RTL Implementation and Verification of A 3×3 Convolution–ReLu Datapath for Underwater Image Edge Filtering

Vikram G[1], Sakthivel Murugan S[2]
[1]*Department of Electronics and Communication Engineering, JCT College of Engineering and Technology, Coimbatore, India*
[2]*Department of Electronics and Communication Engineering, National Institute of Technical Teachers Training and Research (NITTTR), Chennai, India*

Abstract*—This paper describes the RTL implementation and simulation-based verification of a small 3×3 convolution–ReLU Datapath intended for edge filtering of underwater images. The Verilog design has input and weight buffers, one multiply–accumulate (MAC) unit with a 32-bit signed accumulator, a ReLU stage, an output buffer, and a nine-state finite-state machine (FSM) embedded in the top module. Each window is processed serially and occupies the datapath for 48 clock cycles. A self-checking testbench compares every output with a reference value computed inside the testbench. In the final QuestaSim run, 9 directed windows, 500 constrained-random windows and 911 image frames (900 windows per 32×32 frame, 819,900 in total) all matched the reference, with no timeouts, protocol errors, frame-load errors or file warnings, and the run ended with STATUS: PASS. The frame windows come from a single video and are strongly correlated, so they are not 819,900 independent tests. The evidence is simulation only: no formal verification, FPGA, gate-level, silicon or measured-power result is claimed. Pre-layout synthesis figures and a separate software classification experiment from the earlier manuscript are retained for completeness but are marked as not re-verified.

***Index Terms*—Convolution–ReLU datapath, RTL verification, self-checking testbench, edge filtering, underwater imaging, logic synthesis.**

I. INTRODUCTION

Underwater imaging supports marine observation, inspection and environmental monitoring. Absorption, scattering and low visibility degrade the images, so operations that extract local structure are a common first step. Convolution-based filtering is one of the simplest such operations and maps naturally onto digital hardware.

A 3×3 convolution forms each output from a neighbour-hood of nine pixels and nine weights; a rectified linear unit (ReLU) then sets negative sums to zero. Describing this data-path at register-transfer level makes the control sequencing, number representation and cycle-level behaviour explicit and open to inspection.

This work presents a Verilog 3×3 convolution–ReLU data-path. Pixels are 8-bit unsigned; weights are 8-bit signed, and accumulation uses a 32-bit signed register. A 32×32 frame yields a 30×30 output, i.e., 900 windows, and 911 frames therefore give 911 × 900 = 819,900 windows.

Contributions. (1) An RTL Datapath with an embedded control FSM, together with an account of which source files are instantiated, compiled and simulated. (2) A self-checking testbench with directed, constrained-random and image-frame phases, and a final 911-frame run whose logged result is stated together with its limits. (3) A description of the clock-edge scheduling used in the testbench to avoid stimulus/sampling races. (4) Earlier pre-layout synthesis estimates and a small software experiment, kept separate from the hardware and flagged as carried over.

Scope. Weights are supplied through an input port, so the RTL is a general serial 3×3 MAC-plus-ReLU datapath; the edge kernel is applied by the testbench during the frame phase. This paper does not report a trained CNN, FPGA implementation, post-layout

timing, measured power, silicon, or a hardware classifier. The only classification result (Section VIII) comes from a software experiment that is not part of the RTL.

II. RELATED WORK

Zhao *et al.* [1] built an embedded FPGA image-recognition system for a CNN using parallelism and pipelining, and report workstation-equivalent accuracy at 25 FPS for 1920×1080 input. Abd El-Maksoud *et al.* [2] describe a GoogLeNet accelerator in native RTL with pruning and quantisation, reporting
25.1 fps at 3.92 W. Kim and Choi [3] apply RTL-level techniques such as clock gating to a reconfigurable accelerator evaluated with ResNet-20 on CIFAR-10. Eyeriss [4] reduces whole-system energy by reconfiguring the dataflow to limit data movement. Lin *et al.* [5] survey low-power, ultra-small edge AI accelerators and note that many articles lack full specifications and comprehensive comparisons. Devic *et al.* [6] combine an RTL MAC unit with an analytical model to assess CNN mappings, and Teerapittayanon *et al.* [7] distribute a DNN over cloud, edge and end devices.

