POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Elettronica

Tesi di Laurea Magistrale

Metodologie di Progettazione Automatica di Reti Neurali Non Ricorrenti su FPGA

Relatori:
prof. Gianluca Setti
prof. Riccardo Rovatti

Candidato:
Luciano Prono (235239)

Aprile 2019

Ringraziamenti

Desidero ringraziare il prof. Gianluca Setti per avermi seguito nella stesura di questo lavoro, per aver alimentato il mio interesse verso la ricerca scientifica e per avermi dato la possibilità di mettermi in gioco in questo campo. Ringrazio inoltre il prof. Riccardo Rovatti che mi ha dato una direzione e mi ha seguito nella progettazione dell'architettura in questione.

Ringrazio i miei genitori che mi sostengono sempre e che mi hanno dato tantissimo nel corso di tutti questi anni.

Ringrazio i miei amici, coi quali fin'ora ho coltivato tantissimi ricordi e che non mi hanno mai fatto mancare il loro appoggio.

Ringrazio infine Elena che mi sostiene sempre e sulla quale so di poter contare.

Sommario

Al giorno d'oggi l'elaborazione avanzata dei segnali si basa su algoritmi di complessità crescente che richiedono l'uso di strutture dedicate come le Artificial Neural Network (ANN). Queste architetture di calcolo altamente specializzate sono in grado di risolvere un ampio raggio di problemi anche quando manca uno specifico modello necessario per l'elaborazione di una soluzione, garantendo risultati anche per operazioni di complessità notevole. Le ANN sono capaci di aiutare in problemi come l'estrazione di informazioni da immagini o video, il riconoscimento del discorso o del linguaggio, la rilevazione di malattie basata sull'imaging medico, il controllo di bracci robotici, robot terrestri o quadricotteri e molto altro.

Grandi compagnie dell'IT come Google o Microsoft tendono a ottenere un'accuratezza crescente per le loro reti incrementando aggressivamente la loro complessità in termini di profondità (numero di layer) e dimensione (neuroni per layer). Questo risulta in sistemi composti da migliaia di processori che consumano centinaia di kilo-watt.

Questo è intrinsecamente incompatibile con molte aree di applicazione come l'acquisizione di segnali biomedici attraverso reti di nodi su aree del corpo che richiede sistemi piccoli e a basso consumo. Se l'obiettivo è la prototipazione rapida di questi nodi, ci sono principalmente due possibilità: i microcontrollori e gli FPGA. Mentre l'implementazione della ANN su microcontrollore a livello software è piuttosto immediata, lo stesso non si può dire per l'implementazione a livello hardware su FPGA. Allo stesso tempo ci si aspetta che l'implementazione su FPGA sia più efficiente sia in termini di consumi energetici che di velocità di calcolo. Lo scopo di questo lavoro è la progettazione di una struttura completamente parametrizzata descritta in un linguaggio di descrizione dell'hardware (VHDL) capace di generare automaticamente l'ANN desiderata. Con questo approccio un progettista sarebbe capace di generare un'intera ANN funzionante utilizzando un tempo comparabile se non inferiore a quello speso scrivendo il software su microcontrollore. Le ANN generate in questo modo sono non ricorrenti e comprendono layer fully connected, convolutivi

e max-pooling; il progettista può scegliere il numero e il tipo di layer, il numero di neuroni impiegati e persino il numero di operazioni parallele effettuate.

Queste architetture vengono testate e le loro prestazioni vengono confrontate con l'equivalente funzionale che si ottiene programmando su microcontrollore: in questo modo si dimostra l'effettiva utilità di questo tipo di approccio. Per esempio, una ANN fully connected con 32-32-3 neuroni impiega 750 ns e 100 nJ per operazione su Intel Cyclone 10 FPGA, mentre su microcontrollore ARM Cortex M4 impiega 3.5 µs e 300 nJ.

Struttura della tesi

Nel capitolo 1 viene spiegato a livello teorico come funzionano le reti neurali prese in considerazione in questo lavoro e vengono dettagliate le operazioni che i layer fully connected, convolutivi e max-pooling eseguono. Viene anche analizzato in breve lo stato dell'arte di questo tipo di architetture.

Nel capitolo 2 viene mostrata estensivamente l'implementazione della rete neurale su FPGA; viene spiegato come i vari blocchi funzionano, come ci si interfaccia al circuito e le particolarità riguardo ad alcune scelte nell'impiego del linguaggio VHDL.

Nel capitolo 3 viene mostrata l'implementazione su microcontrollore della rete neurale; tale implementazione è quella che verrà confrontata con l'equivalente su FPGA.

Nel capitolo 4 vengono descritte le modalità di test delle reti neurali, i casi studio che vengono impiegati per i test e i risultati e i confronti in termini di performance tra FPGA e microcontrollore. L'FPGA garantisce frequenze operative superiori, meno cicli per operazione, un consumo energetico per operazione inferiore e una capacità di archivio dei pesi superiore con l'unico svantaggio di essere leggermente più costoso rispetto ai microcontrollori.

Indice

Ringraziamenti I

Sommario II

1 Reti Neurali 1
- 1.1 Il perceptron e il neurone . . . 1
- 1.2 Reti neurali fully connected . . . 3
- 1.3 Reti neurali convolutive . . . 5
  - 1.3.1 Layer convolutivi . . . 5
  - 1.3.2 Layer max-pooling . . . 9
- 1.4 Applicazioni e stato dell'arte . . . 9

2 Struttura su FPGA 12
- 2.1 I Field Programmable Gate Array (FPGA) . . . 12
  - 2.1.1 Composizione . . . 13
  - 2.1.2 Prestazioni . . . 13
  - 2.1.3 Dispositivi in commercio . . . 13
- 2.2 Struttura del layer . . . 14
  - 2.2.1 Il Multiply and Accumulate (MAC) . . . 14
  - 2.2.2 Struttura di base dei layer fully connected o convolutivi: gruppo `neurons` . . . 15
  - 2.2.3 Struttura di base dei layer max-pooling . . . 22
  - 2.2.4 Stadi funzionali del layer . . . 24
  - 2.2.5 Sistema di controllo dei layer fully connected . . . 26
  - 2.2.6 Sistema di controllo dei layer convolutivi e max-pooling . . . 33
  - 2.2.7 Pipeline del layer . . . 40
- 2.3 Struttura complessiva della rete neurale . . . 40
  - 2.3.1 Caricamento degli input nella rete neurale . . . 42

2.3.2 Interfaccia di uscita della rete neurale 42
2.3.3 Caricamento dei pesi nella rete neurale 44
2.4 Utilizzo del VHDL per generare una rete neurale parametrizzata . . . 47
2.4.1 Parallelismo e tipi di dato 48
2.4.2 Top level entity 48
2.4.3 File di gestione dei parametri 48

3 Struttura su microcontrollore 51
3.1 Struttura del codice 52
3.2 Dichiarazione dei parametri 53
3.3 Dichiarazione delle strutture dati 55
3.4 Caricamento dei pesi 57
3.5 Elaborazione 58
3.5.1 Layer fully connected 59
3.5.2 Layer convolutivo 60
3.5.3 Layer max-pooling 62
3.6 Generazione dinamica del codice col preprocessore 63

4 Prestazioni e comparazione 64
4.1 Modalità di test 64
4.1.1 Generazione dei parametri delle reti 64
4.1.2 Testing dell'implementazione su FPGA 65
4.1.3 Testing dell'implementazione su microcontrollore 66
4.2 Prestazioni della rete fully-connected 67
4.2.1 Caso studio: riconoscimento di segnali elettromiografici per il movimento di una mano 67
4.2.2 Prestazioni della rete su FPGA 68
4.2.3 Prestazioni della rete su microcontrollore 71
4.2.4 Comparazione dei risultati 72
4.3 Prestazioni della rete convolutiva 76
4.3.1 Caso studio: MNIST data-set 76
4.3.2 Prestazioni della rete su FPGA 77
4.3.3 Prestazioni della rete su microcontrollore 79
4.3.4 Comparazione dei risultati 81
4.4 Conclusioni 83

A Codice completo per l'architettura su FPGA 86
A.1 Packages 86
A.1.1 Pacchetto per la gestione degli array di valori interi di dimensione indefinita (`generic_array_pkg.vhd`) 86
A.1.2 Pacchetto per la valutazione del numero di bit per la rappresentazione di un valore (`bit_evaluation_pkg.vhd`) 88
A.1.3 Pacchetto per la gestione dei record contenenti i parametri dei layer (`layer_record_pkg.vhd`) 91
A.1.4 Pacchetto per la valutazione di parametri interni della rete neurale (`nn_utils_pkg.vhd`) 92
A.2 Codice sorgente dell'architettura 94
A.2.1 Contatore (`counter.vhd`) 94
A.2.2 Contatore "multidimensionale" (`counter_multid.vhd`) 96
A.2.3 Funzione di attivazione (`activation.vhd`) 98
A.2.4 Multiply and accumulate (`mac.vhd`) 99
A.2.5 Gruppo MAC e memoria dei pesi (`mac_group.vhd`) 101
A.2.6 Gruppo di neuroni (`neurons.vhd`) 103
A.2.7 Blocco per il calcolo del valore massimo (`max.vhd`) 107
A.2.8 Datapath del layer (`datapath.vhd`) 109
A.2.9 Layer della rete neurale (`nn_layer.vhd`) 116
A.2.10 Rete neurale completa (`neural_network.vhd`) 122
A.2.11 Top level entity (`neural_network_top.vhd`) 126

B Codice completo per il microcontrollore 129
B.1 File di espansione delle funzionalità del preprocessore (`pp_helper.h`) 129
B.2 Parametri definiti dall'utente (`user_parameters.h`) 131
B.3 File di definizione delle macro per la rete neurale (`nn_macros.h`) . . . 131
B.4 File principale (`main.cpp`) 136

Riferimenti bibliografici 139

Elenco delle figure

1.1 Funzione sigmoid 3
1.2 Schema del neurone 3
1.3 Schema di una rete neurale fully connected 4
1.4 Disposizione dei dati che compongono un'immagine 6
1.5 Campo di ricezione del layer convolutivo 6
1.6 Stride e padding nei layer convolutivi 7
1.7 Schema di un layer convolutivo 8
1.8 Schema del layer max-pooling 9

2.1 Implementazione circuitale del MAC 16
2.2 Struttura hardware generica di un layer 17
2.3 Struttura di un MAC-group 18
2.4 Funzioni di attivazione implementate in hardware 19
2.5 Struttura del gruppo di neuroni di un layer 21
2.6 Memorie dei pesi e dei bias 22
2.7 Struttura `max` per il calcolo del massimo 23
2.8 Struttura set-reset 25
2.9 Connessione tra le unità di controllo 26
2.10 Datapath del layer della rete neurale 27
2.11 Associazione del datapath del layer con le macchine a stati 28
2.12 Datapath dello stadio *load inputs* del layer fully connected 29
2.13 Unità di controllo dello stadio *load inputs* del layer fully connected . . 29
2.14 Unità di controllo dello stadio *elaborate* del layer fully connected . . . 31
2.15 Unità di controllo dello stadio *save outputs* del layer fully connected . 33
2.16 Architettura per l'indirizzamento di volumi di dati multidimensionali 34
2.17 Contatori aggiuntivi utilizzati dai layer convolutivi 35
2.18 Unità di controllo dello stadio *load inputs* per il layer convolutivo . . 36
2.19 Unità di controllo dello stadio *elaborate* per il layer convolutivo 38

2.20 Unità di controllo dello stadio *save outputs* per il layer convolutivo . . 39
2.21 Schema del pipeline del layer fully connected 41
2.22 Rete neurale come singolo blocco 42
2.23 Architettura completa della rete neurale 43
2.24 Timing dell'interfaccia di ingresso della rete neurale 44
2.25 Timing della ricezione degli output 44
2.26 Timing del caricamento dei pesi 45
2.27 Unità di controllo *load weights* della rete neurale 46

4.1 Cicli per operazione e frequenza su FPGA (rete fully-connected) . . . 68
4.2 Tempo per operazione e potenza su FPGA (rete fully-connected) . . . 69
4.3 Energia per operazione su FPGA (rete fully-connected) 70
4.4 Cicli per operazione su microcontrollore (rete fully-connected) 71
4.5 Tempo ed energia per operazione su microcontrollore *mainstream* (rete fully-connected) 73
4.6 Tempo ed energia per operazione su microcontrollore *high performance* (rete fully-connected) 73
4.7 Tempo ed energia per operazione su microcontrollore *ultra low power* (rete fully-connected) 74
4.8 Cicli per operazione e frequenza su FPGA (rete convolutiva) 78
4.9 Tempo per operazione e potenza su FPGA (rete convolutiva) 80
4.10 Energia per operazione su FPGA (rete convolutiva) 80
4.11 Cicli per operazione su microcontrollore (rete convolutiva) 81
4.12 Tempo ed energia per operazione su microcontrollore *mainstream* (rete convolutiva) 82
4.13 Tempo ed energia per operazione su microcontrollore *high performance* (rete convolutiva) 82
4.14 Tempo ed energia per operazione su microcontrollore *ultra low power* (rete fully-connected) 83

Elenco delle tabelle

4.1 Caratteristiche elettriche dei microcontrollori esaminati per il confronto 72
4.2 Costi dei chip della famiglia Intel Cyclone 10 LP 76
4.3 Costi dei microcontrollori Arm Cortex M4 utilizzati per i confronti . . 76

Listings

2.1 Descrizione in C del comportamento del MAC. 14
2.2 Descrizione in C del comportamento del MAC con l'aggiunta del segnale di bypass. 15
2.3 Descrizione in C del comportamento generico di un layer. 16
2.4 Descrizione in C del comportamento del blocco `max`. 23
2.5 Esempio di file dei parametri per la rete neurale su FPGA. 49
3.1 Struttura generica del codice per l'implementazione della rete neurale su microcontrollore. 52
3.2 Esempio di file per la definizione della rete neurale su microcontrollore (`user_parameters.h`). 55
3.3 Dichiarazione delle strutture di dati utilizzate durante le operazioni del microcontrollore, per una rete con layer convolutivo, max-pooling e fully connected. `inputs3` è la matrice di output della rete neurale. . 56
3.4 Salvataggio dei pesi e dei bias di un layer convolutivo; il caricamento per i layer fully connected è simile. Ovviamente l'indice utilizzato nel nome degli array e delle costanti varia a seconda dell'indice del layer a cui ci si riferisce. 58
3.5 Preparazione della matrice di input della rete. 58
3.6 Salvataggio della matrice di output della rete. 59
3.7 Implementazione del layer fully connected su microcontrollore. 60
3.8 Implementazione del layer convolutivo su microcontrollore. 61
3.9 Implementazione del layer max-pooling su microcontrollore. 62
A.1 Pacchetto per la gestione degli array di valori interi di dimensione indefinita (`generic_array_pkg.vhd`). 86
A.2 Pacchetto per la valutazione del numero di bit per la rappresentazione di un valore (`bit_evaluation_pkg.vhd`). 88
A.3 Pacchetto per la gestione dei record contenenti i parametri dei layer (`layer_record_pkg.vhd`). 91

A.4 Pacchetto per la gestione dei record contenenti i parametri dei layer (`nn_utils_pkg.vhd`). 92
A.5 File di descrizione del contatore (`counter.vhd`). 94
A.6 File di descrizione del contatore "multidimensionale" per la generazione di sotto-indirizzi (`counter_multid.vhd`). 96
A.7 File di descrizione del blocco funzione di attivazione (`activation.vhd`). 98
A.8 File di descrizione del multiply and accumulate (`mac.vhd`). 99
A.9 File di descrizione del gruppo MAC con memoria dei pesi e registro di output (`mac_group.vhd`). 102
A.10 File di descrizione del gruppo di neuroni (`neurons.vhd`). 103
A.11 File di descrizione del blocco per il calcolo del valore massimo (`max.vhd`).107
A.12 File di descrizione del datapath del layer (`datapath.vhd`). 109
A.13 File di descrizione del layer della rete neurale (`nn_layer.vhd`). 116
A.14 File di descrizione della rete neurale completa (`neural_network.vhd`). 122
A.15 File di descrizione dell'entity di incapsulamento della rete neurale (`neural_network_top.vhd`). 126
B.1 File di espansione delle funzionalità del preprocessore (`pp_helper.h`). 129
B.2 File di definizione delle macro per la rete neurale (`nn_macros.h`). . . 132
B.3 File principale dell'implementazione su microcontrollore (`main.cpp`) . 136

Capitolo 1

Reti Neurali

Le reti neurali forniscono un metodo efficace di risoluzione automatica di problemi difficilmente modellizzabili a causa della loro complessità; la loro struttura prende ispirazione dal cervello umano. Tipico è ad esempio l'impiego per il riconoscimento di caratteri scritti a mano. Tuttavia gli utilizzi possono essere dei più disparati, dall'identificazione di un suono alla classificazione di un'immagine. Sono ideali per sistemi di carattere generale dei quali non si conoscono gli input a priori con certezza.

Esistono reti neurali feed-forward e ricorrenti: le prime non presentano strutture cicliche e gli output dipendono direttamente e unicamente dagli input correnti, senza dipendenze temporali; le seconde invece presentano strutture cicliche e sono attraversate da segnali che hanno una dipendenza temporale, rendendo gli output dipendenti sia dagli input che dallo stato attuale della rete. In questo lavoro si analizza e si implementa unicamente il primo tipo di rete. Il secondo si avvicina maggiormente a ciò che è il cervello umano, tuttavia ha una complessità estremamente più alta che attualmente lo rende di difficile impiego, tant'è che gli algoritmi di apprendimento attuali non sono efficaci come quelli delle reti feed-forward [1].

1.1 Il perceptron e il neurone

In letteratura il perceptron è un oggetto che riceve in input N valori, li pesa moltiplicandoli per N pesi e li somma tra di loro. Se il valore così ottenuto supera una certa soglia (*threshold*) allora il perceptron si attiva, altrimenti rimane spento. Questo si può esprimere come segue

$$\text{output} = \begin{cases} 0 & \text{se } \sum_{j=0}^{N-1} x_j \cdot w_j \leq \textit{threshold} \\ 1 & \text{se } \sum_{j=0}^{N-1} x_j \cdot w_j > \textit{threshold} \end{cases} \tag{1.1}$$

L'equazione può essere rimaneggiata spostando il valore *threshold* dal membro di destra a quello di sinistra della disequazione: questo valore fisso che viene sommato agli input pesati viene chiamato *bias*.

$$\text{output} = \begin{cases} 0 & \text{se } b + \sum_{j=0}^{N-1} x_j \cdot w_j \leq 0 \\ 1 & \text{se } b + \sum_{j=0}^{N-1} x_j \cdot w_j > 0 \end{cases} \tag{1.2}$$

Un modello del genere tuttavia rende difficoltosi i processi di apprendimento, poiché la funzione di uscita è una funzione gradino [1]. Occorre sostituire la funzione gradino con una funzione derivabile, che chiamiamo funzione di attivazione. Fatto ciò otteniamo quello che viene chiamato neurone. Il comportamento ora è il seguente

$$\begin{cases} z = b + \sum_{j=0}^{N-1} x_j \cdot w_j \\ \texttt{output} = \sigma(z) \end{cases} \tag{1.3}$$

La funzione di attivazione più utilizzata è la cosiddetta sigmoid (eq. 1.4, figura 1.1), il cui comportamento è quello di un gradino smorzato. Tuttavia si possono utilizzare funzioni alternative di implementazione più facile, come mostrato nel paragrafo 2.2.2.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{1.4}$$

In figura 1.2 si può vedere la struttura schematica del neurone: agli input viene applicato un peso, successivamente gli input pesati vengono sommati (con l'aggiunta del bias) e infine viene applicata la funzione di attivazione.

Il comportamento del neurone alla fine non è altro che quello di un filtro digitale a cui viene applicata una funzione generalmente non lineare in uscita.

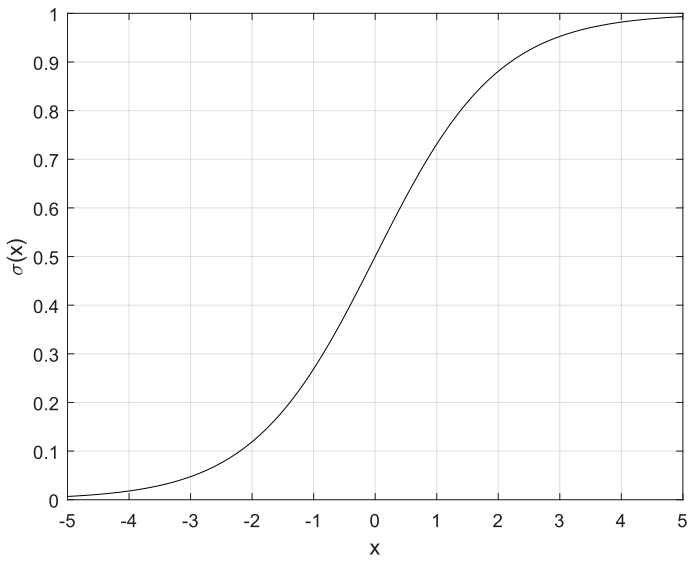


Figura 1.1: Funzione sigmoid.

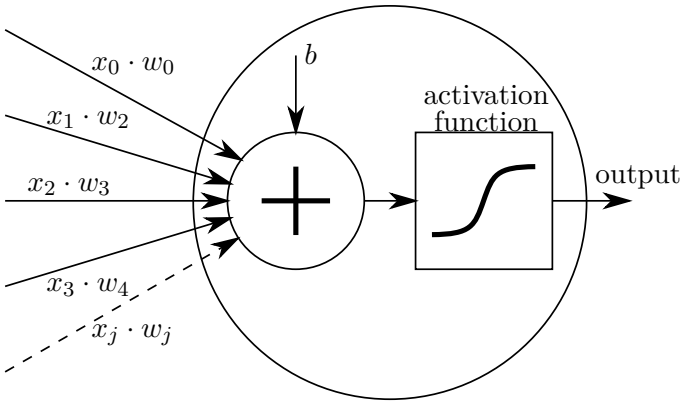


Figura 1.2: Schema del neurone.

1.2 Reti neurali fully connected

Le reti neurali sono organizzate in layer (livelli). Ogni rete dispone di un layer di input, di un layer di output e di una serie di layer nascosti (hidden). Lo scopo del primo layer è semplicemente quello di inviare gli input al primo layer nascosto, senza alcun tipo di elaborazione. Per semplicità in questo lavoro si conterà il numero di layer omettendo il primo poiché esiste solo dal punto di vista teorico e non implementativo.

Le reti neurali fully connected sono reti composte unicamente da layer di tipo

fully connected. I layer di questo tipo sono i più generici possibili: dati S neuroni e N input, tutti gli input vengono inviati a tutti i neuroni; si avranno così S output. Ad ogni neurone vengono associati N pesi più un bias perché tale è il numero di input che riceve. In figura 1.3 si può vedere la struttura schematica di una rete composta da layer di questo tipo.

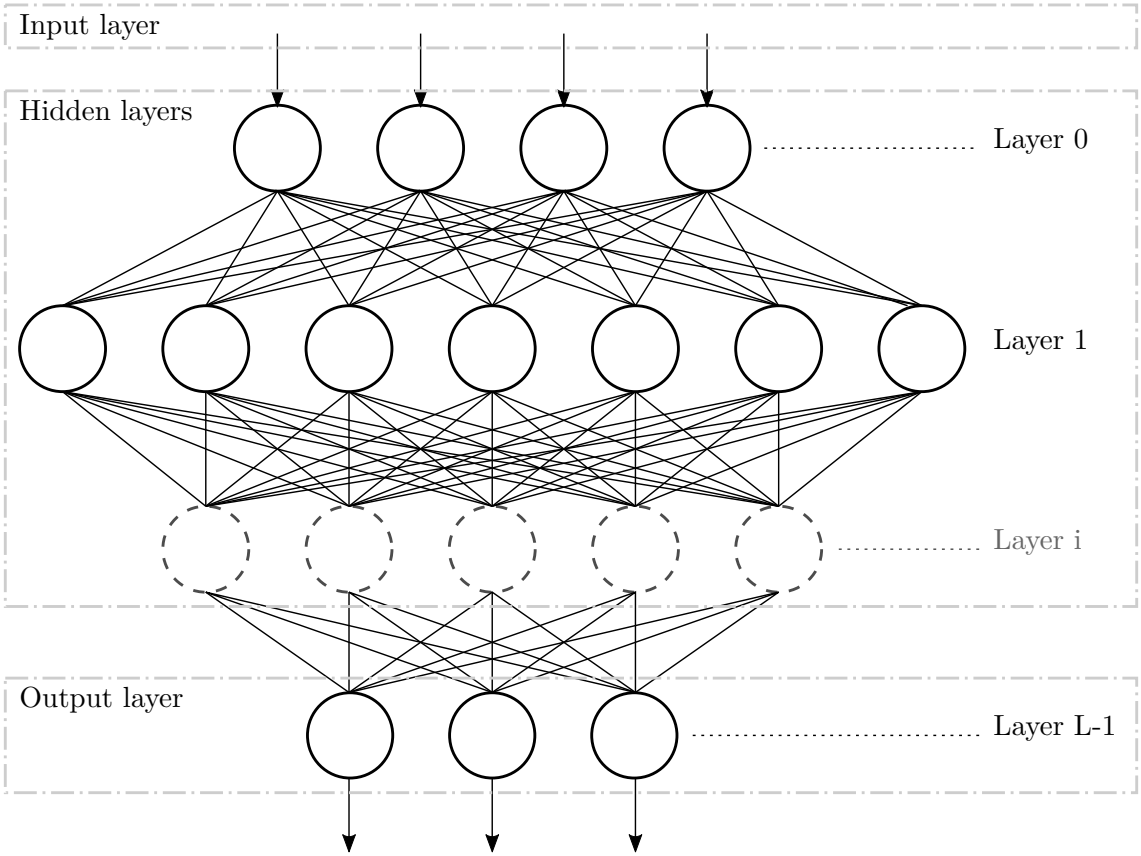


Figura 1.3: Schema di una rete neurale fully connected.

I pesi associati agli input di ciascun neurone vengono ricavati mediante tecniche di apprendimento come la discesa del gradiente, che sfrutta una funzione di costo per minimizzare l'errore dei valori di output. L'approfondimento di questo aspetto è al di fuori degli scopi del presente lavoro. Basti sapere che l'apprendimento avviene utilizzando dei data-set: ad ogni gruppo di input viene associato un gruppo di output e utilizzando tecniche di back-propagation la rete viene indotta a riprodurre questa associazione. Questi data-set sono detti di addestramento (training). L'efficacia della rete viene quindi testata utilizzando un altro data-set detto di validazione (validation): applicati degli input alla rete, si osserva quanto gli output generati si

avvicinino a quelli desiderati. Per quanto concerne questo lavoro i pesi e i bias delle reti in esame vengono generati utilizzando il Deep Learning Tool™ del software Matlab™ [2].

1.3 Reti neurali convolutive

Una tipologia di rete neurale molto utilizzata soprattutto per l'analisi di immagini è la rete neurale convolutiva. È generalmente composta da layer di tipo differente l'uno dall'altro, ma la caratteristica principale di queste reti è la presenza appunto di almeno un layer convolutivo. Mentre i layer fully connected collegano tutti gli input a tutti i neuroni e dunque ciascun output dipende da tutti gli input, i layer convolutivi impiegano la cosiddetta connettività locale: ciascun output dipende solo da una porzione di input.

1.3.1 Layer convolutivi

I layer convolutivi lavorano su volumi di dati organizzati in più matrici di uguale dimensione: questo è l'ideale ad esempio per le immagini, in cui ad ogni valore corrisponde un pixel e per ogni colore (rosso, verde e blu) si ha una matrice diversa, come viene schematizzato in figura 1.4. I dati all'interno di questi tensori tridimensionali vengono identificati da una tripletta di indici i, j e k dove i indica la riga, j la colonna e k la matrice. Si definisce con H l'altezza del volume (lungo i), con W la larghezza (lungo j) e con D la profondità (lungo k).

Il campo di ricezione del layer (*receptive field* [3]) è un parametro che descrive l'estensione della connettività dei neuroni con gli input: indica la dimensione della porzione degli input che vengono utilizzati. In figura 1.5 ciò viene mostrato schematicamente: l'altezza e la larghezza del campo sono arbitrarie mentre la profondità è pari al numero totale di matrici del volume di dati e non è modificabile. Si chiama l'altezza del campo F_h e la larghezza del campo F_w; come detto, la profondità è pari a quella dell'intero volume di dati e quindi è D.

Il neurone contiene un numero di pesi pari al numero di dati presenti nel campo di ricezione. Il suo funzionamento è identico a quello dei layer fully connected: preleva gli input, li moltiplica per i pesi, somma un bias e infine applica una funzione di attivazione. Poiché ad ogni input è associato un peso, anche i pesi all'interno del neurone possono essere organizzati come un volume di valori di dimensioni identiche al campo di ricezione.

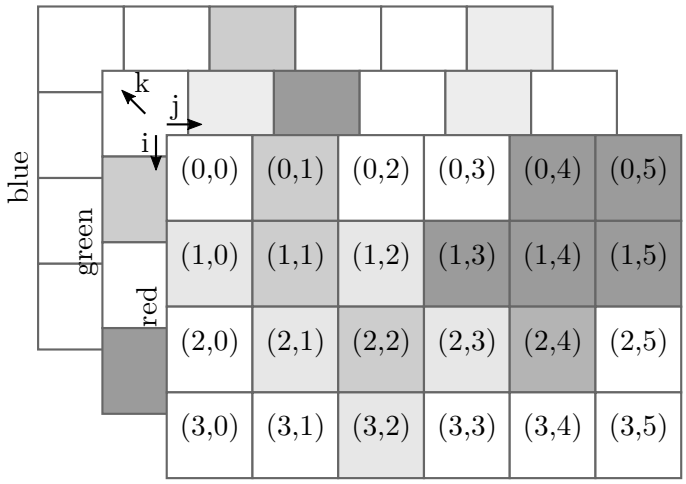


Figura 1.4: Disposizione dei dati di un'immagine: si hanno diverse matrici, ciascuna corrispondente ad un colore diverso. L'indice i corrisponde alla coordinata x del pixel mentre l'indice j corrisponde alla coordinata y.

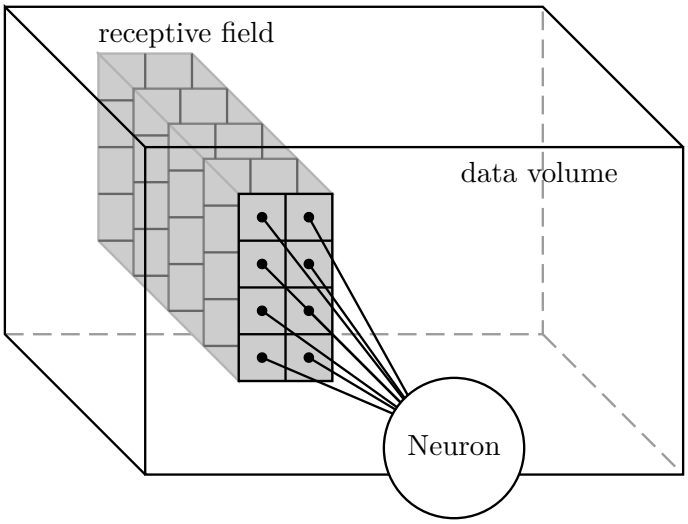


Figura 1.5: Campo di ricezione del layer convolutivo. Tutti i valori presenti nel volume racchiuso dal campo vengono inviati al neurone e a ciascuno di essi è associato un peso diverso.

È possibile fare un'assunzione: se il riconoscimento di una caratteristica avviene in una certa porzione del volume, è ragionevole assumere che questa possa avvenire anche in un'altra porzione [3]. Questo implica che si possa spostare il campo di ricezione lungo l'altezza e la larghezza del volume di dati utilizzando sempre lo stesso neurone con gli stessi pesi. È evidente che fare ciò permetta di utilizzare un numero

ridotto di pesi, il che è conveniente. Durante la valutazione degli output il campo di ricezione percorre tutto il volume. Ad ogni posizione corrisponde un output. Il parametro che indica di quanto si sposta il campo volta per volta si chiama *stride*. Lo spostamento lungo l'indice i non è necessariamente uguale a quello lungo j, per cui il parametro *stride* è composto da due valori. Inoltre esiste un altro parametro denominato *padding*: esso permette di aggiungere una cornice di zeri intorno alle matrici di input, il che è molto utile per rilevare le caratteristiche cercate anche sugli spigoli del volume di dati. È composto di 4 valori che in ordine indicano quante file di zeri aggiungere in alto, in basso, a sinistra e a destra. In figura 1.6 è riassunto il significato di questi due parametri.

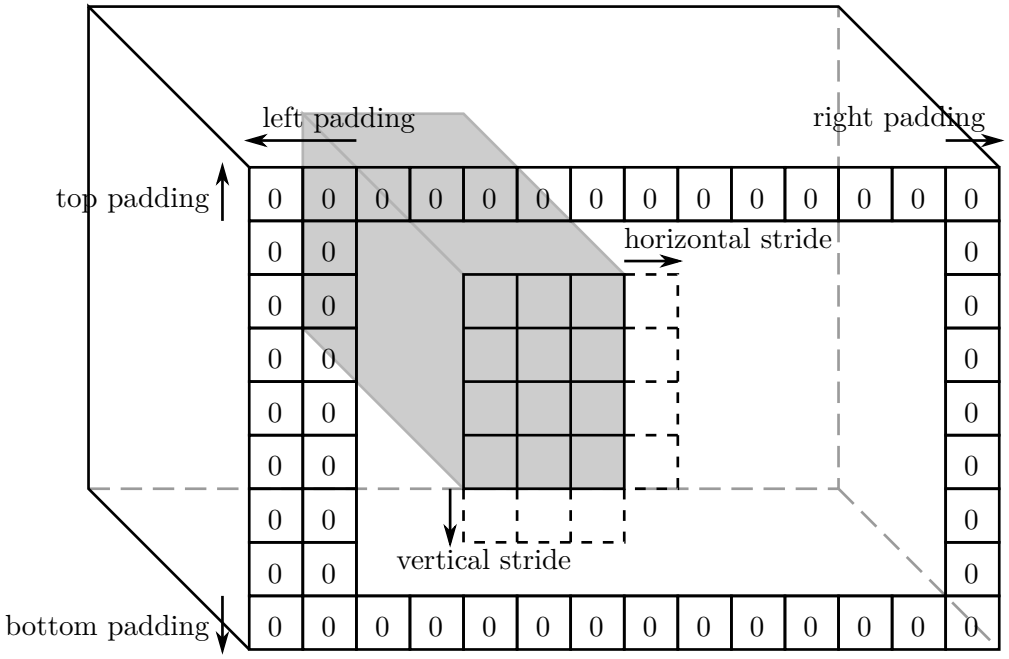


Figura 1.6: Stride e padding nei layer convolutivi.

Se per ogni posizione del campo di ricezione si ottiene un output è allora possibile valutare il numero di valori in uscita da layer. Anche gli output vengono organizzati in una matrice. La disposizione dei valori nella matrice di uscita è coerente con la posizione del campo di ricezione nel momento in cui l'output viene calcolato. Le dimensioni della matrice di output sono

$$\begin{cases} H_{\texttt{out}} = \lfloor (H - F_h + \textit{top padding} + \textit{bottom padding}) / \textit{vertical stride} \rfloor + 1 \\ W_{\texttt{out}} = \lfloor (W - F_w + \textit{left padding} + \textit{right padding}) / \textit{horizontal stride} \rfloor + 1 \end{cases} \tag{1.5}$$

Occorre che i quozienti appena riportati siano esatti affinché venga analizzato il volume nella sua interezza.

La descrizione riportata fin'ora assume che si stia utilizzando un solo neurone. È tuttavia possibile applicare più filtraggi allo stesso volume di input utilizzando S diversi neuroni, ciascuno con gli stessi parametri (*receptive field*, *stride* e *padding*) ma diversi pesi e bias. A partire da ciascun neurone si ottiene una matrice bidimensionale di output e in uscita complessivamente si avrà quindi un tensore tridimensionale di dimensione $H_{\text{out}} \times W_{\text{out}} \times S$. Gli output di diversi neuroni si esplorano modificando l'indice k del volume di uscita. L'ordine dei piani è uguale all'ordine in cui sono disposti i neuroni. In figura 1.7 si può osservare quanto detto fin'ora in forma schematica.

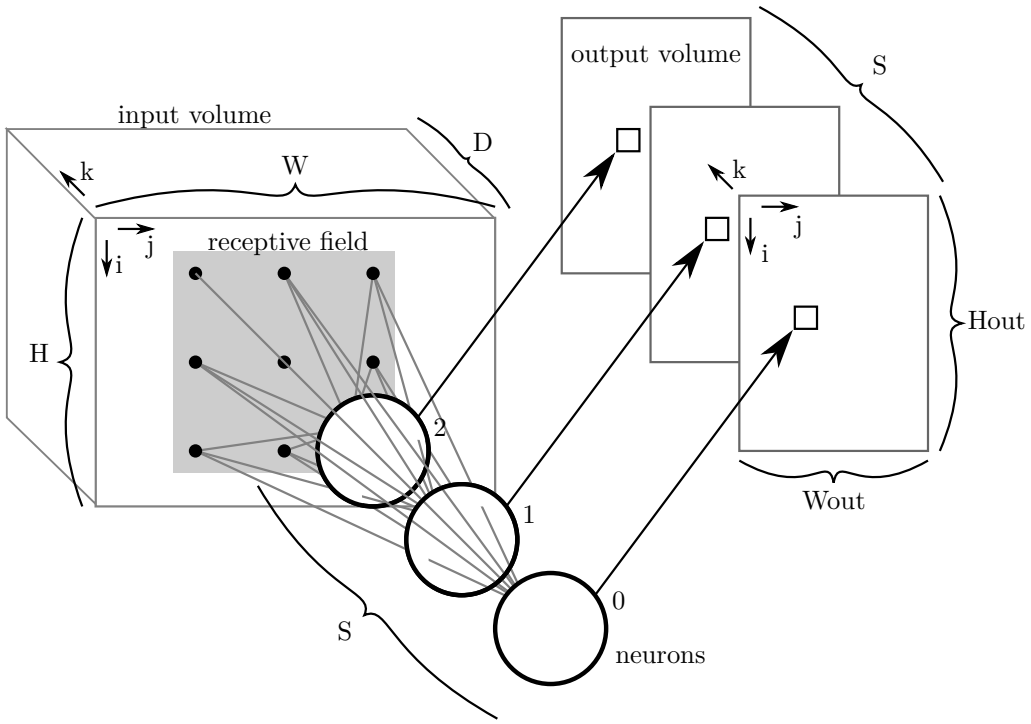


Figura 1.7: Schema di un layer convolutivo.

Qualora si effettui un collegamento da un layer di questo tipo ad un layer fully connected, i cui input hanno una disposizione monodimensionale, i dati vengono inviati colonna per colonna in ordine e quindi matrice per matrice.

1.3.2 Layer max-pooling

Esistono delle tipologie di layer affini ai layer convolutivi che permettono la riduzione del numero di campioni da un layer a un altro senza bisogno di memorizzare ulteriori pesi. Questi layer sono detti di pooling. In particolare in questo lavoro viene implementato il layer max-pooling.

Il funzionamento di questo tipo di layer è semplice. Come per i layer convolutivi, si hanno come parametri il campo di ricezione, lo *stride* e il *padding*. Viene calcolato, indipendentemente per ognuna delle matrici del tensore di input, il valore massimo tra quelli evidenziati dal campo di ricezione. In uscita quindi si avrà per ogni posizione del campo di ricezione un vettore composto da D valori. Le dimensioni delle matrici in uscita si calcolano esattamente come per i layer convolutivi, mentre la profondità del volume è pari a quella del volume di input ($H_{\text{out}} \times W_{\text{out}} \times D$). In figura 1.8 si può vedere uno schema del layer max-pooling.

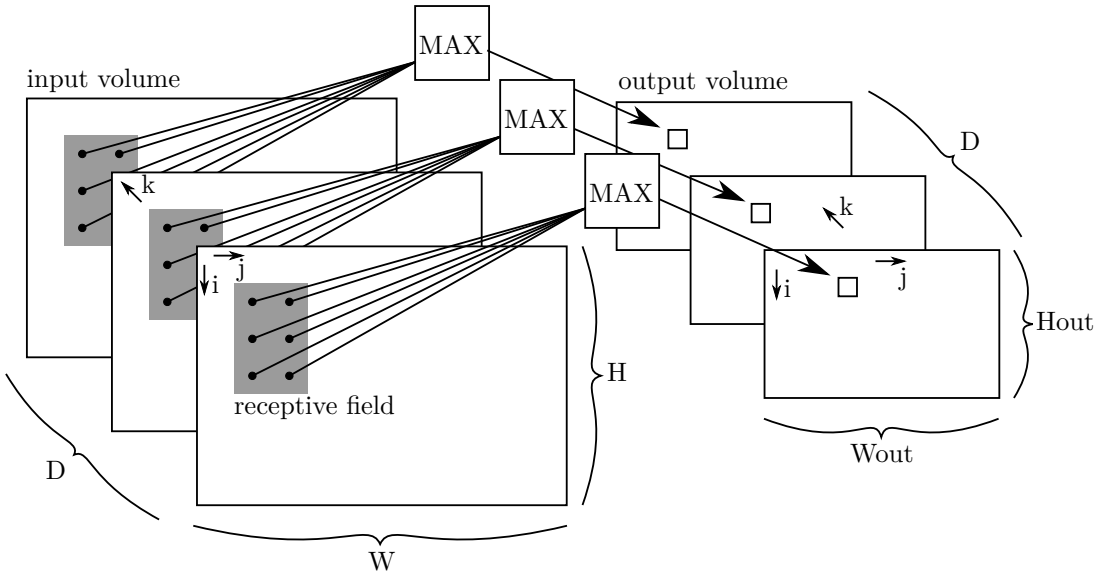


Figura 1.8: Schema di un layer max-pooling.

1.4 Applicazioni e stato dell'arte

La capacità di risolvere problemi di carattere generale consente alle reti neurali di venire impiegate in numerosi campi. Tra questi si ha il riconoscimento ed elaborazione di immagini, del parlato, diagnosi mediche, analisi di finanza con previsione

dell'andamento dei mercati, data mining o anche il controllo di bracci robotici, robot terrestri e quadricotteri [4].

C'è un grande impegno da parte della comunità scientifica per lo sviluppo di tecnologie efficienti che garantiscano una sempre maggiore precisione nei risultati riducendo al contempo l'area, la potenza e il tempo di calcolo richiesti. In ambito digitale i progressi nel campo sono dovuti in larga parte all'impiego di GPU (Graphical Processing Units), in particolare grazie alla loro capacità di moltiplicare velocemente tra di loro matrici con rapporto di aspetto vicino all'unità [5]. Si ricercano miglioramenti mediante la progettazione di strutture simil-GPU votate però esplicitamente all'implementazione di reti neurali. Questo può avvenire sia tramite la progettazione di un Application Specific Integrated Circuit (ASIC) oppure con l'impiego molto meno costoso di Field Programmable Gate Array (FPGA): il blocco circuitale principale per queste architetture è il Multiply And Accumulate (MAC) che esegue una somma di prodotti. Inoltre uno degli aspetti cruciali in questo tipo di applicazioni è il trasferimento di ingenti quantità di dati al di fuori del chip, che spesso comporta consumi e rallentamenti ben superiori a quelli causati dall'acceleratore hardware stesso: per questa ragione è importante far sì che almeno i dati intermedi tra i vari layer non viaggino al di fuori del chip e col parallelismo minimo necessario per evitare sprechi.

È dunque importante minimizzare la quantità di dati trasportati e la distanza dei trasferimenti: per questa ragione i progettisti si focalizzano sulla costruzione di strutture forward inference-only, come del resto avviene in questo lavoro. Tali strutture permettono la rapida valutazione degli output a partire da un gruppo di input ma non permettono l'apprendimento in tempo reale, che deve invece avvenire a priori mediante altri mezzi (nel caso presente, utilizzando tool come il Deep Learning Toolbox™ di Matlab™). Le strutture digitali sono dunque ideali per l'elaborazione di immagini e quindi ottime per l'implementazione di reti convolutive, tuttavia non vanno bene per strutture che richiedono una maggiore elasticità. Per questo si stanno studiando architetture basate su in-memory computing che eseguono le operazioni di calcolo direttamente all'interno di strutture di memoria in maniera analogica e che si mostrano ben predisposte all'implementazione di reti di tipo fully connected e permettono la costruzione più efficiente di reti neuromorfe, ovvero che si modificano nel tempo [5].

Tuttavia le reti neurali presentano un grosso limite: il loro funzionamento è puramente a "black box", ovvero non è possibile esaminare e modellizzare nel dettaglio i singoli stadi che permettono di giungere alla risoluzione del problema [1] e la costruzione della rete stessa avviene tramite un approccio di tipo trial and error [6].

In breve, si sa come funzionano, ma non si riesce a determinare esattamente come risolvono un determinato problema, a causa della loro grande complessità. Questo è anche uno tra i molteplici motivi per cui è difficile comprendere nel dettaglio il funzionamento dell'intelligenza umana.

