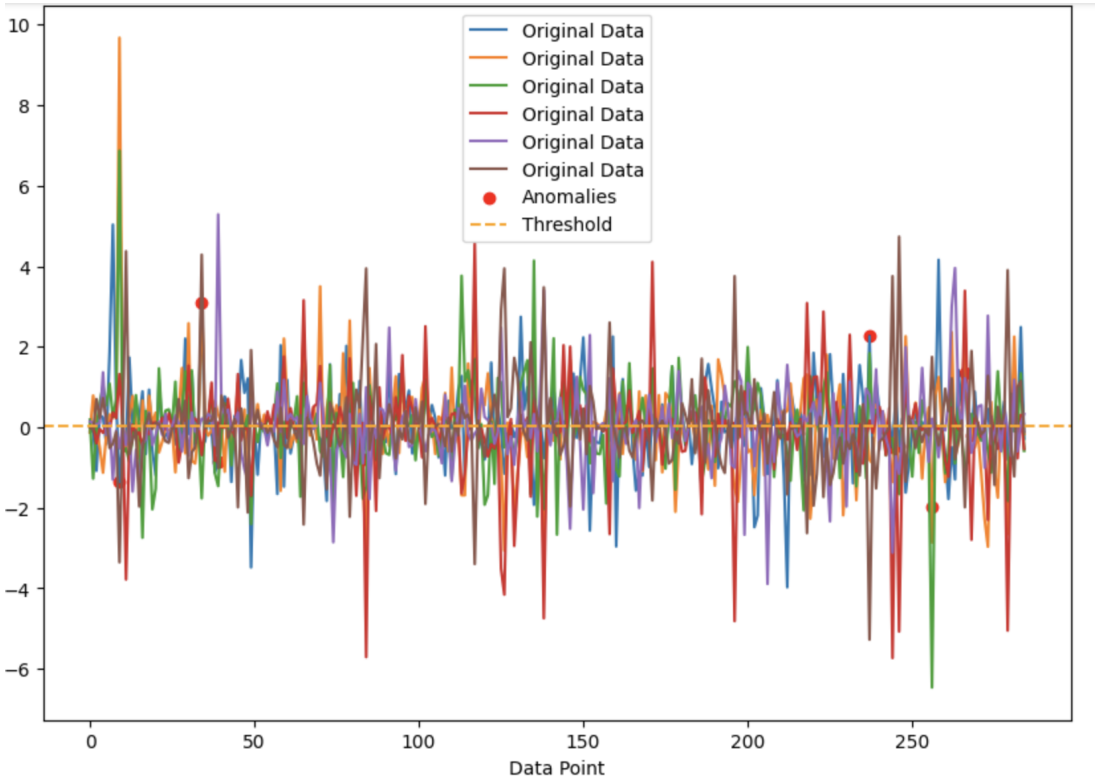


Figure 56: Only few anamoly detected after training neural network

the IQR as the basis for the data. The first quartile, or lower quartile, is designated as Q1. The upper quartile is identified as Q3, and the second quartile called the median, is labeled Q2. The difference between the first quartile (25 percentile) and the third quartile (75 percentile), as indicated by the figure 59, is the fundamental idea of IQR. The formula presented in the figure 59 is used to identify outliers. The Siegen dataset with outlier values figure 53 contains values that are more unlikely than those in the total dataset; we can now identify them using IQR.

6.3.1. Detection of abnormal values by inter quartile range(IQR)

The figure 59 process is iterated for acceleration in 3 dimension and gyroscope in 3 dimension until we obtain values that fall within a specific range and exhibit no sudden fluctuations in values. It displays upper and lower bounds to the full dataset using the IQR formula, allowing us to see data in a specific range without fluctuation or rapid change. The figure 57 shows the values of ACCE-X, which could be considered abnormalities. In the appendix H, the additional acceleration values in both the y and z axes can be viewed. The value range of the gyroscope in three dimensions can be found using the figure 59 same process . The following figure 58 displays the gyroscope outlier detection values in the x-axis direction GYRO-X. The appendix I provides the remaining gyroscope detecting values in the y-z dimension.

