ALMA MATER STUDIORUM
UNIVERISTÀ DI BOLOGNA – SEDE DI BOLOGNA
SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA IN BIOINGEGNERIA ELETTRONICA

TESI DI LAUREA MAGISTRALE

elaborata in:

ELABORAZIONE DEI SEGNALI NEI SISTEMI ELETTRONICI M

IMPLEMENTAZIONE IN FPGA DI UNA RETE NEURALE CONVOLUTIVA PROFONDA PER L'ELABORAZIONE IN TEMPO REALE DI IMMAGINI

Relatore:
Prof. Riccardo Rovatti

Tesi presentata da:
Elisa Naldi

Correlatori:
Prof. Mauro Mangia
Ing. Alex Marchioni
Ing. Stefano Santi - referente Datalogic

II[a] SESSIONE
ANNO ACCADEMICO 2017/2018

INDICE

1. INTRODUZIONE

1.1. Reti neurali

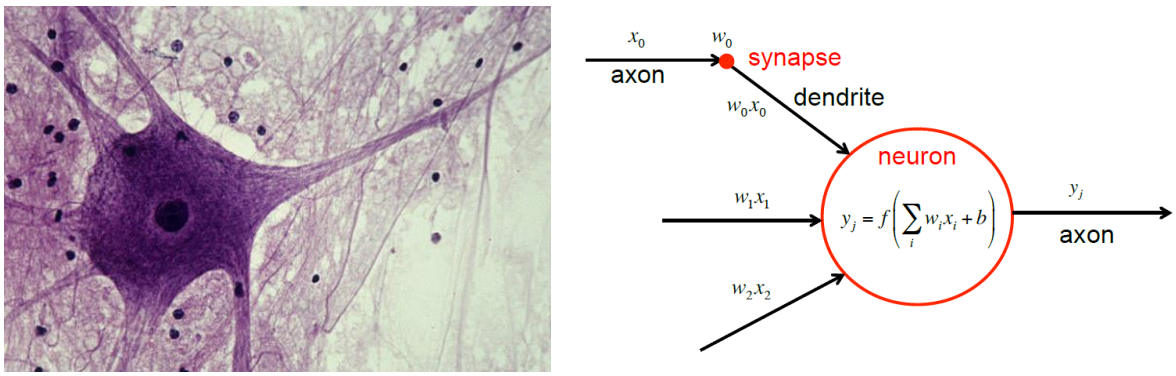
Le reti neurali sono sistemi di elaborazione adattivi composti da un insieme di unità elementari altamente connesse dette, per analogia biologica col sistema nervoso centrale umano, neuroni [1]. Generalmente le reti neurali, come suggerito dal nome, sono architetture e circuiti ispirate dalla struttura del cervello umano, nel quale sono presenti singole unità di elaborazione interconnesse in una rete di comunicazione molto complessa, attraverso strutture biologiche quali i dendriti e assoni.

Le analogie tra la struttura celebrale e le reti neurali sono numerose:

1. Presenza di nodi interconnessi (si veda l'immagine Imm.1.1.a.);
2. Presenza di neuroni che scalano e modulano il segnale trasmesso: nel caso biologico, ad esempio, in corrispondenza della sinapsi, ovvero la connessione tra strutture quali dendriti e assoni, il segnale viene scalato, aumentato o attenuato in base ai segnali e ai trasmettitori biochimici presenti;
3. Capacità di apprendere: le reti neurali infatti, attraverso una fase di training, ottimizzano la loro capacità di elaborazione e imparano senza essere specificatamente programmate per evolvere in una certa direzione;
4. Apprendimento quale ottimizzazione di funzioni di costo, dimostrato nel caso biologico attraverso approfonditi studi nel campo delle neuroscienze [2];
5. Esistenza di più funzioni di costo da ottimizzare: esistono infatti nella modellazione del cervello umano diversi funzionali in differenti aree celebrali, esattamente come si presentano molteplici e specifici layer nella rete neurale [3];
6. Plasticità: nel caso biologico legata alla diversa risposta agli impulsi modulata dalla concentrazione di neurotrasmettitori quali acetilcolina e dopamina; nel caso artificiale basata sulla progressiva variazione dei parametri interni durante la fase di learning (o training);
7. Intelligenza: le reti neurali si ritrovano ad essere il primo passo verso la creazione di un'intelligenza artificiale (artificial intelligence o AI); queste infatti possono essere viste come un vero e proprio modello del processo cognitivo umano (la comprensione dell'intelligenza naturale infatti può aiutare nel

processo di programmazione di una AI e, viceversa, la costruzione di agenti artificiali che emulano le caratteristiche di un apprendimento naturale può favorire una sua maggiore comprensione) [4];

8. Presenza di sistemi specializzati nella comprensione del problema affrontato: in entrambe le strutture infatti esistono siti specifici che analizzano la problematica computazionale da affrontare e permettono di ottenere soluzioni altamente efficienti.



Imm.1.1.a.: Analogia grafica tra un neurone umano e un neurone artificiale

Le immagini qui riportate sono state tratte dal lavoro di Ed Reschke e dall'articolo [5]

Come riportato dall'immagine Imm.1.1.a. soprariportata, la funzione descrittiva caratteristica del singolo neurone si riduce a una funzione generalmente non lineare basata su di una somma di prodotti, di cui si riporta un semplice esempio:

$$y_i = \begin{cases} \sum_{i=1}^{N} w_i * x_i + b_i & se \sum w * x + b > 0 \\ 0 & se \sum w * x + b \leq 0 \end{cases}$$

Dove:

- y_i è l'uscita del neurone, ovvero una somma dei valori di input pesati;
- x_i è l'i-simo ingresso del nodo elaborativo;
- w_i è il peso (weight) applicato ad uno specifico ingresso (per analogia biologica, questo peso deriva dalla particolare risposta sinaptica che si ha in corrispondenza dell'interconnessione tra assone e dendrite);
- b_i è il bias del nodo;
- L'uguaglianza con 0 nel caso di un risultato negativo rappresenta l'operazione non lineare caratteristica del neurone che verrà in seguito definita ReLU.

L’unione di un cospicuo numero di queste semplici unità di elaborazione così descritte porta a una struttura con un’enorme capacità di processing, ovvero a una rete neurale.
Esistono numerose e diverse applicazioni di una rete neurale, di cui si riportano alcuni esempi:

1. Elaborazione di una grande quantità di dati (ad esempio data mining, studi di correlazione o regressione e classificazione);
2. Elaborazione di immagini (ad esempio immagini biomedicali o letture di codici a barre o QR code);
3. Estrazione di particolari informazioni e feature dal dato elaborato (spesso caratterizzato da un’elevata complessità e non linearità) ed eventuale detection di anomalie o caratteristiche rilevanti [5];
4. Riduzione della dimensionalità del dato in ingresso attraverso la selezione e raccolta di un numero contenuto di dati in uscita esplicativi dell’interezza dell’input consegnato (questo può risultare particolarmente vantaggioso in termini di ottimizzazione dell’elaborazione, della trasmissione e della memorizzazione di un dato di dimensioni minori, ma con contenuto informativo uguale o approssimabile a quello originale) [6].

Le reti neurali possono essere utilizzate in queste applicazioni appena citate e molte altre grazie alla loro capacità di apprendimento; quest’ultimo risulta associato ad una delle due principali fasi di funzionamento delle reti neurali:

1. Fase di training (o learning), vera e propria fase di apprendimento;
2. Fase di processing (o testing), fase successiva alla prima, in cui il sistema si ritrova già ottimizzato e in grado di compiere l’elaborazione richiesta sul dato.

Come si può intuire, il training rappresenta uno step fondamentale del funzionamento di una rete neurale, in quanto durante quest’ultimo il sistema può andare ad affinare la sua elaborazione, andando a ricercare i pesi (weights w_i) dei neuroni che ottimizzano il processing richiesto. Il learning che la rete neurale si ritrova ad affrontare consta di una elaborazione iterativa di una elevata quantità di dati che, nel caso più comune di un supervised learning (apprendimento supervisionato), risultano etichettati (ovvero associati a una label) e classificati (con classe nota al sistema); il sistema elabora così il dataset di training disponibile e con continue correzioni dei w_i e delle bias b_i, attraverso,

ad esempio, un incremento basato sul gradiente, arriva al migliore set dei pesi w_i (detti anche fattori di scala) e delle polarizzazioni. Complessivamente questo viene definito quale "machine learning algorithm".

Un esempio di supervised learning è l'allenamento a cui si sottopone una rete neurale che ha come scopo la classificazione di una serie di immagini quali contenenti un certo oggetto (immagini positive) o quali non riportanti lo stesso oggetto (immagini negative). Il sistema neurale in questo caso affronta perciò un training in cui gli vengono sottoposte un elevato numero di immagini labelled (etichettate quali positive o negative) e durante il quale ricerca i migliori pesi e bias che minimizzano l'errore quadratico medio (usato come funzione di costo da ottimizzare) relativo, ad esempio, alla posizione dello specifico oggetto ricercato nell'immagine.

All'interno della famiglia delle reti neurali rientrano diverse architetture caratterizzate da differente complessità. In questa prima introduzione vengono trattate tre tipologie di reti neurali, in ordine sempre più complesse:

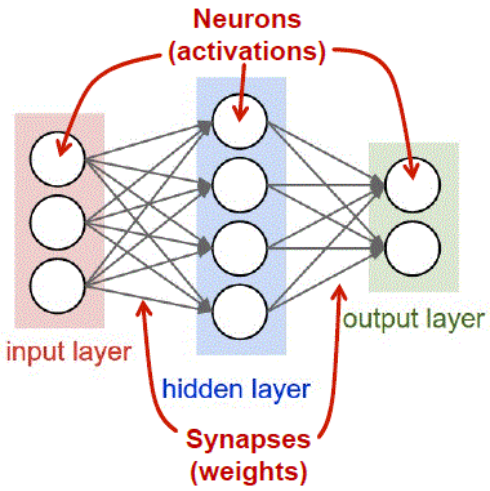
1. ANN: artificial neural network, ovvero rete neurale artificiale;
2. DNN: deep neural network, ovvero rete neurale profonda, caratterizzata perciò dalla capacità di affrontare un deep learning (la DNN è un sottoinsieme della più generica ANN);
3. CNN: convolutional neural network, ovvero rete neurale convolutiva profonda, sottoinsieme delle DNN che presenta dei particolari layer di elaborazione basati sull'operazione di convoluzione.

Le ANN rappresentano le più semplici e comuni reti neurali, rappresentabili da un grafo orientato di nodi e archi pesati e caratterizzate (solitamente, nella loro forma più semplice) da tre compartimenti elementari, detti layer. Nel grafo rappresentante le ANN ogni nodo, detto artificial neuron, contiene una funzione di attivazione (activation function) simile a quella descritta in precedenza [7].

Come si può osservare dall'immagine Imm.1.1.b., la struttura base delle artificial neural network presenta tre layer fondamentali:

1. Layer di input: layer di ingresso passivo (ovvero che non apporta modifiche al dato) che riceve il dato dal blocco a monte e lo duplica e distribuisce a tutti i nodi dell'hidden layer;

2. Hidden layer (ovvero strato nascosto): layer attivo di elaborazione che compie un'operazione di calcolo sul dato basata su pesi (si noti che in architetture più performanti e complesse, quali ad esempio le DNN e CNN, possono esistere più hidden layer);
3. Layer di output: layer attivo di uscita che sintetizza, attraverso una specifica combinazione dei risultati dell'hidden layer, la feature di uscita (o il contenuto set delle output-features, in caso si abbiano più uscite dalla rete neurale).



Imm.1.1.b.: Struttura base delle ANN – immagine tratta dall'articolo [5]

Le DNN (deep neural network) sono reti neurali più complesse che permettono un deep learning e deep processing; in altre parole queste sono neural network capaci di estrarre dai dati in ingresso features di alto livello (high-level features) [8]. Le DNN sono caratterizzate e identificabili dalla presenza di più di tre layer differenti (solitamente in un numero compreso tra 5 e 100).

Possedendo più comparti di elaborazione, le DNN (messe a confronto con le ANN) sono in grado di eseguire elaborazioni più complesse e approfondite quali (ad esempio):

1. Complicate elaborazioni di immagini, quali per esempio il detection dell'azione o movimento eseguito;
2. Riconoscimento vocale e dettatura;
3. Controllo degli autoveicoli auto-guidati;
4. Complessi processing di classificatori e riconoscitori;
5. Applicazioni nell'automazione industriale (ad esempio nell'Industry 4.0) [9].

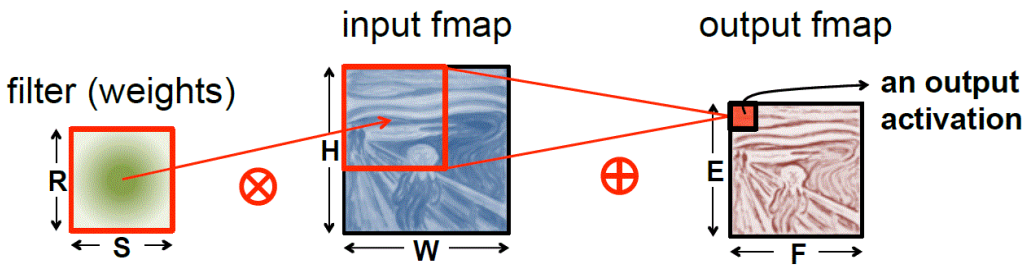
Presentando un numero maggiore di layer, le DNN si ritrovano a gestire un numero di parametri interni (pesi e bias) estremamente elevato; quest'ultimo se da una parte

aumenta la complessità del sistema, dall'altra permette al circuito di svolgere un maggior numero di compiti diversi e di affrontare e gestire con più facilità gli eventuali rumori (aumenta la robustezza ai rumori). Si noti infine che esiste un trade-off insito nella progettazione di una DNN: per aumentare le performance e l'accuratezza dell'output (obiettivi da massimizzare) è necessario aumentarne la complessità, il dispendio energetico e l'occupazione d'area (obiettivi da minimizzare).

Le CNN (convolutional neural network) sono DNN che presentano almeno un layer convolutivo (CONV) in aggiunta ai layer di tipo fully-connected (FC) già presenti nelle ANN e DNN (i layer FC sono così nominati per via del fatto che i nodi in essi contenuti sono completamente interconnessi, ovvero totalmente utilizzati nel calcolo interno). Le CNN sono nuovamente ispirate a una struttura biologica celebrale: l'organizzazione della corteccia visiva umana, che risponde a determinati stimoli luminosi in base alla porzione attivata di recettori nella retina. L'analogia biologica qui consta nella corrispondenza che esiste tra la geometria segmentata del campo visivo o recettivo umano e le diverse aree di elaborazione esistenti nelle CNN che compiono un processing partizionato del dato in ingresso, input che può essere rappresentato ad esempio da un'immagine.

Entrando più nel dettaglio, le CNN sono DNN fenestrate (windowed) e a condivisione di pesi (weight-shared) che si basano su di una operazione di convoluzione applicata a un ristretto dominio dell'input. Questo input, che per semplicità si assocerà a un'immagine, viene così strutturato e segmentato, attraverso il layer CONV, in piccole mappe 2D, dette kernel o 2D input feature maps (2D-IF-map), che vanno a convolversi con un'opportuna maschera di filtraggio 2D delle stesse dimensioni del kernel.

L'immagine Imm.1.1.c. mostra l'operazione di convoluzione applicata a una 2D-IF-map: un certo dominio dell'input (evidenziato da un riquadro in rosso) viene convoluto con la maschera di filtraggio (filter) per ottenere l'output feature map (2D-OF-map).



Imm.1.1.c.: Struttura base delle ANN – immagine tratta dall'articolo [5]

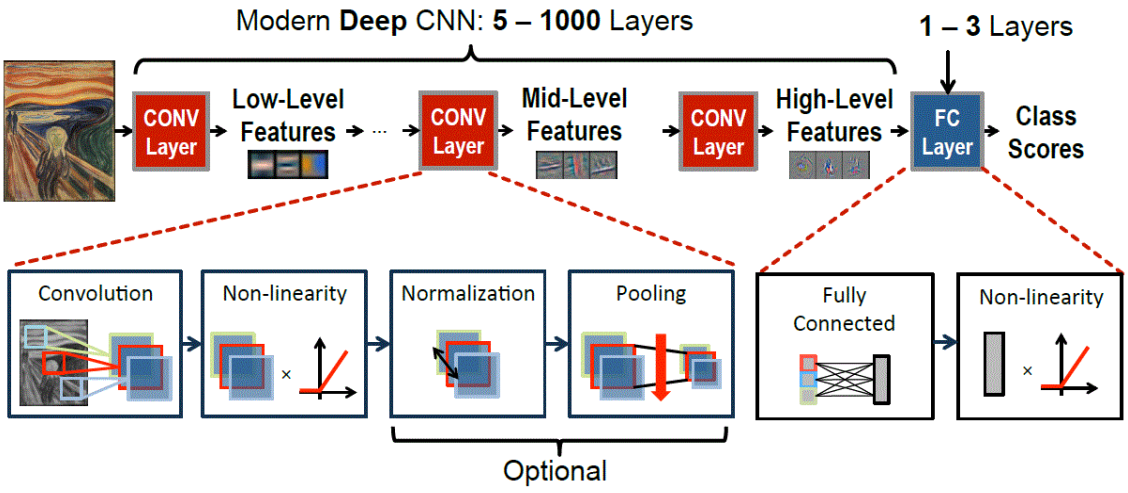
L’uscita dal layer convolutivo è esprimibile con la seguente espressione:

$$o(x,y) = \sum_{j=\frac{-(H-1)}{2}}^{\frac{H-1}{2}} \sum_{i=\frac{-(W-1)}{2}}^{\frac{W-1}{2}} i(x+i, y+j) * w\left(i + \frac{W+1}{2}, j + \frac{H+1}{2}\right)$$

Dove:

- $o(x,y)$ è l’uscita dal layer CONV (output ottenuto dall’esecuzione di un’operazione di selezione del kernel e di convoluzione;
- $i(x+i, y+j)$ è un generico ingresso (x+i; y+j) del CONV;
- $w\left(i + \frac{W+1}{2}; j + \frac{H+1}{2}\right)$ è uno specifico peso della matrice di filtraggio definita in un range orizzontale [1, W] e un range verticale [1, H];
- W e H sono la larghezza (width) e l’altezza (height) sia della 2D-IF-map sia della maschera di filtraggio.

Come si può osservare dall’immagine Imm.1.1.d., le CNN solitamente non sono composte solo da un CONV e da uno a tre FC (successivi al convolutivo), ma presentano molti altri layer differenti e opzionali che elaborano in modo specifico il dato consegnato loro.

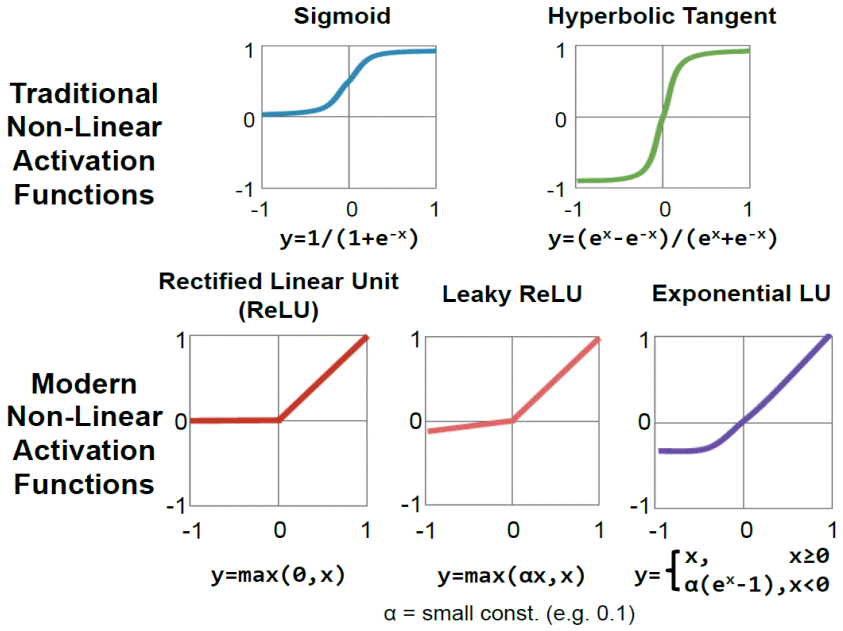


Imm.1.1.d.: Struttura di una CNN – immagine tratta dall’articolo [5]

I possibili layer opzionali presenti in una CNN sono:

1. Non linearità;
2. Pooling;
3. Normalizzazione.

Il layer di non linearità, che viene solitamente posto di seguito a un CONV o a un FC, è associato a un'operazione non lineare espressa, come osservabile nell'immagine Imm.1.1.e, da diverse tipologie di funzioni diverse. Storicamente le prime nonlinearity presentavano forme complesse quali le funzioni sigmoid o hyperbolic tangent; attualmente invece vengono utilizzate funzioni più semplici definite complessivamente quali LU (linear unit), perché riportano in uscita l'ingresso immutato se questo risulta essere positivo ($y = x \quad se\ x \geq 0$).



Imm.1.1.e.: Diverse non linearità applicabili in una CNN – immagine tratta dall'articolo [5]

All'interno della famiglia delle LU si ritrovano:

1. La ReLU (REctified Linear Unit): rappresenta sicuramente la LU più semplice e facile da implementare, presentando la seguente funzione:

$$y = \max(0, x) = \begin{cases} x & se\ x \geq 0 \\ 0 & se\ x < 0 \end{cases}$$

2. La leaky ReLU: presenta una leggera complicazione rispetto alla ReLU in quanto associa ai valori negativi non lo zero ma una risposta che si discosta leggermente da esso (α infatti è settato generalmente minore o uguale a 0.1):

$$y = \begin{cases} x & se\ x \geq 0 \\ \alpha * x & se\ x < 0 \end{cases}$$

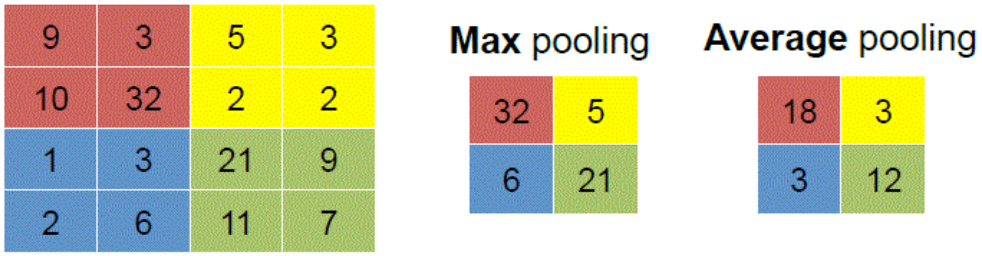
3. La exponential LU: presenta una funzione esponenziale al suo interno che rimodella le funzioni non lineari storiche:

$$y = \begin{cases} x & se\ x \geq 0 \\ \alpha * (e^x - 1) & se\ x < 0 \end{cases}$$

Nella pratica la non linearità solitamente utilizzata è la ReLU, perché più semplice, più facilmente implementabile e capace di offrire tempi di training e di testing più contenuti. Le altre tipologie di LU sono utilizzate invece in quelle elaborazioni che richiedono alte accuratezze e possono tollerare alti tempi di processing.

Il pooling è invece un layer che ha come scopo la riduzione della dimensione del dato; per compiere questa operazione questo blocco seleziona all'interno dell'immagine piccole maschere di pooling (solitamente non sovrapposte) su cui effettua una delle seguenti elaborazioni:

1. Un max-pooling, ovvero una selezione del massimo contenuto nella maschera;
2. Un average-pooling, ovvero la riproposizione in uscita della media dei valori contenuti nella maschera (si veda l'immagine Imm.1.1.f.);
3. Un'altra forma specifica di pooling necessaria per una certa elaborazione [10].



Imm.1.1.f.: Esempi di Pooling – immagine tratta dall'articolo [5]

Il layer di normalizzazione infine è un blocco che compie una vera e propria normalizzazione sul dato in ingresso (shiftandolo e scalandolo), con lo scopo di ottenere determinate e desiderabili proprietà statistiche (ad esempio media e varianza specifiche: $\mu = 0$ e $\sigma = 1$). Questo blocco apporta perciò modifiche al dato per aumentare la capacità di estrazione di informazioni e features dallo stesso; questa ottimizzazione è possibile grazie alla fase di training, grazie alla quale è capace di settare al meglio i parametri di normalizzazione presenti nella funzione normalizzante di seguito riportata:

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \varepsilon}} * \gamma + \beta$$

Dove:

- y è l'uscita dal blocco di normalizzazione;
- x è l'ingresso al blocco di normalizzazione;
- μ è la media ottima necessaria per ottimizzare l'estrazione di informazioni;
- σ è la deviazione standard ottima;

- ε è un parametro di correzione che evita problematiche numeriche;
- γ e β sono i parametri della normalizzazione da ottimizzare durante il training.

