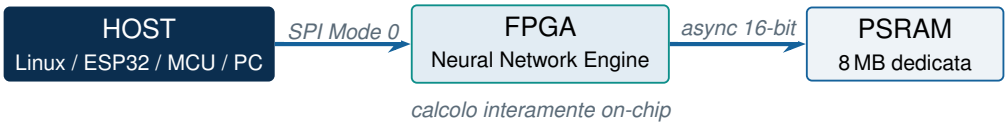
FPGA – Neural

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** ($O(\log_2 \text{PARALLEL})$) 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 (70 ns), 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.
- 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

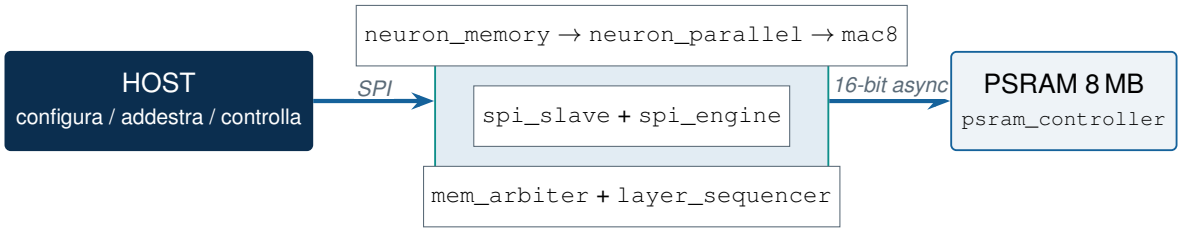
- FPGA: Lattice ECP5 `LFE5U-45F-8BG381C` (-8, CABGA381).
- Sintesi: Yosys; place&route: nextpnr-ecp5; bitstream: Project Trellis (`ecppack`).
- Simulazione: Icarus Verilog (`-g2012`).
- PSRAM: ISSI `IS66WVE4M16EBLL-70BLI` (64 Mb, 4M×16).

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)	75.30 MHz	post timing closure, 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

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.

Indice

1 Panoramica del sistema **1**
- 1.1 Obiettivo del progetto . . . 1
- 1.2 Configurazione hardware contro configurazione di rete . . . 1
- 1.3 Boot e inizializzazione . . . 2
- 1.4 Addestramento e inferenza . . . 2
- 1.5 Filosofia di progetto e riuso . . . 2

2 Architettura RTL **3**
- 2.1 Organizzazione gerarchica . . . 3
- 2.2 Ruolo di ciascun modulo . . . 3
- 2.3 Due percorsi di esecuzione . . . 4

3 Datapath di calcolo **6**
- 3.1 Catena aritmetica INT8/INT32 . . . 6
- 3.2 `mac_unit` — moltiplicatore-accumulatore . . . 6
- 3.3 `mac8` — MAC parallelo e balanced adder tree . . . 6
- 3.4 `neuron_parallel` — FSM del neurone . . . 7
  - 3.4.1 Guard di parametri (elaboration-time) . . . 7
- 3.5 Funzioni di attivazione . . . 8
- 3.6 Saturazione INT8 . . . 8
- 3.7 `layer` — neuroni in parallelo . . . 8

4 Parametri e configurabilità **9**
- 4.1 Parametri di build (synthesis-time) . . . 9
- 4.2 Larghezza di rete a runtime . . . 9
  - 4.2.1 Risparmio misurato . . . 10
- 4.3 Configurazioni caratterizzate . . . 10
- 4.4 Riepilogo build contro runtime . . . 10

5 Sottosistema di memoria **11**
- 5.1 Catena di memoria . . . 11
- 5.2 `int8_memory_access` — conversione byte/word . . . 11
- 5.3 `memory_interface` — handshake . . . 11
- 5.4 `psram_controller` — bus fisico . . . 11
  - 5.4.1 Temporizzazione . . . 12
- 5.5 Mappa degli indirizzi e convenzioni . . . 12
  - 5.5.1 Indirizzamento fisico della PSRAM . . . 12
- 5.6 Larghezza di banda . . . 12

6 Memoria, multi-neurone e multi-layer **13**
- 6.1 `neuron_memory` — ponte memoria/neurone . . . 13
- 6.2 `layer_sequencer` — rete multi-layer . . . 13
  - 6.2.1 Buffer ping-pong . . . 14
  - 6.2.2 Tabella descrittori . . . 14
- 6.3 Gerarchia dei segnali `busy`/`done` . . . 14

7 Rete a grafo (Tipo #2) 15
7.1 Due tipi di rete . . . 15
7.2 Buffer di attivazione globale . . . 15
7.3 DAG feed-forward e vincolo `src_id < out_id` . . . 15
7.4 Formati dati . . . 15
7.4.1 Descrittore Tipo #2 (grafo) . . . 16
7.4.2 Edge del grafo (4 byte, allineato) . . . 16
7.5 `graph_engine` — motore del grafo . . . 16
7.6 Opcode e registri del Tipo #2 . . . 17
7.7 Occupazione (Tipo #2 abilitato) . . . 17
7.8 Banda del gather (misurata) . . . 18
7.9 Compilatore host . . . 18

8 Interfaccia host SPI 19
8.1 Livello fisico . . . 19
8.2 Framing e lunghezza esplicita . . . 19
8.3 Tabella degli opcode . . . 19
8.4 Selettori `SET_BASE` . . . 20
8.5 `STATUS.done` sticky / clear-on-read . . . 21
8.6 Pin di attenzione host (`data_ready_n`, `irq_n`) . . . 21
8.7 `READ_CONFIG` . . . 21
8.8 Sequenze di sessione . . . 21
8.8.1 Percorso single-layer . . . 21
8.8.2 Percorso multi-layer (RUN_NETWORK) . . . 22