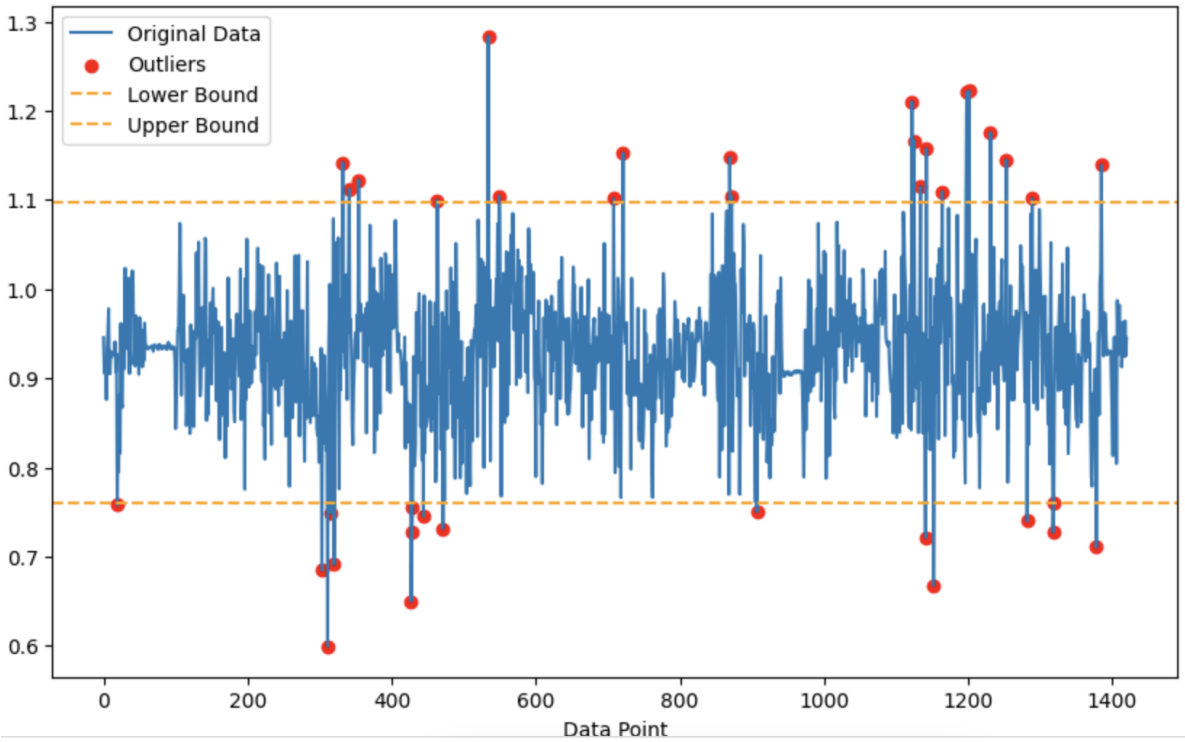


Figure 57: Outlier detection of acceleration in x direction

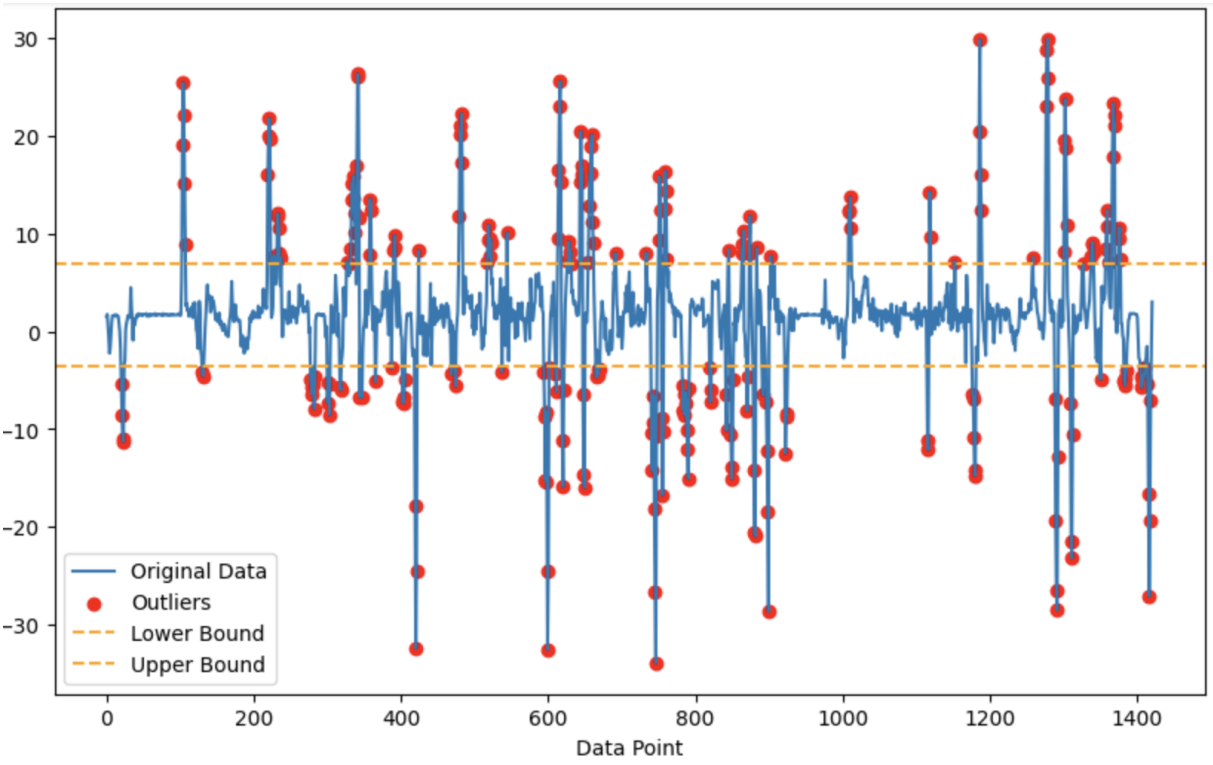


Figure 58: Outlier detection of gyroscope in x direction

6.3.2. Removing Outlier values and machine learning analysis

The following step removes these oscillating values and substitutes the mean value for that specific column. Since it represents the average value for that specific column, it makes sense to utilize it. The remaining fill-in choices include some particular numbers, unique business rules, the maximum, minimum, median, mode, forward or backward fill, etc. The values need to be removed and replaced with the mean value of that specific column, such as acceleration value in three dimensions and a gyroscope value in three dimensions.

After removing abnormal values and combining all data together, figure 60 displays the value range of the new Siegen dataset, and it is clear that the range is bounded and does not experience abrupt changes in value. Thus, we use this dataset to train a machine-learning neural network model. The dataset can be used to train the neural network after all anomalies have been eliminated from each column and merged together. After training with the normal dataset, we need to provide an abnormal dataset to test the trained neural network's performance. The visualisation 61 displays the anomaly detection result after the neural network has been trained. When comparing the outlier detection performance to the previously trained network figure 56, a noticeable improvement can be observed figure 61.

6.3.3. Resource utilization when using 6D sensor dataset

To produce hardware based on weight and bias values, we need to collect them from learned network layers after training the network. The images 62,63 display the hardware of the two networks, while the table displays the hardware resource usage. Similar to the MNIST dataset was designed, the FGPA footprint design employed the same principle for the 6 D sensor dataset.

Table 4: Resources utilization 6-D Sensor dataset

Architecture	Dataset	Number of parameters	Applications	Power	Platform and Surface Area
Neural Network 0	6-D Sensor	581	Sensor data application	35.28 W	Zynq-7000,FPGA LUT 2379,FF 1631,IO 97, BUFG 1

Table 4: Resources utilization 6-D Sensor dataset

Architecture	Dataset	Number of parameters	Applications	Power	Platform and Surface Area
Neural Network 1	6-D Sensor	4461	Sensor data application	106.280 W	Zynq-7000,FPGA LUT 10333,FF 6526, IO 97, BUFG 1

