TinyMAC: From 2×2 8-bit Mac Array to 4×4 Micro-Accelerator

Huancheng Yang Jiangyue Chu

NYU Tandon School of Engineering

CogniChip Hackathon — Feb 20, 2026

Repository

GitHub: https://github.com/Torzirael36/Cognichip-Hackson.git

Outline

1 Introduction

2 Architecture and RTL

3 Using the Cognichip Platform

4 Verification and Simulation Evidence

Problem & Motivation

- Multiply-Accumulate (MAC) is the core operation behind matrix multiplication and convolution:

$$\text{acc} \leftarrow \text{acc} + (a \times b)$$

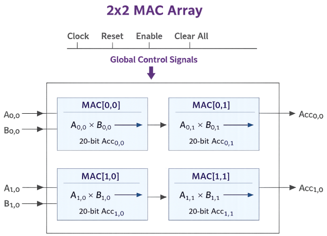
- Modern AI and DSP pipelines perform massive numbers of MAC operations; small compute blocks are the building units.
- **Goal:** build a compact, synthesizable **2×2 8-bit MAC array** with simple control and strong simulation evidence.

What we submit

RTL (SystemVerilog) + simulation testbench evidence (waveforms/logs) + final slides + demo video.

What We Built: TinyMAC 2×2

- A hierarchical, modular design with two layers:
  - **mac_unit**: single MAC with 8-bit inputs and a wide accumulator
  - **mac_array_2x2**: 4 MACs in parallel with global control signals
- Parallel compute: **4 MAC operations per cycle** (when enabled)
- Simple control interface: `enable` (pause/resume) and `clear_all` (reset accumulators)



What We Built: TinyMAC 4×4 Micro-Accelerator

- **Extended (4×4 micro-accelerator):**
  - **Throughput:** 16 MACs in parallel (**16 MACs/cycle**)
  - **A/B register buffers:** 4×4 each (`A_buf`, `B_buf`)
  - **N-cycle accumulation:** matrix-multiply style with `k` loop (`k=0..3`)
  - **Controller FSM:** `IDLE/LOAD/CLEAR/SETUP/COMPUTE/DONE`
  - **Readout:** register-mapped interface `out_addr` / `out_rdata`
  - **Verification:** self-checking SV testbench + golden model + waveform/log evidence

MAC Semantics and Timing

- Operation:

$$\text{accumulator} \leftarrow \text{accumulator} + (a \times b)$$

- **Synchronous accumulation**: updates occur on the rising edge of `clock`.
- **Reset/Clear behavior**:
  - `reset` or `clear` forces accumulator to zero
  - `enable` gates accumulation (no update when disabled)
- Product width: 8-bit $\times$ 8-bit $\Rightarrow$ 16-bit product.

Design choice

We use a wider accumulator to reduce overflow risk during repeated accumulation.

Accumulator Width Choice (int8 $\times$ int8 $\rightarrow$ int32)

- Default parameters (4×4 accelerator RTL):
  - `DATA_W = 8` (signed int8 inputs, range: $[-128, 127]$)
  - `ACC_W = 32` (signed accumulator)
- Single-product bound (signed int8):

$$\max|a \cdot b| = 128 \times 128 = 16384 \quad \text{(fits in 16 bits)}$$

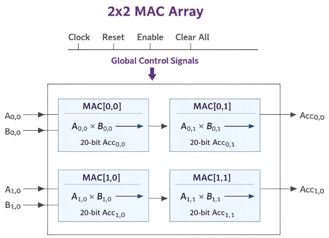
- For 4×4 matmul, each output accumulates $N = 4$ terms:

$$\max|C[i][j]| \leq 4 \times 16384 = 65536 \quad (\approx 17 \text{ bits magnitude})$$

- Using a 32-bit accumulator provides robust headroom for sign, corner cases, and future scaling (e.g., larger N or deeper accumulation).