9 Programmazione della rete 23
9.1 Flusso generale . . . 23
9.2 Registri e opcode coinvolti . . . 23
9.3 Tipo #1 — rete densa . . . 24
9.3.1 Layout in memoria . . . 24
9.3.2 Esempio completo: rete $4 \to 4 \to 2$ . . . 24
9.3.3 Pseudocodice host (dense) . . . 24
9.4 Tipo #2 — rete a grafo . . . 25
9.4.1 Layout in memoria . . . 25
9.4.2 Esempio completo . . . 25
9.4.3 Pseudocodice host (graph) . . . 25
9.4.4 Sorgente YAML . . . 26

10 Progettazione della rete con YAML 27
10.1 Elementi comuni . . . 27
10.2 Tipo #1 — rete dense . . . 27
10.3 Tipo #2 — rete a grafo . . . 27
10.4 Pesi: inline o file esterno . . . 28
10.5 Regole di validazione (compile-time) . . . 28
10.6 Casi d'uso applicativi . . . 28
10.6.1 Classificazione di volti (dense) . . . 28
10.6.2 Manutenzione predittiva / anomalie (dense) . . . 29
10.6.3 Parole chiave / gesti (dense) . . . 29
10.6.4 Controllore a riflessi / rete sparsa (graph) . . . 29

11 Arbitraggio e top-level 31
11.1 `mem_arbiter` — arbitro a tre porte . . . 31
11.2 `spi_neuron_top` — integrazione completa . . . 31

12 Implementazione ECP5 **33**
12.1 Flusso e verifica . . . 33
12.2 Benchmark del datapath (256×4) . . . 33
12.2.1 Fmax e throughput contro parallelismo . . . 34
12.2.2 Interpretazione . . . 34
12.2.3 Percorso critico e limite a 100 MHz . . . 34
12.3 Sistema integrato completo . . . 34
12.3.1 Causa: catena di saturazione/ReLU . . . 34
12.3.2 Timing closure (2026-09-03) . . . 35

13 Progetto hardware e pinout **36**
13.1 Dispositivo target . . . 36
13.2 Budget dei pin . . . 36
13.3 Mappa dei segnali (top-level `spi_neuron_top`) — ball reali . . . 36
13.4 Allocazione per banchi (geometria reale del die) . . . 38
13.5 Sottosistema PSRAM . . . 38
13.5.1 Collegamento PSRAM (esclusivo della FPGA) . . . 38
13.6 Clock . . . 39
13.7 Alimentazione . . . 39
13.8 Configurazione e programmazione . . . 39
13.8.1 JTAG (sviluppo / debug) . . . 39
13.8.2 Config-SPI verso boot flash . . . 39
13.8.3 Modi di configurazione (`CFGMDN`) . . . 40
13.9 Attività aperte prima della cattura schematica . . . 40

14 Riferimento rapido **41**
14.1 Opcode SPI . . . 41
14.2 Byte di STATUS . . . 41
14.3 Selettori SET_BASE . . . 41
14.4 Tabella descrittori (11 byte/layer, MSB-first) . . . 41
14.5 Parametri di build . . . 41

15 Roadmap e stato di sviluppo **42**
15.1 Fasi di sviluppo . . . 42
15.2 Stato dei componenti . . . 42
15.3 Principio architetturale (sintesi) . . . 43
15.4 Visione a lungo termine . . . 43

A Moduli e toolchain **44**
A.1 Elenco dei moduli RTL . . . 44
A.2 Porte del top-level `spi_neuron_top` . . . 44
A.3 Toolchain . . . 44
A.3.1 Parametri nextpnr principali . . . 44
A.3.2 Esempio di simulazione . . . 45
A.4 Testbench principali . . . 45

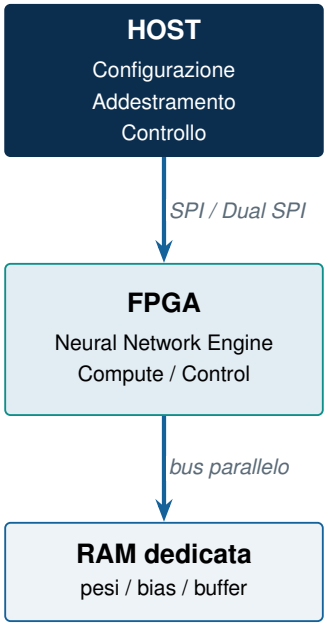
CAPITOLO 1

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.

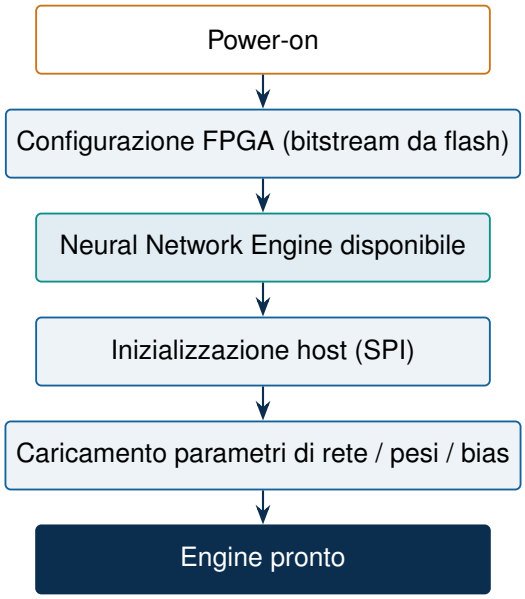
Principio architetturale centrale

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. 15). 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

