Streaming CNN Pipeline on 1024×1024 Input

SystemVerilog RTL, designed from scratch · NCSU ECE 564 (FPGA/ASIC Design with Verilog) · Fall 2025

Topics: *SystemVerilog RTL design · two-FSM DRAM controllers · streaming-pipeline memory architecture (ring buffer + line buffer) · synthesis discipline (timing closure, latch-free, cell count) · Synopsys DC + Nangate 45nm flow*

> **The one-liner:** A streaming CNN datapath in SystemVerilog - 4×4 conv with 16 MACs → Leaky ReLU → 2×2 average pool - operating end-to-end DRAM-in to DRAM-out on 1024×1024 images. No SRAM. Two custom DRAM controllers as FSMs.
>
> **What landed:** 6/6 tests passing · 16.0 ns clock met (slack +0.0018 ns) · no inferred latches · 535,158 cells · 36.15M cycles for full image.
>
> **Why it's interesting:** A pure-RTL design built from scratch in SystemVerilog that hit tight timing on the first synthesis pass. The two memory-engineering decisions (ring-buffer input + 2-row line buffer) are real choices that shrink area dramatically vs. naïve full-frame buffering.

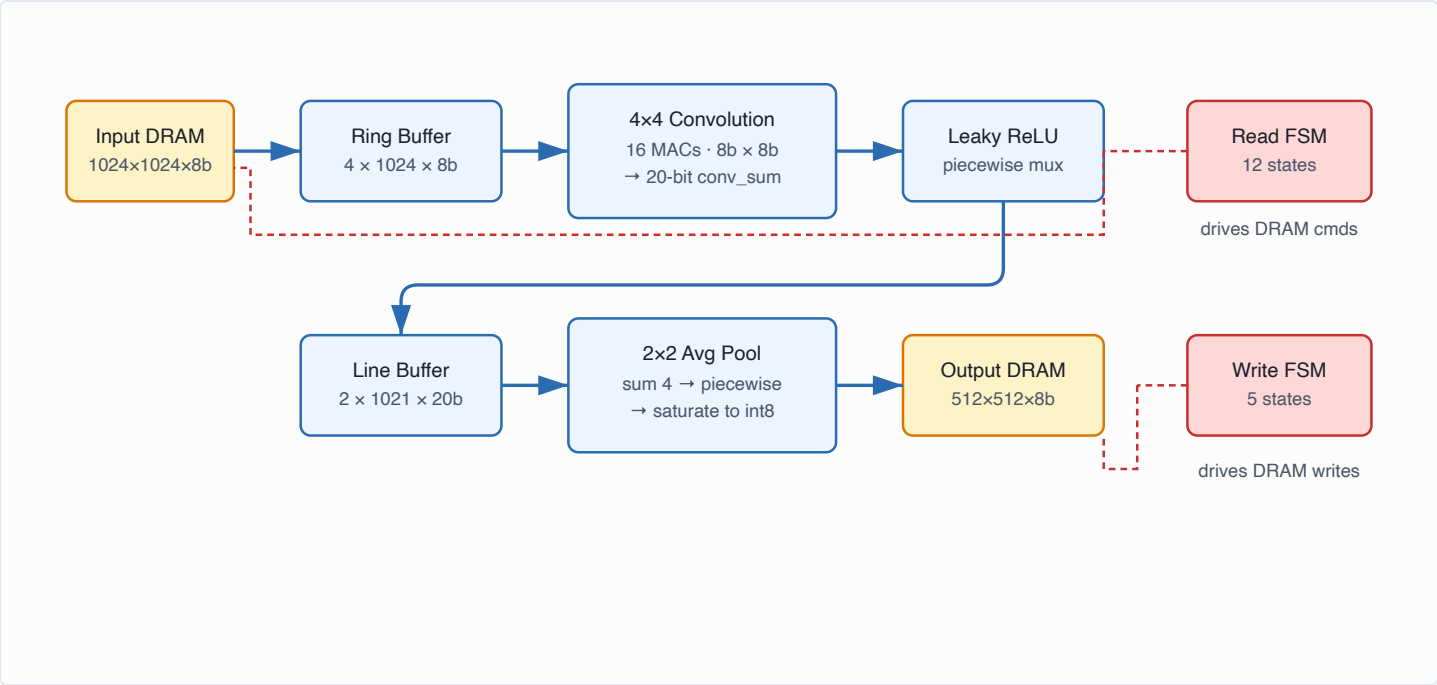
What it does

Takes a 1024×1024 image stored in input DRAM, streams it through a 6-stage pipeline, and writes the result back to output DRAM. The math:

1. **Load kernel** - pull the 4×4 kernel weights from DRAM into 16 kernel registers.
2. **Load inputs** - pull the first window's worth of input pixels from DRAM into the input buffer.
3. **Convolution** - 4×4 kernel × 4×4 input window → 16 parallel MACs → sum to a 20-bit `conv_sum`.
4. **Leaky ReLU** - piecewise on `conv_sum`: `x` for `x > 0`, `0` for `−4 < x ≤ 0`, `⌈x/4⌉` for `x ≤ −4`.
5. **2×2 average pooling** - sum of 4 neighbors, then piecewise: `⌈x/4⌉` for `x ≥ 4`, `0` for `−4 < x < 4`, `⌈x/4⌉` for `x ≤ −4`. Output saturated to 8-bit signed (–127 to +128).
6. **Writeback** - burst the 8-bit pooled pixel back to output DRAM.

Stages 3-6 stream continuously once the pipeline is primed. The Read Controller keeps the input buffer fed; the Write Controller drains pooled outputs back to DRAM.

Architecture at a glance



Top-level dataflow: streaming pipeline DRAM-in → DRAM-out. No SRAM. Two FSMs (red) drive the two DRAM interfaces independently.

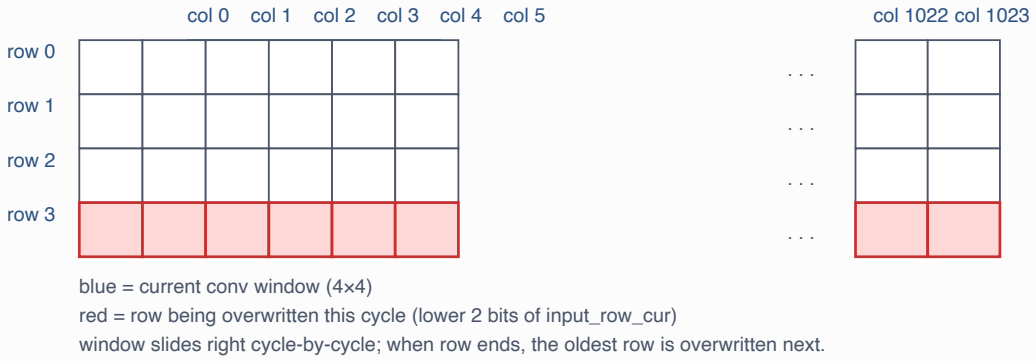
The key architectural calls

- **No SRAM, deliberate.** Stuck with DRAM the entire way. Trade-off: tighter memory-controller logic in exchange for no SRAM area and one fewer memory boundary to manage.
- **Streaming, not batched.** Operations begin as soon as the input buffer holds a valid 4×4 window. No "load full frame, then process."
- **Two independent FSMs.** Read and Write controllers run in parallel - the Read FSM stages input data while the Write FSM drains pooled outputs back, so the pipeline never idles between the two DRAM interfaces.

Memory engineering

These two buffer choices are the heart of the design. They show how the design trades area against naïve choices.

Decision 1: 4×1024 ring buffer for the input window



4×1024 ring buffer. Holds only enough rows for one 4×4 conv window. Lower 2 bits of `input_row_cur` select which physical row to write incoming pixels into - new rows naturally overwrite the oldest.

Why this matters: Naive would be a 1024×1024 = 1 MB frame buffer. The ring buffer is 4×1024 = 4 KB. Same correctness, ~256× less storage. Every DRAM burst brings in 8 pixels (burst length 8) into the correct columns of one of the four physical rows.

