BINARY NEURAL NETWORK IMPLEMENTATION FOR HANDWRITTEN DIGIT RECOGNITION ON FPGA

by
Emir Devlet Ertörer

Engineering Project Report

Yeditepe University
Faculty of Engineering
Department of Computer Engineering
2025

BINARY NEURAL NETWORK IMPLEMENTATION FOR HANDWRITTEN DIGIT RECOGNITION ON FPGA

APPROVED BY:

Prof. Cem Ünsalan . . . . . .
(Supervisor)

(Examiner 1 Name) . . . . . .

(Examiner 2 Name) . . . . . .

DATE OF APPROVAL: .../.../2025

ACKNOWLEDGEMENTS

ABSTRACT

BINARY NEURAL NETWORK IMPLEMENTATION FOR HANDWRITTEN DIGIT RECOGNITION ON FPGA

Image recognition has gained significant importance over the past decades, with neural network models widely deployed across a range of applications. While traditional deep learning models achieve high accuracy, they often require substantial computational resources and memory. Binary Neural Networks (BNNs) present a highly efficient alternative by reducing both the size and complexity of the model, making them especially suitable for deployment on resource constrained platforms. Due to their binarized weights and activations, BNNs can achieve faster inference and lower power consumption compared to full precision models. This project focuses on the implementation of a trained BNN for handwritten digit recognition and its deployment on a Field Programmable Gate Array (FPGA). The design leverages the parallel processing capabilities of FPGAs to achieve real-time inference while maintaining a compact hardware.

ÖZET

FPGA ÜZERİNDE EL YAZISI RAKAM TANIMA İÇİN BİNARY NÖRAL AĞ UYGULAMASI

Görüntü tanıma, son yıllarda önemli bir gelişme göstermiş ve sinir ağı modelleri birçok alanda yaygın şekilde kullanılmaya başlanmıştır. Geleneksel derin öğrenme modelleri yüksek doğruluk oranlarına ulaşsa da, genellikle yüksek hesaplama gücü ve bellek gereksinimi duyarlar. Binary Neural Network (BNN) modelleri ise, modelin boyutunu ve karmaşıklığını azaltarak özellikle donanım kaynakları sınırlı olan sistemler için oldukça verimli bir alternatif sunar. Ağırlıkların ve aktivasyonların binarize edilmesi sayesinde BNN'ler, tam hassasiyetli modellere kıyasla daha hızlı çıkarım (inference) yapabilirler ve daha az güç tüketirler. Bu proje, el yazısı rakamların tanınması için eğitilmiş bir BNN modelinin Field Programmable Gate Array (FPGA) üzerinde uygulanmasına odaklanmaktadır. Tasarım, FPGA'nin paralel işlem yeteneklerinden faydalanarak gerçek zamanlı çıkarım sağlamayı hedeflemekte ve donanımın kompakt yapısını korumaktadır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS/ABBREVIATIONS

FPGA	Field Programmable Gate Array
BNN	Binary Neural Network
CLB	Configurable Logic Block
LUT	Look-Up Table
CNN	Convolutional Neural Network
CPU	Central Processing Unit
GPU	Graphics Processing Unit
ReLU	Rectified Linear Unit
STE	Straight-Through Estimator
XNOR	Exclusive NOR (bitwise operation)
SBN	Shift-based Batch Normalization
AdaMax	Adaptive Moment Estimation Max Optimizer
QAT	Quantization Aware Training
HLS	High Level Synthesis

1. INTRODUCTION

1.1. Binarized Neural Networks (BNNs)

The continuous progress in the field of deep learning [1] has led to major advancements in areas such as speech recognition, generative AI, large language models and computer vision. The most widely used architectures is Convolutional Neural Networks (CNNs) which owe their popularity to their deep structure and large number of parameters which ranges from millions to billions. These models have achieved state of the art performance across a wide range of tasks.

Despite their high accuracy, CNNs rely heavily on precise floating-point computations that typically use 32-bit representations. This leads to significant computational and memory usage. As a result, deploying CNN models typically requires high performance hardware such as modern Graphics Processing Units (GPUs) or specialized Central Processing Units (CPUs).

Research has shown that CNNs include large redundancies in their structure [2]. This has led to the development of various model compression techniques that try to reduce the memory and computational resource usage without severely impacting performance and accuracy. One of the most effective compression techniques is quantization, where weights and activations are represented using low-precision numbers. Within this category, binary quantization is the most aggressive form of quantization. It restricts weights and activations to just two possible values: -1 and +1 (or 0 and 1).

This form of quantization forms a special class of models known as Binary Neural Networks (BNNs). By representing both weights and activations using only 1 bit instead of 32 bits, BNNs drastically reduce memory usage. Additionally, expensive floating-point matrix multiplications can be replaced with lightweight bitwise operations such as the XNOR and bitcount, which offer significant speedup. For instance, BNNs can achieve up to 32× memory savings and up to 58× computational speedup on CPUs [3, p. 4].

Due to their lightweight and efficient structure, BNNs are particularly well suited for deployment on embedded and resource constrained platforms such as FPGAs and mobile devices [3].

1.1.1. Forward Propagation in BNNs

To understand how forward propagation works in BNNs, it is helpful to first understand how standard neural networks work. In any neural network, each layer contains a set of *weights* (learnable parameters) that determine how much influence each input has on the output of a neuron. These weights are initially assigned random values and are updated through training so that the network can learn patterns from the provided data.

During forward propagation, each neuron receives a set of input values (called *activations*) and computes a weighted sum of these inputs:

$$z = \sum_{i=1}^{n} w_i \cdot x_i$$

This value z is then passed through an *activation function* (such as the ReLU or sigmoid function) to produce the neuron's output. This step helps the network understand and respond to the input in a meaningful way. The weighted sum shows how much the input matches what the neuron has learned to look for. The activation function then decides whether the neuron should react to this input. By doing this selectively, the network can pick up on useful patterns and better tell different outputs or classes apart.

BNNs simplify this process by constraining both the weights w_i and activations x_i to binary values, typically $\{-1, +1\}$. This allows computationally expensive floating-point multiplications to be replaced with bitwise *XNOR* operation. In binary logic, the XNOR gate outputs 1 when both inputs are the same and 0 otherwise. Therefore if x_i and w_i are encoded as 1 for +1 and 0 for –1, then:

$$\text{XNOR}(x_i, w_i) = \begin{cases} 1, & x_i = w_i \\ 0, & x_i \neq w_i \end{cases}$$

Applying the XNOR operation to each element in two binary vectors x and w, we obtain a new binary vector indicating which elements match. The number of 1s in the result (called the *popcount*) tells us how many inputs matched their corresponding weights:

$$\text{popcount}(\text{XNOR}(x, w)) = \sum_{i=1}^{n} [x_i = w_i]$$

This count by itself doesnt represent the full dot product. In BNNs, the idea is to simulate the dot product using only bitwise logic. When x_i and w_i are the same (both $+1$ or both -1), their product is $+1$. If they differ, the product is -1. So, each match adds $+1$ and each mismatch subtracts 1 from the total.

Let m be the number of matching elements and n the total number of elements in the vectors. The number of mismatches is then $n - m$, and the signed dot product becomes:

$$z = (m \cdot +1) + ((n - m) \cdot -1)$$

Which can be simplified as:

$$z = m - (n - m) = 2m - n$$

Since $m = \text{popcount}(\text{XNOR}(x, w))$, we can express the full dot product approximation as:

$$z \approx 2 \cdot \text{popcount}(\text{XNOR}(x, w)) - n$$

This transformation allows us to simulate the original full dot product between binary vectors using only XNOR and popcount operations, which are much more efficient in digital hardware than multiplications and additions.

To make this XNOR based computation possible, both the input activations and the weights must first be binarized. This is done using the **sign function** which converts real valued inputs into binary form:

$$\text{sign}(x) = \begin{cases} +1, & x \geq 0 \\ -1, & x < 0 \end{cases}$$

By applying the sign function in the forward pass, the network converts values into binary form. This binarization allows it to replace standard multiplications with faster XNOR and popcount operations.

Once the binary activations are computed, they are compared to the true labels to calculate the loss. This loss is then used to adjust the weights during backpropagation.

1.1.2. Backward Propagation and the Straight-Through Estimator (STE)

To understand how backward propagation works in BNNs, it is important to first understand the standard backpropagation algorithm used in traditional neural networks. In a typical neural network, once the forward pass produces predictions, the network calculates a loss to measure how different the predictions are from the actual labels. This loss is then sent backward through the network using the chain rule so that the weights can be updated to reduce future errors.

For a weight w, the update is computed as:

$$w \leftarrow w - \eta \cdot \frac{\partial L}{\partial w}$$

where η is the learning rate and $\frac{\partial L}{\partial w}$ is the gradient of the loss with respect to that weight. These gradients are computed by applying the chain rule through every layer of the network:

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial w_i}$$

Here, z is the weighted sum of inputs and $\frac{\partial z}{\partial w_i} = x_i$ is the input to the neuron. This process works smoothly for differentiable functions like ReLU or sigmoid.

However, in Binary Neural Networks, backpropagation faces a major problem due to the use of the *sign function* during binarization:

$$\text{sign}(x) = \begin{cases} +1, & x \geq 0 \\ -1, & x < 0 \end{cases}$$

This function is non differentiable at $x = 0$ and its derivative is zero almost everywhere else. As a result, the gradient $\frac{d}{dx}\text{sign}(x)$ is either undefined or vanishes, making it impossible to propagate error signals backward. This causes what's known as the *vanishing gradient problem* where gradients become too small for the network to keep learning. In BNNs this issue is even more serious because the sign function doesn't give any meaningful gradient during backpropagation. Without a workaround, the model can't update its weights properly, which effectively stops learning altogether [4].

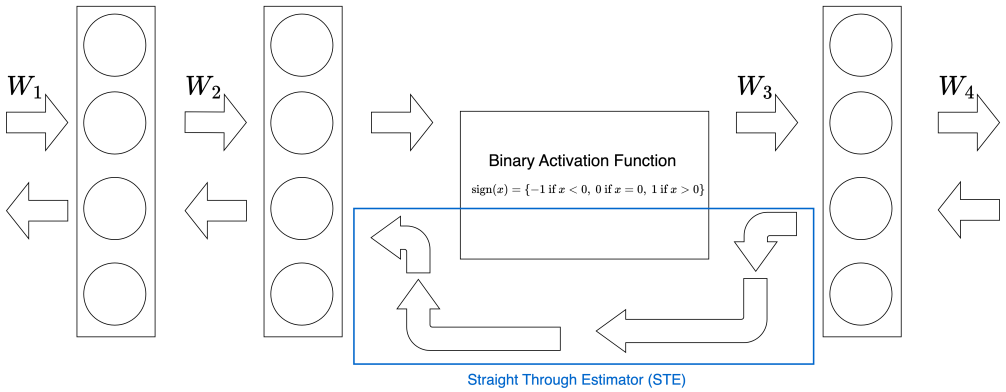

Figure 1.1. Illustration of the Binary Activation Function and use of the Straight-Through Estimator (STE) in backpropagation.

To solve this issue, BNNs use a method called the **Straight-Through Estimator (STE)**. Instead of relying on the actual derivative of the sign function, which doesn't exist, STE treats the sign function as if it were the identity function during backpropagation. This way, it skips over the non-differentiable binarization step and assigns an approximate gradient to keep the learning going.

The STE approximation is typically written as:

$$\frac{d}{dx}\text{sign}(x) \approx \begin{cases} 1, & |x| \leq 1 \\ 0, & \text{otherwise} \end{cases}$$

This means that if the input is within a certain range, the gradient is approximated as 1 and outside that range it is zero. This keeps gradients flowing during training and allows standard optimizers like Adam or AdaMax to work effectively.

This approach lets the network keep updating its weights even though it doesn't use the real gradient from the binarization step. While the exact size of the gradient might not be accurate, STE keeps the direction of the update, which is often good enough for the model to learn. Thanks to this, BNNs can still be trained using standard optimization methods, even though they use only binary weights and activations during the forward pass.

1.1.3. Batch Normalization and Activation Functions

In Binary Neural Networks, the *sign function* is used to binarize both weights and activations. But because the function is very sensitive near zero, even small changes in input can flip the sign and change the binary result. This makes it very important to carefully control the distribution of inputs before binarization to ensure stable and meaningful outputs across layers.

To address this, *Batch Normalization (BatchNorm)* is used right before each binarization step. BatchNorm adjusts the inputs so they have a mean of zero and a standard deviation of one across the mini-batch. This keeps the values in a predictable range, making the binarization step more stable and effective during training.

The batch normalization operation is defined as:

$$\text{BN}(x) = \gamma \cdot \frac{x - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta$$

Here, μ_B and σ_B^2 represent the mean and variance of the input x over the batch. The terms γ and β are learnable parameters that allow the model to rescale and shift the normalized output. The small constant ϵ is added to prevent division by zero. In BNN implementations that target hardware efficiency, the normalization is typically simplified by fixing $\gamma = 1$, $\beta = 0$ and disabling scaling during inference.

Once the input is normalized, it is binarized using the same sign function as before:

$$\text{sign}(\text{BN}(x)) = \begin{cases} +1, & \text{BN}(x) \geq 0 \\ -1, & \text{BN}(x) < 0 \end{cases}$$

In BNNs, batch normalization and binarization work together to replace traditional activation functions like ReLU. Instead of applying a separate non-linear function, the network simply takes the sign of the normalized activation to produce a binary output for the next layer.

BatchNorm helps keep activations centered so that the neurons don't always output just $+1$ or -1. This keeps the binary outputs more balanced, which improves gradient flow during training. As a result, the model trains more smoothly and can reach higher performances faster.

1.1.4. Advantages and Trade-Offs

Binary Neural Networks are designed to reduce computational cost, memory usage and power consumption by replacing floating-point operations with binary logic. These properties make BNNs well suited for deployment on resource-constrained hardwares like FPGAs.

Memory Efficiency: Since all weights and activations are stored using a single bit (-1 or $+1$), the overall model size is reduced by a factor of 32 compared to networks that use 32-bit parameters. This allows for much smaller memory usage, which is particularly useful in embedded systems.

Computation Speed: BNNs replace multiplication heavy operations with bitwise XNOR and popcount. These operations are faster and simpler to implement in hardware. As a result, inference latency is significantly reduced, especially when compared to traditional floating-point neural networks.

Power Efficiency: Binary operations use less switching and need fewer memory accesses, which helps keep power consumption low. This makes BNNs a good fit for situations where saving energy matters like in battery powered devices or low-power systems that need to run in real time.

However, these advantages come with certain trade-offs:

- **Accuracy Loss:** Binarization introduces approximation error by replacing real valued operations with binary ones. This lowers accuracy, particularly on complex or high resolution datasets.
- **Limited Expressiveness:** Since weights and activations are restricted to just two values, each neuron can represent less information. To make up for this, BNNs often need to use more neurons or additional layers to achieve similar performance.
- **Difficult Hyperparameter Tuning:** Due to the aggressive quantization, BNNs tend to be more sensitive to settings like learning rate, batch size and weight initialization. This means training often requires more careful and deliberate tuning to work well.
- **Limited Compatibility:** Many standard layers and functions used in deep learning don't work well with binary values, which can reduce design flexibility or require custom solutions.

1.2. Field Programmable Gate Arrays (FPGAs)

Field Programmable Gate Arrays (FPGAs) are reconfigurable integrated circuits designed to implement custom digital logic. Unlike fixed function hardware like Application Specific Integrated Circuits (ASICs) or general purpose processors like CPUs and GPUs, FPGAs allow developers to define how the underlying hardware behaves even after the device has been manufactured. This flexibility enables precise control over timing, parallelism and data flow, which makes FPGAs suitable for applications requiring low latency, high throughput or hardware-level customization.

1.2.1. Internal Structure of an FPGA

An FPGA is composed of a regular array of logic and routing elements that can be programmed to implement a wide range of digital circuits. Its architecture is modular and consists of several key building blocks, as shown in Figure 1.2.

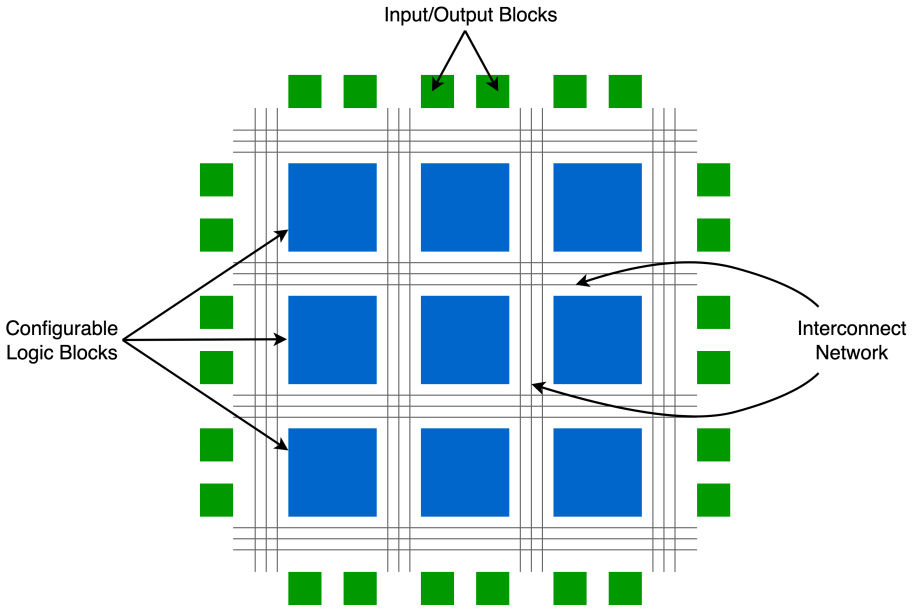

Figure 1.2. High-level layout of an FPGA showing Configurable Logic Blocks (CLBs), Input/Output Blocks (IOBs) and the interconnect network.