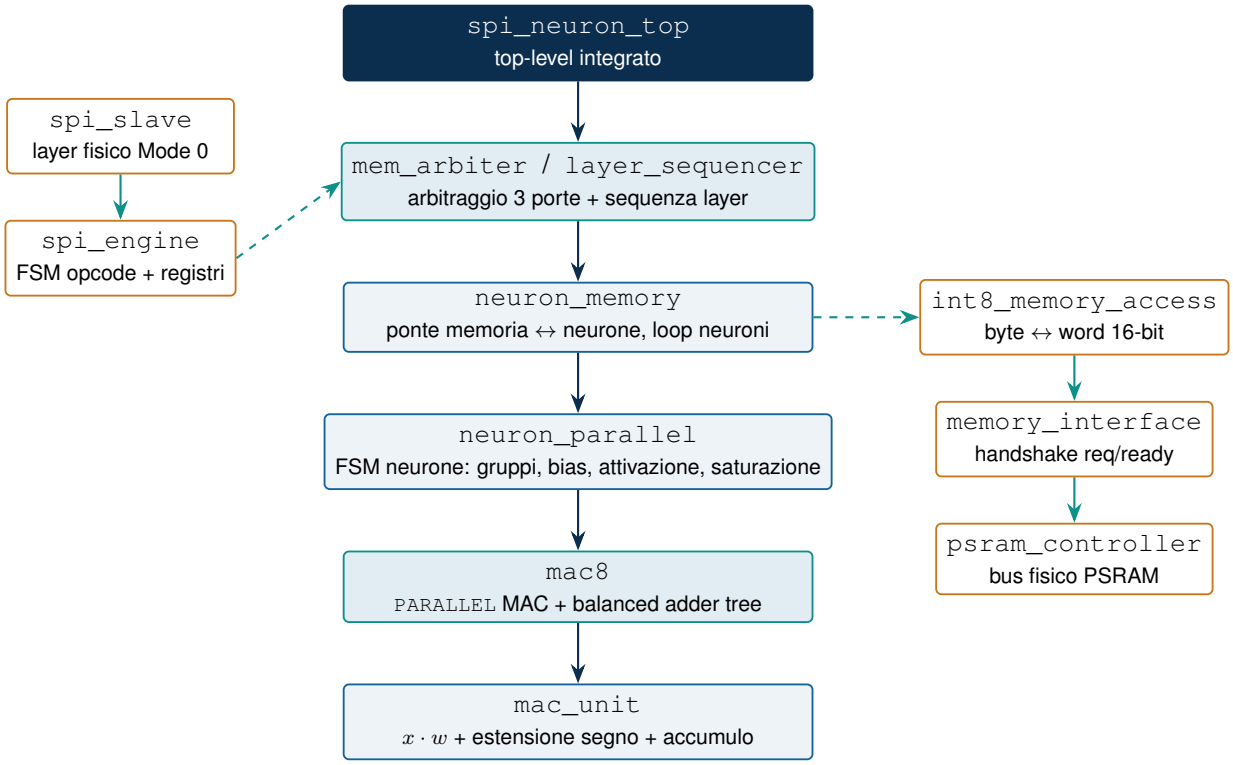
Il progetto va inteso come una *piattaforma di accelerazione neurale FPGA riusabile* più che come una singola rete. L'applicazione determina dimensione degli ingressi, topologia, numero di layer e neuroni, parallelismo, precisione numerica, funzioni di attivazione, requisiti di memoria e prestazioni; 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: sequenza INIT/IDLE/READ/WRITE con temporizzazione a 70 ns derivata da `CLK_FREQ_MHZ`; 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:

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

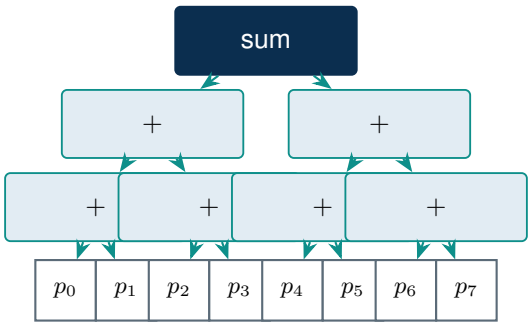
Il prodotto ha larghezza $2\times$`DATA_WIDTH` e viene esteso con segno replicando il bit più significativo. Su ECP5 la moltiplicazione mappa su un blocco DSP `MULT18X18D`.

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.



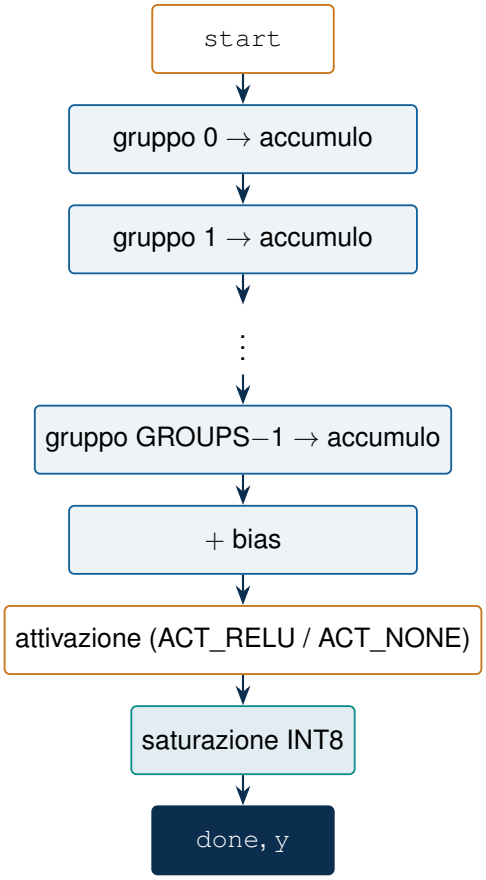
Esempio con PARALLEL=8: 3 livelli. PARALLEL=16 → 4 livelli; PARALLEL=32 → 5 livelli.