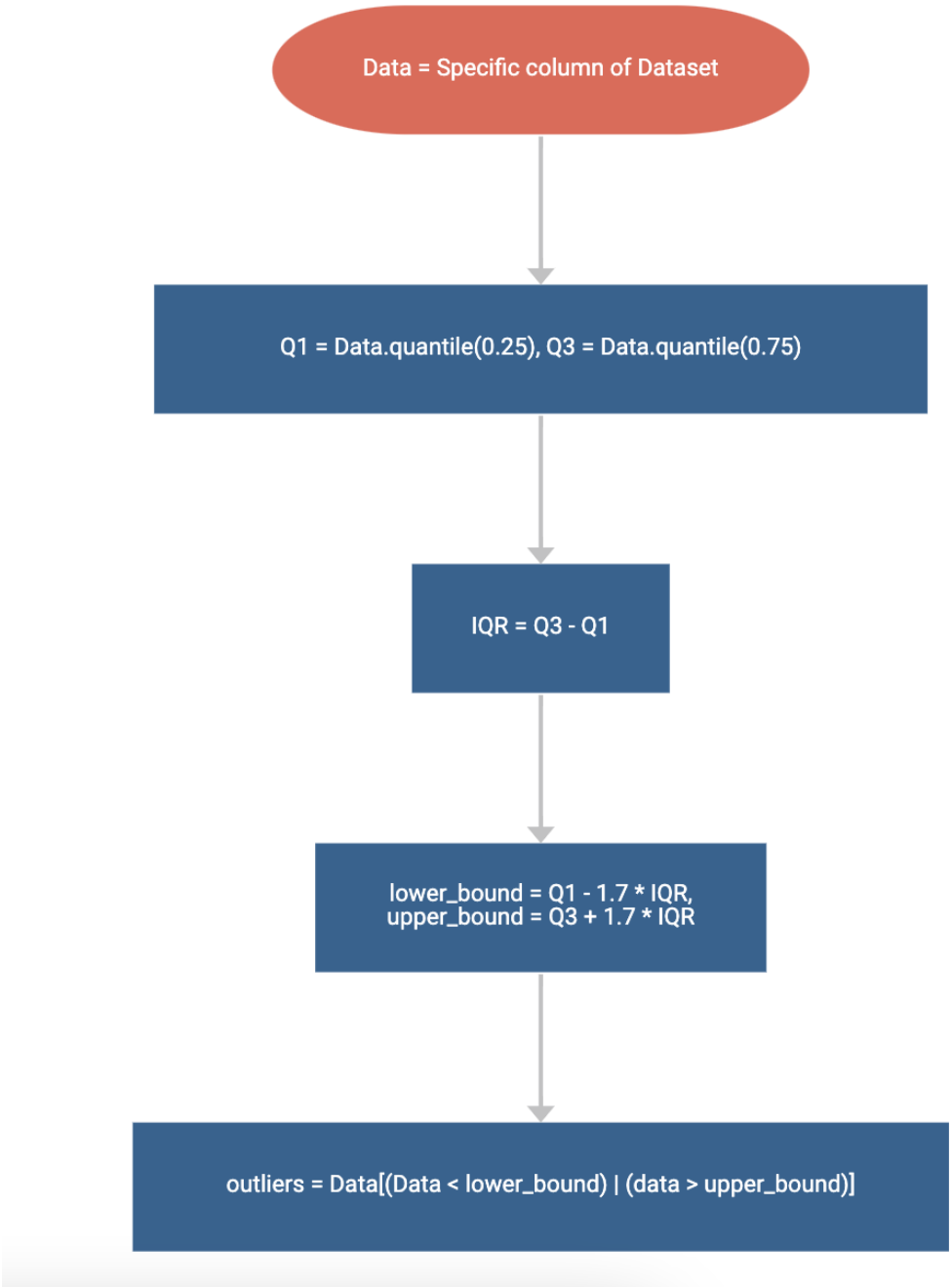


Figure 59: Steps to calculate the interquartile range(IQR) and outlier values

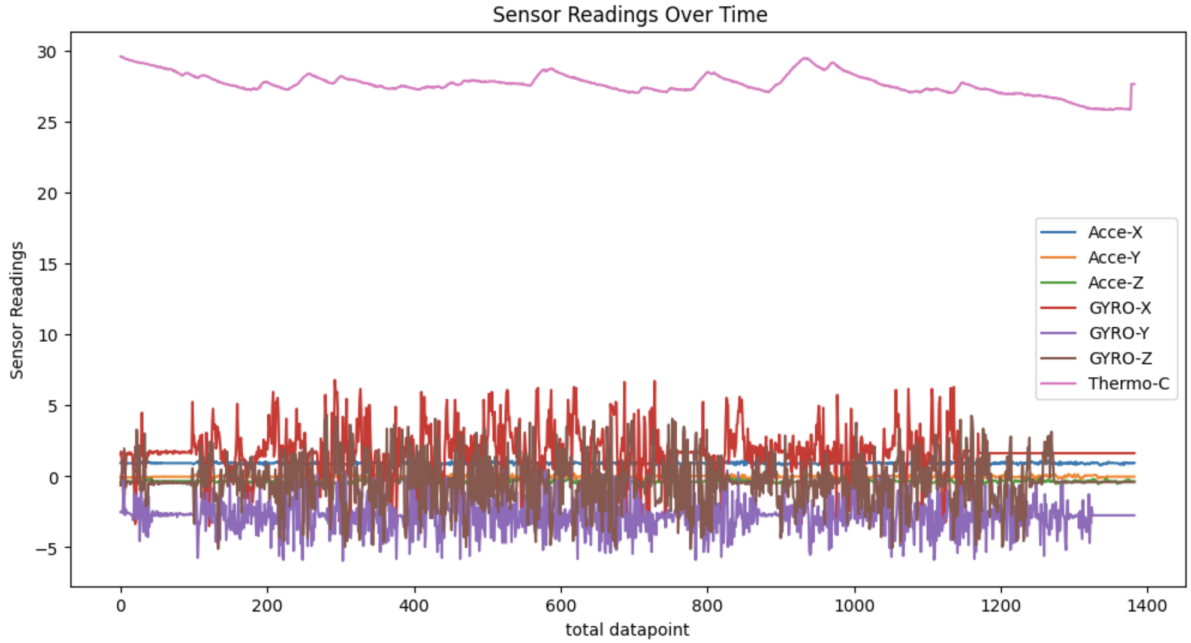


Figure 60: 6-D sensor Dataset for city Siegen with no Outlier

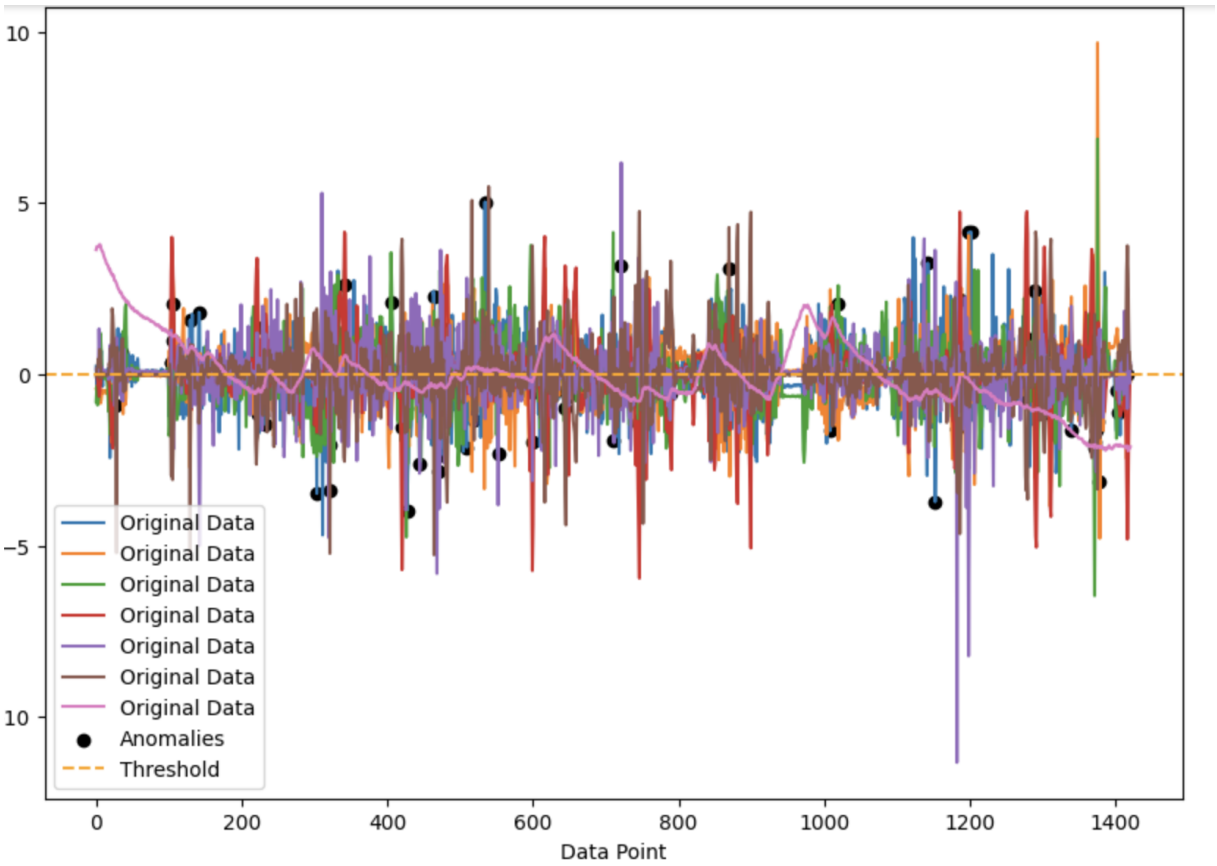


Figure 61: Anamoly detection after training neural network

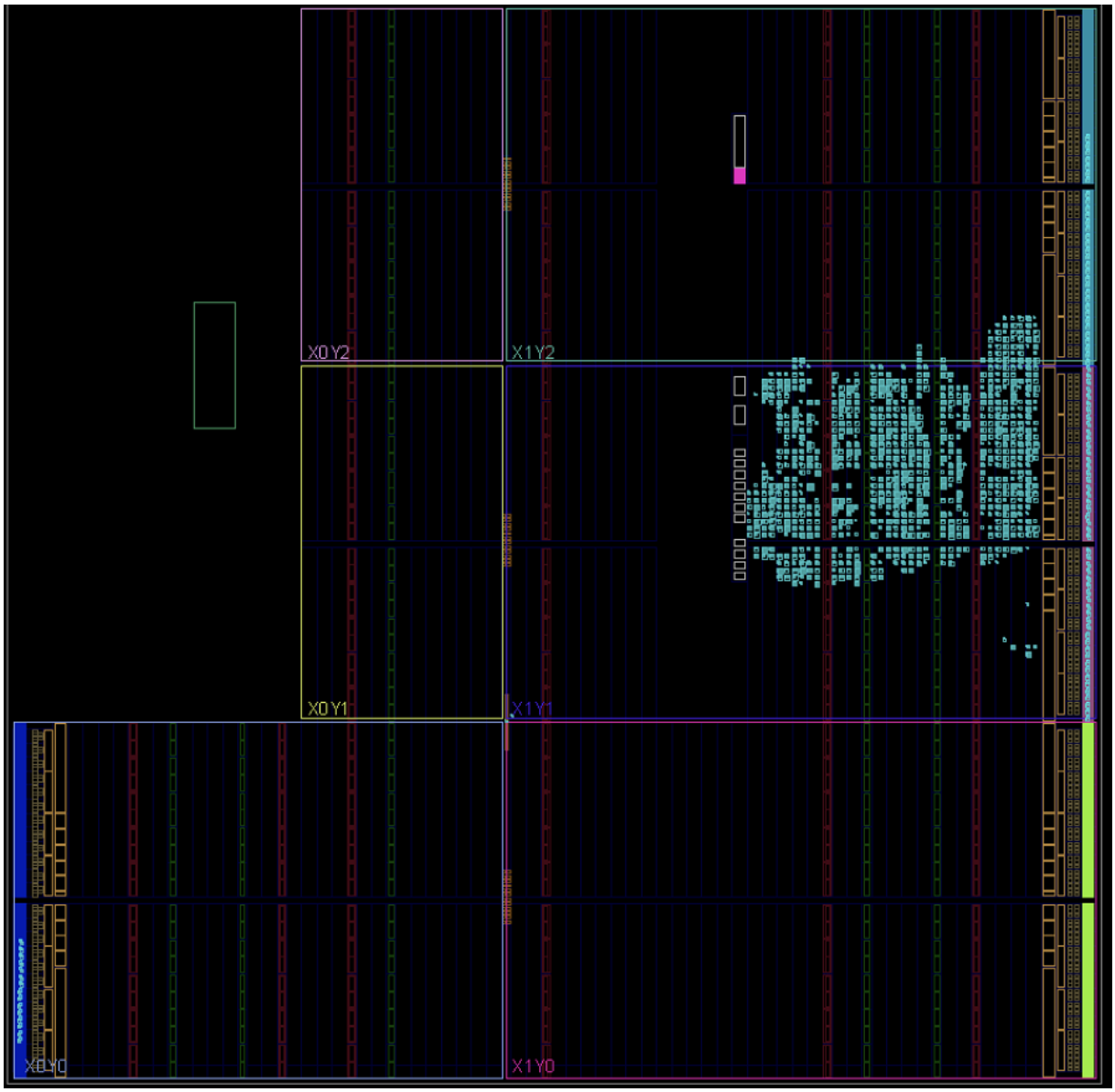


Figure 62: Neural network 0 with Siegen city dataset

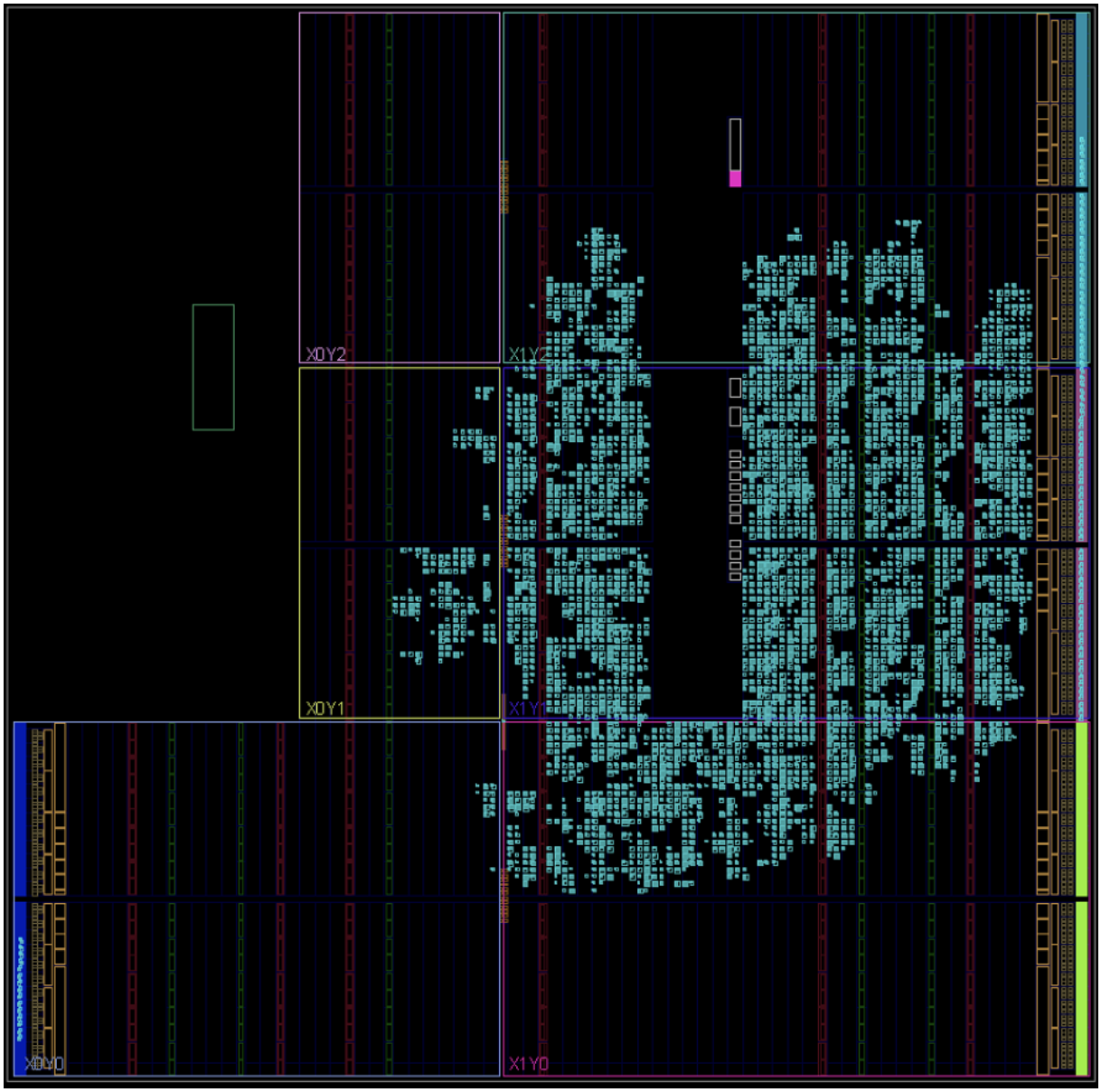


Figure 63: Neural network 1 with Siegen city dataset

7. Summary/Conclusion

Throughout this thesis, the main goal was to develop a neural network on FPGA. The MNIST dataset is ideal because it is easily accessible, comprises 60,000 training examples, and is frequently utilized in machine learning tasks. The other dataset is a 6D sensor dataset that we gathered ourselves for data analysis and footprint fabrication. The neural network was developed and trained using the Python TensorFlow framework, and the hardware was designed using the Zynet framework. After training in the Tensorflow framework, the model weights and bias were extracted and saved as a Txt file, and these data were applied to generate hardware. The mechanism described in the research study allows for enormous versatility in adding or removing layers from network design. After conducting a thorough literature review and discussing the process, I created a software concept for producing hardware based on MNIST data, which is outlined in the chapter 5. Following that, I used a dataset for City Siegen to investigate the network's behavior in the presence of outliers, and hardware fabrication as detailed in the chapter 6. The Siegen data must be cleaned because it contains a large number of missing and illegible values, as is typical with sensor datasets. Following that, the network was created to observe how it behaved in the presence of outlier data, and its performance was below par. After that, I used the