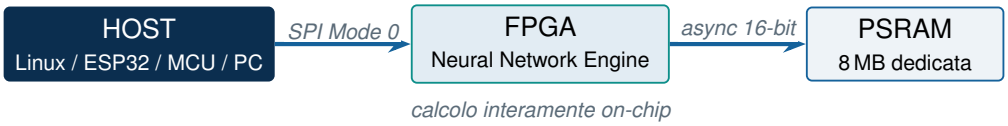
FPGA – Neural

INT8 Neural Network Engine per FPGA

Acceleratore hardware parametrico – Datasheet e manuale di riferimento

Rev. A1 • Settembre 2026



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

Configurazione baseline: INT8/INT32, `N_INPUTS`=256, `N_NEURONS`=4, `PARALLEL` parametrico, memoria di lavoro PSRAM ISSI `IS66WVE4M16EBLL-70BLI`.

Stato: RTL verificato in simulazione (Icarus) e sintesi reale (Yosys + nextpnr-ecp5). Documento descrittivo del progetto allo stato del Settembre 2026.

Autore del progetto: Michele Bigi • MIKILAB / manvalan.

Questo datasheet documenta il codice RTL, la documentazione e i benchmark presenti nella repository `github.com/manvalan/FPGA-Neural`.

FPGA-Neural — Descrizione generale e caratteristiche

FPGA-Neural è un **acceleratore hardware parametrico per reti neurali** feed-forward completamente contenuto nell'FPGA. Il calcolo (moltiplicazione, accumulo, bias, attivazione, saturazione) avviene interamente on-chip in aritmetica intera INT8/INT32; il sistema host fornisce solo configurazione, pesi, dati di ingresso e controllo attraverso una semplice interfaccia SPI, senza mai far parte del datapath computazionale. Un unico bitstream serve qualunque topologia fino al massimo di build.

Caratteristiche

- Datapath **INT8** $\times$ **INT8** $\rightarrow$ **INT16** $\rightarrow$ **INT32**, accumulo a 32 bit con estensione di segno.
- **Balanced binary adder tree** ($O(\log_2$ PARALLEL$)$) al posto della riduzione lineare.
- MAC parallelo configurabile: `PARALLEL` MAC hardware simultanei per neurone, mappati su DSP `MULT18X18D`.
- Architettura completamente **parametrica**: `N_INPUTS`, `N_NEURONS`, `PARALLEL`, `DATA_WIDTH`, `ACC_WIDTH`, `N_LAYERS`.
- **Larghezza di rete a runtime**: `n_inputs_real`/`n_neurons_real` per-layer, un solo bitstream per ogni topologia fino al massimo.
- Attivazioni configurabili: `ACT_RELU` (default) e `ACT_NONE` (lineare con saturazione bilaterale), con saturazione INT8.
- **Due tipi di rete**: classica multi-layer dense (`layer_sequencer`, buffer ping-pong) e **grafo arbitrario sparse** (`graph_engine` + buffer di attivazione in block RAM `DP16KD`), selezionabili a runtime.
- Sottosistema di **memoria dedicata**: interfaccia byte↔word, controller PSRAM parallelo asincrono con **page mode** (70 ns accesso casuale, 20 ns burst di pagina), 8 MB indirizzabili (23 bit).
- Interfaccia host **SPI Mode 0** MSB-first, `SET_NET_TYPE`+dispatch, `STATUS.done` sticky/clear-on-read, `READ_CONFIG` runtime.
- **Sottosistema flash** boot/persistenza: accesso esclusivo della FPGA a una `W25Q128JV` SPI NOR (16 MB) via SPI master dedicato, copy engine flash↔PSRAM e catalogo a 16 slot con CRC32, 8 opcode host.
- Verificato in **simulazione** (Icarus Verilog) e **sintesi reale** (Yosys + nextpnr-ecp5 + ecppack).

Applicazioni

- Inferenza a bassa latenza deterministica come periferica di SoC Linux, Raspberry-Pi-like, ESP32, microcontrollori.
- Blocco hardware riusabile integrabile in progetti eterogenei (piattaforma, non singola rete).
- Edge AI su reti dense compatte quantizzate INT8.
- Off-loading del carico neurale dalla CPU host verso hardware dedicato con throughput prevedibile.

Target & toolchain

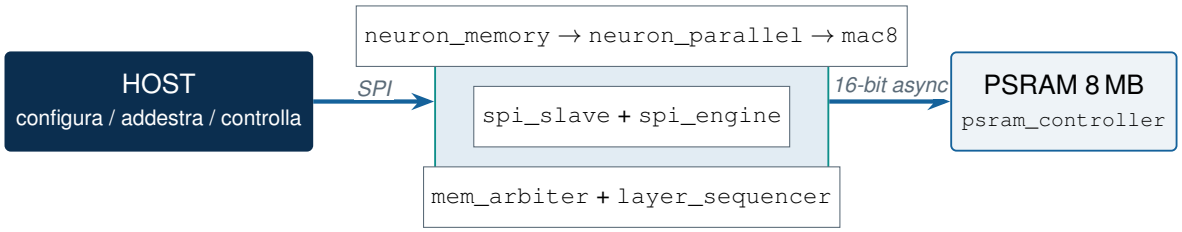
- FPGA: Lattice ECP5 `LFE5U-45F-8BG381C` (-8, CABGA381).
- Sintesi: Yosys; place&route: nextpnr-ecp5; bitstream: Project Trellis (`ecppack`).
- Simulazione: Icarus Verilog (`-g2012`).
- PSRAM: ISSI `IS66WVE4M16EBLL-70BLI` (64 Mb, 4M×16).

Parametri chiave (configurazione baseline caratterizzata)

Grandezza	Valore	Note
Precisione dati	INT8 (signed)	`DATA_WIDTH`=8
Accumulatore	INT32 (signed)	`ACC_WIDTH`=32
Ingressi / neuroni	256 / 4	baseline benchmark datapath
MAC simultanei	$2 \ldots 64$	=`PARALLEL`×`N_NEURONS`
Attivazioni	ReLU, lineare	`ACT_RELU` / `ACT_NONE`
Fmax (P=2, datapath)	87.88 MHz	benchmark datapath isolato
Fmax (P=2, sistema integrato)	67.91 MHz	sistema completo incl. sottosistema flash, place&route reale

Throughput MAC (P=16)	≈3.34 G MAC/s	teorico, solo datapath
Memoria di lavoro	8 MB PSRAM	bus parallelo 16-bit, 70 ns / 20 ns page mode
Spazio indirizzi	23 bit (byte)	`ADDR_WIDTH`=23

Diagramma a blocchi del sistema



Il datapath neurale è interamente nell'FPGA; l'host non partecipa alle singole operazioni MAC.

Sunto del pinout — collegamento pin per pin

Tabella di riferimento rapido: i **57 segnali reali** del top-level `spi_neuron_top`, ciascuno con la propria ball `CABGA381` individuale (**non** un intervallo di bus) — dati reali dal database di dispositivo di Project Trellis (`iodb.json`), **verificati da un place&route `nextpnr-ecp5` completo a 0 errori** (non un pinout pianificato). Descrizione completa, razionale di collocazione per banco e schema di collegamento pin-per-pin verso la PSRAM ISSI: cap. 12.

Segnale	Ball	Banco	Dir	Pin corrispondente / funzione
Clock e reset				
`clk`	H5	7	IN	Clock di sistema, pad `GR_PCLK7_0` (clock globale dedicato).
`rst`	B4	7	IN	Reset globale sincrono, attivo alto.
SPI applicativo (host ↔ FPGA, Mode 0)				
`sclk`	B5	7	IN	SPI clock (CPOL=0, CPHA=0).
`mosi`	C5	7	IN	Master-Out Slave-In.
`miso`	A3	7	OUT	Master-In Slave-Out.
`cs_n`	B3	7	IN	Chip-select, attivo basso.
Attenzione host (attivi bassi, di livello)				
`data_ready_n`	C3	7	OUT	Basso finché un risultato attende lettura.
`irq_n`	C4	7	OUT	Basso se il guard load-time del grafo è scattato.
Flash subsystem — SPI verso W25Q128JV (boot/persistenza)				
`flash_sclk`	E3	7	OUT	SPI clock verso la flash — GPIO ordinario, indipendente (Fase F7, cap. 12).
`flash_mosi`	D3	7	OUT	Master-Out Slave-In verso la flash NOR onboard.
`flash_miso`	D5	7	IN	Master-In Slave-Out dalla flash.
`flash_cs_n`	E4	7	OUT	Chip-select flash, attivo basso.
Bus indirizzi PSRAM — `psram_a[21:0]` (22 linee reali)				
`psram_a[0]`	E16	2	OUT	PSRAM A0
`psram_a[1]`	F16	2	OUT	PSRAM A1
`psram_a[2]`	D18	2	OUT	PSRAM A2
`psram_a[3]`	E17	2	OUT	PSRAM A3
`psram_a[4]`	E18	2	OUT	PSRAM A4
`psram_a[5]`	F18	2	OUT	PSRAM A5
`psram_a[6]`	F17	2	OUT	PSRAM A6
`psram_a[7]`	G16	2	OUT	PSRAM A7
`psram_a[8]`	G18	2	OUT	PSRAM A8
`psram_a[9]`	H16	2	OUT	PSRAM A9
`psram_a[10]`	H17	2	OUT	PSRAM A10
`psram_a[11]`	H18	2	OUT	PSRAM A11
`psram_a[12]`	J16	2	OUT	PSRAM A12
`psram_a[13]`	J17	2	OUT	PSRAM A13
`psram_a[14]`	C20	2	OUT	PSRAM A14
`psram_a[15]`	D19	2	OUT	PSRAM A15
`psram_a[16]`	E19	2	OUT	PSRAM A16
`psram_a[17]`	E20	2	OUT	PSRAM A17
`psram_a[18]`	F19	2	OUT	PSRAM A18
`psram_a[19]`	F20	2	OUT	PSRAM A19
`psram_a[20]`	G20	2	OUT	PSRAM A20

`psram_a[21]`	H20	2	OUT	PSRAM A21
`psram_a[22]`	P18	3	OUT	Sempre 0 (shift byte→word) — NC su scheda.
Bus dati PSRAM — `psram_dq[15:0]` (bidirezionale)				
`psram_dq[0]`	K18	2	IO	PSRAM DQ0
`psram_dq[1]`	C18	2	IO	PSRAM DQ1
`psram_dq[2]`	D17	2	IO	PSRAM DQ2
`psram_dq[3]`	D20	2	IO	PSRAM DQ3
`psram_dq[4]`	G19	2	IO	PSRAM DQ4
`psram_dq[5]`	J18	2	IO	PSRAM DQ5
`psram_dq[6]`	J19	2	IO	PSRAM DQ6
`psram_dq[7]`	J20	2	IO	PSRAM DQ7
`psram_dq[8]`	K19	2	IO	PSRAM DQ8
`psram_dq[9]`	K20	2	IO	PSRAM DQ9
`psram_dq[10]`	L17	3	IO	PSRAM DQ10
`psram_dq[11]`	M18	3	IO	PSRAM DQ11
`psram_dq[12]`	M17	3	IO	PSRAM DQ12
`psram_dq[13]`	N16	3	IO	PSRAM DQ13
`psram_dq[14]`	N18	3	IO	PSRAM DQ14
`psram_dq[15]`	P17	3	IO	PSRAM DQ15
Controllo PSRAM				
`psram_ce_n`	N17	3	OUT	PSRAM CE# — chip enable, attivo basso.
`psram_oe_n`	R16	3	OUT	PSRAM OE# — output enable (lettura).
`psram_we_n`	R17	3	OUT	PSRAM WE# — write enable (scrittura).
`psram_lb_n`	T16	3	OUT	PSRAM LB# — lower-byte enable (DQ[7:0]).
`psram_ub_n`	N19	3	OUT	PSRAM UB# — upper-byte enable (DQ[15:8]).
`psram_zz_n`	N20	3	OUT	PSRAM ZZ# — sleep/snooze (alto in funzionamento).

Standard I/O: LVCMOS33 su tutti i 57 segnali. Ball di JTAG e config-SPI di boot (pin dedicati a funzione fissa, senza porta RTL) non compaiono in questa tabella — vedi cap. 12 §“Configurazione e programmazione”. Sorgente: `synth/ecp5/spi_neuron_top.lpf`, generato da `tools/pinout/gen_lpf.py` contro `iodb.json` di Project Trellis.

Indice

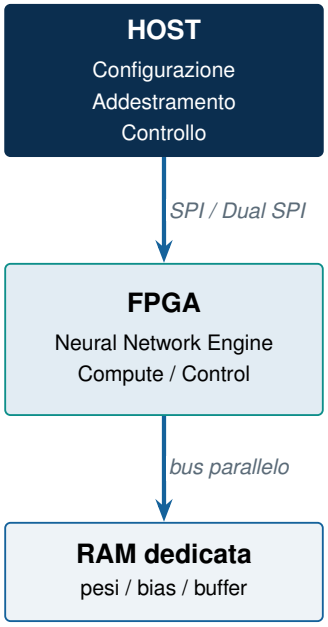
CAPITOLO 1

Panoramica del sistema

1.1 Obiettivo del progetto

FPGA-Neural implementa un **Neural Network Engine riusabile in hardware FPGA**. L'insieme è composto da tre elementi: l'FPGA, che è il vero acceleratore; una RAM dedicata fisicamente associata all'FPGA e non condivisa con l'host; e un'interfaccia host indipendente dal sistema operativo, inizialmente SPI (con possibile estensione futura a Dual SPI).

Il principio fondante è la separazione fra chi *esegue* il calcolo e chi lo *usa*: il calcolo della rete neurale avviene interamente dentro l'FPGA, mentre il sistema host fornisce solo configurazione, parametri di rete, dati di ingresso, controllo e lettura dei risultati. L'host non fa parte del datapath computazionale. Sistemi host possibili includono SoC Linux, sistemi tipo Raspberry Pi, ESP32, microcontrollori e PC di sviluppo: la stessa architettura di engine deve poter essere usata in sistemi completamente diversi.



1.2 Configurazione hardware contro configurazione di rete

Il progetto distingue con precisione fra l'**architettura hardware** dell'acceleratore e i **parametri della rete neurale**.

L'architettura fisica dell'engine è definita al momento della sintesi e dell'implementazione dell'FPGA. I parametri hardware tipici sono `N_INPUTS`, `N_NEURONS`, `N_LAYERS`, `PARALLEL`, `DATA_WIDTH`, `ACC_WIDTH`: sono parametri Verilog risolti in fase di sintesi e determinano il datapath contenuto nel bitstream. I parametri della rete — pesi, bias, parametri di attivazione e di quantizzazione, costanti specifiche — vengono invece caricati a runtime attraverso l'interfaccia host e memorizzati nella RAM associata all'FPGA.

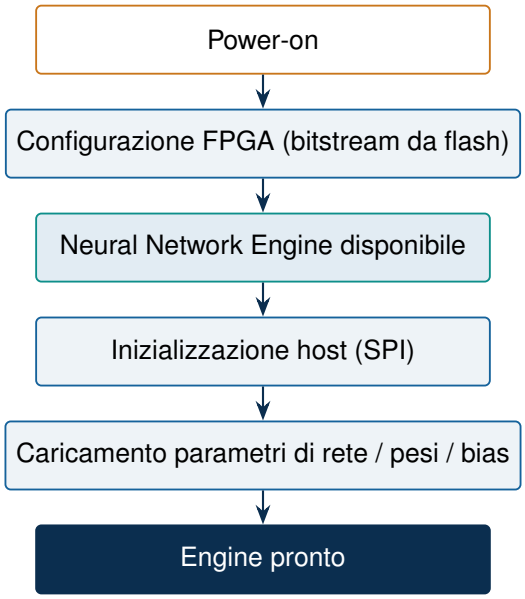
Principio architetturale centrale

Una build fissa il *soffitto* della macchina (numero massimo di layer, larghezza massima, `PARALLEL`); l'host configura la rete *reale* — numero di layer, larghezza ingressi/uscite per-layer, attivazione per-layer e parametri addestrati — interamente a runtime, via SPI, nella memoria

locale dell'FPGA. Un solo bitstream serve qualunque topologia fino a quel soffitto.

1.3 Boot e inizializzazione

L'FPGA viene configurato all'accensione tramite il consueto meccanismo di configurazione (caricamento del bitstream da flash SPI). Il bitstream definisce l'architettura hardware dell'engine; l'host non costruisce dinamicamente il datapath durante il funzionamento normale, ma configura i dati di rete su cui il datapath già esistente opera.



