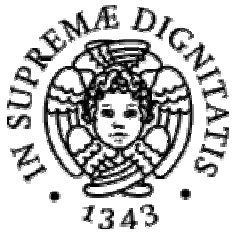


Università di Pisa

Facoltà di Ingegneria

Corso di Laurea Specialistica in Ingegneria Elettronica

Indirizzo Sistemi Elettronici

Anno Accademico 2005 – 2006

Tesi di Laurea

STUDIO E PROGETTAZIONE VLSI DI UN ACCELERATORE HARDWARE PER LA RICOSTRUZIONE DI AMBIENTI SONORI

Relatori:

prof. ing. Luca Fanucci

ing. Nicola E. L'Insalata

Candidato:

Tommaso Cecchini

RINGRAZIAMENTI

La mia famiglia, Emi, la chitarra, la Music Academy, il mio gruppo, i miei amici, i miei compagni di corso (soprattutto Andrea & Iacopo senza i quali non avrei mai resistito qui a ingegneria!), tutti quelli del Laboratorio di Sistemi Elettronici (soprattutto il grande Nko), il prof. Fanucci, i chitarristi che continuamente mi ispirano, la musica, le Harley Davidson, il compianto Dime e tutti i supereroi...

INTRODUZIONE

I recenti sviluppi dei sistemi di misura nell'ambito della caratterizzazione acustica degli ambienti consentono una descrizione sempre più accurata delle proprietà acustiche degli spazi per mezzo della risposta impulsiva.
La creazione di un *database* contenente questo tipo di informazioni riveste un ruolo estremamente importante sia dal punto di vista della conservazione dei beni culturali, sia dal punto di vista della produzione audio cinematografica e musicale.
L'utilizzo della risposta impulsiva, infatti, permette la ricostruzione sonora del contesto acustico, sia in fase di registrazione che di riproduzione, a partire da segnali pseudo-anecoici.
L'elaborazione di questi dati risulta estremamente onerosa dal punto di vista computazionale a causa delle lunghezze delle risposte impulsive, che possono raggiungere durate dell'ordine delle decine di secondi.
In questo lavoro di tesi viene affrontato lo studio e la progettazione di un acceleratore *hardware* per la convoluzione tra un segnale audio pseudo-anecoico ed una risposta impulsiva di un ambiente sonoro. L'obiettivo è la realizzazione di una piattaforma dedicata per applicazioni audio professionali. I criteri di progettazione seguiti sono stati scelti in base alla necessità di provvedere ad un compromesso tra prestazioni, intese come capacità di calcolo aggiuntiva senza la quale sarebbe impossibile il

processamento dei dati, e flessibilità, intesa come possibilità di programmazione dell'intero sistema per far fronte alle varie necessità di dimensionamento relative alle molteplici situazioni di impiego (lunghezza della risposta impulsiva, frequenza di campionamento, numero di vettori processabili in sequenza continua…).

Dopo aver preso visione dello stato dell'arte, si è proceduto ad un'analisi di alto livello del sistema per determinare le specifiche di massima. In seguito, il sistema è stato partizionato nelle sue componenti *hardware* e *software* e si è operata la scelta della piattaforma di prototipazione. Il sottosistema di convoluzione *hardware* è stato quindi descritto in linguaggio VHDL e verificato funzionalmente. Lo stesso codice è stato sintetizzato e verificato su dispositivi FPGA, assieme alle interfacce appositamente sviluppate per la comunicazione verso il sottosistema *software*.

Nel **capitolo 1** andremo ad esporre le modalità di misura della risposta impulsiva di un ambiente, le caratteristiche del formato *wave* e lo stato dell'arte delle soluzioni *software* e *hardware* per la convoluzione.

Nel **capitolo 2** dopo uno studio degli algoritmi di convoluzione al quale segue la scelta del più idoneo ad un'implementazione *hardware*, sarà effettuata un'analisi ad alto livello del sistema convolutore andandone ad individuare le specifiche, l'architettura di massima ed i blocchi principali.

Nel **capitolo 3** sarà trattato il processore FFT/IFFT in tutte le sue caratteristiche, a partire dalla descrizione dell'algoritmo implementato. Saranno analizzati i singoli blocchi costituenti l'architettura, realizzati sulla base dell'algoritmo stesso.

Nel **capitolo 4** analizzeremo i restanti blocchi costituenti il *core* di convoluzione partendo dalla descrizione di un singolo ramo fino ad arrivare all'intera catena di moltiplicazione – accumulo.

Nel **capitolo 5** sarà introdotta la piattaforma di prototipazione nelle sue caratteristiche fondamentali, complete di un'analisi dei blocchi di memoria, dei moduli FPGA e delle interfacce disponibili, alla quale seguirà una descrizione dei blocchi di gestione della comunicazione attraverso il *bus*

PCI, appositamente realizzati per l'applicazione, con un’analisi dei protocolli per il trasferimento dei dati.

Nel **capitolo 6** saranno riportate le modifiche per l’ottimizzazione mirata all’implementazione su dispositivi FPGA apportate al processore FFT, la comparazione con altri IP esistenti ed i risultati delle sintesi dei vari blocchi architetturali, per presentare infine l’analisi dei dati ottenuti relativamente al sistema nel suo complesso, completi dei vincoli di *timing* rilevati dal sintetizzatore logico.

CAPITOLO 1
CARATTERIZZAZIONE ACUSTICA DEGLI AMBIENTI

La propagazione del campo acustico in un ambiente è un fenomeno fisico caratterizzato da equazioni differenziali lineari. Questa proprietà permette di studiare l'acustica degli ambienti utilizzando la teoria dei sistemi lineari; in particolare è possibile caratterizzare completamente un ambiente mediante la misura della sua risposta impulsiva.
Dall'analisi della risposta impulsiva vengono definiti dei parametri, sia qualitativi che quantitativi, che sintetizzano le proprietà acustiche dell'ambiente in esame [Bonsi et al, 2005].
Alcuni parametri qualitativi sono ad esempio:

- *Chiarezza*: grado in cui i suoni appaiono disgiunti.

- *Uniformità del suono*: omogeneità di comportamento al variare della posizione.

- *Coesione / affinità*: esecuzione all'unisono dovuto alla possibilità di ascolto reciproco.

- *Spazialità / ampiezza apparente della sorgente*: impressione di provenienza da una sorgente allargata.

- *Riverberazione / vivezza*: persistenza del suono dopo l'interruzione della sorgente (si parla di sale "vive" o "secche"). Importante è la riverberazione alle medie e alte frequenze (> 350 Hz).

- *Intimità / presenza*: percezione delle dimensioni spaziali della sala.

Alcuni esempi di parametri quantitativi, ricavabili mediante manipolazione della risposta impulsiva, sono invece:

- *Intelligibilità e chiarezza*: relativi rispettivamente al parlato ed alla musica sono ricavati dallo studio della distribuzione temporale dell'energia durante il decadimento.

- *Initial Time Delay Gap (ITDG)*: intervallo di tempo (in ms) che separa il suono diretto dalla prima riflessione.

- T_{60}: tempo necessario affinché l'intensità si riduca di 60 dB rispetto al valore iniziale.

I due problemi principali di questo tipo di processamento risultano la misura della risposta impulsiva e la gestione dei dati ottenuti. Argomento di questa tesi è la convoluzione della risposta impulsiva di un ambiente con un segnale audio pseudo-anecoico (ossia privo di effetti derivati dall'ambiente circostante), per poter imprimere le proprietà acustiche desiderate. Non entreremo quindi nel merito delle informazioni

caratterizzanti l'ambiente ricavabili dall'analisi risposta impulsiva ma ci occuperemo principalmente dell'impiego dei dati ottenuti, con un breve accenno alla procedura di misurazione.

1.1 Cenni sul sistema di misura della risposta Impulsiva

1.1.1 Acquisizione dei dati

La fase di misura e la fase di *processing* risultano tra loro ben distinte e possono essere svolte in ambiti sia temporali che spaziali differenti.
Per quanto riguarda la misura, occorre eccitare l'ambiente in maniera omogenea, cioè uguale in tutte le direzioni spaziali, ed uniformemente in frequenza. A tal fine si utilizza una cassa acustica di forma dodecaedrica (v. figura 1.1), con risposta in frequenza pressoché piatta in tutta la banda audio.
Alla cassa audio è collegato un generatore di segnale che produce *sweep* sinusoidali oppure segnali MLS (*Maximum Lenght Sequence*) che sono assimilabili a sorgenti di rumore bianco.

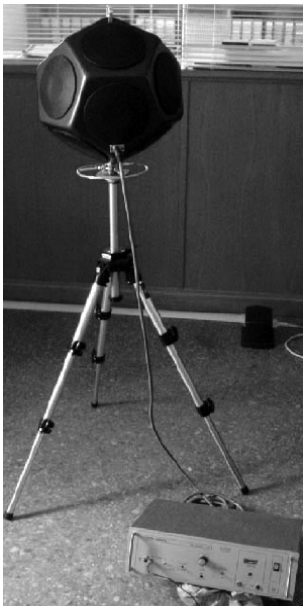


Figura 1.1: La sorgente sonora dodecaedrica.

Il comportamento acustico dell'ambiente è misurato da una particolare sonda, collegata ad una scheda di acquisizione audio (figura 1.2). Attraverso opportune operazioni di *post-processing* (decorrelazione) otterremo la IR relativa rispetto ad un punto ben preciso dell'ambiente.

Figura 1.2: Un'interfaccia audio firewire.

In figura 1.3 è mostrato un esempio di sistema di misura della risposta impulsiva di un ambente.

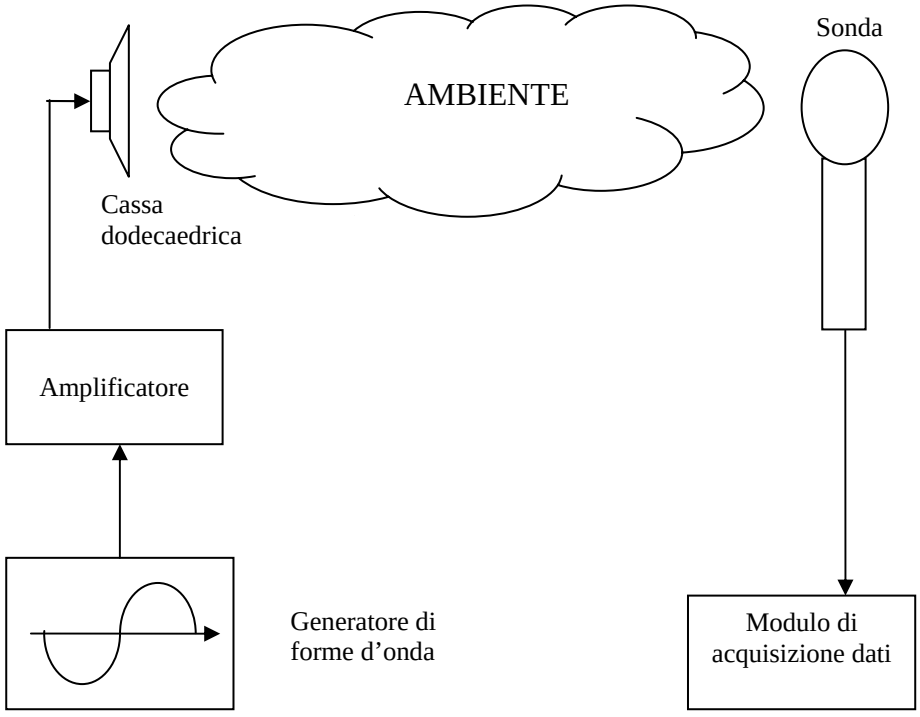


Figura 1.3: Schema dell'apparato necessario alla misura della risposta impulsiva di un ambiente.

1.1.2 Sensori per la misura

Per quanto riguarda le sonde per la rilevazione degli effetti mostrati da un ambiente relativamente all'eccitazione della sorgente, possiamo individuarne tre tipi principali:

- *Sonde in tecnologia p-p*: sono basate su due sensori di pressione per ogni componente spaziale, attraverso i quali si risale alle componenti della velocità mediante un'operazione di integrazione nel tempo del gradiente di pressione. Questo tipo di sonda non è adatta alle applicazioni audio a causa della banda estremamente limitata, l'alto numero di canali coinvolti, gli errori sistematici causati dall'incongruenza di fase tra i canali stessi e l'eccessiva approssimazione compiuta.

- *Microfono Soundfield* [SOU, 2004]: consiste in 4 microfoni di cui uno omni-direzionale e tre bidirezionali orientati secondo gli assi cartesiani, formanti così un tetraedro. Combinando linearmente i quattro gradienti di pressione ottenuti, attraverso un'opportuna matrice, si ottiene una particolare codifica detta "*B-Format*". In un ambiente anecoico questa sonda si rivela affidabile ma non si dimostra altrettanto idonea in un ambiente reale poiché utilizza l'approssimazione lineare dell'equazione di Eulero per ricavare le informazioni di velocità a partire da quelle di pressione e necessita sia di perfetta coincidenza spaziale tra le capsule che di assenza di ogni tipo di effetto di diffrazione dovuto alle loro dimensioni finite.

- *Sonda Microflown* [MIC, 2004]: è costituita da tre sensori di velocità a filo caldo posti secondo le direzioni dei tre assi cartesiani ed un sensore centrale che rileva la pressione. A causa di fenomeni di inerzia termica la banda è piuttosto limitata. Attraverso una serie di

filtraggi successivi è possibile estendere la banda sino ad un massimo di 15 kHz.

In figura 1.4 riportiamo a titolo di esempio un'immagine della sonda *Microflown*.

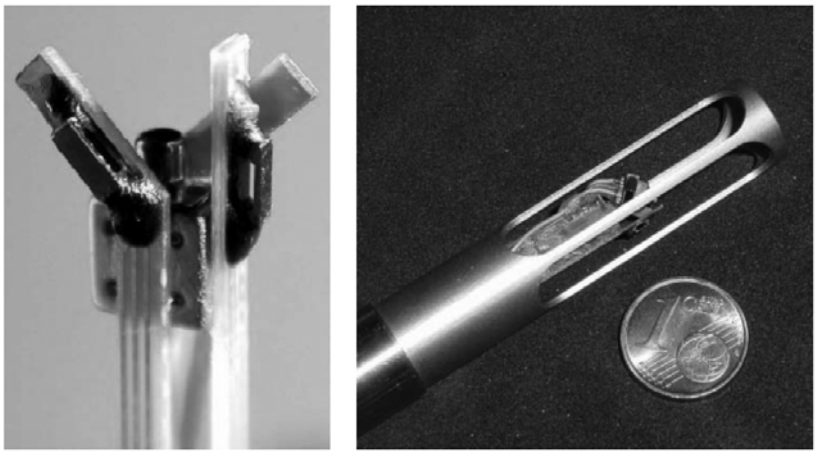


Figura 1.4: La sonda Microflown

1.1.3 Caratteristiche del formato *Wave*

Il fine ultimo delle sessioni di misura della risposta impulsiva è quello della creazione di un *database* informatico contenente tutte le informazioni acquisite, corrispondenti ciascuna ad un diverso ambiente, classificate in base ai valori degli indici sopracitati. In questo modo sarà possibile sia disporre in fase di *editing* di una libreria contenente i dati relativi ad un certo numero di ambienti distinti, classificati in base alle caratteristiche acustiche, sia mantenere nel tempo informazioni riguardanti le proprietà sonore di edifici storici.
Il formato *wave* è nato per l'immagazzinamento dei dati audio digitali su piattaforme *Windows*. Nell'ottica di una creazione di un "*database* acustico", è stato scelto questo formato in quanto:

- Consente la memorizzazione di dati in maniera "*lossless*" (nessun tipo di perdita).

- Supporta tracce audio multicanale.

- Supporta diverse frequenze di campionamento e diverse *bit-width*.

- L'*header* è estremamente flessibile e può contenere dati più o meno arbitrari.

In relazione all'ultimo punto, lo *standard* prevede l'inserimento di una qualsiasi stringa in codice ASCII preceduta dal numero di *bytes* che andrà ad occupare, come mostrato in figura 1.5.

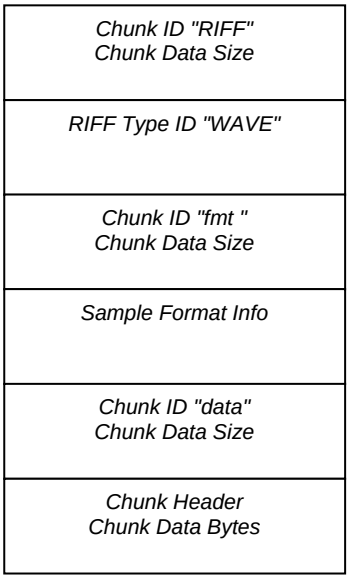


Figura 1.5: Schema della struttura di un file wave.

I *file wave* utilizzano lo *standard* RIFF che raggruppa i diversi contenuti (formato di campionamento, etichette, cursori…), in blocchi separati detti *chunk*. Ciascun *chunk* riporta un'intestazione e la dimensione del *chunk* stesso, intesa a multipli di 2 *bytes*.

Il *chunk* più interessante è quello denominato "*ltxt*", che permette di memorizzare un testo; nel nostro caso esso potrebbe contenere, ad esempio, una descrizione dell'ambiente sonoro cui corrisponde la risposta impulsiva e degli indicatori quantitativi che caratterizzano la IR. La struttura è molto semplice (v. figura 1.6): occorre specificare l'ID "*ltxt*" e la dimensione in *words*. Nella sezione indicata in figura con *"Text"*, possiamo inserire un qualsiasi tipo di informazione. Il resto dei campi sono opzionali e servono a specificare se ci si riferisce ad un istante preciso del file, qual è la lingua in cui è scritto il testo, l'eventuale dialetto…

Offset	**Dimensione**	**Descrizione**	**Valore (hex)**
0x00	4	ID del chunk	"ltxt" (0x6C747874)
0x04	4	Dimensioni del *chunk*	Dipende dal testo
0x08	4	ID del Cue Point	0 - 0xFFFFFFFF
0x0c	4	Lunghezza campioni	0 - 0xFFFFFFFF
0x10	4	ID dello scopo	0 - 0xFFFFFFFF
0x12	2	Paese	0 - 0xFFFF
0x14	2	Lingua	0 - 0xFFFF
0x16	2	Dialetto	0 - 0xFFFF
0x18	2	*Code Page*	0 - 0xFFFF
0x1A	Text		

Figura 1.6: Il chunk "ltxt".

La gestione *software* di queste funzionalità risulta estremamente semplice, grazie all'esistenza di librerie compilate per diverse piattaforme, dedicate alla lettura ed alla scrittura di file contenenti suoni campionati, come ad esempio la libreria *libsndfile* [LIB, 2006].

1.2 Lo stato dell'arte

1.2.1 Soluzioni *software*

Con il termine soluzione *software*, ci riferiamo a programmi per PC, o *plug-in* per programmi di *editing* audio.
Le prestazioni di queste soluzioni, dunque, sono dipendenti dalle caratteristiche del PC all'interno del quale sono installate. Sono molto diffuse a causa del loro costo estremamente contenuto e della loro totale compatibilità ed integrazione con gli ambienti per PC. Alcuni esempi sono:

- *Acoustic Mirror* [SON, 1997]: convolutore stereo per *Sony Sound Forge* con supporto per la registrazione delle IR.

- *Aurora* [AUR, 2004]: *plug-in* di *Adobe Audition* con misurazione della IR sperimentale, creazione di filtri di inversione, reti per cancellazione del *cross-talk*, calcolo dei parametri acustici ISO 3382, *Speech Transmission Index* e *Active Speech Level*.

- *Voxengo Pristine Space* [VOX, 2004]: *plug-in* VST che supporta fino a 8 canali indipendenti, dotato di un convertitore della frequenza di campionamento e di una modalità a bassa qualità per minimizzare l'utilizzo della CPU.

1.2.2 Soluzioni *hardware*

Analizzeremo due prodotti attualmente sul mercato: lo *Yamaha SREV1 Digital Sampling Reverb* [YAM, 2000] e la *workstation Huron* della *LakeDSP* [LAK, 2003].

Il primo è un prodotto *stand-alone* da studio, basato su un *set* di 64 ASIC di convoluzione da 16384 prese. Può eseguire convoluzioni con IR fino a 520000 campioni con frequenze di 44.1 e 48 kHz, sebbene anche gli ingressi a 44.1 siano poi internamente convertiti a 48 kHz.
I segnali di ingresso sono quantizzati su 24 *bit* mentre le aritmetiche interne sono su 32 *bit* per limitare le distorsioni introdotte dal processamento in virgola fissa. I quattro canali di I/O possono essere configurati indipendentemente, malgrado alcune combinazioni vadano a limitare la lunghezza massima della IR processabile.
La *workstation Huron* della *LakeDSP* è un PC a *rack* con un *bus custom* sul quale possono essere montate fino a sette schede acceleratrici, ciascuna contenente un DSP *Motorola/Freescale* 56002 con una potenza di calcolo di 33 MIPS. Questo sistema rappresenta una soluzione completa per il *processing* audio, supportando anche il *B-Format* che, come già accennato, è una particolare codifica delle informazioni acquisite.
Utilizza convolutori basati sull'algoritmo di *Gardner*, che verrà chiarito in seguito (v. capitolo 2). Con una frequenza di campionamento a 48 kHz, la *Huron* può gestire IR fino a 278244 campioni (circa sei secondi) con una latenza di dieci campioni. Queste prestazioni richiedono però quattro DSP per canale.
Il software *Huron* può lavorare con frequenza di campionamento pari a 96 kHz, anche se questa non è supportata dall'*hardware*. Gli ingressi a disposizione dell'utente infatti sono esclusivamente a 48 kHz, cosicché sarà necessaria un'unità esterna per lo *splitting* di un unico segnale audio a 96 kHZ in due segnali a 48 kHz, separando campioni pari e dispari.

CAPITOLO 2
STUDIO DELL'IMPLEMENTAZIONE

Questo lavoro di tesi ha come obiettivo l'implementazione di un prototipo *hardware/software* per la ricostruzione acustica degli spazi a partire dalle misurazioni della risposta impulsiva di ambienti sonori (v. capitolo 1).
Nei paragrafi successivi saranno descritti e confrontati i principali algoritmi di convoluzione al fine di scegliere il più conveniente per l'applicazione, così da poter procedere al partizionamento *hardware/software* e giungere ad un'architettura ad alto livello dell'acceleratore *hardware*.

2.1 Analisi ad alto livello del sistema

Il sistema dovrà essere in grado di gestire tutte le fasi riguardanti il processo di registrazione sia di una traccia mono anecoica che della risposta all'impulso di un ambiente, archiviandole in formato *wave*.
A tal fine il sistema deve generare la traccia di *test*, dando la possibilità all'utente di impostarne i parametri caratteristici, così da registrare la risposta ottenuta, applicando eventualmente dei filtri digitali.
Una volta completati questi compiti sarà necessario passare al calcolo della risposta all'impulso o all'analisi *off-line* di quelle presenti in archivio.

Come ultimo passo avremo la convoluzione tra i due segnali, con la possibilità di svolgere questa operazione anche in modalità *real-time* per quanto riguarda la traccia anecoica, per poter giungere infine all'archiviazione, sempre in formato *wave*, delle tracce ottenute.

.

2.2 Algoritmi di convoluzione

Si definisce convoluzione il processo matematico che mette in relazione l'uscita y(t) di un sistema lineare tempo-invariante con il suo ingresso x(t) e la sua risposta impulsiva h(t).
Per sistemi di questo tipo e sequenze discrete finite, l'operatore di convoluzione si esprime come:

$$y(n) = \sum_{k=0}^{M-1} h(k) \cdot x(n-k) \qquad (1)$$

Dove y(n) rappresenta l'uscita, x(n) l'ingresso e h(n) la risposta impulsiva del sistema.
Come si può osservare dalla relazione (1) il processo di convoluzione risulta molto oneroso in termini di numero di operazioni (moltiplicazioni).

2.2.1 La convoluzione nel dominio del tempo

La prima soluzione da analizzare è senz'altro la più immediata tra tutte le possibili, ovvero l'applicazione diretta dell'equazione (1) in un'architettura a linea di ritardo, nota come filtro FIR (*Finite Impulse Response*), mostrata in figura 2.1.

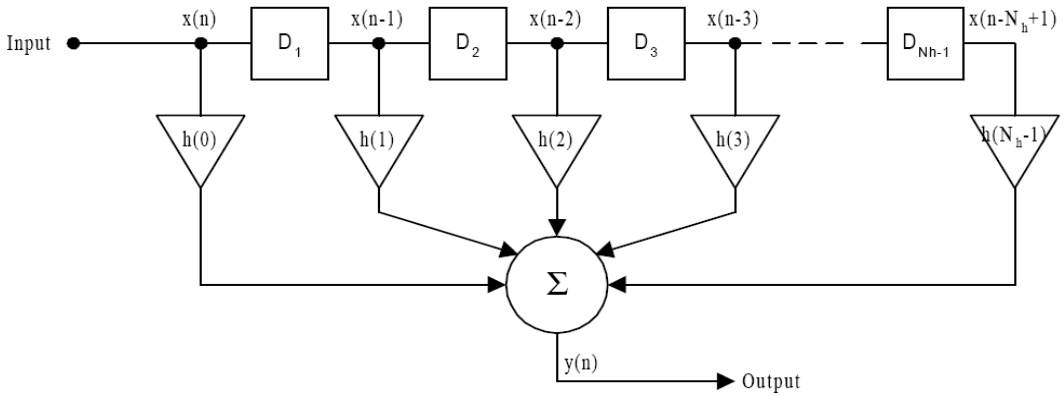


Figura 2.1: Architettura a linea di ritardo per l'implementazione del processo di convoluzione nella sua forma tempo-discreta per sequenze finite.

Detta N_h la lunghezza della risposta impulsiva, il costo per singolo campione dell'uscita è pari a $2\cdot(N_h - 1)$ operazioni di moltiplicazione e somma. Ad esempio, nel caso di una IR di tre secondi campionata a 48 kHz, per ottenere un'uscita di durata pari a tre secondi sono necessarie circa $2\cdot(3\cdot48000)^2 = 41{,}472\cdot10^9$ operazioni. Questa soluzione è dunque perseguibile solo in caso di risposte impulsive estremamente brevi.

2.2.2 La convoluzione nel dominio della frequenza

Un'altra possibilità per il calcolo della convoluzione è costituita dal passaggio al dominio della frequenza che riduce sensibilmente il numero delle operazioni necessarie, al prezzo però di un sensibile incremento della latenza dovuto all'indispensabile operazione di trasformazione discreta di Fourier (DFT). Per un numero abbastanza elevato di campioni da processare, otteniamo un notevole abbassamento dei costi attraverso l'impiego di algoritmi di FFT (*Fast Fourier Transform*) per i quali la crescita computazionale in funzione delle lunghezze in gioco risulta logaritmica anziché lineare [Proakis and Manolakis, 1996], tanto da rendere più conveniente operare nel dominio della frequenza anziché nel dominio del tempo.

Consideriamo una sequenza finita x(n) di lunghezza N_x che vada a stimolare un filtro FIR con risposta h(n) di lunghezza N_h. Senza perdere di generalità, possiamo scrivere:

$$x(n) = 0 \qquad \text{per } n < 0 \text{ e } n \geq N_x$$
$$h(n) = 0 \qquad \text{per } n < 0 \text{ e } n \geq N_h$$

Avremo, secondo la definizione, una sequenza in uscita pari a alla convoluzione di x(n) con h(n):

$$y(n) = \sum_{k=0}^{N_h-1} h(k) \cdot x(n-k) \qquad (2)$$

Otterremo una sequenza di lunghezza finita, poiché sono di durata finita entrambi i termini del prodotto di convoluzione e questa sarà pari ad $N_y=N_x+N_h-1$, cosicché avremo bisogno di un numero di campioni pari almeno ad N_y per rappresentare correttamente y(n) nel dominio della frequenza. Se risulta:

$$Y(k) \equiv Y(\omega)|_{\omega=2\pi k/N} = X(\omega)\cdot H(\omega)|_{\omega=2\pi k/N} \quad \text{per } k = 0,1, \ldots, N_y-1$$

Avremo allora:

$$Y(k) = X(k)\cdot H(k) \qquad \text{per } k = 0,1, \ldots, N_y-1$$

Se le sequenze x(n) ed h(n) hanno lunghezze inferiori ad N_y, basterà aggiungere a ciascuna di esse gli zeri necessari al raggiungimento della durata desiderata (*zero padding*). Questo incremento non modifica gli spettri dei due segnali che risultano continui nel caso di sequenze aperiodiche. Campionando comunque ad intervalli regolari in frequenza

per avere N_y punti, avremo solo aumentato il numero di campioni che rappresentano le sequenze nel dominio della frequenza.
In definitiva, aggiungendo gli zeri alle sequenze e convolvendo le due, possiamo ottenere y(n) per trasformazione inversa. In questo modo abbiamo un metodo applicabile ai processi di filtraggio FIR, del tutto equivalente alla convoluzione lineare.
Emerge infine l'importanza della prevenzione di effetti indesiderati di *aliasing* dovuti ad una scelta di una lunghezza insufficiente, cioè inferiore ad N_x+N_h-1. Nella pratica verrà operata una trasformata di dimensioni pari alla prima potenza di due maggiore o uguale ad N_y [Oppenheim and Schafer, 1975].
Riportiamo in figura 2.2 una schematizzazione a blocchi del flusso di calcolo.

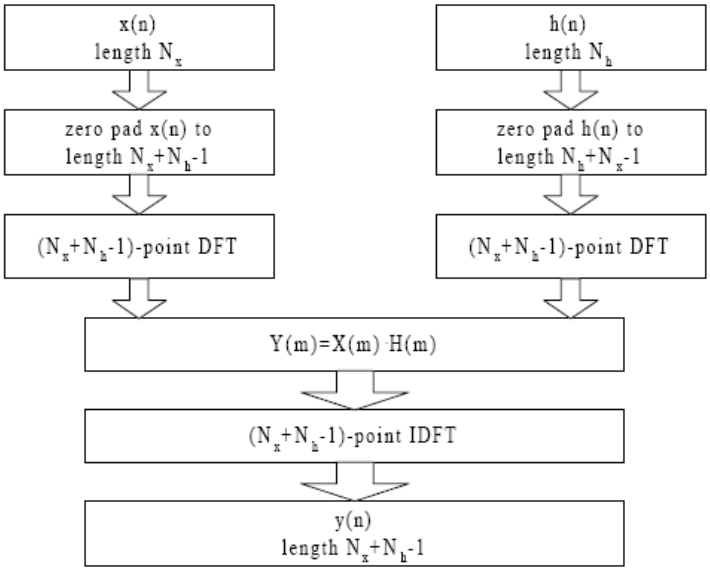


Figura 2.2: Schematizzazione dei passi necessari al processo di convoluzione operato come prodotto nel dominio della frequenza mediante utilizzo di FFT.

