FYP PROGRESS REPORT:RTL(STAGE 1)

MICRO-ARCHITECTURE OF 2-LAYER CNN PROTOTYPE

GROUP MEMBERS:
ISMAIL KHAN (El-22075)
EMAN JAVED (El-22050)
MUHAMMAD ZAID (El-22066)
SHAHMIR AMIR (El-22123)

1. Introduction

This report describes the RTL design of a 2-layer convolutional neural network (CNN) accelerator prototype. The design implements the first convolution layer for an 8×8 input image using a 3×3 kernel, ReLU activation, and storage of the resulting 6×6 feature map in on-chip memory. The aim is to demonstrate the core dataflow of a CNN layer in hardware, validate the architecture on FPGA, and provide a modular guide for team members to implement and test individual blocks.

2. Design Requirements

- **Input image**: 8×8 pixels (8-bit each).
- **Kernel**: 3×3 weights (8-bit each).
- **Compute**: 3 parallel MAC units, 3 cycles per patch.
- **Activation**: ReLU applied to each patch output.
- **Output**: 6×6 feature map stored in conv1_mem.
- **Interface**: AXI-Stream for pixel input.
- **Storage**:

DRAM → source of input image & weights.

Line buffer → temporary patch storage.

conv1_mem (SRAM/BRAM) → output feature map.

3. System Overview

The dataflow of the accelerator is:

1. Pixels stream in via AXI interface and fill the line buffer.

2. Once three rows are loaded, the line buffer outputs a 3×3 patch.
3. Controller FSM orchestrates patch processing by sending pixels and weights to the MAC engine.
4. MAC engine computes dot products in three cycles and updates the accumulator.
5. After accumulation, the result is captured, passed through ReLU, and registered.
6. Final value is written into conv1_mem at the correct address.
7. Controller shifts window across columns and loads new rows until the entire 6×6 feature map is generated.

4. Module Description

4.1 AXI Stream Input

- Provides pixel rows as 64-bit beats (8 pixels per beat).
- Signals: `tvalid`, `tdata[63:0]`, `tready`.
- Handshake ensures pixels are only written when both valid and ready.

4.2 Line Buffer

- Stores 3 rows of pixels and generates sliding 3×3 windows.
- Inputs: `tdata`, `shift_right`, `drop_row`.
- Outputs: `patch_valid`, `pix_out[71:0]`.
- Behavior: after filling 3 rows, asserts `patch_valid`. Shifts horizontally for new patches and vertically after row completion.

4.3 Weight Register File

- 9 registers for kernel weights (w0..w8).
- Loaded once before convolution begins.

4.4 Pixel & Weight Mux

- Controlled by `cycle_sel` from FSM.
- Selects groups of 3 pixels and 3 weights per cycle.
- Cycle 0 → (p0..p2, w0..w2), cycle 1 → (p3..p5, w3..w5), cycle 2 → (p6..p8, w6..w8).

4.5 MAC Engine

- Contains 3 multipliers and an adder to produce `sum3` each cycle.
- Input: 3 pixels + 3 weights.
- Output: partial sum of 3 products.

4.6 Accumulator

- Internal register `acc_reg` updated with `acc_reg + sum3` each cycle.
- Controls: `acc_reset`, `acc_enable`.
- After final cycle, FSM asserts `acc_capture` to transfer result into `out_reg`.

4.7 ReLU + Write Data Register

- Applies activation: `relu_out = max(0, out_reg)`.
- Result stored in `write_data_reg` and marked valid.
- FSM uses `out_valid` to trigger memory write.

