Università degli Studi di Padova

DEPARTMENT OF INFORMATION ENGINEERING
UNIVERSITY OF PADOVA

Dipartimento di Ingegneria dell'Informazione
Corso di Laurea in Ingegneria Elettronica

Core FFT
per
implementazione su FPGA

(FFT core for FPGA implementation)

Laureando:
Francesco Verenini

Relatore:
Prof. Daniele Vogrig

Anno accademico 2009/2010 - Padova, 23 Luglio 2010

Sommario

Nel presente lavoro di Tesi è stato sviluppato il *core* di un dispositivo in grado di svolgere l'algoritmo FFT. Tale *core* è costituito fondamentalmente dalla *buttefly* ovvero l'unità aritmetica dell'algoritmo. Il dispositivo è stato implementato, tramite linguaggio VHDL, per essere utilizzato su FPGA della *Xilinx*.
L'utilizzo di tale tecnologia consente di ottenere un oggetto riconfigurabile nelle sue grandezze principali.

Indice

Capitolo 1

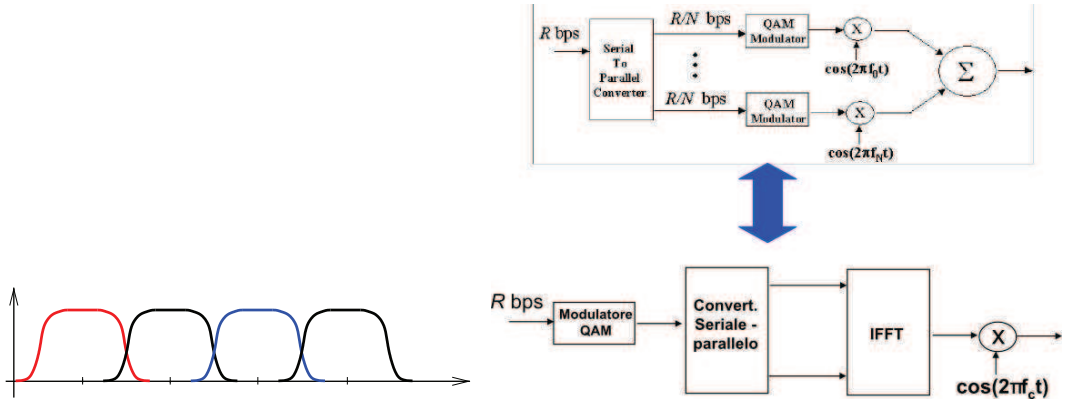
Introduzione

1.1 Informazioni preliminari

1.1.1 L'algoritmo FFT: descrizione e suo utilizzo

Con il termine FFT si indicano una serie di algoritmi ottimizzati per il calcolo della **Trasformata di Fourier** discreta. L'acronimo FFT significa infatti **Fast Fourier Transform** e vuole indicare un procedimento in grado di eseguire velocemente il calcolo della Trasformata di Fourier discreta. Queste tipologie di algoritmi risultano essere molto utili in diverse applicazioni: dall'elaborazione numerica dei segnali digitali, alla soluzione di equazioni differenziali alle derivate parziali, agli algoritmi per moltiplicare numeri interi di grande dimensione.
In particolar modo, nel campo dell'elaborazione numerica dei segnali, viene fatto largo uso della FFT: negli oscilloscopi e nei dispositivi utilizzati per le telecomunicazioni.
Per quanto riguarda questi ultimi, l'algoritmo FFT viene impiegato, in particolar modo, nella modulazione OFDM (*Orthogonal Frequency Digital Multiplexing*). Tale tipo di modulazione permette di trasmettere un segnale in modo parallelo sfruttando una serie di sottoportanti ortogonali ed equispaziate tra loro (Figura 1.1a), in modo tale da coprire l'intera banda disponibile per la trasmissione. Come è possibile vedere dallo schema (Figura 1.1b), l'algoritmo FFT svolge la sua azione andando a filtrare il segnale precedentemente modulato attraverso un qualsiasi sistema di modulazione digitale. In questo tipo di applicazione si utilizza la FFT come algoritmo di convoluzione, questo tipo di approccio permette di evitare di dover utilizzare dei filtri analogici. Sfruttano questo tipo di modulazione: la televisione digitale terrestre (DVB), alcune tipologie di cellulari, le connessioni WiFi WiMax e Hyperlan.

Il dispositivo implementato in questa tesi è stato sviluppato con l'obiettivo di costituire parte integrante di un suddetto sistema di modulazione.



(a) Occupazione in banda di un sistema OFDM

(b) Schema del modulatore OFDM

Figura 1.1: Modulazione OFDM

1.1.2 Tecnologia FPGA

Il dispositivo progettato in questa tesi, è stato implementato su un componente FPGA. La tecnologia FPGA (*Field Gate Programmable Array*) permette di configurare direttamente via Software la funzione logica che deve svolgere un circuito digitale. Un FPGA è un circuito integrato costituito da una serie di strutture tipicamente utilizzate in elettronica digitale (Multiplexer, Flip Flop, porte logiche...) raggruppate tra loro all'interno di blocchi logici (Figura 1.2), le quali possono essere interconnesse tra loro elettronicamente. Per poter verificare il funzionamento di un dispositivo digitale è quindi sufficiente definire le connessioni tra i diversi blocchi da utilizzare e testare l'efficienza del dispositivo. L'utilità di questa tecnologia deriva inoltre dal fatto che durante la fase di progettazione, non è necessario dedicarsi all'interconnessione dei singoli transistors, ma è sufficiente progettare a livello strutturale ovvero riferendosi ai diversi blocchi logici presenti nel dispositivo. É possibile inoltre generare il circuito non mediante uno schema, ma basandosi sulla descrizione comportamentale del circuito stesso cioè definendo come deve comportarsi nelle diverse situazioni in cui si può trovare.
La descrizione comportamentale è effettuata utilizzando linguaggi di descrizione Hardware, in questa tesi il VHDL. Questi linguaggi, molto simile a quelli di programmazione, permettono di definire il comportamento del dispositivo in base ai segnali che esso riceve al suo ingresso e di generare lo

schema delle interconnessioni tra le diverse unità logiche.
I dispositivi FPGA sono quindi general purpose, in grado perciò di essere configurati per diversi tipi di applicazioni, inoltre non è raro che contengano al loro interno alcune strutture complesse come moltiplicatori o blocchi di RAM. In molti casi vengono utilizzati in fase di progettazione di ASIC: sfruttando questo tipo di tecnologia è infatti possibile verificare se un certo tipo di struttura logica risulta funzionare correttamente; una volta effettuate tutte le verifiche allora si può passare alla produzione della stessa su ASIC.

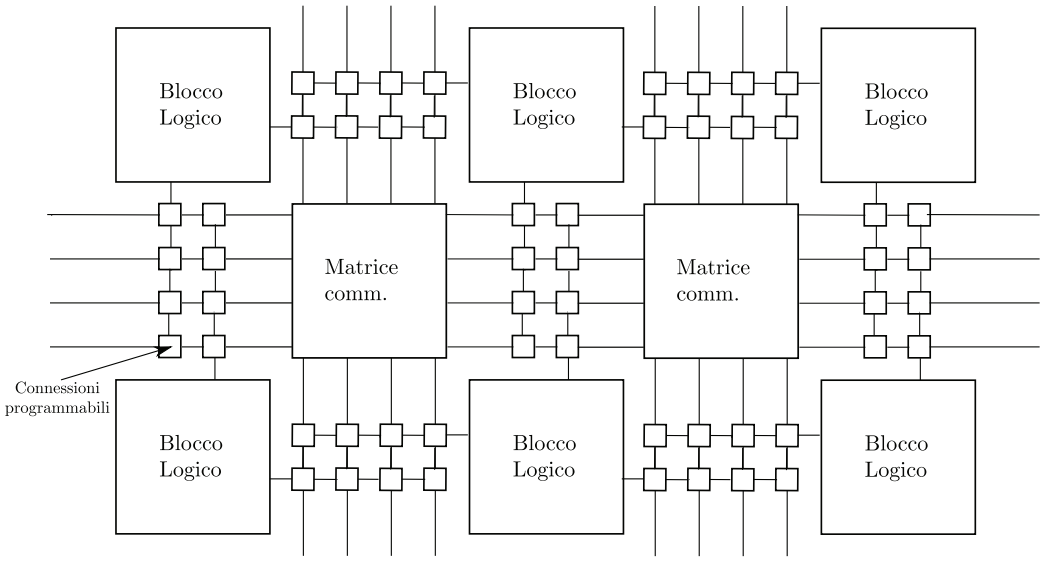


Figura 1.2: Struttura di una FPGA

Esistono svariati tipi di FPGA che differiscono tra loro per il numero e il tipo di risorse di cui dispongono. Il dispositivo progettato in questa tesi è stato sviluppato con l'obiettivo di essere implementato su un FPGA della famiglia *Sparten 3* della *Xilinx*. Come è possibile vedere dalla tabella (Figura 1.3) fanno parte di tale famiglia diversi modelli. I principali parametri che li diversificano sono: numero di blocchi logici (CLB), bit di memoria SRAM (distribuita e a blocchi), numero di moltiplicatori, numero di terminali di I/O disponibili.

La struttura interna di un blocco logico, come è stato detto, è costituita da una serie di componenti (Multiplexer, Latch, Flip Flop) che possono essere interconnessi tra loro; nel caso di una CLB della famiglia *Spartan 3*, si può dire che contiene al suo interno quattro macroblocchi ognuno dei quali prende il nome di *Slice*(Figura 1.5). Le quattro *slices* presenti all'interno di una CLB possono essere facilmente connesse tra loro e con le *slice* delle CLB confinanti,

Device	System Gates	Equivalent Logic Cells	CLB Array (One CLB = Four Slices)				Distributed RAM Bits	Block RAM Bits	Dedicated Multipliers	DCMs	Maximum User I/O	Maximum Differential I/O Pairs
			Rows	Columns	Total CLBs	Total Slices						
XC3S50	50K	1,728	16	12	192	768	12K	72K	4	2	124	56
XC3S200	200K	4,320	24	20	480	1,920	30K	216K	12	4	173	76
XC3S400	400K	8,064	32	28	896	3,584	56K	288K	16	4	264	116
XC3S1000	1000K	17,280	48	40	1,920	7,680	120K	432K	24	4	391	175
XC3S1500	1500K	29,952	64	52	3,328	13,312	208K	576K	32	4	487	221
XC3S2000	2000K	46,080	80	64	5,120	20,480	320K	720K	40	4	565	270
XC3S4000	4000K	62,208	96	72	6,912	27,648	432K	1,728K	96	4	633	300
XC3S5000	5000K	74,880	104	80	8,320	33,280	520K	1,872K	104	4	633	300