Questo metodo, come già esposto, è più efficiente del calcolo diretto nel dominio del tempo. Per un dato ingresso, il numero di operazioni richieste è circa pari a $2\cdot(N_x+N_h-1)\cdot\log_2(N_x+N_h-1)$ per le DFT, (N_x+N_h-1) per il prodotto ed altre $2\cdot(N_x+N_h-1)\cdot\log_2(N_x+N_h-1)$ per le IDFT [Proakis and

Manolakis, 1996]. Per un ingresso di tre secondi (campionato a 48 kHz come la risposta impulsiva) otteniamo:

48000*3+48000*3-1=287999 numero di campioni di x(n) e di h(n)
$287999 + 4 \cdot 287999 \cdot \log_2(287999) \approx 21{,}18 \cdot 10^6$

Abbiamo dunque una riduzione netta del numero di operazioni necessarie, costataci però in termini di latenza (che sarà almeno pari a $2 \cdot N_x$ [Proakis and Manolakis, 1996]). A causa delle operazioni di modifica della lunghezza, questa tecnica è estremamente svantaggiosa qualora la sequenza x(n) sia molto più lunga di h(n).
Adesso introdurremo una serie di altri metodi di calcolo del prodotto di convoluzione mirati al decremento della latenza, del numero di operazioni richieste e della possibilità di operare su segnali arbitrariamente lunghi, che fanno uso di processamenti ibridi e a blocchi, tenendo dunque conto anche delle dimensioni limitate delle memorie.

2.2.3 Metodo Overlap – Add

Andremo adesso a dimostrare che l'operazione di convoluzione può essere compiuta anche partizionando l'ingresso in più blocchi, attraverso opportuni accorgimenti.
Si consideri l'esempio di figura 2.3. La convoluzione di x(n) (figura 2.3 a) e h(n) (figure 2.3 b) è mostrata in figura 2.3 c. Da notare poi le figure 2.3 d-f che illustrano invece convoluzioni consecutive tra una serie di partizioni di x(n), ognuna con l'h(n) stessa.
Ciascuna convoluzione ha la lunghezza attesa di $N_{xi}+N_h-1$ così da non incorrere in problemi di *aliasing*, avendo aggiunto N_h-1 zeri al blocco in ingresso.
Nel caso specifico illustrato, le sequenze risultanti dalle tre convoluzioni dei blocchi hanno ciascuna una lunghezza pari a 5. La loro successione dà luogo dunque ad una sequenza di 15 campioni, anziché agli 11 del

caso tradizionale. Da un'ispezione visiva è logico assumere che ciascuna sequenza successiva corrispondente ai singoli blocchi, vada a sovrapporsi a quella che la segue per un numero pari a (15 -11) / 2 = 2 campioni, come mostrato in figura 2.3 g. Questo valore coincide con N_h-1. Se si procede alla somma dei campioni che si sovrappongono tra le sequenze (figura 2.3 h), a causa dell'aggiunta stessa degli zeri, otteniamo esattamente una sequenza identica a quella ottenuta applicando l'equazione (1), riportata, come già detto, in figura 2.3 c. Questo metodo prende il nome di *Overlap – Add*. La lunghezza delle partizioni di x(n) non deve essere necessariamente pari ad N_h ma, al fine di evitare lunghe convoluzioni, conviene scegliere una dimensione simile in termini di ordine di grandezza. Il numero di campioni sovrapposti rimane comunque pari ad N_h-1 [Smith, 1997].

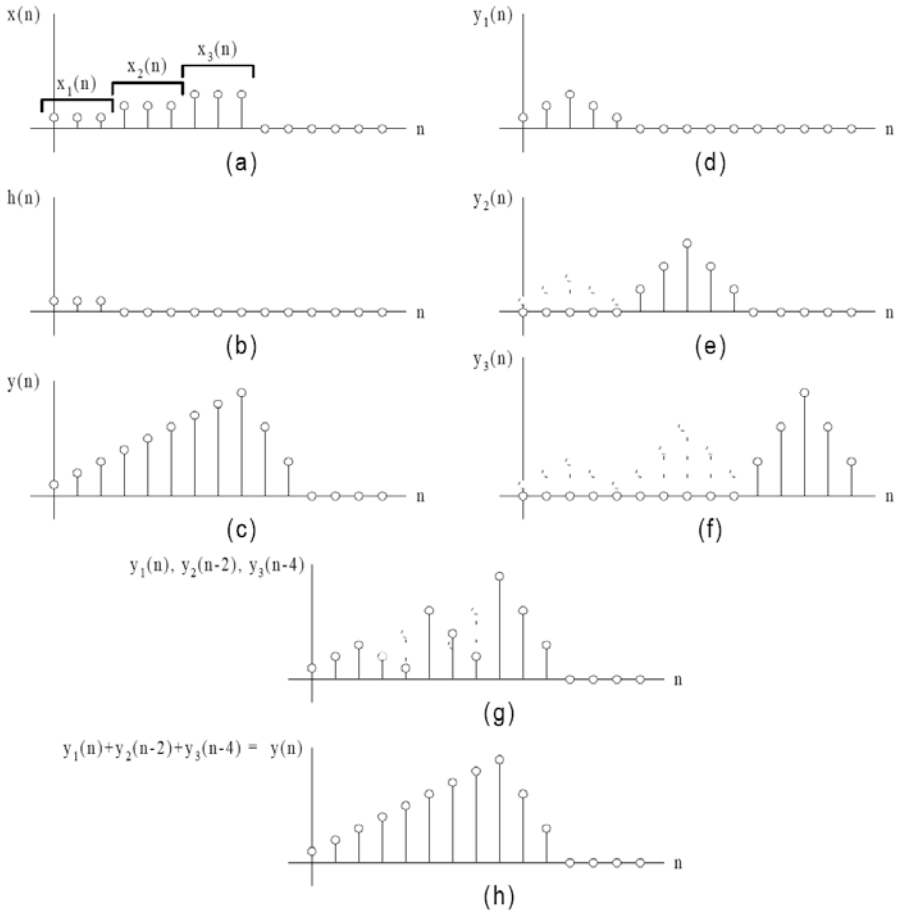


Figura 2.3: Metodo standard nel dominio del tempo (a-c) confrontato con il metodo Overlap – Add (d-h).

Matematicamente la situazione può essere descritta nel modo seguente:

$$x(n) = \sum_{i=0}^{+\infty} x_i(n) \qquad \text{con} \quad \begin{cases} x_i(n) = x(n) & \text{per} \quad i \cdot N_{block} \le n \le (i+1) \cdot N_{block} \\ x_i(n) = 0 & \text{altrove} \end{cases}$$

$$y(n) = h(n) * x(n) = \sum_{k=0}^{n} h(m) \cdot \sum_{i=0}^{+\infty} x_i(n-k) = \sum_{i=0}^{+\infty} h(n) \cdot x_i(n) = \sum_{i=0}^{+\infty} y_i(n)$$

Cioè, riassumendo quanto detto:

$$x_1(n) = \{x(0), x(1), \dots, x(N_{xi}-1), \underbrace{0, 0, \dots, 0}_{N_h\text{-}1 \text{ zeri}}\}$$

$$x_2(n) = \{x(N_{xi}), x(N_{xi}+1), \dots, x(2 \cdot N_{xi}-1), \underbrace{0, 0, \dots, 0}_{N_h\text{-}1 \text{ zeri}}\}$$

$$x_3(n) = \{x(2 \cdot N_{xi}), x(2 \cdot N_{xi}+1), \dots, x(3 \cdot N_{xi}-1), \underbrace{0, 0, \dots, 0}_{N_h\text{-}1 \text{ zeri}}\}$$

$$Y(n) = \{ y_1(0), y_1(1), \dots , y_1(N_{xi}-1), y_1(N_{xi}) + y_2(0), y_1(N_{xi}+1) + y_2(1), \dots , y_1(N_y-1) + y_2(N_h-1), y_2(N_h), \dots \}$$

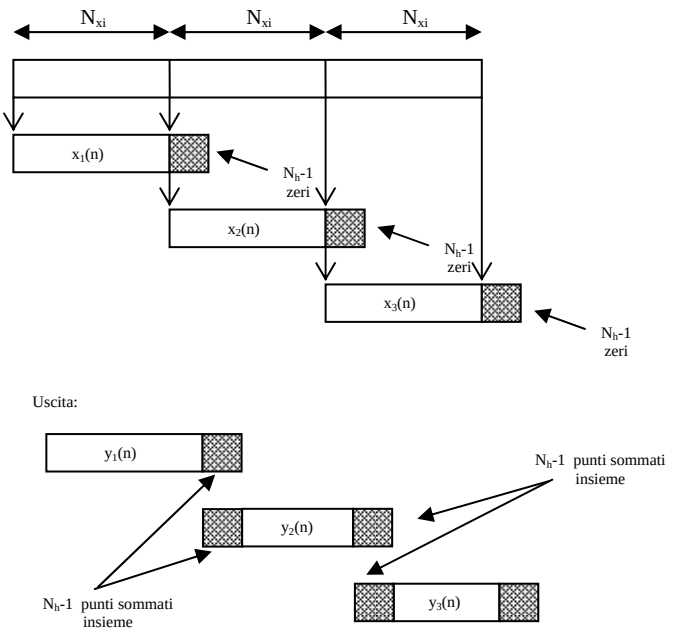


Figura 2.4: Ingressi e uscite del metodo Overlap – Add.

Avendo dimostrato precedentemente che operare nel dominio della frequenza anziché in quello del tempo risulta più conveniente e che sia possibile eseguire un partizionamento del segnale ottenendo il medesimo risultato, è del tutto ovvia la convenienza nell'applicare contemporaneamente entrambe le strategie.

2.2.4 Metodo Overlap – Save

Questo metodo necessita di sovrapposizione dei campioni all'interno dei blocchi di ingresso stessi. Le partizioni (che potranno essere o meno uniformi) risultano poi convolute con la risposta impulsiva completata con N_{xi}-1 zeri. A causa della ridondanza già presente in ingresso, che dovrà

essere pari almeno ad N_h-1 campioni (ogni blocco conterrà gli ultimi N_h-1 dati del precedente – tutti zeri nel caso del primo blocco – seguiti da N_{xi} nuovi dati), ciascuna sequenza di uscita sarà affetta da *aliasing*, per cui un medesimo numero di campioni dovranno essere semplicemente scartati da ciascuna sequenza ottenuta, come mostrato in figura 2.5. Per tenere conto della percentuale di sovrapposizione adottata occorre specificare il cosiddetto fattore di *overlap*.

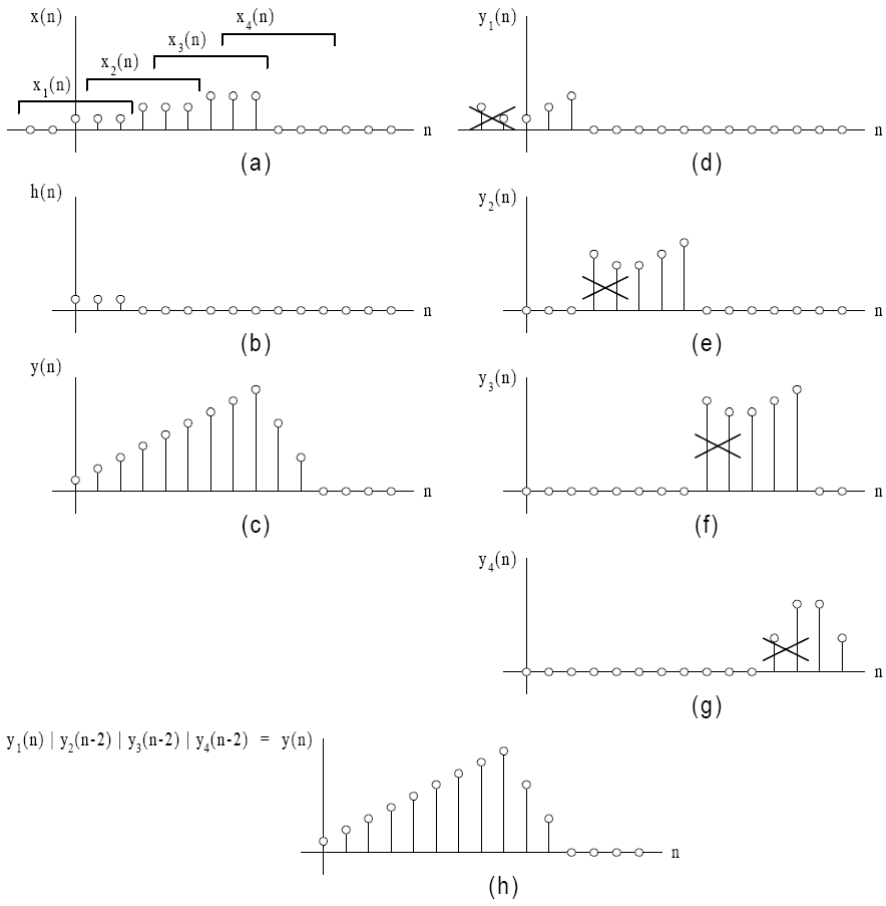


Figura 2.5: Metodo standard nel dominio del tempo (a-c) e metodo Overlap – Save (d-h).

Da notare che per l'applicazione di questo algoritmo occorre conoscere solamente h(n) e N_{xi} [Proakis and Manolakis, 1996], per cui possiamo pensare di calcolare a priori la trasformata della risposta impulsiva

completa degli zeri aggiunti H(m) e salvarla in una memoria prima di iniziare il processamento degli ingressi. Per quanto riguarda la scelta del numero di partizioni dell'ingresso, non esistono specifiche costrizioni ma, poiché ciascun blocco richiede $N_{block} \cdot \log_2(N_{block})$ operazioni per la FFT, N_{block} moltiplicazioni ed ulteriori $N_{block} \cdot \log_2(N_{block})$ operazioni per la IFFT, risulta senz'altro conveniente scegliere N_{block} dello stesso ordine di N_h [Oppenheim and Schafer, 1975].
Segue un'espressione matematica di quanto detto finora.

$x(n) = \sum_{i=0}^{+\infty} x_i(n) - x_{i-1}(m)$, tale che:

$x_i(n) = x(n)$ per $i \cdot N_{block} - (i+1) \cdot (N_h-1) \leq n \leq (i+1) \cdot N_{block} - (i+1) \cdot (N_h-1) - 1$

$x_i(n) = 0$ altrove

$x_{i-1}(m) = x_{i-1}(n)$ per $i \cdot N_{block} - (i+1) \cdot (N_h-1) \leq m \leq i \cdot N_{block} - i \cdot (N_h-1) - 1$

Si ha poi :

$y_i(n) = h(n) * x_i(n)$

per cui,

$y(n) = y_0(m) \mid y_1(m) \mid y_2(m) \mid \ldots$

dove | indica l'operazione di concatenazione ed m indicizza il campione "$N_{block} - (N_h - 1)$" – esimo di ciascun blocco.

Tutte queste operazioni possono essere schematizzate nel diagramma a blocchi di figura 2.6, dove sono stati scelti $2 \cdot N_h$ blocchi ciascuno di $2 \cdot N_h$ campioni. In uscita avremo così N_h campioni affetti da *aliasing* da scartare.

Per non utilizzare una simbologia estremamente pesante, possiamo scrivere:

$$h_i * x(n,N)$$

con cui intendiamo la convoluzione ad N punti tra l'ingresso x e la risposta impulsiva h_i, a partire dal campione n. L'uscita potrà essere così espressa, sempre sotto forma di concatenazione, come:

$$y(n) = h * x(0,N) \mid h * x(N,N) \mid \mid h * x(2N,N) \mid \ldots$$

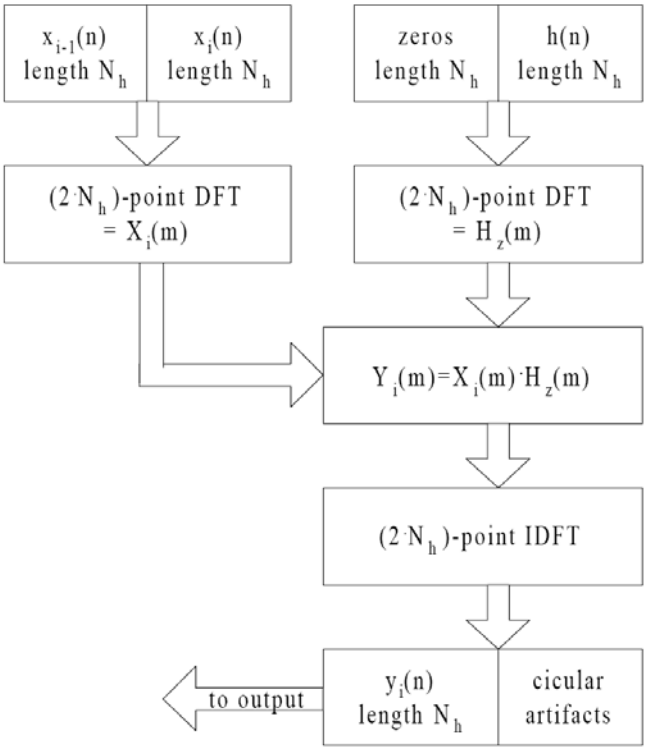


Figura 2.6: Schematizzazione del metodo Overlap – Save.

Andiamo a riassumere gli ingressi e le uscite con lo schema seguente e la figura 2.7.

$$x_1(n) = \{ \underbrace{0, 0, \dots, 0}_{N_h\text{-}1 \text{ punti}}, x(0), x(1), \dots, x(N_{xi} - 1) \}$$

$$x_2(n) = \{ \underbrace{x(N_{xi}–N_h+1), \dots, x(N_{xi} - 1)}_{N_h\text{-}1 \text{ punti da x1(n)}}, \underbrace{x(N_{xi}), \dots, x(2N_{xi}–1)}_{N_{xi} \text{ nuovi dati}} \}$$

$$x_3(n) = \{ \underbrace{x(2N_{xi}–N_h+1), \dots, x(2N_{xi} - 1)}_{N_h\text{-}1 \text{ punti da } x_1(n)}, \underbrace{x(2N_{xi}), \dots, x(3N_{xi}–1)}_{N_{xi} \text{ nuovi dati}} \}$$

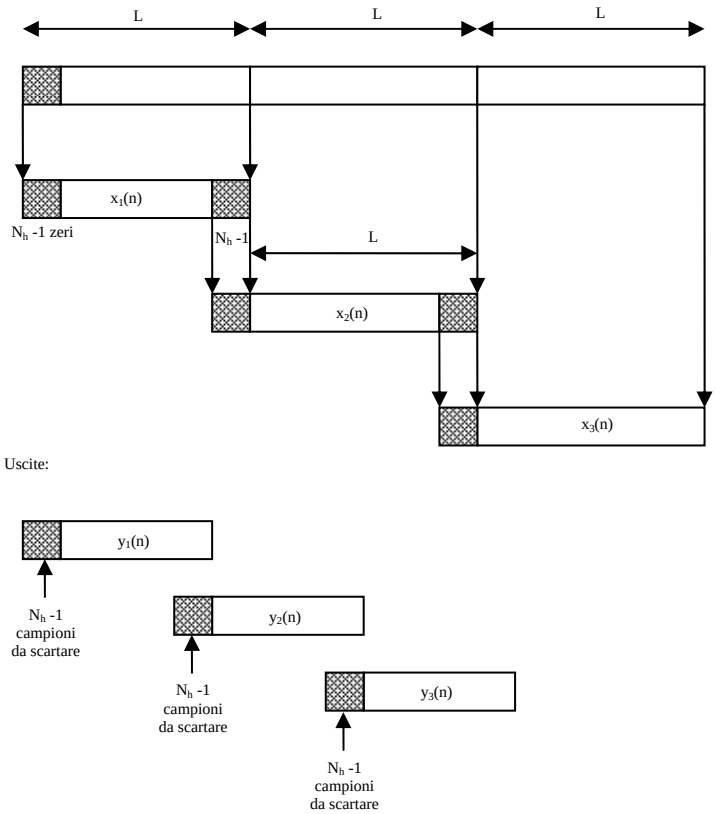


Figura 2.7: Ingressi e uscite del metodo Overlap – Save.

2.2.5 Overlap – Save partizionato

La latenza totale del sistema, nel caso ottimo di fattore di *overlap* pari a 0.5, equivale alla lunghezza di un blocco dell'ingresso, mentre la latenza dell'intero *Overlap – Save* è pari alla lunghezza della IR, che può essere anche molto elevata, senz'altro inadatta ad un sistema *real-time* e soprattutto limitata dalla dimensione delle memorie [Shynk, 2003]. Per questi motivi si ricorre ad un ulteriore partizionamento relativo alla risposta impulsiva, in modo tale da ridurre la latenza del sistema e facilitarne l'implementazione. Questo metodo, detto *Overlap – Save* partizionato, richiederà una somma tra i risultati ottenuti da due partizioni successive, tra loro traslate nel tempo [Proakis and Manolakis, 1996].
Nel caso di partizione uniforme (*Overlap – Save* uniformemente partizionato), si va a ridurre anche la dimensione della FFT cosicché la latenza totale del sistema è ridotta alla dimensione di una singola partizione della IR [Gray, 2003].

2.2.6 Metodo Zero – *Delay*

Questo metodo fu proposto da *Gardner* e consiste in una soluzione ibrida che combina il calcolo diretto con le tecniche di partizionamento appena discusse. Si ottiene in questo modo la minore latenza possibile, così da permetterne l'impiego all'interno di sistemi *real-time*. L'algoritmo è riportato di seguito in figura 2.8.
Malgrado la linearità dell'operazione di convoluzione permetta il partizionamento della risposta impulsiva, occorre notare che al procedere dell'algoritmo la complessità aumenta [Gardner, 1995].
L'algoritmo si sviluppa come segue: i primi N campioni di y_0 sono calcolati usando il metodo diretto con il filtro FIR. Il calcolo di questo primo blocco richiede il tempo necessario all'acquisizione degli N campioni successivi per poter iniziare la prima convoluzione da N punti di y_1. Il calcolo dei nuovi blocchi di y_0 e y_1 necessita del tempo che serve ad acquisire i 2N

campioni per il calcolo del primo blocco di y_2. Così per i 4N campioni necessari a y_3, acquisiti durante il calcolo dei blocchi di y_0, y_1 ed y_2.

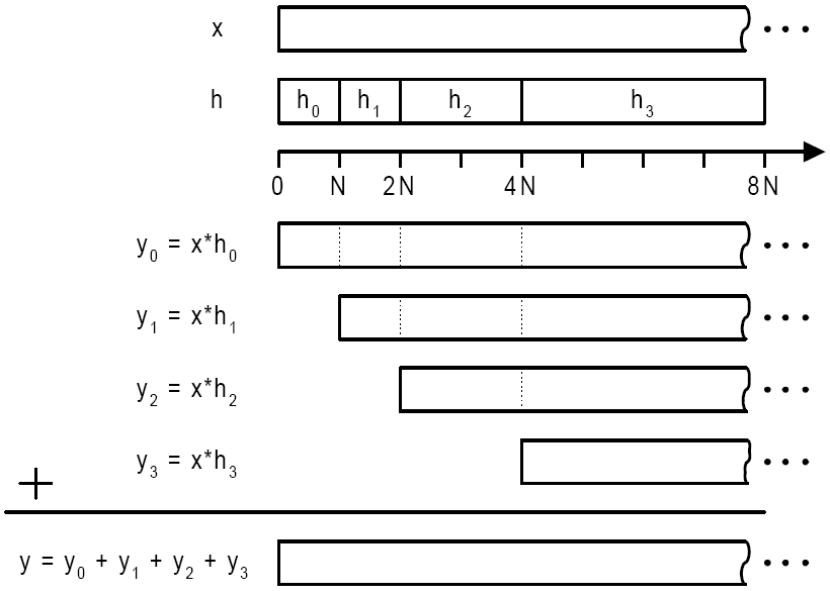


Figura 2.8: L'algoritmo di Gardner per il metodo Zero-Delay.

Gardner definì questa soluzione "a minimo costo" perché non c'è modo di aumentare le dimensioni dei blocchi senza violare la condizione di assenza di ritardo [Gardner, 1995]. Nella pratica questa soluzione non è adottabile poiché tutte le convoluzioni dei singoli blocchi devono essere terminate entro un periodo di campionamento ed inoltre perché la richiesta di calcolo del processore non è uniforme, per cui esso impiega la maggior parte del suo tempo in stati di *idle*, in attesa di un nuovo campione.
Alla luce di questi problemi, *Gardner* ha fornito una soluzione più pratica. Egli propose infatti di rendere uniforme la domanda di capacità di calcolo del processore facendo in modo che blocchi di dimensione N siano processati dai filtri con un ritardo di 2N campioni. La nuova versione dell'algoritmo è mostrata in figura 2.9.

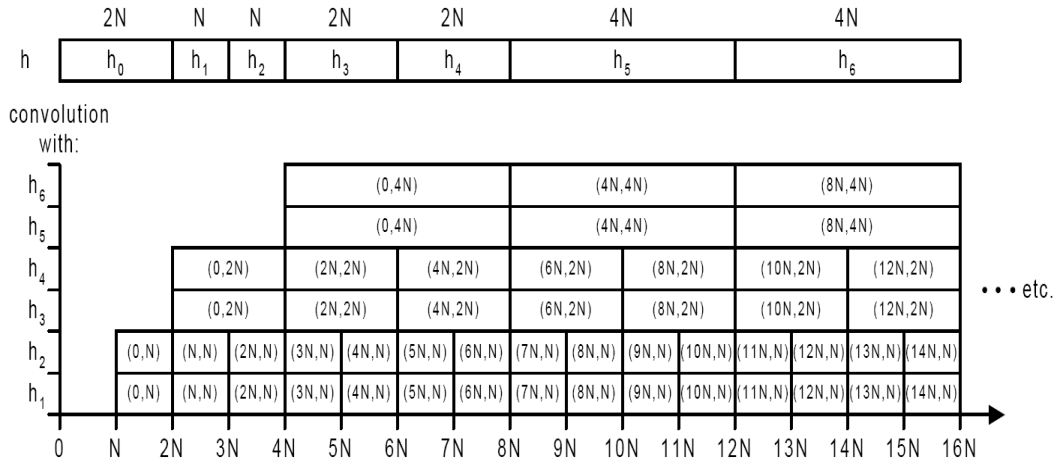


Figura 2.9: L'algoritmo di Gardner nella sua versione implementabile.

In questo caso al processore è assegnato un tempo di calcolo per il blocco pari a quello necessario ad acquisire il numero di campioni dell'ingresso richiesti. Per esempio, la convoluzione dell'ingresso con h_1 inizia al tempo N con il calcolo di $x^*h_1(0,N)$. Quest'operazione deve essersi compiuta al tempo 2N, così che possa iniziare il calcolo di $x^*h_1(N,N)$ e così via. Sebbene le partizioni con indice pari non debbano essere completate allo stesso tempo delle loro corrispondenti con indice dispari, cioè, per esempio, $x^*h_2(0,N)$ deve essere completata entro 3N e non 2N, se si sceglie invece sempre la scadenza più immediata per la coppia pari/dispari corrispondenti, possiamo assicurare una richiesta uniforme del processore [Gardner, 1995].

Malgrado siano necessari sia una significativa quantità di calcoli che uno *scheduling* dei processi estremamente preciso, questo algoritmo fornisce un modo per realizzare convoluzioni a latenza nulla, senza dover implementare dei filtri FIR dalle dimensioni consistenti. L'algoritmo necessita circa di $34 \cdot \log_2(n) - 151$ moltiplicazioni per ciascun campione dell'uscita [Gardner, 1995]. Per una risposta impulsiva da tre secondi, campionata a 48 kHz otteniamo $34 \cdot \log_2(48000 \cdot 3) \approx 582$ moltiplicazioni, contro le 144000 necessarie con il metodo tradizionale a linea di ritardo.

2.3 Confronto tra gli algoritmi

In questa sezione confronteremo tra loro gli algoritmi appena proposti. Non prenderemo in considerazione il metodo *Overlap – Add*, poiché si può dimostrare che richiede sempre un numero maggiore di somme dell'*Overlap – Save* [Proakis and Manolakis, 1996].
I confronti sono avvenuti valutando il numero di operazioni in virgola mobile al secondo (FLOPS) necessarie, così da avere un termine di paragone della necessità di calcolo. Questa figura di merito non può essere l'unico fattore per determinare la soluzione migliore, ma è comunque utile per un confronto tra gli algoritmi, indipendente dalla piattaforma di destinazione, visto che questo metodo non tiene in considerazione operazioni come ad esempio gli accessi in memoria, ma considera solo il numero di istruzioni necessarie all'esecuzione dei calcoli.
Per il calcolo dei FLOPS sono state assunte le seguenti ipotesi:

- I FLOPS sono normalizzati rispetto al numero dei campioni utili in uscita.

- E' stato assunto un fattore di *overlap* del 50%, poiché è stato dimostrato essere la scelta ottimale [Shynk, 2003].

- E' stato considerato un algoritmo di FFT *radix-2* senza ottimizzazioni, cioè con $(N/2)\cdot\log(N/2)$ moltiplicazioni complesse e $N\cdot\log(N)$ somme complesse, detta N la dimensione dell'FFT [Oppenheim and Schafer, 1975].

- Per l'algoritmo *Overlap – Save* non sono state prese in considerazione le operazioni necessarie alla trasformazione della IR.

- Sono state applicate le seguenti equivalenze:
  - somma complessa = 2 somme reali.
  - moltiplicazione complessa = 4 moltiplicazioni reali + 2 somme reali = 6 operazioni reali.

- Moltiplicazioni e addizioni reali hanno il medesimo peso nel conteggio dei FLOPS, considerando un'architettura di riferimento nella quale un'operazione reale impiega un ciclo di *clock* per essere eseguita.

