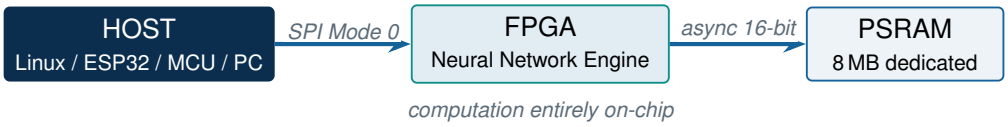
FPGA – Neural

INT8 Neural Network Engine for FPGA

Parametric hardware accelerator – Datasheet and reference manual

Rev. A1 • September 2026



Reference target device: Lattice ECP5 `LFE5U-45F-8BG381C` (speed grade -8, CABGA381, 72×MULT18X18D, ≈44k LUT).

Baseline configuration: INT8/INT32, `N_INPUTS`=256, `N_NEURONS`=4, parametric `PARALLEL`, PSRAM working memory ISSI `IS66WVE4M16EBLL-70BLI`.

Status: RTL verified in simulation (Icarus) and real synthesis (Yosys + nextpnr-ecp5). Document describing the project as of September 2026.

Project author: Michele Bigi • MIKILAB / manvalan.

This datasheet documents the RTL code, documentation and benchmarks present in the repository `github.com/manvalan/FPGA-Neural`.

FPGA-Neural — General description and features

FPGA-Neural is a **parametric hardware accelerator for feed-forward neural networks** contained entirely within the FPGA. Computation (multiplication, accumulation, bias, activation, saturation) takes place entirely on-chip in INT8/INT32 integer arithmetic; the host system only provides configuration, weights, input data and control through a simple SPI interface, without ever being part of the computational datapath. A single bitstream serves any topology up to the build maximum.

Features

- **INT8** $\times$ **INT8** $\rightarrow$ **INT16** $\rightarrow$ **INT32** datapath, 32-bit accumulation with sign extension.
- **Balanced binary adder tree** ($O(\log_2$ PARALLEL$)$) instead of linear reduction.
- Configurable parallel MAC: `PARALLEL` simultaneous hardware MACs per neuron, mapped onto `MULT18X18D` DSPs.
- Fully **parametric** architecture: `N_INPUTS`, `N_NEURONS`, `PARALLEL`, `DATA_WIDTH`, `ACC_WIDTH`, `N_LAYERS`.
- **Runtime network width**: per-layer `n_inputs_real`/`n_neurons_real`, a single bitstream for every topology up to the maximum.
- Configurable activations: `ACT_RELU` (default) and `ACT_NONE` (linear with bilateral saturation), with INT8 saturation.
- **Two network types**: classic multi-layer dense (`layer_sequencer`, ping-pong buffers) and **arbitrary sparse graph** (`graph_engine` + activation buffer in `DP16KD` block RAM), selectable at runtime.
- **Dedicated memory** subsystem: byte↔word interface, asynchronous parallel PSRAM controller (70 ns), 8 MB addressable (23 bit).
- **SPI Mode 0** MSB-first host interface, `SET_NET_TYPE`+dispatch, `STATUS.done` sticky/clear-on-read, runtime `READ_CONFIG`.
- Verified in **simulation** (Icarus Verilog) and **real synthesis** (Yosys + nextpnr-ecp5 + ecppack).

Applications

- Deterministic low-latency inference as a peripheral of a Linux SoC, Raspberry-Pi-like board, ESP32, microcontrollers.
- Reusable hardware block integrable into heterogeneous projects (a platform, not a single network).
- Edge AI on compact dense INT8-quantized networks.
- Off-loading the neural workload from the host CPU to dedicated hardware with predictable throughput.

Target & toolchain

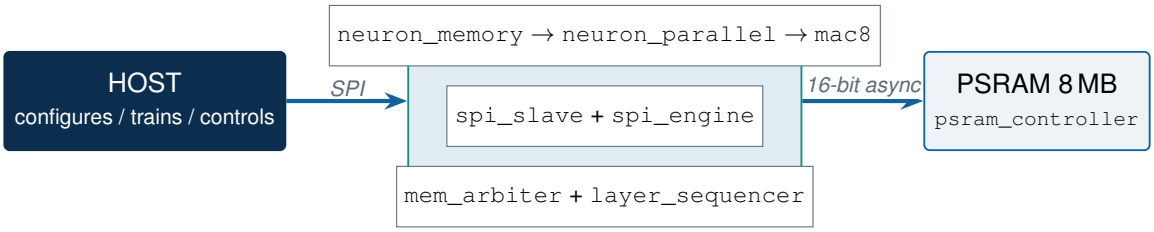
- FPGA: Lattice ECP5 `LFE5U-45F-8BG381C` (-8, CABGA381).
- Synthesis: Yosys; place&route: nextpnr-ecp5; bitstream: Project Trellis (`ecppack`).
- Simulation: Icarus Verilog (`-g2012`).
- PSRAM: ISSI `IS66WVE4M16EBLL-70BLI` (64 Mb, 4M×16).

Key parameters (characterized baseline configuration)

Quantity	Value	Notes
Data precision	INT8 (signed)	`DATA_WIDTH`=8
Accumulator	INT32 (signed)	`ACC_WIDTH`=32
Inputs / neurons	256 / 4	datapath benchmark baseline
Simultaneous MACs	$2 \ldots 64$	=`PARALLEL`×`N_NEURONS`
Activations	ReLU, linear	`ACT_RELU` / `ACT_NONE`
Fmax (P=2, datapath)	87.88 MHz	isolated datapath benchmark
Fmax (P=2, integrated system)	75.30 MHz	after timing closure, real place&route
MAC throughput (P=16)	≈3.34 G MAC/s	theoretical, datapath only
Working memory	8 MB PSRAM	16-bit parallel bus, 70 ns

Address space	23 bit (byte)	`ADDR_WIDTH`=23

System block diagram



The neural datapath is entirely inside the FPGA; the host does not take part in the individual MAC operations.

Contents

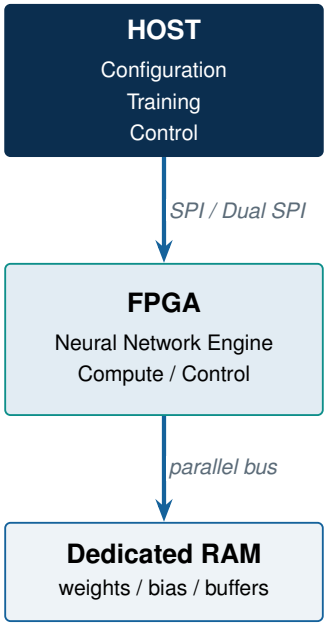
CHAPTER 1

System overview

1.1 Project goal

FPGA-Neural implements a **reusable Neural Network Engine in FPGA hardware**. The whole is made of three elements: the FPGA, which is the actual accelerator; a dedicated RAM physically associated with the FPGA and not shared with the host; and a host interface independent of the operating system, initially SPI (with possible future extension to Dual SPI).

The founding principle is the separation between who *executes* the computation and who *uses* it: the neural network computation happens entirely inside the FPGA, while the host system only provides configuration, network parameters, input data, control and result readback. The host is not part of the computational datapath. Possible host systems include Linux SoCs, Raspberry Pi-like systems, ESP32, microcontrollers and development PCs: the same engine architecture must be usable in completely different systems.



1.2 Hardware configuration versus network configuration

The project draws a precise distinction between the accelerator's **hardware architecture** and the **neural network parameters**.

The physical architecture of the engine is defined at FPGA synthesis and implementation time. Typical hardware parameters are `N_INPUTS`, `N_NEURONS`, `N_LAYERS`, `PARALLEL`, `DATA_WIDTH`, `ACC_WIDTH`: they are Verilog parameters resolved at synthesis and they determine the datapath contained in the bitstream. The network parameters — weights, bias, activation and quantization parameters, specific constants — are instead loaded at runtime through the host interface and stored in the RAM associated with the FPGA.

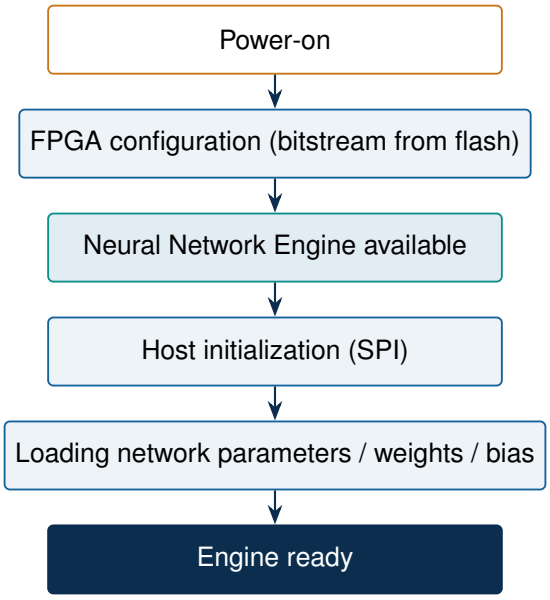
Central architectural principle

A build fixes the *ceiling* of the machine (maximum number of layers, maximum width, `PARALLEL`); the host configures the *actual* network — number of layers, per-layer input/output width, per-layer activation and trained parameters — entirely at runtime, over SPI, into the

FPGA's local memory. A single bitstream serves any topology up to that ceiling.

1.3 Boot and initialization

The FPGA is configured at power-on through the usual configuration mechanism (bitstream loading from SPI flash). The bitstream defines the hardware architecture of the engine; the host does not dynamically build the datapath during normal operation, but rather configures the network data on which the already existing datapath operates.



1.4 Training and inference

Training and inference are conceptually separate. The first implementation does not require the FPGA to perform training: weights can be computed externally (PC/Linux/other host) and transferred over SPI into the FPGA's RAM, which then performs inference. This drastically reduces the complexity of the initial hardware, without precluding a future implementation of assisted or fully hardware training (roadmap Phase 8, ch. 15). During inference the host only provides the input data and retrieves the result, obtaining deterministic computation, reduced host load, hardware parallelism, predictable latency and independence from the host CPU architecture.

1.5 Design philosophy and reuse

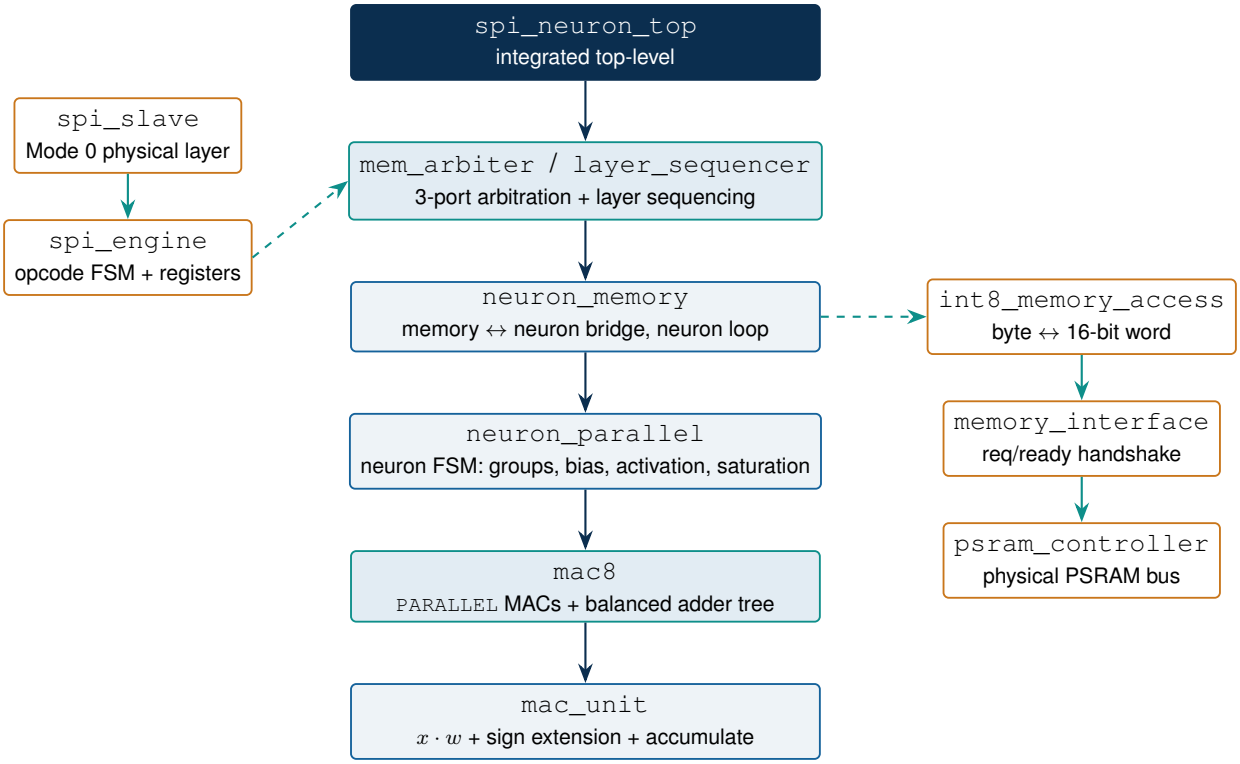
The project should be understood as a *reusable FPGA neural acceleration platform* rather than a single network. The application determines input size, topology, number of layers and neurons, parallelism, numeric precision, activation functions, memory and performance requirements; the hardware generation process produces the corresponding FPGA implementation. The same HDL architecture remains conceptually unchanged while the synthesis parameters generate implementations appropriate to the different application targets.