1.4 Addestramento e inferenza

Addestramento e inferenza sono concettualmente separati. La prima implementazione non richiede che l'FPGA esegua l'addestramento: i pesi possono essere calcolati esternamente (PC/Linux/altro host) e trasferiti via SPI nella RAM dell'FPGA, che poi esegue l'inferenza. Questo riduce drasticamente la complessità dell'hardware iniziale, senza precludere una futura implementazione di training assistito o interamente hardware (Fase 8 della roadmap, cap. 14). Durante l'inferenza l'host fornisce solo i dati di ingresso e recupera il risultato, ottenendo calcolo deterministico, carico ridotto sull'host, parallelismo hardware, latenza prevedibile e indipendenza dall'architettura della CPU host.

1.5 Filosofia di progetto e riuso

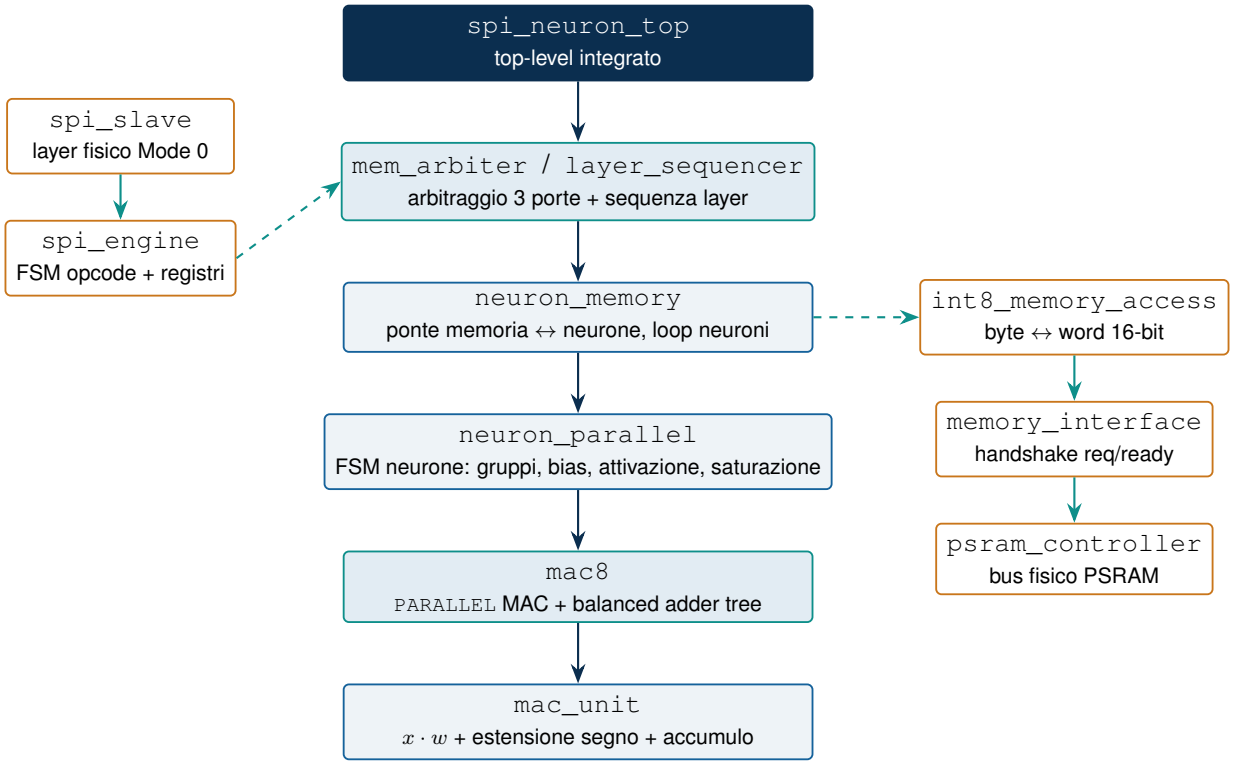
Il progetto va inteso come una *piattaforma di accelerazione neurale FPGA riusabile* più che come una singola rete. L'applicazione determina dimensione degli ingressi, topologia, numero di layer e neuroni, parallelismo, precisione numerica, funzioni di attivazione, requisiti di memoria e prestazioni; il processo di generazione hardware produce l'implementazione FPGA corrispondente. La stessa architettura HDL rimane concettualmente invariata mentre i parametri di sintesi generano implementazioni appropriate ai diversi target applicativi.

CAPITOLO 2

Architettura RTL e gerarchia dei moduli

2.1 Organizzazione gerarchica

Il design è organizzato per livelli, dal moltiplicatore-accumulatore elementare fino al top-level integrato con interfaccia SPI e PSRAM. Ogni livello incapsula il precedente e ne astrae i dettagli: il datapath validato (`mac_unit`, `mac8`, `neuron_parallel`) non viene mai modificato dai livelli di orchestrazione superiori.



2.2 Ruolo di ciascun modulo

Modulo	Funzione
`mac_unit`	Singolo prodotto-accumulatore: $\text{acc\_out} = \text{acc\_in} + (x \cdot w)$, con estensione di segno del prodotto ad `ACC_WIDTH`. Parametrico su `DATA_WIDTH`/`ACC_WIDTH`.
`mac8`	`PARALLEL` istanze di `mac_unit` i cui prodotti vengono sommati da un *balanced binary adder tree* di profondità $\log_2(\text{PARALLEL})$; il risultato è aggiunto all'accumulatore in ingresso.
`neuron_parallel`	FSM di un singolo neurone: elabora `N_INPUTS` ingressi in gruppi di `PARALLEL`, accumula tra i gruppi, somma il bias, applica l'attivazione e satura a INT8. Include il guard di elaborazione su `N_INPUTS % PARALLEL` e la larghezza runtime `n_inputs_real`.

`layer`	Istanzia `N_NEURONS` neuroni *in parallelo* sullo stesso vettore di ingresso; `busy`=OR, `done`=AND dei neuroni. Percorso puramente combinatorio-di-dati usato nei benchmark del datapath.
`neuron_memory`	Integra il calcolo con la memoria: legge X (condiviso) una volta, poi per ogni neurone rilegge W e bias dalla RAM e riusa una singola istanza `neuron_parallel` (memory-bound, un neurone per volta). Uscita `y_bus` packed neuron-major.
`layer_sequencer`	Concatena fino a `N_LAYERS` esecuzioni di `neuron_memory` leggendo una tabella descrittori scritta dall'host e alternando i buffer ping-pong in RAM (Fase 5).
`act_buffer`	Buffer di attivazione globale in block RAM `DP16KD`, indicizzato per id di segnale (Tipo #2).
`graph_engine`	Motore della rete a grafo (Tipo #2): gather da `act_buffer`, riusa `neuron_parallel`, scrive le uscite per id (cap. 7).
`int8_memory_access`	Converte l'interfaccia byte/INT8 (indirizzo di byte) nell'interfaccia a parola 16-bit, selezionando il byte basso/alto tramite `lb_n`/`ub_n` e `addr»1`.
`memory_interface`	FSM di handshake a 2 stati (IDLE/WAIT) che serializza la singola transazione verso il controller.
`psram_controller`	Controller del bus PSRAM parallelo asincrono con **page mode** di lettura: accesso casuale a 70 ns (`tAA`), burst nella stessa pagina a 20 ns (`tAPA`) con CE#/OE# tenuti attivi; abilita il page mode sul chip all'avvio via registro di configurazione (cap. 5, § 5.5). Pilota `ce_n/oe_n/we_n/lb_n/ub_n/zz_n` e il bus dati tri-state.
`mem_arbiter`	Arbitro a priorità fissa (B>C>A) fra tre master byte-level: `spi_engine` (A), `neuron_memory` (B), `layer_sequencer` (C).
`spi_slave`	Layer fisico SPI Mode 0, MSB-first, sincronizzatore CDC a 3 stadi su SCLK/MOSI/CS_N, shift-register e framing di CS.
`spi_engine`	FSM di protocollo/opcode e banco registri (`x_base`, `w_base`, `bias_addr`, base ping-pong, attivazione, larghezze runtime…), con `STATUS.done` sticky/clear-on-read.
`spi_neuron_top`	Top-level: collega SPI, arbitro, sequencer, `neuron_memory` e catena PSRAM; multiplexa il controllo di `neuron_memory` fra sequencer e percorso diretto single-layer.

Modelli di simulazione

`psram_model.v` (in `sim/`) e `memory_model.v` sono modelli comportamentali della memoria usati nei testbench; non fanno parte del design sintetizzabile ma riproducono la latenza reale per la verifica end-to-end.

2.3 Due percorsi di esecuzione

Il top-level espone due modalità mutuamente esclusive verso lo stesso motore di calcolo `neuron_memory`:

- **Percorso single-layer / manuale**: l'host imposta le basi con `SET_BASE`, avvia con `START` e legge con `READ_OUTPUT`. `spi_engine` pilota direttamente `neuron_memory`.
- **Percorso multi-layer**: l'host scrive la tabella descrittori e avvia con `RUN_NETWORK`; `layer_sequencer` prende possesso del controllo di `neuron_memory` (mentre `seq_busy` è alto) e concatena i layer.

Il multiplexer del top-level commuta le linee di controllo di `neuron_memory` in base a `seq_busy`, restituendo il motore al percorso diretto a fine sequenza.

CAPITOLO 3

Datapath di calcolo

3.1 Catena aritmetica INT8/INT32

Il datapath elementare implementa la sequenza tipica di un neurone quantizzato:

Ogni prodotto INT8×INT8 sta in 16 bit; viene esteso con segno a 32 bit prima dell'accumulo, così l'accumulatore non trabocca su vettori lunghi. Bias e attivazione operano a 32 bit; solo l'uscita finale viene saturata a INT8.

3.2 `mac_unit` — moltiplicatore-accumulatore

Il modulo `mac_unit` è puramente combinatorio e parametrico su `DATA_WIDTH` e `ACC_WIDTH`. Calcola:

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

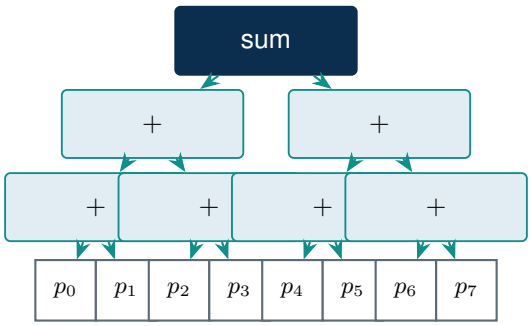
Il prodotto ha larghezza $2\times$`DATA_WIDTH` e viene esteso con segno replicando il bit più significativo. Su ECP5 la moltiplicazione mappa su un blocco DSP `MULT18X18D`.