Capitolo 2

Struttura su FPGA

Si vuole progettare una rete neurale non ricorrente da implementare su chip FPGA. L'implementazione viene descritta con un linguaggio di descrizione dell'hardware (VHDL); deve essere il più possibile parametrica in modo tale che l'utente che voglia utilizzare il codice per generare la propria versione della rete possa scegliere le sue dimensioni a seconda delle proprie esigenze. L'implementazione deve dunque garantire un parallelismo dei dati variabile, un quantitativo di neuroni arbitrario e la scelta di tipologie di layer differenti. In questo capitolo viene presentata la descrizione dettagliata dell'architettura; per un ulteriore dettaglio si può consultare l'appendice A, dove viene mostrato il codice VHDL impiegato per la realizzazione dell'architettura.

2.1 I Field Programmable Gate Array (FPGA)

I Field Programmable Gate Array (FPGA) sono dei circuiti integrati riprogrammabili con l'ausilio di linguaggi di descrizione dell'hardware come il VHDL o il Verilog. Sono la via di mezzo tra una soluzione costosa come la progettazione di un chip integrato ad hoc e il meno costoso ma anche meno performante impiego di un microcontrollore. Un singolo chip FPGA costa all'utente finale mediamente più di un chip dedicato ma garantisce costi di produzione enormemente inferiori, rendendo questa tecnologia adatta per volumi di produzione piccoli, senza rinunciare del tutto alle prestazioni dell'hardware dedicato.

2.1.1 Composizione

Un FPGA è costituito da una serie di blocchi logici configurabili (Configurable Logic Block, CLB) generalmente composti da alcune Look-up Table (LUT) e da un registro. Queste unità possono venire riprogrammate e combinate in modo da formare strutture logiche più o meno complesse, da semplici multiplexer a dei sommatori o moltiplicatori. I CLB sono ovviamente limitati e il loro numero dipende dalle dimensioni (e quindi dal costo) del chip.

Per garantire delle prestazioni buone e non consumare velocemente tutti i CLB i chip FPGA dispongono di una serie di blocchi integrati dalla funzionalità predefinita. Il numero di questi oggetti e il loro scopo varia a seconda della famiglia di FPGA. Normalmente si trovano delle strutture di memoria programmabili in maniera diversa (come FIFO, RAM e così via), dei moltiplicatori dedicati (molto più veloci di quello che si otterrebbe a partire dai CLB) oppure semplicemente dei Data Signal Processor (DSP) di varia natura.

2.1.2 Prestazioni

Come accennato queste strutture sono a metà tra i chip dedicati e i microcontrollori. Sono più veloci dei microcontrollori perché permettono la progettazione di acceleratori hardware dedicati e consentono la parallelizzazione e l'inserimento di livelli di pipeline per aumentare la frequenza di funzionamento e incrementare il throughput. Tuttavia restano più lenti di un chip dedicato perché l'hardware non è realmente fatto su misura: le interconnessioni sono lunghe e le capacità del circuito sono elevate. Questo peggiora anche i consumi e la dissipazione di potenza.

2.1.3 Dispositivi in commercio

I principali produttori di FPGA sono Xilinx e Altera, ora comprata da Intel, rivali storiche nel settore. Ciascuna delle due case produttrici fornisce un'ampia scelta di famiglie di chip e dei tool di sviluppo per il testbench e il testbed dei progetti, in versioni gratuite o a pagamento. In questo lavoro un marchio vale l'altro; si sceglie di progettare su chip Intel Cyclone 10 LP [7], recente, generico come funzionalità e pubblicizzato come low power, come indicato dal nome stesso. Il software di sviluppo è Intel Quartus Prime Starter Edition (versione gratuita di Intel Quartus Prime Pro) che permette la compilazione del VHDL, la compilazione di una descrizione gate-level, l'analisi del timing e di potenza nonché ovviamente il fitting del chip FPGA.

Il testbench viene effettuato utilizzando Altera-Modelsim sia a livello RTL che a livello gate-level.

2.2 Struttura del layer

Il circuito nel suo complesso è costituito da una serie di macrostrutture modulari che si identificano come i singoli layer della rete neurale. Ogni layer presenta la stessa struttura generale; le differenze tra un layer e l'altro dipendono dal tipo di layer e dalle risorse che l'utente sceglie di impiegare. Per rendere efficiente l'implementazione su FPGA occorre sfruttare al meglio i blocchi integrati in essa contenuti: per il progetto si utilizza come punto di riferimento il chip Intel Cyclone 10 LP. Si hanno a disposizione delle strutture di memoria programmabili e dei moltiplicatori; per ciascun layer si hanno una serie di strutture impiegate:

- due memorie RAM sincrone dual-port (con un ingresso e un'uscita per i dati) vengono impiegate come interfaccia, una di ingresso e l'altra di uscita del layer;
- l'unità principale di elaborazione è una struttura Multiply and Accumulate (MAC) che utilizza uno o più moltiplicatori integrati a 9 bit;
- ad ogni MAC viene associata una memoria RAM sincrona dual-port per la memorizzazione dei pesi.

2.2.1 Il Multiply and Accumulate (MAC)

Il MAC è una struttura che riceve due input in ingresso, li moltiplica tra di loro e li somma ad un valore accumulatore. Il valore ottenuto sostituisce il precedente valore accumulatore e viene inviato in uscita. Nel caso delle reti neurali, il primo valore è un input in ingresso nel layer mentre il secondo è un peso per il quale quest'ultimo deve essere moltiplicato. Il MAC non effettua dunque nient'altro che una somma pesata, il che è esattamente quello che fa un neurone (il cui comportamento a sua volta non è altro che quello di un filtro digitale). Nel listato 2.1 viene mostrata la descrizione comportamentale del MAC in C.

```
1 //a -> primo input
2 //b -> secondo input
3 //out -> valore accumulatore
4
5 out = out + a * b;
```

Listing 2.1: Descrizione in C del comportamento del MAC.

Come fare tuttavia ad inizializzare il valore accumulatore? Le opzioni sono due: utilizzare un segnale di reset per azzerarlo oppure impiegare un segnale di bypass per sostituirlo con un valore esterno (nel caso presente, il secondo input). Si opta per la seconda opzione perché permette di inserire immediatamente il valore di bias. Nel listato 2.2 viene mostrata la descrizione del MAC in C con aggiunta di bypass.

```
//a -> primo input
//b -> secondo input
//out -> valore accumulatore

if(bypass)
{
    out = b;
}
else
{
    out = out + a * b;
}
```

Listing 2.2: Descrizione in C del comportamento del MAC con l'aggiunta del segnale di bypass.

L'implementazione a livello circuitale si può vedere in figura 2.1. Il segnale `enable` viene impiegato per abilitare i registri di input e il registro accumulatore, in modo da evitare un'inutile dissipazione di potenza se in disuso. Il segnale `bypass` pilota un multiplexer che seleziona l'ingresso dell'accumulatore. Sono presenti alcuni livelli di pipeline necessari a massimizzare la frequenza di lavoro della macchina. Per convenzione viene anche inserito un segnale di reset che però non rispetta il timing imposto dalla pipeline (viene resettato tutto in un colpo).

2.2.2 Struttura di base dei layer fully connected o convolutivi: gruppo `neurons`

A livello generico, un layer deve essere in grado di ricevere degli input, effettuare delle somme pesate con l'aggiunta di un valore di bias (prestando attenzione alla corretta associazione tra input e peso), applicare una funzione di attivazione al risultato e inviare gli output in uscita nel corretto ordine. Per migliorare le prestazioni occorre poter filtrare più valori contemporaneamente. Nel listato 2.3 viene mostrata

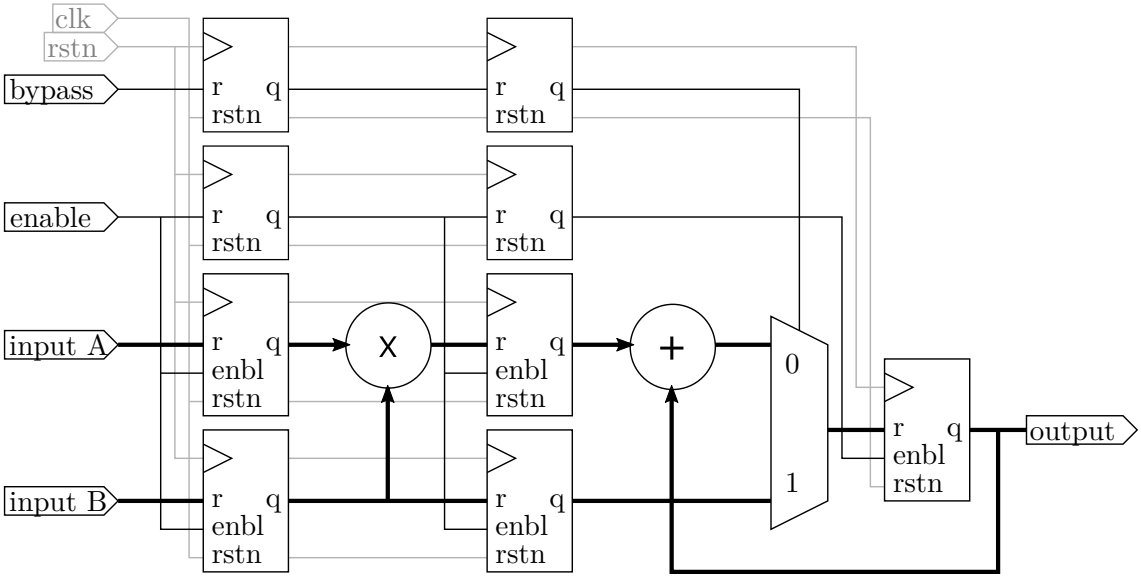


Figura 2.1: Implementazione circuitale del MAC.

la descrizione in C della struttura base del layer: il codice è stato scritto per mettere in evidenza che un input può essere impiegato per svolgere più operazioni in parallelo.

```
//inputs[] -> input inviati ai neuroni
//weights[][] -> pesi assegnati ai neuroni
//bias[] -> bias assegnato ai neuroni
//activation_function(x) -> funzione di attivazione

//i bias vengono caricati negli accumulatori in parallelo
for(s = 0; s < S; s++) //S e' il numero di neuroni o filtri
{
        acc[s] = bias[s];
}

for(n = 0; n < N; n++) //N e' il numero di input
{
    //un input viene pesato e sommato a diversi accumulatori in parallelo
    for(s = 0; s < S; s++)
    {
        acc[s] = acc[s] + inputs[n] * weights[n][s];
    }
}
//dopo che tutti gli input sono stati elaborati i valori vengono inoltrati in
    uscita; questa operazione non puo' avvenire in parallelo perche' la memoria in
    uscita ha un solo ingresso
for(s = 0; s < S; s++)
{
    //ad ogni neurone e' associato un output
```

```
24      out[s] = activation_function(acc[n]);
25  }
```

Listing 2.3: Descrizione in C del comportamento generico di un layer.

La somma pesata effettuata dal neurone viene eseguita da un MAC. Questi prima salva il bias (memorizzato nella stessa memoria dei pesi) utilizzando il segnale di bypass, dopodiché somma gli input moltiplicati per i rispettivi pesi uno dopo l'altro. Un singolo MAC può fare le veci di più neuroni e allo stesso tempo si possono utilizzare più MAC in parallelo per accelerare il processo di calcolo. Il modo in cui la memoria di input e di output sono collegate alla struttura base della rete, che comprende i MAC, le memorie dei pesi e la funzione di attivazione, si può vedere a livello generico nella figura 2.2.

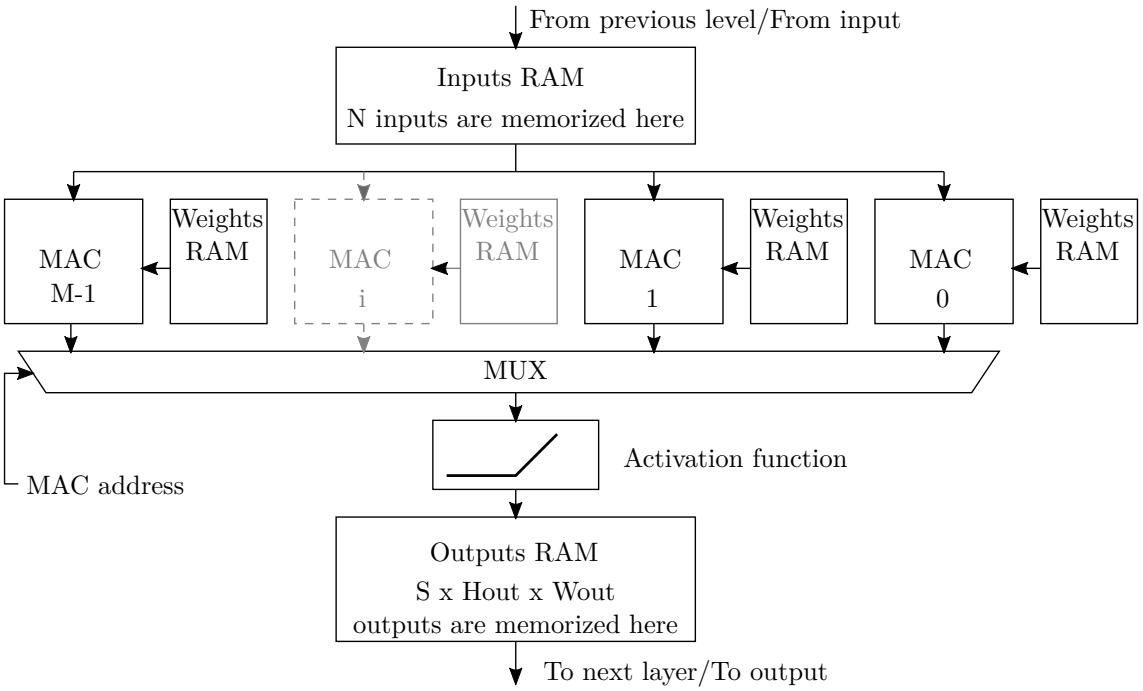


Figura 2.2: Struttura generica del layer.

La memoria dei pesi e il MAC: `MAC-group`

Come viene mostrato nella figura 2.2 ad ogni MAC viene associata una memoria RAM nella quale vengono salvati unicamente i bias e i pesi dei neuroni ad esso associati. L'utente può scegliere arbitrariamente il numero di neuroni e di MAC assegnati a ciascun layer. Si indica con S_j e con M_j rispettivamente il numero di

neuroni e di MAC del j-esimo layer. Il numero di neuroni assegnati a ciascun MAC è allora

$$\text{neurons per MAC} = \left\lceil \frac{S_j}{M_j} \right\rceil \tag{2.1}$$

La struttura costituita da MAC, memoria dei pesi e registro finale viene chiamata `MAC group` e si può osservare in figura 2.3; i due registri di pipeline del segnale `write_output` servono per ritardare la scrittura del risultato nel registro finale dello stesso numero di colpi di clock dei segnali che attraversano il MAC.

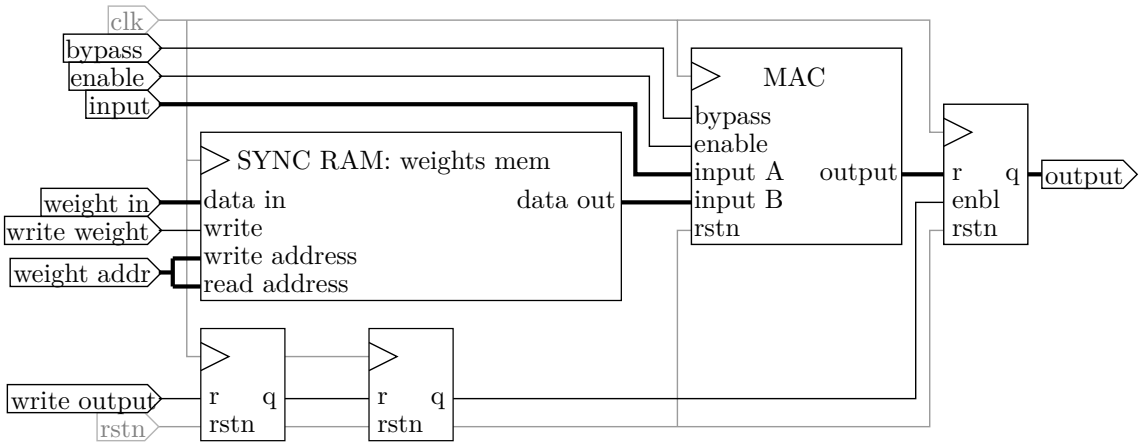


Figura 2.3: Struttura del MAC-group.

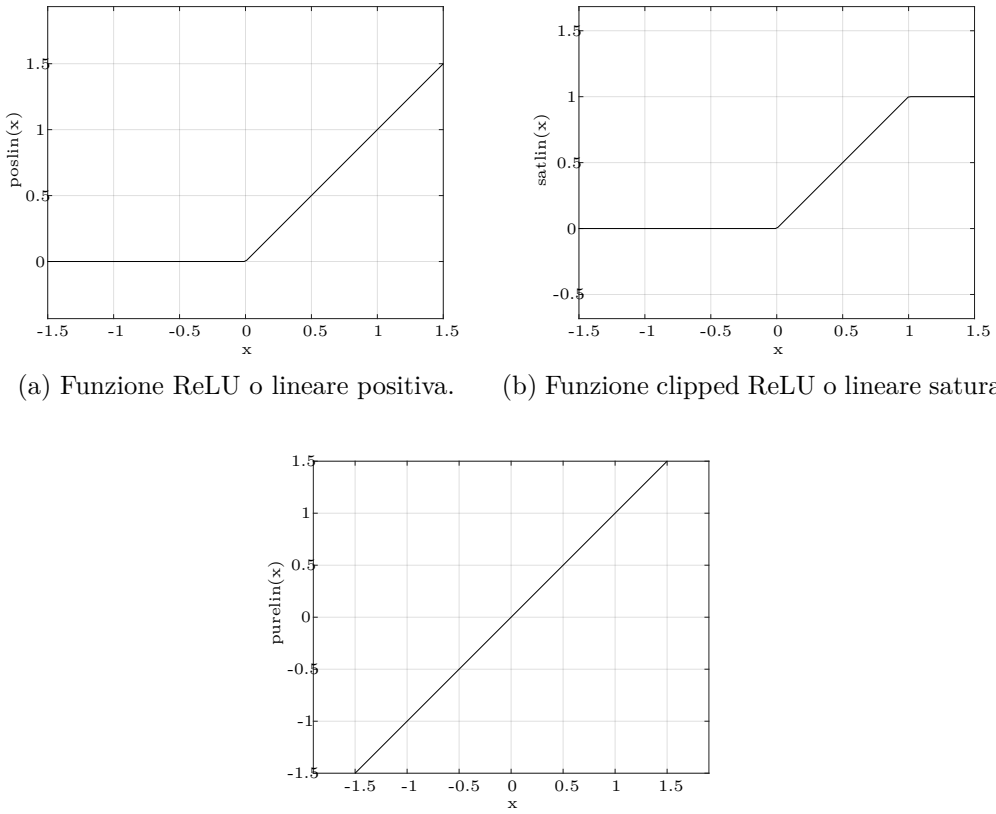
Funzioni di attivazione

Per semplificare la struttura hardware occorre selezionare delle funzioni di attivazione facilmente implementabili. In questo lavoro vengono utilizzati la funzione Rectified Linear Unit (ReLU o lineare positiva, figura 2.4a, Eq. 2.2), la clipped Rectified Linear Unit (o lineare saturata, figura 2.4b, Eq. 2.3) e la lineare pura (figura 2.4c, Eq. 2.4). Tutte e tre le funzioni possono venire implementate facilmente ed efficientemente con del codice VHDL comportamentale. Il codice è strutturato in modo che possano ancora venire aggiunte anche funzioni di attivazione più complesse, come la sigmoid; questo potrebbe ridurre la frequenza di funzionamento e quindi potrebbe essere necessaria l'introduzione di ulteriori livelli di pipeline.

$$\text{poslin}(x) = \begin{cases} 0 & \text{per } x \leq 0 \\ x & \text{per } x > 0 \end{cases} \tag{2.2}$$

$$\text{satlin}(x) = \begin{cases} 0 & \text{per } x \leq 0 \\ x & \text{per } 0 < x < 1 \\ 1 & \text{per } x \geq 1 \end{cases} \tag{2.3}$$

$$\text{purelin}(x) = x \tag{2.4}$$



(a) Funzione ReLU o lineare positiva.

(b) Funzione clipped ReLU o lineare saturata.

(c) Funzione lineare pura.

Figura 2.4: Funzioni di attivazione implementate in hardware.

Gestione dei MAC-group in parallelo

In figura 2.5 possiamo vedere l'architettura definitiva utilizzata sia dai layer fully connected che dai layer convolutivi a cui daremo il nome `neurons`; gli input passano

attraverso M_j MAC-group in parallelo che elaboreranno i dati con S_j diversi filtri. Per completare il calcolo di tutti gli output è necessario compiere più iterazioni durante le quali tutti gli input che vanno filtrati vengono pesati e sommati; ad ogni iterazione il valore dei pesi cambia perché si esegue la valutazione di diversi neuroni. Una fila di registri serve da pipeline per incrementare la frequenza del circuito. Anche prima del blocco che implementa la funzione di attivazione occorre un registro di pipeline a causa del grosso fan-in del multiplexer che lo precede.

Notare che se S_j non è esattamente un multiplo di M_j, abbiamo un certo numero di MAC a cui viene assegnato un neurone in meno rispetto al valore indicato nell'equazione 2.1; si tiene conto di questo durante la sintesi del circuito onde evitare consumi energetici superflui. Il segnale `last_row` che si può vedere in figura 2.5 viene attivato durante l'ultima iterazione di elaborazione e disattiva i MAC che non servono; il numero dei MAC che vengono disattivati è

$$\text{MACs without last neuron} = M_j - (S_j \mod M_j) \tag{2.5}$$

Per capire meglio si fa un esempio. Se si avessero 12 neuroni ($S_j = 12$) che vengono gestiti da 5 MAC ($M_j = 5$), allora ad ogni MAC secondo l'equazione 2.1 dovrebbero venire assegnati 3 neuroni. Tuttavia, poiché 12 non è un multiplo esatto di 5, durante l'ultima iterazione 3 MAC vengono disattivati secondo l'equazione 2.5 e solo 2 rimangono in funzione. Per questo motivo durante la prima iterazione vengono elaborati gli output di 5 neuroni, durante la seconda altri 5 e alla terza (l'ultima) solamente gli ultimi 2. Avremo quindi M_j MAC che in un tempo pari a $\lceil S_j/M_j \rceil$ iterazioni generano l'output di S_j neuroni diversi e durante l'ultima iterazione non tutti i MAC sono in funzione.

A ciascun MAC è assegnata una memoria contenente il bias e i pesi dei neuroni ad esso assegnati. È importante osservare il modo in cui viene generato l'indirizzo dei pesi associato ugualmente a tutti i MAC-group. Le memorie dei pesi contengono i valori in un ordine preciso che si può osservare in figura 2.6: il bias viene seguito da N_j valori che rappresentano i pesi; il primo MAC conterrà i valori associati al primo neurone, il secondo quelli del secondo e così via. Raggiunto l'ultimo MAC si ricomincia dal primo inserendo dopo i valori già salvati il bias e i pesi associati ad un altro neurone, sempre in ordine, e si prosegue così.

Il valore `weight_addr` in ingresso indica quale peso si vuole utilizzare mentre il valore `iterat_addr` indica quali neuroni i MAC-group devono utilizzare: se i MAC-group sono 3 ad esempio, `iterat_addr` = 0 indica che verranno impiegati i neuroni 0, 1 e 2, mentre se `iterat_addr` = 1 verranno impiegati i neuroni 3, 4 e 5 e così di seguito. A questo si aggiunge ancora il segnale `weight1_bias0` che indica se

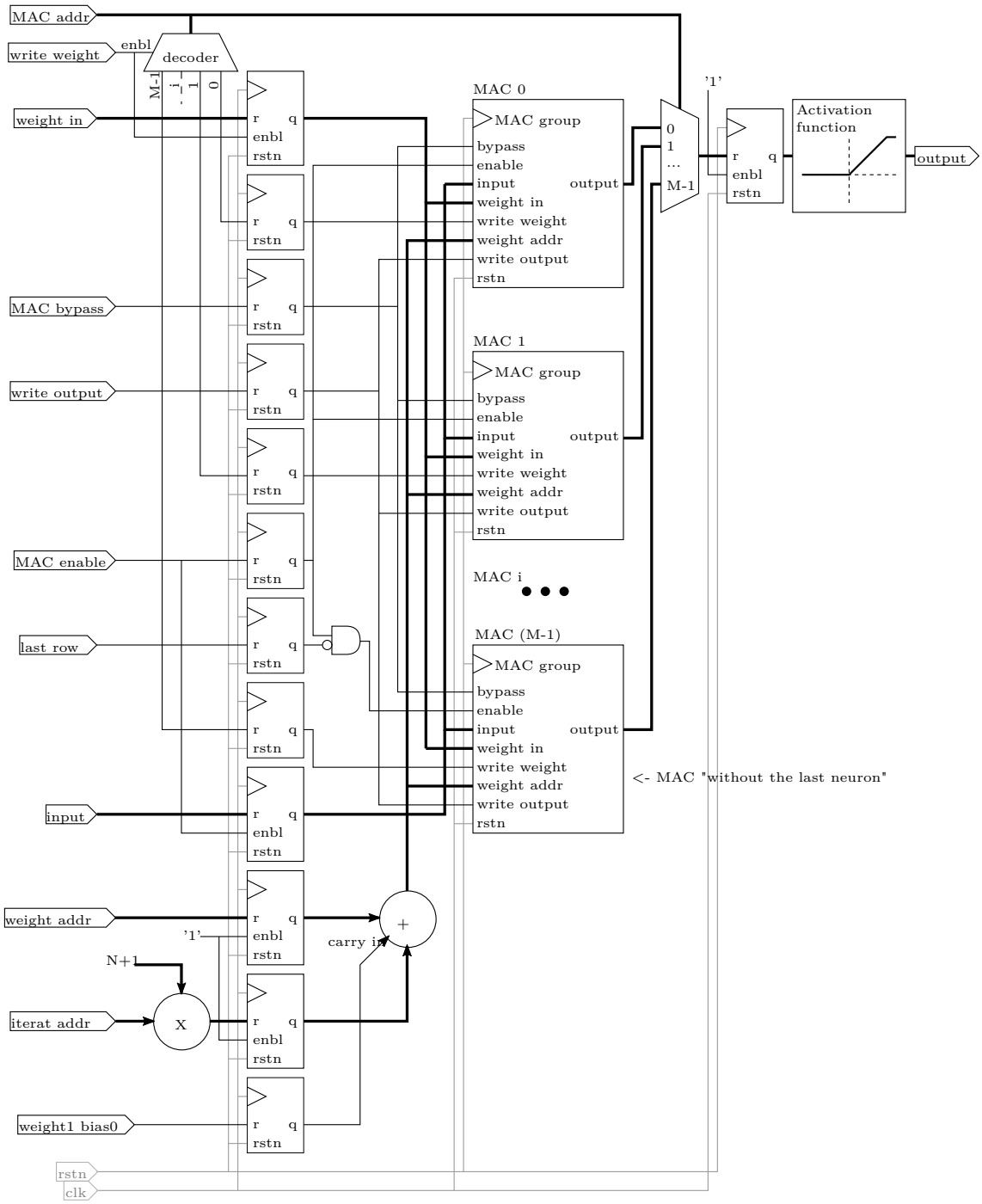


Figura 2.5: Struttura base di un layer fully connected o convolutivo denominata `neurons`.

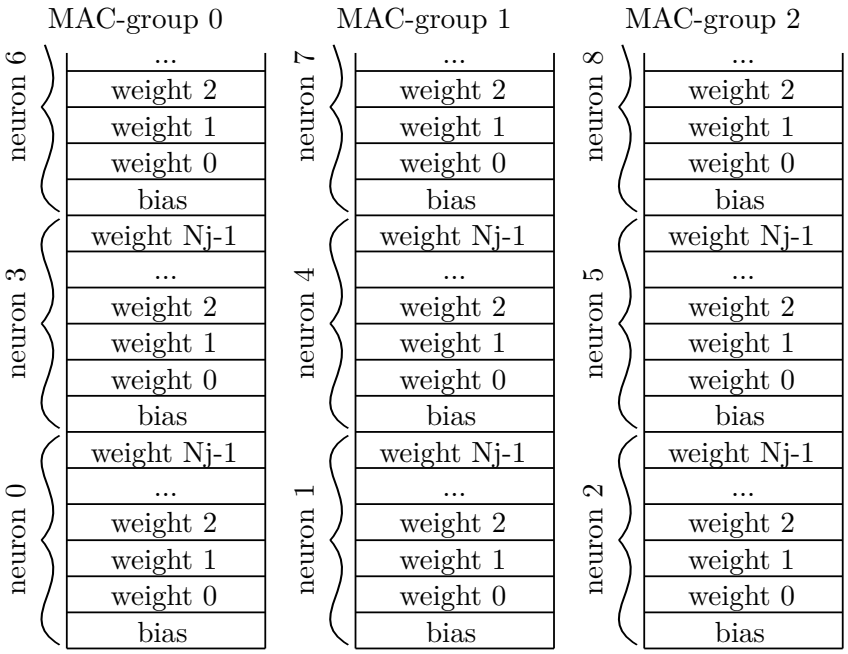


Figura 2.6: Disposizione dei pesi e dei bias all'interno delle memorie contenute nei vari MAC-group; nel caso preso in considerazione, si hanno 3 MAC-group.

stiamo utilizzando un bias o un peso; se stiamo utilizzando un bias `weight_addr` dovrà essere 0. Notare che il bias viene utilizzato per inizializzare l'accumulatore dei MAC, per cui occorre anche abilitare il segnale di bypass. Il calcolo dell'indirizzo della memoria dei pesi si può riassumere con la seguente formula

$$\texttt{weight_mem_addr} = \texttt{input_addr} + \texttt{neuron_addr} \cdot (N_j + 1) + \texttt{weight1_bias0} \quad (2.6)$$

2.2.3 Struttura di base dei layer max-pooling

I layer di tipo max-pooling non devono eseguire somme pesate, bensì devono valutare il massimo tra una serie di valori di input. Per questa ragione non viene impiegata la struttura `neurons` descritta nel paragrafo 2.2.2. Viene al suo posto utilizzata invece una struttura che chiamiamo `max`, che si può vedere in figura 2.7.

L'intera struttura viene inizializzata attraverso il segnale `init` che consente la memorizzazione nel registro accumulatore il minimo valore rappresentabile, ovvero un '1' per il bit di segno seguito da soli '0'. Il segnale `init` impedisce inoltre al segnale `input` di attraversare i registri di pipeline per risparmiare energia. Se `enable`

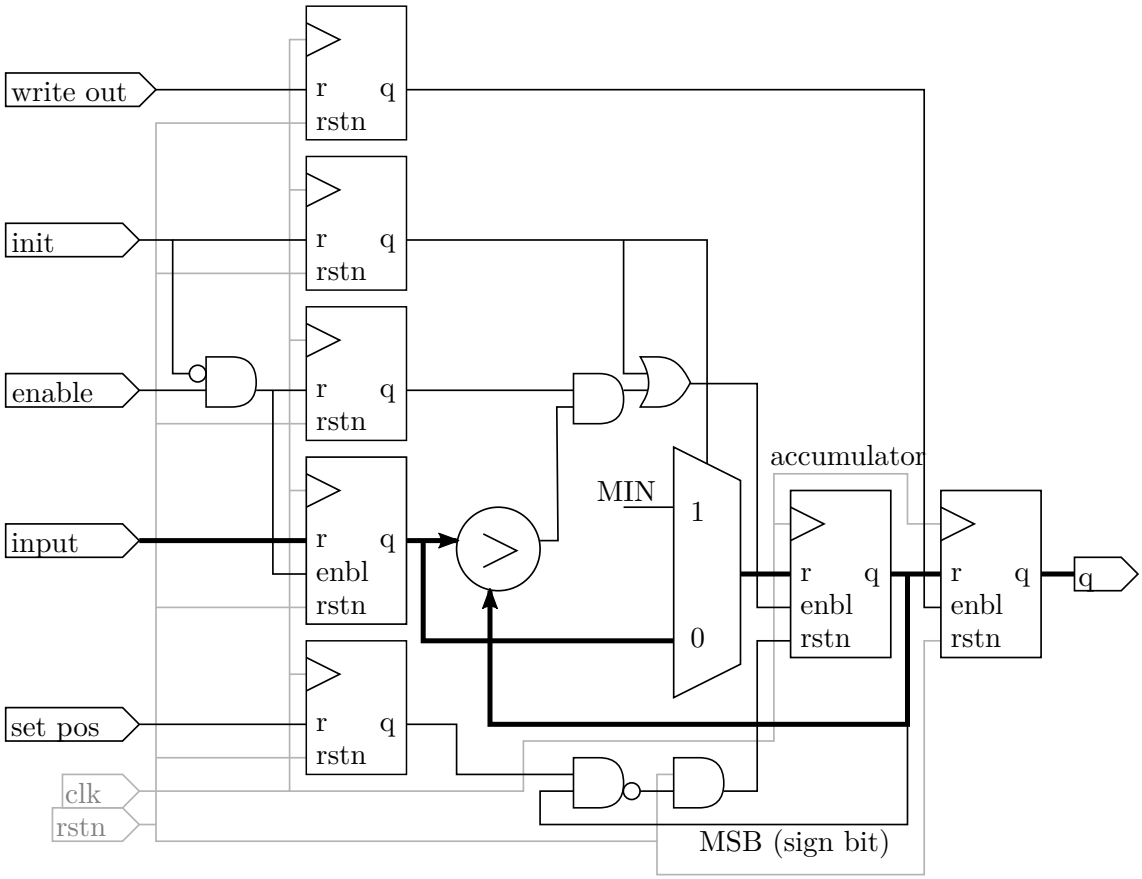


Figura 2.7: Struttura `max` per la valutazione del valore massimo tra una serie di valori di input.

è attivato, il valore `input` viene confrontato col valore salvato nel registro accumulatore; se risulta maggiore, viene salvato, altrimenti non viene fatto nulla. Il segnale `set_pos` dice alla macchina di verificare se il segnale accumulatore sia negativo e, in tal caso, salva il valore 0 nell'accumulatore. Infine il segnale `write_out` consente il trasferimento del valore accumulatore all'interno del registro di uscita. La descrizione in codice C di questo componente viene mostrata nel listato 2.4, per maggiore comprensione.

```
//maximum -> valore accumulatore

if(set_pos && maximum < 0)
{
    maximum = 0;
}
else if(init)
```

```
8  {
9      maximum = 1 << (n_bits-1); //set minimum value
10 }
11 else if(enbl && input > maximum)
12 {
13     maximum = input;
14 }
```

Listing 2.4: Descrizione in C del comportamento del blocco `max`.

Non è possibile parallelizzare questa architettura in quanto nei layer max-pooling un singolo gruppo di input genera unicamente un output. Per le reti fully connected o convolutive lo stesso gruppo di input invece può generare diversi output e questi possono venire calcolati in parallelo. Per questa ragione i layer max-pooling generano un solo output per volta; il numero ridotto di operazioni che devono eseguire tuttavia non li rende più lenti di un layer convolutivo totalmente parallelizzato.

2.2.4 Stadi funzionali del layer

Possiamo dividere il layer in 3 stadi funzionali:

- durante il primo, che verrà indicato come *load inputs*, gli output del layer precedente - oppure gli input della rete se si tratta del layer di ingresso - vengono caricati nella memoria di ingresso;

- durante il secondo, che verrà indicato come *elaborate*, gli input passano attraverso la struttura di elaborazione (MAC per le reti fully connected o convolutive o comparatori per il max-pooling) e vengono salvati in una serie di registri di uscita; questa fase viene ripetuta più volte fino a che non vengono elaborati gli output di tutti i neuroni;

- durante il terzo, che verrà indicato come *save outputs*, i valori salvati nei registri di uscita vengono trasformati dalla funzione di attivazione (se va applicata) e vengono salvati nella memoria di output; questa operazione avviene a più riprese perché occorre aspettare che ci siano degli output validi pronti per essere salvati.

Ciascuno dei tre stadi viene gestito da una propria macchina a stati. La macchina deve assicurarsi di effettuare ciascuna delle operazioni solamente quando i dati che vengono forniti sono validi e quando lo stadio successivo sia pronto a riceverne. Per questa ragione vengono impiegati una serie di segnali flag.

Comunicazione tra le unità di controllo del layer

Le tre macchine a stati, che gestiscono altrettanti stadi all'interno del layer, comunicano tra di loro attraverso dei valori flag che possono venire attivati o disattivati. Ognuno di questi valori è memorizzato e gestito tramite una struttura set-reset che si può vedere in figura 2.8: un valore alto del segnale set attiva la flag, un valore basso del segnale reset la disattiva. Qualora set sia alto e reset sia basso nell'istante di campionamento, prevale set. Si definiscono diverse flag, ciascuna necessaria per la comunicazione tra le unità di controllo:

- `input_ready` viene asserito dalla macchina a stati *load inputs* quando un numero sufficiente di dati nella memoria di ingresso sono pronti; viene resettato da *elaborate* quando gli input non servono più;

- `data_ready` viene asserito dalla macchina a stati *elaborate* quando una porzione di dati è stata elaborata ed è pronta; viene resettato da *save outputs* quando tutti i valori forniti sono stati memorizzati nella memoria di uscita;

- `elab_done` viene asserito dalla macchina a stati *elaborate* quando tutti i valori di output sono stati calcolati; viene resettato da *save outputs* quando tutti gli output sono stati memorizzati nella memoria di uscita;

- `output_ready` viene asserito dalla macchina a stati *save outputs* quando comincia a salvare l'ultima porzione dei valori di output e quindi lo stadio successivo può cominciare già a prelevare i dati; viene resettato da *load inputs* del livello successivo quando questi comincia a trasferire i valori nella successiva memoria di ingresso e quindi i dati appena spostati possono essere sostituiti.

In figura 2.9 è mostrato graficamente come le varie flag collegano le diverse unità di controllo.

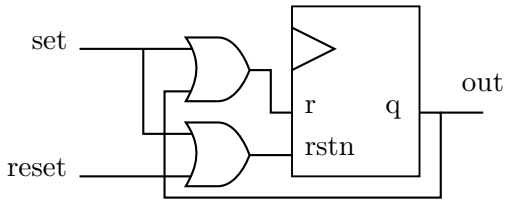


Figura 2.8: Struttura set-reset impiegata per le flag utili per gli handshake. `rstn` è sincrono.

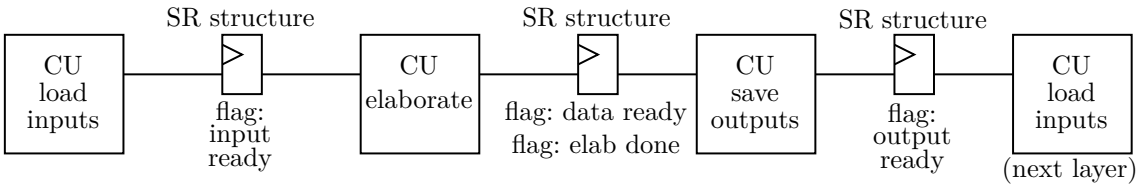


Figura 2.9: Connessioni per la comunicazione tra le unità di controllo.

2.2.5 Sistema di controllo dei layer fully connected

Il layer fully connected deve far passare per ciascun neurone tutti gli input. Per questo motivo dal punto di vista del controllo il layer fully connected è il più semplice da gestire: non deve far altro che prelevare gli input in ordine e inviarli ai MAC, tante volte qual è il rapporto $\lceil S_j/M_j \rceil$. L'implementazione del datapath del layer si trova in figura 2.10. L'associazione del datapath alle varie macchine a stati si trova in figura 2.11

Primo stadio del layer: *load inputs*

Lo stadio *load inputs* del layer si occupa di memorizzare gli input all'interno della memoria RAM di ingresso. Il datapath dello stadio in questione è visibile in figura 2.12, che è un'estratto dello schema completo del datapath del circuito che si trova in figura 2.10. Il contatore che viene indicato come `IN` genera l'indirizzo di scrittura della memoria di ingresso (indicata come `memA`) e quello di lettura della memoria di uscita del layer precedente. La macchina a stati gestisce il segnale di scrittura della memoria e controlla il contatore. L'Algorithmic State Machine chart (ASM chart) dell'unità di controllo si può visionare in figura 2.13.

La macchina attende che ci sia almeno il primo dato pronto da caricare, ovvero che `output_ready` sia asserito, e che i dati correntemente memorizzati non stiano venendo utilizzati, ovvero che `input_ready` non sia asserito. Dopodiché resetta il valore di `output_ready` e asserisce `input_ready`. Infine trasferisce i dati dalla memoria di uscita del layer precedente alla memoria di ingresso di quello attuale fino a che il contatore non arriva al valore terminale, che è pari al numero di input che la macchina riceve (N_j). Occorre notare che il valore delle flag arriva alla precedente e successiva macchina a stati dopo un colpo di clock e le azioni di queste perturberanno il datapath dopo un ulteriore ritardo. In questo modo quando il valore basso di `output_ready` viene recepito dal layer precedente questi può assumere che i primi valori contenuti nella propria memoria di uscita siano già stati trasferiti e per questo motivo può cominciare già a sostituire i valori che non servono più. Lo stesso vale

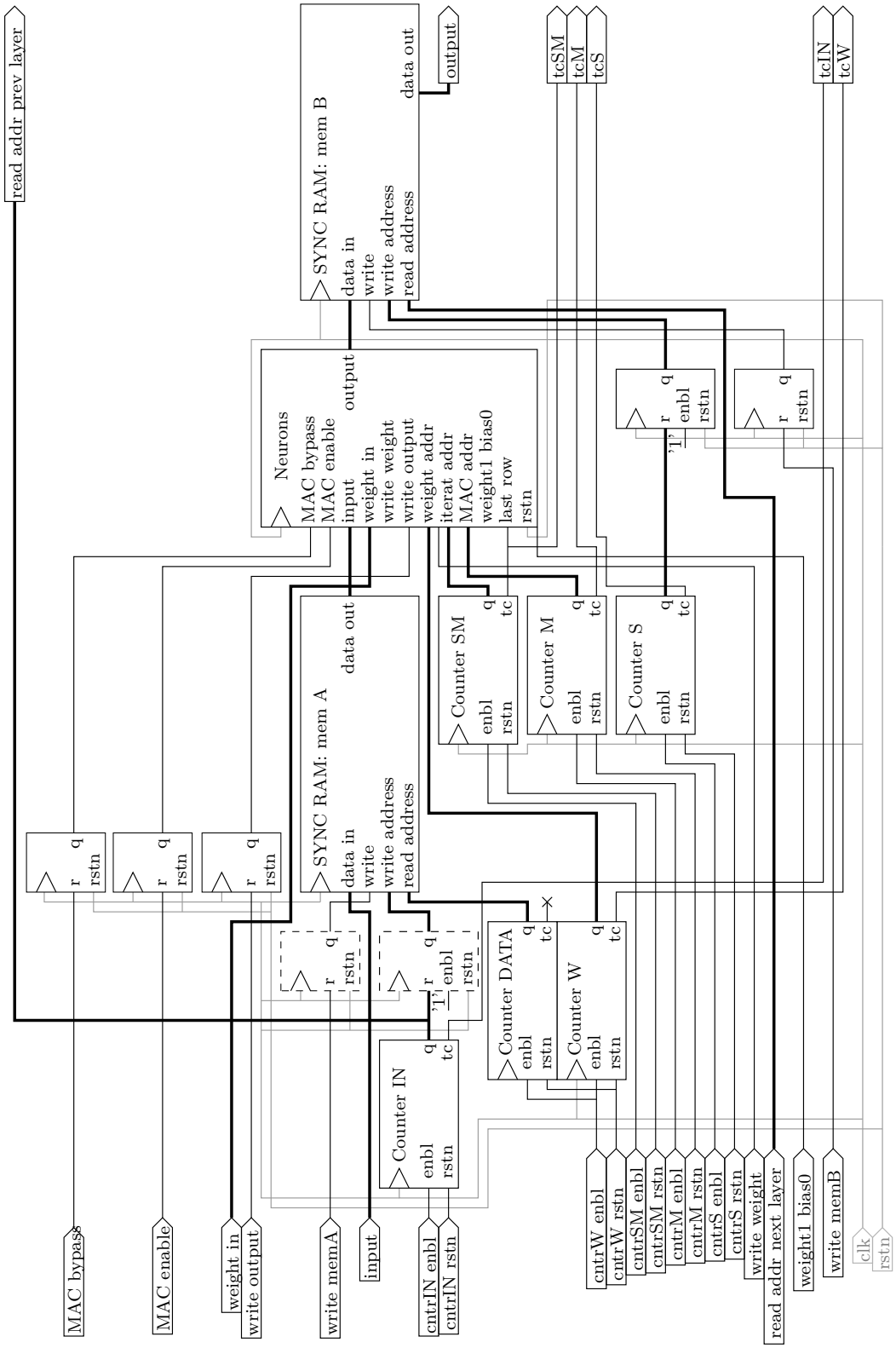


Figura 2.10: Implementazione del datapath del layer fully connected.