Module 2: `mac_array_2x2`

- A 2×2 grid of 4 independent MAC units:
  - MAC[0,0], MAC[0,1], MAC[1,0], MAC[1,1]
- Shared global signals:
  - `clock`, `reset`, `enable`, `clear_all`
- Independent channels:
  - each MAC has its own `a_ij`, `b_ij` inputs and `acc_ij` output



4×4 Accelerator Micro-architecture

Compute Target (Matrix Multiply Kernel)

$$C[i][j] = \sum_{k=0}^{3} A[i][k] \cdot B[k][j]$$

Architecture Highlights

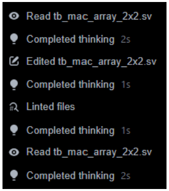
- **Parallelism:** 16 MACs per cycle (all i, j in parallel)
- **N-cycle accumulation:** $N = 4$ cycles over k
- **Per-cycle dataflow:**
  - fixed `k_index`
  - broadcast $A[:, k]$ and $B[k, :]$ into array
- **Internal state:**
  - `A_buf[4][4]`
  - `B_buf[4][4]`
  - `C_acc[4][4]`

How We Used Cognichip

- Generated and iterated on SystemVerilog RTL modules using the Cognichip workflow.
- Created/updated simulation targets (RTL + testbench + run configuration).
- Ran simulation to produce logs and waveform traces for correctness evidence.

How We Used Cognichip for RTL Development

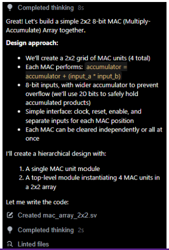
- We used Cognichip as an AI-assisted RTL co-design platform.
- Started with a high-level architectural description:
  - 2x2 MAC array
  - 8-bit inputs
  - 20-bit accumulator
  - Global control signals
- Cognichip generated synthesizable SystemVerilog automatically.
- Built hierarchical modules:
  - mac_unit
  - mac_array_2x2



```
Read tb_mac_array_2x2.sv
Completed thinking 2s
Edited tb_mac_array_2x2.sv
Completed thinking 1s
Linted files
Completed thinking 1s
Read tb_mac_array_2x2.sv
Completed thinking 2s
```

Prompt Engineering: Guiding the Design

- Instead of vague requests, we provided:
  - Explicit data widths
  - Accumulator overflow constraints
  - Clear/reset semantics
  - Hierarchical module structure
- Structured design approach:
  - Step 1: Define mac_unit
  - Step 2: Instantiate 2x2 grid
  - Step 3: Add control signals
- We treated Cognichip like a hardware co-engineer.



Completed thinking 8s

Great! Let's build a simple 2x2 8-bit MAC (Multiply-Accumulate) Array together.

Design approach:

- We'll create a 2x2 grid of MAC units (4 total)
- Each MAC performs: `accumulator = accumulator + (input_a * input_b)`
- 8-bit inputs, with wider accumulator to prevent overflow (we'll use 20 bits to safely hold accumulated products)
- Simple interface: clock, reset, enable, and separate inputs for each MAC position
- Each MAC can be cleared independently or all at once

I'll create a hierarchical design with:

1. A single MAC unit module
2. A top-level module instantiating 4 MAC units in a 2x2 array

Let me write the code:

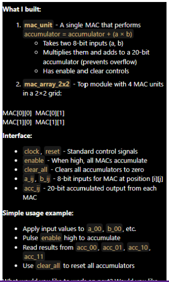
Created mac_array_2x2.sv

Completed thinking 2s

Linted files

Iterative Refinement and Self-Verification

- After initial generation:
  - Ran lint checks
  - Requested improvements
  - Verified coding style
- Cognichip automatically:
  - Corrected structural issues
  - Ensured synthesizable constructs
  - Maintained clean hierarchical separation
- Achieved zero lint warnings.



What I built:

1. `mac_unit` - A single MAC that performs `accumulator = accumulator + (a × b)`
   - Takes two 8-bit inputs (a, b)
   - Multiplies them and adds to a 20-bit accumulator (prevents overflow)
   - Has enable and clear controls
2. `mac_array_2x2` - Top module with 4 MAC units in a 2×2 grid:

```
MAC[0][0]  MAC[0][1]
MAC[1][0]  MAC[1][1]
```

Interface:

- `clock`, `reset` - Standard control signals
- `enable` - When high, all MACs accumulate
- `clear_all` - Clears all accumulators to zero
- `a_ij`, `b_ij` - 8-bit inputs for MAC at position [i][j]
- `acc_ij` - 20-bit accumulated output from each MAC

Simple usage example:

- Apply input values to `a_00`, `b_00`, etc.
- Pulse `enable` high to accumulate
- Read results from `acc_00`, `acc_01`, `acc_10`, `acc_11`
- Use `clear_all` to reset all accumulators

Why Cognichip Is Powerful

- Rapid RTL generation:
  - Full 2x2 MAC array built in seconds
- Clean, parameterized, synthesizable code
- Supports hierarchical hardware abstraction
- Enables faster design exploration

Engineering Impact

Cognichip significantly reduces development time while maintaining hardware-level correctness and modularity.