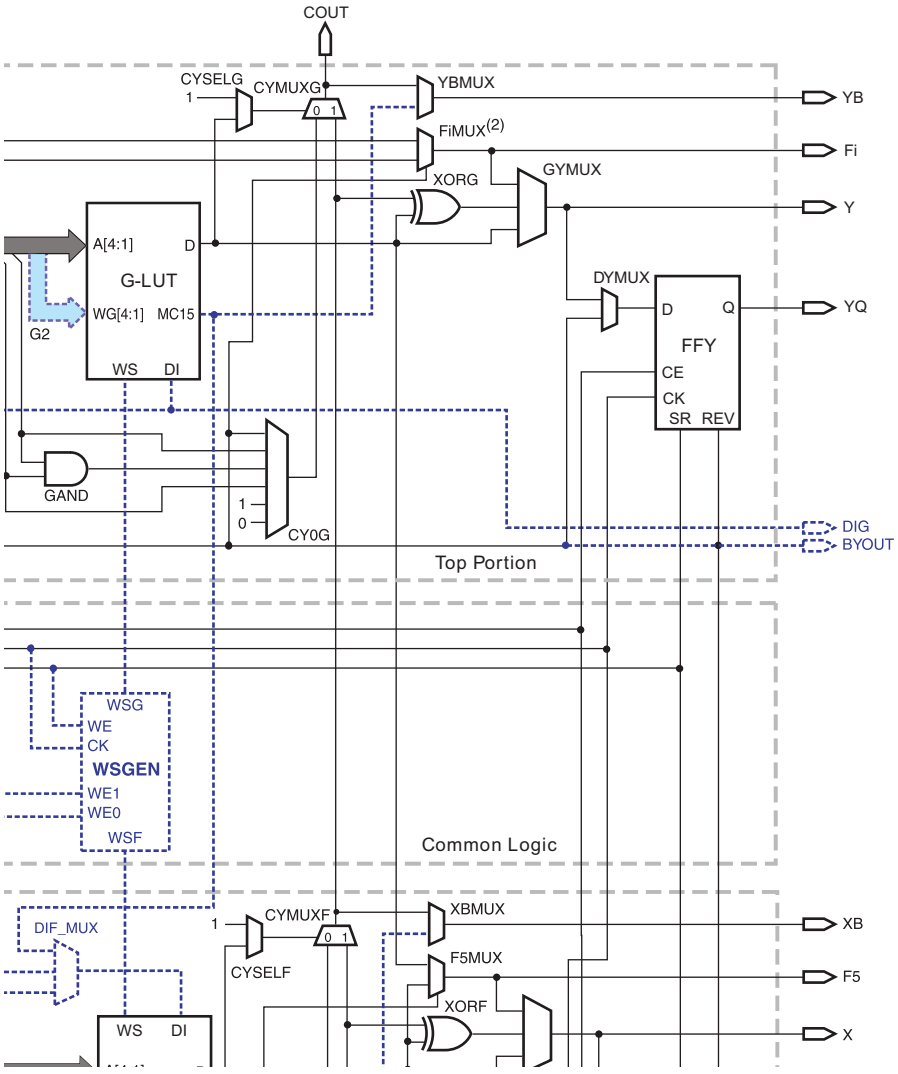
Figura 1.3: Sommario delle risorse degli FPGA della famiglia Spartan 3

inoltre, per ognuna delle due coppie di *slices*, è presente una linea dedicata alla trasmissione del *carry*(riporto) in modo tale da consentire, dopo aver configurato il dispositivo come sommatore, una trasmissione veloce del *carry*. Ogni *slice* presenta al suo interno una struttura complessa (Figura 1.4): due lookup table (LUT) a quattro ingressi (multiplexer i cui ingressi sono collegati a singoli bit di memoria), due elementi di memoria configurabili come Flip Flop o Latch, Multipliexer, la linea per il trasporto del carry e della logica dedicata (XOR) a funzioni aritmetiche. Risulta evidente che, potendo agire sulle diverse interconnessioni tra le *slices* è possibile generare le più svariate tipologie di circuiti digitali.

1.2 Analisi matematica. Dalla DFT alla FFT

1.2.1 DFT per segnali periodici

Per poter descrivere le operazioni svolte dal dispositivo in esame è necessario capire qual è l'operazione matematica che sta alla base del suo funzionamento.
Questo capitolo di introduzione, non vuole essere una descrizione approfondita dell'argomento (a tale proposito si rimanda il lettore alla bibliografia [6]) ma, semplicemente, ha il compito di richiamare alcuni concetti chiave della Teoria dei Segnali utili per la comprensione del lavoro svolto.



Notes:

1. Options to invert signal polarity as well as other options that enable lines for various functions are not shown.
2. The index i can be 6, 7, or 8, depending on the slice. The upper SLICEL has an F8MUX, and the upper SLICEM has an F7MUX. The lower SLICEL and SLICEM both have an F6MUX.

Figura 1.4: Schema semplificato della struttura di una slice

Figura 1.5: Struttura interna di una CLB della famiglia *Spartan3*

Come è noto, per i segnali continui è possibile ottenere la Trasformata di Fourier (TDF) attraverso l'espressione:

$$X(f) = \int_{-\infty}^{\infty} x(t)e^{-j2\pi ft}dt$$

Per i segnali continui e periodici, tale espressione si riduce ad una sommatoria di impulsi traslati su determinate frequenze e scalati opportunamente con i valori della Serie di Fourier:

$$X(f) = \sum_{n=-\infty}^{+\infty} FX_n\delta(f - nF) \qquad F = \frac{1}{T_p} \tag{1.1}$$

Dove il δ si ottiene dal fatto che la TDF di $e^{-j2\pi nFt}$ è appunto $\delta(f - nF)$ e X_n è l'integrale sul periodo del segnale (T_p) di $x(t)e^{-j2\pi nFT}$ ovvero il coefficiente della Serie di Fourier.

L'operazione che il dispositivo svolge (cioè la FFT) è effettuata su un numero di campioni acquisiti in un tempo finito quindi risulta evidente che per comprenderne il funzionamento ci si deve concentrare sulla (1.1).
Il risultato che si ottiene dalla (1.1) è di tipo continuo e quindi non può essere generato da un dispositivo digitale, che può operare solamente su segnali di-

screti. Attraverso opportuni aggiustamenti è possibile ottenere l'espressione in grado di fornire quella che viene definita Trasformata Discreta di Fourier (DFT):

$$X(kF) = \sum_{n=n_0}^{n_0+N-1} T \cdot x(nT) e^{-j2\pi kFnt} \tag{1.2}$$

Questo tipo di Trasformata è di importanza fondamentale poiché è l'unica che può essere calcolata per via numerica e quindi costituisce l'operazione che sta alla base del funzionamento del dispositivo implementato. Il risultato che si ottiene dalla (1.2) è, dato un segnale discreto con periodo $T_p = NT$, una trasformata periodica a frequenza discreta $X(kF)$ con periodo $F_c = 1/T = NF$.
Di tutti i parametri che intercorrono nel determinare la DFT, al fine di ottenere un risultato corretto, è necessario porre particolare attenzione a N, tale valore determina il numero di campioni utilizzati per il calcolo.

1.2.2 DFT per segnali a durata finita: relazione tra Trasformata di Fourier e DFT

Come si evince dal paragrafo precedente, la (1.2) è la trasposizione nel dominio discreto della (1.1) utilizzata per il calcolo della TDF di segnali periodici. Nel caso in cui la medesima espressione venisse utilizzata per calcolare la TDF di un qualunque segnale, non necessariamente periodico o su un numero di campioni che non copre un numero finito di periodi, è evidente che il risultato trovato si potrebbe considerare corretto solamente se si accettasse un certo errore introdotto dalla non periodicità del segnale [4],[1],[6].
Dato che la trasformata inversa della (1.2) come risultato restituisce un segnale periodico, nel caso in cui il segnale acquisito non sia periodico, si preferisce modellarlo attraverso una finestra di campionamento (Figura 1.6) tale da far

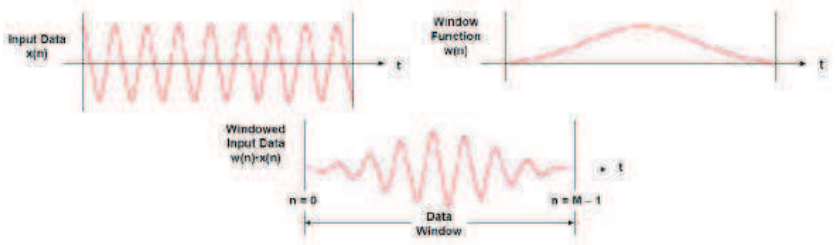


Figura 1.6: Utilizzo della finestra di campionamento

sembrare i campioni acquisiti un periodo del segnale stesso.
Questo tipo di approccio è in grado di ridurre gli errori legati alla non idealità del segnale nei confronti dell'espressione con cui viene calcolata la sua DFT.

Capitolo 2

Struttura

2.1 Fast Fourier Transform (FFT)

Con il termine FFT si fa riferimento ad una serie di algoritmi veloci di calcolo della DFT in grado di ridurre il numero di operazioni associate ad una trasformata di $N = 2^b$ campioni, da un tempo di calcolo proporzionale a N^2 ad uno proporzionale a $N \cdot \log_2 N$.
La caratteristica principale di questi algoritmi è che possono essere ottenuti come un'applicazione del principio *"Divide et Impera"*, ovvero un problema principale viene scomposto in più sotto problemi di minor dimensione (*"Divide"*); una volta risolti questi problemi "semplici", i risultati parziali ottenuti vengono ricombinati (*"Impera"*) in modo da ottenere la soluzione del problema originario.
All'operazione di divisione è chiaramente associato un certo guadagno in termini di complessità di calcolo il quale deve essere confrontato con il costo legato alla ricombinazione e/o alla costruzione di sottoproblemi.
É necessario porre particolare attenzione al fatto che ad ogni scelta della partizione dell'insieme dei dati di partenza in sottoinsiemi, corrisponde un algoritmo diverso, con costi di partizione-ricombinazione diversi.