Interquartile range (IQR) to remove aberrant values and replace them with the mean value of each column. Additionally, the network performed well after being trained on this new dataset that contained no anomalous values. The suggested technique in this research is trivial to implement neural networks on FPGA; extracting weights and bias from Tensorflow implementation allows great flexibility in experimenting with alternative network architectures.

List of Figures

List of Tables

A. Biblography

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Inamul Hussain, Avtar Singh, and Saurabh Chaudhury. A review on the effects of technology on cmos and cpl logic style on performance, speed and power dissipation. In *2018 IEEE Electron Devices Kolkata Conference (EDKCON)*, pages 332–336. IEEE, 2018.

[3] Alex Pappachen James. A hybrid memristor–cmos chip for ai. *Nature Electronics*, 2(7):268–269, 2019.

[4] Pentti Kanerva. Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive computation*, 1:139–159, 2009.

[5] Denis Kleyko, Mike Davies, E Paxon Frady, Pentti Kanerva, Spencer J Kent, Bruno A Olshausen, Evgeny Osipov, Jan M Rabaey, Dmitri A Rachkovskij, Abbas Rahimi, et al. Vector symbolic architectures as a computing framework for nanoscale hardware. *arXiv preprint arXiv:2106.05268*, 2021.

[6] Yann LeCun, C Cortes, and C Burges. The mnist database of handwritten digits. yann lecun, 2021.

[7] Vasileios Leon, Kiamal Pekmestzi, and Dimitrios Soudris. Exploiting the potential of approximate arithmetic in dsp & ai hardware accelerators. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pages 263–264. IEEE, 2021.

[8] Ahmed Mamdouh, Mbonea Mjema, Gurtac Yemiscioglu, Satoshi Kondo, and Ali Muhtaroglu. Design of efficient ai accelerator building blocks in quantum-dot cellular automata (qca). *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 12(3):703–712, 2022.

[9] Michael Nielsen. Neural networks and deeplearning on mnist dataset, Dec 2019.