CHAPTER 2

RTL architecture and module hierarchy

2.1 Hierarchical organization

The design is organized in layers, from the elementary multiply-accumulator up to the integrated top-level with SPI interface and PSRAM. Each layer encapsulates the previous one and abstracts away its details: the validated datapath (`mac_unit`, `mac8`, `neuron_parallel`) is never modified by the higher orchestration layers.



2.2 Role of each module

Module	Function
`mac_unit`	Single multiply-accumulate: $\text{acc\_out} = \text{acc\_in} + (x \cdot w)$, with sign extension of the product to `ACC_WIDTH`. Parametric on `DATA_WIDTH`/`ACC_WIDTH`.
`mac8`	`PARALLEL` instances of `mac_unit` whose products are summed by a *balanced binary adder tree* of depth $\log_2(\text{PARALLEL})$; the result is added to the input accumulator.
`neuron_parallel`	FSM of a single neuron: processes `N_INPUTS` inputs in groups of `PARALLEL`, accumulates across groups, adds the bias, applies the activation and saturates to INT8. Includes the processing guard on `N_INPUTS % PARALLEL` and the runtime width `n_inputs_real`.

`layer`	Instantiates `N_NEURONS` neurons *in parallel* on the same input vector; `busy`=OR, `done`=AND of the neurons. A purely data-combinational path used in the datapath benchmarks.
`neuron_memory`	Integrates computation with memory: reads X (shared) once, then for each neuron re-reads W and bias from RAM and reuses a single `neuron_parallel` instance (memory-bound, one neuron at a time). Output `y_bus` packed neuron-major.
`layer_sequencer`	Chains up to `N_LAYERS` executions of `neuron_memory` by reading a descriptor table written by the host and alternating the ping-pong buffers in RAM (Phase 5).
`act_buffer`	Global activation buffer in `DP16KD` block RAM, indexed by signal id (Type #2).
`graph_engine`	Graph-network engine (Type #2): gather from `act_buffer`, reuses `neuron_parallel`, writes outputs by id (ch. 7).
`int8_memory_access`	Converts the byte/INT8 interface (byte address) into the 16-bit word interface, selecting the low/high byte via `lb_n`/`ub_n` and `addr»1`.
`memory_interface`	2-state handshake FSM (IDLE/WAIT) that serializes the single transaction toward the controller.
`psram_controller`	Asynchronous parallel PSRAM bus controller: INIT/IDLE/READ/WRITE sequence with 70 ns timing derived from `CLK_FREQ_MHZ`; drives `ce_n/oe_n/we_n/lb_n/ub_n/zz_n` and the tri-state data bus.
`mem_arbiter`	Fixed-priority arbiter (B>C>A) among three byte-level masters: `spi_engine` (A), `neuron_memory` (B), `layer_sequencer` (C).
`spi_slave`	SPI Mode 0 physical layer, MSB-first, 3-stage CDC synchronizer on SCLK/MOSI/CS_N, shift register and CS framing.
`spi_engine`	Protocol/opcode FSM and register bank (`x_base`, `w_base`, `bias_addr`, ping-pong base, activation, runtime widths…), with sticky/clear-on-read `STATUS.done`.
`spi_neuron_top`	Top-level: connects SPI, arbiter, sequencer, `neuron_memory` and the PSRAM chain; multiplexes control of `neuron_memory` between the sequencer and the direct single-layer path.

Simulation models

`psram_model.v` (in `sim/`) and `memory_model.v` are behavioral memory models used in the testbenches; they are not part of the synthesizable design but they reproduce the real latency for end-to-end verification.

2.3 Two execution paths

The top-level exposes two mutually exclusive modes toward the same `neuron_memory` compute engine:

- **Single-layer / manual path**: the host sets the bases with `SET_BASE`, starts with `START` and reads with `READ_OUTPUT`. `spi_engine` drives `neuron_memory` directly.
- **Multi-layer path**: the host writes the descriptor table and starts with `RUN_NETWORK`; `layer_sequencer` takes over control of `neuron_memory` (while `seq_busy` is high) and chains the layers.

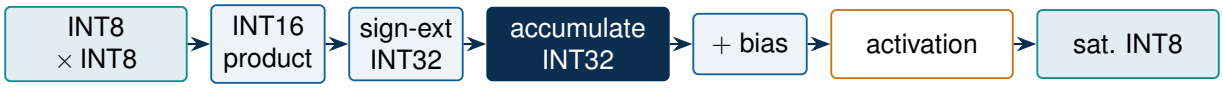
The top-level multiplexer switches the control lines of `neuron_memory` based on `seq_busy`, returning the engine to the direct path at the end of the sequence.

CHAPTER 3

Compute datapath

3.1 INT8/INT32 arithmetic chain

The elementary datapath implements the typical sequence of a quantized neuron:



Each INT8×INT8 product fits in 16 bits; it is sign-extended to 32 bits before accumulation, so the accumulator does not overflow on long vectors. Bias and activation operate at 32 bits; only the final output is saturated to INT8.

3.2 `mac_unit` — multiply-accumulator

The `mac_unit` module is purely combinational and parametric on `DATA_WIDTH` and `ACC_WIDTH`. It computes:

$$\text{acc\_out} = \text{acc\_in} + \text{signext}_{ACC}(x \cdot w)$$

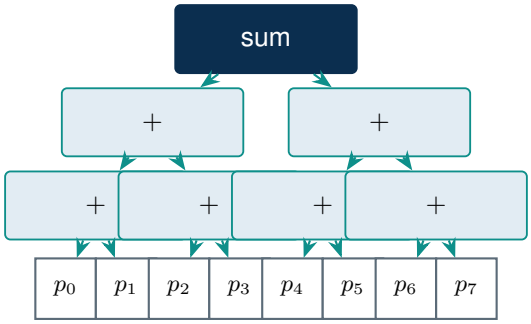
The product has width $2\times$`DATA_WIDTH` and is sign-extended by replicating the most significant bit. On ECP5 the multiplication maps onto a `MULT18X18D` DSP block.

Listing 3.1. `rtl/mac_unit.v` — arithmetic core

```
localparam PROD_WIDTH = 2 * DATA_WIDTH;
wire signed [PROD_WIDTH-1:0] product = x * w;
wire signed [ACC_WIDTH-1:0]  product_ext =
    {{(ACC_WIDTH-PROD_WIDTH){product[PROD_WIDTH-1]}}, product};
assign acc_out = acc_in + product_ext;
```

3.3 `mac8` — parallel MAC and balanced adder tree

`mac8` instantiates `PARALLEL` `mac_unit` units that generate `PARALLEL` independent products, then sums them with a *balanced binary adder tree*. Compared to the linear reduction $((((p_0+p_1)+p_2)+p_3)+\dots)$, of depth $O(\text{PARALLEL})$, the tree has depth $O(\log_2 \text{PARALLEL})$, drastically reducing the combinational path.



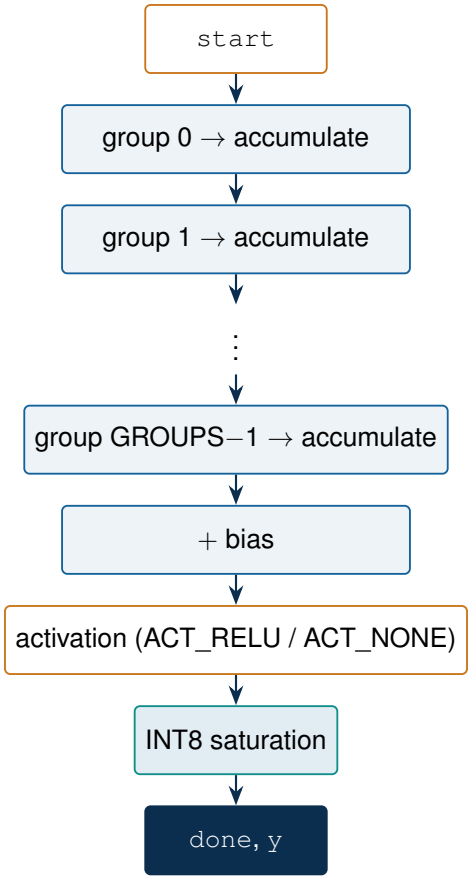
Example with PARALLEL=8: 3 levels. PARALLEL=16 → 4 levels; PARALLEL=32 → 5 levels.

PARALLEL as a power of two

The tree is designed for `PARALLEL` as a power of two (8, 16, 32...). This is also the value used in all project configurations.

3.4 `neuron_parallel` — neuron FSM

`neuron_parallel` processes `N_INPUTS` inputs in groups of `PARALLEL`, maintaining the accumulator from one group to the next. At the end it adds the bias, applies the activation and saturates to INT8. The number of groups is GROUPS $=$ N_INPUTS/PARALLEL.



3.4.1 Parameter guard (elaboration-time)

If `PARALLEL` does not exactly divide `N_INPUTS` two failures occur, both confirmed empirically in `sim/parameter_sweep_tb.v`:

- integer division truncates `GROUPS` and the excess inputs are never read → **wrong** result, with no error and no warning;
- if `PARALLEL > N_INPUTS`, `GROUPS=0` and the terminal condition is never satisfied → the neuron **hangs** (busy high, done never asserted).

The solution does not modify the validated datapath: a `generate` block instantiates a deliberately undefined module when N_INPUTS $\bmod$ PARALLEL $\neq 0$, forcing an error at *elaboration* both in simulation and in synthesis. For valid configurations the branch is never elaborated.

Listing 3.2. `rtl/neuron_parallel.v` — parameter guard

```
generate
  if (N_INPUTS % PARALLEL != 0) begin : PARAMETER_ERROR
     neuron_parallel_requires_N_INPUTS_multiple_of_PARALLEL
        invalid_parameter_combination();
  end
endgenerate
```

3.5 Activation functions

`neuron_parallel` accepts a 2-bit `activation` port. The default is `ACT_RELU`, the only behavior that existed before the port was introduced, so every pre-existing caller remains unchanged.

Encoding	Value	Behavior
`ACT_NONE`	`2'd0`	Linear: no clamp to zero, bilateral saturation to the INT8 range $[-128, +127]$.
`ACT_RELU`	`2'd1`	$\max(0, x)$, then positive saturation to $+127$ (default; also the fallback for reserved encodings).

3.6 INT8 saturation

After bias and activation, the 32-bit accumulator is reduced to INT8:

$$y = \begin{cases} +127 & \text{if } \text{final\_acc} > 127 \\ -128 & \text{if } \text{final\_acc} < -128 \text{ (ACT\_NONE only)} \\ 0 & \text{if } \text{final\_acc} \leq 0 \text{ (ACT\_RELU only)} \\ \text{final\_acc}[7:0] & \text{otherwise} \end{cases}$$

3.7 `layer` — neurons in parallel

`layer` instantiates `N_NEURONS` neurons that share the input vector `x_bus` but have distinct weights and bias; `busy` is the OR and `done` the AND of the neurons' signals. It is the module used in the datapath benchmarks (ch. 12), where all neurons work simultaneously. The addressing convention is neuron-major: the weights of neuron n occupy `weights_bus[n*N_INPUTS*DATA_WIDTH +: N_INPUTS*DATA_WIDTH]`.

CHAPTER 4

Parameters and configurability

4.1 Build parameters (synthesis-time)

The hardware architecture is fixed at synthesis through the following Verilog parameters. They determine the datapath contained in the bitstream and its capacity *ceiling*.

Parameter	Default	Meaning
`DATA_WIDTH`	8	Data width (INT8).
`ACC_WIDTH`	32	Accumulator width (INT32).
`N_INPUTS`	32 / 256	Maximum number of inputs per neuron (benchmark baseline: 256).
`N_NEURONS`	1 / 4	Maximum number of neurons per layer.
`PARALLEL`	8	Simultaneous hardware MACs per neuron; must divide `N_INPUTS` and should be a power of two.
`N_LAYERS`	4	Maximum number of layers chainable by `layer_sequencer`.
`ADDR_WIDTH`	23	Byte-address width (8 MB).
`MEM_DATA_WIDTH`	16	Width of the physical PSRAM data bus.
`CLK_FREQ_MHZ`	80	Frequency used in the PSRAM timing formulas (must be aligned to the real oscillator).

`N_INPUTS` % `PARALLEL` constraint

`PARALLEL` must divide `N_INPUTS` exactly, otherwise the elaboration guard fires (§3). The same constraint applies at runtime to `n_inputs_real`.

4.2 Runtime network width

A single bitstream serves any topology *up to* the build maximum. The actual width of each execution is a separate value, set by the host:

- `n_inputs_real` — inputs actually used in this execution (must be a multiple of `PARALLEL`);
- `n_neurons_real` — neurons actually computed in this execution.

