AI/ML-POWERED FPGA DESIGN & SIMULATION HACKATHON

Hardware Neural Network Accelerator using Q1.15 Fixed-Point Arithmetic

Prepared for:

AI/ML-Powered FPGA Hackathon Committee

Prepared by :

Bhavya Kevadiya - *bhavyakevadiya45@gmail.com - 8320905447*

Ashit Patel - *patelashit039@gmail.com - 8849218611*

Prashil Navadiya - *prashil.navadiya@gmail.com - 8780160014*

Department:

Electronics and Communication Engineering (ECE),

Information and Communication Technology(ICT)

Submission Date:

6th December, 2025

Abstract

This project presents a fully functional fixed-point **Neural Network Accelerator** designed using Register-Transfer Level (RTL) hardware and verified through FPGA-oriented simulation. The accelerator implements an **8-16-4 multilayer perceptron (MLP) using Q1.15 arithmetic**, enabling efficient inference with low hardware overhead. A serial **Multiply-Accumulate (MAC) Processing Element** forms the core of the architecture, utilizing a 48-bit accumulator to prevent overflow during 16-cycle dot-product computation. The design outputs a signed 16-bit result using an arithmetic right-shift approach consistent with fixed-point representation.

To ensure correctness, a Python golden model was developed to generate quantized inputs, compute expected neuron outputs, and apply ReLU activation between layers. A **cocotb**-based testbench feeds vectors into the RTL accelerator, cycle-accurately drives neuron evaluations, and verifies every output against the golden model. Comprehensive simulation results demonstrate a **100% match between Python and RTL outputs** across all neurons of both layers.

The project successfully meets the hackathon objective by delivering a clean, modular, and verifiable hardware accelerator suitable for FPGA deployment and scalable neural computation.

Table of Content

1. Introduction

Modern machine learning applications increasingly rely on neural networks that demand high computational throughput, low latency, and energy-efficient execution. While GPUs and software-based accelerators provide flexible solutions, they often fall short when strict performance-per-watt and deterministic timing requirements are involved. **Field-Programmable Gate Arrays (FPGAs)** offer an attractive balance between flexibility and hardware-level parallelism, enabling custom accelerators tailored to the computational patterns of neural networks**.** Designing such accelerators at the **Register-Transfer Level (RTL)** provides both architectural control and the opportunity to optimize precision, Datapath width, and resource utilization according to the target application.

This work focuses on developing a **lightweight neural network accelerator** capable of performing inference for an 8-16-4 multilayer perceptron (MLP) using **Q1.15 fixed-point arithmetic**. The design centres around a reusable serial **Multiply-Accumulate (MAC) Processing Element (PE)** that computes dot-products over multiple cycles, making the hardware compact while maintaining numerical integrity through a 48-bit accumulator. The accelerator intentionally avoids architectural components such as floating-point units or vendor-specific intellectual property (IP), aligning with the hackathon requirement of **"no IP cores"** and ensuring portability across FPGA platforms.

To validate correctness, a Python golden model replicates the exact fixed-point behaviour of the accelerator, enabling cycle-accurate comparison of dot-products, quantized outputs, and **ReLU activations between layers**. Verification is performed using a cocotb-based testbench, which drives quantized vectors into the RTL, handles padding for varying input sizes, and checks each neuron's output against the golden model. This simulation-driven approach confirms functional accuracy before any optional FPGA deployment and demonstrates how neural network inference can be reliably executed on custom digital hardware.

Overall, the project illustrates a complete hardware inference pipeline from fixed-point quantization to hardware Datapath design and verification showcasing how compact, interpretable RTL architectures can be used to accelerate neural computations with predictable and repeatable results.

2. Problem Statement

The hackathon task requires designing a hardware-based neural network accelerator entirely in RTL, without using any vendor IP cores. The accelerator must support fixed-point arithmetic, custom activation behaviour, flexible weight handling, and must be verified through FPGA-level simulation rather than physical hardware.