The main structural components of an FPGA include:

1. Configurable Logic Blocks (CLBs):
These are the main building blocks responsible for implementing logic functions. Each CLB contains multiple Look-Up Tables (LUTs) and flip-flops (FFs), which together enable both combinational and sequential logic.

- *Look-Up Tables (LUTs):* Small memory arrays that store precomputed outputs for logic functions. A k-input LUT can implement any Boolean function of k variables. LUTs are what lead to the reconfigurability of FPGAs [5, see p. 33].
- *Flip-Flops (FFs):* 1-bit registers used to implement state-holding elements for sequential logic.

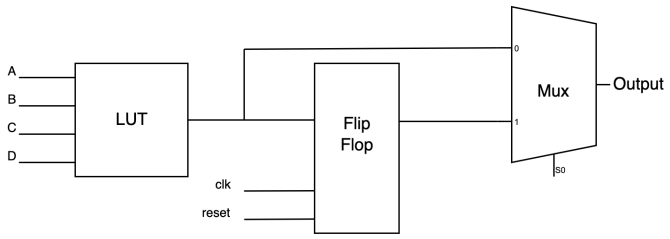

Figure 1.3. Illustration of a Configurable Logic Block (CLB) containing LUTs and flip-flops.

2. Interconnect Network:

The interconnect fabric is a hierarchical and programmable wiring structure that allows logic blocks to be connected to each other or to I/O pins. It includes configurable switches, multiplexers and routing tracks. The flexibility of the interconnect enables the same physical fabric to implement very different designs.

3. Input/Output Blocks (IOBs):

IOBs interface the FPGA with external signals. They support various I/O standards and can be configured as inputs, outputs or bidirectional ports.

4. Dedicated Hardware Resources:

Modern FPGAs include specialized blocks beyond the generic logic in CLBs. These blocks are optimized for common digital design tasks.

- *Block RAM (BRAM):* On-chip memory blocks that can be configured as single or dual-port RAMs. These are used to store buffers, image data or network weights, which is particularly useful in applications like BNNs where fast memory access is needed.
- *DSP Slices:* Fixed function blocks designed for fast arithmetic operations such as multiplication, addition, accumulation and bit-shifting. These slices offload high speed math operations from CLBs, enabling efficient implementation of signal processing.
- *Clocking Resources:* FPGAs include a network of global and regional clocks that distribute timing signals to different parts of the device. These networks ensure that all sequential elements operate in sync, even across large designs.

1.2.2. Reconfigurability and Design Flow

The term ”field programmable” refers to the FPGA's ability to be configured after manufacturing. This is achieved by loading a bitstream file, generated by synthesis and implementation tools, into the FPGA. The design process typically includes:

- Writing HDL code (such as Verilog or VHDL) to describe hardware behavior
- Synthesizing the HDL into logic primitives
- Mapping the logic to available resources (LUTs, FFs, BRAM etc.)
- Placing and routing the design to satisfy timing constraints
- Generating and loading the bitstream onto the FPGA

This process takes more time and is more complex than software compilation but it produces fast, parallel hardware that's specifically built for the task.

1.2.3. Advantages and Trade-Offs

FPGAs combine flexibility, parallelism and low power usage, making them well-suited for running Binary Neural Networks (BNNs). Compared to CPUs, GPUs and ASICs, they offer a hardware platform that can closely match the needs of the task.

Custom Parallelism: FPGA designs let developers build deeply pipelined or highly parallel data paths that match the model's needs. This allows the hardware structure to follow the model, not the other way around.

Predictable Timing: Unlike CPUs and GPUs which rely on operating systems and complex memory hierarchies, FPGAs offer cycle-accurate execution. This makes them ideal for real-time systems where timing must be consistent.

Reusability and Flexibility: FPGA designs can be updated after deployment, allowing changes to the architecture, model structure or logic without changing the hardware. This is especially useful in prototyping or evolving systems.

Power Efficiency: When configured carefully, FPGAs can be more power efficient than CPUs or GPUs for certain workloads, especially when high levels of parallelism reduce idle cycles and memory movement.

However, these strengths come with several trade-offs:

- **Steeper Learning Curve:** Developing for FPGAs requires knowledge of hardware design and HDL languages like Verilog or VHDL. Debugging and verifying hardware also adds complexity and time.
- **Lower Clock Speeds:** FPGA logic usually operates at lower frequencies compared to CPUs, GPUs and ASICs, which can limit performance for tasks that aren't highly parallel.
- **Limited Resources:** The number of logic blocks, memory elements and DSP slices is fixed and must be managed carefully. High parallelization can quickly exhaust available resources.
- **Longer Compilation Times:** Unlike software, FPGA designs must go through synthesis, placement and routing steps. These can take minutes to hours depending on design size and complexity.

1.3. FPGA Deployment for Neural Networks

As shown in [6], FPGAs have been used in areas like signal processing and control systems. More recently, their support for custom dataflows and low-precision operations has made them a strong candidate for efficient machine learning models like BNNs.

As deep learning models grow increasingly complex and resource-intensive, they become harder to deploy on limited hardware. However, studies show that many deep networks contain redundant parameters [7], suggesting that similar accuracy can often be achieved using smaller or lower-precision models. This creates an opportunity for more efficient models like BNNs, which are easier to implement on FPGAs. BNNs can be either fully connected or convolutional and are ideal for embedded devices where power and hardware constraints are critical.

A practical approach for BNN deployment on an FPGA is to handle training on conventional hardware like CPUs or GPUs, since training directly on FPGAs is inefficient and complex. After training, the binarized weights and thresholds are extracted and loaded into the FPGA's memory. The FPGA can then run inference efficiently using these parameters.

1.4. Terms

- *Quantization-Aware Training (QAT)* is a training technique that accounts for quantization effects during both the forward and backward passes. It helps neural networks retain performance after being quantized for deployment in hardware environments.
- *XNOR-Popcount* is the bitwise alternative to multiplication and addition in BNNs. The XNOR operation compares input and weight bits and popcount counts the number of ones, representing the similarity between inputs and weights.
- *Threshold* in this project refers to a precomputed integer derived from batch normalization parameters. During inference, a neuron's output is compared against this threshold to decide whether it activates (1) or not (0).
- *Finite State Machine (FSM)* is a control structure used in the Verilog implementation to coordinate each stage of inference. It defines states such as image loading, layer computation, output extraction and completion.
- *ROM (Read-Only Memory)* is used to store binarized weights, thresholds and input images on the FPGA. These are loaded using `.mem` initialization files during simulation and deployment.
- *Inference* refers to the process of making predictions with the trained model. In this project, inference is implemented entirely on the FPGA using custom Verilog logic to process input images and produce digit outputs.
- *Simulation* is the process of verifying the design before hardware deployment. Functional simulations with waveform analysis were used to debug signal behavior, control flow and output correctness.

1.5. Problem Definition

Modern Binary Neural Networks (BNNs) offer a powerful way to reduce computational cost by using bitwise logic instead of traditional floating-point operations. This makes them ideal for deployment on hardware-constrained platforms like FPGAs. However, many BNN implementations in hardware are built using high-level synthesis (HLS) tools or generic frameworks that abstract away the underlying logic. While these tools accelerate development, they also limit visibility into how the network operates at a low level, especially in terms of data flow, timing and logic decisions. This lack of transparency makes it harder to optimize, debug or fully understand the hardware behavior of binarized inference systems.

The core problem addressed in this project is the lack of accessible, low-level BNN implementations that can be directly studied and customized. To solve this, the project builds a complete BNN inference system from scratch in Verilog without any reliance on HLS. The target is handwritten digit classification using the MNIST dataset. All inference steps: binarized weight loading, XNOR-popcount operations, threshold comparison and argmax output, are implemented using cycle-accurate FSMs on a Nexys A7-100T FPGA. This allows full control over memory layout, logic structure and timing which provides a transparent and hands-on framework for understanding binary inference at the hardware level.

1.6. Scope and Limitations

This project focuses on the FPGA based inference stage of a trained BNN. The model is trained offline using Python and TensorFlow with QAT and then exported in binary format for hardware deployment.

The design has a few limitations. First, it only supports fully connected architectures and convolutional layers are not included. Second, test images are stored in static ROMs, which means changing the input set requires re-synthesis. Third, the Verilog code is specific to the trained model architecture because the FSM and memory accesses are hardcoded for a particular number of layers and neurons. Supporting a different model would require significant code changes, as each layer is implemented as a separate FSM state with fixed dimensions.

1.7. Requirements

1.7.1. Functional Requirements

(i) **Input Acquisition and Binarization**

- The system must accept 28×28 grayscale images converted offline into 1 bit binary form, compatible with the pre-trained model's input format and load them into ROMs for inference.

(ii) **XNOR-Popcount Based Layer Computation**

- The system must compute binary matrix multiplications using XNOR and popcount operations.
- Each layer must perform threshold comparisons using precomputed thresholds.

(iii) **Parallel Inference Execution**

- The system must support variable levels of neuron parallelism.
- Finite State Machine logic must coordinate the sequential processing of neuron batches within each layer.

(iv) **Weight and Threshold Memory Access**

- The system must access weights using dual-port ROMs instantiated from .mem files and optimized for parallel reads.
- Thresholds must be retrieved from single-port ROMs stored as signed integer values.

(v) **Digit Prediction Output**

- The system must compute a 4-bit digit output via an argmax logic over final layer scores.

(vi) **Correctness**

- The inference result must match expected predictions from the trained model.

(vii) **Real-Time Response**

- The system must complete inference within a bounded and deterministic latency, ensuring results are produced fast enough for time sensitive applications.

1.7.2. Nonfunctional Requirements

(i) **Performance**

- The system must operate with minimal latency across tested parallelization levels.
- Throughput must be sufficient for real-time classification on the Nexys A7 FPGA at 80 MHz.

(ii) **Scalability**

- The design must support different levels of parallelism via parametrization without structural rewrites.

(iii) **Resource Efficiency**

- The system must utilize only the minimum number of memory and logic blocks necessary to support a given parallelization level, ensuring no redundant ROM instantiations or unused modules.

(iv) **Portability**

- The system must be deployable on other mid-tier FPGA boards with minimal changes, assuming similar resource profiles.

(v) **Maintainability and Extensibility**

- The Verilog implementation must remain modular, with clearly separated modules for FSM, memory, computation and display logic.

(vi) **Verification and Simulation**

- A testbench must be maintained for simulation based waveform inspection and debugging.
- Simulation results must confirm that the hardware inference outputs match the predictions of the trained model, achieving accuracy consistent with the model's baseline performance.

(vii) **Usability**

- The design must include visual output via seven-segment display to confirm inference predictions on hardware.

(viii) **Compliance**

- The system must comply with synchronous digital design principles, using standard Verilog and applying blocking and non-blocking assignments appropriately.

2. RELATED WORK

2.1. Binary Neural Networks

Courbariaux et al. introduced BinaryNet [8], one of the first methods to train deep networks using binary weights and activations. Their approach applies the sign function during the forward pass and uses a straight-through estimator (STE) during backpropagation to approximate gradients. Real-valued shadow weights are maintained to improve convergence. These choices reduce memory usage and replace multiplications with efficient XNOR and popcount operations, enabling faster and more lightweight inference.

XNOR-Net by Rastegari et al. [9] takes the idea further by applying binarization to convolutional layers. To improve accuracy, they introduce scaling factors that help binary convolutions better approximate their full-precision counterparts. This approach achieves a balance between computational efficiency and performance, showing that BNNs can work well even on complex tasks like ImageNet classification when carefully tuned.

2.2. BNN Deployment on FPGAs

BNNs are a perfect match for FPGA deployment because their binary operations are simple and efficient to implement in hardware. As a result, there has been growing interest in using FPGAs to run BNNs. FINN [10] developed by Xilinx, is a framework for building and evaluating BNNs on reconfigurable platforms. It uses high-level synthesis (HLS) and connects building blocks like matrix-vector units in a dataflow setup, making it easy to scale and efficient to run on FPGAs.

Another implementation by Zhao et al. [11] targets convolutional BNNs using software-programmable FPGAs. Their design is written in C++ and synthesized to Verilog using HLS tools and deployed on a Xilinx Zynq platform. The work focuses on accelerating binary convolutional layers and achieving high performance in terms of GOPS and energy efficiency compared to traditional CNN accelerators.

2.3. Limitations of Current Approaches

Although previous studies have demonstrated the potential of BNNs for both training and deployment, several limitations still exist. Many hardware implementations depend on HLS

tools which abstract away the underlying logic. This reduces visibility into how data moves through the system and makes it harder to debug or fine-tune performance at the hardware level.

In addition, most existing work focuses on convolutional models. These are useful for complex vision tasks but often unnecessary for simpler datasets like MNIST where fully connected networks can perform well using fewer resources.

2.4. Comparison with This Work

This project differs in several key aspects. It implements a fully connected BNN trained with QAT and deploys it on the Nexys A7-100T FPGA entirely in Verilog with no use of high-level synthesis. Every part of the inference process from XNOR-popcount computation to thresholding and state-machine control is written manually, offering full control and transparency.

Unlike frameworks like FINN or other HLS-based designs, this implementation provides direct insight into how each bit is processed, how intermediate values are handled and how control flows between layers. No existing BNN implementation has been found that provides this level of manual control over both the model architecture and its bit-level execution on FPGA.

3. ANALYSIS & DESIGN

3.1. Input Preprocessing and Dataset

The MNIST dataset was used as the primary source of input data. It consists of 60,000 training images and 10,000 test images, each representing a handwritten digit between 0 and 9. Every image is a grayscale 28×28 pixel matrix which is 784 pixel values per sample.

The dataset was chosen because of its simplicity and wide use in evaluating classification models. Its low resolution and clearly defined labels make it ideal for training lightweight neural networks such as BNNs.

Before feeding the data into the model, all pixel values were normalized to the range of -1 to 1. This normalization is important for compatibility with the binarization functions used in BNNs. Additionally the input images were reshaped into one-dimensional vectors to fit the dense layer based model architecture used in this work.

3.2. Binary Neural Network Architecture

This section describes the BNN architecture used during the training phase. The model was developed and trained using software tools and the final binarized weights and thresholds were later exported for deployment on the FPGA.

The architecture of the BNN designed for this project is detailed in Table 3.1 and Figure 3.1. The network follows a simple but highly efficient structure using fully connected (`Dense`) layers combined with binarization at both the weight and activation levels.

The input to the network consists of 28×28 grayscale images from the MNIST dataset which are flattened into 784 dimensional vectors. These vectors are inputs to the first fully connected layer, which maps the 784 input features to 128 binary activations. Both the weights and activations at this stage are binarized using the sign function and batch normalization is applied to the outputs to stabilize training.

The second layer takes the 128 dimensional binary activations and produces 64 binary activations through another fully connected binarized layer. Similar to the first layer, batch normalization is applied to maintain training stability.

The final layer maps the 64 binary features to 10 outputs, corresponding to the ten digit classes (0–9). Unlike the hidden layers, the outputs are left as raw logits which are later processed using a softmax activation during evaluation or classification.

All weight parameters in the hidden and output layers are binarized to 1 bit precision which significantly reduces the memory usage and computational complexity of the network. The activations are also binarized in the hidden layers.