Both default to the build maximum, so any caller that leaves them unconnected processes the full width as before the ports were introduced.

Real early termination

This is not mere address bookkeeping: the two values directly bound the hardware loops (X/W reads of `neuron_memory`, MAC group count of `neuron_parallel` and the length of the ping-pong copy for `RUN_NETWORK`). A narrower layer actually *computes* and *copies* faster and does not require zero-padding of the RAM for the unused tail: data beyond `n_inputs_real`/`n_neurons_real` is never read.

This lets a network taper within a single chained execution, for example $256 \to 64 \to 16 \to 4$, with each layer declaring its own actual width in the descriptor table (ch. 6).

4.2.1 Measured savings

Early termination was measured end-to-end:

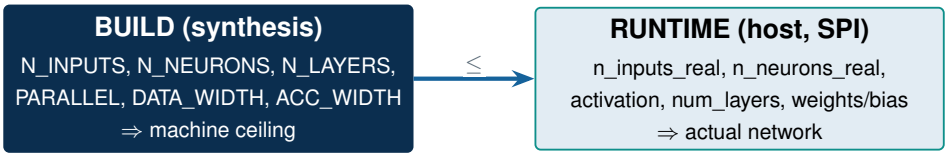
Test	Cycles	Comparison
`neuron_parallel_tb.v` (T7)	3 vs 6	reduced vs full, with “garbage” data in the skipped lanes (proof that they are not read).
`neuron_memory_tb.v` (T5)	209 vs 788	8-of-32 vs full 32, through the real PSRAM stack.

4.3 Characterized configurations

Some combinations validated in simulation and/or synthesis:

N_INPUTS	N_NEURONS	PARALLEL	Notes
32	4	8	First functional parametric test (Phase 1).
256	4	2/4/8/16	Datapath benchmark sweep (Phase 7).
32	1..3	8	Single/multi-neuron memory integration (Phase 3).
4	4	2	End-to-end 2-layer `RUN_NETWORK` test over real SPI.

4.4 Build versus runtime summary

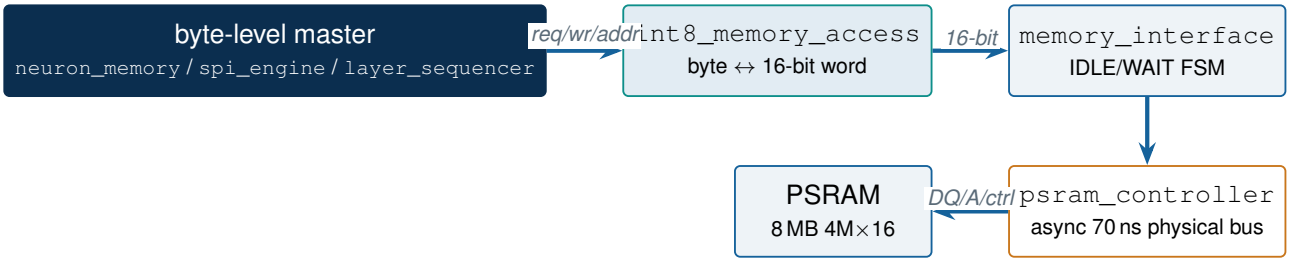


CHAPTER 5

Memory subsystem

5.1 Memory chain

The compute engine works with addresses and data at the *byte* level (INT8), while the PSRAM is a 16-bit word device. Three cascaded modules realize the conversion and the physical access:



5.2 `int8_memory_access` — byte/word conversion

Converts the INT8 interface (byte address) into the word interface. The byte address is divided by two (`addr»1`) to obtain the word address; the least significant bit selects the byte:

- `addr[0]=0` → low byte: `lb_n=0`, `ub_n=1`, data on DQ[7:0];
- `addr[0]=1` → high byte: `lb_n=1`, `ub_n=0`, data on DQ[15:8].

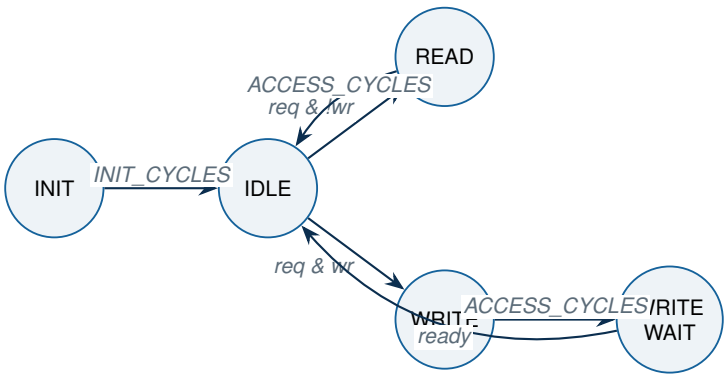
On read it extracts the correct byte from `mem_rdata`. The FSM has two states (IDLE, WAIT) and returns `ready` as a one-cycle pulse.

5.3 `memory_interface` — handshake

Two-state FSM that serializes a single transaction: in IDLE, on the `req` request, it latches `wr/addr/wdata/lb_n/ub_n` and emits a one-cycle `mem_req` pulse toward the controller; in WAIT it waits for `mem_ready`, captures `rdata` on read and asserts `ready`. It guarantees the “one transaction at a time” contract.

5.4 `psram_controller` — physical bus

Asynchronous parallel PSRAM bus controller. The state machine is:



5.4.1 Timing

Timing formulas

ACCESS_CYCLES $= \lceil (70 \times \text{CLK\_FREQ\_MHZ})/1000 \rceil$ (70 ns access latency)
INIT_CYCLES $= 150 \times$ CLK_FREQ_MHZ (power-up initialization)

The data bus is tri-state driven: `psram_dq = dq_oe ? dq_out : Z`. On read `dq_oe=0`; on write `dq_oe=1` during the `we_n` pulse. A WRITE_WAIT state keeps `ce_n/lb_n/ub_n` active for the final hold before release.

This is not QSPI

This is a classic asynchronous-SRAM interface, **not** QSPI: most commercial serial/QSPI "PSRAM" parts are not compatible with this controller without a rewrite. See ch. 13 for the recommended part (parallel ISSI).

5.5 Address map and conventions

The addressing space is `ADDR_WIDTH`=23 bits (*byte* address), for a full 8 MB. The regions do not have hardwired addresses: their bases are registers set by the host via `SET_BASE` (single-layer path) or read from the descriptor table (multi-layer path).

Region	Base	Content / convention
Input X	`x_base`	Shared input vector, read once per invocation.
Weights W	`w_base`	Neuron-major: weights of neuron n at `w_base + n*N_INPUTS` bytes.
Bias	`bias_addr`	One byte per neuron: bias of neuron n at `bias_addr + n`.
Descriptor table	`table_base`	`N_LAYERS` 11-byte entries (ch. 6).
Ping-pong buffers A/B	`buf_a_base` / `buf_b_base`	Intermediate outputs between layers.

5.5.1 PSRAM physical addressing

The recommended PSRAM is 4M×16 (8 MB), which requires a 22-bit word address (A0–A21). `int8_memory_access` computes `addr»1`, turning the 23-bit byte address into a 22-bit word address that maps exactly onto A0–A21; bit 22 of `psram_a` is therefore always 0 and 22 real address lines remain on the PCB.

5.6 Bandwidth

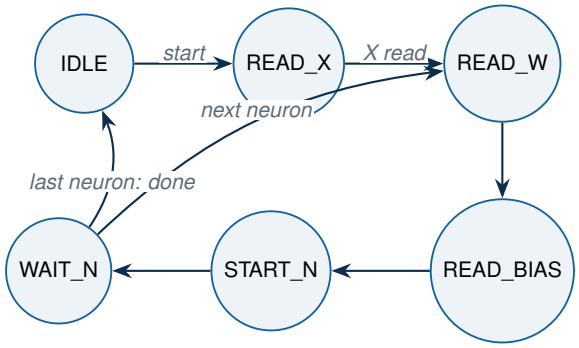
Bandwidth analysis against the real PSRAM timing is deferred to Phase 7. The current model is memory-bound by construction: `neuron_memory` reads X once and re-reads W/bias for each neuron (ch. 6), one neuron at a time, without modifying the validated compute datapath.

CHAPTER 6

Memory integration, multi-neuron and multi-layer

6.1 `neuron_memory` — memory/neuron bridge

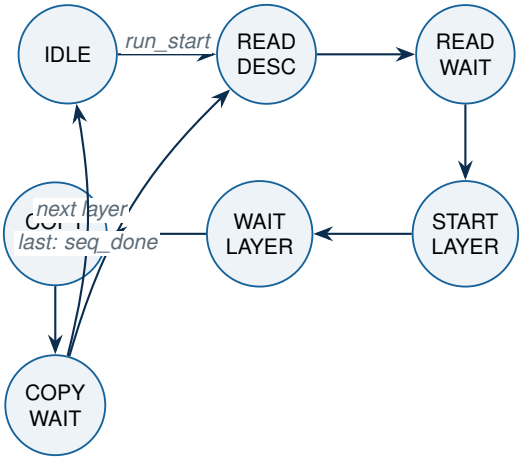
`neuron_memory` connects the compute datapath to memory and manages the loop over the neurons. It reads the X vector only once (shared input), then for each neuron re-reads W and bias from RAM and feeds them to a single reused instance of `neuron_parallel`: the design is memory-bound, one neuron computed at a time, without duplicating the datapath. The output is `y_bus`, packed neuron-major (`DATA_WIDTH*N_NEURONS` bits).



The states are IDLE, READ_X, READ_W, READ_BIAS, START_N, WAIT_N. After the last neuron the FSM returns to IDLE and asserts `done`. The count of neurons and inputs actually processed is given by `n_neurons_real`/`n_inputs_real` (ch. 4).

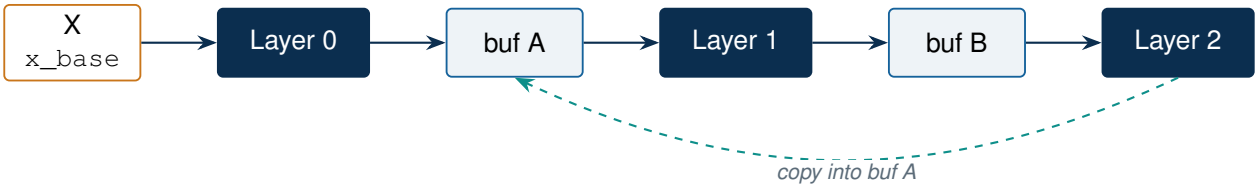
6.2 `layer_sequencer` — multi-layer network

`layer_sequencer` chains up to `N_LAYERS` executions of the same `neuron_memory` instance, realizing a dense feed-forward network *without* touching the validated compute core. It reads a descriptor table written by the host and alternates the two output buffers in RAM (ping-pong).



6.2.1 Ping-pong buffers

Layer 0 reads the external input `x_base`. Layer $k > 0$ reads from the buffer written by layer $k - 1$; the output of each layer is copied into the other buffer, alternating A and B. The final output remains both in `y_bus` (readable with `READ_OUTPUT`) and in the ping-pong buffer into which it was copied.



6.2.2 Descriptor table

Written by the host into RAM at `table_base` with `WRITE_RAM`; `N_LAYERS` entries of 11 bytes each, MSB-first:

Field	Bytes	Meaning
`w_base`	3	Weight base of the layer.
`bias_addr`	3	Bias base of the layer.
`activation`	1	Layer activation (low 2 bits, cf. `ACT_*`).
`n_inputs_real`	2	Actual inputs of the layer (multiple of `PARALLEL`).
`n_neurons_real`	2	Actual neurons of the layer.
Total	**11**	per entry/layer

Copy proportional to the actual width

The sequencer copies exactly `n_neurons_real` bytes of `y_bus` into the ping-pong buffer (not the full build width): a narrower layer is copied faster, without zero-padding in RAM. Each activation is read per-layer from the table, independent of the `activation` register of the single-layer path.

6.3 Hierarchy of the `busy`/`done` signals

In the multi-layer path, `STATUS.busy` is the OR of the single-layer and sequencer busy signals, while `STATUS.done` latches only at completion of the *last* layer, not at each intermediate layer (ch. 8). The top-level returns control of `neuron_memory` to the direct `START` path at the end of the sequence.

CHAPTER 7

Two-level configuration: graph network (Type #2)

7.1 Two network types

The engine exposes two *network types* selectable by the host, with the same start command dispatching to the correct engine:

- **Type #1 — classic network (dense).** Layers with neurons per layer, fully connected between consecutive layers. It is the `layer_sequencer` path (ch. 6), started by `RUN_NETWORK`. Connections are *implicit by position*: nothing is enumerated, only the weights are defined, addressed as `w_base + k*n_inputs + j`.
- **Type #2 — arbitrary graph (sparse).** Starting from the input neuron ids, each neuron's connections up to the output are defined through a per-neuron *sparse edge-list*. Connections are *explicit by enumeration*: each connection is an edge `(src_id, weight)`; if it is not in the list, it does not exist.