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

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. 12), 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:

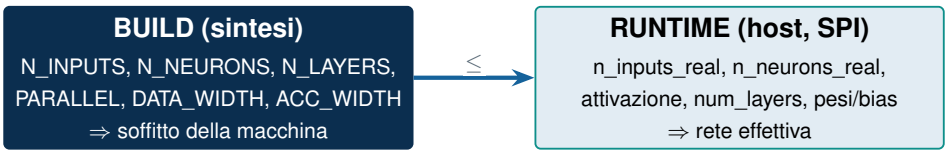
Test	Cicli	Confronto
`neuron_parallel_tb.v` (T7)	3 vs 6	ridotto vs pieno, con dati "spazzatura" nelle corsie saltate (prova che non vengono lette).
`neuron_memory_tb.v` (T5)	209 vs 788	8-di-32 vs 32 pieni, attraverso lo stack PSRAM reale.

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

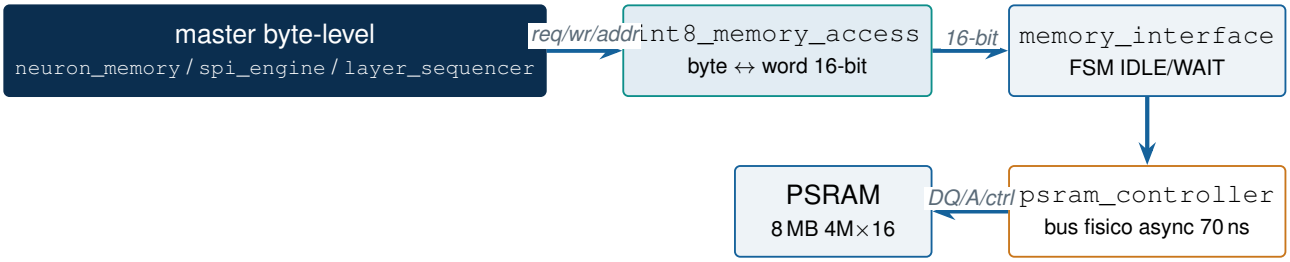


CAPITOLO 5

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].

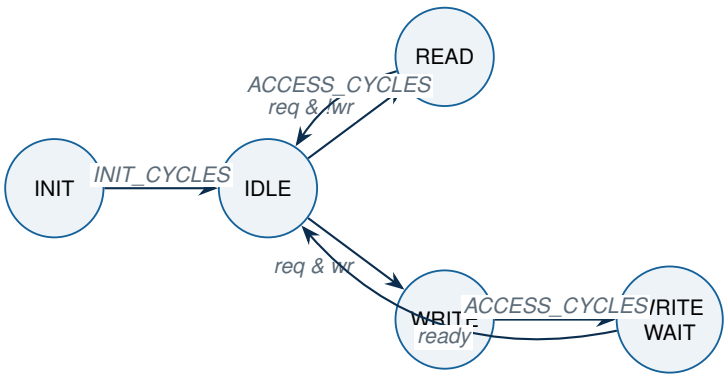
In lettura estrae il byte corretto da `mem_rdata`. La FSM ha due stati (IDLE, WAIT) e restituisce `ready` come impulso di un ciclo.

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. La macchina a stati è:



5.4.1 Temporizzazione

Formule di temporizzazione

ACCESS_CYCLES $= \lceil (70 \times \text{CLK\_FREQ\_MHZ})/1000 \rceil$ (latenza di accesso 70 ns)
INIT_CYCLES $= 150 \times$ CLK_FREQ_MHZ (inizializzazione di power-up)

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. 13 per la parte raccomandata (ISSI parallela).

5.5 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.

5.5.1 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.

5.6 Larghezza di banda

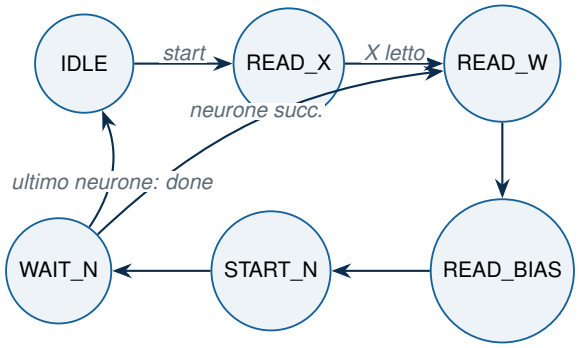
L'analisi di banda contro la temporizzazione reale della PSRAM è rinviata alla Fase 7. Il modello attuale è memory-bound per costruzione: `neuron_memory` legge X una volta e rilegge W/bias per ciascun neurone (cap. 6), un neurone per volta, senza modificare il datapath di calcolo validato.

CAPITOLO 6

Integrazione memoria, multi-neurone e multi-layer

6.1 `neuron_memory` — ponte memoria/neurone

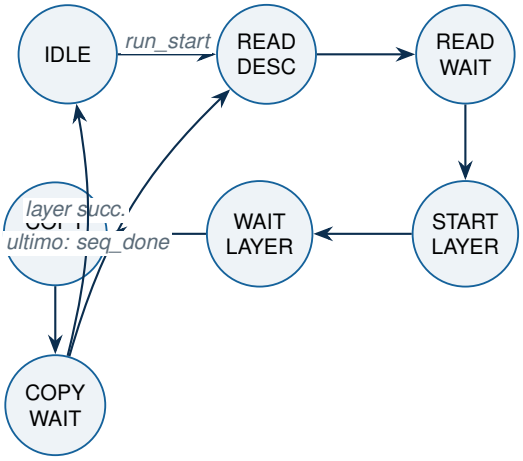
`neuron_memory` collega il datapath di calcolo alla memoria e gestisce il loop sui neuroni. Legge il vettore X una sola volta (ingresso condiviso), poi per ciascun neurone rilegge W e bias dalla RAM e li invia a una singola istanza riusata di `neuron_parallel`: il progetto è memory-bound, un neurone 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).