Layer	Type	Input Size	Output Size	Activation Function
Input Layer	–	28×28 grayscale	784 (flattened)	–
Dense Layer 1	Fully Connected	784	128	Sign + BatchNorm
Dense Layer 2	Fully Connected	128	64	Sign + BatchNorm
Output Layer	Fully Connected	64	10	None (raw logits)

Table 3.1. Detailed Architecture of the Binary Neural Network Used in This Project

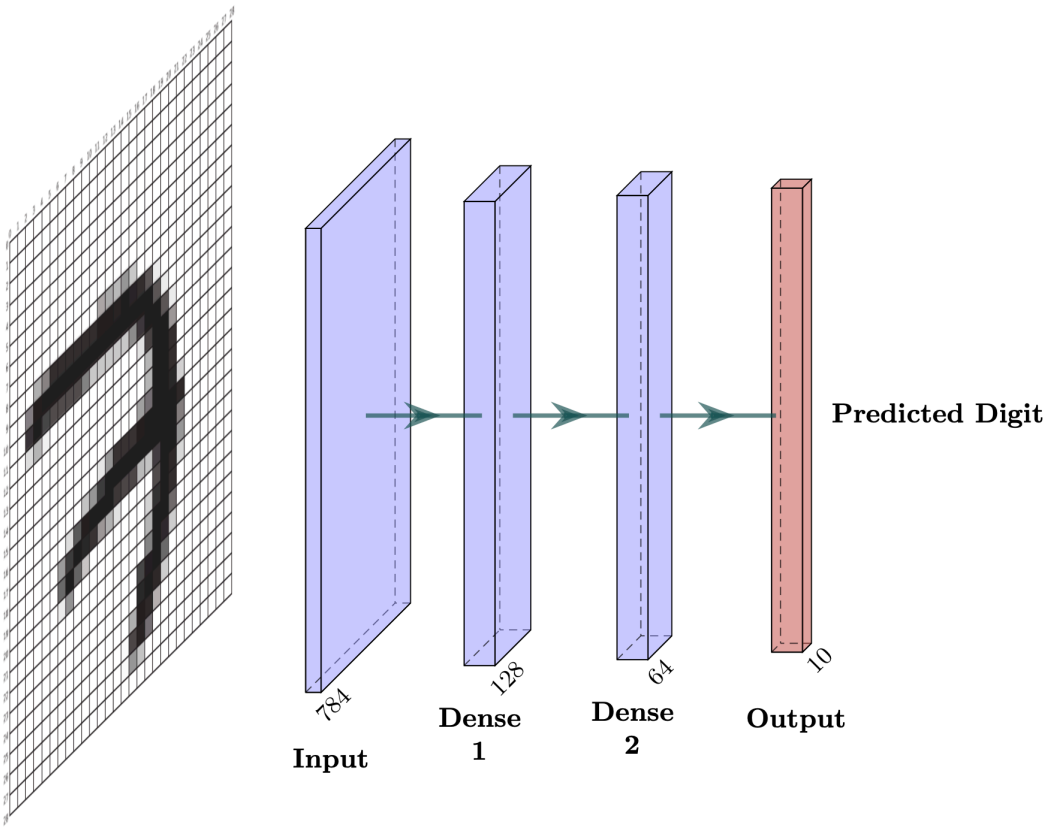

Figure 3.1. High-level visual representation of the BNN architecture.

Figure 3.2 shows the sequence of operations performed during the training phase of the BNN. This sequence diagram summarizes how the dataset is processed, the network is configured and the model is iteratively trained and validated. After convergence the trained weights are exported for hardware deployment.

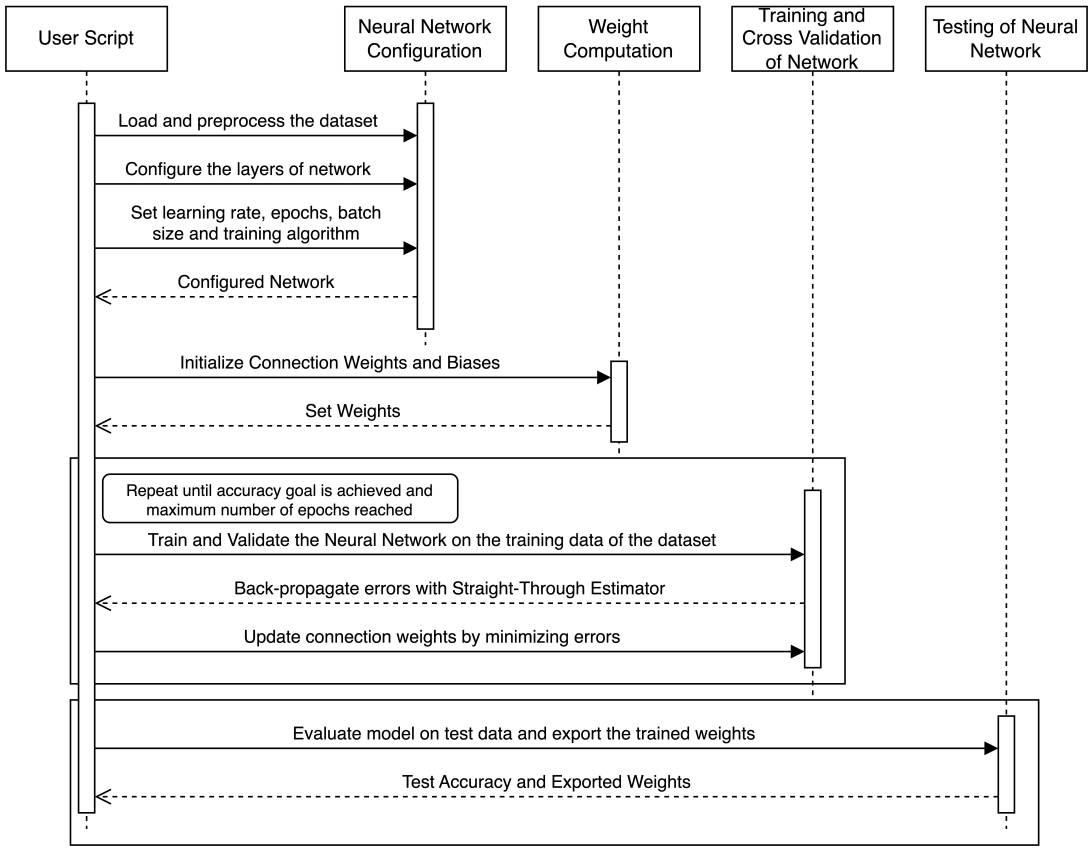

Figure 3.2. Sequence diagram of the training process for the Binary Neural Network.

3.3. Hardware Inference Design on FPGA

This section describes how the trained BNN is implemented for inference in hardware, detailing the control logic, memory organization and output handling.

3.3.1. Inference Operation

The inference design starts from the input image and progresses through binary matrix operations at each layer. For every neuron, the activations from the previous layer are XNORed with the binarized weights. The XNOR results are passed through a popcount operation which counts the number of matching bits.

The popcount value is then compared with a predefined threshold specific to each neuron. If the popcount exceeds the threshold, the neuron output is set to 1 and otherwise it's set to 0. This process replaces standard activation functions with a binary decision step based on thresholds.

Intermediate outputs between the layers are kept in binary form to reduce memory and computation requirements. In the final output layer, the results are stored as integer scores rather than binary outputs to allow the predicted class to be determined with increased accuracy. The inference pseudocode summarizing these steps is shown in Figure 3.3.

```
function BNN_Inference(input_vector, weight_matrices, thresholds):
    // input_vector: binary input activations
    // weight_matrices: list of binarized weight matrices for each layer
    // thresholds: list of threshold vectors

    for layer_index in 0 to number_of_layers - 1:
        current_weights = weight_matrices[layer_index] // shape:[input_size][output_size]
        current_thresholds = thresholds[layer_index]   // shape:[output_size]
        next_activations = []

        for output_neuron in 0 to output_size - 1:
            popcount = 0
            for input_index in 0 to input_size - 1:
                input_bit = input_vector[input_index]
                weight_bit = current_weights[input_index][output_neuron]
                if XNOR(input_bit, weight_bit) == 1:
                    popcount += 1

            z = 2 * popcount - input_size
            if z >= current_thresholds[output_neuron]:
                next_activations.append(1)
            else:
                next_activations.append(0)

        input_vector = next_activations  // output of this layer becomes input to the next

    return input_vector  // Final output activations (logits or class scores)
```

Figure 3.3. Pseudocode for BNN inference using binary matrix operations and thresholds.

While the pseudocode illustrates the core arithmetic steps involved in inference, the sequencing of these operations is managed by a finite state machine (FSM). The control flow of this FSM is presented in Figure 3.4 where each state represents a distinct stage of the inference process and transitions occur based on completion flags at the layer level.

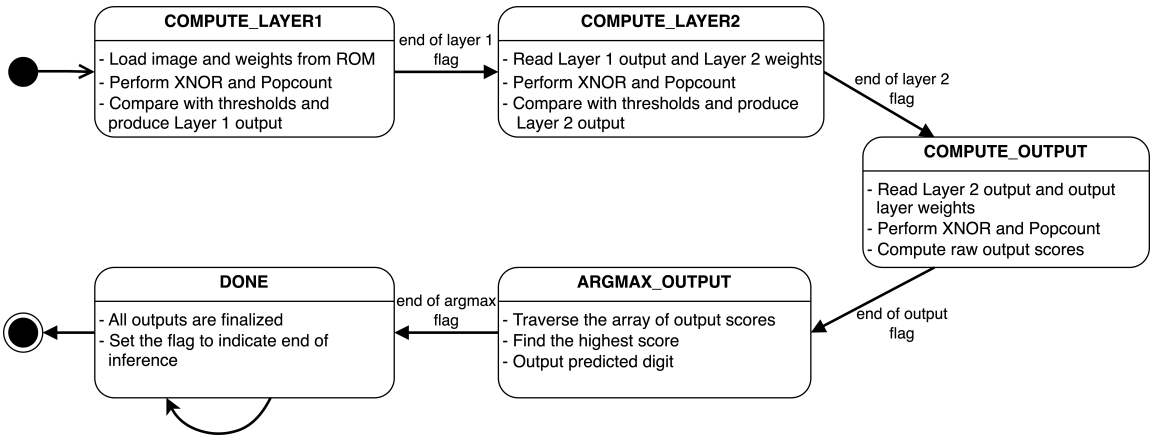

Figure 3.4. State diagram of the BNN inference controller FSM.

3.3.2. Memory Architecture

To support fast and reliable inference, all weights, thresholds and input images are stored in read-only memory blocks. These are generated from the trained model and formatted as binary files containing only the values needed during inference. Each ROM is accessed in a specific pattern: weights are read column-wise to match neuron indexing while images are read row-wise to stream pixel inputs into the first layer.

All memory blocks are built using dual-port access. This allows the system to read two values per clock cycle, which improves throughput. Address values for the ROMs are incremented in hardware based on the current computation stage and internal counters managed by the FSM. This structure keeps the memory usage simple but effective since only the relevant values are accessed at each step and no dynamic memory allocation is required.

Using dual-port ROMs with well managed address control lets the system read data efficiently without wasting resources, keeping the inference flow fast and lightweight.

3.3.3. Module Hierarchy

At the top level, the system connects three main components: the image memory, the core inference logic and the output display. The image ROM is instantiated in the top module and provides input data directly to the inference unit, which contains all the control and computation logic. This includes the FSM controller, XNOR, popcount, threshold blocks and registers to hold intermediate results.

The binarized weights and thresholds are stored in dedicated ROMs inside the inference unit. These are accessed using address lines generated by the FSM and their outputs feed directly into the binary computation flow.

Once inference is complete, the predicted digit is passed to a display module that formats the output for a seven-segment display. The entire flow from input fetch to display update is coordinated by control signals within the inference unit to keep the system synchronized and modular.

3.4. Deployment Platforms

Selecting the right hardware platform is crucial when designing machine learning systems for embedded applications where power, speed and cost are constrained. BNNs are particularly suited to certain platforms due to their lightweight, bitwise operations. This section compares four platforms: CPU, GPU, ASIC and FPGA, focusing on their trade-offs in performance, flexibility, development effort and suitability for BNN deployment.

This comparison is included in the *Analysis & Design* chapter because platform choice directly shapes both model architecture and hardware design decisions.

3.4.1. CPU (Central Processing Unit)

CPUs are general-purpose processors commonly used in traditional computing systems. They offer low initial cost and fast prototyping through familiar high-level software environments. While CPUs can handle BNN inference for testing or development, they are not optimized for bitwise operations or high parallelism. Their limited thread-level concurrency and higher power usage make them less suitable for large-scale or real-time deployments.

- **Initial Cost:** Low. Commodity CPUs are widely available in the $100-$400 range.
- **Development Time:** Very short. High-level languages and software stacks allow immediate prototyping.
- **Parallelism:** Moderate. Typically supports 4-16 threads with shared memory, limiting scalability for high-throughput tasks.
- **Energy Efficiency:** Low for parallel tasks. Power consumption increases significantly with workload size.
- **Suitability for BNNs:** Acceptable for validation or CPU based benchmarking, but not viable for deployment at scale.

3.4.2. GPU (Graphics Processing Unit)

GPUs consist of thousands of smaller cores designed to handle parallel workloads efficiently. They were originally developed for graphics rendering but their architecture has been repurposed for general-purpose computationally intensive tasks, especially in deep learning. GPUs offer high throughput for neural networks and excel in floating-point operations.

- **Initial Cost:** Moderate to high. High-end GPUs cost between $1,000 and $10,000.
- **Development Time:** Moderate. Frameworks like CUDA, PyTorch and TensorFlow accelerate development but require some learning curve.
- **Parallelism:** Very high. Thousands of threads can be executed concurrently, ideal for large scale training.
- **Energy Efficiency:** Moderate. High compute density but power usage scales with workload and memory usage.
- **Suitability for BNNs:** Good for training or batch inference but not optimal for lightweight, real-time applications.

3.4.3. ASIC (Application-Specific Integrated Circuit)

ASICs are custom designed chips fabricated to perform a specific task with extreme efficiency. Inference on an ASIC is typically orders of magnitude faster than on general purpose hardware due to dedicated logic for every operation. Despite their performance, ASICs suffer from lack of flexibility. Any change in model structure or logic requires a new fabrication cycle.

- **Initial Cost:** Extremely high. NRE costs can range from $100,000 to several million dollars depending on complexity.
- **Per-Unit Cost (High Volume):** Very low. Once produced, ASICs can cost as little as $5-$20 per chip in mass production.
- **Development Time:** Long. Typical design and tape-out cycles exceed 6 months.
- **Parallelism:** Maximum. Entire BNN pipeline can be hardwired into the chip.
- **Energy Efficiency:** Highest. Tailored dataflow and power gating yield unmatched performance-per-watt.
- **Suitability for BNNs:** Ideal for commercial deployment at massive scale, but impractical for research or evolving models.

3.4.4. FPGA (Field-Programmable Gate Array)

FPGAs offer a reconfigurable fabric of logic blocks that can be tailored for specific workloads post-manufacturing. This allows for customized data paths, pipelining and parallelism, making FPGAs particularly effective for BNNs, where the majority of operations are binary.

- **Initial Cost:** Low to moderate. Boards like the Basys 3 and Nexys A7 are priced around $150-$400 with no NRE overhead.
- **Development Time:** Moderate. Verilog/VHDL design and synthesis flows require hardware familiarity.
- **Parallelism:** High and configurable. Varies with resource availability.
- **Energy Efficiency:** High for bitwise workloads. Unused logic can be disabled during synthesis.
- **Suitability for BNNs:** Excellent. Especially valuable in research, edge deployment and model-specific customization.

Each platform has strengths depending on the use case. CPUs and GPUs are easy to develop on but dont perform well in low-latency or resource constrained scenarios. ASICs offer unmatched efficiency but are only practical for large-scale production due to high upfront costs and fixed designs.

FPGAs achieve a balance between speed, flexibility and resource efficiency. Their reprogrammable nature and support for bit-level customization make them ideal for BNNs, especially in research and development where models evolve frequently.

Section Platform Comparison Results in Chapter 5 presents detailed measurements comparing FPGA performance against these platforms in terms of latency, power and resource usage.

4. IMPLEMENTATION

The development of the Binary Neural Network (BNN) model and its deployment onto the FPGA required a combination of software frameworks and hardware design tools, each specific to different stages of the project. High-level tools were used during model development and training while low-level HDL design and simulation tools were employed for hardware implementation. Table 4.1 summarizes the main tools and frameworks used throughout the project along with their roles.

Component	Tool	Purpose
Training Framework	TensorFlow	Training the BNN model and managing neural network layers.
BNN Library	Larq	Provides binarized layers and quantization-aware training support.
Programming Language	Python	Used for scripting, training and weight export.
Simulation + Synthesis	Xilinx Vivado	Used for RTL design, waveform inspection, synthesis and implementation.
HDL Language	Verilog	Describes all hardware modules, FSMs, memory logic and inference operation.
Input/Output Medium	`.mem` Files	Store binarized weights, thresholds and test images for ROM initialization.

Table 4.1. Software and Hardware Tools Used in the Project

While comparative analysis was conducted across multiple levels of neuron parallelization, all implementation screenshots, synthesis reports and power analysis results presented in this chapter correspond to the configuration using **64-level parallelism** with **dual-port BRAMs** for weight storage. This configuration was selected because of its balanced trade-off between inference speed and resource utilization.

4.1. Model Architecture and Training

The Binary Neural Network (BNN) model was trained using the **TensorFlow** deep learning framework with **Keras** as its high-level API. To support binarized operations, the **Larq** library was used. Larq extends Keras by introducing binary compatible layers and quantization-aware training capabilities, enabling both weights and activations to be reduced to 1-bit during training.

To ensure reproducibility, fixed random seeds were set for Python, NumPy and TensorFlow, keeping weight initializations consistent across training runs. The model architecture consists of:

- **Input Layer:** 784 features corresponding to 28×28 grayscale image pixels, normalized to the range $[-1, 1]$.
- **First Hidden Layer:** 128 binarized neurons fully connected to the input, followed by batch normalization.
- **Second Hidden Layer:** 64 binarized neurons fully connected to the first layer, followed by batch normalization.
- **Output Layer:** 10 neurons with binarized weights but real-valued outputs, used for final argmax classification.