[10] Kizheppatt Vipin. Zynet: Automating deep neural network implementation on low-cost reconfigurable edge computing platforms. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 323–326, 2019.

[11] Yunhe Wang, Mingqiang Huang, Kai Han, Hanting Chen, Wei Zhang, Chunjing Xu, and Dacheng Tao. Addernet and its minimalist hardware design for energy-efficient artificial intelligence. *arXiv preprint arXiv:2101.10015*, 2021.

[12] Ying Wei, Jun Zhou, Yin Wang, Yinggang Liu, Qingsong Liu, Jiansheng Luo, Chao Wang, Fengbo Ren, and Li Huang. A review of algorithm & hardware design for ai-based biomedical applications. *IEEE transactions on biomedical circuits and systems*, 14(2):145–163, 2020.

[13] Zakia Yahya, Muhammad Hassan, Shahzad Younis, and Muhammad Shafique. Probabilistic analysis of targeted attacks using transform-domain adversarial examples. *IEEE Access*, 8:33855–33869, 2020.

B. Appendix

C. Additional Details Neural Network 0

The neural network 0 depicted in the diagram 64 comprises three layers, each containing 20, 20, and 10 neurons. Using the Adam optimizer and sparse categorical cross-entropy loss, the network is trained over a 25-epoch period. Finally, we store weights and bias values from the second layer to the last layer. These values will be used for the hardware footprint generation. Hardware generation based on weight and bias from Zynet is depicted in the figure 65. It contains the same number of layers as software implementation plus an extra layer called the hardmax layer, which is used to identify the neuron that is giving the maximum output.

```
import tensorflow as tf
import json
import time
mnist = tf.keras.datasets.mnist
(x_train,y_train),(x_test,y_test) = mnist.load_data()
x_train=tf.keras.utils.normalize(x_train,axis=1)
x_test =tf.keras.utils.normalize(x_test,axis=1)

mdl = tf.keras.models.Sequential()

mdl.add(tf.keras.layers.Flatten())
mdl.add(tf.keras.layers.Dense(20,activation=tf.nn.sigmoid))
mdl.add(tf.keras.layers.Dense(20,activation=tf.nn.sigmoid))
mdl.add(tf.keras.layers.Dense(10,activation=tf.nn.sigmoid))


mdl.compile(optimizer ='adam',
loss='sparse_categorical_crossentropy',
metrics = ['accuracy'])
start_time = time.time()
mdl.fit(x_train,y_train,epochs=25)
(val_loss,val_accuracy) = mdl.evaluate(x_test,y_test)

end_time = time.time()

inference_time = end_time - start_time
print(f"Inference time: {inference_time} seconds")
num_params=mdl.count_params()
print(f"Number of parameters:{num_params}" )
weightLst =[]
biasLst =[]

for i in range (1,len(mdl.layers)):
    weights = mdl.layers[i].get_weights()[0]
    weightLst.append((weights.T).tolist())
    bias =  [[float(b)] for b in mdl.layers[i].get_weights()[1]]
    biasLst.append(bias)
# print("weightLst",weightLst,"biases",biasLst)
data ={"weights":weightLst,"biases":biasLst}
f = open('WeightsandBiases.txt',"w")
# print(data,f)
json.dump(data,f)
f.close()
```

Figure 64: Tensorflow implementation of neural network 0

```python
from zynet import zynet
from zynet import utils
import numpy as np
def MnistZynet(dataWidth,sigmoidSize,weightIntSize,inputIntSize):


    mdl = zynet.model()

    mdl.add(zynet.layer("flatten",784))
    mdl.add(zynet.layer("Dense",20,"sigmoid"))
    mdl.add(zynet.layer("Dense",20,"sigmoid"))
    mdl.add(zynet.layer("Dense",10,"sigmoid"))


    mdl.add(zynet.layer("Dense",10,"hardmax"))



    weightArray =utils.genWeightArray('WeightsandBiases.txt')
    biasArray = utils.genBiasArray('WeightsandBiases.txt')

    mdl.compile(pretrained ='Yes',
    weights =weightArray,
    biases=biasArray,
    dataWidth=dataWidth,
    weightIntSize=weightIntSize,
    inputIntSize=inputIntSize,
    sigmoidSize=sigmoidSize
    )

    zynet.makeXilinxProject('ai','xc7z020clg400-1')
    zynet.makeIP('ai')
    zynet.makeSystem('ai','block_design')
if __name__=="__main__":
    MnistZynet(dataWidth=8,sigmoidSize=10,weightIntSize=3,inputIntSize=1)
```

Figure 65: Zynet implementation of neural network 0

D. Additional Details Neural Network 1

The neural network 1 depicted in the diagram 66 comprises four layers, each containing 30, 20, 10, and 10 neurons. Using the Adam optimizer and sparse categorical cross-entropy loss, the network is trained over a 25-epoch period. Finally, we store weights and bias values from the second layer to the last layer, and these values will be used for hardware fabrication. Hardware generation based on weight and bias from Zynet is depicted in the figure 67. It contains the same number of layers as software implementation plus an extra layer called the hardmax layer, which is used to identify the neuron that is giving the higher output.

```
import tensorflow as tf
import json
import time
mnist = tf.keras.datasets.mnist
(x_train,y_train),(x_test,y_test) = mnist.load_data()
x_train=tf.keras.utils.normalize(x_train,axis=1)
x_test =tf.keras.utils.normalize(x_test,axis=1)



mdl = tf.keras.models.Sequential()

mdl.add(tf.keras.layers.Flatten())
mdl.add(tf.keras.layers.Dense(30,activation=tf.nn.sigmoid))
mdl.add(tf.keras.layers.Dense(20,activation=tf.nn.sigmoid))
mdl.add(tf.keras.layers.Dense(10,activation=tf.nn.sigmoid))
mdl.add(tf.keras.layers.Dense(10,activation=tf.nn.sigmoid))


mdl.compile(optimizer ='adam',
loss='sparse_categorical_crossentropy',
metrics = ['accuracy'])
start_time = time.time()
mdl.fit(x_train,y_train,epochs=25)
(val_loss,val_accuracy) = mdl.evaluate(x_test,y_test)

end_time = time.time()

inference_time = end_time - start_time
print(f"Inference time: {inference_time} seconds")
num_params=mdl.count_params()
print(f"Number of parameters:{num_params}" )
weightLst =[]
biasLst =[]

for i in range (1,len(mdl.layers)):
    weights = mdl.layers[i].get_weights()[0]
    weightLst.append((weights.T).tolist())
    bias =  [[float(b)] for b in mdl.layers[i].get_weights()[1]]
    biasLst.append(bias)
# print("weightLst",weightLst,"biases",biasLst)
data ={"weights":weightLst,"biases":biasLst}
f = open('weightsandbiases.txt',"w")
# print(data,f)
json.dump(data,f)
f.close()
```

Figure 66: Tensorflow implementation of neural network 1

```python
from zynet import zynet
from zynet import utils
import numpy as np
def MnistZynet(dataWidth,sigmoidSize,weightIntSize,inputIntSize):


    mdl = zynet.model()

    mdl.add(zynet.layer("flatten",784))
    mdl.add(zynet.layer("Dense",30,"sigmoid"))
    mdl.add(zynet.layer("Dense",20,"sigmoid"))
    mdl.add(zynet.layer("Dense",10,"sigmoid"))


    mdl.add(zynet.layer("Dense",10,"hardmax"))

    # mdl.summary()

    weightArray =utils.genWeightArray('weightsandbiases.txt')
    biasArray = utils.genBiasArray('weightsandbiases.txt')

    mdl.compile(pretrained ='Yes',
    weights =weightArray,
    biases=biasArray,
    dataWidth=dataWidth,
    weightIntSize=weightIntSize,
    inputIntSize=inputIntSize,
    sigmoidSize=sigmoidSize
    )

    zynet.makeXilinxProject('ai','xc7z020clg400-1')
    zynet.makeIP('ai')
    zynet.makeSystem('ai','block_design')
if __name__=="__main__":
    MnistZynet(dataWidth=8,sigmoidSize=10,weightIntSize=3,inputIntSize=1)
```