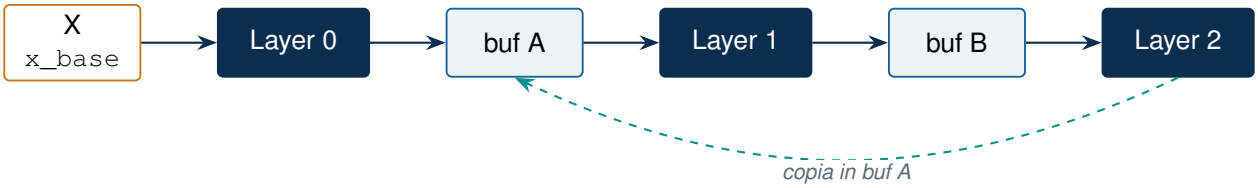
6.2 `layer_sequencer` — rete multi-layer

`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)

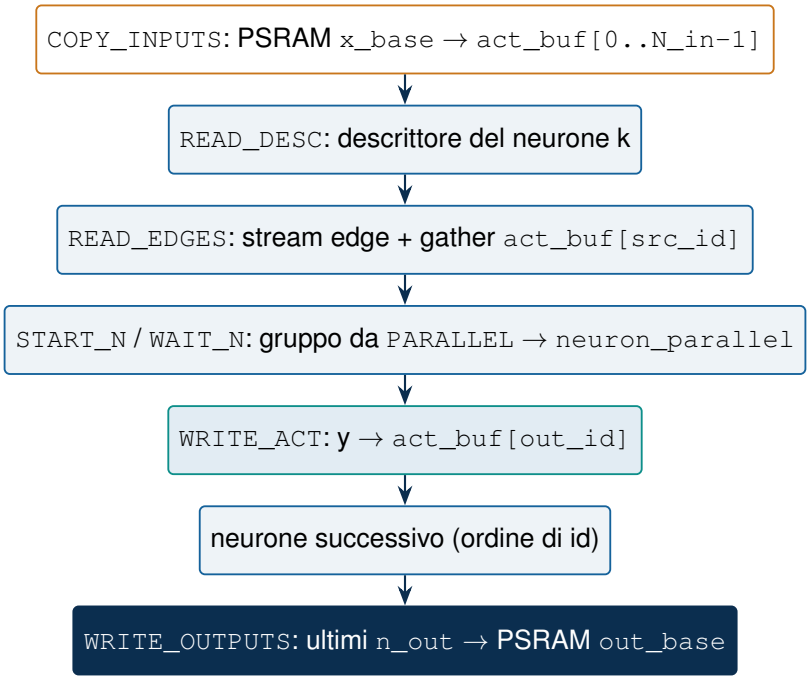
Campo	Byte	Significato
`src_id`	2	Id sorgente (uint16 BE).
`weight`	1	Peso (INT8).
`reserved`	1	0 (allineamento a 4 byte).

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	2367
TRELLIS_FF	2406
`$_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.

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

53.25 cicli/edge, coerente con la teoria: 4 byte/edge × ≈13 cicli/byte via PSRAM asincrona ≈52. A 80 MHz: ≈1.5 M edge/s (≈6.0 MB/s); al clock reale di 16 MHz: ≈300 k edge/s (≈1.2 MB/s).

Ogni edge costa oggi un round-trip PSRAM intero: è l'accesso sequenziale che il page-mode read (roadmap G7, cap. 15) accelererebbe. La misura trasforma quella ottimizzazione da apertura vaga a **decisione quantificabile**: per una rete di dimensione data si può stimare a priori se il guadagno di banda giustifica lo sforzo.

7.9 Compilatore host

La configurazione leggibile della rete non richiede logica dedicata in FPGA: si descrive la rete in **YAML** e un compilatore *sull'host* la traduce nei byte esatti delle tabelle e degli edge, poi caricati con `WRITE_RAM`. Il compilatore valida a compile time (`src_id < out_id`, limiti `N_TOTAL`, padding a `PARALLEL`), complementando il guard runtime. Formato, regole ed esempi nel cap. 10.

CAPITOLO 8

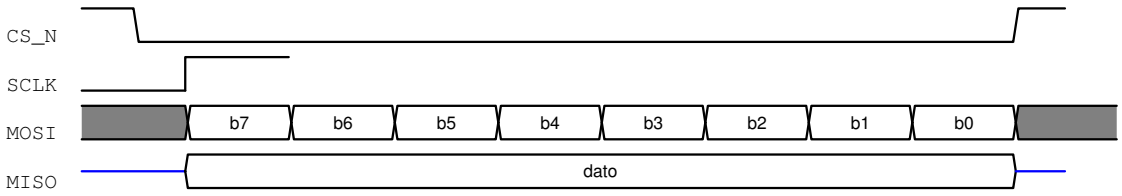
Interfaccia host SPI

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); bit7:3=0.
`0x22`	READ_OUTPU	—	`N_NEURONS` byte	`y_bus` neuron-major (byte 0 = neurone 0).
`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).

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.

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. 13) 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.