To improve convergence and stability, Batch Normalization layers were placed after each hidden and output layer. This ensures consistent feature scaling and supports threshold folding during FPGA inference.

The model is compiled using the **Adam optimizer** which is a widely used gradient-based optimization algorithm. Adam adjusts the learning rate for each parameter individually by tracking both the average and variability of recent gradients. This helps the optimizer converge efficiently and reliably, even when the gradients are sparse or noisy.

To improve training stability and convergence more, an *exponential learning rate decay schedule* is applied. The learning rate starts at 0.001 and decays by a factor of 0.96 every 1000 steps using a staircase strategy, which means that the rate remains constant within each interval before dropping at fixed steps. These hyperparameters were tuned experimentally: higher starting rates like 0.01 led to unstable training and sharp fluctuations in accuracy while lower values like 0.0005 resulted in slow convergence. The selected configuration provided the best balance to achieving smooth and fast convergence without overfitting or divergence.

The loss function used is **Sparse Categorical Crossentropy**, which is suitable for multi-class classification tasks where the target labels are integers. Crossentropy measures the difference between the true class distribution and the predicted probability distribution output by the model. It penalizes incorrect predictions more heavily when the model is confident

but wrong which encourages the network to assign high probability to the correct class. In this case, with digit labels ranging from 0 to 9, the model learns to minimize the crossentropy between the actual digit label and the softmax output. Training was conducted over **15 epochs** with a **batch size of 64**, meaning the model's parameters were updated after every 64 examples u sing mini-batch gradient descent. This batch size offers a good balance between training speed and stable gradient updates to allow the model to converge efficiently. It was also found to be the optimal setting, along with 15 epochs, before further increases in training time began to show diminishing returns in accuracy.

The final trained model achieved a test accuracy of **87.97%** on the standard MNIST test set. This performance is very high given the binarization constraints and shows the efficiency of the network architecture.

The model was saved in the `.h5` format containing both the network weights and batch normalization statistics. This format made it easy to extract and convert the data for FPGA implementation.

4.2. Model Export and Hardware Formatting

After training, weights and Batch Normalization parameters were extracted using structured access paths corresponding to each `QuantDense` and `BatchNormalization` layer from the `.h5` file that stored the trained model. Since the model uses no bias terms, only kernel weights and normalization statistics (moving mean, variance and beta offset) were extracted. These values were later converted into binary or threshold representations and exported as `.mem` files.

To match the memory access pattern of the FPGA, the weight matrices were transposed after exporting them. This ensures that each row in the memory corresponds to a single neuron's full set of weights to enable efficient ROM access during inference.

Batch Normalization parameters were folded into static integer thresholds to simplify the hardware implementation. Folding refers to the process of merging the learned BatchNorm behavior that was originally applied as a floating-point operation at runtime, into a fixed threshold value that can be used during inference. This is essential for FPGA deployment where minimizing floating-point computation is important for speed and resource efficiency.

The folding is performed using the formula:

$$\theta = \left\lfloor \beta - \frac{\mu}{\sqrt{\sigma^2 + \epsilon}} \right\rfloor$$

where β is the learned offset, μ is the moving mean, σ^2 is the variance and ϵ is a small constant added for numerical stability. The resulting thresholds were quantized to signed 11-bit integers and stored in `.mem` files which were directly loaded into threshold ROMs on the FPGA.

Thresholds were extracted and folded for the first two hidden layers to allow the XNOR–popcount outputs to be compared directly against them using simple comparators. However for the output layer, raw floating-point logits were retained instead of binarizing the outputs. This decision was made to preserve classification accuracy, as the final activation outputs are more sensitive to quantization and directly influence the predicted class.

4.3. Target FPGA Board

The final hardware design was implemented on the **Nexys A7-100T FPGA Development Board** by Digilent based on the Xilinx Artix-7 XC7A100T FPGA. This board was selected because of its large amount of logic and memory resources.

4.3.1. Key specifications

- **FPGA Chip:** Xilinx Artix-7 XC7A100T-1CSG324C, providing 15,850 logic slices, each consisting of four 6-input Look-Up Tables (LUTs) and eight Flip-Flops.
- **Logic Resources:** 63,400 LUTs and 126,800 Flip-Flops available for combinational and sequential logic design.
- **Block RAM:** 4,860 Kbits of on-chip BRAM that's equivalent to 135 RAMB18 blocks, used for high throughput dual-port ROMs to store model weights.
- **DSP Slices:** 240 DSP slices are available on the chip but they are not utilized in this project since the BNN relies only on bitwise XNOR and popcount operations.
- **Clocking:** Although the board supports internal clock speeds exceeding 450 MHz, the design operates at a fixed frequency of 80 MHz. This value was selected after timing analysis showed that the design failed to meet setup time requirements at 100 MHz due

to long combinational paths and increased routing complexity introduced by higher levels of parallelism. At 80 MHz the design consistently achieves timing closure and ensures stable and reliable operation without critical warnings.

- **Visual Output:** A 4-digit seven-segment displays is used to show the final digit prediction for inference verification on hardware.

This platform provides sufficient capacity to test multiple configurations of the binary neural network, especially under varying levels of neuron parallelization.

While all simulations, synthesis and implementation steps were performed specifically targeting the Nexys A7-100T board using the Vivado Design Suite, the bitstream was not programmed onto the physical FPGA and no on-board testing was conducted during this project.

4.4. Hardware Architecture Overview

The hardware design implements the inference operation of a trained BNN using a state driven control mechanism and modular memory interfaces. The inference logic processes a 28×28 binarized image input through three fully connected layers.

- **Input Interface:** The input is a 784-bit wire named `image`, representing a flattened 28×28 binarized image. It is driven by the `image_rom` module which retrieves data from `.mem` files containing preprocessed MNIST test images. Each image was converted from grayscale to binary using a Python script that applied normalization. Pixels were mapped to a single bit: `1` for ink and `0` for background. This preprocessing ensured compatibility with the trained BNN model and avoided on-chip computation during inference.
- **Memory Modules:** All model weights are stored in dedicated dual-port ROMs, one for each layer:
  - `dense_kernel_rom_dualport` outputs to `dense_w0[]` for Layer 1
  - `dense_1_kernel_rom_dualport` outputs to `dense_w1[]` for Layer 2
  - `dense_2_kernel_rom_dualport` outputs to `dense_w2[]` for the Output Layer

  Each ROM module is initialized with a corresponding `.mem` file containing the binarized weights for that layer. These files were generated during the export process in Python and formatted to match the access pattern expected by the Verilog design.

The modules are instantiated using `generate` blocks with a `genvar` loop which programmatically replicates the hardware structure for each pair of parallel neurons to enable scalable parallelization without manual duplication. The loop increments by 2 to match the dual-port access, enabling two weight rows to be fetched per clock cycle, one on each port (`addr_a` and `addr_b`). This setup enables high-throughput inference by servicing multiple neurons simultaneously.

- **Threshold Modules:** For the first two layers, thresholds derived from folded BatchNorm parameters are stored in LUT-based single-port ROMs:
  - `dense_kernel_thresholds_rom_lut` outputs to `thresh_w0[]` for Layer 1
  - `dense_1_kernel_thresholds_rom_lut` outputs to `thresh_w1[]` for Layer 2

  These are also instantiated using a `genvar` loop for parallel access. Each iteration creates two separate threshold ROMs, one per neuron, using the same indexing as the dual-port weight ROMs. Threshold values were also precomputed and quantized in Python and saved as signed integers in `.mem` files.

  No threshold ROM is instantiated for the final output layer. Instead, its raw integer logits are passed directly to the `argmax` comparison stage to preserve accuracy to avoid quantization loss at the final classification step.

- **Output Buffers:** Each stage of the inference operation stores its results in dedicated output registers, which act as intermediate buffers between layers:
  - `layer1_out` is a `reg [127:0]` signal that stores the binary activation results of Layer 1. Each bit corresponds to the activation of one neuron that is obtained by thresholding the popcount result of the XNOR operation with the preloaded thresholds.
  - `layer2_out` is a `reg [63:0]` signal that captures the binary activations of Layer 2 neurons. Like `layer1_out`, it is written in parallel once all popcount computations and comparisons are completed.
  - `output_scores` is a register array defined as `reg signed [9:0] output_scores [0:9]`, which holds the final integer logits produced by the Output Layer. Unlike earlier layers the outputs here are not binarized. Instead, the raw accumulated popcount values are retained for accuracy and forwarded to the argmax logic.

  The bit widths of each buffer directly match the number of neurons in each layer to maintain the alignment with the model's architecture.

- **Computation Flow:** For each neuron in a layer, the module performs bitwise XNOR between input activations and stored weights, accumulates the popcount and applies a threshold to produce the next layer's activation.
- **Digit Output:** The predicted digit is stored in the `digit` wire after performing an argmax operation over the `output_scores` array.

The architecture is parametrized using the `PARALLEL_NEURONS` macro, which determines the number of neurons processed concurrently by instantiating parallel memory and compute logic for each layer.

4.4.1. Module Descriptions

The Verilog design follows a hierarchical structure. Each module handles a specific functionality to enable clean separation between computation, memory access, input feeding and output display. The overall module hierarchy is as follows:

- `bnn_top.v` *(Top-Level Module)*
  - Instantiates and connects the following components:
    - `bnn_inference`: Core inference logic
    - `image_rom`: Provides the binary input image
    - `seven_segment_decoder`: Displays the output digit
  - Handles reset, clock and port wiring
- `bnn_inference.v` *(Inference Core)*
  - Implements the main finite state machine for controlling inference flow
  - Contains:
    - Parallel XNOR-popcount logic for binary matrix operations
    - Threshold comparison and binary activation per neuron
    - Argmax logic to select the final prediction
  - Interfaces with weight and threshold ROM modules using address signals and memory fetch wires
- `model_rom.v` *(Central ROM Module for All Weights and Thresholds)*
  - Includes three dual-port BRAM based ROM modules:

        - `dense_kernel_rom_dualport.v`: Layer 1 weights (`dense_w0[]`)
        - `dense_1_kernel_rom_dualport.v`: Layer 2 weights (`dense_w1[]`)
        - `dense_2_kernel_rom_dualport.v`: Output layer weights (`dense_w2[]`)
    - Also includes:
        - `dense_kernel_thresholds_rom_lut.v`: Layer 1 thresholds (`thresh_w0[]`)
        - `dense_1_kernel_thresholds_rom_lut.v`: Layer 2 thresholds (`thresh_w1[]`)
    - No threshold ROM is used for the output layer since it outputs raw scores for argmax

- **`image_rom.v`** *(Input Image Memory)*
    - Stores a single binary encoded 28×28 image (784 bits) as `image[783:0]`
    - Image is initialized from an external `.mem` file
    - Used exclusively for simulation and synthesis time preload because real-time image loading is not supported in the current design

- **`seven_segment_decoder.v`** *(Display Interface)*
    - Converts the 4-bit `digit` output into control signals for 7-segment displays
    - Outputs a 7-bit vector (`segments`) corresponding to the numerical digit pattern
    - Used for visual verification of inference results on hardware

4.4.2. Finite State Machine Control

The inference process is driven by a centralized Finite State Machine (FSM), implemented using two 3-bit registers: `state` for the current state and `next_state` for the upcoming transition. The FSM coordinates the sequential computation of each layer, accumulation of results and generation of the final predicted digit.

Each major layer computation is decomposed into a three phase sequence:

- XNOR operations with parallel weight fetches
- Popcount accumulation into `popcount[]` registers
- Threshold comparison to set output activations or scores

The FSM consists of five named states:

- `COMPUTE_LAYER1`: Computes activations for the first hidden layer
- `COMPUTE_LAYER2`: Computes activations for the second hidden layer
- `COMPUTE_OUTPUT`: Computes raw output scores from the final layer
- `ARGMAX_OUTPUT`: Performs maximum value comparison to determine the predicted digit
- `DONE`: Signals completion of the entire inference process

The FSM always begins in the `COMPUTE_LAYER1` state upon reset. Transitions between states are governed by dedicated `done` flags for each stage: `done_layer1`, `done_layer2`, `done_output` and `done_argmax`. For instance, when all neurons in the first layer have completed their computation and thresholding, `done_layer1` is set and the FSM transitions to `COMPUTE_LAYER2`. This process continues until the final `DONE` state is reached. When the `reset` signal is asserted, all FSM states and internal registers such as counters, flags, and accumulators are cleared to prepare for a fresh inference cycle. This structured control mechanism ensures orderly and synchronized execution of all stages

The FSM traverses these five states sequentially:

I. COMPUTE_LAYER1:

- This state performs inference for the first hidden layer using binarized inputs and weights.
- A counter named `addr1` is used to iterate over all 784 input pixels (indexed from 0 to 783). It acts as the address for accessing each bit of the input image in a sequential manner.
- For each pixel (bit) a bitwise XNOR operation is performed between the input bit `image[addr1]` and the corresponding weight bit `dense_w0[j][783 - addr1]` for each neuron j in the current parallel batch. The reverse indexing (`783 - addr1`) is necessary because the weights were transposed during preprocessing for memory alignment so that each ROM row holds a complete set of weights for a neuron in reverse bit order.
- The output of the XNOR is 1 if the input and weight bits match (either both 0 or both 1) and 0 otherwise. These results are accumulated in the `popcount[j]` registers across the 784 input cycles.

- After all inputs are processed (`addr1 == 783`), a control signal `process_neuron1` is set to 1 in the next clock cycle which signals that popcount accumulation is complete for this neuron batch.
- For each neuron j, the intermediate sum z_j is computed using the formula:

$$z_j = 2 \cdot \texttt{popcount[j]} - 784$$

  This matches the mathematical formulation of a dot product between input and weight vectors where binary values are interpreted as {-1, +1}.
- Once z_j is computed and `z_ready` is still 0, the result is stored in `z[j]` and `z_ready` is set to 1.
- The folded threshold `thresh_w0[j]` is then compared against z_j. In the comparison if $z_j \geq$ `thresh_w0[j]`, the corresponding bit in `layer1_out[neuron_idx1 + j]` is set to 1, otherwise it is set to 0. This mimics the binary activation function used during training.
- After all neurons in the current parallel group are processed, the state resets `addr1`, `z_ready`, `process_neuron1` and all `popcount[j]` registers to prepare for the next group.
- The index `neuron_idx1` is incremented by `PARALLEL_NEURONS` to point to the next batch of neurons. Once all 128 neurons in Layer 1 have been processed (`neuron_idx1 >= 128 - PARALLEL_NEURONS`), the control flag `done_layer1` is set to 1 which signals the FSM to transition to the next state.

II. COMPUTE_LAYER2:

- This state performs inference for the second hidden layer using the output of Layer 1 as its input.
- A counter named `addr2` is used to iterate over all 128 binary outputs of Layer 1 (indexed from 0 to 127). It acts as the address for accessing each bit of the previous layer's activation in a sequential manner.
- For each input bit a bitwise XNOR operation is performed between `layer1_out[addr2]` and the corresponding weight bit `dense_w1[j][127 - addr2]` for each neuron j in the current parallel batch. As in the previous layer, reverse indexing is used to match the pre-transposed memory format of the weights in ROM.

- The output of the XNOR is 1 if the layer output bit and the weight bit match (either both 0 or both 1) and 0 otherwise. These results are accumulated in the `popcount[j]` registers across the 128 input cycles.
- After all inputs are processed (`addr2 == 127`), a control signal `process_neuron2` is set to 1 in the next clock cycle which signals that popcount accumulation is complete for this neuron batch.
- For each neuron j, the intermediate sum z_j is computed using the formula:

$$z_j = 2 \cdot \texttt{popcount[j]} - 128$$

- Once z_j is computed and `z_ready` is still 0, the result is stored in `z[j]` and `z_ready` is set to 1.
- The folded threshold `thresh_w1[j]` is then compared against z_j. In the comparison if $z_j \geq$ `thresh_w1[j]`, the corresponding bit in `layer2_out[neuron_idx2 + j]` is set to 1, otherwise it is set to 0. This follows the same binary activation logic as the first layer.
- After all neurons in the current parallel group are processed, the state resets `addr2`, `z_ready`, `process_neuron2` and all `popcount[j]` registers to prepare for the next group.
- The index `neuron_idx2` is incremented by `PARALLEL_NEURONS` to point to the next batch of neurons. Once all 64 neurons in Layer 2 have been processed (`neuron_idx2 >= 64 - PARALLEL_NEURONS`), the control flag `done_layer2` is set to 1 which signals the FSM to transition to the next state.