Figure 67: Zynet implementation of neural network 1

E. Additional Details Neural Network 2

The neural network 2 depicted in the diagram 68 comprises four layers, each containing 40,30, 20, and 10 neurons. Using the Adam optimizer and sparse categorical cross-entropy loss, the network is trained over a 25-epoch period. Finally, we store weights and bias values from the second layer to the final layer. These values will be used for hardware fabrication with zynet. Hardware generation based on weight and bias from Zynet is depicted in the figure 69. It contains the same number of layers as software implementation plus an extra layer called the hardmax layer, which is used to identify the neuron that is giving the maximum value output.

```python
import tensorflow as tf
import json
import time
mnist = tf.keras.datasets.mnist
(x_train,y_train),(x_test,y_test) = mnist.load_data()
x_train=tf.keras.utils.normalize(x_train,axis=1)
x_test =tf.keras.utils.normalize(x_test,axis=1)



mdl = tf.keras.models.Sequential()

mdl.add(tf.keras.layers.Flatten())
mdl.add(tf.keras.layers.Dense(40,activation=tf.nn.sigmoid))
mdl.add(tf.keras.layers.Dense(30,activation=tf.nn.sigmoid))
mdl.add(tf.keras.layers.Dense(20,activation=tf.nn.sigmoid))
mdl.add(tf.keras.layers.Dense(10,activation=tf.nn.sigmoid))
# 40,30,20, 10

mdl.compile(optimizer ='adam',
loss='sparse_categorical_crossentropy',
metrics = ['accuracy'])
start_time = time.time()
mdl.fit(x_train,y_train,epochs=25)
(val_loss,val_accuracy) = mdl.evaluate(x_test,y_test)
end_time = time.time()

inference_time = end_time - start_time
print(f"Inference time: {inference_time} seconds")
# print(mdl.count.params()) this is error here
num_params=mdl.count_params()
print(f"Number of parameters:{num_params}" )
weightLst =[]
biasLst =[]

for i in range (1,len(mdl.layers)):
    weights = mdl.layers[i].get_weights()[0]
    weightLst.append((weights.T).tolist())
    bias =  [[float(b)] for b in mdl.layers[i].get_weights()[1]]
    biasLst.append(bias)
# print("weightLst",weightLst,"biases",biasLst)
data ={"weights":weightLst,"biases":biasLst}
f = open('weightsandbiases.txt',"w")
# print(data,f)
json.dump(data,f)
f.close()
```

Figure 68: Tensorflow implementation of neural network 2

```
from zynet import zynet
from zynet import utils
import numpy as np
def MnistZynet(dataWidth,sigmoidSize,weightIntSize,inputIntSize):


    mdl = zynet.model()

    mdl.add(zynet.layer("flatten",784))
    mdl.add(zynet.layer("Dense",40,"sigmoid"))
    mdl.add(zynet.layer("Dense",30,"sigmoid"))
    mdl.add(zynet.layer("Dense",20,"sigmoid"))
    mdl.add(zynet.layer("Dense",10,"sigmoid"))
    mdl.add(zynet.layer("Dense",10,"hardmax"))

    weightArray =utils.genWeightArray('WeigntsAndBiases.txt')
    biasArray = utils.genBiasArray('WeigntsAndBiases.txt')

    mdl.compile(pretrained ='Yes',
    weights =weightArray,
    biases=biasArray,
    dataWidth=dataWidth,
    weightIntSize=weightIntSize,
    inputIntSize=inputIntSize,
    sigmoidSize=sigmoidSize
    )

    zynet.makeXilinxProject('ai','xc7z020clg400-1')
    zynet.makeIP('ai')
    zynet.makeSystem('ai','block1')
if __name__=="__main__":
    MnistZynet(dataWidth=8,sigmoidSize=10,weightIntSize=3,inputIntSize=1)
```