Listing 3.1. `rtl/mac_unit.v` — nucleo aritmetico

```
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` — MAC parallelo e balanced adder tree

`mac8` istanzia `PARALLEL` unità `mac_unit` che generano `PARALLEL` prodotti indipendenti, poi li somma con un *albero di addizione binario bilanciato*. Rispetto alla riduzione lineare $((((p_0{+}p_1){+}p_2){+}p_3){+}\dots)$, di profondità $O(\text{PARALLEL})$, l'albero ha profondità $O(\log_2 \text{PARALLEL})$, riducendo drasticamente il percorso combinatorio.



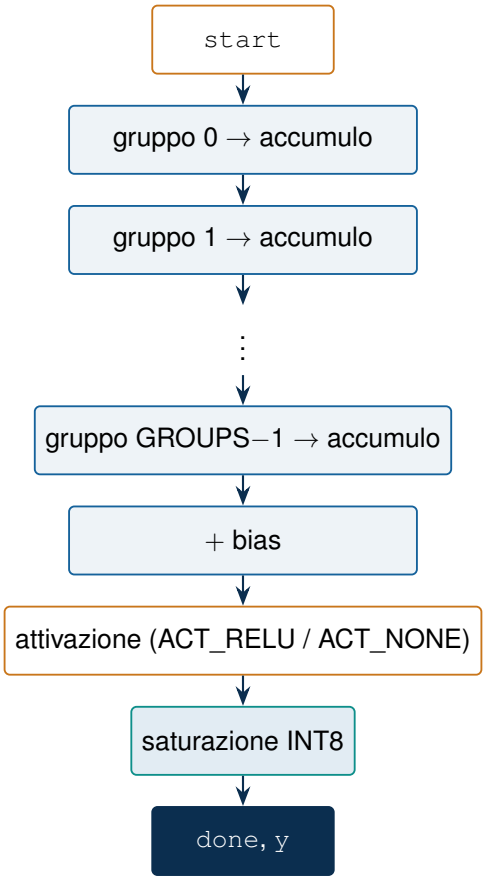
Esempio con PARALLEL=8: 3 livelli. PARALLEL=16 → 4 livelli; PARALLEL=32 → 5 livelli.

PARALLEL potenza di due

L'albero è pensato per `PARALLEL` potenza di due (8, 16, 32...). Questo è anche il valore usato in tutte le configurazioni del progetto.

3.4 `neuron_parallel` — FSM del neurone

`neuron_parallel` elabora `N_INPUTS` ingressi in gruppi di `PARALLEL`, mantenendo l'accumulatore tra un gruppo e il successivo. Alla fine somma il bias, applica l'attivazione e satura a INT8. Il numero di gruppi è GROUPS $=$ N_INPUTS/PARALLEL.



3.4.1 Guard di parametri (elaboration-time)

Se `PARALLEL` non divide esattamente `N_INPUTS` si verificano due guasti, entrambi confermati empiricamente in `sim/parameter_sweep_tb.v`:

- la divisione intera tronca `GROUPS` e gli ingressi in eccesso non vengono mai letti → risultato **errato**, senza errore né avviso;
- se `PARALLEL > N_INPUTS`, `GROUPS=0` e la condizione terminale non è mai soddisfatta → il neurone **si blocca** (busy alto, done mai asserito).

La soluzione non modifica il datapath validato: un blocco `generate` istanzia un modulo deliberatamente indefinito quando N_INPUTS $\bmod$ PARALLEL $\neq 0$, forzando un errore in *elaborazione* sia in simulazione sia in sintesi. Per le configurazioni valide il ramo non viene mai elaborato.

Listing 3.2. `rtl/neuron_parallel.v` — guard di parametri

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

Caso limite `N_INPUTS=0` (corretto 2026-09-04)

La condizione originale (`N_INPUTS % PARALLEL != 0`) non intercetta `N_INPUTS=0`, poiché $0 \bmod \text{PARALLEL} = 0$ per ogni `PARALLEL`: il modulo elaborava con successo (sia in simulazione sia in sintesi reale Yosys) lasciando `x_bus`/`w_bus` non pilotati e `start` silenziosamente inefficace. Trovato durante la campagna di ri-certificazione (`docs/validation/bugs.md`, BUG-002) e corretto estendendo il guard come sopra — `N_INPUTS=0` ora fallisce l'elaborazione esattamente come gli altri casi degeneri.

3.5 Funzioni di attivazione

`neuron_parallel` accetta una porta `activation` a 2 bit. Il default è `ACT_RELU`, l'unico comportamento esistente prima dell'introduzione della porta, così ogni chiamante preesistente resta invariato.

Codifica	Valore	Comportamento
`ACT_NONE`	`2'd0`	Lineare: nessun clamp a zero, saturazione bilaterale al range INT8 $[-128, +127]$.
`ACT_RELU`	`2'd1`	$\max(0, x)$, poi saturazione positiva a $+127$ (default; fallback anche per codifiche riservate).

3.6 Saturazione INT8

Dopo bias e attivazione, l'accumulatore a 32 bit viene ridotto a INT8:

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

3.7 `layer` — neuroni in parallelo

`layer` istanzia `N_NEURONS` neuroni che condividono il vettore di ingresso `x_bus` ma hanno pesi e bias distinti; `busy` è l'OR e `done` l'AND dei segnali dei neuroni. È il modulo usato nei benchmark del datapath (cap. 11), dove tutti i neuroni lavorano simultaneamente. La convenzione di indirizzamento è neuron-major: i pesi del neurone n occupano `weights_bus[n*N_INPUTS*DATA_WIDTH +: N_INPUTS*DATA_WIDTH]`.

CAPITOLO 4

Parametri e configurabilità

4.1 Parametri di build (synthesis-time)

L'architettura hardware è fissata alla sintesi tramite i parametri Verilog seguenti. Determinano il datapath contenuto nel bitstream e il suo *soffitto* di capacità.

Parametro	Default	Significato
`DATA_WIDTH`	8	Larghezza dei dati (INT8).
`ACC_WIDTH`	32	Larghezza dell'accumulatore (INT32).
`N_INPUTS`	32 / 256	Numero massimo di ingressi per neurone (baseline benchmark: 256).
`N_NEURONS`	1 / 4	Numero massimo di neuroni per layer.
`PARALLEL`	8	MAC hardware simultanei per neurone; deve dividere `N_INPUTS` e conviene sia potenza di due.
`N_LAYERS`	4	Numero massimo di layer concatenabili da `layer_sequencer`.
`ADDR_WIDTH`	23	Larghezza dell'indirizzo di byte (8 MB).
`MEM_DATA_WIDTH`	16	Larghezza del bus dati fisico PSRAM.
`CLK_FREQ_MHZ`	80	Frequenza usata per le formule di temporizzazione PSRAM (va allineata all'oscillatore reale).

Vincolo `N_INPUTS % PARALLEL`

`PARALLEL` deve dividere esattamente `N_INPUTS`, altrimenti scatta il guard di elaborazione (§3). Lo stesso vincolo vale a runtime su `n_inputs_real`.

4.2 Larghezza di rete a runtime

Un singolo bitstream serve qualunque topologia *fino* al massimo di build. La larghezza reale di ciascuna esecuzione è un valore separato, impostato dall'host:

- `n_inputs_real` — ingressi realmente usati in questa esecuzione (deve essere multiplo di `PARALLEL`);
- `n_neurons_real` — neuroni realmente calcolati in questa esecuzione.

Entrambi hanno default pari al massimo di build, così ogni chiamante che li lascia scollegati elabora l'intera larghezza come prima dell'introduzione delle porte.

Terminazione anticipata reale

Non si tratta di semplice contabilità di indirizzi: i due valori limitano direttamente i loop hardware (letture X/W di `neuron_memory`, conteggio gruppi MAC di `neuron_parallel` e lunghezza della copia ping-pong per `RUN_NETWORK`). Un layer più stretto *calcola* e *copia* davvero più in fretta e non richiede zero-padding della RAM per la coda non usata: i dati oltre `n_inputs_real`/`n_neurons_real` non vengono mai letti.

Questo permette a una rete di rastremarsi dentro una sola esecuzione concatenata, ad esempio $256 \to 64 \to 16 \to 4$, con ogni layer che dichiara la propria larghezza reale nella tabella descrittori (cap. 6).

4.2.1 Risparmio misurato

La terminazione anticipata è stata misurata end-to-end:

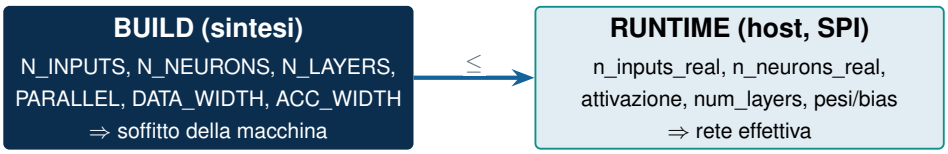
Test	Cicli	Confronto
`neuron_parallel_tb.v` (T7)	3 vs 6	ridotto vs pieno, con dati "spazzatura" nelle corsie saltate (prova che non vengono lette).
`neuron_memory_tb.v` (T5)	209 vs 788	8-di-32 vs 32 pieni, attraverso lo stack PSRAM reale.

4.3 Configurazioni caratterizzate

Alcune combinazioni convalidate in simulazione e/o sintesi:

N_INPUTS	N_NEURONS	PARALLEL	Note
32	4	8	Primo test parametrico funzionale (Fase 1).
256	4	2/4/8/16	Sweep di benchmark del datapath (Fase 7).
32	1..3	8	Integrazione memoria mono/multi-neurone (Fase 3).
4	4	2	Test end-to-end `RUN_NETWORK` a 2 layer su SPI reale.

4.4 Riepilogo build contro runtime

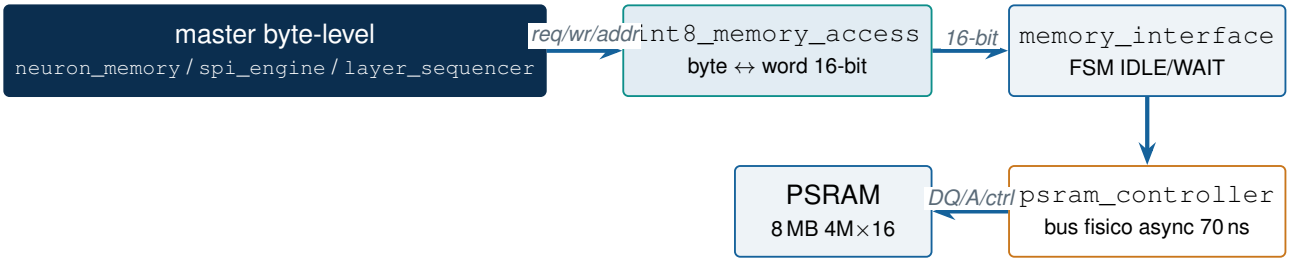


CAPITOLO 5

Sottosistema di memoria

5.1 Catena di memoria

Il motore di calcolo lavora con indirizzi e dati a livello di *byte* (INT8), mentre la PSRAM è un dispositivo a parola da 16 bit. Tre moduli in cascata realizzano la conversione e l'accesso fisico:



5.2 `int8_memory_access` — conversione byte/word

Converte l'interfaccia INT8 (indirizzo di byte) nell'interfaccia a parola. L'indirizzo di byte viene diviso per due (`addr»1`) per ottenere l'indirizzo di parola; il bit meno significativo seleziona il byte:

- `addr[0]=0` → byte basso: `lb_n=0`, `ub_n=1`, dato su DQ[7:0];
- `addr[0]=1` → byte alto: `lb_n=1`, `ub_n=0`, dato su DQ[15:8].

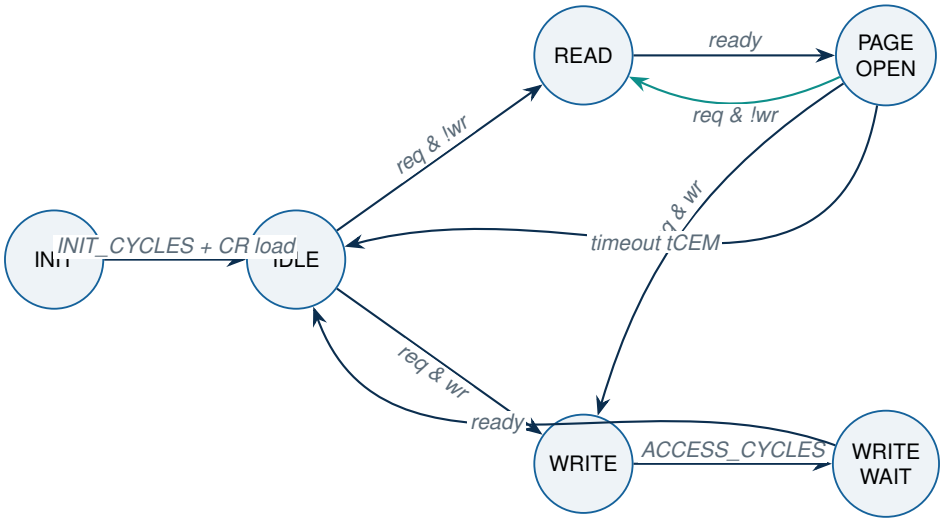
In lettura estrae il byte corretto da `mem_rdata`. La FSM ha due stati (IDLE, WAIT) e restituisce `ready` come impulso di un ciclo.

5.3 `memory_interface` — handshake

FSM a due stati che serializza una singola transazione: in IDLE, alla richiesta `req`, latcha `wr/addr/wdata/lb_n/ub_n` ed emette un impulso `mem_req` di un ciclo verso il controller; in WAIT attende `mem_ready`, cattura `rdata` in lettura e asserisce `ready`. Garantisce il contratto "una transazione per volta".

5.4 `psram_controller` — bus fisico

Controller del bus PSRAM parallelo asincrono, con supporto al **page mode** di lettura del chip (§ 5.5). La macchina a stati principale è:



Da INIT il controller passa automaticamente per una sotto-sequenza di caricamento del registro di configurazione (`STATE_CR_INIT`, 4 passi) prima di raggiungere IDLE per la prima volta — vedi § 5.5. La transizione PAGE OPEN → WRITE (freccia in basso a destra) passa internamente per due micro-stati di transito, `STATE_PAGE_CLOSE` e `STATE_PAGE_REOPEN` (un ciclo ciascuno): il primo forza CE#/OE# alti per almeno un ciclo prima che il controller inizi a pilotare il bus dati, evitando contesa con l'uscita ancora attiva della PSRAM ($\geq t_{HZ}$); il secondo riavvia la transazione già latchata esattamente come farebbe IDLE. Non sono disegnati come nodi separati per non appesantire la figura.

5.4.1 Temporizzazione

Formule di temporizzazione

ACCESS_CYCLES $= \lceil (70 \times \text{CLK\_FREQ\_MHZ})/1000 \rceil$ (latenza di accesso casuale, t_{AA}/t_{RC} = 70 ns)

PAGE_CYCLES $= \lceil (20 \times \text{CLK\_FREQ\_MHZ})/1000 \rceil$ (continuazione nella stessa pagina, t_{APA}/t_{PC} = 20 ns)

INIT_CYCLES $= 150 \times \text{CLK\_FREQ\_MHZ}$ (inizializzazione di power-up, t_{PU} = 150 µs)

PAGE_TIMEOUT_CYCLES $= \lceil (6000 \times \text{CLK\_FREQ\_MHZ})/1000 \rceil$ (chiusura automatica della pagina, margine di sicurezza sotto t_{CEM} = 8 µs)

Il bus dati è pilotato in tri-state: `psram_dq = dq_oe ? dq_out : Z`. In lettura `dq_oe=0`; in scrittura `dq_oe=1` durante l'impulso di `we_n`. Uno stato di WRITE_WAIT mantiene attivi `ce_n/lb_n/ub_n` per l'hold finale prima del rilascio.

Non è QSPI

Questa è un'interfaccia SRAM-asincrona classica, **non** QSPI: la maggior parte delle "PSRAM" serie/QSPI in commercio non è compatibile con questo controller senza riscrittura. Vedere il cap. 12 per la parte raccomandata (ISSI parallela).

5.5 Page mode di lettura

Il chip raccomandato (cap. 12) è "asynchronous/**page mode**": una volta fatto un primo accesso casuale a t_{AA} = 70 ns, letture successive all'interno della stessa pagina da 16 word (bit di indirizzo sopra `A[3]` invariati) costano solo t_{APA}/t_{PC} = 20 ns, perché CE#/OE# restano attivi e cambia solo il bus indirizzi. Il page mode è **disabilitato di default** all'accensione (bit 7 del registro di configurazione, CR = `0x0070` di default) e va abilitato esplicitamente.

- **Abilitazione all'avvio**: subito dopo INIT, il controller esegue la "software-access sequence" del datasheet (2 letture dummy + 2 scritture, `0x0000` di sblocco poi CR reale `0x00F0` = default con il bit Page attivo) all'indirizzo più alto del chip — riusa esattamente la stessa logica READ/WRITE di ogni altra transazione, quindi passa dagli stessi controlli di temporizzazione.
- **Burst di pagina**: dopo una READ il controller non chiude più CE#/OE# (stato PAGE OPEN). Una READ successiva nella stessa pagina aspetta solo PAGE_CYCLES; una READ che attraversa pagina resta comunque senza toggle di CE# ma paga un ACCESS_CYCLES pieno per quella parola (qualunque cambio a `A[4]` o superiore richiede un nuovo t_{AA}). Un contatore chiude la pagina prima del limite t_{CEM} con margine di sicurezza.
- **Solo una WRITE chiude la pagina.** I cambi di `lb_n`/`ub_n` *non* la chiudono: `int8_memory_access` alterna questi segnali a quasi ogni accesso (accesso byte-granulare su bus a 16 bit), quindi trattarli come condizione di chiusura — primo tentativo di implementazione — rendeva il workload reale *più lento*, non più veloce (misurato: 53.25→61.25 cicli/edge sul gather di `graph_engine`); rimosso, corretto a 53.25→37.53 cicli/edge (banda +42%, § 5.7).

Nessun beneficio senza pattern sequenziale

Il page mode accelera solo accessi che restano nella stessa pagina (o quasi) mentre il controller resta in attesa di una nuova richiesta con la pagina ancora aperta. Accessi isolati e sparsi (indirizzo casuale ogni volta) pagano comunque ACCESS_CYCLES pieno, più un piccolo overhead di chiusura/riapertura se preceduti da una WRITE o da un timeout t_{CEM}: non è un guadagno universale, dipende dal pattern di accesso del chiamante.

Fmax reale (`nextpnr-ecp5`, cap. 11) sul sistema integrato `spi_neuron_top` con Tipo #2 abilitato: **75.73 MHz** a `PARALLEL`=2 (era 55.59 MHz prima dell'aggiunta del page mode) e **65.13 MHz** a `PARALLEL`=8, entrambe ancora FAIL all'obiettivo di 80 MHz ma non regredite. Il percorso critico resta, in entrambi i casi, interamente dentro `u_graph_engine.u_neuron` (catena di accumulo `mac8`/`neuron_parallel`, cap. 11) — `psram_controller` non compare mai nel percorso critico nonostante la crescita di risorse del page mode.

5.6 Mappa degli indirizzi e convenzioni

Lo spazio di indirizzamento è di `ADDR_WIDTH`=23 bit (indirizzo di *byte*), per 8 MB pieni. Le regioni non hanno indirizzi cablati: le loro basi sono registri impostati dall'host via `SET_BASE` (percorso single-layer) o lette dalla tabella descrittori (percorso multi-layer).

Regione	Base	Contenuto / convenzione
Ingresso X	`x_base`	Vettore di ingresso condiviso, letto una volta per invocazione.
Pesi W	`w_base`	Neuron-major: i pesi del neurone n a `w_base + n*N_INPUTS` byte.
Bias	`bias_addr`	Un byte per neurone: bias del neurone n a `bias_addr + n`.
Tabella descrittori	`table_base`	`N_LAYERS` voci da 11 byte (cap. 6).
Buffer ping-pong A/B	`buf_a_base` / `buf_b_base`	Uscite intermedie tra layer.

5.6.1 Indirizzamento fisico della PSRAM

La PSRAM raccomandata è 4M×16 (8 MB), che richiede un indirizzo di parola a 22 bit (A0–A21). `int8_memory_access` calcola `addr»1` portando l'indirizzo di byte a 23 bit in un indirizzo di parola

a 22 bit che mappa esattamente su A0–A21; il bit 22 di `psram_a` è quindi sempre 0 e sul PCB restano 22 linee di indirizzo reali.

5.7 Larghezza di banda

Misurata sul gather della lista di edge di `graph_engine` (cap. 7), per differenza tra due dimensioni di grafo per isolare il costo per-edge dall'overhead fisso per-neurone (`sim/graph_engine_bandwidth_tb.v`):

	Prima (no page mode)	Dopo (page mode)	Δ
Cicli/edge	53.25	37.53	-29.5%
Banda @80 MHz	6.01 MB/s	8.53 MB/s	$+41.9\%$
Banda @16 MHz*	1.20 MB/s	1.71 MB/s	$+41.9\%$

*oscillatore reale raccomandato (cap. 12).

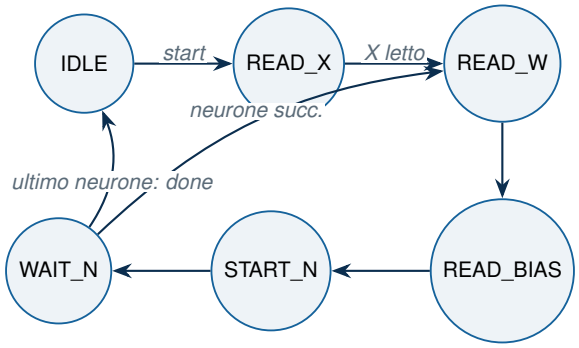
Il modello resta comunque memory-bound per costruzione: `neuron_memory` legge X una volta e rilegge W/bias per ciascun neurone (cap. 6), un neurone per volta; il page mode riduce il costo per-byte dell'accesso sequenziale, non elimina il pattern di accesso stesso.

CAPITOLO 6

Integrazione memoria, multi-neurone e multi-layer

6.1 `neuron_memory` — ponte memoria/neurone

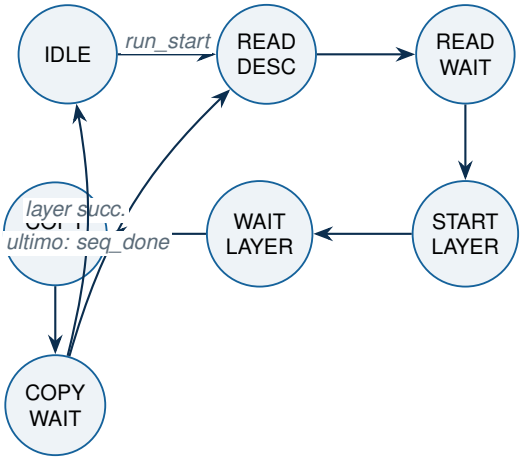
`neuron_memory` collega il datapath di calcolo alla memoria e gestisce il loop sui neuroni. Legge il vettore X una sola volta (ingresso condiviso), poi per ciascun neurone rilegge W e bias dalla RAM e li invia a una singola istanza riusata di `neuron_parallel`: il progetto è memory-bound, un neurone calcolato per volta, senza duplicare il datapath. L'uscita è `y_bus`, packed neuron-major (`DATA_WIDTH*N_NEURONS` bit).



Gli stati sono IDLE, READ_X, READ_W, READ_BIAS, START_N, WAIT_N. Dopo l'ultimo neurone la FSM torna in IDLE e asserisce `done`. Il conteggio di neuroni e ingressi realmente elaborati è dato da `n_neurons_real`/`n_inputs_real` (cap. 4).

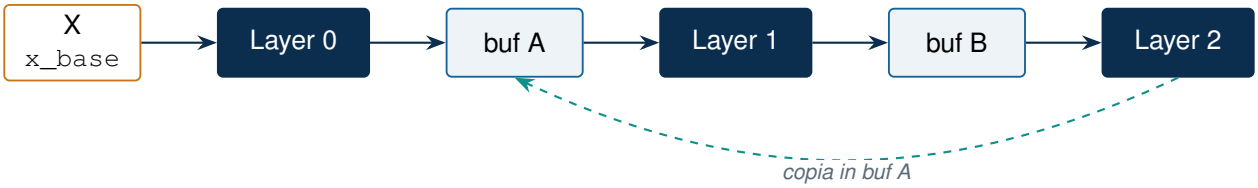
6.2 `layer_sequencer` — rete multi-layer

`layer_sequencer` concatena fino a `N_LAYERS` esecuzioni della stessa istanza `neuron_memory`, realizzando una rete densa feed-forward *senza* toccare il core di calcolo validato. Legge una tabella descrittori scritta dall'host e alterna i due buffer di uscita in RAM (ping-pong).



6.2.1 Buffer ping-pong

Il layer 0 legge l'ingresso esterno `x_base`. Il layer $k > 0$ legge dal buffer scritto dal layer $k-1$; l'uscita di ciascun layer viene copiata nell'altro buffer, alternando A e B. L'uscita finale resta sia in `y_bus` (leggibile con `READ_OUTPUT`) sia nel buffer ping-pong su cui è stata copiata.



6.2.2 Tabella descrittori

Scritta dall'host in RAM a `table_base` con `WRITE_RAM`; `N_LAYERS` voci da 11 byte ciascuna, MSB-first:

Campo	Byte	Significato
`w_base`	3	Base dei pesi del layer.
`bias_addr`	3	Base dei bias del layer.
`activation`	1	Attivazione del layer (2 bit bassi, cfr. `ACT_*`).
`n_inputs_real`	2	Ingressi reali del layer (multiplo di `PARALLEL`).
`n_neurons_real`	2	Neuroni reali del layer.
Totale	**11**	per voce/layer

Copia proporzionale alla larghezza reale

Il sequencer copia esattamente `n_neurons_real` byte di `y_bus` nel buffer ping-pong (non l'intera larghezza di build): un layer più stretto viene copiato più in fretta, senza zero-padding in RAM. Ogni attivazione è letta per-layer dalla tabella, indipendente dal registro `activation` del percorso single-layer.

6.3 Gerarchia dei segnali `busy`/`done`

Nel percorso multi-layer, `STATUS.busy` è l'OR dei busy single-layer e sequencer, mentre `STATUS.done` latcha solo al completamento dell'*ultimo* layer, non a ogni layer intermedio (cap. 8). Il top-level restituisce il controllo di `neuron_memory` al percorso diretto `START` al termine della sequenza.

CAPITOLO 7

Configurazione a due livelli: rete a grafo (Tipo #2)

7.1 Due tipi di rete

L'engine espone due *tipi di rete* selezionabili dall'host, con lo stesso comando di avvio che instrada verso il motore corretto:

- **Tipo #1 — rete classica (dense).** Layer con neuroni per layer, fully connected tra layer consecutivi. È il percorso di `layer_sequencer` (cap. 6), avviato da `RUN_NETWORK`. Connessioni *implicite per posizione*: non si enumera nulla, si definiscono solo i pesi indirizzati come `w_base + k*n_inputs + j`.
- **Tipo #2 — grafo arbitrario (sparse).** A partire dagli id dei neuroni di ingresso si definiscono le connessioni di ogni neurone fino all'uscita, tramite una *edge-list sparsa* per-neurone. Connessioni *esplicite per enumerazione*: ogni connessione è un edge `(src_id, peso)`; se non è nella lista, non esiste.

La differenza in una riga

Dense: definisci i *pesi* per posizione in una matrice. Graph: definisci ogni *connessione* come edge `(src_id, peso)` in una lista per-neurone. Le due tabelle descrittori hanno lo stesso formato di 11 byte ma campi diversi; il registro `net_type` dice al motore quale interpretazione usare.

7.2 Buffer di attivazione globale

Il Tipo #2 introduce un **buffer di attivazione** indicizzato per *id di segnale*, un byte INT8 per id, realizzato in **block RAM on-chip** `DP16KD` (`rtl/act_buffer.v`). Gli id `0..N_in-1` sono gli ingressi; ogni neurone scrive la propria uscita nel proprio id. Il gather delle sorgenti legge da qui a *accesso random a un ciclo*: è ciò che rende economico il grafo, perché è l'accesso che la PSRAM (70 ns, sequenziale) non potrebbe accelerare.

Dimensionamento V1

`N_TOTAL`=4096 segnali, id a 16 bit (spazio fino a 65.536 senza cambiare formato). Buffer = 4 KB, cioè 2 blocchi `DP16KD` su 108. Il vincolo reale diventa la capacità PSRAM per gli edge (≈2 M edge a 4 B), non la block RAM.

7.3 DAG feed-forward e vincolo `src_id < out_id`

Il grafo è un DAG feed-forward: ogni connessione punta a un id **già calcolato** (`src_id < out_id`). I neuroni si elaborano in ordine di id crescente, così quando si calcola un neurone tutte le sue sorgenti sono pronte nel buffer. Cicli e ricorrenza sono fuori scope per la V1. Il vincolo è verificato a due livelli: dall'assemblatore host (a compile time) e da un guard a runtime in `graph_engine` (`STATUS.err`), nella stessa filosofia del guard di elaborazione su `N_INPUTS % PARALLEL`.

7.4 Formati dati

Entrambi i descrittori sono da 11 byte/voce, MSB-first, a `table_base`.

7.4.1 Descrittore Tipo #2 (grafo)

Campo	Byte	Significato
`conn_ptr`	3	Indirizzo byte in PSRAM del blocco edge del neurone.
`n_conn`	2	Connessioni reali (pre-padding).
`out_id`	2	Id in cui scrivere l'uscita del neurone.
`activation`	1	`ACT_RELU` / `ACT_NONE` (2 bit bassi).
`bias`	1	Bias del neurone (INT8).
`reserved`	2	0.
Totale	**11**	voci in ordine di `out_id` crescente

7.4.2 Edge del grafo (4 byte, allineato)

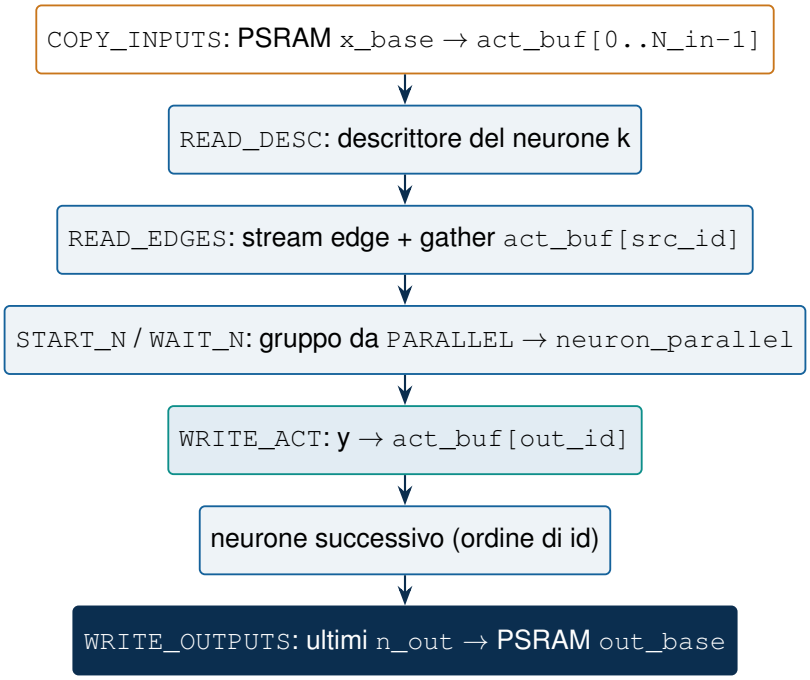
Campo	Byte	Significato
`src_id`	2	Id sorgente (uint16 BE).
`weight`	1	Peso (INT8).
`reserved`	1	0 (allineamento a 4 byte).

Padding a `PARALLEL`

`n_conn` arbitrario non è multiplo di `PARALLEL`: la edge-list del neurone è riempita fino al multiplo con edge a **peso zero** (spreco $\leq$`PARALLEL`-1 per neurone). Così il datapath e il suo guard restano intatti.

7.5 `graph_engine` — motore del grafo

`rtl/graph_engine.v` orchestra il Tipo #2 **riusando `neuron_parallel` senza modificarlo**, come fa `neuron_memory` per il caso denso. Differenza chiave: tra i due modi cambia *solo l'indirizzamento di X*. In Tipo #1 l'input è contiguo (`x_base + i`); in Tipo #2 è un gather (`act_buf[src_id]`). Il core aritmetico non si tocca.



Le uscite sono gli **ultimi `n_out`** id: nei DAG con l'ordinamento `src_id < out_id` i neuroni di uscita (sink, non riusati come sorgente) finiscono naturalmente con gli id più alti. A fine esecuzione `graph_engine` copia questi `n_out` byte in una regione PSRAM a `out_base`, che l'host rilegge con `READ_RAM`.

7.6 Opcode e registri del Tipo #2

La selezione del tipo avviene con un nuovo opcode; `RUN_NETWORK` fa il dispatch sul registro `net_type` (dettagli in cap. 8).

Opcode / sel	Nome	Funzione
`0x11`	SET_NET_TYPE	`type(1B)`: `0x01`=dense (#1), `0x02`=graph (#2). Default dopo `RESET`=dense.
`SET_BASE sel 9`	num_neurons_graph	Numero di neuroni del grafo (uint16).
`SET_BASE sel 10`	n_out	Numero di id di uscita (uint16).

Zero regressioni sul Tipo #1

Con `net_type=dense` (valore di default dopo `RESET`) il percorso #1 è bit-identico a prima: `RUN_NETWORK` mantiene il payload `num_layers(1B)` e il framing degli opcode esistenti non cambia.

7.7 Occupazione (Tipo #2 abilitato)

Sintesi Yosys del sistema completo `spi_neuron_top` con Tipo #2 abilitato (`PARALLEL`=2):

Risorsa	Uso
`DP16KD` (block RAM)	2 (buffer di attivazione)
`MULT18X18D` (DSP)	4 (2 `neuron_memory` + 2 `graph_engine`)

LUT4	2619
TRELLIS_FF	2467
`$_TBUF_` (bus PSRAM)	16

Il device (108 `DP16KD`, 72 DSP, ≈44k LUT/FF) resta ampiamente sotto la saturazione: il Tipo #2 aggiunge una modalità completa a costo di risorse contenuto. LUT4/TRELLIS_FF sono cresciuti rispetto a una misura precedente (2367/2406) per via del page mode PSRAM aggiunto al controller (cap. 5, § 5.5) — sotto il 6% di utilizzo, nessun impatto pratico.

7.8 Banda del gather (misurata)

Il costo per-edge del gather è stato **isolato** costruendo due grafi identici per struttura ma con conteggio edge diverso e differenziando i cicli: la sottrazione cancella l'overhead fisso per-neurone e lascia il solo costo dell'edge.

Costo per-edge

37.53 cicli/edge con il page mode PSRAM abilitato (cap. 5, § 5.5) — **53.25 cicli/edge** senza (baseline pre-page-mode, coerente con la teoria: 4 byte/edge × ≈13 cicli/byte via PSRAM asincrona ≈52). A 80 MHz: ≈2.13 M edge/s (≈8.5 MB/s, +42% vs baseline); al clock reale di 16 MHz: ≈426 k edge/s (≈1.71 MB/s).

Il page-mode read (roadmap G7, cap. 14) è stato implementato e misurato: l'accesso sequenziale del gather ne beneficia direttamente, riducendo il costo per-edge del 29.5% (53.25→37.53 cicli/edge). Ogni edge continua comunque a pagare l'accesso byte-granulare di `int8_memory_access` (4 byte/edge); il page mode riduce il costo di ciascun byte sequenziale, non il numero di accessi.

7.9 Assemblatore host `netasm`

La configurazione leggibile della rete non richiede logica dedicata in FPGA: uno pseudo-assembly viene compilato *sull'host* (`tools/netasm/`) nei byte esatti delle tabelle e degli edge, poi caricati con `WRITE_RAM`. L'assemblatore valida a compile time (`src_id < out_id`, limiti `N_TOTAL`, padding a `PARALLEL`), complementando il guard runtime.

Listing 7.1. Esempio di pseudo-assembly (grafo)