1.2. Reti neurali per l'elaborazione delle immagini

Negli ultimi decenni un enorme avanzamento nel campo dell'elaborazione delle immagini è stato possibile attraverso due principali elementi:

1. Un sempre più vasto dataset di immagini annotate (offerto ad esempio da ImageNet che presenta più di 1.2 milioni di immagini classificate in più di 1000 classi diverse) sul quale gli elaboratori possono compiere un adeguato training;
2. Le DNN ed in particolar modo le CNN [11].

L'utilizzo infatti di reti neurali convolutive profonde ha portato a delle performance molto elevate nell'image-processing, permettendo di ampliare molto le sue funzionalità. Esistono infatti molte diverse applicazioni delle CNN nell'elaborazione delle immagini, quali ad esempio:

1. Classificazione delle immagini (immagini positive o negative rispetto a una certa entità ricercata);
2. Riconoscimento di caratteri e analisi di documenti [12];
3. Object localization & detection (localizzazione di un oggetto nelle immagini);
4. Action recognition (riconoscimento dell'azione svolta nel frame in esame) e controllo della mobilità nei trasporti e nel movimento umano;
5. Image understanding (comprensione dell'immagine) e biometria;
6. Image modification (modifica dell'immagine, quali ad esempio detection dei bordi, sfocatura, goffratura e aumento del contrasto);
7. Analisi di immagini biomedicali e supporto nella diagnostica;
8. Automazione industriale e robotica;
9. Modellazione ambientale e riconoscimento di immagini radar in ambito militare.

Quello che permette alle CNN di spaziare in ambiti così diversificati è la loro plasticità caratteristica, associata a tre elementi fondamentali:

1. La variabilità del kernel e della matrice di filtraggio, che può essere settata opportunamente in base all'elaborazione da svolgersi;

2. La programmazione dei layer di elaborazione che possono estrarre feature a diversi livelli di profondità;
3. Il training, che va a variare i pesi e le bias dei neuroni in base alla specifica operazione da svolgere.

La scelta del kernel e della corrispondente matrice di filtraggio permette un'elaborazione diversificata e profonda del dato. Generalmente i kernel, per convenzione, vengono scelti di dimensioni contenute e con un numero di dati (o pixel) dispari sia per la base che per l'altezza (ad esempio kernel 3x3, 5x5, 7x7, 3x5, etc.), questo per avere un pixel su cui poter centrare la maschera di convoluzione. La maschera di filtraggio invece viene scelta in base all'operazione da applicare; nel caso di un image modification si possono scegliere ad esempio le seguenti maschere 3x3:

1. $\begin{bmatrix}0 & 0 & 0\\0 & 1 & 0\\0 & 0 & 0\end{bmatrix}$ matrice *no-effect* che non apporta alcuna modifica al dato;
2. $\begin{bmatrix}0 & 1 & 0\\1 & -4 & 1\\0 & 1 & 0\end{bmatrix}$ matrice *edge-detection* che permette l'individuazione dei contorni;
3. $\begin{bmatrix}0 & -1 & 0\\-1 & 5 & -1\\0 & -1 & 0\end{bmatrix}$ matrice *sharpening* (affilatura) che aumenta il contrasto;
4. $\begin{bmatrix}1 & 1 & 1\\1 & 1 & 1\\1 & 1 & 1\end{bmatrix}$ matrice *smoothing* che apporta una sfocatura (operazione di media);
5. $\begin{bmatrix}-1 & -1 & -1\\0 & 0 & 0\\1 & 1 & 1\end{bmatrix}$ matrice *gradient* che esegue un filtraggio derivativo;

La programmazione dei layer di elaborazione rappresenta un grado di libertà ulteriore che aumenta la plasticità del sistema neurale. Come si può osservare anche nell'immagine Imm.1.1.d. precedentemente proposta, è possibile programmare i diversi comparti della rete neurale con lo scopo di ottenere l'estrazione di features a livello di profondità sempre maggiore. Nell'ipotesi di avere tre livelli convolutivi CONV (come nell'immagine) questi possono essere dedicati all'individuazione di informazioni diverse; per esempio si possono settare i blocchi in questo modo:

1. Il primo layer estrae low-level features, quali ad esempio i contorni dell'immagine;

2. Il secondo layer individua informazioni e features di medio livello, come ad esempio la disposizione cromatica e il riconoscimento della forma e delle linee principali;
3. Il terzo layer estrae features di alto livello, quali ad esempio la classe di appartenenza dell'elaborato (classificazione) [13].

Il training rappresenta l'ultimo grado di plasticità delle CNN che permette di ottimizzare l'image-processing. Questo, andando a ottimizzare i parametri interni del sistema (weights e bias), minimizza gli errori di elaborazione e permette alla rete di eseguire compiti anche molto differenti tra loro. Un esempio classico di training è il supervised learning di una rete neurale che ha come scopo la classificazione di immagini. In questo caso il sistema va ad associare alle diverse classi di riconoscimento (ad esempio cane, gatto, tavolo, etc.) uno score, ovvero un punteggio che rappresenta la probabilità di appartenenza a quella classe; la classe con lo score più alto rappresenterà la classificazione del dato. Il learning in tutto questo ha il delicato compito di individuare i parametri ottimali che massimizzano la probabilità di una corretta classificazione. Una tipologia classica di aggiornamento dei parametri è chiamata gradient descent; in quest'ultima i pesi e le bias vengono aggiornati attraverso un incremento che è funzione del gradiente di un funzionale o funzione di costo F. Un esempio di gradient descent è:

$$w_{ij}^{t+1} = w_{ij}^{t} - \alpha \frac{\partial F}{\partial w_{ij}^{t}}$$

Dove:

- w_{ij}^{t+1} è il nuovo peso al tempo t+1;
- w_{ij}^{t} è il peso precedente al tempo t;
- α è il coefficiente di aggiornamento, che indica come i pesi devono variare per massimizzare la probabilità dell'ottenimento di una corretta classificazione;
- $\frac{\partial F}{\partial w_{ij}^{t}}$ è il gradiente della funzione di costo F definita su una proprietà dell'input.

1.3. Reti neurali per applicazioni in Industry 4.0

L'Industria 4.0 è il nome che si associa alla cosiddetta quarta rivoluzione industriale che risulta caratterizzata da tre fattori fondamentali:

1. Estesa e capillare automazione industriale e uso di sistemi cyberfisici, ovvero sistemi fisici strettamente coadiuvati e interconnessi con sistemi informatici e altre entità dello stesso tipo;
2. Applicazione di tecnologie produttive innovative che permettano un tasso di crescita e produttività molto elevato;
3. Aumento considerevole della capacità produttiva e della qualità del prodotto e parallela diminuzione dei tempi di produzione [14].

L'Industry 4.0 è perciò sinonimo di grande innovazione, qualità e velocità di produzione e si ritrova ad essere rappresentata dalla smart factory ("fabbrica intelligente") che a sua volta è caratterizzata da tre smart-element:

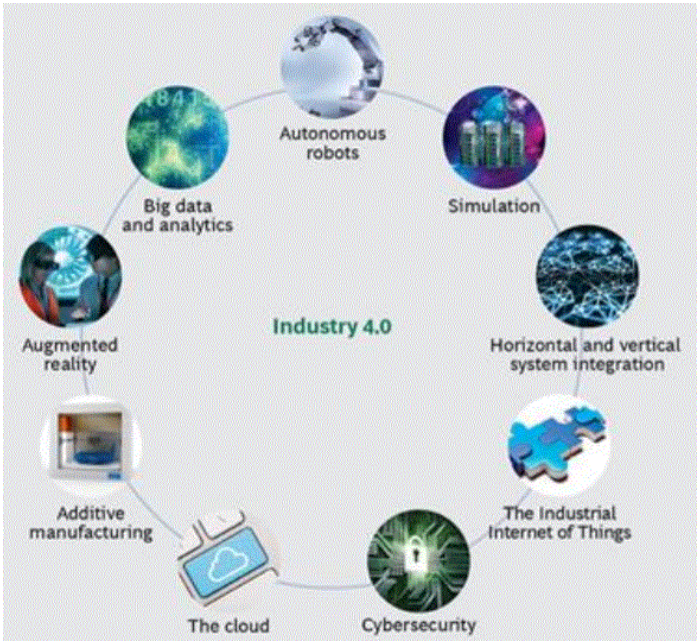
1. Smart production: uso di tecnologie di produzione innovative che permettono una coordinazione e cooperazione tra tutti gli elementi produttivi presenti (si ha così collaborazione tra strumenti e macchine diverse);
2. Smart services: compresenza di servizi informatici e tecnici che vanno a integrare e coordinare diverse macchine e sistemi, diversi reparti e persino diverse aziende (ad esempio coordinazione tra fornitore e cliente);
3. Smart energy: tendenza a ottimizzare i consumi energetici e a prediligere l'energia di tipo sostenibile.

Come descritto nello studio *Industry 4.0: The Future of Productivity and Growth in Manufacturing Industries* del Boston Consulting Group [15] sono nove le tecnologie (definite abilitanti) alla base dell'Industria 4.0 (si veda l'immagine Imm.1.3.a.):

1. Automazione robotica: sistemi di produzione automatici e avanzati con elevato livello di interconnessione e autogestione;
2. Uso di simulazioni: simulazioni sempre più presenti e capaci di aumentare l'affidabilità del prodotto e diminuirne i tempi di progettazione e produzione (la simulazione permette perciò un'ottimizzazione dei processi produttivi);
3. Sistemi di integrazione orizzontali e verticali: scambio di informazioni su più livelli, sia tra strutture poste su differenti livelli gerarchici (integrazione

verticale), sia tra entità appartenenti allo stesso livello della gerarchia produttiva (integrazione orizzontale);

4. Internet of things: comunicazione tra le diverse unità di produzione elementari (presenza di un internet, ovvero una connessione, che lega le diverse macchine e comparti di produzione);
5. Cybersecurity: aumento della cybersicurezza per minimizzare la probabilità di una manomissione esterna con conseguente perdita economica e produttiva;
6. Cloud: storage di dati e sistemi di cloud-computing che permettono una ottimale gestione dei dati raccolti e una parallela analisi degli stessi (qui è presente anche un sistema di riduzione della dimensione del dato e di estrazione delle features dallo stesso);
7. Produzione additiva: produzione, contrapposta a quella sottrattiva, in cui un oggetto 3D è costruito attraverso la sovrapposizione e stratificazione di materiali diversi;
8. Realtà aumentata: sistemi VR (sistemi di visione con realtà aumentata) che aumentano la produttività e migliorano le azioni svolte dagli operatori;
9. Analisi e gestione di big data: tecniche di gestione di quantità cospicue di dati attraverso predizione, riduzione della dimensionalità ed estrazione di features e informazioni significative.



Imm.1.3.a.: Le nove tecnologie abilitanti dell'industry 4.0 – immagine tratta dall'articolo [15]

Osservando con attenzione queste nove tecnologie abilitanti, si può capire il motivo dell'utilizzo delle reti neurali nell'Industry 4.0: le DNN e le CNN infatti possono essere utilizzate in quattro delle suddette nove tecnologie, ovvero l'automazione robotica, le simulazioni, il cloud e l'analisi e gestione di big data (senza comunque escludere eventuali altre piccole collaborazioni e utilizzi nelle restanti tecnologie).

1.4. FPGA per reti neurali e confronto con ASIC

FPGA e ASIC sono due tipi di tecnologie per l'implementazione di circuiti integrati che presentano caratteristiche molto differenti e spesso discordanti. Come si argomenterà in seguito, la tecnologia utilizzata preferenzialmente nella progettazione di una rete neurale (soprattutto DNN e CNN) è rappresentata dalle FPGA. Si vuole perciò in questo capitolo descrivere le caratteristiche specifiche dell'una e dell'altra tecnologia, le principali differenze tra esse e la motivazione per una scelta preferenziale orientata verso le FPGA.

Le FPGA (field programmable gate array) rappresentano una tipologia di circuito integrato le cui funzionalità sono riprogrammabili e gestibili attraverso linguaggi hardware o HDL (hardware description language), quali ad esempio VHDL e verilog. La struttura base di un FPGA è rappresentata da una matrice di CLB (configurable logic block), ovvero blocchi logici riconfigurabili, che si ritrovano interconnessi attraverso delle programmable connections (connessioni programmabili) realizzate attraverso un insieme di switch matrix composte da pass transistor programmabili. Come si può intuire da questa semplice descrizione, in questa tipologia di struttura sono presenti molti gradi di libertà per una possibile riprogrammazione, sia a livello dei CLB (la riprogrammazione di CLB permette di realizzare le diverse funzioni logiche necessarie), sia a livello delle connessioni programmabili. Per completare la struttura base delle FPGA, oltre all'organizzazione interna di blocchi logici e connessioni, è necessario descrivere anche gli IOB (input-output block); gli IOB sono blocchi che hanno il compito di interfacciarsi col mondo esterno, gestendo l'input da consegnare ai CLB interni e l'output presente ai pin di uscita. Entrando più nel dettaglio nei CLB, questi internamente sono composti da un numero contenuto di logic cell alla cui base vi sono una o più LUT (look-up table) programmabili, almeno un registro di memoria e una

serie di multiplexer. Attraverso questa struttura e alla possibile interconnessione tra più LUT, si possono implementare anche funzioni logiche molto complesse.

I principali vantaggi delle FPGA sono:

1. Il contenuto time to market, in quanto l'hardware di una FPGA (in confronto a una ASIC) risulta più facile da progettare;
2. La programmabilità e riprogrammabilità delle FPGA (riprogrammazione eseguita direttamente dall'utente finale) che permette loro di essere utilizzate in svariati campi di applicazione, essere aggiornate nelle funzionalità ed essere modificate e corrette (in caso di errore);
3. La semplicità del design;
4. L'elevato tempo di vita associato alla capacità di aggiornarsi;
5. La capacità di correggere eventuali errori presenti.

Gli svantaggi invece delle FPGA sono:

1. La maggiore occupazione d'area (anche molto maggiore rispetto agli ASIC);
2. Un maggiore costo di produzione rispetto gli ASIC nel caso di una produzione su vasta scala (con un numero di pezzi prodotti molto elevato, maggiore di 400K unità);
3. Le performance che non sono ottimizzate per la specifica operazione da eseguire;
4. Il consumo di potenza elevato.

Gli ASIC (application specific integrated circuit) sono una tipologia di circuito integrato progettato per rispondere a una particolare e ben precisa applicazione. Essendo una tipologia di device special purpose, ovvero focalizzata sulla risoluzione di una specifica applicazione di calcolo, l'ASIC può consentire notevoli ottimizzazioni in termini di minimizzazione della velocità di calcolo, dei consumi di potenza e della massimizzazione generale delle performance (ottimizzazioni non ottenibili attraverso un device general purpose). Nonostante questi notevoli vantaggi, gli ASIC presentano un grosso svantaggio: costi e tempi di sviluppo molto elevati (che si traducono in elevati time to market). Essendo lo sviluppo così esoso, gli ASIC sono perciò generalmente scelti esclusivamente per volumi ingenti di unità prodotte. Si noti inoltre che, presentando un circuito permanente, gli ASIC non sono riconfigurabili.

Le differenze principali tra FPGA e ASIC, già descritte precedentemente, sono raccolte nella seguente tabella Tab.1.4.a. [16] (dove vengono sottolineate le ottimizzazioni delle specifiche).

Specifica da ottimizzare	FPGA	ASIC
Time to market	Contenuto	Elevato
Design del circuito	Semplice	Complesso
Costo di produzione (per volume prodotto elevato)	Elevato	Contenuto
Performance	Medie	Elevate
Consumo di potenza	Elevato	Contenuto
Occupazione d'area	Media-alta	Bassa
Programmabile e riconfigurabile	Si	No
Capacità di correggere gli errori	Si	No
Tempo di vita	Alto	Medio-basso

Tab.1.4.a.: Confronto tra FPGA e ASIC

Analizzando le caratteristiche e le differenze di queste due tecnologie, è possibile capire perché si prediligono gli FPGA nella programmazione delle reti neurali. Le reti neurali infatti hanno come principali caratteristiche la plasticità e riprogrammabilità (ad esempio nel numero e tipologia di layer per svolgere azioni e operazioni differenti), caratteristiche importanti soprattutto nella fase di iniziale sviluppo e aggiornamento, e una tecnologia che può offrire loro queste proprietà è rappresentata dalle architetture FPGA. Inoltre la facilità di progetto di quest'ultime, permette una semplice programmazione di un componente caratterizzato da una sua insita complessità.

2. SCOPO DELLA TESI

Lo scopo della tesi qui esposta è quello di fornire una descrizione dettagliata della struttura di un coprocessore neurale (CNN), implementato in VHDL, per l'elaborazione di immagini in tempo reale e una trattazione delle performance della sua sintesi.

La descrizione seguirà la gerarchia dell'implementazione. Si descriverà dapprima la struttura generale del coprocessore presentando i diversi sottosistemi e layer presenti; in seguito si esamineranno nel dettaglio i diversi componenti riportando esempi grafici esplicativi e piccoli estratti del codice implementato.

La trattazione delle performance andrà invece a descrivere due aspetti fondamentali del corretto funzionamento di questo progetto:

1. La correttezza logica funzionale;
2. Le performance in termini di velocità di funzionamento e occupazione di area espressa in numero di blocchi base.

Per trattare la correttezza logica funzionale si utilizzeranno due serie di codici di programmazione (di cui si citeranno brevi estratti):

1. Un testbench per l'applicativo ModelSim con codici di assegnazione del dato;
2. Un codice Matlab che riproduce a livello software il comportamento del componente implementato a livello hardware.

La verifica consisterà nell'ottenimento degli stessi risultati attraverso la simulazione Vsim e l'analogo software via Matlab.

Per ottenere invece i dati relativi alla performance del sistema si utilizzerà una funzionalità dell'applicativo Quartus Prime 17.1 (programma usato per la scrittura, verifica e sintesi del coprocessore neurale qui presentato), ovvero la compilazione e sintesi (Processing – Start Compilation). Attraverso la sintesi finale sarà così possibile ottenere un resoconto dettagliato delle prestazioni del componente e una valutazione quantitativa della sua velocità e occupazione d'area.

In ultimo, verranno riportati alcuni possibili sviluppi, miglioramenti e generalizzazioni applicabili al coprocessore implementato per aumentarne la funzionalità e migliorarne le prestazioni. In particolar modo si sottolineerà l'importanza della parametrizzazione per rendere il funzionamento del progetto il più possibile generico e ottimizzabile per diverse esigenze e campi d'applicazione.

3. PROGETTO DI TESI

3.1. *ChainOfComponents* – struttura del coprocessore neurale

ChainOfComponents è il nome del coprocessore neurale implementato per il seguente progetto di tesi; deriva il suo nome dalla struttura a catena o a cascata dei diversi blocchi di elaborazione in esso presenti.

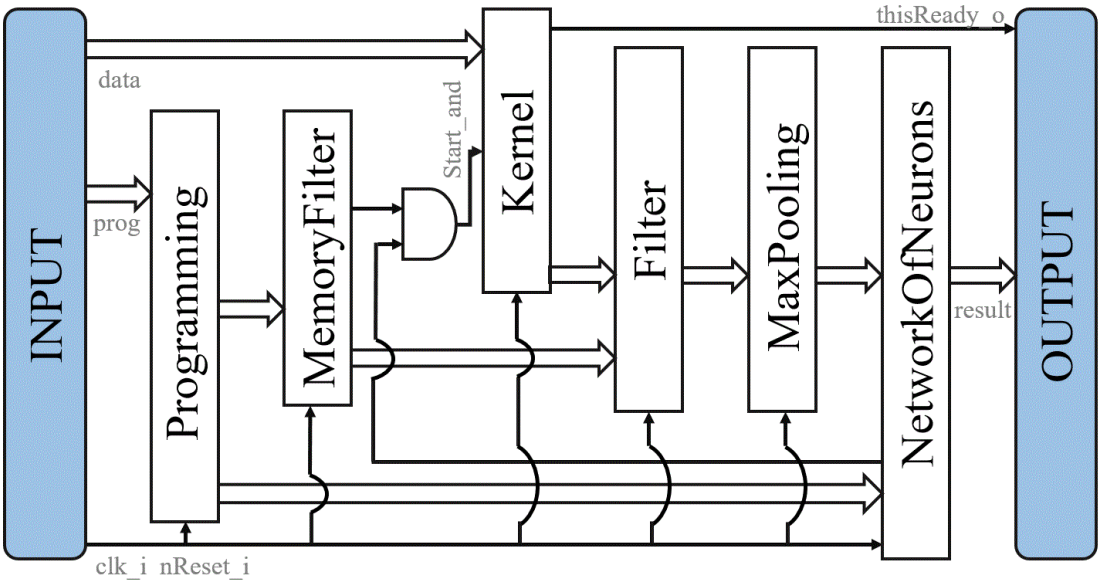
La funzionalità di *ChainOfComponents* consta nell'elaborazione di una immagine letta e caricata pixel per pixel in modo sequenziale con lo scopo di ottenere in uscita otto feature qualificanti le caratteristiche del caricato. Attraverso i suoi diversi layer *ChainOfComponents* applica un'operazione di convoluzione su opportune maschere dell'immagine (gestita ai bordi con la politica dello zero-padding) ottenendo una nuova immagine filtrata con le stesse dimensioni; successivamente riduce la dimensione dell'elaborato applicando un operazione di max-pooling (ovvero la selezione del massimo entro una maschera di opportuna grandezza) e consegna infine l'immagine così ridotta ad una rete di neuroni, costruita su due stadi fully-connected, che ne riduce ulteriormente la numerosità fino ad ottenere in uscita gli otto parametri ricercati.

Si riporta nella tabella Tab.3.1.a le dimensioni dei diversi parametri utilizzati nella sintesi finale del progetto *ChainOfComponents*; queste grandezze verranno successivamente citate sia nelle descrizioni dei diversi componenti, sia nei capitoli successivi come standard di sintesi.

Grandezza	Dimensione
Dimensione dell'immagine in ingresso	64x64 dati (pixel)
Numero di bit del dato in ingresso	8 bit
Dimensione della maschera del blocco *Kernel* (convoluzione)	3x3 dati
Numero di bit dei pesi del filtraggio	4 bit
Numero di bit del dato dopo il filtraggio	12 bit
Dimensione della maschera del blocco *MaxPooling*	4x4 dati
Numero di bit dei pesi dei neuroni nei due diversi stadi fully-connected	4 bit
Numero di dati su cui il singolo neurone del primo stadio fully-connected compie il suo calcolo	256 dati
Numero di neuroni del primo stadio fully-connected	64 neuroni
Numero di bit del dato dopo il primo stadio fully-connected	16 bit
Numero di dati su cui il singolo neurone del secondo stadio fully-connected compie il suo calcolo	64 dati
Numero di neuroni del secondo stadio fully-connected	8 neuroni
Numero di bit del dato dopo il secondo stadio fully-connected, ovvero numero di bit della singola uscita	16 bit
Numero di bit in uscita dal *ChainOfComponents*	8x16 bit

Tab.3.1.a.: Riassunto delle dimensioni dei dati

Nell'immagine Imm.3.1.a. è riportata una semplificazione grafica riassuntiva che evidenzia i componenti base di *ChainOfComponents*.



Imm.3.1.a.: Struttura base di ChainOfComponents

ChainOfComponents presenta i seguenti sottocomponenti:

1. *Programming*: blocco di programmazione che accoglie tutti gli ingressi inerenti al settaggio dei pesi del filtro e dei neuroni della rete neurale restituendoli ai relativi blocchi *MemoryFilter* e *MemoryNeurons* (quest'ultimo non visibile nell'immagine Imm.3.1.a. in quanto presente nel blocco *NetworkOfNeurons*);
2. *MemoryFilter*: blocco di sincronizzazione tra i dati in ingresso a *Filter* e i pesi del filtraggio, precedentemente precaricati in memoria durante la programmazione;
3. *AND dello Start_and*: semplice blocco logico AND che compie l'operazione logica & tra i segnali *Start* provenienti dai blocchi *MemoryFilter* e *MemoryNeurons* ottenendo *Start_and*, segnale di avvenuta programmazione che rende attiva la catena di componenti, attivando in primis il blocco *Kernel*;
4. *Kernel*: blocco di gestione dell'operazione di convoluzione che seleziona dall'immagine caricata un pixel alla volta le maschere di interesse su cui applicare poi l'operazione di filtraggio;
5. *Filter*: blocco che applica il filtraggio (composto da una somma di prodotti tra dato e peso) e la successiva operazione ReLU ai dati della maschera del *Kernel*;

6. *MaxPooling*: blocco di max-pooling che seleziona per ogni maschera individuata sull'immagine filtrata (senza overlap) il massimo e lo restituisce in uscita;
7. *NetworkOfNeurons* (letteralmente rete di neuroni): blocco di rete neurale costituito da due stadi fully-connected da 64 e 8 neuroni (ogni neurone sarà preceduto da un opportuno blocco *MemoryNeurons* che sincronizzerà il dato in ingresso al neurone col peso opportuno).