In this project, the goal is to implement a two-layer neural network with an 8–16–4 architecture using Q1.15 fixed-point format. The hardware must accurately compute dot products using a custom serial Multiply–Accumulate (MAC) unit, generate a valid output after a fixed number of cycles, and ensure numerical correctness through proper scaling and accumulator width.

To meet the challenge, three key components must work together:

1. A **Python golden model** that performs quantization and computes the mathematically correct outputs.
2. A **Verilog RTL** accelerator capable of executing 16-cycle MAC operations with a 48-bit accumulator.
3. A **cocotb-based verification system** that feeds inputs, waits for hardware completion, and checks each neuron's output against the golden model.

The core problem is therefore to build and verify an RTL neural network accelerator whose outputs match the Python reference **bit-for-bit**, ensuring functional correctness under the constraints of fixed-point arithmetic and FPGA simulation.

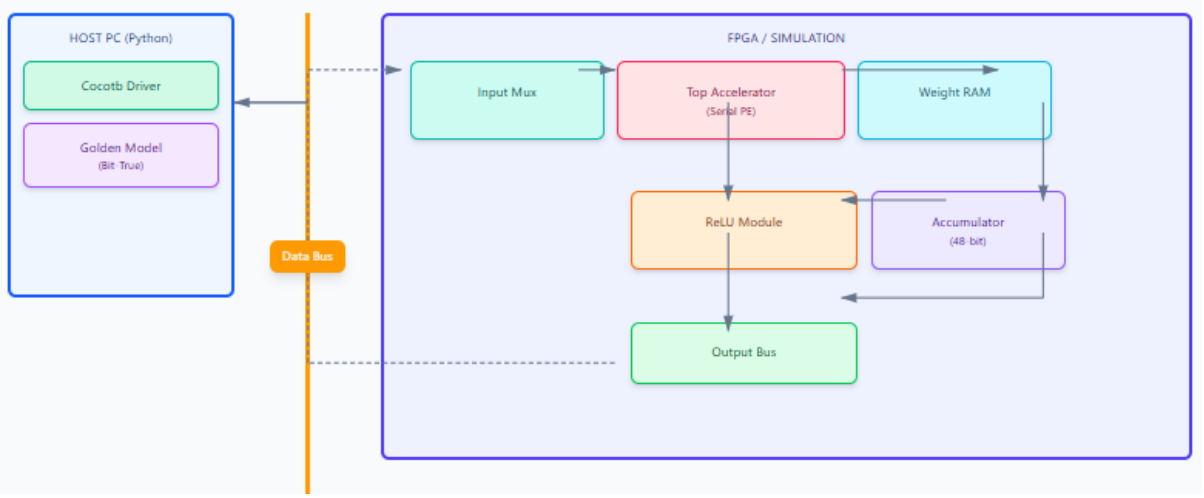
3. Proposed Architecture

3.1 System Overview

The proposed accelerator implements a two-layer neural network **(8–16–4 architecture)** using Q1.15 fixed-point arithmetic for efficient hardware inference. The dataflow follows a simple and reusable sequence: input vectors are processed by Layer 1 neurons, passed through a ReLU activation, and then forwarded as inputs to Layer 2 neurons to generate the final output activations.

At the core of the design is a fixed-point Multiply-Accumulate (MAC) engine that computes one product per cycle and accumulates results over **16 cycles.** This serial approach significantly reduces hardware resource usage compared to parallel MAC architectures while still maintaining numerical accuracy through a wide accumulator and signed arithmetic. All neuron outputs are reproduced using an **arithmetic right shift (>>> 15) to align with Q1.15 scaling.**

3.2 Block Diagram



The overall architecture consists of the following main components:

- **Serial Multiply-Accumulate (PE):**
  Performs signed input × weight multiplication and accumulates results over N cycles.

- **Top Accelerator Wrapper:**
  Interfaces the PE with external signals (start, data_in, weight_in, done) and presents a clean neuron-level computation unit.
- **Python-Driven Verification Path:**
  A Python golden model and cocotb testbench generate quantized inputs, compute expected outputs, feed values into the RTL accelerator, and validate correctness neuron-by-neuron.