III. COMPUTE_OUTPUT:

- This state performs inference for the output layer using the binary activations from Layer 2.
- A counter named `addr3` is used to iterate over all 64 bits of `layer2_out` (indexed from 0 to 63). It acts as the address to sequentially access each input bit for the final layer.
- At each step, a bitwise XNOR operation is performed between `layer2_out[addr3]` and the corresponding weight bit `dense_w2[j][63 - addr3]` for each output neuron

j in the current parallel batch. As in the previous layers, reverse indexing is used to match the pre-transposed memory format of the weights in ROM

- The results of the XNOR operation are accumulated in the `popcount[j]` registers over 64 cycles to measure how many input bits matched their corresponding weights for each neuron.
- Once all inputs are processed (`addr3 == 63`), the control signal `process_neuron3` is set to 1 in the next clock cycle, indicating that accumulation is complete.
- For each output neuron j, the final dot product approximation is computed as:

$$z_j = 2 \cdot \texttt{popcount[j]} - 64$$

- Unlike the hidden layers, no thresholding or activation is applied in this state. The raw z_j values are directly stored into `output_scores[neuron_idx3 + j]` to preserve full resolution for classification.
- After completing this step, `addr3`, `z_ready`, `process_neuron3` and all `popcount[j]` registers are reset to prepare for the next group of output neurons.
- The index `neuron_idx3` is incremented by `PARALLEL_NEURONS`. Once all 10 output neurons have been processed (`neuron_idx3 >= 10 - PARALLEL_NEURONS`), the control flag `done_output` is set to 1, allowing the FSM to move to the final comparison stage.

IV. ARGMAX_OUTPUT:

- This state performs the final classification step by identifying the index of the maximum value in the `output_scores` array which corresponds to the predicted digit.
- On initial entry into this state, the comparison process begins by initializing two temporary registers: `temp_max` is set to `output_scores[0]`, and `temp_index` is initialized to 0. A separate counter `compare_idx` is also set to 1 to start comparisons from the second score.
- At each clock cycle, `output_scores[compare_idx]` is compared to `temp_max`. If a larger value is found, both `temp_max` and `temp_index` are updated with the new maximum value and its corresponding index.

- The index `compare_idx` is then incremented to process the next score. This continues until all 10 class scores have been examined (`compare_idx == 9`).
- Once the maximum value has been found, `temp_index` which now holds the index of the highest score, is written to `predicted_digit` to complete the classification.
- At this point, the control flag `done_argmax` is set to 1 to signal completion of this stage and the internal control flag `argmax_started` is reset to prepare for future inference cycles.

V. DONE:

- This terminal state signals the end of the inference process.
- No further computation is performed; the FSM remains idle.
- The output control signal `done` is held high indefinitely to indicate completion.
- The FSM stays in this state until a `reset` signal is asserted, which restarts the process.

At this point, the predicted digit value is available on the `digit` output wire which is assigned directly from the internal `predicted_digit` register. The output signal `done` is also asserted when the FSM enters the `DONE` state.

In the top-level `bnn_top` module this digit is received by the `seven_segment_decoder`, which converts the 4-bit digit value into the corresponding 7-segment display encoding.

4.4.3. Parametrization and Scalability

Scalability in the design is achieved through the `PARALLEL_NEURONS` parameter, which defines how many neurons are processed in parallel during inference. Weight and threshold ROMs are instantiated using `genvar` loops that adapt to this parameter while internal indices increment accordingly to support batch-wise processing. By adjusting `PARALLEL_NEURONS`, the architecture scales to trade off between inference speed and hardware resource usage. Higher parallelism achieves lower latency but increases BRAM consumption with Vivado falling back to LUT-based ROMs when necessary.

4.5. Simulation, Synthesis and Implementation

The complete hardware flow was executed using the **Xilinx Vivado Design Suite**, covering simulation, synthesis, implementation and power analysis stages. Each stage was carefully evaluated to ensure the design met functional correctness, resource efficiency and timing closure targets for the selected FPGA platform.

4.5.1. Simulation and Verification

The inference design was verified using behavioral simulation in Vivado. Testbenches provided input to the `bnn_top` module, with pre-loaded `.mem` files supplying the input image, weights and thresholds. Waveform inspection via `.vcd` files allowed cycle-level monitoring of internal signals.

Correctness was evaluated by comparing the output digit predictions against the true labels of the test images. Intermediate outputs like `layer1_out`, `layer2_out` and `output_scores[]` were manually cross-checked to ensure alignment at each stage.

Figures 4.1 and 4.2 present a waveform capture of the inference process for a test image of the digit 3. The vertical yellow marker indicates the clock cycle where inference completes and the predicted digit is finalized.

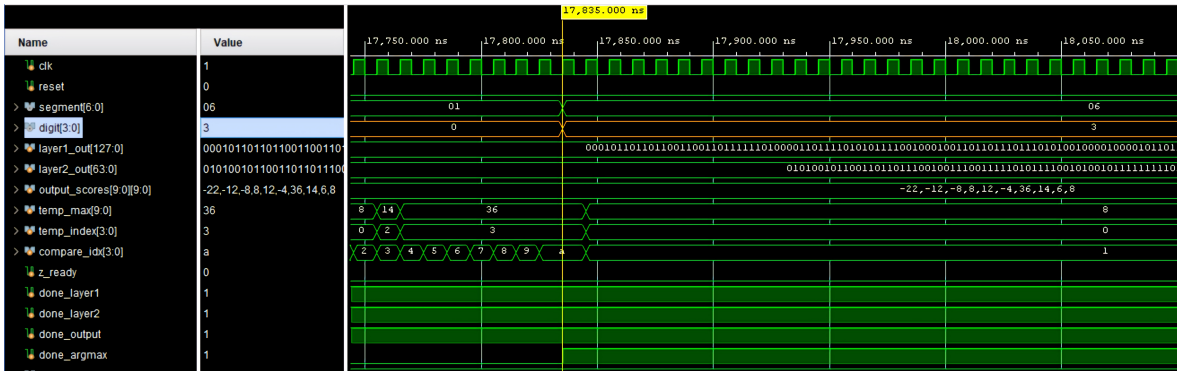

Figure 4.1. Upper section of simulation waveform showing FSM transitions, output digit, final logits (`output_scores`) and prediction logic.

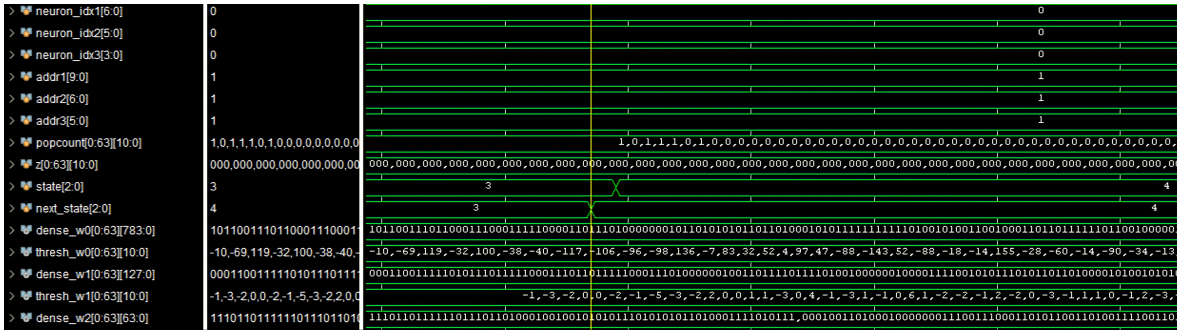

Figure 4.2. Lower section of simulation waveform showing neuron traversal counters, threshold comparisons, intermediate accumulators and ROM contents.

4.5.2. RTL Synthesis

Vivado's RTL Synthesis tool translated the Verilog source into a gate-level netlist customized for the Artix-7 XC7A100T architecture. This process was performed after verifying the functionality of all modules through behavioral simulation. The synthesis process included HDL parsing, optimization of logic expressions and mapping to specific hardware primitives such as LUTs, Flip-Flops and BRAMs.

Resource utilization results were extracted for the design configured at `PARALLEL_NEURONS = 64`, with dual-port BRAMs used for weight storage. The key goals of synthesis were to:

- Evaluating logic resource usage including Look-Up Tables (LUTs), Flip-Flops (FFs), BRAMs and I/O buffers
- Verifying the proper instantiation of memory modules, FSM logic and datapath structures
- Ensuring synthesis passes without critical warnings to confirm structural correctness

Figure 4.3 shows the synthesized RTL schematic of the top-level module. The design is flattened into its post-synthesis netlist, with the `bnn_inference` core module interfaced with I/O buffers and key internal registers. This view confirms correct synthesis and routing of major signal paths, including clock, reset and digit display.

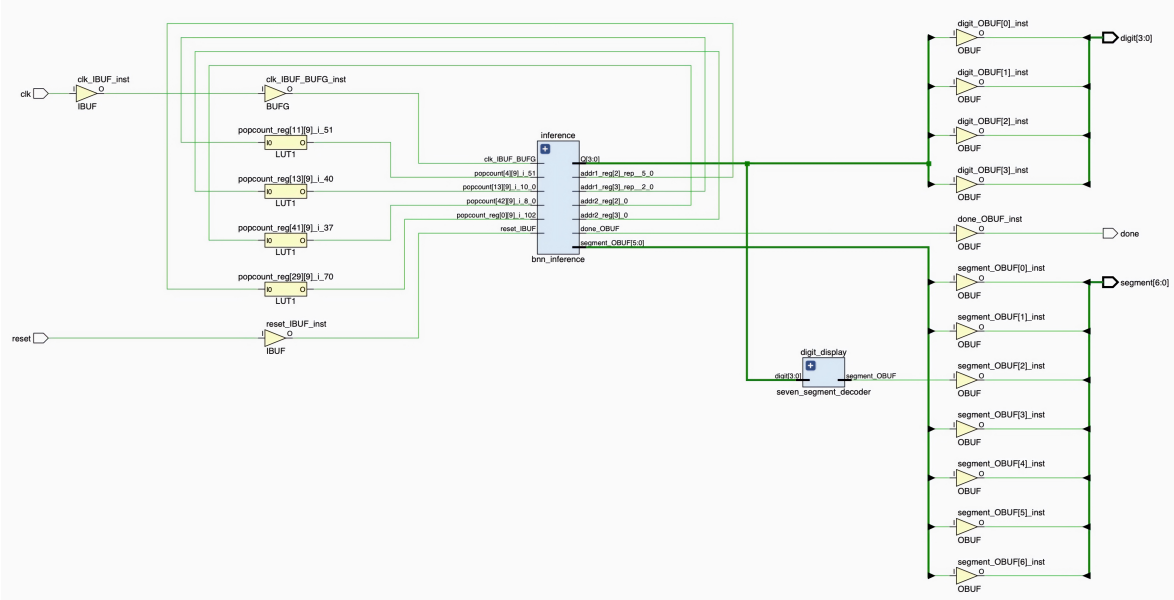

Figure 4.3. Synthesized schematic view for the top-level module at 64-parallel configuration.

To confirm correctness and visualize the design hierarchy, Figure 4.4 shows the synthesized RTL netlist as viewed after elaboration. This view confirms that all submodules were instantiated as expected.

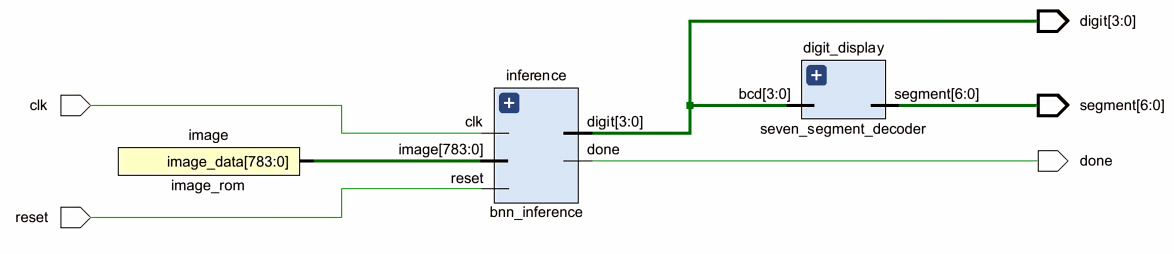

Figure 4.4. Elaborated RTL netlist showing structural hierarchy.

These synthesis results validated that the design was structurally sound and resource-efficient given the project constraints and formed the baseline for implementation and power analysis.

4.5.3. Post-Synthesis Implementation

Following synthesis, the design was passed through Vivado's physical implementation stages, which includes logic optimization, placement and routing across the FPGA fabric. This stage maps logical blocks to specific physical resources while optimizing for timing and congestion.

Timing Verification.
Vivado's implementation tools ensured that all constraints for clock timing, signal integrity and interconnect delay were satisfied. The implementation process addressed the following:

- Ensuring setup and hold timing requirements were met across all paths for the 80 MHz system clock.
- Eliminating any critical path violations by adjusting placement and routing.
- Managing BRAM usage and interconnect congestion, especially under high parallelism (`PARALLEL_NEURONS = 64`).

Post-implementation timing was verified using Vivado's static timing analyzer. The design achieved zero timing violations with all slack margins reported as positive. Figure 4.5 summarizes the setup, hold and pulse width timing results.

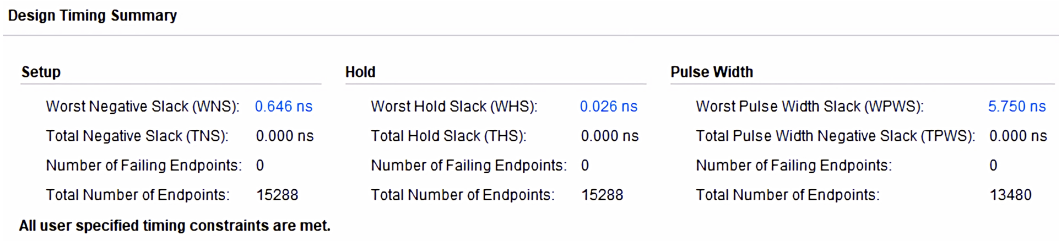

Figure 4.5. Timing summary confirming successful closure at 80 MHz.

In the setup section, the **Worst Negative Slack (WNS)** is 0.646 ns, meaning that the critical path is within acceptable limits with positive slack, satisfying setup timing constraints. For hold timing, the **Worst Hold Slack (WHS)** is 0.026 ns, indicating that no hold violations are present. The pulse width check shows a **Worst Pulse Width Slack (WPWS)** of 5.750 ns, confirming that all signals maintain sufficient high or low durations to ensure reliable operation. With zero negative slack and zero failing endpoints across all checks, the design is fully timing safe at the target frequency of 80 MHz.

Resource Utilization Summary.

Figure 4.6 presents the post-implementation **resource utilization report** for the design configured with `PARALLEL_NEURONS = 64`. The utilization summary indicates that 16,629 of the 63,400 available LUTs were used, corresponding to 26.23% of the total logic capacity. This reflects the logic needed for operations like XNOR and popcount, controlling the FSM and generating addresses for memory access in each layer.

In terms of sequential resources, the design utilizes 13,215 out of 126,800 flip-flops (10.42%), which shows that despite the design's complexity, the usage remains comfortably within device limits.

The most heavily consumed resource is Block RAM. A total of 132 out of 135 BRAM blocks are used, reaching a utilization of 97.78%. This high percentage is expected due to several large dual-port ROMs instantiated for storing weights across the three layers of the BNN. The aggressive parallelization further increases BRAM usage as more ROM blocks are instantiated to support concurrent access.

Lastly, only 14 out of 210 general purpose I/O ports (6.67%) are utilized. These are dedicated to interfacing with the clock, reset, 7-segment display output and debugging interfaces.

This breakdown shows that the design fits well within the XC7A100T's resources for LUTs and flip-flops but comes close to the BRAM limit. This was a necessary tradeoff to achieve faster processing with wide memory access.

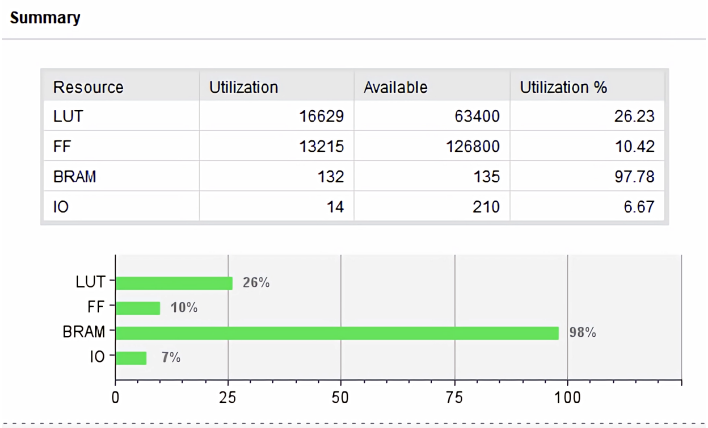

Figure 4.6. Post-implementation resource utilization for `PARALLEL_NEURONS = 64`.

4.5.4. Power Analysis

Vivado's Power Analyzer was used after implementation to estimate the power consumption of the design. This analysis is based on actual signal activity from the post-place-and-route netlist. Both static (leakage) and dynamic (switching) power consumptions were included.

As shown in Figure 4.7, the total estimated on-chip power is approximately **0.617,W**. Of this, **0.512,W** (83%) is dynamic power and **0.105,W** (17%) is static. The high dynamic power usage is mostly due to the extensive use of BRAM which alone accounts for **0.376,W** or 74% of the total dynamic power.