ChainOfComponents rappresenta perciò il file riassuntivo di progettazione che connette ognuno dei componenti precedentemente elencati e descritti. Per rendere possibile questa connessione si sono utilizzate le seguenti righe di codice e accorgimenti:

1. Utilizzo delle opportune librerie e componenti:

   ```
   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_arith.all;
   use work.MemoryFilter; (uso del componente MemoryFilter) etc.
   ```

2. Assegnazioni dei *generic* necessari attraverso i quali si rende parametrica l'implementazione del coprocessore rendendolo adattabile a diverse situazioni applicative:

   ```
   KERNEL_WIDTH: integer :=3;
   POOL_DIM: integer :=4; ect.
   ```

3. Definizione delle porte di ingresso e uscita e dei segnali interni:

   ```
   clk_i:  in std_logic; (ingresso)
   resultValid_o: out std_logic_vector(NUM_OF_NEURONS2-1 downto 0); (uscita)
   signal weightValid_prog: std_logic; (segnale interno)
   ```

4. Piazzamento dei diversi componenti e mapping dei relativi *generic* e *port* (esempio del blocco *MemoryFilter*):

   ```
   MemoryFilter_component: entity MemoryFilter
   generic map(
           MAX_WEIGHTS_WIDTH => MAX_WEIGHTS_WIDTH,
           WEIGHT_WIDTH => WEIGHT_WIDTH,
           KERNEL_WIDTH => KERNEL_WIDTH,
           KERNEL_HEIGHT => KERNEL_HEIGHT)
   port map(
           clk_i => clk_i,
           nReset_i => nReset_i,
           weightValid_i => weightValid_prog,
           weightValue_i => weightValue_prog,
           endOfWeights_i => endOfWeights_prog,
           weightsValue_o => weightsValue_m1,
           Start_o => Start_m1);
   ```

   In questa breve citazione del codice si può osservare il mapping tra i parametri *generic* del componente e quelli di *ChainOfComponents* e le connessioni tra i segnali di quest'ultimo coi segnali in ingresso e uscita da *MemoryFilter*.

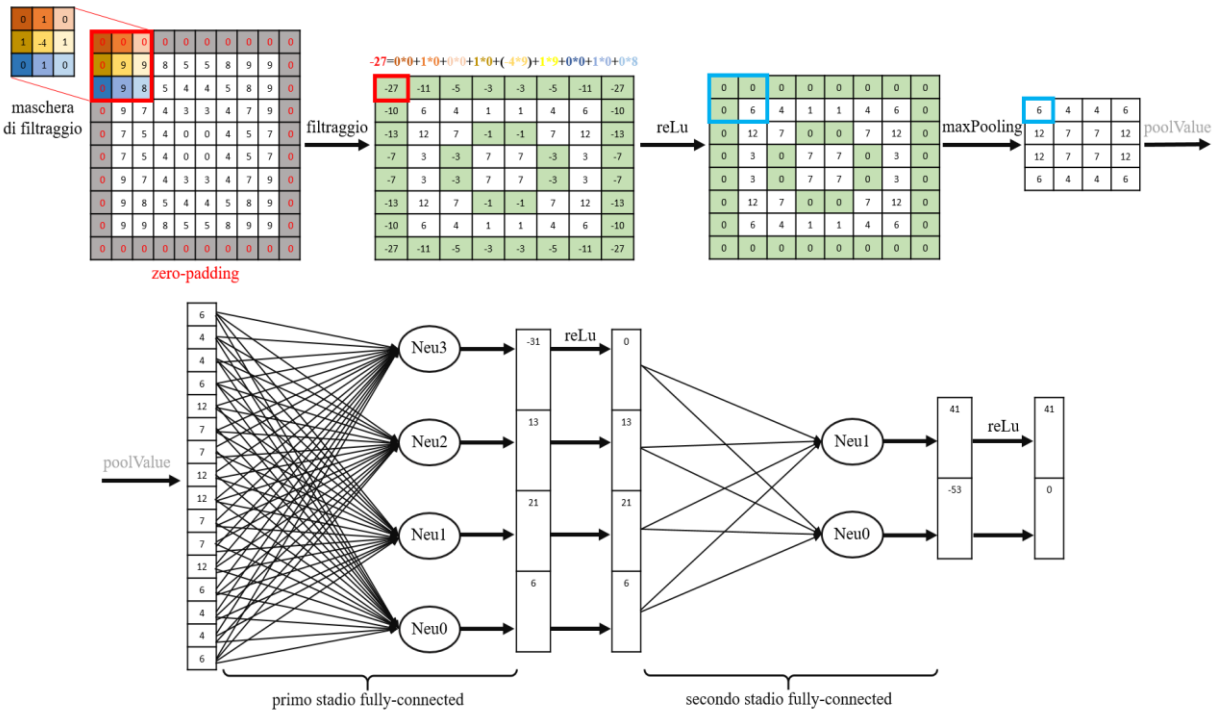
ChainOfComponents presenta complessivamente 42 pin di ingresso e 137 pin di uscita. Il numero di pin in ingresso e in uscita è stato scelto in base alle esigenze di elaborazione e calcolo e alla dimensione dei diversi dati e maschere usati (si veda la tabella Tab.3.1.a. per un riassunto delle dimensioni dei dati).

Nella tabella Tab.3.1.b. sono riportate gli ingressi e le uscite di *ChainOfComponents* con relativo numero di bit e utilità.

Nome del segnale	Ingresso/Uscita	Numero di bit	Utilità del segnale
clk_i	Ingresso	1 bit	Clock di sistema
nReset_i	Ingresso	1 bit	Reset dell'esecuzione dell'elaborazione
nReset_prog_i	Ingresso	1 bit	Reset dell'esecuzione della programmazione
progValid_i	Ingresso	1 bit	Validità del dato di programmazione
progValue_i	Ingresso	26 bit	Valore del dato di programmazione
dataValid_i	Ingresso	1 bit	Validità del dato in ingresso
dataValue_i	Ingresso	8 bit	Valore del dato in ingresso
endOfLine_i	Ingresso	1 bit	Segnale che indica la fine della linea dell'immagine
endOfFrame_i	Ingresso	1 bit	Segnale che indica la fine dell'immagine
nextReady_i	Ingresso	1 bit	Segnale che il blocco a valle è pronto a ricevere il dato
resultValid_o	Uscita	8 bit	Validità del dato in uscita
resultValue_o	Uscita	128 bit	Valore del dato in uscita
thisReady_o	Uscita	1 bit	Segnale che il blocco *Kernel* è pronto a ricevere nuovi dati dal blocco a monte

Tab.3.1.b.: Riassunto dei segnali di ingresso e di uscita di ChainOfComponents

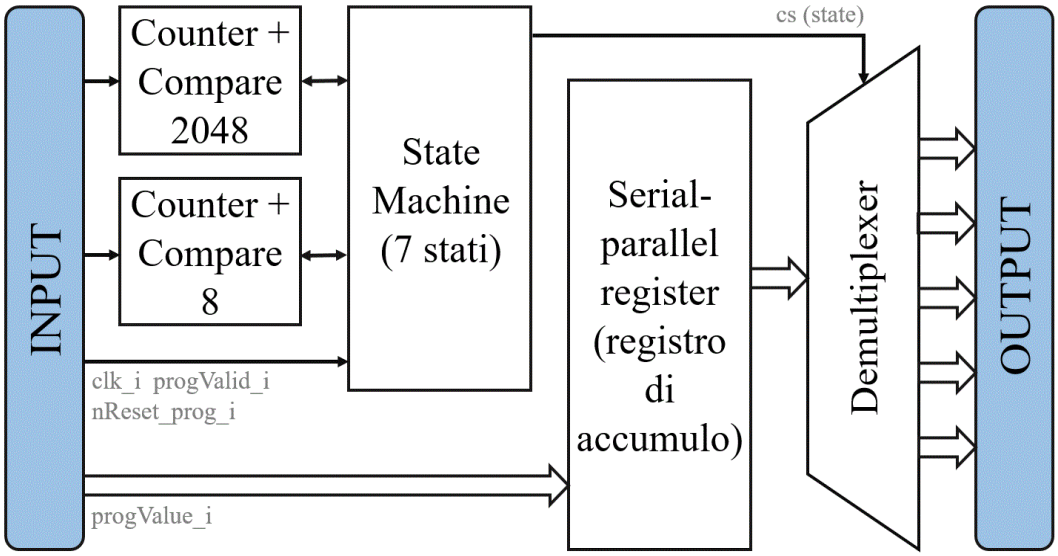
Il funzionamento di *ChainOfComponents* è riassunto nell'immagine Imm.3.1.b.; verranno esplicate le diverse elaborazioni ivi presenti nei successivi sottocapitoli.



Imm.3.1.b.: Funzionamento di ChainOfComponents

3.2. *Programming* – blocco di programmazione

Il blocco *Programming* è il primo componente che si incontra in *ChainOfComponents*. Il suo scopo è gestire la programmazione del filtro e dei neuroni degli stadi fully-connected, permettendo il salvataggio, la memorizzazione e la corretta assegnazione dei pesi del filtro e dei diversi neuroni presenti. Nell'immagine Imm.3.2.a. è riportata la struttura interna di questo componente con l'esplicazione di alcuni dei segnali più importanti.



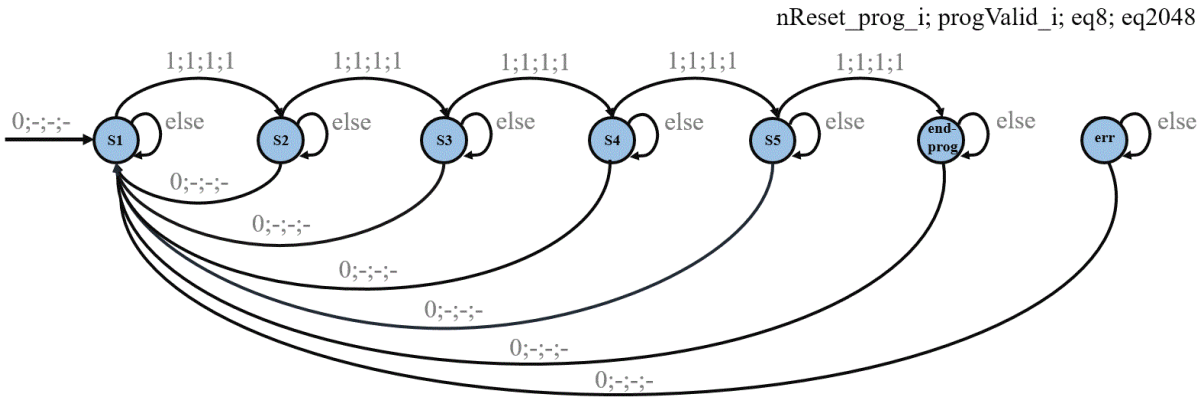
Imm.3.2.a.: Struttura del blocco di programmazione Programming

Il funzionamento di *Programming* è controllato da una state-machine a sette stati, la cui alternanza permette, attraverso il segnale *current state* (*cs*), il controllo del demultiplexer in uscita; quest'ultimo blocco collega l'ingresso a una delle possibili uscite disponibili, consegnando perciò il dato ad un'unica destinazione (in altre parole presenta un comportamento opposto al componente "multiplexer" che porta in uscita uno solo dei segnali in ingresso). L'immagine Imm.3.2.b. riporta il diagramma di stato della state-machine di *Programming*.

Si possono osservare i sette diversi stati presenti:

1. Cinque stati di programmazione (*S1*, *S2*, *S3*, *S4* e *S5*), la cui numerazione è associata alla diversa destinazione del dato di programmazione;
2. Uno stato accessorio *ENDPROG*, che segnala la fine della programmazione (è lo stato mantenuto da *Programming* durante il normale funzionamento del coprocessore);

3. Uno stato accessorio *ERR*, che informa l'utilizzatore dell'errata programmazione del coprocessore.



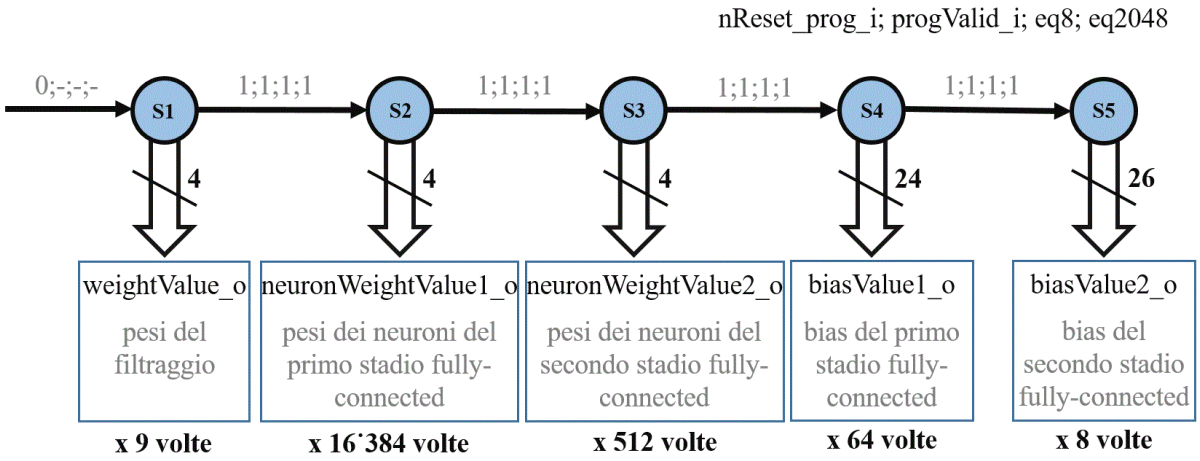
Imm.3.2.b.: Diagramma di stato della state-machine del blocco Programming (particolare dalla netlist)

Si può notare dall'immagine soprariportata come l'uscita dallo stato *ERR* sia possibile solo in seguito a un reset di programmazione, ovvero con *nReset_prog_i* pari a 0; quest'ultimo è un segnale attivobasso che provoca (in caso in cui sia pari a 0) il passaggio allo stato *S1*, qualunque sia lo stato corrente. È inoltre possibile osservare la combinazione di segnali che permettono il passaggio da uno stato all'altro della normale programmazione: per saltare da S_{n-1} a S_n (o eventualmente *ENDPROG*) è necessario avere pari a 1 i segnali *nReset_prog_i*, *progValid_i*, *eq8* e *eq2048*. Il segnale *progValid_i* è un bit in ingresso (un bit di validità) che comunica a *Programming* la validità del dato presente in *progValue_i*; in altre parole quest'ultimo conferma che si ha in ingresso un corretto valore di un peso del filtro o di un neurone. I segnali *eq8* e *eq2048* sono invece segnali internamente prodotti in *Programming* che segnalano (se pari a 1) l'equivalenza tra l'uscita di un contatore *counter8* o *counter2048* e un corrispondente riferimento settato opportunamente a seconda dello stato corrente. Nella tabella Tab.3.2.a. sono riportati i valori di questi riferimenti (chiamati all'interno del componente *compared_data_8* o *_2048*) in base allo stato corrente.

Nome del segnale	Stato corrente (*cs*)	Valore assunto dal segnale
compared_data_8	*S1*	0_{10} ovvero 000_2
compared_data_8	*S2, S3, S4, S4*	7_{10} ovvero 111_2
compared_data_2048	*S1*	9_{10} ovvero 00000001001_2
compared_data_2048	*S2*	2047_{10} ovvero 11111111111_2
compared_data_2048	*S3*	63_{10} ovvero 00000111111_2
compared_data_2048	*S4*	7_{10} ovvero 00000000111_2
compared_data_2048	*S5*	0_{10} ovvero 00000000000_2

Tab.3.2.a.: Valori assunti dai segnali compared_data

Per comprendere il perché di questi valori è necessario studiare più nel dettaglio la struttura della state-machine, soffermandosi sui pesi rilasciati in uscita nei diversi stati. Nell'immagine Imm.3.2.c. è riportato un dettaglio delle uscite di *Programming* in riferimento allo stato corrente.



Imm.3.2.c.: Funzionamento del blocco Programming

Si può notare come a diversi stati corrisponda una diversa quantità di weight (ovvero pesi) con un differente numero di bit:

1. In *S1* vengono caricati e consegnati opportunamente nove pesi da 4 bit utilizzati nel filtraggio (si noti la dimensione di progValue_i pari a 26 bit; tra S1 e S4 i bit del peso di interesse saranno i LSB, ovvero least significant bit);
2. In corrispondenza dello stato *S2* vengono raccolti 64x256 pesi da 4 bit per i neuroni del primo stadio fully-connected, ovvero 256 weight per ogni neurone del primo livello che sono in numero pari a 64;
3. Similmente a *S2*, in *S3* vengono raccolti 8x64 weight da 4 bit per il secondo stato;
4. In *S4* vengono raccolti 64 bias da 24 bit corrispondenti ai 64 neuroni del primo stadio (il bias qui raccolto sarà un offset di partenza per il calcolo del neurone);
5. Similmente a *S4*, in *S5* vengono raccolti 8 bias da 26 bit (si notino le diverse dimensioni dei bias, scelte in base al numero di bit utilizzati all'interno dei calcoli dei neuroni, nel primo stadio pari a 24 e nel secondo pari a 26).

Come mostrato nella precedente immagine Imm.3.2.a., in *Programming* è presente, prima del demultiplexer, un registro di accumulo (*serialParallelRegister*); lo scopo di questo componente è facilitare l'assegnazione dei pesi e bias ai neuroni corrispondenti e controllarne la correttezza. *NetworkOfNeurons*, componente al quale *Programming*

consegna i diversi pesi e offset, presenta infatti in ingresso dodici serie di input, in 4 categorie, che devono essere serviti dal blocco di programmazione:

1. I segnali *neuronWeightValue1*, *neuronWeightValid1* e *endOfWeight1* (che rappresentano la prima categoria: segnali dei pesi del 1° stadio) sono rispettivamente un blocco di otto valori di pesi del primo stadio (8x4 bit), un byte di validità (8 bit) e un byte di fine della programmazione dei pesi del primo livello (8 bit);
2. Similmente per il secondo stadio si hanno *neuronWeightValue2* (8x4 bit), *neuronWeightValid2* (8 bit) e *endOfWeight2* (8 bit);
3. I segnali *biasValue1* (8x24 bit), *biasValid1* (8 bit) e *endOfBias1* (8 bit) corrispondono ai valori di otto bias impacchettati in un blocco da 8x24 bit, otto bit di validità e otto bit di conferma dell'avvenuto caricamento;
4. Similmente si hanno *biasValue2* (8x26 bit), *biasValid2* (8 bit) e *endOfBias2* (8 bit) per il secondo stadio.

Questa architettura di segnali (che preleva un dato alla volta in maniera seriale, ne impacchetta otto in una struttura a molteplicità 8 e consegna quest'ultima in uscita) ha il vantaggio di avere una ridondanza sfruttabile per controllare la correttezza della programmazione; questo controllo ridondante si attua sia a livello dei bit di validità sia attraverso gli *endOf-*:

1. Una corretta programmazione dei neuroni è associata in primis a un set di validità (*-Valid*) completo, ovvero con bit tutti pari a 1; in presenza di un set incompleto si ha un errore che deve essere opportunamente gestito e che viene segnalato dallo stato *ERR*;
2. Il completamento della programmazione senza errori corrisponde alla conclusione del salvataggio dei dati in *NetworkOfNeurons* che attiva il segnale *Start_o* e al rilascio in uscita di un set completo di *endOf-*; la compresenza delle due commutazioni di segnali (*Start_o* e *endOf-* finale) è la prova del corretto caricamento (se *Start_o* commuta a 1 prima che arrivi l'*endOfBias2*, ultimo *endOf-* presente in base agli stati scelti, allora è avvenuto un errore e si salta allo stato *ERR*).

Oltre che per la rilevazione di eventuali errori, questa struttura "ad impacchettamento di dati" viene utilizzata perché permette la consegna in parallelo dei dati di programmazione a tutti gli otto neuroni fisici e perciò semplifica molto l'assegnazione associata e i segnali di controllo annessi.

Volendo sfruttare i vantaggi annessi a questa architettura seriale-parallela, è stato necessario l'utilizzo non di un solo contatore, ma di due diversi contatori indipendenti (*counter8* e *counter2048*); il primo conta quanti valori sono già stati inseriti nel *serialParallelRegister* (ovvero quanti dati sono stati già impacchettati) e si resetta con un valore pari a sette (conta tra zero e sette), il secondo invece conta quanti pacchetti da otto valori sono stati consegnati in uscita. Quest'ultimo contatore si resetta ad opportuni valori in base allo stato corrente (si veda la tabella Tab.3.2.a. per i valori di reset del counter).

Per completare l'analisi di questo componente, si riporta nella tabella Tab.3.2.b. un riassunto dei segnali in ingresso e uscita dal blocco *Programming*, con numero di bit e breve spiegazione dell'utilità del segnale.