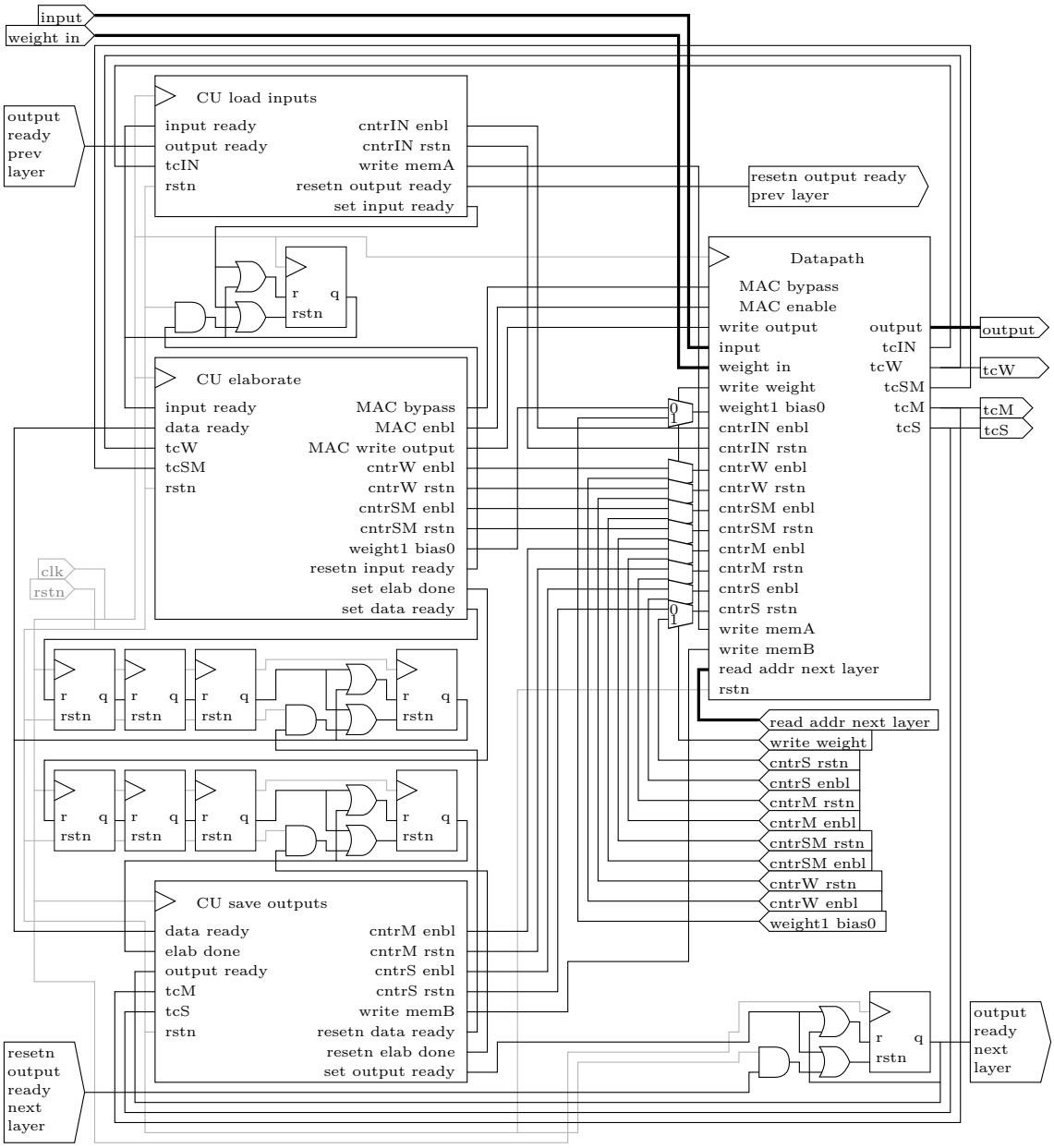


Figura 2.11: Implementazione completa del layer con macchine a stati. Notare che i segnali di controllo dei contatori possono anche provenire dall'esterno e un multiplexer li seleziona attraverso il segnale `write weight`; questo serve per caricare i pesi come viene descritto nel paragrafo 2.3.3. I registri aggiuntivi sono registri di pipeline come descritto nel paragrafo 2.2.7.

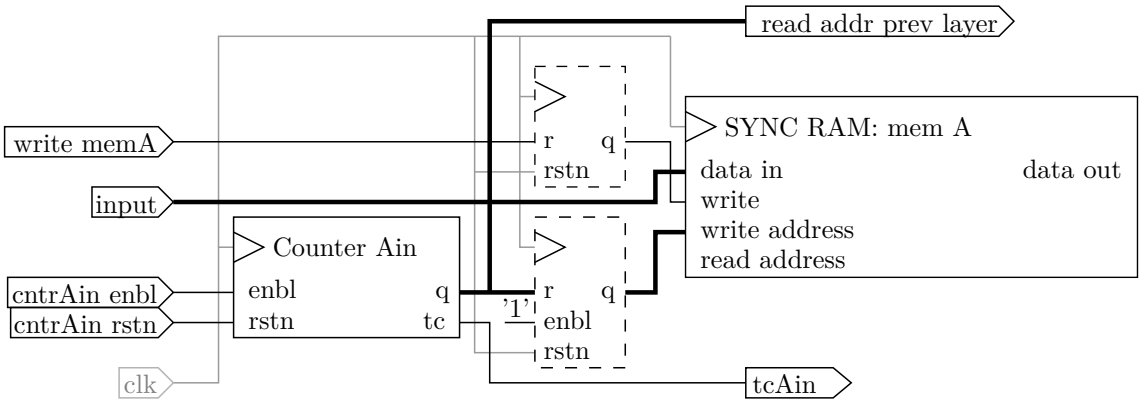


Figura 2.12: Porzione di circuito che rappresenta *load inputs*. I registri tratteggiati vengono inseriti solamente se il layer in questione non è il primo.

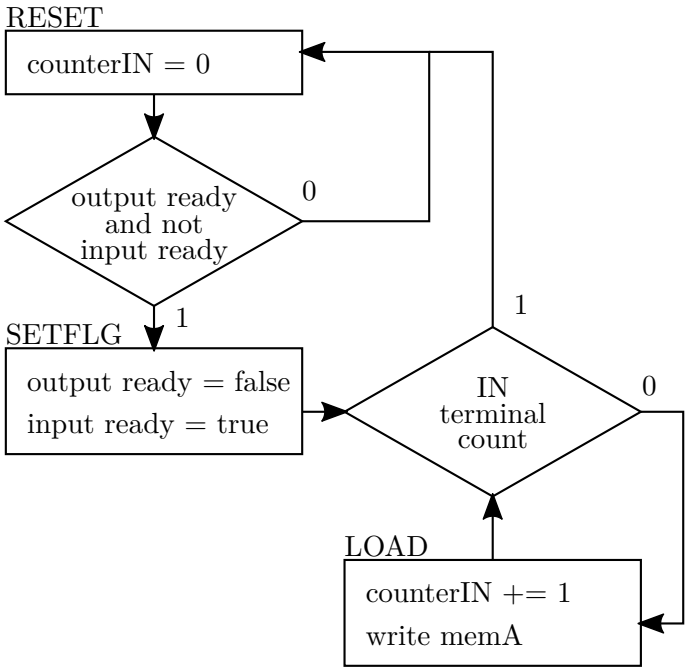


Figura 2.13: ASM chart dell'unità di controllo *load inputs*.

per l'altra flag: quando il valore alto di `input_ready` viene ricevuto dallo stadio *elaborate* questi sa che può cominciare ad effettuare le operazioni di calcolo e dopo un ulteriore ritardo dovuto all'inizializzazione degli accumulatori con i valori di bias inizia a prelevare i valori di input a partire dai primi che sono appena stati caricati. Il risultato è che mentre i valori vengono trasferiti uno per volta questi possono venire immediatamente sostituiti e possono essere già utilizzati mano a mano che

vengono caricati, aumentando il throughput della rete neurale.

La struttura del datapath varia leggermente in due differenti casistiche: il layer può essere il primo, con gli input che vengono inseriti dall'esterno, oppure semplicemente può essere un layer qualsiasi che preleva i dati da un layer precedente. Nel caso in cui il layer in questione sia quello di ingresso i registri tratteggiati in figura 2.12 non vengono inseriti. Si rimanda al paragrafo 2.3.1 per i dettagli inerenti la gestione degli input della rete neurale.

Qualora invece il layer in questione non sia il primo, i registri in figura 2.12 vengono inseriti. Il motivo è il comportamento delle memorie, che dispongono di un'uscita sincrona: il dato corrispondente ad un dato indirizzo viene posto in uscita il colpo di clock successivo rispetto a quando l'indirizzo stesso viene asserito. Per questa ragione si può modellizzare la memoria come se fosse uno stadio di pipeline (inserisce un ritardo). Poiché dunque la memoria di uscita del layer precedente inserisce un ritardo, i segnali che viaggiano parallelamente al suo valore di uscita (`write memA` e il valore di uscita del contatore) necessitano di venire ritardati per mantenere la coerenza dei dati.

Secondo stadio del layer: *elaborate*

Per questo stadio ci si può riferire alla figura 2.10 che mostra come i contatori e le memorie siano connesse al blocco `neurons`, la cui struttura si può osservare tornando alla figura 2.5. In figura 2.14 si trova l'ASM chart della macchina stati riferita a questo stadio. In questo stadio non viene ancora applicata la funzione di attivazione.

Lo stadio *elaborate* si occupa dell'elaborazione delle somme pesate che verranno salvate nei registri a valle dei MAC, all'interno dei MAC-group. Trattandosi di un layer fully connected ciascun neurone deve ricevere tutti gli input. Due contatori regolano il funzionamento di questa parte del layer: il contatore `DATA` genera l'indirizzo di uscita della memoria `memA` che invia i valori ai MAC mentre il contatore `SM` indica quante volte sono già stati inoltrati i valori di ingresso o in altri termini l'indice dell'iterazione di elaborazione corrente. In pratica, ogni volta che tutti gli N_j input vengono fatti passare attraverso i MAC e che quindi sono stati calcolati gli output di M_j filtri, il valore del contatore `SM` cresce; il suo valore massimo è $\lceil S_j/M_j \rceil$. Il contatore `W` viene invece utilizzato per indirizzare la memoria dei pesi all'interno dei `MAC-group`.

Lo stadio comincia a lavorare non appena viene ricevuto il segnale `input_ready` che indica la presenza di dati validi nella memoria di ingresso. Sfruttando il segnale

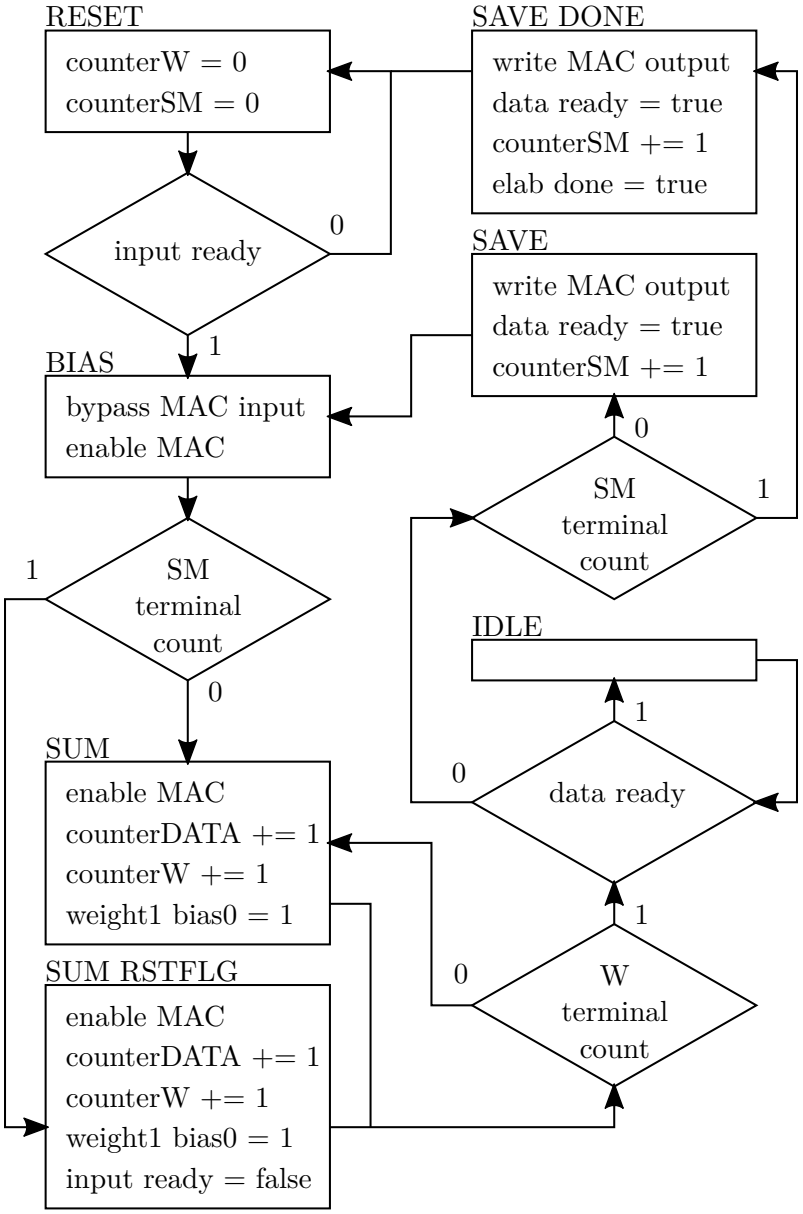


Figura 2.14: ASM chart dell'unità di controllo *elaborate*.

di bypass per prima cosa si inizializza l'accumulatore dei MAC con il valore di bias. Dopodiché si sommano uno per volta gli input pesati; volta per volta il contatore `W` e il contatore `DATA` vengono incrementati. Qualora si stia effettuando l'ultima iterazione (`SM terminal count`), viene resettato il segnale `input_ready` per segnalare allo stadio *load inputs* che può cominciare a caricare dei nuovi input (infatti mentre i vecchi input vengono utilizzati per l'ultima volta si può già cominciare a caricarne

di nuovi). Quando tutti gli input vengono sommati la macchina attende (stato `IDLE`) finché i registri in uscita del MAC-group non sono liberi, il che viene decretato dallo stadio *save outputs* quando resetta il valore di `data_ready`. Successivamente i dati vengono inoltrati nei registri a valle dei MAC-group, cresce il conteggio dei loop (il contatore `SM`) e viene asserito `data_ready`. La macchina può ricominciare subito perché i dati possono venire salvati temporaneamente negli accumulatori dei MAC nonostante i registri finali dei MAC-group siano occupati. Qualora si sia effettuata l'ultima iterazione, la macchina a stati asserisce anche il segnale `elab_done` che indica allo stadio successivo che tutti gli output per quel gruppo di input sono stati calcolati e ritorna allo stato `RESET`.

Terzo stadio del layer: *save outputs*

L'ultimo stadio del layer si occupa di salvare i dati generati dai MAC nella memoria di uscita. Durante questa operazione i dati passano attraverso il blocco che applica la funzione di attivazione, se presente. Anche per questo stadio ci si può riferire alla figura 2.10, nel particolare per la connessione della memoria di uscita al blocco `neurons`, e alla figura 2.5 per quanto riguarda il dettaglio sulla selezione dell'output da salvare e il passaggio attraverso il blocco della funzione di attivazione. In figura 2.15 si trova l'ASM chart della macchina a stati.

Lo stadio *elaborate* genera M_j output per volta che lo stadio *save outputs* salva in serie ad ogni iterazione nella memoria finale finché non ha salvato S_j valori totali; per questa ragione utilizza un contatore `M` che conta gli output dei MAC che sono stati salvati e un contatore `S` che invece conta il numero di output complessivi che sono stati salvati; il contatore `S` è importante perché non è detto che S_j sia un multiplo esatto di M_j. Inizialmente la macchina a stati controlla che ci siano dei dati pronti in ingresso (`data_ready`) e che i valori nella memoria in uscita siano già stati trasferiti nella memoria di ingresso del layer successivo oppure in uscita alla rete (`output_ready`). I dati cominciano a venire salvati passando attraverso il blocco che applica la funzione di attivazione e i contatori di volta in volta vengono incrementati; se lo stadio *elaborate* informa che ha finito di calcolare gli ultimi output (`elab_done` viene asserito) allora la macchina a stati avverte il layer successivo che può già cominciare a prelevare i dati mentre gli ultimi vengono caricati. Quando tutti i valori in uscita dai MAC sono stati salvati (`M terminal count`) si avverte lo stadio `elaborate` che i registri possono essere sovrascritti e si attende finché non sono pronti dei nuovi dati. Qualora tutti gli S_j output siano stati salvati la macchina

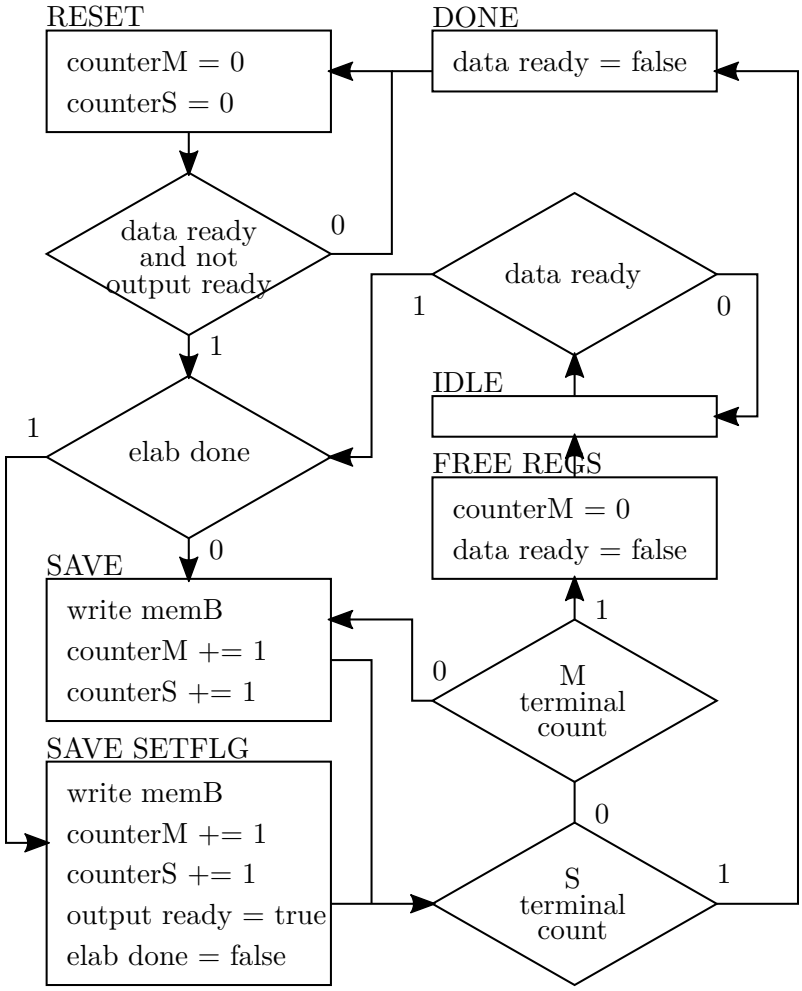


Figura 2.15: ASM chart dell'unità di controllo *save outputs*.

invece si resetta: poiché non è detto che S_j sia un multiplo di M_j, durante l'ultima iterazione potrebbe non essere necessario salvare tutti gli output dei MAC.

2.2.6 Sistema di controllo dei layer convolutivi e max-pooling

Il sistema di controllo dei layer convolutivi e max-pooling è molto simile a quello dei layer fully connected, con qualche eccezione; per questa ragione ci si può riferire sempre alla figura 2.10 per il datapath della struttura.

I dati utilizzati da questo tipo di layer vengono disposti in un tensore a 3 dimensioni, come descritto nel paragrafo 1.3.1. Per questa ragione alle memorie di

ingresso e di uscita viene assegnato un indirizzo composto da 3 sotto-indirizzi, rispettivamente riferiti alla riga, alla colonna e alla matrice del valore che si vuole leggere o salvare (ovvero gli indici i, j e k del tensore). Ad ogni sotto-indirizzo corrisponde un certo numero n di bit. Qualora una delle dimensioni del tensore non sia esattamente pari a 2^n esisteranno inevitabilmente porzioni di memoria che non vengono utilizzate. Per automatizzare l'indirizzamento delle memorie con dati disposti in questo modo si utilizza l'architettura mostrata in figura 2.16: ad ogni contatore corrisponde una dimensione del tensore, quando si termina una colonna automaticamente viene incrementato l'indice per le colonne e quando si termina la matrice automaticamente viene incrementato l'indice per le matrici. Il numero di contatori è arbitrario e dipende dal numero di dimensioni del tensore. Ad ogni contatore è associato un valore di `terminal_count` e la disposizione dei valori di uscita può variare se necessario. L'insieme dei contatori viene trattato in figura 2.10 come un singolo contatore "multidimensionale".

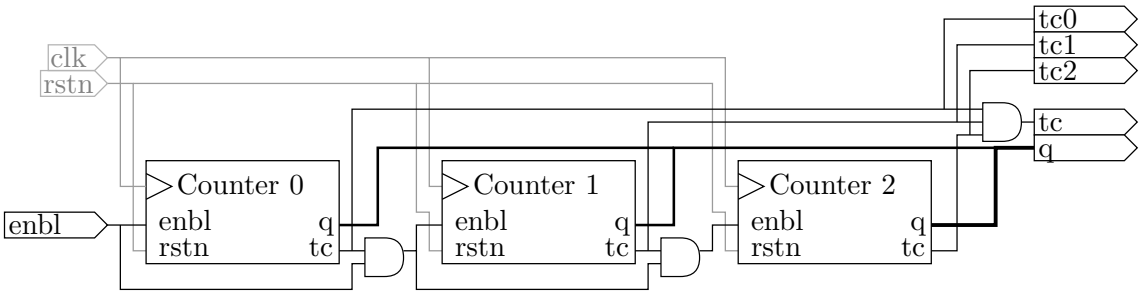


Figura 2.16: Architettura per l'indirizzamento dei volumi di dati multidimensionali.

Aggiuntivamente, oltre ai contatori che vengono descritti nel paragrafo 2.2.5, vengono impiegati anche i contatori rappresentati in figura 2.17. Il contatore `R` conta quanti input sono stati caricati; il contatore `DATA` è un contatore multidimensionale che conta i valori all'interno del campo di ricezione; il contatore `F` indica la posizione del campo di ricezione. L'indirizzo inviato alla memoria A viene controllato per verificare che si riferisca ad una posizione interna al volume di input assegnato. Il counter `DATA` è multidimensionale anche per i layer fully connected e gestisce tutto il volume di ingresso, qualora i dati di input non siano monodimensionali.

Primo stadio del layer: *load inputs*

Esattamente come per i layer fully connected, questo stadio carica uno ad uno gli input dalla memoria di uscita del layer precedente a quella di ingresso di questo layer, oppure carica i valori direttamente dall'esterno della rete neurale nel caso si tratti

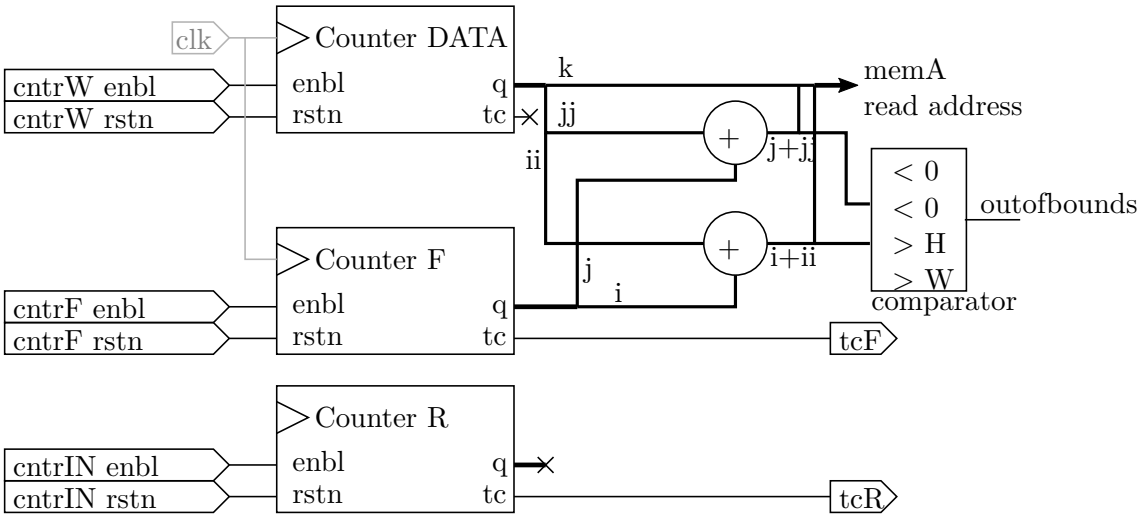


Figura 2.17: Contatori aggiuntivi utilizzati dai layer convolutivi.

del primo layer. I dati vengono prelevati e disposti all'interno dei volumi di memoria utilizzando l'architettura di figura 2.16. È possibile far iniziare l'elaborazione dei dati per mezzo dello stadio *elaborate* una volta che un certo numero di input è stato caricato. Questo valore viene raggiunto dal contatore `R` quando invia un segnale `terminal_count`.

Poiché gli input vengono caricati riga per riga, colonna per colonna, matrice per matrice, il numero totale di input che devono essere caricati per avere a disposizione tutti i valori contenuti all'interno del campo di ricezione alla sua posizione di partenza è pari a

$$N_{\text{loaded}} = H \cdot W \cdot (D-1) + H \cdot (F_w - 1) + F_h \tag{2.7}$$

Infatti vengono prima caricati tutti i valori di $D-1$ piani, poi sull'ultimo piano un numero di colonne pari a $F_w - 1$ e infine gli ultimi F_h valori per completare il campo di ricezione. Tuttavia quando si effettua l'elaborazione si parte dal primo valore e si arriva all'ultimo dopo un numero di cicli pari al numero di valori all'interno del campo di ricezione più il bias, ovvero $F_h \cdot F_w \cdot D + 1$. Sottraendo questo valore alla precedente equazione si ottiene

$$N\text{min}_{\text{loaded}} = H \cdot W \cdot (D-1) + H \cdot (F_w - 1) + F_h - (F_h \cdot F_w \cdot D + 1) \tag{2.8}$$

Questo è il numero di input che vanno caricati prima che si possa iniziare la prima elaborazione dei layer convolutivi (o max-pooling). In figura 2.18 viene mostrata la macchina a stati riferita allo stadio *load inputs*.

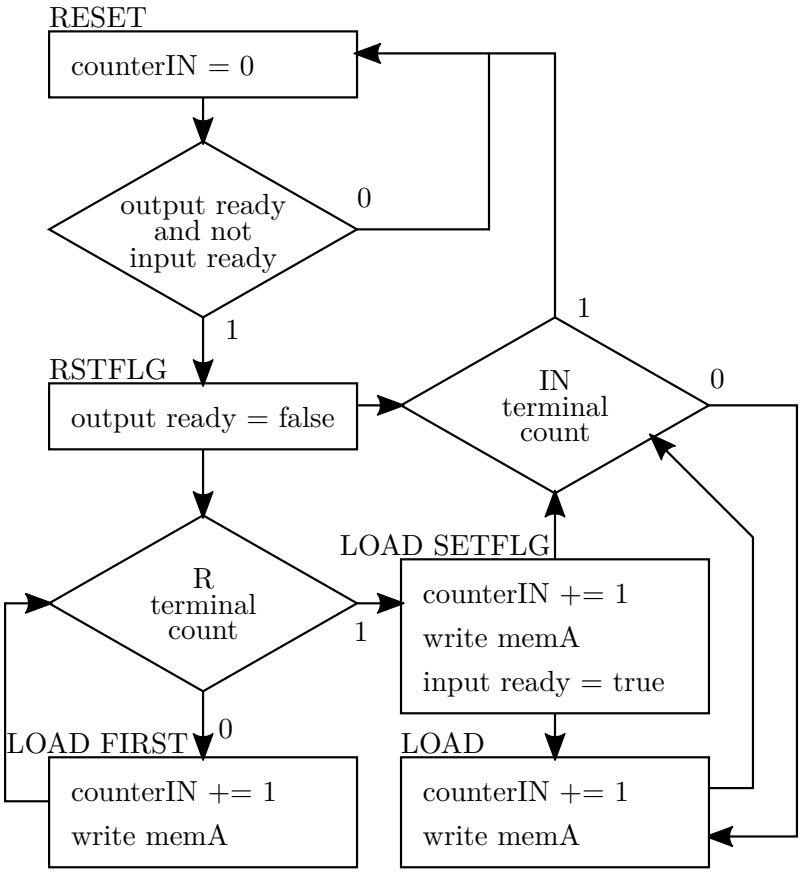


Figura 2.18: Unità di controllo dello stadio *load inputs* per il layer convolutivo.

La flag `input_ready` viene asserita solamente quando è stato caricato un numero di input sufficiente per iniziare l'elaborazione. Per i layer fully connected è sufficiente un solo input per cui il valore di `terminal_count` di `R` è sempre '1': in questo modo la macchina a stati diventa molto simile a quella che si trova in figura 2.13.

Secondo stadio del layer: *elaborate*

La macchina a stati per lo stadio *elaborate* si può trovare in figura 2.19. La modalità di funzionamento è molto simile a quella dei layer fully connected che si può trovare nel paragrafo 2.2.5, con alcune differenze. Mentre coi layer fully connected si eseguono una serie di iterazioni su un unico volume di dati, coi layer convolutivi

si eseguono più elaborazioni su diversi volumi di dati racchiusi all'interno del campo di ricezione. Per questa ragione non si raggiunge l'ultima elaborazione una volta che il contatore `SM` è arrivato al `terminal_count`, ma occorre anche attendere che si sia raggiunto l'ultimo conteggio anche per il contatore `F`, che indica la posizione del campo di ricezione. Inoltre, ogni volta che finisce un'iterazione e si completa l'analisi dei dati sul campo di ricezione, occorre spostarne la posizione incrementando il valore del contatore `F`. Notare che se il `terminal_count` di `F` viene posto sempre a '1', si ottiene la stessa macchina a stati che si trova in figura 2.14.

Il *padding* e lo *stride* non vengono gestiti direttamente dalla macchina a stati. Il contatore bidimensionale `F` è costruito in modo da utilizzare valori con segno che vengono incrementati in modo arbitrario. L'incremento viene deciso a priori ed è pari allo *stride*. Il *padding* superiore e laterale sinistro vengono applicati sul contatore facendo sì che al reset questo si inizializzi ad un valore pari all'opposto del *padding*. Ad esempio se il contatore conta gli indici i e j e i valori del padding sono `padding_top` e `padding_left`, i valori del contatore vengono inizializzati come $i = -$`padding_top` e $j = -$`padding_left`. Il contatore viene programmato per contare lungo tutto il volume di input con l'aggiunta dei valori di *padding* inferiore e laterale destro. Il `terminal_count` viene innescato quando si raggiunge l'ultima posizione in cui il campo di ricezione può ancora essere contenuto nel volume di input (tenendo conto del padding).

Una serie di comparatori controllano se l'indirizzo generato per la memoria A si riferisca ad un valore all'interno del volume di input, come mostrato in figura 2.17. In caso contrario, si sta cercando di ottenere un valore dalla cornice di zero padding che si trova intorno al volume di input. Per i layer convolutivi viene bloccata l'operazione per il ciclo corrente, poiché si sommerebbe uno 0 e il risultato non cambierebbe, mentre per i layer max-pooling viene impostato il valore dell'accumulatore nel blocco `max` a 0, se questo è negativo.

Per quanto riguarda i layer max-pooling le cose cambiano leggermente, senza però rendere necessaria l'alterazione della macchina a stati: poiché non ci sono pesi, il contatore `W` viene invece utilizzato per contare i valori all'interno di una singola matrice nella porzione contenuta nel campo di ricezione. Ogni volta che una porzione di matrice viene analizzata, viene generato un output che è il massimo dei valori presi in considerazione. Il contatore `SM` invece conta le matrici e il suo `terminal_count` viene asserito ogni volta che si raggiunge l'ultima matrice durante la valutazione dei dati e indica che occorre spostare la posizione del campo di ricezione.

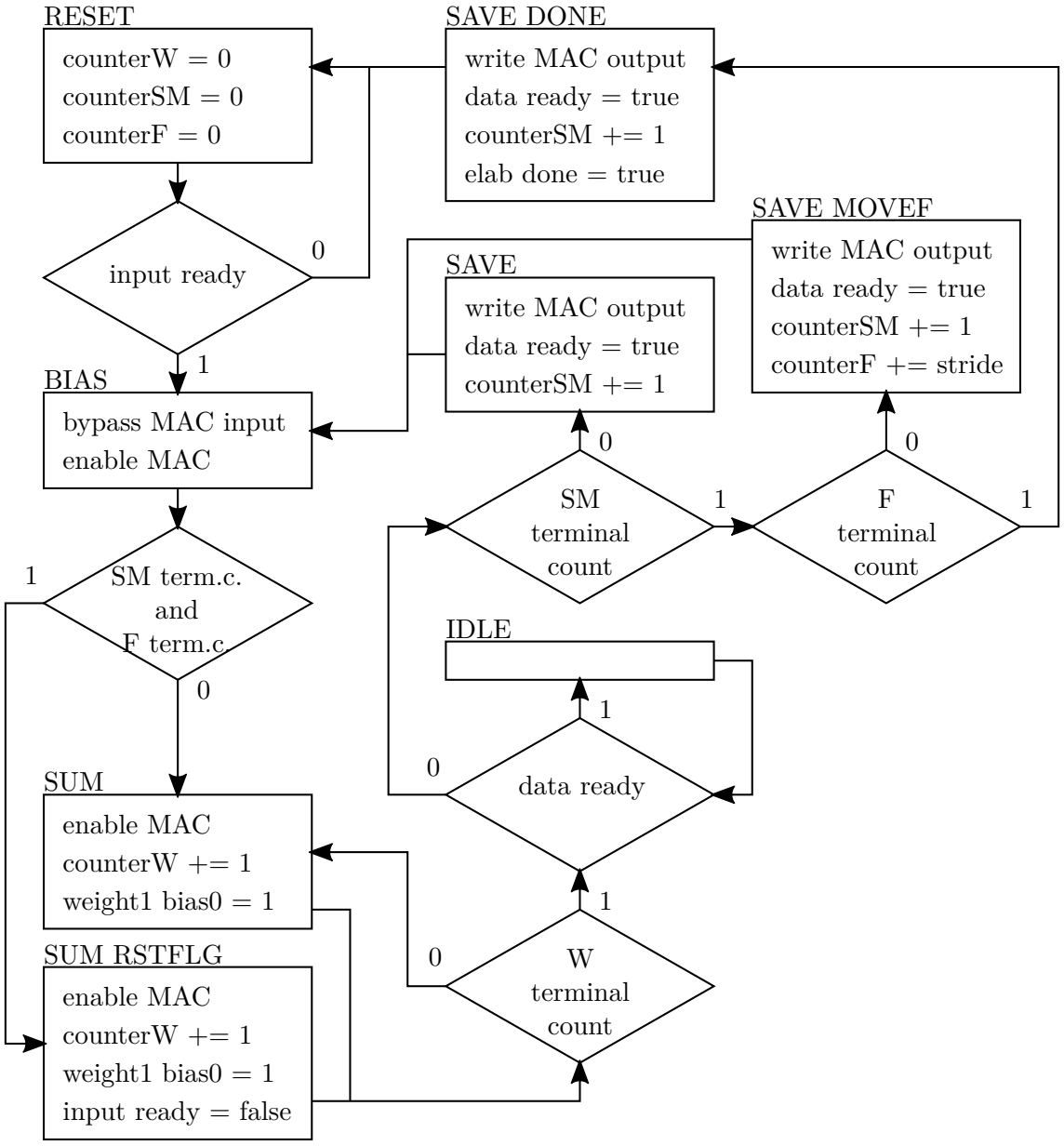


Figura 2.19: Unità di controllo dello stadio *elaborate* per il layer convolutivo.

Terzo stadio del layer: *save outputs*

I valori di output vengono disposti nella memoria di uscita in modo che ad ogni matrice corrispondano tutti i valori generati da un singolo neurone; si hanno tante matrici quanto è il numero di neuroni assegnati al layer. Nel caso dei layer max-pooling il numero di neuroni è sostituito dalla profondità del volume di input. Un

contatore tridimensionale conta prima il numero di neuroni, poi le righe e le colonne del volume di uscita. I valori, prima di venire assegnati come indirizzo della memoria B, devono essere riordinati in modo da avere i bit per il conteggio dei neuroni nella posizione più significativa. Viene generato un `terminal_count` ogni volta che vengono contati tutti i neuroni (`tcS`) e uno ogni volta che vengono contati tutti i valori nel volume (`tcOUT`). La macchina a stati del terzo stadio, *save outputs*, si trova in figura 2.20. In questo caso la macchina a stati è pressoché identica a quella della figura 2.15, con la differenza che alla fine di un ciclo di salvataggio viene verificato se tutti i dati sono stati salvati. Per le reti fully connected infatti ad ogni neurone è associato un solo output, mentre per le reti convolutive ad ogni neurone viene associato un intero piano di valori di uscita. Qualora `tcOUT` sia sempre pari a '1', la macchina a stati diventa esattamente come quella del layer fully connected.

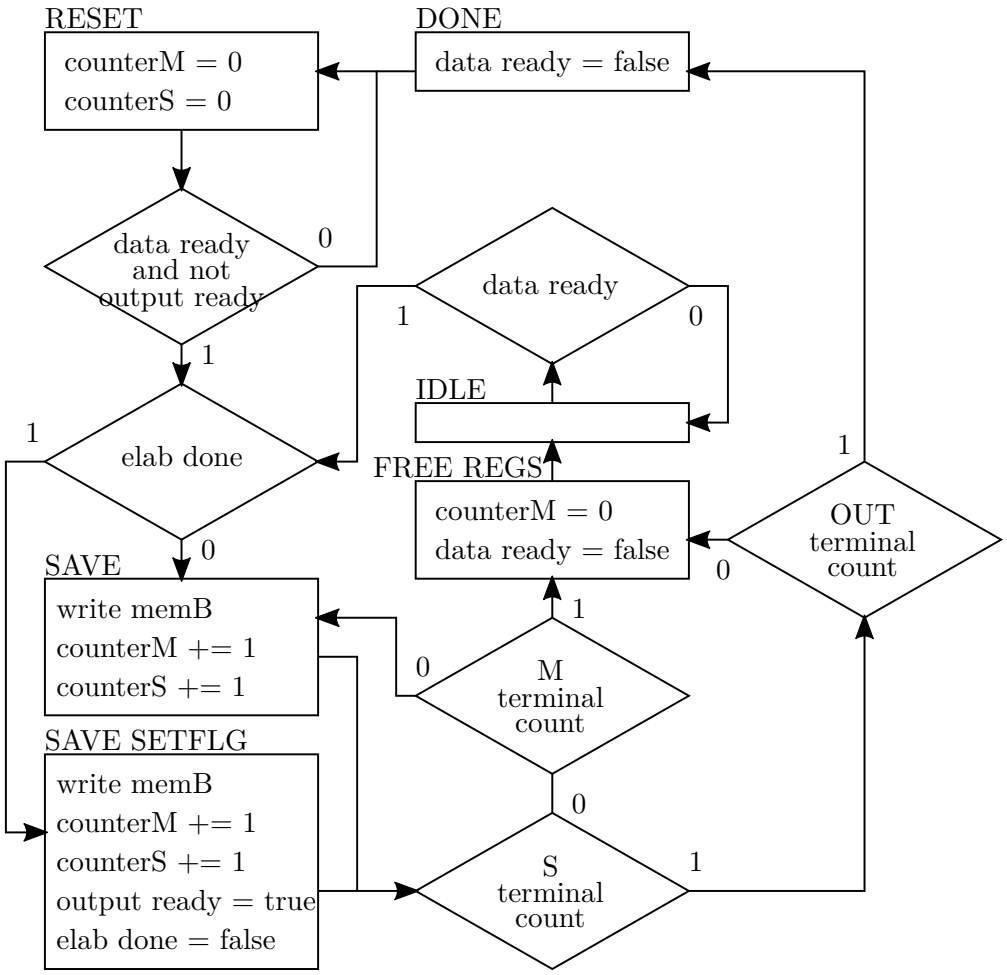


Figura 2.20: Unità di controllo dello stadio *save outputs* per il layer convolutivo.

2.2.7 Pipeline del layer

Per aumentare la frequenza del circuito è necessario inserire diversi livelli di pipeline all'interno del layer. Inoltre si deve tener conto che le memorie utilizzate sono tutte sincrone (perché tali sono le memorie impiegate in FPGA come l'Intel Cyclone 10 LP) per cui i valori in uscita da tali strutture sono attivi il colpo di clock successivo a quello in cui viene inviato l'indirizzo di lettura. I registri di pipeline che vengono inseriti nel datapath e nei collegamenti tra le unità di controllo per il layer si trovano in figura 2.21.

Di seguito l'elenco dei livelli di pipeline:

1. occorre un livello di pipeline per gestire il ritardo introdotto dalla memoria di uscita del layer precedente; questo non è necessario se il layer in questione è il primo della rete neurale;
2. occorre un livello di pipeline per ridurre la critical path dovuta al calcolo dell'indirizzo dei pesi, essendo necessari un moltiplicatore e un sommatore in serie;
3. anche la memoria dei pesi introduce un ritardo per cui occorre un terzo livello di pipeline;
4. il fan-out della memoria di input può diventare grande qualora vengano impiegati molti MAC, per cui occorre un ulteriore livello di pipeline;
5. la critical path all'interno del MAC è lunga e va ridotta;
6. il grande fan-in del multiplexer combinato con la lunghezza del percorso nel blocco della funzione di attivazione richiede un'ultimo livello di pipeline.

Per quanto riguarda i layer max-pooling, il numero di livelli di pipeline è minore perché viene sostituito il blocco `neurons` che, insieme al MAC, comprende diversi livelli di ritardo.

2.3 Struttura complessiva della rete neurale

La rete neurale nel suo complesso è costituita da vari layer messi in successione. La tipologia e i parametri di tali layer vengono definiti dall'utente, come viene spiegato nel paragrafo 2.4. L'output di ciascun layer viene collegato all'input del layer successivo; così avviene anche per i segnali di flag e per i valori dell'indirizzo della memoria

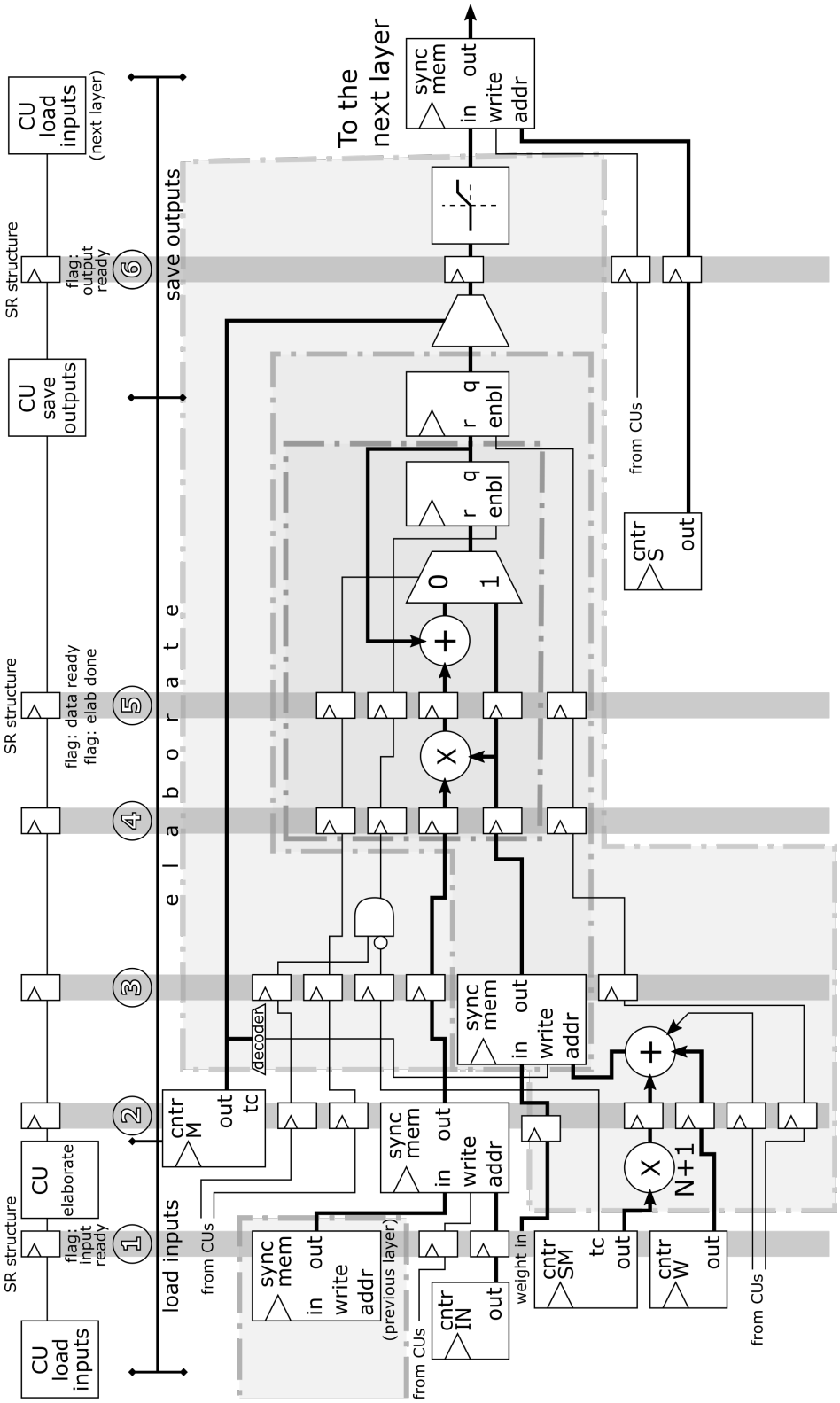


Figura 2.21: Schema che mette in evidenza i registri di pipeline impiegati nel layer fully connected; occorre tenere conto che le memorie introducono esse stesse un ritardo tra l'indirizzo di lettura e il valore di uscita.

di uscita. I layer vengono affiancati da un'unità di controllo *load weights* che gestisce il corretto caricamento dei bias e dei pesi all'interno delle rete. Un segnale gestito da questa macchina a stati (`write_weight`) permette di prendere il controllo dei contatori all'interno dei vari layer per poter indirizzare i pesi all'interno delle memorie (vedere figura 2.11): questo ovviamente implica che durante il caricamento delle memorie la rete neurale non possa entrare in funzione. In aggiunta un contatore `L` viene utilizzato per selezionare i valori di `terminal_count` a cui la macchina a stati debba essere sensibile e i contatori di quale layer debba controllare. In figura 2.22 viene mostrata la rete neurale come singolo blocco con i suoi input e output. In figura 2.23 si può osservare l'architettura completa che mostra come i layer vengono collegati tra loro e con la macchina a stati per il caricamento dei pesi.