2.1.1 Complessità di calcolo della DFT

Analizzando l'espressione della DFT

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot W_N^{kn} \qquad dove \qquad \begin{array}{l} W_N^{kn} \triangleq e^{-j\frac{2\pi}{N}} \\ k = 0, 1, \ldots, N-1 \\ x(n) \in \mathbb{C} \end{array} \tag{2.1}$$

è facile notare che per uno qualsiasi dei k valori da calcolare, sono necessarie N moltiplicazioni e $N - 1 \simeq N$ somme, entrambe complesse. Tenendo in considerazione il fatto che un numero complesso è da considerare come una coppia di numeri reali, in termini di operazioni (reali), si ottiene:

$$N_*^C + N_+^C = 4 \cdot N_*^R + 2 \cdot N_+^R + 2 \cdot N_+^R = 4(N_*^R + N_+^R) \tag{2.2}$$

a cui vanno aggiunti anche gli accessi in memoria per la lettura/scrittura dei dati che sono, rispettivamente, N operazioni di lettura e 1 di scrittura perciò

$$N - 1 \simeq N \qquad \text{operazioni di r/w} \tag{2.3}$$

c'è inoltre da aggiungere il tempo di calcolo dei fattori W_N^{kn} o in ogni caso di accesso alla ROM che li contiene.
Concludendo, il numero di operazioni da svolgere risulta composto da:

$$\begin{cases} N & \text{moltiplicazioni} \\ N & \text{somme} \\ N & \text{operazioni di r/w} \\ N & \text{fattori } W_N^{kn} \text{ o accessi ad una ROM} \end{cases}$$

per una complessità totale di:

$$C = K_*^C N^2 + K_+^C N^2 + K_a N^2 + K_w N^2 + (\text{calcolo degli indici}) \tag{2.4}$$

Il calcolo di N coefficienti risulta quindi proporzionale a N^2

$$T(N) = K \cdot N^2 \tag{2.5}$$

2.1.2 Complessità di calcolo della FFT

Come è stato precedentemente scritto, esistono diversi algoritmi per il calcolo della DFT che utilizzano il metodo *"Divide et Impera"*, tali algoritmi differiscono tra loro solamente per il tipo di partizione che si fa dell'insieme originario dei campioni. La complessità temporale è però legata al tipo di partizionamento e differisce a seconda della dimensione minima degli insiemi dei campioni.
Verrà trattato solamente il caso in cui la dimensione minima degli insiemi sia 2, questo valore deriva dal fatto che, utilizzando dispositivi digitali, risulta semplice lavorare con potenze e multipli di 2, inoltre tale scelta risulta essere la più efficiente in termini di complessità temporale, se si opera con un numero di campioni pari $N = 2^b$, come nel caso del dispositivo implementato.
In base al principio *"Divide et Impera"* è necessario scomporre l'insieme costituito dagli N campioni in insiemi più piccoli, quindi, sapendo che N è una potenza 2, è possibile dividere N in due insiemi ognuno dei quali ha dimensione $N/2$.

$$N \to \frac{N}{2} + \frac{N}{2} \tag{2.6}$$

Se il tempo di calcolo associato al problema con cardinalità N è pari a $T(N) \simeq N^2$ allora, dopo il partizionamento, ad ognuno dei due sottoinsiemi con cardinalità $N/2$ è associato un tempo di calcolo pari a $T(N/2) = (N/2)^2$, quindi ricombinando le soluzioni si ottiene il seguente tempo di calcolo:

$$T(N) = 2 \cdot T\left(\frac{N}{2}\right) + R(N) \tag{2.7}$$

dove $R(N)$ è il tempo necessario alla ricombinazione dei risultati ottenuti. Risulta evidente che se $R(N) = O(N)$, e si reitera lo stesso procedimento fino a raggiungere insiemi di dimensione minima, il tempo necessario al calcolo risulta:

$$T(N) = O(N \cdot \log_2(N)) \tag{2.8}$$

2.1.3 Funzionamento della FFT implementata. *Algoritmi a decimazione nel tempo*

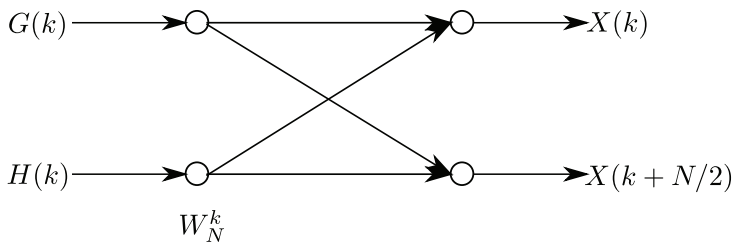
Considerando il solito insieme di N campioni, come precedentemente illustrato, lo si suddivide in due sottoinsiemi con la stessa cardinalità. In questo caso, per effettuare la suddetta suddivisione, si raggruppano fra loro tutti i campioni con indice pari e tutti quelli con indice dispari, in modo tale che risulti:

$$n = \begin{cases} 2 \cdot r \\ 2 \cdot r + 1 \end{cases}$$

Attraverso alcuni passaggi matematici, è facile verificare inoltre che $W_N^2 = W_{N/2}$ quindi dall'espressione della DFT si ricava:

$$\begin{aligned} X(k) &= \sum_{n=0}^{N-1} x(n) W_N^{nk} \qquad\qquad (W_N = e^{-j\frac{2\pi}{N}}) \\ &= \sum_{r=0}^{N/2-1} x(2r) W_N^{2rk} + \sum_{r=0}^{N/2-1} x(2r+1) W_N^{(2r+1)k} \\ &= \underbrace{\sum_{r=0}^{N/2-1} x(2r) W_{\frac{N}{2}}^{2rk}}_{G(k)} + W_N^k \underbrace{\sum_{r=0}^{N/2-1} x(2r+1) W_{\frac{N}{2}}^{2rk}}_{H(k)} \\ &= G(k) + W_N^k H(k) \qquad\qquad k = 0, 1, \ldots, N-1 \end{aligned} \tag{2.9}$$

da cui risulta evidente che il calcolo della DFT su N campioni, è stato scomposto nel calcolo di due DFT entrambe su $N/2$ campioni. Tenendo conto inoltre che la DFT è periodica, $G(k)$ e $H(k)$ sono anch'esse periodiche di periodo $N/2$, quindi $G(k) = G(k + N/2)$ e $H(k) = H(k + N/2)$.
Reiterando l'operazione di suddivisione in sottoinsiemi via via più piccoli, ad un certo punto (dipendente dal numero di campioni da elaborare) l'insieme minimo su cui l'algoritmo opera è di due soli elementi. Il compito dell'algoritmo è infatti quello di calcolare $X(k) = G(k) + W_N^k H(k)$ e $X(k+N/2) = G(k) + W_N^k H(k)$, a partire da alcune coppie di campioni scelte correttamente e di reiterare tale operazione per un numero di volte pari al $\log_2 N$.
Da queste espressioni deriva la struttura base dell'algoritmo che prende il nome di *butterfly* e che costituisce inoltre parte integrante del dispositivo implementato:



Tale elemento è l'unità fondamentale di calcolo dell'intero algoritmo.
Per calcolare, per esempio, la FFT di un segnale di cui sono stati acquisiti 8 campioni, è necessario sviluppare il calcolo precedente per ognuno dei $k = 0, 1, \ldots, 7$ campioni; il risultato ottenuto, come si può vedere dalla (Figura 2.1), è la ripetizione, secondo un determinato schema che deriva dallo sviluppo matematico dell'algoritmo, della struttura butterfly.

Risulta evidente che il calcolo del risultato, può essere effettuato in modo *"parallelo"*, quindi è possibile calcolare simultaneamente più di un risultato e perciò risparmiare tempo. Definendo ognuno dei gruppi verticali di butterfly uno stadio, il numero di stadi da eseguire è pari a $\log_2 N$, in questo caso quindi pari a 3.
Va posta inoltre una particolare attenzione ad altri fattori che concorrono nel determinare il risultato: per ottenere, al termine dell'esecuzione dell'intero algoritmo, dei risultati ordinati correttamente (cioè da $X(0)$ a $X(7)$), è necessario che i campioni posti all'ingresso del blocco di calcolo siano posizionati secondo un ordine diverso da quello naturale (cioè temporale), questo tipo di ordinamento prende il nome di *ordinamento a bit rovesciati*. L'*ordinamento a bit rovesciati* prevede che le posizioni dei campioni all'interno del vettore che li contiene, vengano scambiate prima di svolgere l'algoritmo, le posizioni in cui ogni campione va posto sono determinate invertendo i bit che compongono la parola che identifica l'indice della posizione di ogni singolo campione, in questo caso si ottiene:

$000 \to 000$		$x(0) \to x(0)$
$001 \to 100$		$x(1) \to x(4)$
$010 \to 010$		$x(2) \to x(2)$
$011 \to 110$		$x(3) \to x(6)$
$100 \to 001$	da cui	$x(4) \to x(1)$
$101 \to 101$		$x(5) \to x(5)$
$110 \to 011$		$x(6) \to x(3)$
$111 \to 111$		$x(7) \to x(7)$