La figura 2.10 mostra i FLOPS in funzione della lunghezza della IR per i diversi algoritmi. Come si vede, il filtraggio FIR risulta sempre la soluzione meno vantaggiosa, escluso il caso di una risposta impulsiva di 64 campioni, che però non rientra nel nostro interesse, vista la sua lunghezza estremamente ridotta. Questo punto rappresenta infatti l'intersezione dei grafici dei costi delle due soluzioni per cui, a monte di esso risulta più conveniente la convoluzione nel dominio del tempo, mentre a valle conviene operare in frequenza.
L'*Overlap – Save* non partizionato presenta il minore numero di FLOPS ma occorre tenere presente che la dimensione della FFT richiesta aumenta con la lunghezza della IR, così come la latenza (direttamente proporzionale alla dimensione della FFT) [Gray, 2003].
L'algoritmo di *Gardner* costituisce il limite superiore per gli algoritmi nel dominio della frequenza (crescita e latenza nulli) [Gardner, 1995].
L'*Overalp – Save* uniformemente partizionato permette una riduzione della latenza rispetto alla forma non partizionata, mantenendo comunque una necessità di potenza di calcolo minore di quella richiesta dall'algoritmo di *Gardner*.

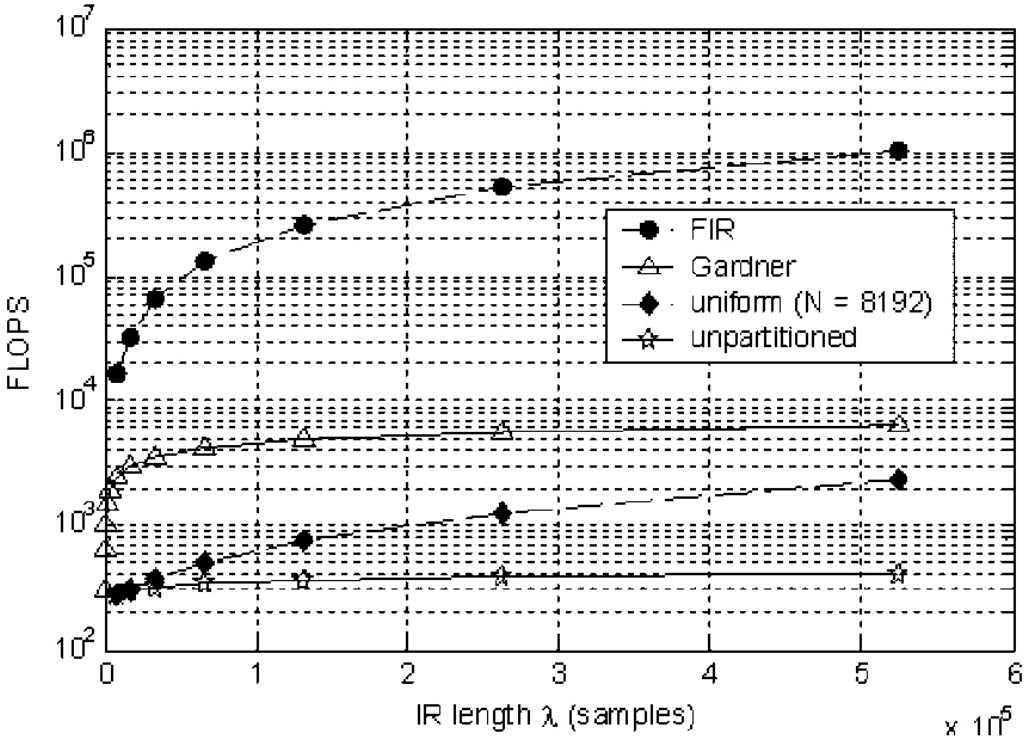


Figura 2.10: FLOPS in funzione della lunghezza della IR per i diversi algoritmi.

Il metodo *Zero – Delay* non presenta problemi dal punto di vista dell'ottimizzazione dello *scheduling* visto che nelle applicazioni *multi-core* questo è del tutto indipendente dal DSP ed è così possibile provvedere alla rimozione delle non linearità rilevate nel processo di *scheduling* stesso. Da un altro punto di vista però, la partizione uniforme della IR offre un'alta regolarità e flessibilità che meglio si adatta alle esigenze di un'implementazione su ASIC o FPGA.

La figura 2.11 illustra i FLOPS necessari ad un algoritmo *Ovelap – Save* uniforme, in funzione della dimensione della FFT per varie lunghezze della risposta impulsiva.

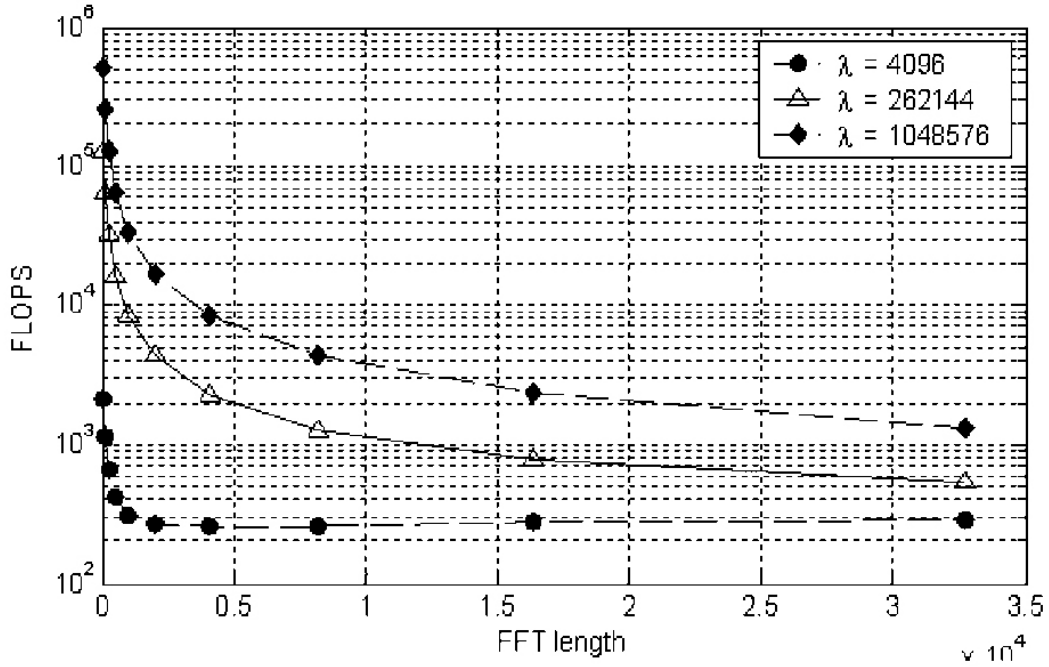


Figura 2.11: FLOPS necessari ad un algoritmo Ovelap – Save uniforme, in funzione della lunghezza della FFT per varie lunghezze della IR.

2.4 Partizionamento *hardware/software*

Una volta analizzati i compiti a cui il sistema dovrà assolvere ed esaminati gli algoritmi di convoluzione, occorre stabilire la piattaforma più idonea all'applicazione. Data la disomogeneità dei requisiti computazionali dei singoli *task* che compongono l'applicazione, i dispositivi FPGA, integrati in un contesto di acceleratore *hardware* per PC, consentano di raggiungere un ottimo compromesso tra esigenze di *real-time processing*, flessibilità e tempo di sviluppo. A questo punto allora si rende necessaria una fase di partizionamento *hardware/software* dell'applicazione.

Al PC sono demandati tutti quei *task* che non pongono dei vincoli stringenti sul tempo di esecuzione (*task software*) mentre sull'acceleratore *hardware* vengono realizzati quelli che possono costituire un collo di bottiglia per l'applicazione (*task hardware*). La definizione dei *task*

hardware è stata fatta tenendo conto anche dei tempi di implementazione più lunghi solitamente richiesti da architetture dedicate.
Il compito più critico risulta la convoluzione della traccia anecoica con la risposta impulsiva, a causa del notevole carico computazionale necessario. L'accesso all'acceleratore *hardware* avviene attraverso un'applicazione *custom* scritta in linguaggio C. Il gestore dell'acceleratore si interfaccia all'ambiente MATLAB attraverso un *bridge software* e all'acceleratore stesso attraverso un *bus* di comunicazione dedicato (*bridge hardware*).
Il filtraggio delle tracce acquisite e l'estrazione della risposta impulsiva da queste ultime sono operazioni *off-line* (che dunque non impongono il rispetto di vincoli *real-time*) e, per questa ragione, possono essere implementati in *software* senza compromettere le prestazioni complessive.
Lo sviluppo dei *task software* avviene all'interno di un ambiente MATLAB/C. Quanto appena esposto è mostrato in figura 2.12.
Una volta stabilito ciò che dovrà essere implementato su *hardware*, occorre determinare il tipo di piattaforma da utilizzare. Dopo aver preso visione dello stato dell'arte possiamo concludere che l'impiego di vari DSP in parallelo, come nel caso della *workstation Huron* (v. capitolo 1), non rappresenta una soluzione convincente poiché estremamente complessa e costosa, sia da un punto di vista delle risorse che da un punto di vista della difficoltà nella gestione e nella programmazione.
Una piattaforma FPGA risulta particolarmente adatta, come detto in precedenza, ad un algoritmo di tipo *Overlap – Save* ed inoltre rappresenta un ottimo compromesso tra un DSP ed una realizzazione *full – custom*. Risulta inoltre un passo obbligato in fase di prototipazione, anche nell'ottica di una futura realizzazione ASIC.

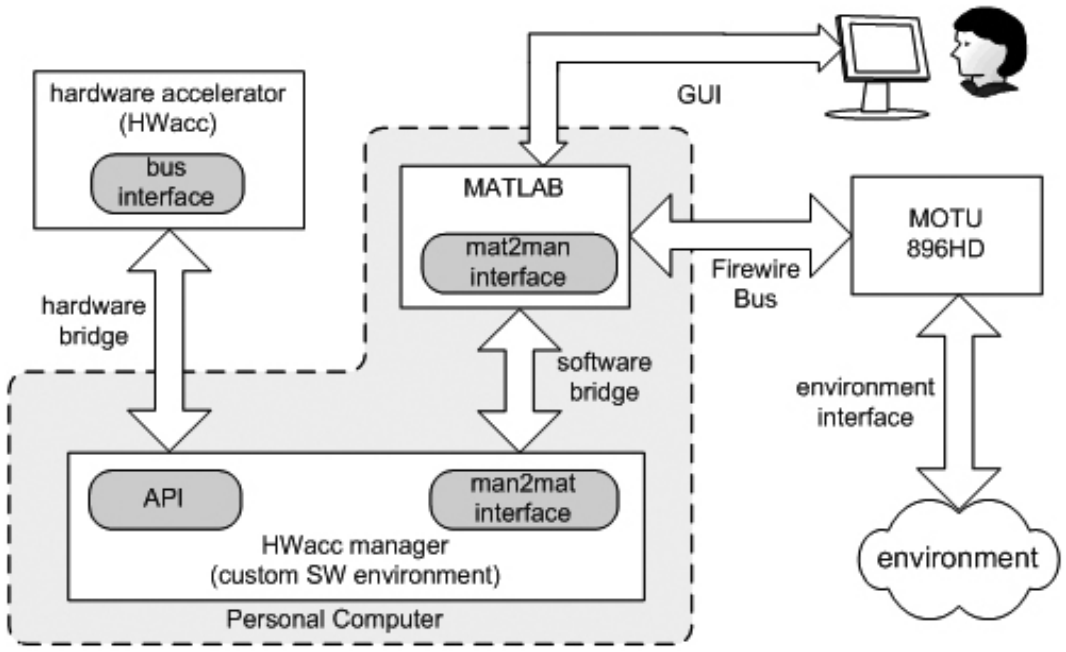


Figura 2.12: Schema del partizionamento hardware/software.

2.5 Architettura VLSI dell'acceleratore hardware

Come primo approccio per la prototipazione, si è optato dunque per un'implementazione su FPGA del convolutore, basandosi sull'algoritmo di *Overlap – Save* uniformemente partizionato, per i motivi appena esposti; occorre sottolineare inoltre che la flessibilità della soluzione adottata permette di operare eventuali rifiniture in fase di progettazione basandosi sullo studio del comportamento del dispositivo sul campo.
La figura 2.13 mostra il *top-level* dell'architettura nel suo insieme, concepita a partire da quanto richiesto dall'applicazione.
La memoria, come si può vedere, è organizzata in tre banchi:

- Memoria della risposta impulsiva: RAM a singola porta implementata sulle memorie *embedded* dell'FPGA, viene

aggiornata ogni volta che vogliamo variare la IR con cui convolvere il segnale.

- Memoria degli ingressi: è una RAM *dual-port* scritta dell'*host* e letta dal convolutore, secondo l'ordine imposto dall'algoritmo *Overlap – Save*. Questo modulo è comune ai quattro canali.

- Memoria delle uscite: è costituita da una RAM *dual-port* scritta dal convolutore e letta dall'*host*.

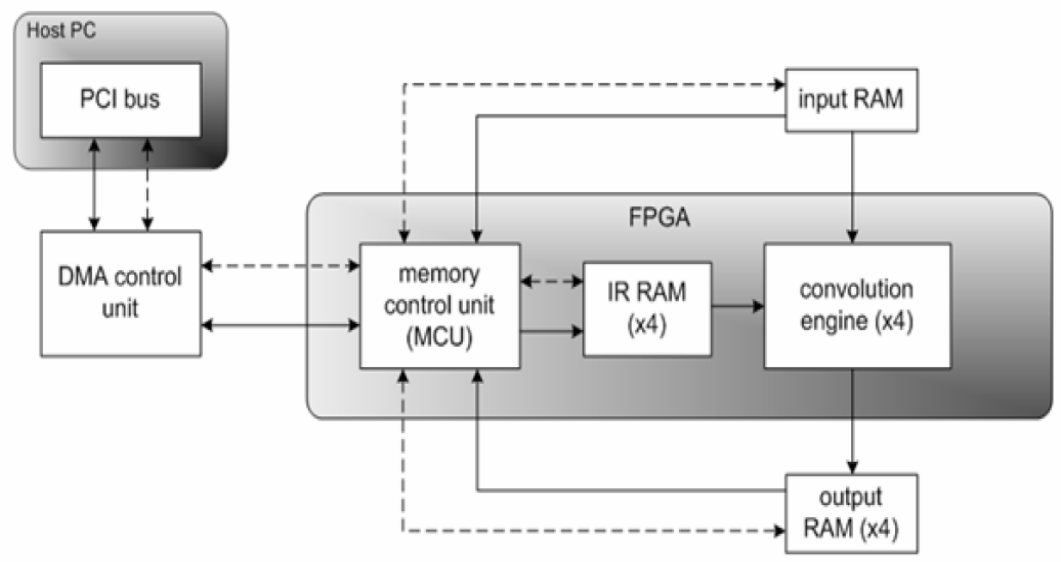


Figura 2.13: Organizzazione ad alto livello del sistema globale: le memorie di ingresso e di uscita sono sulla scheda PCI, l'unità di controllo ed il core di convoluzione sono implementati su FPGA.

Il trasferimento dati tra *host* e scheda, a causa dell'alto numero di informazioni coinvolte, deve avvenire attraverso il canale DMA (*Direct Memory Access*) che ha un *throughput* di circa 80 Mbyte/s. Ciascun trasferimento via DMA, nel caso ad esempio di quattro canali, consiste in un ciclo di scrittura nella memoria degli ingressi e quattro cicli di lettura dalla memoria delle uscite; se si considera una frequenza di campionamento di 192 kHz, notiamo che esiste un fattore 100 di

differenza rispetto alla velocità del PCI, cosicché l'interfaccia di comunicazione non costituisce una limitazione per il sistema.
Entrando nel merito dei componenti da implementare su FPGA, la figura 2.14 mostra l'architettura di alto livello dell'acceleratore *hardware* vero e proprio, secondo quanto richiesto all'algoritmo prescelto [Gray, 2003]. I principali componenti identificati sono: il processore FFT/IFFT, la catena di moltiplicazione ed accumulo (*MAC chain*) ed un insieme di banchi di memoria indipendenti.

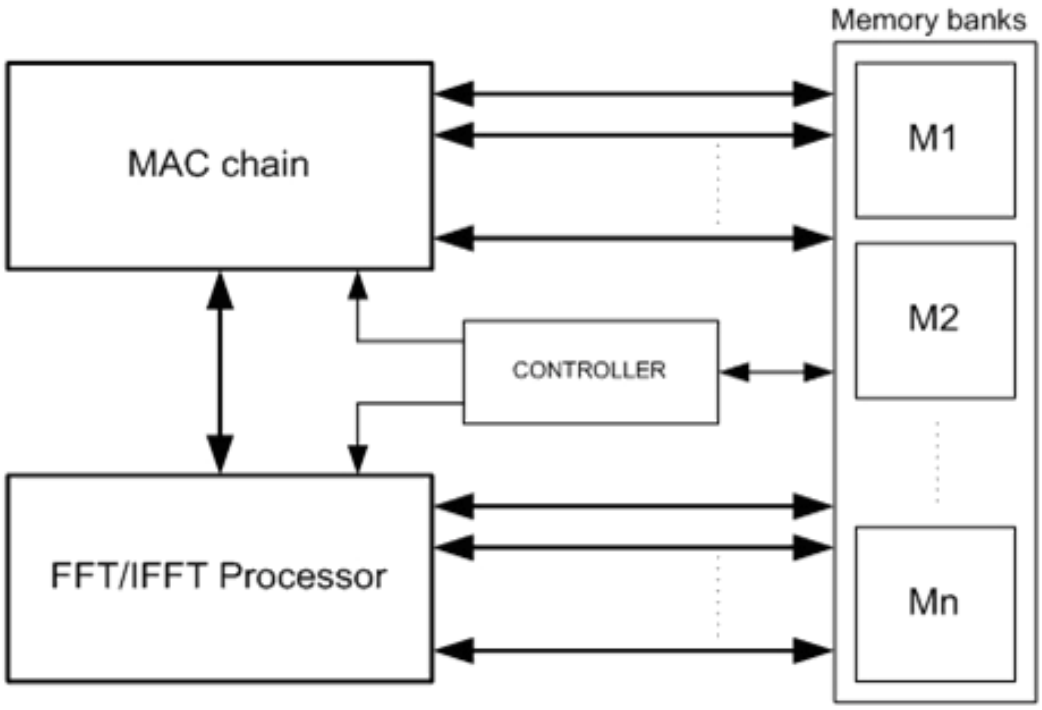


Figura 2.14: Schematizzazione di alto livello dell'acceleratore hardware

Per un corretto funzionamento del sistema proposto è necessario che le memorie posseggano i seguenti requisiti:

– indipendenza tra i banchi: le memorie logiche individuabili all'interno dell'architettura (risposta impulsiva, ingressi e uscite), devono essere

mappate sui banchi di memoria fisica in modo tale che l'accesso a due risorse logiche distinte non origini conflitti;

– memorie *pipelined*: per garantire prestazioni di alto livello servono memorie SRAM ad elevato *throughput*, dette anche *pipelined* o ZBT (*zero bus turnaround*). Questa tipologia di memorie permette un accesso continuo per la lettura e la scrittura dei dati.

Alla luce dell'analisi compiuta a riguardo dell'architettura, possiamo trarre le specifiche di massima alle quali l'acceleratore *hardware* dovrà sottostare:

- Frequenza di campionamento di tutti i segnali: 48 kHz;
- Risoluzione dei segnali: 16/24 *bit*;
- Massima lunghezza della risposta impulsiva: 2 s;

Nei prossimi capitoli andremo ad analizzare in dettaglio ciascuno dei blocchi che compongono l'architettura.

CAPITOLO 3
IL PROCESSORE FFT

In questo capitolo andremo ad analizzare l'architettura VLSI del processore FFT/IFFT. Il processore ha lunghezza programmabile e *bit-width* delle aritmetiche completamente configurabile. Dopo aver illustrato l'algoritmo, sarà descritta l'architettura del processore completa dell'analisi delle precisione e delle aritmetiche finite di macchina.

3.1 Descrizione dell'algoritmo

3.1.1 Algoritmi di tipo *radix*

Il processore FFT utilizzato nel sistema fa riferimento all'algoritmo di tipo *BI & Jones* con architettura *Pipeline – Cascade* [Bi and Jones, 1989].
Gli algoritmi di tipo *radix*, nei quali rientra anche il suddetto, si basano sulla fattorizzazione del numero N di campioni. Solitamente i sottomultipli adottati sono i numeri 4 e/o 2. Se N viene dunque suddiviso in una serie di prodotti del solo numero 2, l'algoritmo si definisce di tipo *radix-2*. Se invece è potenza di 4 e viene scomposto in un prodotto di un certo numero di fattori, tutti uguali a 4, si definisce algoritmo di tipo *radix-4*. Se

invece si adotta una tecnica mista, cioè si fattorizza N secondo una serie di prodotti dei soli numeri 4 e 2 si ha un algoritmo di tipo *mixed-radix*. Nel nostro caso comunque, al massimo N sarà scomposto sempre in un prodotto di fattori tutti uguali a 4, escluso l'ultimo che sarà 4 solo se N ne è una potenza, altrimenti varrà 2.

3.1.2 Algoritmo di *Bi & Jones*

Vediamo adesso di descrivere più esaurientemente l'algoritmo. Detto *v* il numero di radici in cui è stato fattorizzato N (che vedremo essere equivalente al numero di stadi dell'architettura), possiamo scrivere:

$$N = r_1 \cdot r_2 \cdot \ldots \cdot r_v .$$

Si definiscono le seguenti quantità:

$N_t = \dfrac{N}{r_1 \cdot r_2 \cdot \ldots \cdot r_t}$ per $1 \le t \le v-1$ e $N_v = 1$.

$W_M^k = e^{-j(\frac{2\pi k}{M})}$ coefficiente di *Twiddle*.

$X_{[n]}$ sequenza di ingresso per $0 \le n \le N-1$.

Come primo passo, cioè per t=1, si scrive il vettore degli ingressi sottoforma di matrice, riempiendo quest'ultima seguendo l'ordine delle colonne:

$$[X]_0 = \begin{bmatrix} x_{[0]} & x_{[N_1]} & x_{[2N_1]} & \cdots & \cdots & x_{[(r_1-1)N_1]} \\ x_{[1]} & x_{[N_1+1]} & x_{[2N_1+1]} & \cdots & \cdots & x_{[(r_1-1)N_1+1]} \\ x_{[2]} & x_{[N_1+2]} & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{[N_1-1]} & x_{[2N_1-1]} & \cdots & \cdots & \cdots & x_{[r_1N_1-1]} \end{bmatrix}$$

matrice N_1 x r_1

Per ottenere il primo risultato intermedio occorre eseguire il calcolo seguente:

$$^{m1}x_1 = A \cdot B$$

Dove si hanno:

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & W_N^{m_1} & 0 & 0 & \cdots & 0 \\ 0 & 0 & W_N^{2m_1} & 0 & \cdots & 0 \\ 0 & 0 & 0 & W_N^{3m_1} & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & W_N^{(N_1-1)m_1} \end{bmatrix}$$

matrice N_1 X N_1, $0 \le m_1 \le r_1-1$

$$B = [X_{[0]}] \cdot \begin{bmatrix} 1 \\ W_N^{m_1} \\ W_N^{2m_1} \\ W_N^{3m_1} \\ \vdots \\ W_N^{(N_1-1)m_1} \end{bmatrix}$$

matrice 1 X N_1, $0 \le m_1 \le r_1-1$

I coefficienti della matrice diagonale A, sono i coefficienti di *Twiddle* per il primo stadio. Otteniamo, alla fine, r_1 vettori di lunghezza N_1: ${}^{0}x_1$, ${}^{1}x_1$, ${}^{2}x_1$, … , ${}^{r_1-1}x_1$.

Possiamo ripetere quanto appena esposto per il passo successivo, facendo le operazioni elencate a partire dal vettore in uscita dal primo stadio, anziché da quello in ingresso. Otterremo r_2 vettori di lunghezza N_2, per ogni vettore proveniente dal primo passo, in totale $r_1 \cdot r_2$ vettori.

$$\left[X\right]_1^{m_1} = \begin{bmatrix} {}^{m1}x_{1_{[0]}} & {}^{m1}x_{1_{[N_2]}} & {}^{m1}x_{1_{[2N_2]}} & \dots & \dots & {}^{m1}x_{1_{[(r_2-1)N_2]}} \\ {}^{m1}x_{1_{[1]}} & {}^{m1}x_{1_{[N_2+1]}} & {}^{m1}x_{1_{[2N_2+1]}} & \dots & \dots & {}^{m1}x_{1_{[(r_2-1)N_2+1]}} \\ {}^{m1}x_{1_{[2]}} & {}^{m1}x_{1_{[N_2+2]}} & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ {}^{m1}x_{1_{[N_2-1]}} & {}^{m1}x_{1_{[2N_2-1]}} & \dots & \dots & \dots & {}^{m1}x_{1_{[r_2N_2-1]}} \end{bmatrix} \quad \text{matrice } N_2 \text{ x } r_2$$

$${}^{m_1m_2}x_2 = \begin{bmatrix} 1 & 0 & 0 & 0 & \dots & 0 \\ 0 & W_{N_1}^{m_2} & 0 & 0 & \dots & 0 \\ 0 & 0 & W_{N_1}^{2m_2} & 0 & \dots & 0 \\ 0 & 0 & 0 & W_{N_1}^{3m_2} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \dots & \vdots \\ 0 & 0 & 0 & 0 & \dots & W_{N_1}^{(N_2-1)m_2} \end{bmatrix} \cdot \left[X\right]_1^{m_1} \cdot \begin{bmatrix} 1 \\ W_{r_2}^{m_2} \\ W_{r_2}^{2m_2} \\ W_{r_2}^{3m_2} \\ \vdots \\ W_{r_2}^{(N_2-1)m_2} \end{bmatrix} \quad 0 \le m_1 \le r_1-1 \text{ e } 0 \le m_2 \le r_2-1$$

Per il passo t generico otterremo:

$$\left[X\right]_{t-1}^{m_1 m_2 \ldots m_{t-1}} = \begin{bmatrix} {}^{m_1 m_2 \ldots m_{t-1}}x_{t-1_{[0]}} & {}^{m_1 m_2 \ldots m_{t-1}}x_{t-1_{[N_t]}} & \cdots & \cdots & \cdots & {}^{m_1 m_2 \ldots m_{t-1}}x_{t-1_{[(r_t-1)N_t]}} \\ {}^{m_1 m_2 \ldots m_{t-1}}x_{t-1_{[1]}} & {}^{m_1 m_2 \ldots m_{t-1}}x_{t-1_{[N_t+1]}} & \cdots & \cdots & \cdots & {}^{m_1 m_2 \ldots m_{t-1}}x_{t-1_{[(r_t-1)N_t+1]}} \\ {}^{m_1 m_2 \ldots m_{t-1}}x_{t-1_{[2]}} & {}^{m_1 m_2 \ldots m_{t-1}}x_{t-1_{[N_t+2]}} & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ {}^{m_1 m_2 \ldots m_{t-1}}x_{t-1_{[N_t-1]}} & {}^{m_1 m_2 \ldots m_{t-1}}x_{t-1_{[2N_t-1]}} & \cdots & \cdots & \cdots & {}^{m_1 m_2 \ldots m_{t-1}}x_{t-1_{[r_t N_t-1]}} \end{bmatrix} \quad \text{matrice } N_t \text{ x } r_t$$

$${}^{m_1 m_2 \ldots m_t}x_t = \begin{bmatrix} 1 & 0 & 0 & 0 & \ldots & 0 \\ 0 & W_{N_{t-1}}^{m_t} & 0 & 0 & \ldots & 0 \\ 0 & 0 & W_{N_{t-1}}^{2m_t} & 0 & \ldots & 0 \\ 0 & 0 & 0 & W_{N_{t-1}}^{3m_t} & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ldots & \vdots \\ 0 & 0 & 0 & 0 & \ldots & W_{N_{t-1}}^{(N_t-1)m_t} \end{bmatrix} \cdot \left[X\right]_{t-1}^{m_1 m_2 \ldots m_t} \cdot \begin{bmatrix} 1 \\ W_{r_t}^{m_t} \\ W_{r_t}^{2m_t} \\ W_{r_t}^{3m_t} \\ \vdots \\ W_{r_t}^{(r_t-1)m_t} \end{bmatrix} \quad 2 \le t \le v\text{-}1 \text{ e } 0 \le m_t \le r_t\text{-}1$$

Infine, per l'ultimo passo, cioè quello per t = *v*, avremo la sequenza di uscita nel modo seguente:

$$X_{(k)} = \left({}^{m_1 m_2 \ldots m_{v-1}}x_{v-1}\right)^T \cdot \begin{bmatrix} 1 \\ W_{r_v}^{m_v} \\ W_{r_v}^{2m_v} \\ W_{r_v}^{3m_v} \\ \vdots \\ W_{r_v}^{(r_t-1)m_v} \end{bmatrix}$$

con $0 \le k \le N\text{-}1$, dove $k = r_1 r_2 \ldots r_{v-1} m v + r_1 r_2 \ldots r_v\text{-}2 m_{v-1} + \ldots + r_1 m_2 + m_1$, per $1 \le t \le v\text{-}1$, $0 \le m_t \le r_t\text{-}1$.

Una rappresentazione grafica dei vettori, per un caso *radix-4* è quella mostrata in figura 3.1.

N_1 sarà pari ad N/4, N_2 sarà N/16, e così via. Il numero totale degli stadi si ricava dunque mediante la semplice formula: $v = \log_4(N)$.

Si nota una forte regolarità nella struttura dell'algoritmo che viene sfruttata nell'implementazione architetturale.