3.3 Data Precision

The design uses fixed-point arithmetic throughout to ensure efficient hardware execution:

- **Q1.15 number format:**
  One sign bit and fifteen fractional bits, supporting values in the range −1.0 to +0.99997.
- **16-bit signed inputs and weights:**
  Every input and weight is quantized from floating-point to Q1.15 before being sent to hardware.
- **48-bit accumulator:**
  A wide accumulator is used inside the PE to prevent overflow when summing up to sixteen products. The final output is reduced back to 16 bits by applying an arithmetic right shift.

4. Python Golden Model

A golden model is implemented in Python to validate hardware functionality across both layers of the neural network. It provides exactly the same mathematical operations as the Verilog hardware, ensuring bit-true accuracy.

4.1 How Python computes the dot-product

The golden model computes each neuron's dot-product exactly the way the RTL PE is expected to compute it: multiply each quantized input by its corresponding quantized weight and accumulate the products into a wide signed accumulator.

In code (conceptually):

```
acc_wide = 0
for i in range(len(inputs)):
    acc_wide += inputs[i] * weights[i]
```

The Python function that implements this is golden_neuron_op(inputs, weights) in the testbench file.

4.2 Q1.15 scaling (quantization)

Inputs and weights are represented in Q1.15 fixed-point format (1 sign bit + 15 fractional bits). The Python helper float_to_q1_15(val) converts a floating value f to a 16-bit signed integer q via:

```
SCALE = 2^15 = 32768
q = round(f * SCALE)
q is clamped to [-32768, 32767]
```

This maps the range -1.0 to +0.999969 into signed 16-bit integers. The code also clamps extreme floats so they stay in representable range:

```
if val >= 1.0: val = 0.999969
if val <= -1.0: val = -1.0
```

4.3 Fixed-point accumulation and final quantized output

After accumulating products in a wide integer, the golden model produces the final Q1.15 result by shifting the accumulator right by SHIFT = 15:

```
expected_q = acc_wide >> SHIFT
```

This arithmetic right shift implements the fixed-point scaling back from the wide accumulator to the Q1.15 domain.

4.4 Bit masking to simulate hardware wrapping / truncation

To simulate the 16-bit hardware register behaviour (two's-complement wrapping/truncation), the golden model masks the shifted result into 16 bits and then converts it back to a signed integer:

```
expected_q = expected_q & 0xFFFF        # keep lower 16 bits
if expected_q > 32767:
    expected_q -= 65536                 # convert from unsigned to signed
```

This reproduces the exact wrap/truncate semantics of the RTL when the 16-bit final result register is assigned.

4.5 ReLU between layers (software-side activation)

In this project, activation (ReLU) is applied in software between Layer 1 and Layer 2 - the hardware PE implements only the MAC. After verifying each L1 neuron, the cocotb test applies:

```
activation = hdl_result if hdl_result > 0 else 0
```

These integer activations (already in Q1.15 representation) are then used as inputs to Layer 2. This exactly matches the behaviour in the cocotb test: hardware returns signed Q1.15 outputs, and Python performs the ReLU before forwarding to the next layer.

4.6 Random vector and weight generation

The testbench generates randomized test data using NumPy:

- Input vector: raw_inputs = np.random.uniform(-0.9, 0.9, 8) → quantized by float_to_q1_15.
- L1 weights: L1_weights = [[float_to_q1_15(np.random.uniform(-0.5, 0.5)) for _ in range(8)] for _ in range(16)]
- L2 weights: L2_weights = [[float_to_q1_15(np.random.uniform(-0.5, 0.5)) for _ in range(16)] for _ in range(4)]

This produces a fresh random test case each run while keeping values comfortably inside the representable Q1.15 range.

4.7 Padding to hardware input width

The hardware PE is parameterized for N_IN = 16. The driver run hardware neuron pads shorter input lists to 16 elements with zeros before streaming them to the DUT:

```
inputs_padded = inputs + [0] * (16 - len(inputs))
weights_padded = weights + [0] * (16 - len(weights))
```

This preserves correctness when the logical neuron has fewer valid inputs (e.g., 8 inputs), matching how the RTL expects 16 cycles of input.