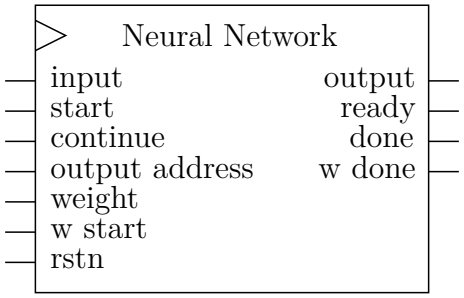


Figura 2.22: Rete neurale mostrata come singolo blocco con input e output.

2.3.1 Caricamento degli input nella rete neurale

Gli input devono essere inseriti dall'esterno uno per volta seguendo il timing riportato in figura 2.24. L'utente asserisce il segnale `start` in ingresso che è collegato direttamente al segnale `output_ready` del primo layer; in questo modo si avverte la rete neurale che ci sono degli input pronti in ingresso. Il segnale va tenuto asserito finché la macchina non è pronta a ricevere i dati in ingresso. A questo punto viene asserito per un colpo di clock il segnale `ready`, che è la negazione del segnale `resetn_output_ready` del primo layer, asserito quando i valori nella memoria di ingresso non servono più. Dal colpo di clock successivo l'utente deve inviare i valori di input che vengono campionati un colpo di clock per volta.

2.3.2 Interfaccia di uscita della rete neurale

Lo stadio *save outputs* dell'ultimo layer della rete neurale si deve interfacciare con l'esterno per poter inviare gli output in uscita dalla rete neurale. Il segnale `done` è

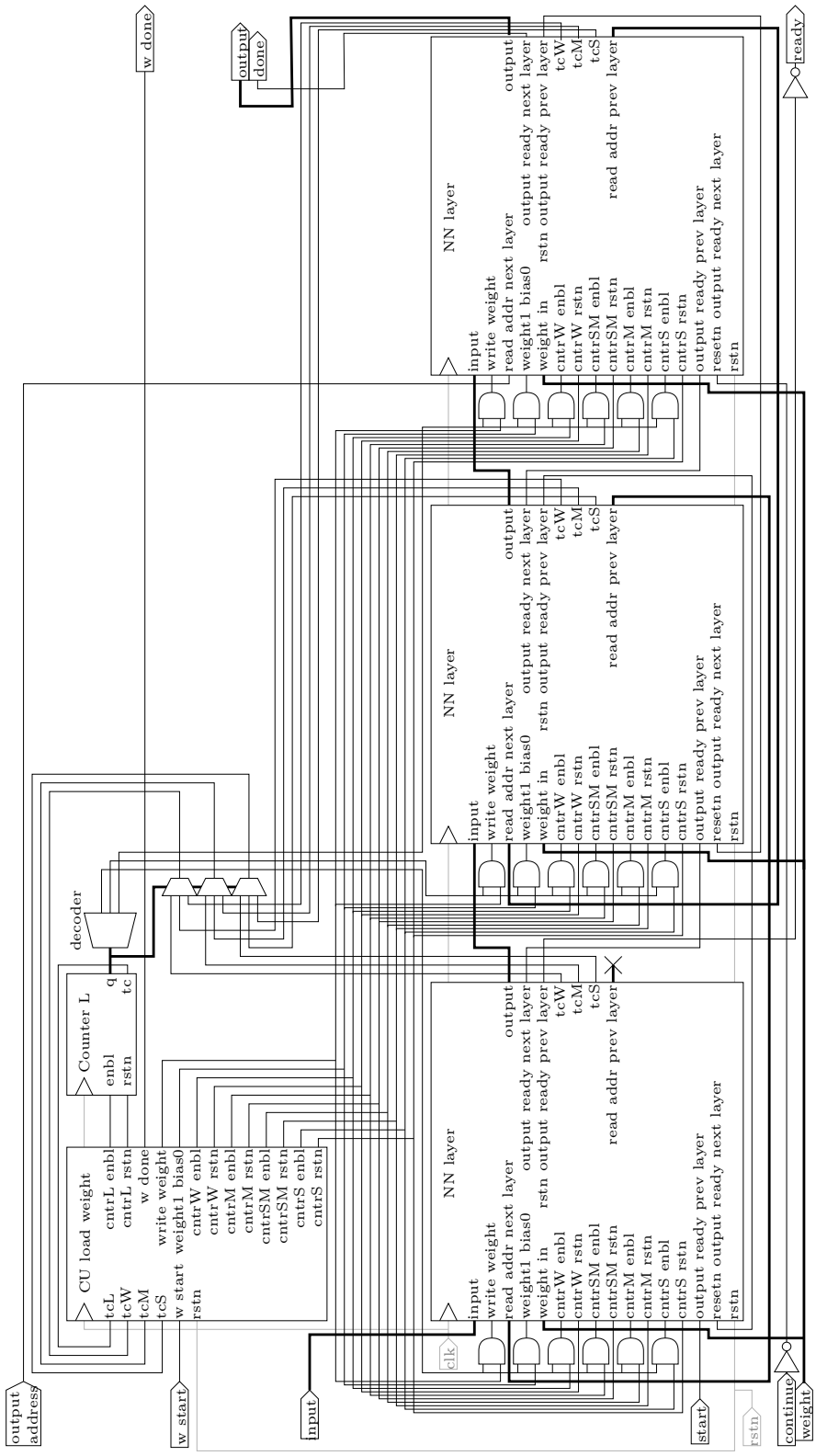


Figura 2.23: Architettura completa della rete neurale con collegamento tra i vari layer e la macchina a stati per il caricamento dei pesi. I segnali `prev_layer` vengono collegati al layer precedente mentre quelli `next_layer` vengono collegati a quello successivo.

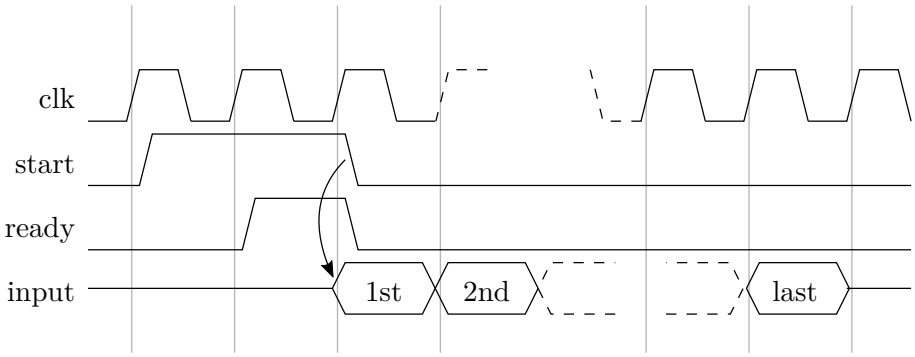


Figura 2.24: Timing dell'interfaccia di ingresso della rete neurale.

connesso direttamente al segnale `outputs_ready` e indica che dei dati sono pronti in uscita. L'utente esterno (che conosce a priori il numero di output che deve prelevare) a questo punto invia l'indirizzo dell'output che vuole prelevare alla macchina e dopo un colpo di clock riceve il dato corrispondente. Quando ha finito avverte la macchina che non gli servono più i dati tramite il segnale `continue` che va a resettare la flag `outputs_ready` dell'ultimo layer della rete neurale; è importante che questo segnale venga asserito o la macchina non potrà proseguire con le operazioni. Il timing di tale operazione si può vedere in figura 2.25.

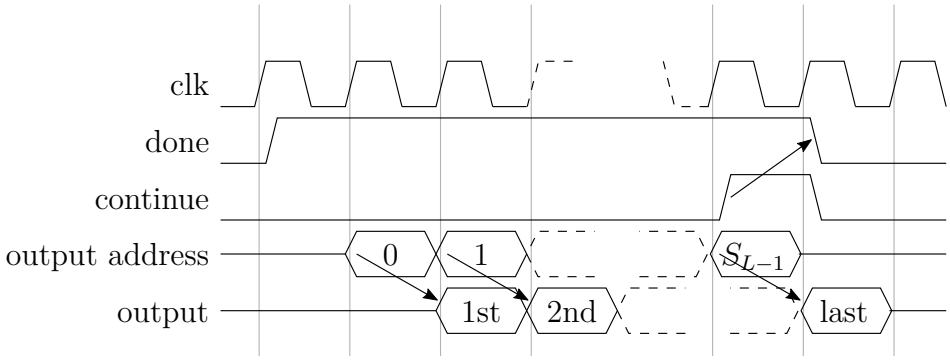


Figura 2.25: Timing della ricezione degli output.

2.3.3 Caricamento dei pesi nella rete neurale

Un'unità di controllo denominata *load weights* gestisce il caricamento dei pesi all'interno delle memorie del circuito; in aggiunta viene impiegato un contatore `L` il cui valore serve a indicizzare uno dei layer della rete neurale. L'utente invia il segnale `w_start` e dal colpo di clock successivo, usando l'input `weights` che si può vedere

in figura 2.22, invia i pesi uno per volta come mostrato nel timing in figura 2.26. In figura 2.27 si può vedere l'ASM chart della macchina a stati. Per necessità legata al tempismo dei dati in ingresso viene impiegata una macchina di Mealy invece che la più classica macchina di Moore: questo vuol dire che i segnali di controllo dipendono dagli input oltre che dallo stato della macchina.

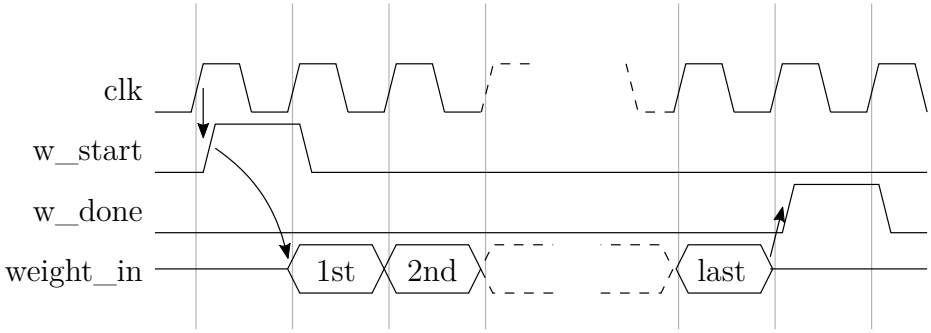


Figura 2.26: Timing del caricamento dei pesi.

La macchina a stati prende il controllo dei contatori dei layer utilizzando il segnale `write_weight`. Questi contatori non hanno bisogno di essere resettati perché sono già stati azzerati dalle unità di controllo dei layer che dopo il segnale di reset generico si trovano nello stato `RESET`. Tuttavia occorre comunque che la macchina a stati *load weights* prenda il controllo dei segnali di reset per poter utilizzare i contatori.

L'unità di controllo attende che l'utente invii il segnale `w_start`: dal colpo di clock successivo cominciano a venire inviati in sequenza i valori dei bias e dei pesi senza alcuna interruzione, da cui la ragione per cui si utilizza una macchina di Mealy: la macchina deve essere in grado di salvare i pesi in sequenza senza interruzioni dovute alla valutazione della fine dei vari cicli di caricamento (ad esempio se viene terminato il caricamento dei pesi su un layer la macchina non può perdere neanche un colpo di clock per incrementare il contatore `L`, oppure si perderanno dei dati). Per ogni neurone viene prima inviato il bias, poi tutti i pesi associati ad esso; finiti tutti i neuroni per un layer si passa a quello successivo e così via fino alla fine. Nel dettaglio:

- `w start` viene asserito;
- viene caricato un bias;

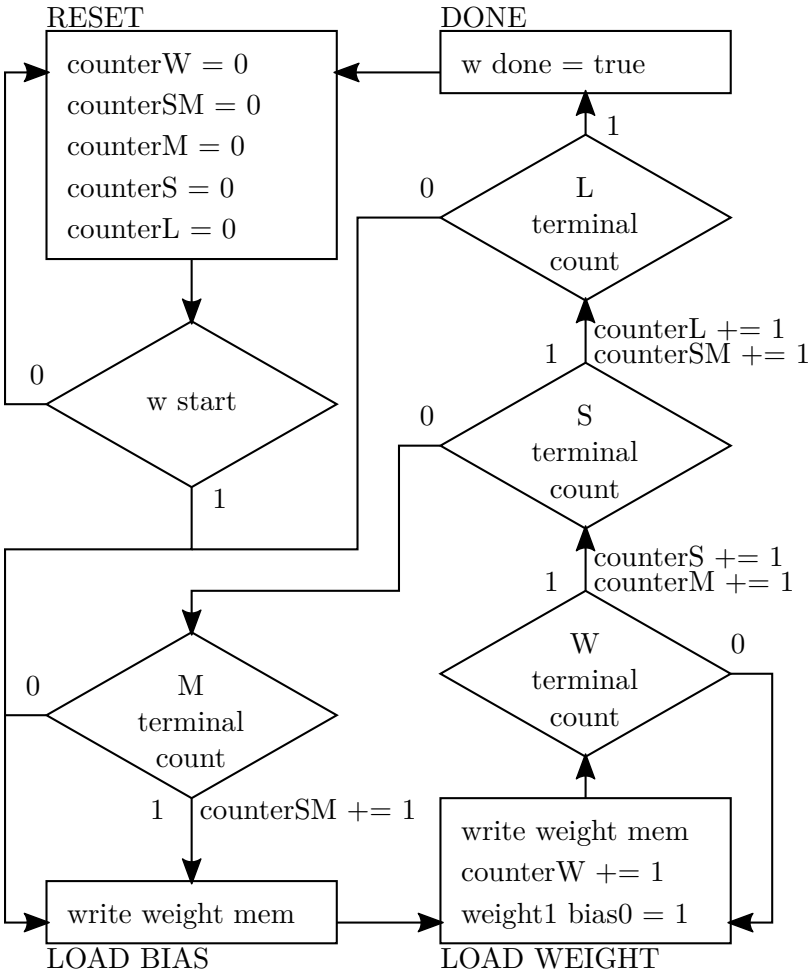


Figura 2.27: ASM chart dell'unità di controllo *load weights*.

- vengono caricati tutti i pesi incrementando volta per volta il contatore `W` del layer che conta il numero di pesi che sono associati ad ogni neurone; a fine operazione vengono incrementati i contatori `M` e `S`;
- se non sono stati caricati i pesi in tutti i neuroni del layer si ricomincia da capo; a questo punto se le memorie di tutti i MAC-group sono già state riempite il contatore `SM` cresce e si ricomincia a riempire dal primo MAC;
- se sono stati caricati i pesi in tutti i neuroni del layer si prosegue col layer successivo;
- al termine il segnale `w_done` informa l'utente che il caricamento è stato concluso.

La sequenza dei pesi varia a seconda del tipo di layer ed è a cura dell'utente che inserisce i valori mantenerne l'ordine. Ovviamente i layer di tipo max-pooling non contengono pesi e durante il caricamento non vengono contati.

Ordine dei pesi per i layer fully connected

Per ogni neurone occorre caricare prima il bias e poi di seguito tutti i pesi; l'ordine dei pesi è tale per cui al primo peso viene associato il primo input del layer, al secondo il secondo e così via. I gruppi di valori associati a ciascun neurone vanno inviati uno di seguito all'altro: al primo gruppo corrisponde il neurone che genera il primo output e così di seguito in ordine. Terminati tutti i gruppi vengono immediatamente inviati i valori associati al layer successivo, se ce n'è uno.

Ordine dei pesi per i layer convolutivi

Similmente al caso precedente per ogni filtro del layer si invia prima il bias, poi tutti i valori dei pesi. L'ordine dei pesi in questo caso va osservato con più attenzione. Il campo di ricezione del filtro comprende dati disposti su più matrici: i pesi vengono inviati a partire dalla prima matrice. Per ciascuna matrice si inviano prima i pesi della prima colonna, a partire dalla prima riga, fino all'ultima colonna. In pratica, immaginando che i pesi vengano disposti in un tensore tridimensionale di indici i, j e k, i viene incrementato per ogni peso, j viene incrementato quando si termina la colonna mentre k viene incrementato quando si termina la matrice. Per quanto riguarda l'ordine dei neuroni i gruppi di pesi associati a questi vanno inviati a partire dal filtro che genera la prima matrice di output. Anche qui, terminati tutti i gruppi di pesi, si procede immediatamente a inviare i pesi del layer successivo, se c'è.

2.4 Utilizzo del VHDL per generare una rete neurale parametrizzata

L'implementazione su FPGA della rete neurale è scritta interamente in VHDL e consente all'utente un totale controllo sui parametri. Si possono disporre un numero variabile di layer di tipologia differente, ciascuno può utilizzare un numero variabile di MAC e di neuroni, i parametri delle reti neurali convolutive (stride, padding, dimensione dei filtri) possono essere scelti a piacere e si può modificare il parallelismo dei dati e dei pesi. I parametri vengono raccolti dalla top-level entity e passati a tutti i componenti del circuito attraverso il generic mapping del VHDL. Sta ovviamente

all'utente selezionare i parametri in maniera accorta in modo da riuscire a generare il circuito voluto all'interno del chip selezionato.

2.4.1 Parallelismo e tipi di dato

L'utente può selezionare separatamente il parallelismo dei dati e dei pesi; poiché l'architettura utilizza valori fixed point, l'utente deve anche decidere il numero di bit che vengono dedicati alla parte frazionaria. Benché in origine si pensasse di utilizzare il pacchetto `fixed` per la gestione di tali valori, questo è risultato non completamente compatibile con il programma utilizzato per il fitting dell'architettura (Intel Quartus Prime) e non è compatibile con le versioni più vecchie del VHDL, quindi si è scelto di utilizzare semplicemente valori di tipo `signed` curando la posizione dei bit durante le moltiplicazioni. I valori di input e pesi vanno infatti allineati prima di eseguire una moltiplicazione. Inoltre il prodotto ha un parallelismo pari alla somma dei parallelismi dei fattori in ingresso: per questa ragione i valori in uscita dai MAC vanno troncati tenendo conto del numero di bit frazionali selezionati dall'utente per i dati. Il pacchetto `fixed` è stato utilizzato unicamente nei file di testbench per trasformare comodamente i valori floating point caricati dai file in valori fixed point e vice versa.

2.4.2 Top level entity

Esistono problemi di compatibilità tra la struttura RTL che viene compilata da Quartus e la struttura gate-level che questo genera per il fitting e le analisi di potenza: i segnali che originariamente sono `signed` o `unsigned` vengono trasformati nei più generici segnali `std_logic_vector`. Per questa ragione viene implementata una top-level entity che incapsula il blocco `neural_network` e converte esplicitamente i valori `signed` e `unsigned` in valori `std_logic_vector`. I testbench scritti per la versione RTL del codice VHDL in questo modo funzionano anche con la versione gate-level. In aggiunta la top-level entity si occupa anche di includere il pacchetto che contiene i parametri selezionati dall'utente e di inviarne i valori tramite generic mapping ai componenti del circuito.

2.4.3 File di gestione dei parametri

I valori dei parametri vengono scritti dall'utente in un file apposito denominato `parameters.vhd`. Questo file deve implementare un pacchetto di nome `parameters`

contenente una serie di costanti di nome predefinito. Nel listato 2.5 viene mostrato un esempio di tale file.

```
use work.generic_array_pkg.all;
use work.layer_record_pkg.all;
use work.nn_utils_pkg.all;

package parameters is

-- Parallelism for the data
constant data_bits              : integer := 9;
constant data_fractional_bits   : integer := 5;
-- Parallelism for the weights
constant weight_bits            : integer := 9;
constant weight_fractional_bits : integer := 5;
-- Size of the input volume in the neural network
constant inputs_size            : int_array := (20, 20, 1);
-- Layers options
constant layers                 : layer_array := (
    0 => (   -- layer 0
        layer => "conv",
        neurons => 10,
        macs => 1,
        r_field => (6,6),
        stride => (2,2),
        padding => (2,2,2,2),
        a_func => "posl"
    ),
    1 => (  -- layer 1
        layer => "maxp",
        neurons => 1,
        macs => 1,
        r_field => (2,2),
        stride => (2,2),
        padding => (0,0,0,0),
        a_func => "purl"
    ),
    2 =>(
        layer => "fulc",
        neurons => 10,
        macs => 1,
        r_field => (0,0),
        stride => (0,0),
        padding => (0,0,0,0),
        a_func => "purl"
    ));

-- RESERVED; do not modify
constant o_size                 : int_array :=
    eval_inputs(inputs_size, layers, layers'length);

end package parameters;
```

Listing 2.5: Esempio di file dei parametri per la rete neurale su FPGA.

Vengono inizialmente dichiarate una serie di costanti che indicano il numero di bit totale e il numero di bit frazionari dei valori fixed point utilizzati per rappresentare i dati e i pesi nella rete neurale. Un array tridimensionale `input_size` serve per indicare la dimensione del volume dei dati di ingresso; per i layer fully connected si pongono i primi due valori ad 1 e il numero di input va scritto nella terza posizione. Le caratteristiche dei vari layer vengono salvati in un array di record di lunghezza indefinita; in questo modo si può generare un numero arbitrario di layer. La natura dei record in VHDL non consente di omettere l'inizializzazione di un valore per cui occorre inizializzati tutti i parametri, anche quelli che non servono al layer. Ogni layer dispone dei seguenti parametri:

- `layer` è una stringa di 4 caratteri che indica la tipologia di layer; può essere "fulc", "conv" oppure "maxp";
- `r_field` è un array di due valori che indica l'altezza e la larghezza del campo di ricezione; non viene considerato dai layer fully connected;
- `stride` è un array di due valori che indica lo spostamento verticale e orizzontale del campo di ricezione; non viene considerato dai layer fully connected;
- `padding` è un array di 4 valori che indica il padding rispettivamente superiore, inferiore, laterale sinistro e destro applicato al volume di input; viene ignorato dai layer fully connected;
- `a_func` è una stringa di 4 caratteri che indica la funzione di attivazione che viene utilizzata dal layer; può essere "posl", "satl" oppure "purl" e non viene considerata dai layer max-pooling.

Il valore `o_size` che si trova in fondo al file indica il numero di output della rete e viene calcolato automaticamente a partire dai parametri dei layer, utilizzando le formule del paragrafo 1.3.1; questo valore è necessario per la sintesi della rete e non va modificato.

Capitolo 3

Struttura su microcontrollore

Implementare una rete neurale su microcontrollore risulta estremamente più semplice e veloce se paragonato al lavoro svolto su FPGA: per questa ragione il codice VHDL è stato scritto in modo che l'utente non debba metterci mano nella generazione della rete neurale di cui ha bisogno. Tuttavia occorre capire quanto sia vantaggiosa in termini di prestazioni l'implementazione su FPGA in paragone a quella su microcontrollore e per questo occorre progettare una rete anche in software.

La principale difficoltà che si incontra quando si confrontano le prestazioni di due dispositivi tanto diversi riguarda la cosiddetta *fairness*: è importante che il paragone venga effettuato nelle condizioni di funzionamento il più simili possibile e con gli stessi dati affinché l'operazione abbia un senso. Per questa ragione il firmware viene scritto in codice C che sia il più di basso livello possibile, con il minimo numero di chiamate a funzione e di qualsiasi cosa che possa introdurre un overhead temporale durante le operazioni. Inoltre, per mantenere il parallelo con il codice scritto in VHDL e facilitare le operazioni di testing, si scrive il codice in modo che modificando solamente alcuni parametri possa essere generato il software per reti neurali di dimensioni e tipologia differente.

Per la progettazione del software si è utilizzato per il debugging e il testing il microcontrollore della STMicroelectronics basato su ARM Cortex M4 la cui sigla è STM32F303RE [8]; il codice viene compilato utilizzando la piattaforma online Arm Mbed [9]. Per la valutazione dei tempi di calcolo viene impiegato un contatore del microcontrollore.

3.1 Struttura del codice

Durante la stesura del codice bisogna tenere conto di quali siano le operazioni che possano inquinare le misurazioni del tempo di calcolo. Come accennato, si scrive in C evitando il più possibile chiamate a funzione e accessi all'heap della memoria. Inoltre occorre in qualche modo inviare al microcontrollore i pesi e i valori di input e ricevere i risultati: questo può avvenire attraverso una semplice interfaccia seriale UART (Universal Asinchronous Receiver Transmitter) in connessione con un PC. Alcuni test effettuati mostrano però che il tempo di calcolo risulti trascurabile rispetto al tempo dovuto alla trasmissione dei dati. Per questa ragione gli input che vengono inviati dal PC vanno salvati a priori nel microcontrollore e gli output vanno mandati indietro solamente a posteriori. Nel caso dei pesi non c'è differenza in quanto, esattamente come per l'architettura su FPGA, vengono comunque memorizzati a priori. Input, pesi e output sono salvati in array dichiarati globalmente: diversamente, a causa della quantità di valori che vengono memorizzati, si incorrerebbe facilmente in uno stack overflow. La struttura generica del codice viene mostrata nel listato 3.1.

```
1  //libreria di funzioni per il microcontrollore fornita da ARM mbed
2  #include "mbed.h"
3
4  //dichiarazione dei parametri
5  ...
6  //dichiarazione degli array contenenti pesi, input, output e valori intermedi
7  ...
8
9  //apertura della linea seriale e dichiarazione di un contatore
10 Serial pc(SERIAL_TX, SERIAL_RX, 57600); // 57600 bauds, 8-bit data, no parity
11 Timer timer;
12
13 int main()
14 {
15     /** LOAD WEIGHTS ******************************/
16     //invia un segnale al PC che decreta l'inizio delle operazioni
17     pc.putc('w');
18     //caricamento dei pesi
19     ...
20
21     /** LOAD INPUTS *******************************/
22     //riceve tutti gli input dal PC e li salva in un array
23     for(int n = 0; n < TOTAL_INPUTS; ++n)
24     {
25         int buff;
26         pc.scanf("%d", &buff);
27         inputs[n] = buff;
28     }
29
```

```
30      /** EVALUATE NET*********************************/
31      //contatore degli input che sono stati utilizzati
32      int inputs_prepared = 0;
33      //contatore degli output che sono stati salvati
34      int outputs_saved = 0;
35      //avvia il contatore
36      timer.start();
37      //valuta gli output per ciascun pacchetto di input
38      for(int m = 0; m < PACKETS; ++m)
39      {
40          //trasferisce i valori di input in un array di ingresso della rete
41          ...
42          //valuta gli output
43          ...
44          //salva gli output correnti nell'array contenente tutti gli output
45          ...
46      }
47      //ferma il timer
48      timer.stop();
49
50      /** SEND OUTPUTS ******************************/
51      //invia tutti gli output al PC
52      for(int n = 0; n < TOTAL_OUTPUTS; ++n)
53      {
54          pc.printf("%d\n", outputs[n]);
55      }
56
57      /** SEND TIMING *********************************/
58      //invia il tempo totale impiegato per compiere tutte le elaborazioni
59      int elapsed_time = timer.read_us();
60      pc.printf("%d\n", elapsed_time);
61
62      //fine del programma
63  }
```

Listing 3.1: Struttura generica del codice per l'implementazione della rete neurale su microcontrollore.

Come si può vedere dal codice, viene impiegato un contatore per la misurazione del tempo totale di elaborazione. Il tempo viene misurato solamente durante le operazioni di calcolo e non tiene conto dei trasferimenti di dati lungo la linea seriale. Il tempo impiegato da una singola operazione può essere facilmente calcolato dividendo il tempo totale per il numero di valutazioni che vengono fatte.

3.2 Dichiarazione dei parametri

I parametri vengono definiti dall'utente in un file di intestazione `user_parameters.h` definendo delle macro del preprocessore. Vanno definiti prima una serie di parametri generici:

- `N_FRAC`: poiché i valori trattati sono fixed point, occorre indicare quanti sono i bit della parte frazionaria;
- `PACKETS`: numero dei gruppi di input che vengono elaborati, a ciascun gruppo corrisponde un'elaborazione completa;
- `INPUTSHEIGHT0`, `INPUTSWIDTH0` e `INPUTSDEPTH0`: dimensione della matrice di input della rete neurale, qualora il primo layer sia fully connected è consigliato settare l'altezza e la larghezza a 1 e assegnare alla profondità il numero di valori nel gruppo di input.
- `LAYERS`: numero di layer contando solo i layer nascosti e quello di output.

Vengono poi definiti i parametri che determinano la composizione della rete. A ciascun layer viene assegnato un indice a partire da 0, escludendo il layer di input che a livello implementativo non è altro che l'ingresso dei dati. L'etichetta di ciascun parametro è composta dal suo nome seguito dall'indice del layer a cui si riferisce. I parametri da definirsi per i layer sono:

- `LAYERn`: tipologia del layer, che può essere `FULLYCONNECTED`, `CONVOLUTIONAL` oppure `MAXPOOLING`;
- solo per i layer fully connected e convolutivi:
  - `NEURONSn`: numero di neuroni impiegati;
  - `AFUNCn`: funzione di attivazione, che può essere `PURELIN`, `POSLIN` oppure `SATLIN` (vedi paragrafo 2.2.2);
- solo per i layer convolutivi o max-pooling:
  - `FILTERHEIGHTn` (convolutivi) o `POOLHEIGHTn` (max-pooling): altezza del campo di ricezione;
  - `FILTERWIDTHn` (convolutivi) o `POOLWIDTHn` (max-pooling): larghezza del campo di ricezione;
  - `STRIDEn (h, v)`: valore dello *stride* verticale (`h`) e orizzontale (`v`);
  - `PADDINGn (t, b, l, r)`: valore del *padding* in alto (`t`), in basso (`b`), a sinistra (`l`) e a destra (`r`).

Nel listato 3.2 viene mostrato un esempio di definizione di una rete neurale che utilizza tutte le tipologie di layer disponibili.

```
// LAYER0
#define LAYER0 CONVOLUTIONAL
#define NEURONS0 10
#define FILTERHEIGHT0 6
#define FILTERWIDTH0 6
#define STRIDE0 (2, 2)
#define PADDING0 (2, 2, 2, 2)
#define AFUNC0 POSLIN
// LAYER1
#define LAYER1 MAXPOOLING
#define POOLHEIGHT1 2
#define POOLWIDTH1 2
#define STRIDE1 (2, 2)
#define PADDING1 (0, 0, 0, 0)
// LAYER2
#define LAYER2 FULLYCONNECTED
#define NEURONS2 10
#define AFUNC2 PURELIN
```

Listing 3.2: Esempio di file per la definizione della rete neurale su microcontrollore (`user_parameters.h`).

3.3 Dichiarazione delle strutture dati

Per prima cosa si salva in una costante il numero totale di input che verranno inviati al microcontrollore come il numero di valori presenti nel volume di input della rete moltiplicato per il numero di pacchetti. A partire da questo valore si dichiara un array monodimensionale (`inputs`) in cui vengono salvati tutti gli input da utilizzare. Si dichiara poi una array tridimensionale (`inputs0`) per il tensore di input della rete.

Vengono effettuate poi per ciascun layer le dichiarazioni di una serie di costanti e di strutture di dati a partire dai parametri definiti dall'utente. Per prima cosa vengono definite le dimensioni dei tensori di output:

- per i layer fully connected, la larghezza e l'altezza sono unitarie mentre la profondità del tensore è pari al numero di output, e quindi al numero di neuroni;
- per i layer convolutivi e max-pooling per l'altezza e la larghezza si applicano le equazioni 1.5 del paragrafo 1.3.1, mentre la profondità è pari al numero di neuroni.

Per i layer fully connected e convolutivi vengono quindi dichiarati gli array dei pesi (`weightsn`, dove `n` è l'indice del layer) e dei bias (`biasn`). L'array dei bias è monodimensionale e a ciascun indice corrisponde il bias del neurone corrispondente. L'array dei pesi invece è multidimensionale: tre indici servono per l'associazione del peso ad un input nella porzione tridimensionale di dati compresa nel campo di ricezione, un quarto indice invece indica il neurone al quale ci si riferisce. Per quanto riguarda i layer fully connected, i primi tre indici sono associati all'intero volume dei dati di input, essendo per definizione del layer tutti gli input connessi a tutti i neuroni.

Viene poi dichiarato un terzo array che serve a contenere gli input del layer successivo (`inputs(n+1)`) e che viene dunque utilizzato anche per salvare gli output del layer corrente. L'array è tridimensionale e le sue dimensioni sono memorizzate nelle costanti menzionate poco sopra nel testo. Le dichiarazioni riferite al layer si concludono con le costanti contenenti le dimensioni del tensore di input del layer successivo, che sono le stesse del tensore di output del layer corrente: infatti i due tensori si riferiscono alla stessa struttura di dati. A questo punto se esiste un layer successivo si prosegue con le dichiarazioni di quel layer, e così via.

Con i valori così dichiarati si ottiene per ultima una matrice di input il cui indice è pari al numero totale di layer. Questa matrice non va in input a nessun layer ed è invece usata come matrice di output della rete. A partire dalla dimensione di questo oggetto e dal numero di elaborazioni che si vogliono effettuare si può ricavare la dimensione di un ultimo array monodimensionale, `outputs`, nel quale verranno salvati tutti gli output calcolati. Nel listato 3.3 viene mostrato un esempio.

```
1  //INPUTS
2  const int TOTAL_INPUTS =
3      INPUTSHEIGHT0*INPUTSWIDTH0*INPUTSDEPTH0*PACKETS;
4  int inputs[TOTAL_INPUTS]; //array degli input
5  int inputs0[INPUTSDEPTH0][INPUTSWIDTH0][INPUTSHEIGHT0];
6
7  // LAYER0 (convolutional)
8  const int OUTPUTSHEIGHT0 =
9      (INPUTSHEIGHT0-FILTERHEIGHT0+PADDING0[0]+PADDING0[1]
10     /STRIDE0[0]+1;
11 const int OUTPUTSHEIGHT0 =
12     (INPUTSWIDTH0-FILTERWIDTH0+PADDING0[2]+PADDING0[3]
13     /STRIDE0[1]+1;
14 const int OUTPUTSDEPTH0 = NEURONS0;
15
16 int weights0[NEURONS0][INPUTSDEPTH0][FILTERWIDTH0][FILTERHEIGHT0];
17 int bias0[NEURONS0];
18 int inputs1[OUTPUTSDEPTH0][OUTPUTSWIDTH0][OUTPUTSHEIGHT0];
19
```

```
20 const int INPUTSHEIGHT1 = OUTPUTSHEIGHT0;
21 const int INPUTSWIDTH1 = OUTPUTSWIDTH0;
22 const int INPUTSDEPTH1 = OUTPUTSDEPTH0;
23
24 //LAYER1 (max-pooling)
25 const int OUTPUTSHEIGHT1 =
26     (INPUTSHEIGHT1-POOLHEIGHT1+PADDING1[0]+PADDING1[1]
27     /STRIDE0[0]+1;
28 const int OUTPUTSWIDTH1 =
29     (INPUTSWIDTH1-POOLWIDTH1+PADDING1[2]+PADDING1[3])
30     /STRIDE[1]+1;
31 const int OUTPUTSDEPTH1 = INPUTSDEPTH1;
32
33 int inputs2[OUTPUTSDEPTH1][OUTPUTSWIDTH1][OUTPUTSHEIGHT1];
34
35 const int INPUTSHEIGHT2 = OUTPUTSHEIGHT1;
36 const int INPUTSWIDTH2 = OUTPUTSWIDTH1;
37 const int INPUTSDEPTH2 = OUTPUTSDEPTH1;
38
39 //LAYER2 (fully connected)
40 const int OUTPUTSHEIGHT2 = 1;
41 const int OUTPUTSWIDTH2 = 1;
42 const int OUTPUTSDEPTH2 = NEURONS2;
43
44 int weights2[NEURONS2][INPUTSDEPTH2][INPUTSWIDTH2][INPUTSHEIGHT2];
45 int bias2[NEURONS2];
46 //matrice di output della rete
47 int inputs3[OUTPUTSDEPTH2][OUTPUTSWIDTH2][OUTPUTSHEIGHT2];
48
49 const int INPUTSHEIGHT3 = 1;
50 const int INPUTSWIDTH3 = 1;
51 const int INPUTSDEPTH3 = OUTPUTSDEPTH2;
52
53 //OUTPUTS
54 const int TOTAL_OUTPUTS =
55     INPUTSHEIGHT3*INPUTSWIDTH3*INPUTSDEPTH3*PACKETS;
56 int outputs[TOTAL_OUTPUTS]; //array degli output
```

Listing 3.3: Dichiarazione delle strutture di dati utilizzate durante le operazioni del microcontrollore, per una rete con layer convolutivo, max-pooling e fully connected. `inputs3` è la matrice di output della rete neurale.

3.4 Caricamento dei pesi

I pesi vengono inviati dal pc una volta che il microcontrollore invia un carattere predeterminato. Questi vengono inviati in successione, layer per layer, neurone per neurone, per primo il bias e poi i pesi, in maniera del tutto analoga a quanto descritto nel paragrafo 2.3.3. La disposizione dei dati all'interno delle strutture dichiarate nel paragrafo 3.3 è molto semplice, è sufficiente per ogni layer utilizzare una serie di

cicli annidati. Nel listato 3.4 viene illustrato il codice per il caricamento dei pesi e dei bias in un layer. Per i layer max-pooling ovviamente non serve fare nulla in quanto non utilizza pesi. Occorre notare che la dimensione della matrice dei pesi di un layer convolutivo è pari a quella del campo di ricezione, mentre per un layer fully connected è pari a tutto il volume di input.

```
//itera attraverso tutti i neuroni del layer
for(int n = 0; n < NEURONS0; ++n)
{
    int buff;
    pc.scanf("%d", &buff);
    bias0 = buff;
    //itera attraverso tutti i pesi
    for(int k = 0; k < INPUTSDEPTH0; ++k)
    {
        for(int j = 0; j < FILTERWIDTH0; ++j)
        //i fully connected usano j < INPUTSWIDTH0
        {
            for(int i = 0; i < FILTERHEIGHT0; ++i)
            //i fully connected usano j < INPUTSHEIGHT0
            {
                pc.scanf("%d", &buff);
                weights0[n][k][j][i] = buff;
            }
        }
    }
}
```

Listing 3.4: Salvataggio dei pesi e dei bias di un layer convolutivo; il caricamento per i layer fully connected è simile. Ovviamente l'indice utilizzato nel nome degli array e delle costanti varia a seconda dell'indice del layer a cui ci si riferisce.

3.5 Elaborazione

La fase di elaborazione ha inizio con il trasferimento dei dati dall'array monodimensionale `inputs` che contiene tutti gli input alla matrice di ingresso della rete `inputs0`. Questo avviene semplicemente con una serie di cicli annidati, con l'ausilio di una variabile che tiene il conto di quanti input siano già stati utilizzati (`inputs_prepared`, la cui dichiarazione si trova nel listato 3.1); nel listato 3.5 viene mostrato il codice.

```
for(int k = 0; k < INPUTSDEPTH0; ++k)
{
    for(int j = 0; j < INPUTSWIDTH0; ++j)
    {
        for(int i = 0; i < INPUTSHEIGHT0; ++i)
```

```
6           {
7               inputs0[k][j][i] = inputs[inputs_prepared++];
8           }
9       }
10  }
```

Listing 3.5: Preparazione della matrice di input della rete.

In maniera simile, dopo che viene effettuata l'elaborazione il processo termina con il salvataggio degli output nell'array `outputs` dal tensore di uscita della rete, anche qui con l'ausilio di una variabile che conta il numero di output già salvati (`outputs_saved`, la cui dichiarazione si trova nel listato 3.1). Nel listato 3.6 viene mostrato un esempio di salvataggio degli output ipotizzando che il numero di layer sia 3.

```
1   #define LAYERS 3
2   ...
3   //il layer di indice 3 non esiste; le costanti si riferiscono alla dimensione
        della matrice di output della rete
4   for(int k = 0; k < INPUTSDEPTH3; ++k)
5   {
6       for(int j = 0; j < INPUTSWIDTH3; ++j)
7       {
8           for(int i = 0; i < INPUTSHEIGHT3; ++i)
9           {
10              outputs[outputs_saved++] = CAT(inputs, LAYERS)[k][j][i];
11          }
12      }
13  }
```

Listing 3.6: Salvataggio della matrice di output della rete.

Il fulcro della fase di elaborazione è invece ovviamente costituito dalle operazioni svolte da ciascuno dei layer della rete neurale. Il codice riferito ad ognuno dei layer viene inserito in successione. Ogni layer utilizza un tensore di ingresso e genera un tensore di output che servirà da input al layer successivo; le operazioni che vengono eseguite variano a seconda della tipologia di layer.

3.5.1 Layer fully connected

I layer fully connected eseguono per ciascun neurone delle somme pesate utilizzando tutti i valori della matrice di ingresso. Un ciclo itera tra tutti i neuroni. Si inizializza un accumulatore `sum` con il valore di bias associato al neurone corrente (prestando attenzione all'allineamento dei bit in quanto si stanno usando valori fixed point). Si itera tra tutti i valori di input e si eseguono le somme pesate. I bit della somma

vanno riallineati e infine si applica la funzione di attivazione, variabile a seconda delle opzioni impostate. Le funzioni di attivazione vengono implementate a parte come funzioni `inline`: queste funzioni non vengono richiamate ma il codice che le costituisce viene direttamente inserito nel corpo del codice in modo da evitare overhead temporali. Per maggiori dettagli sui tipi di funzione di attivazione che vengono implementati vedere il paragrafo 2.2.2. Il tensore di output ha altezza e larghezza unitarie e profondità pari al numero di output. Nel listato 3.7 viene mostrato il codice riferito a questo tipo di layer.

```
1  //ricordare che l'indice utilizzato nei nomi degli array e delle costanti varia a
       seconda del layer
2  for(int n = 0; n < NEURONS0; ++n)
3  {
4          //si usano valori fixed point per cui occorre allineare il bias con il
               prodotto tra input e pesi
5      int sum = bias0[n] << N_FRAC;
6      for(int k = 0; k < INPUTSDEPTH0; ++k)
7      {
8          for(int j = 0; j < INPUTSWIDTH0; ++j) \
9          {
10             for(int i = 0; i < INPUTSHEIGHT0; ++i)
11             {
12                 sum = sum + inputs0[k][j][i]*weights[n][k][j][i];
13             }
14         }
15     }
16     //si usano valori fixed point per cui dopo le moltiplicazioni i valori vanno
           riallineati
17     sum = sum >> N_FRAC;
18     //la funzione di attivazione varia a seconda delle impostazioni del layer
19     inputs1[n][0][0] = poslin(sum);
20 }
```

Listing 3.7: Implementazione del layer fully connected su microcontrollore.

3.5.2 Layer convolutivo

L'implementazione dei layer convolutivi è più complessa. Innanzi tutto un ciclo itera tra tutti i neuroni. Dopodiché si hanno due cicli che iterano attraverso gli output: vengono definiti gli indici `i` e `j` che indicano la posizione del campo di ricezione, mentre gli indici `iout` e `jout` sono riferiti all'output che si sta calcolando. I valori `i` e `j` vengono inizializzati con un valore negativo di modulo pari al padding: in questo modo si può "emulare" l'aggiunta della cornice di 0 senza sprecare spazio in memoria. Quando si tenta l'accesso ad un indice al di fuori dei limiti della matrice di input si sa che il valore da utilizzare è 0. Si inizializza un accumulatore `sum` esattamente come

per i layer fully connected. Si itera attraverso tutti i valori racchiusi nel campo di ricezione utilizzando altri tre indici `k`, `ii` e `jj`. Gli indici del valore di input impiegato si calcolano semplicemente sommando gli indici `i` e `j` rispettivamente con `ii` e `jj`; si controlla se gli indici così ottenuti superano i limiti della matrice di ingresso. Se li superano, si assume che l'input sia 0 e non si effettua alcuna operazione, altrimenti si effettua la somma pesata dell'input. Infine si applica la funzione di attivazione e viene aggiornata la posizione del campo di ricezione utilizzando i valori dello *stride*. Nel listato 3.8 si può osservare il codice.

```
1  //ricordare che l'indice utilizzato nei nomi degli array e delle costanti varia a
       seconda del layer
2  for(int n = 0; n < NEURONS0; ++n)
3  {
4          //i e j indicano la posizione del campo di ricezione
5          //iout e jout sono gli indici dell'output che si sta calcolando
6      int j = -PADDING0[2];
7      for(int jout = 0; jout < OUTPUTSWIDTH0; ++jout)
8      {
9          int i = -PADDING0[0];
10         for(int iout = 0; iout < OUTPUTSHEIGHT0; ++iout)
11         {
12                 //si usano valori fixed point per cui occorre allineare il bias
                       con il prodotto tra input e pesi
13             int sum = bias0[n] << N_FRAC;
14             for(int k = 0; k < INPUTSDEPTH0; ++k)
15             {
16                 for(int jj = 0; jj < FILTERWIDTH0; ++jj)
17                 {
18                     for(int ii = 0; ii < FILTERHEIGHT0; ++ii)
19                     {
20                         if(i+ii >= 0 && i+ii < INPUTSHEIGHT0 &&
21                            j+jj >= 0 && j+jj < INPUTSWIDTH0)
22                         {
23                             sum += inputs0[k][j+jj][i+ii]
24                                   *weights0[n][k][jj][ii];
25                         }
26                     }
27                 }
28             }
29             //si usano valori fixed point per cui dopo le moltiplicazioni i valori
                   vanno riallineati
30             sum = sum >> N_FRAC;
31             //la funzione di attivazione varia a seconda delle impostazioni del
                   layer
32             inputs1[n][jout][iout] = poslin(sum);
33             i += STRIDE0[0];
34         }
35         j += STRIDE0[1];
36     }
37 }
```

Listing 3.8: Implementazione del layer convolutivo su microcontrollore.