Those works concern trained networks and report frame rate or power on FPGAs or silicon. The present design is a deliberately small serial datapath evaluated by simulation and pre-layout synthesis. It is a stepping stone rather than a competitor, and no performance, power or accuracy compar-ison with [1]– [4] is made or implied. (The figures quoted above are taken from the cited papers as summarised in the earlier manuscript and have not been re-checked against the originals.)

III. SYSTEM ARCHITECTURE

3-A. Overall Architecture

The top module ai_accelerator instantiates five modules: input_buffer, veight_buffer, nac_unit, relu and output_ buffer (Fig. 1). The control FSM, the address counters and the operation counter are written directly in the top module. Pixels and weights arrive through input_data and veight_ data and are written into two 16-entry, 8-bit buffers with synchronous read. The FSM reads one entry of each buffer at a time into two capture registers (nac_data_reg, unsigned; nac_veight_reg, signed). The single MAC accumulates nine products, ReLU clamps the sum, and the result is written to a 16-entry, 32-bit output buffer with combinational read.

The module parameters are DATA_WIDTH=8, ACC_WIDTH=32, BUFFER_DEPTH=16 and NUH_0PERATI0NS=9.

TABLE 1. RTL source files and their role in the simulated design

File	Role (from source)	Used
ai_accelerator. v	Top: FSM, counters, capture regs	yes
input_buffer. v	16×8 b, synchronous read	yes
veight_buffer. v	16×8 b, synchronous read	yes
nac_unit. v	u8×s8 MAC, 32-b accumulator	yes
relu.v	32-b signed ReLU	yes
output_buffer. v	16×32 b, combinational read	yes
controller_fsn. v	generic 6-state FSM	no
accunulator. v	generic 16-bit accumulator	no

TABLE 1 lists the eight RTL files supplied. All eight are compiled by run. do, but only the top module and five leaf modules are elaborated: the simulator loads ai_accelerator, input_buffer, veight_buffer, nac_unit, relu and output_ buffer, and reports one FSM recognised in the top module. controller_fsn. v and accumulator. v is therefore syntax-checked only; neither is part of the simulated datapath and no validation claim is made for them.

3-B. Control Sequence and Timing

The FSM has nine states (Fig. 2): S_IDLE, S_L0AD, S_READ, S_CAPTURE, S_WAIT, S_C0HPUTE, S_RELU, S_ST0RE and S_D0NE.

One pixel/weight pair is written per cycle: address 0 in the S_ IDLE cycle that samples start, and addresses 1–8 in S_L0AD. Each of the nine MAC operations then passes through READ, CAPTURE, WAIT and COMPUTE, and the window finishes with RELU, STORE and DONE. The datapath is therefore occupied for $1 + 8 + 9 \times 4 + 3 = 48$ cycles per window, and done is high only in the 48th. At the 10 ns testbench clock this is 480 ns per window and a MAC utilisation of 9⁄48 = 18.75%. The input and weight pins are sampled only while start is accepted and during S_L0AD; afterwards they are ignored, which the testbench checks by driving junk values on them.

Reset rst_n is active low and synchronous; it clears the state, counters, capture registers and the three buffers. A start is accepted only in S_IDLE, and undefined state encodings return to S_IDLE. The output buffer writes the ReLU result at the current write address and points the read address at that same entry, so it behaves as a single result register rather than a frame store: the write address wraps modulo 16 and only the latest entry is ever read. Weights are re-streamed for every window, so the kernel is not held stationary.