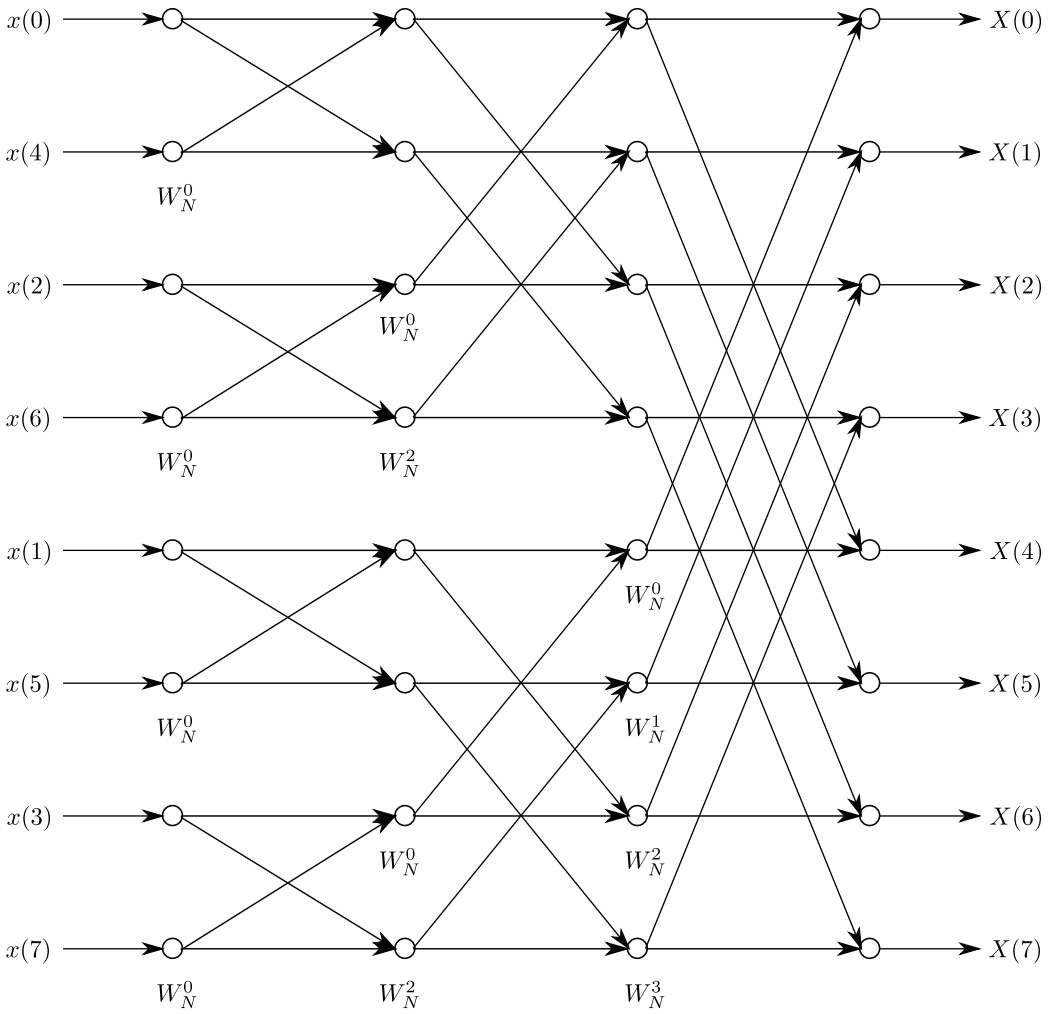


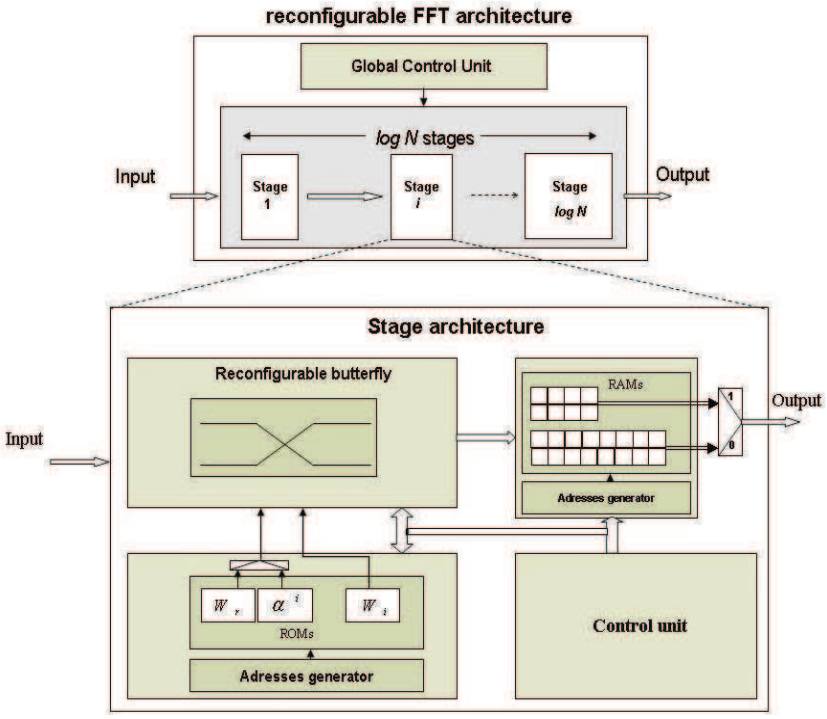
Figura 2.1: Sviluppo dell'FFT

Un'ultima attenzione va posta ai coefficienti W_N^k, in particolar modo all'esponente k che determina il valore del coefficiente stesso, dato che il parametro N è fissato a priori ed è pari al numero di campioni.
Implementando l'algoritmo risulta infatti necessario capire come evolve il termine k durante il calcolo. Sviluppando l'espressione (2.9) per ogni campione si ottiene come risultato che, per tutto il primo stadio, k è sempre pari a 0, durante il secondo stadio invece acquista valori pari a 0 e 2^{b-2}, mentre durante il terzo stadio è $0, 1, 2, 3$ che sono tutti valori distanziati tra loro di 2^{b-3}, in questo caso(per $N = 8$) pari a 1. In generale quindi, presi $N = 2^b$ campioni, per ognuno degli i stadi ($i = 1, \ldots, (\log_2 N) - 1$), la distanza tra i vari k è pari a 2^{b-1-i}.

É bene tenere conto che $k < N/2$, ciò si traduce nel fatto che per la realizzazione del dispositivo sono state previste strutture in grado di contenere solamente tutti i coefficienti W_N^k con $k \leq (N/2)-1$. In ogni caso, anche se tali valori fossero stati calcolati, sarebbe risultato sufficiente limitarsi a calcolare quelli per cui $k \leq (N/2) - 1$.

2.2 Struttura implementativa di base. *FPGA Implementation of a Re-configurable FFT*

Individuati i parametri di base che concorrono allo svolgimento dell'algoritmo, viene descritta ora la struttura di base con cui è stato realizzato il dispositivo. Dovendo infatti utilizzare strutture riconfigurabili quali FPGA, ci si può rendere facilmente conto che esistono svariate configurazioni in grado di svolgere il medesimo compito, sia volendo prediligere la rapidità di esecuzione dell'algoritmo o la minimizzazione dell'area occupata nel dispositivo stesso oppure altri tipi di grandezze.
Nella scelta della struttura è stato quindi preso in considerazione l'articolo di Ghouwayel e Louët [3]. In tale lavoro viene mostrata la possibile struttura costitutiva di un dispositivo in grado di svolgere la FFT sia nel dominio Complesso(quindi infinito) sia su domini finiti di Galois. L'obiettivo di questo tipo di realizzazione è quello di definire uno schema di base che può poi essere sviluppato sia nel caso in cui si prediliga la velocità di esecuzione e quindi penalizzando l'area occupata, sia se si preferisce invece occupare l'area minore e ovviamente impiegare più tempo per il calcolo.



L'immagine riportata, mostra la struttura la quale prevede che il calcolo sia suddiviso in $\log_2 N$ *stages* ognuno dei quali composto da una serie di elementi:

1. la ***butterfly***: parte fondamentale di tutto lo stage, come è stato visto in precedenza, costituisce l'unità aritmetica del dispositivo stesso. Dato che la FFT opera su segnali complessi accetta al suo ingresso tre coppie di segnali reali e dopo averle opportunamente elaborate, restituisce all'uscita due coppie di segnali reali.

2. una **RAM**: ha il compito di contenere i dati che vengono elaborati da ogni stage, la sua capienza è quindi tale da memorizzare tutti i campioni acquisiti.

3. una **ROM**: contiene i coefficienti W_N^k necessari al calcolo.

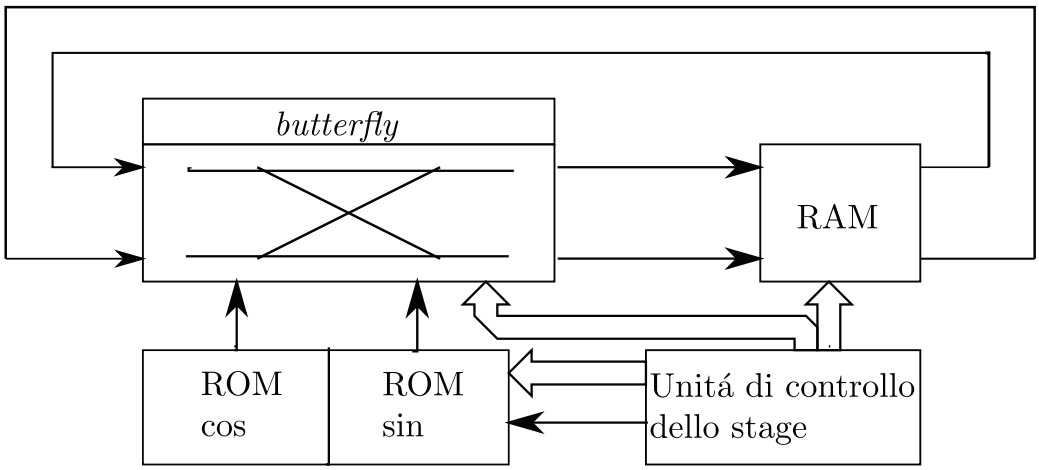
4. l'**unità di controllo**: gestisce lo svolgimento delle operazioni, i dati contenuti nella RAM vengono infatti letti dalle giuste posizioni della

memoria, vanno poi all'ingresso della *butterfly* e infine, dopo essere stati elaborati, vengono nuovamente memorizzati nella RAM. Inoltre durante l'elaborazione dei dati da parte della *butterfly*, il coefficiente W_N^k posto al suo ingresso viene prelevato dalla ROM, ciò si traduce nel fatto che l'unità di controllo calcola il giusto indirizzo di memoria che contiene il coefficiente corretto per l'elaborazione che si sta effettuando.

Tutta l'unità è a sua volta controllata da un'**unità di controllo globale** la quale si occupa di adattare opportunamente i dati all'ingresso del dispositivo, di inserirli nella RAM e di avviare l'esecuzione di ogni stage. Una volta eseguiti tutti gli stage legge sequenzialmente il risultato contenuto nella RAM e lo restituisce come uscita del dispositivo.
L'intero dispositivo è parametrizzato sia per quanto riguarda la quantità di campioni da elaborare sia per la risoluzione dei campioni stessi quindi per il numero di bit che compongono un campione.

2.3 Struttura implementata

Dopo aver individuato la struttura di base è necessario definire alcune delle specifiche seguite nella progettazione del dispositivo.
Innanzitutto l'oggetto che viene presentato in questa tesi costituisce solamente il *core* del dispositivo in grado di eseguire l'intero algoritmo, manca infatti di quella parte che nel paragrafo precedente è stata chiamata unità di controllo globale.
Nell'implementazione si è scelto di prediligere una struttura che fosse in grado di occupare la minor quantità di area:



Come si vede dallo schema, l'intero dispositivo è costituito da un unico stage che viene iterato, per ognuno dei $\log_2 N$ stadi, $N/2$ volte. Tale struttura risulta avere un'occupazione minima, pur mantenendo alte le prestazioni dell'algoritmo stesso. Differentemente dallo schema mostrato nell'articolo, il dispositivo è stato implementato per poter eseguire solamente la FFT classica, quindi nel dominio complesso (infinito). La ROM è stata divisa in due banchi differenti, infatti il termine W_N^k altro non è che un esponenziale complesso e quindi la coppia di due valori di seno e coseno.
L'intero dispositivo è stato sviluppato per poter essere implementato sulla board *Spartan 3* della *Xilinx*.
Dovendo utilizzare un FPGA specifico, le caratteristiche di riconfigurabilità sono state mantenute nella maggior parte del dispositivo, con un'unica eccezione fatta per quelle strutture che, essendo integrate nella board stessa, sono state utilizzate direttamente. Sfruttando questi moduli embedded però sono state bloccate di fatto alcune grandezze:

- Quantità di campioni: è stata posta a $N = 2^{10} = 1024$

- Quantità di bit per campione: avendo a disposizione moltiplicatori embedded a 18 bit si è scelto di operare con campioni costituiti da 18 bit per la parte reale e 18 per quella complessa.

2.3.1 Codifica

Risulta evidente che il dispositivo implementato costituisce un'unità adibita al calcolo numerico, in quanto tale è stato necessario definire una codifica numerica in grado di esprimere al meglio i valori elaborati.
Per poter avere una discreta precisione, ma anche una buona velocità di calcolo, la scelta è ricaduta su una codifica di tipo frazionario in virgola fissa. Dovendo inoltre operare con numeri positivi e negativi, si è preferito utilizzare la rappresentazione numerica in complemento a due poiché tale scelta rendeva semplici le operazioni aritmetiche da eseguire, sia dal punto di vista pratico (stesura del codice), sia per quanto riguarda l'implementazione fisica in quanto la board contiene strutture che sono facilmente configurabili per questo tipo di utilizzo.