Nome del segnale	Ingresso/Uscita	Numero di bit	Utilità del segnale
clk_i	Ingresso	1 bit	Clock di sistema
nReset_prog_i	Ingresso	1 bit	Reset dell'esecuzione della programmazione
progValid_i	Ingresso	1 bit	Validità del dato di programmazione
progValue_i	Ingresso	26 bit	Valore del dato di programmazione
weightValid_o	Uscita	1 bit	Validità del peso del filtraggio
weightValue_o	Uscita	4 bit	Valore del peso del filtraggio
endOfWeights_o	Uscita	1 bit	Segnale di fine della programmazione del filtro
neuronWeightValid1	Uscita	8 bit	Validità dei pesi dei neuroni (1° stadio)
neuronWeightValue1	Uscita	8x4 bit	Valore dei pesi dei neuroni (1° stadio)
endOfWeight1	Uscita	8 bit	Segnale di fine della programmazione dei pesi dei neuroni (1° stadio)
neuronWeightValid2	Uscita	8 bit	Validità dei pesi dei neuroni (2° stadio)
neuronWeightValue2	Uscita	8x4 bit	Valore dei pesi dei neuroni (2° stadio)
endOfWeight2	Uscita	8 bit	Segnale di fine della programmazione dei pesi dei neuroni (2° stadio)
biasValid1	Uscita	8 bit	Validità dei bias dei neuroni (1° stadio)
biasValue1	Uscita	8x24 bit	Valore dei bias dei neuroni (1° stadio)
endOfBias1	Uscita	8 bit	Segnale di fine della programmazione dei bias dei neuroni (1° stadio)
biasValid2	Uscita	8 bit	Validità dei bias dei neuroni (2° stadio)
biasValue2	Uscita	8x26 bit	Valore dei bias dei neuroni (2° stadio)
endOfBias2	Uscita	8 bit	Segnale di fine della programmazione dei bias dei neuroni (2° stadio)

Tab.3.2.b.: Riassunto dei segnali di ingresso e di uscita di Programming

Per concludere la trattazione del blocco *Programming* si riportano alcuni frammenti di codice significativi e alcuni accorgimenti utilizzati:

1. Uso di componenti provenienti dal IP Catalog di Quartus Prime 17.1 (intellectual property (o IP) generator); questo prevede due differenti step:
    a. Dichiarazione del componente all'interno del blocco *architecture* (prima del begin, insieme ai segnali interni):

```
COMPONENT counter8
        PORT
        (       aclr            : IN STD_LOGIC ;
                clock           : IN STD_LOGIC ;
                cnt_en          : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (2 DOWNTO 0) );
END COMPONENT;
```

    b. Mapping del componente con l'uso del *PORT MAP* (a sinistra del simbolo => si hanno i segnali propri del componente dichiarato, a destra invece i segnali interni o I/O di *Programming*):

```
counter_8: counter8 PORT MAP    (
                aclr            => clear_cnt8,
                clock           => clk_i,
                cnt_en          => enable_cnt8,
                q               => cnt8);
```

2. Utilizzo di *process* per la gestione della state-machine:
    a. Il *process Go_To_Next_State* che si occupa del reset dello stato iniziale e del salto da uno stato al successivo:

```
Go_To_Next_State: process(clk_i, nReset_prog_i)
begin
        if clk_i'event and clk_i='1' and nReset_prog_i='0' then
                cs<=S1;                                 --! RESET
        elsif clk_i'event and clk_i='1' then
                cs<=ns;                                 --! JUMP
        end if;
end process;
```

    b. Il *process Management_Flags_Next_State* che rappresenta il vero e proprio cuore della state-machine (qui si riporta una versione ridotta che ne esplica solo la struttura di base):

```
Management_Flags_Next_State: process([sensitivity list])
begin
        if nReset_prog_i='0' then
                ns<= S1; […]                            --! RESET
        else
                case cs is
                when S1=>                               --! STATE S1
                        if eq8='1' and eq2048='1' and progValid_i='1' then
                                ns<=S2; […]             --! NEW_STATE
                        else
```

```
                        ns<=S1; [...]
                    end if;
                    enable_cnt2048 <= progValid_i;   --! ASSIGNMENT
                    enable_cnt8 <= '0'; [...]
                when S2=> [...]
                end case;
            end if;
        end process;
```

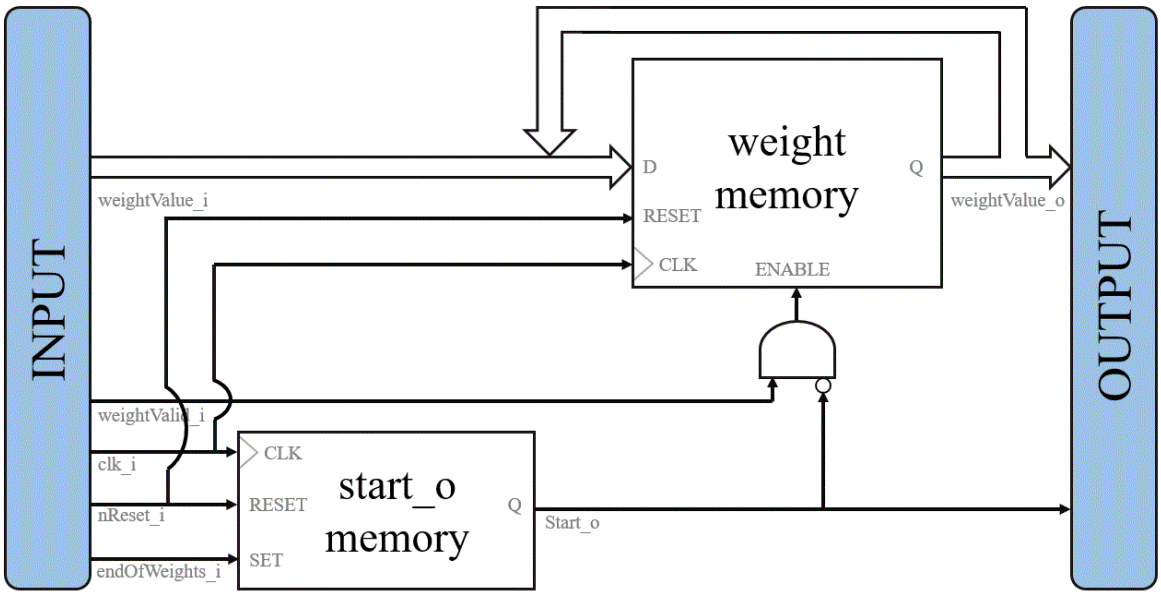
3. Uso di *process* per gestire le uscite nei diversi stati e i *compared_data* per il confronto del conteggio attuale con gli opportuni riferimenti (si veda la tabella Tab.3.2.a.):

```
compared_data8_management: process(cs)
begin
        case cs is
        when S1=>
                compared_data8<="000";
        when others=>
                compared_data8<="111";
        end case;
end process;
```

3.3. *MemoryFilter* – memoria dei pesi del filtro

MemoryFilter è il componente di *ChainOfComponents* che gestisce i pesi del filtraggio. La sua struttura risulta essere piuttosto semplice essendo costituita da solo due memorie e un AND logico opportunamente connessi.

Si riporta nell'immagine Imm.3.3.a. una rappresentazione grafica della struttura di *MemoryFilter*.



Imm.3.3.a.: Struttura del blocco MemoryFilter

Le memorie qui presenti sono:

1. *WeightMemory*, uno shift-register che, in fase di programmazione, viene caricato serialmente con gli opportuni pesi del filtro e che, durante il normale funzionamento del coprocessore, consegna il set di pesi necessario in uscita (lo shift-register qui presente prende in ingresso un dato alla volta e restituisce in uscita il set completo di dati caricati);
2. *Start_oMemory*, un semplice flipflop che presenta in ingresso al *RESET* il segnale *nReset_i* (ovvero con *nReset_i* pari a 0, *Q* si resetta a 0) e in ingresso al *SET* il segnale *endOfWeights_i* (ovvero con *endOfWeights_i* pari a 1, *Q* commuta a 1).

Si noti come *Start_o*, oltre ad essere una delle uscite di *MemoryFilter*, rappresenta anche uno dei segnali di comando dell'*enable* (ovvero attivazione) della memoria *WeightMemory*; questo impedisce di sovrascrivere i pesi correttamente caricati, poiché Start_o viene settato a 1 solo ad avvenuto salvataggio di tutti i pesi del filtro.

Si riporta nella tabella Tab.3.3.a. un riassunto dei segnali in ingresso e uscita dal blocco *MemoryFilter*, con numero di bit e breve spiegazione dell'utilità del segnale.

Nome del segnale	Ingresso/Uscita	Numero di bit	Utilità del segnale
clk_i	Ingresso	1 bit	Clock di sistema
nReset_i	Ingresso	1 bit	Reset dell'esecuzione dell'elaborazione
weightValid_i	Ingresso	1 bit	Validità del peso in ingresso
weightValue_i	Ingresso	4 bit	Valore del peso in ingresso
endOfWeights_i	Ingresso	1 bit	Segnale in corrispondenza dell'ultimo peso in ingresso
weightValue_o	Uscita	441 bit	Valore del peso del filtraggio
Start_o	Uscita	1 bit	Segnale di avvenuta programmazione del filtro

Tab.3.3.a.: Riassunto dei segnali di ingresso e di uscita di MemoryFilter

Si noti il numero di bit in uscita dal componente *MemoryFilter* pari a 442; questo numero è stato scelto considerando il caso peggiore di maschera di filtraggio, ovvero la maschera 7x7 (per ipotesi *ChainOfComponent* deve essere in grado di gestire tutte le combinazioni di 3, 5 e 7, come verrà poi spiegato nel capitolo 3.5.), la quale può contenere coefficienti estremamente grandi, esprimibili in complemento a due solo con 9 bit (441 bit = 7 x 7 pesi x 9 bit).

L'implementazione in linguaggio VHDL di questo componente è estremamente semplice e presenta solo tre parti fondamentali:

1. Mapping del componente *shift_register_reverse* (*generic map* e *port map*):

```
shift_register_weights: entity shift_register_reverse generic map(
        KERNEL_WIDTH => KERNEL_WIDTH*KERNEL_HEIGHT,
        DATA_WIDTH => WEIGHT_WIDTH)
        port map(
        clk_i => clk_i,
        writeEnable_i=> writeEnable,
        Reset_i => Reset_c,
        dataIn_i => weightValue_i,
        dataOut_o  =>  weightsValue_c(KERNEL_HEIGHT*KERNEL_WIDTH*WEIGHT_WIDTH-1
downto 0));
```

2. Processo di gestione del segnale *Start_o* e *Start_c* (si noti che i due *Start* sono uguali tra loro; poiché il valore di *Start_o* deve essere utilizzato all'interno del circuito, è necessario definire il segnale interno *Start_c*):

```
Start_o_management: process(nReset_i, clk_i, endOfWeights_i)
begin
        if nReset_i='0' then
                Start_o<='0';
                Start_c<='0';
        elsif clk_i'event and clk_i='1' and endOfWeights_i='1' then
                Start_o<='1';
                Start_c<='1';
        end if;
end process;
```

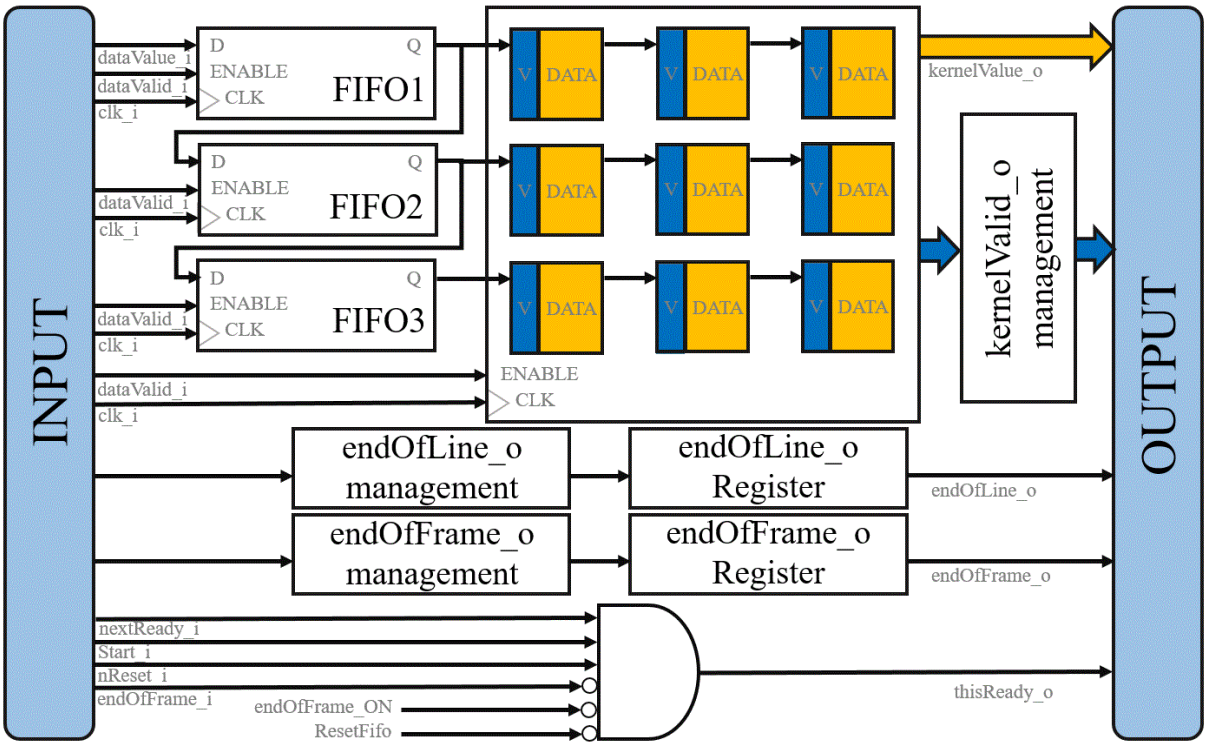
3. Assignment dei segnali in ingresso e uscita dall'*entity shift_register_reverse*, ad esempio: `writeEnable<= weightValid_i and not(Start_c);`, assegnazione ben riconoscibile anche nell'immagine Imm.3.3.a.

3.4. *Kernel* – primo stadio del filtro convolutivo

Kernel è un componente fondamentale di *ChainOfComponents* che permette la gestione dell'immagine quale dato bidimensionale e la selezione dell'opportuna maschera di filtraggio sui dati (chiamata anche kernel). Presenta una struttura alquanto complessa con diversi elementi costitutivi di base:

1. Un sistema di memorizzazione costituito da *FIFO* (first in first out) e registri a scorrimento (il numero di livelli di memorizzazione *FIFO* + *shift-register* è pari all'altezza della maschera di kernel *KERNEL_HEIGHT*);
2. Due componenti di gestione degli *endOf-*;
3. Una semplice logica di management del *thisReady_o*, segnale che comunica al componente a monte la disponibilità di *ChainOfComponents* di acquisire nuovi dati.

Nell'immagine Imm.3.4.a. è riportato lo schema della struttura di *Kernel*, con un particolare dettaglio sulla struttura di memorizzazione del dato.



Imm.3.4.a.: Struttura del blocco Kernel

Nell'immagine soprariportata si possono osservare i tre componenti principali di *Kernel* ed in particolare i collegamenti tra le memorie *FIFO* e *shift-register*. Si noti la struttura della catena di memorizzazione, in cui le diverse *FIFO* sono connesse l'una all'altra a costituire un unico grande registro a scorrimento, che accoglie i dati dell'immagine in ingresso e ne riproduce la struttura bidimensionale. Avendo scelto una maschera di memorizzazione 3x3, sono presenti, nella struttura complessiva di *Kernel*, tre livelli di memorizzazione (*FIFO* + *shift-register*) e, all'interno degli shift-register, tre blocchi di memoria (in figura riconoscibili come rettangoli blu e gialli) costituiti da nove bit (un bit di validità, in figura in blu, e otto bit di dato, in giallo). In questa architettura di memoria scorrono i diversi pixel dell'immagine, provenienti dall'ingresso *dataValue_i*. L'avanzamento nella catena di memoria è legato al segnale di *enable dataValid_i* (modulato con una opportuna logica a seconda del livello di memorizzazione considerato) che rappresenta la validità del pixel; questo segnale non solo funge da segnale principale dell'*enable* della memoria di *Kernel*, ma costituisce anche una parte fondamentale del dato in memoria (componente blu nell'immagine Imm.3.4.a.), in

quanto la posizione dei diversi bit di validità all'interno degli *shift-register* permette la gestione del kernel di uscita e della validità dei dati al suo interno.

Questa scelta di accompagnare ogni dato col proprio bit di validità permette quattro fondamentali azioni:

1. Permette di conoscere in ogni istante temporale la posizione della maschera di filtraggio sull'immagine in ingresso;
2. Permette una gestione semplificata e ottimizzata della struttura di memorizzazione con attivazione di *enable* specifici legati alle validità;
3. Permette una politica di zero-padding (ovvero il considerare i pixel ignoti del contorno tutti settati a 0) per il mantenimento delle dimensioni dell'immagine in ingresso;
4. Permette una costruzione semplice e intuitiva dei segnali *kernelValue_o* e *kernelValid_o*.

La gestione dello zero-padding sopracitata è associata a una semplice logica a selezione: ad ogni possibile combinazione delle validità, unita ad un valore memorizzato di *endOfLine_i,* è associata un determinato *kernelValid_o*. Si deve notare come, in seguito a un reset dell'elaborazione o in seguito a una conclusione dell'elaborazione di una precedente immagine, tutte le validità presenti negli *shift-register* siano settate a 0; non appena l'immagine comincia ad essere caricata si ha una sequenza ininterrotta di validità attive (pari a 1) che avanza nella struttura di memorizzazione e, a conclusione del caricamento, si ha al contrario una sequenza ininterrotta di validità pari a 0. Di conseguenza le possibili combinazioni di attivazioni di validità all'interno della struttura non sono pari a 2^9 ma molte di meno. Considerando anche le commutazioni di *endOfLine_c* (*endOfLine_i* ritardato opportunamente ad ogni clock) si ottengono le seguenti possibili combinazioni di segnali associati al corrispondente *kernelValid_o* (si riportano le combinazioni di segnali nella tabella Tab.3.4.a.; le validità vengono ivi riportate in un'unica riga riassuntiva costituita dalle tre righe di bit di validità poste in sequenza l'una dopo l'altra).

Si deve notare come l'uso degli *endOfLine_c* permette di associare a una matrice di validità completamente settata a 1 l'opportuno *Valid* di uscita; senza questo accorgimento, la gestione a zero padding dei bordi laterali destro e sinistro non sarebbe

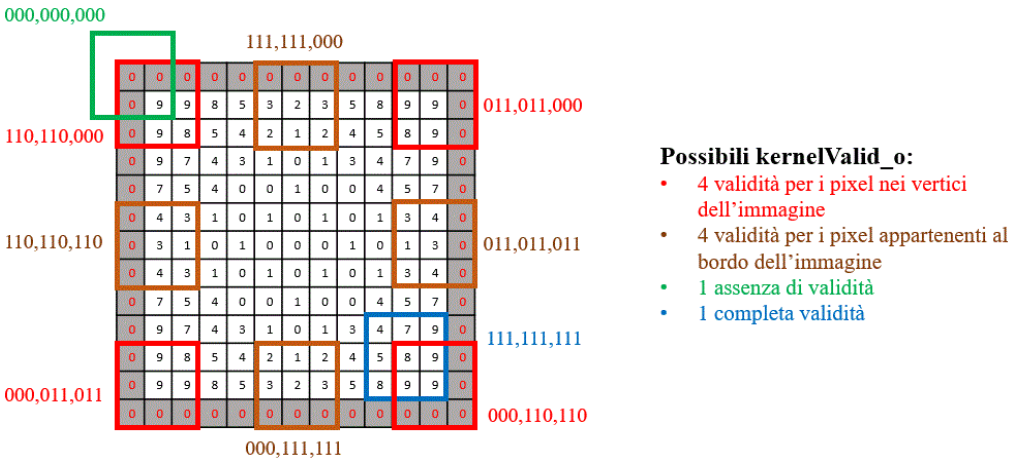
possibile. Si noti inoltre il significato dei segnali presenti nella tabella Tab.3.4.a.: *shift-registerValid* corrisponde alle validità in memoria (nella struttura 3x3 degli *shift-register* di *Kernel*) ed *endOfLine_c(3)* e *(4)* corrispondono al segnale endOfLine_i ritardato di tre o quattro cicli di clock.

shiftRegisterValid	endOfLine(3)	endOfLine(4)	kernelValid_o	Significato kernelValid_o
---,-0-,--- (ogni combinazione con 0 nel bit centrale)	- (qualunque)	-	000,000,000	Nessuna validità: maschera non centrata in un pixel dell'immagine
111,110,000	-	-	110,110,000	Maschera centrata sul primo pixel in alto a sinistra (esempio di kernelValue_o dall'immagine Imm.3.4.b. = $899,990,000_d$)
111,111,000	0	0	111,111,000	Maschera centrata in un pixel della prima riga (esempio: $458,589,000_d$)
111,111,000	1	0	011,011,000	Maschera centrata nell'ultimo pixel della prima riga (es: $998,999,900_d$)
111,111,11-	-	1	110,110,110	Maschera centrata in un pixel della prima colonna (es: $579,799,899_d$)
111,111,11-	0	0	111,111,111	Maschera centrata in un pixel completamente interno all'immagine (es: $540,743,854_d$)
-11,111,111	1	0	011,011,011	Maschera centrata in un pixel dell'ultima colonna (es: $775,797,998_d$)
001,111,111	-	-	000,110,110	Maschera centrata nel primo pixel dell'ultima riga (es: $009,999,899_d$)
000,111,111	-	-	000,111,111	Maschera centrata in un pixel dell'ultima riga (es: $000,558,544_d$)
000,011,111	-	-	000,011,011	Maschera centrata nell'ultimo pixel dell'immagine (es: $000,099,998_d$)

Tab.3.4.a.: Combinazioni di validità e endOfLine_c con relativi kernelValid_o (maschera 3x3)

Come osservabile dalla tabella Tab.3.4.a., esistono dieci possibili *kernelValid_o*:

1. Quattro *kernelValid_o* corrispondenti ai pixel ai vertici;
2. Quattro *kernelValid_o* corrispondenti ai bordi superiore, inferiore, destro e sinistro;
3. Un *kernelValid_o* di assenza di validità (completamente settato a 0);
4. Un *kernelValid_o* di completa validità (completamente settato a 1).



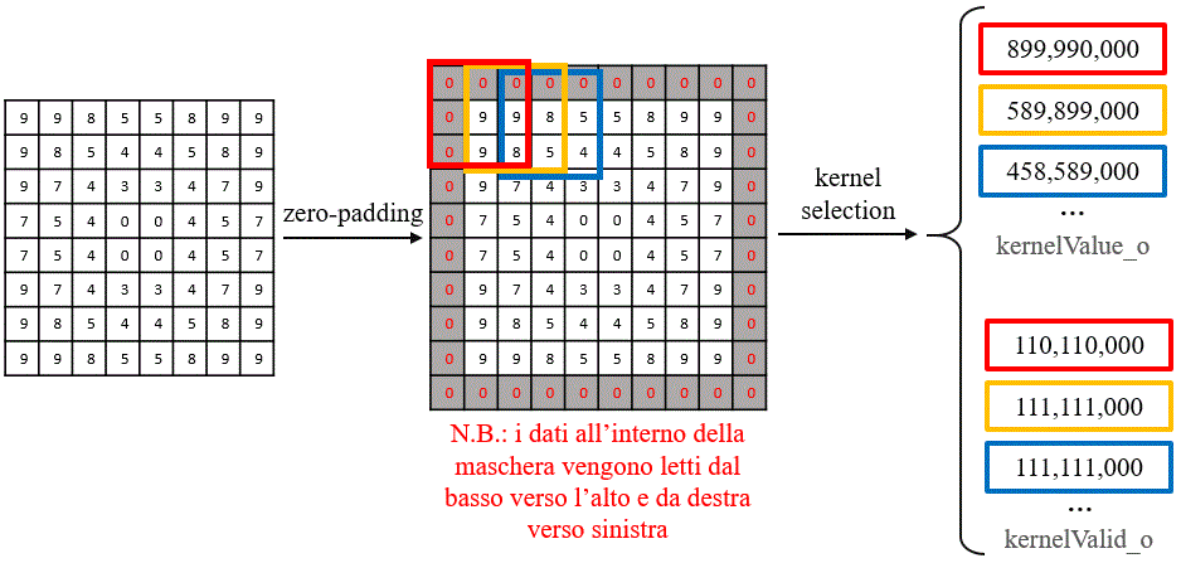
Imm.3.4.b.: Esempi delle diverse tipologie di kernelValid_o

Nell'immagine Imm.3.4.a. si possono osservare, oltre al blocco di memoria, tre componenti che gestiscono altrettanti segnali di uscita (*endOfLine_o*, *endOfFrame_o* e *thisReady_o*). La gestione degli *endOf-* è a carico di opportuni management che, attraverso specifici ritardi dei segnali *endOf-_i* corrispondenti, portano in uscita una attivazione del segnale, ovvero una commutazione a 1, in corrispondenza dei *kernelValid_o* centrati nei pixel appartenenti all'ultima colonna (nel caso di *endOfLine_o*) o in corrispondenza dell'ultimo kernel dell'immagine (nel caso di *endOfFrame_o*). La gestione invece del segnale *thisReady_o* (segnale che non rappresenta solo un'uscita del blocco *Kernel* ma anche un importante output di *ChainOfComponents*) è affidata a un blocco logico piuttosto semplice, un *AND* a sei ingressi; questo significa che, per aver *thisReady_o* attivo e settato a 1, è necessaria una particolare combinazione degli ingressi. Di seguito è riportata una tabella esplicativa degli ingressi dell'*AND* logico del *thisReady_o* e del loro corrispondente significato di attivazione.

Nome del segnale	Ingresso/segnale interno	Valore assunto per attivare *thisReady_o*	Significato del segnale
nextReady_i	Ingresso	1	Il componente a valle è pronto a ricevere il dato *resultValid_o*
Start_i	Ingresso	1	La programmazione si è conclusa e l'elaborazione può aver luogo
nReset_i (attivobasso)	Ingresso	1	Non si ha un reset dell'esecuzione
endOfFrame_i	Ingresso	0	Non si sta ricevendo l'ultimo pixel dell'immagine
endOfFrame_ON	Interno	0	Non si sta elaborando l'ultima riga dell'immagine
ResetFifo	Interno	0	Non si stanno resettando le FIFO in attesa di una nuova immagine

Tab.3.4.b.: Segnali di controllo dell'uscita thisReady_o

Il funzionamento di *Kernel* è riportato nell'immagine Imm.3.4.c.; questo componente ha lo scopo di selezionare i pixel dell'immagine appartenenti a opportune maschere di filtraggio e di collocarli in vettori chiamati *KernelValue_o,* che verranno poi forniti in uscita. La selezione dei pixel è possibile grazie alla complessiva memoria a scorrimento composta dalle *FIFO* e relativi *shift-register,* in quanto i pixel appartenenti alle diverse maschere di filtraggio si ritroveranno raccolti e ordinati nella struttura di *shift-register* a valle delle *FIFO*. È necessario notare come i dati vengono letti dalla memoria: entrando nella prima *FIFO* in alto e scorrendo da sinistra verso destra e dall'alto verso il basso, i dati risulteranno (rispetto alla posizione nell'immagine) doppiamente specchiati in *kernelValue_o*, ovvero risulteranno letti da destra verso sinistra e dal basso verso l'alto. Considerando, ad esempio, la maschera in rosso sotto riportata nell'immagine Imm.3.4.c., il *kernelValue_o* ipotetico ottenuto in uscita risulta essere pari a *890,990,000* (leggendo il contenuto della maschera da destra verso sinistra e dal basso verso l'alto). Si può però notare che l'uscita corrispondente presenta una discrepanza col valore teorico, essendo pari a *899,990,000*; questo è dovuto al fatto che gli zeri dello zero-padding non sono zeri fisicamente letti dal lettore e trasmessi a *Kernel*, ma sono virtuali e creati da *Kernel* stesso. Il valore *9* presente in terza posizione altro non è che l'ultimo pixel della prima riga, ovvero il valore letto immediatamente prima del pixel in seconda posizione. La correzione di questi errori, dovuti allo scorrimento dei dati nella memoria, è affidata al vettore *kernelValid_o* il quale, con il corrispondente valore pari a *0* in terza posizione, annulla l'errato valore di *kernelValue_o*.