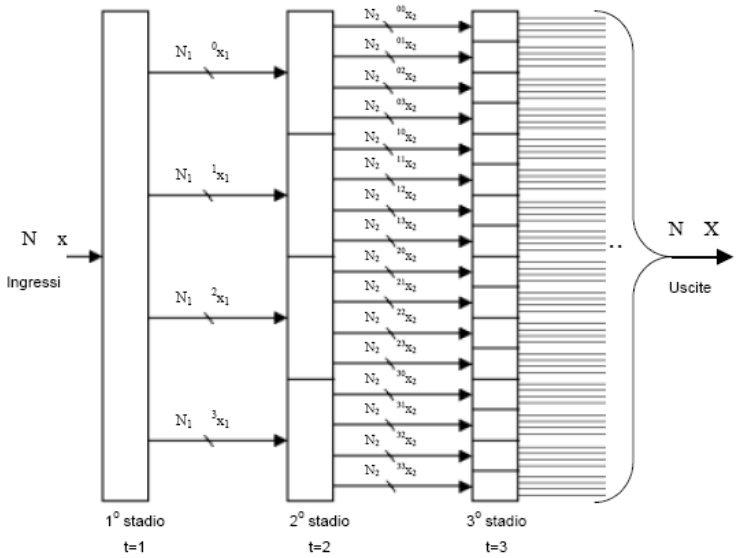


Figura 3.1: Schematizzazione dei vettori nei primi tre passi dell'algoritmo di calcolo per un caso radix-4.

Per chiarire ulteriormente questo concetto, in figura 3.2 è riportato l'algoritmo di *BI & Jones*, evidenziando con dei puntini neri i risultati intermedi e con dei cerchi bianchi le operazioni di somma e sottrazione. I numeri dentro i cerchi rappresentano i valori di m_t per lo stadio *t-esimo*. I coefficienti di *Twiddle* dello stadio sono dati da $W_{N_t-1}^{m_t q}$, con *q* compreso tra 0 ed N_t-1, dove, i valori dei suddetti indici, sono indicati dai numeri che si trovano sopra le linee. È fondamentale notare che i dati in uscita non sono

ordinati, bensì risultano *digit-reversal order* [Proakis and Manolakis, 1996].
In questo tipo di algoritmo, si possono dunque individuare tre tipi di operazioni principali:

- Organizzazione degli ingressi di ciascuno stadio sottoforma di matrice. Operazione di commutazione e miscelazione.

- Operazioni di somma e sottrazione (dette *butterfly*). Ciascuna componente del vettore B_t, di dimensione N_t, rappresentato qui di seguito (figura 3.3), è l'uscita della *butterfly* dello stadio *t-esimo*. Ciascuna di queste è ottenuta con operazioni di somme e/o sottrazioni sulle componenti reali ed immaginarie di $[X]_{t-1}^{m_1 m_2 \ldots m_t}$.

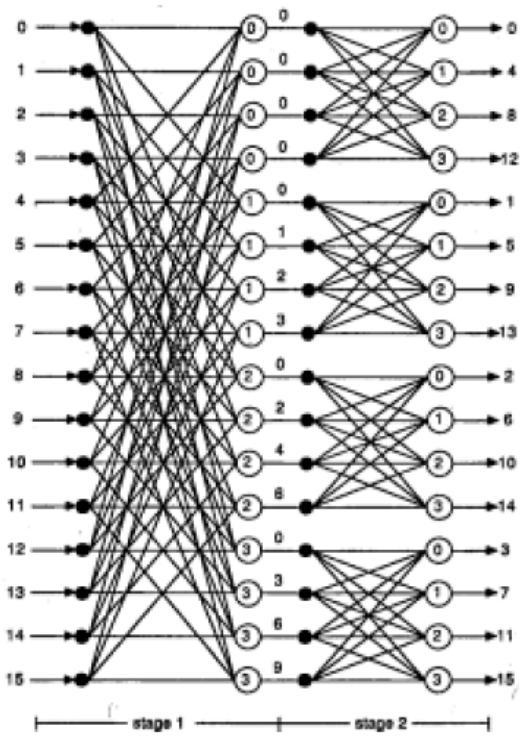


Figura 3.2: Algoritmo BI & Jones radix-4 con N=16.

$$B_t = [X]_{t-1}^{m_1 m_2 \ldots m_t} \cdot \begin{bmatrix} 1 \\ W_{r_t}^{m_t} \\ W_{r_t}^{2m_t} \\ W_{r_t}^{3m_t} \\ \vdots \\ W_{r_t}^{(r_t-1)m_t} \end{bmatrix}$$

Figura 3.3: Vettore delle componenti in uscita dalle butterfly dello stadio t-esimo.

- Moltiplicazione per i coefficienti di *Twiddle*.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \ldots & 0 \\ 0 & W_{N_{t-1}}^{m_t} & 0 & 0 & \ldots & 0 \\ 0 & 0 & W_{N_{t-1}}^{2m_t} & 0 & \ldots & 0 \\ 0 & 0 & 0 & W_{N_{t-1}}^{3m_t} & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ldots & \vdots \\ 0 & 0 & 0 & 0 & \ldots & W_{N_{t-1}}^{(N_t-1)m_t} \end{bmatrix} \cdot B_t$$

Figura 3.4 : moltiplicazione per i coefficienti di Twiddle.

Ogni componente del vettore così ottenuto rappresenta l'uscita dello stadio t-esimo. Ognuna di esse viene presentata in uscita in tempi diversi. Da notare che l'ultimo stadio dell'architettura non presenta operazioni di moltiplicazione per i coefficienti di *Twiddle*. Si riduce quindi alle sole operazioni compiute dalle *butterfly*.

3.1.3 Considerazioni sull'aritmetica finita di macchina

Per stabilire il tipo di aritmetica da adottare, abbiamo utilizzato i risultati ottenuti in [Fanucci et al, 2002] e [Saponara et al, 2005]. Da quanto emerso nell'analisi contenuta nei documenti citati, la soluzione migliore risulta l'aritmetica CBFP (*Convergent Block Floating Point*) per la quale è stato messo a disposizione un *software* in grado di dimensionare le *bit-*

width interne a seconda delle dimensioni degli ingressi e delle uscite desiderate.

Con questo tipo di aritmetica, supponendo uno stadio *radix-2*, viene calcolato il modulo di tutti gli *N* dati in ingresso e viene confrontato il modulo massimo con il valore 1/2. Se è inferiore viene evitato lo *scaling* (divisione per 2) all'interno del primo stadio, se invece è maggiore, il dato viene scalato. Comunque sia, i dati di uscita dal primo stadio hanno modulo minore di 1. In ingresso al secondo stadio abbiamo ancora *N* dati ma la prima metà può essere processata indipendentemente dalla seconda. È possibile calcolare il modulo massimo dei primi *N*/2 dati e decidere se effettuare lo *scaling* all'interno del secondo stadio. In maniera del tutto indipendente è possibile ripetere il calcolo e il confronto per l'altra metà di dati relativi sempre al secondo stadio (senza curarsi se vi sia stato o meno *scaling* nella prima metà).

In uscita dal secondo stadio abbiamo due gruppi di dati, ciascuno di *N*/2 campioni, che in generale saranno associati ad esponenti differenti, quindi, per memorizzare questa informazione, sarà necessario disporre di 2 esponenti, uno per gruppo, codificati su un certo numero di *bit*. In maniera analoga la tecnica può essere ripetuta anche per gli stadi successivi: entrambi i gruppi di *N*/2 dati in uscita dal secondo stadio possono a loro volta essere divisi in due sottogruppi di *N*/8 dati e così via per gli stadi seguenti. Iterando questa tecnica, i singoli dati finali di uscita avranno tutti esponenti diversi tra loro e per questo motivo ad ogni dato sarà associato un esponente che tiene il conto di quanti *scaling* non sono stati effettuati lungo il percorso del dato dall'ingresso all'uscita. Per stadi *radix-4* il discorso è analogo: invece di confrontare il modulo massimo con 1/2, viene confrontato con 1/4 e, per esempio, in uscita al primo stadio abbiamo 4 gruppi indipendenti di dati anziché 2.

Questa tecnica di *scaling* dei dati permette di escludere a priori la possibilità di *overflow* all'interno della catena. La parola *convergent* sta ad indicare che i dati di ingresso sono associati al medesimo esponente mentre i dati di uscita assumono ciascuno un proprio esponente: l'intero

schema converge dunque da una rappresentazione *fixed-point* iniziale a una sorta di *floating-point* con un esponente limitato a pochi valori. In realtà in uscita ci saranno, nel caso *radix-2*, coppie di valori complessi che avranno lo stesso esponente (nel caso *radix-4* saranno gruppi di quattro). La tecnica BFP classica per algoritmi *radix* consiste invece nel valutare se effettuare lo *scaling* ad ogni stadio sempre su tutti gli *N* valori intermedi, per cui i risultati finali avranno tutti un unico esponente che sarà in generale diverso da quello iniziale. Osserviamo quindi che l'aritmetica CBFP si presenta come una tecnica superiore rispetto alla tecnica classica BFP in quanto i dati di uscita avranno a piccoli gruppi (massimo di 4 valori) esponenti tra loro differenti e questo permette di ottenere una maggiore precisione. Inoltre, la BFP classica su architetture *pipeline* non è applicabile. Non sarebbe mai possibile infatti poter calcolare il modulo massimo su tutti gli N valori intermedi in uscita ad ogni stadio.

Nel caso in cui i dati in ingresso alla FFT non soddisfino la condizione di modulo minore di uno, ma questa fosse soddisfatta solo per la parte reale ed immaginaria singolarmente, è possibile riutilizzare la tecnica del *right-shift* (RSH). Nel passaggio da SWL *bit* (*bit* con i quali opera internamente il sistema) a OWL *bit* (*bit* di uscita), bisogna considerare anche l'esponente del dato.

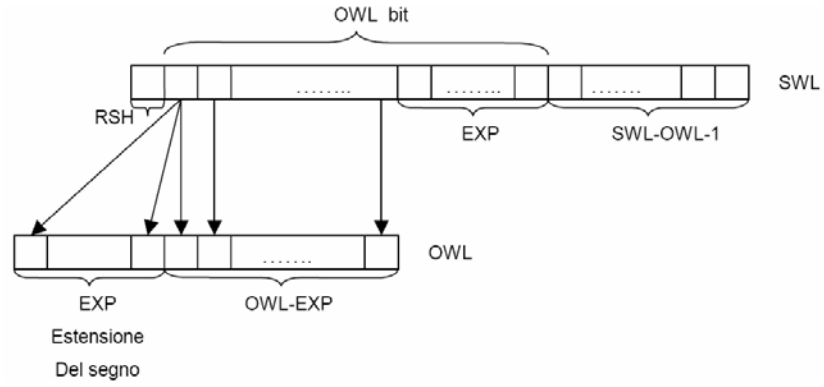


Figura 3.5: Passaggio da SWL a OWL bit.

Nel passaggio a OWL *bit*, tutti i dati di uscita vengono riportati allo stesso esponente (uguale a quello della BFP), così da avere un formato *fixed-point* dei dati di uscita uguale a quello delle altre aritmetiche. In questo modo però, si riducono i vantaggi della tecnica CBFP, infatti, nel riportare tutti i dati di uscita al medesimo esponente, vengono eliminati *bit* significativi. Ricordiamo che l'esponente codifica il numero totale di *shift* (relativi agli *scaling*) che non sono stati effettuati lungo la catena degli stadi, che saranno dunque recuperati nel passaggio da SWL a OWL *bit*.
Grazie al calcolo del modulo complesso dei dati in ingresso, seguito da un confronto con il valore 1, è il processore stesso a decidere se utilizzare tale tecnica, in funzione del risultato della comparazione stessa. Se il modulo massimo è minore allora la tecnica RSH non viene impiegata, mentre se è maggiore viene utilizzata. Nel caso in cui la RSH non sia necessaria, dal momento che il valore di SWL rimane lo stesso in entrambi i casi, il processore dispone di un *bit* meno significativo in più da sfruttare per avere una accuratezza migliore in uscita (tecnica dell'RSH dinamica). Il numero di *bit* meno significativi aggiunti a DWL è pari a (SWL-DWL) mentre, nel caso di RSH, è dato da (SWL-DWL-1).

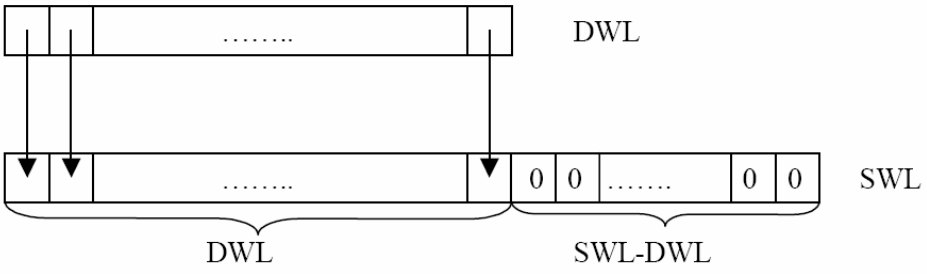


Figura 3.6: Passaggio da DWL a SWL bit nel caso di assenza di RSH.

3.2 Architettura VLSI

Per l'implementazione del blocco di calcolo della FFT siamo partiti dall'algoritmo illustrato in 3.1 per definire una prima architettura. In seguito sono state apportate alcune ottimizzazioni mirate all'implementazione su dispositivi FPGA (v. capitolo 6).
Il processore, come detto in precedenza, si basa sull'algoritmo *BI & Jones* con architettura *Pipeline-Cascade* e aritmetica CBFP. I principali blocchi del processore sono:

- I Commutatori
- Le *Butterfly*
- I Moltiplicatori Complessi
- Le ROM dei coefficienti di *Twiddle*
- La *Control Unit*

L'interfaccia del processore comprende due *bus* di *input* (*Input Re* - ingresso per la parte reale - ed *Input Im* - ingresso per la parte immaginaria), due bus di *output* (*UI* - parte reale dei dati in uscita, *UR* - parte immaginaria dei dati di uscita) ed i segnali di *clock* e *reset*.
L'architettura è completamente parametrica e la configurazione è gestibile attraverso un programma in linguaggio C++ che provvede alla creazione in automatico dei *file* necessari, come sarà approfondito nel capitolo 6.
Il numero di stadi presenti nell'architettura è pari al logaritmo in base 4 di N, dove N rappresenta la massima lunghezza della FFT/IFFT. Nel caso in cui N non sia potenza di quattro (ma sia comunque una potenza di due), il numero di stadi sarà pari al troncato superiore, dove l'ultimo stadio risulterà di tipo *radix-2* (*mixed-radix*).
Per avere una visione generale dell'architettura, si consideri dunque, in base a quanto appena esposto, che solo il primo e l'ultimo stadio differiranno dal generico stadio *j-esimo*, per cui gli stadi intermedi saranno

generati automaticamente, tutti con la medesima struttura. Questo concetto è espresso in figura 3.4.

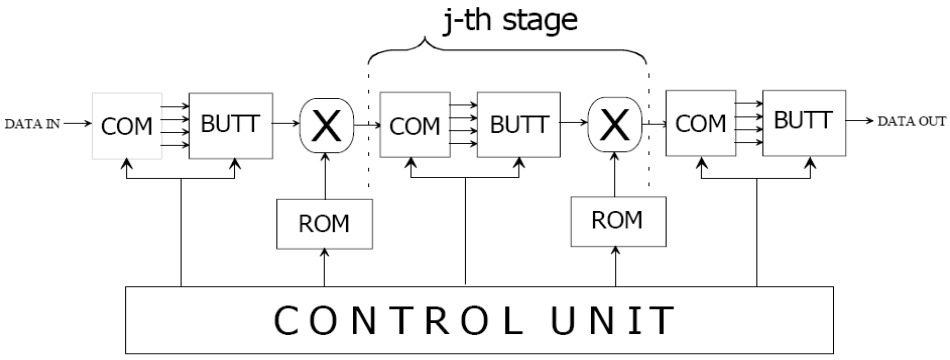


Figura 3.4: Panoramica dell'architettura degli stadi

Come si può immediatamente notare, sono individuabili nel *j-esimo* stadio, i blocchi principali elencati in precedenza.

3.2.1 Il commutatore

Il commutatore riceve in ingresso un unico dato complesso per poi generarne 2 o 4 a seconda del tipo di stadio nel quale è contenuto, cioè se, rispettivamente, *radix*-2 o *radix*-4. Questo blocco contiene gli opportuni ritardi per equalizzare il *data-path* e gestire i dati secondo l'organizzazione a matrice illustrata in 3.1.

3.2.2 La butterfly ed il moltiplicatore complesso

La *butterfly* compie le operazioni di somma e sottrazione necessarie alla determinazione del vettore che verrà moltiplicato per i coefficienti di *Twiddle*, provenienti dalla ROM associata allo stadio. Il moltiplicatore complesso è stato implementato con quattro *Booth Multipliers*. Un

Multiplexer seguito da un blocco di ritardo (v. figura 3.5) realizza le moltiplicazioni per il fattore *(1+ j0)*, evitando un'inutile approssimazione sull'ingresso che rimane così invariato. Il controllo per la scelta dell'uscita è facilmente generabile dato che i coefficienti di *Twiddle* sono noti a priori e, di conseguenza, sono note le posizioni dei coefficienti puramente reali di modulo unitario. Per l'esattezza, se la ROM ha profondità M, i *Twiddle* unitari si troveranno nelle prime M/4 locazioni e nella prima locazione di tutti i successivi blocchi di dimensione M/4, come mostrato in figura 3.6.

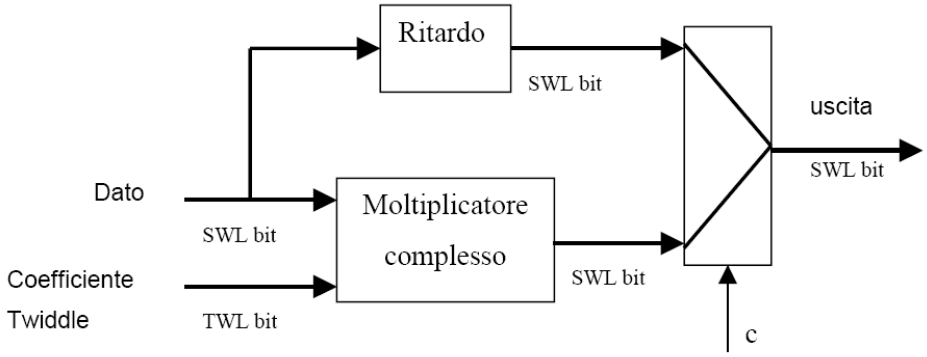


Figura 3.5: Schema della struttura completa del blocco di moltiplicazione

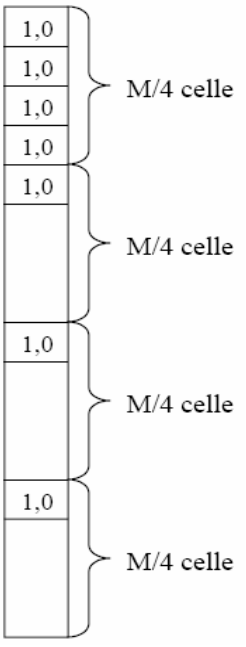


Figura 3.6: Rappresentazione di un generico vettore dei coefficienti di Twiddle presente nelle varie memorie ROM

3.2.3 Le memorie ROM

In ciascuno stadio è presente una ROM, fatta eccezione per l'ultimo che non contiene il moltiplicatore complesso. Ciascuna memoria contiene *words* di medesima dimensione ma ognuna ha una dimensione pari ad $N/4^{(stage-1)}$, dove *stage* indica lo stadio in cui la ROM stessa è inserita. Il contenuto di ciascuna memoria viene generato a partire da quello della precedente prendendo un elemento ogni quattro. I *file* con il contenuto delle ROM sono generate in automatico da un programma C++ sulla base della configurazione del processore.

3.2.4 L'unità di controllo

L'unità di controllo (figura 3.7) ha il compito di sincronizzare tra loro sia tutti i blocchi costituenti lo stadio, sia i vari stadi tra loro. Ogni stadio possiede una propria unità di controllo che genera gli indirizzi per l'accesso alle memorie e fornisce il segnale di sincronizzazione.

3.2.5 Il blocco di decisione

Il blocco denominato modulo massimo e decisione (figura 3.7) controlla l'eventuale variazione di esponente all'interno della *Butterfly* in accordo all'aritmetica CBFP (v. paragrafo 3.1.). La rete calcola il modulo approssimato dell'ingresso e lo confronta con quello del precedente generando il segnale di *scaling*. L'esponente è aumentato di 1 o di 2 se lo stadio è *radix-2* oppure *radix-4*, rispettivamente.

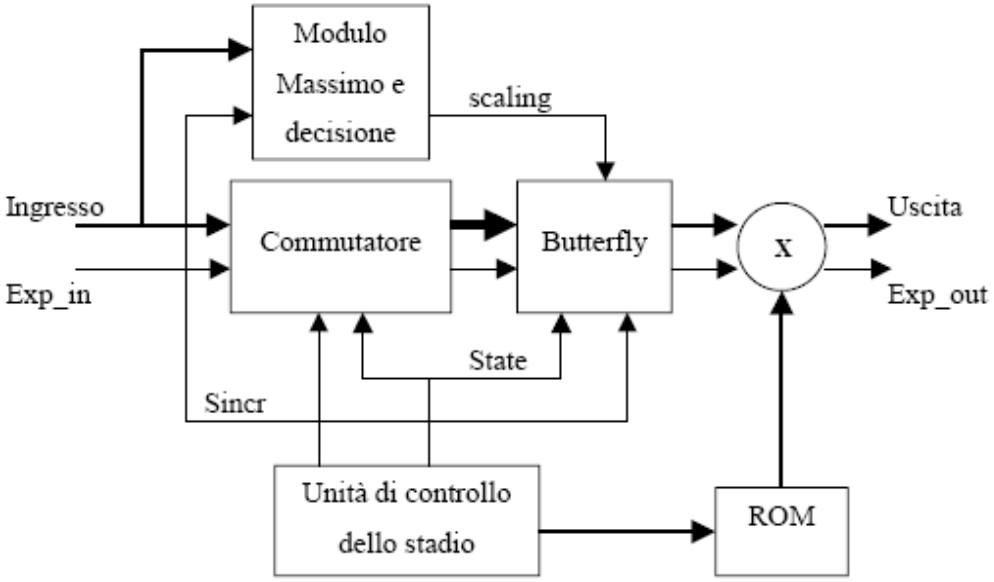


Figura 3.7: stadio j-esimo completo per aritmetica CBFP

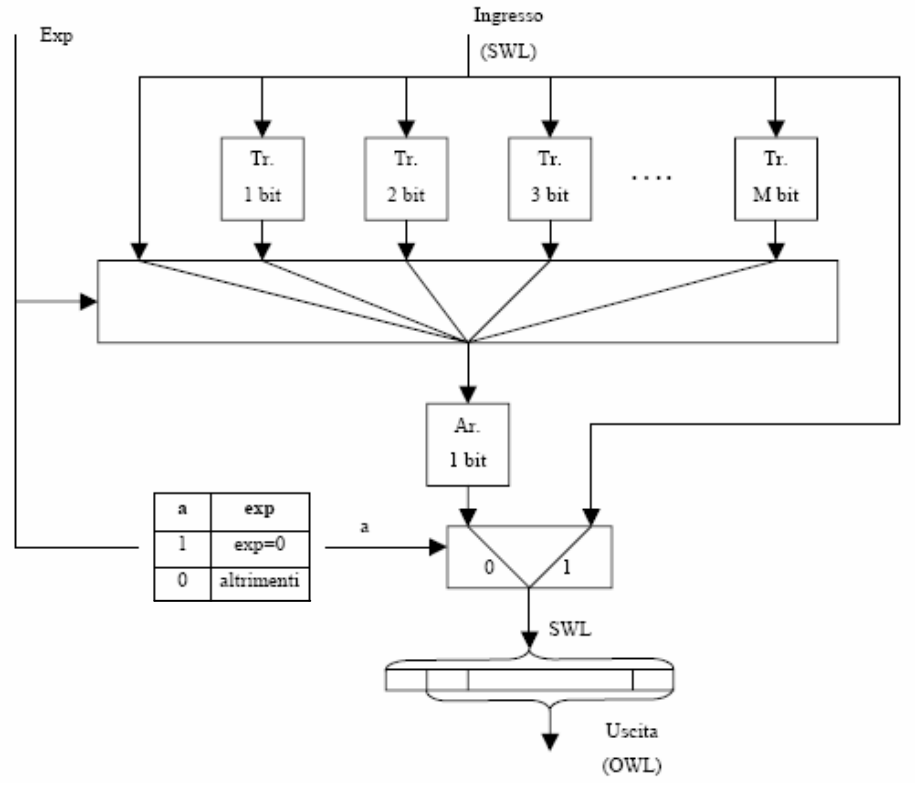


Figura 3.8: Il blocco di arrotondamento finale

3.2.6 Approssimazioni

Per quanto riguarda le approssimazioni, si hanno dei blocchi inseriti dopo ciascun moltiplicatore reale ed un blocco di arrotondamento finale (entrambi contenuti in *arrotondamento.vhd*). Per quanto riguarda i primi, l'operazione di arrotondamento è stata compiuta troncando il numero di *bit* di interesse e sommando ad essi il *bit* più significativo tra quelli scartati.
Per quanto riguarda invece l'arrotondamento finale, questo deve tenere conto dell'esponente accumulato dai dati durante le varie elaborazioni negli stadi. Come si può vedere in figura 3.8, ci sono M reti, a complessità nulla, selezionate a seconda del valore dell'esponente di uscita, che troncano ciascuna un numero via via crescente di *bit* da 1 al valore massimo di *Exp*. Se l'esponente vale *k*, viene selezionata la rete che tronca *k-1 bit* che saranno successivamente privati del *bit* meno significativo dalla rete Ar_1 *bit*. In questo modo l'esponente è stato ridotto a 0. Se *Exp* valeva già 0 allora un *Multiplexer* provvede a far uscire il dato senza troncamenti.

3.2.7 Latenza del processore FFT

In questo paragrafo sarà ricavata la formula per il calcolo della latenza del processore FFT. Innanzitutto, occorre calcolare il numero di stadi dei quali è composta l'architettura, già accennato in precedenza:

$$stadi = ceil\left(\log_4\left(\frac{N}{2^{MD}}\right)\right) \qquad \textit{numero di stadi}$$

MD rappresenta il valore del parametro *mode* selezionato in fase di configurazione, incrementando il quale, mantenendo la stessa architettura, è possibile dimezzare i punti della FFT e, conseguentemente, ridurre la latenza. La funzione *ceil(*x*)* restituisce il minimo intero maggiore o uguale a x.

Occorre adesso specificare i ritardi introdotti dai singoli blocchi, riportati nella tabella di figura 3.9.

	Ritardo (cicli di *clock*)	**Descrizione**
CLK_{butt_r4}	3	Ritardo della *Butterfly radix-4*
CLK_{butt_r2}	3	Ritardo della *Butterfly radix-2*
CLK_{mult}	3	Ritardo del moltiplicatore complesso
CLK_{modulo}	3	Ritardo della rete per il calcolo del modulo massimo
$CLK_{last\_com\_r4}$	4	Ritardo del commutatore *radix-4* dell'ultimo stadio
$CLK_{last\_com\_r2}$	2	Ritardo del commutatore *radix-2* dell'ultimo stadio

Figura 3.9: Latenze dei blocchi costituenti il processore FFT.

Definiamo il fattore Q_t come:

$$Q_t = \frac{4}{2^{MD}} \cdot \frac{N}{4^t}$$

Possiamo a questo punto esplicitare la latenza come:

$$latenza = 3 + \sum_{t=1}^{stadi-1} [Q_t] + (CLK_{modulo} + CLK_{butt\_r4} + CLK_{mult}) \cdot (stadi - 1) + last(MD)$$

Dove con *last*(MD) si intende:

$$last(\mathrm{MD}) = \begin{cases} CLK_{last\_com\_r4} + CLK_{modulo} + CLK_{last\_butt\_r4} & \text{per MD pari} \\ CLK_{last\_com\_r2} + CLK_{modulo} + CLK_{last\_butt\_r2} & \text{per MD dispari} \end{cases}$$

CAPITOLO 4
IL CORE DI CONVOLUZIONE

In questo capitolo analizzeremo i blocchi che completano l'architettura del convolutore nel dominio della frequenza, secondo l'algoritmo *Overlap – Save* uniformemente partizionato. I moduli presentati sono il gestore della memoria degli ingressi, la catena di moltiplicazione e accumulo (MAC) e le memorie delle partizioni della IR, con i loro rispettivi gestori.

4.1 Gestore della memoria degli ingressi

Questo modulo fa sì che i dati in ingresso al processore FFT siano organizzati in accordo all'algoritmo di *Overlap – Save* uniformemente partizionato [Proakis and Manolakis, 1996]. La memoria degli ingressi dovrà essere letta dapprima interamente, leggendo però due volte la seconda metà e poi, ciclicamente, dovranno essere lette a turno, due volte ciascuna, le due metà, come mostrato in figura 4.1 per il caso di quattro locazioni.

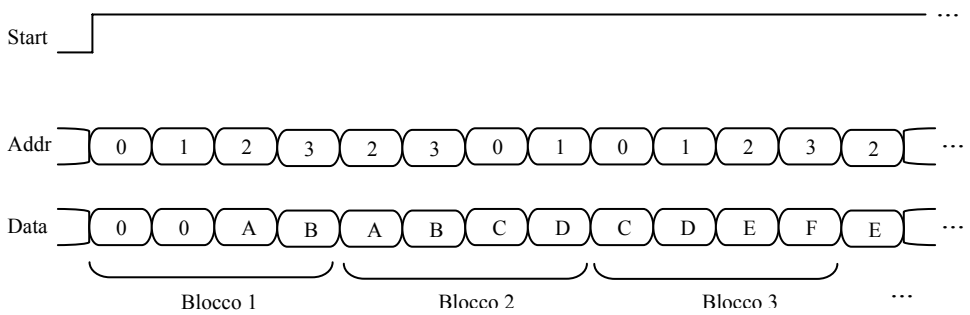


Figura 4.1: Lettura della memoria di Input secondo l'algoritmo overlap - save (dimensione del blocco N=4).

4.2 Catena di moltiplicazione – accumulo

La catena di moltiplicazione – accumulo si basa su un modulo MAC (*multiply and accumulate*) iterato un numero di volte pari alle partizioni della risposta impulsiva [Gray, 2003]. La figura 4.2 riporta una visione schematica della catena.