3.5.3 Layer max-pooling

Il funzionamento di questo layer è del tutto simile a quello del layer convolutivo. In questo caso non c'è iterazione attraverso i neuroni e viene effettuata l'operazione di calcolo del massimo all'interno del campo di ricezione. Un accumulatore `maximum` viene inizializzato con il minimo valore rappresentabile in una variabile intera; per ogni input si controlla se è maggiore dell'accumulatore e in caso affermativo il suo valore viene salvato nell'accumulatore stesso. Poiché i valori possono essere anche negativi occorre tenere in considerazione che il massimo potrebbe essere uno 0 del padding. Si ricorda che le operazioni di calcolo del massimo avvengono separatamente su ciascuno dei piani di input. Nel listato 3.9 viene mostrato il codice per questo layer.

```
1  //valuto il minimo valore di tipo int possibile
2  static const int MINIMUM_INT = -(1 << (sizeof(int)*8-2));
3  int j = -PADDING0[2];
4  for(int jout = 0; jout < OUTPUTSWIDTH0; ++jout)
5  {
6      int i = -PADDING0[0];
7      for(int iout = 0; iout < OUTPUTSHEIGHT0; ++iout)
8      {
9          for(int k = 0; k < INPUTSDEPTH0; ++k)
10         {
11                 //si effettua un'operazione indipendente per ciascuno dei piani di
                       input
12             int maximum = MINIMUM_INT;
13             for(int jj = 0; jj < POOLWIDTH0; ++jj)
14             {
15                 for(int ii = 0; ii < POOLHEIGHT0; ++ii)
16                 {
17                     if(i+ii >= 0 && i+ii < INPUTSHEIGHT0 &&
18                        j+jj >= 0 && j+jj < INPUTSWIDTH0)
19                     {
20                         if(inputs0[k][j+jj][i+ii] > maximum)
21                         {
22                             maximum = inputs0[k][j+jj][i+ii];
23                         }
24                     }
25                     else
26                     {
27                         if(0 > maximum)
28                         {
29                             maximum = 0;
30                         }
```

```
31                      }
32                  }
33              }
34              inputs1[k][jout][iout] = maximum;
35          }
36          i += STRIDE0[0];
37      }
38      j += STRIDE0[1];
39  }
```

Listing 3.9: Implementazione del layer max-pooling su microcontrollore.

3.6 Generazione dinamica del codice col preprocessore

Come si è accennato all'inizio del capitolo il codice viene scritto in modo che modificando solo i parametri che si trovano nel file di intestazione `user_parameters.h` (paragrafo 3.2) si possano generare reti neurali di tipo e di dimensione diversa. Allo stesso tempo però si vuole evitare di rendere il codice lento mediante chiamate a funzione o memoria allocata dinamicamente. Per rispettare queste premesse si genera il codice interamente utilizzando il preprocessore con la definizione di macro e la possibilità di concatenare i nomi degli identificatori; è infatti possibile definire semplici operazioni o strutture simili ad array col processore e persino effettuare espansioni ricorsive delle macro [10]. Si evidenzia anche il fatto che nonostante il codice sia scritto interamente in C, si utilizza un compilatore C++ che permette di calcolare il valore delle costanti durante la compilazione. Il codice dell'implementazione su microcontrollore completo delle istruzioni per il preprocessore viene mostrato nell'appendice B.

Capitolo 4

Prestazioni e comparazione

Una volta implementate le reti neurali su FPGA e su microcontrollore si valutano le prestazioni. Occorre utilizzare dei metodi di testing validi che garantiscano la *fairness* del confronto. Per effettuare dei test realistici inoltre si devono scegliere dei problemi validi e la rete impiegata deve essere in grado di risolverli correttamente in gran parte dei casi. I test devono mettere in evidenza quando sia più opportuno utilizzare un tipo di implementazione piuttosto che l'altro.

4.1 Modalità di test

La valutazione delle prestazioni su ciascuno dei due dispositivi è differente. Il modo in cui la rete neurale viene implementata è infatti profondamente differente e le informazioni, i tool di sviluppo e le simulazioni disponibili per ciascuna delle due architetture varia.

4.1.1 Generazione dei parametri delle reti

Per testare le varie implementazioni della rete neurale occorre scegliere un data-set associato ad un particolare problema da risolvere. Una volta scelto il data-set che si vuole impiegare occorre addestrare una rete neurale in modo che i risultati che ottiene siano in gran parte corretti. Per fare ciò si utilizza il Deep Learning Tool™ di Matlab™ [2]. La rete neurale generata in Matlab™ viene addestrata utilizzando il data-set per l'apprendimento (training) e viene testata con il data-set per la validazione (validation) per vedere se si ottengono dei buoni risultati. I valori di input provenienti dal data-set vengono salvati in un file che servirà da input per le

implementazioni fisiche della rete neurale. I bias e i pesi della rete neurale vengono estrapolati e salvati in un file in modo che possano successivamente venire caricati su microcontrollore e su FPGA; l'ordine con cui vengono salvati si trova nei paragrafi 2.3.3 e 3.4. Anche gli output generati dalla rete neurale su Matlab™ vengono salvati in un file necessario per verificare il funzionamento delle implementazioni fisiche.

4.1.2 Testing dell'implementazione su FPGA

Il testing della rete neurale su FPGA avviene interamente tramite simulazioni. La famiglia di FPGA utilizzata per i test è l'Intel Cyclone 10 LP FPGA [7] come descritto nel paragrafo 2.1.3. Il codice viene compilato utilizzando l'IDE Intel Quartus Prime Starter Edition. I test vengono effettuati con Altera Modelsim utilizzando un file di testbench che preleva i pesi della rete neurale e gli input da file in formato floating point; il testbench trasforma tutti i valori in fixed point e invia alla rete prima i bias e i pesi, poi gli input, rispettando i segnali descritti nel paragrafo 2.3.1. Allo stesso tempo salva gli output in un file per valutare se la rete stia operando correttamente. I valori di output di riferimento utilizzati sono quelli generati con Matlab™.

I test per la valutazione del numero di colpi di clock medio impiegati per ciascuna operazione vengono effettuati utilizzando la versione Register Transfer Level (RTL) del circuito. Si valuta il numero totale di cicli utilizzati per il calcolo e lo si divide per il numero totale di operazioni eseguite. Le operazioni vengono eseguite con il massimo throughput possibile: i dati vengono inviati non appena la macchina è pronta a riceverne e vengono prelevati non appena sono validi in uscita. È importante notare che il numero di cicli medio per operazione non è pari al numero di cicli che intercorrono tra l'inserimento dell'input e la generazione dell'output: poiché la macchina ha un numero di livelli di pipeline molto elevato dopo l'inizio di un'operazione molte altre possono essere iniziate prima che questa finisca. Questo parametro è indicativo per valutare facilmente l'energia consumata per operazione; invertendolo si può ottenere il throughput del circuito.

Quartus è in grado di valutare i percorsi critici del circuito e quindi la frequenza massima a cui questo può lavorare. Utilizzando questa informazione è possibile valutare il tempo di calcolo effettivo, moltiplicando il numero di cicli per operazione per il periodo del segnale di clock (eq. 4.1). In questo modo è stato possibile anche minimizzare la lunghezza dei percorsi critici inserendo registri di pipeline dove necessario.

$$t_{\text{op}}[s] = \frac{\text{cicli}_{\text{op}}}{f[Hz]} \tag{4.1}$$

Utilizzando la versione gate-level del circuito ricavata dal compilatore è inoltre possibile ricavare un file Value Change Dump (.vcd) dove vengono registrate tutte le variazioni di tensione all'interno dell'architettura. Questo consente a Quartus di ricavare le switching activity dei segnali e quindi la potenza dissipata dal dispositivo, sia a livello statico che a livello dinamico, con un'incertezza del 20%. Si trascura la potenza dissipata per gli input e per gli output in quanto non è definito il modo in cui i valori vengono inviati al chip e poiché anche su microcontrollore le operazioni di input e output non vengono considerate. Bisogna tenere conto del fatto che se si vuole valutare la dissipazione di potenza dinamica a frequenze diverse rispetto a quella in cui è stata misurata occorre effettuare una proporzione: la potenza dinamica dipende linearmente dalla frequenza. Moltiplicando infine la potenza dissipata per il tempo di operazione è possibile valutare l'energia spesa per singola operazione (eq. 4.2).

$$E_{\text{op}}[J] = t_{\text{op}}[s] \cdot P[W] \tag{4.2}$$

4.1.3 Testing dell'implementazione su microcontrollore

Il testing dell'implementazione della rete neurale su microcontrollore avviene su scheda NUCLEO-STM32F303RE della ST Microelectronics [11] che dispone di un microcontrollore [8] basato su Arm Cortex M4, di utilizzo piuttosto diffuso.

Viene programmata una semplice interfaccia grafica utilizzando Visual C# il cui compito è quello di prelevare i valori dei pesi e degli input da file, convertirli in valori fixed point e inviarli al microcontrollore tramite linea seriale UART; i dettagli riguardanti al modo in cui i dati vengono ricevuti dal dispositivo si trovano nel paragrafo 3.1. La stessa interfaccia si occupa poi di ricevere i valori di output e di salvarli in un file per verificarne la correttezza.

Le prestazioni vengono valutate utilizzando un contatore del microcontrollore stesso, anche questo descritto nel paragrafo 3.1. Il tempo totale di calcolo misurato in *us* viene inviato all'interfaccia grafica su PC alla fine delle operazioni. Per ottenere il tempo medio necessario a effettuare una singola operazione si divide questo tempo per il numero totale di operazioni effettuate.

Dal datasheet del microcontrollore si può ricavare la frequenza massima di funzionamento e la potenza media dissipata durante l'elaborazione. La potenza segnata

dipende dalla frequenza, ma non linearmente poiché contiene anche il contributo statico. A partire dalla frequenza massima di funzionamento si può calcolare il numero di cicli necessari per un'operazione per poter fare un confronto con l'FPGA indipendente dal periodo del clock (eq. 4.3).

$$\text{cicli}_{\text{op}} = t_{\text{op}}[s] \cdot f[Hz] \tag{4.3}$$

A partire invece dalla potenza dissipata e dal tempo per operazione si può ricavare l'energia utilizzata per operazione, come nell'equazione 4.2.

4.2 Prestazioni della rete fully-connected

Per prima cosa si valutano le prestazioni dell'implementazione di una rete neurale fully connected, ideale quando il numero di input della rete è basso.

4.2.1 Caso studio: riconoscimento di segnali elettromiografici per il movimento di una mano

Si testa l'implementazione della rete neurale fully connected con la valutazione di segnali elettromiografici (EMG) provenienti da un avambraccio per il riconoscimento automatico dei movimenti di una mano [12]. I segnali vengono prelevati con degli elettrodi dall'avambraccio e campionati con la tecnica del compressed sensing: si ottengono dei valori in un dominio "compresso", non di Nyquist. Si hanno 16 elettrodi e quindi altrettanti segnali; si valuta il valore efficace (root-mean-square, valore RMS) di ciascun segnale per ogni finestra temporale di lunghezza definita. Nel dettaglio, i segnali elettromiografici vengono campionati con una frequenza di 4 ksample/s e il valore RMS viene estratto su finestre di 64 ms pari a 256 campioni. Per ogni finestra temporale si hanno dunque 16 valori che vengono inviati in input alla rete neurale.

I segnali si riferiscono ad una mano che può aprirsi, chiudersi o rimanere rilassata. Si hanno quindi in output 3 valori, ciascuno corrispondente ad un diverso movimento. Il movimento selezionato dalla rete è quello riferito all'output di valore massimo.

Il data-set impiegato per il training dispone di 434 gruppi di input, ciascuno composto di 16 elementi. Per il test vengono allenate diverse reti neurali. La rete neurale originariamente utilizzata per la risoluzione del problema consiste di un layer nascosto di 20 neuroni e di uno di uscita di 3 elementi; questa viene principalmente utilizzata per testare il funzionamento della rete su FPGA e su microcontrollore.

Per i test sulle prestazioni vengono impiegate diverse reti neurali composte da 1, 2 o 3 layer nascosti con 32 neuroni; il layer di uscita è ovviamente sempre composto da 3 neuroni, mentre in ingresso si hanno sempre 16 valori. L'accuratezza della rete non varia particolarmente, ma in questo lavoro ci si focalizza piuttosto sulle prestazioni fisiche delle implementazioni prese in considerazione.

4.2.2 Prestazioni della rete su FPGA

Vengono generate reti su FPGA che utilizzino per ogni layer 1, 2, 4, 8, 16 o 32 MAC. Il parallelismo dei dati e dei pesi è di 9 bit, con 5 bit per la parte frazionaria. Per ciascuna implementazione si effettuano valutazioni sulla frequenza, sul numero di cicli impiegati per operazione, sulla potenza e sull'energia.

In figura 4.1 vengono mostrati i cicli impiegati per operazione e la massima frequenza del circuito.

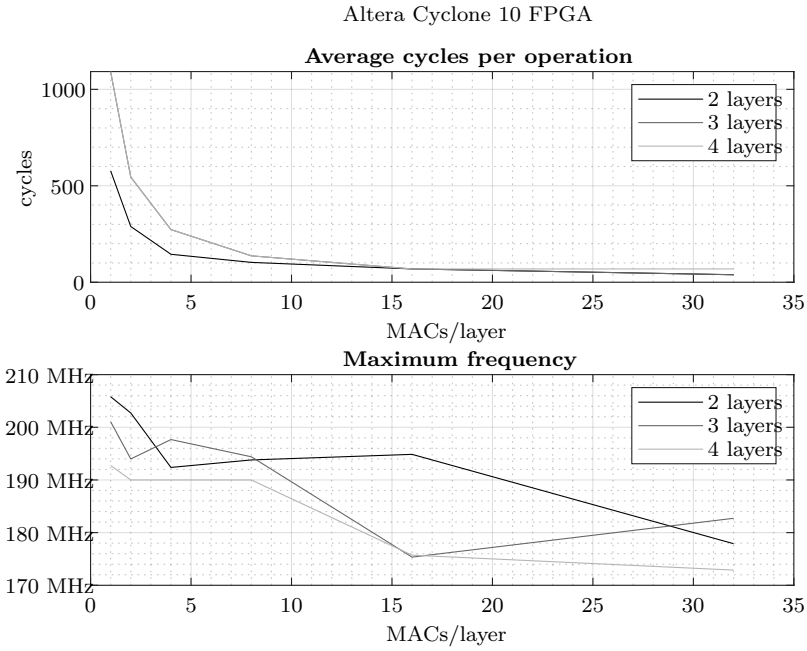


Figura 4.1: Numero di colpi di clock impiegato per ciascuna operazione e frequenza massima supportata dall'implementazione.

Il numero di cicli decresce come $1/x$ con l'aumentare del numero di MAC. Il punto ideale di funzionamento si attesta intorno agli 8 MAC per layer (ovvero si hanno 4 neuroni assegnati ad ogni MAC) oppure 4 MAC per layer se si utilizzano solo

2 layer; aumentare ulteriormente le risorse impiegate non comporta una riduzione significativa del tempo di calcolo. Il numero di cicli aumenta ovviamente anche con l'aumentare dei layer ma tende ad arrestarsi per un numero elevato di layer poiché la macchina dispone di molti livelli di pipeline. Ci si attesta intorno ai 150 cicli per operazione. La frequenza tende a ridursi con l'aumentare del numero di MAC (e quindi dell'area del circuito) perché diventa più difficile per il fitter ottimizzare il circuito. I punti nel grafico in cui si ha un calo di frequenza improvviso di più di una decina di MHz è dovuto al numero limitato di moltiplicatori dedicati presenti nei chip. Quando questi non sono più sufficienti per coprire la sintesi di tutti i MAC vengono generati dei moltiplicatori utilizzando i CLB dell'FPGA: il risultato è un circuito lento che utilizza più area. Quando anche i CLB non sono più sufficienti viene selezionato un chip più grande (e più costoso) e la frequenza cresce nuovamente. Si rimane comunque sempre sopra i 170 MHz e si hanno dei picchi di 205 MHz.

In figura 4.2 viene mostrato invece il tempo per operazione calcolato come prodotto dei valori della figura 4.1, come illustrato nell'equazione 4.1, oltre che la dissipazione di potenza.

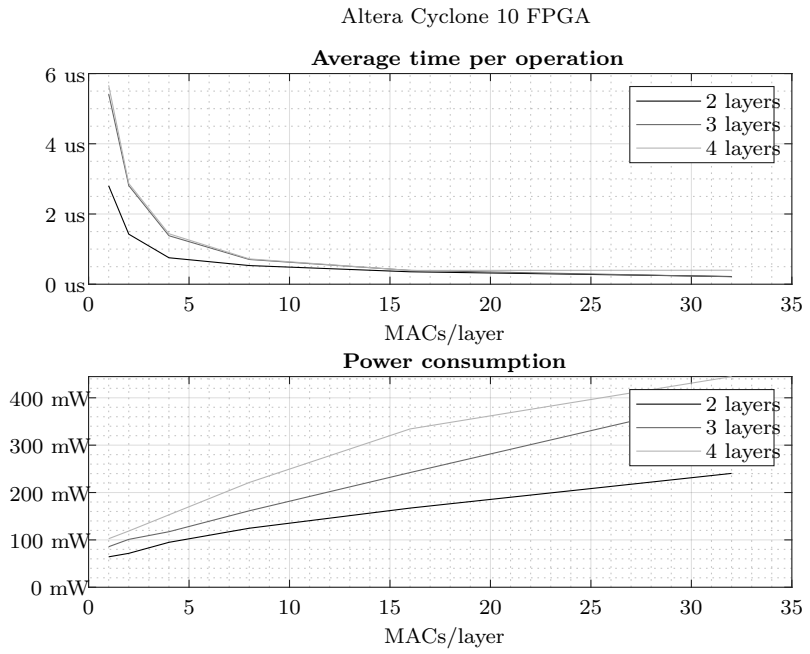


Figura 4.2: Tempo impiegato per ciascuna operazione e potenza dissipata; i valori sono calcolati alla massima frequenza di funzionamento del circuito nelle date condizioni.

L'andamento del tempo per operazione è del tutto simile a quello dei cicli per

operazione in quanto la variazione della frequenza al variare del numero di MAC non è particolarmente significativa. Ci si attesta intorno ai 750 ns per operazione al punto ottimo. La dissipazione di potenza è calcolata alla frequenza massima di funzionamento e cresce quasi linearmente con l'aumentare dei MAC; l'aumento è dettato in particolare dalla crescita della potenza dinamica. È presente una crescita anche con l'aumentare del numero di layer perché ovviamente aumenta il numero di risorse impiegate. Si va da valori pari a 60 mW a valori di 440 mW; stando ai risultati mostrati da Quartus, non si arriva ad una densità di potenza tale da richiedere l'utilizzo di dissipatori.

Si valuta infine l'energia media utilizzata per operazione moltiplicando i valori della figura 4.2 secondo l'equazione 4.2. Il risultato si può vedere in figura 4.3.

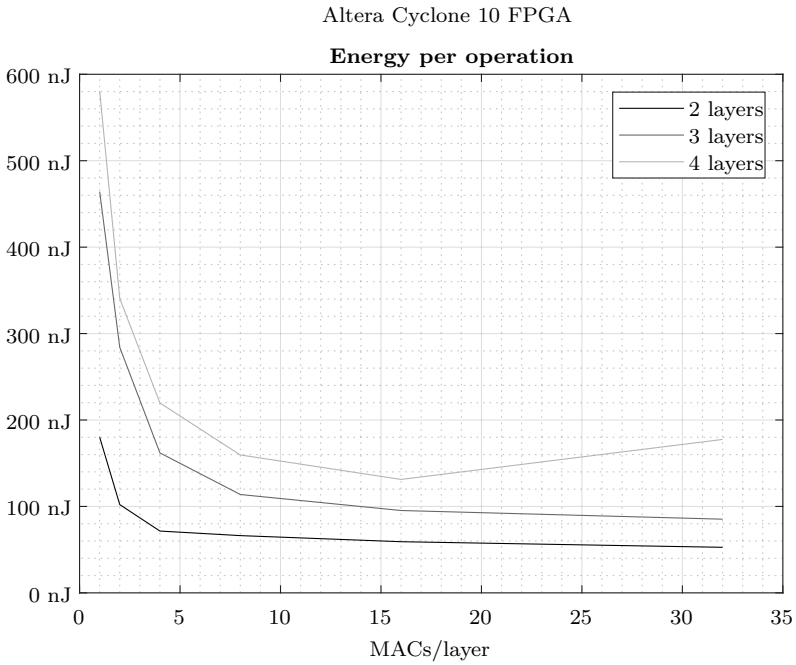


Figura 4.3: Energia utilizzata per ciascuna operazione.

La potenza cresce come x col numero di MAC per layer mentre il tempo per operazione decresce come $1/x$ per cui ci si aspetterebbe come risultato un valore costante; tuttavia esiste una componente di potenza nel circuito indipendente dal numero di MAC. Questo fa sì che con l'aumentare del numero di MAC l'energia consumata per operazione effettivamente diminuisca. Anche in questo caso il punto ottimo si trova utilizzando 8 MAC per layer oppure 4 MAC per layer con soli 2

layer. Per 2 layer si ha un consumo intorno ai 60 nJ, per 3 si hanno 100 nJ e per 4 si hanno 140 nJ.

4.2.3 Prestazioni della rete su microcontrollore

Su microcontrollore vengono misurati i tempi per operazione per reti con 2, 3 e 4 layer come nel caso dell'FPGA. Internamente il microcontrollore lavora con dati a 32 bit; si utilizzano 8 bit per la parte frazionaria dei valori fixed point. In questo caso però non è possibile aumentare il livello di parallelizzazione del sistema: avviene sempre e solo un'operazione per volta. Il numero di cicli per operazione viene riportato in figura 4.4: questi valori sono calcolati sapendo che le misurazioni vengono effettuate con un clock di frequenza pari a 8 MHz [11] e utilizzando l'equazione 4.3.

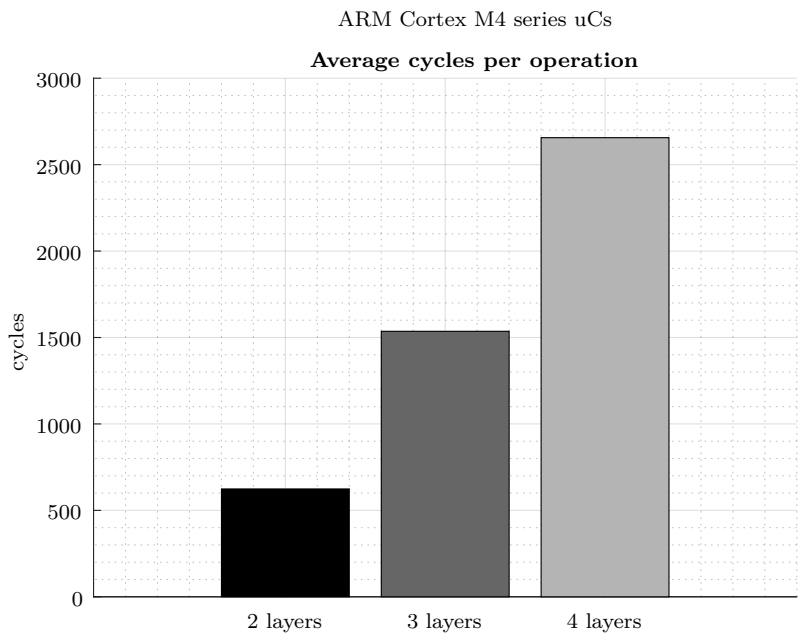


Figura 4.4: Cicli per operazione su microcontrollore.

Si assume verosimilmente che tutti i microcontrollori che si basano sull'architettura Arm Cortex M4 impieghino lo stesso numero di colpi di clock per eseguire le stesse operazioni. Si hanno valori che vanno dai 600 cicli per 2 layer a 2650 per 4 layer. L'incremento tra un layer e l'altro è costante. Il numero totale di cicli per 2 layer è minore dell'incremento che si ha con 3 layer perché il secondo layer nascosto deve processare 32 input contro i 16 del primo.

Poiché esistono molte tipologie di microcontrollori in commercio, si effettuano confronti tra dispositivi della famiglia Arm Cortex M4 pubblicizzati differentemente: si sceglie un chip *mainstream* (stm32f303re), uno *high performance* (stm32f446re) e uno *ultra low power* (stm32l452re). Si visionano i datasheet dei microcontrollori presi in esame [8, 13, 14]. Nella tabella 4.1 vengono elencate le caratteristiche elettriche dei dispositivi alla loro massima frequenza di funzionamento in *run mode*, ovvero durante l'esecuzione di operazioni di calcolo.

uC	voltage	current	power
`STM32F446RE`	1.7 V	81.0 mA @ 180 MHz	137.7 mW @ 180 MHz
`STM32F303RE`	2 V	27.3 mA @ 72 MHz	54.6 mW @ 72 MHz
`STM32L452RE`	1.7 V	8.9 mA @ 80 MHz	15.1 mW @ 80 MHz

Tabella 4.1: Corrente (run mode, peripherals disabled), tensione e potenza dissipata dei microcontrollori in esame alla loro massima frequenza.

Utilizzando i valori della frequenza massima e del numero di cicli per operazione si valuta il tempo medio per operazione per ciascuno dei dispositivi, secondo l'equazione 4.1. Si calcola quindi l'energia per operazione a partire dalle caratteristiche elettriche, con l'equazione 4.2. I valori così ricavati per ciascuno dei tre dispositivi si trovano nelle figure 4.5, 4.6 e 4.7.

Il chip *high performance* eccelle per velocità rispetto agli altri due con 3.5 µs per 2 layer e 15 µs per 4: la frequenza di clock è infatti molto più elevata. Tuttavia per quanto riguarda l'energia consumata per operazione il miglior chip è di gran lunga quello *ultra low power* con 100 nJ per 2 layer e 500 nJ per 4 layer; i tempi in questo caso sono decisamente più lunghi però.

4.2.4 Comparazione dei risultati

Per quanto riguarda i tempi migliori di esecuzione i risultati sono indubbiamente a vantaggio dell'FPGA. Il numero di cicli richiesto per operazione dall'FPGA al suo punto di funzionamento ottimo è 150 con una crescita quasi trascurabile all'aumentare del numero di layer, mentre per l'Arm Cortex M4 si contano 600 colpi di clock per 2 layer e addirittura 2650 per 4 layer. A questo si aggiunge che la frequenza massima di funzionamento su FPGA si mantiene al di sopra dei 180 MHz, ovvero la frequenza massima del microcontrollore *high performance*. I tempi in secondi alla massima frequenza sono 750 ns per operazione su FPGA contro i 3.5 µs per operazione su chip *high performance*.

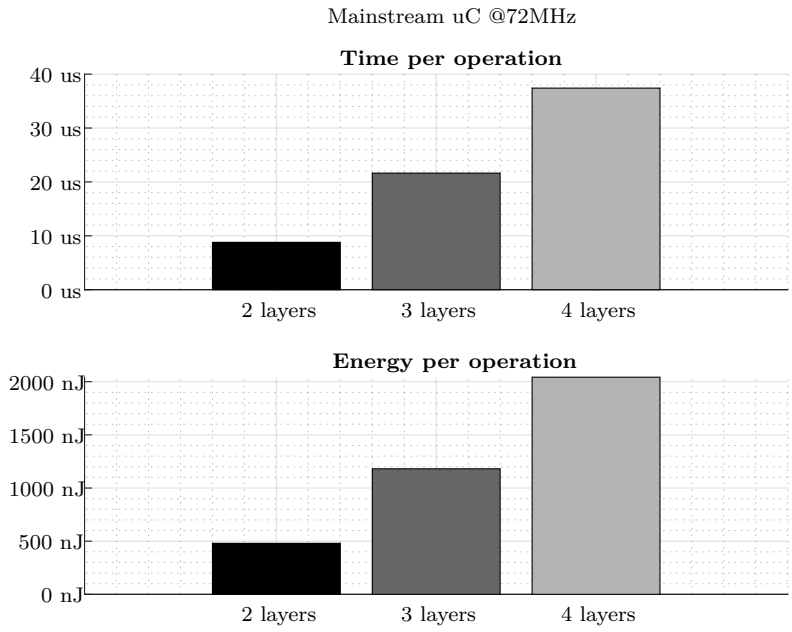


Figura 4.5: Tempo ed energia per operazione su microcontrollore *mainstream*.

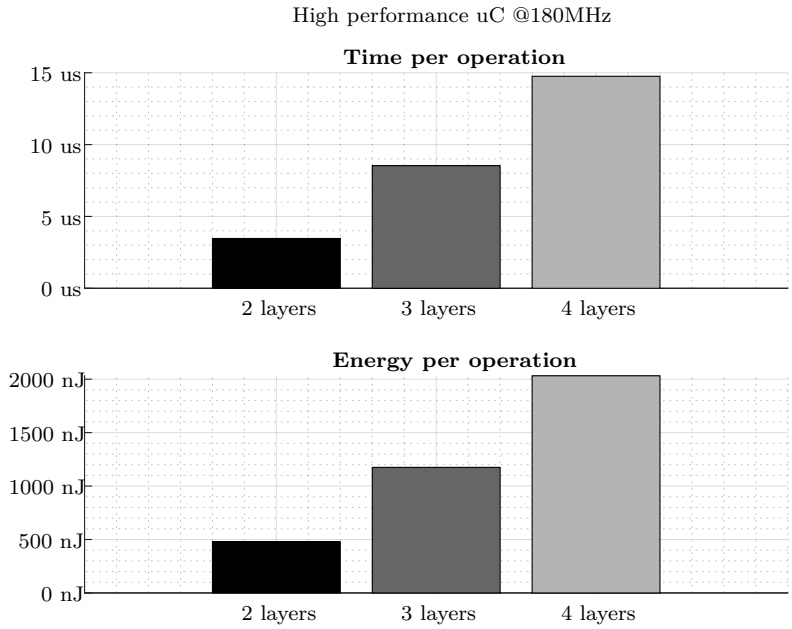


Figura 4.6: Tempo ed energia per operazione su microcontrollore *high performance*.

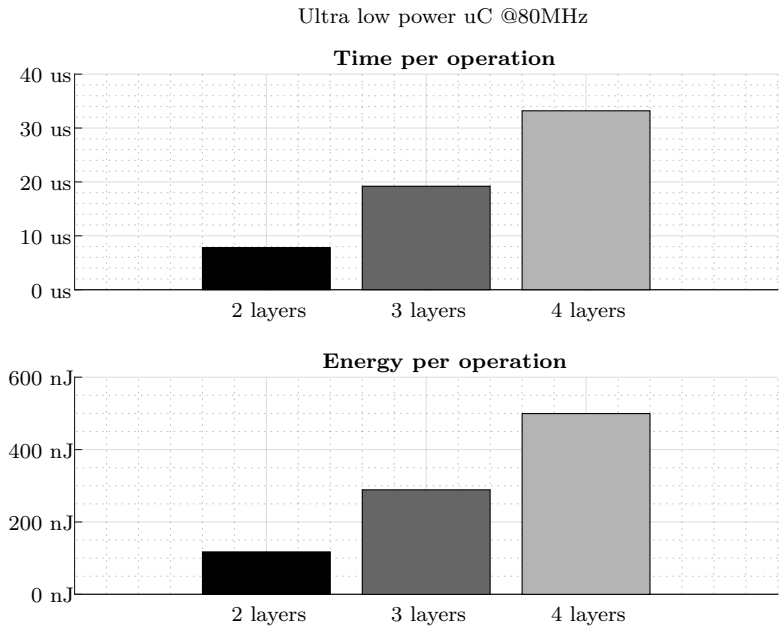


Figura 4.7: Tempo ed energia per operazione su microcontrollore *ultra low power*.

Anche per quanto riguarda l'energia consumata per operazione l'FPGA si dimostra più conveniente. L'FPGA al suo punto di funzionamento ottimo consuma dai 60 nJ ai 140 nJ per operazione al variare del numero di layer. Il microcontrollore *ultra low power* consuma invece dai 100 nJ ai 500 nJ. La distanza tra i due dispositivi aumenta all'aumentare del numero di layer coinvolto e quindi all'aumentare del numero di operazioni che vanno eseguite.

Occorre sottolineare che queste prestazioni sono state ottenute facendo lavorare i dispositivi al massimo throughput possibile; la dissipazione statica di potenza è generalmente minore per i microcontrollori e questo potrebbe portare l'implementazione su FPGA ad essere svantaggiosa nel caso l'ingresso dei dati avvenga dopo larghi intervalli di tempo, a meno di non disattivare l'alimentazione quando possibile.

Parallelismo dei dati

È importante tenere conto del fatto che i microcontrollori impiegati lavorano a 32 bit, mentre su FPGA si sono utilizzati solamente 9 bit. Questo ovviamente porta ad una differenza notevole di velocità e di consumi. Dai test risulta che con 9 bit l'accuratezza della rete non degrada. Con 8 bit tuttavia comincia a notarsi una riduzione della qualità della rete. Quando si esegue un prodotto occorre un numero

di bit pari alla somma del numero di bit dei fattori per evitare una perdita di informazioni. Su FPGA si può lavorare internamente dove serve con 18 bit. Se si usasse invece un microcontrollore a 16 bit con l'intento di ridurre i consumi, non sarebbe possibile lavorare con più di 8 bit per i dati; diversamente non si avrebbe la certezza di non avere perdite di informazione durante l'elaborazione.

Quantità di memoria

La dimensione delle reti neurali può raggiungere anche dimensioni notevoli. Poiché le risorse impiegate su FPGA non vengono riutilizzate per tutte le operazioni, l'implementazione della rete su questo tipo di supporto può incontrare problemi legati alla mancanza fisica di risorse come ad esempio il numero limitato di moltiplicatori dedicati. Su microcontrollore questo non avviene. Si può evitare la problematica sacrificando le prestazioni dell'architettura e riducendo il numero di MAC per layer.

Ciò che invece marca maggiormente il limite della dimensione della rete neurale che si può implementare su FPGA o su microcontrollore è invece la quantità di pesi che queste strutture possono memorizzare. Si tratta dunque di un problema legato alla quantità di memoria di cui questi chip dispongono. Il chip più piccolo della famiglia Intel Cyclone 10 LP [7] dispone di 30 memorie da 9kB che possono essere configurate con un parallelismo da 9 bit [7]. Questo vuol dire che possono venire salvati 245760 pesi con 9 bit di parallelismo, trascurando le memorie che vengono utilizzate per salvare gli input e gli output dei layer. Il microcontrollore tra quelli in esame con la RAM più capiente, ovvero l'*ultra low power* STM32L452RE [14], dispone invece di 160kB a 32 bit. Sebbene sia possibile salvare in un unica locazione di memoria più pesi di parallelismo inferiore a 32 bit, questo comporterebbe delle notevoli riduzioni di prestazioni dovute alle operazioni di estrazione dei dati. Quindi in questo caso si possono salvare solo 40960 pesi, trascurando la porzione di memoria che viene occupata dal codice. Lo stesso vale qualora si salvassero i pesi nella memoria flash del microcontrollore da 512kB: si potrebbero salvare 131072 pesi. Le FPGA dunque garantiscono l'implementazione di reti neurali di dimensioni maggiori, tenendo conto del fatto che esistono chip di dimensioni ben superiori a quelle analizzate ora.

Confronto dei prezzi

Vanno tenuti in considerazione anche i prezzi dei dispositivi in analisi per poter completare il confronto. Il prezzo dei chip FPGA è grandemente variabile a seconda delle dimensioni. Nella tabella 4.2 viene mostrato il prezzo del chip più piccolo

e di quello più grande. Come visto nel paragrafo 4.2.4 il chip più piccolo della famiglia Intel Cyclone 10 LP è in grado di contenere reti neurali persino più grosse di quelle implementabili su microcontrollore, per cui si può utilizzare il prezzo di questo chip come termine di confronto. Nella tabella 4.3 viene mostrato invece il prezzo dei microcontrollori in esame. L'FPGA è più cara di circa 3$: questo può essere significativo a seconda dell'applicazione.

Cyclone 10 chip	purchase price for 25 units
`10CL006YU256I7G` (smallest)	6.919 US$
`10CL120YF484I7G` (largest)	117.517 US$

Tabella 4.2: Costi del chip più piccolo e più grande della famiglia di FPGA Cyclone 10 (novembre 2018), dal negozio di Mouser Electronics (rivenditore autorizzato Intel). Il prezzo è stato convertito da euro a dollaro ed è stata rimossa l'IVA (22%).

uC	budgetary price for 10000 units
`STM32F446RE`	4.374 US$
`STM32F303RE`	3.4832 US$
`STM32L452RE`	3.4935 US$

Tabella 4.3: Costi dei microcontrollori Arm Cortex-M4 esaminati forniti ufficialmente dalla STmicroelectronics (novembre 2018). Il prezzo non è stato rimaneggiato ed è netto.

4.3 Prestazioni della rete convolutiva

Si valutano le prestazioni dell'implementazione di una rete neurale convolutiva, composta da diverse tipologie di layer. Questo tipo di reti è ideale per l'elaborazione di immagini e dunque per il filtraggio di grandi quantità di input.

4.3.1 Caso studio: MNIST data-set

Per testare le reti neurali convolutive si utilizza il MNIST data-set [15]. Questo consiste di una serie di immagini che raffigurano delle cifre scritte a mano: lo scopo della rete neurale è quello di associare il corretto valore numerico a ciascuna cifra che le viene inviata. Ognuna delle immagini è in bianco e nero e ha dimensione 20×20 pixel. Questo vuol dire che il tensore di ingresso della rete ha dimensione $20 \times 20 \times 1$. Il data-set consta di 30000 immagini utilizzate per l'addestramento della rete e di 10000 utilizzate per la validazione. Vengono generati due setup di rete

neurale utilizzando il Deep Learning Toolbox™ di Matlab™ che hanno la seguente composizione:

- Setup A:
  - matrice di input di dimensione $20 \times 20 \times 1$;
  - layer convolutivo con campo di ricezione di dimensione 6×6, 10 neuroni, *stride* $= [2, 2]$, *padding* $= [2, 2, 2, 2]$, funzione di attivazione ReLU 2.2.2;
  - layer max-pooling con campo di ricezione di dimensione 2×2, *stride* $= [2, 2]$, *padding* $= [0, 0, 0, 0]$;
  - layer fully connected con 10 neuroni, funzione di attivazione lineare pura (paragrafo 2.2.2).
- Setup B:
  - matrice di input di dimensione $20 \times 20 \times 1$;
  - layer convolutivo con campo di ricezione di dimensione 6×3, 10 neuroni, *stride* $= [2, 1]$, *padding* $= [2, 2, 1, 1]$, funzione di attivazione ReLU 2.2.2;
  - layer convolutivo con campo di ricezione di dimensione 3×6, 10 neuroni, *stride* $= [1, 2]$, *padding* $= [1, 1, 2, 2]$, funzione di attivazione ReLU 2.2.2;
  - layer max-pooling con campo di ricezione di dimensione 2×2, *stride* $= [2, 2]$, *padding* $= [0, 0, 0, 0]$;
  - layer fully connected con 10 neuroni, funzione di attivazione lineare pura (paragrafo 2.2.2).

La rete ha 10 output, ciascuno riferito ad un valore che va da 0 a 9. La cifra scelta dalla rete è quella associata all'output di valore massimo. La rete mostra avere un'accuratezza superiore al 96% per entrambi i setup. Si nota che il secondo layer del setup B ha molti valori in input per cui deve eseguire moltissime operazioni che rallentano il dispositivo.

4.3.2 Prestazioni della rete su FPGA

Il test su FPGA avviene utilizzando 1, 2, 5 o 10 MAC contemporaneamente per il layer convolutivo e per quello fully-connected. Non avvengono modifiche sul layer max-pooling in quanto questo non dispone di alcun grado di libertà. Il numero e la tipologia di layer è quella descritta nel paragrafo 4.3.1 per entrambi i setup impiegati.

Sia i dati che i pesi, come nel caso della rete neurale fully connected del paragrafo 4.2.2, hanno un parallelismo di 9 bit di cui 5 bit frazionari.

In figura 4.8 vengono mostrati i cicli impiegati per operazione e la massima frequenza del circuito.

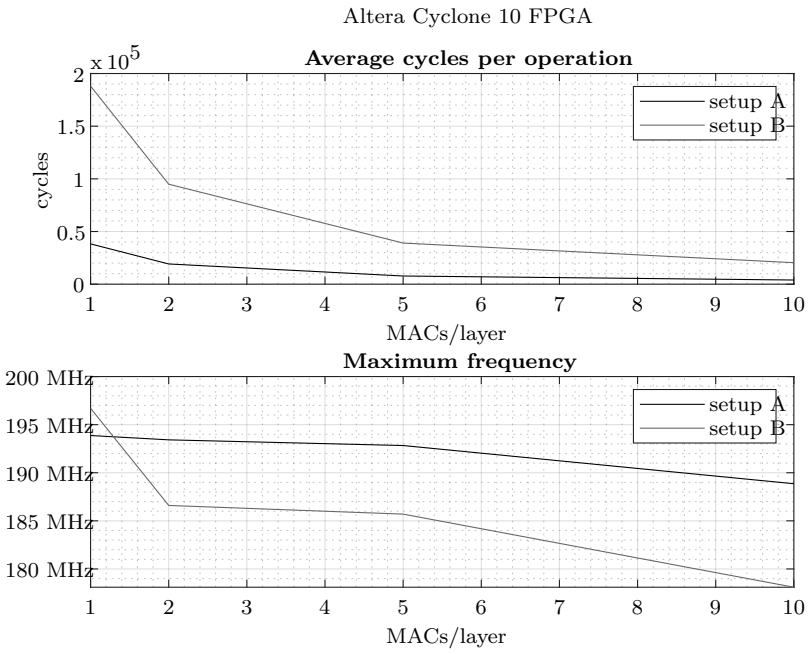


Figura 4.8: Numero di colpi di clock impiegato per ciascuna operazione e frequenza massima supportata dall'implementazione.

Il numero di cicli decresce come $1/x$ con l'aumentare del numero di MAC, come per le reti fully connected. È più difficile in questo caso decretare un punto ideale di funzionamento perché il numero complessivo di neuroni è inferiore; si può dire che intorno ai 5 MAC/layer si ottengano dei buoni risultati senza utilizzare troppe risorse, anche se i tempi di calcolo possono essere ancora ridotti. Il range varia dai 40000 ai 5000 cicli per operazione per il setup A, mentre col setup B si hanno valori che vanno dai 200000 cicli ai 25000. Con 5 MAC ci si attesta sui 10000 cicli per il setup A, col setup B invece 40000. Come accennato nel paragrafo 4.3.1, il setup B necessita di un numero di cicli molto più elevato a causa del secondo layer convolutivo.

Anche in questo caso, come per la rete fully connected, la frequenza tende a ridursi con l'aumentare del numero di MAC; non conviene aumentare eccessivamente

il numero di MAC utilizzati. Ci si attesta intorno ai 190 MHz per il setup A e intorno ai 185 MHz per il setup B.

In figura 4.9 viene mostrato il tempo per operazione calcolato come prodotto dei valori della figura 4.8, esattamente come nel caso della rete fully connected.

L'andamento del tempo per operazione è simile a quello dei cicli per operazione. Ci si attesta intorno ai 50 µs per operazione per il setup A, intorno ai 200 µs per il setup B. La dissipazione di potenza è calcolata alla frequenza massima di funzionamento e cresce linearmente con l'aumentare dei MAC; il contributo che aumenta principalmente è legato alla potenza dinamica, mentre la potenza statica cresce in maniera quasi trascurabile. Si va da valori pari a 120 mW a valori di 160 mW per il setup A, nel complesso valori inferiori a quelli della rete fully connected in quanto le operazioni sono molto più lunghe e tengono occupate meno risorse contemporaneamente. Col setup B invece la concorrenza delle operazioni cresce e si arriva a superare i 200 mW.

Si valuta infine l'energia media utilizzata per operazione moltiplicando i valori della figura 4.9 secondo l'equazione 4.2, come nel caso fully connected. Il risultato si può vedere in figura 4.10.