Imm.3.4.c.: Funzionamento del blocco Kernel

Nella precedente immagine si possono vedere le operazioni fondamentali eseguite dal blocco di elaborazione *Kernel*:

1. Acquisizione dei pixel/dati dell'immagine;
2. Gestione con zero-padding dei bordi (aggiunta di zeri virtuali);
3. Scorrimento della maschera di filtraggio (in questo caso 3x3) da sinistra a destra con passo di un pixel (una volta raggiunta la fine di una riga, lo scorrimento della maschera ricomincia dal primo pixel della riga successiva);
4. Rilascio degli opportuni segnali di uscita.

Per completare lo studio del funzionamento di *Kernel*, si riportano nella tabella Tab.3.4.c. i suoi segnali di ingresso e uscita.

Nome del segnale	Ingresso/Uscita	Numero di bit	Utilità del segnale
clk_i	Ingresso	1 bit	Clock di sistema
nReset_i	Ingresso	1 bit	Reset dell'esecuzione dell'elaborazione
Start_i	Ingresso	1 bit	Segnale di avvenuta programmazione (enable dell'esecuzione)
dataValid_i	Ingresso	1 bit	Validità del dato in ingresso
dataValue_i	Ingresso	8 bit	Valore del dato in ingresso
endOfLine_i	Ingresso	1 bit	Segnale di fine riga dell'immagine (corrispondente all'ultimo pixel della riga)
endOfFrame_i	Ingresso	1 bit	Segnale di fine immagine (corrispondente all'ultimo pixel dell'immagine)
nextReady_i	Ingresso	1 bit	Segnale che il blocco a valle è pronto a ricevere il dato
kernelValid_o	Uscita	1 bit	Validità del kernel in uscita
kernelValue_o	Uscita	72 bit	Valore del kernel in uscita
endOfLine_o	Uscita	1 bit	Segnale di fine riga (corrispondente all'ultimo kernel della riga)
endOfFrame_o	Uscita	1 bit	Segnale di fine immagine (corrispondente all'ultimo kernel consegnato in uscita)
thisReady_o	Uscita	1 bit	Segnale che il blocco *Kernel* è pronto a ricevere nuovi dati dal blocco a monte

Tab.3.4.c.: Riassunto dei segnali di ingresso e di uscita di Kernel

Si riportano, anche per questo componente, dei frammenti di codice e spiegazioni delle più importanti scelte progettuali; in *Kernel* si ha l'uso delle seguenti funzioni:

1. *Generate*, per permettere un piazzamento dei componenti parametrico, ovvero variabile con *KERNEL_HEIGHT*, poiché il numero di *FIFO* e *shift-register* da generare è pari all'altezza della maschera di filtraggio; qui il numero di *FIFO* collocate è pari a *KERNEL_HEIGHT* – 1 (partendo *L* da 2), questo perché esiste un piazzamento a parte per la prima *FIFO*:

```
fifo_2_KH_generate: for L in 2 to KERNEL_HEIGHT generate
        fifo_L : FifoMemoria PORT MAP (
```

```
                aclr        => Reset_inter,
                clock       => clk_i,
                data        => [...]);
end generate fifo_2_KH_generate;
```

2. *Entity*, per piazzare componenti scritti in altri file di progettazione (dichiarati precedentemente con *use work.component;*):

```
shift_register_endOfLine_ON: entity shift_register generic map(
        KERNEL_WIDTH => KERNEL_WIDTH-2,
        DATA_WIDTH => 1)
        port map(
        clk_i => clk_i,
        writeEnable_i=> [...]);
```

3. *Process*, per gestire processi sincroni e non (ad esempio, i management sopracitati); si riporta il management del segnale *endOfFrame_ON*:

```
endOfFrame_ON_management: process(Reset_inter, clk_i, endOfFrame_i, nextReady_i)
begin
        if Reset_inter='1' then
                endOfFrame_ON<='0';
        elsif clk_i'event and clk_i='1' and endOfFrame_i='1' and nextReady_i='1' then
                endOfFrame_ON<='1';
        end if;
end process;
```

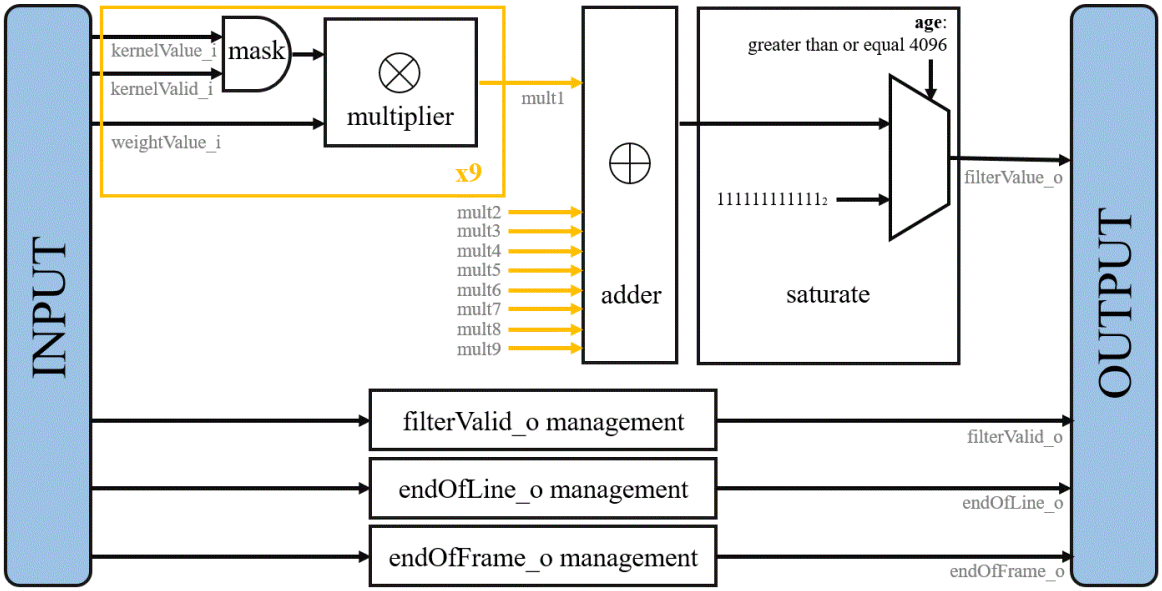
3.5. *Filter* – secondo stadio del filtro convolutivo e ReLU

Il blocco *Filter* è il componente immediatamente a valle di *Kernel*, che, con quest'ultimo, costituisce nel complesso il filtro convolutivo (layer convolutivo CONV della CNN) di *ChainOfComponents*.

Il file Filter.vhd in realtà comprende (per comodità di implementazione) due componenti logici di base del coprocessore neurale: il calcolo del filtro convolutivo e la successiva *ReLU*, ovvero il rettificatore *REctified Linear Unit*, che apporta la non linearità tipica delle reti neurali.

Qui di seguito viene trattato perciò prima il componente *Filter* (senza l'operazione di non linearità) e poi la *ReLU*.

La struttura base di *Filter* è riportata nell'immagine Imm.3.5.a.; si possono notare i quattro componenti alla base del suo funzionamento, ovvero tre semplici management e un blocco di calcolo costituito da moltiplicatori, sommatore e saturatore.



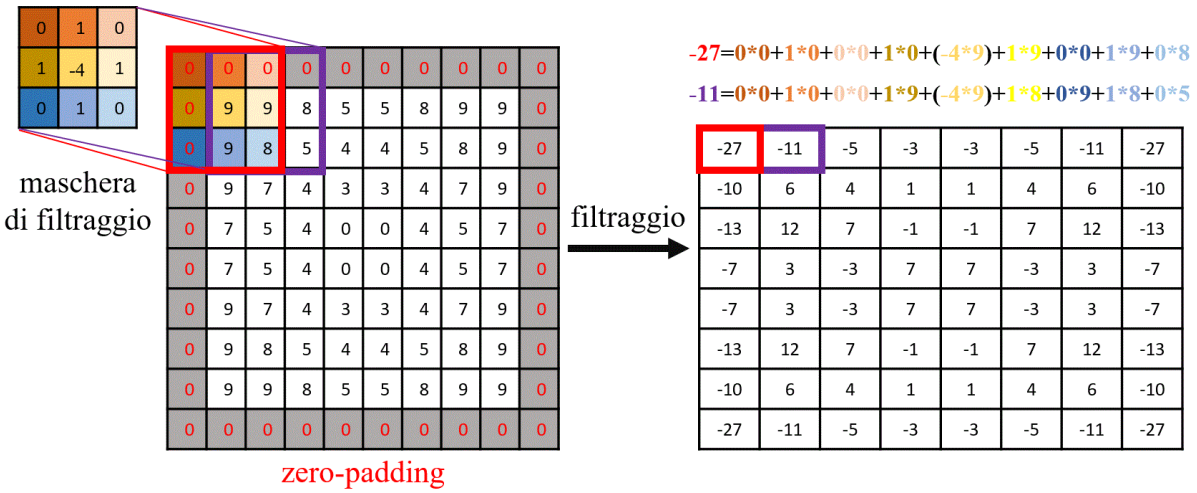
Imm.3.5.a.: Struttura del blocco Filter

I tre blocchi di management gestiscono la validità dell'uscita di *Filter*, la fine della riga e la fine dell'immagine (*filterValid_o*, *endOfLine_o* e *endOfFrame_o*), semplicemente ritardando opportunamente il quinto bit di *kernelValid_i* (che rappresenta la validità generale del kernel consegnatogli in ingresso), *endOfLine_i* ed *endOfFrame_i*.

Il cuore però di *Filter* è rappresentato dal blocco di calcolo; questo è costituito da tre tipi di blocchi aritmetici o logici fondamentali:

1. Un blocco di mascheramento e moltiplicazione, che permette di mascherare, ovvero annullare, gli eventuali errori della gestione dello zero-padding e di compiere la moltiplicazione del dato così corretto con l'opportuno peso del filtro consegnatogli da *MemoryFilter* (questo blocco è piazzato un numero di volte pari al numero dei dati della maschera di filtraggio, in questo esempio pari a 9);
2. Un *adder* a 9 porte (dove naturalmente il numero di ingressi è pari al numero di moltiplicazioni eseguite), che somma tutti i risultati precedentemente prodotti dai blocchi *mask+multiply*;
3. Un blocco di saturazione (*saturate*) che permette un'intelligente riduzione della dimensione del dato in uscita: questo componente confronta il risultato della somma col massimo numero esprimibile col numero di bit dell'uscita e, in caso in cui la somma risulti maggiore del massimo esprimibile, setta l'uscita pari al massimo.

L’immagine Imm.3.5.b. riporta un riassunto del funzionamento del blocco *Filter*; è qui possibile osservare l’esplicazione delle operazioni compiute dal blocco di calcolo.



Imm.3.5.b.: Funzionamento del blocco Filter

Prendendo a riferimento la prima maschera in rosso nell’immagine, le operazioni compiute risultano essere (considerando il mascheramento già avvenuto e moltiplicando peso per valore del pixel): 0*0 + 1*0 + 0*0 + 1*0 + (-4)*9 + 1*9 + 0*0 + 1*9 + 0*8 con risultato -27. Questa operazione di mascheramento, moltiplicazioni e somma conclusiva si ripete per ogni maschera e produce perciò in uscita una nuova immagine con le stesse dimensioni di quella di partenza, in cui ogni nuovo dato è prodotto da una operazione di convoluzione compiuta grazie a una maschera centrata nel pixel corrispondente.

La maschera usata nell’esempio soprariportata è una maschera 3x3 con valori pari a 010,1-41,010; questa maschera permette l’individuazione dei bordi o contorni dell’immagine in esame. Esistono naturalmente diverse maschere per diversi scopi; nell’immagine Imm.3.5.c. si riportano esempi di maschere di dimensioni diverse con valori settati per scopi differenti. Si deve inoltre notare che il blocco *Filter* qui implementato è programmato per esser in grado di gestire maschere *KERNEL_WIDTH*KERNEL_HEIGHT* con valori di *WIDTH* e *HEIGHT* pari a numeri dispari di piccole dimensioni (3, 5 e 7), ovvero per gestire ogni possibile combinazione di questi numeri (in totale si ottengono nove casi).

maschera 3x3

0	1	0
1	-4	1
0	1	0

Individuazione bordi/contorni

maschera 3x5

-1	-1	-1	-1	-1
0	0	0	0	0
1	1	1	1	1

Gradient Prewitt esteso

maschera 3x7

-1	-1	-2	-4	-2	-1	-1
0	0	0	0	0	0	0
1	1	2	4	2	1	1

Gradient Sobel esteso

maschera 5x3

-1	0	1
-1	0	1
-1	0	1
-1	0	1
-1	0	1

Gradient Prewitt esteso (trasposto della maschera 3x5)

maschera 5x5

2	1	0	-1	-2
2	1	0	-1	-2
4	2	0	-2	-4
2	1	0	-1	-2
2	1	0	-1	-2

Gradient Y-direction

maschera 5x7

0	0	0	0	0	0	0
0	0	0	-1	0	0	0
0	0	-1	5	-1	0	0
0	0	0	-1	0	0	0
0	0	0	0	0	0	0

Aumento del contrasto (high-pass)

maschera 7x3

-1	0	1
-1	0	1
-2	0	2
-4	0	4
-2	0	2
-1	0	1
-1	0	1

Gradient Sobel esteso (trasposto della maschera 3x7)

maschera 7x5

0	0	0	0	0
0	0	0	0	0
0	0	-1	0	0
0	-1	5	-1	0
0	0	-1	0	0
0	0	0	0	0
0	0	0	0	0

Aumento del contrasto (trasposto della maschera 5x7)

maschera 7x7

2	4	5	5	5	4	2
4	5	3	0	3	5	4
5	3	-12	-24	-12	3	5
5	0	-24	-40	-24	0	5
5	3	-12	-24	-12	3	5
4	5	3	0	3	5	4
2	4	5	5	5	4	2

Laplaciano di gaussiana

Imm.3.5.c.: Esempi di diverse maschere di filtraggio

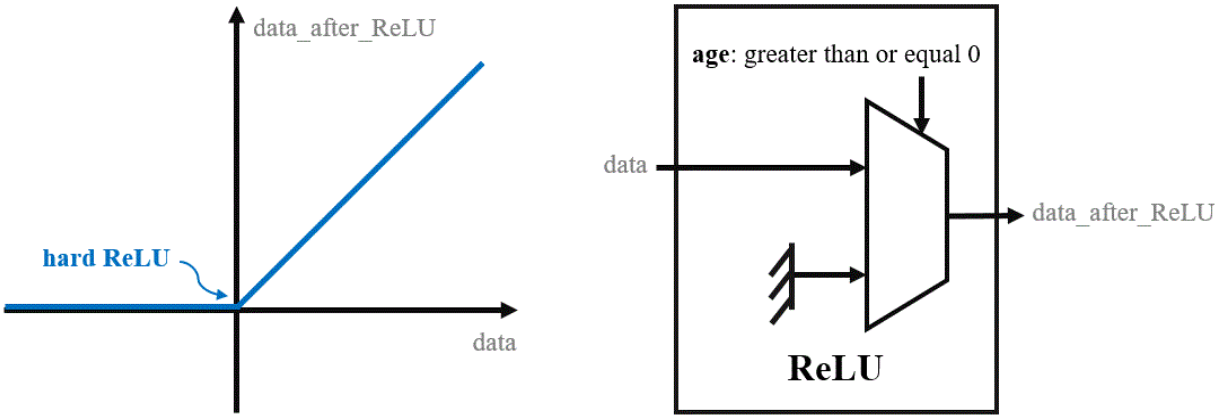
Dall'immagine soprariportata si possono osservare diverse tipologie di maschere di filtraggio che presentano differenti scopi (seguendo l'ordine delle maschere):

1. Individuazione di bordi o contorni nelle direzioni orizzontale e verticale;
2. Operatore Prewitt, utilizzato nel processing delle immagini per l'edge detection, ovvero l'individuazione dei bordi (questo è un operatore differenziale che approssima il gradiente in direzione verticale, la presenza nella matrice di più differenze finite in senso orizzontale è associata ad un operatore passabasso di media che corregge eventuali errori ad alta frequenza);
3. Operatore Sobel, anch'esso operatore di edge detection, che presenta però coefficienti differenti;
4. Gradient Y-direction (o X-direction), altro esempio di maschera a gradiente che agisce in una opportuna direzione; qui i coefficienti sono distribuiti differentemente rispetto ai due casi precedenti in quanto il calcolo del gradiente può sfruttare un numero maggiore di dati (essendo una maschera 5x5);
5. Aumento del contrasto locale;
6. Operatore laplaciano di gaussiana, costituito dalla composizione di due operatori, un gaussiano e un operatore di Laplace, dove il primo consiste nel convolvere l'immagine con una risposta all'impulso consistente in una campana di Gauss

normalizzata (costituisce in pratica un lowpass modificato) e il secondo invece permette di evidenziare i bordi e zero-crossing.

Esistono naturalmente altre tipologie di maschere, ad esempio una maschera completamente settata in ogni suo valore a 1 rappresenta una maschera passabasso (o di media). Grazie a questa varietà di filtraggi, perciò, in base al particolare obiettivo da soddisfare, è possibile selezionare ed utilizzare la maschera più opportuna.

Come precedentemente introdotto, il componente *Filter* presenta al suo interno anche una non linearità *ReLU*, di cui si riportano schema e grafico nell'immagine Imm.3.5.d.



Imm.3.5.d.: Funzione ReLU – schema e grafico

Questo operatore ha lo scopo di annullare eventuali risultati negativi; la caratteristica ingresso-uscita si presenta con un comportamento a rampa (nel caso di una non linearità hard; esistono anche rettificatori soft, più complessi da implementare e con prestazioni non sufficientemente superiori al caso qui in esame, si veda l'immagine Imm.1.1.e.): i valori negativi vengono annullati, mentre quelli positivi vengono lasciati invariati. Il componente corrispondente alla *ReLU* è un semplice multiplexer poiché, attraverso il confronto del MSB (most significan bit) con lo 0 (si ricorda che i numeri negativi in complemento a 2 presentano nel MSB il valore 1), si è in grado di discriminare numeri positivi da quelli negativi e settare a 0 quest'ultimi.

Per completare l'analisi del componente *Filter* si riportano nella tabella Tab.3.5.a. gli ingressi e uscite del blocco con esplicazioni sul numero di bit e relativi significati.

Nome del segnale	Ingresso/Uscita	Numero di bit	Utilità del segnale
clk_i	Ingresso	1 bit	Clock di sistema
nReset_i	Ingresso	1 bit	Reset dell'esecuzione dell'elaborazione
kernelValid_i	Ingresso	1 bit	Validità del kernel in ingresso
kernelValue_i	Ingresso	72 bit	Valore del kernel in ingresso
endOfLine_i	Ingresso	1 bit	Segnale di fine riga dell'immagine (corrispondente all'ultimo kernel della riga)
endOfFrame_i	Ingresso	1 bit	Segnale di fine immagine (corrispondente all'ultimo kernel dell'immagine)
weightsValue_i	Ingresso	441 bit	Valore del peso del filtro
filterValid_o	Uscita	1 bit	Validità dell'uscita del filtro
kernelValue_o	Uscita	12 bit	Valore dell'uscita del filtro
endOfLine_o	Uscita	1 bit	Segnale di fine riga (corrispondente all'ultimo dato della riga)
endOfFrame_o	Uscita	1 bit	Segnale di fine immagine (corrispondente all'ultimo dato consegnato in uscita)

Tab.3.5.a.: Riassunto dei segnali di ingresso e di uscita di Filter

In ultimo si riporta per *Filter* un unico frammento di codice relativo alla gestione delle maschere 3x3 (questo rappresenta uno dei sei *generate* del file); al suo interno si possono notare i diversi componenti del blocco di calcolo e la ReLU. Si segmenta il codice per evidenziare le diverse parti costitutive e i blocchi di calcolo:

1. Dichiarazioni dei segnali interni e componenti del generate:

```
generate3x3: if KERNEL_HEIGHT=3 and KERNEL_WIDTH=3 generate
signal peso3x3: std_logic_vector(35 downto 0);                    --! INTERNAL SIGNAL
signal result3x3: std_logic_vector(116 downto 0);
signal out3x3: std_logic_vector(16 downto 0);
COMPONENT mult3x3                                                 --! COMPONENTS
        PORT(   dataa           : IN STD_LOGIC_VECTOR (8 DOWNTO 0);
                datab           : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
                result          : OUT STD_LOGIC_VECTOR (12 DOWNTO 0));
END COMPONENT;
COMPONENT adder3x3
        PORT(   data0x          : IN STD_LOGIC_VECTOR (12 DOWNTO 0);
                […]
                data8x          : IN STD_LOGIC_VECTOR (12 DOWNTO 0);
                result          : OUT STD_LOGIC_VECTOR (16 DOWNTO 0));
END COMPONENT;
begin
```

2. Moltiplicatori gestiti con un *generate for* (con I da 1 a 9):

```
peso3x3<=weightsValue_i(35 downto 0);
        mult3x3_generate: for I in 1 to KERNEL_WIDTH*KERNEL_HEIGHT generate
                mult_3x3 : mult3x3 PORT MAP (
                        dataa => data_a((DATA_WIDTH+1)*I-1 downto (DATA_WIDTH+1)*(I-1)),
                        datab => peso3x3(4*I-1 downto 4*(I-1)),
                        result => result3x3(13*I-1 downto 13*(I-1)));
        end generate mult3x3_generate;
```

3. Piazzamento e mapping del sommatore:

```
adder_3x3:adder3x3 PORT MAP (
                        data0x => result3x3(12 downto 0),
                        […]
```

```
                    data8x => result3x3(116 downto 104),
                    result =>  out3x3);
```

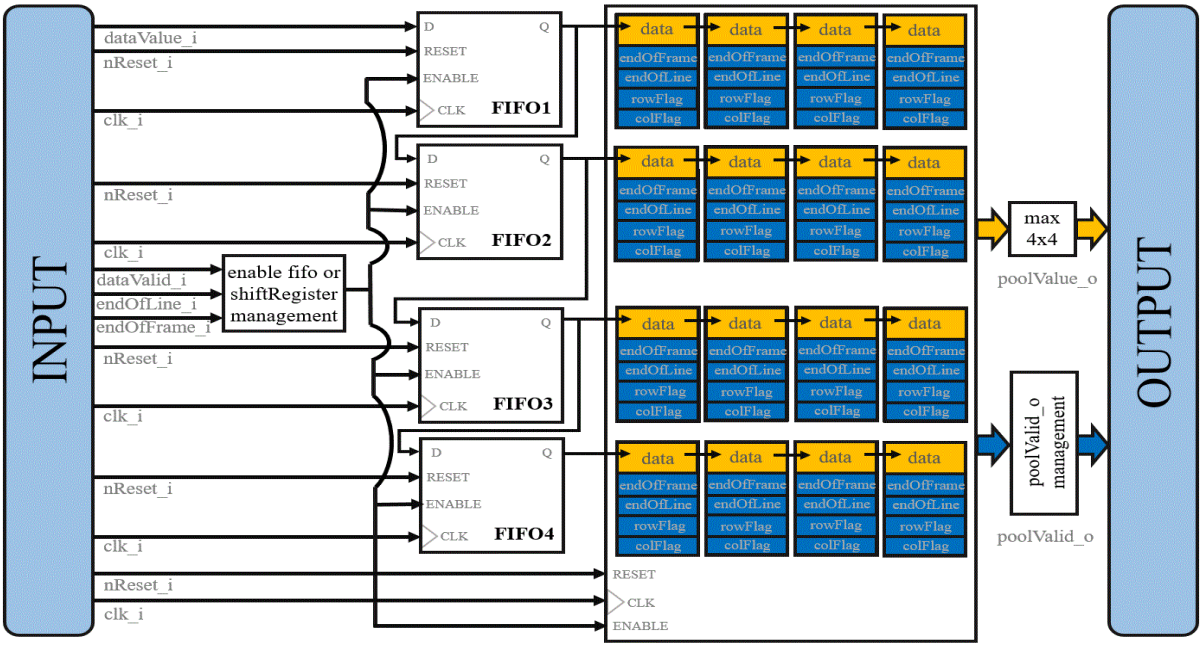
4. *ReLU* gestita da un semplice *process* di selezione e assegnazione:

```
ReLU3x3: process(out3x3)
begin
        if out3x3(16)='1' then
                filterValue_c<=(others=>'0');
        else
                filterValue_c<="0000000" & out3x3;
        end if;
end process;
end generate generate3x3;
```

3.6. *MaxPooling* – stadio di max-pooling

Il componente *MaxPooling*, che rappresenta il blocco a valle di *Filter*, compie una prima forte riduzione della dimensione dell'immagine, andando a decimare i dati in essa contenuti (nell'ipotesi di aver un *POOL_DIM* pari a 4, ovvero una maschera di selezione 4x4, il numero di dati in uscita dal *MaxPooling* sarà un sedicesimo del numero originario in ingresso). Il funzionamento di *MaxPooling*, come verrà esplicato meglio in seguito, si basa sulla ricerca di un massimo all'interno di opportune maschere di ricerca, che scorrono, senza overlap, sull'immagine.