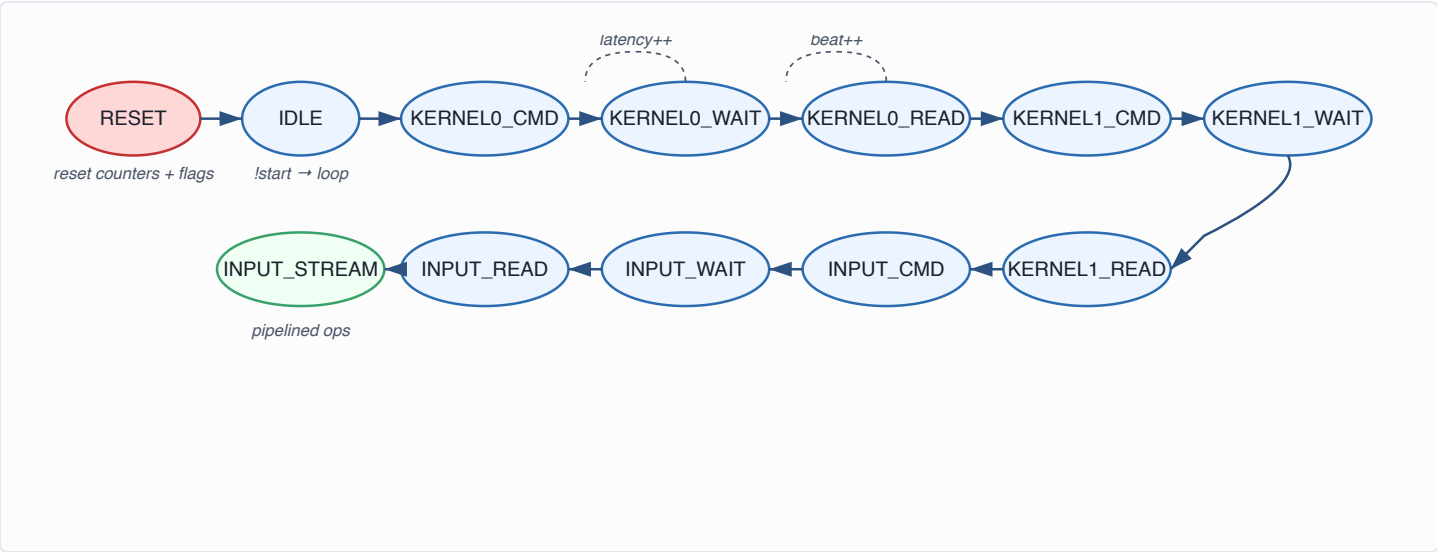
Decision 2: 2×1021 line buffer between ReLU and pool

Same idea, applied to the ReLU → pool seam. A 2×2 pool needs two ReLU outputs from the previous row paired with two from the current row. Buffer exactly two rows of 1021 ReLU outputs (the post-conv resolution is 1021×1021 - a 4×4 conv with stride 1 on a 1024×1024 input gives 1024 − 4 + 1 = 1021 per side), not the full 1021×1021 conv-output frame. Storage: 2 × 1021 × 20b ≈ 40 Kbit, vs. the naive 1021 × 1021 × 20b ≈ 20.9 Mbit - roughly a 510× reduction.

For each 2×2 block: read two from the stored "previous" row + two from the current row → sum all four → apply the piecewise averaging rule → saturate to int8 → produce one pooled pixel.

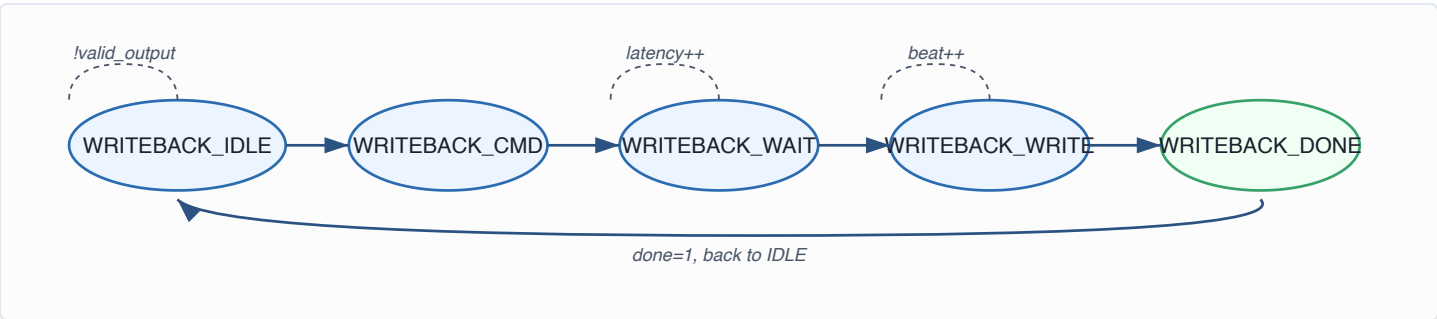
The two FSMs

Read Controller - 12 states



Read Controller FSM. Two kernel halves (8 beats each via burst-len-8 reads with latency-5 wait) → input window load → INPUT_STREAM does the pipelined compute. `latency_count` *counts wait cycles;* `beat_count` *counts beats in a burst.*

Write Controller - 5 states



Write Controller FSM. Waits in IDLE until a valid pooled output, then issues a WRITE command, waits for latency-3, bursts 8 beats out, signals done.