Scendendo più nel dettaglio: avendo a disposizione 18 bit e dovendo rappresentare per i coefficienti W_N^k tutti i valori compresi tra -1 e 1, utilizzando la codifica frazionaria sopraccitata, un qualunque numero n con $-1 \leq n \leq 1$ per essere rappresentato correttamente deve essere:

1. per prima cosa moltiplicato per $2^{b-n\_int\_bits} - 1$, dove n_int_bits è un valore che indica quanti dei 18 bit sono utilizzati per rappresentare la parte intera del numero n e tale valore deve essere almeno pari a 1 poiché la rappresentazione è in $C2$ e se tale valore fosse minore di 1 si perderebbe l'informazione relativa al segno che sta sull'**MSB**.

2. successivamente, dopo aver effettuato un arrotondamento all'intero più vicino del risultato ottenuto al punto precedente, solamente nel caso in cui $n < 0$, è necessario traslare il risultato di 2^b per poterlo rappresentare correttamente in $C2$. É poi possibile convertire in binario il valore numerico così ottenuto.

In formule:

$$\text{Campione} = \begin{cases} n \cdot (2^{b-n\_int\_bits} - 1) & \text{se } n \geq 0 \\ n \cdot (2^{b-n\_int\_bits} - 1) + 2^b & \text{se } n < 0 \end{cases}$$

Ad esempio il numero $n_{10} = -0.273618926$, dopo essere stato scalato e traslato (per $n\_int\_bits = 1$) diventerebbe 226280 quindi convertendolo in un numero binario su 18 bit si otterrebbe $n_2 = 110111001111101000$.

Svolgendo l'operazione inversa si possono invece trovare le corrispondenze tra il numero espresso in codice binario e l'effettivo valore rappresentato, per esempio se si hanno a disposizione $b = 3$ bit e $n\_int\_bits = 1$ si ottengono le seguenti corrispondenze:

$$\begin{cases} 011 \to 3/3 \\ 010 \to 2/3 \\ 001 \to 1/3 \\ 000 \to 0 \\ 111 \to -1/3 \\ 110 \to -2/3 \\ 101 \to -3/3 \\ 100 \to -4/3 \end{cases}$$

Da cui risulta evidente che tale codifica è stata scelta in quanto oltre a mantenere "semplici" le operazioni, consente di rappresentare tutti i valori compresi tra -1 e 1.

2.3.2 Gestione dell'overflow

É evidente che all'interno di una struttura composta da diversi blocchi aritmetici posti in serie, è alto il rischio di incorrere in un overflow dei risultati. Innanzitutto si deve precisare che utilizzando una codifica frazionaria, tale problema si verifica solamente in seguito a somme/sottrazioni, infatti i moltiplicatori non possono generare overflow perché in ogni caso sulla loro uscita è sempre presente un valore inferiore rispetto ai valori in ingresso (si stanno moltiplicando infatti valori minori o uguali all'unità).
Per evitare l'overflow sono state utilizzate due soluzioni in grado di minimizzare la probabilità del suo verificarsi e di arginarlo nel caso in cui si verifichi.

É stato detto che il calcolo effettuato dalla *butterfly* viene reiterato sui medesimi campioni per un numero di volte pari a $\log_2 N$. In seguito ad alcune verifiche sperimentali è stato riscontrato che, anche utilizzando all'ingresso del dispositivo segnali con ampiezze piccole, giunti ad un certo stadio del calcolo (che dipende principalmente dall'ampiezza del segnale in ingresso), i valori in uscita dalla *butterfly* (ovvero l'unità di calcolo) risultavano scorretti; era evidente che la causa di questo errore era da attribuire a valori del segnale in ingresso troppo grandi i quali facevano saturare il risultato. La soluzione a questo tipo di problema è stata quella di dividere a metà tutti i valori in ingresso alla *butterfly*, in questo modo si riesce a limitare il verificarsi dell'overflow. Tale affermazione sarà più chiara nel momento in cui verrà analizzata dettagliatamente la struttura dei diversi blocchi che costituiscono il *core*.

La seconda soluzione riguarda la gestione del risultato nel caso in cui si dovesse verificare un overflow. É noto dall'algebra binaria che, utilizzando una codifica in $C2$ ed effettuando una serie di somme/sottrazioni, è necessario verificare solamente alla fine di tutte le operazioni la presenza di un possibile overflow. Ciò che è stato fatto è costituire dei sommatori e sottrattori "speciali" in grado di riconoscere il verificarsi dell'overflow e, nel caso in cui il risultato ottenuto sia scorretto, saturarlo al valore massimo o minimo esprimibile dalla codifica a seconda del tipo di overflow generato. Tali strutture, che chiaramente hanno un'occupazione d'area maggiore rispetto ad un sommatore semplice, sono state utilizzate solamente al termine di una serie di somme/sottrazioni. Non è necessario infatti controllare i risultati intermedi alla serie di blocchi somma/sottrazione in quanto, anche se generano overflow, tale errore può essere "riassorbito" dai blocchi successivi. Per chiarezza viene esposto un esempio nel caso in cui vengano effettuate due somme consecutive su numeri espressi in $C2$ a 3 bit, la prima delle quali genera un

overflow.

Si vuole ottenere il risultato della seguente espressione:

$$n = n_1 + n_2 + n_3$$

Sapendo che

$$\begin{cases} n_1 = 2 \rightarrow 010 \\ n_2 = 3 \rightarrow 011 \\ n_3 = -2 \rightarrow 110 \end{cases}$$

allora risulterà:

$$n_1 + n_2 = 5$$

il sommatore avendo a disposizione solo 3 bit genererà un overflow, restituendo come risultato il numero $101 \rightarrow -3$. A questo punto si somma al risultato ottenuto da $n_1 + n_2$ il valore $n_3 = -2$. Quindi:

$$\begin{cases} n_1 + n_2 = 5 \\ (n_1 + n_2) + n_3 = 3 \end{cases}$$

infatti

$$\begin{array}{cccc|l} 1 & 0 & 1 & + & \rightarrow -3 \\ 1 & 1 & 0 & = & \rightarrow -2 \\ \hline 0 & 1 & 1 & & \rightarrow 3 \end{array}$$

Risulta evidente quindi che l'overflow che si verifica nella prima somma, viene ricompensato da un ulteriore overflow nella somma successiva, ottenendo in questo modo un risultato corretto.

Capitolo 3

Implementazione VHDL e descrizione del dispositivo

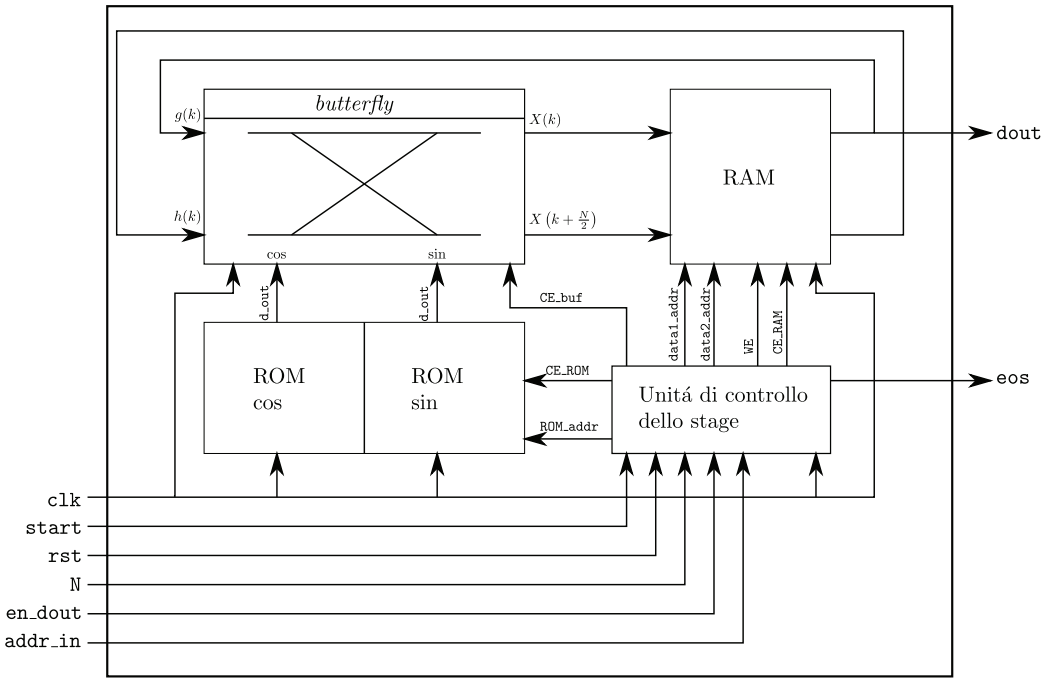


Figura 3.1: Schema a blocchi del dispositivo implementato

L'intero *core* è implementato, in linguaggio VHDL, nel file *stage.vhd*. Analizzando dettagliatamente la sua struttura, il dispositivo presenta 6 ingressi e 2 uscite, rispettivamente:

INGRESSI

- `en_dout`: portando a livello alto questo segnale è possibile abilitare l'uscita `dout` per la lettura dei dati.

- `addr_in`: consente di inviare alla RAM gli indirizzi che si vogliono leggere, una volta abilitato il segnale `en_dout`.

- `start`: è il segnale che ha il compito di avviare ogni singolo stage, portandolo a livello alto per almeno un periodo di clock si avvia l'esecuzione dell'intero stage.

- `rst`: portando a livello alto questo segnale si resetta l'intero stage, quindi si azzerano i registri in esso contenuti.

- `N`: questo segnale proviene dall'unità di controllo ed è pari al numero dello stadio che si sta eseguendo, va da 0 a $(\log_2 N) - 1$.

- `clk`: segnale di clock.

USCITE

- `dout`: in accordo con i segnali `en_dout` ed `addr_in` restituisce, ad ogni ciclo di clock il contenuto della RAM presente all'indirizzo specificato in `addr_in`.

- `eos`: portandosi a livello alto segnala all'unità di controllo globale l'avvenuta esecuzione di un intero stage.