Figure 69: Zynet implementation of neural network 2

F. Acceleration reading in Y-Z direction

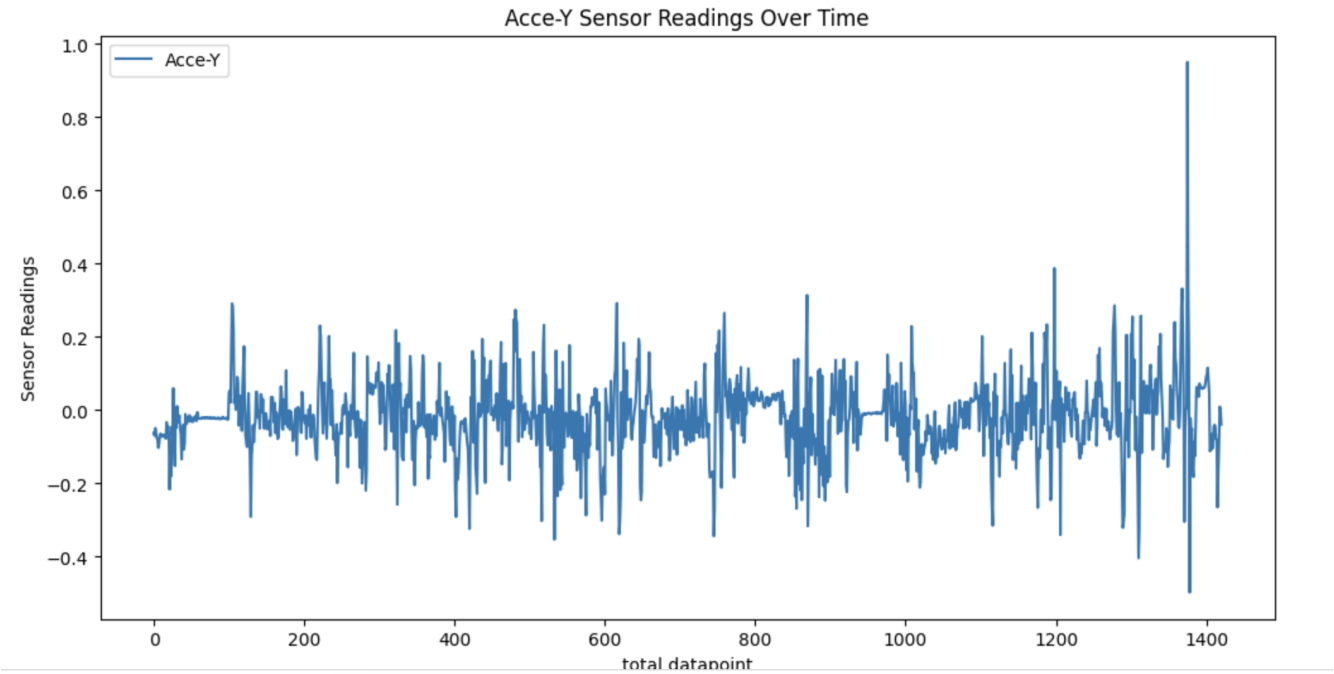


Figure 70: Acceleration reading in Y direction

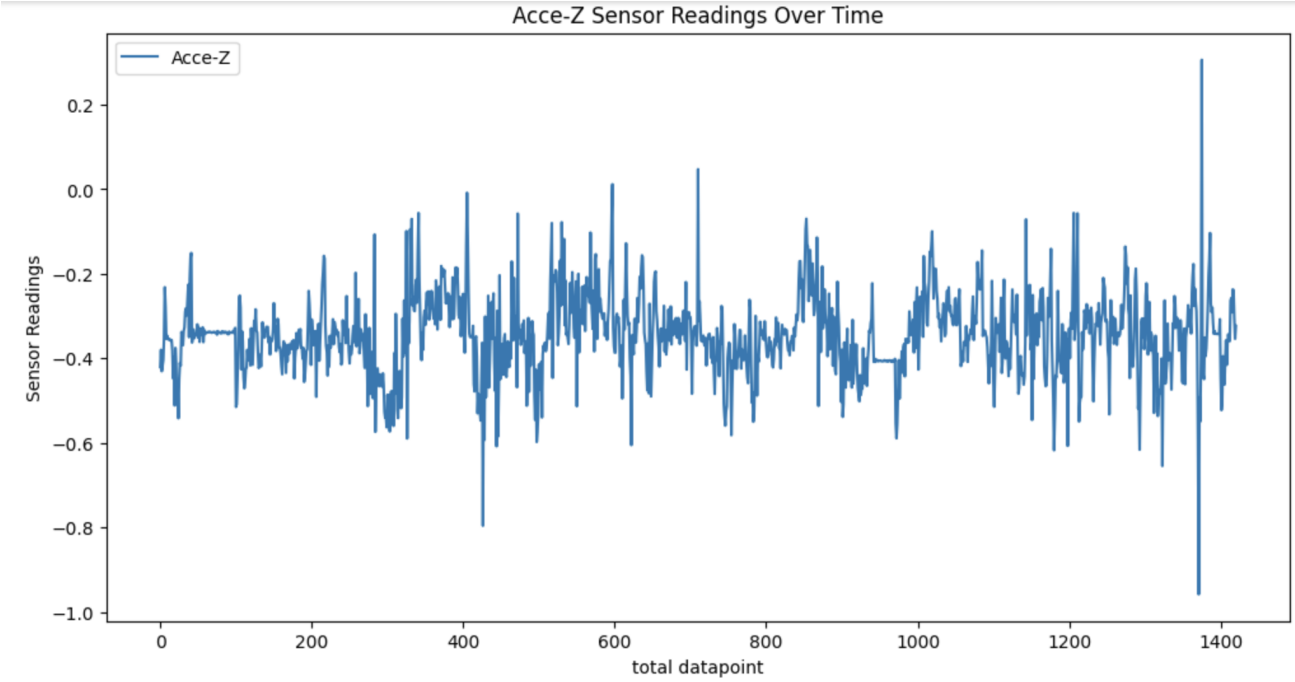


Figure 71: Acceleration reading in Z direction

G. Gyroscope reading in Y-Z direction

Figure 72: Gyroscope reading in Y direction

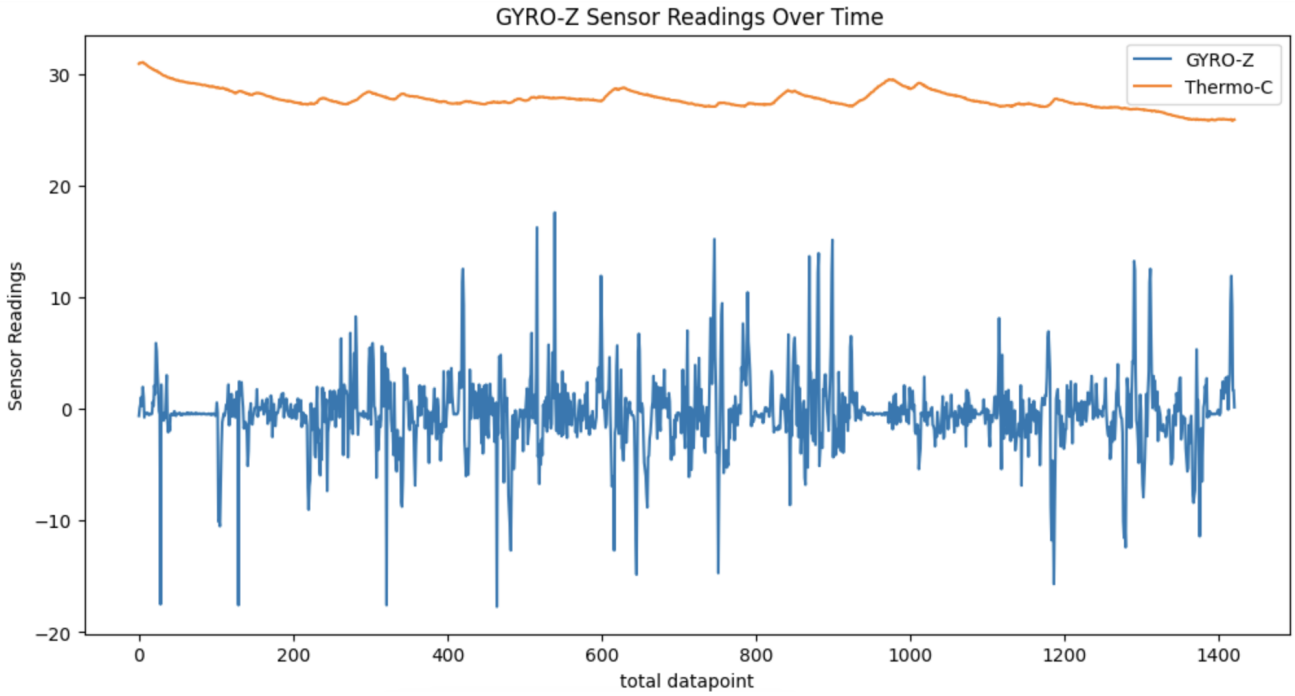


Figure 73: Gyroscope reading in Z direction

H. Outlier detection of Acceleration reading in Y-Z direction