```
NET graph
INPUTS 4                ; id 0..3
NEURON n4 relu bias=2
  CONN 0 w=5
  CONN 1 w=-3
NEURON n5 none bias=0
  CONN n4 w=2           ; riferimento simbolico all'uscita di n4
  CONN 2 w=7
OUTPUT n5
END
```

CAPITOLO 8

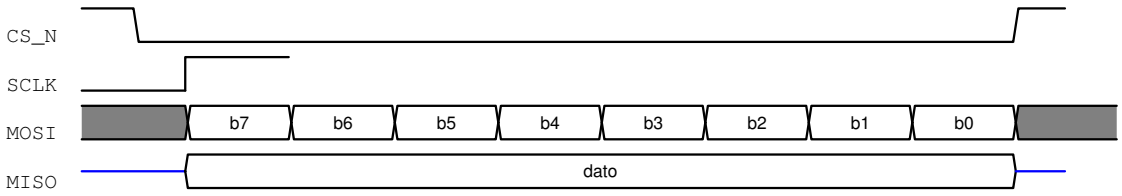
Interfaccia host SPI

8.1 Livello fisico

L'FPGA è sempre **slave** SPI. Il protocollo v1 usa SPI **Mode 0** (CPOL=0, CPHA=0), MSB-first, single-SPI. Un comando per periodo di CS basso; il byte 0 di ogni transazione è l'opcode. I campi multi-byte sono big-endian.

Campionamento Mode 0

`mosi` è campionato sul fronte di **salita** di `sclk`; `miso` è pilotato sul fronte di **discesa** (stabile prima del successivo campionamento del master). `spi_slave` sincronizza `sclk/mosi/cs_n` con un doppio flip-flop (CDC a 3 stadi) prima di ogni rilevazione di fronte.



Framing di un byte: CS scende, 8 colpi di SCLK, MSB per primo; MISO in tri-state fuori transazione.

Contratto `tx_byte_req`

`tx_byte_req` è un *prefetch hint*, non un evento "byte consumato": un consumatore deve avanzare i puntatori (indirizzo RAM, indice byte di risposta) su `rx_valid`, che pulsa esattamente una volta per byte reale trasferito.

8.2 Framing e lunghezza esplicita

La lunghezza dei trasferimenti RAM è **esplicita**, non delimitata dal fronte di CS: `WRITE_RAM`/`READ_RAM` portano un campo lunghezza a 2 byte, così il controller SPI necessita solo di un contatore di byte. Gli indirizzi di byte sono a 23 bit, trasportati in un campo di 3 byte con il bit più alto riservato a 0.

8.3 Tabella degli opcode

Op	Nome	Payload (host→FPGA)	Risposta	Funzione
`0x00`	NOP	—	—	Nessuna operazione (idle/dummy clocking).
`0x01`	WRITE_RAM	addr(3B)+len(2B)+dati	—	Scrive un blocco in PSRAM (X, pesi, bias, parametri).
`0x02`	READ_RAM	addr(3B)+len(2B)	`len` byte	Rilegge un blocco da PSRAM.

Op	Nome	Payload	Risposta	Funzione
`0x0F`	RESET	—	—	Reset sincrono del motore e azzeramento del latch STATUS; non cancella la PSRAM.
`0x10`	SET_BASE	sel(1B)+addr(3B)	—	Imposta le basi/registri (vedi §8.4).
`0x11`	SET_NET_TYP	type(1B)	—	Tipo di rete: `0x01`=dense (#1), `0x02`=graph (#2). Default dopo RESET=dense.
`0x20`	START	—	—	Avvia `neuron_memory` (percorso single-layer); ignorato se busy.
`0x21`	STATUS	—	1 byte	bit0=`busy` (live), bit1=`done` (sticky, clear-on-read), bit2=`err` (guard grafo), bit3=`flash_err` (sticky, clear-on-read), bit4=`flash_busy` (live); bit7:5=0.
`0x22`	READ_OUTPU	—	`N_NEURONS` byte	`y_bus` neuron-major (byte 0 = neurone 0); solo percorso dense (Tipo #1).
`0x23`	RUN_NETWORK	num_layers(1B)	—	Avvia l'esecuzione: dispatch su `net_type` verso `layer_sequencer` (#1) o `graph_engine` (#2); ignorato se busy.
`0x30`	READ_CONFIC	—	11 byte	Record di configurazione hardware (§8.7).
`0x40`	FLASH_READ_BLOCK	flash_addr(3B)+psram_addr(3B)+len(3B)	—	Lettura raw flash→PSRAM, bypassa il catalogo.
`0x41`	FLASH_WRITE	psram_addr(3B)+flash_a	—	Scrittura raw PSRAM→flash (erase-before-write interno + loop Page Program ≤256B + poll WIP, trasparente all'host), bypassa il catalogo.

Op	Nome	Payload	Risposta	Funzione
`0x42`	FLASH_ERASE	sector_addr(3B)	—	Erase di un settore da 4 KB (deve essere sector-aligned), bypassa il catalogo.
`0x43`	CAT_READ	—	—	Ricarica il catalogo a 16 slot (registri on-chip) dal settore riservato in flash.
`0x44`	CAT_WRITE_SLOT	slot_id(1B)+offset(3B)+len(3B)+tipo(1B)		Registra/aggiorna (offset, lunghezza, tipo) dello slot nel catalogo on-chip e lo persiste in flash; marca lo slot *non valido* finché `SAVE_SLOT` non lo conferma.
`0x45`	LOAD_SLOT	slot_id(1B)+psram_addr(	—	Flash→PSRAM per lo slot (offset/lunghezza dal catalogo), verifica CRC32 live; `STATUS.flash_err` se lo slot non è valido o il CRC non torna.
`0x46`	SAVE_SLOT	slot_id(1B)+psram_addr(3B)+len(3B)		PSRAM→flash all'offset già registrato dello slot, calcola il CRC32 live; a esito positivo aggiorna e persiste la entry di catalogo (lunghezza, CRC, valid=1).
`0x47`	CAT_INSPECT	slot_id(1B)	16 byte	Lettura sincrona di una entry di catalogo già caricata: offset[3]+len[3]+tipo[1]+valid[1]+CRC32[4]+ MSB-first.

Tutti gli opcode flash sono *fire-and-forget*: l'host fa polling su `STATUS` (bit4=`flash_busy`, bit3=`flash_err`) o sui pin `irq_n`/`data_ready_n` per l'esito, eccetto `CAT_INSPECT` che risponde in modo sincrono.

Gli 8 opcode flash (`0x40`–`0x47`) sono descritti in dettaglio, con razionale di progetto e latenze reali misurate, in §8.8 sotto.

8.4 Selettori `SET_BASE`

sel	Registro	Uso
0	`x_base`	Base ingresso X.
1	`w_base`	Base pesi.

2	`bias_addr`	Base bias.
3	`table_base`	Base tabella descrittori (multi-layer).
4	`buf_a_base`	Buffer ping-pong A.
5	`buf_b_base`	Buffer ping-pong B.
6	`activation`	Attivazione (2 bit bassi) — solo percorso single-layer.
7	`n_inputs_real`	Larghezza ingressi runtime (16-bit BE) — single-layer.
8	`n_neurons_real`	Larghezza neuroni runtime (16-bit BE) — single-layer.
9	`num_neurons_gra`	Numero neuroni del grafo (16-bit BE) — Tipo #2.
10	`n_out`	Numero id di uscita (16-bit BE) — Tipo #2.

I selettori 6–8 riguardano solo il percorso single-layer/manuale; con `RUN_NETWORK` i valori equivalenti sono letti per-layer dalla tabella descrittori.

Casi limite "reale=0" corretti (2026-09-04)

La campagna di ri-certificazione (`docs/validation/bugs.md`) ha trovato che diversi valori runtime pari a zero non erano protetti da alcun guard, con esiti che andavano da un risultato silenziosamente ignorato fino a hang o scritture PSRAM a indirizzi arbitrari. Tutti e cinque i casi seguenti sono ora no-op sicuri, verificati indipendentemente:

- `n_inputs_real=0` (selettore 7): completa in 1 ciclo con $y = \text{activation}(\text{bias})$ (BUG-003).
- `n_neurons_real=0` (selettore 8): completa senza eseguire alcun calcolo per-neurone, molto più rapido di un run a piena larghezza (BUG-004).
- `num_neurons_graph=0` (selettore 9): completa immediatamente dopo la copia degli ingressi, senza mai entrare nel loop dei descrittori (BUG-006).
- `RUN_NETWORK` con `num_layers=0` (percorso dense): no-op immediato — **prima del fix eseguiva 256 layer fasulli leggendo dati PSRAM arbitrari come descrittori** (BUG-005, CRITICO, vedi §8.9.2 sotto).
- `SET_NET_TYPE` ricevuto mentre un run è in corso: ora rifiutato silenziosamente (nessun effetto, nessun errore SPI) invece di rimappare il multiplexer dell'arbitro a metà esecuzione — **prima del fix causava un hang permanente del motore in corso** (BUG-007, CRITICO).

Dettagli, evidenza e verifica di ciascun fix in `docs/validation/bugs.md`.

8.5 `STATUS.done` sticky / clear-on-read

In `neuron_memory` il segnale `done` è un impulso di un solo ciclo. Un host che effettua polling via SPI (molto più lento del clock FPGA) mancherebbe quasi certamente un impulso grezzo di un ciclo. Il banco registri SPI latcha quindi `done` in un bit sticky sull'impulso e lo azzera quando l'host legge `STATUS` (o `RESET`). Il bit `busy` è invece mantenuto a livello per tutta la computazione e si legge live.

Race corretto (2026-09-02)

Una race reale nel meccanismo sticky (presente dalla Fase 4) è stata corretta latchando uno `status_snapshot` all'accettazione dell'opcode `STATUS` e condizionando la pulizia del bit sticky a `status_snapshot[1]` (si azzera solo se il byte effettivamente trasmesso mostrava `done=1`). Un `done` che arriva troppo tardi per uno snapshot viene riportato al polling successivo invece di essere perso.

8.6 Pin di attenzione host (`data_ready_n`, `irq_n`)

Oltre al polling di `STATUS`, il top-level espone due pin fisici attivi bassi (banco 7, cap. 12) che rispecchiano i bit sticky senza richiedere una transazione SPI, utili per pilotare un GPIO/IRQ dell'host:

- `data_ready_n` = ∼`STATUS.done` (sticky): basso quando un risultato è pronto da leggere, torna alto alla lettura di `STATUS` (clear-on-read).
- `irq_n` = ∼`STATUS.err` (guard grafo): basso quando il guard load-time di `graph_engine` è scattato. **Non** è clear-on-read: si azzera solo con `RESET` o un nuovo avvio di grafo, così un errore non passa inosservato tra un polling e l'altro.

Sono porte aggiuntive: non toccano gli opcode né i registri esistenti.

`flash_err` non ha un pin dedicato

`STATUS.flash_err` (bit3) è riportato **solo** nel byte `STATUS`, per scelta di progetto: riusare `irq_n` lo avrebbe confuso con gli errori del guard grafo (due domini di errore indipendenti sullo stesso pin), mentre un'operazione flash è sempre avviata dall'host con un opcode appena emesso, quindi il polling di `STATUS` subito dopo — già implicito nella convenzione "fire-and-forget, poi polling `STATUS`/ `data_ready_n`" — è già naturale, senza bisogno di un pin asincrono in più. `data_ready_n` invece *si azzera anche al termine di un'operazione flash*: lo specchia `STATUS.done` (bit1), che ora latcha anche sul completamento di un op flash, non solo su `RUN_NETWORK`/`START`.

8.7 READ_CONFIG

Payload fisso di **11 byte**: permette a un unico firmware host di funzionare con bitstream diversi senza ricompilare. I valori `N_INPUTS`/`N_NEURONS` riportano il *massimo* di build (il soffitto), non necessariamente la rete correntemente caricata.

Byte	Campo	Sorgente
0	`ADDR_WIDTH` (bit)	`neuron_memory.ADDR_WIDTH`
1–2	`N_INPUTS` (16-bit BE)	massimo di build
3	`N_NEURONS`	massimo di build
4	`PARALLEL`	parametro di build
5	`DATA_WIDTH` (bit)	parametro di build
6–7	versione protocollo (BE)	`0x0001`
8–9	`N_TOTAL` (16-bit BE)	massimo segnali grafo (Tipo #2)
10	flag di capacità	bit0=`GRAPH_SUPPORTED`=1

8.8 Sottosistema flash (opcode 0x40–0x47, completato 2026-09-04)

La FPGA ha accesso **esclusivo** alla flash di boot/persistenza onboard (Winbond `W25Q128JV`, 16 MB SPI NOR, cap. 12 §6/§7) tramite un SPI master dedicato e fisicamente separato (`rtl/spi_flash_master.v`), mai per accesso diretto dell'host ai pin della flash. **Non** è un filesystem: un catalogo a dimensione fissa (16 slot, `rtl/flash_slot_manager.v`) mappa `slot_id` → (offset, lunghezza, tipo, valid, CRC32) in un settore riservato della flash (settore 0) — nessuna allocazione dinamica, nessun garbage collection.

Stratificazione (ogni livello testabile a sé)

- `rtl/spi_flash_master.v` — SPI master grezzo verso il chip flash (RDID/READ/WREN/PP/SE/RDSR-1). Bus a 4 fili completamente indipendente (`sclk`/`mosi`/`miso`/`cs_n`, tutti GPIO ordinario — Fase F7, 2026-09-04): una versione precedente riusava il pad `CCLK` di boot via la primitiva ECP5 `USRMCLK` per risparmiare un pin, abbandonato perché rendeva fuorviante l'affermazione di "bus esclusivo" (elettricamente

dipendeva comunque dal motore di configurazione) e comportava un gap di verifica mai chiuso (timing di `USRMCLKTS` mai verificato contro la guida Lattice primaria).
- `rtl/flash_copy_engine.v` — motore di streaming a blocchi: flash→PSRAM (`DIR_LOAD`), PSRAM→flash con erase-before-write interno + loop Page Program ≤256B + poll WIP (`DIR_SAVE`), erase di settore standalone (`DIR_ERASE`). Master a bassa priorità (Porta D) su `rtl/mem_arbiter.v`: le operazioni flash sono su scala dei ms e non bloccano mai l'inferenza.
- `rtl/flash_slot_manager.v` — il catalogo a slot sopra, più un CRC32 (`rtl/crc32.v`, IEEE 802.3/zlib) calcolato live sul flusso di byte reale durante `LOAD_SLOT`/`SAVE_SLOT`, così uno slot corrotto o scritto a metà (es. alimentazione persa durante l'erase) è rilevato anche quando l'operazione flash sottostante ha riportato successo.

Allineamento a settore obbligatorio

`SAVE_SLOT` (e i raw `FLASH_WRITE_BLOCK`/`FLASH_ERASE`) richiedono che l'indirizzo flash target sia allineato a settore da 4 KB — rifiutato come errore altrimenti, invece di un silenzioso read-modify-erase-write parziale del settore (non esiste un buffer di scratch abbastanza grande per farlo, e ogni `SAVE_SLOT` reale scrive già uno slot intero e allineato per costruzione).

Razionale completo, ogni citazione da datasheet, ogni test avversariale (CRC non corrispondente, slot mai salvato, attraversamento di confine pagina, simulazione di perdita di alimentazione, contesa sull'arbitro) e i due bug reali trovati e corretti durante il bring-up (uno pre-esistente in `psram_controller.v`, uno nel nuovo handshake di richiesta dell'arbitro) sono in `WORKLOG.md` (voci Fasi F1-F6) e `docs/FPGA-Neural-Flash-Subsystem-Verification.md` (sunto di copertura per modulo, non ripetuto qui).

Operazione	Latenza reale misurata
ERASE (settore 4 KB)	≈400 ms (dominata dal tSE interno del chip flash, indipendente dal clock host)
SAVE (pagina 256 B, incl. erase interno)	≈403 ms (idem, tSE+tPP)
LOAD (4096 B)	1.74 ms (2.35 MB/s) @80 MHz; 8.71 ms (0.47 MB/s) @16 MHz (solo SPI-clock-bound)

Metodologia di misura completa in `docs/FPGA-Neural-Flash-Subsystem-Verification.md`.

8.9 Sequenze di sessione

8.9.1 Percorso single-layer

Listing 8.1. Sessione single-layer

