



# NATIONAL INSTITUTE OF TECHNOLOGY CALICUT

DEPARTMENT OF ELECTRONICS AND COMMUNICATION  
ENGINEERING

## Verilog-Based Hardware Accelerator for Lightweight CNN Inference

**Course:** EC3048E - Hardware Architectures for Deep Learning

**Instructor:** Dr. Sudhish N. George

**Demo Video:** 

Submitted by:

MUMMANA JAGADEESH    B231113EC

# Contents

|          |                                                                   |           |
|----------|-------------------------------------------------------------------|-----------|
| <b>1</b> | <b>Abstract</b>                                                   | <b>3</b>  |
| <b>2</b> | <b>Introduction</b>                                               | <b>4</b>  |
| <b>3</b> | <b>CNN Model Design and Training</b>                              | <b>5</b>  |
| 3.1      | CIFAR10 Dataset                                                   | 5         |
| 3.2      | Training Setup and Data Augmentation                              | 6         |
| 3.3      | Learning Rate and Early Stopping                                  | 8         |
| 3.4      | Model Training                                                    | 8         |
| 3.5      | Exploring Multiple Architectures                                  | 9         |
| 3.5.1    | Summary of Architectures                                          | 10        |
| 3.5.2    | Training and Loss Curves                                          | 11        |
| <b>4</b> | <b>Model Quantisation</b>                                         | <b>12</b> |
| 4.1      | Types of Quantization                                             | 12        |
| 4.1.1    | Post-Training Quantization (PTQ)                                  | 13        |
|          | Post-Training Quantization (PTQ)                                  | 13        |
| 4.1.2    | Quantization-Aware Training (QAT)                                 | 14        |
|          | Quantization-Aware Training (QAT)                                 | 14        |
| 4.1.3    | Comparison Summary                                                | 14        |
|          | Comparison Summary                                                | 14        |
| 4.2      | Floating-Point vs Fixed-Point Representation                      | 15        |
| 4.3      | Fixed-Point Representation: Qm.n Format and Bit Structure         | 16        |
| 4.4      | Hardware Considerations for Qm.n Quantization                     | 17        |
| 4.5      | Quantization Evaluation on CIFAR-10                               | 18        |
| 4.5.1    | MODEL_ARCH_4 (72,730 Parameters)                                  | 18        |
| 4.5.2    | MODEL_ARCH_8 (52,622 Parameters)                                  | 18        |
| <b>5</b> | <b>Hardware Implementation in Verilog</b>                         | <b>20</b> |
| 5.1      | Verilog HDL and Its Limitations                                   | 20        |
| 5.2      | On-Chip vs. Off-Chip Memory                                       | 22        |
| 5.3      | Types of LUTs                                                     | 22        |
| 5.4      | Fixed-Point Quantization and ROM Generation                       | 23        |
| 5.5      | Two-Cycle Handshake Protocol                                      | 24        |
| 5.6      | Image ROM Generation and CNN Input Handling                       | 25        |
| 5.7      | Convolution Core Implementation ('conv2d')                        | 26        |
| 5.8      | Max Pooling Core Implementation ('max_pool')                      | 30        |
| 5.9      | Residual / Shortcut Add Module ('add')                            | 33        |
| 5.10     | Global Average Pooling ('gap')                                    | 35        |
| 5.11     | Dense / Fully Connected Layer ('dense')                           | 36        |
| 5.12     | Softmax and Class Prediction ('softmax')                          | 38        |
| 5.13     | Top-Level CNN Module and Testbench                                | 40        |
| 5.13.1   | Top-Level Module Architecture                                     | 40        |
| 5.14     | MODEL_ARCH_4 (Initial)                                            | 42        |
| 5.15     | MODEL_ARCH_8 (Full CNN)                                           | 43        |
| <b>6</b> | <b>Systolic Array Based Matrix Multiplication and Convolution</b> | <b>44</b> |
| 6.1      | Mathematical Background and Efficiency                            | 44        |
| 6.2      | GEMM Implementation                                               | 45        |

|          |                                                                    |           |
|----------|--------------------------------------------------------------------|-----------|
| 6.3      | Convolutional Layer Implementation (Conv2D) . . . . .              | 47        |
| 6.4      | Yosys Netlists of Modules . . . . .                                | 50        |
| 6.5      | Comparison of MAC Unit Architectures . . . . .                     | 52        |
| 6.5.1    | 8-bit Signed Adders . . . . .                                      | 52        |
| 6.5.2    | 8-bit Signed Multipliers . . . . .                                 | 53        |
| 6.5.3    | Summary of MAC Unit Efficiency . . . . .                           | 54        |
| 6.6      | Timing Estimate (MODEL_ARCH_8) . . . . .                           | 55        |
| 6.7      | GDS and Layout Visualizations for Adders and Multipliers . . . . . | 57        |
| 6.7.1    | Adder Visualizations . . . . .                                     | 57        |
| 6.7.2    | Multiplier Visualizations . . . . .                                | 57        |
| <b>7</b> | <b>References &amp; Listings . . . . .</b>                         | <b>59</b> |
| <b>8</b> | <b>Demo Video . . . . .</b>                                        | <b>60</b> |

# 1 Abstract

Deep learning models, while highly accurate, are often computationally expensive and resource-intensive, making their deployment on hardware accelerators both challenging and essential. In this work, we explore the design, optimization, and hardware realization of convolutional neural network (CNN) architectures targeted for efficient implementation. Our approach has two main directions: (1) model-level exploration with quantization, and (2) hardware-level implementation using Verilog and systolic array accelerators.

At the model level, we extensively explored various training techniques, CNN architectures, architectural styles, and quantization strategies to identify an optimal balance between accuracy and hardware cost. Our most minimal variant achieves 83% test accuracy with only 52k parameters, using 8-bit post-training quantization (Q1.7 fixed-point). This quantized model was implemented and deployed on a Verilog-based hardware prototype, providing an end-to-end evaluation of model-hardware interaction and inference behavior. On the hardware side, we compared multiple adder and multiplier architectures within a systolic array-based MAC framework, analyzing trade-offs in latency, throughput, area, and frequency. Through this exploration, we derived several empirical insights on efficient CNN-accelerator co-design that, to our knowledge, have not been systematically reported before.

At the hardware realization level, all implementations were carried out entirely using open-source EDA tools and other freeware tools, ensuring full reproducibility and transparency of the workflow. Despite the constraints of an open-source toolchain, we achieved a fully functional and synthesizable design flow. The second-generation Verilog implementation employs a two-cycle handshake protocol, auto-generated ROM blocks mapped to LUT structures, and a hierarchical modular architecture for layer-level flexibility. This was later consolidated into a synchronous top-level design integrating convolution, pooling, and fully connected blocks, validated through simulation and synthesis. The entire system can be reproduced or extended from scratch using only open-source frameworks, serving as a baseline for future CNN accelerator research and educational hardware prototyping.

Our results show that careful co-design of CNN architectures and hardware accelerators enables efficient fixed-point implementations with minimal accuracy loss. The experiments highlight the tradeoffs between accuracy, parameter size, and hardware complexity, while also establishing a foundation for scalable systolic-array-based deep learning accelerators



## 2 Introduction

The rapid evolution of deep learning has transformed fields such as computer vision, speech processing, and natural language understanding. The impressive accuracy of modern neural networks, however, comes at the expense of high computational and memory demands. As model sizes and data volumes grow, conventional processors struggle to sustain the required throughput within realistic power and area limits. This has made hardware acceleration essential for efficient, scalable deep learning deployment across cloud, edge, and embedded systems.

Hardware accelerators achieve performance efficiency through parallelism, pipelining, and locality of reference while tailoring compute and memory subsystems to specific workloads. Their design centers on efficient realization of multiply–accumulate (MAC) operations, which dominate both computation and power consumption. Optimizing MAC flow—via datapath refinement, resource sharing, and latency hiding—is key to sustaining high utilization. Deep pipelines raise throughput and clock frequency but require careful handling of hazards and synchronization to prevent stalls.

At the architectural level, efficiency depends on balancing computation and data movement. Memory accesses often consume more energy than arithmetic, making dataflow and memory hierarchy design critical. Techniques like tiling, buffering, and double-buffered scheduling maintain data availability, while bandwidth-aware partitioning limits off-chip traffic. The chosen dataflow—input-, output-, or weight-stationary—directly shapes reuse patterns and energy efficiency.

Microarchitectural design must optimize for area, latency, and power under strict timing constraints. Pipeline depth, logic fanout, interconnect topology, and clock gating all affect frequency and power. Likewise, arithmetic precision and quantization influence accuracy and complexity, creating trade-offs between numerical fidelity and resource use. Fixed-point and low-bit representations thus remain popular for inference accelerators.

Beyond datapath and memory tuning, hardware–software co-design emphasizes joint optimization of model and hardware configurations to maximize energy efficiency while preserving accuracy. This includes restructuring algorithms for locality, exploiting sparsity, and mapping computation effectively to hardware fabrics.

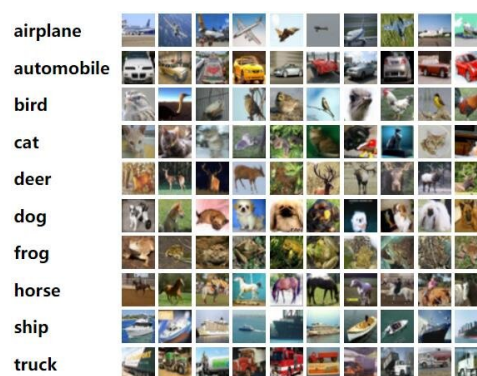
Despite steady progress, accelerator design faces ongoing challenges: balancing performance, area, and energy efficiency; ensuring scalability as models evolve; and maintaining reproducibility across toolchains. Addressing these demands a holistic view spanning arithmetic unit design, pipeline scheduling, memory organization, and system integration. As deep learning proliferates from data centers to edge devices, custom, power-aware, and reconfigurable hardware will remain vital to sustaining its growth and practical adoption.

## 3 CNN Model Design and Training

### 3.1 CIFAR10 Dataset

The CIFAR-10 dataset is a widely used benchmark in the field of computer vision and deep learning. It consists of 60,000 color images divided into 10 distinct classes, with 6,000 images per class. Among these, 50,000 images are used for training and 10,000 images for testing. Each image has a fixed resolution of  $32 \times 32$  pixels and contains three color channels (Red, Green, and Blue), making the dataset suitable for RGB-based image processing tasks.

The ten classes in the CIFAR-10 dataset are:



Sample images from the CIFAR-10 dataset representing all 10 classes.

- **Airplane:** Images of airplanes in various orientations and backgrounds, typically capturing the full aircraft.
- **Automobile:** Images of cars, trucks, and other vehicles, usually seen from the side or front view.
- **Bird:** Various species of birds captured in different poses, often perched or in flight.
- **Cat:** Images of domestic cats in diverse postures and backgrounds.
- **Deer:** Deer in natural settings, often standing or grazing.
- **Dog:** Domestic dogs of various breeds and poses.
- **Frog:** Frogs captured in natural environments, frequently shown on plants or water surfaces.
- **Horse:** Horses in different stances, sometimes in fields or riding contexts.
- **Ship:** Images of ships, boats, and other watercraft, generally on water bodies.
- **Truck:** Trucks and lorries, often seen on roads or in industrial settings.

Each image is small, low-resolution, and contains varied backgrounds, which makes the dataset challenging for classification tasks. Despite its simplicity, CIFAR-10 is an effective benchmark for testing image recognition models and evaluating their ability to learn features from RGB images with limited resolution.

### 3.2 Training Setup and Data Augmentation

All models in this study are pure convolutional neural networks trained end-to-end on **Colab** using GPU acceleration. To improve generalization and reduce overfitting, extensive **data augmentation** was applied to the CIFAR-10 dataset. Augmentation techniques included random rotations, horizontal flips, and small translations in both width and height directions. This allows the model to see varied versions of the same image and encourages robustness to small perturbations in the input.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Data augmentation
datagen = ImageDataGenerator(
    rotation_range=15,
    horizontal_flip=True,
    width_shift_range=0.1,
    height_shift_range=0.1,
)
datagen.fit(x_train)
```

The training pipeline was designed to be robust and adaptive, ensuring stable convergence across different neural network architectures. Specifically, all models were compiled using the Adam optimizer, which combines the benefits of adaptive learning rates and momentum-based updates. Given a set of parameters  $\theta$  at training step  $t$ , the Adam optimizer updates the parameters according to:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla_{\theta} \mathcal{L}(\theta_{t-1}) \quad (1)$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) (\nabla_{\theta} \mathcal{L}(\theta_{t-1}))^2 \quad (2)$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \quad (3)$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \quad (4)$$

$$\theta_t = \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \quad (5)$$

where  $m_t$  and  $v_t$  are the first and second moment estimates,  $\beta_1$  and  $\beta_2$  are decay rates,  $\alpha$  is the learning rate, and  $\epsilon$  is a small constant to prevent division by zero.

For the **loss function**, we employed **categorical cross-entropy** to reflect the multi-class classification nature of CIFAR-10. Given a true label vector  $y = [y_1, \dots, y_C]$  and predicted probabilities  $\hat{y} = [\hat{y}_1, \dots, \hat{y}_C]$  for  $C$  classes, the loss is computed as:

$$\mathcal{L}_{\text{cross-entropy}}(y, \hat{y}) = - \sum_{i=1}^C y_i \log(\hat{y}_i) \quad (6)$$

Finally, **accuracy** was monitored as the primary performance metric:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} \quad (7)$$

In code, the model compilation step can be expressed as:

```
model = build_model()
model.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
```

### 3.3 Learning Rate and Early Stopping

Training employed two complementary mechanisms to improve convergence and prevent overfitting:

1. **Learning Rate Reduction on Plateau:** This callback monitors validation accuracy and reduces the learning rate by a factor of 0.5 if the metric does not improve for 5 consecutive epochs. This allows the model to make finer adjustments once it approaches a local minimum.
2. **Early Stopping:** Training halts if validation accuracy does not improve for 15 epochs, restoring the best model weights. This prevents unnecessary overfitting and reduces training time.