Accumulation takes place in nac_unit. Its 32-bit signed accout is cleared by reset or by a clear strobe in the idle cycle that samples start, and otherwise adds the product while valid is high in S_C0HPUTE. Fig. 4 shows this behaviour in simulation: valid pulses for one cycle in every fourth cycle, account changes one clock after each pulse, and state cycles through 2, 3, 4, 5 (READ, CAPTURE, WAIT, COMPUTE) while operation_count and the two read addresses advance together.

IV. DATAPATH AND NUMERICAL REPRESENTATION

4-A. Convolution, Arithmetic and Range

For input I and kernel K, each output is

$$S(i,j) = \sum_{m=0}^{2} \sum_{n=0}^{2} I(i+m, j+n)K(m,n), \quad (1)$$

$$O(i,j) = \max\{0, S(i,j)\}, \quad (2)$$

with $0 \le i, j \le 29$, I an 8-bit unsigned pixel, K an 8-bit signed weight and S a 32-bit signed sum. The window is not flipped, so (1) is a cross-correlation. For an $N \times N$ input, a $k \times k$ kernel, unit stride and no padding the output has $(N-k+1)^2$ elements; for $N = 32$, $k = 3$ this is $30 \times 30 = 900$.

In nac_unit the pixel is zero-extended and the weight sign-extended to 32 bits before a signed multiplication, which is the correct treatment for unsigned pixels and signed weights.

The kernel used in the frame phase of the testbench has every row equal to (32, 0, −32), a Prewitt-type horizontal gradient scaled by 32 that, with ReLU, passes only responses where the left column is brighter than the right. For this kernel each sum is 32 times the sum of three left-minus-right differences, each at most 255 in magnitude, so

$$|S| \le 255 \cdot (3 \cdot 32) = 24{,}480, \quad (3)$$

which fits in 16 signed bits. The 32-bit accumulator is there-fore generous for this particular kernel, but it is not wasted for arbitrary weights: the directed test with all pixels at 255 and all weights at 127 produces $9 \cdot 255 \cdot 127 = 291{,}465$, which needs 20 signed bits. Because ReLU output is non-negative, the top bit of output_data is always zero.

4-B. Preprocessing

The earlier manuscript describes a script that converts a video frame to 8-bit grayscale, resizes it to 32×32 with nearest-neighbour sampling and writes one two-digit hexadecimal pixel per line in row-major order, giving the frane_NNNN. hex files that the testbench reads. Resizing a 1280×720 frame to 32×32 changes the aspect ratio and subsamples roughly 40× horizontally and 22× vertically, so the result is heavily aliased; it is a hardware test vector, not an image-quality-preserving representation. Fig. 3 shows one example. The scripts and frame files were not among the files supplied for this revision, so this figure and the preprocessing description are carried over and could not be re-checked.

V. TESTBENCH AND VERIFICATION METHODOLOGY

5-A. Testbench Structure

The supplied self-checking testbench ai_accelerator_tb. v (468 lines) instantiates the design as dut and generates a 100 MHz clock. After reset it runs three phases, summarised in TABLE 2.

Phase 1 applies nine directed windows: all-zero pixels, maximum positive products, maximum negative products (ReLU clamp), cancelling products, a mixed-sign ramp, −1 weights (sign extension), alternating extremes, an MSB-set pixel (unsigned treatment) and zero weights.

Phase 2 applies 500 constrained-random windows with random pixels and random signed weights from a fixed seed. *Phase 3* reads each frane_NNNN. hex with $readnenh, pre-filled with unknown values so that a short or missing file is detected, and applies the 900 windows of each 32×32 frame in row-major order using the edge kernel above. Each window is streamed as nine pixel/weight pairs, the first together with start.