```
RESET               -> 0x0F
READ_CONFIG         -> 0x30      (l'host apprende N_INPUTS/N_NEURONS/...)
WRITE_RAM (pesi)    -> 0x01 ...
WRITE_RAM (bias)    -> 0x01 ...
SET_BASE (X/W/BIAS) -> 0x10 x3
WRITE_RAM (input X) -> 0x01 ...
START               -> 0x20
poll STATUS         -> 0x21      (finche' done=1; si azzera a questa lettura)
READ_OUTPUT         -> 0x22
```

8.9.2 Percorso multi-layer (RUN_NETWORK)

Listing 8.2. Sessione multi-layer

```
WRITE_RAM (tabella descrittori)          -> 0x01 ...
WRITE_RAM (pesi/bias per layer, X layer0)-> 0x01 ...
SET_BASE (X/TABLE/BUF_A/BUF_B)           -> 0x10 x4
RUN_NETWORK(num_layers)                  -> 0x23 <num_layers>
poll STATUS                              -> 0x21   (finche' done=1)
READ_OUTPUT                              -> 0x22   (y_bus del layer finale)
```

Fuori ambito per v1

Dual SPI e CRC/checksum sui trasferimenti host (SPI assunto affidabile su traccia di scheda — da non confondere con il CRC32 del catalogo flash, §8.8, che protegge un dominio diverso: la persistenza flash↔PSRAM, non il link SPI host).

`WRITE_RAM`/`READ_RAM` senza backpressure verso l'host — rischio reale, non teorico

Ogni byte ricevuto/prodotto deve essere completamente processato da `spi_engine` prima che arrivi il successivo confine di byte scandito da SCLK — ragionevole per il bulk-loading iniziale di pesi/ingressi, non un percorso real-time. Il rischio concreto: se un host emette `WRITE_RAM`/`READ_RAM` prima che la sequenza di power-up di `psram_controller.v` sia completata (∼150 µs dopo il reset, `STATE_INIT`+`STATE_CR_INIT`), `spi_engine` si blocca in attesa che il primo accesso PSRAM completi, mentre l'host — non rallentato da alcun handshake — continua a scandire byte. I byte ricevuti durante quello stallo vengono **scartati silenziosamente**, senza errore e senza hang: solo dati sbagliati in PSRAM. Trovato durante il lavoro sul sottosistema flash (`WORKLOG.md`, Fase F5) con una riproduzione minimale solo-`WRITE_RAM`, senza alcun opcode flash coinvolto: è un rischio generale per qualunque host, non specifico agli opcode flash. **Mitigazione attuale: l'host deve attendere il power-up della PSRAM (o assicurarsi che la FPGA sia fuori reset da >150 µs) prima del suo primo `WRITE_RAM`/`READ_RAM`.** Non risolto a livello di protocollo (richiederebbe una vera backpressure, una modifica più ampia) — dichiarato qui come rischio aperto, non aggirato silenziosamente.

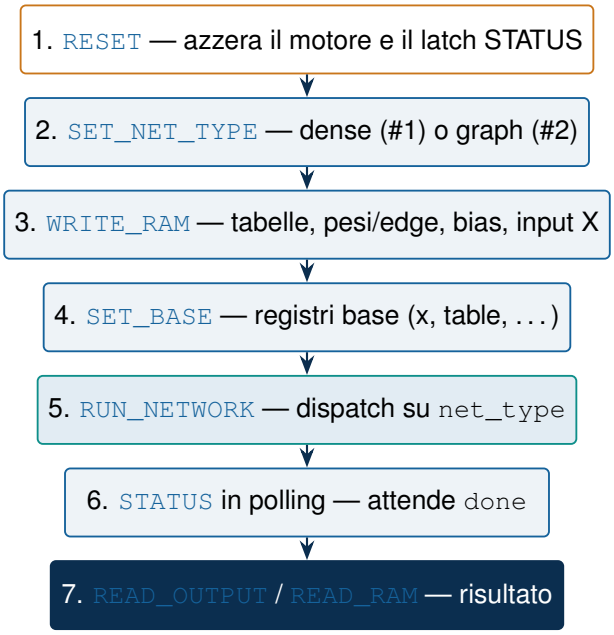
CAPITOLO 9

Programmazione della rete neurale

Questo capitolo è la guida pratica alla codifica di una rete per FPGA-Neural: come si dispone in memoria, quali registri si impostano e come si avvia, per entrambe le topologie. Presuppone gli opcode SPI (cap. 8) e i formati descrittore (cap. 6, 7).

9.1 Flusso generale

Qualunque sia il tipo, il ciclo è lo stesso: l'host *costruisce le strutture dati in RAM*, imposta i *registri base*, dichiara il *tipo di rete*, *avvia* e *rilegge* il risultato.



9.2 Registri e opcode coinvolti

Tutti i valori base si impostano con `SET_BASE` `sel(1B)+addr(3B)`. Selettori:

sel	Registro	Tipo #1	Tipo #2	Uso
0	`x_base`	✓	✓	Base input X.
3	`table_base`	✓	✓	Tabella descrittori.
4	`buf_a_base`	✓	✓*	Ping-pong A (#1) / `out_base` riuso (#2).
5	`buf_b_base`	✓	—	Ping-pong B (#1).
9	`num_neurons_graph`	—	✓	Numero neuroni del grafo.
10	`n_out`	—	✓	Numero id di uscita.

∗ *In Tipo #2 i buffer ping-pong non servono: il selettore 4 è riusato come `out_base` (regione dove copiare le uscite). I selettori 1/2/6/7/8 riguardano solo il percorso single-layer manuale (`START`), non `RUN_NETWORK`.*

Per il Tipo #1, i `w_base`/`bias_addr` *per-layer* **non** si impostano con `SET_BASE`: sono campi della tabella descrittori. `SET_NET_TYPE` default dopo `RESET` è *dense*, quindi una rete #1 funziona anche senza emetterlo.

9.3 Tipo #1 — rete densa

9.3.1 Layout in memoria

Struttura	Formato
Input X	`n_inputs_real` byte INT8 a `x_base`.
Pesi (per layer)	Neuron-major: neurone k a `w_base + k*n_inputs_real`, `n_neurons*n_inputs` byte.
Bias (per layer)	Un byte INT8 per neurone a `bias_addr`.
Tabella descrittori	`num_layers` voci da 11 byte a `table_base`.
Buffer A/B	Uscite intermedie ping-pong.

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

9.3.2 Esempio completo: rete $4 \to 4 \to 2$

Layer 0: 4 input, 4 neuroni, ReLU. Layer 1: 4 input, 2 neuroni, lineare (`PARALLEL`=2, quindi ogni `n_inputs_real` è multiplo di 2). Indirizzi scelti: `table_base`=`0x000000`, `x_base`=`0x001000`, pesi/bias L0 a `0x002000`/`0x002100`, L1 a `0x002200`/`0x002300`, buffer a `0x003000`/`0x003100`.

Listing 9.1. Tabella descrittori dense (22 byte)

```
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. Sessione SPI (dense)

```
0x0F                              RESET
0x11 01                           SET_NET_TYPE = dense
0x01 000000 0016 <22 byte tabella>   WRITE_RAM tabella
0x01 002000 0010 <16 byte pesi L0>   WRITE_RAM pesi L0 (neuron-major)
0x01 002100 0004 <4 byte bias L0>
0x01 002200 0008 <8 byte pesi L1>
0x01 002300 0002 <2 byte bias L1>
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 finche' done=1
0x22                              READ_OUTPUT -> 2 byte (layer finale)
```

9.3.3 Pseudocodice host (dense)

Listing 9.3. Codifica e caricamento di una rete densa

```
def load_dense(layers, X):          # layers in ordine di esecuzione
    spi(RESET); spi(SET_NET_TYPE, DENSE)
    table = b""
    for L in layers:                # L: pesi[n][k], bias[n], act, n_in, n_out
        assert L.n_in % PARALLEL == 0
        w = alloc(L.weights_neuron_major)   # k lento, input veloce
        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 Tipo #2 — rete a grafo

9.4.1 Layout in memoria

Struttura	Formato
Input X	`N_in` byte a `x_base`; copiati in `act_buf[0..N_in-1]` all'avvio.
Tabella descrittori	`num_neurons_graph` voci da 11 byte a `table_base`, in ordine di `out_id` crescente.
Blocchi edge	Per neurone: `n_conn` edge da 4 byte a `conn_ptr`, con padding a multiplo di `PARALLEL` (edge peso 0).
Uscite	`n_out` byte scritti a `out_base` (=selettore 4).

Descrittore graph (11 byte): `conn_ptr`(3) | `n_conn`(2) | `out_id`(2) | `activation`(1) | `bias`(1) | `reserved`(2). Edge (4 byte): `src_id`(2) | `weight`(1) | `reserved`(1). Vincolo: `src_id < out_id` (DAG feed-forward).

9.4.2 Esempio completo

4 ingressi (id 0–3). Neurone n4 (`out_id`=4, ReLU, bias=2) connesso agli id 0 e 1; neurone n5 (`out_id`=5, lineare, bias=0) connesso a n4 (id 4) e all'id 2; uscita = n5 (`n_out`=1). `PARALLEL`=2, entrambi hanno 2 connessioni (nessun padding). Indirizzi: `table_base`=`0x000000`, edge a `0x000100`, `x_base`= `0x001000`, `out_base`=`0x002000`.

Listing 9.4. Descrittori + edge grafo

```
Descrittori (a 0x000000, 22 byte):
 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

Blocchi edge (a 0x000100, 4 byte/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 = uscita di n4
                00 02 07 00   (src=2, w=+7)
```

Listing 9.5. Sessione SPI (graph)

```
0x0F                              RESET
0x11 02                           SET_NET_TYPE = graph
0x01 000000 0016 <22 byte tabella>   WRITE_RAM descrittori
0x01 000100 0010 <16 byte edge>      WRITE_RAM blocchi edge
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 (riuso sel 4)
0x10 09 000002                    SET_BASE num_neurons_graph = 2
0x10 0A 000001                    SET_BASE n_out = 1
0x23 00                           RUN_NETWORK (dispatch a graph_engine)
0x21 ...                          poll STATUS (bit2=err se src_id>=out_id)
0x02 002000 0001                  READ_RAM out_base -> 1 byte (uscita n5)
```

9.4.3 Pseudocodice host (graph)

Listing 9.6. Codifica e caricamento di un grafo

```
def load_graph(neurons, X, n_out):  # neurons ordinati per out_id crescente
    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   # regola DAG
        conn_ptr = EDGE_BASE + len(edges)
        padded = pad(N.conns, PARALLEL, fill=(0,0))   # edge peso 0
        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 ignorato in graph
    wait_status_done()
    return read_ram(OUT_BASE, n_out)
```

9.4.4 Pseudo-assembly `netasm`

La descrizione leggibile viene compilata dall'assemblatore host (`tools/netasm/`) esattamente nei byte delle tabelle e degli edge sopra. Esempio equivalente al grafo dell'esempio:

Listing 9.7. netasm: sorgente e byte generati

```
; --- sorgente ---
NET graph
INPUTS 4                ; id 0..3
NEURON n4 relu bias=2
  CONN 0 w=5
  CONN 1 w=-3
NEURON n5 none bias=0
  CONN n4 w=2           ; riferimento simbolico -> id 4
  CONN 2 w=7
OUTPUT n5
END