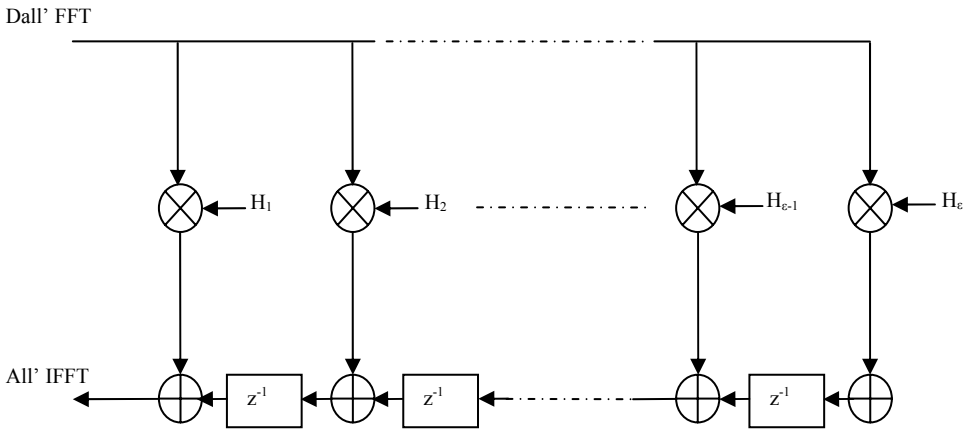


Figura 4.2: Schematizzazione del core di convoluzione.

Ciascun ramo provvede alla moltiplicazione di una partizione della risposta impulsiva con i campioni trasformati. Un sommatore procede poi all'addizione del risultato ottenuto con l'uscita del ramo a monte relativa al campione corrispondente come posizione nella partizione degli ingressi,

ma appartenente al blocco precedente. Di seguito analizzeremo i moduli costituenti l'architettura di un singolo ramo. Da ricordare che per quanto riguarda la programmabilità, l'utente dispone di un *package* nel quale può definire:

- numero di partizioni della IR;
- numero di *bit* dei campioni della IR;
- numero di *bit* dei campioni di ingresso;
- numero minimo di *bit* per l'aritmetica interna;
- numero dei *bit* dei campioni in uscita;

Sono presenti inoltre alcune costanti di utilità ricavate sotto forma di combinazioni lineari dei valori appena citati.

4.2.1 Memorie della IR

*R*ealizzate con il *tool* generatore di *Xilinx* sono due memorie RAM (una per la parte reale e una per l'immaginaria) con inizializzazione a zero e con fasi di scrittura e lettura non sovrapposte (v. capitolo 5). Ciascuna *word* corrisponde alla dimensione di un campione in frequenza della IR e la profondità è determinata dalla dimensione della FFT (dimensione del blocco dell'*overlap – save*).

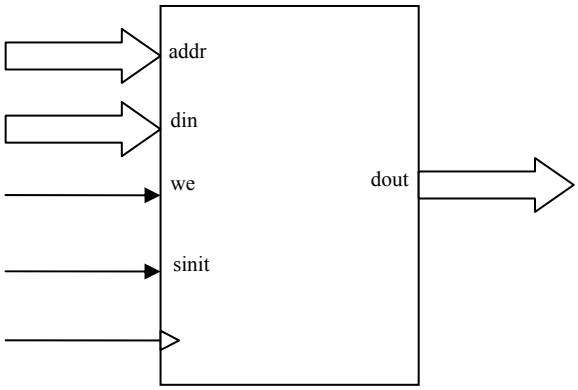


Figura 4.3: La memoria della risposta impulsiva.

4.2.2 Gestore delle memorie della IR

Gestisce le memorie che contengono i campioni in frequenza della risposta impulsiva. Agisce in due fasi distinte. Nella prima fase carica le memorie con i campioni provenienti dall'interfaccia di comunicazione (*bus* PCI). L'operazione deve essere segnalata settando il segnale *ir_charge* prima di inviare il primo campione. Il modulo risponderà all'utente mantenendo settato il segnale *charging* per notificare la ricezione dei dati in corso. Contestualmente il blocco invia alle memorie il segnale *we* di abilitazione alla scrittura. L'operazione termina con la disattivazione di *charging* e di *we*.

La seconda fase inizia con il settaggio da PCI del segnale di *start* e consiste nell'invio sequenziale al moltiplicatore dei dati contenuti nelle memorie. Questo avviene generando, ad ogni ciclo di *clock*, indirizzi successivi per le memorie interessate. È prevista la modalità *burst* nel caso di processamento di più vettori di dati. La procedura si arresta una volta campionato il livello alto sul segnale *stop* (proveniente dal PC *host*).

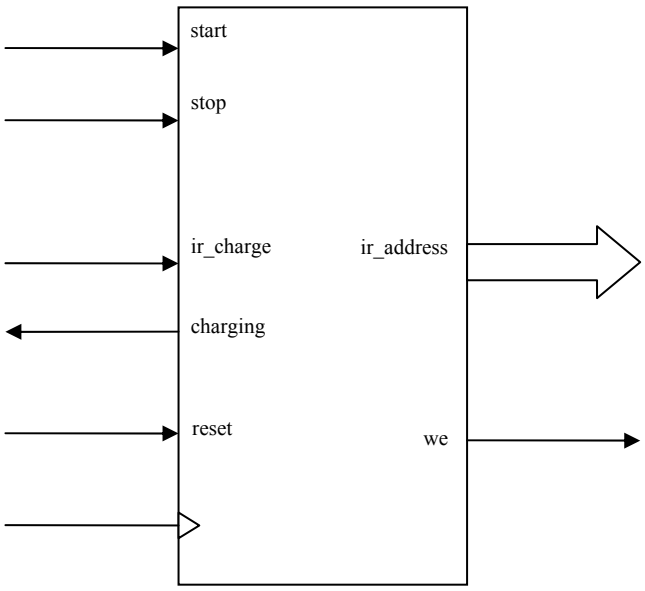


Figura 4.4: Il gestore delle memorie della IR.

4.2.3 Moltiplicatore

Questo blocco esegue una moltiplicazione complessa tra i dati provenienti dal processore FFT ed i campioni della risposta impulsiva. Per non compromettere già in questa prima fase la precisione, ciascun risultato (reale ed immaginario) viene fornito su un numero di *bit* pari alla somma dei *bit* degli ingressi (ovviamente parte reale ed immaginaria di ciascun ingresso hanno il medesimo numero di *bit*). Il moltiplicatore presenta una latenza di due cicli di *clock* in virtù della *pipeline* interna.

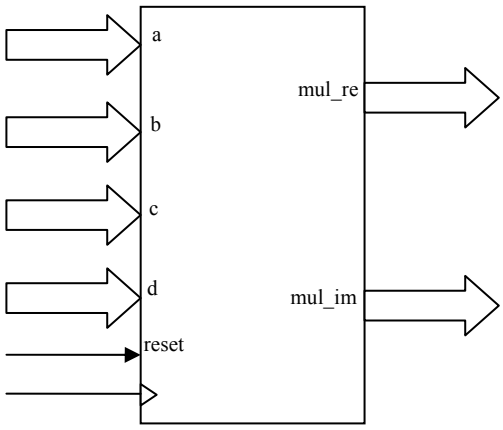


Figura 4.5: Il moltiplicatore.

4.2.4 Sommatore

Questo blocco realizza una somma complessa tra il risultato *k-esimo* proveniente dal moltiplicatore e l'uscita k-esima del ramo a monte relativa alla partizione precedente. Il risultato viene fornito su un numero di *bit* pari alla maggiore tra le dimensioni dei due ingressi, aumentato di un'unità. Questa quantità, che garantisce un'esatta precisione, è automaticamente calcolata nel *package* operando l'algoritmo seguente:

$$\frac{(bit\_in1 + bit\_in2) + |bit\_in1 - bit\_in2|}{2} + 1 = \max\{bit\_in1, bit\_in2\} + 1$$

Anche il sommatore ha una latenza di due cicli di *clock*, per i medesimi motivi esposti nella descrizione del moltiplicatore.

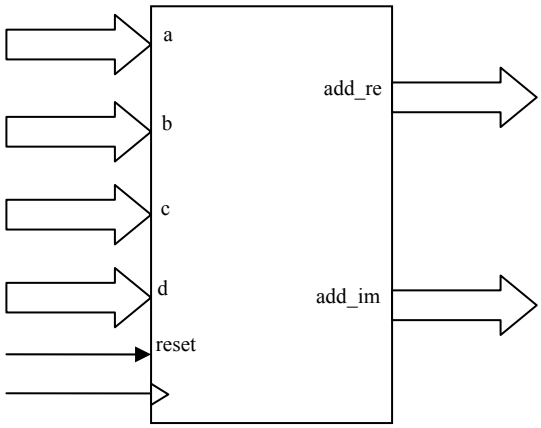


Figura 4.6: Il sommatore.

4.2.5 Elemento di ritardo

Costituito da due memorie RAM *dual – port* tra loro identiche (reale ed immaginaria), con inizializzazione a zero e modalità di lettura e scrittura non sovrapposte (v. capitolo 5). Per capire il funzionamento di questi elementi di ritardo dobbiamo considerare due indici temporali: uno a livello dei campioni contenuti in una singola partizione dell'ingresso e uno a livello delle partizioni stesse. All'inizio le memorie verranno scritte con ciascun valore ottenuto per ogni campione, successivamente verranno rilette in modo che al campione *k-esimo* del blocco *i-esimo* venga sommato il campione *k-esimo* del blocco *i-1 esimo*. Questi elementi di ritardo dunque, a regime, saranno scritti ad una certa locazione (corrispondente al campione successivo rispetto a quello in elaborazione nel ramo a valle) e letti alla precedente (corrispondente al campione attuale).

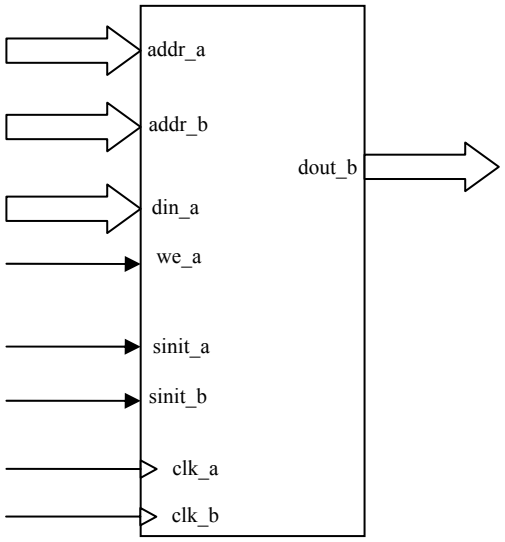


Figura 4.7: La memoria per la generazione dei ritardi (la porta a è di sola scrittura e la b di sola lettura).

4.2.6 Gestore dei ritardi

Questo blocco si occupa della gestione delle memorie poc'anzi analizzate. Attraverso un segnale di *start* generato dell'utente, il modulo innanzitutto segnala l'avvenuta ricezione dell'impulso di partenza settando per un ciclo di *clock* il segnale *ack*. In seguito esso abilita alla scrittura entrambe le memorie (segnale *we_a*) ed invia loro gli indirizzi per la scrittura e la lettura, incrementati ad ogni ciclo di *clock*. La procedura si arresta una volta campionato il livello alto sul segnale *stop*, a cura dell'utente.

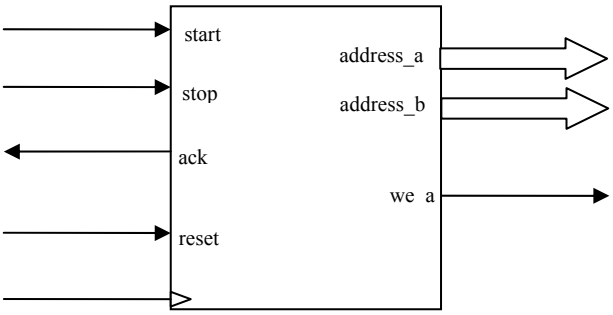


Figura 4.87: Il gestore delle memorie delle uscite dei rami a monte.

4.2.7 Il blocco di approssimazione

Questo blocco è dedicato all'approssimazione dei dati provenienti dal sommatore per poter rappresentare l'uscita sul numero di *bit* richiesti dall'utente. Da notare che è ragionevole supporre che questa specifica darà senz'altro luogo ad una decimazione poiché, a causa del moltiplicatore e del sommatore, il numero di *bit* provenienti dal processore FFT risulterà senz'altro più che raddoppiato. Il sistema acquisisce i dati dal sommatore e li approssima troncando i *bit* più significativi (corrispondentemente a quanti desiderati in uscita) e somma loro il primo *bit* tra quelli esclusi. Se la dinamica non è sufficiente alla rappresentazione del dato, le uscite saturano al maggior valore rappresentabile (tutti i *bit* a uno). A causa dei registri sull'uscita anche questo blocco dà luogo ad un ciclo di latenza.

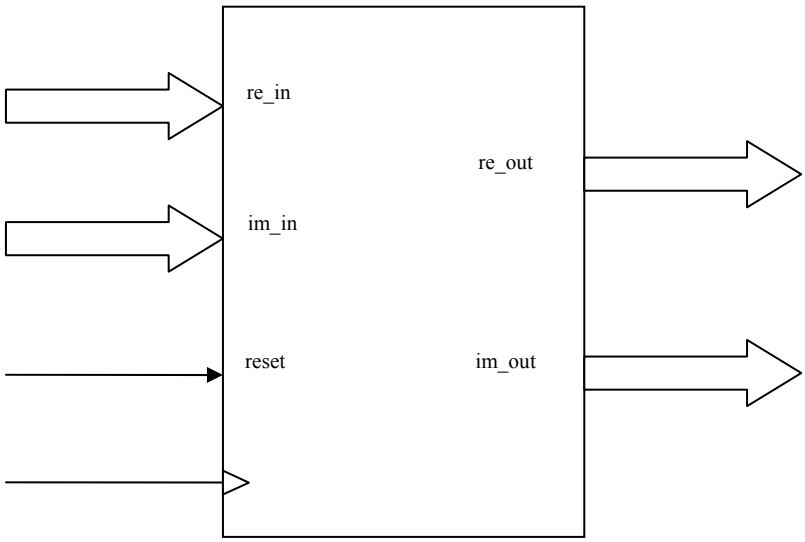


Figura 4.9: Il blocco di approssimazione.

Segue il diagramma completo di un ramo (figura 4.10).

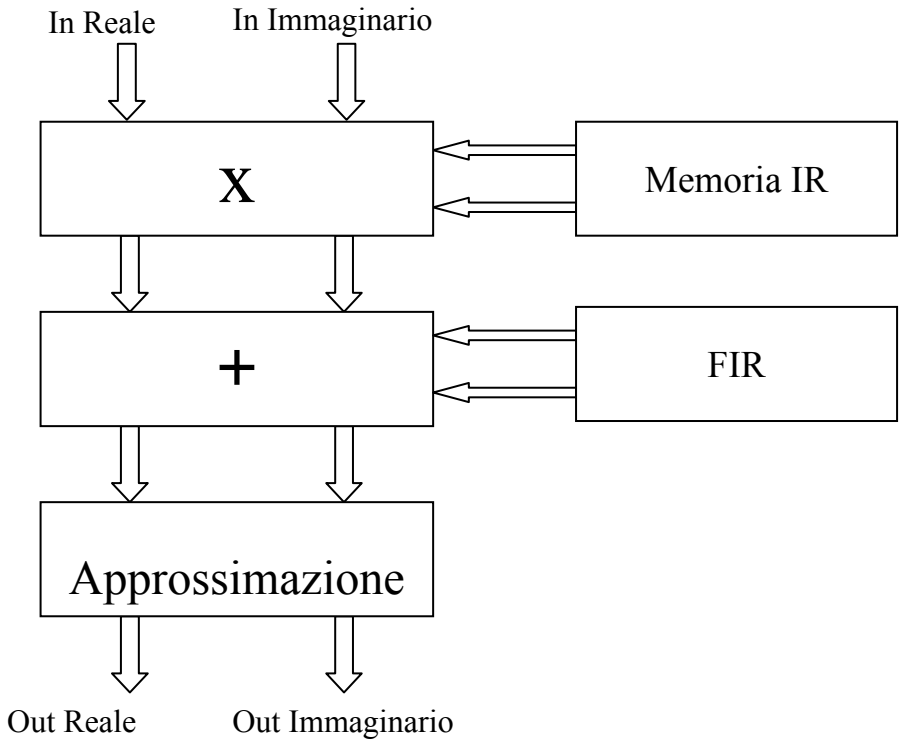


Figura 4.10: Architettura di un ramo del convolutore.

4.3 Architettura completa

Presentiamo adesso l'architettura completa, rappresentata in figura 4.11 nel caso di partizionamento in tre blocchi della IR.

Come era ovvio attendersi, il ramo a monte di tutti gli altri ha un ingresso del sommatore fissato allo zero logico poiché non esistono stadi precedenti a questo. Non è stato del tutto abolito il blocco sommatore perché altrimenti avremmo avuto una latenza diversa per questo stadio e ciò avrebbe reso difficile una corretta sincronizzazione.

La latenza totale dunque, considerando il tempo che intercorre tra un dato generato dal processore FFT e l'uscita generata dal decimatore, risulta essere pari a cinque cicli di *clock*.

Come si può vedere in figura 4.11, il gestore delle memorie della IR ed il gestore dei ritardi sono unici per l'intera architettura poiché, generando solo gli indirizzi e le abilitazioni senza particolari accorgimenti di temporizzazione, non richiedono di essere customizzati per ogni singolo ramo.

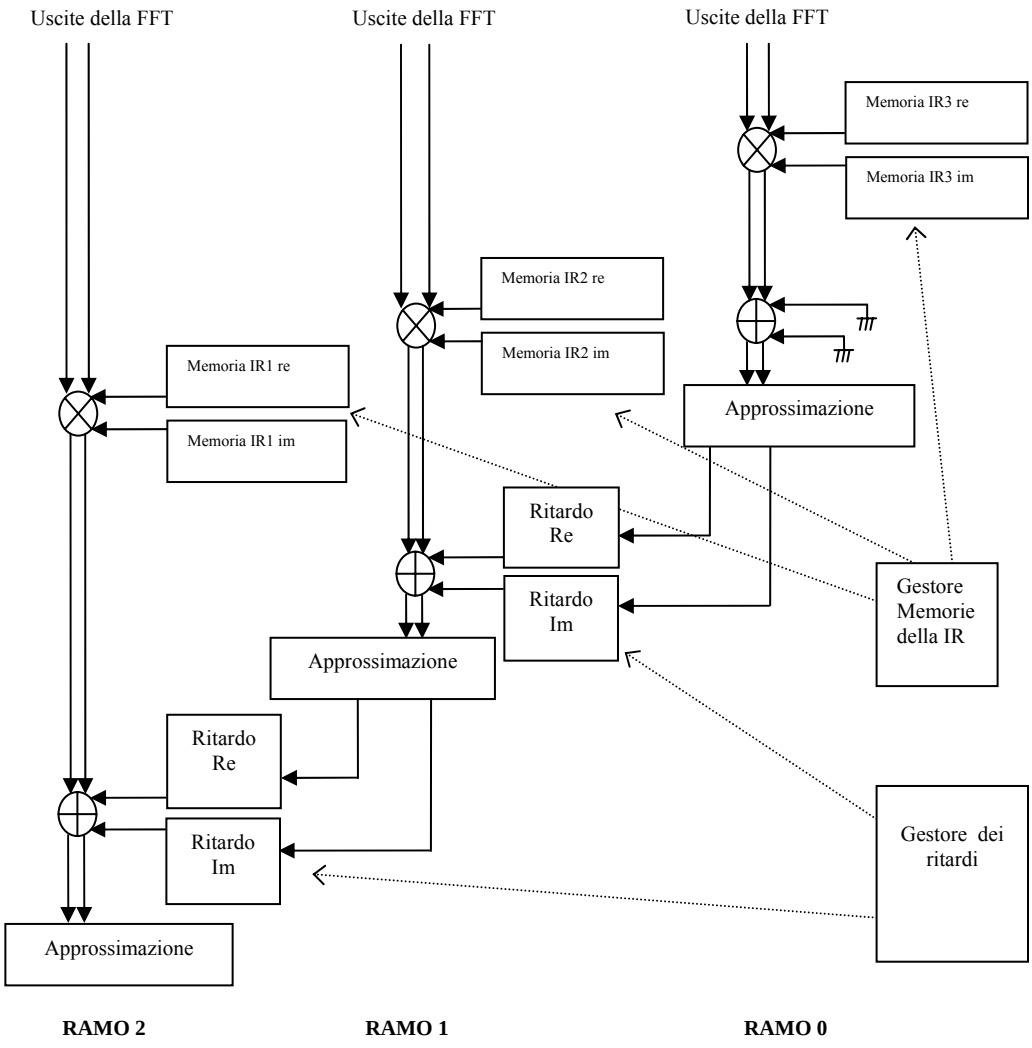


Figura 4.11: La catena di moltiplicazione – accumulo nel caso di una IR partita in tre blocchi .

CAPITOLO 5
LA PIATTAFORMA DI PROTOTIPAZIONE

In questo capitolo esamineremo le caratteristiche della piattaforma di prototipazione, descrivendo brevemente le risorse disponibili, sia per quanto riguarda gli FPGA che i banchi di memoria esterni. Sarà analizzata anche l'interfaccia per la comunicazione con il PC *host*, tramite *bus* PCI, completa dei protocolli per il trasferimento dei dati, realizzata appositamente per l'applicazione.

5.1 *Overview* della piattaforma

La scheda utilizzata per la realizzazione dell'acceleratore *hardware* è una *Nallatech BenNUEY-PCI*, equipaggiata con due moduli di espansione *BenBLUE-III* [NAL, 2005].
Principali caratteristiche:

- FPGA *Xilinx Spartan-II* (XC2S200) preconfigurato per il controllo a basso livello del *bus* PCI [XIL, 2005a]; questo dispositivo non è accessibile all'utente.

- Un FPGA *Xilinx Virtex-II* (XC2V8000) completamente programmabile per l'applicazione [XIL, 2005b].
- Tre moduli di espansione di tipo DIME-II.
- Interfaccia PCI da 64 bit/33MHz.
- Tre generatori di *clock* programmabili utilizzabili anche per il pilotaggio di componenti esterni.
- Otto LED bicolore.
- Due banchi da 2 *Mbyte* di SSRAM ZBT.
- *Software Nallatech* FUSE per la configurazione degli FPGA via PCI.
- Libreria *software Nallatech* FUSE per l'interfacciamento e il controllo della scheda.

Quanto esposto finora può essere può essere riscontrato in figura 5.1, dove ogni elemento è stato individuato nella sua posizione sulla scheda.

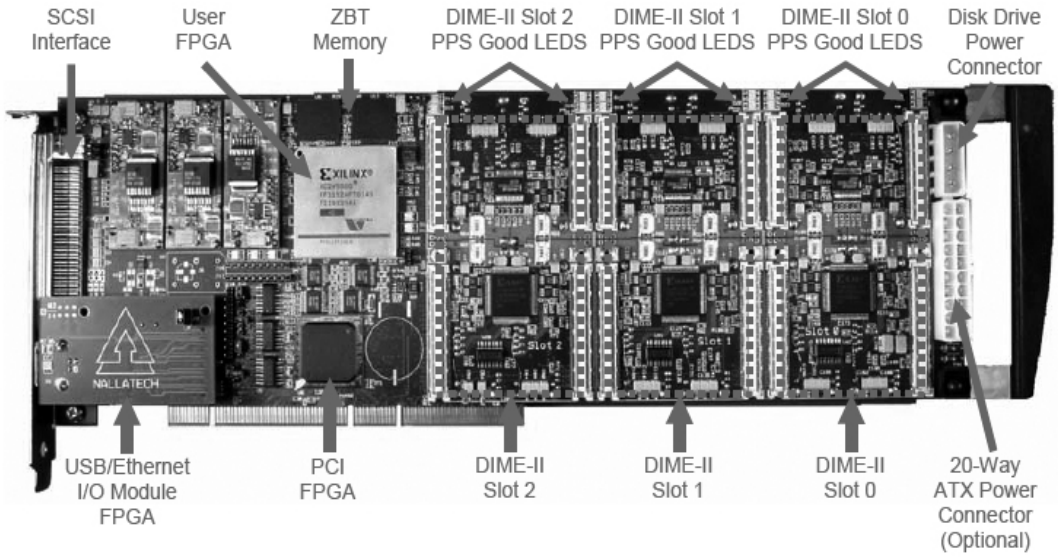


Figura 5.1: La scheda di sviluppo ed i suoi componenti principali.

Uno slot di espansione è stato popolato con un modulo BenBLUE-III. Un modulo BenBLUE-III consiste in due FPGA *Xilinx Virtex-II* PRO (XC2VP100) e 72 *Mbyte* di ZBT SRAM montati come otto banchi indipendenti; ciascun FPGA può accedere solamente a quattro degli otto

banchi a disposizione. La scelta di questo modulo è stata compiuta in considerazione del fatto che la necessità di calcolo richiesta dall'applicazione era molto elevata ed era necessario non partizionare eccessivamente l'architettura su troppi FPGA. Infatti si tenga presente che gli XC2VP100 hanno una capacità notevolmente maggiore degli XC2V8000, sebbene questo non sia immediatamente confrontabile data la diversa architettura dei CLB dei due dispositivi [NAL, 2004a].
In figura 5.2 è mostrato uno schema di un singolo modulo BenBLUE-III.

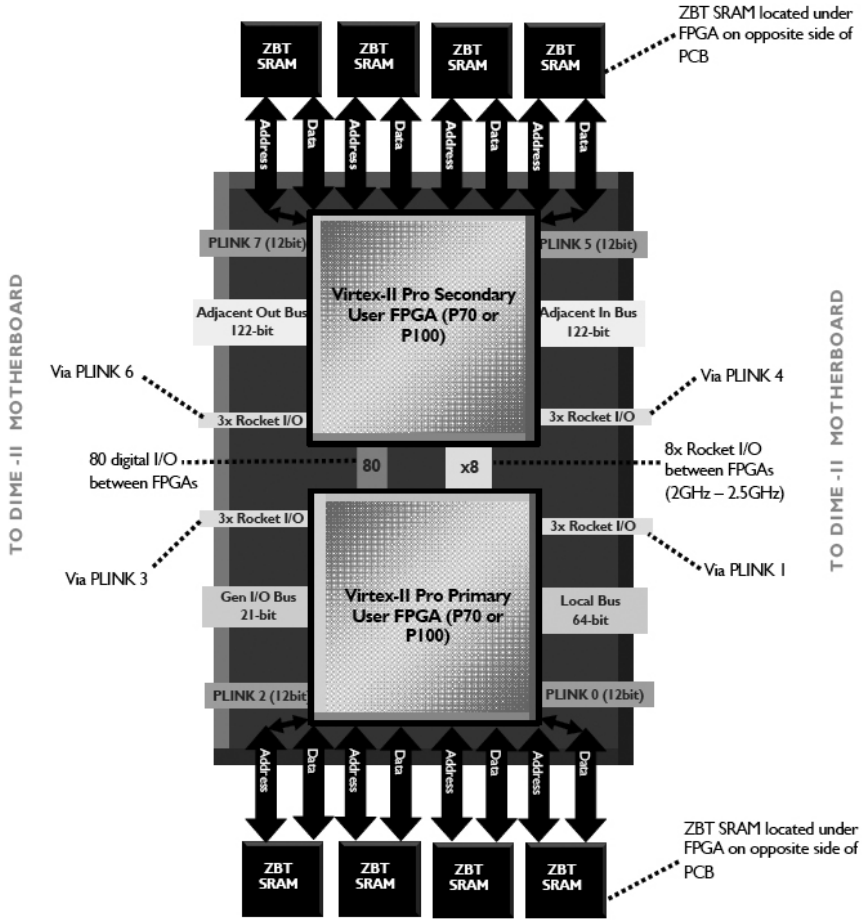


Figura 5.2: Il modulo di espansione BenBLUE-III.

Di seguito un diagramma a blocchi che schematizza l'architettura completa della piattaforma di prototipazione.

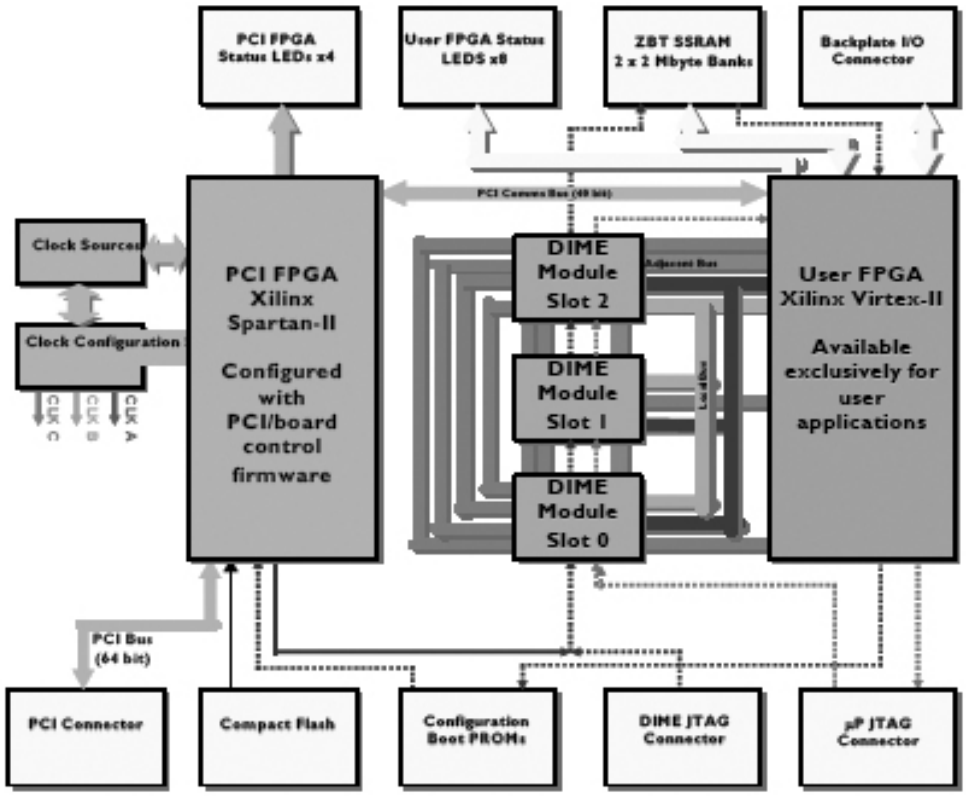


Figura 5.3: La scheda di sviluppo ed i suoi componenti principali.