Il risultato è molto simile a quello che si vede per la rete fully connected: l'energia spesa decresce come x al crescere del numero di MAC. Si può dire che il punto di funzionamento migliore sia intorno ai 5 MAC anche in questo caso: in queste condizioni si ha un consumo energetico intorno ai 5 µJ per il setup A. Il setup B come ci si poteva aspettare consuma decisamente di più e con 5 MAC si attesta intorno ai 35 µJ.

4.3.3 Prestazioni della rete su microcontrollore

Le condizioni di funzionamento del microcontrollore sono le stesse impiegate per la rete fully connected descritte nel paragrafo 4.2.3 e il numero e il tipo di layer impiegati sono quelli descritti nel paragrafo 4.3.1. Per i consumi di potenza e le frequenze impiegate dalle varie famiglie di microcontrollori ci si riferisce sempre la tabella 4.1. Le prestazioni della rete convolutiva implementata sulle varie famiglie di microcontrollori si possono trovare nella figura 4.11, dove viene mostrato il numero di cicli per operazione, e nelle figura 4.12, 4.13 e 4.14, dove vengono mostrati i tempi effettivi di esecuzione e l'energia per operazione. I valori sono calcolati esattamente come è stato fatto per la rete fully connected. Anche in questo caso vengono utilizzati entrambi i setup descritti nel paragrafo 4.3.1.

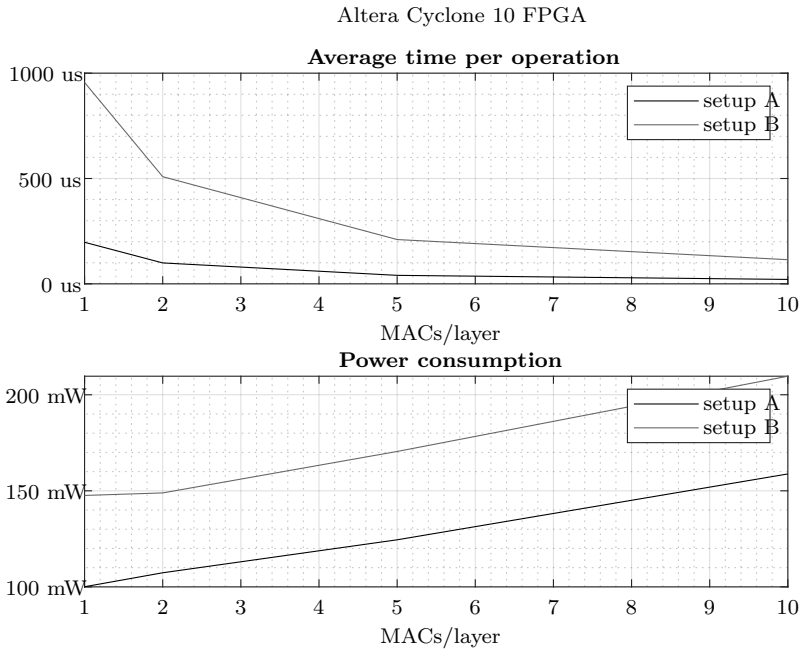


Figura 4.9: Tempo impiegato per ciascuna operazione e potenza dissipata; i valori sono calcolati alla massima frequenza di funzionamento del circuito nelle date condizioni.

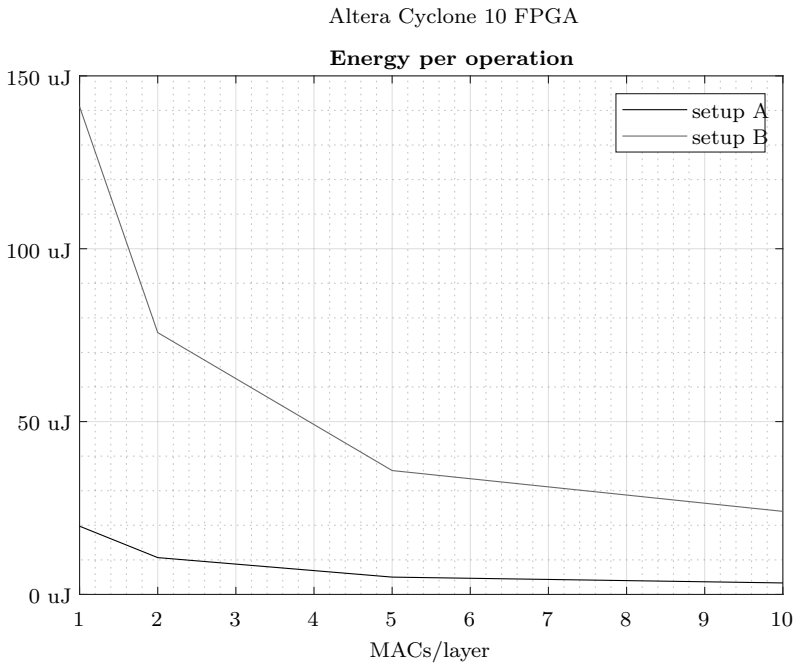


Figura 4.10: Energia utilizzata per ciascuna operazione.

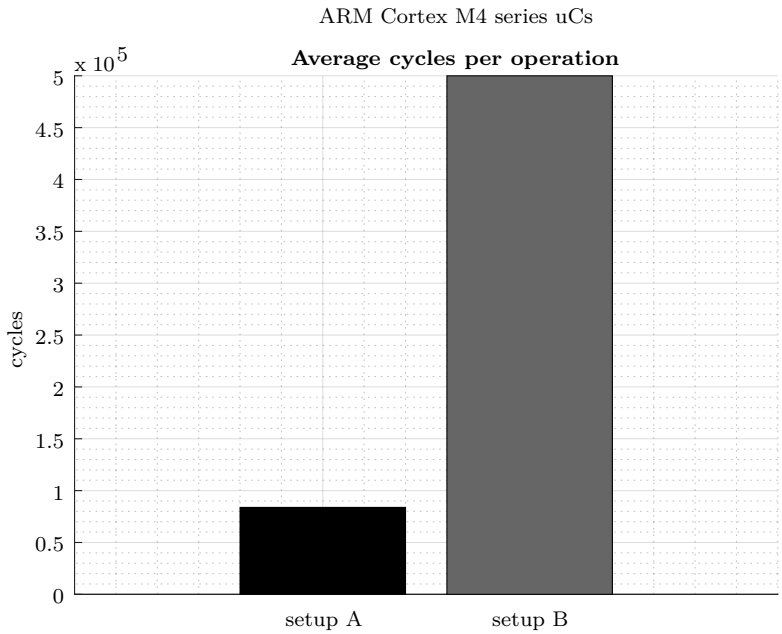


Figura 4.11: Cicli per operazione su microcontrollore.

Si osserva come, esattamente come su FPGA, il setup B sia molto più dispendioso del setup A, avendo un layer convolutivo in più e dovendo eseguire molti più calcoli sul secondo layer. Per quanto riguarda la velocità di calcolo il chip migliore è quello *high performance* grazie all'elevata frequenza di clock, come si può vedere in figura 4.13. I consumi energetici invece sono più bassi per il chip *ultra low power* che consuma intorno ai 16 µJ per operazione per il setup A e intorno ai 90 µJ per il setup B, come si può vedere in figura 4.14.

4.3.4 Comparazione dei risultati

I tempi di esecuzione sono di gran lunga minori per l'implementazione su FPGA che impiega fino a meno di un decimo del numero di cicli impiegati dal microcontrollore e in più garantisce frequenze di funzionamento superiori. I tempi su microcontrollore sono tanto lunghi poiché questo deve impiegare inevitabilmente degli array multidimensionali i cui tempi di accesso sono elevati.

Per quanto riguarda l'energia consumata per operazione, anche in questo caso l'FPGA si rivela vantaggiosa potendo arrivare a consumare meno di 5 µJ per il setup A e meno di 35 µJ per il setup B, ovvero meno della metà di quello che consuma

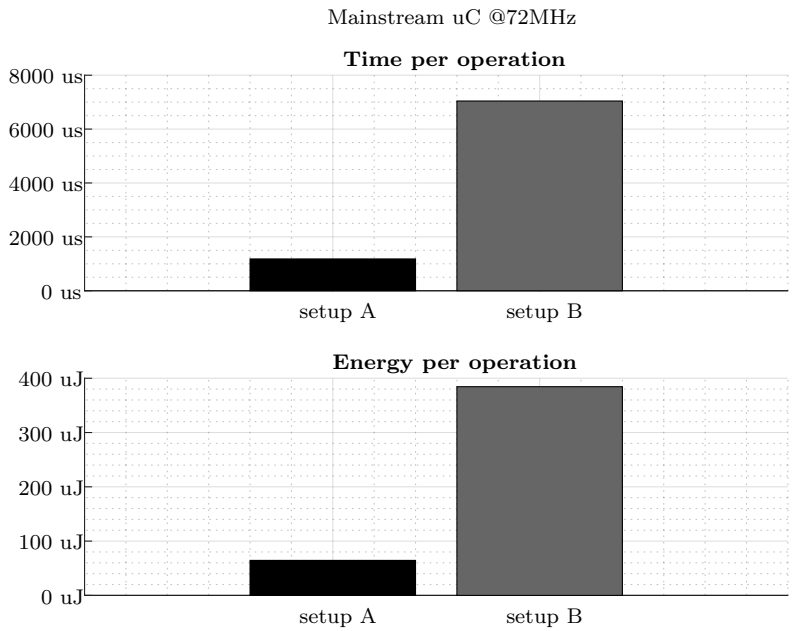


Figura 4.12: Tempo ed energia per operazione su microcontrollore *mainstream*.

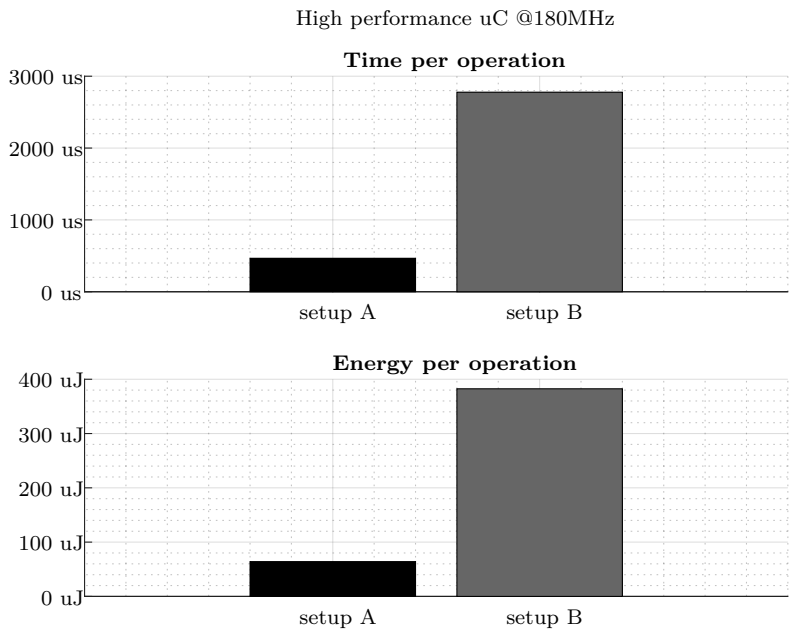


Figura 4.13: Tempo ed energia per operazione su microcontrollore *high performance*.

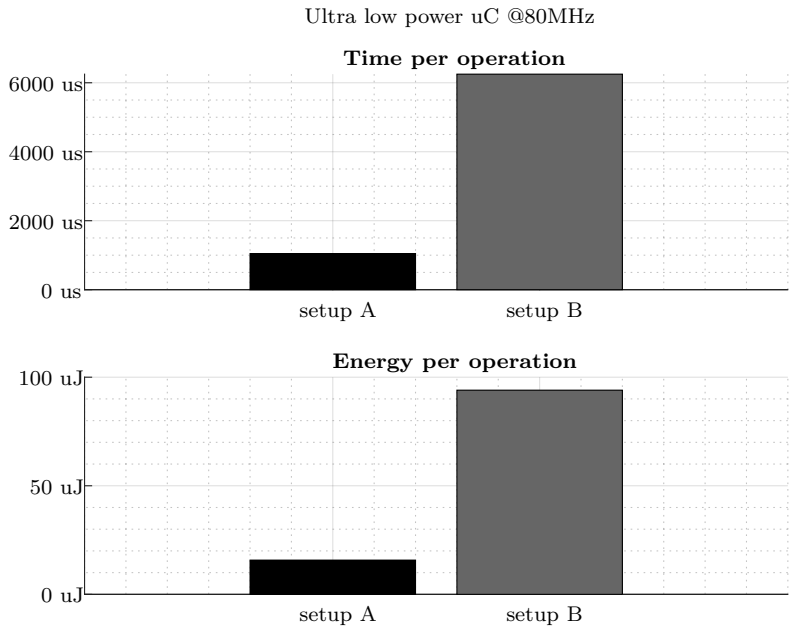


Figura 4.14: Tempo ed energia per operazione su microcontrollore *ultra low power*.

il migliore dei microcontrollori. Occorre notare tuttavia che se si utilizzasse un solo MAC nell'implementazione su FPGA si avrebbero risultati migliori su microcontrollore; i vantaggi dell'implementazione hardware derivano dunque in particolare dalla possibilità di parallelizzare le operazioni e di sfruttare dunque meglio l'energia consumata per tutte le operazioni non strettamente di calcolo.

Per quanto riguarda le constatazioni legate alla quantità di memoria, il parallelismo dei dati e il confronto dei prezzi, ci si riferisce al paragrafo 4.2.4.

4.4 Conclusioni

Con questo lavoro si è voluto progettare l'implementazione di una rete neurale su chip FPGA la cui generazione sia totalmente automatica e dettata a partire da una serie di parametri definibili da un utente qualsiasi. In questo modo un progettista che avesse bisogno in tempi brevi di avere un acceleratore hardware performante di questo tipo potrebbe optare per questa soluzione. Il tutto avviene sfruttando al massimo le potenzialità del codice VHDL e senza necessità di generare codice utilizzando software esterni al codice sorgente per l'architettura. Il progetto comprende l'implementazione di layer di tipo fully connected, max-pooling e convolutivi: le tre

tipologie utilizzano lo stesso codice durante la generazione disponendo della stessa struttura generica di base.

Una soluzione del genere è tuttavia in diretta concorrenza con un tipo di implementazione che si basa su microcontrollore: progettare in software è più facile e richiede meno tempo. Per questa ragione si è effettuato un confronto di prestazioni tra l'implementazione su FPGA e su microcontrollore per valutare se fosse opportuno impiegare la prima. La progettazione del software su microcontrollore è avvenuta cercando di ridurre al minimo operazioni che possano allungare i tempi di calcolo in maniera non necessaria. Il codice su microcontrollore viene generato dinamicamente sfruttando il preprocessore, in maniera simile all'architettura su FPGA: in questo modo risulta più veloce effettuare il confronto tra diverse tipologie di rete.

I risultati del confronto sono a favore dell'implementazione in hardware: i tempi di esecuzione sono molto più brevi e a livello di consumi energetici si hanno notevoli vantaggi; aggiuntivamente su FPGA si possono implementare reti più grandi. Il costo su FPGA è superiore ma non tale da comprometterne l'utilizzo per applicazioni che abbiano la stessa portata di quelle per microcontrollore.

I ridotti consumi energetici dell'implementazione su FPGA aprono la possibilità di impiego in ambito mobile del sistema, ad esempio per applicazioni in ambito biomedico oppure IoT, laddove reti estremamente accurate costruite da giganti dell'IT come Google o Microsoft non possono essere utilizzate a causa dei costi e delle dimensioni elevatissimi e dei consumi energetici di centinaia di kilo-watt. La possibilità di modificare le caratteristiche dell'architettura senza grandi sforzi inoltre garantisce la possibilità di una prototipazione veloce: può venire effettuato un elevato numero di test della rete neurale per ricercare la soluzione migliore ad un dato problema in termini di performance e accuratezza.

Sviluppi futuri

La natura di questo tipo di implementazione, che si può concretizzare in un numero pressoché infinito di diverse architetture, fa sì che ci sia un infinito numero di oggetti da testare per verificarne le prestazioni. Sarebbe utile effettuare altri test che si basino su diversi tipi di applicazioni per verificare quando questa implementazione garantisca dei maggiori vantaggi. Effettuare test su diverse tipologie di FPGA inoltre permetterebbe di capire quanto il codice sia effettivamente versatile. Per velocizzare e rendere più precisa la valutazione delle prestazioni potrebbe essere opportuno sfruttare la riprogrammabilità dei chip FPGA ed eseguire i test direttamente su

dispositivo; diversamente effettuare simulazioni per le valutazioni di potenza su reti neurali di grosse dimensioni potrebbe diventare estremamente lungo.

In questo lavoro si è testata la rete immaginando che questa venga utilizzata al massimo del suo throughput. Tuttavia i dati potrebbero anche arrivare al dispositivo dopo lunghi intervalli di tempo e per questo occorrerebbe valutare anche lo spegnimento del chip quando inutilizzato.

Sarebbe infine importante incrementare ulteriormente le funzionalità della rete neurale in modo tale da aumentare la tipologia di layer a disposizione, il numero di funzioni di attivazione e magari ottimizzare ulteriormente la struttura. In particolare sarebbe interessante l'implementazione di layer e funzioni di attivazione non lineari in modo tale che possano venire generate tutte le tipologie di rete che software come Matlab™ sono in grado di addestrare.

Appendice A

Codice completo per l'architettura su FPGA

A.1 Packages

Affinché si possa generare l'architettura in modo dinamico occorrono una serie di pacchetti di funzioni ausiliari.

A.1.1 Pacchetto per la gestione degli array di valori interi di dimensione indefinita (`generic_array_pkg.vhd`)

Nel codice vengono molto spesso impiegati array di valori interi di dimensione non definita a priori. Il pacchetto in questione ne dichiara il tipo e ne permette la gestione tramite funzioni che calcolano il valore massimo, la somma, il prodotto dei valori contenuti nell'array. Una funzione consente anche di incrementare tutti i valori di 1.

```
1  package generic_array_pkg is
2      type int_array is array (natural range <>) of integer;
3      -- maximum value of the array
4      function int_array_max(arg : int_array) return integer;
5      -- sum of all the values in the array
6      function int_array_sum(arg : int_array) return integer;
7      -- sum of the first n values of the array
8      function int_array_sum(arg : int_array; n : natural) return integer;
9      -- product of all the values in the array
10     function int_array_mul(arg : int_array) return integer;
11     -- increase all the elements of the array by 1
12     function int_array_incr(arg : int_array) return int_array;
13 end package generic_array_pkg;
14
```

```
15 package body generic_array_pkg is
16     function int_array_max(arg : int_array) return integer is
17         variable maximum : integer;
18     begin
19         maximum := 0;
20         for i in arg'left to arg'right loop
21             if(arg(i) > maximum)then
22                 maximum := arg(i);
23             end if;
24         end loop;
25         return maximum;
26     end function int_array_max;
27 
28     function int_array_sum(arg : int_array) return integer is
29         variable sum : integer;
30     begin
31         sum := 0;
32         for i in 0 to arg'length-1 loop
33             sum := sum + arg(i);
34         end loop;
35         return sum;
36     end function int_array_sum;
37 
38     function int_array_sum(arg : int_array; n : natural) return integer is
39         variable sum : integer := 0;
40     begin
41         if n > 0 then
42             for i in 0 to n-1 loop
43                 sum := sum + arg(i);
44             end loop;
45         end if;
46         return sum;
47     end function int_array_sum;
48 
49     function int_array_mul(arg : int_array) return integer is
50         variable mul : integer;
51     begin
52         mul := 1;
53         for i in 0 to arg'length-1 loop
54             mul := mul * arg(i);
55             report "mul␣=␣" & integer'image(mul) severity warning;
56         end loop;
57         return mul;
58     end function int_array_mul;
59 
60     function int_array_incr(arg : int_array) return int_array is
61         variable out_array  : int_array(0 to arg'length-1);
62     begin
63         for i in 0 to arg'length-1 loop
64             out_array(i) := arg(i)+1;
65         end loop;
66         return out_array;
67     end function int_array_incr;
68 end package body generic_array_pkg;
```

Listing A.1: Pacchetto per la gestione degli array di valori interi di dimensione indefinita (`generic_array_pkg.vhd`).

A.1.2 Pacchetto per la valutazione del numero di bit per la rappresentazione di un valore (`bit_evaluation_pkg.vhd`)

Questo pacchetto contiene funzioni per valutare il numero di bit necessari a rappresentare un certo intervallo di valori. Si può anche effettuare la valutazione della somma del numero di bit necessari a rappresentare una serie di valori contenuti in un array (utile qualora si abbiano dei sotto-indirizzi da assegnare ad una memoria, vedi il paragrafo 2.2.6). Le funzioni con denominazione "safe" non restituiscono valori inferiori ad 1 e garantiscono che non vengano generati dei vettori nulli durante la compilazione. Una serie di funzioni aggiuntive sono necessarie perché sono necessarie per la generazione di alcune strutture nel codice.

```
library ieee;
use ieee.math_real.all;
use work.generic_array_pkg.all;

package bit_evaluation_pkg is
    -- 'Safe' assures the output is > 0
    -- evaluate the number of bit needed to represent a value
    function bitEval(max_val : integer) return integer;
    function bitEvalSafe(max_val : integer) return integer;
    -- evaluate the number of bit needed to represent a value for each value of
    -- the int_array
    function bitEval(max_val : int_array) return int_array;
    function bitEvalSafe(max_val : int_array) return int_array;
    -- evaluate the number of bit needed to represent a value for each value of
    -- the int_array and sum them
    function bitEval_sum(max_val : int_array) return integer;
    function bitEvalSafe_sum(max_val : int_array) return integer;
    -- evaluate the number of bit needed to represent a value for n values of
    -- the int_array and sum them
    function bitEval_sum(max_val : int_array; n : natural) return integer;
    function bitEvalSafe_sum(max_val : int_array; n : natural) return integer;
    -- evaluate the ceil integer ratio
    function ceil_ratio(a, b : integer) return integer;
    -- evaluate the floor integer ratio
    function floor_ratio(a, b : integer) return integer;
    -- evaluate the maximum
    function max(a, b : integer) return integer;
    --evaluate maximum for each coupled values in the arrays
    function max(a, b : int_array) return int_array;
    -- evaluate the modulus of the difference
    function abs_diff(a, b : integer) return integer;
end package bit_evaluation_pkg;
```

```
33
34 package body bit_evaluation_pkg is
35     function bitEval(max_val : integer) return integer is
36         variable bit_num : integer;
37     begin
38         if(max_val <= 1)then
39             bit_num := 0;
40         else
41             bit_num := integer(ceil(log2(real(max_val))));
42         end if;
43         return bit_num;
44     end function bitEval;
45
46     function bitEvalSafe(max_val : integer) return integer is
47         variable bit_num : integer;
48     begin
49         bit_num := bitEval(max_val);
50         if bit_num = 0 then
51             bit_num := 1;
52         end if;
53         return bit_num;
54     end function bitEvalSafe;
55
56     function bitEval(max_val : int_array) return int_array is
57         variable bit_num : int_array(0 to max_val'length-1);
58     begin
59         for i in 0 to max_val'length-1 loop
60             bit_num(i) := bitEval(max_val(i));
61         end loop;
62         return bit_num;
63     end function bitEval;
64
65     function bitEvalSafe(max_val : int_array) return int_array is
66         variable bit_num : int_array(0 to max_val'length-1);
67     begin
68         for i in 0 to max_val'length-1 loop
69             bit_num(i) := bitEvalSafe(max_val(i));
70         end loop;
71         return bit_num;
72     end function bitEvalSafe;
73
74     function bitEval_sum(max_val : int_array) return integer is
75     begin
76         return int_array_sum(bitEval(max_val));
77     end function bitEval_sum;
78
79     function bitEvalSafe_sum(max_val : int_array) return integer is
80         variable sum    : integer := 0;
81     begin
82         sum := bitEval_sum(max_val);
83         if sum <= 0 then
84             sum := 1;
85         end if;
86         return sum;
87     end function bitEvalSafe_sum;
88
```

```
    function bitEval_sum(max_val : int_array; n : natural) return integer is
        variable sum    : integer := 0;
    begin
        return int_array_sum(bitEval(max_val), n);
    end function bitEval_sum;

    function bitEvalSafe_sum(max_val : int_array; n : natural) return integer is
        variable sum    : integer := 0;
    begin
        sum := bitEval_sum(max_val, n);
        if sum <= 0 then
            sum := 1;
        end if;
        return sum;
    end function bitEvalSafe_sum;

    function ceil_ratio(a, b : integer) return integer is
    begin
        if b /= 0 then
            return integer(ceil(real(a)/real(b)));
        else
            report "Error, division by 0." severity failure;
        end if;
    end function ceil_ratio;

    function floor_ratio(a, b : integer) return integer is
    begin
        if b /= 0 then
            return integer(floor(real(a)/real(b)));
        else
            report "Error, division by 0." severity failure;
        end if;
    end function floor_ratio;

    function max(a, b : integer) return integer is
    begin
        if(a > b)then
            return a;
        else
            return b;
        end if;
    end function max;

    function max(a, b : int_array) return int_array is
        variable res : int_array(0 to a'length-1);
    begin
        for i in 0 to a'length-1 loop
            if(a(i) > b(i))then
                res(i) := a(i);
            else
                res(i) := b(i);
            end if;
        end loop;
        return res;
    end function max;

```

```
    function abs_diff(a, b : integer) return integer is
        variable diff : integer;
    begin
        diff := a-b;
        if(diff < 0)then
            return -diff;
        else
            return diff;
        end if;
    end function abs_diff;
end package body bit_evaluation_pkg;
```

Listing A.2: Pacchetto per la valutazione del numero di bit per la rappresentazione di un valore (`bit_evaluation_pkg.vhd`).

A.1.3 Pacchetto per la gestione dei record contenenti i parametri dei layer (`layer_record_pkg.vhd`)

Le gestione dei parametri di ogni singolo layer avviene tramite l'utilizzo di un record. Un array di record di dimensione indefinita permette di istanziare un numero arbitrario di layer. Per maggiori dettagli riguardo ai vari parametri vedere il paragrafo 2.4.3.

```
use work.generic_array_pkg.all;

package layer_record_pkg is

    type typestr is array(1 to 4) of character;

    type layer_record is record
        layer   : typestr; -- fulc, conv or maxp
        neurons : integer;
        macs    : integer;
        r_field : int_array(0 to 1); -- (Fh, Fw)
        stride  : int_array(0 to 1); -- (h, v)
        padding : int_array(0 to 3); -- (t, b, l, r)
        a_func  : typestr; -- posl, satl or purl
    end record layer_record;

    type layer_array is array (natural range <>) of layer_record;

end package layer_record_pkg;
```

Listing A.3: Pacchetto per la gestione dei record contenenti i parametri dei layer (`layer_record_pkg.vhd`).

A.1.4 Pacchetto per la valutazione di parametri interni della rete neurale (`nn_utils_pkg.vhd`)

Durante la generazione della rete neurale sono necessarie una serie di funzioni per valutare il numero di output di un layer, il numero di pesi che un layer utilizza e simili.

```
library ieee;
use ieee.math_real.all;
use work.generic_array_pkg.all;
use work.layer_record_pkg.all;
use work.bit_evaluation_pkg.all;

package nn_utils_pkg is
    -- evaluate the input matrix dimensions for the n-th layer
    -- Narg : input matrix dimension
    -- Larg : layers array
    -- n : index of the layer; use layer_number+1 to evaluate the number of
    -- outputs of the last layer
    function eval_inputs(Narg : int_array; Larg : layer_array; n : integer)
        return int_array;
    -- evaluate the number of weights for each neuron (or, for pooling layers,
    -- the number of inputs to be elaborated to have a valid output)
    -- Narg : input matrix dimension
    -- Larg : layer record
    function eval_weights(Narg : int_array; Larg : layer_record) return integer;
    -- evaluate the maximum number of outputs between the layers
    -- Narg : input matrix dimension
    -- Larg : layers array
    function max_output(Narg : int_array; Larg : layer_array) return integer;

    -- evaluate the input connectivity matrix dimensions
    -- Narg : layer input matrix dimensions
    -- Larg : layer record
    function connect_matrix_dim(Narg : int_array; Larg : layer_record)
        return int_array;
    -- evaluate the number of layers with weights
    -- Narg : layer input matrix dimensions
    -- Larg : layer record
    function getLayersWithWeights(Larg : layer_array) return integer;
end package nn_utils_pkg;

package body nn_utils_pkg is
    function eval_inputs(Narg : int_array; Larg : layer_array; n : integer)
        return int_array is
        variable inmat : int_array(0 to 2);
        variable inmat_temp : int_array(0 to 2);
        variable F : int_array(0 to 1);
        variable P : int_array(0 to 3);
        variable S : int_array(0 to 1);
    begin
        inmat := Narg;
        if(n > 0)then
            --evaluate the outputs of previous layer, which is the input of the
```

```
48              --n-th layer
49              for i in 0 to n-1 loop
50                  F := Larg(i).r_field;
51                  P := Larg(i).padding;
52                  S := Larg(i).stride;
53                  if(Larg(i).layer = "fulc")then
54                      inmat_temp := (1, 1, Larg(i).neurons);
55                  elsif(Larg(i).layer = "conv")then
56                      inmat_temp := (
57                          ceil_ratio(inmat(0)-F(0)+P(0)+P(1), S(0))+1,
58                          ceil_ratio(inmat(1)-F(1)+P(2)+P(3), S(1))+1,
59                          Larg(i).neurons
60                          );
61                  elsif(Larg(i).layer = "maxp")then
62                      inmat_temp := (
63                          ceil_ratio(inmat(0)-F(0)+P(0)+P(1), S(0))+1,
64                          ceil_ratio(inmat(1)-F(1)+P(2)+P(3), S(1))+1,
65                          inmat(2)
66                          );
67                  else
68                      report "Error:␣wrong␣layer␣name." severity failure;
69                  end if;
70                  inmat := inmat_temp;
71              end loop;
72          end if;
73          report "layer␣" & integer'image(n) &
74                 "␣output␣:=␣(" & integer'image(inmat(0)) & ",␣"
75                                & integer'image(inmat(1)) & ",␣"
76                                & integer'image(inmat(2)) & ")" severity note;
77          return inmat;
78      end function eval_inputs;
79
80      function eval_weights(Narg : int_array; Larg : layer_record)
81          return integer is
82      begin
83          if(Larg.layer = "fulc")then
84              return int_array_mul(Narg);
85          elsif(Larg.layer = "conv")then
86              return int_array_mul(Larg.r_field)*Narg(2);
87          else
88              return int_array_mul(Larg.r_field);
89          end if;
90      end function;
91
92      function max_output(Narg : int_array; Larg : layer_array) return integer is
93          variable outputs : integer;
94          variable maximum : integer;
95      begin
96          maximum := 0;
97          for i in 0 to Larg'length loop
98              outputs := int_array_mul(eval_inputs(Narg, Larg, i));
99              if(outputs > maximum)then
100                 maximum := outputs;
101             end if;
102         end loop;
103         return maximum;
```

```
    end function max_output;

    function connect_matrix_dim(Narg : int_array; Larg : layer_record)
        return int_array is
    begin
        if Larg.layer = "fulc" then
            return Narg;
        else
            return (Larg.r_field(0), Larg.r_field(1), Narg(2));
        end if;
    end function connect_matrix_dim;

    function getLayersWithWeights(Larg : layer_array) return integer is
        variable res : integer := 0;
    begin
        for i in 0 to Larg'length-1 loop
            if(Larg(i).layer /= "maxp")then
                res := res + 1;
            end if;
        end loop;
        return res;
    end function getLayersWithWeights;
end package body nn_utils_pkg;
```

Listing A.4: Pacchetto per la gestione dei record contenenti i parametri dei layer (`nn_utils_pkg.vhd`).

A.2 Codice sorgente dell'architettura

Il codice sorgente che viene riportato comprende la descrizione dei vari blocchi che compongono la rete neurale. Non tutti i blocchi vengono inseriti perché molti di essi sono di implementazione banale e non è necessario approfondirli. Tra questi in particolare le memorie, i registri e i flip-flop.

A.2.1 Contatore (`counter.vhd`)

Il contatore è uno dei blocchi più importanti per il controllo dell'architettura. Viene riportata la versione `unsigned`, tuttavia ne esiste anche una `signed` che permette di scegliere il valore a cui inizializzare il contatore (la quale viene utilizzata per il *padding* nei layer convolutivi.) che non viene riportata qui.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;
use work.bit_evaluation_pkg.all;

-- ENTITY ==================================================================
```

```
8  entity counter is
9      generic
10     (
11         num_max : integer := 256;   --maximum number of counted values
12         incr    : integer := 1 --increase value
13     );
14     port
15     (
16         clk     : in std_logic;
17         rstn    : in std_logic;
18         enbl    : in std_logic;
19         tc      : out std_logic;
20         q       : out unsigned(bitEvalSafe(num_max)-1 downto 0)
21     );
22 end entity counter;
23
24 -- ARCHITECTURE ================================================================
25 architecture behavior of counter is
26     constant num_tc     : integer := floor_ratio(num_max-1, incr)*incr;
27     constant n_bits     : integer := bitEvalSafe(num_max);
28     signal q_int        : unsigned(n_bits-1 downto 0);
29     signal q_int_incr   : unsigned(n_bits-1 downto 0);
30 begin
31     proc_counter:
32     process(clk) begin
33         if(clk'event and clk = '1' and num_max > 1)then
34             if(rstn = '0')then
35                 q_int <= (others => '0');
36             elsif(enbl = '1')then
37                 if(to_integer(q_int) = num_tc)then
38                     q_int <= (others => '0');
39                 else
40                     q_int <= q_int_incr;
41                 end if;
42             end if;
43         end if;
44     end process;
45
46     proc_tc:
47     process(q_int) begin
48         if(to_integer(q_int) = num_tc) or (num_max <= 1)then
49             tc <= '1';
50         else
51             tc <= '0';
52         end if;
53     end process;
54
55     goutput: if num_max > 1 generate
56         q_int_incr <= q_int + to_unsigned(incr, n_bits);
57         q <= q_int;
58     end generate;
59 end architecture behavior;
```

Listing A.5: File di descrizione del contatore (`counter.vhd`).

A.2.2 Contatore "multidimensionale" (`counter_multid.vhd`)

Per l'indirizzamento delle memorie che contengono matrici multidimensionali vengono utilizzati blocchi che contengono insiemi di contatori adibiti rispettivamente all'indicizzazione di righe, colonne, piani. Viene riportata la versione `unsigned` nonostante esista anche una versione `signed` che qui non viene riportata; notare che la versione `signed` ha un bit in più per ciascuno dei contatori perché deve tenere conto dei bit di segno. Per maggiori dettagli riferirsi al paragrafo 2.2.6.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use ieee.math_real.all;
5  use work.bit_evaluation_pkg.all;
6  use work.generic_array_pkg.all;
7
8  -- ENTITY ==================================================================
9  entity counter_multid is
10     generic
11     (
12         -- dimensions of the Ndimensional counter
13         num_max : int_array := (24, 6, 12);
14         -- increase amount for each counter
15         stride  : int_array := (1, 1, 1);
16         -- output order
17         order   : int_array := (0, 1, 2)
18     );
19     port
20     (
21         clk     : in std_logic;
22         rstn    : in std_logic;
23         enbl    : in std_logic := '0'; --enable for the first counter
24         enbls   : in std_logic_vector(num_max'length-1 downto 0)
25                                                          := (others => '0');
26         tc      : out std_logic; --tc for the first counter
27         tcs     : out std_logic_vector(num_max'length-1 downto 0);
28         q       : out unsigned(bitEvalSafe_sum(num_max)-1 downto 0)
29     );
30 end entity counter_multid;
31
32 -- ARCHITECTURE ============================================================
33 architecture behavior of counter_multid is
34     signal enbls_int    : std_logic_vector(num_max'length-1 downto 0);
35         signal tcs_int          : std_logic_vector(num_max'length-1 downto 0);
36     signal q_int        : unsigned(bitEvalSafe_sum(num_max)-1 downto 0);
37 begin
38     -- Counters generation
39     gcounters: for i in 0 to num_max'length-1 generate
40         gcounter: if num_max(i) > 1 generate
41             icounter:entity work.counter
42             generic map(
43                 num_max => num_max(i),
44                 incr => stride(i)
45             )
```

```
46              port map(
47                  clk => clk,
48                  rstn => rstn,
49                  enbl => enbls_int(i),
50                  tc => tcs_int(i),
51                  q => q_int(bitEval_sum(num_max, i+1)-1 downto
52                             bitEval_sum(num_max, i))
53              );
54          end generate;
55          gnot_counter: if num_max(i) <= 1 generate
56              tcs_int(i) <= '1';
57          end generate;
58      end generate;
59
60      -- Output reordering, based on the order generic parameter
61      porder: process(q_int)
62          constant n          : integer := num_max'length;
63          variable posfrom    : int_array(0 to n-1);
64          variable posto      : int_array(0 to n-1);
65          variable rangesto   : int_array(0 to n-1);
66          variable rangesfrom : int_array(0 to n-1);
67      begin
68          rangesfrom := bitEval(num_max); --ranges of original vectors
69          posto(0) := 0;
70          for i in 0 to n-1 loop
71              rangesto(i) := rangesfrom(order(i)); --ranges of target vectors
72              if i > 0 then --generate the position of target vectors
73                  posto(i) := posto(i-1) + rangesto(i-1);
74              end if;
75          end loop;
76          for i in 0 to n-1 loop
77              posfrom(i) := 0;
78              if order(i) > 0 then
79                  for j in 1 to order(i) loop --generate the pos. of original vec.
80                      posfrom(i) := posfrom(i) + rangesfrom(j-1);
81                  end loop;
82              end if;
83              if(rangesto(i) > 0)then --check if not null range
84                  q(posto(i)+rangesto(i)-1 downto posto(i)) <=
85                      q_int(posfrom(i)+rangesto(i)-1 downto posfrom(i));
86              end if;
87          end loop;
88      end process;
89
90      -- The enable of each counter is activated if it's active an enable of a
91      -- less significal counter and all the terminal counts of the lower counters
92      -- or if it's activated the enable signal of the actual counter.
93      penbls: process(enbl, enbls, enbls_int, tcs_int)
94      begin
95          enbls_int(0) <= enbls(0) or enbl;
96          for i in 1 to num_max'length-1 loop
97              enbls_int(i) <= (enbls_int(i-1) and tcs_int(i-1)) or enbls(i);
98          end loop;
99      end process;
100
101         -- terminal count generation
```

```
    tcs <= tcs_int;
    ptcs: process(tcs_int)
        variable tc_temp : std_logic_vector(num_max'length-1 downto 0);
    begin
        tc_temp(0) := tcs_int(0);
        for i in 1 to num_max'length-1 loop
            tc_temp(i) := tc_temp(i-1) and tcs_int(i);
        end loop;
        tc <= tc_temp(num_max'length-1);
    end process;
end architecture behavior;
```

Listing A.6: File di descrizione del contatore "multidimensionale" per la generazione di sotto-indirizzi (`counter_multid.vhd`).

A.2.3 Funzione di attivazione (`activation.vhd`)

Questo blocco molto semplicemente implementa le funzioni di attivazione. È sufficiente utilizzare un `process` per farlo perché si tratta di funzioni lineari. Il process si occupa anche di selezionare il tipo di funzione di attivazione da impiegare. Per maggiori dettagli riferirsi al paragrafo 2.2.2.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;
use work.layer_record_pkg.all;

-- ENTITY ==================================================================
entity activation is
    generic
    (
        n_tot   : integer := 8; --total number of bits
        n_frac  : integer := 5; --number of fractional bits
        a_func  : typestr := "purl" --type of activation function
    );
    port
    (
        x       : in signed(n_tot-1 downto 0);
        y       : out signed(n_tot-1 downto 0)
    );
end entity activation;

-- ARCHITECTURE ============================================================
architecture struct of activation is
begin
    pactivation:
    process(x)
    begin
        if(a_func = "posl")then     -- positive linear function (ReLU)
            if(x <= 0)then
                y <= (others => '0');
```

```
31              else
32                  y <= x;
33              end if;
34          elsif(a_func = "satl")then  -- saturated linear function
35              if(x <= 0)then
36              y <= (others => '0');
37              elsif (x >= 2**n_frac)then --2**n_frac is 1 in fixed point notation
38                  y <= (others => '0');
39                  y(n_frac) <= '1';
40              else                    -- pure linear function
41                  y <= x;
42              end if;
43          else
44              y <= x;
45          end if;
46      end process;
47  end architecture struct;
```

Listing A.7: File di descrizione del blocco funzione di attivazione (`activation.vhd`).

A.2.4 Multiply and accumulate (`mac.vhd`)

Questo è il blocco Multiply and Accumulate (MAC), così come viene descritto nel paragrafo 2.2.1.

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4   use work.bit_evaluation_pkg.all;
5
6   -- ENTITY ==================================================================
7   entity mac is
8       generic
9       (
10          a_n_tot     : integer := 8;   --total number of bits of input A
11          a_n_frac    : integer := 5;   --number of fractional bits of input A
12          b_n_tot     : integer := 8;   --total number of bits of input B
13          b_n_frac    : integer := 5    --number of fractional bits of input B
14      );
15      port
16      (
17          clk         : in std_logic;
18          rstn        : in std_logic;
19          a           : in signed(a_n_tot-1 downto 0);
20          b           : in signed(b_n_tot-1 downto 0);
21          enbl        : in std_logic;
22          bypass      : in std_logic;
23          q           : out signed(a_n_tot+b_n_tot-1 downto 0)
24      );
25  end entity mac;
26
27  -- ARCHITECTURE =============================================================
28  architecture behavior of mac is
```

```
29      -- Note: signals followed by dX (Where X is a number) are delayed signals
30      constant tot_n_tot  : integer := a_n_tot+b_n_tot;
31      constant tot_n_frac : integer := a_n_frac+b_n_frac;
32      constant a_n_int    : integer := a_n_tot-a_n_frac;
33      constant b_n_int    : integer := b_n_tot-b_n_frac;
34      constant max_n_frac : integer := max(a_n_frac, b_n_frac);
35      constant max_n_int  : integer := max(a_n_int, b_n_int);
36      constant max_n_tot  : integer := max_n_int+max_n_frac;
37      constant diff_n_int : integer := abs_diff(a_n_int, b_n_int);
38      constant diff_n_frac: integer := abs_diff(a_n_frac, b_n_frac);
39
40      signal enblOrBypass : std_logic;
41      signal a_d0         : signed(a_n_tot-1 downto 0);
42      signal b_d0         : signed(b_n_tot-1 downto 0);
43      signal a_d0_ext     : signed(max_n_tot-1 downto 0);
44      signal b_d0_ext     : signed(max_n_tot-1 downto 0);
45      signal b_d1         : signed(b_n_tot-1 downto 0);
46      signal b_bypass     : signed(tot_n_tot-1 downto 0);
47      signal mul_ext      : signed(2*max_n_tot-1 downto 0);
48      signal mul          : signed(tot_n_tot-1 downto 0);
49      signal mul_d0       : signed(tot_n_tot-1 downto 0);
50      signal sum          : signed(tot_n_tot-1 downto 0);
51      signal acc_in       : signed(tot_n_tot-1 downto 0);
52      signal acc_out      : signed(tot_n_tot-1 downto 0);
53      signal enbl_d0      : std_logic;
54      signal enbl_d1      : std_logic;
55      signal bypass_d0    : std_logic;
56      signal bypass_d1    : std_logic;
57  begin
58      -- Pipe0 flip flops -------------------------------------------------------
59      enblOrBypass <= enbl or bypass;
60      ienbl_d0:entity work.ff port map(enblOrBypass, clk, rstn, enbl_d0);
61      ibypass_d0:entity work.ff port map(bypass, clk, rstn, bypass_d0);
62
63      -- Pipe0 registers --------------------------------------------------------
64      ia_d0:entity work.sreg generic map(a_n_tot)
65      port map(a, clk, rstn, enblOrBypass, a_d0);
66      ib_d0:entity work.sreg generic map(b_n_tot)
67      port map(b, clk, rstn, enblOrBypass, b_d0);
68
69      -- Multiply ================================================================
70      -- The values must be aligned
71      -- example
72      -- a    - - - - - o o o.o o o o o o - *
73      -- b    - - - o o o o o.o - - - - - - =
74      -- q    o o o o o o o o.o o o o o o o
75      palign_values: process(a_d0, b_d0)
76      begin
77          if(max_n_frac < max_n_tot)then -- extend sign bit
78              a_d0_ext(max_n_tot-1 downto max_n_frac) <=
79                                              (others => a_d0(a_n_tot-1));
80              b_d0_ext(max_n_tot-1 downto max_n_frac) <=
81                                              (others => b_d0(b_n_tot-1));
82          end if;
83          if(max_n_frac > 0)then -- extend lsb
84              a_d0_ext(max_n_frac-1 downto 0) <= (others => '0');
```

```
85              b_d0_ext(max_n_frac-1 downto 0) <= (others => '0');
86          end if;
87          -- assign values
88          a_d0_ext(a_n_int+max_n_frac-1 downto max_n_frac-a_n_frac) <= a_d0;
89          b_d0_ext(b_n_int+max_n_frac-1 downto max_n_frac-b_n_frac) <= b_d0;
90      end process;
91      -- multiply
92      mul_ext <= a_d0_ext*b_d0_ext;
93      -- truncate the msb of the result
94      mul <= mul_ext(2*max_n_tot-diff_n_int-1 downto diff_n_frac);
95
96      -- Pipe1 flip flops -------------------------------------------------------
97      ienbl_d1:entity work.ff port map(enbl_d0, clk, rstn, enbl_d1);
98      ibypass_d1:entity work.ff port map(bypass_d0, clk, rstn, bypass_d1);
99
100     -- Pipe1 registers --------------------------------------------------------
101     imul_d0:entity work.sreg generic map(tot_n_tot)
102     port map(mul, clk, rstn, enbl_d0, mul_d0);
103     ib_d1:entity work.sreg generic map(b_n_tot)
104     port map(b_d0, clk, rstn, enbl_D0, b_d1);
105
106     -- Sum ==================================================================
107     sum <= mul_d0 + acc_out;
108
109     -- Accumulate ===============================================================
110     pbypass: -- bypass value must be aligned to the output
111     process(b_d1)
112     begin
113         b_bypass(tot_n_tot-1 downto tot_n_frac) <= (others => b_d1(b_n_tot-1));
114         b_bypass(tot_n_frac-1 downto 0) <= (others => '0');
115         b_bypass(tot_n_frac+b_n_tot-b_n_frac-1 downto tot_n_frac-b_n_frac) <=
116                                                                     b_d1;
117     end process pbypass;
118
119     -- choose whether to use the sum value or the bypass value
120     with bypass_d1 select acc_in <= b_bypass when '1', sum when others;
121
122     -- accumulator register
123     iaccreg:entity work.sreg generic map(tot_n_tot)
124     port map(acc_in, clk, rstn, enbl_d1, acc_out);
125
126     -- Output ===================================================================
127     q <= acc_out;
128 end architecture behavior;
```

Listing A.8: File di descrizione del multiply and accumulate (`mac.vhd`).