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 Sequenze di sessione

8.8.1 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
```

8.8.2 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, CRC/checksum sui trasferimenti (SPI assunto affidabile su traccia di scheda), e backpressure sui trasferimenti RAM: l'host non deve clockare comandi RAM più velocemente di una transazione RAM per byte SPI (vincolo ragionevole per il bulk-loading iniziale, non un percorso real-time).

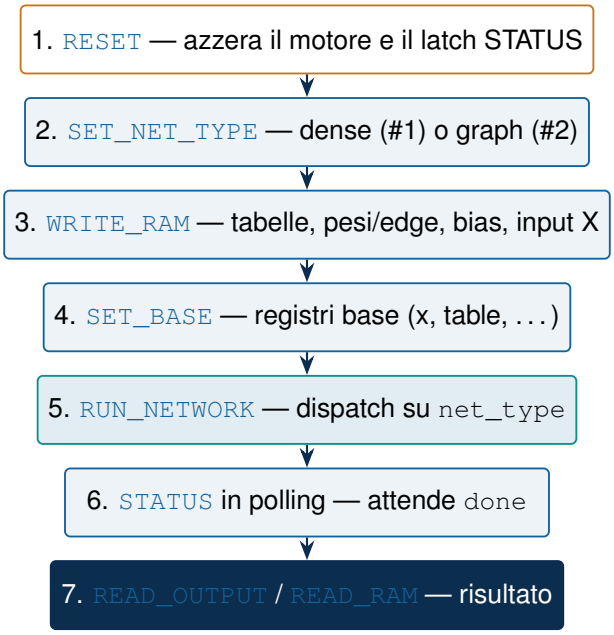
CAPITOLO 9

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 Sorgente YAML

La descrizione leggibile è in **YAML** (cap. 10), compilata dall'host esattamente nei byte delle tabelle e degli edge sopra. Sorgente equivalente al grafo dell'esempio:

Listing 9.7. YAML: sorgente e byte generati

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

# --- il compilatore emette ---
#  id assegnati:  n4=4, n5=5   (garantito src < 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, N_TOTAL, padding a PARALLEL
```

Un solo formato sorgente

Il YAML (cap. 10) e lo pseudocodice host producono gli *stessi byte*: il YAML è il formato sorgente unico, versionabile e scritto a mano o generato dall'addestratore; lo pseudocodice mostra solo cosa fa il compilatore sotto. L'FPGA riceve comunque solo tabelle e dati via `WRITE_RAM`: nessun interprete a bordo.

CAPITOLO 10

Progettazione della rete con YAML

Le reti si descrivono in un formato dichiarativo **YAML**, compilato *sull'host* nei byte nativi (tabelle descrittori ed edge-list, cap. 9) e caricato via `WRITE_RAM`. Nessun linguaggio custom e nessun interprete a bordo: il YAML si legge con un parser standard in una rappresentazione intermedia (IR), e un unico encoder emette il formato del chip. Lo stesso IR alimenta compilatore, simulatore e addestratore (framework software, livello 1).

10.1 Elementi comuni

Chiave	Significato
`type`	`dense` (Tipo #1) o `graph` (Tipo #2). Obbligatorio.
`requires`	Vincoli di build attesi (`parallel`, `n_total`); il compilatore li verifica contro `READ_CONFIG` del bitstream.
`name` / `version`	Metadati opzionali.

Aritmetica INT8 con segno ovunque: pesi e bias $\in [-128, +127]$ (accumulo INT32 interno al chip). Attivazioni: `relu` (default) o `none` (lineare, saturazione bilaterale).

10.2 Tipo #1 — rete dense

Connessioni *implicite* (fully connected tra layer): si definiscono solo pesi, bias e dimensioni.

```
type: dense
inputs: <int>                # input del primo layer
layers:
  - neurons: <int>           # n_neurons_real del layer
    activation: relu | none
    weights: [[...], [...]]  # matrice neuron-major: weights[k] = pesi del neurone k
    bias:    [...]           # un bias per neurone
```

Regole: l'input di ogni layer deve essere multiplo di `parallel`; l'input del layer k è l'uscita del layer $k-1$ (le dimensioni concatenano); `layers` $\leq$ `N_LAYERS`. Ogni layer diventa una voce di descrittore da 11 byte (cap. 9).

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

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

10.3 Tipo #2 — rete a grafo

Connessioni *esplicite*: ogni neurone elenca i suoi ingressi come edge `(src, w)`.

```
type: graph
n_total: 4096                 # spazio id (default 4096)
inputs: <int>                 # id di ingresso: 0 .. inputs-1
neurons:                      # in ordine topologico (sorgenti prima dei consumatori)
  - id: <nome|int>
    activation: relu | none
    bias: <int>
    connections:
      - {src: <id>, w: <int>} # src = ingresso o id di un neurone gia' dichiarato
outputs: [<id>, ...]          # neuroni sink -> ultimi n_out
```

Regole: gli ingressi prendono gli id `0..inputs-1`, i neuroni id crescenti da `inputs` in poi (ordine di dichiarazione); vincolo **`src < id`** del neurone (DAG feed-forward, niente cicli) e `src < n_total`; il padding di `n_conn` a multiplo di `parallel` (edge peso 0) lo fa il compilatore; gli `outputs` sono sink (non usati come `src`).

Listing 10.2. Graph n4/n5

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

10.4 Pesi: inline o file esterno

Reti piccole/scritte a mano: pesi inline. Reti addestrate: file esterno (prodotto dall'addestratore), con `weights_file`/`bias_file` (int8, shape verificata dal compilatore). `weights` e `weights_file` sono mutuamente esclusivi.

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

10.5 Regole di validazione (compile-time)

Il compilatore rifiuta il file se: pesi/bias fuori da $[-128, +127]$; (dense) input di layer non multiplo di `parallel` o dimensioni che non concatenano, lunghezze errate, `layers>N_LAYERS`; (graph) `src >= id` (ciclo/non topologico) o `src >= n_total`, o un'uscita usata come sorgente; `type` assente; `requires` incompatibile col target. Ogni errore indica il punto (layer/neurone, campo). Il chip ha comunque il guard runtime (`STATUS.err`/`irq_n`) come seconda difesa.

10.6 Casi d'uso applicativi

FPGA-Neural è un motore per **classificatori dense INT8 compatti** (o grafi sparsi): tipicamente lo *stadio finale* di una pipeline, dove le feature grezze le estrae l'host o un sensore e il chip fa la decisione, deterministica e a bassa latenza, scaricando la CPU.