```
from tensorflow.keras.callbacks import ReduceLROnPlateau, EarlyStopping
```

```
# Callbacks
```

```
lr_reduction = ReduceLROnPlateau(monitor='val_accuracy', patience=5, factor=0.5, verbose=1)
early_stop = EarlyStopping(monitor='val_accuracy', patience=15, restore_best_weights=True)
```

### 3.4 Model Training

Training was performed using a **batch size of 64** over a maximum of **100 epochs**, leveraging the data generator to feed augmented batches. The use of online augmentation ensures continuous exposure to varied inputs during training, reducing the model's reliance on memorizing the dataset.

```
history = model.fit(
    datagen.flow(x_train, y_train, batch_size=64),
    validation_data=(x_test, y_test),
    epochs=100,
    callbacks=[lr_reduction, early_stop],
    verbose=1
)
```

This setup provides a balance between convergence speed, model stability, and generalization, making it suitable for experiments with multiple architectures. Adam's adaptive updates accelerate convergence for deeper networks, while learning rate scheduling refines weight updates in later stages. Early stopping minimizes wasted computation and mitigates overfitting risks.

### 3.5 Exploring Multiple Architectures

The architectural exploration began with **MODEL\_ARCH\_1**, a deep convolutional neural network (CNN) consisting of three convolutional blocks with  $64 \rightarrow 128 \rightarrow 256$  filters. Each block was followed by Batch Normalization (BN) and ReLU activations. A fully connected (Dense) layer with 512 units preceded the final softmax classifier. The inclusion of BN stabilized training by normalizing intermediate activations, reducing internal covariate shift, and enabling higher learning rates. Although this model achieved the highest test accuracy of 90.91%, its parameter count ( $\sim 3.25\text{M}$ ) made it less suitable for hardware-constrained environments.

To reduce computational complexity and improve deployment efficiency, **MODEL\_ARCH\_2** reduced the convolutional filter sizes to  $32 \rightarrow 64 \rightarrow 128$  and replaced the Dense512 layer with a smaller Dense256 layer, while retaining BN. This design achieved a modest accuracy drop (88.84%) but substantially lowered the parameter count to approximately 0.82M.

Further optimization led to **MODEL\_ARCH\_3**, where Batch Normalization was removed entirely to minimize floating-point operations and simplify inference hardware. This modification resulted in an accuracy of 85.53% with around 0.52M parameters, offering a balance between efficiency and performance.

In **MODEL\_ARCH\_4**, the fully connected layers were replaced by a *Global Average Pooling* (GAP) layer, which aggregates spatial features by averaging across each feature map prior to classification. This change drastically reduced the parameter count to approximately 72.7k while maintaining competitive accuracy (83.05%), mitigating overfitting, and preserving essential feature representations.

Subsequent simplifications were explored through **MODEL\_ARCH\_5** and **MODEL\_ARCH\_6**. **MODEL\_ARCH\_5** employed only two convolutional blocks ( $16 \rightarrow 32$  filters) with GAP and a Dense10 output, achieving 67.94% accuracy with a minimal parameter count of about 17k. To regain lost representational capacity, **MODEL\_ARCH\_6** increased the filter count to  $32 \rightarrow 64$ , improving accuracy to 78.92% while keeping the parameters low ( $\sim 66\text{k}$ ).

To enhance feature propagation without increasing depth, **MODEL\_ARCH\_7** introduced *residual connections*. Each convolutional block incorporated a  $1 \times 1$  convolutional shortcut, enabling direct gradient flow and mitigating vanishing gradient issues. Combined with GAP and Dense10 output, this architecture achieved 81.08% accuracy with approximately 68k parameters.

Finally, **MODEL\_ARCH\_8** refined the residual structure by reducing the number of filters to  $28 \rightarrow 56$ , thereby lowering the parameter count to roughly 52.6k while maintaining a comparable accuracy of 80.45%. This model represented a well-balanced trade-off between compactness and representational strength.

### 3.5.1 Summary of Architectures

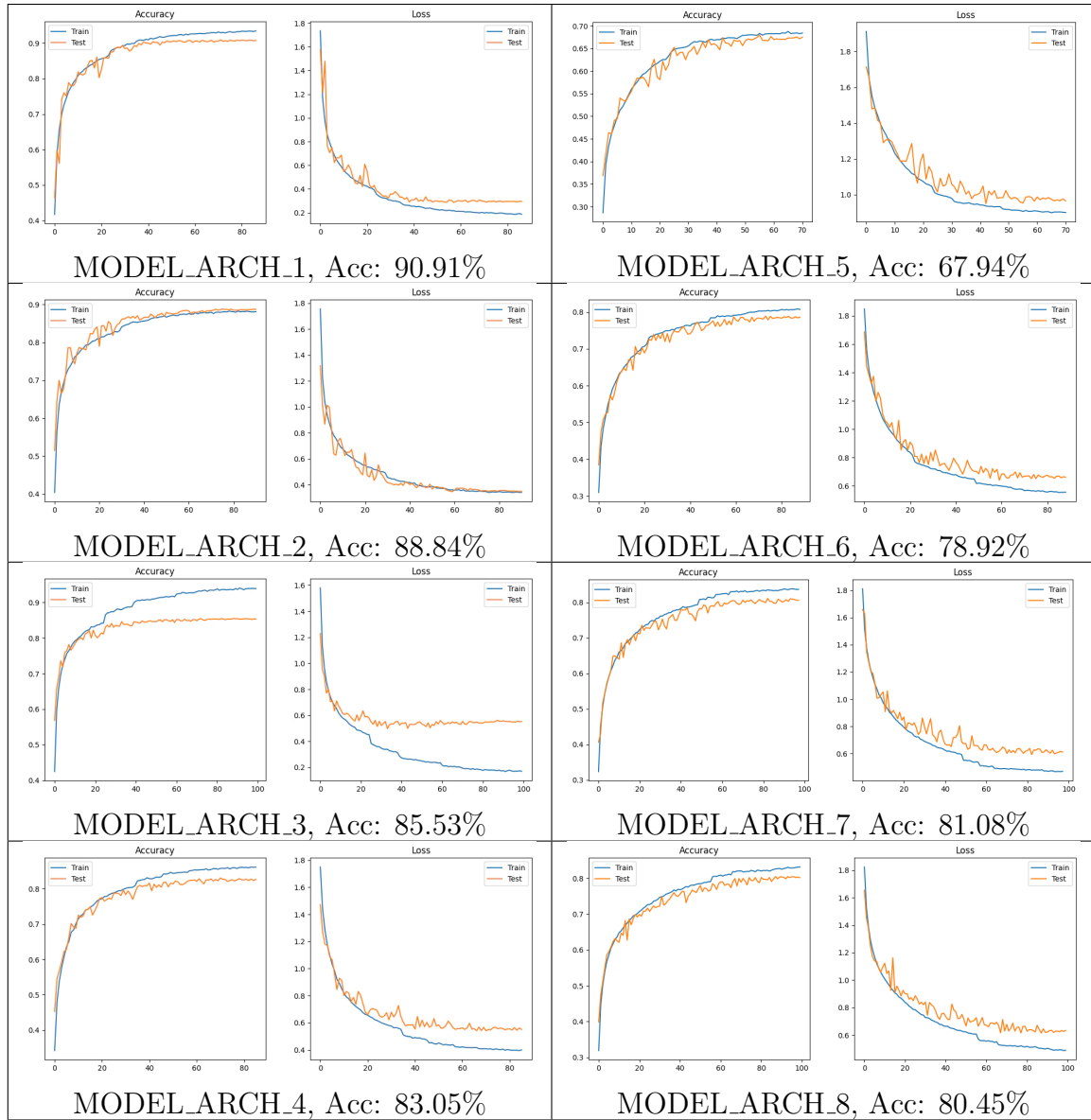
| Model        | Short Name /<br>Key Layers                                                         | Test Accuracy (%) | Total Parameters |
|--------------|------------------------------------------------------------------------------------|-------------------|------------------|
| MODEL_ARCH_1 | 64→128→256<br>convs, BN,<br>Dense512 (first<br>long model)                         | 90.91             | 3,253,834        |
| MODEL_ARCH_2 | 32→64→128<br>convs, BN,<br>Dense256                                                | 88.84             | 816,938          |
| MODEL_ARCH_3 | 32→64→96 convs,<br>no BN, Dense256                                                 | 85.53             | 517,002          |
| MODEL_ARCH_4 | 16→32→64 convs,<br>GlobalAvgPool,<br>Dense10                                       | 83.05             | 72,730           |
| MODEL_ARCH_5 | 16→32 convs,<br>GlobalAvgPool,<br>Dense10 (smaller)                                | 67.94             | 16,986           |
| MODEL_ARCH_6 | 32→64 convs,<br>GlobalAvgPool,<br>Dense10                                          | 78.92             | 66,218           |
| MODEL_ARCH_7 | Residual: [32→32<br>+ 1×1 shortcut],<br>[32→64 + 1×1<br>shortcut], GAP,<br>Dense10 | 81.08             | 68,458           |
| MODEL_ARCH_8 | Residual (28→56<br>filters): same<br>structure as #7                               | 80.45             | 52,622           |

Comparison of different model architectures

| Model        | FLOPs ( $\times 10^6$ ) |
|--------------|-------------------------|
| MODEL_ARCH_1 | 276.5                   |
| MODEL_ARCH_2 | 99.7                    |
| MODEL_ARCH_3 | 68.2                    |
| MODEL_ARCH_4 | 20.4                    |
| MODEL_ARCH_5 | 7.2                     |
| MODEL_ARCH_6 | 13.8                    |
| MODEL_ARCH_7 | 15.2                    |
| MODEL_ARCH_8 | 12.6                    |

FLOPs of each model architecture in millions.

### 3.5.2 Training and Loss Curves



Training and loss curves for each architecture

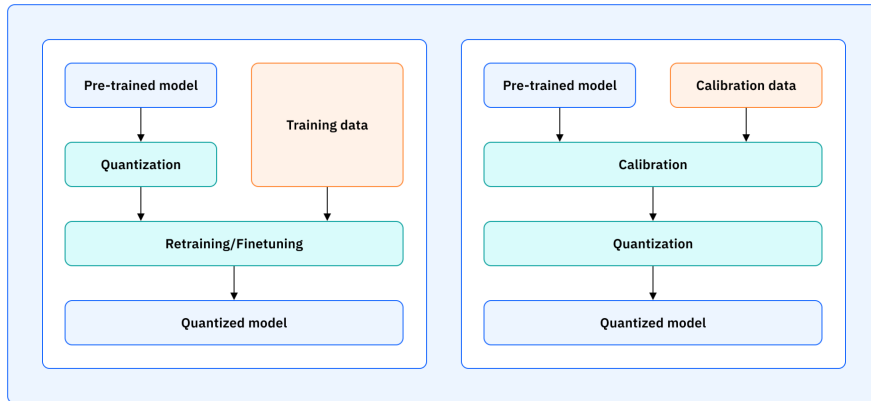


## 4 Model Quantisation

Quantization is a fundamental technique in modern deep learning, especially when deploying models on resource-constrained hardware like FPGAs, microcontrollers, or edge devices. At its core, quantization involves approximating high-precision floating-point values, which are typically stored using 32-bit or 16-bit floating-point representations, with lower-precision fixed-point or integer representations. This reduces the memory footprint, decreases memory bandwidth requirements, lowers power consumption, and accelerates inference. While this approximation can introduce small errors, careful application of quantization allows models to retain nearly the same predictive performance while enabling efficient hardware implementation.

The motivation for quantization is primarily practical. Floating-point arithmetic, especially in deep neural networks with millions of parameters, is expensive in terms of both computation and memory. On FPGAs or ASICs, high-precision floating-point units occupy significant silicon area and consume considerable power. By reducing the bit-width of weights and activations, we can implement simpler arithmetic units, reduce memory usage, and increase parallelism, which results in faster and more energy-efficient execution.

### 4.1 Types of Quantization



Quantization methods in deep learning generally fall into two categories: Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT). Each has distinct mechanisms, advantages, and trade-offs.

#### 4.1.1 Post-Training Quantization (PTQ)

Post-Training Quantization converts a pre-trained floating-point model into a lower-precision representation *after training*, without modifying the training process. This makes PTQ simple and fast, but it may introduce small accuracy loss due to quantization errors.

```
# Example: PTQ in PyTorch
import torch
from torch.quantization import quantize_dynamic

# Pre-trained floating-point model
model_fp = MyModel()

# Convert model weights to 8-bit integers dynamically
model_int8 = quantize_dynamic(model_fp, {torch.nn.Linear}, dtype=torch.qint8)
```

In PTQ, weights and activations are scaled and rounded to integer or fixed-point values. For example, a weight  $w_{fp}$  in the range  $[-1,1)$  is converted to fixed-point  $w_q$  as:

$$w_q = \text{round}(w_{fp} \cdot (2^n - 1))$$

This allows the model to run efficiently on hardware without retraining.

### 4.1.2 Quantization-Aware Training (QAT)

Quantization-Aware Training simulates the effects of quantization *during training*. Fake quantization nodes are inserted in the forward pass to emulate rounding and truncation, while gradients are computed using the Straight-Through Estimator (STE) to allow learning of quantization-resilient weights.