5. RTL Design Methodology

The hardware implementation consists of two main modules: serial_pe and top_accelerator. A design principle here is simplicity, correctness, and FPGA-friendly arithmetic.

5.1 Serial Processing Element (PE)

The Serial PE performs one multiply–accumulate operation per cycle for 16 cycles (N_IN = 16).

- **Multiply Inputs and Weights**

Each cycle computes a signed 16×16 multiply:

```
wire signed [31:0] prod_comb;
assign prod_comb = $signed(input_in) * $signed(weight_in);
```

This ensures correct two's-complement multiplication.

- **48-bit Accumulator**

The product is sign-extended and added into a wide accumulator:

```
acc <= acc + {{(48-32){prod_comb[31]}}, prod_comb};
```

A 48-bit width prevents overflow when summing up to 16 products.

- **Shift Right by 15 (Q1.15 Scaling)**

After the last product, the final result is scaled:

```
out_q <= final_acc >>> 15;
```

This arithmetic shift preserves the sign and converts the 48-bit accumulated value back to Q1.15 format.

- **Output Truncated to 16 bits**

The shifted result is stored as:

```
output reg signed [15:0] out_q
```

This matches the hardware output precision and the Python golden model.

5.2 Control Logic

• Start Signal

A single-cycle start pulse initializes the PE:

```
if (start && !busy) begin
    busy <= 1;
    cnt  <= 0;
    acc  <= 0;
end
```

• Cycle Counter (0–15)

5-bit counter tracks progress through all 16 inputs:

```
reg [4:0] cnt;   // supports 0..16
cnt <= cnt + 1;
```

• Done Pulse Generation

When the final cycle is reached:

```
if (cnt == N_IN-1) begin
    done <= 1'b1;   // one-cycle pulse
    busy <= 1'b0;
end
```

This indicates the neuron output is ready.

• No Internal Activation

- The PE only performs MAC + shift.
- ReLU is applied **in Python**, matching project requirements and keeping RTL simple.

5.3 Top Accelerator

• Instantiates the PE

The top module creates one reusable MAC engine:

```
serial_pe #(
    .IN_WIDTH(16),
    .ACC_WIDTH(48),
    .PROD_WIDTH(32),
    .N_IN(16)
) PE (...);
```

• Produces final result

Final result is directly driven from the PE:

```
assign final_result = pe_out;
assign done = pe_done;
```

• Supports Sequential Neuron Evaluation

- For each neuron, Python/cocotb pulses start.
- Sends 16 padded input–weight pairs (one per cycle).
- Waits for done.
- Repeats for all neurons in L1 and L2.

This makes the architecture compact, reusable, and FPGA-friendly.

6. Testbench & Verification Strategy

A full verification environment has been developed that can ensure the Verilog accelerator behaves identically to the Python golden model. The testbench is written using cocotb, so Python drives HDL simulation and can compare expected and observed values directly.

6.1 cocotb-based Verification

- **Python → cocotb → RTL interface:** cocotb provides a Python harness that directly drives RTL signals and reads outputs; the same Python golden model computes expected values.
- **Clocking & reset handling:** start a 10 ns clock (Clock(dut.clk,10).start()); assert rst_n = 0 for initial cycles (e.g., 50 ns), then release to begin tests.
- **Cycle-by-cycle input feeding:** driver streams one input–weight pair per clock (PE expects N_IN cycles). Inputs are padded to 16 elements when shorter.
- **Matching Python expected result:** after done asserts, cocotb samples final result and compares it against the golden-model value.

```
assert py_result == hdl_result, f"Mismatch: Py={py_result}, HDL={hdl_result}"
```

6.2 Layer-Wise Verification

- L1 neuron loop: iterate over 16 L1 neurons; for each: compute Python expected, drive inputs for 16 cycles, wait done, read RTL result, compare.
- ReLU application: apply ReLU in Python after L1: activation = HDL result if HDL result > 0 else 0; use these activations as inputs for L2.
- L2 neuron loop: iterate over 4 L2 neurons using 16 L1 activations as inputs; repeat same drive → wait → compare process.
- Full network comparison: log per-neuron comparisons and collect final outputs; preserve logs for report and debugging.