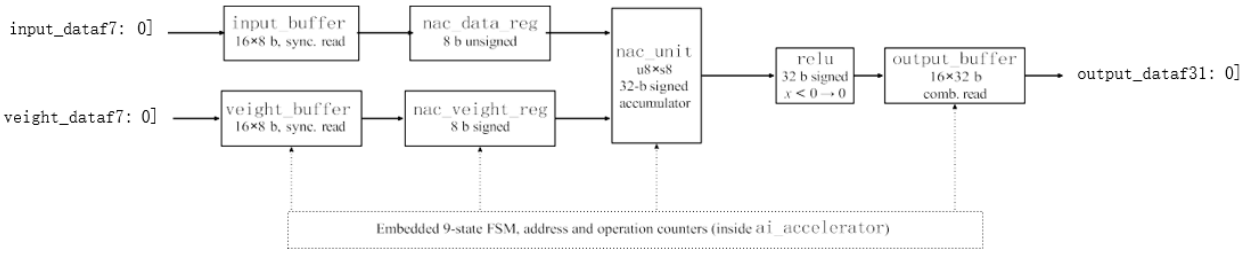

Fig. 1. Instantiated architecture of ai_accelerator, redrawn from the top-level source. Solid arrows show the data path; dotted arrows show control from the embedded FSM (addresses, write enables, capture, clear and valid strobes). The write/read control to the two input-side buffers is shown once for clarity. controller_fsn. v and accumulator. v is not instantiated.

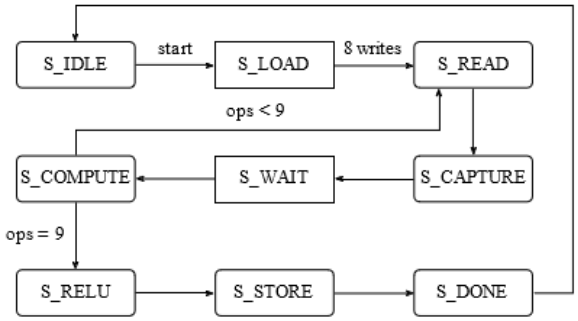

Fig. 2. Embedded controller FSM. One window takes 1 (IDLE) + 8 (LOAD) + 9×4 (READ, CAPTURE, WAIT, COMPUTE) + 3

For every window the testbench computes the expected value as the exact integer sum of nine products followed by ReLU, and compares it with output_data using case inequal-ity (! ==), so, X or Z outputs count as failures. It also checks the handshake: the DUT must be idle (busy=done=0) before each start; busy must be high during loading and until done; busy must be low while done is high; done must last exactly one cycle; and the start-to-done count measured on the first window must repeat for every later window. Waiting for done is bounded by a 200-cycle timeout that is followed by a reset. Results are also written to one text file per frame. The final summary line is STATUS: PASS only if all frames completed, the window count equals frames×900, and the failure, time-out, protocol-error and load-error counts are all zero. Note that the output-file warning counter is printed but does not enter this condition.

The reference value and the stimulus use the same weights, and both are written by the authors, so the check confirms that the RTL implements the intended arithmetic and handshake. It is not an independent validation of the kernel or of the specification itself.

5-B. Clock-Edge Scheduling

Driving inputs with blocking assignments immediately after 0(posedge clk) and reading outputs at the same edge is race-prone, because the DUT samples on that edge and the outcome depends on simulator event ordering. The sup-plied testbench avoids this pattern: start, input_data and veight_data change only after 0(negedge clk), so the DUT samples them half a period (5 ns) later, and done, busy and output_data are observed only on falling edges, where the registered output of the preceding rising edge is stable.

TABLE 2. Testbench phases and checks

Phase	Stimulus	Purpose
1	9 directed windows	zero, extremes, ReLU clamp, sign/zero extension
2	500 random windows (fixed seed)	random pixels and signed weights
3	emergency services frames × 900 windows	image data, edge kernel (32, 0, -32)

Reset is held low across three rising edges and released on a falling edge. This removes the identified race-prone pattern; it does not prove that no race exists elsewhere. The earlier testbench is not among the supplied files, so the comparison with it rests on the notes in the earlier manuscript.