From idea to working RTL in minutes.

Verification Methodology (4×4)

Golden Reference Model

- SystemVerilog matrix-multiply reference
- Signed int8 inputs → int32 accumulation

Directed Tests

- all zeros
- identity × random ($A = I \Rightarrow C = B$)
- max positive / max negative
- mixed signs

Random Regression

- 5–20 random matrix pairs per run
- full 16-element compare

Verification Checks and Evidence

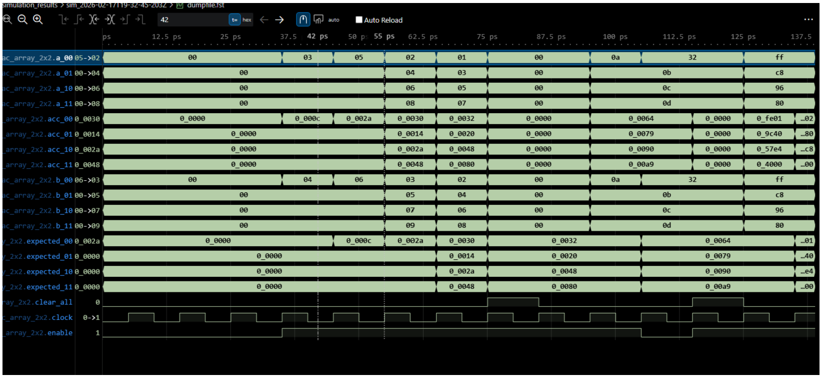
Correctness Checks

- `done` latency: CLEAR + SETUP + N compute cycles
- Per-element comparison
  - All 16 outputs verified
  - via `out_addr / out_rdata`

Artifacts

- Waveform inspection
- PASS / FAIL summary logs
- Automatic mismatch detection

Simulation Results: 2×2 8-bit Mac Array Waveform Evidence


simulation_results > sim_2026-02-17T19-32-45-203Z > dumpfile.fst
42
hex
auto
Auto Reload
ps 12.5 ps 25 ps 37.5 42 ps 50 ps 55 ps 62.5 ps 75 ps 87.5 ps 100 ps 112.5 ps 125 ps 137.5
ac_array_2x2.a_00 05->02 00 03 05 02 01 00 0a 32 ff
ac_array_2x2.a_01 00->04 00 04 03 00 0b c8
ac_array_2x2.a_10 00->06 00 06 05 00 0c 96
ac_array_2x2.a_11 00->08 00 08 07 00 0d 80
_array_2x2.acc_00 0_0030 0_0000 0_000c 0_002a 0_0030 0_0032 0_0000 0_0064 0_0000 0_fe01 …02
_array_2x2.acc_01 0_0014 0_0000 0_0014 0_0020 0_0000 0_0079 0_0000 0_9c40 …80
_array_2x2.acc_10 0_002a 0_0000 0_002a 0_0048 0_0000 0_0090 0_0000 0_57e4 …c8
_array_2x2.acc_11 0_0048 0_0000 0_0048 0_0080 0_0000 0_00a9 0_0000 0_4000 …00
ac_array_2x2.b_00 06->03 00 04 06 03 02 00 0a 32 ff
ac_array_2x2.b_01 00->05 00 05 04 00 0b c8
ac_array_2x2.b_10 00->07 00 07 06 00 0c 96
ac_array_2x2.b_11 00->09 00 09 08 00 0d 80
y_2x2.expected_00 0_002a 0_0000 0_000c 0_002a 0_0030 0_0032 0_0064 …01
y_2x2.expected_01 0_0000 0_0000 0_0014 0_0020 0_0079 …40
y_2x2.expected_10 0_0000 0_0000 0_002a 0_0048 0_0090 …e4
y_2x2.expected_11 0_0000 0_0000 0_0048 0_0080 0_00a9 …00
ray_2x2.clear_all 0
c_array_2x2.clock 0->1
_array_2x2.enable 1