3.1 La *butterfly*

3.1.1 Struttura

Ciò che costituisce l'unità di calcolo fondamentale dell'intero algoritmo, è la *butterfly*. La struttura di questo blocco deriva direttamente dalle formule del capitolo precedente, in particolar modo da:

$$X(k) = G(k) + W_N^k H(k)$$
$$X(k + N/2) = G(k) - W_N^k H(k)$$

I valori contenuti in queste formule sono tutti numeri complessi, la prima operazione svolta è stata quella di scomporre la precedente espressione in parte reale e parte immaginaria. L'operazione è necessaria in quanto utilizzando un dispositivo digitale, non è possibile rappresentare direttamente un numero complesso, l'unico modo per poterlo esprimere è come coppia di due valori reali. Ovviamente i blocchi di elaborazione contenuti nella *butterfly* sono tali per cui questa coppia numerica viene trattata come un unico segnale complesso composto da parte reale e parte immaginaria.
Dalla scomposizione si ottiene:

1. per il coefficiente W_N^k, ricordando che

$$W_N^k = e^{-j\frac{2\pi}{N}k}$$

e che un esponenziale complesso altro non è che una coppia di segnali reali, rispettivamente coseno e seno, deriva:

$$W_N^k = e^{-j\frac{2\pi}{N}k} = \cos\left(-\frac{2\pi}{N}k\right) + j\sin\left(-\frac{2\pi}{N}k\right)$$

inoltre da:

$$\begin{cases} \cos(\alpha) = \cos(-\alpha) \\ \sin(-\alpha) = -\sin(\alpha) \end{cases}$$

risulta

$$W_N^k = e^{-j\frac{2\pi}{N}k} = \cos\left(-\frac{2\pi}{N}k\right) + j\sin\left(-\frac{2\pi}{N}k\right) =$$
$$= \cos\left(\frac{2\pi}{N}k\right) - j\sin\left(\frac{2\pi}{N}k\right)$$

2. per il prodotto $H(k)W_N^k$

$$H(k) = H(k)_{\Re} + jH(k)_{\Im}$$

e

$$W_N^k = \cos\left(\frac{2\pi}{N}k\right) - j\sin\left(\frac{2\pi}{N}k\right)$$

ponendo

$$R(k) = H(k)W_N^k$$

si ottiene

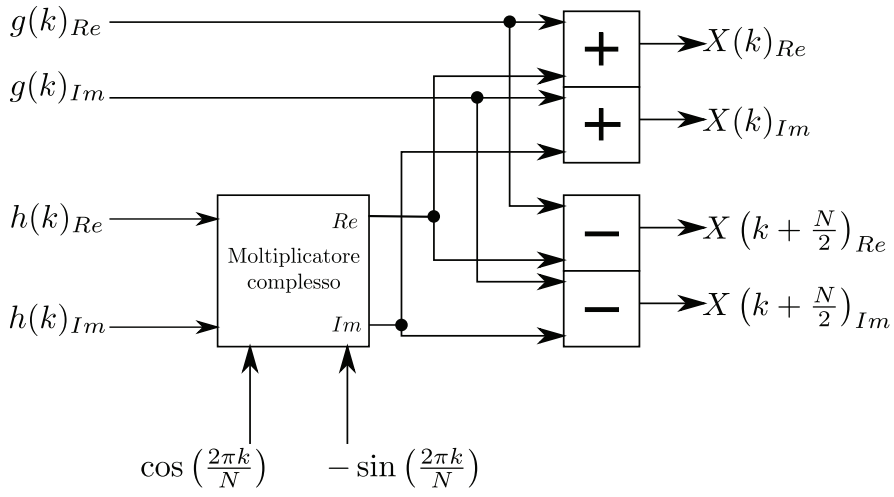
$$R(k)_{\Re} = \left[H(k)_{\Re}\cos\left(\frac{2\pi}{N}k\right) + H(k)_{\Im}\sin\left(\frac{2\pi}{N}k\right)\right]$$
$$R(k)_{\Im} = \left[H(k)_{\Im}\cos\left(\frac{2\pi}{N}k\right) - H(k)_{\Re}\sin\left(\frac{2\pi}{N}k\right)\right]$$

3. il risultato finale risulta essere

$$\begin{cases} X(k)_{\Re} = G(k)_{\Re} + R(k)_{\Re} \\ X(k)_{\Im} = G(k)_{\Im} + R(k)_{\Im} \end{cases}$$

$$\begin{cases} X(k+\frac{N}{2})_{\Re} = G(k)_{\Re} - R(k)_{\Re} \\ X(k+\frac{N}{2})_{\Im} = G(k)_{\Im} - R(k)_{\Im} \end{cases}$$

Da queste espressioni deriva lo schema a blocchi relativo alla *butterfly*.



Come mostrato nello schema, tale dispositivo ha come ingressi 3 coppie di segnali complessi quindi 6 segnali reali e restituisce dopo l'elaborazione 2 coppie di segnali complessi e quindi 4 segnali reali. Inoltre contiene al suo interno un moltiplicatore e un sommatore entrambi complessi. Per quanto riguarda quest'ultimo, è semplicemente costituito da due sommatori e due sottrattori, per quanto riguarda il moltiplicatore invece la sua struttura risulta essere più articolata e quindi viene descritta dettagliatamente nel paragrafo seguente.

Il moltiplicatore complesso

Il compito del moltiplicatore, nel calcolo del risultato della *butterfly*, è quello di moltiplicare il segnale che è stato definito $H(k)$ per il coefficiente W_N^k. Questo blocco, essendo a tutti gli effetti un moltiplicatore complesso, è stato implementato in modo indipendente rispetto all'intera struttura, il che lo rende un apparato del tutto generico in grado di operare anche su altri dispositivi. Proprio per il motivo sopraccitato viene descritto a prescindere dai segnali con cui deve operare (e che quindi deve elaborare), nella successiva descrizione verranno infatti utilizzati ingressi e uscite generiche le quali permettono una più chiara esposizione degli argomenti.
Come per lo schema generale della *butterfly*, anche per il moltiplicatore è stato necessario scomporre il calcolo in parte reale e parte immaginaria. Considerando una semplice moltiplicazione complessa, volendo moltiplicare tra loro i due valori X e Y, a partire da:

$$X = x_{\Re} + jx_{\Im}$$
e
$$Y = y_{\Re} + jy_{\Im}$$

il risultato del loro prodotto è pari a:

$$Z = X \cdot Y$$
ovvero
$$z_{\Re} = x_{\Re}y_{\Re} - x_{\Im}y_{\Im}$$
$$z_{\Im} = x_{\Re}y_{\Im} + x_{\Im}y_{\Re}$$

da cui risulta evidente che per poter svolgere tale operazione sono necessari quattro moltiplicatori, un sommatore e un sottrattore (quindi un ulteriore sommatore). Utilizzando moltiplicatori di tipo embedded e quindi presenti in quantità limitata all'interno della *board*, è stato necessario rielaborare le

espressioni precedenti per cercare di limitare l'utilizzo di tali dispositivi. Un'ottima soluzione è risultata essere quella proposta in una *application note* della *Xilinx*([8]) relativa all'utilizzo dei moltiplicatori. Nel documento si consiglia, per ridurre il numero di moltiplicatori utilizzati e l'area occupata (c'è da tenere conto del fatto che i moltiplicatori embedded hanno un'occupazione non indifferente), di eseguire la moltiplicazione complessa utilizzando solamente tre moltiplicatori e alcuni blocchi somma/sottrazione. In termini matematici, l'iter che porta alla soluzione adottata è il seguente:

$$Z = (x_{\Re} + jx_{\Im}) \cdot (y_{\Re} + jy_{\Im})$$

definendo

$$\begin{cases} p = x_{\Re} y_{\Re} \\ q = x_{\Im} y_{\Im} \\ r = (x_{\Re} + x_{\Im}) \cdot (y_{\Re} + y_{\Im}) = \\ = x_{\Re} y_{\Re} + x_{\Re} y_{\Im} + x_{\Im} y_{\Re} + x_{\Im} y_{\Im} \end{cases}$$

allora

$$\begin{aligned} z_{\Re} &= p - q = x_{\Re} y_{\Re} - x_{\Im} y_{\Im} \\ z_{\Im} &= r - (p + q) = x_{\Re} y_{\Re} + x_{\Re} y_{\Im} + x_{\Im} y_{\Re} + x_{\Im} y_{\Im} - \\ &\quad - x_{\Re} y_{\Re} - x_{\Im} y_{\Im} = x_{\Re} y_{\Im} + x_{\Im} y_{\Re} \end{aligned}$$

che risulta essere la soluzione della moltiplicazione complessa tra X e Y generici. Dalle formule precedenti deriva direttamente lo schema a blocchi del moltiplicatore (Figura 3.2). Al suo interno sono presenti: in un primo stadio due moltiplicatori embedded e due sommatori, in un secondo stadio l'ultimo moltiplicatore utilizzato, un sommatore e un sottrattore e infine, per concludere il calcolo della parte immaginaria, si trova un ultimo sommatore. La struttura implementata è evidentemente più complessa e ha prestazioni inferiori (in termini di velocità di computazione) rispetto a ciò che si sarebbe potuto ottenere utilizzando quattro moltiplicatori, ma risulta essere un buon compromesso tra la rapidità di calcolo e l'occupazione di risorse.

3.1.2 Sviluppo del codice e ulteriori specifiche

VHDL e calcoli

L'intera struttura finora descritta, visibile nello schema (Figura 3.3), è stata implementata nei files:

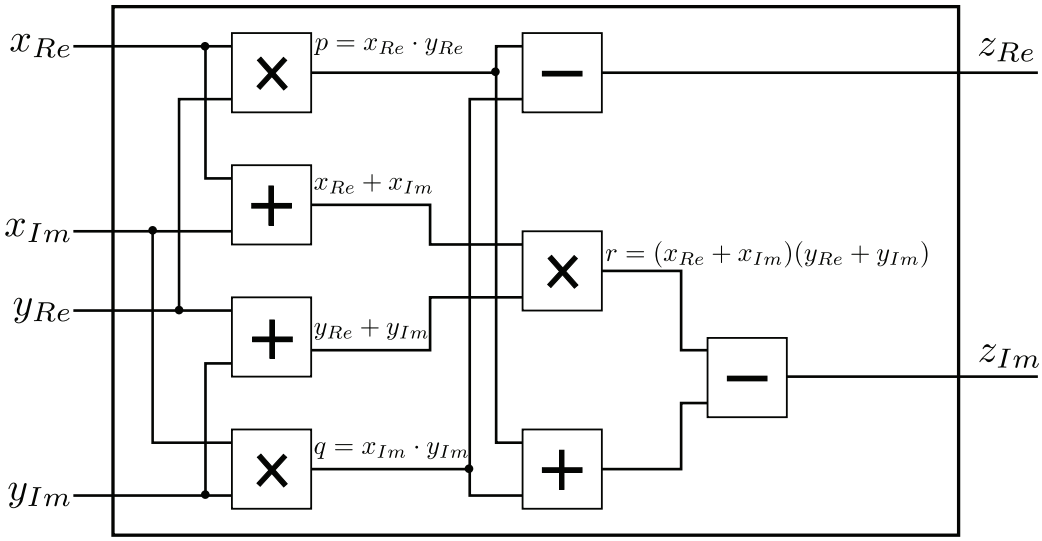


Figura 3.2: Schema del moltiplicatore complesso

- *butterfly.vhd*;
- *complex_mult.vhd*;
- *sum.vhd*;
- *sub.vhd*;

In base alla codifica definita nel paragrafo 2.3.1, è stato possibile estrapolare il funzionamento vero e proprio (cioè bit a bit) delle singole operazioni da svolgere nella *butterfly*.

Le operazioni (aritmetiche) svolte sono:

- addizione
- sottrazione
- moltiplicazione

Per le prime due, non è stato necessario predisporre alcun tipo di modifica o controllo sui bit, infatti utilizzando una codifica in virgola fissa, addizione e sottrazione vengono svolte allo stesso identico modo in cui vengono eseguite nel caso di una codifica intera. A supporto di tale affermazione è sufficiente fare un semplice esempio nel caso in cui (come visto precedentemente) $b = 3$ e $n\_int\_bits = 1$.