; --- l'assemblatore emette ---
;  id assegnati:  n4=4, n5=5   (garantito src_id < out_id)
;  descrittori:   00 01 00 00 02 00 04 01 02 00 00
;                 00 01 08 00 02 00 05 00 00 00 00
;  edge:          00 00 05 00 00 01 FD 00   (n4)
;                 00 04 02 00 00 02 07 00   (n5)
;  registri:      table_base, x_base, out_base, num_neurons=2, n_out=1
;  validato a compile-time: src_id<out_id, N_TOTAL, padding a PARALLEL
```

Perche' due livelli di codifica

Lo pseudocodice host e il `netasm` producono gli *stessi byte*. Il primo è utile quando la rete è generata a runtime (es. pesi da training); il secondo quando la topologia è scritta a mano o versionata come sorgente. In entrambi i casi l'FPGA riceve solo tabelle e dati via `WRITE_RAM`: nessun interprete a bordo.

CAPITOLO 10

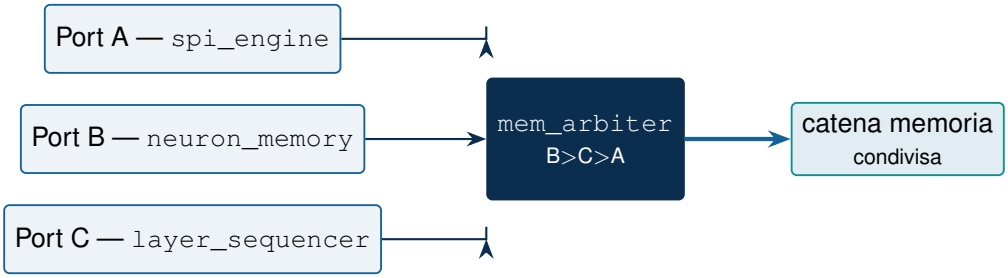
Arbitraggio e integrazione top-level

10.1 `mem_arbiter` — arbitro a tre porte

Un unico master di memoria byte-level (che alimenta la catena condivisa `int8_memory_access` → `memory_interface` → `psram_controller`) è arbitrato tra tre richiedenti:

Porta	Master	Accessi
A	`spi_engine`	`WRITE_RAM` / `READ_RAM`.
B	`neuron_memory`	Letture X/W/bias durante un'esecuzione.
C	`layer_sequencer`	Letture descrittori + scritture buffer tra layer.

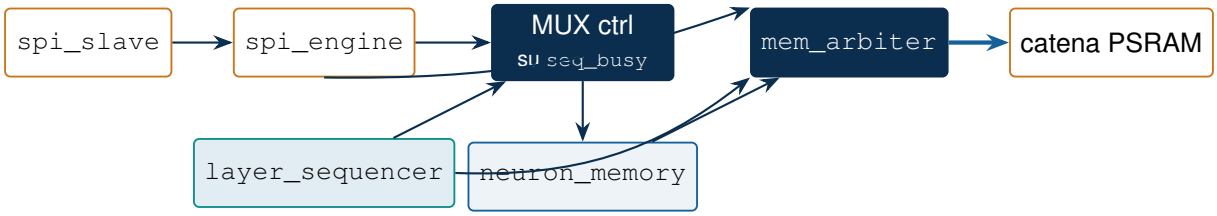
Priorità fissa **B** > **C** > **A**: un'inferenza in corso è più critica della contabilità del sequencer, che a sua volta è più critica di un accesso SPI manuale appena arrivato. In funzionamento normale B e C sono comunque temporalmente disgiunti (`neuron_memory` richiede solo durante un'esecuzione, `layer_sequencer` solo nelle pause tra layer), quindi la priorità conta soprattutto per il caso limite di un `WRITE_RAM`/`READ_RAM` manuale che arriva durante un'esecuzione multi-layer.



Concesso l'accesso, l'arbitro mantiene la proprietà fino all'impulso `m_ready` della singola transazione, poi rilascia: tutti e tre i master emettono `req` come impulso pulito di un ciclo, quindi è sufficiente un design grant-and-forward senza code.

10.2 `spi_neuron_top` — integrazione completa

Il top-level collega SPI (`spi_slave`+`spi_engine`), l'arbitro, il sequencer, `neuron_memory` e la catena PSRAM. Il reset di `neuron_memory` è l'OR del reset globale con l'impulso di soft-reset dell'opcode `RESET`, così l'host può recuperare il motore via SPI senza reset fisico (la RAM resta intatta).



Il multiplexer commuta le linee di controllo di `neuron_memory` tra il sequencer (mentre `seq_busy` è alto) e il percorso diretto di `spi_engine` (modalità single-layer legacy), restituendo il motore al percorso diretto a fine sequenza.

Verifica end-to-end

`spi_neuron_top` è verificato in simulazione con PSRAM reale (`psram_model.v`, nessun mock): RESET/READ_CONFIG/WRITE_RAM/READ_RAM/SET_BASE/ START/STATUS/-READ_OUTPUT e `RUN_NETWORK` sono esercitati puramente su SPI simulato (cap. 11).

CAPITOLO 11

Implementazione e caratterizzazione ECP5

11.1 Flusso e verifica

Il progetto è verificato su due piani complementari: **simulazione** funzionale con Icarus Verilog (algebra signed, prodotti, accumulo, gruppi, bias, ReLU, saturazione, segnali busy/done) e **implementazione** reale con Yosys (sintesi) + nextpnr-ecp5 (place&route, timing) + Project Trellis (`ecppack`).

Fase di verifica	Esito	Copre
RTL funzionale	**PASS**	correttezza del datapath
Simulazione parametrica	**PASS**	sweep di configurazioni
Sintesi ECP5	**PASS**	sintetizzabilità, mapping
Placement / Routing	**PASS**	LUT/FF/DSP, timing
Bitstream (`ecppack`)	**PASS**	flusso completo, 0 errori (P2 e P8)

Toolchain end-to-end fino al bitstream

L'intero flusso RTL → Yosys → nextpnr-ecp5 → `ecppack` produce un bitstream valido per P2 e P8, **0 errori in ogni stadio**. Header verificato byte-per-byte: `Part: LFE5U-45F-8CABGA381`, il part number reale del target, non un placeholder. Verificata la sola *generazione*: nessun test su hardware fisico in questa sessione.

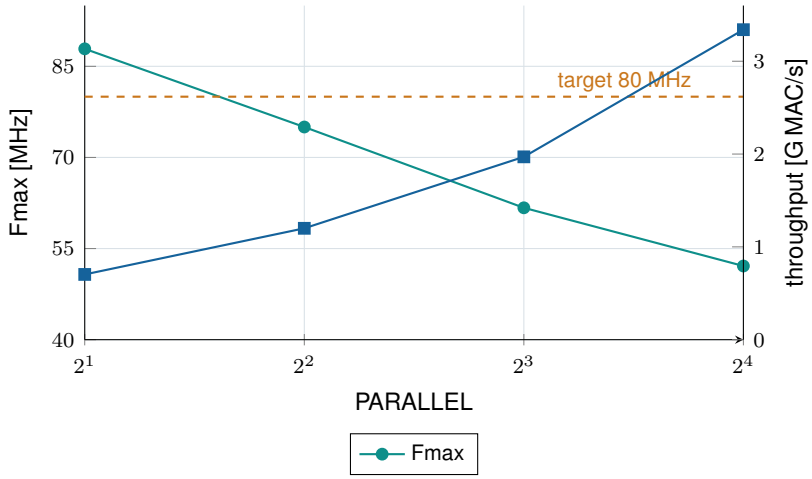
11.2 Benchmark del datapath (256×4)

Configurazione: INT8/INT32, `N_INPUTS`=256, `N_NEURONS`=4, `PARALLEL` variabile, target 80 MHz, dispositivo `LFE5U-45F-8BG381C` (−8). I bus di test sono generati *dentro* il wrapper di benchmark per non esporre migliaia di I/O; il top-level espone solo `clk/rst/start/y_bus/busy/done`.

PAR	MAC tot	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 e Tcrit in MHz e ns. MAC totali = PARALLEL×4 neuroni.

11.2.1 Fmax e throughput contro parallelismo



Trade-off fondamentale: al crescere di PARALLEL la Fmax cala (routing/albero più profondi) ma il throughput teorico sale. La linea blu (quadrati) è il throughput ≈MAC/ciclo×Fmax.

11.2.2 Interpretazione

Riducendo `PARALLEL` calano MAC simultanei, DSP, profondità dell'adder tree e congestione di routing, quindi la Fmax sale; ma aumenta il numero di gruppi e quindi la latenza. La sola frequenza non basta a scegliere: conta il throughput complessivo ≈MAC/ciclo×frequenza.

Scelte architetturali

`PARALLEL=8` è il candidato per la V1 orientata al throughput: esattamente 32 MAC simultanei con 4 neuroni, DSP al ≈44%, lasciando risorse per controller, buffer, SPI e pipeline future. `PARALLEL=2` è il riferimento orientato alla frequenza: 87.88 MHz, unico a superare il target 80 MHz, ma richiede 128 gruppi per un neurone da 256 ingressi.

11.2.3 Percorso critico e limite a 100 MHz

Il target 100 MHz non è raggiunto (miglior risultato 87.88 MHz con P2). Il limite è *temporale*, non di occupazione: con P2 l'FPGA è usato pochissimo (DSP ≈11%, LUT ≈1%). Il percorso critico attraversa FF pesi → `MULT18X18D` → prodotti → adder/carry → `acc_next` → ReLU/saturazione → FF uscita. Superare 100 MHz richiederà una o più pipeline interne, non ancora necessarie per proseguire.

11.3 Sistema integrato completo

Sintesi reale di `spi_neuron_top` (SPI + arbitro + `neuron_memory` + `graph_engine` + catena PSRAM), speed grade -8. Prima della timing closure il sistema integrato mancava il target 80 MHz (P2 ≈55 MHz, P8 ≈45 MHz), con un percorso critico interamente interno a `neuron_parallel`.

11.3.1 Causa: catena di saturazione/ReLU

L'utilizzo di risorse non è la causa (device sotto il 10% ovunque). Il percorso critico del sistema integrato è la **catena di riporto CCU2C del comparatore di saturazione/ReLU** in `neuron_parallel.v` — *non* lo SPI, l'arbitro, la PSRAM né i moduli del Tipo #2. La saturazione era scritta come confronto aritmetico (`acc > 127`, `acc < -128`), mappato dal sintetizzatore su un sottrattore a 32 bit con carry chain lunga.

11.3.2 Timing closure (2026-09-03)

Deroga esplicita al vincolo "datapath intoccabile" per un task separato di timing closure, con l'unico vincolo dell'**equivalenza bit-esatta** su tutta la regressione. Due passi:

- **Passo 1 — saturazione/ReLU come bit-test.** Un valore signed a 32 bit sta in INT8 se e solo se `acc[31:7]` sono tutti uguali: riduzione AND/OR su una fetta di bit invece di 32 bit di riporto. Semplificazione corretta e verificata bit-esatta, guadagno di logica reale ma da solo sommerso dal rumore di piazzamento.
- **Passo 2 — registro di pipeline** tra accumulo e attivazione (`+1` ciclo di latenza per neurone, assorbito dall'handshake `start`/`done`, trasparente per i chiamanti). È il passo decisivo.

Config	Prima	Dopo	Δ
P2, `.lpf` reale	54.58	**75.30**	$+38\%$
P2, sweep 5 seed	55.59	73.38–75.55	robusto
P8, unconstrained	45.47	**60.26**	$+33\%$
P8, sweep 5 seed	43.15–50.48	60.26–68.87	non sovrapposto

Fmax in MHz, place&route reale (`nextpnr-ecp5`). Guadagno robusto su 5 seed, non attribuibile a fortuna di placement.

Criterio di stop e margine reale

80 MHz non è raggiunto (75.30 MHz a P2, 94% del target) ma il guadagno è enorme e reale ($+38\%$/$+33\%$). Il passo successivo (registro di uscita del `MULT18X18D`, che toccherebbe `mac_unit.v`) è stato lasciato: gli 80 MHz sono *headroom* in vista del `.lpf` reale, non un requisito operativo. Con l'oscillatore previsto a 16 MHz, anche il numero peggiore misurato (≈45 MHz a P8) ha $2.8\times$ di margine. **Superato 2026-09-04**: dopo l'aggiunta del sottosistema flash (cap. 8 §8.8, cap. 14) la Fmax del sistema completo (P2, stesso pinout reale + 3 nuovi segnali flash) era 66.68 MHz, percorso critico ancora sulla stessa catena di accumulo di `neuron_parallel` identificata qui sopra — non un nuovo collo di bottiglia, la differenza rispetto a 75.30 MHz rumore di piazzamento/routing dovuto ai pin/logica aggiuntivi. **Aggiornato di nuovo lo stesso giorno (Fase F7)**: reso il bus SPI della flash genuinamente indipendente (rimosso il riuso del pad `CCLK` via `USRMCLK`, aggiunto un 4° pin `flash_sclk` ordinario), Fmax ri-misurata **67.91 MHz** (leggero miglioramento, percorso critico confermato ancora identico). Margine sull'oscillatore 16 MHz: $4.2\times$.

Ottimizzazione futura separata

Indipendente dalla timing closure: gli array `x_mem`/`w_mem` di `neuron_memory` sono ancora inferiti come RAM distribuita su LUT anziché su `DP16KD`. Spostarli su block RAM libererebbe LUT ed è un candidato per la Fase 7 — non era però sul percorso critico risolto qui.

CAPITOLO 12

Progetto hardware e mappa dei segnali

Stato del pinout — assegnato e verificato

Esiste ora un `.lpf` reale (`synth/ecp5/spi_neuron_top.lpf`) con i **57 segnali** del top-level assegnati a ball CABGA381 concrete, **verificato da un place&route `nextpnr-ecp5` completo a 0 errori** (non più `--lpf-allow-unconstrained`). Le ball derivano dal database di dispositivo di Project Trellis (`iodb.json`, lo stesso che usa nextpnr) e sono state validate in modo indipendente contro la §4.3.2 del datasheet Lattice ufficiale (conteggi GPIO per banco: coincidenza esatta su 6 banchi su 7, scostamento di 1 ball sul banco 3, irrilevante perché nessun segnale assegnato lo usa). `TRELLIS_IO`: 57/245 (23%). Fmax del build corrente (sistema completo incl. sottosistema flash con bus SPI indipendente, Fase F7, 2026-09-04) **67.91 MHz**, percorso critico confermato ancora sulla catena di accumulo di `neuron_parallel`, invariato rispetto alle build precedenti (cap. 11). Sunto pin-per-pin a inizio documento (pagg. 2–3). Le ball di config-SPI di boot e JTAG non compaiono qui perché sono pin dedicati a funzione fissa, senza porta RTL corrispondente: nextpnr non le richiede mai (0 errori), contano solo per lo schematic PCB.

12.1 Dispositivo target

Parametro	Valore
Dispositivo	Lattice ECP5 `LFE5U-45F-8BG381C`
Package	CABGA381 (381 ball)
Speed grade	-8 (il più veloce della famiglia ECP5)
Risorse	≈44k LUT/FF, 72×`MULT18X18D`, block RAM `DP16KD`
I/O utilizzabili	≈232 ball su 381 (resto: alimentazione/massa/NC)

12.2 Budget dei pin

Il progetto richiede circa 60 segnali su ≈232 I/O utilizzabili: ampio margine (>170 pin liberi), quindi la scheda non è pin-constrained.

Funzione	Pin
PSRAM (indirizzi 22, dati 16, controllo 6)	fino a 44
SPI applicativo (`sclk/mosi/miso/cs_n`)	4
Clock, reset	2
Pin attenzione host (`irq_n`, `data_ready_n`)	2
Bus SPI flash runtime (`flash_sclk/flash_mosi/flash_miso/flash_cs_n`, GPIO ordinario, bus indipendente — Fase F7)	4
JTAG (bring-up / debug, consigliato)	4
Totale	**≈60**

12.3 Mappa dei segnali (top-level `spi_neuron_top`) — ball reali

Assegnazione reale dei 57 segnali del top-level, verificata da place&route, **ball individuale per ogni bit** (mai un intervallo di bus). Standard I/O: LVCMOS33 (alimentazione I/O a 3.3 V). Le ball provengono dal `.lpf` reale place&route-verified. Sunto compatto della stessa tabella anche a inizio documento (pagg. 2–3).

Segnale	Dir	Ball	Banco	Funzione
Clock e reset (banco 7, lato sinistro)				
`clk`	IN	H5	7	Clock di sistema su pad `GR_PCLK7_0` (clock globale dedicato).
`rst`	IN	B4	7	Reset globale sincrono, attivo alto.
SPI applicativo (banco 7, opposto al bus PSRAM)				
`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 (pilotato sul fronte di discesa).
`cs_n`	IN	B3	7	Chip-select attivo basso.
Pin di attenzione host (banco 7, attivi bassi, di livello)				
`data_ready_n`	OUT	C3	7	Basso finché un risultato attende lettura (specchio di `STATUS.done`, clear su lettura STATUS).
`irq_n`	OUT	C4	7	Basso se il guard load-time del grafo è scattato (specchio di `STATUS.err`); si azzera solo su `RESET` o nuovo `run_start`, *non* su lettura STATUS.
Flash subsystem — SPI verso W25Q128JV onboard, bus indipendente (banco 7, Fasi F1-F7)				
`flash_sclk`	OUT	E3	7	SPI clock verso la flash — GPIO ordinario, nessuna primitiva di config coinvolta (Fase F7).
`flash_mosi`	OUT	D3	7	Master-Out Slave-In verso la flash.
`flash_miso`	IN	D5	7	Master-In Slave-Out dalla flash.
`flash_cs_n`	OUT	E4	7	Chip-select flash, attivo basso.
Bus PSRAM indirizzi `psram_a[21:0]` — 22 ball individuali (banco 2)				
`psram_a[0]`	OUT	E16	2	PSRAM A0
`psram_a[1]`	OUT	F16	2	PSRAM A1
`psram_a[2]`	OUT	D18	2	PSRAM A2
`psram_a[3]`	OUT	E17	2	PSRAM A3
`psram_a[4]`	OUT	E18	2	PSRAM A4
`psram_a[5]`	OUT	F18	2	PSRAM A5
`psram_a[6]`	OUT	F17	2	PSRAM A6
`psram_a[7]`	OUT	G16	2	PSRAM A7
`psram_a[8]`	OUT	G18	2	PSRAM A8
`psram_a[9]`	OUT	H16	2	PSRAM A9
`psram_a[10]`	OUT	H17	2	PSRAM A10
`psram_a[11]`	OUT	H18	2	PSRAM A11
`psram_a[12]`	OUT	J16	2	PSRAM A12
`psram_a[13]`	OUT	J17	2	PSRAM A13
`psram_a[14]`	OUT	C20	2	PSRAM A14

psram_a[15]	OUT	D19	2	PSRAM A15
psram_a[16]	OUT	E19	2	PSRAM A16
psram_a[17]	OUT	E20	2	PSRAM A17
psram_a[18]	OUT	F19	2	PSRAM A18
psram_a[19]	OUT	F20	2	PSRAM A19
psram_a[20]	OUT	G20	2	PSRAM A20
psram_a[21]	OUT	H20	2	PSRAM A21
psram_a[22]	OUT	P18	3	Sempre 0 (shift byte→word): NC sulla scheda.
Bus PSRAM dati psram_dq[15:0] — 16 ball individuali (banchi 2 e 3)				
psram_dq[0]	IO	K18	2	PSRAM DQ0
psram_dq[1]	IO	C18	2	PSRAM DQ1 (dual-function, usata come GPIO ordinario).
psram_dq[2]	IO	D17	2	PSRAM DQ2
psram_dq[3]	IO	D20	2	PSRAM DQ3
psram_dq[4]	IO	G19	2	PSRAM DQ4
psram_dq[5]	IO	J18	2	PSRAM DQ5
psram_dq[6]	IO	J19	2	PSRAM DQ6
psram_dq[7]	IO	J20	2	PSRAM DQ7
psram_dq[8]	IO	K19	2	PSRAM DQ8
psram_dq[9]	IO	K20	2	PSRAM DQ9
psram_dq[10]	IO	L17	3	PSRAM DQ10
psram_dq[11]	IO	M18	3	PSRAM DQ11
psram_dq[12]	IO	M17	3	PSRAM DQ12
psram_dq[13]	IO	N16	3	PSRAM DQ13
psram_dq[14]	IO	N18	3	PSRAM DQ14
psram_dq[15]	IO	P17	3	PSRAM DQ15 (bus dati bidirezionale tri-state, dq_oe = direzione).
Controllo PSRAM (banco 3)				
psram_ce_n	OUT	N17	3	Chip enable, attivo basso.
psram_oe_n	OUT	R16	3	Output enable (lettura).
psram_we_n	OUT	R17	3	Write enable (scrittura).
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 (inattivo=alto in funzionamento).