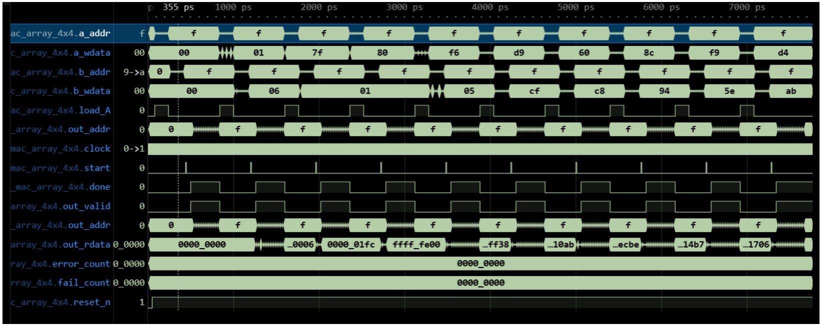
Test Case Walkthrough: Multiply-Accumulate Behavior

- **Step 1: 3 × 4 = 12**
  - Inputs: `a_00 = 3`, `b_00 = 4`, `enable = 1`
  - Product: $3 \times 4 = 12 = 0x000C$
  - Accumulator update:
    $$acc_{00} : 0x0000 \to 0x000C$$
  - `acc_00` matches `expected_00`
- **Step 2: 5 × 6 = 30**
  - $5 \times 6 = 30 = 0x001E$
  - Accumulation:
    $$0x000C + 0x001E = 0x002A\ (42)$$
  - Waveform shows step-wise increase to `0x002A`
- **Step 3: 2 × 3 = 6**
  - $2 \times 3 = 6$
  - Accumulation:
    $$0x002A + 0x0006 = 0x0030\ (48)$$

Simulation Results: Log Evidence

- The cognichip runs the testbenches.
- All directed tests passed (9/9).
- The reset,enable signals are all tested.
- Logs record key checks and final PASS/FAIL summary.

Simulation Results: 4×4 Accelerator Waveform Evidence

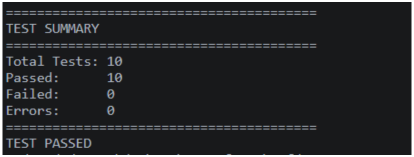


4×4 Example: Identity $\times$ Random (Sanity Check)

- Test setup:
  - Set $A = I$ (4×4 identity), random B
  - Expected result: $C = A \cdot B = B$
- Test flow (self-checking):
  - TB loads `A_buf` and `B_buf` via `load_A`/`load_B` + address/data
  - Assert `start`; controller runs `k=0..3` accumulation
  - Read back all 16 outputs via `out_addr`/`out_rdata`
  - Compare against golden model (SV reference matmul)
- Takeaway: validates end-to-end datapath + control + readout interface

Simulation Results: 4×4 Log Evidence

- Directed tests: **5/5 passed**
- Random regression: **5/5 passed**
- Total mismatches: **0**
- Testbench structure: Tests 1–5 directed, Tests 6–10 random
- Logs include per-element checks and final PASS/FAIL summary.



```
========================================
TEST SUMMARY
========================================
Total Tests: 10
Passed:      10
Failed:      0
Errors:      0
========================================
TEST PASSED
```

Challenges, Lessons, and Future Directions

Technical Challenges

- N-cycle accumulation with correct k-loop control
- Off-by-one / skip-k FSM sequencing bugs
- Clean load–compute–readout interface

Key Lessons

- Verification must validate microarchitecture, not just arithmetic
- Self-checking TB exposed real control-path misalignment
- Modular design (MAC $\rightarrow$ array $\rightarrow$ controller) enables scaling

Future Directions

- Scale to 8×8 and larger arrays
- Add pipelining / retiming for higher clock frequency
- Explore systolic dataflow variants
- FPGA demo and PPA measurement