Lo schema di questo blocco è visibile nell'immagine Imm.3.6.a. Si può subito notare una certa somiglianza tra *MaxPooling* e il blocco di memoria di *Kernel*, presentando *FIFO* connesse a catena e *shift-register* di uscita; questa similitudine è dovuta all'effettiva implementazione analoga dei due componenti.



Imm.3.6.a.: Struttura del blocco MaxPooling

I componenti fondamentali di *MaxPooling* sono:

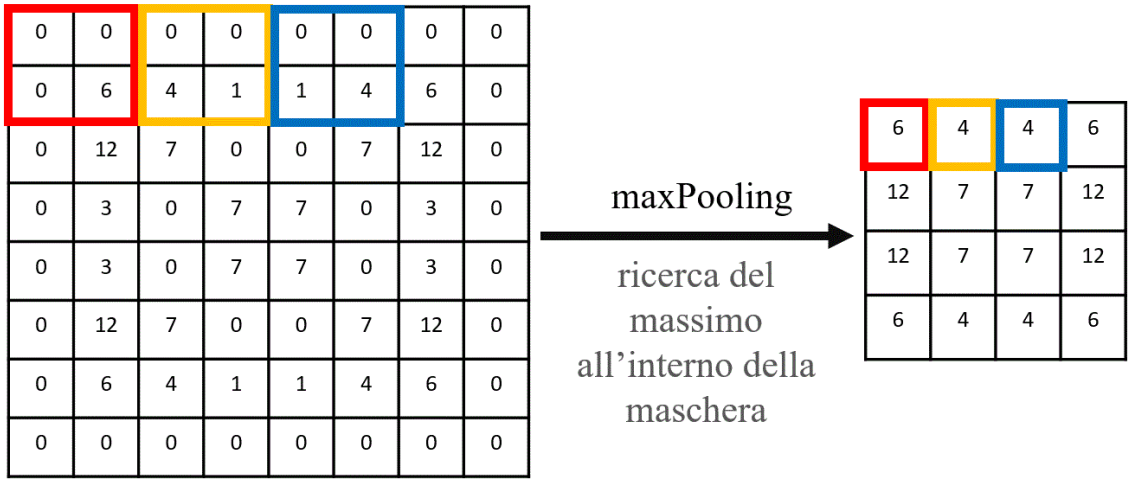
1. Un componente di gestione dell'enable delle *FIFO* e *shift-register*;
2. Una catena di *FIFO* e memorie a scorrimento (il numero di livelli di memoria è pari a 4, perché, per ipotesi, *POOL_DIM* risulta essere pari a 4);
3. Un componente di calcolo del massimo (in figura *max 4x4*) che cerca il massimo tra i sedici elementi (in figura *data* in riquadri gialli) della struttura di *shift-register*;
4. Un management per la validità dell'uscita (*poolValid_o*) che sfrutta i bit annessi al dato posto in memoria.

Come si può notare dalla figura precedente, il dato collocato in memoria non è infatti isolato, ma accompagnato da 4 bit identificativi:

1. Un bit per informare della fine dell'immagine (*endOfFrame*): questo segnale accompagna il dato per avere in uscita l'*endOfFrame_o* associato all'ultimo massimo identificato nell'immagine (si noti però che gli *endOf-* qui prodotti non verranno utilizzati dai componenti successivi, per questo motivo non vengono riportati nell'immagine Imm.3.6.a.);
2. Un bit per identificare la fine di una riga (*endOfLine*): questo similmente viene sfruttato per aver il corretto *endOfLine_o* in uscita;
3. Un bit per gestire le righe: questo è pari a 1 esclusivamente in corrispondenza di righe il cui numero identificativo è divisibile per quattro, in altre parole questo bit delinea il bordo inferiore della maschera di selezione (*rowFlag*);
4. Un bit per gestire le colonne: questo similmente a *rowFlag*, risulta essere pari a 1 in corrispondenza di colonne il cui numero identificativo è divisibile per quattro (*colFlag* delinea il bordo destro della finestra di selezione).

Proprio attraverso questi ultimi due flag è possibile capire se i dati presenti nell'architettura degli *shift-register* corrispondano a una selezione di ricerca del massimo; nel momento in cui la selezione è completa infatti, il dato posto nel vertice in alto a sinistra dell'architettura dei registri a scorrimento presenta un doppio flag (*rowFlag*; *colFlag*) = (1; 1), questa combinazione di segnali viene così tradotta in un poolValid_o pari a 1, che identifica un corretto massimo di selezione.

Il funzionamento di *MaxPooling* è esplicato dall'immagine Imm.3.6.b. sotto riportata. Si può notare come viene eseguita la ricerca del massimo, ovvero su finestre disgiunte collocate le une di seguito alle altre. Per ognuna di queste finestre viene riportato in uscita il massimo in esse contenuto.



Imm.3.6.b.: Funzionamento del blocco MaxPooling

Si deve inoltre notare che il blocco *MaxPooling* è stato implementato per essere in grado di gestire finestre di selezione di dimensione 2x2, 3x3 e 4x4, ovvero *POOL_DIM* pari a 2, 3 o 4. Non sono gestibili maschere di dimensioni maggiori, in quanto il componente di ricerca del massimo, chiamato *Max*, presenta solo tre generate associati alle dimensioni della finestra sopracitati; è naturalmente possibile estendere la funzionalità di *Max*, permettendogli di accettare *POOL_DIM* maggiori, ma questo comporterebbe un carico computazionale molto maggiore, in quanto sarebbero necessari 25, 36, etc. confronti (si noti che *Max*, come si vedrà di seguito, rappresenta già in questo caso il percorso più critico dell'intera struttura *ChainOfComponents*, perciò complicarlo ulteriormente non è desiderabile). Si noti inoltre che generalmente nella pratica non vengono mai utilizzate maschere di ricerca del massimo maggiori di una 4x4; questo perciò giustifica la scelta fatta nell'implementare il componente *MaxPooling*.

Si riporta nelle tabelle Tab.3.6.a. e Tab.3.6.b. un riassunto degli ingressi e delle uscite di *MaxPooling* e di *Max*, con spiegazioni sul significato dei nomi dei segnali e numero di bit associati.

Nome del segnale	Ingresso/Uscita	Numero di bit	Utilità del segnale
clk_i	Ingresso	1 bit	Clock di sistema
nReset_i	Ingresso	1 bit	Reset dell'esecuzione dell'elaborazione
dataValid_i	Ingresso	1 bit	Validità dell'uscita del filtro
dataValue_i	Ingresso	12 bit	Valore dell'uscita del filtro
endOfLine_i	Ingresso	1 bit	Segnale di fine riga dell'immagine (corrispondente all'ultima uscita del filtro della riga)
endOfFrame_i	Ingresso	1 bit	Segnale di fine immagine (corrispondente all'ultima uscita del filtro)
poolValid_o	Uscita	1 bit	Validità dell'uscita del *MaxPooling*
poolValue_o	Uscita	12 bit	Valore dell'uscita del *MaxPooling*
endOfLine_o	Uscita	1 bit	Segnale di fine riga (corrispondente all'ultimo dato della riga)
endOfFrame_o	Uscita	1 bit	Segnale di fine immagine (corrispondente all'ultimo dato consegnato in uscita)

Tab.3.6.a.: Riassunto dei segnali di ingresso e di uscita di MaxPooling

Nome del segnale	Ingresso/Uscita	Numero di bit	Utilità del segnale
dataIn_i	Ingresso	192 bit	Valore di 4x4 uscite del filtro (da 12 bit), poste nell'opportuna maschera di ricerca del massimo
dataOut_o	Uscita	12 bit	Valore del massimo all'interno della maschera di ricerca del massimo

Tab.3.6.b.: Riassunto dei segnali di ingresso e di uscita di Max

Anche per il componente *MaxPooling* si riportano frammenti di codice significativi e relative spiegazioni; in questo blocco si fa uso delle funzioni:

1. *For generate*, per generare un opportuno numero di livelli di memoria *FIFO* + *shift-register*:

```
shift_register_generate: for L in 1 to POOL_DIM generate
        shift_register_L: entity shift_register generic map(
        KERNEL_WIDTH => POOL_DIM,
        DATA_WIDTH => NEW_DATA_WIDTH+4)
        port map(
        clk_i => clk_i,
        writeEnable_i=> wr_vector(L+1),
        […]);
end generate shift_register_generate;
```

2. *Entity*, per piazzare componenti singoli nel circuito:

```
row_management: entity shift_register_MaxPooling generic map(
POOL_DIM => POOL_DIM)
port map(
clk_i => clk_i,
writeEnable_i=> dataValid_i,
Reset_i => Reset_row,
dataOut_o => row_flag);
```

   Si noti, in questo esempio, l'uso di un componente appositamente creato per gestire i flag *rowFlag* e *colFlag*, ovvero *shift_register_MaxPooling*, un particolare registro ciclico con enable e reset che non prende in ingresso alcun

dato, ma fa solo ricircolare un valore 1 al suo interno; l'uscita di questo componente sarà perciò (nel caso di maschera di ricerca del massimo di dimensioni 4x4) pari a $0 \xrightarrow{enable} 0 \xrightarrow{enable} 0 \xrightarrow{enable} 1 \xrightarrow{enable} 0 \xrightarrow{enable} ...$

3. *Process*, per gestire processi sincroni e non, come il processo *out_management*, di seguito riportato, che si occupa del buffer in uscita:

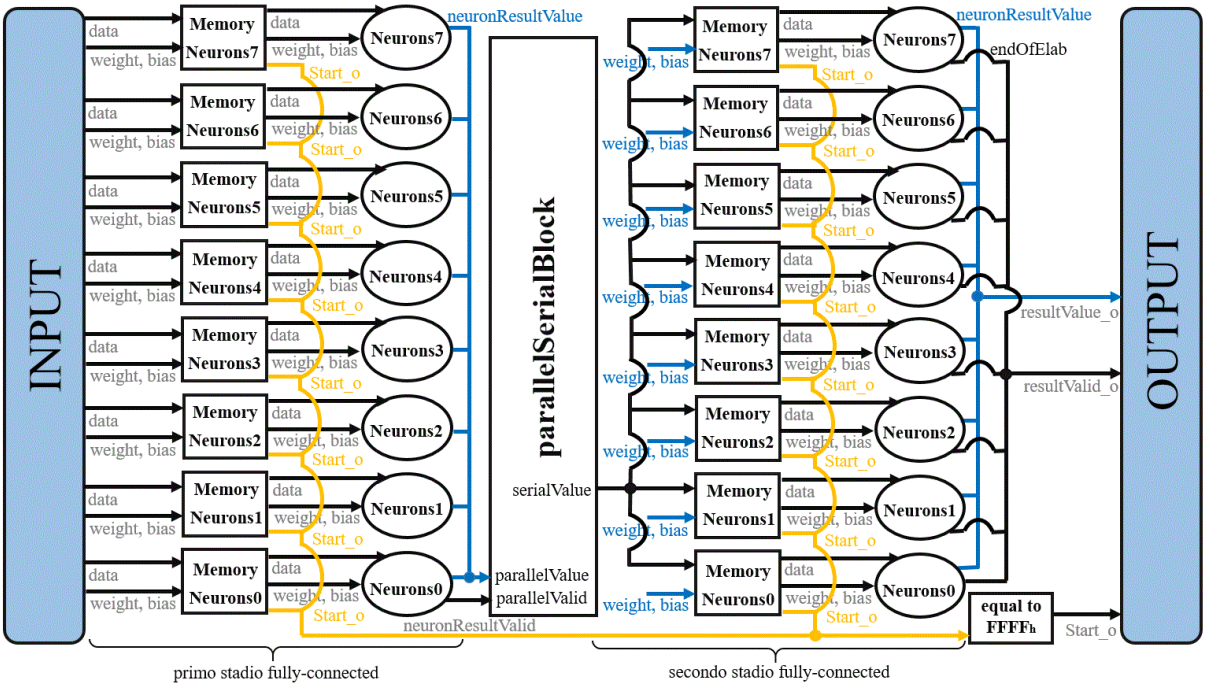
```
output_management: process(clk_i)
begin
        if clk_i'event and clk_i='1' then
                if (enable_out='1' [...]) then
                        poolValue_o<=poolValue_c;
                end if;
                poolValid_o<=poolValid_c;
                endOfLine_o<=endOfLine_c1;
                endOfFrame_o<=endOfFrame_c1;
        end if;
end process;
```

3.7. *NetworkOfNeurons* – stadi fully-connencted della rete di neuroni

Con *NetworkOfNeurons* si entra nell'ultimo stadio di *ChainOfComponents*; questo rappresenta il componente più complesso dell'intera catena e il cuore della rete neurale. È costituito, come visibile nell'immagine Imm.3.7.a., dalla ripetizione di molti blocchi elementari:

1. Il componente *MemoryNeurons*, che permette la sincronizzazione tra il dato rilasciato dal *MaxPooling* o *ParallelSerialBlock* e i corrispondenti pesi del neurone;
2. Il blocco *Neurons*, vero e proprio neurone della rete neurale, che esegue i calcoli di moltiplicazione e accumulatore opportuni, rilasciando il risultato in uscita;
3. Il gestore *ParallelSerialBlock*, un componente che converte un set di segnali forniti in parallelo in un flusso sequenziale di dati.

Come visibile dall'immagine successiva, in *NetworkOfNeurons* sono presenti due stadi fully-connected, ovvero due stadi di calcolo in cui le operazioni vengono eseguite sull'interezza del set di dati forniti.



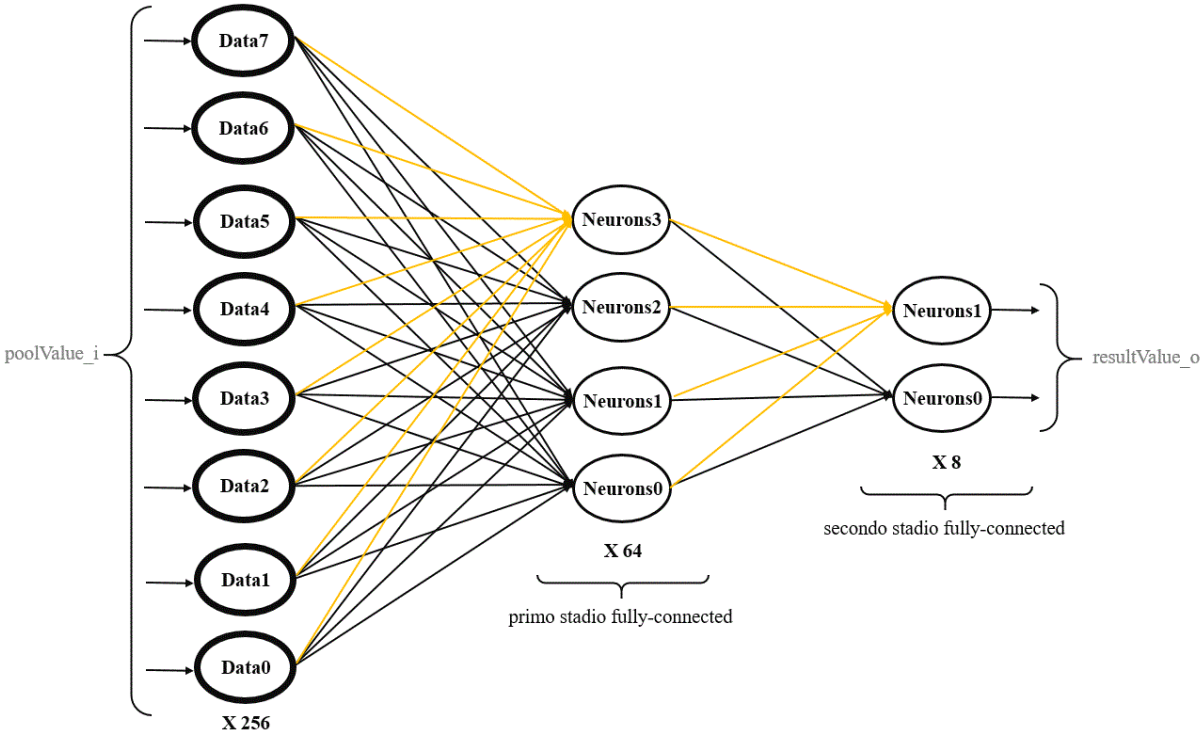
Imm.3.7.a.: Struttura del blocco NetworkOfNeurons

Si può osservare la complessità dei dati forniti ai diversi ingressi dei sottocomponenti e rilasciati in uscita dagli stessi:

1. *MemoryNeurons* prende in ingresso il dato fornito dal *MaxPooling* (*data*), i pesi dei neuroni *(weight*) e le *bias* (quest'ultime fornite durante la programmazione); in uscita invece rilascia, oltre agli analoghi degli ingressi, anche il segnale *Start_o*, segnale di avvenuta programmazione della memoria del neurone;
2. *Neurons* ha in ingresso *data*, *weight* e *bias* opportunamente sincronizzati dal *MemoryNeurons* a monte; in uscita portano tre possibili segnali (non mostrati, per comodità, in uscita a ogni neurone), ovvero *neuronResultValid_o* (validità del risultato in uscita dal neurone), *neuronResultValue_o* (valore del risultato) e *endOfElab_o* (segnale di avvenuta elaborazione). Si noti come quest'ultimo segnale sia usato come verifica della correttezza del risultato: se *endOfElab_o* si attiva contemporaneamente a *neuronResultValid_o*, allora non sono avvenuti errori durante il calcolo (questo, come molti altri accorgimenti e verifiche descritti in passato, aumentano l'affidabilità di *ChainOfComponents*);
3. *ParallelSerialBlock* prende in ingresso i valori rilasciati dai neuroni del primo stadio fully-connected (un set di 8x12 bit) e una validità di uno degli stessi neuroni, in uscita invece rilascia uno alla volta i dati da 12 bit, che diverranno gli ingressi dei *MemoryNeurons* del secondo stadio fully-connected.

Si noti inoltre la gestione dei segnali *Start_o*: questi vengono raccolti in un unico vettore di segnali di avvenuta programmazione e il vettore così ottenuto viene confrontato con un vettore settato completamente a 1; *Start_o*, quale segnale di uscita di *NetworkOfNeurons* sarà pari a 1 solo se tutti gli *Start_o* prodotti dai 16 *MemoryNeurons* qui presenti saranno posti a 1 (questo rappresenta un ulteriore controllo ai fini della correttezza della programmazione).

Nell'immagine Imm.3.7.b. è riportata una esplicazione grafica del funzionamento di *NetworkOfNeurons*; ivi è possibile osservare la connessione di ogni dato con ogni neurone presente (fully-connected) e l'organizzazione a due layer/stadi fully-connected.



Imm.3.7.b.: Funzionamento del blocco NetworkOfNeurons

Naturalmente, l'immagine qui proposta rappresenta una semplificazione del caso reale, in quanto (come evidenziato dalle didascalie poste sotto ogni blocco di elaborazione) il numero dei dati consegnati in ingresso al componente e il numero dei neuroni per ogni stadio è molto maggiore di quello qui visibile.

È necessario notare la motivazione dietro l'utilizzo del componente *ParallelSerialBlock*: quest'ultimo permette di ottenere un numero di neuroni (che saranno "virtuali") per il primo stadio molto maggiore rispetto alla quantità di *Neurons* fisicamente piazzati nel circuito. Si noti infatti che i neuroni fisicamente presenti nel circuito sono pari a otto, quando invece il numero di risultati ottenuti, ovvero di neuroni

virtuali, risulta essere pari a 64. Questo è reso possibile grazie al *ParallelSerialBlock* che gestisce la ciclicità del dato prodotto dal primo stadio e grazie ai componenti *MemoryNeurons* posti a monte dei neuroni del primo stadio, che non solo sincronizzano i dati rispetto ai pesi, ma memorizzano i diversi ingressi e li ripropongono ciclicamente ai componenti *Neurons* annessi; questo permette di avere, a seguito di una programmazione piuttosto onerosa (256x8 pesi per ogni neurone), il comportamento ciclico del primo stadio fully-connected qui presentato.

Anche per *NetworkOfNeurons* si riporta una tabella degli ingressi e delle uscite del componente (tabella Tab.3.7.a).