Segnali di scheda non esposti come porte RTL

Non sono porte di spi_neuron_top ma vanno previsti a livello di scheda: le linee di **SPI di configurazione** verso la flash NOR onboard (PROGRAMN/INITN/ DONE/CCLK. . . , i “Miscellaneous Dedicated Pins” del datasheet) e le 4 linee **JTAG** (TCK/TMS/TDI/TDO), l’**oscillatore** sul pad PCLK, le **alimentazioni**. I loro numeri di ball non sono nel datasheet Lattice (file separato) ma non servono qui: sono pin dedicati senza porta RTL, nextpnr non li richiede mai (0 errori), contano solo per lo schematic PCB.

SPI applicativo separato dallo SPI di configurazione

L'SPI applicativo (`sclk/mosi/miso/cs_n`) deve cadere su I/O ordinarie, **mai** sui pin dell'SPI di configurazione: il pin di clock della config-SPI non è riutilizzabile come ingresso generico dopo la configurazione senza workaround a livello di scheda. Tenerli fisicamente separati evita quel problema.

12.4 Allocazione per banchi (geometria reale del die)

La collocazione segue la geometria dei bordi del die (da `globals.json` di Trellis, ball → (col,row) → banco): i banchi **2 e 3** sono contigui lungo il bordo **destro** del chip e ospitano insieme l'intero bus PSRAM (44+1 segnali) — esattamente gli "uno o due banchi adiacenti" raccomandati. Il banco **7** (bordo **sinistro**, fisicamente opposto al bus PSRAM) ospita SPI applicativo e clock/reset, deliberatamente sul lato opposto per non far incrociare i due bus. `clk` è sul pad dedicato `H5` (`GR_PCLK7_0`). Dove un banco esauriva le ball "plain" (parte di `psram_dq`), è stata usata la ball dual-function successiva come GPIO ordinario, confermata utilizzabile dal place&route reale.

Gruppo di segnali	N. pin	Banco (reale)
Indirizzi PSRAM `psram_a[21:0]`	22	banco 2 (bordo destro)
Dati PSRAM `psram_dq[15:0]`	16	banchi 2 + 3 (adiacenti)
Controllo PSRAM (ce/oe/we/lb/ub/zz)	6	banco 3
SPI applicativo	4	banco 7 (bordo sinistro)
Pin attenzione host (`irq_n`, `data_ready_n`)	2	banco 7
Bus SPI flash indipendente (`flash_sclk/flash_mosi/flash_miso/flash_cs`	4	banco 7
Clock / reset	2	banco 7, `clk` su `GR_PCLK7_0`
Config SPI boot / JTAG	—	pin dedicati (fuori RTL, solo PCB)

12.5 Sottosistema PSRAM

Il controller `psram_controller.v` implementa un'interfaccia **parallela asincrona** (bus indirizzi, dati 16-bit, `ce_n/oe_n/we_n` e byte-lane `lb_n/ub_n`, più `zz_n`) con latenza di accesso **70 ns** cablata come $\lceil 70\,\text{ns} \times f_{clk} \rceil$. È un bus in stile SRAM asincrona, non QSPI.

Ruolo	Componente
Memoria di lavoro	ISSI `IS66WVE4M16EBLL-70BLI` — PSRAM parallela 64 Mbit (4M×16, 8 MB), async, 70 ns, corrispondente esatto alla temporizzazione del controller.
Fallback	ISSI `IS61WV6416DBLL` / `IS61WV102416BLL` (SRAM async vera, drop-in sugli stessi segnali, `zz_n` inattivo, ∼10 ns, densità minore).
Storage persistente	Winbond `W25Q128JV` — flash NOR SPI 16 MB per bitstream, pesi, bias, metadati di rete.

12.5.1 Collegamento PSRAM (esclusivo della FPGA)

La PSRAM è pilotata **esclusivamente dalla FPGA** tramite `psram_controller.v`: nessun master esterno accede al bus. L'host esterno (RPi/ESP32/MCU) parla solo SPI con la FPGA e non tocca

mai queste linee. Collegamento pin-per-pin FPGA ↔ ISSI `IS66WVE4M16EBLL-70BLI`:

Segnale FPGA	Pin PSRAM	Funzione
`psram_a[21:0]`	A0–A21	Bus indirizzi (22 linee, 8 MB word address).
`psram_dq[15:0]`	DQ0–DQ15	Bus dati bidirezionale (tri-state, `dq_oe`=direzione).
`psram_ce_n`	CE#	Chip enable (attivo basso).
`psram_oe_n`	OE#	Output enable (lettura).
`psram_we_n`	WE#	Write enable (scrittura).
`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 (tenuto alto in funzionamento).

Alimentazione PSRAM: **3.3 V** (variante BLL), sullo stesso rail I/O dei banchi 2/3 a cui è cablata (cap. 12, ball reali). Disaccoppiamento per pin di alimentazione secondo il datasheet ISSI.

12.6 Clock

Non esiste ancora alcun PLL nell'RTL: `CLK_FREQ_MHZ` è un *parametro di temporizzazione* (alimenta le formule di accesso PSRAM), non un generatore di clock. L'oscillatore montato pilota `clk` direttamente. Raccomandazione: oscillatore MEMS 16 MHz (famiglia SiTime SiT2001B), ben al di sotto dei 67.91 MHz di Fmax del sistema integrato completo (incl. sottosistema flash, cap. 11). `CLK_FREQ_MHZ` deve essere impostato al valore reale dell'oscillatore montato, altrimenti la temporizzazione PSRAM risulta errata.

12.7 Alimentazione

Albero a **tre rail** (la sezione SERDES dell'eval board Lattice non serve e va omessa: niente `VCCA`/`VCCHTX` a 1.2 V):

Rail	Tensione	Alimenta / regolatore
`VCC` (core)	1.1 V	Core logico FPGA. Buck `TLV62568`, ≥600 mA.
`VCCIO0/2/3/6/7`	3.3 V	I/O di tutti i banchi usati + PSRAM. Buck `TLV62568`, 1 A.
`VCCAUX`	2.5 V	Ausiliario FPGA. LDO `TLV73325`, 10 mA.

Disaccoppiamento: almeno un condensatore per pin di alimentazione + bulk per rail, secondo la checklist hardware ECP5 Lattice. Ingresso: 12 V esterno (o adatta i buck alla sorgente).

12.8 Configurazione e programmazione

La "scrittura della mappa" dell'FPGA (bitstream) avviene tramite pin dedicati del silicio, **non** porte del top-level RTL. Modo di default: **MSPI** — boot automatico dalla flash NOR all'accensione (prodotto standalone); JTAG disponibile per lo sviluppo.

12.8.1 JTAG (sviluppo / debug)

Segnale	Ball[†]	Funzione
`TCK`	T5	Test clock.
`TDI`	R5	Test data in.

TDO	V4	Test data out.
TMS	U5	Test mode select.

12.8.2 Config-SPI verso boot flash

La FPGA carica il bitstream dalla **Winbond W25Q128JV** (128 Mbit SPI NOR, Quad read) all'accensione. Il sottosistema flash (rtl/flash_slot_manager.v, Fasi F1-F7, cap. 11) usa la **stessa flash fisica** per pesi/bias/metadati di rete a runtime, accesso esclusivo della FPGA: dopo la configurazione, la FPGA riprende il controllo del chip via un bus SPI a 4 fili completamente indipendente, flash_sclk/flash_mosi/flash_miso/flash_cs_n (tutti GPIO ordinario, pagg. 2–3 e §"Mappa dei segnali" — nessuna primitiva di configurazione ECP5 coinvolta, Fase F7) — implica comunque un doppio collegamento a livello di scheda (DI/DO/CS/CLK della flash cablati sia ai pin dedicati di boot sotto sia a queste 4 ball ordinarie, poiché è lo stesso chip fisico a svolgere entrambi i ruoli), non ancora riportato in uno schematico (nessuno esiste ancora, vedi checklist sotto).

Segnale	Ball†	Funzione
CCLK/MCLK/SCK	U3	Clock di configurazione.
DQ0_MOSI	W2	Dato config (MOSI).
DQ1_MISO	V2	Dato config (MISO).
BUSY_CSSPIN	R2	Chip-select flash.
DQ2 / DQ3	Y2 / W1	Linee per Quad read.
PROGRAMN	W3	Avvia riconfigurazione (pulsante, attivo basso).
INITN	V3	Init / errore di configurazione (LED).
DONE	Y3	Configurazione completata (LED).
CFGMDN[2:0]	R4/T4/U4	Selezione modo (vedi sotto).

12.8.3 Modi di configurazione (CFGMDN)

Modo	CFGMDN[2:0]	Uso
MSPI (boot da flash)	010	**Default** — standalone.
SSPI (slave SPI)	001	Config da host esterno.
SCM (slave serial)	101	Config seriale.
SPCM (slave parallel)	111	Config parallela 8-bit.

Ball di configurazione da verificare sul 45F

†Le ball di JTAG e config-SPI qui riportate sono il *riferimento* dell'eval board Lattice (device 85F). JTAG e config-SPI sono pin dedicati e in gran parte fissi nella famiglia ECP5, ma le posizioni esatte sul target LFE5U-45F-8BG381C vanno confermate sul file pinout Lattice del 45F (Diamond/Radiant o database Trellis) prima di committarle nello schematico, come già fatto per i segnali applicativi (cap. 12).

12.9 Attività aperte prima della cattura schematica

OK ADDR_WIDTH=23 (8 MB pieni) su tutti i moduli e testbench.
OK .lpf reale con l'assegnazione ball CABGA381, place&route-verified a 0 errori (synth/ecp5/spi_neuron_top.lpf, 57 segnali incl. sottosistema flash).

OK Sottosistema flash boot/persistenza (Fasi F1-F7): SPI master, copy engine, catalogo a slot con CRC32, bus SPI a 4 fili indipendente (nessuna primitiva di configurazione condivisa), sintesi reale a 0 errori, Fmax 67.91 MHz (`WORKLOG.md`).

☐ Confermare signal integrity PSRAM/SPI al clock effettivamente montato.

☐ Schema di doppio collegamento DI/DO/CS/CLK della flash (pin dedicati di boot + le 4 ball ordinarie del sottosistema flash) — non ancora catturato a schematico.

☐ Scelta del footprint del connettore JTAG.

☐ Cattura schematica (KiCad o altro): nessuno schema esiste ancora per questa combinazione dispositivo/package.

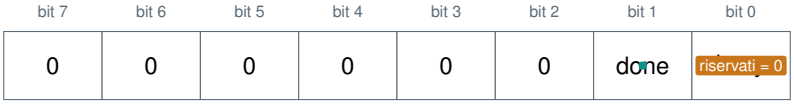
CAPITOLO 13

Riferimento rapido

13.1 Opcode SPI

Valore	Nome	Risposta	Sintesi
0x00	NOP	—	idle
0x01	WRITE_RAM	—	scrittura blocco PSRAM
0x02	READ_RAM	len B	lettura blocco PSRAM
0x0F	RESET	—	reset motore + latch STATUS
0x10	SET_BASE	—	imposta base/registro (sel 0..10)
0x11	SET_NET_TYPE	—	tipo rete #1/#2
0x20	START	—	avvio single-layer
0x21	STATUS	1 B	busy(live)/done(sticky)
0x22	READ_OUTPUT	N_NEURONS B	y_bus
0x23	RUN_NETWORK	—	avvio multi-layer
0x30	READ_CONFIG	11 B	record configurazione

13.2 Byte di STATUS



done è sticky, clear-on-read; busy è live; bit2=err (guard grafo).

13.3 Selettori SET_BASE

- 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 (Tipo #2)
- 10 — n_out (Tipo #2)

13.4 Tabella descrittori (11 byte/layer, MSB-first)

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

13.5 Parametri di build

- DATA_WIDTH — 8 (INT8)
- ACC_WIDTH — 32 (INT32)
- N_INPUTS — max ingressi
- N_NEURONS — max neuroni
- PARALLEL — MAC simultanei

- `N_LAYERS` — max layer
- `ADDR_WIDTH` — 23 (8 MB)
- `MEM_DATA_WIDTH` — 16
- `CLK_FREQ_MHZ` — timing PSRAM

CAPITOLO 14

Roadmap e stato di sviluppo

14.1 Fasi di sviluppo

Fase	Titolo	Stato	Contenuto
1	Layer parametrico	**OK**	ingressi/neuroni/parallelismo, accumulo, bias, ReLU; test 32×4/P=8.
2	Parameter sweep	**OK**	configurazioni multiple incl. non-multiple e degeneri; guard di elaborazione aggiunto.
3	Architettura di memoria	**OK**	`neuron_memory` mono/multi-neurone, PSRAM reale testata; buffer multi-layer → Fase 5.
4	Interfaccia SPI	**OK**	`spi_slave`+`spi_engine`, 17 opcode incl. sottosistema flash, Fmax controllata a livello di sistema completo.
5	Rete multi-layer	**OK**†	`layer_sequencer`, attivazioni configurabili, larghezza runtime; toolchain reale controllata.
6	Software host	pianificata	driver Linux ed ESP32 sullo stesso protocollo.
7	Ottimizzazione	in corso	timing closure fatta (55→75 MHz); page-mode PSRAM fatto (banda gather +42%); resta block RAM per x/w.
8	Training hardware (opz.)	futura	backprop, gradienti, aggiornamento pesi.
9	Sottosistema flash (F1-F7)	**OK**	SPI master dedicato, copy engine flash↔PSRAM, catalogo a 16 slot con CRC32, bus SPI a 4 fili indipendente (F7), 8 opcode (`0x40`–`0x47`, cap. 8 §8.8); sintesi reale 0 errori.

*† RTL, unit test ed end-to-end su SPI simulato completi; timing closure eseguita: 75.30 MHz (P2) / 60.26 MHz (P8) al tempo della Fase 5, bit-esatta su tutta la regressione; Fmax del sistema completo dopo Fase 9 (incl. sottosistema flash indipendente): **67.91 MHz** (cap. 11).*

14.2 Stato dei componenti

Componente	Stato
Layer neurale parametrico	**OK** funzionante
Ingressi/neuroni/parallelismo parametrici	**OK**
Accumulo, bias, ReLU	**OK**
Validazione 32×4 / P=8	**OK**

RAM dedicata (interfaccia + controller + accesso INT8)	**OK** testata su PSRAM reale
Interfaccia SPI (17 opcode incl. RUN_NETWORK + flash)	**OK** Fmax a livello di sistema completo
Dual SPI	futura
Motore multi-layer	**OK** timing closure 75.30 MHz (P2) al tempo della Fase 5
Attivazioni configurabili (ACT_NONE/ACT_RELU)	**OK**
Larghezza rete runtime (un bitstream, ogni topologia)	**OK** risparmio misurato
Rete a grafo Tipo #2 (act_buffer, graph_engine, netasm)	**OK** RTL + test + sintesi
Pinout CABGA381 (`.lpf` reale, 57 segnali incl. flash)	**OK** place&route-verified 0 errori
Page-mode PSRAM (G7)	**OK** fatto (37.53 cicli/edge, banda +42%)
Sottosistema flash (SPI master, copy engine, catalogo CRC32, bus indipendente F7)	**OK** sintesi reale 0 errori, Fmax 67.91 MHz
Bitstream reale (`ecppack`, P2/P8)	**OK** 0 errori, part LFE5U-45F-8CABGA381
Driver host Linux / ESP32	pianificato
Training hardware	futuro

14.3 Principio architetturale (sintesi)

Fondamento del progetto

L'FPGA implementa la macchina neurale e possiede la propria RAM; l'host configura e usa la macchina. Una build fissa il *soffitto* (max layer, max larghezza, PARALLEL); l'host configura la rete *reale* — numero di layer, larghezza per-layer, attivazione per-layer, parametri addestrati — interamente a runtime, via SPI, nella memoria locale dell'FPGA. Un solo bitstream serve qualunque topologia fino a quel soffitto.

14.4 Visione a lungo termine

L'obiettivo finale è un blocco hardware riusabile integrabile in progetti futuri diversi: la piattaforma host può cambiare (Linux, ESP32, MCU, PC) senza cambiare l'architettura fondamentale dell'engine. L'FPGA diventa una periferica di computazione neurale dedicata, ottimizzata per la topologia richiesta da ciascuna applicazione.

CAPITOLO A

Moduli, porte e toolchain

A.1 Elenco dei moduli RTL

File	Tipo	Ruolo
`rtl/mac_unit.v`	combinatorio	prodotto-accumulatore singolo
`rtl/mac8.v`	combinatorio	MAC parallelo + adder tree bilanciato
`rtl/neuron_parallel.v`	FSM	neurone: gruppi, bias, attivazione, saturazione
`rtl/layer.v`	strutturale	N_NEURONS neuroni in parallelo
`rtl/neuron_memory.v`	FSM	ponte memoria/neurone, loop neuroni
`rtl/layer_sequer`	FSM	sequenza multi-layer, ping-pong
`rtl/int8_memory_acces.v`	FSM	conversione byte ↔ word
`rtl/memory_inter`	FSM	handshake req/ready
`rtl/psram_controller.v`	FSM	bus fisico PSRAM async, page mode 70/20 ns
`rtl/mem_arbiter.`	arbitro	3 porte, priorità B>C>A
`rtl/spi_slave.v`	FSM	layer fisico SPI Mode 0 + CDC
`rtl/spi_engine.v`	FSM	opcode + banco registri
`rtl/act_buffer.v`	block RAM	buffer di attivazione DP16KD (Tipo #2)
`rtl/graph_engine`	FSM	motore rete a grafo (Tipo #2)
`rtl/spi_neuron_top.v`	top	integrazione completa
`rtl/memory_model`	modello	RAM comportamentale (sim)

A.2 Porte del top-level `spi_neuron_top`

Vedere la tabella segnale-per-segnale completa nel cap. 12. In sintesi: clock e reset (`clk`, `rst`); SPI applicativo (`sclk`, `mosi`, `miso`, `cs_n`); bus PSRAM (`psram_a[22:0]`, `psram_dq[15:0]`, `psram_ce_n/oe_n/we_n/lb_n/ub_n/zz_n`).

A.3 Toolchain

Strumento	Versione	Uso
Yosys	0.68+post	sintesi RTL → netlist JSON, mapping ECP5
nextpnr-ecp5	0.11.1-19-g8dbcee5	placement, routing, timing
Project Trellis	install	`ecppack/ecppll/ecpbram`
Icarus Verilog	`-g2012`	simulazione funzionale

A.3.1 Parametri nextpnr principali