5.2 L'FPGA XC2VP100

Abbiamo scelto di implementare l'applicazione su questo FPGA poiché l'XC2V8000 non aveva risorse di memoria sufficienti (v. capitolo 6). Andremo adesso a dare una breve descrizione delle caratteristiche principali del dispositivo.

5.2.1 Le memorie

L'FPGA contiene delle memorie *embedded*, ossia banchi *on-chip* all'interno del dispositivo stesso, di dimensioni fino a 7992 *Kbits*. Questo tipo di memorie sono state impiegate per le partizioni della IR e per le ROM dei coefficienti di *twiddle* (v. capitoli 3 e 4). Vediamo adesso in dettaglio le caratteristiche di ciascun modello a disposizione.

I blocchi RAM (v. figura 5.4) sono totalmente sincroni e possono essere *single-port* o *dual-port*. Ciascun blocco può essere connesso ad un altro per formare memorie più grandi.

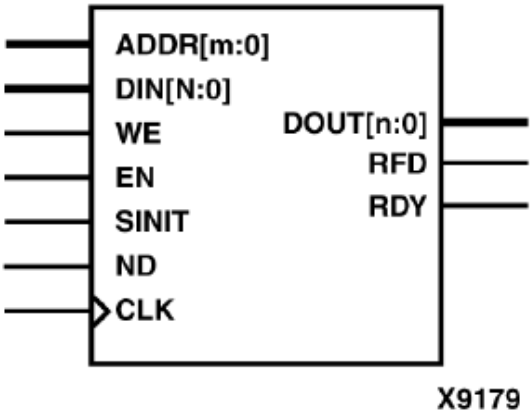


Figura 5.4: RAM single-port.

La versione *dual-port* ha due porte funzionalmente identiche ma ciascuna è dotata ingressi e uscite del tutto indipendenti, come si può osservare in figura 5.5.

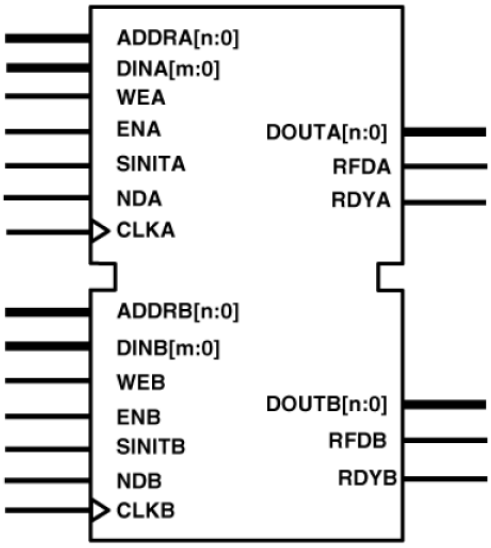


Figura 5.5: RAM dual-port.

Possiamo dunque esporre le caratteristiche principali della versione a singola porta, visto che la versione a due porte ha in più solo una replica della piedinatura.

Le uscite delle memorie possono essere forzate ad un valore specifico (stabilito in fase di generazione) attraverso il segnale *SINIT*. Per quanto riguarda gli altri ingressi, abbiamo i *pin* dedicati all'abilitazione globale (*EN*) e alla scrittura (*WEN*), il *bus* indirizzi (*ADDR*), il *bus* dati (*DIN*), il *clock* (*CLK*) ed infine il segnale che indica la presenza di nuovi dati validi in ingresso (*ND*). In uscita avremo i dati letti (*DOUT*), il segnale che indica che la memoria è pronta a ricevere un nuovo dato (*RFD*) ed infine il segnale che notifica dati validi sul *bus* di uscita (*RDY*).
È inoltre possibile aggiungere fino a 2 livelli di *pipeline* in uscita, oltre a quello presente di *default*.
Per quanto riguarda le RAM *dual-port*, le letture simultanee dalla stessa locazione sono possibili, al contrario delle scritture che invece sono proibite. Per sincronizzare le due fasi esistono comunque tre modalità possibili tra le quali scegliere:

- *No read on write*: I processi di lettura e scrittura non possono mai avvenire simultaneamente.

- *Read after write*: Il dato è simultaneamente immagazzinato in memoria ed inviato ai registri di uscita.

- *Read before write*: Il dato precedentemente immagazzinato nella locazione è memorizzato nei registri di uscita e contemporaneamente la locazione è sovrascritta.

Le memorie RAM, scegliendo in fase di generazione l'opzione *Read Only*, sono implementate come ROM. La piedinatura rimane pressoché la medesima, fatta esclusione per i *pin* relativi alla scrittura.
Altre risorse a disposizione sono le *Distribuited Memory*, le cui celle sono ricavate utilizzando i *flip-flop* all'interno dei CLB (*Configurable Logic Blocks*) dell'FPGA (v. paragrafo 5.2.3). Questo tipo di risorsa si utilizza

solitamente per l'implementazione di piccoli *buffer* (es. FIFO), consentendo una maggiore velocità in virtù della località della memoria.

5.2.2 Le memorie ZBT

Come introdotto nel capitolo 2, oltre ai tipi finora descritti, necessitiamo di un altro tipo di memorie che supportino la modalità *burst* e risultino estremamente veloci, al fine di immagazzinare i vettori di ingresso e di uscita dell'intero sistema, durante le fasi di comunicazione con il PC *host*. Si ricorre, come accennato, alle ZBT – SRAM (*Zero Bus Turnaround*) che sono delle memorie sincrone ad alta velocità realizzate appositamente per la riduzione della latenza tra operazioni di scrittura e di lettura [XIL, 2000], [MIC, 2000]. Queste risorse sono posizionate sul modulo di espansione, esternamente agli FPGA, ma sono collegate ciascuna ad un dispositivo specifico (v. figura 5.6).
Come indicato dall'acronimo, non sono necessari cicli di *idle* tra le operazioni di lettura e scrittura, consentendo così un aumento della banda (v. figura 5.7). Questo tipo di memorie possono funzionare in modalità *burst*, grazie ad un contatore interno.

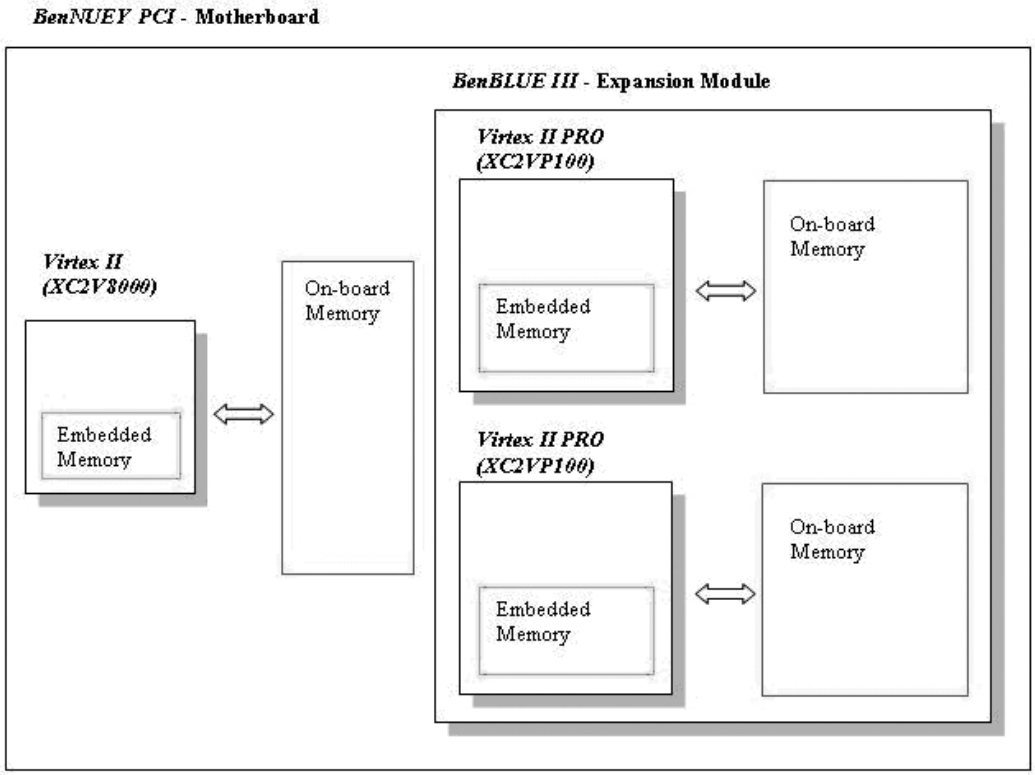


Figura 5.6: Le risorse di memoria sulla scheda Nallatech BenNUEY-PCI.

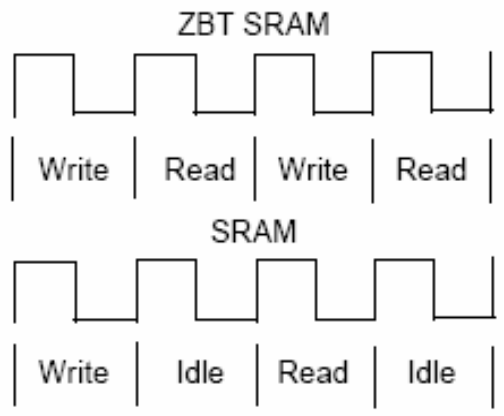


Figura 5.7: Differenza nelle operazioni tra ZBT SRAM e SRAM.

Per quanto riguarda gli ingressi, come possiamo vedere in figura 5.8, abbiamo: il *clock* (*CLK*), il *bus* indirizzi (*ADDR*), quattro abilitazioni - alla scrittura (*WE*), alla scrittura dei singoli *byte* ($BWE_{a/b/c/d}$), delle uscite (*OE*) e globale (*EN*) -, il segnale di avanzamento del contatore *on-chip* o

caricamento di un nuovo indirizzo (*ADV/LD#*) ed infine il selettore della modalità del *burst* (*MODE*). In uscita abbiamo due *bus* bidirezionali contenenti uno i dati e l'altro i *bit* di parità, considerati esattamente allo stesso modo.
Tutti i segnali sono completamente registrati, sia in ingresso che in uscita, come le *embedded* RAM vista in precedenza.
I parametri di *timing* sono i seguenti:

- Massima frequenza di *clock on-board* : 166 MHz (periodo: 6 ns).
- Massimo tempo di accesso : 3.5 ns (@ f_{clkmax} = 166 MHz).

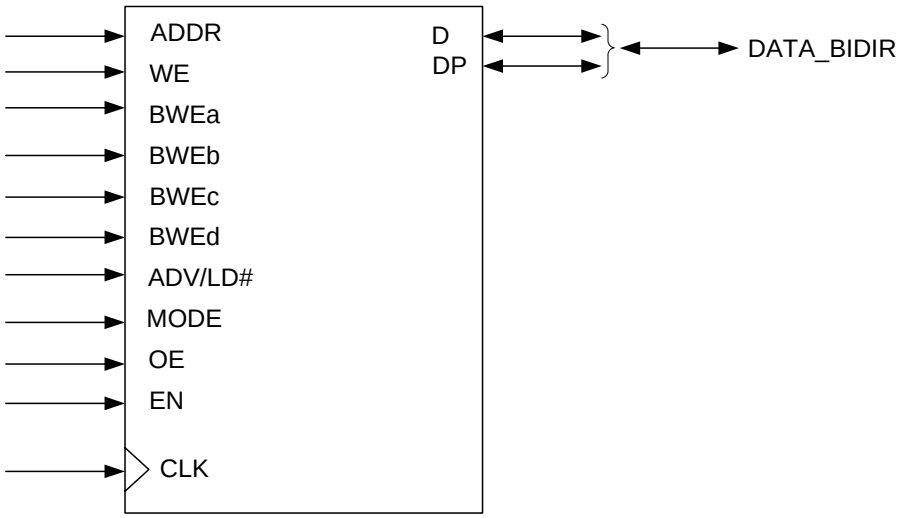


Figura 5.8: La memoria ZBT SRAM.

5.2.3 I CLB

L'FPGA da noi considerato è costituito dai blocchi logici configurabili (CLB – *Configurable Logic Blocks*) mostrati in figura 5.9. Essi sono organizzati in *array* e sono impiegati per l'implementazione di funzioni sia combinatorie che sequenziali. Ciascun CLB è connesso ad una matrice di *switch* che forma un collegamento con le risorse di connessione del *chip*.

Ogni CLB comprende quattro *slice* a loro volta suddivisi in due colonne. Ciascun *slice* contiene due generatori di funzioni con ingressi da 4 *bit*, basati su *look-up-table*, capaci di implementare una qualsiasi funzione booleana. Sono disponibili inoltre due tipi di *multiplexer* capaci di gestire fino a 8 ingressi e due registri configurabili (v. figura 5.10).

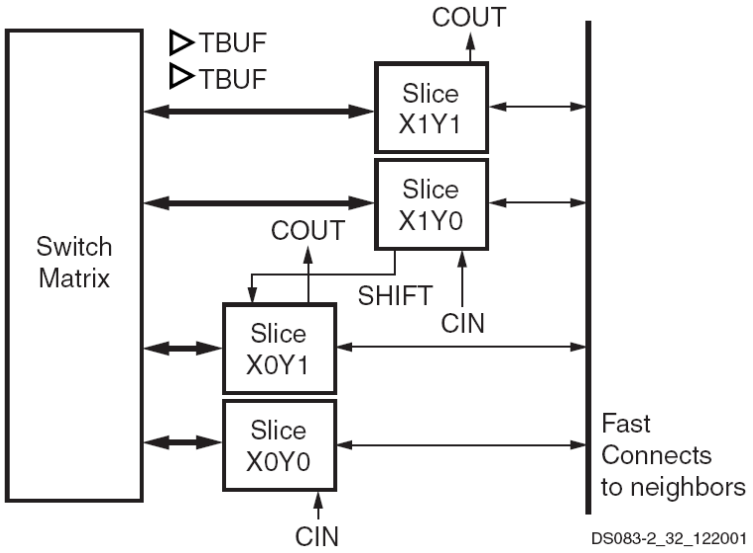


Figura 5.9: I CLB.

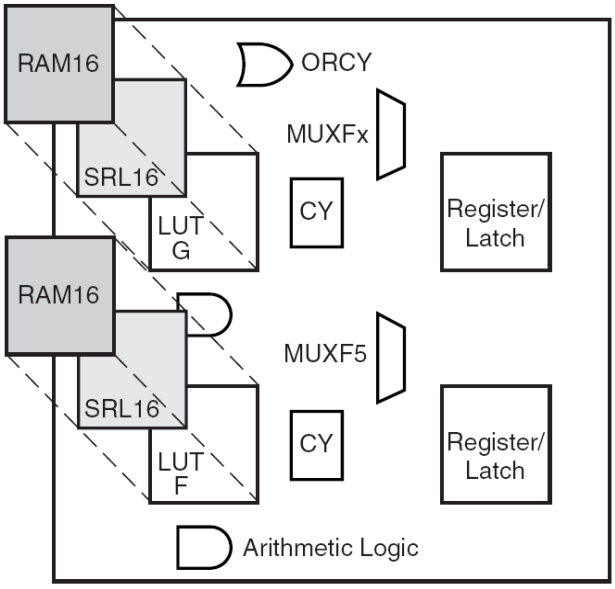


Figura 5.10: Gli slice all'interno dei CLB.

5.2.4 I moltiplicatori

L'FPGA XC2VP100 contiene 444 moltiplicatori da 18 *bit* x 18 *bit* in complemento a due. Queste risorse sono ottimizzate per garantire alte velocità di computazione e basso consumo di potenza. Se queste risorse non fossero sufficienti per l'applicazione o se, al contrario, essa richiedesse un numero estremamente esiguo di moltiplicatori, i CLB possono essere programmati per realizzare delle risorse logiche capaci di implementare le operazioni di prodotto necessarie.

5.3 La comunicazione *host*/scheda

5.3.1 L'interfaccia di comunicazione

In figura 5.11 è mostrata l'architettura completa del modulo per la gestione delle comunicazioni tra PCI e applicazione.

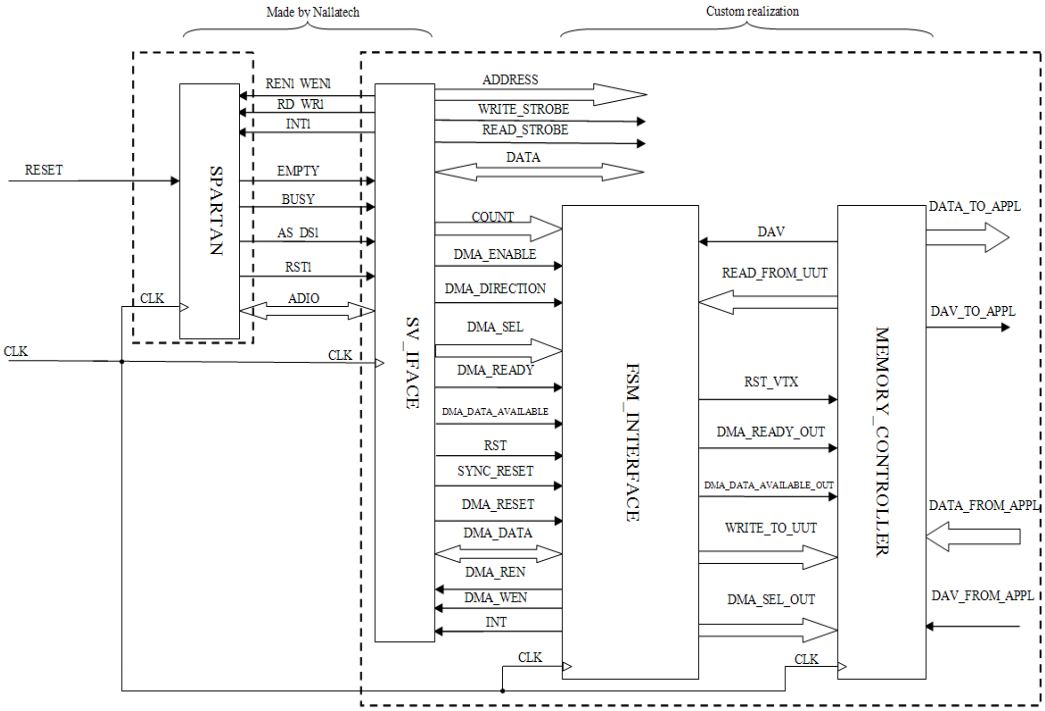


Figura 5.11 : Architettura completa dell'interfaccia di comunicazione

L'architettura di questo modulo consiste in cinque blocchi principali:

1. *PCI Communications Interface*
2. *Memory Map Interface*
3. *DMA Interface*
4. *FSM Interface*
5. *Memory Manager*

5.3.2 La PCI Communication Interface

Fornita da *Nallatech* assieme alla piattaforma di sviluppo, è realizzata sul modulo *Spartan* e non necessita dunque di sintesi. Questa interfaccia pilota a basso livello il bus PCI e comunica direttamente con il *bus* stesso. La prima fase di *testing* è stata resa possibile attraverso un modello funzionale messo a disposizione dalla casa madre [NAL, 2004b].

5.3.3 La *Memory Map Interface*

Si trova sull'FPGA dedicato all'applicazione e deve essere sintetizzata assieme al progetto da realizzare. La *Nallatech* comunque ne fornisce tutti i *file* VHDL necessari. Questo blocco fa parte della cosiddetta SV *Interface* (*Spartan* to *Virtex*). Necessita del settaggio di tre *generic*, attraverso i quali si dà una descrizione dello spazio di memoria impiegato:

- *Num_blocks*: numero di *bit* necessari ad individuare i blocchi di suddivisione.
- *Block_size*: numero di *bit* necessari ad individuare una locazione all'interno di un blocco.
- *Num_regs*: numero totale di registri presenti nello spazio di memoria.

In figura 5.12 è mostrata l'interpretazione di un indirizzo.

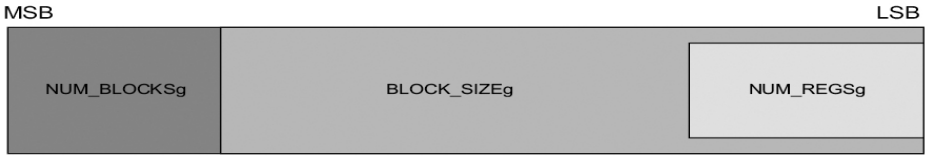


Figura 5.12: Suddivisione dei campi di un indirizzo.

Questa tecnica risulta molto conveniente dal punto di vista della logica di decodifica nel caso di grandi blocchi con un ridotto numero di registri.

5.3.4 La *DMA Interface*

Si trova anch'essa sull'FPGA dedicato all'applicazione ed è fornita dello stesso tipo di documentazione del blocco precedente, poiché costituente la seconda ed ultima parte della *SV Interface*. Permette il trasferimento di larghe porzioni di dati, anziché di una singola *word*. È costituita principalmente da due FIFO, una di ingresso e una di uscita. Possiede inoltre due registri da 32 *bit* dedicati alla sua programmazione, posti obbligatoriamente agli indirizzi 0x00000000 e 0x00000001 dello spazio di memoria. Il primo registro è detto CSR (*Control/Status Register*) ed il secondo è detto *Counter Register*. Il CSR serve a settare il controllore DMA (abilitazione, direzione del bus dati, *reset*…), secondo quanto mostrato nella tabella di figura 5.13. Vedremo più in dettaglio la funzione di ciascun *bit* quando in seguito parleremo dei protocolli di trasferimento dei dati.

Label	Read/Write	Number of Bits	Register Bits
DMA Enable	R/W	1	0
DMA Direction	R/W	1	1
DMA Select	R/W	4	2-5
DMA Reset	R/W	1	6
DMA In FIFO Full	R	1	7
DMA In FIFO Empty	R	1	8
DMA Out FIFO Full	R	1	9
DMA Out FIFO Empty	R	1	10
RESERVED	-	21	11-31

Figura 5.13: Bit del Control/Staus Register del controllore DMA.

Il *Counter Register* contiene il numero delle *words* ancora da trasferire nell'operazione corrente, per cui all'inizio esso conterrà il numero totale dei dati coinvolti nella trasmissione. Esistono in realtà altri due contatori per i dati trasferiti da e verso le FIFO del DMA.

Come *generic*, occorre settare il valore di *WAIT_COUNT* che corrisponde ai cicli di *clock* da attendere nell'acquisizione di due dati successivi dalle FIFIO, nel caso di una lettura dalla memoria da parte del PCI via DMA.

Di questo blocco e dei due precedenti possiamo vedere una rappresentazione schematica in figura 5.14, dove risulta chiara la suddivisione tra i due FPGA.

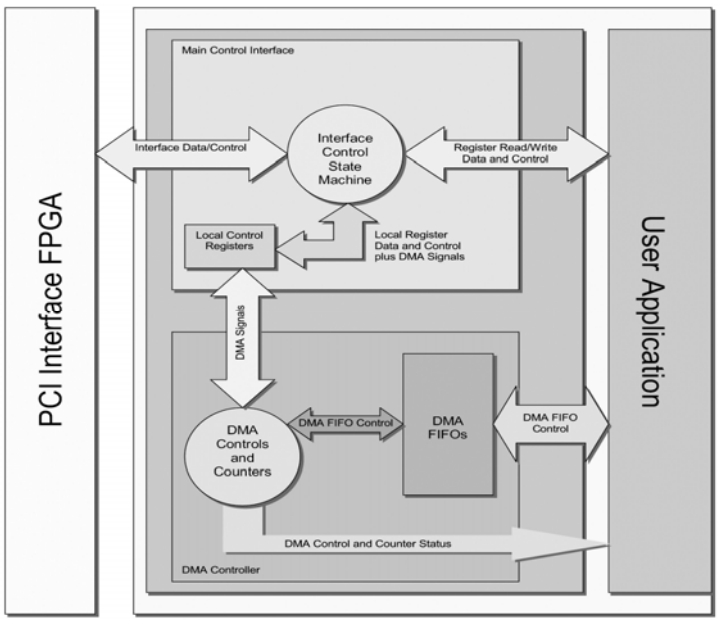


Figura 5.14: Diagramma a blocchi delle interfacce fornite da Nallatech.

5.3.5 La *FSM Interface*

Tenendo presente quanto fornito sulla scheda di sviluppo, abbiamo realizzato questo modulo, mostrato in figura 5.15, volto alla comunicazione tra *DMA Interface* e *Memory Manager*. L'acronimo FSM sta per *Finite State Machine*, entità logica su cui questo modulo si basa.

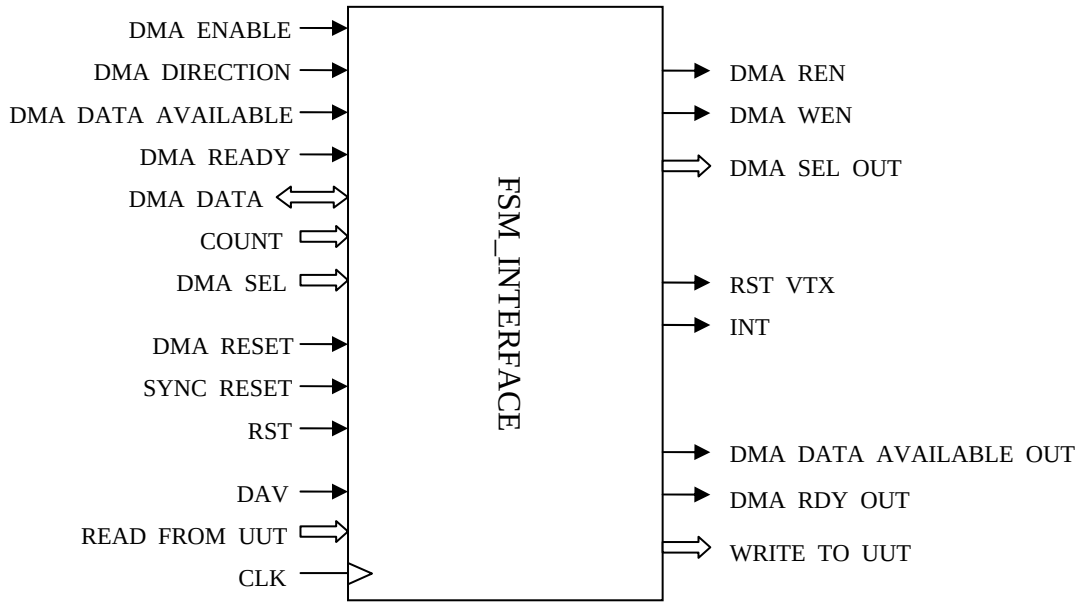


Figura 5.15 : Il blocco FSM Interface.

I segnali di ingresso sono:

- *DMA_ENABLE*: *bit* contenuto nel CSR (*Control Status Register*) del controllore DMA, del quale segnala l'abilitazione.
- *DMA_DIRECTION*: *bit* contenuto nel CSR che segnala la direzione del trasferimento dati sul *bus*. Quando vale 0 significa chi i dati scorrono dal controllore verso l'interfaccia, mentre quando vale 1 si ha il verso opposto.
- *DMA_DATA_AVAILABLE*: segnale che indica la disponibilità sul *bus* dei dati inviati dal controllore DMA.

- *DMA_READY*: segnale che indica che il controllore DMA è pronto per leggere dati dal *bus*.
- *DMA_DATA*: *bus* da 32 bit per il trasferimento dei dati tra interfaccia e controllore DMA.
- *COUNT*: dato contenuto nel CSR che indica il numero di *words* che devono essere ancora trasferite, sia in un senso che nell'altro.
- *DMA_SEL*: vettore da 4 *bit* che indica su quale dei 16 canali DMA possibili sta avvenendo il trasferimento dati.
- *DMA_RESET*: *bit* contenuto nel CSR che indica il *reset* del controllore DMA.
- *SYNC_RESET*: segnale di *reset* sincrono.
- *RST*: segnale di *reset* per l'FPGA.
- *DAV*: segnale attraverso il quale il gestore delle memorie segnala la presenza del dato richiesto su *READ_FROM_UUT*.
- *READ_FROM_UUT*: vettore da 32 *bit* attraverso il quale il gestore delle memorie comunica i dati richiesti.
- *CLK*: segnale di *clock*.

I segnali di uscita sono:

- *DMA_REN*: segnale attraverso il quale l'interfaccia comunica al controllore DMA la fase di lettura di un dato dal *bus*.
- *DMA_WEN*: segnale attraverso il quale l'interfaccia abilita la lettura da parte del controllore DMA di un dato presente sul *bus*.
- *DMA_SEL_OUT*: Copia di DMA_SEL da fornire al gestore delle memorie.
- *RST_VTX*: copia di *RST* da inviare agli FPGA *Virtex*.
- *INT*: Segnale per la richiesta di *interrupt*. Considerando l'architettura del modulo *Spartan*, questo segnale non è stato preso in considerazione ed è stato dunque fissato allo zero logico.

- *DMA_DATA_AVAILABLE_OUT*: segnale attraverso il quale si inoltra al gestore la richiesta di scrittura in memoria.
- *DMA_READY_OUT*: segnale attraverso il quale si inoltra al gestore la richiesta di lettura dalle memorie.
- *WRITE_TO_UUT*: vettore da 32 *bit* attraverso il quale vengono passati i dati al gestore delle memorie.

5.3.6 Il Memory Manager

Questo blocco gestisce l'accesso alle memorie presenti sulla piattaforma, sia quelle *embedded* presenti sul modulo FPGA, che quelle esterne presenti sulla scheda stessa (ZBT). Il modulo è stato realizzato sulla base delle specifiche del blocco FSM *Interface* per permettere la fase finale di memorizzazione degli ingressi destinati all'applicazione e delle uscite che essa genererà a partire da questi, nonché della lettura di queste ultime per poterle trasmettere al PCI e renderle così visibile all'utente. L'architettura dettagliata è mostrata in figura 5.16.

I dati provenienti dall'*host*, sono smistati dal *Memory Manager* verso il controllore della ZBT per poi essere memorizzati nelle ZBT stesse, le quali supportano la scrittura in modalità *burst*. Il gestore delle memorie provvede poi, una volta completati tutti i cicli di scrittura, all'invio dei dati presenti nelle ZBT al *buffer* di ingresso (*buff_in_ram*), così da poter essere acquisiti dall'applicazione. Durante l'elaborazione, il gestore inserisce i dati generati nel *buffer* di uscita (*buff_out_ram*) per poi inviarne il contenuto alla ZBT attraverso il loro controllore, così da poter gestire in seguito la comunicazione *burst* verso l'*host*.