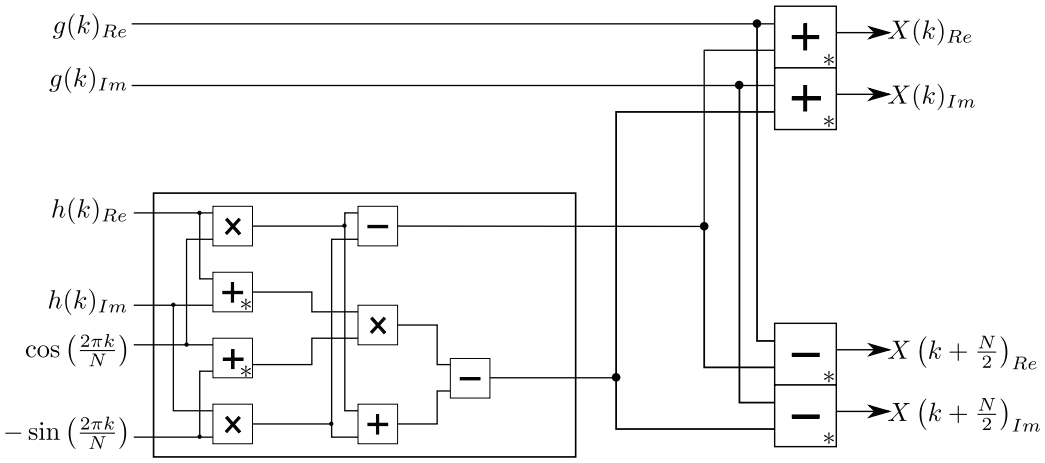


Figura 3.3: Schema completo della buttefly

Volendo per esempio ottenere $1/3 = 2/3 - 1/3$ è sufficiente svolgere una somma binaria tra $010 \to 2/3$ e $111 \to -1/3$, da cui risulta $010 + 111 = 001 \to 1/3$.

Diverso invece è il caso dei moltiplicatori, per i quali è stato necessario compiere uno studio più approfondito. I moltiplicatori presenti nella board operano su segnali a $b = 18$ bit di tipo *signed* cioè su numeri interi rappresentati in $C2$ e restituiscono, come valore in uscita, un segnale a $2 \cdot b = 18 \cdot 2 = 36$ bit sempre rappresentato in $C2$. Il risultato che si ottiene dalla moltiplicazione di due numeri in codifica frazionaria è (a partire dall'**MSB**) pari al prodotto dei due valori diviso per $2^{n\_int\_bits}$. Per ottenere un risultato corretto è quindi necessario prelevare il risultato (ovvero prelevare b dei $2 \cdot b$ bit disponibili) a partire da n_int_bits in meno rispetto all'**MSB**. Facendo un esempio (per $b = 3$ e $n\_int\_bits = 1$):

Si vuole moltiplicare $-2/3 \to 110$ per $1/3 \to 001$

			1	1	0	$\cdot$
			0	0	1	$=$
1	1	1	1	1	0	
5	4	3	2	1	0	posizioni

Da cui è possibile ricavare il risultato corretto ($111 \to -1/3$) prendendo i bit $4, 3, 2$ e cioè i primi b bit a partire dal bit (**MSB**$-n\_int\_bits$). L'operazione appena illustrata viene svolta all'uscita dei moltiplicatori contenuti all'interno del moltiplicatore complesso ed è possibile riconoscerla in

un estratto del codice contenuto nel file *complex_ mult.vhd*, in cui è descritto il moltiplicatore stesso:

Listing 3.1: Metodo di prelievo dei bit all'uscita dei moltiplicatori

```
s <= p(2*data_bits-1-n_int_bits downto data_bits-n_int_bits) +
+ q(2*data_bits-1-n_int_bits downto data_bits-n_int_bits);

Re <= p(2*data_bits-1-n_int_bits downto data_bits-n_int_bits) -
- q(2*data_bits-1-n_int_bits downto data_bits-n_int_bits);

Im <= r(2*data_bits-1-n_int_bits downto data_bits-n_int_bits) -
- s;
```

Dove `data_bits` è l'equivalente di b ed i segnali `p q` ed `r` sono le uscite dei tre moltiplicatori.

Per concludere questa sezione, la *butterfly* è stata resa sincrona nei confronti di un segnale di clock esterno che può essere abilitato o inibito tramite l'ingresso `CE` anch'esso sincrono. Ogni volta che il segnale di clock si porta a livello alto, ed il segnale `CE` è a `'1'`, i dati all'ingresso della *butterfly* vengono elaborati e portati all'uscita restando disponibili fino al successivo fronte di clock.

Controllo dell'overflow

Riprendendo quanto detto nel paragrafo 2.3.2 tutti i valori in ingresso all'intera *butterfly* sono stati divisi per due attraverso uno shift a destra(Codice 3.2).

Per quanto riguarda invece i sommatori/sottrattori a saturazione del risultato in caso di overflow, sono stati utilizzati all'uscita della *butterfly* e prima dei moltiplicatori (in sostanza tutti coloro che nella Figura 3.3 sono contrassegnati da un asterisco).
L'algoritmo utilizzato per determinare il verificarsi di un overflow è molto semplice. Viene effettuata l'operazione aritmetica (somma o sottrazione), successivamente, andando ad analizzare solo il primo bit dei due operandi e del risultato si determina la presenza dell'overflow nel seguente modo (nel caso della somma, analogo è quello della sottrazione):

- se i due operandi sono entrambi positivi e il risultato restituito è negativo allora si è verificato un overflow e viene restituito un risultato pari al massimo valore positivo rappresentabile (che corrisponde alla configurazione $011\ldots1$).

- se i due operandi sono entrambi negativi e il risultato restituito è positivo allora si è verificato un overflow e quindi viene restituito un risultato pari al massimo valore negativo rappresentabile (che corrisponde alla configurazione $100\ldots0$).
- nei restanti casi il risultato risulta sempre corretto ed esente da overflow.

É riportato un estratto del codice relativo al processo che gestisce la saturazione dei valori del sommatore (Codice 3.3).

Listing 3.2: Esempio di dimezzamento del segnale all'ingresso della *butterfly*

```
142 -- dimezzo i dati in ingresso alla butterfly per evitare overflow
gr_s(data_bits-1) <= gr(data_bits-1); -- serve per mantenere il segno
gr_s(data_bits-2 downto 0) <= gr(data_bits-1 downto 1);
```

Listing 3.3: Processo di saturazione del segnale in uscita in seguito ad overflow

```
owfl : process(A,B,D_OUT)
46 begin
  if A(data_bits-1) = '0' then
48   if B(data_bits-1) = '0' then
       if D_OUT(data_bits-1) = '0' then
50       C <= D_OUT;
       else
52       C(data_bits-1) <= '0'; -- saturo al massimo positivo
         C(data_bits-2 downto 0) <= (others => '1');
54     end if;
     else
56     C <= D_OUT;
     end if;
58 else
     if B(data_bits-1) = '0' then
60     C <= D_OUT;
     else
62     if D_OUT(data_bits-1) = '0' then
         C(data_bits-1) <= '1'; -- saturo al massimo negativo
64       C(data_bits-2 downto 0) <= (others => '0');
       else
66       C <= D_OUT;
       end if;
68   end if;
  end if;
70 end process;
```

3.2 Le ROM: mappatura dei coefficienti W_N^k

3.2.1 Implementazione e codice

Le due ROM (rispettivamente per il coseno e il seno) sono state implementate come ROM generiche, all'interno dei file:

- *cos.vhd*
- *sin.vhd*

Prima di descrivere il funzionamento, tra l'altro piuttosto semplice, di questi dispositivi è bene ricordare che la presenza delle due ROM deriva da

$$W_N^k = e^{-j\frac{2\pi}{N}k} = \cos\left(\frac{2\pi}{N}k\right) - j\sin\left(\frac{2\pi}{N}k\right)$$

e poiché $W_{\frac{N}{2}} = W_N^2$, è sufficiente che il numero di coefficienti presenti nelle ROM sia solamente $\frac{N}{2}$, perciò le due ROM contengono $\frac{N}{2}$ valori di coseno e seno, calcolati entrambi per $0 \leq \theta < \pi$ con $\theta = \frac{2\pi k}{N}$ e quindi $k = 0, \ldots, \frac{N}{2} - 1$.

La struttura implementata rispecchia in pieno le formule appena illustrate, ognuna delle due ROM presenta infatti:

- un segnale di `clock` all'ingresso per il sincronismo.
- un segnale `k` all'ingresso che corrisponde all'esponente k presente in W_N^k.
- un segnale `CE` che abilita/disabilita il segnale di `clock`.
- un segnale `d_out` che costituisce il valore del coefficiente letto nella ROM.

Per l'implementazione in linguaggio VHDL si è scelto di descriverle come array di `STD_LOGIC_VECTOR (data_bits-1 downto 0)` con dimensione pari a $\frac{N}{2}$.

Listing 3.4: Array che descrive la ROM

```
44 type ROMType is array(0 to (samples/2)-1)
       of STD_LOGIC_VECTOR(data_bits-1 downto 0);
```

In entrambe le memorie è presente un processo che gestisce l'abilitazione del `clock` e che porta sull'uscita `d_out`, ad ogni fronte di salita del clock, il valore corrispondente a $\cos(\frac{2\pi}{N}\mathtt{k})$ oppure $-\sin(\frac{2\pi}{N}\mathtt{k})$, a seconda che si stia leggendo dalla ROM del coseno o del seno.

Listing 3.5: Gestione del clock e lettura dei dati

```
564 read_rom: process(clk,k,ROM,CE)
    begin
566   if clk = '1' and clk'event then
        if CE = '1' then
```

```
568         d_out <= ROM(conv_integer(k));
        end if;
570   end if;
end process;
```

3.2.2 Mappatura delle ROM

Il codice in grado di fornire la corretta mappatura delle ROM, è stato generato attraverso uno script MATLAB contenuto nei files *cos2.m* e *sin2.m*.
Lo script è costituito principalmente da un ciclo all'interno del quale vengono calcolati i valori di coseno o seno, a seconda del file che si utilizza, una volta calcolato tale valore viene salvata su un file di testo la stringa di codice VHDL relativa al coefficiente calcolato. Il numero di iterazioni compiute è pari a $N/2$ cioè pari al numero di locazioni di memoria di ogni ROM.
Per il calcolo dei coefficienti vengono utilizzate le seguenti formule:

- per il coseno $d\_out(k) = round\left[\cos(\frac{2\pi k}{N}) \cdot (2^{b-n\_int\_bits} - 1)\right]$
- per il seno $d\_out(k) = round\left[-\sin(\frac{2\pi k}{N}) \cdot (2^{b-n\_int\_bits} - 1)\right]$

Dove *round*() indica che il risultato ottenuto è stato arrotondato all'intero più vicino e $(2^{b-n\_int\_bits} - 1)$ serve per rappresentare correttamente il valore nei confronti della codifica utilizzata.