A.2.5 Gruppo MAC e memoria dei pesi (`mac_group.vhd`)

Questo è il blocco che contiene il MAC, la memoria dei pesi e il registro a valle del MAC in cui vengono salvati i risultati temporanei. La sua descrizione si trova nel paragrafo 2.2.2.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use ieee.math_real.all;
5  use work.bit_evaluation_pkg.all;
6
7  -- ENTITY ==================================================================
8  entity mac_group is
9      generic
10     (
11         d_n_tot     : integer := 8;     --data total number of bits
12         d_n_frac    : integer := 5;     --data number of fractionary bits
13         w_n_tot     : integer := 8;     --weight total number of bits
14         w_n_frac    : integer := 5;     --weight number of fractionary bits
15         Wj          : integer := 8;     --layer inputs number
16         neurons_num : integer := 8      --neurons assigned to this MAC
17     );
18     port
19     (
20         clk         : in std_logic;
21         rstn        : in std_logic;
22
23         -- I/O
24         input       : in signed(d_n_tot-1 downto 0);
25         output      : out signed(d_n_tot-1 downto 0);
26         weight_in   : in signed(w_n_tot-1 downto 0);
27
28         -- Addresses
29         weight_write_addr, weight_read_addr
30                     : in unsigned(bitEvalSafe((Wj+1)*neurons_num)-1 downto 0);
31
32         -- Control signals
33         MAC_enbl    : in std_logic;
34         MAC_bypass  : in std_logic;
35         write_output: in std_logic;
36         write_weight: in std_logic
37     );
38 end entity mac_group;
39
40 -- ARCHITECTURE ============================================================
41 architecture behavior of mac_group is
42     -- Note: signals followed by dX (Where X is a number) are delayed signals
43     constant mac_n_tot      : integer := d_n_tot+w_n_tot;
44     constant mac_n_frac     : integer := d_n_frac+w_n_frac;
45
46     signal weight_to_MAC    : signed(w_n_tot-1 downto 0);
47     signal MAC_output       : signed(mac_n_tot-1 downto 0);
48     signal MAC_output_trunc : signed(d_n_tot-1 downto 0);
49     signal write_output_d0  : std_logic;
50     signal write_output_d1  : std_logic;
51 begin
52     -- Instantiation of the MAC ==============================================
53     iMAC:entity work.mac
54     generic map(
55         a_n_tot => d_n_tot,
```

```
56          a_n_frac => d_n_frac,
57          b_n_tot => w_n_tot,
58          b_n_frac => w_n_frac
59      )
60      port map(
61          clk => clk,
62          rstn => rstn,
63          a => input,
64          b => weight_to_MAC,
65          enbl => MAC_enbl,
66          bypass => MAC_bypass,
67          q => MAC_output
68      );
69      -- Truncate the output to d_n_tot bits from 2*d_n_tot bits
70      MAC_output_trunc <=
71          MAC_output(mac_n_frac+d_n_tot-d_n_frac-1 downto mac_n_frac-d_n_frac);
72
73      -- Register used to save the MAC result ====================================
74      ioutput_reg:entity work.sreg generic map(d_n_tot)
75      port map(MAC_output_trunc, clk, rstn, write_output_d1, output);
76
77      -- Bias/Weights Memory ======================================================
78      -- NOTE: values are saved as "b w w w w ... b w w w w ... b w w w w ..."
79      iweights_mem:entity work.mem
80      generic map(bitEvalSafe((Wj+1)*neurons_num), w_n_tot)
81      port map(clk, weight_in, write_weight, weight_read_addr, weight_write_addr,
82          weight_to_MAC);
83
84      -- Delay registers ==========================================================
85      iwrite_output_d0:entity work.ff
86      port map(write_output, clk, rstn, write_output_d0);
87      iwrite_output_d1:entity work.ff
88      port map(write_output_d0, clk, rstn, write_output_d1);
89  end architecture behavior;
```

Listing A.9: File di descrizione del gruppo MAC con memoria dei pesi e registro di output (`mac_group.vhd`).

A.2.6 Gruppo di neuroni (`neurons.vhd`)

Questo blocco contiene i blocchi `mac_group` in parallelo e il blocco per la funzione di attivazione. La sua descrizione si trova nel paragrafo 2.2.2.

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4   use ieee.math_real.all;
5   use work.bit_evaluation_pkg.all;
6   use work.layer_record_pkg.all;
7
8   -- ENTITY ==================================================================
9   entity neurons is
10      generic
```

```
11      (
12          d_n_tot         : integer := 8; --data total number of bits
13          d_n_frac        : integer := 5; --data number of fractionary bits
14          w_n_tot         : integer := 8; --weight total number of bits
15          w_n_frac        : integer := 5; --weight number of fractionary bits
16          Wj              : integer := 8; --layer weights per neuron number
17          Sj              : integer := 8; --layer neurons number
18          Mj              : integer := 2;  --layer MACs number
19          a_func          : typestr := "purl" --activation function
20      );
21      port
22      (
23          clk             : in std_logic;
24          rstn            : in std_logic;
25          -- I/O
26          input           : in signed(d_n_tot-1 downto 0);
27          output          : out signed(d_n_tot-1 downto 0);
28          weight_in       : in signed(w_n_tot-1 downto 0);
29          -- Addresses
30          weight_addr     : in unsigned(bitEvalSafe(Wj)-1 downto 0);
31          iterat_addr     : in unsigned(bitEvalSafe(ceil_ratio(Sj,Mj))-1 downto 0);
32          MAC_addr        : in unsigned(bitEvalSafe(Mj)-1 downto 0);
33          -- Control signals
34          MAC_enbl        : in std_logic;
35          MAC_bypass      : in std_logic;
36          write_output    : in std_logic;
37          write_weight    : in std_logic;
38          weight1_bias0   : in std_logic;
39          last_row        : in std_logic
40      );
41  end entity neurons;
42
43  -- ARCHITECTURE ================================================================
44  architecture behavior of neurons is
45      -- Note: signals followed by dX (Where X is a number) are delayed signals
46      -- Note: signals marked as "woln" (without last neuron) are needed to avoid
47      --       that the MACs work when they don't need to (reduced power
48      --       consumption).
49      type MAC_array_type is array (0 to Mj-1) of signed(d_n_tot-1 downto 0);
50      constant SMrat               : integer := ceil_ratio(Sj, Mj);
51      constant Wtot                : integer := (Wj+1)*SMrat;
52      constant Wtot_woln           : integer := (Wj+1)*(SMrat-1);
53
54      signal input_d0              : signed(d_n_tot-1 downto 0);
55      signal weight_in_d0          : signed(w_n_tot-1 downto 0);
56      signal weight_addr_d0        : unsigned(bitEvalSafe(Wj)-1 downto 0);
57      signal iterat_addr_mul       : unsigned(bitEvalSafe(Wtot)-1 downto 0);
58      signal iterat_addr_mul_d0    : unsigned(bitEvalSafe(Wtot)-1 downto 0);
59      signal MAC_enbl_d0           : std_logic;
60      signal MAC_bypass_d0         : std_logic;
61      signal MAC_output_array      : MAC_array_type;
62      signal activation_input      : signed(d_n_tot-1 downto 0);
63      signal activation_input_d0   : signed(d_n_tot-1 downto 0);
64      signal write_weight_vec      : std_logic_vector(Mj-1 downto 0);
65      signal write_weight_vec_d0   : std_logic_vector(Mj-1 downto 0);
66      signal MAC_enbl_woln         : std_logic;
```

```
67      signal write_output_d0       : std_logic;
68      signal last_row_d0           : std_logic;
69      signal weight1_bias0_d0      : std_logic;
70      signal weight_tot_addr       : unsigned(bitEvalSafe(Wtot)-1 downto 0);
71      signal weight_tot_addr_woln : unsigned(bitEvalSafe(Wtot_woln)-1 downto 0);
72  begin
73      -- Generation of the MAC groups ==========================================
74      -- Each mac group has already its own weights memory. Whether the value
75      -- Sj/Mj is not integer, while computing the last neuron outputs not all the
76      -- MACs will work. For example if Sj = 5 and Mj = 3, first the three neurons
77      -- will generate the outputs 1, 2 and 3, then only Sj mod Mj = 2 MACs will
78      -- work to generate outputs 4 and 5.
79      -- Last_row is the signals which tells us when we are using the last neurons
80      gmac_groups:
81      for i in 0 to Mj-1 generate
82          gwith_last_neuron:
83          if (i < Sj mod Mj) or (Sj mod Mj = 0) generate
84              imac_group:entity work.mac_group
85              generic map(d_n_tot, d_n_frac, w_n_tot, w_n_frac, Wj, SMrat)
86              port map(
87                  clk => clk,
88                  rstn => rstn,
89                  input => input_d0,
90                  weight_write_addr => weight_tot_addr,
91                  weight_read_addr => weight_tot_addr,
92                  weight_in => weight_in_d0,
93                  write_weight => write_weight_vec_d0(i),
94                  MAC_enbl => MAC_enbl_d0,
95                  MAC_bypass => MAC_bypass_d0,
96                  write_output => write_output_d0,
97                  output => MAC_output_array(i)
98              );
99          end generate;
100         gwithout_last_neuron:
101         if (i >= Sj mod Mj) and (Sj mod Mj /= 0) generate
102             imac_group_woln:entity work.mac_group
103             generic map(d_n_tot, d_n_frac, w_n_tot, w_n_frac, Wj, SMrat-1)
104             port map(
105                 clk => clk,
106                 rstn => rstn,
107                 input => input_d0,
108                 weight_write_addr => weight_tot_addr_woln,
109                 weight_read_addr => weight_tot_addr_woln,
110                 weight_in => weight_in_d0,
111                 write_weight => write_weight_vec_d0(i),
112                 MAC_enbl => MAC_enbl_woln,
113                 MAC_bypass => MAC_bypass_d0,
114                 write_output => write_output_d0,
115                 output => MAC_output_array(i)
116             );
117         end generate;
118     end generate;
119
120     -- Gating the MACs not useful for the last output generation
121     ginput_gating:
122     if Sj mod Mj /= 0 generate --generate if there are woln MACs
```

```
123         MAC_enbl_woln <= MAC_enbl_d0 and not last_row_d0;
124         weight_tot_addr_woln <= weight_tot_addr(bitEvalSafe(Wtot_woln)-1
125                                                                 downto 0);
126     end generate;
127
128     -- MAC output multiplexer ==================================================
129     gmultiple_MACs_output:
130     if Mj > 1 generate
131         activation_input <= MAC_output_array(to_integer(MAC_addr));
132     end generate;
133     gsingle_MAC_output:
134     if Mj = 1 generate
135         activation_input <= MAC_output_array(0);
136     end generate;
137
138     -- Activation function ======================================================
139     iactivation:entity work.activation
140     generic map(d_n_tot, d_n_frac, a_func)
141     port map(activation_input_d0, output);
142
143     -- Weight address evaluation ================================================
144     -- weight_tot_addr <= iterat_addr*(Wj+1) + weight_addr (carry_in = w1_b0)
145     iterat_addr_mul <=
146         to_unsigned((Wj+1)*to_integer(iterat_addr), bitEvalSafe((Wj+1)*SMrat));
147
148     pweight_write_addr:
149     process(weight_addr_d0, iterat_addr_mul_d0, weight1_bias0_d0)
150         variable weight_addr_ext       : unsigned(bitEvalSafe(Wtot)-1 downto 0);
151         variable weight_addr_ext_lsb   : unsigned(bitEvalSafe(Wtot) downto 0);
152         variable iterat_addr_mul_lsb   : unsigned(bitEvalSafe(Wtot) downto 0);
153         variable weight_tot_addr_lsb   : unsigned(bitEvalSafe(Wtot) downto 0);
154     begin
155         weight_addr_ext := (others => '0');
156         weight_addr_ext(bitEvalSafe(Wj)-1 downto 0) := weight_addr_d0;
157         weight_addr_ext_lsb := weight_addr_ext & weight1_bias0_d0;
158         iterat_addr_mul_lsb := iterat_addr_mul_d0 & '1';
159         weight_tot_addr_lsb := iterat_addr_mul_lsb + weight_addr_ext_lsb;
160         weight_tot_addr <= weight_tot_addr_lsb(bitEvalSafe(Wtot) downto 1);
161     end process;
162
163     -- Address decoder ==========================================================
164     -- MAC address to weight/bias register write decoding with enable
165     proc_MAC_addr_to_write:
166     process(write_weight, MAC_addr)
167     begin
168         write_weight_vec <= (others => '0');
169         write_weight_vec(to_integer(MAC_addr)) <= write_weight;
170     end process;
171
172     -- Delay registers ==========================================================
173     iactivation_input_d0:entity work.sreg generic map(d_n_tot)
174     port map(activation_input, clk, '1', rstn, activation_input_d0);
175
176     iweight_addr_d0:entity work.ureg generic map(bitEvalSafe(Wj))
177     port map(weight_addr, clk, '1', rstn, weight_addr_d0);
178
```

```
179     iiterat_addr_mul_d0:entity work.ureg generic map(bitEvalSafe(Wtot))
180     port map(iterat_addr_mul, clk, '1', rstn, iterat_addr_mul_d0);
181
182     iweight1_bias0_d0:entity work.ff
183     port map(weight1_bias0, clk, rstn, weight1_bias0_d0);
184
185     iwrite_output_d0:entity work.ff
186     port map(write_output, clk, rstn, write_output_d0);
187
188     glast_row_d0: if Sj mod Mj /= 0 generate --generate if there are woln MACs
189         ilast_row_d0:entity work.ff
190         port map(last_row, clk, rstn, last_row_d0);
191     end generate;
192
193     iinput_d0:entity work.sreg generic map(d_n_tot)
194     port map(input, clk, MAC_enbl, rstn, input_d0);
195
196     iweight_in_d0:entity work.sreg generic map(w_n_tot)
197     port map(weight_in, clk, write_weight, rstn, weight_in_d0);
198
199     iwrite_weight_vec_d0:entity work.reg generic map(Mj)
200     port map( write_weight_vec, clk, '1', rstn, write_weight_vec_d0);
201
202     iMAC_enbl_d0:entity work.ff
203     port map(MAC_enbl, clk, rstn, MAC_enbl_d0);
204
205     iMAC_bypass_d0:entity work.ff
206     port map(MAC_bypass, clk, rstn, MAC_bypass_d0);
207
208 end architecture behavior;
```

Listing A.10: File di descrizione del gruppo di neuroni (`neurons.vhd`).

A.2.7 Blocco per il calcolo del valore massimo (`max.vhd`)

Questo blocco viene impiegato dai layer max-pooling per il calcolo del valore massimo in un gruppo di input. La sua descrizione si trova nel paragrafo 2.2.3

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  -- ENTITY ==================================================================
6  entity max is
7      generic
8      (
9          n_tot   : integer := 8  --total number of bits of input
10     );
11     port
12     (
13         clk         : in std_logic;
14         rstn        : in std_logic; --this sets the acc to the lowest signed
15         input       : in signed(n_tot-1 downto 0);
```

```
16          enbl        : in std_logic;
17          init        : in std_logic; --initialization signal
18          write_out   : in std_logic;
19          set_pos     : in std_logic; --if acc is negative, set to 0
20          q           : out signed(n_tot-1 downto 0)
21      );
22  end entity max;
23
24  -- ARCHITECTURE ==============================================================
25  architecture behavior of max is
26      -- Note: signals followed by dX (Where X is a number) are delayed signals
27      signal input_enbl   : std_logic;
28      signal input_d0     : signed(n_tot-1 downto 0);
29      signal enbl_d0      : std_logic;
30      signal init_d0      : std_logic;
31      signal set_pos_d0   : std_logic;
32      signal write_out_d0 : std_logic;
33      signal is_max       : std_logic;
34      signal save_max     : std_logic;
35      signal maximum      : signed(n_tot-1 downto 0);
36      signal to_acc       : signed(n_tot-1 downto 0);
37      signal acc_rstn     : std_logic;
38  begin
39      -- Pipe0 registers and ff ---------------------------------------------------
40      input_enbl <= enbl and not init;
41
42      iinput_d0:entity work.sreg generic map(n_tot)
43      port map(input, clk, rstn, input_enbl, input_d0);
44
45      ienbl_d0:entity work.ff port map(enbl, clk, rstn, enbl_d0);
46      iinit_d0:entity work.ff port map(init, clk, rstn, init_d0);
47      iset_pos_d0:entity work.ff port map(set_pos, clk, rstn, set_pos_d0);
48      iwrite_out_d0:entity work.ff port map(write_out, clk, rstn, write_out_d0);
49
50      -- Evaluation of the maximum ==============================================
51      is_max <= '1' when input_d0 > maximum else '0';
52      save_max <= (is_max and enbl_d0) or init_d0;
53
54      -- Accumulator
55      imaximum:entity work.sreg generic map(n_tot)
56      port map(to_acc, clk, acc_rstn, save_max, maximum);
57
58      -- Accumulator initialization
59      preset_acc:
60      process(init_d0, input_d0)
61          variable minimum    : signed(n_tot-1 downto 0);
62      begin
63          -- Evaluate the minimum possible value
64          minimum := (others => '0');
65          minimum(n_tot-1) := '1';
66          -- Assign the minimum value when resetting
67          if(init_d0 = '1')then
68              to_acc <= minimum;
69          else
70              to_acc <= input_d0;
71          end if;
```

```
    end process;

    -- Reset the accumulator  register if the value it holds is negative, when
    -- the signal set_pos is asserted.
    acc_rstn <= not(maximum(n_tot-1)and set_pos_d0) and rstn;

    -- Output register
    ioutput:entity work.sreg generic map(n_tot)
    port map(maximum, clk, rstn, write_out_d0, q);
end architecture behavior;
```

Listing A.11: File di descrizione del blocco per il calcolo del valore massimo (`max.vhd`).

A.2.8 Datapath del layer (`datapath.vhd`)

Il datapath contiene il blocco principale di elaborazione, che può essere il blocco `max` oppure `neurons`, e una serie di contatori che vengono utilizzati dalle macchina a stati per gestire la rete. Viene fatto largo uso del costrutto `generate` per modificare la struttura del layer a seconda della sua tipologia e dei suoi parametri. Un'immagine di questo blocco si trova in figura 2.10.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;
use work.bit_evaluation_pkg.all;
use work.generic_array_pkg.all;
use work.layer_record_pkg.all;
use work.nn_utils_pkg.all;

-- ENTITY ======================================================================
entity datapath is
    generic
    (
        d_n_tot         : integer := 9; --data total number of bits
        d_n_frac        : integer := 5; --data number of fractionary bits
        w_n_tot         : integer := 9; --weight total number of bits
        w_n_frac        : integer := 5; --weight number of fractionary bits
        Nj_mat          : int_array; --input matrix dimensions
        L_info          : layer_record; --description of the layer
        Oj_mat          : int_array; --output matrix dimensions
        index           : integer := 0 --index of the layer
    );
    port
    (
        clk                 : in std_logic;
        rstn                : in std_logic;
        --I/O
        input               : in signed(d_n_tot-1 downto 0);
        output              : out signed(d_n_tot-1 downto 0);
        weight_in           : in signed(w_n_tot-1 downto 0);
        --Addresses
```

```
32          read_addr_prev_layer: out unsigned(bitEvalSafe_sum(Nj_mat)-1 downto 0);
33          read_addr_next_layer: in unsigned(bitEvalSafe_sum(Oj_mat)-1 downto 0);
34          --Control signals
35          cntrIN_enbl          : in std_logic;
36          cntrIN_rstn          : in std_logic;
37          cntrW_enbl           : in std_logic;
38          cntrW_rstn           : in std_logic;
39          cntrSM_enbl          : in std_logic;
40          cntrSM_rstn          : in std_logic;
41          cntrF_enbl           : in std_logic;
42          cntrF_rstn           : in std_logic;
43          cntrM_enbl           : in std_logic;
44          cntrM_rstn           : in std_logic;
45          cntrS_enbl           : in std_logic;
46          cntrS_rstn           : in std_logic;
47          MAC_enbl             : in std_logic;
48          MAC_bypass           : in std_logic;
49          MAC_write_output     : in std_logic;
50          write_memA           : in std_logic;
51          write_memB           : in std_logic;
52          write_weight         : in std_logic;
53          weight1_bias0        : in std_logic;
54          --Status signals
55          tcIN                 : out std_logic;
56          tcR                  : out std_logic;
57          tcW                  : out std_logic;
58          tcSM                 : out std_logic;
59          tcF                  : out std_logic;
60          tcM                  : out std_logic;
61          tcS                  : out std_logic;
62          tcOUT                : out std_logic
63      );
64  end entity datapath;
65
66  -- ARCHITECTURE ==============================================================
67  architecture behavior of datapath is
68
69      -- Signals ------------------------------------------------------------
70      -- Note1: signals followed by dX (Where X is a number) are delayed signals
71      -- Note2: many signal could have null length depending on the parameters.
72      --        When they are null length they are not employed.
73
74      --dimensions of the matrix connected to the neurons
75      constant c_mat       : int_array := connect_matrix_dim(Nj_mat, L_info);
76      --bits to represent the connectivity matrix
77      constant c_mat_bits  : int_array := bitEvalSafe(c_mat);
78      --bits to represent the input matrix
79      constant Nj_bits     : int_array := bitEvalSafe(Nj_mat);
80      --dimensions of the receptive field
81      constant Fj_mat      : int_array := L_info.r_field;
82      --padding array
83      constant Pj_mat      : int_array := L_info.padding;
84      --bits for the F counter
85      constant F_bits      : int_array := (bitEvalSafe(Nj_mat(0)+Pj_mat(0)),
86                                           bitEvalSafe(Nj_mat(1)+Pj_mat(1)));
87      --stride array
```

```
88      constant STj_mat      : int_array := L_info.stride;
89      --number of weights (without bias) for each neuron
90      constant Wj           : integer := eval_weights(Nj_mat, L_info);
91      --number of neurons
92      constant Sj           : integer := L_info.neurons;
93      --number of MACs
94      constant Mj           : integer := L_info.macs;
95      --memories address bits
96      constant addrAbits    : integer := bitEvalSafe_sum(Nj_mat);
97      constant addrBbits    : integer := bitEvalSafe_sum(Oj_mat);
98      --number of neurons associated to each MAC
99      constant SMrat        : integer := ceil_ratio(Sj, Mj);
100     --total number of weights for MAC
101     constant Wtot         : integer := (Wj+1)*SMrat;
102     --minimum number of inputs loaded sufficient to start the elaboration (conv)
103     constant Nj_min       : integer := Nj_mat(0)*(Nj_mat(1)*(Nj_mat(2)-1)+
104                                        Fj_mat(1)-1)+Fj_mat(0)-(Wj+1);
105
106     signal dataToMac      : signed(d_n_tot-1 downto 0);
107     signal dataFromMac    : signed(d_n_tot-1 downto 0);
108     signal cntrIN_val     : unsigned(addrAbits-1 downto 0);
109     signal cntrIN_val_d0  : unsigned(addrAbits-1 downto 0);
110     signal cntrDATA_val   : unsigned(bitEvalSafe_sum(c_mat)-1 downto 0);
111     signal cntrDATA_row   : unsigned(c_mat_bits(0)-1 downto 0);
112     signal cntrDATA_col   : unsigned(c_mat_bits(1)-1 downto 0);
113     signal cntrDATA_dep   : unsigned(c_mat_bits(2)-1 downto 0);
114     signal cntrF_val      : signed(F_bits(0)+F_bits(1)+1 downto 0);
115     signal cntrF_row      : signed(F_bits(0) downto 0);
116     signal cntrF_col      : signed(F_bits(1) downto 0);
117     signal cntrW_val      : unsigned(bitEvalSafe(Wj)-1 downto 0);
118     signal cntrSM_val     : unsigned(bitEvalSafe(SMrat)-1 downto 0);
119     signal cntrM_val      : unsigned(bitEvalSafe(Mj)-1 downto 0);
120     signal cntrS_val      : unsigned(bitEvalSafe_sum(Oj_mat)-1 downto 0);
121     signal cntrS_val_d0   : unsigned(bitEvalSafe_sum(Oj_mat)-1 downto 0);
122     signal cntrSM_tc      : std_logic;
123     signal cntrS_tc       : std_logic;
124     signal cntrS_tcs      : std_logic_vector(2 downto 0);
125     signal MAC_enbl_d0    : std_logic;
126     signal MAC_enbl_d0_bounded : std_logic;
127     signal MAC_bypass_d0 : std_logic;
128     signal MAC_write_output_d0  : std_logic;
129     signal set_pos_d0     : std_logic;
130     signal outofbounds    : std_logic;
131     signal write_memA_d0 : std_logic;
132     signal write_memB_d0 : std_logic;
133     signal write_memB_d1 : std_logic;
134     signal addrArow       : signed(F_bits(0) downto 0);
135     signal addrAcol       : signed(F_bits(1) downto 0);
136     signal addrArow_d0    : signed(F_bits(0) downto 0);
137     signal addrAcol_d0    : signed(F_bits(1) downto 0);
138     signal addrAread      : unsigned(addrAbits-1 downto 0);
139     signal addrBread      : unsigned(addrBbits-1 downto 0);
140
141 begin
142     -- Counters ==============================================================
143     -- Load inputs counters ------------------------------------------------
```

```
144     -- IN counts the input to be written in memory A with a 3D disposition
145     icounterIN:entity work.counter_multid generic map(Nj_mat)
146     port map(clk, cntrIN_rstn, cntrIN_enbl, open, tcIN, open, cntrIN_val);
147     read_addr_prev_layer <= cntrIN_val;
148
149     gcounterR: if L_info.layer /= "fulc" generate --if this is not fully-conn.
150         icounterR:entity work.sat_counter generic map(Nj_min)
151         port map(clk, cntrIN_rstn, cntrIN_enbl, tcR, open);
152     end generate;
153     gnot_counterR: if L_info.layer = "fulc" generate --if this is fully-conn.
154         tcR <= '1';
155     end generate;
156
157     -- Elaborate counters-------------------------------------------------------
158     -- DATA counts the inputs to be read from memory A with a 3D disposition
159     icounterDATA:entity work.counter_multid generic map(c_mat)
160     port map(clk, cntrW_rstn, cntrW_enbl, open, open, open, cntrDATA_val);
161
162     gsplit_cntrDATA: if L_info.layer = "conv" or L_info.layer = "maxp" generate
163         cntrDATA_row <= cntrDATA_val(bitEvalSafe_sum(c_mat, 1)-1
164                                      downto 0);
165         cntrDATA_col <= cntrDATA_val(bitEvalSafe_sum(c_mat, 2)-1
166                                      downto bitEvalSafe_sum(c_mat, 1));
167         gcntrDATA_dep: if c_mat(2) > 1 generate
168             cntrDATA_dep <= cntrDATA_val(bitEvalSafe_sum(c_mat)-1
169                                      downto bitEvalSafe_sum(c_mat, 2));
170         end generate;
171     end generate;
172
173     -- W counts the weights to be used in the weights memory
174     gWnot_maxp: if L_info.layer /= "maxp" generate -- for not-maxpol
175         icounterW:entity work.counter generic map(Wj)
176         port map(clk, cntrW_rstn, cntrW_enbl, tcW, cntrW_val);
177     end generate;
178     gWmaxp: if L_info.layer = "maxp" generate -- for maxpol
179         icounterW:entity work.counter generic map(Wj)
180         port map(clk, cntrW_rstn, cntrW_enbl, tcW, open);
181     end generate;
182
183     -- SM counts the iterations done with the same inputs with different weights
184     -- aka the number of neurons already evaluated associated with each MAC
185     gcounterSM_fulc_conv: if L_info.layer /= "maxp" generate -- if not maxp
186         gcounterSM: if SMrat > 1 generate
187             icounterSM:entity work.counter generic map(SMrat)
188             port map(clk, cntrSM_rstn, cntrSM_enbl, cntrSM_tc, cntrSM_val);
189         end generate;
190         gnot_counterSM: if SMrat <= 1 generate
191             cntrSM_tc <= '1'; cntrSM_val <= (others => '0');
192         end generate;
193         tcSM <= cntrSM_tc;
194     end generate;
195     -- For maxpooling layers, SM counts the number of outputs for every
196     -- position of the receptive field.
197     gcounterSM_maxp: if L_info.layer = "maxp" generate -- if maxp
198         gcounterSM: if Oj_mat(2) > 1 generate
199             icounterSM:entity work.counter generic map(Oj_mat(2))
```

```
200             port map(clk, cntrSM_rstn, cntrSM_enbl, cntrSM_tc, open);
201         end generate;
202         gnot_counterSM: if Oj_mat(2) <= 1 generate
203             cntrSM_tc <= '1';
204         end generate;
205         tcSM <= cntrSM_tc;
206     end generate;
207 
208     gcntrF: if L_info.layer = "conv" or L_info.layer = "maxp" generate
209         -- F counts the position of the receptive field; padding is used to
210         -- modify the dimensions of the input field
211         iscounterF:entity work.scounter_multid
212         generic map((Nj_mat(0)+Pj_mat(1),Nj_mat(1)+Pj_mat(3)), STj_mat,
213         (-Pj_mat(0), -Pj_mat(2)), Fj_mat)
214         port map(clk, cntrF_rstn, cntrF_enbl, open, tcF, open, cntrF_val);
215         -- Note: the sign bit is to be taken into account!
216         cntrF_row <= cntrF_val(F_bits(0) downto 0);
217         cntrF_col <= cntrF_val(F_bits(1)+F_bits(0)+1 downto F_bits(0)+1);
218     end generate;
219 
220     gnot_ctnrF: if(L_info.layer = "fulc")generate
221         tcF <= '1';
222     end generate;
223 
224     -- Save outputs counters ---------------------------------------------------
225     -- M counts the MACs
226     gcounterM: if L_info.layer /= "maxp" generate -- useful only for not-maxpool
227         gcounterM: if Mj > 1 generate -- necessary only if M > 1
228             icounterM:entity work.counter generic map(Mj)
229             port map(clk, cntrM_rstn, cntrM_enbl, tcM, cntrM_val);
230         end generate;
231         gnot_counterM: if Mj <= 1 generate
232             tcM <= '1'; cntrM_val <= (others => '0');
233         end generate;
234     end generate;
235     gnot_counterM: if L_info.layer = "maxp" generate --if max-pool
236         tcM <= '1';
237     end generate;
238 
239     -- S counts the neurons and the outputs
240     -- neurons are counted first; when all the neurons have been counted the
241     -- output value increase. This needs a reordering of the outputs.
242     -- For maxpooling layer "neurons" means the depth of output volume.
243     icounterS:entity work.counter_multid
244     generic map((Oj_mat(2),Oj_mat(0),Oj_mat(1)), (1,1,1), (1,2,0))
245     port map(clk, cntrS_rstn, cntrS_enbl, open, cntrS_tc, cntrS_tcs, cntrS_val);
246 
247     gtcS: if L_info.layer /= "maxp" generate --if not maxp
248         tcS <= cntrS_tcs(0); --when all the neurons are counted
249     end generate;
250     gnot_tcS: if L_info.layer = "maxp" generate --if maxp
251         -- There is only 1 elaboration unit (max block) instead than many MACS
252         -- (with many neurons).
253         tcS <= '1';
254     end generate;
255     tcOUT <= cntrS_tc; --when all the values from the neurons are counted
```

```
256
257     -- Out of boundaries check and read address A evaluation ===================
258     -- evaluate the read address of memA
259     gaddrAread_conv: if L_info.layer = "conv" or L_info.layer = "maxp" generate
260         addrArow <= to_signed(to_integer(cntrF_row)
261                    +to_integer(cntrDATA_row), F_bits(0)+1);
262         addrAcol <= to_signed(to_integer(cntrF_col)
263                    +to_integer(cntrDATA_col), F_bits(1)+1);
264         gcntrDATA_dep: if c_mat(2) > 1 generate
265             addrAread <= cntrDATA_dep
266                         & unsigned(addrAcol(Nj_bits(1)-1 downto 0))
267                         & unsigned(addrArow(Nj_bits(0)-1 downto 0));
268         end generate;
269         gnot_cntrDATA_dep: if c_mat(2) <= 1 generate
270             addrAread <= unsigned(addrAcol(Nj_bits(1)-1 downto 0))
271                         & unsigned(addrArow(Nj_bits(0)-1 downto 0));
272         end generate;
273
274         --check if out of boundaries (padding regions)
275         outofbounds <= '1' when to_integer(addrArow_d0) < 0
276                               or to_integer(addrArow_d0) >= Nj_mat(0)
277                               or to_integer(addrAcol_d0) < 0
278                               or to_integer(addrAcol_d0) >= Nj_mat(1) else '0';
279     end generate;
280
281     gaddrAread_fulc: if L_info.layer = "fulc" generate
282         outofbounds <= '0';
283         addrAread <= cntrDATA_val;
284     end generate;
285
286     -- Memories ==============================================================
287     -- input memory
288     imemA:entity work.mem generic map(addrAbits, d_n_tot)
289     port map(clk, input, write_memA_d0, cntrIN_val_d0, addrAread, dataToMac);
290     -- output memory
291     imemB:entity work.mem generic map(addrBbits, d_n_tot)
292     port map(clk, dataFromMac, write_memB_d0, cntrS_val_d0, addrBread, output);
293     addrBread <= read_addr_next_layer;
294
295     -- Neurons/Max ===========================================================
296     -- generate neurons block
297     MAC_enbl_d0_bounded <= MAC_enbl_d0 and not outofbounds;
298
299     gneurons:
300     if L_info.layer = "fulc" or L_info.layer = "conv" generate
301         ineurons:entity work.neurons generic map(d_n_tot, d_n_frac, w_n_tot,
302             w_n_frac, Wj, Sj, Mj, L_info.a_func)
303         port map(
304             clk => clk,
305             rstn => rstn,
306             input => dataToMac,
307             output => dataFromMac,
308             weight_in => weight_in,
309             weight_addr => cntrW_val,
310             iterat_addr => cntrSM_val,
311             MAC_addr => cntrM_val,
```

```
312             MAC_enbl => MAC_enbl_d0_bounded,
313             MAC_bypass => MAC_bypass_d0,
314             write_output => MAC_write_output_d0,
315             write_weight => write_weight,
316             weight1_bias0 => weight1_bias0,
317             last_row => cntrSM_tc
318         );
319     end generate;
320     --generate max block
321     gmax:
322     if L_info.layer = "maxp" generate
323         imax:entity work.max generic map(d_n_tot)
324         port map(
325             clk => clk,
326             rstn => rstn,
327             input => dataToMac,
328             init => MAC_bypass_d0,
329             enbl => MAC_enbl_d0_bounded,
330             write_out => MAC_write_output_d0,
331             set_pos => outofbounds,
332             q => dataFromMac
333         );
334     end generate;
335
336     -- Delay registers ==========================================================
337     icntrS_d0:entity work.ureg generic map(bitEvalSafe_sum(Oj_mat))
338     port map(cntrS_val, clk, '1', rstn, cntrS_val_d0);
339
340     gaddrAread_d0: if L_info.layer /= "fulc" generate
341         iaddrArow_d0:entity work.sreg generic map(Nj_bits(0)+1)
342         port map(addrArow, clk, '1', rstn, addrArow_d0);
343         iaddrAcol_d0:entity work.sreg generic map(Nj_bits(1)+1)
344         port map(addrAcol, clk, '1', rstn, addrAcol_d0);
345     end generate;
346
347     iMAC_enbl_d0:entity work.ff
348     port map(MAC_enbl, clk, rstn, MAC_enbl_d0);
349
350     iMAC_bypass_d0:entity work.ff
351     port map(MAC_bypass, clk, rstn, MAC_bypass_d0);
352
353     iwrite_memB_d0:entity work.ff
354     port map(write_memB, clk, rstn, write_memB_d0);
355
356     iMAC_write_output_d0:entity work.ff
357     port map(MAC_write_output, clk, rstn, MAC_write_output_d0);
358
359     ginput_pipe: if index > 0 generate --if it's not the first layer
360         icntrIN_d0:entity work.ureg generic map(n_tot => addrAbits)
361         port map(cntrIN_val, clk, '1', rstn, cntrIN_val_d0);
362         iwrite_memA_d0:entity work.ff
363         port map(write_memA, clk, rstn, write_memA_d0);
364     end generate;
365
366     gnot_input_pipe: if index = 0 generate --if it's the first layer
367         cntrIN_val_d0 <= cntrIN_val;
```

```
368         write_memA_d0 <= write_memA;
369     end generate;
370 end architecture behavior;
```

Listing A.12: File di descrizione del datapath del layer (`datapath.vhd`).

A.2.9 Layer della rete neurale (`nn_layer.vhd`)

Questo blocco contiene il datapath del layer e le macchine a stati che gestiscono i vari stadi dell'esecuzione (*load inputs*, *elaborate*, *save outputs*). Il codice delle macchine a stati viene omesso; la loro struttura può essere agevolmente osservata nelle figure 2.13, 2.14, 2.15, 2.18, 2.19, 2.20. Le macchine a stati realmente implementate sono quelle riferite alla rete convolutiva ma, come viene spiegato nel paragrafo 2.2.6, queste vengono facilmente ridotte a quelle fully connected rendendo costanti determinati segnali.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use ieee.math_real.all;
5  use work.bit_evaluation_pkg.all;
6  use work.generic_array_pkg.all;
7  use work.layer_record_pkg.all;
8
9  -- ENTITY ==================================================================
10 entity nn_layer is
11     generic
12     (
13         d_n_tot         : integer := 9; --data total number of bits
14         d_n_frac        : integer := 5; --data number of fractionary bits
15         w_n_tot         : integer := 9; --weight total number of bits
16         w_n_frac        : integer := 5; --weight number of fractionary bits
17         Nj_mat          : int_array; --input matrix dimensions
18         L_info          : layer_record; --description of the layer
19         Oj_mat          : int_array; --output matrix dimensions
20         index           : integer := 0 --index of the layer
21     );
22     port(
23         clk                  : in std_logic;
24         rstn                 : in std_logic;
25         -- I/O
26         input                : in signed(d_n_tot-1 downto 0);
27         output               : out signed(d_n_tot-1 downto 0);
28         weight_in            : in signed(w_n_tot-1 downto 0);
29         -- Addresses
30         read_addr_prev_layer : out unsigned(bitEvalSafe_sum(Nj_mat)-1 downto 0);
31         read_addr_next_layer : in unsigned(bitEvalSafe_sum(Oj_mat)-1 downto 0);
32         -- Control signals
33         output_ready_prev_layer : in std_logic;
34         resetn_output_ready_next_layer  : in std_logic;
35         weight1_bias0_lw        : in std_logic;
```

```
36          write_weight            : in std_logic;
37          cntrW_enbl_lw           : in std_logic;
38          cntrW_rstn_lw           : in std_logic;
39          cntrSM_enbl_lw          : in std_logic;
40          cntrSM_rstn_lw          : in std_logic;
41          cntrM_enbl_lw           : in std_logic;
42          cntrM_rstn_lw           : in std_logic;
43          cntrS_enbl_lw           : in std_logic;
44          cntrS_rstn_lw           : in std_logic;
45          -- Status signals
46          output_ready_next_layer : out std_logic;
47          resetn_output_ready_prev_layer  : out std_logic;
48          tcW_lw                  : out std_logic;
49          tcM_lw                  : out std_logic;
50          tcS_lw                  : out std_logic
51      );
52  end entity nn_layer;
53
54  ------------------------------------------------------------------------------
55  -- ARCHITECTURE
56  ------------------------------------------------------------------------------
57
58  architecture behavior of nn_layer is
59      -- Note: signals followed by dX (Where X is a number) are delayed signals.
60      -- Counter resets, enables, terminal counts
61      signal cntrIN_enbl              : std_logic;
62      signal cntrIN_rstn              : std_logic;
63      signal cntrW_enbl               : std_logic;
64      signal cntrW_enbl_to_dp         : std_logic;
65      signal cntrW_rstn               : std_logic;
66      signal cntrW_rstn_to_dp         : std_logic;
67      signal cntrSM_enbl              : std_logic;
68      signal cntrSM_enbl_to_dp        : std_logic;
69      signal cntrSM_rstn              : std_logic;
70      signal cntrSM_rstn_to_dp        : std_logic;
71      signal cntrF_enbl               : std_logic;
72      signal cntrF_rstn               : std_logic;
73      signal cntrM_enbl               : std_logic;
74      signal cntrM_enbl_to_dp         : std_logic;
75      signal cntrM_rstn               : std_logic;
76      signal cntrM_rstn_to_dp         : std_logic;
77      signal cntrS_enbl               : std_logic;
78      signal cntrS_enbl_to_dp         : std_logic;
79      signal cntrS_rstn               : std_logic;
80      signal cntrS_rstn_to_dp         : std_logic;
81      signal tcIN                     : std_logic;
82      signal tcR                      : std_logic;
83      signal tcW                      : std_logic;
84      signal tcSM                     : std_logic;
85      signal tcF                      : std_logic;
86      signal tcM                      : std_logic;
87      signal tcS                      : std_logic;
88      signal tcOUT                    : std_logic;
89      -- Memory enables, addresses, statuses and controls
90      signal write_memA               : std_logic;
91      signal write_memB               : std_logic;
```

```
92      signal weight1_bias0            : std_logic;
93      signal weight1_bias0_to_dp      : std_logic;
94      signal enough_inputs            : std_logic;
95      signal MAC_enbl                 : std_logic;
96      signal MAC_bypass               : std_logic;
97      signal MAC_write_output         : std_logic;
98      -- Control unit handshake signals
99      signal input_ready              : std_logic;
100     signal set_input_ready          : std_logic;
101     signal resetn_input_ready       : std_logic;
102     signal resetn_input_ready_to_sr : std_logic;
103     signal data_ready               : std_logic;
104     signal set_data_ready           : std_logic;
105     signal set_data_ready_d0        : std_logic;
106     signal set_data_ready_d1        : std_logic;
107     signal set_data_ready_d2        : std_logic;
108     signal resetn_data_ready        : std_logic;
109     signal resetn_data_ready_to_sr  : std_logic;
110     signal elab_done                : std_logic;
111     signal set_elab_done            : std_logic;
112     signal set_elab_done_d0         : std_logic;
113     signal set_elab_done_d1         : std_logic;
114     signal set_elab_done_d2         : std_logic;
115     signal resetn_elab_done         : std_logic;
116     signal resetn_elab_done_to_sr   : std_logic;
117     signal output_ready             : std_logic;
118     signal set_output_ready         : std_logic;
119     signal resetn_output_ready_to_sr: std_logic;
120     signal cu_rstn                  : std_logic;
121 begin
122     -- Datapath ==============================================================
123     idatapath:entity work.datapath
124     generic map(
125         d_n_tot => d_n_tot,
126         d_n_frac => d_n_frac,
127         w_n_tot => w_n_tot,
128         w_n_frac => w_n_frac,
129         Nj_mat => Nj_mat,
130         L_info => L_info,
131         Oj_mat => Oj_mat,
132         index => index
133     )
134     port map(
135         clk => clk,
136         rstn => rstn,
137         input => input,
138         output => output,
139         weight_in => weight_in,
140         read_addr_prev_layer => read_addr_prev_layer,
141         read_addr_next_layer => read_addr_next_layer,
142         cntrIN_enbl => cntrIN_enbl,
143         cntrIN_rstn => cntrIN_rstn,
144         cntrW_enbl => cntrW_enbl_to_dp,
145         cntrW_rstn => cntrW_rstn_to_dp,
146         cntrSM_enbl => cntrSM_enbl_to_dp,
147         cntrSM_rstn => cntrSM_rstn_to_dp,
```