Da notare che i *bus* dati sono predefiniti a 32 *bit* e, nel caso della nostra applicazione, che ha ingressi uscite con parte reale e immaginaria da 16 *bit* ciascuna, i dati vengono gestiti allineati.

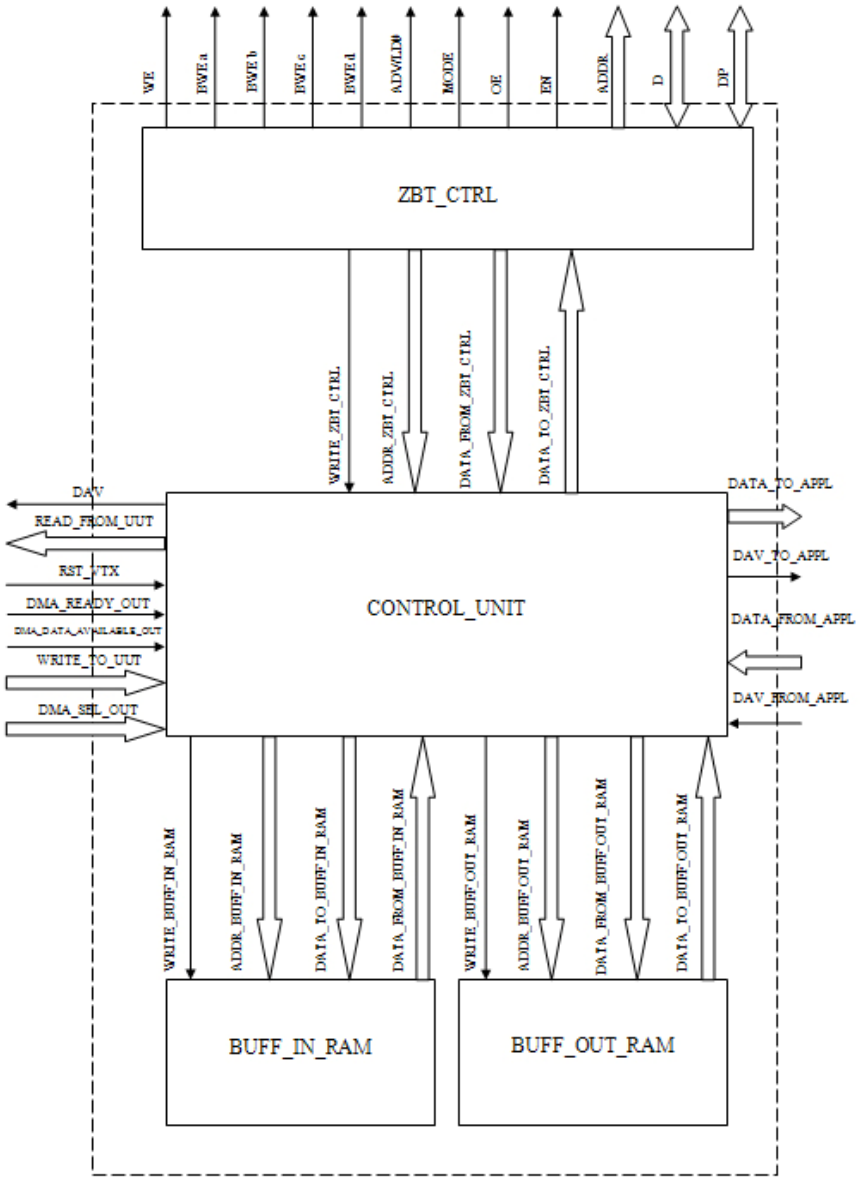


Figura 5.16 : Il blocco Memory Manager nel dettaglio dei suoi componenti.

5.3.7 Il controllore delle ZBT – SRAM

Questo blocco, sviluppato da *Xilinx*, agisce da interfaccia verso le memorie ZBT, rendendole funzionalmente identiche a semplici memorie *Select* RAM analizzate in precedenza. Occorre implementarlo sull'FPGA dedicato all'applicazione ed è fornito come VHDL sintetizzabile.

5.4 I protocolli di trasferimento dati

5.4.1 La comunicazione attraverso la PCI Communications Interface

Nello scambio di dati tra *PCI Communications Interface* e *Memory Map Interface*, possiamo individuare i principali segnali coinvolti:

- EMPTY: segnala la disponibilità alla ricezione dei dati provenienti dalla *Spartan* da parte della *Memory Map Interface*.
- AS/DSl : se alto significa che sul bus è presente un indirizzo, altrimenti siamo nel caso di un dato.
- ADIO: *bus* unico per dati e indirizzi.
- RD/WRl: se alto significa che siamo in una fase di lettura da PCI, altrimenti di scrittura.
- RDl/WRl : abilitazione attiva bassa per quanto riguarda sia la lettura che la scrittura.
- BUSY: segnala l'eventuale non disponibilità da parte del PCI alla ricezione di dati.

In figura 5.17 sono mostrate le forme d'onda riguardanti rispettivamente la scrittura su un registro (a) e la lettura (b).

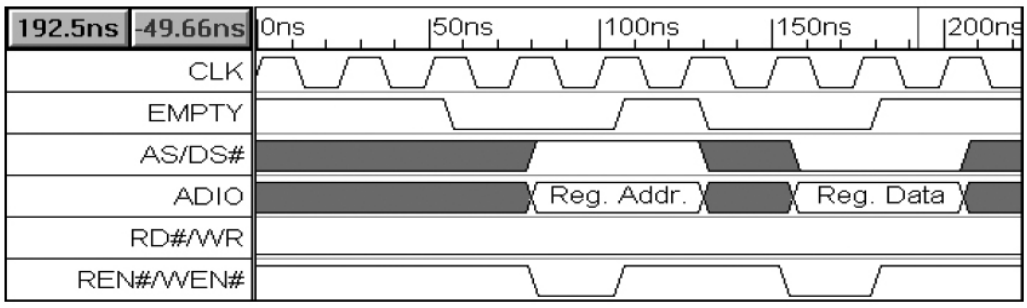


Figura 5.17.a : Temporizzazioni della fase di scrittura su un registro.

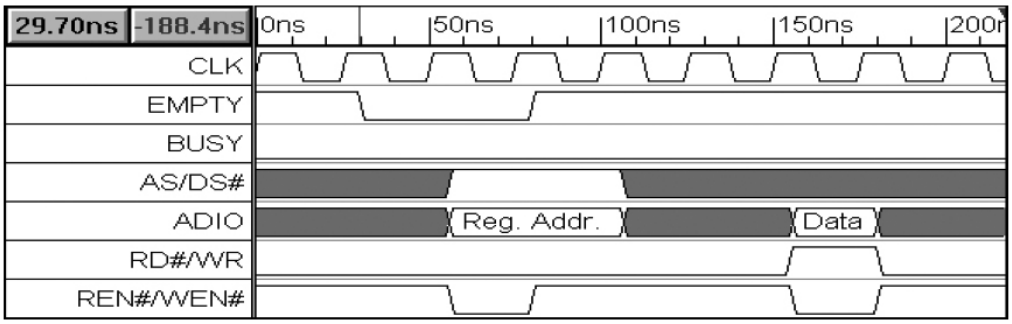


Figura 5.17.b : Temporizzazioni della fase di lettura da un registro.

La *Memory Map Interface* ha accesso allo spazio di memoria attraverso un *bus* indirizzi (*ADDRESS*), due segnali attivi alti che indicano rispettivamente se la fase è di scrittura o di lettura (*WRITE_STROBE* e *READ_STROBE*) e, ovviamente, un *bus* dati bidirezionale (*DATA*).
Per quanto riguarda invece le comunicazioni tra *PCI Communications Interface* e DMA *Interface*, i segnali coinvolti sono gli stessi che abbiamo per la comunicazione con la *Memory Map Interface* con l'aggiunta dei seguenti:

- *DMA_COUNT*: contiene il numero di *words* ancora da trasferire e viene decrementato ogni volta che viene inserito un dato sul *bus ADIO*, sia in lettura che in scrittura.
- *DMA_ENABLE*: deve essere alto ogni volta che il DMA viene coinvolto in un'operazione, poiché segnale di abilitazione di quest'ultimo.
- *DMA_DIRECTION*: determina sia chi dovrà pilotare il *bus* durante la fase in corso che la natura dei dati su esso presenti, se destinati al PCI o all'applicazione.

In figura 5.18 sono mostrate le due temporizzazioni delle trasmissioni via DMA: scrittura del PCI verso le memorie (a) e lettura dei dati provenienti dall'applicazione da parte del PCI (b).

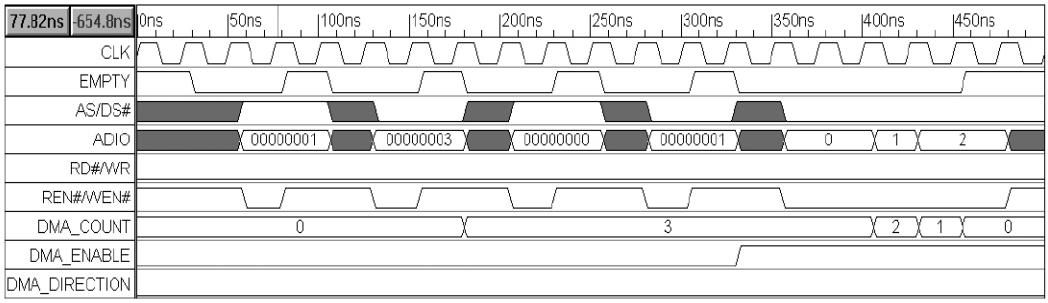


Figura 5.18.a : Temporizzazioni della fase di scrittura in memoria del PCI via DMA.

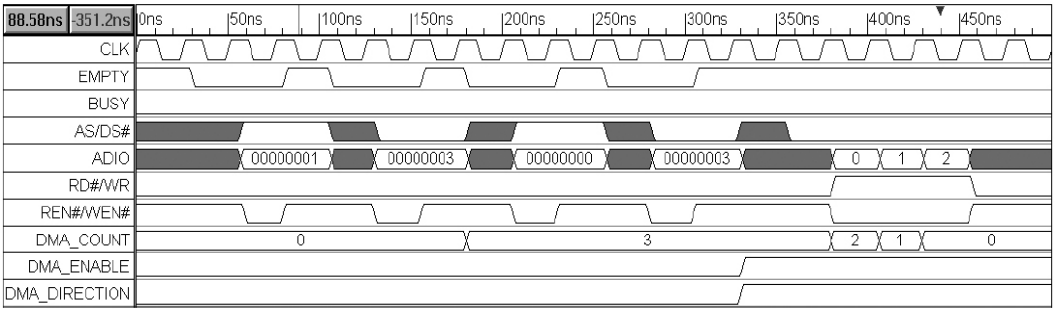


Figura 5.18.b : Temporizzazioni della fase di lettura dalla memoria verso il PCI via DMA.

Come si può notare, all'inizio è necessario programmare i due registri dedicati al DMA, secondo l'operazione da compiere. La trasmissione procede quindi in automatico fino all'esaurimento delle *words* specificate (il *Counter Register* è aggiornato automaticamente).

5.4.2 La comunicazione tra DMA Interface e applicazione utente

Nel caso in cui vi sia una comunicazione tra la DMA *Interface* verso la FSM *Interface*, si ha che dalla programmazione del CSR sono stabiliti: l'abilitazione (1 - abilitato), il canale (quello che corrisponderà allo spazio di memoria dedicato all'immagazzinamento degli ingressi), il *counter* (numero di *words* da scrivere) e la direzione del *bus* dati del DMA (0 – dal PCI verso l'applicazione). Quando un dato è disponibile sul *bus*, il segnale *DMA_DATA_AVAILABLE* viene settato a uno. L'interfaccia risponde

settando a 1 il segnale *DMA_REN*. Il dato passa al controllore delle memorie attraverso il segnale *WRITE_TO_UUT*. Quando il dato fosse indisponibile, il segnale *DMA_DATA_AVAILABLE* viene rimesso a 0 e l'interfaccia risponde togliendo il *DMA_REN*. Il dato su *WRITE_TO_UUT* viene mantenuto al valore che aveva in precedenza. Il segnale *DMA_COUNT* viene decrementato ad ogni trasferimento andato a buon fine e, quando raggiunge il valore zero, la comunicazione termina.

Per quanto riguarda invece il flusso di dati dalla FSM *Interface* verso la DMA *Interfce*, il CSR è programmato come il caso precedente, salvo per quanto riguarda il canale (che sarà quello che corrisponderà allo spazio di memoria dedicato all'immagazzinamento delle uscite), il *counter* (numero di *words* da leggere) e la direzione del *bus* dati del DMA che sarà settata a uno (dall'applicazione verso il PCI). Il DMA segnala la sua disponibilità alla ricezione dei dati, settando a uno il segnale *DMA_RDY*. L'interfaccia risponde attraverso il segnale *DMA_WEN* quando sul *bus* è presente il dato che il PCI ha richiesto di leggere, precedentemente inviato dal gestore delle memorie su *READ_FROM_UUT*. Nel caso di DMA non disponibile ad accettare dati, il segnale *DMA_RDY* si abbassa, provocando di conseguenza anche l'abbassamento del *DMA_WEN*. Se invece il dato sul *bus* non fosse disponibile, il segnale *DMA_WEN* si abbassa, segnalando così al controllore DMA di non leggere il dato.

Veniamo ora alle comunicazioni tra la *FSM Interfac*e verso il *Memory Manager*. Si alza *DMA_DATA_AVAILABLE_OUT* e, al ciclo successivo, su *WRITE_TO_UUT* viene riportato il contenuto del bus *DMA_DATA*. Il segnale *DMA_SEL_OUT* indica la memoria sulla quale occorre andare a scrivere. Se *DMA_DATA_AVAILABLE_OUT* si abbassa, il controllore di memoria si arresta una volta scritto il dato in corso.

Nelle comunicazioni dal *Memory Manager* verso la *FSM Interface* invece, abbiamo il settaggio di *DMA_READY_OUT* seguito dal segnale *DAV*, una volta che il controllore di memoria ha posto il dato richiesto sul piedino *READ_FROM_UUT*. Di conseguenza, l'interfaccia alza il *DMA_WEN*. Se il *DMA_READY* si abbassa, il controllore della memoria si arresta e registra

il dato non ancora inviato, così da renderlo disponibile per la trasmissione successiva.

I segnali della DMA *Interface* inviati al *Memory Manager* (*DMA_DATA_AVAILABLE* e *DMA_READY*) non sono stati ulteriormente registrati poiché, considerato il tempo di accesso alle memorie e la macchina a stati presente nel gestore stesso (contenente a sua volta dei registri), si è potuto risparmiare un inutile ciclo di *clock* di ritardo, ovviando così ad un campionamento ridondante.

Mediante vari *test*, abbiamo potuto appurare che questo tipo di protocollo non comporta la perdita di dati anche in situazioni di interruzioni anomale sia da parte del *bus* PCI che da parte dell'applicazione utente.

CAPITOLO 6
RISULTATI

In questo capitolo andremo a descrivere il flusso di progetto seguito per passare dalla descrizione VHDL dei singoli blocchi alla programmazione della scheda. In seguito saranno analizzati i risultati ottenuti in fase di simulazione funzionale e di sintesi per il processore FFT (comparandolo con altri IP) e quelli dell'altra parte del *core* di convoluzione, per giungere infine al sistema completo, ricavando anche i vincoli di *timing*.

6.1 Flusso di progetto

Andremo adesso a descrivere il flusso di progetto, per quanto riguarda l'utilizzo dei CAD, dalla descrizione dell'architettura in linguaggio VHDL sino alla verifica *post – place & route* ed implementazione su FPGA (v. figura 6.1).
Per quanto riguarda le simulazioni funzionali, post-sintesi e post – *place & route*, abbiamo utilizzato il *software Modelsim SE 64 PLUS 6.1* di *Mentor Graphics*.
Nella fase di sintesi è stato utilizzato *Precision RTL Synthesis 2005a.56* di *Mentor Graphics*. La *netlist* generata da *Precision*, che rappresenta il

design mappato sulle risorse logiche dell'FPGA, è inviata a *ModelSim* per le simulazioni post-sintesi, per le quali sarà utilizzato lo stesso *test-bench* della fase precedente.

Per la fase di *place & route* su FPGA è necessario utilizzare il *software* fornito dal *vendor*, nel nostro caso *Xilinx* ISE 8.1. ISE accetta in ingresso una *netlist* post – sintesi (formato *VHDL*, *Verilog* o *EDIF*) ed un eventuale file per vincolare il *pinout* del *design* (UCF – *User Constraint File*). Quest'ultimo è stato ricavato attraverso un processo di customizzazione dei *file* messi a disposizione da *Nallatech* (*BenNUEY_XC2V8000_FF1152.ucf* e *Primary_USER_FPGA_P70_P100_UCF.ucf* – ciascuno relativo al corrispondente dispositivo), in base alle risorse su scheda che si desidera collegare all'FPGA (sorgenti di *clock*, memorie, *bus* PCI, etc.).

In uscita, ISE genera la *bit – stream* di programmazione dell'FPGA ed i *file* per le simulazioni *post – place & route*. In particolare, il *software* crea una nuova netlist (VHDL o *Verilog*) ed un file SDF (*Standard Delay Format*) per consentire al simulatore di annotare le *snapshot* di simulazione con i ritardi introdotti dalle interconnessioni.

Per l'ultima fase, quella del trasferimento delle informazioni sulla piattaforma, abbiamo utilizzato il *software Nallatech FUSE Probe for Windows* che, a partire dal *file .bit*, va a programmare la scheda stessa.

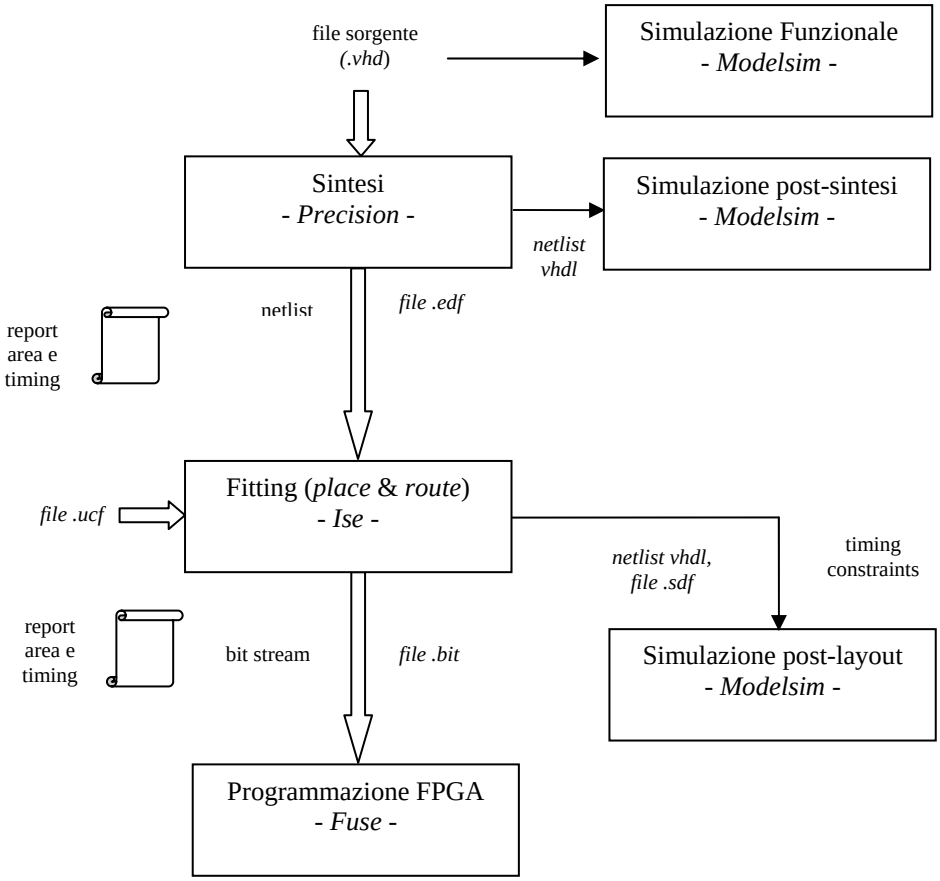


Figura 6.1: Il flusso di progetto.

6.2 Considerazioni sul processore FFT

Per procedere alla sintesi del processore FFT abbiamo apportato alcune modifiche volte all'ottimizzazione per l'implementazione su dispositivi FPGA, che hanno riguardato principalmente le ROM ed i moltiplicatori.

Per quanto riguarda le memorie, sono stati creati dei file VHDL per simularne il comportamento, in modo da poter svolgere delle simulazioni *technology-independent*. Il programma C++ genera automaticamente sia il contenuto delle ROM che i *file* necessari alla loro simulazione, in accordo ai parametri scelti per il processore. Per la sintesi sulla piattaforma di

prototipazione, il programma genera anche degli *script* per il *Core Generator*, così da poter utilizzare le macro delle *embedded* RAM di *Xilinx*.
Dall'analisi del *report* di sintesi abbiamo inoltre notato che i moltiplicatori venivano mappati in *hardware* senza utilizzare le risorse dedicate presenti nell'FPGA. Per ottimizzare sia l'occupazione di area che l'efficienza delle risorse a disposizione, abbiamo allora implementato i moltiplicatori complessi senza vincoli di codifica. È stato comunque mantenuto l'accorgimento per evitare le moltiplicazioni per (*1+ j0*) per il quale è stato reso necessario l'inserimento di una catena di ritardo.
L'analisi dei risultati è stata effettuata mediante confronto con un *file* contenente i vettori di uscita corretti, generati dal programma C++.

6.3 Analisi di altri *core*

6.3.1 Il processore FFT *Spiral*

Come termine di paragone è stato analizzato un *core* FFT *open-source* programmabile, disponibile *on-line*, facente parte del progetto *Spiral* [Nordin et al, 2005]. Questa IP è stata testata su un F*PGA Xilinx Virtex2-Pro*, cioè la medesima piattaforma a nostra disposizione per la realizzazione del progetto. L'analisi è stata compiuta sulla descrizione *Verilog RTL*. Ricordiamo brevemente le principali caratteristiche contraddistinguono questa soluzione:

- Presenza del parametro *p*, che permette di controllare il grado di parallelismo dell'architettura. Mediante questo valore è possibile ottimizzare il *tradeoff* tra minimizzazione dei costi (in termini di area occupata e di potenza impiegata) e massimizzazione delle prestazioni (a livello di latenza e *throughput*).
- Ingressi programmabili *scaled fixed point*.

- Uscite in ordine *bit-reverse*.
- Calcolo basato su algoritmo di Pease radix-*4* e *radix-2* (opzionale) [Kumhom et al, 2000].
- Sviluppo e valutazione su *FPGA Xilinx Virtex2-Pro XC2VP100*.
- Tre tipi si settaggi principali:

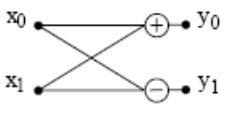
1. Utilizzo minimo degli *slice* (blocco fondamentale dell'FPGA contenente due *look-up tables 4 to 1* basate su *SRAM* e due registri a singolo *bit*).
2. Utilizzo minimo delle *BRAM* (banchi di memoria *embedded* da 2 *KByte*).
3. Salvataggio delle tabelle dei coefficienti di *Twiddle* e delle FIFO all'interno *BRAM*.

Questo *core* si basa sull'algoritmo di *Pease* [Kumhom et al, 2000], che è un caso particolare di FFT . La normale DFT, ad esempio per un ingresso a 2 componenti, sarà:

$$y = DFT_2\, x \quad \text{con}$$

$$DFT_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Questa operazione può essere rappresentata, come visto in precedenza, sottoforma di *butterfly*:



L'algoritmo di *Pease*, si basa sulla seguente fattorizzazione:

$$DFT_{2^k} = R_{2^k}\left(\prod_{i=0}^{k-1} T_i (I_{2^{k-1}} \otimes F_2) L^{2^k}_{2^{k-1}}\right)$$

Dove abbiamo:

R_{2^k} : Matrice per il riordino dei *bit* (*bit reversing*).

T_i : Matrice diagonale contenente i coefficienti di *Twiddle*.

$I_{2^{k-1}}$: Matrice Identica di ordine 2^{k-1}.

F_2 : Matrice *butterfly* di ordine 2.

$L^{2^k}_{2^{k-1}}$: Matrice di *shuffle* per riorganizzare il vettore di ingresso.

Ad esempio, L^8_4 trasforma il vettore [0,1,2,3,4,5,6,7] in [0,4,1,5,2,6,3,7].

Il prodotto $I_n \otimes F_2$ dà luogo ad una matrice diagonale quadrata di ordine $2n$ avente gli elementi di F_2.

Come si può notare, la produttoria implica k moltiplicazioni tra loro identiche, fatta eccezione per il coefficiente T_i che cambia di volta in volta.

Per quanto riguarda l'architettura, come abbiamo già fatto in precedenza, traduciamo l'algoritmo sottoforma di grafico, come indicato in figura 6.2.

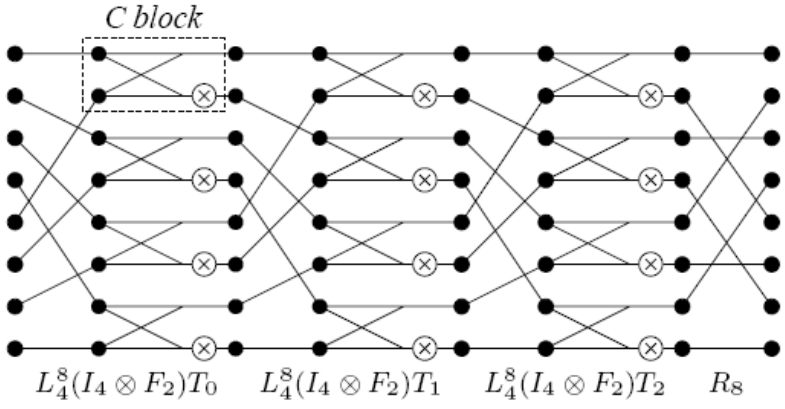


Figura 6.2: Algoritmo di Pease per il calcolo di una DFT_8.

In figura è mostrato il caso di una DFT_8 che risulta:

$$DFT_8 = R_8\left(T_0(I_4 \otimes F_2)L_4^8\right)\left(T_1(I_4 \otimes F_2)L_4^8\right)\left(T_2(I_4 \otimes F_2)L_4^8\right)$$

Il numero degli stadi è pari a 3 e, generalizzando, si ricava la formula generale: numero_stadi = $\log_2(n)$.
Il blocco evidenziato in figura 6.2, indicato con C, rappresenta la *butterfly* completa dell'unico coefficiente di *Twiddle*, visto che uno dei due vale sempre 1. Si ottiene comunque una griglia di $\log_2(n)$ per n/2 blocchi C, dove le colonne sono separate dalle matrici L. Si potrebbe pensare ad una mappatura diretta su *hardware* della struttura ma, fatta esclusione per valori estremamente bassi di n, questa soluzione si rivela proibitiva in termini di risorse impiegate. Conviene dunque optare per il riutilizzo iterativo della logica che si ripete, sviluppando più o meno ciascuna delle due dimensioni (*hardware time multiplexing*). Portando all'estremo queste metodologie di riutilizzo della logica ci possiamo ridurre ad una singola colonna composta da n/2 blocchi (figura 6.3), per poi passare ad un singolo blocco (figura 6.4).

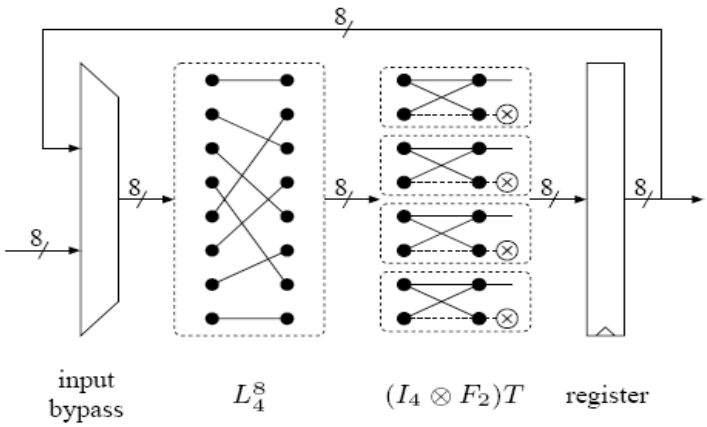


Figura 6.3: Hardware time multiplexing totalmente orizzontale dell'algoritmo di Pease per una DFT_8.

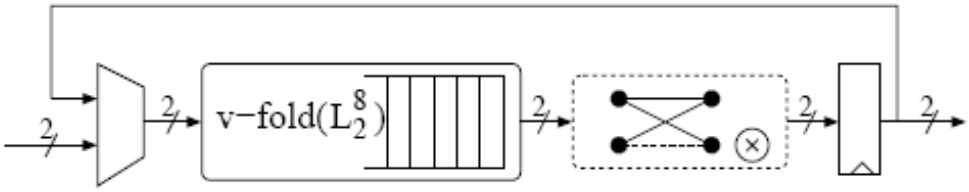


Figura 6.4: Hardware time multiplexing totale sia in orizzontale che in verticale dell'algoritmo di Pease per una DFT_8.

Nel primo caso (*horizontal-folding*), ad ogni iterazione, un segnale di *input* seleziona da una tabella il set di coefficienti di *Twiddle* appropriato. Approssimativamente le risorse necessarie sono ridotte di un fattore $\log_2(n)$ mentre invece la latenza ed il *throughput* rimangono praticamente invariati. Non emerge alcun vantaggio in un parziale *folding* orizzontale, quindi ne è stato adottato uno totale [Nordin et al, 2005].

Nel secondo caso invece entra in gioco il parametro *p* poiché il *folding* verticale va ad incidere sul *tradeoff* costo/prestazioni. Per $p = n/2$ si ha la configurazione originale, mentre per $p = 1$ si ha il *vertical-folding* totale. In generale si ha sia una riduzione delle risorse che un degrado di latenza e *throughput* pari ad un fattore di circa n/2p [Nordin et al, 2005].