Nome del segnale	Ingresso/Uscita	Numero di bit	Utilità del segnale
clk_i	Ingresso	1 bit	Clock di sistema
nReset_i	Ingresso	1 bit	Reset dell'esecuzione dell'elaborazione
poolValid_i	Ingresso	1 bit	Validità dell'uscita del *MaxPooling*
poolValue_i	Ingresso	12 bit	Valore dell'uscita del *MaxPooling*
neuronWeightValid1	Ingresso	8 bit	Validità dei pesi dei neuroni (1° stadio)
neuronWeightValue1	Ingresso	8x4 bit	Valore dei pesi dei neuroni (1° stadio)
endOfWeight1	Ingresso	8 bit	Segnale di fine della programmazione dei pesi dei neuroni (1° stadio)
neuronWeightValid2	Ingresso	8 bit	Validità dei pesi dei neuroni (2° stadio)
neuronWeightValue2	Ingresso	8x4 bit	Valore dei pesi dei neuroni (2° stadio)
endOfWeight2	Ingresso	8 bit	Segnale di fine della programmazione dei pesi dei neuroni (2° stadio)
biasValid1	Ingresso	8 bit	Validità dei bias dei neuroni (1° stadio)
biasValue1	Ingresso	8x24 bit	Valore dei bias dei neuroni (1° stadio)
endOfBias1	Ingresso	8 bit	Segnale di fine della programmazione dei bias dei neuroni (1° stadio)
biasValid2	Ingresso	8 bit	Validità dei bias dei neuroni (2° stadio)
biasValue2	Ingresso	8x26 bit	Valore dei bias dei neuroni (2° stadio)
endOfBias2	Ingresso	8 bit	Segnale di fine della programmazione dei bias dei neuroni (2° stadio)
Start_o	Uscita	1 bit	Segnale di avvenuta programmazione (enable dell'esecuzione)
resultValid_o	Uscita	8 bit	Validità del dato in uscita
resultValue_o	Uscita	128 bit	Valore del dato in uscita

Tab.3.7.a.: Riassunto dei segnali di ingresso e di uscita di NetworkOfNeurons

Per gestire la complessità del componente *NetworkOfNeurons*, il suo codice di programmazione presenta molte funzioni precedentemente già citate ed alcuni accorgimenti riportati in seguito:

1. Funzione *for generate*, che rappresenta la funzione più usata nella realizzazione di *NetworkOfNeurons*:

   memoryNeurons1_generate: for L in 0 to NUM_OF_NEURONS1-1 generate
   memoryNeurons1: entity memoryNeurons

```
generic map(
        NEURONSET_NUM => NEURONSET_NUM1,
        [...])
port map(
        clk_i => clk_i,
        nReset_i => nReset_i,
        endOfWeights_i => endOfWeight1(L),
        [...]
        Start_o => Start_vector(L));
end generate memoryNeurons1_generate;
```

2. Funzione *for loop*, usata per costruire le opportune connessioni tra i segnali:

```
parallelValue_loop: for L in 1 to NUM_OF_NEURONS1 loop
                parallelValue(L*(PS_DATA_WIDTH+1)-1 downto (L-1)*(PS_DATA_WIDTH+1)) <=
        '1'     &     neuronResultValuePS(RESULT_WIDTH*(L-1)+PS_DATA_WIDTH-1     downto
RESULT_WIDTH*(L-1));
        end loop parallelValue_loop;
```

3. Uso della concatenazione dei dati (uso di &), riportato ad esempio nella citazione precedente, utile per apporre la validità ai dati in ingresso a *ParallelSerialBlock*:

```
'1' & neuronResultValuePS([range])
```

4. Funzione *Entity*, per piazzare il blocco *ParallelSerialBlock*:

```
parallel_serial_block_component: entity parallel_serial_block
generic map(
        NUM_OF_NEURONS => NUM_OF_NEURONS1,
        PS_DATA_WIDTH => PS_DATA_WIDTH)
port map(
        clk_i => clk_i,
        [...]
        serialValid_o => serialValid2);
```

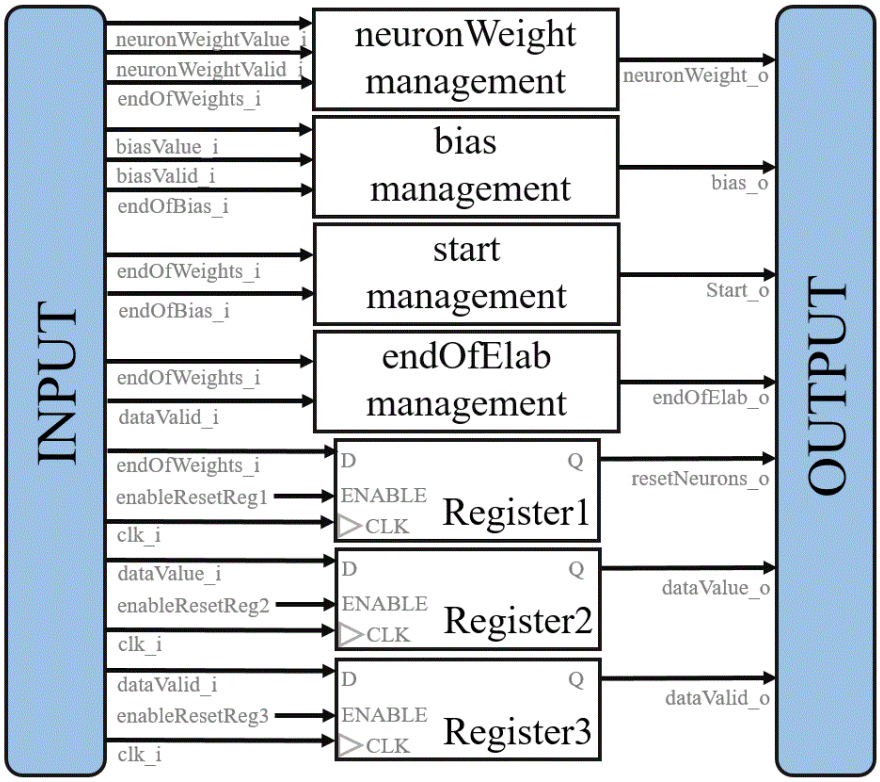
5. Funzione *Process*, utilizzato per gestire il *for loop* e la generazione del segnale *Start_o*:

```
Start_o_management: process(Start_vector, Start_vector_equal)
begin
        if Start_vector=Start_vector_equal then
                Start_o<='1';
        else
                Start_o<='0';
        end if;
end process;
```

3.7.1. *MemoryNeurons* – memoria dei pesi dei neuroni

MemoryNeurons rappresenta il blocco a monte di ogni neurone degli stadi fully-connected. Il suo nome può trarre in inganno, in quanto non funge semplicemente da memoria, ma anche da sistema di gestione e sincronizzazione dei dati e dei pesi da consegnare a valle. *MemoryNeurons* presenta una struttura piuttosto intuitiva in cui ogni sottoblocco è adibito a una specifica gestione; infatti, come visibile nell'immagine Imm.3.7.c., l'intero sistema è costituito da management (blocchi di gestione costituiti

da memorie e selettori) e registri a scorrimento che memorizzano e ripropongono ciclicamente i dati da elaborare.



Imm.3.7.c.: Struttura del blocco MemoryNeurons

MemoryNeurons presenta molti ingressi, che vengono consegnati a più management contemporaneamente, e diverse uscite, ognuna prodotta da un gestore. Si riporta nella tabella Tab.3.7.b. un riassunto degli ingressi e delle uscite di *MemoryNeurons*.

Nome del segnale	Ingresso/Uscita	Numero di bit	Utilità del segnale
clk_i	Ingresso	1 bit	Clock di sistema
nReset_i	Ingresso	1 bit	Reset dell'esecuzione dell'elaborazione
endOfWeights_i	Ingresso	1 bit	Segnale di fine della trasmissione dei pesi
neuronWeightValid_i	Ingresso	1 bit	Validità del peso
neuronWeightValued_i	Ingresso	4 bit	Valore del peso
dataValid_i	Ingresso	1 bit	Validità del dato
dataValue_i	Ingresso	12 bit	Valore del dato
endOfBias_i	Ingresso	1 bit	Segnale di fine della trasmissione delle bias
biasValid_i	Ingresso	1 bit	Validità del bias
biasValue_i	Ingresso	26 bit	Valore del bias
neuronWeight_o	Uscita	4 bit	Valore del peso
bias_o	Uscita	26 bit	Valore del bias
dataValue_o	Uscita	12 bit	Valore del dato
dataValid_o	Uscita	1 bit	Validità del dato (e del peso annesso)
endOfElab_o	Uscita	1 bit	Segnale di fine elaborazione
resetNeurons_o	Uscita	1 bit	Segnale di reset del neurone (nuovo inizio del calcolo)
Start_o	Uscita	1 bit	Segnale di avvenuta programmazione dei neuroni

Tab.3.7.b.: Riassunto dei segnali di ingresso e di uscita di MemoryNeurons

Si riporta qui di seguito una breve trattazione dei diversi management e dei relativi ingressi e uscite, con frammenti del codice di programmazione:

1. *neuronWeight management*: blocco di gestione dei pesi dei neuroni, consegna gli opportuni pesi associati ai *dataValue_o*. Si basa su di una memoria che viene letta a seguito dell'attivazione di un *enable* di lettura, che inizialmente (durante l'acquisizione dei *poolValid_o*) equivale alla validità del dato e poi va ad eguagliare il segnale di clock di sistema.

```
Lettura_memoryWeight: process([sensitivity list])
begin
        if nReset_i='0' or Start_c='0' or endOfElab_c1='1' then
                neuronWeight_o<=(others=>'0');
        elsif clk_i'event and clk_i='1' and (dataValid_c='1' or readingData_ON='0') then --!se leggo il
valore la programmazione è già avvenuta
                neuronWeight_o<=memoryWeight(NEURONS_WIDTH-1 downto 0);
        end if;
end process;
```

2. *bias management*: gestore che consegna, all'inizio di un nuovo calcolo del neurone, l'opportuno bias; questa consegna è associata all'attivazione del segnale *resetNeurons_o*, in quanto segue la sua commutazione a 1. Il *bias management* si basa anch'esso su di una memoria opportunamente scritta in fase di programmazione e letta in fase di calcolo ed elaborazione.

```
Lettura_memoryBias: process(dataValid_i, clk_i, nReset_i, Start_c, readingData_ON, endOfElab_c1)
begin
        if nReset_i='0' or Start_c='0' or endOfElab_c1='1' then
                bias_o<=(others=>'0');
        elsif clk_i'event and clk_i='1' and (dataValid_i='1' or readingData_ON='0') then
        --!se leggo il valore la programmazione è già avvenuta (=readingData_ON='0')
                bias_o<=memoryBias(NEW_DATA_WIDTH-1 downto 0);
        end if;
end process;
```

3. *start management*: semplice blocco logico basato su un *process* e un'operazione logica AND; consegna in uscita *Start_o* che viene poi utilizzato dal componente *Kernel* come segnale di avvio all'elaborazione.

```
Start_o_management: process(nReset_i, clk_i)
begin
        if nReset_i='0' then
                Start_o<='0';
        elsif clk_i'event and clk_i='1' then
                Start_o<=Start_c;
        end if;
end process;
Start_c<=endOfWeights_ON and endOfBias_ON;
```

4. *endOfElab management*: gestore del segnale che indica la fine dell'elaborazione. endOfElab_o commuta a 1 non appena si riporta in uscita l'ultimo dato dell'elaborazione; questo segnale viene sfruttato dall'ultimo stadio fully-connected come validità del risultato finale.

```
Lettura_memoryWeight: process([sensitivity list])
begin
        if nReset_i='0' or Start_c='0' or endOfElab_c1='1' then
                endOfElab_c<='0';
        elsif clk_i'event and clk_i='1' and (dataValid_c='1' or readingData_ON='0') then
                endOfElab_c<=memoryWeight(NEURONS_WIDTH);
        end if;
end process;
endOfElab_o<=endOfElab_c;
```

5. Registri per *resetNeurons_o*: memoria di gestione del reset dei neuroni. Il segnale *resetNeurons_o* si attiva e commuta a 1 a seguito della conclusione dei calcoli del blocco *Neuron*s, ovvero a seguito della completa consegna dei pesi e dei dati.

```
Lettura_memoryData: process(clk_i, nReset_i, Start_c, endOfElab_c1)
begin
                resetNeurons_c<='0';
                resetNeurons_c1<='0';
        elsif clk_i'event and clk_i='1' then --! se leggo il valore la programmazione è già avvenuta
                resetNeurons_c<=memoryData(NEW_DATA_WIDTH);
                resetNeurons_c1<= resetNeurons_c; --! Ritardo il resetNeurons
        end if;
end process;
resetNeurons_o_management: process(firstResetNeurons, dataValid_c, resetNeurons_c1)
begin
        if firstResetNeurons='1' and dataValid_c='1' then
                resetNeurons_o<='1';
        else
                resetNeurons_o<=resetNeurons_c1;
        end if;
end process;
```

6. Registri per *dataValue_o* e *dataValid_o*: registri per la memorizzazione e rilascio dei dati da elaborare e per la relativa validità. I *dataValue_o* derivano da una memoria appositamente adibita ai dati, chiamata *memoryData*, invece la validità annessa è calcolata da un'operazione logica basata su NOT, AND e OR.

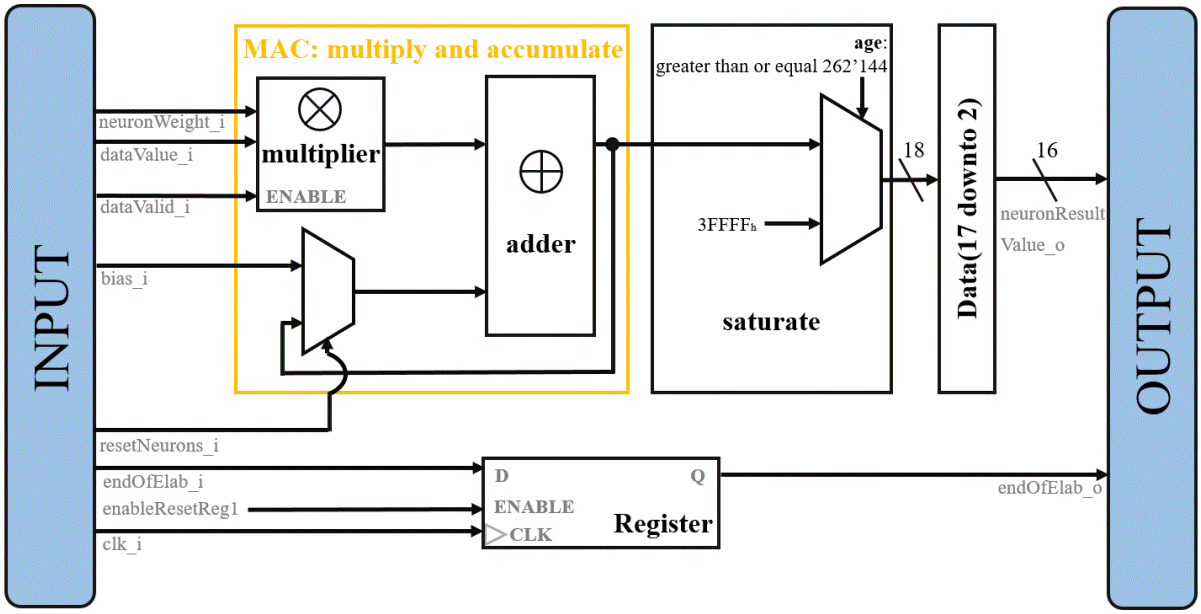
```
Lettura_memoryData: process(clk_i, nReset_i, Start_c, endOfElab_c1)
begin
        if nReset_i='0' or Start_c='0' or endOfElab_c1='1' then
                dataValue_o<=(others=>'0');
                dataValid_o<='0';
        elsif clk_i'event and clk_i='1' then --!se leggo il valore la programmazione è già avvenuta
                dataValue_o<=memoryData((NEURONSET_NUM+1)*(NEW_DATA_WIDTH+1)-2
downto (NEURONSET_NUM)*(NEW_DATA_WIDTH+1));
                dataValid_o<=(not(resetNeurons_c) and (dataValid_c or not(readingData_ON)));
        end if;
end process;
```

3.7.2. *Neurons* – neurone della rete neurale

Neurons è il componente di elaborazione corrispondente al neurone della rete neurale a due stadi fully-connected. Questo è costituito da due soli macroblocchi di gestione:

1. Un blocco di calcolo, vero e proprio cuore del neurone;
2. Un registro per l'*endOfElab_i*, che genera *endOfElab_o* semplicemente ritardando il segnale omonimo in ingresso.

Nell'immagine Imm.3.7.d. è riportata la struttura alla base del componente *Neurons*.



Imm.3.7.d.: Struttura del blocco Neurons

Nell'immagine soprariportata è ben visibile il macroblocco di calcolo, che presenta tre layer fondamentali:

1. Un *MAC*, ovvero un *multiply and accumulate*, che prende in ingresso gli opportuni dati e pesi, li moltiplica assieme e li somma col risultato precedentemente calcolato dall'*adder*; si noti la presenza di un multiplexer di retroazione che consegna all'*adder* o il suo precedente risultato o il bias in ingresso, questo perché, all'inizio di un nuovo calcolo, il risultato precedente viene resettato e l'offset iniziale viene subito sommato col primo risultato del moltiplicatore;
2. Un blocco di gestione della politica *saturate*, in cui il dato prodotto dal *MAC* viene confrontato col massimo numero esprimibile con 18 bit (unsigned perché

è presente all'interno del calcolo anche una ReLU) e settato pari al minimo tra il massimo esprimibile e il risultato così ottenuto;

3. Un blocco divisore, che riduce la dimensione del dato in conseguenza della sua divisione per 4 (ovvero la selezione dei 16 bit più significativi sui 18 presenti).

Nella seguente tabella Tab.3.7.c. sono riportate gli ingressi e le uscite del blocco *Neurons*, con una breve spiegazione dei diversi segnali.

Nome del segnale	Ingresso/Uscita	Numero di bit	Utilità del segnale
clk_i	Ingresso	1 bit	Clock di sistema
nReset_i	Ingresso	1 bit	Reset dell'esecuzione dell'elaborazione
dataValid_i	Ingresso	1 bit	Validità del dato in ingresso
dataValue_i	Ingresso	12/16 bit	Valore del dato in ingresso (di 12 bit nel 1° stadio e di 16 bit nel 2° stadio)
neuronWeight_i	Ingresso	4 bit	Valore del peso del neurone
bias_i	Ingresso	24/26 bit	Valore del bias (di 24 bit nel 1° stadio e di 26 bit nel 2° stadio)
resetNeurons_i	Ingresso	1 bit	Segnale di reset del neurone
endOfElab_i	Ingresso	1 bit	Segnale di fine elaborazione
neuronResultValid_o	Uscita	1 bit	Validità del risultato in uscita
neuronResultValue_o	Uscita	16 bit	Valore del risultato in uscita
endOfElab_o	Uscita	1 bit	Segnale di fine elaborazione

Tab.3.7.c.: Riassunto dei segnali di ingresso e di uscita di Neurons

Utilizzando all'interno di *Neurons* solo le precedentemente citate *Entity* e *Process*, si citano qui non frammenti di codice del neurone, ma estratti di programmazione del suo primo layer di calcolo, ovvero del *MAC*. All'interno del componente *MAC* si ritrovano diverse funzioni:

1. Funzione *process*, utilizzata nei diversi management dei segnali interni e dei prodotti intermedi dell'elaborazione (ad esempio addendo1 e addendo2):

```
addendo2_management: process(nReset_i, resetNeurons_i, clk_i, bias_i)
begin
if nReset_i='0' then
        addendo2<=(others=>'0');
elsif resetNeurons_i='1' then
        addendo2(NEW_DATA_WIDTH-1 downto 0)<=bias_i;
        if bias_i(NEW_DATA_WIDTH-1)='1' then
                addendo2(RESULT_MAC_WIDTH-1 downto NEW_DATA_WIDTH)<=(others=>'1');
        else
                addendo2(RESULT_MAC_WIDTH-1 downto NEW_DATA_WIDTH)<=(others=>'0');
        end if;
elsif clk_i'event and clk_i='1' then
        addendo2<=resultAdder;
end if;
end process;
```

Si può notare nel codice soprariportato come l'addendo2 sia posto uguale al bias, opportunamente esteso alla giusta dimensione con degli 0 se positivo o con degli

1 se negativo (nel rispetto dell'espressione a complemento a 2), quando si ha un reset del neurone (*resetNeurons_i = '1'*) o posto pari al risultato dell'*adder* altrimenti (ovvero durante il suo normale funzionamento).

2. Funzione *if generate*, grazie alla quale è possibile la gestione di due diversi dimensionamenti del dato (12 e 16 bit del dato):

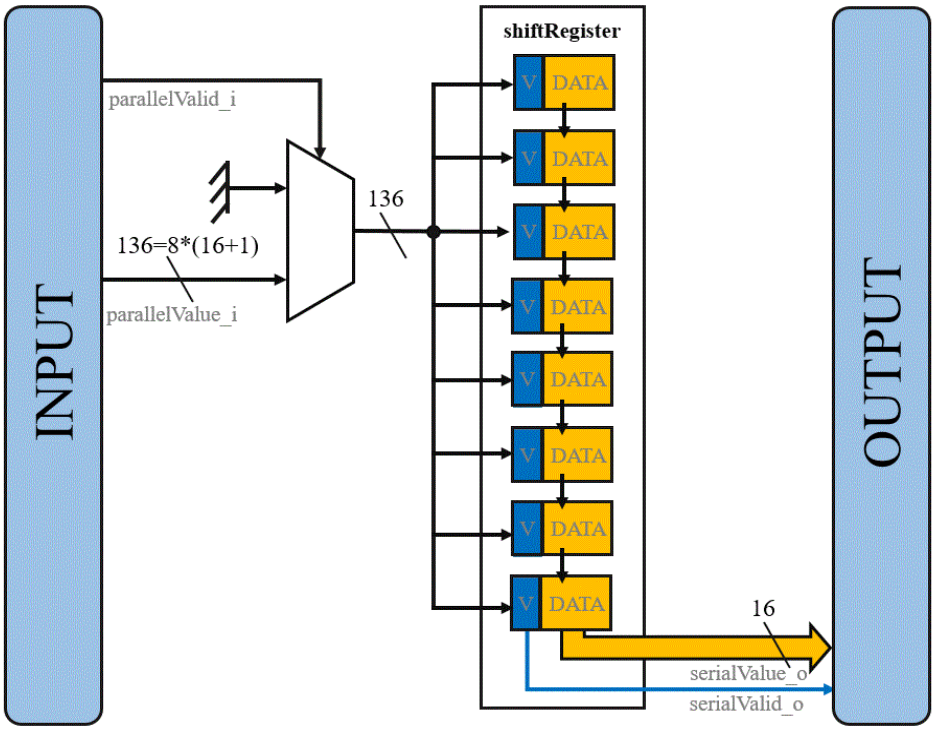
```
mult12_4_generate: if NEW_DATA_WIDTH=12 generate
mult12_4_component : mult12_4 PORT MAP (
        dataa => dataValue_i,
        datab => neuronWeight_i,
        result => resultMult);
end generate mult12_4_generate;
```

3. Piazzamento attraverso mapping (uso di *port map*) per piazzare *adder* e il gestore del flag di saturazione *saturate_Neurons*:

```
saturate_Neurons: saturateNeurons PORT MAP(
                dataa => resultAdder,
                ageb => saturate_flag);
```

3.7.3. *ParallelSerialBlock* – blocco parallelo seriale

ParallelSerialBlock rappresenta l'anello di giunzione dei due diversi stadi fully-connected; è un convertitore parallelo-seriale, ovvero un componente che riceve in ingresso un set di dati in modo parallelo e rilascia in uscita serialmente un dato alla volta. Nell'immagine Imm.3.7.e. è riportata la struttura base di *ParallelSerialBlock*.



Imm.3.7.e.: Struttura del blocco ParallelSerialBlock

È possibile notare nell'immagine Imm.3.7.e. i due elementi base del componente in esame:

1. Un multiplexer che consegna i dati prodotti dai diversi neuroni a monte di una particolare memoria a scorrimento, esclusivamente in corrispondenza di una validità dei dati (*parallelValid_i = '1'*) (altrimenti l'uscita del MUX è settata a un vettore di soli 0);
2. Uno *shift-register* inizializzabile in parallelo, che accoglie al suo interno dati e opportune validità.

ParallelSerialBlock presenta diversi pin di ingresso e di uscita, in particolare 129 pin di ingresso, di cui uno solo di validità e 128 di dato, e 17 pin di uscita, di cui nuovamente solo uno di validità. Nella tabella Tab.3.7.d. è riportato un riassunto degli ingressi e delle uscite del componente *ParallelSerialBlock*.

Nome del segnale	Ingresso/Uscita	Numero di bit	Utilità del segnale
clk_i	Ingresso	1 bit	Clock di sistema
nReset_i	Ingresso	1 bit	Reset dell'esecuzione dell'elaborazione
parallelValid_i	Ingresso	1 bit	Validità del set di dati in ingresso
parallelValue_i	Ingresso	128 bit	Valore del set di dati in ingresso
serialValid_o	Uscita	1 bit	Validità del dato seriale in uscita
serialValue_o	Uscita	16 bit	Valore del dato seriale in uscita

Tab.3.7.d.: Riassunto dei segnali di ingresso e di uscita di ParallelSerialBlock

Il funzionamento di questo componente è estremamente semplice e strutturato in due fasi principali (vengono riportati anche estratti del codice per completarne l'analisi):

1. Fase di *Reading*, ovvero fase di lettura dei dati in ingresso e di scrittura degli stessi in memoria; in questa fase viene letto e memorizzato l'intero set parallelo e, in assenza della validità di lettura, vengono fatti scorrere i dati all'interno della memoria:

```
Reading: process(nReset_i, clk_i, parallelValid_i)
begin
        if nReset_i='0' then
                memory <= (others=>'0');
        elsif clk_i'event and clk_i='1' then
                if parallelValid_i='1' then
                        memory <= parallelValue_i;
                else
                        writing_loop: for L in 1 to NUM_OF_NEURONS-1 loop
                                memory([range n])<=memory(([range n+1]);
                        end loop writing_loop;
                        memory([last range])<=(others=>'0');
                end if;
        end if;
end process;
```

2. Fase di *Writing*, ovvero fase di scrittura del dato sull'uscita del componente *serialValue_o* con parallela generazione della sua validità:

```
Writing: process(clk_i, nReset_i)
begin
        if nReset_i='0' then
                serialValid_o<='0';
                serialValue_o<=(others=>'0');
        elsif clk_i'event and clk_i='1' then
                serialValid_o<=memory(PS_DATA_WIDTH);
                serialValue_o<=memory(PS_DATA_WIDTH-1 downto 0);
        end if;
end process;
```

4. RISULTATI E DISCUSSIONI

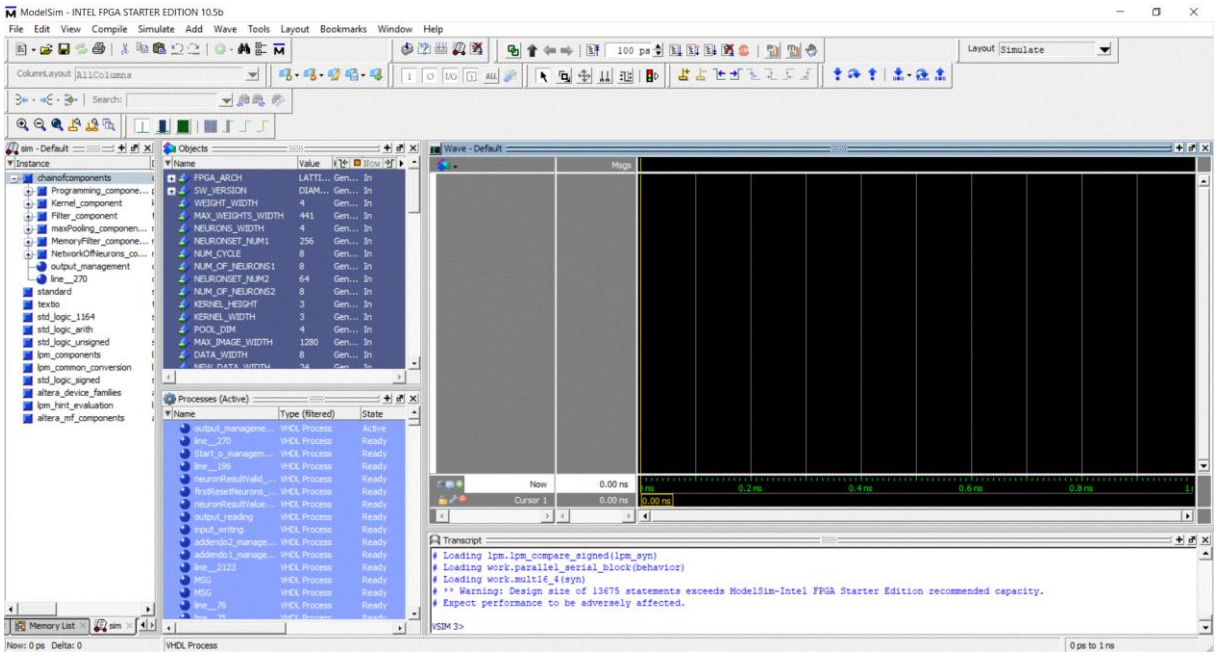
4.1. Verifica funzionale con testbench

La verifica funzionale di *ChainOfComponents*, ovvero la verifica del corretto funzionamento logico del componente è attuata attraverso un duplice controllo:

1. Una simulazione attraverso l'applicativo ModelSim, grazie alla quale (attraverso un opportuno testbench) si possono osservare il rilascio (output) dei dati e le temporizzazioni annesse (espresse come numero di colpi di clock);
2. Una riproposizione software del componente hardware così implementato attraverso l'uso del programma Matlab (successivamente definita "simulazione software").