10.6.1 Classificazione di volti (dense)

L'host (es. ESP32-CAM) rileva il volto ed estrae un *embedding* INT8; il chip classifica tra identità note. Registrare un volto nuovo = raffinare l'ultimo layer (runtime, livello 2).

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

```
  - {neurons: 4,  activation: none, weights_file: fc1_w.npy, bias_file: fc1_b.npy}
# uscite: [A, B, C, sconosciuto] -> l'host fa argmax
```

10.6.2 Manutenzione predittiva / anomalie (dense)

Feature INT8 da sensori di macchina (vibrazione, corrente, temperatura, già estratte dall'MCU) → classificazione *normale/anomalia* o modo di guasto. Inferenza continua a bassa latenza sul campo, senza inviare dati al cloud.

```
name: machine_health
type: dense
requires: {parallel: 8}
inputs: 64                    # feature spettrali/statistiche
layers:
  - {neurons: 16, activation: relu, weights_file: h0_w.npy, bias_file: h0_b.npy}
  - {neurons: 3,  activation: none, weights_file: h1_w.npy, bias_file: h1_b.npy}
# uscite: [ok, usura, guasto]
```

10.6.3 Parole chiave / gesti (dense)

Feature audio (MFCC) o IMU estratte dall'host → classe parola/gesto. Wake-word o comandi gestuali a bassissimo consumo.

```
name: keyword_spotter
type: dense
requires: {parallel: 8}
inputs: 40                    # 40 (n_input multiplo di 8 -> usa 40? -> serve 40%8!=0)
```

Attenzione al vincolo `parallel`

`inputs` e l'input di ogni layer devono essere multipli di `parallel`. Per 40 feature con `parallel`=8: 40 è multiplo di 8, ok. Se le feature non sono un multiplo, si porta l'ingresso al multiplo con zero-padding a monte (lato host): scelta di preparazione, non del chip.

10.6.4 Controllore a riflessi / rete sparsa (graph)

Quando ogni uscita dipende da un *sottoinsieme specifico* di ingressi (connettività irregolare, disegnata a mano), il Tipo #2 evita di pagare i pesi a zero di una dense: cabli solo le connessioni che contano. Esempio: un robot dove ogni attuatore reagisce a pochi sensori mirati.

```
name: reflex_controller
type: graph
requires: {parallel: 2}
inputs: 6                     # 6 sensori: id 0..5
neurons:
  - id: avoid_L
    activation: relu
    bias: 0
    connections: [{src: 0, w: 4}, {src: 1, w: -3}]   # solo sensori frontali sx
  - id: avoid_R
    activation: relu
    bias: 0
    connections: [{src: 4, w: 4}, {src: 5, w: -3}]   # solo frontali dx
  - id: motor_L
    activation: none
    bias: 0
    connections: [{src: avoid_L, w: 2}, {src: 2, w: 1}]
  - id: motor_R
    activation: none
    bias: 0
    connections: [{src: avoid_R, w: 2}, {src: 3, w: 1}]
outputs: [motor_L, motor_R]
```

Dense o graph?

Scegli **dense** quando la connettività è piena e regolare (classificatori su embedding, feature vector): è il percorso più veloce. Scegli **graph** quando la connettività è sparsa e specifica: risparmi cicli e memoria non pagando le connessioni a zero.

CAPITOLO 11

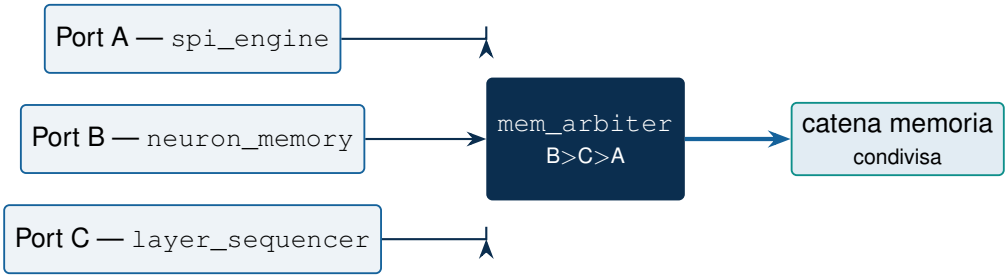
Arbitraggio e integrazione top-level

11.1 `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.

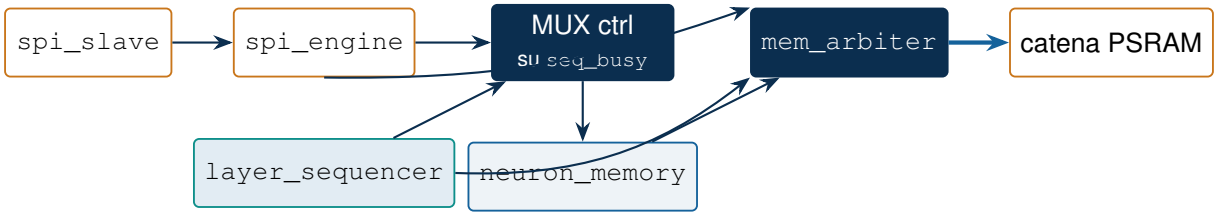
Priorità fissa **B** > **C** > **A**: un'inferenza in corso è più critica della contabilità del sequencer, che a sua volta è più critica di un accesso SPI manuale appena arrivato. In funzionamento normale B e C sono comunque temporalmente disgiunti (`neuron_memory` richiede solo durante un'esecuzione, `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.

11.2 `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. 12).

CAPITOLO 12

Implementazione e caratterizzazione ECP5

12.1 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.