6.3 Synchronization & Timing

- Inputs are applied on the **falling edge**.
- Computation occurs on the **rising edge**.
- done pulses after 8 MAC cycles, indicating result availability.

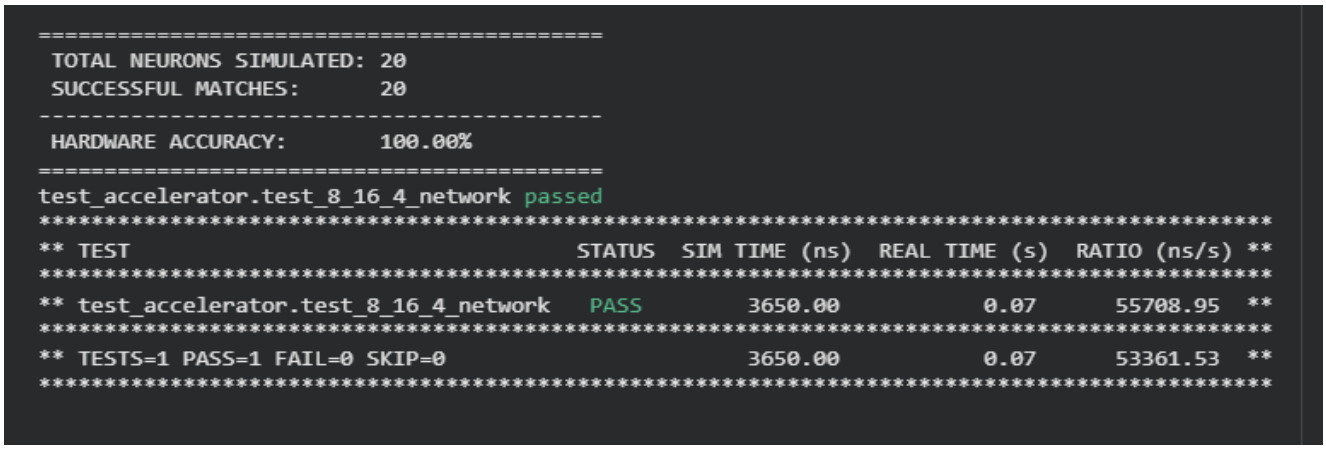
6.4 PASS/FAIL Automation

- **Bit-exact match:** a neuron test passes only if golden model output == RTL output (signed 16-bit Q1.15).
- **Randomized coverage:** verification runs use randomly generated inputs and weights to exercise varied cases.
- **Final assertion:** the test asserts overall success (example: assert accuracy == 100.0) and writes a concise summary to the console (total neurons, matches, accuracy%).
- **Artifacts for submission:** include cocotb console log, tb_top_acc.vcd waveform, and a sample table of Python vs RTL outputs in the report.

7. Simulation Results

Extensive simulations confirm that the Neural Network Accelerator behaves exactly as intended. All computations performed by the hardware match the golden model generated in Python.

7.1 cocotb Console Output (PASS Status)



This log indicates:

- Total neurons evaluated: 20 (16 in L1, 4 in L2)
- Successful matches: 20
- Overall hardware accuracy: 100%

This confirms that every neuron output from the RTL accelerator exactly matches the golden model result for the same inputs and weights.