The difference in one line

Dense: you define the *weights* by position in a matrix. Graph: you define each *connection* as an edge `(src_id, weight)` in a per-neuron list. The two descriptor tables share the same 11-byte format but different fields; the `net_type` register tells the engine which interpretation to use.

7.2 Global activation buffer

Type #2 introduces an **activation buffer** indexed by *signal id*, one INT8 byte per id, implemented in **on-chip `DP16KD` block RAM** (`rtl/act_buffer.v`). Ids `0..N_in-1` are the inputs; each neuron writes its own output into its own id. The source gather reads from here with *single-cycle random access*: this is what makes the graph cheap, because it is the access that PSRAM (70 ns, sequential) could not accelerate.

V1 sizing

`N_TOTAL`=4096 signals, 16-bit id (room to 65 536 without changing the format). Buffer = 4 KB, i.e. 2 `DP16KD` blocks out of 108. The real constraint becomes the PSRAM edge capacity (≈2 M edges at 4 B), not block RAM.

7.3 Feed-forward DAG and the `src_id < out_id` rule

The graph is a feed-forward DAG: every connection points to an **already-computed** id (`src_id < out_id`). Neurons are processed in ascending id order, so that when a neuron is computed all its sources are ready in the buffer. Cycles and recurrence are out of scope for V1. The rule is checked at two levels: by the host assembler (compile time) and by a runtime guard in `graph_engine` (`STATUS.err`), in the same philosophy as the elaboration guard on `N_INPUTS % PARALLEL`.

7.4 Data formats

Both descriptors are 11 bytes/entry, MSB-first, at `table_base`.

7.4.1 Type #2 descriptor (graph)

Field	Bytes	Meaning
`conn_ptr`	3	Byte address in PSRAM of the neuron's edge block.
`n_conn`	2	Real connections (pre-padding).
`out_id`	2	Id into which the neuron's output is written.
`activation`	1	`ACT_RELU` / `ACT_NONE` (low 2 bits).
`bias`	1	Neuron bias (INT8).
`reserved`	2	0.
Total	**11**	entries in ascending `out_id` order

7.4.2 Graph edge (4 bytes, aligned)

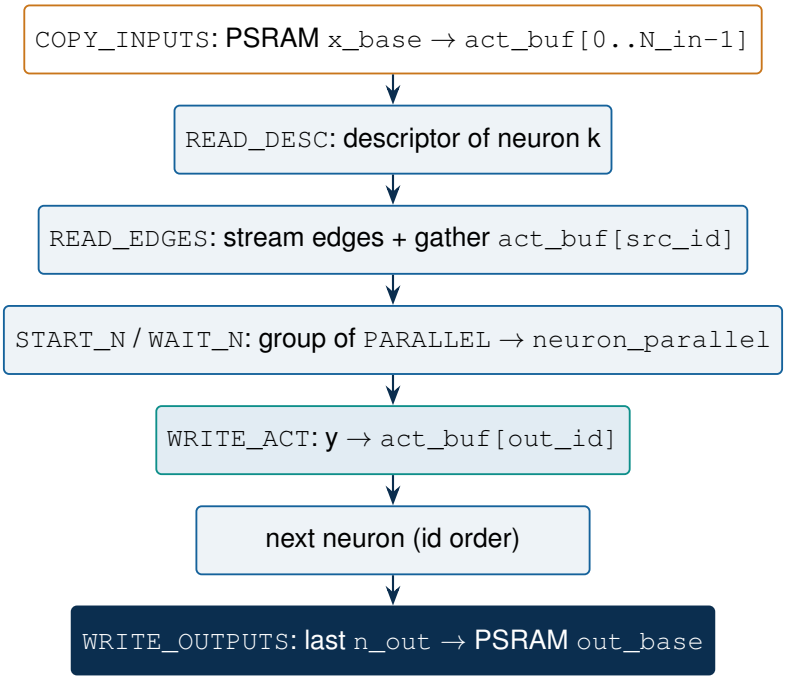
Field	Bytes	Meaning
`src_id`	2	Source id (uint16 BE).
`weight`	1	Weight (INT8).
`reserved`	1	0 (4-byte alignment).

Padding to `PARALLEL`

An arbitrary `n_conn` is not a multiple of `PARALLEL`: the neuron's edge-list is padded up to the multiple with **zero-weight** edges (waste $\leq$`PARALLEL`-1 per neuron). This keeps the datapath and its guard intact.

7.5 `graph_engine` — graph engine

`rtl/graph_engine.v` orchestrates Type #2 **reusing `neuron_parallel` unmodified**, as `neuron_memory` does for the dense case. Key difference: between the two modes only the *X addressing* changes. In Type #1 the input is contiguous (`x_base + i`); in Type #2 it is a gather (`act_buf[src_id]`). The arithmetic core is untouched.



The outputs are the **last `n_out`** ids: in a DAG with the `src_id < out_id` ordering the output neurons (sinks, not reused as sources) naturally end up with the highest ids. At the end `graph_engine` copies these `n_out` bytes into a PSRAM region at `out_base`, which the host reads back with `READ_RAM`.

7.6 Type #2 opcodes and registers

The type is selected with a new opcode; `RUN_NETWORK` dispatches on the `net_type` register (details in ch. 8).

Opcode / sel	Name	Function
`0x11`	SET_NET_TYPE	`type(1B)`: `0x01`=dense (#1), `0x02`=graph (#2). Default after `RESET`=dense.
`SET_BASE sel 9`	num_neurons_graph	Number of graph neurons (uint16).
`SET_BASE sel 10`	n_out	Number of output ids (uint16).

Zero regression on Type #1

With `net_type=dense` (the default value after `RESET`) the #1 path is bit-identical to before: `RUN_NETWORK` keeps its `num_layers(1B)` payload and the framing of the existing opcodes does not change.

7.7 Occupancy (Type #2 enabled)

Yosys synthesis of the full `spi_neuron_top` system with Type #2 enabled (`PARALLEL`=2):

Resource	Use
`DP16KD` (block RAM)	2 (activation buffer)
`MULT18X18D` (DSP)	4 (2 `neuron_memory` + 2 `graph_engine`)

LUT4	2367
TRELLIS_FF	2406
`$_TBUF_` (PSRAM bus)	16

The device (108 `DP16KD`, 72 DSP, ≈44k LUT/FF) stays well below saturation: Type #2 adds a complete mode at a contained resource cost.

7.8 Gather bandwidth (measured)

The per-edge gather cost was **isolated** by building two structurally-identical graphs with different edge counts and differencing the cycles: the subtraction cancels the fixed per-neuron overhead and leaves the edge cost alone.

Per-edge cost

53.25 cycles/edge, consistent with theory: 4 bytes/edge × ≈13 cycles/byte over async PSRAM ≈52. At 80 MHz: ≈1.5 M edges/s (≈6.0 MB/s); at the real 16 MHz clock: ≈300 k edges/s (≈1.2 MB/s).

Each edge today costs a full PSRAM round-trip: it is the sequential access that page-mode read (roadmap G7, ch. 15) would accelerate. The measurement turns that optimization from a vague opening into a **quantifiable decision**: for a network of a given size one can estimate up front whether the bandwidth gain is worth the effort.

7.9 Host compiler

Readable network configuration needs no dedicated FPGA logic: the network is described in **YAML** and a host compiler translates it into the exact bytes of the tables and edges, then loaded with `WRITE_RAM`. The compiler validates at compile time (`src_id < out_id`, `N_TOTAL` bounds, padding to `PARALLEL`), complementing the runtime guard. Format, rules and examples in ch. 10.

CHAPTER 8

SPI host interface

8.1 Physical layer

The FPGA is always an SPI **slave**. The v1 protocol uses SPI **Mode 0** (CPOL=0, CPHA=0), MSB-first, single-SPI. One command per low-CS period; byte 0 of each transaction is the opcode. Multi-byte fields are big-endian.

Mode 0 sampling

`mosi` is sampled on the **rising** edge of `sclk`; `miso` is driven on the **falling** edge (stable before the master's next sampling). `spi_slave` synchronizes `sclk/mosi/cs_n` with a double flip-flop (3-stage CDC) before every edge detection.

Framing of one byte: CS falls, 8 SCLK pulses, MSB first; MISO in tri-state outside a transaction.

`tx_byte_req` contract

`tx_byte_req` is a *prefetch hint*, not a "byte consumed" event: a consumer must advance its pointers (RAM address, response byte index) on `rx_valid`, which pulses exactly once per real byte transferred.

8.2 Framing and explicit length

The length of RAM transfers is **explicit**, not delimited by the CS edge: `WRITE_RAM`/`READ_RAM` carry a 2-byte length field, so the SPI controller only needs a byte counter. Byte addresses are 23-bit, carried in a 3-byte field with the most significant bit reserved to 0.

8.3 Opcode table

Op	Name	Payload (host→FPGA)	Response	Function
`0x00`	NOP	—	—	No operation (idle/dummy clocking).
`0x01`	WRITE_RAM	addr(3B)+len(2B)+data	—	Writes a block into PSRAM (X, weights, bias, parameters).
`0x02`	READ_RAM	addr(3B)+len(2B)	`len` bytes	Reads a block back from PSRAM.

Op	Name	Payload	Response	Function
0x0F	RESET	—	—	Synchronous reset of the engine and clearing of the STATUS latch; does not erase PSRAM.
0x10	SET_BASE	sel(1B)+addr(3B)	—	Sets the bases/registers (see §8.4).
0x11	SET_NET_TYP	type(1B)	—	Network type: 0x01=dense (#1), 0x02=graph (#2). Default after RESET=dense.
0x20	START	—	—	Starts neuron_memory (single-layer path); ignored if busy.
0x21	STATUS	—	1 byte	bit0=busy (live), bit1=done (sticky, clear-on-read), bit2=err (graph guard); bit7:3=0.
0x22	READ_OUTPU	—	N_NEURONS bytes	y_bus neuron-major (byte 0 = neuron 0).
0x23	RUN_NETWORK	num_layers(1B)	—	Starts execution: dispatches on net_type to layer_sequencer (#1) or graph_engine (#2); ignored if busy.
0x30	READ_CONFIC	—	11 bytes	Hardware configuration record (§8.7).

8.4 SET_BASE selectors

sel	Register	Use
0	x_base	Input base X.
1	w_base	Weight base.
2	bias_addr	Bias base.
3	table_base	Descriptor table base (multi-layer).
4	buf_a_base	Ping-pong buffer A.
5	buf_b_base	Ping-pong buffer B.
6	activation	Activation (low 2 bits) — single-layer path only.
7	n_inputs_real	Runtime input width (16-bit BE) — single-layer.
8	n_neurons_real	Runtime neuron width (16-bit BE) — single-layer.
9	num_neurons_gra	Number of graph neurons (16-bit BE) — Type #2.

10	n_out	Number of output ids (16-bit BE) — Type #2.

Selectors 6–8 concern only the single-layer/manual path; with RUN_NETWORK the equivalent values are read per-layer from the descriptor table.

8.5 STATUS.done sticky / clear-on-read

In neuron_memory the done signal is a single-cycle pulse. A host polling over SPI (much slower than the FPGA clock) would almost certainly miss a raw one-cycle pulse. The SPI register bank therefore latches done into a sticky bit on the pulse and clears it when the host reads STATUS (or RESET). The busy bit is instead held at level for the whole computation and is read live.

Race corrected (2026-09-02)

A real race in the sticky mechanism (present since Phase 4) was corrected by latching a status_snapshot on acceptance of the STATUS opcode and conditioning the clearing of the sticky bit on status_snapshot[1] (it clears only if the byte actually transmitted showed done=1). A done that arrives too late for a snapshot is reported on the next poll instead of being lost.

8.6 Host attention pins (data_ready_n, irq_n)

Besides STATUS polling, the top-level exposes two active-low physical pins (bank 7, ch. 13) that mirror the sticky bits without an SPI transaction, handy for driving a host GPIO/IRQ:

- data_ready_n = ∼STATUS.done (sticky): low when a result is ready to read, returns high on the STATUS read (clear-on-read).
- irq_n = ∼STATUS.err (graph guard): low when graph_engine's load-time guard has tripped. It is **not** clear-on-read: it clears only on RESET or a fresh graph start, so an error is not missed between polls.

These are additive ports: they touch neither the existing opcodes nor the registers.

8.7 READ_CONFIG

Fixed **11-byte** payload: it lets a single host firmware work with different bitstreams without recompiling. The N_INPUTS/N_NEURONS values report the build *maximum* (the ceiling), not necessarily the currently loaded network.