4.8 conv1_mem

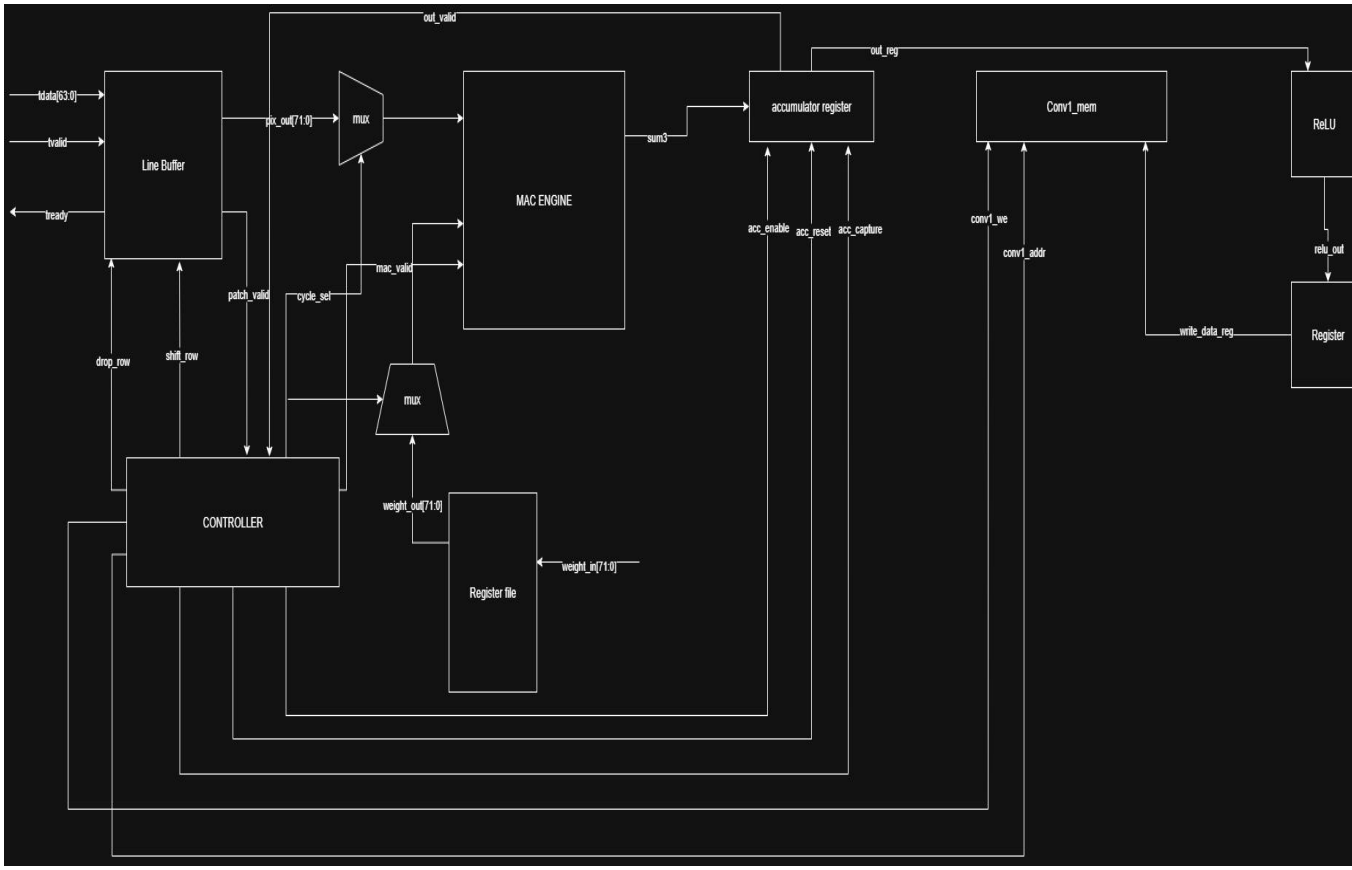
- Stores 6×6 feature map results.
- Write interface: `conv1_addr`, `conv1_wdata`, `conv1_we`.

4.9 Controller FSM

Controls sequencing of all modules.

Tasks:

- Load weights and rows.
- Manage patch setup and 3 MAC cycles.
- Capture final result and trigger ReLU/write.
- Shift columns, drop rows, and track row/column counters.
- Signal `done` when feature map complete.