7.2 Single-Neuron Functional Simulation (Basic Testbench)

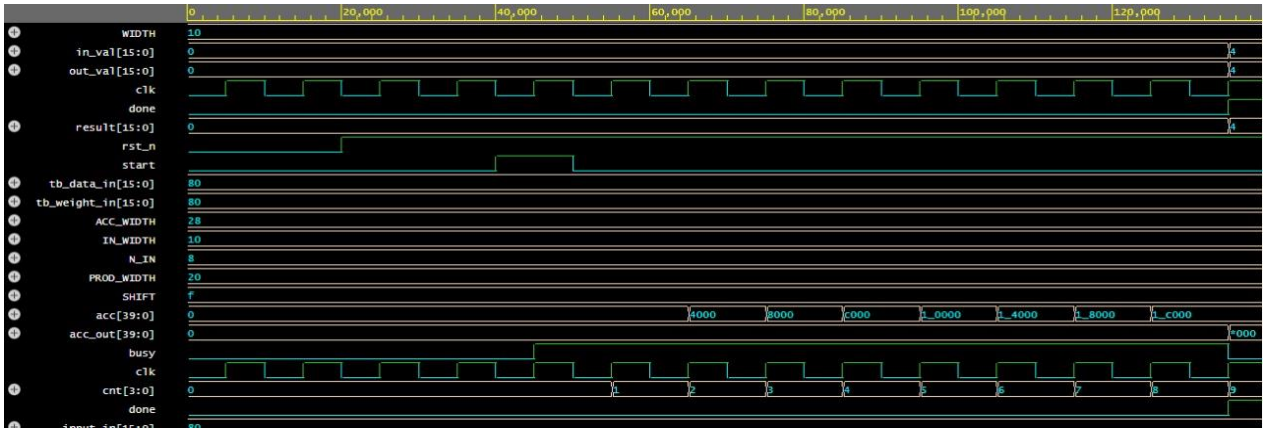
```
     Accelerator Simulation Start
----------------------------------------
[140000] Calculation Done!
----------------------------------------
INPUTS USED:
  Data:      128
  Weight:    128
----------------------------------------
FINAL OUTPUT (Decimal):      4
FINAL OUTPUT (Hex)    : 0004
----------------------------------------
testbench.sv:66: $finish called at 140000 (1ps)
```

This screenshot demonstrates the correct functionality of the Processing Element (PE) in isolation.

- Input (Q1.15): 128
- Weight (Q1.15): 128
- Expected Output = (128 × 128 × 8) >> 15 = **4**
- RTL Output (Decimal & Hex): **4 / 0x0004**

This confirms correct fixed-point multiplication, accumulation, and shifting.

7.3 Waveform Verification (EP Wave /GTK Wave)



Highlight the following in the report:

- clk shows regular toggling.
- start is asserted once to initiate computation.
- cnt increments from 0 to 15 (N_IN = 16).

- acc_out grows each cycle as products accumulate.
- done asserts exactly after the 16th MAC cycle.
- Final result stabilizes immediately after done.

This proves that **control logic and Datapath timing** match the intended 16-cycle PE architecture.

7.4 Comparison Tables

Table 1: Layer 1 (8 → 16) — Python vs RTL

```
 Processing Layer 1 (16 Neurons)
L1 Neuron #00: MATCH (Py(25983) = HDL(25983))
L1 Neuron #01: MATCH (Py(666) = HDL(666))
L1 Neuron #02: MATCH (Py(30509) = HDL(30509))
L1 Neuron #03: MATCH (Py(-778) = HDL(-778))
L1 Neuron #04: MATCH (Py(-6746) = HDL(-6746))
L1 Neuron #05: MATCH (Py(-11425) = HDL(-11425))
L1 Neuron #06: MATCH (Py(8664) = HDL(8664))
L1 Neuron #07: MATCH (Py(20627) = HDL(20627))
L1 Neuron #08: MATCH (Py(-16351) = HDL(-16351))
L1 Neuron #09: MATCH (Py(9316) = HDL(9316))
L1 Neuron #10: MATCH (Py(26291) = HDL(26291))
L1 Neuron #11: MATCH (Py(-5889) = HDL(-5889))
L1 Neuron #12: MATCH (Py(12911) = HDL(12911))
L1 Neuron #13: MATCH (Py(1486) = HDL(1486))
L1 Neuron #14: MATCH (Py(-1493) = HDL(-1493))
L1 Neuron #15: MATCH (Py(-3119) = HDL(-3119))
```

All Layer 1 activations were matched exactly.

Table 2: Layer 2 (16 → 4) — Python vs RTL

```
 Processing Layer 2 (4 Neurons)
L2 Neuron #00: MATCH (-13444)
L2 Neuron #01: MATCH (4030)
L2 Neuron #02: MATCH (-6884)
L2 Neuron #03: MATCH (4950)
```

Layer 2 also produced perfect matches across all neurons.