```
# Example: QAT in PyTorch
import torch
from torch.quantization import prepare_qat, convert

model_fp = MyModel()
model_fp.train()

# Prepare the model for QAT
model_qat = prepare_qat(model_fp)

# Train the model with simulated quantization
train(model_qat, train_loader, epochs=5)

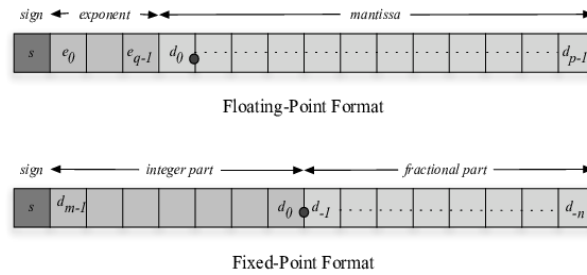
# Convert to actual quantized model after training
model_int8 = convert(model_qat)
```

QAT typically results in higher post-quantization accuracy than PTQ, particularly for very low bit-widths, but requires retraining and additional computation during training.

### 4.1.3 Comparison Summary

- **PTQ:** Fast, simple, no retraining, small accuracy loss.
- **QAT:** Higher accuracy, adapts to quantization noise, requires retraining.

## 4.2 Floating-Point vs Fixed-Point Representation



In deep learning, numerical values such as model weights, activations, and biases can be stored using either floating-point or fixed-point representations. Floating-point formats, such as FP32 (32-bit float) or FP16 (16-bit float), provide a wide dynamic range and high precision, thanks to their structure: a sign bit, exponent bits, and mantissa bits. The exponent allows the number to scale up or down over a large range, which is particularly useful for training deep networks where values can vary dramatically. For instance, FP32 has 1 sign bit, 8 exponent bits, and 23 mantissa bits, supporting extremely small or large numbers.

While floating-point provides flexibility and precision, it comes with significant drawbacks in hardware implementations. Floating-point arithmetic units are resource-intensive, consume more power, and have higher latency. On FPGAs and ASICs, implementing FP32 multipliers and adders can occupy a large portion of logic resources or DSP blocks, limiting the number of parallel computations.

Fixed-point representation, in contrast, divides a number into a fixed number of bits for the integer and fractional parts. There is no exponent; the scaling is implicit. This approach reduces hardware complexity because arithmetic operations can be performed using simple integer addition and multiplication, which are much more hardware-efficient. For example, a Q1.7 fixed-point number uses 1 bit for the sign, 0 bits for additional integer magnitude (range [-1, 0.992]), and 7 bits for the fractional part (precision approx 0.0078). Fixed-point representation is therefore widely used in inference on embedded devices, where memory and power are limited.

To illustrate the difference:

```
# Representing 0.625
# FP8 (floating point, simplified 8-bit)
0.625 -> sign:0, exponent:011, mantissa:1010000

# Q1.7 fixed-point
0.625 * 2^7 = 80 -> binary: 01010000
```

From this example, it is clear that fixed-point can represent values with sufficient precision for many inference tasks while using simpler and smaller hardware units, at the cost of dynamic range.

### 4.3 Fixed-Point Representation: Qm.n Format and Bit Structure

Fixed-point numbers are commonly represented using the **Qm.n notation**, where  $m$  denotes the number of integer bits (including the sign bit) and  $n$  denotes the number of fractional bits. The total bit-width is  $m+n$ , with each bit serving a specific purpose. *Note that some resources may describe the format differently or use alternative bit structures; in this work, we follow the convention based on the resources listed at the end.*

- The **most significant bit (MSB)** represents the sign (0 for positive, 1 for negative).
- The next  $m - 1$  bits represent the integer part.
- The remaining  $n$  least significant bits (LSBs) represent the fractional part.

A Qm.n number represents values in the range:

$$x \in [-2^{m-1}, 2^{m-1} - 2^{-n}]$$

The fractional bits determine the smallest step (resolution) between two representable numbers:

$$\text{Step size} = 2^{-n}$$

For example, Q1.7 numbers can represent values from -1 to 0.992 in increments of 0.0078. Increasing  $n$  increases precision but requires larger multipliers for arithmetic operations. Increasing  $m$  expands the representable range but also increases the bit-width, consuming more memory and hardware resources.

| Total bits | Integer bits (m) | Fractional bits (n) | Range / Step size       |
|------------|------------------|---------------------|-------------------------|
| 8          | 1                | 7                   | [-1, 0.992] / 0.0078    |
| 8          | 2                | 6                   | [-2, 1.984] / 0.0156    |
| 16         | 1                | 15                  | [-1, 0.99997] / 0.00003 |
| 16         | 4                | 12                  | [-8, 7.9998] / 0.000244 |

This notation also directly informs hardware implementation: multipliers and adders can be sized according to the total bit-width, and overflow/underflow handling can be incorporated depending on the number of integer bits.

## 4.4 Hardware Considerations for Qm.n Quantization

Implementing Qm.n quantization on hardware requires careful consideration of the trade-offs between accuracy, resource usage, and performance:

**Integer Bits (m):** The integer bits define the range of representable numbers. Choosing too few integer bits can result in overflow during computation, especially in deeper layers where activation values may accumulate. Choosing too many bits increases the bit-width of multipliers and adders, consuming more LUTs and DSP blocks in FPGAs or more area in ASICs.

**Fractional Bits (n):** Fractional bits determine precision. More fractional bits reduce quantization error, improving model accuracy, but require larger multipliers and increase power consumption. Fewer fractional bits reduce resource usage and latency but can cause a noticeable drop in inference quality.

**Memory Footprint:** The total bit-width  $m + n$  directly affects memory usage for storing weights, biases, and activations. For large models, reducing bit-width from 32 bits (FP32) to 8 bits (Q1.7 or Q2.6) can reduce memory usage by 4x, enabling deployment on devices with limited RAM or cache.

**Arithmetic Unit Design:** Multiplication and addition are implemented using integer arithmetic. For example, an 8-bit Q1.7 multiplier can be realized with a single DSP slice in an FPGA. Doubling the fractional bits requires doubling the size of the multiplier, which increases latency and resource usage. Accumulation must also be carefully managed: when summing many fixed-point numbers, intermediate results may require additional integer bits to avoid overflow.

**Latency and Power:** Reducing bit-width reduces switching activity and the number of gates required, which decreases both latency and dynamic power consumption. For real-time or battery-powered systems, this is critical. Choosing the right Qm.n format is therefore a balance between acceptable accuracy and minimal hardware cost.

**Practical Example:** For a normalized neural network with weights in  $[-1, 1)$ ,  $m = 1$  is usually sufficient. Choosing  $n = 7$  allows reasonable precision for inference with minimal hardware usage. Increasing  $n$  to 15 improves accuracy but doubles the size of multipliers and increases memory footprint, often unnecessarily for edge applications.

## 4.5 Quantization Evaluation on CIFAR-10

We evaluated two different model architectures, **MODEL\_ARCH\_4** and **MODEL\_ARCH\_8**, using a small subset of the CIFAR-10 dataset (100 images, 10 images per class). Both Python simulations and Verilog hardware implementations were compared across multiple fixed-point formats (Qm.n) as well as floating-point formats. We also calculated the corresponding model sizes for each representation.

**Model Size Calculation:** The size of a model in bytes can be estimated as:

$$\text{Model Size (bytes)} = \text{Number of Parameters} \times \frac{\text{Bit-width}}{8}$$

where the bit-width corresponds to the total bits of the numeric format (e.g., Float32 = 32 bits, Q1.7 = 8 bits).

### 4.5.1 MODEL\_ARCH\_4 (72,730 Parameters)

| Format  | Accuracy (%) | Model Size (KB) |
|---------|--------------|-----------------|
| Float32 | 85           | 291.0           |
| Q1.31   | 84           | 291.0           |
| Q1.15   | 84           | 145.0           |
| Q1.7    | 83           | 72.7            |
| Q1.3    | 65           | 36.4            |

**Observations:** As expected, reducing the fractional bit-width  $n$  in Qm.n leads to smaller model sizes. Accuracy remains high for Q1.31 and Q1.15 formats, while Q1.3 shows a significant drop. The Verilog implementation closely matches Python results, confirming that fixed-point arithmetic can be accurately realized in hardware.

### 4.5.2 MODEL\_ARCH\_8 (52,622 Parameters)

| Format  | Accuracy (%) | Model Size (KB) |
|---------|--------------|-----------------|
| Float32 | 80           | 210.49          |
| Q1.31   | 84           | 210.49          |
| Q1.15   | 84           | 105.24          |
| Q1.7    | 83           | 52.62           |
| Q1.3    | 78           | 26.31           |

**Observations:** Similar trends are observed for MODEL\_ARCH\_8. High-precision fixed-point formats (Q1.31 and Q1.15) maintain accuracy comparable to floating-point, while

lower bit-widths reduce model size at the cost of accuracy. Even with Q1.7, the model retains reasonable performance, making it suitable for embedded hardware deployment.

**Summary:** These evaluations demonstrate a clear trade-off between model size and accuracy. High-bit fixed-point formats allow near-floating-point accuracy with moderate memory reduction. Lower-bit formats dramatically reduce model size but incur an accuracy penalty. This provides guidance for selecting Qm.n formats in hardware implementations, balancing resource usage and performance.



## 5 Hardware Implementation in Verilog

### 5.1 Verilog HDL and Its Limitations

A CNN implemented in hardware is fundamentally different from its software counterpart. In software, an image is typically loaded into memory and processed in batch. On an FPGA, however, images are captured directly from a camera as a continuous stream of pixel data. The camera outputs pixels sequentially, often in RGB format, along with synchronization signals such as **VSYNC** (Vertical SYNC) and **HSYNC** (Horizontal SYNC).

#### Camera Interface on FPGA

- **VSYNC:** Indicates the start of a new frame. When this signal is asserted, the FPGA knows that the first row of the image is being transmitted.
- **HSYNC:** Indicates the start of a new line. For every assertion, the FPGA knows that the next row of pixels is incoming.
- **Pixel Clock (PCLK):** Driven by the camera, this clock signals when a new pixel is valid on the data bus.

The FPGA uses these signals to latch pixel values in real time. Typically, a FIFO or line buffer is implemented to temporarily store pixels before feeding them to the CNN hardware. For simulation purposes, instead of a live camera feed, images are preloaded as text files, `.mem` files, or auto-generated ROM modules. Pretrained CNN kernels and biases are similarly stored in memory for easy access.

#### Example: Pixel Capture Logic (Simplified)

```
always @(posedge pclk) begin
    if (vsync) begin
        row <= 0; col <= 0;
    end else if (hsync) begin
        col <= 0; row <= row + 1;
    end else begin
        pixel_buffer[row][col] <= pixel_data;
        col <= col + 1;
    end
end
```

**Limitations in Verilog HDL** While Verilog is powerful for RTL design, it has limitations:

- Not all constructs are synthesizable. For instance, `$readmemh` works in simulation but may not be supported in synthesis for some FPGA flows.
- Real-time camera interfacing requires precise timing management, which cannot always be verified in simulation without proper testbenches or stimulus generators.
- Complex data structures (like multi-dimensional arrays) must often be flattened to 1D for hardware storage, requiring careful index calculations.

### Example: Simple ROM Initialization Using `$readmemh`

```
reg [7:0] rom [0:1023];  
initial begin  
    $readmemh("weights_layer1.mem", rom);  
end
```

### Tools Used

- **Iverilog:** Simulation of Verilog modules.
- **Python:** Automation of ROM generation and fixed-point conversion.
- **TCL/Python:** Scripted inference automation for testing.
- **EDA tools:** Yosys, OpenLane, OpenROAD, KLayout for synthesis, layout, and routing.

## 5.2 On-Chip vs. Off-Chip Memory

Memory in FPGA-based CNNs can reside on-chip (BRAM, LUTRAM) or off-chip (external DDR, SRAM). The choice affects latency, throughput, and resource usage.

- **On-Chip Memory:** Low latency and high bandwidth make on-chip memory ideal for small, frequently-accessed data like feature maps or partial kernels. BRAMs or LUTRAMs store these values.
- **Off-Chip Memory:** Large datasets, full CNN weights, or images are stored off-chip. Access is slower and requires careful timing, but allows handling of large models.
- **Hybrid Approach:** In this project, small weight matrices and feature maps are stored on-chip, while full layer weights and images are placed in ROMs or off-chip memory.

### Example: BRAM Instantiation

```
(* ram_style = "block" *) reg [7:0] bram[0:1023];
always @(posedge clk) begin
    if (we) bram[addr] <= data_in;
    data_out <= bram[addr];
end
```

## 5.3 Types of LUTs

Look-Up Tables (LUTs) are the smallest logic units in FPGAs and can be used to implement combinational logic or small memory blocks.

- **1- to 2-input LUTs:** Implement simple combinational logic functions.
- **4- to 6-input LUTs:** Used for small multipliers, adders, or logic in CNN modules.
- **Distributed RAM LUTs:** LUTs configured as small RAM blocks to store CNN weights or activations.

### Example: LUT as RAM