Other contributors to dynamic power include:

- Clocks: **0.056,W** (11%)
- Signals: **0.040,W** (8%)
- Logic (LUTs/Flip-Flops): **0.039,W** (8%)
- I/O: **0.001,W** (negligible)

The high BRAM consumption reflects the design's memory-intensive architecture. Since a large number of dual-port ROMs are used to support parallel neuron access, the overall memory switching activity is high. This is the main source of dynamic power.

The junction temperature is reported as **27.8°C**, with a thermal margin of over **57°C**, indicating that the chip operates well within safe thermal limits.

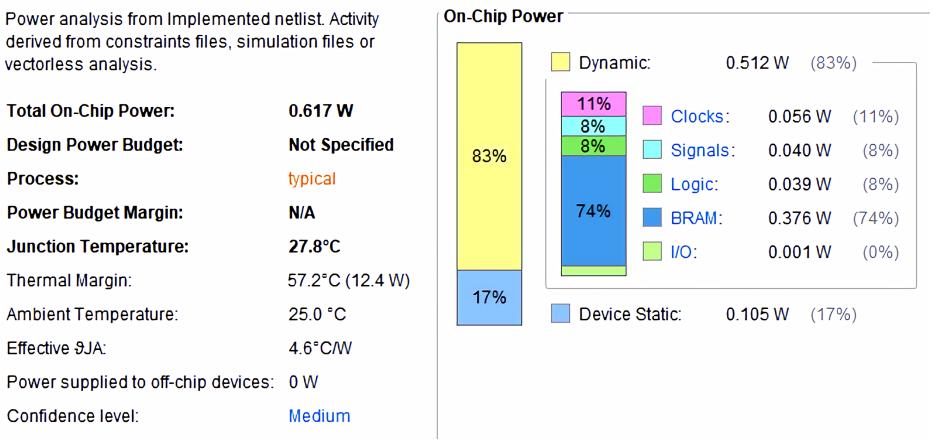

Figure 4.7. Post-implementation power consumption breakdown for 64-level parallelization.

The moderate total power consumption means that the design is suitable for deployment on mid-range FPGAs like the Artix-7 without exceeding thermal or power budgets. But the high BRAM usage and associated power cost means that scaling of the design may require a higher-end FPGA with more power and memory resources. This affects not only energy efficiency but also system cost as higher capacity devices generally come with increased price and board complexity.

4.5.5. Summary

The entire workflow from simulation to hardware mapping ensured that the Binary Neural Network implementation remained functionally correct, resource efficient and compliant with timing and power constraints. Although the design was not deployed to the physical board, the results validate its readiness for hardware deployment.

5. TEST & RESULTS

5.1. Testing Methodology

All testing was performed using Vivado 2024.1, targeting the Xilinx Artix-7 XC7A100T FPGA. Functional and timing correctness were evaluated using testbenches with `.mem` files providing binarized weights, thresholds and MNIST test images. These files were loaded into ROM modules at simulation start.

The `bnn_top_tb` testbench applied clock/reset signals and observed the `digit` and `done` outputs. Waveform viewers and simulation logs were used to monitor FSM transitions, intermediate layer outputs and final predictions.

Correctness was validated by comparing predicted digits against CPU-based inference results from the trained Python model. Tests were repeated across parallelization levels and memory configurations to enable detailed performance comparisons.

5.2. Correctness Verification

To validate the correctness of the hardware implementation, a total of 100 binarized MNIST test images were evaluated on the FPGA design, with 10 unique images for each digit from 0 to 9. These images were normalized and then converted into `.mem` files and fed into the design one by one while the predicted digit was recorded after each simulation.

The resulting predictions were compared against the expected digit labels. Out of 100 test cases 87 were classified correctly, resulting in an overall accuracy of **87%**. This is highly consistent with the trained software model's test accuracy of **87.97%**, confirming that the quantized model behavior has been accurately replicated on hardware.

Table 5.1 shows the confusion matrix summarizing the predictions. Each row represents the true label, and each column represents the predicted digit. Correct predictions are located on the diagonal and highlighted in green while incorrect classifications appear off-diagonal in red.

True\Pred	0	1	2	3	4	5	6	7	8	9
0	10	0	0	0	0	0	0	0	0	0
1	0	9	0	1	0	0	0	0	0	0
2	3	0	6	0	0	0	0	1	0	0
3	0	0	1	9	0	0	0	0	0	0
4	1	0	0	0	8	0	0	0	1	0
5	0	0	0	1	1	8	0	0	0	0
6	1	0	1	0	0	0	8	0	0	0
7	1	0	0	0	0	0	0	9	0	0
8	0	0	0	0	0	0	0	0	10	0
9	0	0	0	0	1	0	0	2	0	7

Table 5.1. Confusion matrix of FPGA predictions on 100 test images

5.3. Speed vs Resource Trade-Off Results

To evaluate the scalability of the design, thirteen hardware configurations were tested by varying the `PARALLEL_NEURONS` parameter from 1 to 128. For each configuration up to and including 64-level parallelism, two versions were synthesized: one using dual-port BRAMs for weight storage and one using LUT based ROMs. At 128-level parallelism, BRAM capacity is fully exhausted and partial fallback is no longer possible, so only the LUT based version could be synthesized. All timing, power and resource utilization results were taken from post-implementation reports in Vivado.

5.3.1. Latency Comparison

As shown in Table 5.2, while the inference latency decreases significantly with each doubling of parallelism, the speedup is not perfectly linear. For instance, at 16× parallelism, the speedup is 15.9× instead of the ideal 16×, and at 64× it drops to 61.4× instead of the ideal 64. This difference becomes more noticeable at 128×, where the speedup reaches only 111.1× compared to the expected 128×.

The BRAM and LUT based versions show nearly identical latency values across all parallelization levels up to 64. The observed difference between the two memory styles is only

10 ns, indicating that from a pure timing perspective both styles perform similarly.

However, these results are derived from behavioral simulation, which does not fully account for hardware specific delays such as routing congestion, interconnect loading or clock skew. In actual implementation, the LUT based versions may experience greater critical path delays due to increased logic utilization and denser placement, especially at high parallelism levels.

The nonlinearity in speedup results from several hardware factors. As the parallelism increases, more computational units are instantiated which puts additional stress on routing resources. This leads to longer interconnect delays and less optimal placement, reducing timing efficiency. In addition, memory contention increases because more parallel units try to read from memory at the same time, creating access bottlenecks even with dual-port BRAM or LUT based ROMs. Control overhead also increases with more FSM logic and wider popcount logic, since more bits need to be counted simultaneously in each clock cycle, adding delay and limiting speedup. Together, these effects cause diminishing returns as parallelism reaches higher levels.

Parallelization	**Inference Latency (ns)**	**Speedup**	**Memory Style**
1	1,096,045	1×	BRAM
1	1,096,035	1×	LUT
4	274,465	4.0×	BRAM
4	274,455	4.0×	LUT
8	137,645	7.96×	BRAM
8	137,635	7.96×	LUT
16	68,905	15.9×	BRAM
16	68,895	15.9×	LUT
32	34,865	31.43×	BRAM
32	34,855	31.45×	LUT
64	17,845	61.42×	BRAM
64	17,835	61.45×	LUT
128	9,865	111.1×	LUT

Table 5.2. Measured inference latency as a function of parallelization and memory style

5.3.2. Resource Utilization

Table 5.3 summarizes post-implementation resource usage across increasing levels of parallelism. Each value is accompanied by its percentage relative to the total available resources on the XC7A100T device. For a clearer visualization, Figures 5.1, 5.2 and 5.3 illustrate how LUT, Flip-Flop and BRAM usage scale with parallelization under both BRAM-based and LUT-only memory configurations.

BRAM utilization increases rapidly up to 16× parallelism, reaching 132 blocks, which is 97.78% of the XC7A100T's BRAM capacity. Beyond 64× parallelism, the BRAM based design cannot synthesize at higher parallelism levels as it no longer supports partial fallback to LUTs. Because of this the 128× configuration was tested using a fully LUT based implementation. Attempts to synthesize even higher parallel configurations with LUT only memory also failed, marking 128× as the practical upper limit for parallelization in this design.

Parallelization	LUTs	FFs	BRAMs	Memory Style
1	786 (1.24%)	455 (0.36%)	13 (9.63%)	BRAM
1	2,483 (3.92%)	477 (0.38%)	0	LUT
4	1,661 (2.62%)	497 (0.39%)	52 (38.52%)	BRAM
4	6,649 (10.49%)	669 (0.53%)	0	LUT
8	3,093 (4.88%)	605 (0.48%)	104 (77.04%)	BRAM
8	12,953 (20.43%)	774 (0.61%)	0	LUT
16	10,364 (16.35%))	5,716 (4.51%)	132 (97.78%)	BRAM
16	13,786 (21.74%))	992 (0.78%)	0	LUT
32	14,401 (22.71%)	15,880 (12.53%)	132 (97.78%)	BRAM
32	11,541 (18.20%)	1217 (0.96%)	0	LUT
64	16,494 (26.02%)	10,670 (8.41%)	132 (97.78%)	BRAM
64	15,272 (24.09%)	1850 (1.46%)	0	LUT
128	18,626 (29.38%)	3,144 (2.48%)	0	LUT

Table 5.3. Post Implementation logic and memory utilization.

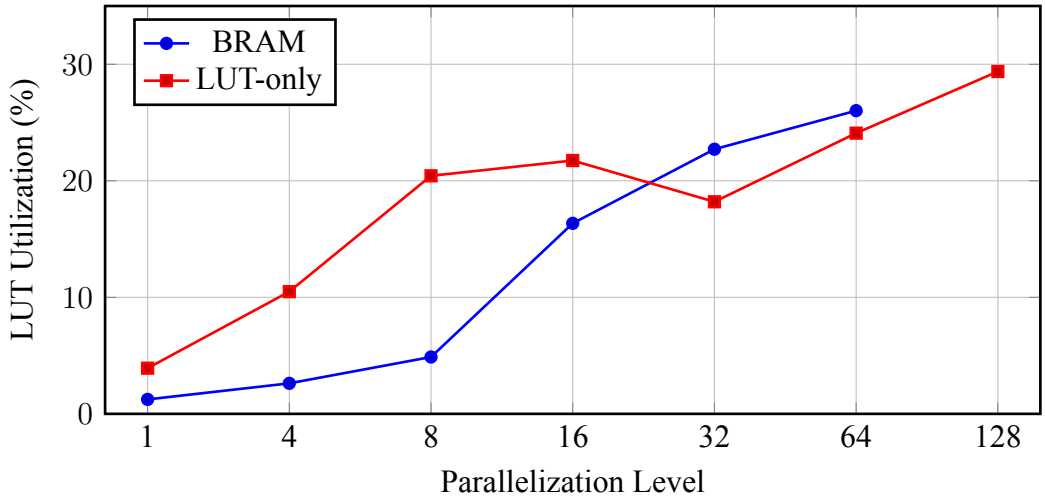

Figure 5.1. LUT utilization across parallelization levels.

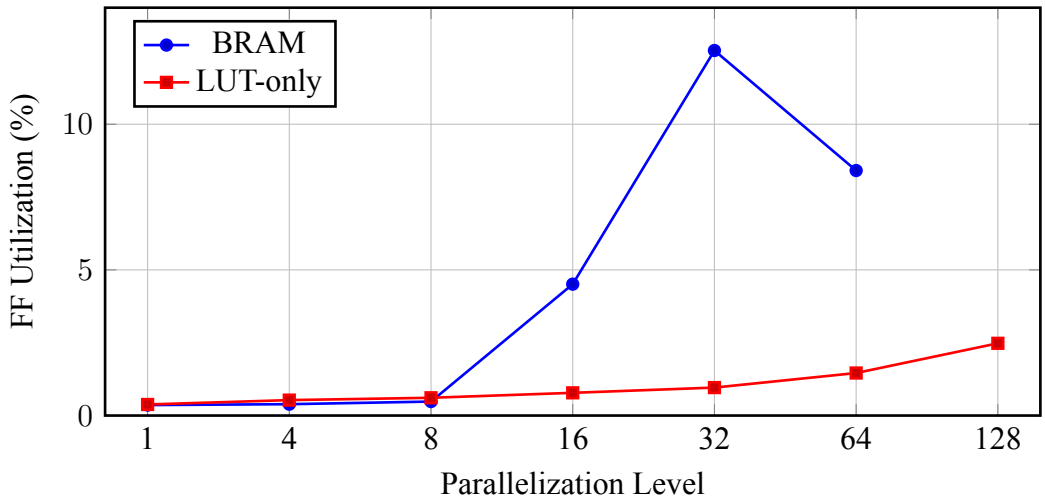

Figure 5.2. Flip-Flop utilization across parallelization levels.

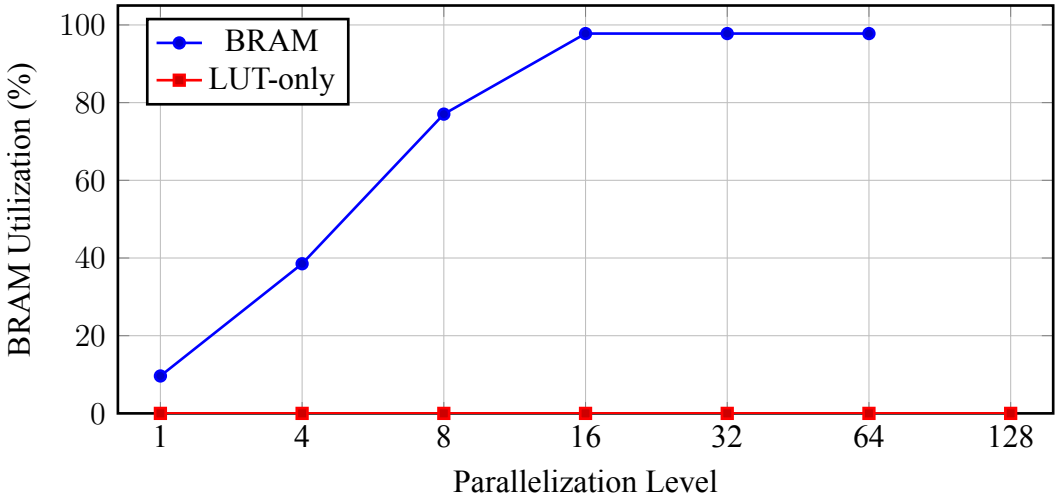

Figure 5.3. BRAM utilization across parallelization levels.

LUT usage in BRAM based configurations grows steadily with parallelism, starting from only 1.24% at 1× and increasing up to 26.02% at 64×. In contrast, LUT based configurations begin with significantly higher LUT usage, 3.92% even at 1×, due to logic overhead from synthesizing ROMs out of LUTs. Interestingly, while the LUT usage continues to grow in BRAM based designs with each level of parallelism, the LUT only designs show a different behaviour where LUT usage rises until 16× parallelism (21.74%) but then drops at 32× to 18.20%, before increasing again. This drop likely reflects optimization effects in synthesis such as logic folding or reduced duplication in control logic when large ROMs dominate the design.

At 32× and 64× levels, the BRAM based designs actually use more LUTs than their LUT only equivalents. This is likely due to the added complexity of managing many parallel dual-port BRAM instances. Each BRAM block requires dedicated address generation logic, read-enable control and synchronization signals, all of which are implemented using LUTs. As the number of parallel accesses increases, this overhead also increases, leading to a sharp increase in control and routing logic. In comparison, the LUT only versions benefit from a more compact memory organization. Since the weights are synthesized directly into the logic fabric as distributed ROMs, address decoding and access logic can be flattened and optimized during synthesis. This eliminates the need for deep FSM coordination and shared BRAM access control, allowing the tools to simplify the datapath.

Flip-flop utilization follows a similarly distinct pattern. In BRAM based configurations, FF usage increases significantly at higher parallelism levels, peaking at 12.53% for 32×, due to deeper sequential control structures and wider data movement. Interestingly, FF usage drops to 8.41% at 64× despite higher parallelism. This reduction may be due to architectural optimizations in the control FSMs or shared datapaths that reduce duplication, as well as potential logic folding applied by the synthesis tool to meet timing constraints under increased routing pressure. However, in LUT based designs FF utilization remains consistently low, reaching only 1.46% at 64× and 2.48% at 128×. This suggests that the logic structure of LUT based memory may require fewer sequential elements for control and buffering or that certain datapath and control logic is implemented as combinational rather than registered logic due to synthesis optimizations.

Overall, the table and the figures demonstrates that while BRAM becomes a limiting factor early in the scaling process, logic resources (LUTs and FFs) remain comfortably within the Artix-7 budget throughout.

5.3.3. Timing Slack

Table 5.4 shows the worst negative slack (WNS) and worst hold slack (WHS) reported by Vivado after place-and-route for each configuration. Figures 5.4 and 5.5 shows the variation of these metrics across different parallelization levels for BRAM and LUT-only memory styles. WNS reflects how close the design is to violating setup timing constraints. Lower values mean less time left before errors may happen. WHS shows if signals are arriving too early which can cause data to change when it shouldn't during a clock cycle.