VI. FUNCTIONAL VERIFICATION RESULTS

6-A. Compilation and the Final Run

The final transcript was produced with Questa Altera Starter FPGA Edition-64 2025.2 on 14 Oct 2026 by run. do, which compiles the eight RTL files and the testbench and then runs vsin -voptargs=" +acc" vork. ai_accelerator_-tb; the frame count is the testbench

default (911). All eight design files and the testbench compiled with 0 errors and 0 warnings. The only simulator warning is vopt-10908 (opti-misations reduced by +acc), plus a vlib-34 notice that the work library already existed. The first lines of the supplied log belong to an earlier session (they end with "Errors: 2, Warnings: 1") and are not part of this run; the content of that earlier session is not available.

TABLE 3 reproduces the summary printed by the testbench. It agrees with the counter values visible at the end of the run in the waveform window (franes_started and franes_ conpleted 0x38F = 911; vindovs_checked and pass_count 0xC82BC = 819,900; dir_checked 9; rnd_checked 0x1F4 = 500; all failure and error counters 0; latency_ref 0x27 = 39).

6-B. Timing and Latency

The reported latency of 39 cycles needs careful reading. The testbench starts counting after the ninth pixel/weight pair has been driven, i.e., after the last S_L0AD beat, and stops when done is seen. The 39 cycles are the 36 cycles of nine READ–CAPTURE–WAIT–COMPUTE groups plus RELU, STORE and DONE. Counting from the cycle in which start is sampled, done is therefore high 47 cycles later, in the 48th cycle of the window, as derived in Section III-B.

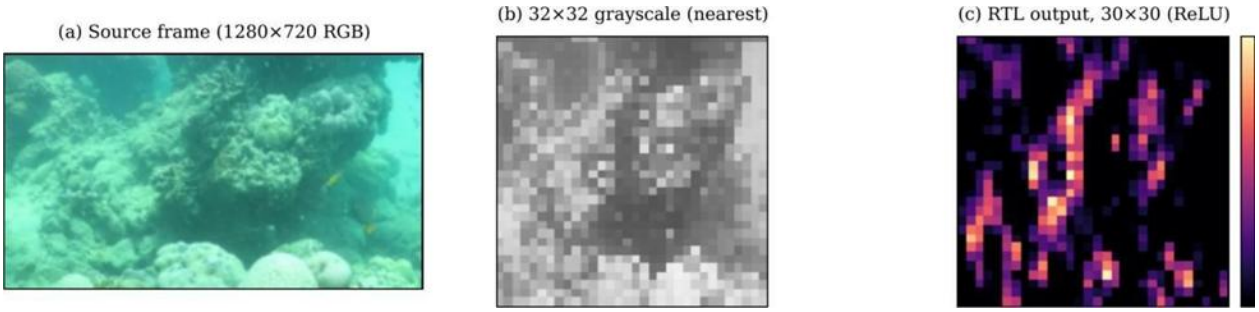

Fig. 3. Preprocessing example carried over from the earlier manuscript (not re-checked against the supplied files): (a) source frame (1280×720 RGB), (b) 32×32 nearest-neighbour grayscale RTL input, (c) 30×30 RTL output after ReLU.

TABLE 3. Final simulation summary (from the supplied log)

Item	Value
Directed windows (fail)	9 (0)
Random windows (fail)	500 (0)
Frames requested / started / completed	911 / 911 / 911
Frame load errors	0
Frame windows checked / passed / failed	819,900 / 819,900 / 0
Timeouts	0
Protocol errors	0
Output-file warnings	0
Reported start → done latency	39 cycles
Final simulation time	402,000,480 ns
Overall status	PASS

The log timestamps give a frame period of exactly 441 μs (for example, frames 1 and 2 finish at 690,460,000 ps and 1,131,460,000 ps), i.e., 490 ns or 49 cycles per window. The extra cycle is idle time introduced by the testbench: after done it spends one falling edge confirming that done has dropped and another on the next idle check before issuing start. The datapath itself needs 48 cycles per window; the supplied test-bench presents one window every 49. The earlier manuscript's 480 ns figure came from the earlier testbench and does not apply to this run. The end time is consistent with this accounting: 509 × 490 ns for the directed and random windows, 50 ns before them, 911 × 441,000 ns for the frames and a final 20 ns give 402,000,480 ns.