Byte	Field	Source
0	ADDR_WIDTH (bit)	neuron_memory.ADDR_WIDTH
1–2	N_INPUTS (16-bit BE)	build maximum
3	N_NEURONS	build maximum
4	PARALLEL	build parameter
5	DATA_WIDTH (bit)	build parameter
6–7	protocol version (BE)	0x0001
8–9	N_TOTAL (16-bit BE)	max graph signals (Type #2)
10	capability flag	bit0=GRAPH_SUPPORTED=1

8.8 Session sequences

8.8.1 Single-layer path

Listing 8.1. Single-layer session

```
RESET                -> 0x0F
READ_CONFIG          -> 0x30      (host learns N_INPUTS/N_NEURONS/...)
WRITE_RAM (weights)  -> 0x01 ...
WRITE_RAM (bias)     -> 0x01 ...
SET_BASE (X/W/BIAS)  -> 0x10 x3
WRITE_RAM (input X)  -> 0x01 ...
START                -> 0x20
poll STATUS          -> 0x21      (until done=1; cleared by this read)
READ_OUTPUT          -> 0x22
```

8.8.2 Multi-layer path (RUN_NETWORK)

Listing 8.2. Multi-layer session

```
WRITE_RAM (descriptor table)              -> 0x01 ...
WRITE_RAM (weights/bias per layer, X L0)  -> 0x01 ...
SET_BASE (X/TABLE/BUF_A/BUF_B)            -> 0x10 x4
RUN_NETWORK(num_layers)                   -> 0x23 <num_layers>
poll STATUS                               -> 0x21   (until done=1)
READ_OUTPUT                               -> 0x22   (y_bus of the final layer)
```

Out of scope for v1

Dual SPI, CRC/checksum on transfers (SPI assumed reliable on a board trace), and back-pressure on RAM transfers: the host must not clock RAM commands faster than one RAM transaction per SPI byte (a reasonable constraint for the initial bulk-loading, not a real-time path).

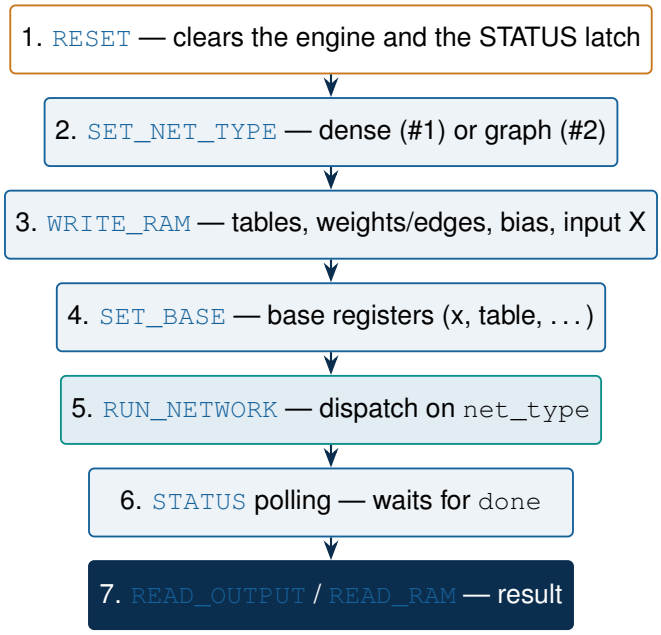
CHAPTER 9

Neural network programming

This chapter is the practical guide to encoding a network for FPGA-Neural: how it is laid out in memory, which registers are set and how it is started, for both topologies. It assumes the SPI opcodes (ch. 8) and the descriptor formats (ch. 6, 7).

9.1 General flow

Whatever the type, the cycle is the same: the host *builds the data structures in RAM*, sets the *base registers*, declares the *network type*, *starts* and *reads back* the result.



9.2 Registers and opcodes involved

All base values are set with `SET_BASE` `sel(1B)+addr(3B)`. Selectors:

sel	Register	Type #1	Type #2	Use
0	`x_base`	✓	✓	Input base X.
3	`table_base`	✓	✓	Descriptor table.
4	`buf_a_base`	✓	✓*	Ping-pong A (#1) / `out_base` reuse (#2).
5	`buf_b_base`	✓	—	Ping-pong B (#1).
9	`num_neurons_graph`	—	✓	Number of graph neurons.
10	`n_out`	—	✓	Number of output ids.

∗ *In Type #2 the ping-pong buffers are unused: selector 4 is reused as `out_base` (region into which outputs are copied). Selectors 1/2/6/7/8 concern only the manual single-layer path (`START`), not `RUN_NETWORK`.*

For Type #1, the *per-layer* `w_base`/`bias_addr` are **not** set with `SET_BASE`: they are fields of the descriptor table. `SET_NET_TYPE` defaults to *dense* after `RESET`, so a #1 network works even without issuing it.

9.3 Type #1 — dense network

9.3.1 Memory layout

Structure	Format
Input X	`n_inputs_real` INT8 bytes at `x_base`.
Weights (per layer)	Neuron-major: neuron k at `w_base + k*n_inputs_real`, `n_neurons*n_inputs` bytes.
Bias (per layer)	One INT8 byte per neuron at `bias_addr`.
Descriptor table	`num_layers` 11-byte entries at `table_base`.
Buffers A/B	Ping-pong intermediate outputs.

Descriptor (11 bytes, MSB-first): `w_base`(3) | `bias_addr`(3) | `activation`(1) | `n_inputs_real`(2) | `n_neurons_real`(2).

9.3.2 Worked example: a $4 \to 4 \to 2$ network

Layer 0: 4 inputs, 4 neurons, ReLU. Layer 1: 4 inputs, 2 neurons, linear (`PARALLEL`=2, so each `n_inputs_real` is a multiple of 2). Chosen addresses: `table_base`=`0x000000`, `x_base`=`0x001000`, L0 weights/bias at `0x002000`/`0x002100`, L1 at `0x002200`/`0x002300`, buffers at `0x003000`/`0x003100`.

Listing 9.1. Dense descriptor table (22 bytes)

```
Layer 0:  00 20 00 | 00 21 00 | 01 | 00 04 | 00 04
          w_base    bias_addr  ReLU  n_in=4  n_neu=4
Layer 1:  00 22 00 | 00 23 00 | 00 | 00 04 | 00 02
          w_base    bias_addr  NONE  n_in=4  n_neu=2
```

Listing 9.2. SPI session (dense)

```
0x0F                              RESET
0x11 01                           SET_NET_TYPE = dense
0x01 000000 0016 <22-byte table>     WRITE_RAM table
0x01 002000 0010 <16-byte L0 wts>    WRITE_RAM L0 weights (neuron-major)
0x01 002100 0004 <4-byte L0 bias>
0x01 002200 0008 <8-byte L1 wts>
0x01 002300 0002 <2-byte L1 bias>
0x01 001000 0004 <x0 x1 x2 x3>       WRITE_RAM input X
0x10 00 001000                    SET_BASE x_base
0x10 03 000000                    SET_BASE table_base
0x10 04 003000                    SET_BASE buf_a
0x10 05 003100                    SET_BASE buf_b
0x23 02                           RUN_NETWORK num_layers=2
0x21 ...                          poll STATUS until done=1
0x22                              READ_OUTPUT -> 2 bytes (final layer)
```

9.3.3 Host pseudocode (dense)

Listing 9.3. Encoding and loading a dense network

```
def load_dense(layers, X):          # layers in execution order
    spi(RESET); spi(SET_NET_TYPE, DENSE)
    table = b""
    for L in layers:                # L: weights[n][k], bias[n], act, n_in, n_out
        assert L.n_in % PARALLEL == 0
        w = alloc(L.weights_neuron_major)   # k slow, input fast
        b = alloc(L.bias)
        table += u24(w)+u24(b)+u8(L.act)+u16(L.n_in)+u16(L.n_out)
    write_ram(TABLE_BASE, table)
    write_ram(X_BASE, X)
    set_base(0, X_BASE); set_base(3, TABLE_BASE)
```

```
    set_base(4, BUF_A);  set_base(5, BUF_B)
    spi(RUN_NETWORK, len(layers))
    wait_status_done()
    return read_output(layers[-1].n_out)
```

9.4 Type #2 — graph network

9.4.1 Memory layout

Structure	Format
Input X	`N_in` bytes at `x_base`; copied into `act_buf[0..N_in-1]` at start.
Descriptor table	`num_neurons_graph` 11-byte entries at `table_base`, in ascending `out_id` order.
Edge blocks	Per neuron: `n_conn` 4-byte edges at `conn_ptr`, padded to a multiple of `PARALLEL` (zero-weight edges).
Outputs	`n_out` bytes written to `out_base` (=selector 4).

Graph descriptor (11 bytes): `conn_ptr`(3) | `n_conn`(2) | `out_id`(2) | `activation`(1) | `bias`(1) | `reserved`(2). Edge (4 bytes): `src_id`(2) | `weight`(1) | `reserved`(1). Rule: `src_id < out_id` (feed-forward DAG).

9.4.2 Worked example

4 inputs (ids 0–3). Neuron n4 (`out_id`=4, ReLU, bias=2) connected to ids 0 and 1; neuron n5 (`out_id`=5, linear, bias=0) connected to n4 (id 4) and id 2; output = n5 (`n_out`=1). `PARALLEL`=2, both have 2 connections (no padding). Addresses: `table_base`=`0x000000`, edges at `0x000100`, `x_base`= `0x001000`, `out_base`=`0x002000`.

Listing 9.4. Graph descriptors + edges

```
Descriptors (at 0x000000, 22 bytes):
 n4: 00 01 00 | 00 02 | 00 04 | 01 | 02 | 00 00
     conn_ptr  n_conn  out_id  ReLU bias  rsv
 n5: 00 01 08 | 00 02 | 00 05 | 00 | 00 | 00 00
     conn_ptr  n_conn  out_id  NONE bias  rsv

Edge blocks (at 0x000100, 4 bytes/edge: src_id, weight, rsv):
 n4 @0x000100:  00 00 05 00   (src=0, w=+5)
                00 01 FD 00   (src=1, w=-3)   ; -3 = 0xFD
 n5 @0x000108:  00 04 02 00   (src=4, w=+2)   ; id4 = n4's output
                00 02 07 00   (src=2, w=+7)
```

Listing 9.5. SPI session (graph)

```
0x0F                              RESET
0x11 02                           SET_NET_TYPE = graph
0x01 000000 0016 <22-byte table>     WRITE_RAM descriptors
0x01 000100 0010 <16-byte edges>     WRITE_RAM edge blocks
0x01 001000 0004 <x0 x1 x2 x3>       WRITE_RAM input X
0x10 00 001000                    SET_BASE x_base
0x10 03 000000                    SET_BASE table_base
0x10 04 002000                    SET_BASE out_base (sel 4 reuse)
0x10 09 000002                    SET_BASE num_neurons_graph = 2
0x10 0A 000001                    SET_BASE n_out = 1
0x23 00                           RUN_NETWORK (dispatch to graph_engine)
0x21 ...                          poll STATUS (bit2=err if src_id>=out_id)
0x02 002000 0001                  READ_RAM out_base -> 1 byte (n5 output)
```

9.4.3 Host pseudocode (graph)

Listing 9.6. Encoding and loading a graph

```
def load_graph(neurons, X, n_out):  # neurons sorted by ascending out_id
    spi(RESET); spi(SET_NET_TYPE, GRAPH)
    edges = b""; table = b""
    for N in neurons:               # N: out_id, conns=[(src_id,w)...], act, bias
        for (src,_) in N.conns:
            assert src < N.out_id and src < N_TOTAL   # DAG rule
        conn_ptr = EDGE_BASE + len(edges)
        padded = pad(N.conns, PARALLEL, fill=(0,0))   # zero-weight edges
        for (src,w) in padded:
            edges += u16(src)+i8(w)+u8(0)
        table += u24(conn_ptr)+u16(len(N.conns))+u16(N.out_id) \
               + u8(N.act)+i8(N.bias)+u16(0)
    write_ram(TABLE_BASE, table); write_ram(EDGE_BASE, edges)
    write_ram(X_BASE, X)
    set_base(0, X_BASE); set_base(3, TABLE_BASE); set_base(4, OUT_BASE)
    set_base(9, len(neurons)); set_base(10, n_out)
    spi(RUN_NETWORK, 0)             # payload ignored in graph
    wait_status_done()
    return read_ram(OUT_BASE, n_out)
```

9.4.4 YAML source

The readable description is in **YAML** (ch. 10), compiled by the host into exactly the table and edge bytes above. Source equivalent to the worked graph:

Listing 9.7. YAML: source and generated bytes

```
# --- source ---
type: graph
requires: {parallel: 2}
inputs: 4                 # ids 0..3
neurons:
  - id: n4
    activation: relu
    bias: 2
    connections: [{src: 0, w: 5}, {src: 1, w: -3}]
  - id: n5
    activation: none
    bias: 0
    connections: [{src: n4, w: 2}, {src: 2, w: 7}]
outputs: [n5]

# --- the compiler emits ---
#  assigned ids: n4=4, n5=5   (guarantees src < id)
#  descriptors:  00 01 00 00 02 00 04 01 02 00 00
#                00 01 08 00 02 00 05 00 00 00 00
#  edges:        00 00 05 00 00 01 FD 00    (n4)
#                00 04 02 00 00 02 07 00    (n5)
#  registers:    table_base, x_base, out_base, num_neurons=2, n_out=1
#  compile-time checks: src<id, N_TOTAL, padding to PARALLEL
```

A single source format

The YAML (ch. 10) and the host pseudocode produce the *same bytes*: YAML is the single source format, version-controlled and hand-written or generated by the trainer; the pseudocode just shows what the compiler does underneath. Either way the FPGA receives only tables and data via `WRITE_RAM`: no on-board interpreter.