Parallelization	**Worst Negative Slack (ns)**	**Worst Hold Slack (ns)**	**Memory Style**
1	1.144	0.169	BRAM
1	3.564	0.115	LUT
4	1.525	0.132	BRAM
4	1.975	0.039	LUT
8	1.043	0.062	BRAM
8	1.708	0.187	LUT
16	0.37	0.033	BRAM
16	1.109	0.05	LUT
32	0.68	0.075	BRAM
32	1.95	0.129	LUT
64	0.939	0.081	BRAM
64	0.519	0.04	LUT
128	1.163	0.025	LUT

Table 5.4. Post Implementation timing slack values across configuration at 80 MHz

Across both memory styles, WNS generally drops with higher parallelism due to added logic and routing delays. In BRAM designs, WNS drops to 0.37 ns at 16× but slightly improves at 32× and 64× likely from better placement or reduced congestion in those builds. For LUT based designs WNS starts high at 1× (3.564 ns), stays above 1 ns up to 32×, goes below 1 at 64× due to increased complexity, then improves at 128× which may be because the fully flattened ROM layout reduces long timing paths.

WHS values remain small across all configurations, ranging from 0.025 ns to 0.187 ns but they always stay positive.

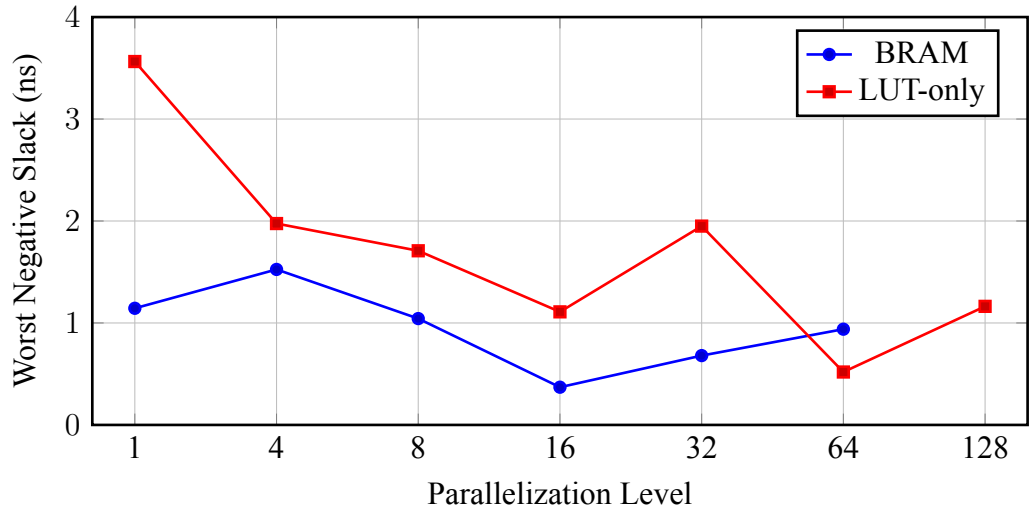

Figure 5.4. Worst Negative Slack across parallelization levels.

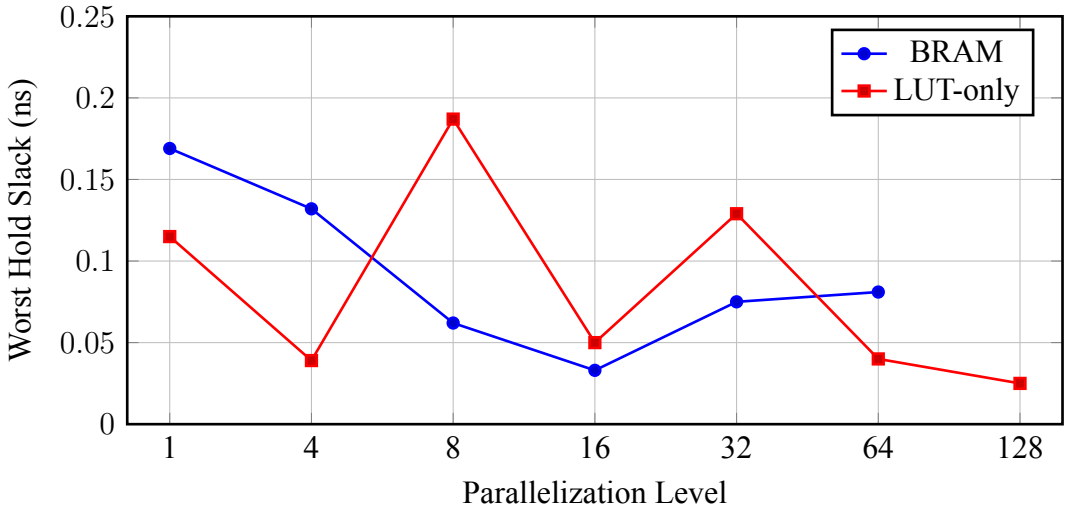

Figure 5.5. Worst Hold Slack across parallelization levels.

Overall all configurations meet the 80 MHz timing target but designs beyond 64× offer very little timing margin. LUT-only designs show better WNS in mid-range configurations like 16× and 32× while BRAM based designs generally have more consistent hold slack performance.

5.3.4. Power and Thermal Estimates

Table 5.5 shows that power and temperature increase with parallelization but the rate and pattern of increase are different between memory types. BRAM designs see a steeper power rise while LUT versions grow more gradually and stay efficient. Figures 5.6 and 5.7 helps visualize this by showing how BRAM and LUT power and dynamic power usage diverge at higher parallel levels.

Parallelization	Total Power (W)	Junction Temperature (°C)	Dynamic/Static Power (%)	Memory Style
1	0.103	25.5	5/95	BRAM
1	0.106	25.5	9/91	LUT
4	0.111	25.5	10/90	BRAM
4	0.119	25.5	19/81	LUT
8	0.127	25.6	20/80	BRAM
8	0.115	25.5	16/84	LUT
16	0.183	25.8	43/57	BRAM
16	0.142	25.6	32/68	LUT
32	0.633	27.9	83/17	BRAM
32	0.147	25.7	34/66	LUT
64	0.617	27.8	83/17	BRAM
64	0.156	25.7	37/63	LUT
128	0.179	25.8	46/54	LUT

Table 5.5. Post Implementation power and temperature estimates across configurations.

Junction temperature represents the internal temperature of the internal chip, influenced by dynamic power consumption and the efficiency of thermal dissipation. It increases as the design consumes more power and utilizes more logic resources. For BRAM designs, the temperature rises from 25.5 °C at 1× to a peak of 27.9 °C at 32×, reflected by the jump in power from 0.103 W to 0.633 W. In contrast, LUT based designs maintain nearly constant temperatures around 25.5–25.8 °C across all levels, reflecting their lower total power consumption.

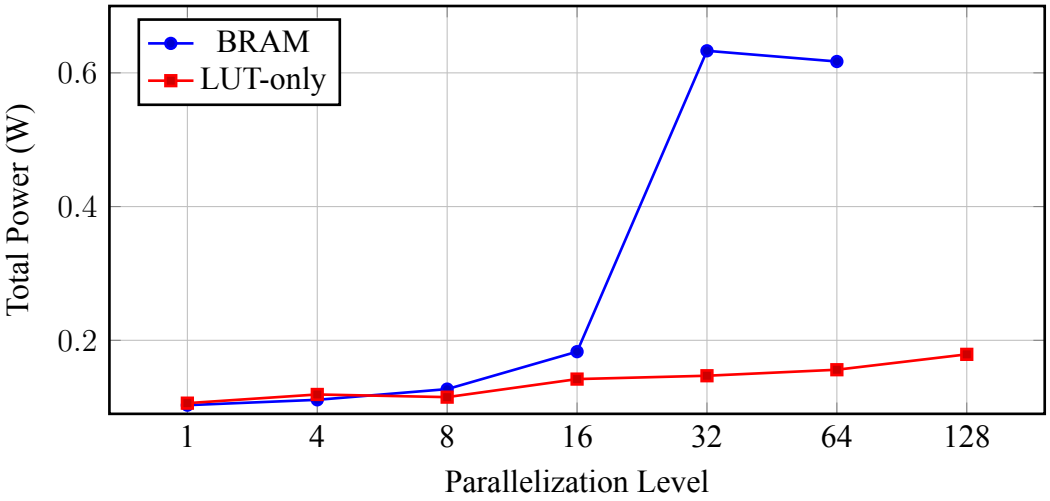

Figure 5.6. Total power consumption across parallelization levels.

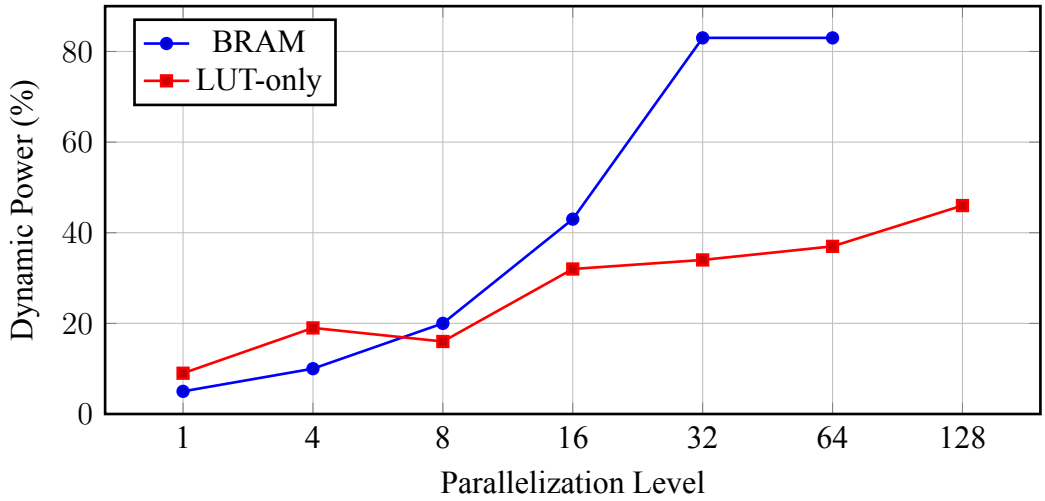

Figure 5.7. Dynamic power consumption percentage across parallelization levels.

The dynamic vs static power ratio reveals how activity changes with parallelism. At low levels most power is static with dynamic power around 5–20%. As more units become active, dynamic power rises due to increased switching especially in BRAM designs where it exceeds 80% at 16–64×. This suggests that avoiding BRAM reduces high activity memory access and keeps dynamic power lower.

Overall, BRAM based designs are more power hungry at higher configurations due to intensive memory access and wide datapath control while LUT only designs trade off slightly higher logic usage for better energy efficiency. This also reflects in thermal behavior where the LUT only designs maintains lower and more stable junction temperatures. Thus, while BRAM configurations deliver higher throughput, they come with higher power and thermal costs.

5.3.5. Summary and Trade-Off Justification

Table summaries and detailed analysis in the previous subsections show the key trade-offs between performance, resource usage and efficiency. Both BRAM and LUT based memory styles differ significantly at higher configurations.

The 64× BRAM based implementation is selected as the final configuration because it provides the best balance between performance, hardware feasibility and realistic deployment conditions. It delivers an inference latency of 17.8 ns, offering a 61.4× speedup over the baseline while fully utilizing the available 132 BRAM blocks (97.78%). Logic utilization remains moderate (26.02% LUTs, 8.41% FFs) and the design meets timing at 80 MHz with 0.939 ns of WNS. Although its power consumption is relatively high at 0.598 W, this is compensated by its throughput and resource realism.

While the 128× LUT only design performs correctly in simulation and consumes significantly less power (0.179 W), its practicality is limited. It achieves only a 1.8× speedup over the 64× design while requiring the highest LUT usage (29.38%) and offers no further scalability as synthesis fails beyond this point. Moreover, relying only on LUT based ROMs to store model weights is not representative of real FPGA deployment scenarios. In practice, model weights are stored in larger, external or on-chip SRAM blocks or mapped to dedicated BRAM when available. Using LUTs to synthesize memory for large models is inefficient and not scalable.

Additionally, BRAM based memory access more accurately reflects hardware implementations where high throughput and low latency access to structured memory is essential. The LUT only implementation would become impractical for larger networks where weights can span millions of parameters, even though it isfunctional in this controlled design. Therefore, the BRAM configuration is a better approximates for a realistic memory hierarchy and it is a more meaningful platform for analyzing power, timing and utilization behaviors under real-world conditions.

In summary, the 64× BRAM based configuration achieves the optimal trade-off: it maximizes parallelism without exceeding physical limits, adheres to practical design rules and remains deployable within the constraints of the target Artix-7 FPGA. It serves as the most effective and scalable implementation for real hardware integration.

5.4. BNN vs CNN Comparison Results

Two neural network models were trained and evaluated for the task of handwritten digit classification. Both were implemented in TensorFlow with fixed random seeds and tested under identical conditions. One uses a binarized fully connected architecture (BNN), while the other follows a standard convolutional structure (CNN). While both models perform the same task and operate on the same input format, they differ significantly in architecture, computational cost and hardware suitability.

5.4.1. Architectural Complexity

The CNN follows a standard convolutional structure with two convolutional layers using 3×3 filters (with 32 and 64 channels), each followed by 2×2 max pooling. The resulting feature maps are flattened and passed through a 128-unit dense layer with ReLU activation and dropout before producing a 10-class softmax output. This architecture is designed to extract multi-scale spatial features and supports high accuracy but it also introduces significant complexity that requires floating-point multiplications, non-linear activations and large intermediate buffers.

In contrast, the BNN uses a fully connected architecture with three binarized layers. The 28×28 input is flattened and passed through QuantDense layers with 128 and 64 binary neurons followed by a final binarized dense layer producing 10 logits. All layers use batch normalization without scaling and doesnt use any bias terms. Inference is performed using XNOR-popcount operations followed by thresholding.

While the CNN prioritizes representational power and accuracy, the BNN is designed for simplicity, speed and efficient deployment.

5.4.2. Accuracy

The CNN model reached a test accuracy of **99.31%** while the BNN achieved **87.97%**. This result reflects the advantages of convolutional layers when it comes to picking up spatial patterns in images. By learning local features and building up more complex representations, the CNN is better at telling apart digits that look similar. On the other hand, the BNN flattens the image and processes it using simple binary transformations, which limits how much structure it can capture. The 11.34% drop in accuracy is a direct result of this trade-off between simplicity and accuracy.

5.4.3. Inference Latency

Mean inference latency over 100 CPU runs was **0.176 ms** for the BNN and **0.213 ms** for the CNN. On average, the BNN completed inference around 17% faster than the CNN. As summarized in Table 5.6, BNN latencies ranged from 0.149 ms to 0.276 ms while CNN latencies ranged from 0.191 ms to 0.293 ms.

Model	Mean (ms)	Min (ms)	Max (ms)	Standard Deviation (ms)
BNN	0.176	0.149	0.276	0.022
CNN	0.213	0.191	0.293	0.016

Table 5.6. CPU inference latency statistics across 100 runs

Figure 5.8 shows the full distribution of inference times across all 100 runs. The BNN not only shows lower latency overall but also reaches its lowest latencies earlier in the run. In contrast, the CNN shows slightly more fluctuation.

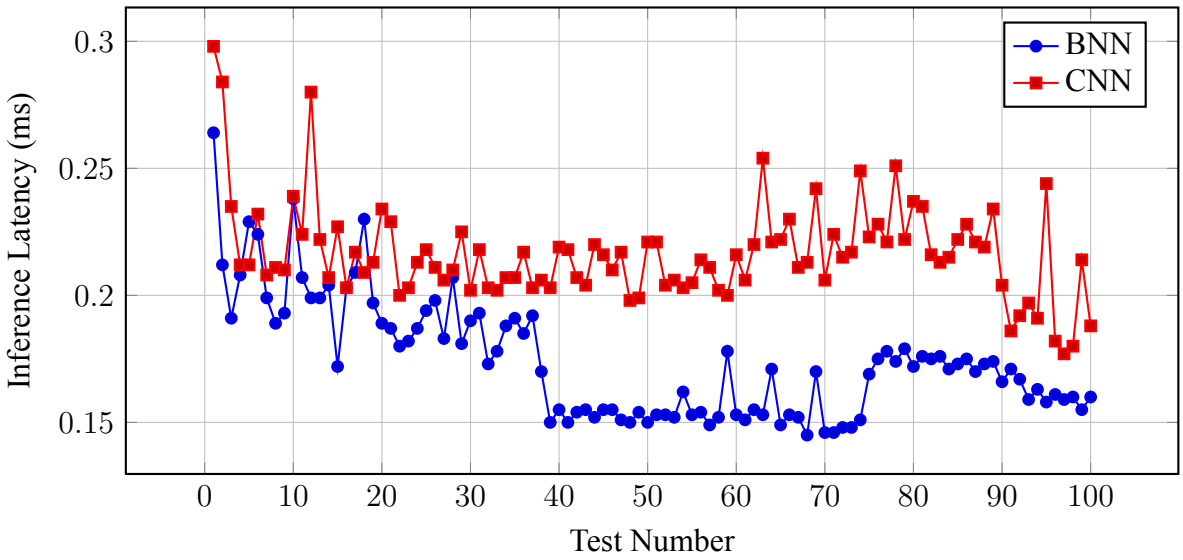

Figure 5.8. CPU inference latency distribution for BNN and CNN models across 100 runs.