```
reg [7:0] lut_ram[0:15]; // 16 elements
always @(posedge clk) begin
    lut_ram[addr] <= data_in;
    data_out <= lut_ram[addr];
end
```

## 5.4 Fixed-Point Quantization and ROM Generation

Floating-point CNN weights are converted to fixed-point (e.g., Q1.7) to reduce memory usage and increase hardware efficiency. Overflow handling is crucial during conversion:

```
SCALE = 1 << 7
MAX_VAL = 127
MIN_VAL = -128

def float_to_fixed(f):
    val = int(round(f*SCALE))
    val = min(max(val, MIN_VAL), MAX_VAL)
    return val
```

Weights are flattened into layer-wise ROM files in hexadecimal format:

```
for val in fixed_values:
    hexval = format((val + 256) % 256, "02X")
    f.write(hexval + "\n")
```

### Parameterized ROM Module

```
module rom_layer #(parameter DEPTH=1024, WIDTH=8)(
    input wire clk, rst,
    input wire addr_valid,
    output reg addr_ready,
    input wire [$clog2(DEPTH)-1:0] addr,
    output reg data_valid,
    input wire data_ready,
    output reg [WIDTH-1:0] data
);
    reg [WIDTH-1:0] rom [0:DEPTH-1];
    initial $readmemh("weights_layer.mem", rom);
endmodule
```

## 5.5 Two-Cycle Handshake Protocol

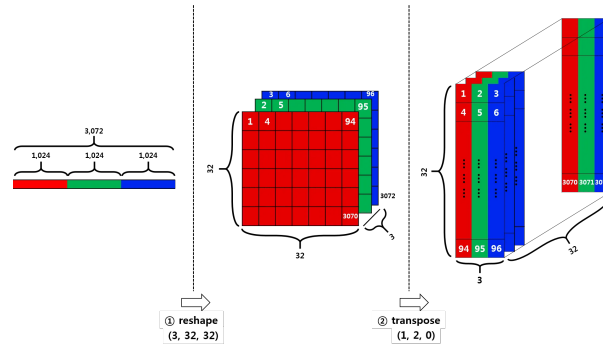
The handshake protocol ensures data integrity and proper timing between the ROM and the consumer module (CNN).

- **Cycle 1 (Address Phase):** The consumer asserts `addr_valid` and provides an address. If the ROM is ready (`addr_ready=1`), the ROM latches the address and prepares the data. The ROM then de-asserts `addr_ready` to prevent new addresses until data is read.
- **Cycle 2 (Data Phase):** The ROM asserts `data_valid` to indicate that valid data is available. The consumer reads the data and asserts `data_ready`. Once the data is consumed, the ROM de-asserts `data_valid` and reasserts `addr_ready` for the next request.
- **Importance:** This protocol prevents race conditions, ensures synchronization across clock domains if needed, and allows the consumer to read data at its own pace without losing or overwriting data.

### Verilog Implementation of Two-Cycle Handshake

```
// Address phase
if (addr_valid && addr_ready) begin
    addr_reg <= addr;
    data <= rom[addr];
    data_valid <= 1'b1;
    addr_ready <= 1'b0;
end
// Data phase
if (data_valid && data_ready) begin
    data_valid <= 1'b0;
    addr_ready <= 1'b1;
end
```

## 5.6 Image ROM Generation and CNN Input Handling



Images are split into RGB channels, converted to 8-bit integers, and stored in `.mem` files. Each channel gets a dedicated ROM module:

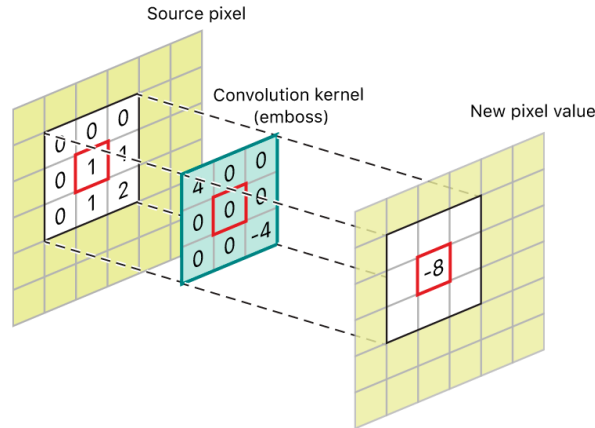
```
for i in range(32):
    for j in range(32):
        f.write(f"{array[i,j]:02X}\n")
```

Verilog ROM:

```
module image_r_rom(
    input wire clk,
    input wire [9:0] addr,
    output reg [7:0] data
);
    reg [7:0] rom[0:1023];
    initial $readmemh("image_r.mem", rom);
    always @(posedge clk) data <= rom[addr];
endmodule
```

The CNN module reads from these ROMs using the two-cycle handshake protocol described earlier, ensuring proper synchronization between image inputs and convolution layers.

## 5.7 Convolution Core Implementation (‘conv2d’)



The ‘conv2d’ module implements a 2D convolution layer in hardware. The module is designed to be highly parametrizable, supporting arbitrary input dimensions, channels, number of filters, and kernel size. It also interfaces directly with ROM modules for images, kernels, and biases.

### Parametrization

- **WIDTH, HEIGHT:** Dimensions of the input feature map.
- **CHANNELS:** Number of input channels (e.g., RGB = 3).
- **FILTERS:** Number of convolutional filters.
- **K:** Kernel size (assumed square).
- **PAD:** Padding applied to input edges.
- **BIAS\_MODE\_POST\_ADD:** Determines if bias is added post MAC normalization or pre.

This parametrization allows a single hardware module to be reused for different layers without rewriting RTL.

**FSM-Based Control** The convolution is implemented using a finite state machine (FSM) that coordinates:

- Requests for image, kernel, and bias data via ROM handshakes.
- Accumulation of multiply-accumulate (MAC) results across kernel dimensions and channels.

- Moving across pixels and filters.
- Output generation and validation signaling.

### FSM Example: Pixel Processing Loop

```
// Determine if kernel element is valid
in_y = i + m - PAD;
in_x = j + n - PAD;
if ((in_y >= 0) && (in_y < HEIGHT) && (in_x >= 0) && (in_x < WIDTH)) begin
    image_addr      <= in_y * WIDTH + in_x;
    image_addr_valid <= 1'b1;
end
```

The FSM ensures all channels and kernel elements are processed systematically, preventing race conditions and handling padding gracefully.

**Handshake Protocol** All ROM accesses (image, kernel, bias) use a two-cycle handshake to synchronize between producer (ROM) and consumer ('conv2d' core):

- **Address Phase:** The FSM asserts {image/kernel/bias\_addr\_valid} and provides an address. The ROM asserts `addr_ready` when it can accept the address.
- **Data Phase:** Once data is available, the ROM asserts `data_valid`. The FSM then asserts `data_ready` to latch the value and proceed to the next element.

This handshake allows the convolution core to wait for memory without stalling the entire pipeline and is essential for synchronizing multiple ROM modules.

**Accumulation and Bias Application** The convolution core uses a signed accumulator ('accum') to store intermediate MAC results. After completing the MAC over all channels and kernel elements:

- The accumulator is normalized (shift/scaling).
- Bias is added based on `BIAS_MODE_POST_ADD`.
- ReLU is applied if the output is negative.

```
// Apply normalization and bias
if (BIAS_MODE_POST_ADD) begin
    numerator = accum;
    out_int = ((numerator * 257) + (1<<15)) >>> 16;
    out_int = out_int + bias16;
end
```



**Top-Level Integration** In a top-level CNN module, multiple ‘conv2d’ cores are instantiated alongside all ROM modules:

- Each ‘conv2d’ core receives broadcast image addresses to three channel ROMs.
- Kernel and bias ROMs are separate for each filter.
- FSMs in each core control MAC operations independently, allowing parallel processing.
- The handshake protocol ensures that multiple cores do not access ROMs simultaneously in a conflicting manner.

### FSM Advantages

- Ensures deterministic sequencing of convolution operations.
- Supports parametrized kernels, channels, and padding without rewriting RTL.
- Facilitates reuse of the convolution module across layers in a CNN.
- Simplifies integration with other layers (MaxPool, Dense, GAP) by providing synchronized output with valid signals.

**Illustrative FSM State Diagram (Simplified)** The main states of the FSM include:

- **S\_IDLE:** Wait for the `start` signal.
- **S\_START\_FILTER:** Request the bias for the current filter.
- **S\_BIAS\_WAIT:** Wait for the bias data to be valid from ROM.
- **S\_SETUP\_PIXEL:** Initialize accumulation for the current pixel.
- **S\_MAC\_DECIDE:** Determine if the current kernel element falls within valid input boundaries, handling padding.
- **S\_IMG\_REQ / S\_IMG\_WAIT:** Request and wait for the corresponding image pixel from the ROM.
- **S\_KERN\_REQ / S\_KERN\_WAIT:** Request and wait for the corresponding kernel weight.
- **S\_MAC\_ACCUM:** Perform MAC operation and iterate across channels and kernel dimensions.
- **S\_PIXEL\_DONE:** Apply normalization and bias, apply ReLU, and generate output.

- **S\_NEXT\_PIXEL / S\_NEXT\_FILTER:** Move to the next pixel or filter.
- **S\_DONE:** Signal that convolution is complete.

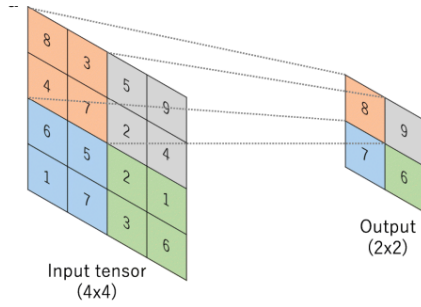
```

S_IDLE --> S_START_FILTER --> S_BIAS_WAIT --> S_SETUP_PIXEL
        --> S_MAC_DECIDE --> S_IMG_REQ --> S_IMG_WAIT
        --> S_KERN_REQ --> S_KERN_WAIT --> S_MAC_ACCUM
        --> S_PIXEL_DONE --> S_NEXT_PIXEL --> S_NEXT_FILTER
        --> S_DONE

```

This linear but looping FSM efficiently traverses all pixels, channels, and filters, making it the core control mechanism of the convolution module.

## 5.8 Max Pooling Core Implementation (‘max\_pool’)



The ‘max\_pool’ module implements a streaming max-pooling operation in hardware. It reads the input feature map (IFM) through a handshake-based interface, computes the maximum within a pooling window, and outputs the pooled results as a stream.

**Parametrization** The module is highly configurable:

- **WIDTH\_IN, HEIGHT\_IN:** Input feature map dimensions.
- **CHANNELS:** Number of input channels.
- **POOL\_SIZE:** Size of the pooling window (e.g., 2 for 2x2 pooling).
- **STRIDE:** Step size between pooling windows.

Output dimensions are derived automatically:

$$\begin{aligned}\text{WIDTH\_OUT} &= (\text{WIDTH\_IN} - \text{POOL\_SIZE}) / \text{STRIDE} + 1 \\ \text{HEIGHT\_OUT} &= (\text{HEIGHT\_IN} - \text{POOL\_SIZE}) / \text{STRIDE} + 1\end{aligned}$$

**FSM-Based Control** The module uses a finite state machine to control the max-pooling operation, systematically iterating over:

- Channels (‘c’)
- Output rows (‘ph’)
- Output columns (‘pw’)
- Window elements within the pooling region (‘pi’, ‘pj’)

**Handshake Protocol** Similar to the convolution core, the module uses a two-phase handshake for reading IFM data:

- **Address Request:** Assert `ifm_addr_valid` with the current address and channel.
- **Data Latching:** When `ifm_data_valid` is asserted, the module asserts `ifm_data_ready` and latches the value.

This ensures the core does not proceed until valid data is available, maintaining synchronization with external memory modules or testbench-provided streams.

### FSM Example: Requesting IFM Data

```
in_y = ph * STRIDE + pi;
in_x = pw * STRIDE + pj;
ifm_addr      <= in_y * WIDTH_IN + in_x;
ifm_chan      <= c;
ifm_addr_valid <= 1'b1;
```

After the handshake, the data is compared with the current maximum:

```
if (sample_q17 > max_val)
    max_val <= sample_q17;
```

**Streaming Output** Once all pixels in the pooling window are processed, the maximum value is sign-extended and streamed out:

```
// 16-bit -> 32-bit
out_data  <= { {16{max_val[15]}}, max_val };
out_valid <= 1'b1;
```

The ‘S\_NEXT’ state then advances output coordinates or moves to the next channel. This streaming approach allows the module to produce outputs continuously without storing the entire IFM internally.

### FSM Advantages

- Provides deterministic control over pooling operations.
- Supports arbitrary input sizes, channels, pooling sizes, and stride.
- Interfaces seamlessly with other modules via handshake signals.
- Enables pipelined streaming output, minimizing memory usage.

**Illustrative FSM Flow** The FSM states are:

- **S\_IDLE:** Wait for the ‘start‘ signal.
- **S\_START:** Begin processing the first channel and output cell.
- **S\_INIT\_CELL:** Reset pooling window counters and maximum value.
- **S\_REQ:** Request current pixel from IFM ROM (handshake interface).
- **S\_WAIT:** Wait for valid IFM data from ROM.
- **S\_ACC:** Update the maximum for the pooling window.
- **S\_OUTPUT:** Stream out the pooled maximum value.
- **S\_NEXT:** Move to the next output cell, channel, or finish.
- **S\_DONE:** Signal completion of max-pooling.

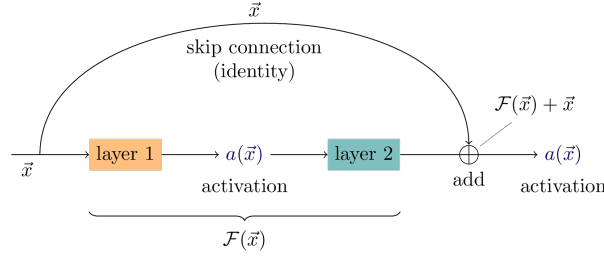
```
S_IDLE --> S_START --> S_INIT_CELL --> S_REQ --> S_WAIT
        --> S_ACC --> S_OUTPUT --> S_NEXT --> ... --> S_DONE
```

Each pooling window element is processed sequentially, while multiple channels and output cells are iterated systematically. The FSM ensures all pixels are considered, and the max is computed accurately.

**Integration in Top-Level CNN** In the top-level module:

- ‘max\_pool‘ cores receive IFM streams from preceding ‘conv2d‘ cores or ROMs.
- Handshake signals synchronize the pool with convolution outputs.
- The FSM ensures that streamed outputs are valid only when ready, enabling pipelined execution for subsequent layers (e.g., addition, GAP, Dense).

## 5.9 Residual / Shortcut Add Module (‘add’)



The **residual add module** implements skip connections commonly used in ResNet-style architectures. It combines the main convolution path with a shortcut path (either identity or  $1 \times 1$  convolution) to improve gradient flow and enable deeper networks. All computations are performed in Q7 fixed-point.

**Shortcut /  $1 \times 1$  Convolution** When the shortcut path requires dimensional alignment, a  $1 \times 1$  convolution is applied to the input to match the number of output channels. Each output pixel is computed as a weighted sum of input channels plus bias:

```
for (i = 0; i < H_IN; i = i + 1) begin
    for (j = 0; j < W_IN; j = j + 1) begin
        for (oc = 0; oc < OUT_CH; oc = oc + 1) begin
            sum = 0;
            for (ic = 0; ic < IN_CH; ic = ic + 1) begin
                prod = kernel_1x1[ic][oc] * input_img[i][j][ic]; // Q7 multiply
                sum = sum + prod;
            end
            shortcut[i][j][oc] = (sum + ROUND_CONST) / SCALE + bias_1x1[oc];
        end
    end
end
```

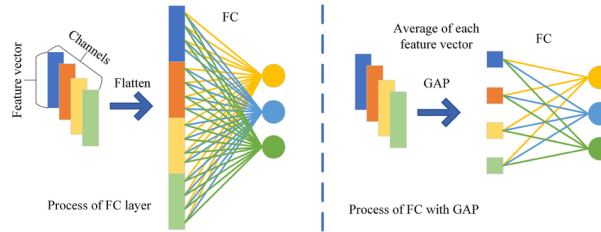
**Residual Addition** Once the shortcut is ready, the main convolution output is added element-wise to the shortcut:

```
for (i = 0; i < H_OUT; i = i + 1) begin
    for (j = 0; j < W_OUT; j = j + 1) begin
        for (oc = 0; oc < OUT_CH; oc = oc + 1) begin
            res_out[i][j][oc] = conv_out[i][j][oc] + shortcut[i][j][oc];
        end
    end
end
```

## Key Points

- Ensures the main and shortcut paths have compatible dimensions (via 1x1 convolution if needed).
- Accumulation uses sufficiently wide integers to avoid overflow in Q7 representation.
- Element-wise addition is highly parallelizable in hardware, allowing channel-wise SIMD operations.
- Supports streaming interfaces: output pixels can be sent sequentially or stored in on-chip buffers.
- Preserves fixed-point scaling and avoids intermediate floating-point computation, keeping the module fully hardware-friendly.

## 5.10 Global Average Pooling ('gap')



The **Global Average Pooling** module computes the mean of each input feature map. Instead of storing all spatial data for post-processing, GAP averages each channel across its width and height, producing a single value per channel. In our design, all data is in Q1.7 format, ensuring consistent fixed-point representation.

### Implementation Overview

- Input: feature maps of size  $H \times W$  per channel.
- Output: single value per channel.
- Computation: sum all pixel values in the channel, divide by the number of elements.

### Example snippet

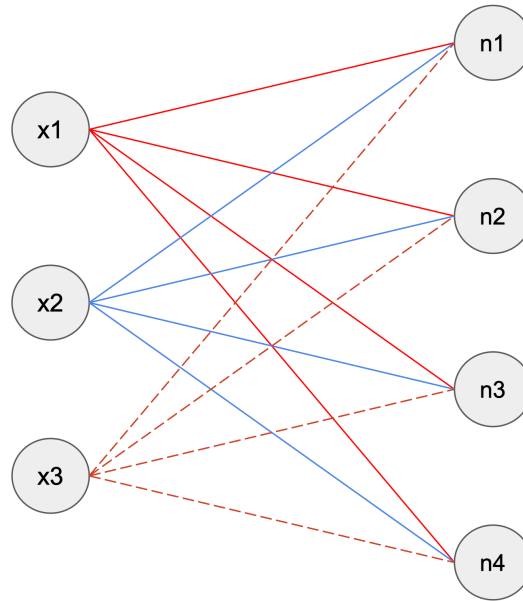
```
for (c = 0; c < CHANNELS; c = c + 1) begin
    sum = 0;
    for (i = 0; i < VALUES_PER_MAP; i = i + 1)
        sum = sum + feature_map[c][i]; // Q7 integer sum
    gap_result[c] = sum / VALUES_PER_MAP; // integer average
end
```

### Key Points

- Keeps scaling consistent (Q7).
- Outputs are directly fed into the dense layer without additional normalization.
- Reduces memory footprint by replacing each channel's feature map with a single value.



## 5.11 Dense / Fully Connected Layer ('dense')



The **dense layer** performs a matrix-vector multiplication between the GAP outputs and the dense layer's weights, followed by bias addition. The design uses Q7 fixed-point arithmetic to maintain precision while avoiding floating-point overhead.

### Implementation Highlights

- Inputs: GAP results per channel (Q7 integer).
- Weights and biases stored in ROM blocks for hardware access.
- Outputs: logits (pre-softmax scores), one per output neuron.

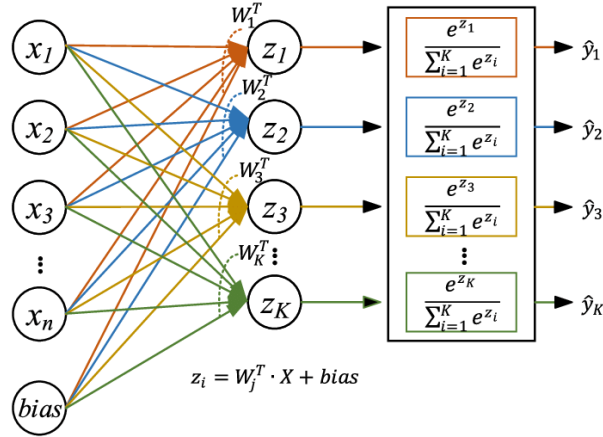
### Example snippet

```
// Accumulate weighted sum for output neuron j
accum = 0;
for (i = 0; i < INPUT_SIZE; i = i + 1) begin
    product = gap_q7[i] * kernel_q7[i][j]; // Q7*Q7
    accum = accum + product;
end
// Add bias
out_q7[j] = accum + bias_q7[j];
```

## Key Points

- Uses wide accumulators to avoid overflow.
- Each output neuron computes independently; outputs can be streamed or stored.
- ROM-based weight storage allows easy replacement or retraining.

## 5.12 Softmax and Class Prediction ('softmax')



The **softmax module** converts dense layer logits into probabilities and selects the predicted class. In hardware, it is common to pass only the index of the predicted class (0–9) to save resources. All operations are performed in Q7 fixed-point for consistency.

### Implementation Highlights

- Inputs: dense layer outputs (logits).
- Softmax computation can be done in approximate integer or floating-point for verification.
- Output: predicted class index (0–9) and optional probability mapping.

### Example snippet

```
// Find max logit for class prediction
max_idx = 0;
max_val = logits_q7[0];
for (i = 1; i < CLASS_NUM; i = i + 1) begin
    if (logits_q7[i] > max_val) begin
        max_val = logits_q7[i];
        max_idx = i;
    end
end
predicted_class = max_idx; // 0-9
```

## Key Points

- Only a single 4-bit index is needed for classification output.
- Mapping to human-readable label can be done externally (e.g., testbench or software).
- Simplifies hardware while maintaining correct top-1 prediction.

## 5.13 Top-Level CNN Module and Testbench

The top-level module integrates multiple convolution cores, max-pooling, residual add, GAP, dense, and softmax blocks into a full CNN pipeline. It is designed for Model Architecture 4 and Model Architecture 8, with streaming input from ROM-based image storage.

### 5.13.1 Top-Level Module Architecture

The top module parameterizes the first convolution core and wires subsequent cores with corresponding weights and biases. Each core has its own ROM interface for kernels and biases, and all modules communicate via handshake-based streaming.

#### Connections and Parameterization

- Conv2D cores: Each core receives pixel streams from upstream blocks or image ROMs. The first core's parameters (WIDTH, HEIGHT, CHANNELS, FILTERS, KERNEL\_SIZE) are configurable.
- ROM blocks: Each convolution core accesses its weights and biases via dedicated ROMs. Handshake signals (`addr_valid`, `addr_ready`, `data_valid`, `data_ready`) ensure proper sequencing.
- Max-Pool / Residual Add / GAP: Modules receive processed feature maps from prior blocks. Streaming handshake ensures data integrity and sequential consumption.
- Dense and Softmax: The GAP output feeds the dense layer, whose outputs feed the softmax module. Only the predicted index (0–9) is passed to the testbench.

#### Example: Conv2D Core Wiring in Top Module

```
// image ROM interface
conv2d_core0 (
    .clk(clk),
    .rst(rst),
    .start(start),
    .image_addr(image_addr),
    .image_addr_valid(image_addr_valid),
    .image_addr_ready(image_addr_ready),
    .image_r_data(image_r_q),
    .image_g_data(image_g_q),
    .image_b_data(image_b_q),
    .image_data_valid(image_data_valid),
```