The **DRAM interface parameters** baked into both FSMs:

- `DRAM_DQ_WIDTH = 8` bits
- Burst length 8 (reads bring in 8 pixels per command)
- Read latency 5 cycles, write latency 3 cycles
- Address width 32-bit; kernel halves at `0x0000_0000` and `0x0000_0008`, inputs starting at `0x0000_0010`

Numbers

Metric	Value	Where it came from
Public tests passed	`6 / 6`	ModelSim/QuestaSim simulation
Total cycle count (1024×1024)	`36,149,772`	final simulation log
Total simulated time	`180,748,860 ns (~180.7 ms)`	final simulation log
Synthesis clock period (target)	`16.0 ns`	Design Compiler constraint
Final slack	`+0.0018 ns (MET)`	`timing_max_slow.rpt`
Inferred latches	`0`	`CheckLatches.tcl`
Total cell count	`535,158`	`cell_report_final.rpt`
Synthesis CPU time	`7.52 hours (elapsed)`	DC log

> **The clean-met-timing story:** Targeted 16 ns and hit it. Slack is +0.0018 ns - basically right at the edge, but cleanly MET. Long path runs through the avg-pool line buffer (endpoint `av_pool_line_buffer_reg[0][132][6]/D`). Did *not* attempt to push lower.

Tooling

Stage	Tool
Simulation	Mentor ModelSim / QuestaSim (headless via `make vsim-dut` / `vsim-dut-c`)
Synthesis	Synopsys Design Compiler T-2022.03-SP4
Standard cells	Nangate OpenCell Library PDKv1_2 v2008_10 (45nm), slow / fast / typical corners + `dw_foundation.sldb`
Build flow	CMake + Make wrapping the Synopsys EDA flow
Compute	NCSU EDA cluster

Design decisions and trade-offs

What it demonstrates: Graduate-coursework RTL design done from scratch in SystemVerilog. Met a 16.0 ns clock at right-on-the-edge slack on the first synthesis attempt, passed every functional test, synthesized with no latches. The two memory decisions (no-SRAM streaming + ring-buffer input + 2-row line buffer) are *real* engineering choices, not defaults.

Scope and limitations: Course-scale, not production-scale.

- No coverage-driven verification, no formal, no power analysis, no place-and-route.
- Course scope did not cover lint, coverage, or timing-exception handling - those belong to a production verification flow.
- The cell library is Nangate 45nm open - fine for the class, not what a production team would use.

Pushing the clock lower

The long path runs through the average-pooling line buffer. Tightening it would mean either registering the conv → ReLU → pool seam more deeply, or pipelining the piecewise mux path. The design targeted 16 ns and hit it cleanly on the first pass; it was not pushed lower than that.

Common questions about this project

Question	Short answer
How large is this design?	535,158 standard cells synthesized in Nangate 45nm. A streaming CNN datapath with two FSMs and no SRAM, meeting a 16 ns clock on the first synthesis pass.
How does the memory architecture work?	Ring buffer + line buffer; both shrink area dramatically vs. naïve full-frame buffering. (See the two SVGs above.)
Why no SRAM?	Deliberate. Wanted a true streaming pipeline DRAM-in/out, no intermediate staging. Trade-off: tighter memory controllers in exchange for one fewer memory boundary.
How would this be verified for production?	Course scope was directed-test based (6 vectors). Production would need coverage-driven random verification, UVM-style scoreboards, formal on the FSMs, and lint-clean per the team's signoff rules. That's the next thing I want to build.
Why two FSMs?	Read and Write touch independent DRAM interfaces. Decoupling them lets one drain pooled outputs while the other is filling the input buffer - no idle cycles waiting on each other.
What was the hardest part?	Getting the ring-buffer indexing right at row boundaries - the lower-two-bits trick is clean but you have to make sure the conv window addressing modular-wraps consistently with the writes. Sim caught the off-by-one.
Was place-and-route done?	No. Synthesis only. Course didn't go to PNR.
What synthesizer does it use?	Synopsys Design Compiler T-2022.03 on NCSU's cluster, against Nangate Open 45nm.
What would change in a revision?	Probably register the avg-pool sum stage to break the long path - would buy headroom to push the clock down. And add a proper testbench harness with self-checking + coverage from day one rather than retrofitting at the end.
How would this map to an FPGA target?	The MACs would map to DSP blocks (16 of them); the ring buffer and line buffer fit comfortably in BRAM; the FSMs are tiny in LUTs. The 16 ns clock (62.5 MHz) is conservative for FPGA - UltraScale+ would clear it easily.

Question	Short answer
How does this compare to a high-level-synthesis approach?	HLS would have written this faster but probably wouldn't have produced the same ring-buffer / line-buffer area win without explicit pragmas. Hand-RTL gave me direct control over the memory architecture.