Il problema principale è costituito dal *folding* delle matrici L. In assenza di *folding* verticale, avviene solo uno scambio spaziale a livello di connessioni. Per $p < n/2$, i vettori di ingresso e quelli intermedi sono organizzati in segmenti di 2p elementi per ciclo e si hanno allora dei blocchi sequenziali che memorizzano e riordinano i dati nello spazio e nel tempo. La soluzione adottata è presentata in figura 6.5.

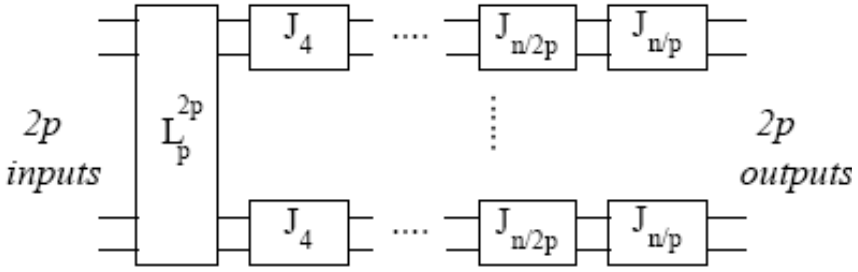


Figura 6.5: Folding delle matrici L.

I blocchi L_p^{2p} rimangono solo delle connessioni scambiate a livello spaziale mentre ciascun blocco J è del tipo mostrato in figura 6.6.

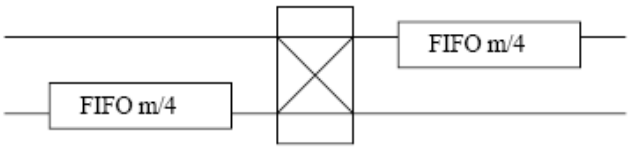


Figura 6.6: Blocchi sequenziali di dimensione generica m, per il folding delle matrici L.

Questa soluzione è adatta per qualunque *p* compreso tra 0 ed n. Ciascun blocco J_m necessita di due FIFO da m/4 ed uno *switch* programmabile che permette ai due dati di attraversare in trasparenza o di essere scambiati, ogni volta alternando per m/4 cicli di *clock*. Tutto questo comporta un ritardo, dovuto alla *pipeline* stessa, pari ad n/2p-1.

La latenza totale è dovuta ovviamente all'intero *processing*: occorrono n/2p cicli per caricare tutti gli ingressi nel registro (o nelle FIFO se p< n/2), ai quali segue l'elaborazione nella *pipeline* di $\log_2(n)$ cicli durante la quale, in corrispondenza dell'ultima iterazione, vengono caricati nuovi *input* [Nordin et al, 2005]. Per quanto riguarda il blocco computazionale C, esso ha il percorso critico costituito da una moltiplicazione e due addizioni *fixed-point* a 16 *bit*. Nel caso di tre livelli di *pipeline*, questo dà luogo ad una latenza di $\left(n/2p+3\right)$ cicli di *clock*. Avremo dunque una latenza totale pari a

$\log_2(n)\left(n/2p+3\right)t$, dove *t* è il periodo di *clock*. Il *throughput* sarà pari a 1/latenza. In questa architettura non è previsto nessun blocco di riordino dei dati (*natural-in*, *bit-reversed-out*).

6.3.2 Il processore FFT *Xilinx* del *Core Generator*

Attraverso il *Core Generator* fornito da *Xilinx*, è possibile generare un processore FFT da implementare direttamente su FPGA [XIL, 2004]. Andiamo ad analizzarne le caratteristiche principali:

- Possibilità di lavorare con un numero di campioni da un minimo di 8 ad un massimo di 65536 (da 2^3 a 2^{16}).
- Impiego dell'algoritmo di *Cooley-Tukey* [Cooley and Tukey, 1965].
- Ingressi suddivisi in blocchi di 8, 12, 16, 20 o 24 bit sia per il modulo che per la fase.
- Aritmetiche *fixed point*, *scaled fixed point*, *block floating point*.
- Possibilità di arrotondamento o troncamento dopo le *butterfly*.
- Utilizzo delle memorie RAM dell'*FPGA*.
- Scelta tra uscita in ordine naturale o in *digit-reversed order*.
- Possibilità di configurazione dei parametri fondamentali durante l'esecuzione.
- Opzioni architetturali:

    1. *Pipelined streaming* I/O per il processamento continuo dei dati (figura 6.7).
    2. Soluzione *radix-4* che permette due distinte fasi: carica e computazione, sovrapponibili per dati successivi. Questo riduce la dimensione ma aumenta la latenza. Numero di stadi pari a $\log_4(N)$.

3. Soluzione *radix-2* che impiega il minimo della risorse ma richiede comunque entrambe le fasi suddette. Numero di stadi pari a $\log_2(N)$.

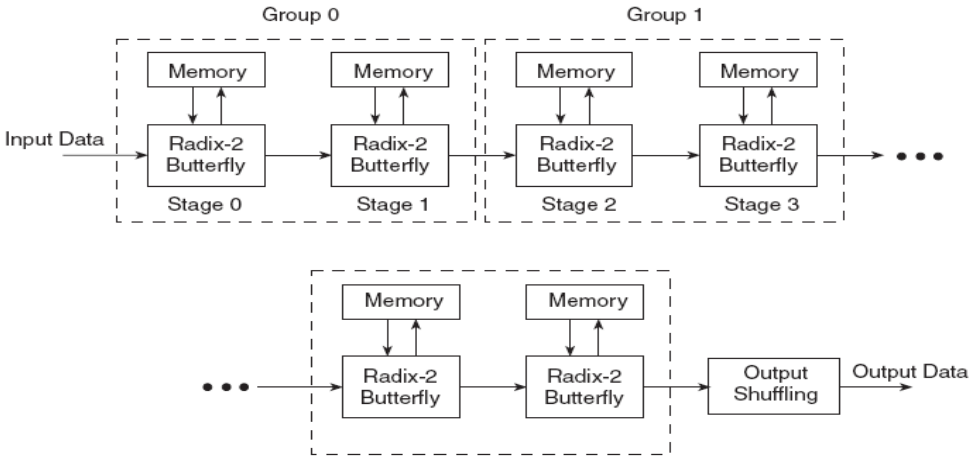


Figura 6.7: Schema a blocchi del pipelined straming I/O.

Per quanto riguarda la scelta *streaming* per gli ingressi, viene impiegata la tecnica di decimazione in frequenza (*DIF*). Nel caso invece della doppia fase viene utilizzata la decimazione nel tempo (*DIT*) [Proakis and Manolakis, 1996].

Nel caso di dimensioni non multiple di 4, viene inserito uno staio *radix-2* aggiuntivo, esattamente come abbiamo visto nei due *core* precedentemente analizzati.

L'ultima considerazione da aggiungere riguarda lo *scaling* che può rivelarsi necessario dopo ogni *butterfly*. Esistono tre modalità attuabili:

1. Assenza di scalatura, portando avanti man mano tutti i *bit* più significativi.
2. Utilizzo di un algoritmo fisso di scalatura ad ogni stadio pari ad un fattore di 1, 2, 4 oppure 8.
3. Introduzione di scalatura automatica attraverso l'aritmetica *block-floating point*.

In figura 6.8 ed in figura 6.9 sono mostrati rispettivamente i moduli di I/O di tipo *radix*-4 e *radix*-2.

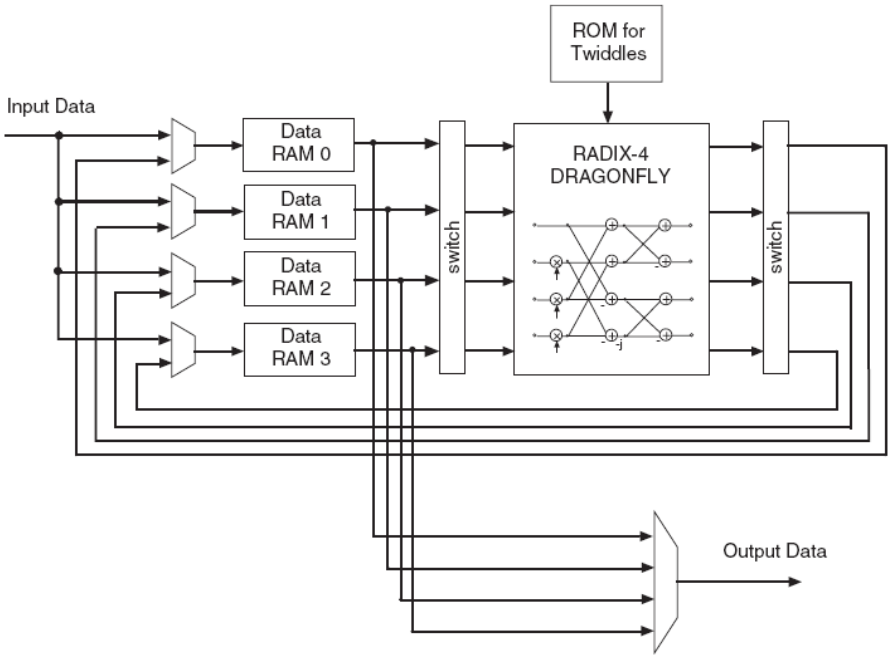


Figura 6.8: Schema a blocchi del modulo di I/O radix-4.

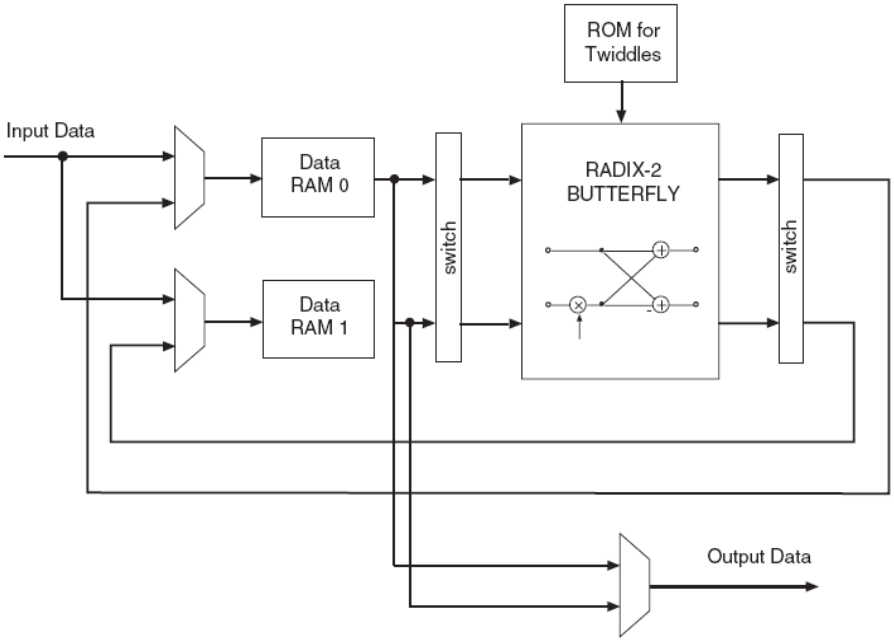


Figura 6.9: Schema a blocchi del modulo di I/O radix-2.

6.3.3 Confronto tra le soluzioni

Alla luce di quanto analizzato, ciascuna soluzione si basa essenzialmente sul medesimo tipo di algoritmo e, dunque, le architetture risultano tra loro molto simili.
Abbiamo a disposizione i dati ottenuti dai creatori stessi dell' IP *Spiral*, che hanno effettuato dei confronti con il core *Xilinx* in modalità r*adix-4 burst I/O* senza riordino. Da queste valutazioni emerge una superiorità del processore generato da *Xilinx*, soprattutto per quanto riguarda valori per noi significativi del numero di campioni. Infatti, per bassi valori di *p* (ad esempio 2 o 4) c'è un notevole accordo tra le due soluzioni. Per *p* = 2, il processore *Spiral* presenta un utilizzo inferiore di *slice* e *BRAM*, pari rispettivamente al 38% ed al 75% di quelle impiegate dal *core Xilinx*. Per valori maggiori di *p*, la situazione però peggiora. Per una DFT_{1024} con *p* = 4 l'IP *Spiral* occupa il 18% di *slice* in più [Nordin et al, 2005].
I dati a nostra disposizione, neil casi più significativi, si riferiscono ad FFT da 1024 e da 16384 punti, con ingressi e uscite a 16 *bit*, testati con un *clock* a 50 Mhz su FPGA *Xilinx* XC2VP100. Abbiamo allora generato il processore FFT per operare la comparazione in base a queste specifiche. Per completezza riportiamo i parametri di configurazione adottati.

- N = 1024 / 16384 - dimensione della FFT.

- SWL = 18 - *System Word Lenght*: dimensione dei dati all'interno del sistema.

- TWL = 12 - *Twiddle Word Lenght*: dimensione dei coefficienti di *Twiddle*.

- RSH = 1 - *Right Shift*: introduzione di un bit più significativo nella catena per evitare un eventuale *overflow*.

- DWL = 16 - *Data Word Lenght*: dimensione dei dati in ingresso.

- OWL = 16 - *Output Word Lenght*: dimensione dei dati in uscita.

Per quanto riguarda la scelta tra le possibili modalità di funzionamento, operata attraverso il programma C++, sono state compiute le seguenti scelte:

- ✓ Tecnica di dimensionamento SWL e TWL originale.
- ✓ Dimensionamento da più ingressi *random*.
- ✓ 100 iterazioni per la determinazione di SWL e TWL.
- ✓ Modulo complesso esatto.
- ✓ Algoritmo FFT.
- ✓ 50 vettori di test.
- ✓ Architettura CBFP programmabile.
- ✓ Mode 0 (FFT a 1024 / 16384 punti).
- ✓ Ingressi interni al quadrato unitario nel piano di Gauss.

Da ricordare che cambiando il parametro *mode*, si può variare il numero di campioni per la FFT. Per ogni incremento del valore di *mode* infatti, si ha un dimezzamento del numero di punti della FFT, a partire da *mode* = 0 che corrisponde al valore stabilito per il parametro N.
In figura 6.10 sono proposte due tabelle riassuntive, relative a quanto emerso nell'analisi dei tre processori proposti, prendendo come parametri di confronto la latenza, l'occupazione degli *slices* e delle risorse di memoria (v. capitolo 5), nel caso di FFT da 1024 e 16384 punti.

	Xilinx	*Spiral*	*Custom*
Latency	2343	1310	1403
CLB Slices *(44096)*	1492 *(3,39%)*	1818 *(4,12%)*	2147 *(4,87%)*
BRAM *(444)*	7 *(1,58%)*	32 *(7,21%)*	36 *(8,11%)*

Figura 6.10.a: Tabella di confronto tra i tre processori per una FFT da 1024 punti.

	Xilinx	*Spiral*	*Custom*
Latency	45176	28714	21899
CLB Slices *(44096)*	1614 *(3,66%)*	4937 *(11,2%)*	3958 *(8,98%)*
BRAM *(444)*	41 *(9,23%)*	180 *(40,54%)*	134 *(30,18%)*

Figura 6.10.b: Tabella di confronto tra i tre processori per una FFT da 16384 punti.

Da questo confronto emerge che il *core Xilinx* è quello che minimizza l'impiego di risorse logiche e di memoria a scapito però della latenza.

Il core *Spiral*, come già anticipato, data la sua estrema flessibilità architetturale, risulta meno efficiente per quanto riguarda l'impiego delle risorse.

Il core da noi sviluppato ha delle prestazioni inferiori per FFT con un numero di punti relativamente contenuto, mentre, per quanto riguarda trasformazioni di notevole entità, garantisce dei risultati migliori in termini di latenza. Da sottolineare inoltre che grazie al parametro *mode*, il nostro *core* può implementare FFT con un numero di punti via via dimezzato, diminuendo la latenza di conseguenza. Questa operazione può essere compiuta dagli altri due processori solo aggiungendo degli zeri agli ingressi, mantenendo quindi la latenza originaria.

6.4 Risultati di sintesi del *core* di convoluzione

In questa sezione sono riportati i risultati ottenuti riguardo alla sintesi del *core* di convoluzione. Abbiamo adottato la seguente configurazione:

- Campioni complessivi della IR: 49152 (1.024 s @ 48 kHz).
- Frequenza di campionamento: 48 kHz.
- Partizioni della IR : 6.
- Dimensione del blocco: 16384 (numero di campioni su ciascuna partizione: 8192).

Abbiamo implementato la catena di moltiplicazione – accumulo e i processori FFT e IFFT da 16384 punti visti nel paragrafo precedente.
Il sistema nel suo complesso ha una latenza pari a 43803 cicli di *clock* (21899 per i processori FFT ed IFFT e 5 del convolutore), cioè, nel caso di una frequenza di 50 MHz, circa 876 µs. Questa latenza è ovviamente relativa solamente al sistema in esame. Per avere il tempo necessario ad un'intera fase di *processing* infatti, dovremmo aggiungere il tempo necessario al trasferimento dei dati via PCI, tenendo presente che, comunque, i campioni si presentano ad una frequenza pari al massimo a 48 kHz.
Riportiamo di seguito il *report* completo del sistema.

```
************************************************
Device Utilization for 2VP100ff1696
************************************************
Resource                Used    Avail   Utilization
------------------------------------------------
IOs                     271     1164     23.28%
Global Buffers          1       16        6.25%
Function Generators     13054   88192    14.80%
CLB Slices              6624    44096    15.02%
Dffs or Latches         13248   91684    14.45%
```

```
Block RAMs              444     444     100.00%
Block Multipliers       72      444      16.22%
Block Multiplier Dffs   0       15984     0.00%
-----------------------------------------------
Number of gates :                  13694

Domain         Clock Name     Min Period (Freq)
------         ----------     -----------------
ClockDomain0   clk_in         14.381 (69.536 MHz)
```

Come si può notare, la maggiore limitazione è costituita dalle memorie. Si potrebbe allora pensare di implementare più canali su di uno stesso FPGA andando ad utilizzare una memoria esterna, la quale però, essendo costituita da un unico banco, dovrebbe funzionare ad una frequenza più alta rispetto a quella dell'applicazione, per gestire gruppi di più dati anziché dati singoli. Si osservi infine che l'implementazione è avvenuta sull'FPGA XC2VP100 presente sul modulo di espansione *BenBLUE-III*, poiché la soluzione su XC8000 non è stata possibile, proprio a causa dell'alto numero di memorie in gioco.
Riportiamo adesso il report della sola catena di moltiplicazione – accumulo per determinarne il peso all'interno dell'architettura completa.

```
*********************************************
Device Utilization for 2VP100ff1696
*********************************************
Resource                Used    Avail   Utilization
-----------------------------------------------
IOs                     267     1164     22.94%
Global Buffers          1       16        6.25%
Function Generators     1735    88192     1.97%
CLB Slices              1515    44096     3.44%
Dffs or Latches         3030    91684     3.30%
Block RAMs              176     444      39.64%
```

```
Block Multipliers       24      444       5.41%
Block Multiplier Dffs   0       15984     0.00%
-----------------------------------------------
Number of gates :                   1259

Domain            Clock Name        Min Period (Freq)
------            ----------        -----------------
ClockDomain0      clk               9.772 (102.333 MHz)
```

Come emerge dai dati di sintesi, la catena di moltiplicazione – accumulo non comporta stringenti vincoli di *timing* ma, al contrario, il collo di bottiglia sotto questo aspetto è costituito dal processore FFT.

CONCLUSIONI E SVILUPPI FUTURI

In questo lavoro di tesi abbiamo affrontato lo studio e la progettazione VLSI di un acceleratore *hardware* per applicazioni audio, da utilizzarsi all'interno di una catena di registrazione e riproduzione. Misurando la risposta impulsiva di un ambiente, è infatti possibile caratterizzarlo completamente dal punto di vista sonoro e ricostruire in un secondo momento le proprietà dello stesso, attraverso l'operazione di convoluzione nel tempo.

La necessità di un acceleratore *hardware* da unire al *Personal Computer* nasce dal notevole carico computazionale associato agli algoritmi di convoluzione.

Il sistema oggetto di questo lavoro deve essere capace di gestire risposte impulsive di durate fino a 2 secondi, da convolvere con un segnale audio di lunghezza arbitraria e campionato ad una frequenza fino a 48 kHz.

Per la realizzazione del progetto è stato scelto un algoritmo *Overlap – Save* uniformemente partizionato poiché il calcolo nel dominio della frequenza presenta un carico computazionale notevolmente ridotto rispetto al dominio del tempo ed offre un'alta regolarità e flessibilità che meglio si adatta ad un'implementazione su dispositivi FPGA.

L'acceleratore *hardware* è composto da un processore per il calcolo della FFT, un processore per il calcolo della IFFT, una catena di moltiplicazione

– accumulazione ed un'interfaccia per la comunicazione tra il PC *host* e l'applicazione, attraverso il *bus* PCI.
Seguendo un flusso di progetto *semi – custom*, abbiamo sviluppato una descrizione di ciascun modulo dell'architettura in linguaggio VHDL, verificato funzionalmente mediante il CAD *ModelSim* di *Mentor Graphics*.
Il codice è stato sviluppato in maniera completamente parametrica per permettere lo studio degli effetti dell'aritmetica finita di macchina direttamente in un contesto reale, anziché appoggiarsi a *framework* di simulazione *software* (sviluppo *hardware-in-the-loop*).
Questo approccio presenta tre vantaggi notevoli:

- la IP sviluppata è flessibile e può adattarsi anche a contesti diversi da quello nel quale è stata concepita;
- I *loop* di progetto per la soluzione del *trade-off* tra precisione dell'algoritmo ed impiego delle risorse *hardware* sono velocizzati, essendo l'emulazione *hardware* intrinsecamente più veloce di una simulazione *software*;
- Il sistema è dimensionato considerando il reale contesto di applicazione anziché una sua versione simulata.

I processori FFT sviluppati sono capaci di calcolare una FFT o una IFFT di lunghezza variabile con precisione dei dati arbitraria. Implementano un'architettura *pipeline Cascade* con aritmetica *convergent – block – floating point* (CBFP), che consente una elevata flessibilità in termini di lunghezza della FFT/IFFT, un'alta regolarità della struttura, un elevato *throughput* e un'occupazione di area (complessità) ridotta. La configurazione dei processori ed il dimensionamento dell'aritmetica interna avvengono in automatico attraverso un programma scritto in linguaggio C++.
La catena di moltiplicazione – accumulo (MAC) è stata anch'essa sviluppata in maniera del tutto parametrica per permettere sia la

programmabilità dei parametri riguardanti i dati da processare che la massima configurabilità delle aritmetiche finite di macchina nell'implementazione in frequenza dell'algoritmo di *Overlap – Save* uniformemente partizionato. Ogni ramo della catena, corrispondente ad una partizione della risposta impulsiva, contiene inoltre un blocco di decimazione programmabile per permettere la configurazione della *bit-width* all'interno del data *path* stesso.

La comunicazione verso il PC *host* è garantita da un'interfaccia con il *bus* PCI realizzata appositamente per l'applicazione che consente la configurazione dello spazio di memoria per adattarsi a partizionamenti diversi dell'*Overlap – Save*.

Il sistema ottenuto è stato sintetizzato su di un FPGA *Xilinx* XC2VP100, attraverso il CAD *Precision* di *Mentor Graphics* e verificato utilizzando nuovamente *ModelSim* SE. La frequenza massima di funzionamento ottenuta è pari a 69 MHz, mentre l'occupazione del dispositivo è pari al 15% delle risorse logiche, al 16% dei moltiplicatori ed al 100% delle risorse di memoria.

Considerando quanto ottenuto in fase di sintesi, si evince che, impiegando un blocco di memoria esterno, con i dovuti vincoli riguardanti la frequenza di funzionamento, derivanti dalla necessità di gestire i dati riuniti in gruppi, sarebbe possibile implementare anche due canali su di un singolo FPGA, nell'ottica di un impiego in ambiti nei quali sia necessario un processamento multi – canale. Da sottolineare inoltre che, data la presenza di due dispositivi FPGA su un singolo modulo di espansione, i quali hanno risorse e connessioni privilegiate, possiamo pensare di implementare quattro canali su un singolo modulo di espansione, senza che questo richieda un'eccessiva frammentazione dell'architettura.

Nella realizzazione di questo progetto, come già accennato, si è tenuto conto di una prototipazione *hardware-in-the-loop*. Abbiamo posticipato infatti la determinazione ottima dei parametri ad una fase successiva alla programmazione della scheda, così da poter rendere possibile il *test* del sistema direttamente sul campo.

La valutazione del sistema può essere effettuata mediante stimoli che coprono l'intera banda audio e risposte impulsive appartenenti ad un insieme eterogeneo di ambienti, misurando indici di merito caratteristici quali ad esempio SNR (Rapporto Segnale/Rumore), THD (Total Harmonic Distortion) ed MSE (Errore quadratico medio).

Attraverso un *software* per rendere i *test* ed il calcolo degli indici del tutto automatici, sarà possibile operare una calibrazione estremamente precisa del sistema, anche per quanto riguarda le aritmetiche finite di macchina e le *bit-width* interne. In questo modo potranno essere garantite delle precise prestazioni sulla base dei valori minimi degli indici suddetti, a seconda delle richieste del contesto di impiego.

Si potrebbe inoltre pensare, attraverso l'impiego dei dispositivi *Power* PC presenti sulla piattaforma di prototipazione, di implementare un *core* realizzato in linguaggio C per giungere alla realizzazione dell'intero sistema come un'unità *stand-alone* integrata ad un modulo di acquisizione dati.

BIBLIOGRAFIA

[Bonsi et al, 2005] D. Bonsi, D. Gonzalez, D. Stanzial, *Quadraphonic impulse responses for acoustic enhancement of audio tracks: measurement and analysis*, in Proc. Forum Acusticum 2005, Budapest, Aug-Sep 2005.

[SOU, 2004] Soundfield Inc., *The Soundfield MKV microphone user guide* 527 – 027, Issue 1.1, 2004.

[MIC, 2004] Microflown Technologies Inc., *Datasheet Ultimate Sound Probe* v. 1.0, Nov 2004.

[LIB, 2006] Libsndfile library, *http://www.mega-nerd.com/libsndfile*, 2006.

[SON, 1997] Sony Acoustic Mirror, plug-in software Sound Forge, *http://www.sonymediasoftware.com/Products*, 1997.

[AUR, 2004] Aurora software, *http://www.ramsete.com/aurora/homepage.html*, beta ver. 4.1, jul 2004.

[VOX, 2004] Voxengo Inc., Voxengo Pristine Space plug-in, *http://www.voxengo.com/product/pspace*, 2004.

[YAM, 2000] Yamaha Corporation, Pro Audio & Digital Music Instrument Division, *SREV1 Digital Sampling Reverb user guide*, October 2000.

[LAK, 2003] Lake DSP Ltd., *Huron Technical Manual*, software ver. 3.2, rev. 6.0 edition, 2003.

[Proakis and Manolakis, 1996] J.G Proakis, D.G.Manolakis, *Digital signal processing: principles, algorithms, and applications*, Prentice-Hall, 1996.

[Oppenheim and Schafer, 1975] A.V. Oppenheim, R.W. Schafer, *Digital signal processing*, Prentice-Hall, 1975.

[Smith, 1997] Steven W. Smith, *The scientist and engineer's guide to digital signal processing*, 1997.

[Shynk, 2003] J. Shynk, *Frequency-domain and multirate adaptive filtering*, IEEE Signal Processing Mag., Jan 1992.

[Gray, 2003] A.A. Gray, *Parallel sub-convolution filter bank architectures*, in International symposium on circuits and systems, volume 4, pages 528–531, May 2003.

[Gardner, 1995] W.G. Gardner, *Efficient convolution without inputoutput delay*, AES J Audio Eng Soc, 43(3):127–136, Mar 1995.

[Bi and Jones, 1989] G. Bi, E. V. Jones, *A pipelined FFT processor for word-sequential data*, IEEE transactions on acoustic, speech and signal processing, vol. 37, n. 12, 1989.

[Fanucci et al, 2002] L. Fanucci, M. Forliti, P. Terreni, *FAST: FFT ASIC automated synthesis*, Integration, the VLSI Journal, dec. 2002, vol. 33, num. 1-2, pag. 23-37.

[Saponara et al, 2005] S. Saponara, L. Serafini, L. Fanucci, P. Terreni, *Automated design of FFT/IFFT processors for advanced telecom applications*, International Symposium on signals, circuits and systems, jul 2005, pag. 103-106, vol. 1.

[NAL, 2005] Nallatech Inc., *BenNUEY reference guide*, NT107-0123 - Issue 11, Mar 2005.

[XIL, 2005a] Xilinx Inc., *Virtex-II platform FPGAs: complete data sheet*, DS031, ver. 3.4, Mar 2005.

[XIL, 2005b] Xilinx Inc., *Virtex-II Pro and Virtex-II Pro X platform FPGAs: complete data sheet*, DS083, ver. 4.2, Mar 2005.

[NAL, 2004a] Nallatech Inc*., BenBLUE-III reference guide*, NT107-0250 - Issue 1, Dec 2004.

[XIL, 2000] Xilinx Inc., *Synthesizable 200 MHz ZBT SRAM interface*, XAPP136 ver. 2.0, Jan 2000.

[MIC, 2000] Micron Technology, Inc. *8Mb: 512K x 18, 256K x 32/36 Pipelined ZBT SRAM*, MT55L512L18P_2.p65, Jul 2000.

[NAL, 2004b] Nallatech Inc., *PCI communication core*, NT 302-0000, Issue 6, Sep 2004.

[Nordin et al, 2005] Grace Nordin, Peter A. Milder, James C. Hoe, and Markus Püschel, *Automatic generation of customized discrete Fourier transform IPs*, 2005.

[Kumhom et al, 2000] P. Kumhom, J. Johnson, P. Nagvajara, *Design, Optimization, and implementation of a universal FFT processor,* in Proc. 13th IEEE ASIC/SOC Conference, 2000.

[XIL, 2004] Xilinx Inc., *Xilinx LogiCore: Fast Fourier transform*, ver. 3.1,

Nov 2004.

[Cooley and Tukey, 1965] J. W. Cooley, J. W. Tukey, *An algorithm for the machine computation of complex Fourier series*, mathematics of computation, vol. 19, pp. 297-301, Apr 1965.