```

    .image_data_ready(image_data_ready),
    .kernel_addr(kernel_addr0),
    .kernel_addr_valid(kernel_addr_valid0),
    .kernel_addr_ready(kernel_addr_ready0),
    .kernel_data(kernel_data0),
    .kernel_data_valid(kernel_data_valid0),
    .kernel_data_ready(kernel_data_ready0),
    .bias_addr(bias_addr0),
    .bias_addr_valid(bias_addr_valid0),
    .bias_addr_ready(bias_addr_ready0),
    .bias_data(bias_data0),
    .bias_data_valid(bias_data_valid0),
    .bias_data_ready(bias_data_ready0),
    .out_data(conv0_out),
    .out_valid(conv0_valid)
);

```

## Testbench (TB) Interface

- Loads images from `.mem` files for R, G, B channels.
- Streams pixel data via handshake to the first Conv2D core.
- Receives outputs sequentially from each core, forwarding to next modules (MaxPool / Residual / GAP / Dense / Softmax).
- Monitors final predicted class index (0–9) and optionally maps it to a string label.

**Top-Level FSM / Data Flow** While individual conv cores have their FSMs for MAC accumulation, the top module itself mostly wires streaming paths:

- Handshake ensures no data is lost between modules.
- Each core signals `out_valid` when its output is ready; downstream modules assert `data_ready` to consume.
- Parameterized cores allow easy scaling of FILTERS or kernel sizes without changing TB or ROM connections.

## 5.14 MODEL\_ARCH\_4 (Initial)

|            |            |            |
|------------|------------|------------|
| clk        | DEPTH=432  | addr_ready |
| rst        |            | WIDTH=8    |
| addr_valid | data_valid |            |
| addr[8:0]  | data[7:0]  |            |
| data_ready |            |            |

|            |            |            |
|------------|------------|------------|
| clk        | DEPTH=16   | addr_ready |
| rst        |            | WIDTH=8    |
| addr_valid | data_valid |            |
| addr[3:0]  | data[7:0]  |            |
| data_ready |            |            |

|            |            |            |
|------------|------------|------------|
| clk        | DEPTH=1024 | addr_ready |
| rst        |            | WIDTH=8    |
| addr[9:0]  | data_valid |            |
| addr_valid | data[7:0]  |            |
| data_ready |            |            |

Conv1 Parameters:  
kernel, bias, and input  
blue image

|                   |                      |                   |
|-------------------|----------------------|-------------------|
| clk               | WIDTH=32             | done              |
| rst               |                      |                   |
| start             |                      |                   |
| image_addr[9:0]   | HEIGHT=32            | image_data_ready  |
| image_addr_valid  |                      |                   |
| image_r_data[7:0] |                      |                   |
| image_g_data[7:0] | CHANNELS=3           | kernel_data_ready |
| image_b_data[7:0] |                      |                   |
| image_addr_ready  |                      |                   |
| image_data_valid  | FILTERS=16           | bias_data_ready   |
| kernel_addr[8:0]  |                      |                   |
| kernel_addr_valid |                      |                   |
| kernel_data[7:0]  | K=3                  | out_data[31:0]    |
| kernel_addr_ready |                      |                   |
| kernel_data_valid |                      |                   |
| bias_addr[3:0]    | PAD=1                | out_valid         |
| bias_addr_valid   |                      |                   |
| bias_data[7:0]    |                      |                   |
| bias_addr_ready   | BIAS_MODE_POST_ADD=1 |                   |
| bias_data_valid   |                      |                   |

Conv2D layer 1

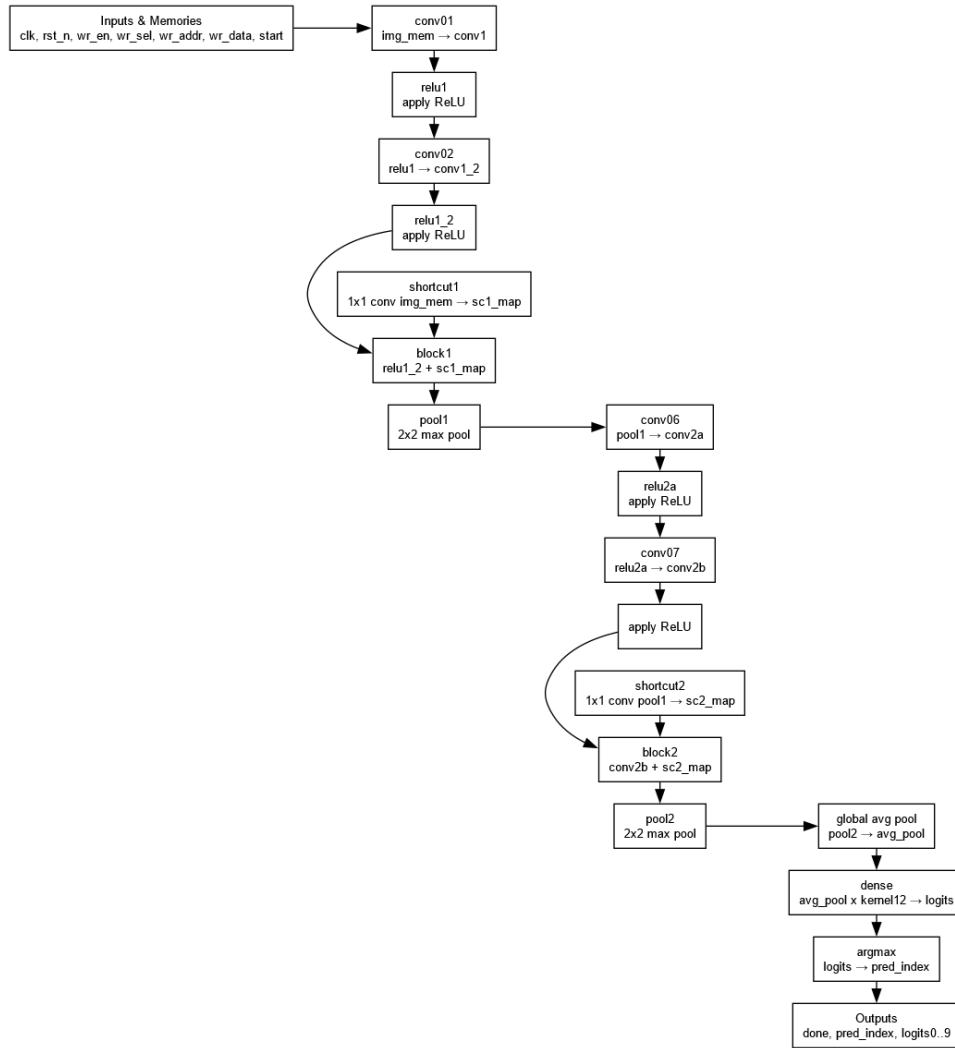
|                   |             |                |
|-------------------|-------------|----------------|
| clk               | WIDTH=32    | done           |
| rst               |             |                |
| start             |             |                |
| ifm_addr[9:0]     | HEIGHT=32   |                |
| ifm_chan[3:0]     |             |                |
| ifm_addr_valid    |             |                |
| ifm_addr_ready    | CHANNELS=16 | out_data[31:0] |
| ifm_data[15:0]    |             |                |
| ifm_data_valid    |             |                |
| ifm_data_ready    | FILTERS=16  |                |
| kernel_addr[11:0] |             |                |
| kernel_addr_valid |             |                |
| kernel_addr_ready | K=3         | out_valid      |
| kernel_data[7:0]  |             |                |
| kernel_data_valid |             |                |
| kernel_data_ready | PAD=1       |                |
| bias_addr[3:0]    |             |                |
| bias_addr_valid   |             |                |
| bias_addr_ready   |             |                |
| bias_data[7:0]    |             |                |
| bias_data_valid   |             |                |
| bias_data_ready   |             |                |

Conv2D layer 2

|                   |             |                |
|-------------------|-------------|----------------|
| clk               | WIDTH=32    | done           |
| rst               |             |                |
| start             |             |                |
| ifm_addr[9:0]     | HEIGHT=32   |                |
| ifm_chan[3:0]     |             |                |
| ifm_addr_valid    |             |                |
| ifm_addr_ready    | CHANNELS=16 | out_data[31:0] |
| ifm_data[15:0]    |             |                |
| ifm_data_valid    |             |                |
| ifm_data_ready    | FILTERS=16  |                |
| kernel_addr[11:0] |             |                |
| kernel_addr_valid |             |                |
| kernel_addr_ready | K=3         | out_valid      |
| kernel_data[7:0]  |             |                |
| kernel_data_valid |             |                |
| kernel_data_ready | PAD=1       |                |
| bias_addr[3:0]    |             |                |
| bias_addr_valid   |             |                |
| bias_addr_ready   |             |                |
| bias_data[7:0]    |             |                |
| bias_data_valid   |             |                |
| bias_data_ready   |             |                |

Maxpool layer

## 5.15 MODEL\_ARCH\_8 (Full CNN)



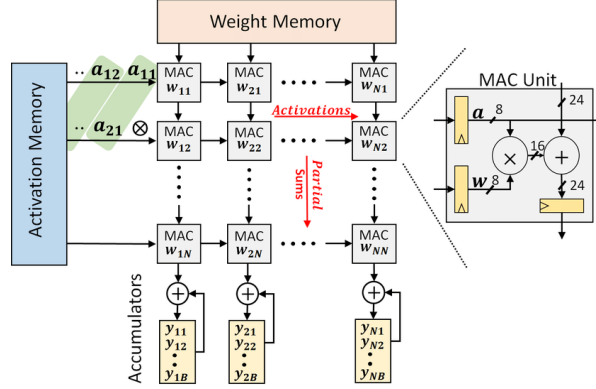
Full CNN block diagram (Model Architecture 8): convolution, residual add, pooling, GAP, dense, softmax.

### Notes

- Images stored in ROM are read sequentially and broadcast to multiple channels of the first convolution core.
- Each Conv2D, MaxPool, Residual Add, GAP, Dense, and Softmax module operates in Q7 fixed-point domain for consistency.
- Handshake ensures sequential and correct data propagation across pipeline stages.
- Top module allows easy swapping of weights and bias ROMs for experiments without modifying FSMs of individual cores.



## 6 Systolic Array Based Matrix Multiplication and Convolution



Systolic arrays are highly efficient hardware architectures designed for performing repetitive numerical computations, particularly matrix multiplication and convolution, in a structured and pipelined manner. The term “systolic” refers to the rhythmic, pulse-like propagation of data through a grid of processing elements (PEs), similar to the heart-beat. This design enables massive parallelism while minimizing external memory access, making it highly suitable for applications in linear algebra, deep learning accelerators, and signal processing.

The key principle behind systolic arrays is that each PE performs a simple computation—typically a multiply-accumulate (MAC)—and forwards its inputs to neighboring PEs. This allows each element of the input matrices to be reused across multiple PEs, reducing memory bandwidth requirements. Additionally, the regular, local communication between PEs makes the array highly scalable and amenable to pipelining.

### 6.1 Mathematical Background and Efficiency

Consider the multiplication of two matrices,  $A \in R^{N \times M}$  and  $B \in R^{M \times K}$ , producing a matrix  $C \in R^{N \times K}$ :

$$C_{ij} = \sum_{k=0}^{M-1} A_{ik} \cdot B_{kj}, \quad 0 \leq i < N, \quad 0 \leq j < K \quad (8)$$

A naive implementation would require fetching elements of  $A$  and  $B$  from memory for each multiplication, leading to high memory bandwidth and latency. Systolic arrays mitigate this by streaming the elements of  $A$  across rows and elements of  $B$  across columns. Each PE receives one element from  $A$  and one from  $B$ , performs a multiplication, adds it to an internal accumulator, and forwards the operands to neighboring PEs.

Mathematically, the computation within a PE can be expressed as:

$$c_{ij}^{(k)} = c_{ij}^{(k-1)} + a_{ik} \cdot b_{kj} \quad (9)$$

where  $c_{ij}^{(k)}$  is the partial sum after processing the  $k$ -th element of the summation. By distributing the summation across a 2D grid of PEs and pipelining the inputs, the systolic array achieves:

1. **Parallelism:** Each PE operates concurrently, producing one partial sum per clock cycle.
2. **Data Locality:** Operands move only to immediate neighbors, minimizing global memory access.
3. **Pipelining:** Once the array is filled, one output per PE per clock cycle can be produced continuously.

This structure allows for high throughput and energy efficiency, especially for large matrices.

## 6.2 GEMM Implementation

The general matrix multiplication (GEMM) is implemented using a 2D grid of PEs. Each PE maintains a local accumulator, performs a multiply-accumulate operation, and forwards the operands to the next PE in its row or column. The top-level module organizes the PEs to multiply an  $N \times M$  matrix  $A$  with an  $M \times K$  matrix  $B$  to produce  $C \in R^{N \times K}$ .

**Processing Element (PE)** Each PE implements a multiply-accumulate operation:

```
module pe(
    input clk,
    input rst,
    input [7:0] a_in,
    input [7:0] b_in,
    output reg [7:0] a_out,
    output reg [7:0] b_out,
    output reg [15:0] c_out
);
    reg [15:0] acc;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            a_out <= 0;
            b_out <= 0;

```

```

        c_out <= 0;
        acc  <= 0;
    end else begin
        a_out <= a_in;
        b_out <= b_in;
        acc  <= acc + a_in * b_in;
        c_out <= acc;
    end
end
endmodule

```

The PE is fully pipelined: ‘a\_in’ and ‘b\_in’ are forwarded to the next PE while the internal accumulator updates the partial sum. This local accumulation avoids repeated memory accesses for intermediate sums.

**Top-Level Systolic Array** The PEs are arranged in an  $N \times K$  grid, enabling the computation of all  $C_{ij}$  elements in parallel. Input streams are delayed appropriately using registers to align operands across the array. The top-level module can be parameterized for any matrix dimensions  $N$ ,  $M$ , and  $K$ :