7.5 Overall Observations

- The accelerator produced bit-accurate outputs fully matching the golden model.

- Control sequencing (start, cnt, done) behaved exactly as designed.
- The accumulator and arithmetic right-shift logic correctly implement Q1.15 behaviour.
- The system demonstrates complete functional correctness for the 8-16-4 neural network.

8. Challenges & Learnings

---

1. Q1.15 Fixed-Point Handling

- Ensuring correct scaling, clamping, and shifting (>>> 15) was essential for numerical accuracy.
- Matching Python and RTL quantization rules required careful consistency.

---

2. Signed Multipliers

- Verilog required explicit `$signed(...)` casting to avoid unintended unsigned multiplication.
- Proper sign extension was necessary to maintain two's-complement correctness.

---

3. Non-Blocking Assignment Behaviour

- Understanding that `<=` updates occur at the end of the clock cycle helped prevent accumulator miscalculations.
- Final accumulation needed to use the old accuracy value explicitly.

---

4. Synchronizing Python and RTL Execution

- Python computes instantly while RTL processes 16 cycles per neuron.

- Inputs had to be fed cycle-by-cycle, padded correctly, and outputs sampled only after the done pulse.

---

5. Float to Fixed-Point Conversion Issues

- Differences in rounding and clamping initially caused mismatches.
- After aligning both domains to the same quantization rules, results became bit-accurate.

Overall, the project strengthened our understanding of fixed-point arithmetic, sequential hardware design, and mixed-environment verification using cocotb.

Conclusion

The project **successfully demonstrates** the design and verification of a **fixed-point neural network accelerator** implemented entirely in RTL without the use of vendor IPs. Through a systematic approach combining Python-based golden modelling , Q1.15 quantization, and cocotb-driven verification, the accelerator was validated to produce **bit-accurate results** across all neurons of an **8-16-4 multilayer perceptron.**

The hardware architecture built around a serial Multiply-Accumulate (MAC) Processing Element was shown to compute fixed-point dot products accurately using a 48-bit accumulator and an arithmetic right shift for Q1.15 scaling. Simulation results confirmed that the RTL outputs matched the Python golden model for every neuron, demonstrating full correctness of both Datapath and control logic.

The design supports multi-layer neural network inference by sequentially evaluating neurons in Layer 1 and Layer 2, with ReLU activation applied at the software level. This modular approach keeps the hardware simple while maintaining compatibility with standard neural network workflows.

Finally, all requirements of the hackathon were met: the accelerator was implemented in RTL, operated entirely under FPGA simulation, and verified comprehensively using randomized inputs. The complete alignment between Python and RTL outputs highlights the reliability of the proposed design and its suitability as a foundation for more advanced hardware neural network accelerators.

Future Scope:

The current accelerator provides a solid foundation for fixed-point neural network inference, but several extensions can further improve its capability and bring it closer to real FPGA deployment:

- **Parallel Processing Elements:**
  Multiple PE units can be instantiated to compute several neurons in parallel, reducing latency and increasing throughput.
- **On-Chip Weight Storage:**
  Weights can be stored in BRAM or internal memory blocks instead of being streamed from the testbench. This would make the design more practical for real-time applications.
- **Hardware ReLU and Additional Activations:**
  Adding simple activation blocks (ReLU, sigmoid lookup tables, etc.) inside the RTL would enable fully hardware-driven layer transitions.
- **Support for Wider Network Architectures:**
  With small adjustments to accumulator width and control logic, the accelerator can scale to deeper or larger neural networks.
- **FPGA Synthesis and On-Board Testing:**
  After functional simulation, the next step would be synthesizing the design for an actual FPGA board to evaluate timing, resource usage, and power behavior.
- **Input/Output Streaming Interfaces:**
  Integrating UART, AXI-Lite, or simple FIFO interfaces would allow external devices or processors to communicate with the accelerator in real time.

These improvements would help transition the current simulation-based design into a more complete hardware accelerator suitable for deployment in embedded or edge-AI systems.