6-C. What the Waveform Shows

Fig. 4 is a cropped QuestaSim excerpt from the middle of the run, covering operations 3 to 6 of one window, and it can be checked by hand against the RTL. With the kernel (32, 0, −32) repeated per row, operation 3 uses weight 0x20 and operation 4 weight 0x00. The accumulator steps from 0x3E0 to 0x1880 when the pixel 0xA5 (165) is multiplied by 32 (992 + 5280 = 6272). It holds 0x1880 for the zero weight, and then falls to 0x240 (6272 − 32 × 178 = 576) when the pixel 0xB2 meets the weight 0xE0 (−32). relu_output follows nac_output because the sums are positive. The remaining operations and the final result of this window lie outside the excerpt. The excerpt therefore

documents the sequencing and arithmetic of the datapath; it does not show reset, a start or done pulse, or a completed window result, and it cannot by itself show correctness over the run.

6-D. What the PASS Result Does and Does Not Establish

The log establishes that, for the stimulus applied, all 9 directed, 500 random and 819,900 frame windows produced outputs equal to the in-testbench reference, that every frame was loaded and completed, and that the handshake checks listed above and a constant start-to-done count held without exception. Directed and random windows exercise corner values and random sign and magnitude combinations; the 819,900-frame windows exercise realistic image data with one kernel.

It does not establish more than that. The frame windows come from 911 consecutive frames of a single video and are strongly correlated, so they should not be read as 819,900 independent random tests. The reference was written by the same authors as the RTL, the frame kernel is fixed, and simulation of this kind samples the input space rather than covering it; it is neither formal verification nor equivalence checking. The testbench does not check an absolute latency value, reset during operation, a start issued while busy, or the timeout-recovery path (the timeout count is zero, so that path never ran). No gate-level simulation, FPGA run, silicon or measured data are reported. The frame and output files were not supplied, so the stored RTL outputs could not be inspected, and the earlier manuscript's separate offline comparison with a Python reference (zero mismatches over 911 frames) could not be re-checked.

VII. SYNTHESIS RESULTS (CARRIED OVER, NOT RE-VERIFIED)

The synthesis reports, constraints and netlist were not among the files supplied for this revision. The figures below are taken from the earlier manuscript, which states that they were read from Synopsys Design Compiler reports dated 8 Oct 2026 (a generic standard-cell flow, SAED32 RVT typical corner, 1.05 V, 25°C, wire-load mode, 10 ns ideal clock with 0.2 ns uncertainty and 1 ns input/output delays). They are pre-layout estimates, are not an FPGA flow, and have not been re-checked here. The reports are dated before the 14 Oct simulation, so it is not established that the synthesised RTL is identical to the simulated RTL.

TABLE 4 summarises the values. The earlier manuscript notes that the 1,933 combinational and 848 sequential cells add to 2,781, not 2,783, and that the discrepancy could not be explained; it is repeated here as an open item. The slack is for one setup path with an ideal clock, with no hold analysis, skew, placement or routing, and it is not an achievable-frequency claim. Power was reported from a low-effort, non-annotated analysis with an ideal clock, and leakage accounts for about two thirds of the total.