CHAPTER 10

Network design with YAML

Networks are described in a declarative **YAML** format, compiled *on the host* into the native bytes (descriptor tables and edge-lists, ch. 9) and loaded via `WRITE_RAM`. No custom language and no on-board interpreter: the YAML is read with a standard parser into an intermediate representation (IR), and a single encoder emits the chip format. The same IR feeds compiler, simulator and trainer (software framework, level 1).

10.1 Common elements

Key	Meaning
`type`	`dense` (Type #1) or `graph` (Type #2). Required.
`requires`	Expected build constraints (`parallel`, `n_total`); the compiler checks them against the bitstream's `READ_CONFIG`.
`name` / `version`	Optional metadata.

Signed INT8 arithmetic throughout: weights and bias $\in [-128, +127]$ (INT32 accumulation on-chip). Activations: `relu` (default) or `none` (linear, bilateral saturation).

10.2 Type #1 — dense network

Implicit connections (fully connected between layers): only weights, bias and sizes are defined.

```
type: dense
inputs: <int>                # inputs of the first layer
layers:
  - neurons: <int>           # layer n_neurons_real
    activation: relu | none
    weights: [[...], [...]]  # neuron-major matrix: weights[k] = neuron k's weights
    bias:    [...]           # one bias per neuron
```

Rules: each layer's input count must be a multiple of `parallel`; layer k's input is layer $k-1$'s output (sizes concatenate); `layers` $\leq$ `N_LAYERS`. Each layer becomes an 11-byte descriptor entry (ch. 9).

Listing 10.1. Dense $4 \to 4 \to 2$

```
name: dense_4_4_2
type: dense
requires: {parallel: 2}
inputs: 4
layers:
  - neurons: 4
    activation: relu
    weights: [[1,-2,0,3],[0,5,-1,2],[-3,1,4,0],[2,0,-2,1]]
    bias: [0, 1, -1, 0]
  - neurons: 2
    activation: none
    weights: [[1,0,-1,2],[-2,3,0,1]]
    bias: [0, 0]
```

10.3 Type #2 — graph network

Explicit connections: each neuron lists its inputs as `(src, w)` edges.

```
type: graph
n_total: 4096                 # id space (default 4096)
inputs: <int>                 # input ids: 0 .. inputs-1
neurons:                      # in topological order (sources before consumers)
  - id: <name|int>
    activation: relu | none
    bias: <int>
    connections:
      - {src: <id>, w: <int>} # src = input or an already-declared neuron id
outputs: [<id>, ...]          # sink neurons -> last n_out
```

Rules: inputs take ids `0..inputs-1`, neurons get ascending ids from `inputs` on (declaration order); rule **`src < id`** of the neuron (feed-forward DAG, no cycles) and `src < n_total`; padding `n_conn` to a multiple of `parallel` (zero-weight edges) is done by the compiler; `outputs` are sinks (not used as `src`).

Listing 10.2. Graph n4/n5

```
name: graph_n4_n5
type: graph
requires: {parallel: 2}
inputs: 4                     # ids 0..3
neurons:
  - id: n4
    activation: relu
    bias: 2
    connections: [{src: 0, w: 5}, {src: 1, w: -3}]
  - id: n5
    activation: none
    bias: 0
    connections: [{src: n4, w: 2}, {src: 2, w: 7}]
outputs: [n5]
```

10.4 Weights: inline or external file

Small/hand-written networks: inline weights. Trained networks: external file (produced by the trainer), with `weights_file`/`bias_file` (int8, shape checked by the compiler). `weights` and `weights_file` are mutually exclusive.

```
  - {neurons: 64, activation: relu, weights_file: fc0_w.npy, bias_file: fc0_b.npy}
```

10.5 Validation rules (compile-time)

The compiler rejects the file if: weights/bias outside $[-128, +127]$; (dense) a layer's input not a multiple of `parallel` or sizes that don't concatenate, wrong lengths, `layers>N_LAYERS`; (graph) `src >= id` (cycle/non-topological) or `src >= n_total`, or an output used as a source; `type` missing; `requires` incompatible with the target. Each error names the location (layer/neuron, field). The chip still has the runtime guard (`STATUS.err`/`irq_n`) as a second line of defense.

10.6 Application use cases

FPGA-Neural is an engine for **compact INT8 dense classifiers** (or sparse graphs): typically the *final stage* of a pipeline, where raw features are extracted by the host or a sensor and the chip makes the decision — deterministic, low-latency, offloading the CPU.

10.6.1 Face classification (dense)

The host (e.g. ESP32-CAM) detects the face and extracts an INT8 *embedding*; the chip classifies among known identities. Enrolling a new face = refining the last layer (runtime, level 2).

```
name: face_classifier_128
type: dense
requires: {parallel: 8}
inputs: 128                   # embedding size
layers:
  - {neurons: 32, activation: relu, weights_file: fc0_w.npy, bias_file: fc0_b.npy}
```

```
  - {neurons: 4,  activation: none, weights_file: fc1_w.npy, bias_file: fc1_b.npy}
# outputs: [A, B, C, unknown] -> host does argmax
```

10.6.2 Predictive maintenance / anomaly (dense)

INT8 features from machine sensors (vibration, current, temperature, already extracted by the MCU) → *normal/anomaly* or fault-mode classification. Continuous low-latency inference in the field, no data sent to the cloud.

```
name: machine_health
type: dense
requires: {parallel: 8}
inputs: 64                    # spectral/statistical features
layers:
  - {neurons: 16, activation: relu, weights_file: h0_w.npy, bias_file: h0_b.npy}
  - {neurons: 3,  activation: none, weights_file: h1_w.npy, bias_file: h1_b.npy}
# outputs: [ok, wear, fault]
```

10.6.3 Keyword / gesture (dense)

Audio (MFCC) or IMU features extracted by the host → word/gesture class. Ultra-low-power wake-word or gesture commands.

```
name: keyword_spotter
type: dense
requires: {parallel: 8}
inputs: 40                    # feature vector
layers:
  - {neurons: 16, activation: relu, weights_file: k0_w.npy, bias_file: k0_b.npy}
  - {neurons: 5,  activation: none, weights_file: k1_w.npy, bias_file: k1_b.npy}
```

Mind the `parallel` constraint

`inputs` and each layer's input must be multiples of `parallel`. For 40 features with `parallel`=8, 40 is a multiple of 8, fine. If the features are not a multiple, zero-pad the input up to the multiple on the host side: a preparation choice, not the chip's.

10.6.4 Reflex controller / sparse network (graph)

When each output depends on a *specific subset* of inputs (irregular, hand-designed connectivity), Type #2 avoids paying for a dense network's zero weights: you wire only the connections that matter. Example: a robot where each actuator reacts to a few targeted sensors.

```
name: reflex_controller
type: graph
requires: {parallel: 2}
inputs: 6                     # 6 sensors: ids 0..5
neurons:
  - id: avoid_L
    activation: relu
    bias: 0
    connections: [{src: 0, w: 4}, {src: 1, w: -3}]   # front-left sensors only
  - id: avoid_R
    activation: relu
    bias: 0
    connections: [{src: 4, w: 4}, {src: 5, w: -3}]   # front-right only
  - id: motor_L
    activation: none
    bias: 0
    connections: [{src: avoid_L, w: 2}, {src: 2, w: 1}]
  - id: motor_R
    activation: none
    bias: 0
    connections: [{src: avoid_R, w: 2}, {src: 3, w: 1}]
outputs: [motor_L, motor_R]
```

Dense or graph?

Choose **dense** when connectivity is full and regular (classifiers on embeddings, feature vectors): it is the faster path. Choose **graph** when connectivity is sparse and specific: you save cycles and memory by not paying for zero connections.

CHAPTER 11

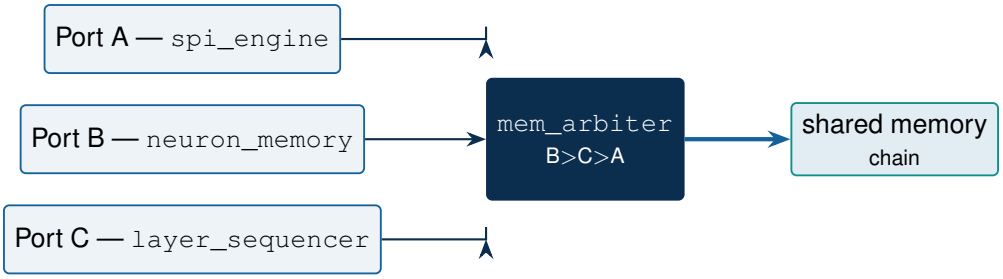
Arbitration and top-level integration

11.1 `mem_arbiter` — three-port arbiter

A single byte-level memory master (which feeds the shared chain `int8_memory_access` → `memory_interface` → `psram_controller`) is arbitrated among three requesters:

Port	Master	Accesses
A	`spi_engine`	`WRITE_RAM` / `READ_RAM`.
B	`neuron_memory`	X/W/bias reads during an execution.
C	`layer_sequencer`	Descriptor reads + buffer writes between layers.

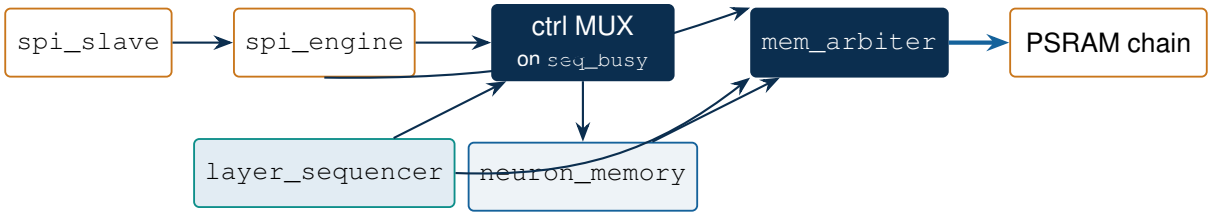
Fixed priority **B** > **C** > **A**: an inference in progress is more critical than the sequencer's bookkeeping, which in turn is more critical than a manual SPI access that has just arrived. In normal operation B and C are anyway temporally disjoint (`neuron_memory` requests only during an execution, `layer_sequencer` only in the pauses between layers), so the priority matters mostly for the corner case of a manual `WRITE_RAM`/`READ_RAM` arriving during a multi-layer execution.



Once access is granted, the arbiter retains ownership until the single transaction's `m_ready` pulse, then releases: all three masters emit `req` as a clean one-cycle pulse, so a queue-less grant-and-forward design suffices.

11.2 `spi_neuron_top` — full integration

The top-level connects SPI (`spi_slave`+`spi_engine`), the arbiter, the sequencer, `neuron_memory` and the PSRAM chain. The reset of `neuron_memory` is the OR of the global reset with the soft-reset pulse of the `RESET` opcode, so the host can recover the engine over SPI without a physical reset (the RAM stays intact).



The multiplexer switches the control lines of `neuron_memory` between the sequencer (while `seq_busy` is high) and the direct path of `spi_engine` (legacy single-layer mode), returning the engine to the direct path at the end of the sequence.

End-to-end verification

`spi_neuron_top` is verified in simulation with real PSRAM (`psram_model.v`, no mock): RESET/READ_CONFIG/WRITE_RAM/READ_RAM/SET_BASE/ START/STATUS/READ_OUTPUT and `RUN_NETWORK` are exercised purely over simulated SPI (ch. 12).

CHAPTER 12

ECP5 implementation and characterization

12.1 Flow and verification

The project is verified on two complementary planes: functional **simulation** with Icarus Verilog (signed algebra, products, accumulation, groups, bias, ReLU, saturation, busy/done signals) and real **implementation** with Yosys (synthesis) + nextpnr-ecp5 (place&route, timing) + Project Trellis (`ecppack`).

Verification stage	Outcome	Covers
Functional RTL	**PASS**	datapath correctness
Parametric simulation	**PASS**	configuration sweep
ECP5 synthesis	**PASS**	synthesizability, mapping
Placement / Routing	**PASS**	LUT/FF/DSP, timing
Bitstream (`ecppack`)	**PASS**	full flow, 0 errors (P2 and P8)

End-to-end toolchain through the bitstream

The full flow RTL → Yosys → nextpnr-ecp5 → `ecppack` produces a valid bitstream for P2 and P8, **0 errors at every stage**. Header verified byte-by-byte: `Part: LFE5U-45F-8CABGA381`, the target's real part number, not a placeholder. Only *generation* is verified: no physical-hardware test in this session.