```
--45k                          seleziona LFE5U-45F
--package CABGA381             package
--speed 8                      speed grade -8
--json <netlist>               netlist da Yosys
--lpf <vincoli>                vincoli di pin (attualmente vuoti)
--lpf-allow-unconstrained      permette I/O non vincolate (benchmark)
--freq 80                      timing target 80 MHz
```

A.3.2 Esempio di simulazione

```
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 Testbench principali

Testbench	Copertura
`parametric_tb.v`	datapath 256×4, casi accumulo/bias/ReLU/saturazione
`parameter_sweep_tb.v`	sweep configurazioni valide
`neuron_parallel_tb.v`	attivazioni, larghezza runtime (T7)
`neuron_memory_tb.v` / `_multi_tb.v`	integrazione memoria mono/multi-neurone, PSRAM reale (T5)
`psram_controller_tb.v`	controller PSRAM
`psram_page_mode_tb.v`	burst di pagina, attraversamento pagina, chiusura su WRITE/timeout t_{CEM}, cambi di byte-enable (§ 5.5)
`spi_slave_tb.v`	layer fisico SPI (4 test)
`spi_engine_tb.v`	opcode, registri (10+ test)
`spi_neuron_top_tb.v`	end-to-end, PSRAM reale su SPI simulato
`spi_neuron_top_runnetwor`	RUN_NETWORK 2 layer end-to-end
`layer_sequencer_tb.v`	sequenza 2 layer, ping-pong, copia byte-exact

Questo datasheet è generato a partire dal codice RTL, dalla documentazione e dai benchmark presenti nella repository `github.com/manvalan/FPGA-Neural` allo stato del Settembre 2026. I valori di Fmax, utilizzo risorse e throughput sono quelli riportati nelle misure della repository (`.lpf` reale già assegnato e verificato da place&route, cap. 12) e vanno riverificati ad ogni variazione sostanziale dell'RTL o della chiusura del timing di Fase 7, tuttora in corso (cap. 14).