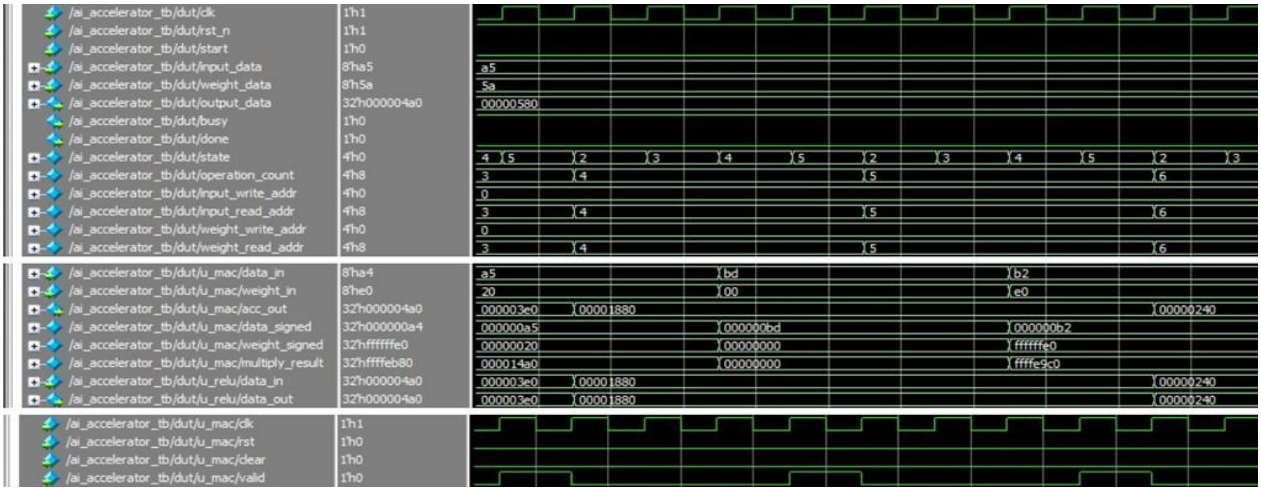

Fig. 4. QuestaSim waveform excerpt (cropped, same time window in all panels, about 211.0502–211.0503 ms, one window of frame 479). Top: DUT ports and control state; state cycles through 2, 3, 4, 5 (READ, CAPTURE, WAIT, COMPUTE) while operation count and the read addresses advance from 3 to 6. Middle: nac_unit operands, product and accumulator, and ReLU output. Bottom: MAC valid strobe, high in the COMPUTE cycles. The value column on the left shows end-of-simulation values and does not correspond to the displayed interval. busy is high and done low throughout this excerpt.

TABLE 4. Synthesis figures from the earlier manuscript (not re-verified)

Metric	Value	Unit
Total cell area	10,397.794	lib. units
Total area incl. interconnect	13,394.677	lib. units
Cells (comb. /seq.)	2,781 (1,933/848)	count
Worst setup slack (1 path)	+7.03 (MET)	ns
Dynamic power	570.742	μW
Leakage power	1.1440	mW
Total power	1.7147	mW

These figures are estimates, not measurements, and without a baseline design or activity-annotated analysis they do not demonstrate low power; no low-power claim is made. Area is dominated by 16-entry buffers of which nine entries are used.

VIII. SEPARATE SOFTWARE EXPERIMENT (CARRIED OVER)

The earlier manuscript also describes a software-only experiment, separate from the RTL, whose source files were not supplied; the numbers below are carried over and only the arithmetic was rechecked.

A logistic-regression classifier was trained on 30 features per frame to predict a manually as-signed Fish or No Fish label for each of the 911 frames (728 training, 183 test; 724 Fish and 187 No Fish in all). On the test set it classified 140 of 145 Fish and 22 of 38 No Fish frames correctly, giving accuracy 162/183 = 88.52%, Fish recall 96.55% and No Fish recall 57.89%.

Always predicting Fish would score 145/183 = 79.23%, so accuracy overstates class separation. The features, scaling, labelling protocol and split are not documented, and because consecutive video frames are strongly correlated, the test set cannot be called in-dependent. This is not a trained CNN, does not show reliable fish detection, and the 88.52% figure is not a property of the RTL Datapath. The earlier manuscript also reports a 97.71% reduction in text-file size when RTL outputs are summarised as compact per-frame statistics; this is only a file-size comparison, and no network latency, energy or bandwidth saving was measured.