5. Dry-Run Example (First Patch Walkthrough)

5.1 Row loading: FSM instructs line buffer to accept 3 beats (row0, row1, row2). Line buffer asserts `patch_valid`.

5.2 Patch setup: FSM resets accumulator, sets `cycle_sel=0`.

5.3 Cycle 1 (T0): Pixels p0..p2 × w0..w2 → sum3 → acc_reg updated.

5.4 Cycle 2 (T1): Pixels p3..p5 × w3..w5 → sum3 → acc_reg updated.

5.5 Cycle 3 (T2): Pixels p6..p8 × w6..w8 → sum3 → acc_reg updated.

5.6 Capture (T3): FSM asserts `acc_capture`. `out_reg <= acc_reg`.

5.7 ReLU + register: relu_out = max(0, out_reg). Stored in write_data_reg.

5.8 Writeback (T4): FSM asserts `conv1_we=1` with address (0,0). Value written into conv1_mem.

5.9 Next patch: FSM increments column counter, signals line buffer `shift_right`. New patch forms, steps repeat.

At the end of row: FSM issues `drop_row`, line buffer accepts a new beat (row3). Processing continues until all 6×6 outputs are generated.

6 Architectural Choices and Justification

6.1 Choice of Line Buffer Based Architecture

Why: CNN convolution requires overlapping 3×3 windows across rows and columns. A line buffer minimizes repeated DRAM fetches by storing only 3 rows at a time.

Impact:

- Reduces DRAM bandwidth usage.
- Ensures local patch availability with minimal latency.
- Makes design scalable to larger input sizes.

6.2 Three Parallel MAC Units (3-lane engine)

Why: A 3×3 kernel has 9 multiplications. With 3 MACs we can compute 3 products per cycle, finishing one patch in exactly 3 cycles.

Impact:

- Balanced design: small hardware footprint, reasonable throughput.
- Achieves **1 output pixel every 4–5 cycles** (3 cycles for MAC + 1–2 cycles for capture/write).

- Saves FPGA area compared to 9-MAC fully parallel design, while still faster than single-MAC sequential design.
- Modular — if needed, more MAC lanes can be added later.

6.3 Accumulator + Capture + ReLU Separation

Why: Accumulator runs during patch cycles, but ReLU and memory write need stable final result. Separation avoids hazards.

Impact:

- Cleaner control logic.
- Easier to pipeline and debug.
- Guarantees correct activation application.

6.4 On-chip conv1_mem for Feature Map Storage

Why: Writing intermediate results back to DRAM is costly. Using BRAM keeps data local.

Impact:

- Saves DRAM bandwidth.
- Enables chaining to next layer without round-trip to DRAM.
- Matches typical CNN accelerator designs (hierarchical memory).

6.5 Controller FSM Based Design

Why: CNN dataflow is highly regular but still requires sequencing (load rows, shift windows, capture, write). An FSM is natural for this flow.

Impact:

- Simplifies synchronization between line buffer, MAC, accumulator, and memory.
- Allows modular testing — FSM state transitions can be traced in waveforms.
- Provides a clean hook for scaling to more complex control later (multiple kernels, pooling).

7) Verification Plan

- Unit test each module (line buffer, MAC, accumulator).
- Integration test: feed known 8×8 image and weights, compare conv1_mem with MATLAB golden results.
- Corner cases: all-zero input, single bright pixel, random input.
- Debug signals: `cycle_sel`, `acc_reg`, `out_reg`, `conv1_we`, `conv1_addr`.

8) Next Steps

- Add pooling stage after conv1_mem.
- Extend architecture to second convolution layer (replicate modules).

- Explore pipelining inside MAC engine for higher clock speed.
- Scale image size to 28×28 for digit classification tasks.

9) Conclusion

This report outlines the RTL design of a 2-layer CNN accelerator prototype. The micro-architecture is modular, clear, and testable. The dry-run confirms correct dataflow: from input pixels → line buffer → MAC engine → accumulator → ReLU → conv1_mem. This document will guide the implementation of each module and serve as the baseline for extensions in the final FYP accelerator.