```

module top #(
    parameter N = 4,
    parameter M = 4,
    parameter K = 4
)(
    input clk,
    input rst,
    input [7:0] a_in [0:N-1][0:M-1],
    input [7:0] b_in [0:M-1][0:K-1],
    output [15:0] c_out [0:N-1][0:K-1]
);
    // Instantiate PEs in an NxK grid
    // Forward a_in along rows, b_in along columns
    // Each PE computes partial sums
endmodule

```

The testbench only sequences the input matrices into the array and observes the output. Because the computation is fully pipelined, the testbench logic is minimal and does not introduce additional complexity. Its primary function is to verify correctness for different input sizes.

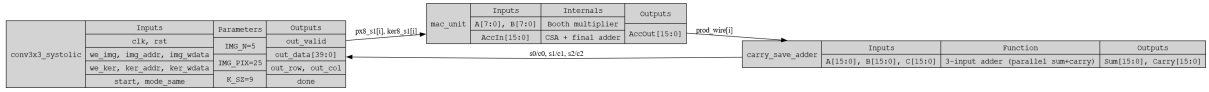
This GEMM implementation using systolic arrays achieves high throughput and energy efficiency by exploiting parallelism, pipelining, and local communication. The parameterized design allows scaling to arbitrary  $N \times M \times K$  matrices, making it suitable for hardware accelerators where matrix multiplication dominates computation time.

### 6.3 Convolutional Layer Implementation (Conv2D)

A 2D convolution operation computes a weighted sum over a sliding window of the input image. Mathematically, for an input image  $I \in R^{N \times N}$  and a kernel  $K \in R^{K \times K}$ :

$$O_{i,j} = \sum_{m=0}^{K-1} \sum_{n=0}^{K-1} I_{i+m,j+n} \cdot K_{m,n}, \quad 0 \leq i, j < N - K + 1 \quad (10)$$

Two common convolution modes are *valid* (no padding, output smaller than input) and *same* (zero-padding, output same size as input).



**Systolic Array Mapping and Inner Modules** The 3x3 convolution is mapped to a systolic structure composed of 9 **processing elements (PEs)**, each being a multiply-accumulate (MAC) unit. Every PE takes one pixel and one kernel weight, computes a product, and contributes it to the convolution sum. These 9 products are then aggregated through a tree of **carry-save adders (CSAs)** to produce the final convolution output.

- **Processing Element (PE):** Each PE is realized as a *MAC unit* that accepts an 8-bit signed pixel ( $A$ ), an 8-bit signed kernel coefficient ( $B$ ), and a 16-bit accumulator input. Internally, it uses:
  1. A **Booth multiplier** for signed  $8 \times 8$  multiplication, producing a 16-bit product.
  2. A **carry-save adder (CSA)** to add the product, the accumulator input, and a zero.
  3. A final **carry-propagate adder** that combines the CSA's sum and shifted carry outputs, generating the accumulated result.

This design minimizes critical path delay while allowing pipelining of MAC results.

- **CSA Tree for Accumulation:** Since 9 MAC units operate in parallel, their outputs must be combined efficiently. A three-level CSA tree is employed:

$$\begin{aligned} (P_0, P_1, P_2) &\rightarrow (s_0, c_0) \\ (P_3, P_4, P_5) &\rightarrow (s_1, c_1) \\ (P_6, P_7, P_8) &\rightarrow (s_2, c_2) \end{aligned}$$

The final result is computed as:

$$Out = (s_0 + s_1 + s_2) + ((c_0 + c_1 + c_2) \ll 1)$$

This reduces delay compared to a single 9-input adder.

- **Pipeline Registers:**

1. **Stage0:** Holds sampled 16-bit pixel values ('px16\_s0') from the input image.
2. **Stage1:** Truncates pixel and kernel values to 8 bits for MACs ('px8\_s1', 'ker8\_s1').
3. **Stage2:** Registers MAC outputs ('prod\_s2') before feeding them to the CSA tree.

**Pipeline Flow and Data Alignment** The convolution pipeline operates as follows:

1. **Sampling Window:** The FSM scans the input image, extracting a 3x3 window for each output pixel. For *same* convolution, padding is applied by inserting zeros where indices go out of bounds.
2. **Kernel Flipping:** The kernel is flipped prior to computation to satisfy the convolution definition.
3. **Stage Propagation:**
  - Pixels from 'px16\_s0' are truncated to 8-bit values and forwarded to MAC units in Stage1.
  - MAC outputs ('prod\_wire') are registered into 'prod\_s2' in Stage2 to align with the CSA pipeline.
4. **Valid Alignment:** A 3-bit shift register ('valid\_pipe') tracks the propagation of valid data through the stages. When 'valid\_pipe[2]' is high, the output corresponds to a fully computed convolution value.
5. **Output Counters:** 'out\_row\_cnt' and 'out\_col\_cnt' track the spatial position of the output pixel. These counters advance only when an output is emitted.

**Verilog Implementation Overview** The design instantiates:

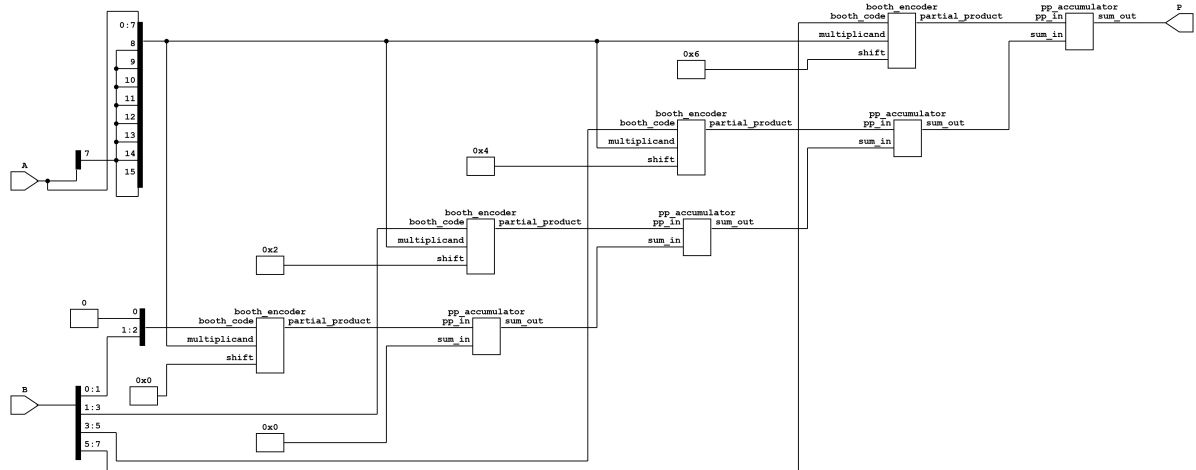
- **9 MAC units**, each forming a PE.
- **3 CSAs**, reducing 9 partial products into final sum.
- **FSM + pipeline control logic**, ensuring continuous throughput, flush handling, and SAME/VALID mode support.

## Key Architectural Features

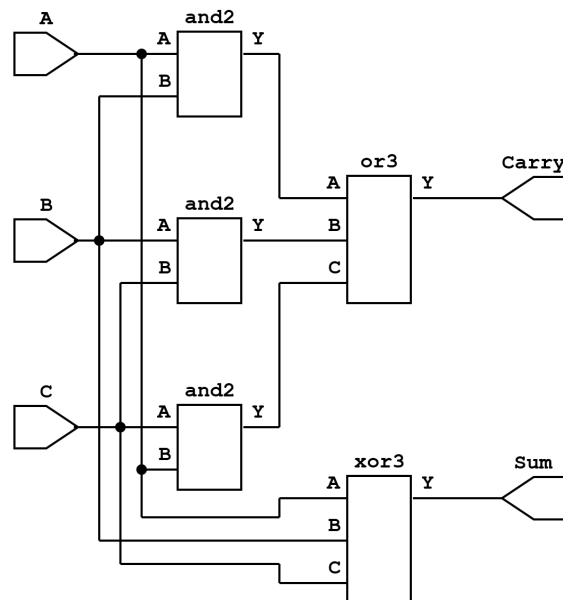
- **PE-level parallelism:** All 9 multiplications happen simultaneously.
- **MAC architecture:** Booth multiplier + CSA + final adder reduces critical path delay.
- **CSA accumulation:** Parallel reduction tree for 9 products avoids long adder chains.
- **Stage-by-stage pipelining:** Continuous data flow, one output per cycle after pipeline fill.
- **Flush support:** Pipeline always propagates valid signals until the last output is produced.
- **Flexible convolution modes:** SAME with zero-padding, or VALID without padding.

## 6.4 Yosys Netlists of Modules

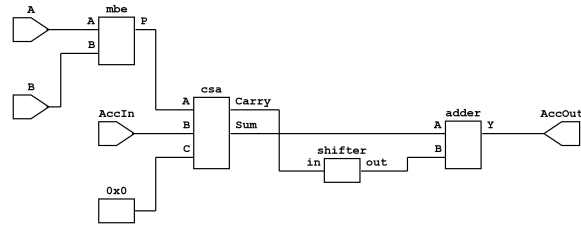
To validate the register-transfer level design and visualize the datapath structure, Yosys was used to generate netlists for the major building blocks. These netlists illustrate how arithmetic primitives and processing elements are interconnected within the systolic arrangement.



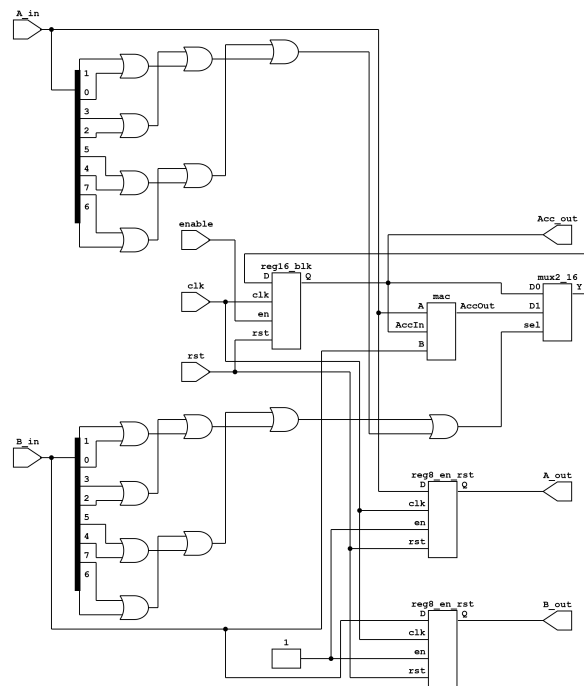
Netlist of the Booth multiplier block. The partial product generation and encoding stages are visible in the datapath.



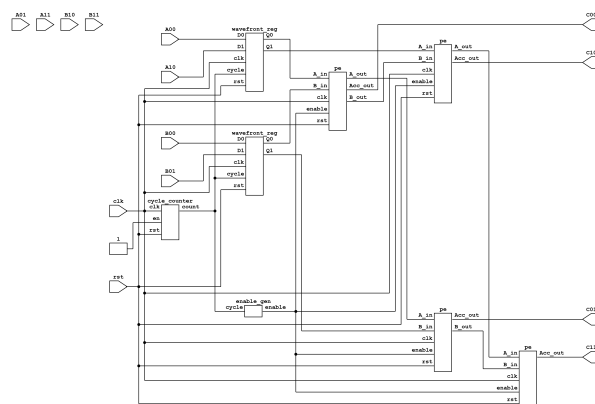
Netlist of the carry-save adder (CSA). The three-input adder structure reduces multiple partial results into sum and carry outputs.



Netlist of a multiply-accumulate (MAC) unit. It integrates the Booth multiplier with a CSA and final adder.



Netlist of a processing element (PE). The PE encapsulates one MAC unit along with local registers for systolic data movement.



Netlist of the complete systolic array arrangement. Multiple PEs are interconnected to form a parallel datapath for matrix-style computation.



## 6.5 Comparison of MAC Unit Architectures

In high-throughput accelerators such as systolic arrays for GEMM and Conv2D, the performance of individual MAC units cannot be considered in isolation. When multiple MAC units are paired or arranged in a 2D array, additional waiting cycles occur due to pipeline alignment, data propagation, and carry-save accumulation. Consequently, the latency of a combined MAC pair or array is always higher than the latency of a single MAC unit, while throughput may be limited by interconnect delays and output readiness. This subsection analyzes both adder and multiplier architectures to highlight these effects.

### 6.5.1 8-bit Signed Adders

Detailed Comparison of 8-bit Signed Adders

| Metric                                       | CSA           | Kogge–Stone   | RCA           |
|----------------------------------------------|---------------|---------------|---------------|
| Core Area ( $\mu\text{m}^2$ )                | 2,522         | 3,689         | 1,126         |
| Die Area ( $\mu\text{m}^2$ )                 | 4,604         | 6,093         | 2,572         |
| Utilization (%)                              | 54.79         | 60.54         | 43.79         |
| Flip-Flops                                   | 0             | 0             | 0             |
| Total Cells                                  | 64            | 110           | 34            |
| Combinational Cells                          | 64            | 110           | 34            |
| Clock Period (ns)                            | 10.00         | 10.00         | 10.00         |
| Critical Path (ns)                           | 5.07          | 6.21          | 7.14          |
| Fmax (MHz)                                   | 197.24        | 161.03        | 140.06        |
| Total Power (mW)                             | 0.0834        | 0.0957        | 0.0326        |
| Energy per Cycle (pJ)                        | 0.834         | 0.957         | 0.326         |
| Latency (ns)                                 | 5.07          | 6.21          | 7.14          |
| Throughput (ops/s)                           | 197,238,700   | 161,030,600   | 140,056,000   |
| Energy per Operation (pJ)                    | 422.84        | 594.30        | 232.76        |
| Power Efficiency (ops/s per mW)              | 2,364,972,000 | 1,682,660,000 | 4,296,197,000 |
| Area Efficiency (ops/s per $\mu\text{m}^2$ ) | 42,839.65     | 26,429.79     | 54,459.20     |

#### Observations:

- CSA achieves the shortest critical path (5.07 ns) and highest Fmax (197 MHz) due to parallel partial sum computation.
- RCA is the most compact and energy-efficient, but its simple ripple structure results in the slowest performance.
- Kogge–Stone has high speed theoretically, but wire-driven delays and dense prefix logic increase area and practical delay.

### 6.5.2 8-bit Signed Multipliers

Detailed Comparison of 8-bit Signed Multipliers

| Metric                                       | MBE         | Booth       | Baugh–Wooley |
|----------------------------------------------|-------------|-------------|--------------|
| Core Area ( $\mu\text{m}^2$ )                | 9,594       | 14,369      | 11,911       |
| Die Area ( $\mu\text{m}^2$ )                 | 13,094      | 18,687      | 15,827       |
| Utilization (%)                              | 73.27       | 76.89       | 75.26        |
| Flip-Flops                                   | 0           | 0           | 0            |
| Total Cells                                  | 294         | 470         | 374          |
| Combinational Cells                          | 294         | 470         | 374          |