IX. LIMITATIONS AND DISCUSSION

The work provides a small, inspectable RTL and verification package, but the evidence is uneven. The main limitations are:

- The RTL is a serial 3×3 MAC-plus-ReLU datapath, not a CNN, classifier or detector; the only accuracy figure belongs to the separate software experiment.
- Verification is by simulation of one testbench. The 911 frames come from one video and are correlated; no for-mal, gate-level, FPGA or silicon evidence exists.
- The testbench does not assert an absolute latency, nor does it exercise reset during operation, start while busy, or timeout recovery. The reported "39-cycle latency" is measured from the last load beat, not from start.
- The reference model shares its kernel and authorship with the stimulus, and the earlier Python comparison cannot be re-checked.
- Synthesis figures are pre-layout, wire-load, ideal-clock, single-corner estimates carried over and not re-verified; there is no hold analysis, activity-annotated power or confirmation that the synthesised and simulated RTL are identical.
- The design is not efficient enough to support a low-power claim: nine of 48 cycles are productive, nine of 16 buffer entries are used, and the output buffer acts as a single register.
- controller_fsn. v and accumulator. v is neither integrated nor validated.
- Aggressive 32×32 down sampling discards most image detail, so edge-filter output says nothing about marine-object detection.

X. CONCLUSION AND FUTURE WORK

A serial 3×3 convolution–ReLU Datapath was implemented in Verilog with a nine-state embedded FSM, unsigned-pixel by signed-weight arithmetic and a 32-bit accumulator. The final simulation log reports a PASS for 9 directed windows, 500 random windows and 911 image frames (819,900 windows) against an in-testbench reference, with no timeouts, proto-col errors or load errors, and a window occupying 48 cycles of the Datapath. This is simulation evidence for the tested stimulus only. Earlier synthesis estimates and the software classification experiment are carried

over and not re-verified. Future work includes adding an absolute latency assertion and directed tests for reset during operation, start while busy and timeout recovery; archiving frame files, output files and the Python comparison; synthesizing the exact simulated RTL withhold analysis and activity-annotated power; physical implementation; integrating or removing the two unused files; narrowing the buffers and Datapath; and a documented video-level classifier evaluation.

ACKNOWLEDGMENT

This work was carried out as part of an internship at the Aazhi Research Laboratory, Department of Electronics and Communication Engineering, National Institute of Technical Teachers Training and Research (NITTTR), Chennai. The authors grate-fully acknowledge the support of NITTTR Chennai.

REFERENCES

[1] M. Zhao, C. Hu, F. Wei, K. Wang, C. Wang, and Y. Jiang, "Real-time underwater image recognition with FPGA embedded system for convolutional neural network," *Sensors*, vol. 19, no. 2, Art. no. 350, Jan. 2019, doi: 10.3390/s19020350.

[2] A. J. Abd El-Maksoud, M. Ebbed, A. H. Khalil, and H. Mostafa, "Power efficient design of high-performance convolutional neural networks hardware accelerator on FPGA: A case study with GoogLeNet," *IEEE Access*, vol. 9, pp. 151897–151911, 2021, doi: 10.1109/ACCESS.2021.3126838.

[3] V. H. Kim and K. K. Choi, "A reconfigurable CNN-based accelerator design for fast and energy-efficient object detection system on mobile FPGA," *IEEE Access*, vol. 11, pp. 59438–59445, 2023, doi: 10.1109/ACCESS.2023.3285279.

[4] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017, doi: 10.1109/JSSC.2016.2616357.

[5] W. Lin, A. Adetomi, and T. Arslan, "Low-power ultra-small edge AI accelerators for image recognition with convolutional neural networks: Analysis and future directions," *Electronics*, vol. 10, no. 17, Art. no. 2048, 2021, doi: 10.3390/electronics10172048.

[6] G. Devic, G. Sassatelli, and A. Gamatié, "A bottom-up methodology for the fast assessment of CNN mappings on energy-efficient accelerators," *Journal of Low Power Electronics and Applications*, vol. 13, no. 1, Art. no. 5, 2023, doi: 10.3390/jlpea13010005.

[7] S. Teerapittayanon, B. McDanel, and H. T. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *Proc. 37th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2017, pp. 328–339.