La perfetta coincidenza dei risultati rilasciati dalle due diverse simulazioni (hardware e software) verifica il corretto funzionamento del componente.

La prima simulazione sopracitata utilizza il comando di lancio (dall'applicativo Quartus): *Tools – Run Simulation Tool – RTL Simulation*; seguendo questo percorso si apre il tool di simulazione ModelSim, di cui si riporta la struttura nell'immagine Imm.4.1.a.



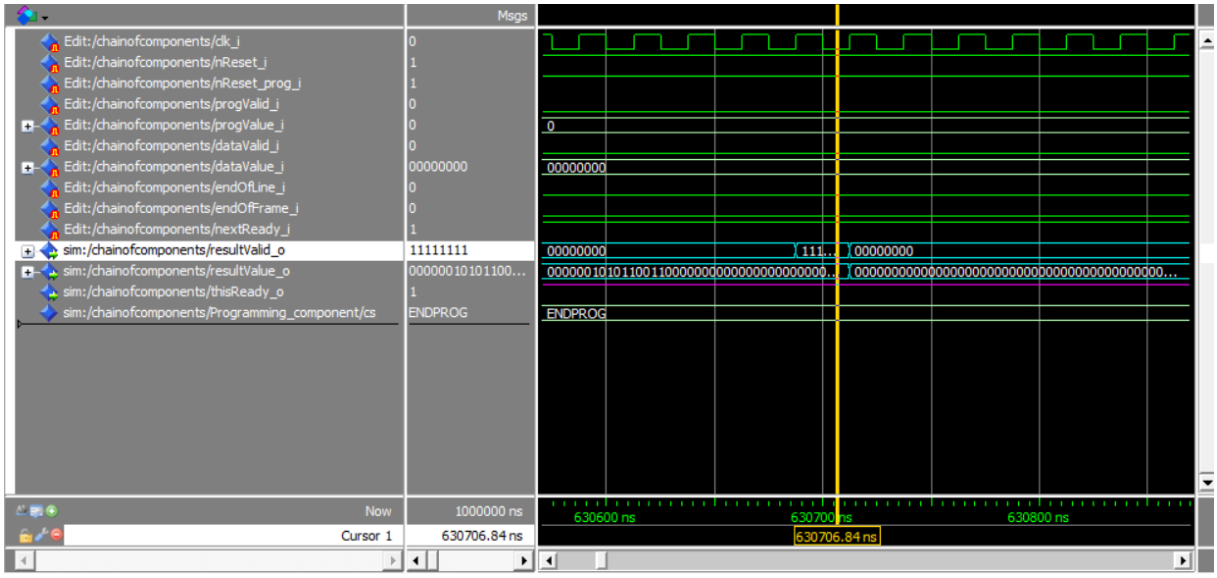
Imm.4.1.a.: Struttura di ModelSim

Attraverso questo applicativo e uno specifico testbench, da applicare alla finestra *Transcript*, è possibile lanciare la simulazione del componente così implementato, dopo aver selezionato i seguenti comandi: *Simulate – Start Simulation – work – filename*.

Il testbench qui utilizzato è generato da un programma Matlab, grazie al quale è possibile ridurre notevolmente il tempo di scrittura del codice (questo perché la stesura a mano risulterebbe troppo onerosa, essendo il testbench composto di 23˙280 istruzioni di assegnazioni). Nel codice qui applicato si sono tradotte le specifiche di progetto presenti nella Tabella Tab.3.1.a. in pixel e pesi specifici, compiendo le seguenti scelte:

1. Il clock qui applicato è stato scelto in base alla verifica della massima frequenza di clock applicabile (si veda il capitolo 4.2. per approfondimenti); questo risulta perciò caratterizzato da un periodo di 25 ns con duty cycle del 50% (scelta solo formale, essendo una simulazione logica-funzionale);
2. L'immagine applicata è un'icona 64x64 in bianco e nero che presenta forti contrasti (immagine ideale sulla quale applicare una selezione dei contorni);
3. Il filtro applicato è rappresentato dalla maschera 3x3 di individuazione dei bordi/contorni riportata nell'immagine Imm.3.5.c.;
4. I pesi dei neuroni degli stati fully-connected si sono scelti caratterizzati da una distribuzione normale (o gaussiana) centrata in zero e con una massima escursione tra -8 e +7; ogni neurone risulta caratterizzato da una distribuzione diversa con media e deviazione standard differenti (questi rappresentano i set di coefficienti di inizializzazione del training).

Applicando tutte le ipotesi sopracitate si è ottenuto il seguente risultato di simulazione (riportato nell'immagine Imm.4.1.b.).



Imm.4.1.b.: Risultato di simulazione raccolto attraverso ModelSim

Si può osservare come a 630˙700 ns *ChainOfComponents* rilascia il proprio risultato di simulazione in *resultValue_o* (quest'ultimo accompagnato da una conferma di validità in *resultValid_o* settato completamente attivo). Come si verificherà in seguito, il numero qui riportato risulta essere equivalente al valore calcolato attraverso l'implementazione software, via Matlab. Si può inoltre osservare come il risultato sia rilasciato dopo 25˙228 colpi di clock (dal reset di sistema); questo elevato numero di cicli è giustificato dall'iniziale programmazione, molto onerosa in termini di cicli di clock in quanto ogni singolo peso del filtro e dei neuroni necessita di un clock per essere assegnato.
Si possono osservare dall'immagine Imm.4.1.b. anche il *thisReady_o* e il *current state* (*cs*):

1. *thisReady_o,* essendo settato a 1, comunica al blocco a monte la sua disponibilità a una nuova elaborazione ancora prima di aver concluso il corrente processing (questo nel rispetto della struttura a pipeline su cui si basa l'intero sistema);
2. *cs* invece è correttamente settato a *ENDPROG*; questo rappresenta una verifica della conclusione dell'iniziale programmazione (se *cs* fosse settato ad uno qualunque degli altri stati possibili, il risultato rilasciato in *resultValue_o* risulterebbe errato).

Si riportano qui di seguito degli estratti del testbench utilizzato (inserito nella finestra *Transcript)*, con precisazioni sul dato settato:

1. Creazione dei segnali e posizionamento degli output da osservare:
   wave create -driver freeze -pattern clock -initialvalue 0 -period 25ns -dutycycle 50 -starttime 0ns -endtime 10000000ns sim:/chainofcomponents/clk_i
   wave create -driver freeze -pattern constant -value 1 -starttime 0ns -endtime 10000000ns sim:/chainofcomponents/nReset_i
   wave create -driver freeze -pattern constant -value 000000000000000000000000 -range 25 0 -starttime 0ns -endtime 10000000ns sim:/chainofcomponents/progValue_i
   […]
   add wave -position end sim:/chainofcomponents/resultValid_o
   add wave -position end sim:/chainofcomponents/resultValue_o
   add wave -position end sim:/chainofcomponents/thisReady_o
2. Modifica dei segnali di assegnazione dei pesi:
   wave modify -driver freeze -pattern constant -value 1 -starttime 125ns -endtime 350ns Edit:/chainofcomponents/progValid_i
   wave modify -driver freeze -pattern constant -value 000000000000000000000001 -range 25 0 -starttime 150ns -endtime 175ns Edit:/chainofcomponents/progValue_i […]
3. Modifica dei segnali di assegnazione del dato (immagine da elaborare):
   wave modify -driver freeze -pattern constant -value 11111111 -range 7 0 -starttime 477900ns -endtime 477925ns Edit:/chainofcomponents/dataValue_i
   […]

```
wave modify -driver freeze -pattern constant -value 1 -starttime 477900ns -endtime 580300ns Edit:/chainofcomponents/dataValid_i
[...]
wave modify -driver freeze -pattern constant -value 1 -starttime 580275ns -endtime 580300ns Edit:/chainofcomponents/endOfFrame_i
```

Come precedentemente descritto, la verifica funzionale di *ChainOfComponents* consta di due simulazioni: una simulazione ModelSim (appena descritta) e una seconda simulazione Matlab. Quest'ultima, utilizzata come confronto e valutazione della correttezza del risultato, permette la verifica dell'output prodotto attraverso l'applicativo ModelSim.

Si riportano anche in questo caso degli estratti dal codice (in questo caso codice Matlab), con esplicazioni dell'elaborazione eseguita:

1. Caricamento dei pesi del filtro, dei neuroni e dell'immagine da elaborare (gestita con politica a zero-padding):

```
load neuron1.mat
A1=A;
[...]
FILT = [0 1 0; 1 -4 1; 0 1 0];
IMM=zeros(66, 66);
IMM(2:65, 2:65)=IMM1; % zero padding
```

2. Equivalente software dei blocchi *Kernel* e *Filter*:

```
for I=2:65
   for J=2:65
      IMM_AF(I-1,J-1)=sum(sum(FILT.*IMM(I-1:I+1,J-1:J+1)));   %immagine after-filtering
      if IMM_AF(I-1,J-1)<0 % reLu
         IMM_AF(I-1,J-1)=0;
      end
   end
end
```

3. Equivalente software del blocco *Maxpooling*:

```
for I=1:16
   for J=1:16
      IMM_AM(I,J)=max(max(IMM_AF((I-1)*4+1:I*4,(J-1)*4+1:J*4))); % immagine after maxpooling
   end
end
```

4. Equivalente software degli stadi fully-connected (blocco *NetworkOfNeurons*):

```
IMM_AM=reshape(IMM_AM',1,256);
IMM_AM=IMM_AM';
J=1;
for I=1:8
   IMM_A1F(J)=A1((I-1)*256+1:I*256)*IMM_AM; % immagine after-first-fully-connected
   if IMM_A1F(J)<0
      IMM_A1F(J)=0;
   end
   J=J+1;
end

[...]
```

In conclusione, essendo identici i risultati ottenuti dalle due simulazioni, si può affermare che, a livello funzionale, il circuito risulta essere perfettamente funzionante.

4.2. Sintesi e studio delle performance

La sintesi e lo studio delle performance di *ChainOfComponents* si attuano attraverso la seguente funzionalità di Quartus: *Processing – Start Compilation*. La compilazione viene eseguita nell'ipotesi di utilizzare l'Intel Cyclone V E FPGA development kit (in particolare un device 5CGXFC7C6U19A7 caratterizzato da 56˙480 ALMs, 268 IOB, 156 DSP e un'alimentazione a 1.1V).

Attraverso questo studio si ottengono informazioni riguardanti i tempi di programmazione ed elaborazione, la massima frequenza di clock, il numero di blocchi ALM (adaptive logic module, ovvero i basic building block dell'implementato) piazzati e il numero di funzioni implementate attraverso ALUT (adaptive look up table).

Per quanto riguarda le durate di programmazione e di elaborazione di un'immagine 64x64, si sono ottenuti i seguenti risultati:

1. Durata della programmazione (con un solo dato di programmazione consegnato per ogni ciclo di clock): circa 0.5 ms;
2. Durata dell'elaborazione della singola immagine: circa 0.13 ms; con però ottimizzazione del tempo di processing dalla seconda immagine elaborata (pari al 40%) grazie alla struttura a pipeline di *ChainOfComponents*, che non attende la conclusione della precedente elaborazione per iniziare il processing successivo.

Si deve notare però come queste misure dei tempi di programmazione ed elaborazione siano dipendenti dalla dimensione dell'immagine elaborata (in questo caso 64x64 pixel).

L'analisi della massima frequenza di clock ricerca all'interno del circuito programmato il percorso più critico (in questo caso tra ingresso e uscita del blocco *Max* del *Maxpooling*) e calcola tempi di setup e hold (tempi minimi per i quali l'input deve risultare stabile prima e dopo il fronte del clock), tempi di riposta dei registi e tempo di percorrenza del percorso più critico andando di conseguenza a calcolare il minimo periodo di clock, il cui reciproco rappresenta la massima frequenza di funzionamento (f_{CK}^{max}). Si sono così ottenuti i seguenti tempi di hold e setup (riportati nella Tab.4.2.a.).

Condizioni operative	Tempo di hold	Tempo di setup
fast-1.1V model a 85°C	0.98 ns	6.649 ns
fast-1.1V model a 0°C	1.028 ns	5.975 ns

Tab.4.2.a.: Tempi di hold e setup ottenuti in due diverse condizioni operative (al variare della temperatura)

La massima frequenza di clock così ritrovata risulta essere pari a f_{CK}^{max} = 67.94 MHz, il che equivale a un periodo di clock pari a T_{CK}^{min} = 14.71 ns. La precedente scelta di avere un periodo di clock pari a 25 ns (anche se scelta solamente formale) risulta perciò essere una scelta in linea coi risultati ottenuti.

L'analisi dell'occupazione d'area, espressa in numero di blocchi base ALM, ha portato al seguente risultato: in *ChainOfComponents* vengono utilizzati 45˙707 ALMs di cui 43˙222 utilizzati come registri.

Infine il numero di combinational ALUT impiegato nel circuito risulta essere pari a 20˙749 ALUTs. Nello specifico, nella tabella Tab.4.2.b. è riportato il numero di funzioni ad almeno tre ingressi utilizzate in *ChainOfComponents*.

Numero di ingressi della funzione	Numero di funzioni utilizzate
Funzione a 3 ingressi	2567
Funzione a 4 ingressi	972
Funzione a 5 ingressi	613
Funzione a 6 ingressi	667
Funzione a 7 ingressi	1

Tab.4.2.b.: Riassunto delle funzioni implementate da ALUT (specificando il numero di ingressi)

5. CONCLUSIONI E PROSPETTIVE FUTURE

5.1. Conclusioni

In base ai risultati raccolti nel capitolo 4 si può concludere che *ChainOfComponents* non solo risulta funzionante a livello logico-funzionale, ma permette di avere risultati ed elaborazioni in ottimi tempi di processing (elaborazione di una piccola immagine 64x64 in 0.052 ms).

Avendo ottime prestazioni temporali, questo circuito può essere perciò impiegato per un'elaborazione in tempo reale di immagini. In base alla sua struttura, all'elaborazione eseguita e a queste proprietà, una sua possibile applicazione potrebbe essere la ricerca di features e informazioni da un'acquisizione e lettura di codici a barre di prodotti presenti su un nastro trasportatore in un processo produttivo altamente automatizzato (questo infatti è il contesto alla base del quale si è implementato il coprocessore neurale *ChainOfComponents*).

In conclusione, *ChainOfComponents* risulta rispettare tutte le specifiche date nel caso specifico in esame (si veda la tabella Tab.3.1.a.); naturalmente però esistono diverse ottimizzazioni e sviluppi futuri applicabili al coprocessore neurale qui presentato, come specificato nei seguenti sottocapitoli dal 5.2. al 5.5.

5.2. Statistica dei segnali elaborati per scegliere il numero di bit ottimo

Una prima ottimizzazione applicabile a *ChainOfComponents* consiste in uno studio della statistica dei segnali elaborati con lo scopo di scegliere ottimi dimensionamenti dei dati presenti in tutta la sua struttura. Questo perché, come si è descritto nel capitolo 3, in molti layer del circuito si sono adottate politiche di riduzione della dimensione del dato, ad esempio attraverso *saturate policy* e troncamento per divisione di potenze del 2. Ognuna di queste però è stata scelta sulla base di pochi esempi esplicativi e simulazioni locali sul singolo layer, non perciò attraverso un approfondito studio della dimensione del dato e della sua statistica nell'attraversare i diversi stadi del coprocessore. Un primo ed importante approfondimento a questo progetto consiste perciò in questa analisi dettagliata che porterà a una ottimizzazione della dimensione dei dati interni (*NEW_DATA_WIDTH*, *DATA_AFTER_FILTERING_WIDTH*, *PS_DATA_WIDTH*, *NEW_DATA_WIDTH2*) e del risultato finale (*RESULT_WIDTH*).

5.3. Troncamento di bit e saturate da programmazione

Una seconda importante ottimizzazione di *ChainOfComponents* consta in uno studio approfondito sull'impatto delle diverse tecniche di troncamento. Come descritto nel punto 5.2., anche la scelta della tecnica di riduzione della dimensione del dato è stata dettata da pochi esempi di elaborazione e considerazioni locali, senza un approfondito studio sull'effettivo impatto del troncamento (dei MSB), della politica del saturate o della riduzione attraverso divisione di potenze del 2 (troncamento dei LSB).
Un futuro possibile studio in merito porterà perciò alla selezione del metodo ottimo per ogni singolo layer del circuito.

5.4. Implementazione completamente parametrica

Un terzo e importante possibile miglioramento di *ChainOfComponents* consiste in una sua implementazione completamente parametrica. La maggior parte dei suoi layer infatti può gestire (già nella formulazione attuale) una varietà di dimensioni del dato, della maschera del filtraggio e della numerosità dei bit dei parametri interni molto estesa. Però non tutti i layer risultano essere così plastici e capaci di gestire diverse combinazioni di *generic* settati dal progettista; in particolar modo risultano critici i blocchi *Maxpooling* e *NetworkOfNeurons* che presentano alcune ipotesi sulle dimensioni dei dati consegnati:

1. Il blocco *Maxpooling* può gestire qualunque dimensione dell'immagine consegnata in ingresso se *POOL_DIM* viene settato a 2; in caso in cui quest'ultimo sia settato a 3 o 4, viene richiesto come ipotesi che l'altezza e la base dell'immagine siano divisibili per 3 o 4.
2. *NetworkOfNeurons* rappresenta un layer estremamente plastico e capace di elaborare qualunque dimensione dei dati in ingresso; presenta però un'unica limitazione nel suo funzionamento, ovvero l'incapacità di elaborare un dato a dimensione variabile all'interno di una sua singola programmazione.

Il superamento di questi due limiti, ampiamente trascurabili nell'esempio preso in esame in questo progetto, può perciò rappresentare un sostanziale miglioramento della struttura di *ChainOfComponents*.

5.5. Interfaccia a un livello di astrazione superiore (Matlab)

La quarta e più importate ottimizzazione di *ChainOfComponents* consta nell'implementazione di un'interfaccia a livello di astrazione superiore, ad esempio basata su Matlab, che permetta un piazzamento facile e ottimizzato del componente e un utilizzo dello stesso attraverso un'automatica selezione dei parametri interni. Attraverso quest'interfaccia, in base alle specifiche date, sarà possibile ottenere un componente ottimizzato (sulla dimensione dei dati interni, sulla gestione del troncamento e sulla varietà di selettori possibili) capace di elaborare le immagini in ingresso nel minor tempo possibile e con una qualità del dato e dei controlli di affidabilità molto elevata.

Lo scopo di quest'interfaccia sarà perciò l'ottimizzazione della qualità dell'output, basata su algoritmi ad alto livello (scritti ad esempio in Matlab) che attraverso specifici selettori (basati sulle specifiche date dal cliente) potranno piazzare solo specifiche porzioni del circuito, le stesse porzioni che ottimizzano il calcolo, l'informazione contenuta nelle features finali e la massima velocità di elaborazione (o il suo duale nel trade-off area-velocità, ovvero la minima occupazione d'area).

BIBLIOGRAFIA

1. Sara Colantonio, Ovidio Salvetti, ***Categorizzazione automatica di immagini mediante algoritmi neurali***, articolo interno – ISTI, Istituto di scienza e tecnologie dell'informazione "Alessandro Faedo" (2003)

2. Marcel van Gerven, Sander Bohte, ***Artificial Neural Networks as Models of Neural Information Processing***, Frontiers in Computational Neuroscience 11 (2017) article 114

3. Adam H. Marblestone, Greg Wayne, Konrad P. Kording, ***Toward an Integration of Deep Learning and Neuroscience***, Frontiers in Computational Neuroscience 11 (2017) article 94

4. Marcel van Gerven, ***Computational Foundations of Natural Intelligence***, Frontiers in Computational Neuroscience 11 (2017) article 112

5. Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, ***Efficient Processing of Deep Neural Networks: A Tutorial and Survey***, Proceedings of the IEEE 105 (2017) 2295 – 2329

6. G. E. Hinton, R. R. Salakhutdinov, ***Reducing the Dimensionality of Data with Neural Networks***, Science 313 (2006) 504-507

7. Tatt Hee Oong, Nor Ashidi Mat Isa, ***Adaptive Artificial Neural Networks for Pattern Classification***, IEEE Transactions on Neural Networks 22 (2011) 1823-1836

8. Barry Lennox, Gary A. Montague, Andy M. Frith, Chris Gent, Vic Bevan, ***Industrial application of neural networks - an investigation***, Journal of Process Control 11 (2001) 497-507

9. Jürgen Schmidhuber, ***Deep learning in neural networks: An overview***, Neural Networks 61 (2015) 85-117

10. Dominik Scherer, Andreas Müller, Sven Behnke, ***Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition***, 20th International Conference on Artificial Neural Networks (ICANN) 3 (2010) 92-101

11. Shin Hoo-Chang, Holger R. Roth, Mingchen Gao, Le Lu, Ziyue Xu, Isabella Nogues, Jianhua Yao, Daniel Mollura, Ronald M. Summers, ***Deep***

Convolutional Neural Networks for Computer-Aided Detection: CNN Architectures, Dataset Characteristics and Transfer Learning, IEEE Transactions on Medical Imaging 35 (2016) 1285-1298

12. Patrice Y. Simard, Dave Steinkraus, John C. Platt, ***Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis***, 7th International Conference on Document Analysis and Recognition 3 (2003) 958-962

13. Dan Cireşan, Ueli Meier, Jurgen Schmidhuber, ***Multi-column Deep Neural Networks for Image Classification***, IEEE Conference on Computer Vision and Pattern Recognition 12 (2012) 3642-3649

14. Roland Petrasch, Roman Hentschke, ***Process modeling for Industry 4.0 applications: Towards an Industry 4.0 process modeling language and method***, 13th International Joint Conference on Computer Science and Software Engineering (JCSSE) (2016) 1-5

15. Michael Rüßmann, Markus Lorenz, Philipp Gerbert, Manuela Waldner, Jan Justus, Pascal Engel, and Michael Harnisch, ***Industry 4.0: The Future of Productivity and Growth in Manufacturing Industries***, Boston Consulting Group 9 (2015)

16. Amara Amara, Frédéric Amiel, Thomas Ea, ***FPGA vs. ASIC for low power applications***, Microelectronics Journal 37 (2006) 669-677