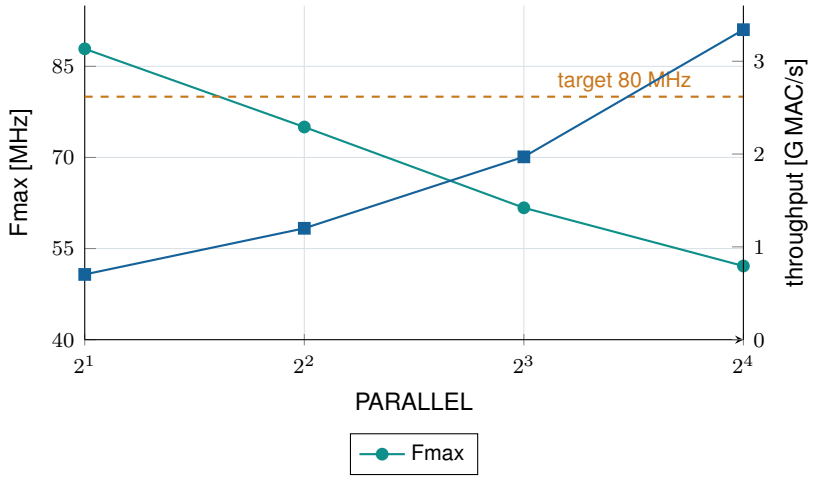
12.2 Benchmark del datapath (256×4)

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.

12.2.1 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.

12.2.2 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.

12.2.3 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.

12.3 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`.

12.3.1 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.

12.3.2 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 ($\approx$45 MHz a P8) ha $2.8\times$ di margine.

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 13

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 53 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) ed 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`: 53/245 (21%). Fmax del build corrente 73.88 MHz, entro la banda di rumore di seed della timing closure (cap. 12). Le ball di config-SPI 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.

13.1 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)

13.2 Budget dei pin

Il progetto richiede circa 58 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
SPI di configurazione (verso flash onboard)	4
JTAG (bring-up / debug, consigliato)	4
Totale	**≈58**

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

Assegnazione reale dei 53 segnali del top-level, verificata da place&route. Standard I/O: LVCMOS33 (alimentazione I/O a 3.3 V). Le ball provengono dal `.lpf` reale place&route-verified.

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.
Bus PSRAM indirizzi (banco 2, lato destro)				
`psram_a[21:0]`	OUT	E16…H20	2	22 linee di indirizzo reali (A0–A21, 8 MB).
`psram_a[22]`	OUT	P18	3	Sempre 0 (shift byte→word): NC sulla scheda.
Bus PSRAM dati (banchi 2 e 3)				
`psram_dq[9:0]`	IO	K18…K20	2	`dq[1..9]` su ball dual-function usate come GPIO ordinario.
`psram_dq[15:10`	IO	L17…P17	3	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.

13.4 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
Clock / reset	2	banco 7, `clk` su `GR_PCLK7_0`
Config SPI / JTAG	—	pin dedicati (fuori RTL, solo PCB)

13.5 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.

13.5.1 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. 13, ball reali). Disaccoppiamento per pin di alimentazione secondo il datasheet ISSI.

13.6 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 75 MHz di Fmax del sistema integrato completo dopo la timing closure (cap. 12). `CLK_FREQ_MHZ` deve essere impostato al valore reale dell'oscillatore montato, altrimenti la temporizzazione PSRAM risulta errata.

13.7 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`, $\geq$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).

13.8 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.

13.8.1 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.

13.8.2 Config-SPI verso boot flash

La FPGA carica il bitstream dalla **Winbond `W25Q128JV`** (128 Mbit SPI NOR, Quad read) all'accensione. La stessa flash può ospitare pesi/bias/metadati di rete.

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).

13.8.3 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. 13).

13.9 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).
- ☐ Confermare signal integrity PSRAM/SPI al clock effettivamente montato.
- ☐ Scelta del footprint del connettore JTAG.
- ☐ Cattura schematica (KiCad o altro): nessuno schema esiste ancora per questa combinazione dispositivo/package.

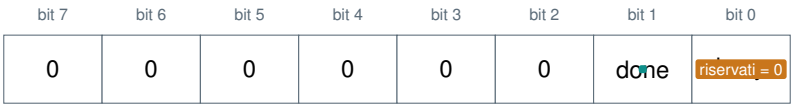
CAPITOLO 14

Riferimento rapido

14.1 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

14.2 Byte di STATUS



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

14.3 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)

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

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

14.5 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 15

Roadmap e stato di sviluppo

15.1 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`, tutti gli opcode, Fmax controllata in isolamento.
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); restano page-mode PSRAM, banda gather, block RAM x/w.
8	Training hardware (opz.)	futura	backprop, gradienti, aggiornamento pesi.

† *RTL, unit test ed end-to-end su SPI simulato completi; timing closure eseguita: 75.30 MHz (P2) / 60.26 MHz (P8), bit-esatta su tutta la regressione (cap. 12).*

15.2 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 (tutti gli opcode incl. RUN_NETWORK)	**OK** Fmax in isolamento
Dual SPI	futura
Motore multi-layer	**OK** timing closure 75.30 MHz (P2)
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, compilatore YAML)	**OK** RTL + test + sintesi
Pinout CABGA381 (`.lpf` reale)	**OK** place&route-verified 0 errori
Page-mode PSRAM (G7)	aperta, ora quantificata (53.25 cicli/edge)
Bitstream reale (`ecppack`, P2/P8)	**OK** 0 errori, part LFE5U-45F-8CABGA381
Driver host Linux / ESP32	pianificato
Training hardware	futuro

15.3 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.

15.4 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_access.v`	FSM	conversione byte ↔ word
`rtl/memory_inter`	FSM	handshake req/ready
`rtl/psram_controller.v`	FSM	bus fisico PSRAM async 70 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. 13. 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
`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_runnetwork_tb.v`	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 e vanno riverificati dopo l'assegnazione di un `.lpf` reale e la chiusura del timing di Fase 7.