5.4.4. Model Size

The saved BNN model occupies **1.4 MB** of storage while the CNN model occupies **2.7 MB** of storage. This roughly 2× difference reflects the CNN's use of 32-bit floating-point weights and its larger number of parameters introduced by convolutional filters and dense layers. While the BNN uses only 1-bit weights and doesnt include bias terms, it also has fewer total parameters due to its simpler, fully connected architecture with smaller layer sizes. This means the memory savings are significant but do not reach the theoretical 32× reduction since the BNN starts with a simpler structure overall.

5.4.5. Training Time

Training the BNN for 15 epochs took **15 seconds** while the CNN required **71 seconds** for just 10 epochs. Even after adjusting for the difference in epoch count, the CNN remains considerably slower to train. This is expected because convolutional operations are more computationally expensive than simple binary matrix multiplications. Additionally, the CNN's use of dropout, activation functions and floating-point arithmetic increases training time.

The speed of BNN training allows for more rapid iteration during development, particularly when tuning hyperparameters or re-training on new datasets. On the other hand, the CNN requires fewer epochs to converge due to its feature extraction capacity.

5.4.6. Suitability for FPGA

The BNN is a natural fit for FPGAs. Its bitwise operations map efficiently to LUTs and the absence of multipliers means DSP blocks are unused. With a uniform layer structure and no branching, the design is easy to parallelize and control using simple FSM logic which makes resource usage and latency predictable.

In contrast, the CNN relies on floating-point multiply-accumulate operations and large intermediate buffers which require DSP slices and more complex memory management. Its dynamic control flow also makes timing closure harder. While the CNN achieves higher accuracy, the BNN offers a more practical balance of performance and efficiency for FPGA deployment.

5.5. Platform Comparison Results

This section compares the performance and efficiency of the final FPGA implementation with alternative platforms. All comparisons are made using the same Binary Neural Network (BNN) architecture, specifically the configuration using 64-level parallelization with BRAM on the Nexys A7-100T FPGA.

5.5.1. Comparison with ASIC

Since a direct ASIC implementation of this BNN model was not feasible within the scope of this project, a rough estimation based comparison is performed using data from YodaNN, which is a low-power CNN accelerator based on binary weights [12]. While YodaNN is optimized for convolutional layers, it is the most relevant ASIC reference for binary weight architectures. Table 5.7 summarizes the key specifications and efficiency metrics of the YodaNN ASIC accelerator which serves as the baseline for this comparison.

Attribute	**Value**
Process Technology	UMC 65 nm LP CMOS
Core Area	$\sim$1.9 mm^2 (1.33 MGE)
Max Clock Frequency	480 MHz at 1.2 V
Low Power Clock Frequency	27.5 MHz at 0.6 V
Peak Throughput	1.5 TOp/s @ 1.2 V
Energy Efficiency	61.2 TOp/s/W at 0.6 V
Core Power	895 μW at 0.6 V
Retail Price Estimate	\$2 - \$6 (in volume)
Architecture	Binary CNN (1×1 - 7×7 kernels)

Table 5.7. ASIC (YodaNN) Properties and Metrics

- **Latency:** The 64-parallel FPGA implementation achieves an inference latency of **17,845 ns** (or **0.0178 ms**) per image at 80 MHz. For comparison, the YodaNN ASIC reports a latency of **7.5 ms** per image for a similar 3-layer BinaryConnect architecture on the CIFAR-10 dataset [12, Table III]. While the CIFAR-10 dataset consists of 16x16 RGB color images and involves convolutional processing stages, this is the closest architectural match available from published ASIC results. Despite its optimizations for energy efficiency, YodaNN includes multiple convolutional layers and a more complex input space which contributing to its significantly higher per-image latency. Both implementations use binarized weights and target 10-class classification tasks, which

makes the comparison useful for understanding the FPGA's strength in low-latency inference, especially when real-time responsiveness is prioritized over throughput or energy efficiency.

- **Power and Energy:** The FPGA implementation reports a total power consumption of approximately **0.617 W** at 80 MHz based on Vivado's post-implementation power analysis. This estimate includes both static leakage and dynamic switching components and reflects total device usage assuming continuous operation.

  In contrast, the YodaNN ASIC consumes only about **0.00034 W** during inference, calculated from its throughput and energy efficiency [12, Table III] as:

$$P_{\text{real}} = \frac{\Theta_{\text{real}}}{EnEff} = \frac{20.1\,\text{GOp/s}}{59.2\,\text{TOp/s/W}} \approx 0.00034\,\text{W}$$

  This indicates that YodaNN uses over **1800**$\times$ less power during active computation. But its important to note that this figure reflects dynamic power consumption during inference only while the FPGA figure includes total board-level power estimated across all logic elements, memory blocks and static leakages.

  For energy per inference the FPGA consumes approximately **11.0** μ**J**, calculated as:

$$E_{\text{FPGA}} = 0.617\,\text{W} \times 17.845 \times 10^{-6}\,\text{s} \approx 11.0\,\mu\text{J}$$

  while YodaNN reports just **2.6** μ**J** for a similar task.

- **Cost:** The retail price of the XC7A100T development board is **$150**, while YodaNN ASICs are estimated to cost between **$2–$6** per unit in volume production. Note that this price is not provided in the YodaNN paper and it is estimated based on publicly available manufacturing data for 65 nm chips, taking into account typical wafer costs, die yield rates, packaging and testing costs and standard pricing markups used in the industry. While ASICs are significantly cheaper at scale for consumer purchases, it's important to mention that the research and development cost for ASICs is significantly higher than for FPGAs due to fabrication, tooling and non-recurring engineering (NRE) expenses. On the other hand FPGA solutions involve higher per-unit costs but much lower upfront development costs and offer faster iteration during prototyping.

- **Flexibility:** FPGA offers full reconfigurability which allows developers to change the neural network architecture entirely such as changing the number of layers, neuron counts or even the dataflow model without hardware changes. In contrast, ASICs are generally fixed post-fabrication. YodaNN for instance provides limited flexibility by

supporting multiple kernel sizes (1×1 to 7×7), per-channel scaling and biasing and run-time channel tiling [12, pg. 2]. However the overall datapath and architecture remain static. This trade-off is expected as ASICs prioritize efficiency over programmability while FPGAs remain adaptable.

In summary, YodaNN demonstrates clear advantages in power consumption, energy efficiency and per-unit cost when deployed at scale. With energy per inference as low as **2.6 μJ** and a dynamic power consumption of only **0.34 mW**, the ASIC is highly optimized for low-power applications and high-throughput inference.

On the other hand, the FPGA implementation achieves much faster inference (**0.0178 ms** per image) while being competitive in energy usage (**11.0 μJ**) despite being a general-purpose platform. More importantly, the FPGA offers full reconfigurability that enables rapid design changes and experimentation without requiring new hardware. This flexibility makes FPGAs ideal for research, prototyping and low-volume deployments where design flexibility is more critical than total efficiency.

Overall, while ASICs are the optimal choice for finalized, mass-produced applications with strict power and cost constraints, FPGAs are the better choice when flexibility, development speed and real-time responsiveness are the priorities.

5.5.2. Comparison with CPU and GPU

Unlike the ASIC comparison, CPU and GPU benchmarks were run directly on the trained BNN model using Google Colab's cloud-based hardware. The same BNN was used for both. To ensure fair timing, warm-up runs were performed and overhead was minimized. Batch sizes ranged from 1 to 10,000 to evaluate scalability, as shown in Table 5.10 and visualized in Figure 5.9.

Table 5.8 shows the specifications of the CPU used in the Colab environment. While the CPU model is virtualized, it is an Intel Xeon class processor running at 2.20 GHz with two logical cores. Exact thermal and power related specifications are unavailable because of the nature of cloud hosted resources.

The GPU used for testing was the NVIDIA Tesla T4. It is passively cooled and operates under a 70 W TDP, making it ideal for datacenter inference. Table 5.9 summarizes the key GPU properties.

Attribute	Value
Model	Intel Xeon CPU @ 2.20GHz
Cores	2 (1 core per socket)
L3 Cache	56.3 MB
Retail Price Estimate	$60-100

Table 5.8. CPU Runtime Hardware

Attribute	Value
Model	NVIDIA Tesla T4 (TU104-895)
Cores	CUDA: 2560, Tensor: 320
Memory	16 GB GDDR6
Max GPU Clock	1590 MHz
Power	70 W
Driver / CUDA Version	550.54 / 12.4
Retail Price Estimate	$400-900

Table 5.9. GPU Runtime Hardware

Batch Size	Device	Mean (ms)	Per Image (ms)	Std Dev (ms)
1	CPU	1.604866	1.604866	0.289
1	GPU	0.815132	0.815132	0.059
10	CPU	1.009551	0.100955	0.139
10	GPU	0.869315	0.086932	0.092
100	CPU	1.745848	0.017458	0.151
100	GPU	1.218901	0.012189	0.134
1000	CPU	6.926128	0.006926	1.134
1000	GPU	0.862603	0.000863	0.080
10000	CPU	63.021132	0.006302	15.473
10000	GPU	1.575326	0.000158	1.078

Table 5.10. Inference Benchmark Results Across Batch Sizes

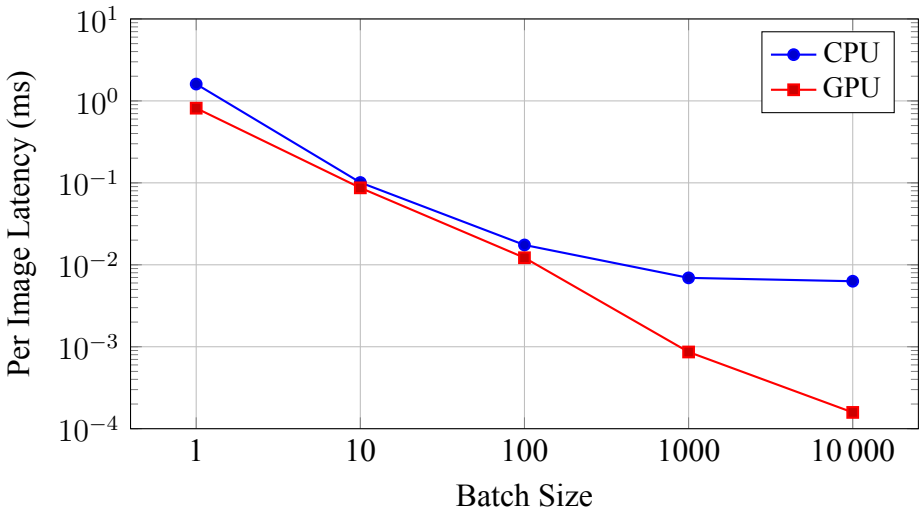

Figure 5.9. Per Image Inference Latency vs Batch Size (Log Scale)

- **Latency:** GPU inference consistently outperforms CPU at higher batch sizes, especially when matrix operations are parallelizable. For batch size 10,000, GPU achieves a per-image latency of **158 ns**, compared to **6302 ns** on CPU. The crossover point where GPU becomes significantly more efficient begins at batches ≥ 100.
- **Scalability:** While the CPU shows good scaling up to batch size 1000, its performance improvements diminish beyond that point due to limited parallelism and memory bandwidth. In contrast the GPU benefits from Tensor Core acceleration and memory-level parallelism, showing significant gains at higher batch sizes. This makes GPUs ideal for large batch, throughput optimized inference operations. On the other hand, CPUs are more effective for smaller workloads or latency-sensitive single image tasks.
- **Deployment Trade-offs:** Despite GPU's high throughput, its cost and energy footprint ($\sim$70W for T4) makes it less desirable for embedded systems or edge applications. CPUs offer decent latency and better availability in low-cost deployments. However both CPU and GPU lack the deterministic timing and low-latency guarantees of FPGA based inference.
- **FPGA Comparison:** The FPGA implementation achieves a latency of **17.845 ns** per image (64-parallel, 80 MHz). While this is slower than GPU at very large batches, it significantly outperforms CPU for real-time, low-batch scenarios. Additionally, FPGA consumes less power ($\sim$0.617 W) than GPUs and CPUs and offers predictable performance with strict resource control.

In summary, GPUs are best suited for handling large workloads with big batch sizes,

which makes them ideal for training large neural networks and running offline model evaluations in data centers. CPUs work well for general tasks or when real-time speed is not a priority. FPGAs offer a strong middle ground, they provide fast and efficient inference at much lower power levels with consistent timing.

5.5.3. FPGA as the Most Practical Platform for BNN Deployment

Based on both the experimental results and platform comparisons, FPGA stands out as the most practical option for running this BNN model. It offers a combination of fast inference, low power use and reconfigurability without the high cost or inflexibility of ASICs or the power-hungry nature of GPUs.

- **Fast Inference Time:** The final FPGA design achieved a per-image latency of **17,845 ns**. This is much faster than the CPU (**1.604 ms** at batch size 1) and only slower than the GPU when using very large batches.
- **Reliable Timing:** The time it takes to process each image on the FPGA doesn't change, no matter the workload or conditions. This makes it ideal for real-time systems where consistent response time is important which is something CPUs and GPUs can't always guarantee.
- **Lower Power Use:** The design runs at about **0.617 W**, far less than the GPU's **70 W** power rating (Table 5.9) and with much less cooling and energy cost. This makes it a great option for edge devices and power-limited environments.
- **Fits on a Mid-Range Board:** All logic fit comfortably on the Artix-7 XC7A100T, using only **26%** of the LUTs and nearly all available BRAM.
- **Easy to Change or Extend:** Unlike ASICs, the FPGA design can be updated at any time. If the network structure changes or new features are added, no new chip is needed.
- **Cost-Effective Development:** The development board costs around $150, and once purchased, no extra fees are required for testing or running the design. This makes it more affordable than GPUs (typically $400–$900) and still competitive when compared to CPUs ($60–$100) or ASICs which although cheap per unit ($2–$6), require very high upfront development costs.

In short, FPGA offered the right balance for this project with good speed, low power, easy reconfigurability and an affordable setup cost. It was especially well suited for this small and efficient neural network and will remain flexible for future updates or similar deployments.

6. CONCLUSION & FUTURE WORK

6.1. Key Findings

6.2. Challenges Encountered

Bibliography

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[2] Y. Izui and A. Pentland, "Analysis of neural networks with redundancy," *Neural Computation*, vol. 2, no. 2, pp. 226–238, 1990. DOI: 10.1162/neco.1990.2.2.226.

[3] H. Qin, R. Gong, X. Liu, X. Bai, J. Song, and N. Sebe, "Binary neural networks: A survey," *Pattern Recognition*, vol. 105, p. 107 281, Sep. 2020, ISSN: 0031-3203. DOI: 10.1016/j.patcog.2020.107281. [Online]. Available: http://dx.doi.org/10.1016/j.patcog.2020.107281.

[4] Y. Bengio, N. Léonard, and A. Courville, *Estimating or propagating gradients through stochastic neurons for conditional computation*, 2013. arXiv: 1308.3432 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1308.3432.

[5] C. Ünsalan and B. Tar, *Digital system design with FPGA : implementation using Verilog and VHDL*. McGraw-Hill Education, 2017. [Online]. Available: https://cir.nii.ac.jp/crid/1130285376382978560.

[6] J. Serrano, "Introduction to FPGA design," 2008. DOI: 10.5170/CERN-2008-003.231. [Online]. Available: https://cds.cern.ch/record/1100537.

[7] Y. Cheng, F. X. Yu, R. S. Feris, S. Kumar, A. Choudhary, and S.-F. Chang, *An exploration of parameter redundancy in deep networks with circulant projections*, 2015. arXiv: 1502.03436 [cs.CV]. [Online]. Available: https://arxiv.org/abs/1502.03436.

[8] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, *Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1*, 2016. arXiv: 1602.02830 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1602.02830.

[9] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, *Xnor-net: Imagenet classification using binary convolutional neural networks*, 2016. arXiv: 1603.05279 [cs.CV]. [Online]. Available: https://arxiv.org/abs/1603.05279.

[10] Y. Umuroglu, N. J. Fraser, G. Gambardella, *et al.*, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17, ACM, Feb. 2017, pp. 65–74. DOI: 10.1145/3020078.3021744. [Online]. Available: http://dx.doi.org/10.1145/3020078.3021744.

[11] R. Zhao, W. Song, W. Zhang, *et al.*, "Accelerating binarized convolutional neural networks with software-programmable fpgas," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17, Monterey, California, USA: Association for Computing Machinery, 2017, pp. 15–24, ISBN: 9781450343541. DOI: 10.1145/3020078.3021741. [Online]. Available: https://doi.org/10.1145/3020078.3021741.

[12] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "Yodann: An ultra-low power convolutional neural network accelerator based on binary weights," in *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2016, pp. 236–241. DOI: 10.1109/ISVLSI.2016.111.