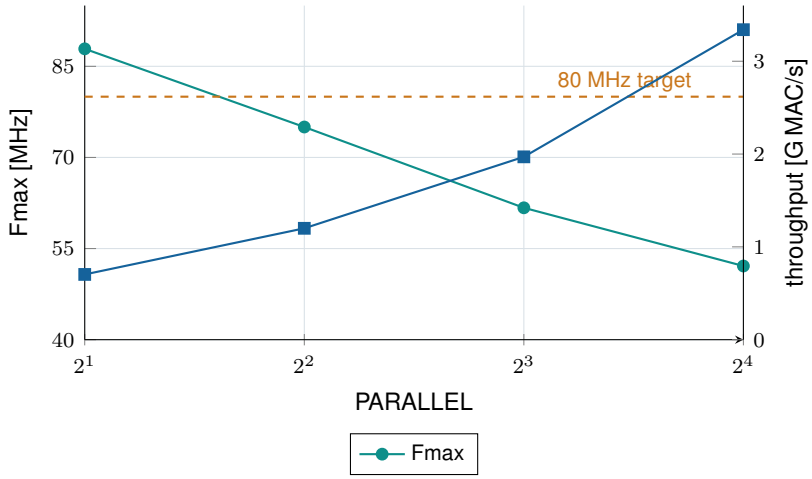
12.2 Datapath benchmark (256×4)

Configuration: INT8/INT32, `N_INPUTS`=256, `N_NEURONS`=4, variable `PARALLEL`, 80 MHz target, device `LFE5U-45F-8BG381C` (−8). The test buses are generated *inside* the benchmark wrapper so as not to expose thousands of I/Os; the top-level exposes only `clk/rst/start/y_bus/busy/done`.

PAR	tot MAC	DSP	Fmax	Tcrit	80 MHz	LUT4
16	64	64/72	52.13	19.18	**FAIL**	≈2531
8	32	32/72	61.71	16.20	**FAIL**	—
4	16	16/72	75.01	13.33	**FAIL**	804
2	8	8/72	87.88	11.38	**PASS**	481

Fmax and Tcrit in MHz and ns. Total MACs = PARALLEL×4 neurons.

12.2.1 Fmax and throughput versus parallelism



Fundamental trade-off: as PARALLEL grows, Fmax drops (deeper routing/tree) but the theoretical throughput rises. The blue line (squares) is the throughput ≈MAC/cycle×Fmax.

12.2.2 Interpretation

Reducing `PARALLEL` lowers simultaneous MACs, DSPs, adder-tree depth and routing congestion, so Fmax rises; but the number of groups increases and hence the latency. Frequency alone is not enough to choose: what matters is the overall throughput ≈MAC/cycle×frequency.

Architectural choices

`PARALLEL=8` is the candidate for the throughput-oriented V1: exactly 32 simultaneous MACs with 4 neurons, DSP at ≈44%, leaving resources for controller, buffers, SPI and future pipelines. `PARALLEL=2` is the frequency-oriented reference: 87.88 MHz, the only one to exceed the 80 MHz target, but it requires 128 groups for a 256-input neuron.

12.2.3 Critical path and the 100 MHz limit

The 100 MHz target is not met (best result 87.88 MHz with P2). The limit is *temporal*, not one of occupancy: with P2 the FPGA is barely used (DSP ≈11%, LUT ≈1%). The critical path runs through weight FF → `MULT18X18D` → products → adder/carry → `acc_next` → ReLU/saturation → output FF. Exceeding 100 MHz will require one or more internal pipelines, not yet necessary to proceed.

12.3 Full integrated system

Real synthesis of `spi_neuron_top` (SPI + arbiter + `neuron_memory` + `graph_engine` + PSRAM chain), speed grade `−8`. Before timing closure the integrated system missed the 80 MHz target (P2 ≈55 MHz, P8 ≈45 MHz), with a critical path entirely inside `neuron_parallel`.

12.3.1 Cause: the saturation/ReLU carry chain

Resource usage is not the cause (device below 10% everywhere). The integrated system's critical path is the **CCU2C carry chain of the saturation/ReLU comparator** in `neuron_parallel.v` — *not* SPI, arbiter, PSRAM, nor the Type #2 modules. The saturation was written as an arithmetic comparison (`acc > 127`, `acc < -128`), mapped by the synthesizer onto a 32-bit subtractor with a long carry chain.

12.3.2 Timing closure (2026-09-03)

An explicit waiver of the “datapath untouchable” rule for a separate timing-closure task, with the single constraint of **bit-exact equivalence** across the whole regression. Two steps:

- **Step 1 — saturation/ReLU as bit-test.** A signed 32-bit value fits INT8 iff `acc[31:7]` are all equal: an AND/OR reduction over a bit slice instead of a 32-bit carry chain. A correct, bit-exact-verified simplification; real logic gain, but on its own submerged by placement noise.
- **Step 2 — pipeline register** between accumulate and activation (`+1` cycle of latency per neuron, absorbed by the `start`/`done` handshake, transparent to callers). This is the decisive step.

Config	Before	After	Δ
P2, real `.lpf`	54.58	**75.30**	$+38\%$
P2, 5-seed sweep	55.59	73.38–75.55	robust
P8, unconstrained	45.47	**60.26**	$+33\%$
P8, 5-seed sweep	43.15–50.48	60.26–68.87	non-overlapping

Fmax in MHz, real place&route (`nextpnr-ecp5`). Robust across 5 seeds, not attributable to placement luck.

Stop criterion and real margin

80 MHz is not reached (75.30 MHz at P2, 94% of target) but the gain is large and real ($+38\%$/$+33\%$). The next step (the `MULT18X18D` output register, which would touch `mac_unit.v`) was left out: 80 MHz is *headroom* toward the real `.lpf`, not an operating requirement. With the planned 16 MHz oscillator, even the worst measured number (≈45 MHz at P8) has $2.8\times$ of margin.

Separate future optimization

Independent of timing closure: the `x_mem`/`w_mem` arrays of `neuron_memory` are still inferred as distributed RAM on LUTs instead of `DP16KD`. Moving them to block RAM would free LUTs and is a Phase 7 candidate — but it was not on the critical path resolved here.

CHAPTER 13

Hardware design and signal map

Pinout status — assigned and verified

A real `.lpf` now exists (`synth/ecp5/spi_neuron_top.lpf`) with the top-level's 53 signals assigned to concrete CABGA381 balls, **verified by a full 0-error `nextpnr-ecp5` place&route** (no longer `--lpf-allow-unconstrained`). The balls come from Project Trellis's device database (`iodb.json`, the same nextpnr uses) and were independently validated against §4.3.2 of the official Lattice datasheet (per-bank GPIO counts: exact match on 6 of 7 banks, off by 1 ball on bank 3, immaterial since no assigned signal uses it). `TRELLIS_IO`: 53/245 (21%). Current-build Fmax 73.88 MHz, within the timing-closure seed-noise band (ch. 12). The config-SPI and JTAG balls do not appear here: they are dedicated fixed-function pins with no corresponding RTL port, nextpnr never requires them (0 errors), they matter only for the PCB schematic.

13.1 Target device

Parameter	Value
Device	Lattice ECP5 `LFE5U-45F-8BG381C`
Package	CABGA381 (381 balls)
Speed grade	-8 (the fastest of the ECP5 family)
Resources	≈44k LUT/FF, 72×`MULT18X18D`, `DP16KD` block RAM
Usable I/O	≈232 balls out of 381 (rest: power/ground/NC)

13.2 Pin budget

The project requires about 58 signals out of ≈232 usable I/Os: ample margin (>170 free pins), so the board is not pin-constrained.

Function	Pins
PSRAM (22 address, 16 data, 6 control)	up to 44
Application SPI (`sclk/mosi/miso/cs_n`)	4
Clock, reset	2
Configuration SPI (to onboard flash)	4
JTAG (bring-up / debug, recommended)	4
Total	**≈58**

13.3 Signal map (top-level `spi_neuron_top`) — real balls

Real assignment of the top-level's 53 signals, verified by place&route. I/O standard: LVCMOS33 (3.3 V I/O supply). The balls come from the real place&route-verified `.lpf`.

Signal	Dir	Ball	Bank	Function

Clock and reset (bank 7, left edge)				
`clk`	IN	H5	7	System clock on pad `GR_PCLK7_0` (dedicated global clock).
`rst`	IN	B4	7	Global synchronous reset, active high.
Application SPI (bank 7, opposite the PSRAM bus)				
`sclk`	IN	B5	7	SPI clock (CPOL=0, CPHA=0).
`mosi`	IN	C5	7	Master-Out Slave-In.
`miso`	OUT	A3	7	Master-In Slave-Out (driven on the falling edge).
`cs_n`	IN	B3	7	Active-low chip-select.
Host attention pins (bank 7, active-low, level)				
`data_ready_n`	OUT	C3	7	Low while a result awaits reading (mirrors `STATUS.done`, clear on STATUS read).
`irq_n`	OUT	C4	7	Low if the graph load-time guard has tripped (mirrors `STATUS.err`); clears only on `RESET` or a fresh `run_start`, *not* on a STATUS read.
PSRAM address bus (bank 2, right edge)				
`psram_a[21:0]`	OUT	E16…H20	2	22 real address lines (A0–A21, 8 MB).
`psram_a[22]`	OUT	P18	3	Always 0 (byte→word shift): NC on the board.
PSRAM data bus (banks 2 and 3)				
`psram_dq[9:0]`	IO	K18…K20	2	`dq[1..9]` on dual-function balls used as plain GPIO.
`psram_dq[15:10`	IO	L17…P17	3	Bidirectional tri-state data bus (`dq_oe` = direction).
PSRAM control (bank 3)				
`psram_ce_n`	OUT	N17	3	Chip enable, active low.
`psram_oe_n`	OUT	R16	3	Output enable (read).
`psram_we_n`	OUT	R17	3	Write enable (write).
`psram_lb_n`	OUT	T16	3	Lower-byte enable (DQ[7:0]).
`psram_ub_n`	OUT	N19	3	Upper-byte enable (DQ[15:8]).
`psram_zz_n`	OUT	N20	3	Sleep/snooze (inactive=high in operation).

Board signals not exposed as RTL ports

Not ports of `spi_neuron_top` but required at board level: the **configuration SPI** lines to the onboard NOR flash (`PROGRAMN`/`INITN`/`DONE`/`CCLK`…, the datasheet's "Miscellaneous Dedicated Pins") and the 4 **JTAG** lines (`TCK`/`TMS`/`TDI`/`TDO`), the **oscillator** on the `PCLK` pad, the **power supplies**. Their ball numbers are not in the Lattice datasheet (separate file) but are not needed here: dedicated pins with no RTL port, nextpnr never requires them (0 errors), they matter only for the PCB schematic.

Application SPI separate from configuration SPI

The application SPI (`sclk/mosi/miso/cs_n`) must land on ordinary I/Os, **never** on the configuration-SPI pins: the config-SPI clock pin is not reusable as a general-purpose input after configuration without a board-level workaround. Keeping them physically separate avoids that problem.

13.4 Per-bank allocation (real die geometry)

The placement follows the die-edge geometry (from Trellis's `globals.json`, ball → (col,row) → bank): banks **2 and 3** sit contiguously along the chip's **right** edge and together hold the entire PSRAM bus (44+1 signals) — exactly the "one or two adjacent banks" recommended. Bank **7** (**left** edge, physically opposite the PSRAM bus) holds the application SPI and clock/reset, deliberately on the far side so the two buses do not cross. `clk` is on the dedicated pad `H5` (`GR_PCLK7_0`). Where a bank ran out of plain balls (part of `psram_dq`), the next dual-function ball was used as ordinary GPIO, confirmed usable by the real place&route.

Signal group	# pins	Bank (real)
PSRAM addresses `psram_a[21:0]`	22	bank 2 (right edge)
PSRAM data `psram_dq[15:0]`	16	banks 2 + 3 (adjacent)
PSRAM control (ce/oe/we/lb/ub/zz)	6	bank 3
Application SPI	4	bank 7 (left edge)
Host attention pins (`irq_n`, `data_ready_n`)	2	bank 7
Clock / reset	2	bank 7, `clk` on `GR_PCLK7_0`
Config SPI / JTAG	—	dedicated pins (outside RTL, PCB only)

13.5 PSRAM subsystem

The `psram_controller.v` controller implements an **asynchronous parallel** interface (address bus, 16-bit data, `ce_n/oe_n/we_n` and byte-lanes `lb_n/ub_n`, plus `zz_n`) with an access latency of **70 ns** wired as $\lceil 70\,\text{ns} \times f_{clk} \rceil$. It is an asynchronous-SRAM-style bus, not QSPI.

Role	Component
Working memory	ISSI `IS66WVE4M16EBLL-70BLI` — 64 Mbit parallel PSRAM (4M×16, 8 MB), async, 70 ns, an exact match to the controller timing.
Fallback	ISSI `IS61WV6416DBLL` / `IS61WV102416BLL` (true async SRAM, drop-in on the same signals, `zz_n` inactive, ∼10 ns, lower density).
Persistent storage	Winbond `W25Q128JV` — 16 MB SPI NOR flash for bitstream, weights, bias, network metadata.