| Clock Period (ns)                            | 10.00       | 10.00       | 10.00        |
| Critical Path (ns)                           | 8.84        | 12.50       | 8.63         |
| Fmax (MHz)                                   | 113.12      | 80.00       | 115.87       |
| Total Power (mW)                             | 0.379       | 0.701       | 0.448        |
| Energy per Cycle (pJ)                        | 3.79        | 7.01        | 4.48         |
| Latency (ns)                                 | 8.84        | 12.50       | 8.63         |
| Throughput (ops/s)                           | 113,122,200 | 80,000,000  | 115,874,900  |
| Energy per Operation (pJ)                    | 3,350.36    | 8,762.50    | 3,866.24     |
| Power Efficiency (ops/s per mW)              | 298,475,400 | 114,122,700 | 258,649,200  |
| Area Efficiency (ops/s per $\mu\text{m}^2$ ) | 8,639.16    | 4,281.08    | 7,321.29     |

#### Observations:

- Baugh–Wooley achieves the shortest critical path (8.63 ns) and highest Fmax (115.9 MHz) due to its regular array layout.
- Booth multiplier (Radix-2) suffers from high latency (12.5 ns) and power consumption (0.701 mW) because of additional partial product handling.
- MBE provides the best energy efficiency and balanced area utilization, making it optimal for paired MAC deployments.

### 6.5.3 Summary of MAC Unit Efficiency

Overall MAC Component Efficiency Summary

| Category           | Adders Best              | Multipliers Best   |
|--------------------|--------------------------|--------------------|
| Speed / Fmax       | CSA                      | Baugh–Wooley / MBE |
| Power              | RCA                      | MBE                |
| Area               | RCA                      | MBE                |
| Overall Efficiency | RCA (compact, efficient) | MBE (balanced)     |

As seen in the previous section, we have selected the CSA and MBE as the final adder–multiplier pair for our design. CSA provides significantly better speed with only a modest increase in area, while MBE offers a balanced trade-off between speed, power, and area, making the combination optimal for overall MAC efficiency

**Key Insight:** While these tables reflect the isolated performance of individual adders and multipliers, the effective latency of a MAC array in GEMM or Conv2D accelerators is higher due to pipeline propagation and waiting cycles. Similarly, throughput is influenced by data alignment and CSA accumulation stages, highlighting that array-level performance always differs from single-unit metrics.

## 6.6 Timing Estimate (MODEL\_ARCH\_8)

**Assumptions.** The following simplifying assumptions are made for estimating the latency of one forward inference:

1. The systolic convolution engine computes one convolution output (one spatial location  $\times$  one output channel) per clock cycle in steady state.
2. A small fixed overhead of approximately 3 cycles per kernel is incurred (kernel flip and pipeline flush).
3. Elementwise residual adds and global pooling reductions are implemented as 1 cycle per arithmetic operation.
4. The fully connected (Dense) layer is implemented with one multiply-accumulate per cycle.
5. No additional stalls are assumed from memory bandwidth or control logic.
6. Clock frequency is treated as a variable ( $f$ ), and estimates are shown for 50, 100, 200, and 250 MHz.

### Output counts per layer.

$$\begin{aligned}\text{conv2d } (32 \times 32 \times 28) &: 32 \cdot 32 \cdot 28 = 28,672 \\ \text{conv2d\_1 } (32 \times 32 \times 28) &: 28,672 \\ \text{conv2d\_2 } (32 \times 32 \times 28) &: 28,672 \\ \text{conv2d\_3 } (16 \times 16 \times 56) &: 16 \cdot 16 \cdot 56 = 14,336 \\ \text{conv2d\_4 } (16 \times 16 \times 56) &: 14,336 \\ \text{conv2d\_5 } (16 \times 16 \times 56) &: 14,336\end{aligned}$$

$$\text{Total convolution outputs} = 28,672 \times 3 + 14,336 \times 3 = 129,024$$

### Cycle accounting.

$$\begin{aligned}\text{Base convolution cycles} &= 129,024 \\ \text{Per-kernel overhead} &= 252 \times 3 = 756 \\ \text{Residual adds} &= (32 \cdot 32 \cdot 28) + (16 \cdot 16 \cdot 56) \\ &= 28,672 + 14,336 = 43,008 \\ \text{Global average pooling} &= 56 \cdot 63 = 3,528 \\ \text{Dense } (56 \rightarrow 10) &= 560\end{aligned}$$

$$\text{Total cycles} = 129,024 + 756 + 43,008 + 3,528 + 560 = 176,876$$

**Wall-clock latency.** Latency is given by

$$T = \frac{\text{cycles}}{f}.$$

| Clock frequency $f$ | Period | Latency $T$                               |
|---------------------|--------|-------------------------------------------|
| 50 MHz              | 20 ns  | $176,876/50 \times 10^6 \approx 3.54$ ms  |
| 100 MHz             | 10 ns  | $176,876/100 \times 10^6 \approx 1.77$ ms |
| 200 MHz             | 5 ns   | $176,876/200 \times 10^6 \approx 0.88$ ms |
| 250 MHz             | 4 ns   | $176,876/250 \times 10^6 \approx 0.71$ ms |

Estimated latency per inference.

## 6.7 GDS and Layout Visualizations for Adders and Multipliers

### 6.7.1 Adder Visualizations

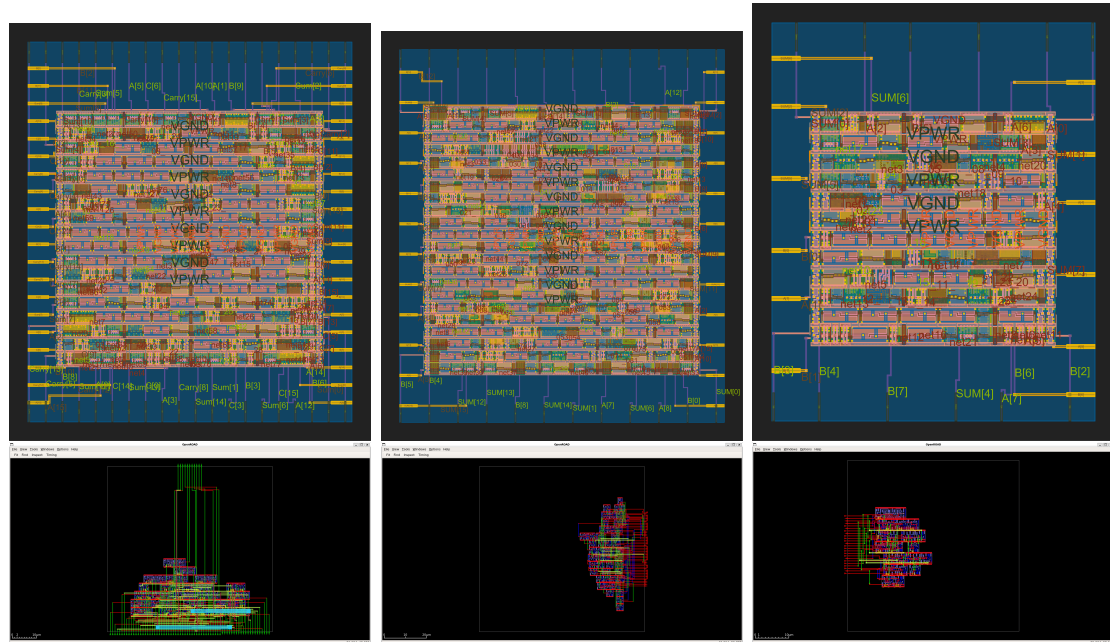


Figure 1: Adder GDS (top) and Layout (bottom) visualizations.

### 6.7.2 Multiplier Visualizations

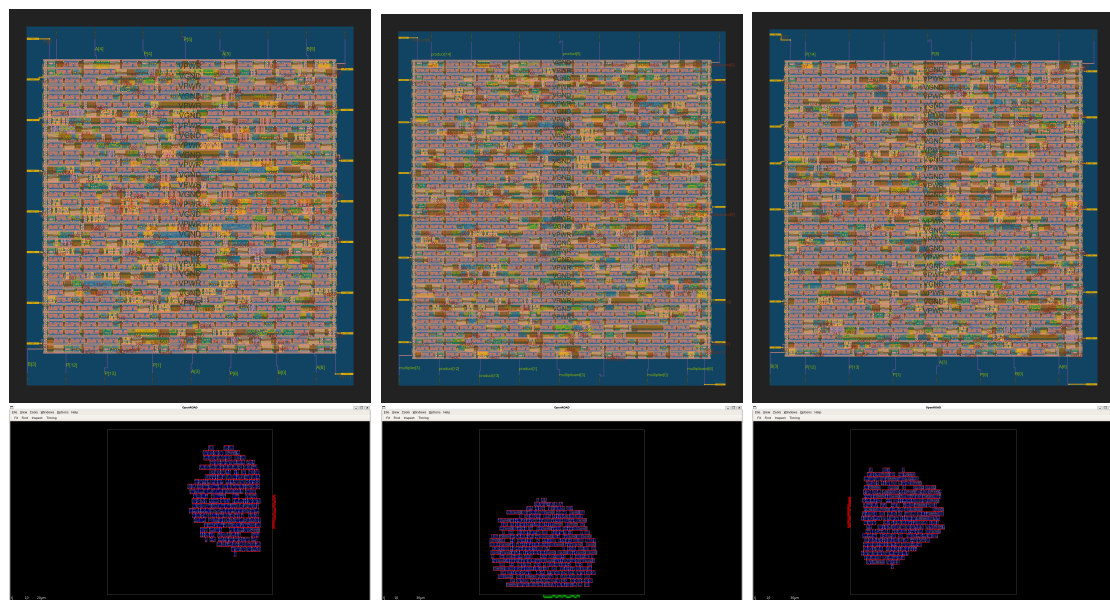


Figure 2: Multiplier GDS (top) and Layout (bottom) visualizations.

## Note on Environment and Verification Setup

All synthesis and verification experiments were carried out in a consistent flow, using the same TCL scripts and Yosys synthesis scripts. This uniform environment ensures that results are directly comparable without any tool-induced bias. Each RTL module was verified against its dedicated testbench, and post-synthesis reports confirmed that all designs met the specified timing requirements.

For technology mapping, only the **SkyWater 130 nm (Sky130) open-source PDK** was used. This PDK is provided through a collaboration between Google and SkyWater Technology, enabling open-source access to a fully characterized standard-cell library. The Sky130 platform allows digital design, synthesis, place-and-route, and verification flows to be tested on a real CMOS technology node.

Specifically, the standard-cell library used for synthesis was:

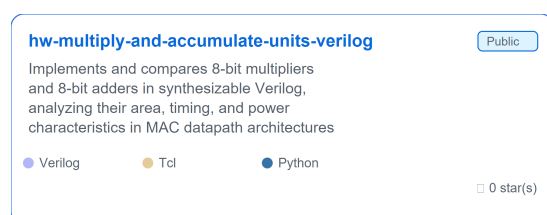
`sky130_fd_sc_hs__tt_025C_1v80.lib`

Breaking down the naming convention:

- **sky130** – Refers to the SkyWater 130 nm technology node.
- **fd** – Denotes the "foundry" designation.
- **sc** – Standard-cell library.
- **hs** – High-speed (HS) flavor of the library, optimized for performance at the cost of area and power.
- **tt** – Typical process corner (*typical NMOS, typical PMOS*), representing nominal manufacturing conditions.
- **025C** – Temperature corner at 25°C, i.e., room temperature.
- **1v80** – Operating supply voltage of 1.80 V.

`sky130_fd_sc_hs__tt_025C_1v80.lib` represents a standard-cell library characterized for the Sky130 process, using the high-speed cell set, under typical conditions at 25°C and 1.8 V supply. This ensures that synthesis, timing analysis, and verification are carried out on a realistic and industry-compatible open-source technology.

For more details, refer to this:



## 7 References & Listings

1. CIFAR-10 Dataset: <https://www.cs.toronto.edu/~kriz/cifar.html>
2. GeeksforGeeks: Convolutional Neural Network in Machine Learning: <https://www.geeksforgeeks.org/deep-learning/convolutional-neural-network-cnn-in-machine-learning/>
3. TensorFlow Model Optimization: Quantization Guide: [https://www.tensorflow.org/model\\_optimization/guide/quantization/training\\_comprehensive\\_guide](https://www.tensorflow.org/model_optimization/guide/quantization/training_comprehensive_guide)
4. Fixed-Point Representation (Q-format): <https://www.allaboutcircuits.com/technical-articles/fixed-point-representation-the-q-format-and-addition-examples/>
5. Lecture Slides on Adders and Multipliers (Auburn University): [https://www.eng.auburn.edu/~uguin/teaching/E4200\\_Spring\\_2021/lecture-slides/Lecture-7-Adders-Multipliers.pdf](https://www.eng.auburn.edu/~uguin/teaching/E4200_Spring_2021/lecture-slides/Lecture-7-Adders-Multipliers.pdf)
6. Yosys Open SYnthesis Suite Documentation: <https://yosyshq.readthedocs.io/projects/yosys/en/latest/>
7. OpenLane2 Documentation: <https://openlane2.readthedocs.io/en/latest/>

All working code, along with multiple variants, versions, and complete documentation, is available in the following repositories. Each repository includes relevant explanations, and for detailed documentation please refer to:

[mummanajagadeesh.github.io/projects/improve/subprojects/](https://github.com/mummanajagadeesh/projects/improve/subprojects/)

The image displays four GitHub repository cards arranged in a 2x2 grid. Each card shows the repository name, a brief description, a list of supported languages/tools, and the star count.

- VISION** (Public): ViSION - Verilog for Image processing and Simulation-based Inference Of Neural Networks. 0 star(s). Languages: Verilog, Perl, Shell.
- NeVer** (Private): NEural Network on VERilog | MLP for (E)MNIST | CNN for CIFAR10. 1 star(s). Languages: Verilog, Jupyter, Notebook, Python, Perl, Tcl, C++, SystemVerilog, Shell, Makefile, Batchfile, CMake.
- systolic-array-matrix-multiplication** (Private): Systolic array of MAC PEs with Booth multipliers and carry-save adders, supporting both GEMM and 3x3 CNN convolutions for hardware-accelerated deep learning and linear algebra. 0 star(s). Languages: Verilog, SystemVerilog, Shell.
- hw-multiply-and-accumulate-units-verilog** (Public): Implements and compares 8-bit multipliers and 8-bit adders in synthesizable Verilog, analyzing their area, timing, and power characteristics in MAC datapath architectures. 0 star(s). Languages: Verilog, Tcl, Python.



## Additional Repositories:

**cordic-algorithm-verilog** Private  
Verilog-based implementation of the CORDIC algorithm for efficient estimation of mathematical functions  
● Verilog  
1 star(s)

**ImProVe** Private  
Image processing using Verilog  
● Verilog ● Python ● Roff ● HTML  
● V  
1 star(s)

## 8 Demo Video