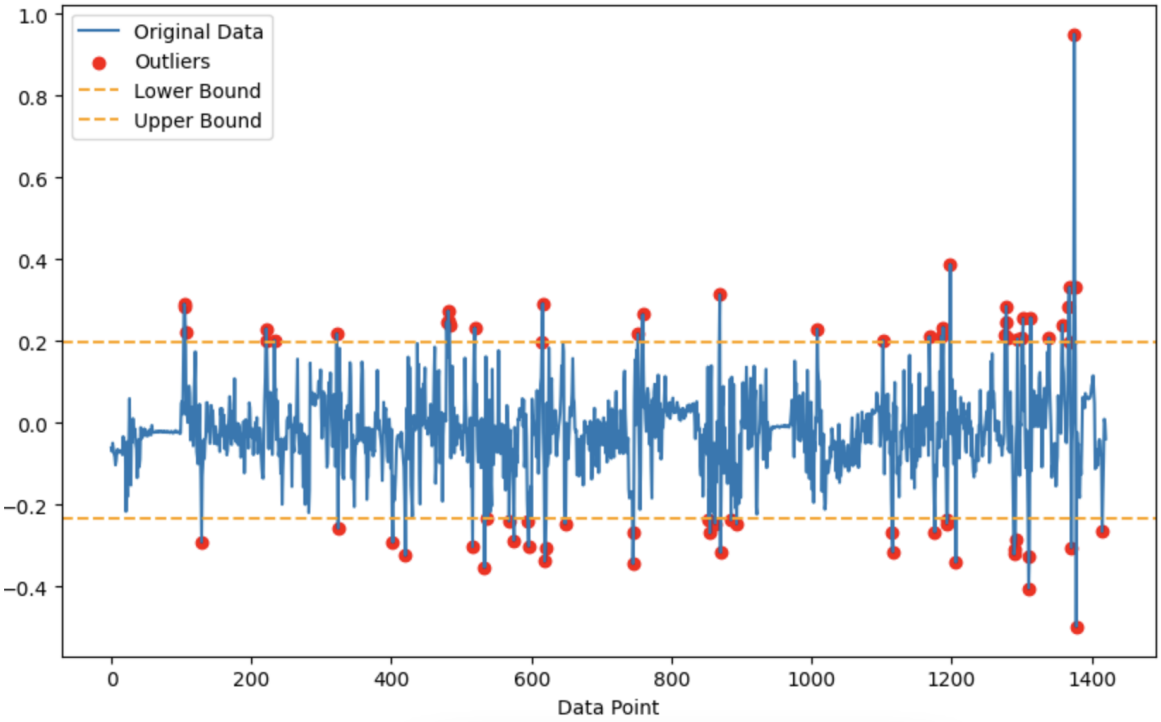


Figure 74: Detection of outlier acceleration in y-direction

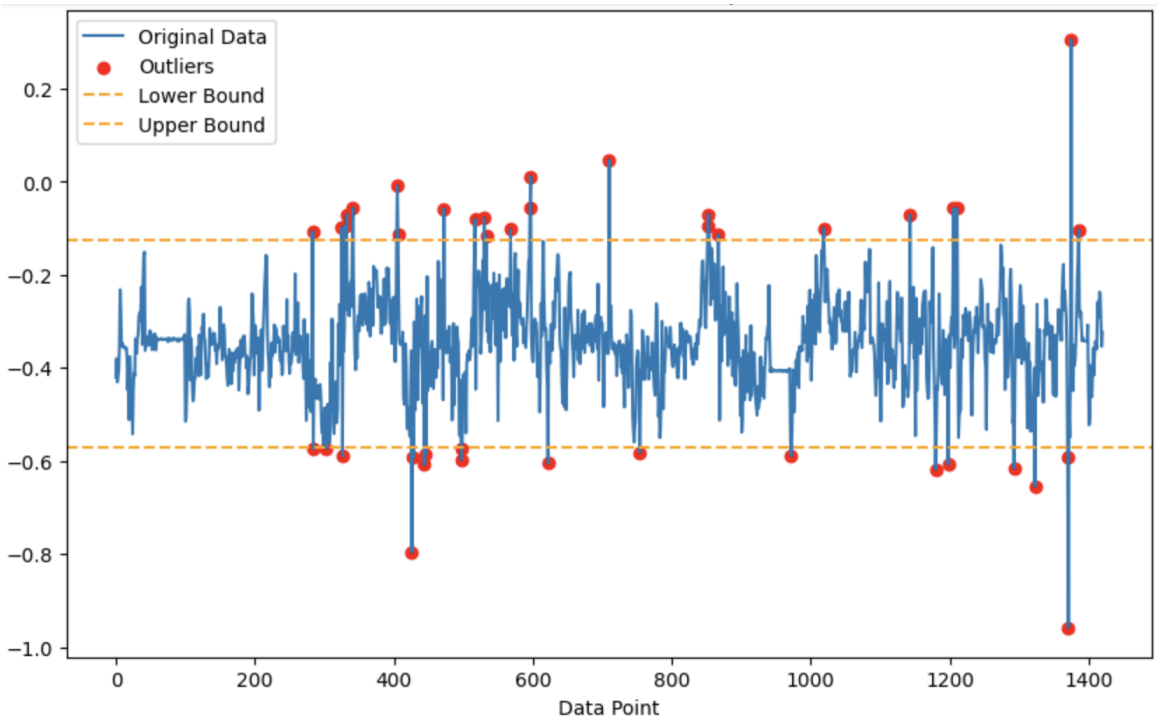


Figure 75: Detection of outlier acceleration in z-direction

I. Outlier detection of gyroscope reading in Y-Z direction

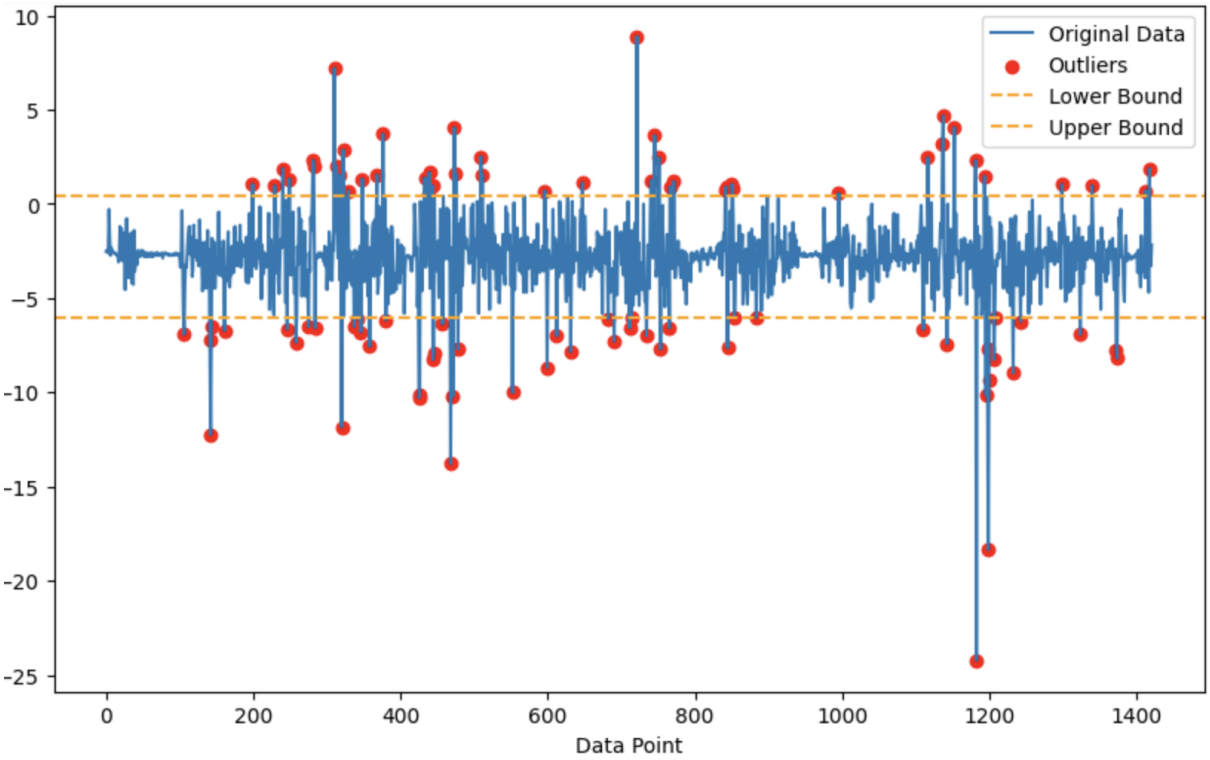


Figure 76: Outlier detection of gyroscope in y-direction

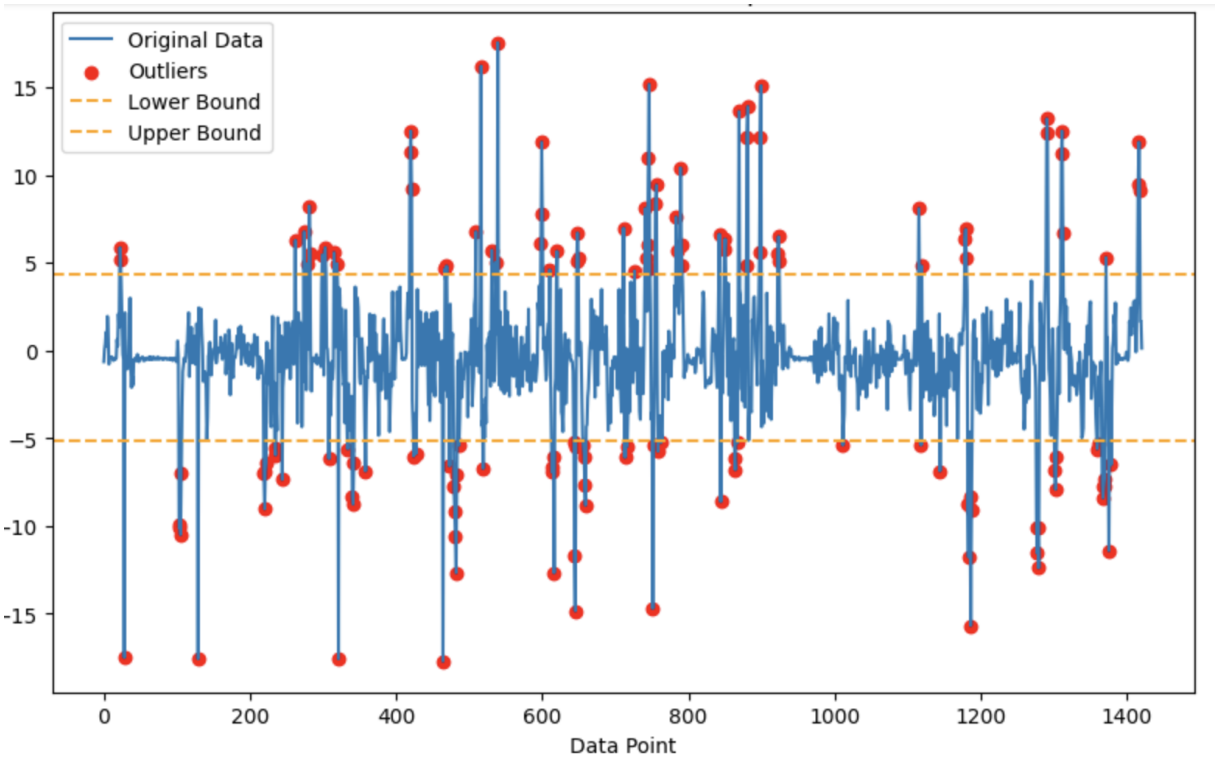


Figure 77: Outlier detection of gyroscope in z-direction