13.5.1 PSRAM connection (FPGA-exclusive)

The PSRAM is driven **exclusively by the FPGA** through `psram_controller.v`: no external master touches the bus. The external host (RPi/ESP32/MCU) only speaks SPI to the FPGA and never touches these lines. Pin-by-pin connection FPGA ↔ ISSI `IS66WVE4M16EBLL-70BLI`:

FPGA signal	PSRAM pin	Function
`psram_a[21:0]`	A0–A21	Address bus (22 lines, 8 MB word address).
`psram_dq[15:0]`	DQ0–DQ15	Bidirectional data bus (tri-state, `dq_oe`=direction).
`psram_ce_n`	CE#	Chip enable (active low).
`psram_oe_n`	OE#	Output enable (read).
`psram_we_n`	WE#	Write enable (write).
`psram_lb_n`	LB#	Lower-byte enable (DQ[7:0]).
`psram_ub_n`	UB#	Upper-byte enable (DQ[15:8]).

psram_zz_n	ZZ#	Sleep/snooze (held high in operation).

PSRAM supply: **3.3 V** (BLL variant), on the same I/O rail as banks 2/3 to which it is wired (ch. 13, real balls). Decoupling per supply pin per the ISSI datasheet.

13.6 Clock

There is no PLL in the RTL yet: CLK_FREQ_MHZ is a *timing parameter* (it feeds the PSRAM access formulas), not a clock generator. The mounted oscillator drives clk directly. Recommendation: a 16 MHz MEMS oscillator (SiTime SiT2001B family), well below the 75 MHz Fmax of the full integrated system after timing closure (ch. 12). CLK_FREQ_MHZ must be set to the real value of the mounted oscillator, otherwise the PSRAM timing comes out wrong.

13.7 Power

A **three-rail** tree (the Lattice eval board's SERDES section is not needed and is omitted: no 1.2 V VCCA/VCCHTX):

Rail	Voltage	Feeds / regulator
VCC (core)	1.1 V	FPGA core logic. Buck TLV62568, ≥600 mA.
VCCIO0/2/3/6/7	3.3 V	I/O of all used banks + PSRAM. Buck TLV62568, 1 A.
VCCAUX	2.5 V	FPGA auxiliary. LDO TLV73325, 10 mA.

Decoupling: at least one capacitor per supply pin + bulk per rail, per the Lattice ECP5 hardware checklist. Input: external 12 V (or match the bucks to the source).

13.8 Configuration and programming

Writing the FPGA "map" (bitstream) happens through dedicated silicon pins, **not** RTL top-level ports. Default mode: **MSPI** — automatic boot from the NOR flash at power-on (standalone product); JTAG available for development.

13.8.1 JTAG (development / debug)

Signal	Ball†	Function
TCK	T5	Test clock.
TDI	R5	Test data in.
TDO	V4	Test data out.
TMS	U5	Test mode select.

13.8.2 Config-SPI to boot flash

The FPGA loads the bitstream from the **Winbond W25Q128JV** (128 Mbit SPI NOR, Quad read) at power-on. The same flash can hold network weights/bias/metadata.

Signal	Ball†	Function
CCLK/MCLK/SCK	U3	Configuration clock.
DQ0_MOSI	W2	Config data (MOSI).
DQ1_MISO	V2	Config data (MISO).
BUSY_CSSPIN	R2	Flash chip-select.

DQ2 / DQ3	Y2 / W1	Quad-read lines.
PROGRAMN	W3	Start reconfiguration (button, active low).
INITN	V3	Init / configuration error (LED).
DONE	Y3	Configuration complete (LED).
CFGMDN[2:0]	R4/T4/U4	Mode select (see below).

13.8.3 Configuration modes (CFGMDN)

Mode	CFGMDN[2:0]	Use
MSPI (boot from flash)	010	**Default** — standalone.
SSPI (slave SPI)	001	Config from external host.
SCM (slave serial)	101	Serial config.
SPCM (slave parallel)	111	8-bit parallel config.

Configuration balls to verify on the 45F

†The JTAG and config-SPI balls listed here are the *reference* from the Lattice eval board (85F device). JTAG and config-SPI are dedicated, largely fixed pins in the ECP5 family, but the exact positions on the LFE5U-45F-8BG381C target must be confirmed against the Lattice 45F pinout file (Diamond/Radiant or the Trellis database) before committing them to the schematic, as already done for the application signals (ch. 13).

13.9 Open tasks before schematic capture

OK ADDR_WIDTH=23 (full 8 MB) across all modules and testbenches.
OK Real .lpf with the CABGA381 ball assignment, place&route-verified with 0 errors (synth/ecp5/spi_neuron_top.lpf).
☐ Confirm PSRAM/SPI signal integrity at the actually mounted clock.
☐ Choice of the JTAG connector footprint.
☐ Schematic capture (KiCad or other): no schematic exists yet for this device/package combination.

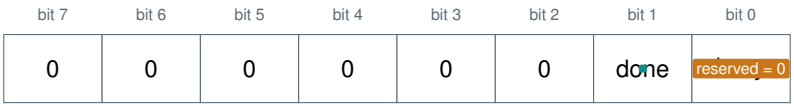
CHAPTER 14

Quick reference

14.1 SPI opcodes

Value	Name	Response	Summary
0x00	NOP	—	idle
0x01	WRITE_RAM	—	PSRAM block write
0x02	READ_RAM	len B	PSRAM block read
0x0F	RESET	—	engine reset + STATUS latch
0x10	SET_BASE	—	set base/register (sel 0..10)
0x20	START	—	single-layer start
0x21	STATUS	1 B	busy(live)/done(sticky)
0x22	READ_OUTPUT	N_NEURONS B	y_bus
0x23	RUN_NETWORK	—	multi-layer start
0x30	READ_CONFIG	11 B	configuration record

14.2 STATUS byte



done is sticky, clear-on-read; busy is live.

14.3 SET_BASE selectors

- 0 — x_base
- 1 — w_base
- 2 — bias_addr
- 3 — table_base
- 4 — buf_a_base
- 5 — buf_b_base
- 6 — activation (single-layer)
- 7 — n_inputs_real (single-layer)
- 8 — n_neurons_real (single-layer)
- 9 — num_neurons_graph (Type #2)
- 10 — n_out (Type #2)

14.4 Descriptor table (11 bytes/layer, MSB-first)

w_base 3B	bias_addr 3B	act 1B	n_inputs_real 2B	n_neurons_real 2B

14.5 Build parameters

- DATA_WIDTH — 8 (INT8)
- ACC_WIDTH — 32 (INT32)
- N_INPUTS — max inputs
- N_NEURONS — max neurons
- PARALLEL — simultaneous MACs
- N_LAYERS — max layers
- ADDR_WIDTH — 23 (8 MB)
- MEM_DATA_WIDTH — 16
- CLK_FREQ_MHZ — PSRAM timing

CHAPTER 15

Roadmap and development status

15.1 Development phases

Phase	Title	Status	Content
1	Parametric layer	**OK**	inputs/neurons/parallelism, accumulation, bias, ReLU; test 32×4/P=8.
2	Parameter sweep	**OK**	multiple configurations incl. non-multiple and degenerate; elaboration guard added.
3	Memory architecture	**OK**	`neuron_memory` single/multi-neuron, real PSRAM tested; multi-layer buffers → Phase 5.
4	SPI interface	**OK**	`spi_slave`+`spi_engine`, all opcodes, Fmax checked in isolation.
5	Multi-layer network	**OK**†	`layer_sequencer`, configurable activations, runtime width; real toolchain checked.
6	Host software	planned	Linux and ESP32 drivers on the same protocol.
7	Optimization	in progress	timing closure done (55→75 MHz); PSRAM page-mode, gather bandwidth, x/w block RAM remain.
8	Hardware training (opt.)	future	backprop, gradients, weight update.

† *RTL, unit tests and end-to-end over simulated SPI complete; timing closure done: 75.30 MHz (P2) / 60.26 MHz (P8), bit-exact across the whole regression (ch. 12).*

15.2 Component status

Component	Status
Parametric neural layer	**OK** working
Parametric inputs/neurons/parallelism	**OK**
Accumulation, bias, ReLU	**OK**
32×4 / P=8 validation	**OK**
Dedicated RAM (interface + controller + INT8 access)	**OK** tested on real PSRAM
SPI interface (all opcodes incl. RUN_NETWORK)	**OK** Fmax in isolation
Dual SPI	future
Multi-layer engine	**OK** timing closure 75.30 MHz (P2)
Configurable activations (ACT_NONE/ACT_RELU)	**OK**

Runtime network width (one bitstream, any topology)	**OK** measured savings
Type #2 graph network (act_buffer, graph_engine, YAML compiler)	**OK** RTL + tests + synthesis
CABGA381 pinout (real `.lpf`)	**OK** place&route-verified, 0 errors
PSRAM page-mode (G7)	open, now quantified (53.25 cycles/edge)
Real bitstream (`ecppack`, P2/P8)	**OK** 0 errors, part LFE5U-45F-8CABGA381
Linux / ESP32 host driver	planned
Hardware training	future

15.3 Architectural principle (summary)

Foundation of the project

The FPGA implements the neural machine and owns its own RAM; the host configures and uses the machine. A build fixes the *ceiling* (max layers, max width, PARALLEL); the host configures the *actual* network — number of layers, per-layer width, per-layer activation, trained parameters — entirely at runtime, over SPI, into the FPGA's local memory. A single bitstream serves any topology up to that ceiling.

15.4 Long-term vision

The final goal is a reusable hardware block integrable into different future projects: the host platform can change (Linux, ESP32, MCU, PC) without changing the fundamental architecture of the engine. The FPGA becomes a dedicated neural computation peripheral, optimized for the topology required by each application.

CHAPTER A

Modules, ports and toolchain

A.1 List of RTL modules

File	Type	Role
`rtl/mac_unit.v`	combinational	single multiply-accumulator
`rtl/mac8.v`	combinationa	parallel MAC + balanced adder tree
`rtl/neuron_parallel.v`	FSM	neuron: groups, bias, activation, saturation
`rtl/layer.v`	structural	N_NEURONS neurons in parallel
`rtl/neuron_memory.v`	FSM	memory/neuron bridge, neuron loop
`rtl/layer_sequer`	FSM	multi-layer sequencing, ping-pong
`rtl/int8_memory_access.v`	FSM	byte ↔ word conversion
`rtl/memory_inter`	FSM	req/ready handshake
`rtl/psram_controller.v`	FSM	async 70 ns physical PSRAM bus
`rtl/mem_arbiter.`	arbiter	3 ports, priority B>C>A
`rtl/spi_slave.v`	FSM	SPI Mode 0 physical layer + CDC
`rtl/spi_engine.v`	FSM	opcode + register bank
`rtl/act_buffer.v`	block RAM	DP16KD activation buffer (Type #2)
`rtl/graph_engine`	FSM	graph-network engine (Type #2)
`rtl/spi_neuron_top.v`	top	full integration
`rtl/memory_model`	model	behavioral RAM (sim)

A.2 Ports of the top-level `spi_neuron_top`

See the complete signal-by-signal table in ch. 13. In summary: clock and reset (`clk`, `rst`); application SPI (`sclk`, `mosi`, `miso`, `cs_n`); PSRAM bus (`psram_a[22:0]`, `psram_dq[15:0]`, `psram_ce_n/oe_n/we_n/lb_n/ub_n/zz_n`).

A.3 Toolchain

Tool	Version	Use
Yosys	0.68+post	RTL synthesis → JSON netlist, ECP5 mapping
nextpnr-ecp5	0.11.1-19-g8dbcee5	placement, routing, timing
Project Trellis	install	`ecppack/ecppll/ecpbram`
Icarus Verilog	`-g2012`	functional simulation

A.3.1 Main nextpnr parameters

```
--45k                       selects LFE5U-45F
--package CABGA381          package
--speed 8                   speed grade -8
--json <netlist>           netlist from Yosys
--lpf <constraints>        pin constraints (currently empty)
```

```
--lpf-allow-unconstrained    allows unconstrained I/Os (benchmark)
--freq 80                    80 MHz timing target
```

A.3.2 Simulation example

```
iverilog -g2012 -Ptb.PARALLEL=16 -o sim/parametric_256x4_p16 \
  sim/parametric_tb.v rtl/mac_unit.v rtl/mac8.v \
  rtl/neuron_parallel.v rtl/layer.v
vvp sim/parametric_256x4_p16
```

A.4 Main testbenches

Testbench	Coverage
`parametric_tb.v`	256×4 datapath, accumulate/bias/ReLU/saturation cases
`parameter_sweep_tb.v`	sweep of valid configurations
`neuron_parallel_tb.v`	activations, runtime width (T7)
`neuron_memory_tb.v` / `_multi_tb.v`	single/multi-neuron memory integration, real PSRAM (T5)
`psram_controller_tb.v`	PSRAM controller
`spi_slave_tb.v`	SPI physical layer (4 tests)
`spi_engine_tb.v`	opcodes, registers (10+ tests)
`spi_neuron_top_tb.v`	end-to-end, real PSRAM over simulated SPI
`spi_neuron_top_runnetwork_tb.v`	RUN_NETWORK 2-layer end-to-end
`layer_sequencer_tb.v`	2-layer sequence, ping-pong, byte-exact copy

This datasheet is generated from the RTL code, the documentation and the benchmarks present in the repository `github.com/manvalan/FPGA-Neural` as of September 2026. The Fmax, resource usage and throughput values are those reported in the repository measurements and must be re-verified after the assignment of a real `.lpf` and the Phase 7 timing closure.