```
148         cntrF_enbl => cntrF_enbl,
149         cntrF_rstn => cntrF_rstn,
150         cntrM_enbl => cntrM_enbl_to_dp,
151         cntrM_rstn => cntrM_rstn_to_dp,
152         cntrS_enbl => cntrS_enbl_to_dp,
153         cntrS_rstn => cntrS_rstn_to_dp,
154         MAC_enbl => MAC_enbl,
155         MAC_bypass => MAC_bypass,
156         MAC_write_output => MAC_write_output,
157         write_memA => write_memA,
158         write_memB => write_memB,
159         write_weight => write_weight,
160         weight1_bias0 => weight1_bias0_to_dp,
161         tcIN => tcIN,
162         tcR => tcR,
163         tcW => tcW,
164         tcSM => tcSM,
165         tcF => tcF,
166         tcM => tcM,
167         tcS => tcS,
168         tcOUT => tcOUT
169     );
170
171     -- Connect terminal count in output.
172     tcW_lw <= tcW; tcM_lw <= tcM; tcS_lw <= tcS;
173
174     -- Select inputs for counters W, SM, S, M (if they come from layer cu or
175     -- from weight cu).
176     with write_weight select cntrW_enbl_to_dp <=
177         cntrW_enbl_lw when '1',
178         cntrW_enbl when others;
179     with write_weight select cntrW_rstn_to_dp <=
180         cntrW_rstn_lw when '1',
181         cntrW_rstn when others;
182     with write_weight select cntrSM_enbl_to_dp <=
183         cntrSM_enbl_lw when '1',
184         cntrSM_enbl when others;
185     with write_weight select cntrSM_rstn_to_dp <=
186         cntrSM_rstn_lw when '1',
187         cntrSM_rstn when others;
188     with write_weight select cntrM_enbl_to_dp <=
189         cntrM_enbl_lw when '1',
190         cntrM_enbl when others;
191     with write_weight select cntrM_rstn_to_dp <=
192         cntrM_rstn_lw when '1',
193         cntrM_rstn when others;
194     with write_weight select cntrS_enbl_to_dp <=
195         cntrS_enbl_lw when '1',
196         cntrS_enbl when others;
197     with write_weight select cntrS_rstn_to_dp <=
198         cntrS_rstn_lw when '1',
199         cntrS_rstn when others;
200     -- Select if weight1_bias0 is controlled by weight loader or layer CU.
201     with write_weight select weight1_bias0_to_dp <=
202         weight1_bias0_lw when '1',
203         weight1_bias0 when others;
```

```
204
205     -- Control units intercommunications flags ================================
206     -- SR structures instantiation
207     iinput_ready_sr:entity work.sr
208     port map(set_input_ready, clk, resetn_input_ready_to_sr, input_ready);
209     idata_ready_sr:entity work.sr
210     port map(set_data_ready_d2, clk, resetn_data_ready_to_sr, data_ready);
211     ielab_done_sr:entity work.sr
212     port map(set_elab_done_d2, clk, resetn_elab_done_to_sr, elab_done);
213     ioutput_ready_sr:entity work.sr
214     port map(set_output_ready, clk, resetn_output_ready_to_sr, output_ready);
215     -- Reset signals for the SR structures
216     resetn_input_ready_to_sr <= rstn and resetn_input_ready;
217     resetn_data_ready_to_sr <= rstn and resetn_data_ready;
218     resetn_elab_done_to_sr <= rstn and resetn_elab_done;
219     resetn_output_ready_to_sr <= rstn and resetn_output_ready_next_layer;
220
221     -- Layer Control Units ======================================================
222
223     icu_load_inputs:entity work.cu_load_inputs
224     port map(
225         clk => clk,
226         rstn => cu_rstn,
227         input_ready => input_ready,
228         output_ready => output_ready_prev_layer,
229         tcIN => tcIN,
230         tcR => tcR,
231         set_input_ready => set_input_ready,
232         resetn_output_ready => resetn_output_ready_prev_layer,
233         write_memA => write_memA,
234         cntrIN_rstn => cntrIN_rstn,
235         cntrIN_enbl => cntrIN_enbl
236     );
237
238     icu_elaborate:entity work.cu_elaborate
239     port map(
240         clk => clk,
241         rstn => cu_rstn,
242         input_ready => input_ready,
243         data_ready => data_ready,
244         tcW => tcW,
245         tcSM => tcSM,
246         tcF => tcF,
247         set_data_ready => set_data_ready,
248         set_elab_done => set_elab_done,
249         resetn_input_ready => resetn_input_ready,
250         cntrW_rstn => cntrW_rstn,
251         cntrW_enbl => cntrW_enbl,
252         cntrSM_rstn => cntrSM_rstn,
253         cntrSM_enbl => cntrSM_enbl,
254         cntrF_rstn => cntrF_rstn,
255         cntrF_enbl => cntrF_enbl,
256         MAC_enbl => MAC_enbl,
257         MAC_bypass => MAC_bypass,
258         MAC_write_output => MAC_write_output,
259         weight1_bias0 => weight1_bias0
```

```
260     );
261
262     icu_save_outputs:entity work.cu_save_outputs
263     port map(
264         clk => clk,
265         rstn => cu_rstn,
266         output_ready => output_ready,
267         data_ready => data_ready,
268         tcM => tcM,
269         tcS => tcS,
270         tcOUT => tcOUT,
271         elab_done => elab_done,
272         set_output_ready => set_output_ready,
273         resetn_data_ready => resetn_data_ready,
274         resetn_elab_done => resetn_elab_done,
275         write_memB => write_memB,
276         cntrM_rstn => cntrM_rstn,
277         cntrM_enbl => cntrM_enbl,
278         cntrS_rstn => cntrS_rstn,
279         cntrS_enbl => cntrS_enbl
280     );
281
282     output_ready_next_layer <= output_ready;
283     -- Data elaboration must be stopped while loading biases and weights.
284     cu_rstn <= rstn or not write_weight;
285
286     -- Delay registers ==========================================================
287     -- Delay0
288     iset_data_ready_d0:entity work.ff
289     port map(set_data_ready, clk, rstn, set_data_ready_d0);
290     iset_elab_done_d0:entity work.ff
291     port map(set_elab_done, clk, rstn, set_elab_done_d0);
292
293     -- Delay1
294     iset_data_ready_d1:entity work.ff
295     port map(set_data_ready_d0, clk, rstn, set_data_ready_d1);
296     iset_elab_done_d1:entity work.ff
297     port map(set_elab_done_d0, clk, rstn, set_elab_done_d1);
298
299     -- Delay2
300     gmaxp: if L_info.layer = "maxp" generate --if this is maxpooling
301         set_data_ready_d2 <= set_data_ready_d1;
302         set_elab_done_d2 <= set_elab_done_d1;
303     end generate;
304     gnot_maxp: if L_info.layer /= "maxp" generate --if this is not maxpooling
305         iset_data_ready_d2:entity work.ff
306         port map(set_data_ready_d1, clk, rstn, set_data_ready_d2);
307         iset_elab_done_d2:entity work.ff
308         port map(set_elab_done_d1, clk, rstn, set_elab_done_d2);
309     end generate;
310 end architecture behavior;
```

Listing A.13: File di descrizione del layer della rete neurale (`nn_layer.vhd`).

A.2.10 Rete neurale completa (`neural_network.vhd`)

Questo blocco mette insieme i layer della rete neurale e gestisce il caricamento dei pesi tramite la macchina a stati che si può visionare in figura 2.27.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;
use work.bit_evaluation_pkg.all;
use work.generic_array_pkg.all;
use work.layer_record_pkg.all;
use work.nn_utils_pkg.all;

-- ENTITY ==================================================================
entity neural_network is
    generic
    (
        d_n_tot         : integer := 16; --data total number of bits
        d_n_frac        : integer := 10; --data number of fractionary bits
        w_n_tot         : integer := 16; --weight total number of bits
        w_n_frac        : integer := 10; --weight number of fractionary bits
        IN_size         : int_array := (1, 1, 16); --number of inputs in the nn
        L_list          : layer_array; --description of the layers
        OUT_size        : int_array := (1, 1, 3) --number of outputs from the nn
    );
    port
    (
        clk             : in std_logic;
        rstn            : in std_logic;
        --input
        input           : in signed(d_n_tot-1 downto 0);
        start           : in std_logic;
        ready           : out std_logic;
        --output
        output          : out signed(d_n_tot-1 downto 0);
        output_address  : in unsigned(bitEvalSafe_sum(OUT_size)-1 downto 0);
        done            : out std_logic;
        continue        : in std_logic;
        --weights loading
        weight          : in signed(w_n_tot-1 downto 0);
        w_start         : in std_logic;
        w_done          : out std_logic
    );
end entity neural_network;

-- ARCHITECTURE ============================================================
architecture behavior of neural_network is

    -- Signals & constants ------------------------------------------------
    -- number of layers
    constant L_num                : integer := L_list'length;
    -- number of layers with weights
    constant L_num_ww             : integer := getLayersWithWeights(L_list);

    type layers_outputs_type is array(0 to L_num) of signed(d_n_tot-1 downto 0);
```

```
52      type layers_output_addresses_type is array (0 to L_num)
53          of unsigned(max_output(IN_size, L_list)-1 downto 0);
54
55      signal layers_inputs          : layers_outputs_type;
56      signal layers_output_addresses: layers_output_addresses_type;
57      signal layers_output_ready    : std_logic_vector(L_num downto 0);
58      signal layers_resetn_output_ready : std_logic_vector(L_num downto 0);
59      signal layers_enblW           : std_logic_vector(L_num-1 downto 0);
60      signal layers_enblSM          : std_logic_vector(L_num-1 downto 0);
61      signal layers_enblM           : std_logic_vector(L_num-1 downto 0);
62      signal layers_enblS           : std_logic_vector(L_num-1 downto 0);
63      signal layers_tcW             : std_logic_vector(L_num-1 downto 0);
64      signal layers_tcM             : std_logic_vector(L_num-1 downto 0);
65      signal layers_tcS             : std_logic_vector(L_num-1 downto 0);
66      signal layers_weight1_bias0   : std_logic_vector(L_num-1 downto 0);
67      signal layers_write_weight    : std_logic_vector(L_num-1 downto 0);
68      signal cntrW_enbl             : std_logic;
69      signal cntrW_rstn             : std_logic;
70      signal cntrSM_enbl            : std_logic;
71      signal cntrSM_rstn            : std_logic;
72      signal cntrM_enbl             : std_logic;
73      signal cntrM_rstn             : std_logic;
74      signal cntrS_enbl             : std_logic;
75      signal cntrS_rstn             : std_logic;
76      signal cntrL_enbl             : std_logic;
77      signal cntrL_rstn             : std_logic;
78      signal tcW_lw                 : std_logic;
79      signal tcM_lw                 : std_logic;
80      signal tcS_lw                 : std_logic;
81      signal tcL_lw                 : std_logic;
82      signal cntrL_val              : unsigned(bitEvalSafe(L_num_ww)-1 downto 0);
83      signal write_weight           : std_logic;
84      signal weight1_bias0          : std_logic;
85      signal layer_select           : std_logic_vector(L_num_ww-1 downto 0);
86      signal tc_select              : std_logic_vector(L_num-1 downto 0);
87
88  begin
89      -- Layers instantiation ==================================================
90      gnn_layers:
91      for i in 0 to L_num-1 generate
92          inn_layer:entity work.nn_layer
93          generic map(
94              d_n_tot => d_n_tot,
95              d_n_frac => d_n_frac,
96              w_n_tot => w_n_tot,
97              w_n_frac => w_n_frac,
98              Nj_mat => eval_inputs(IN_size, L_list, i),
99              L_info => L_list(i),
100             Oj_mat => eval_inputs(IN_size, L_list, i+1),
101             index => i
102         )
103         port map(
104             clk => clk,
105             rstn => rstn,
106             input => layers_inputs(i),
107             output => layers_inputs(i+1),
```

```
108             weight_in => weight,
109             read_addr_prev_layer => layers_output_addresses(i)
110                 (bitEvalSafe_sum(eval_inputs(IN_size, L_list, i))-1 downto 0),
111             read_addr_next_layer => layers_output_addresses(i+1)
112                 (bitEvalSafe_sum(eval_inputs(IN_size, L_list, i+1))-1 downto 0),
113             output_ready_prev_layer => layers_output_ready(i),
114             output_ready_next_layer => layers_output_ready(i+1),
115             resetn_output_ready_prev_layer => layers_resetn_output_ready(i),
116             resetn_output_ready_next_layer => layers_resetn_output_ready(i+1),
117             weight1_bias0_lw => layers_weight1_bias0(i),
118             write_weight => layers_write_weight(i),
119             cntrW_enbl_lw => layers_enblW(i),
120             cntrW_rstn_lw => cntrW_rstn,
121             cntrSM_enbl_lw => layers_enblSM(i),
122             cntrSM_rstn_lw => cntrSM_rstn,
123             cntrM_enbl_lw => layers_enblM(i),
124             cntrM_rstn_lw => cntrM_rstn,
125             cntrS_enbl_lw => layers_enblS(i),
126             cntrS_rstn_lw => cntrS_rstn,
127             tcW_lw => layers_tcW(i),
128             tcM_lw => layers_tcM(i),
129             tcS_lw => layers_tcS(i)
130         );
131     end generate;
132     -- inputs management
133     layers_inputs(0) <= input;
134     layers_output_ready(0) <= start;
135     ready <= not layers_resetn_output_ready(0);
136     -- outputs management
137     output <= layers_inputs(L_num);
138     layers_output_addresses(L_num)
139         (bitEvalSafe_sum(eval_inputs(IN_size, L_list, L_num))-1 downto 0) <=
140         output_address;
141     layers_resetn_output_ready(L_num) <= not continue;
142     done <= layers_output_ready(L_num);
143
144     -- Control Unit for weight and bias load ==================================
145     icu_load_weights:entity work.cu_load_weights
146     port map(
147         clk => clk,
148         rstn => rstn,
149         w_start => w_start,
150         tcW => tcW_lw,
151         tcM => tcM_lw,
152         tcS => tcS_lw,
153         tcL => tcL_lw,
154         cntrW_rstn => cntrW_rstn,
155         cntrW_enbl => cntrW_enbl,
156         cntrSM_rstn => cntrSM_rstn,
157         cntrSM_enbl => cntrSM_enbl,
158         cntrM_rstn => cntrM_rstn,
159         cntrM_enbl => cntrM_enbl,
160         cntrS_rstn => cntrS_rstn,
161         cntrS_enbl => cntrS_enbl,
162         cntrL_rstn => cntrL_rstn,
163         cntrL_enbl => cntrL_enbl,
```

```
164         write_weight => write_weight,
165         weight1_bias0 => weight1_bias0,
166         w_done => w_done
167     );
168 
169     -- Counter for weight and bias load ========================================
170     gcounterL: if L_num_ww > 1 generate
171         icounterL:entity work.counter generic map(L_num_ww)
172         port map(clk, cntrL_rstn, cntrL_enbl, tcL_lw, cntrL_val);
173     end generate;
174     gnot_counterL: if L_num_ww <= 1 generate
175         tcL_lw <= '1'; cntrL_val <= (others => '0');
176     end generate;
177 
178     -- Weight cu anc layers interface ==========================================
179     -- tc selection to cu
180     ptc_selection: process(layers_tcW, layers_tcM, layers_tcS, tc_select)
181         variable tcW_lw_temp    : std_logic_vector(L_num-1 downto 0);
182         variable tcM_lw_temp    : std_logic_vector(L_num-1 downto 0);
183         variable tcS_lw_temp    : std_logic_vector(L_num-1 downto 0);
184     begin
185         if L_num_ww > 0 then
186             tcW_lw_temp(0) := (layers_tcW(0) and tc_select(0));
187             tcM_lw_temp(0) := (layers_tcM(0) and tc_select(0));
188             tcS_lw_temp(0) := (layers_tcS(0) and tc_select(0));
189             for i in 1 to L_num-1 loop
190                 tcW_lw_temp(i) := tcW_lw_temp(i-1)
191                                or (layers_tcW(i) and tc_select(i));
192                 tcM_lw_temp(i) := tcM_lw_temp(i-1)
193                                or (layers_tcM(i) and tc_select(i));
194                 tcS_lw_temp(i) := tcS_lw_temp(i-1)
195                                or (layers_tcS(i) and tc_select(i));
196             end loop;
197             tcW_lw <= tcW_lw_temp(L_num-1);
198             tcM_lw <= tcM_lw_temp(L_num-1);
199             tcS_lw <= tcS_lw_temp(L_num-1);
200         else
201             tcW_lw <= '1';
202             tcM_lw <= '1';
203             tcS_lw <= '1';
204         end if;
205     end process;
206 
207     -- layer selector decoder
208     player_select_decoder:
209     process(cntrL_val)
210     begin
211         if(L_num_ww > 0)then
212             layer_select <= (others => '0');
213             layer_select(to_integer(cntrL_val)) <= '1';
214         end if;
215     end process;
216 
217     -- save and enable signals gating
218     psave_enbl_gating: process(write_weight, cntrW_enbl, cntrSM_enbl,
219         cntrM_enbl, cntrS_enbl, weight1_bias0, layer_select)
```

```
        variable nw         : int_array(L_num-1 downto 0);
    begin
        nw(0) := 0; --count the layers without weights
        for i in 0 to L_num-1 loop
            if(L_list(i).layer = "maxp")then
                if(i > 0)then
                    nw(i) := nw(i-1) + 1;
                end if;
                layers_write_weight(i) <= '0';
                layers_enblW(i) <= '0';
                layers_enblSM(i) <= '0';
                layers_enblM(i) <= '0';
                layers_enblS(i) <= '0';
                layers_weight1_bias0(i) <= '0';
                tc_select(i) <= '0';
            else
                if(i > 0)then
                    nw(i) := nw(i-1);
                end if;
                layers_write_weight(i) <= write_weight
                                        and layer_select(i-nw(i));
                layers_enblW(i) <= cntrW_enbl and layer_select(i-nw(i));
                layers_enblSM(i) <= cntrSM_enbl and layer_select(i-nw(i));
                layers_enblM(i) <= cntrM_enbl and layer_select(i-nw(i));
                layers_enblS(i) <= cntrS_enbl and layer_select(i-nw(i));
                layers_weight1_bias0(i) <= weight1_bias0
                                         and layer_select(i-nw(i));
                tc_select(i) <= layer_select(i-nw(i));
            end if;
        end loop;
    end process;

end architecture behavior;
```

Listing A.14: File di descrizione della rete neurale completa (`neural_network.vhd`).

A.2.11 Top level entity (`neural_network_top.vhd`)

Questo blocco "incapsula" il blocco `neural_network`, passando i valori che vengono recuperati dal file `parameters.vhd` e convertendo i segnali da `signed` e da `unsigned` a `std_logic_vector` per garantire la massima compatibilità col software che effettua il mapping della struttura.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;
use work.bit_evaluation_pkg.all;
use work.generic_array_pkg.all;
use work.layer_record_pkg.all;
use work.nn_utils_pkg.all;
use work.parameters.all;
```

```
10
11 -- ENTITY ==================================================================
12 entity neural_network_top is
13     port
14     (
15         clk             : in std_logic;
16         rstn            : in std_logic;
17         --input
18         input           : in std_logic_vector(data_bits-1 downto 0);
19         start           : in std_logic;
20         ready           : out std_logic;
21         --output
22         output          : out std_logic_vector(data_bits-1 downto 0);
23         output_address  : in std_logic_vector
24                                     (bitEvalSafe_sum(o_size)-1 downto 0);
25         done            : out std_logic;
26         continue        : in std_logic;
27         --weights loading
28         weight          : in std_logic_vector(weight_bits-1 downto 0);
29         w_start         : in std_logic;
30         w_done          : out std_logic
31     );
32 end entity neural_network_top;
33
34 -- ARCHITECTURE ==============================================================
35 architecture behaviour of neural_network_top is
36     signal s_input          : signed(data_bits-1 downto 0);
37     signal s_output         : signed(data_bits-1 downto 0);
38     signal s_weight         : signed(weight_bits-1 downto 0);
39     signal u_output_address : unsigned(bitEvalSafe_sum(o_size)-1 downto 0);
40 begin
41     --neural network instantiation
42     ineural_network:entity work.neural_network
43     generic map(
44         d_n_tot => data_bits,
45         d_n_frac => data_fractional_bits,
46         w_n_tot => weight_bits,
47         w_n_frac => weight_fractional_bits,
48         IN_size => inputs_size,
49         L_list => layers,
50         OUT_size => o_size
51     )
52     port map(
53         clk => clk,
54         rstn => rstn,
55         input => s_input,
56         start => start,
57         ready => ready,
58         output => s_output,
59         output_address => u_output_address,
60         done => done,
61         continue => continue,
62         weight => s_weight,
63         w_start => w_start,
64         w_done => w_done
65     );
```

```
66      --signal conversion
67      s_input <= signed(input);
68      output <= std_logic_vector(s_output);
69      s_weight <= signed(weight);
70      u_output_address <= unsigned(output_address);
71  end architecture behaviour;
```

Listing A.15: File di descrizione dell'entity di incapsulamento della rete neurale (`neural_network_top.vhd`).

Appendice B

Codice completo per il microcontrollore

B.1 File di espansione delle funzionalità del preprocessore (pp_helper.h)

Il codice di questo file contiene una serie di macro necessarie per espandere le funzionalità del preprocessore. In questo modo si possono eseguire espansioni condizionali e ricorsive e si possono utilizzare delle strutture simili a degli array [10].

```
//preprocessor basic operators
#define CAT(a, ...) PRIMITIVE_CAT(a, __VA_ARGS__)
#define PRIMITIVE_CAT(a, ...) a ## __VA_ARGS__

#define IIF(c) PRIMITIVE_CAT(IIF_, c)
#define IIF_0(t, ...) __VA_ARGS__
#define IIF_1(t, ...) t

#define COMPL(b) PRIMITIVE_CAT(COMPL_, b)
#define COMPL_0 1
#define COMPL_1 0

#define BITAND(x) PRIMITIVE_CAT(BITAND_, x)
#define BITAND_0(y) 0
#define BITAND_1(y) y

#define INC(x) PRIMITIVE_CAT(INC_, x)
#define INC_0 1
#define INC_1 2
#define INC_2 3
#define INC_3 4
#define INC_4 5
#define INC_5 6
```

```
24 #define INC_6 7
25 #define INC_7 8
26 #define INC_8 9
27 #define INC_9 9
28
29 #define DEC(x) PRIMITIVE_CAT(DEC_, x)
30 #define DEC_0 0
31 #define DEC_1 0
32 #define DEC_2 1
33 #define DEC_3 2
34 #define DEC_4 3
35 #define DEC_5 4
36 #define DEC_6 5
37 #define DEC_7 6
38 #define DEC_8 7
39 #define DEC_9 8
40
41 //preprocessor detection
42 #define CHECK_N(x, n, ...) n
43 #define CHECK(...) CHECK_N(__VA_ARGS__, 0,)
44 #define PROBE(x) x, 1,
45
46 #define IS_PAREN(x) CHECK(IS_PAREN_PROBE x)
47 #define IS_PAREN_PROBE(...) PROBE(~)
48
49 #define NOT(x) CHECK(PRIMITIVE_CAT(NOT_, x))
50 #define NOT_0 PROBE(~)
51
52 #define BOOL(x) COMPL(NOT(x))
53 #define IF(c) IIF(BOOL(c))
54
55 #define EAT(...)
56 #define EXPAND(...) __VA_ARGS__
57 #define WHEN(c) IF(c)(EXPAND, EAT)
58
59 //preprocessor recursion
60 #define EMPTY()
61 #define DEFER(id) id EMPTY()
62 #define OBSTRUCT(...) __VA_ARGS__ DEFER(EMPTY)()
63 #define EXPAND(...) __VA_ARGS__
64
65 #define EVAL(...)  EVAL1(EVAL1(EVAL1(__VA_ARGS__)))
66 #define EVAL1(...) EVAL2(EVAL2(EVAL2(__VA_ARGS__)))
67 #define EVAL2(...) EVAL3(EVAL3(EVAL3(__VA_ARGS__)))
68 #define EVAL3(...) EVAL4(EVAL4(EVAL4(__VA_ARGS__)))
69 #define EVAL4(...) EVAL5(EVAL5(EVAL5(__VA_ARGS__)))
70 #define EVAL5(...) __VA_ARGS__
71
72 #define REPEAT(count, macro, ...) \
73     WHEN(count) \
74     ( \
75         OBSTRUCT(REPEAT_INDIRECT) () \
76         ( \
77             DEC(count), macro, __VA_ARGS__ \
78         ) \
79         OBSTRUCT(macro) \
```

```
        ( \
            DEC(count), __VA_ARGS__ \
        ) \
    )
#define REPEAT_INDIRECT() REPEAT

//preprocessor comparison
#define PRIMITIVE_COMPARE(x, y) IS_PAREN \
( \
    COMPARE_ ## x ( COMPARE_ ## y) (())  \
)

#define IS_COMPARABLE(x) IS_PAREN( CAT(COMPARE_, x) (()) )

#define NOT_EQUAL(x, y) \
IIF(BITAND(IS_COMPARABLE(x))(IS_COMPARABLE(y)) ) \
( \
    PRIMITIVE_COMPARE, \
    1 EAT \
)(x, y)

#define EQUAL(x, y) COMPL(NOT_EQUAL(x, y))

//preprocessor array
#define ARGN(LIST, N) CAT(ARG_,N) LIST
#define ARG_0(A0, ...) A0
#define ARG_1(A0, A1, ...) A1
#define ARG_2(A0, A1, A2, ...) A2
#define ARG_3(A0, A1, A2, A3, ...) A3
#define ARG_4(A0, A1, A2, A3, A4, ...) A4
#define ARG_5(A0, A1, A2, A3, A4, A5, ...) A5
#define ARG_6(A0, A1, A2, A3, A4, A5, A6, ...) A6
#define ARG_7(A0, A1, A2, A3, A4, A5, A6, A7, ...) A7
#define ARG_8(A0, A1, A2, A3, A4, A5, A6, A7, A8, ...) A8
#define ARG_9(A0, A1, A2, A3, A4, A5, A6, A7, A8, A9, ...) A9
```

Listing B.1: File di espansione delle funzionalità del preprocessore (`pp_helper.h`).

B.2 Parametri definiti dall'utente (`user_parameters.h`)

I dettagli relativi a questo file insieme ad un esempio si possono trovare nel paragrafo 3.2.

B.3 File di definizione delle macro per la rete neurale (`nn_macros.h`)

In questo file vengono definite le macro per la generazione del codice necessario alla dichiarazione delle strutture dei dati, al caricamento dei pesi e all'esecuzione delle

operazioni per i vari layer della rete neurale. Il codice come appare una volta espanso si può trovare nei paragrafi 3.3, 3.4 e 3.5. Viene fatto un ampio utilizzo della macro `CAT` che permette di concatenare due identificatori per generarne un terzo: è l'ideale per associare un indice al nome di un array.

```
#define SIZEOUT(W,F,P1,P2,S) (W-F+P1+P2)/S+1

/** Compare macros **/
#define COMPARE_FULLYCONNECTED(x) x
#define COMPARE_MAXPOOLING(x) x
#define COMPARE_CONVOLUTIONAL(x) x
#define COMPARE_POSLIN(x) x
#define COMPARE_SATLIN(x) x

/** Weights and outputs macro **/
#define DECLARE_WEIGHTS(i,_) \
IF(EQUAL(CAT(LAYER, i), FULLYCONNECTED)) \
( \
  const int CAT(OUTPUTSHEIGHT, i) = 1; \
  const int CAT(OUTPUTSWIDTH, i) = 1; \
  const int CAT(OUTPUTSDEPTH, i) = CAT(NEURONS, i); \
  int CAT(weights, i)[CAT(NEURONS, i)] \
                     [CAT(INPUTSDEPTH, i)] \
                     [CAT(INPUTSWIDTH, i)] \
                     [CAT(INPUTSHEIGHT, i)]; \
  int CAT(bias, i)[CAT(NEURONS, i)]; \
  int CAT(inputs, INC(i))[CAT(OUTPUTSDEPTH, i)] \
                         [CAT(OUTPUTSWIDTH, i)] \
                         [CAT(OUTPUTSHEIGHT, i)]; \
  const int CAT(INPUTSHEIGHT, INC(i)) = 1; \
  const int CAT(INPUTSWIDTH, INC(i)) = 1; \
  const int CAT(INPUTSDEPTH, INC(i)) = CAT(OUTPUTSDEPTH, i); \
) \
IF(EQUAL(CAT(LAYER, i), MAXPOOLING))\
( \
  const int CAT(OUTPUTSHEIGHT, i) = \
    SIZEOUT( \
      CAT(INPUTSHEIGHT, i), \
      CAT(POOLHEIGHT, i), \
      ARGN(CAT(PADDING, i), 0), \
      ARGN(CAT(PADDING, i), 1), \
      ARGN(CAT(STRIDE, i), 0) \
    ); \
  const int CAT(OUTPUTSWIDTH, i) = \
    SIZEOUT( \
      CAT(INPUTSWIDTH, i), \
      CAT(POOLWIDTH, i), \
      ARGN(CAT(PADDING, i), 2), \
      ARGN(CAT(PADDING, i), 3), \
      ARGN(CAT(STRIDE, i), 1) \
    ); \
  const int CAT(OUTPUTSDEPTH, i) = CAT(INPUTSDEPTH, i); \
  int CAT(inputs, INC(i))[CAT(OUTPUTSDEPTH, i)] \
                         [CAT(OUTPUTSWIDTH, i)] \
                         [CAT(OUTPUTSHEIGHT, i)]; \
```

```
51    const int CAT(INPUTSHEIGHT, INC(i)) = CAT(OUTPUTSHEIGHT, i); \
52    const int CAT(INPUTSWIDTH, INC(i)) = CAT(OUTPUTSWIDTH, i); \
53    const int CAT(INPUTSDEPTH, INC(i)) = CAT(OUTPUTSDEPTH, i); \
54  ) \
55  IF(EQUAL(CAT(LAYER, i), CONVOLUTIONAL))\
56  ( \
57    const int CAT(OUTPUTSHEIGHT, i) = \
58      SIZEOUT( \
59        CAT(INPUTSHEIGHT, i), \
60        CAT(FILTERHEIGHT, i), \
61        ARGN(CAT(PADDING, i), 0), \
62        ARGN(CAT(PADDING, i), 1), \
63        ARGN(CAT(STRIDE, i), 0) \
64      ); \
65    const int CAT(OUTPUTSWIDTH, i) = \
66      SIZEOUT( \
67        CAT(INPUTSWIDTH, i), \
68        CAT(FILTERWIDTH, i), \
69        ARGN(CAT(PADDING, i), 2), \
70        ARGN(CAT(PADDING, i), 3), \
71        ARGN(CAT(STRIDE, i), 1) \
72      ); \
73    const int CAT(OUTPUTSDEPTH, i) = CAT(NEURONS, i); \
74    int CAT(weights, i)[CAT(NEURONS, i)] \
75                        [CAT(INPUTSDEPTH, i)] \
76                        [CAT(FILTERWIDTH, i)] \
77                        [CAT(FILTERHEIGHT, i)]; \
78    int CAT(bias, i)[CAT(NEURONS, i)]; \
79    int CAT(inputs, INC(i))[CAT(OUTPUTSDEPTH, i)] \
80                            [CAT(OUTPUTSWIDTH, i)] \
81                            [CAT(OUTPUTSHEIGHT, i)]; \
82    const int CAT(INPUTSHEIGHT, INC(i)) = CAT(OUTPUTSHEIGHT, i); \
83    const int CAT(INPUTSWIDTH, INC(i)) = CAT(OUTPUTSWIDTH, i); \
84    const int CAT(INPUTSDEPTH, INC(i)) = CAT(OUTPUTSDEPTH, i); \
85  )
86  //end define
87
88  /** Weights load macro**/
89  #define LOAD_WEIGHTS(index,_) \
90  IF(EQUAL(CAT(LAYER, index), FULLYCONNECTED))\
91  ( \
92    for(int n = 0; n < CAT(NEURONS, index); ++n) \
93    { \
94      int buff; \
95      pc.scanf("%d", &buff); \
96      CAT(bias, index)[n] = buff; \
97      for(int k = 0; k < CAT(INPUTSDEPTH, index); ++k) \
98      { \
99        for(int j = 0; j < CAT(INPUTSWIDTH, index); ++j) \
100       { \
101         for(int i = 0; i < CAT(INPUTSHEIGHT, index); ++i) \
102         { \
103           pc.scanf("%d", &buff); \
104           CAT(weights, index)[n][k][j][i] = buff; \
105         } \
106       } \
```

```
107     } \
108   } \
109 ) \
110 IF(EQUAL(CAT(LAYER, index), CONVOLUTIONAL))\
111 ( \
112   for(int n = 0; n < CAT(NEURONS, index); ++n) \
113   { \
114     int buff; \
115     pc.scanf("%d", &buff); \
116     CAT(bias, index)[n] = buff; \
117     for(int k = 0; k < CAT(INPUTSDEPTH, index); ++k) \
118     { \
119       for(int j = 0; j < CAT(FILTERWIDTH, index); ++j) \
120       { \
121         for(int i = 0; i < CAT(FILTERHEIGHT, index); ++i) \
122         { \
123           pc.scanf("%d", &buff); \
124           CAT(weights, index)[n][k][j][i] = buff; \
125         } \
126       } \
127     } \
128   } \
129 )
130 //end define
131
132 /** NEURAL NETWORK macro **/
133 #define NEURAL_NETWORK(index,_) \
134 IF(EQUAL(CAT(LAYER, index), FULLYCONNECTED))\
135 ( \
136   for(int n = 0; n < CAT(NEURONS, index); ++n) \
137   { \
138     int sum = CAT(bias, index)[n] << N_FRAC; \
139     for(int k = 0; k < CAT(INPUTSDEPTH, index); ++k) \
140     { \
141       for(int j = 0; j < CAT(INPUTSWIDTH, index); ++j) \
142       { \
143         for(int i = 0; i < CAT(INPUTSHEIGHT, index); ++i) \
144         { \
145           sum += CAT(inputs, index)[k][j][i] \
146                 *CAT(weights, index)[n][k][j][i]; \
147         } \
148       } \
149     } \
150     sum = sum >> N_FRAC; \
151     CAT(inputs, INC(index))[n][0][0] = \
152       IF(EQUAL(CAT(AFUNC, index), POSLIN))(poslin) \
153       IF(EQUAL(CAT(AFUNC, index), SATLIN))(satlin) \
154       (sum); \
155   } \
156 ) \
157 IF(EQUAL(CAT(LAYER, index), MAXPOOLING))\
158 ( \
159   int j = -ARGN(CAT(PADDING, index), 2); \
160   for(int jout = 0; jout < CAT(OUTPUTSWIDTH, index); ++jout) \
161   { \
162     int i = -ARGN(CAT(PADDING, index), 0); \
```

```
163     for(int iout = 0; iout < CAT(OUTPUTSHEIGHT, index); ++iout) \
164     { \
165       for(int k = 0; k < CAT(INPUTSDEPTH, index); ++k) \
166       { \
167         int maximum = MINIMUM_INT; \
168         for(int jj = 0; jj < CAT(POOLWIDTH, index); ++jj) \
169         { \
170           for(int ii = 0; ii < CAT(POOLHEIGHT, index); ++ii) \
171           { \
172             if(i+ii >= 0 && i+ii < CAT(INPUTSHEIGHT, index) && \
173                j+jj >= 0 && j+jj < CAT(INPUTSWIDTH, index)) \
174             { \
175               if(CAT(inputs, index)[k][j+jj][i+ii] > maximum)  \
176               { \
177                  maximum = CAT(inputs, index)[k][j+jj][i+ii]; \
178               } \
179             } \
180             else \
181             { \
182               if(0 > maximum)  \
183               { \
184                 maximum = 0; \
185               } \
186             } \
187           } \
188         } \
189         CAT(inputs, INC(index))[k][jout][iout] = maximum; \
190       } \
191       i += ARGN(CAT(STRIDE, index), 0); \
192     } \
193     j += ARGN(CAT(STRIDE, index), 1); \
194   } \
195 ) \
196 IF(EQUAL(CAT(LAYER, index), CONVOLUTIONAL))\
197 ( \
198   for(int n = 0; n < CAT(NEURONS, index); ++n) \
199   { \
200     int j = -ARGN(CAT(PADDING, index), 2); \
201     for(int jout = 0; jout < CAT(OUTPUTSWIDTH, index); ++jout) \
202     { \
203       int i = -ARGN(CAT(PADDING, index), 0); \
204       for(int iout = 0; iout < CAT(OUTPUTSHEIGHT, index); ++iout) \
205       { \
206         int sum = CAT(bias, index)[n] << N_FRAC; \
207         for(int k = 0; k < CAT(INPUTSDEPTH, index); ++k) \
208         { \
209           for(int jj = 0; jj < CAT(FILTERWIDTH, index); ++jj) \
210           { \
211             for(int ii = 0; ii < CAT(FILTERHEIGHT, index); ++ii) \
212             { \
213               if(i+ii >= 0 && i+ii < CAT(INPUTSHEIGHT, index) && \
214                  j+jj >= 0 && j+jj < CAT(INPUTSWIDTH, index)) \
215               { \
216                 sum += CAT(inputs,index)[k][j+jj][i+ii] \
217                        *CAT(weights, index)[n][k][jj][ii]; \
218               } \
```

```
            } \
          } \
        } \
        sum = sum >> N_FRAC; \
        CAT(inputs, INC(index))[n][jout][iout] = \
          IF(EQUAL(CAT(AFUNC, index), POSLIN))(poslin) \
          IF(EQUAL(CAT(AFUNC, index), SATLIN))(satlin) \
          (sum); \
        i += ARGN(CAT(STRIDE, index), 0); \
      } \
      j += ARGN(CAT(STRIDE, index), 1); \
    } \
  } \
)
//end define
```

Listing B.2: File di definizione delle macro per la rete neurale (`nn_macros.h`).

B.4 File principale (`main.cpp`)

Nel file principale vengono dichiarate le strutture di dati ed eseguite le operazioni; gran parte del codice viene generato mediante l'espansione delle macro definite nel file `nn_macros.h`. Le espansioni vengono effettuate ricorsivamente utilizzando la macro `EVAL(macro)`, che permette di scansionare una macro più e più volte per effettuare delle espansioni, e la macro `REPEAT(loops, macro, args...)` che permette l'espansione di una macro tante volte quanto definito. Nel caso presente per ogni layer si esegue un'iterazione e si genera il codice secondo quanto definito nel file `user_parameters.h`. Per ulteriori dettagli leggere il paragrafo 3.1.

```
#include "mbed.h"
#include "pp_helper.h"
#include "user_parameters.h"
#include "nn_macros.h"

/**************************************************
    Constants and Variables declarations
**************************************************/

const int TOTAL_INPUTS = INPUTSHEIGHT0*INPUTSWIDTH0*INPUTSDEPTH0*PACKETS;
int inputs[TOTAL_INPUTS]; //declare the input vector
int inputs0[INPUTSDEPTH0][INPUTSWIDTH0][INPUTSHEIGHT0]; //input matrix

EVAL(REPEAT(LAYERS, DECLARE_WEIGHTS, ~)); //declare the weights and bias arrays

const int TOTAL_OUTPUTS = CAT(INPUTSHEIGHT, LAYERS)*
                          CAT(INPUTSWIDTH, LAYERS)*
                          CAT(INPUTSDEPTH, LAYERS)*
                          PACKETS;
int outputs[TOTAL_OUTPUTS]; //declare the output vector
```

```
21
22 /****************************************************
23     Functions declarations
24 ****************************************************/
25
26 inline int poslin(int x);
27 inline int satlin(int x);
28
29 /****************************************************
30     MICRO declarations
31 ****************************************************/
32
33 Serial pc(SERIAL_TX, SERIAL_RX, 57600); // 57600 bauds, 8-bit data, no parity
34 DigitalOut led(LED1);
35 Timer timer;
36
37 /****************************************************
38     MAIN
39 ****************************************************/
40
41 int main()
42 {
43     /** LOAD WEIGHTS ******************************/
44     pc.putc('w'); //send request
45     EVAL(REPEAT(LAYERS, LOAD_WEIGHTS, ~)); //expand load weights code
46
47     /** LOAD INPUTS *******************************/
48     for(int n = 0; n < TOTAL_INPUTS; ++n) //receive all the inputs
49     {
50         int buff;
51         pc.scanf("%d", &buff);
52         inputs[n] = buff;
53     }
54
55     led = 1;    //led ON -> the micro is elaborating
56     int inputs_prepared = 0;
57     int outputs_saved = 0;
58     timer.start();  //start time measurement
59     for(int m = 0; m < PACKETS; ++m)
60     {
61         /** PREPARE INPUTS ****************************/
62         for(int k = 0; k < INPUTSDEPTH0; ++k)
63         {
64             for(int j = 0; j < INPUTSWIDTH0; ++j)
65             {
66                 for(int i = 0; i < INPUTSHEIGHT0; ++i)
67                 {
68                     inputs0[k][j][i] = inputs[inputs_prepared++];
69                 }
70             }
71         }
72
73         /** EVALUATE NET*******************************/
74         static const int MINIMUM_INT = -(1 << (sizeof(int)*8-2));
75         EVAL(REPEAT(LAYERS, NEURAL_NETWORK, ~)); //expand neural network code
76
```

```
77          /** SAVE OUTPUTS ******************************/
78          for(int k = 0; k < CAT(INPUTSDEPTH, LAYERS); ++k)
79          {
80              for(int j = 0; j < CAT(INPUTSWIDTH, LAYERS); ++j)
81              {
82                  for(int i = 0; i < CAT(INPUTSHEIGHT, LAYERS); ++i)
83                  {
84                      outputs[outputs_saved++] =
85                          CAT(inputs, LAYERS)[k][j][i];
86                  }
87              }
88          }
89      }
90      led = 0;    //led OFF -> the micro has finished the elaboration
91      timer.stop();   //stop time measurement
92
93      /** SEND OUTPUTS ******************************/
94      for(int n = 0; n < TOTAL_OUTPUTS; ++n) //send the outputs to the pc
95      {
96          pc.printf("%d\n", outputs[n]);
97      }
98
99      /** SEND TIMING *******************************/
100     int elapsed_time = timer.read_us();
101     pc.printf("%d\n", elapsed_time);
102
103     /** PROGRAM END *******************************/
104     for(;;) //infinite loop
105     {
106         led = !led; //blinking led -> the micro has terminated
107         wait(1);
108     } //end of the program
109 }
110
111 /****************************************************
112     Functions implementation
113 ****************************************************/
114
115 inline int poslin(int x)
116 {
117     if(x < 0) return 0;
118     else return x;
119 }
120
121 inline int satlin(int x)
122 {
123     static const int ONE = 1 << N_FRAC; //constant for value '1' in fixed point
124     if(x < 0) return 0;
125     else if(x > ONE) return ONE;
126     else return x;
127 }
```

Listing B.3: File principale dell'implementazione su microcontrollore (`main.cpp`)

Bibliografia

[1] Michael Nielsen. *Neural Networks and Deep Learning*. 2018. URL: http://neuralnetworksanddeeplearning.com/.

[2] MathWorks™. *Deep Learning Toolbox™: User's Guide (R2018b)*. 2018. URL: https://it.mathworks.com/help/matlab/.

[3] *CS231n Convolutional Neural Networks for Visual Recognition*. Retrieved November 2018. URL: http://cs231n.github.io/convolutional-networks/.

[4] V. Sze et al. «Efficient Processing of Deep Neural Networks: A Tutorial and Survey». In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329. ISSN: 0018-9219. DOI: 10.1109/JPROC.2017.2761740.

[5] Hsinyu Tsai et al. «Recent progress in analog memory-based accelerators for deep learning». In: *J. Phys. D: Appl. Phys* 51.283001 (2018). DOI: https://doi.org/10.1088/1361-6463/aac8a5.

[6] M.Y Rafiq, G Bugmann e D.J Easterbrook. «Neural network design for engineering applications». In: *Computers & Structures* 79.17 (2001), pp. 1541–1552. ISSN: 0045-7949. DOI: https://doi.org/10.1016/S0045-7949(01)00039-6. URL: http://www.sciencedirect.com/science/article/pii/S0045794901000396.

[7] Intel. *Intel Cyclone 10 LP Device Datasheet*. 2018. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-10/c10lp-51002.pdf.

[8] STMicroelectronics. *STM32F303xD STM32F303xE datasheet*. 2016. URL: https://www.st.com/resource/en/datasheet/stm32f303re.pdf.

[9] *Arm Mbed website*. Retrieved November 2018. URL: https://www.mbed.com/en/.

[10] *C Preprocessor tricks, tips, and idioms*. Retrieved December 2018. URL: https://github.com/pfultz2/Cloak/wiki/C-Preprocessor-tricks,-tips,-and-idioms.

[11] *STM32 Nucleo-64 boards user manual*. 2017. URL: https://www.st.com/content/ccc/resource/technical/document/user_manual/98/2e/fa/4b/e0/82/43/b7/DM00105823.pdf/files/DM00105823.pdf/jcr:content/translations/en.DM00105823.pdf.

[12] A. Marchioni et al. «Rakeness-based Compressed Sensing of Surface Electro-MyoGraphy for Improved Hand Movement Recognition in the Compressed Domain». In: *2018 IEEE Biomedical Circuits and Systems Conference (BioCAS)*. 2018, pp. 1–4. DOI: 10.1109/BIOCAS.2018.8584763.

[13] STMicroelectronics. *STM32F446xC/E datasheet*. 2016. URL: https://www.st.com/resource/en/datasheet/stm32f446re.pdf.

[14] STMicroelectronics. *STM32F452xx datasheet*. 2018. URL: https://www.st.com/resource/en/datasheet/stm32l452re.pdf.

[15] Yann LeCun, Corinna Cortes e Christopher J.C. Burges. *The MNIST database of handwritten digits website*. Retrieved November 2018. URL: http://yann.lecun.com/exdb/mnist/.