# Multi-Layer, Multi-Image $\alpha$ Validation on 8×8 Core

## 1. Purpose of This Testbench

- We extend the original `core_tb` into a **multi-layer, multi-image** validation environment.
- Goal: systematically verify our **$\alpha$-scaled, quantized** weights and activations on the 8×8 systolic core:
  - across **multiple network layers** (`NUM_LAYER`),
  - across **multiple input images** (`NUM_IMG`),
  - under both **2-bit** and **4-bit** operating modes.
- The testbench checks that the RTL core reproduces the **golden outputs** generated by our PyTorch + $\alpha$ pipeline.

## 2. High-Level Structure

- Bit-width loop:

$$\texttt{run\_i} = 0 \Rightarrow \text{2-bit mode}, \quad \texttt{run\_i} = 1 \Rightarrow \text{4-bit mode}.$$

- Nested loops:

```
for layer = 0 ..  NUM_LAYER-1for img = 0 ..  NUM_IMG-1
```

- For each (mode, layer, img) combination, the testbench:
  1. Resets the core.
  2. Loads the corresponding **activation tile** into `xmem`.
  3. Iterates over all 9 spatial kernel positions (`kij=0..8`).
  4. Streams the kernel and activations into the core.
  5. Reads out partial sums through OFIFO, writes them into `pmem`.
  6. Performs accumulation over `kij` according to `acc_address_layer.txt`.
  7. Compares the final output with the golden output file for this layer and image.

## 3. File Naming and Data Interface

All data consumed by `core_tb` are generated offline by our PyTorch scripts, which already apply **Conv+BN fusion**, **$\alpha$-scaling**, and **quantization**.

### 3.1 Activation Files

- Format:

```
activation_layer%d_img%d_tile0.txt
```

- Example:

```
activation_layer0_img2_tile0.txt
```

- Each file contains:
  - 3 header lines (comments),
  - followed by `len_nij` lines of packed activation bits matching the 8×8 systolic array input format.

### 3.2 Weight Files

- For each layer and each spatial kernel index `kij`:

  `weight_layer%d_itile0_otile0_kij%d.txt`

- Example:

  `weight_layer1_itile0_otile0_kij3.txt`

- These weights already encode:
  - Conv+BN fusion for that layer,
  - the chosen $\alpha$ scaling for that layer/mode,
  - and the 2-bit or 4-bit quantization scheme.
- In the testbench, each file is:
  1. Opened once per `kij`,
  2. Parsed into 8 rows × 8 columns,
  3. Written into high-address region of `xmem`,
  4. Then streamed into `L0` and loaded into PEs.

### 3.3 Accumulation Address Files

- For each layer:

  `acc_address_layer%d.txt`

- This file tells the testbench how to map partial sums in `pmem` to their corresponding output feature-map positions during accumulation.

### 3.4 Golden Output Files

- For each mode, layer, and image:

  2-bit: `out_2bit_layer%d_img%d.txt`

  4-bit: `out_4bit_layer%d_img%d.txt`

- Each file contains:
  - 3 header lines,
  - followed by `len_onij` lines of 128-bit outputs (8 columns × 16-bit psum), representing the first eight output positions required by the project.

# 4. Control Flow Inside the Testbench

## 4.1 Mode and Reset Handling

- The global mode is selected once per `run_i`:
  - `inst_w = 1` for 2-bit mode,
  - `inst_w = 0` for 4-bit mode.
- For each `(layer, img)` pair:
  - A local reset sequence is applied to ensure a clean starting state.
  - All control signals (`CEN_xmem`, `WEN_xmem`, `l0_wr`, `execute`, etc.) are re-initialized.

## 4.2 Activation and Kernel Flow

1. **Activation load:**
   - Read from $\texttt{activation\_layer}_i mg_t ile0.txt$.
   - `Write sequentially into xmem addresses` $[0..len\_nij - 1]$.
2. **Kernel loop (`kij = 0..8`):**
   - For each `kij`:
     (a) Load kernel weights from $\texttt{weight\_layer}_k ij.txtintothehighaddressregionof\texttt{xmem}$.
     `(a) Stream this kernel from xmem to L0.`
     `(b) Trigger weight loading into the PE array via l0_rd and load.`
     `(c) Stream all activation tiles from xmem to L0.`
     `(d) Assert execute while reading activations from L0 to perform MAC operations.`
     `(e) Read the OFIFO outputs and write partial sums into pmem.`

## 4.3 Accumulation and Verification

1. For each output position index (`i = 1 .. len_onij`):
   - Reset the core and then iterate over `kij`:
     - Use `acc_address_layer.txt` to fetch appropriate `pmem` addresses.
     - Assert `acc` to accumulate contribution from each kernel index.
   - After accumulation, compare the resulting `sfp_out` with the expected 128-bit value read from the corresponding golden output file.
   - Log `PASS` or `ERROR` for each feature-map position.
2. If no mismatches are detected:

   `PASS: Mode = {2,4}-bit, Layer, Img`

   otherwise:

   `FAIL: Mode = {2,4}-bit, Layer, Img`

## 5. Relation to $\alpha$-Scaling Pipeline

- The $\alpha$-scaling logic (weight combine, Conv+BN fusion, and PTQ) is implemented in Python / PyTorch.
- This Verilog testbench assumes that all file inputs (`activation_*.txt`, `weight_*.txt`, `out_*.txt`) already encode:
  - The desired bit-width (2-bit or 4-bit),
  - The fused Conv+BN parameters,
  - The chosen $\alpha$ for each layer.
- Therefore, **core_tb** serves as a pure RTL checker:
  - If the core passes for all modes, layers, and images, then the entire $\alpha$-scaled quantization pipeline is **functionally consistent** between PyTorch and hardware.

## 6. Key Points (for Poster)

- Multi-layer, multi-image testbench for the 8×8 systolic core.
- Supports both 2-bit and 4-bit modes through a single `mode` signal controlled by `inst_w`.
- Uses structured file naming:
  - `activation_layer_img_tile0.txt`,
  - `weight_layer_kij_.txt`,
  - `acc_address_layer_.txt`,
  - `out_{2bit,4bit}_layer_img.txt`.
- The RTL core is verified against golden outputs generated by our $\alpha$-aware PTQ pipeline.
- Once this testbench passes with zero mismatches, the $\alpha$ **scaling + quantization** design is validated end-to-end from VGG/ResNet to hardware.