Da un estratto del codice MATLAB ci si può facilmente rendere conto dell'operazione svolta dallo script:

Listing 3.6: Generazione dei coefficienti del coseno

```
for i=1:(N/2)
8       x(i)=round((cos(2*pi*(i-1)/N)/k)*(2^(b-n_int_bits)-1));
end;
10  toSave = fopen('cos.txt','w');
for i=1:(N/2)
12      fprintf(toSave,'conv_std_logic_vector(%d,data_bits), -- %d\n',x(i),i-1);
end;
```

cui corrispondono, come risultato, una serie di righe in VHDL del tipo:

Listing 3.7: Codice VHDL generato da cos2.m

```
conv_std_logic_vector(COEFF,data_bits), -- INDICE_LOCAZIONE
```

3.3 RAM: memoria dei campioni

3.3.1 Implementazione e codice

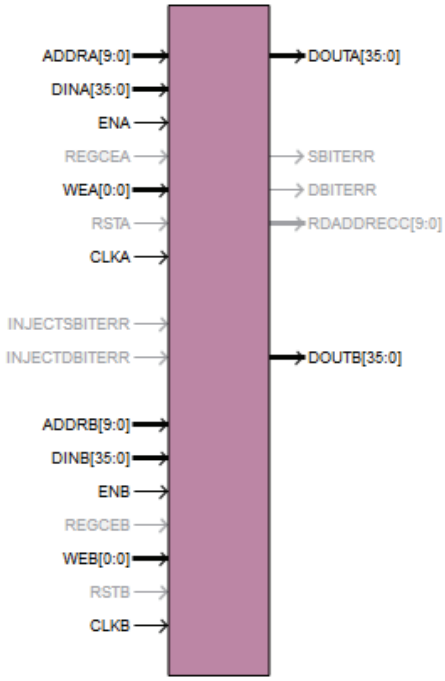
Il ruolo della RAM è quello di memorizzare i campioni del segnale acquisito prima di essere elaborati dall'algoritmo e di contenere i risultati intermedi e finali generati dall'algoritmo stesso. Come risulta dal capitolo secondo infatti, l'algoritmo FFT permette di elaborare i valori direttamente "sul posto" quindi non è necessario utilizzare memorie intermedie per l'elaborazione. Il numero di locazioni di memoria che ha la RAM è perciò pari al numero di campioni elaborati dall'algoritmo stesso e quindi, secondo la notazione finora utilizzata, pari a N.
Per l'implementazione è stata utilizzata, come nel caso dei moltiplicatori, una delle RAM embedded, opportunamente configurata come modulo IP. La memoria utilizzata è di tipo *dual port*, questo perché si è voluto fare in modo che le coppie di valori prelevate dalla RAM fossero portate alla *butterfly* contemporaneamente, tale sistema permette infatti di leggere o scrivere due dati su due locazioni di memoria differenti nel medesimo istante. Come già detto, il numero di locazioni è stato posto pari a N mentre, dato che ogni campione è composto da b bit per la parte reale e b bit per quella immaginaria, ogni locazione di memoria ha un'ampiezza di $2{\cdot}b$ bit. I valori ottenuti vengono memorizzati nel seguente modo:

- i b bit più alti, cioè dal $(2 \cdot b - 1)$-esimo al b-esimo bit, costituiscono la parte reale del campione elaborato.
- i b bit più bassi, cioè dal $(b - 1)$-esimo al bit 0, costituiscono la parte immaginaria del campione elaborato.

Questo tipo di suddivisione permette di scindere le due componenti del segnale semplicemente controllando le interconnessioni tra i diversi blocchi. I segnali di controllo della RAM sono:

- `clka` e `clkb` rispettivamente il segnale di clock della porta A e quello della porta B.
- `addra` e `addrb`, indirizzi delle locazioni a cui fare riferimento.
- `dina` e `dinb`, linee che portano i segnali in ingresso (quindi in scrittura) alla memoria.

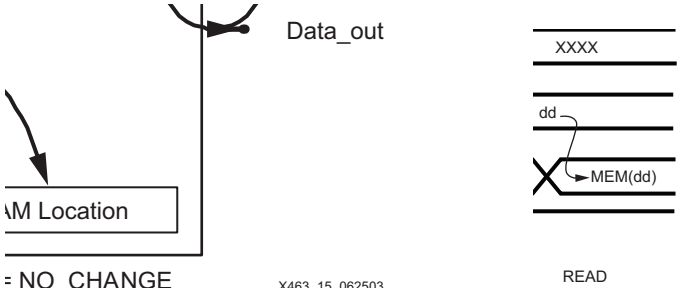
- `douta` e `doutb`, linee che portano il segnale in uscita (quindi in lettura) dalla memoria.
- `wea` e `web`, segnali che abilitano o inibiscono la scrittura dei dati.
- `ena` e `enb`, segnali che abilitano o inibiscono i clock delle due porte.



Il funzionamento del dispositivo è quindi molto semplice, ad ogni fronte di salita del clock, il valore contenuto all'indirizzo `addra` o `addrb`, viene letto e portato sulle rispettive uscite. Se il segnale `wea` o `web` è abilitato allora viene prima letta la locazione di memoria e successivamente viene scritta con il valore contenuto in `dina` o `dinb`. In ogni caso il valore restituito alle uscite `dout` è sempre relativo a dati letti dalla memoria e non riguarda mai il valore scritto nel medesimo ciclo di clock, in accordo con l'opzione *no change* (Figura 3.4) che è stata attribuita alla memoria([8]).

3.3.2 Inizializzazione della memoria

Come è stato descritto nel paragrafo relativo all'implementazione dell'intero dispositivo, avendo realizzato un *core* (quindi una struttura ancora incompleta), manca l'unità di controllo globale che si occupa oltretutto di inizializzare



(a) Schema relativo all'opzione no change (b) Schema dell'evoluzione dei segnali relativo all'opzione No Change

Figura 3.4: Opzione No Change

la RAM. Avendo la necessità di effettuare delle simulazioni per verificare il corretto funzionamento del dispositivo, è stato necessario inizializzare "manualmente" la RAM. Nella configurazione del modulo IP è possibile inserire un file, opportunamente strutturato, con estensione .coe, contenente i valori di inizializzazione della RAM [8].
All'interno di questo file i diversi valori sono espressi in formato esadecimale senza segno quindi per avere dei risultati corretti è necessario convertire i valori numerici (in base 10) tramite il metodo descritto nel paragrafo relativo alla codifica utilizzata. Per rendere più agevole questa operazione è stato usato uno script MATLAB contenuto nel file *ram_ testbench.m*. Tale script permette di convertire correttamente i valori in ingresso dalla base 10 alla base 16 inoltre ordina correttamente i valori posti nella RAM secondo il criterio *bitreverse* e infine genera il file *ram_ testbench.coe* che costituisce il file di inizializzazione della memoria.

Listing 3.8: Conversione e ordinamento bitreverse dei valori

```
for i=1:N
    j = round((sin(2*pi*f*(i-1)/N)/k)*(2^(b-n_int_bits)-1));
    if j >= 0
        x(i)=j;
    else
        x(i)=j+(2^b);
    end;
end;
% riordino i campioni correttamente, per poter effettuare la FFT i campioni
% non devono essere disposti in ordine temporale,
% ma con le posizioni a "bit rovesciati"
x = bitrevorder(x);
```

Listing 3.9: Esempio del file .coe di inizializzazione della memoria

```
;inizializzazione della ram per il testbench
;file generato automaticamente in matlab
```

```
memory_initialization_radix = 16;
memory_initialization_vector =
000000000,
000000000,
7FFFC0000,
800040000,
```

3.4 L'unità di controllo dello Stage

In accordo con quanto detto nel paragrafo riguardante il tipo di struttura implementata, verrà descritto il blocco che prende il nome di **unità di controllo dello Stage**. L'obiettivo di questo dispositivo è quello di sequenzializzare correttamente tutte le operazioni da svolgere, in sostanza tale unità ha il compito di scandire nel tempo lo svolgimento dell'algoritmo.
Riprendendo quanto detto nel Capitolo 2 il blocco di controllo dello stage genera nel tempo i coefficienti utilizzati per calcolare gli indirizzi di memoria dai quali vengono prelevati i valori necessari alla *butterfly* per effettuare il calcolo e nei quali vengono salvati i risultati.
Per comprendere a pieno quali sono le operazioni svolte dall'**unità di controllo dello Stage** risulta utile mostrare uno stralcio di codice C (contenuto nel file *fft.c*) utilizzato per simulare l'algoritmo:

Listing 3.10: Simulazione in C dell'algoritmo

```
/** Operazione di fft, viene passato un vettore di N campioni
/** e al suo interno vengono svolte le operazioni
/** Il parametro r permette di arrestare il calcolo allo stadio r-esimo */
void fft_k(complex sample[N], int r){
  order(sample);
  int count;
  for(count = 1; count <= (N/(pow(2,(log2(N)-r)))); count = count*2){
    int i;
    for(i = 0; i < N; i = i + (count*2)){
      int k;
      for(k = 0; k < count; k++){
        butterfly(&sample[i + k],&sample[i + k + count],(k*(N/(count*2))));
      }
    }
  }
}
```

Risulta evidente che l'intero algoritmo è stato scomposto in tre cicli differenti: il primo, che fa capo a `count`, relativo ad ognuno dei gruppi verticali di butterfly visibili nello schema generale di evoluzione dell'algoritmo(Figura 2.1); il secondo, che fa capo a `i`, relativo al numero di butterfly eseguite in ognuno dei singoli stadi ed infine il terzo, che fa capo a `k`, che serve per calcolare gli indirizzi di memoria contenenti i dati da elaborare.
In particolar modo gli indirizzi utilizzati per il calcolo sono quelli contenuti nei parametri della funzione `butterfly()`:

- `i + k` costituisce l'indirizzo del primo valore da prelevare dalla RAM ed inviare alla *butterfly*;
- `i + k + count` costituisce l'indirizzo del secondo valore da prelevare dalla RAM ed inviare alla *butterfly*;
- `k*(N/count*2)` è l'indirizzo delle ROM contenente le due componenti del coefficiente W_N^k.

L'unità di controllo dello stage genera questi indici ed in base ad essi calcola gli indirizzi di memoria contenenti i dati da elaborare.
Per poter implementare con una certa agilità tale dispositivo si è pensato di scomporlo in due parti: una parte si occupa di generare solamente gli indici di controllo (`k,i,count`) mentre l'altra utilizza gli indici per generare gli indirizzi di memoria e sequenzializza le operazioni abilitando e disabilitando il clock dei diversi blocchi che compongono lo stage (*butterfly*, RAM, ROM).

3.4.1 Blocco di generazione degli indici

Prima di descrivere come è costituita questa unità, è bene capire come evolvono nel tempo gli indici. Per quanto riguarda l'indice `count`, esso risulta essere funzione dello stadio che si sta eseguendo (valore che è direttamente fornito dall'unità di controllo globale, pari a `N` e tale che $0 \leq \texttt{N} \leq log_2 N - 1$). Al variare di `N`, come si può facilmente dedurre da (Codice 3.10), $\texttt{count}=2^{\texttt{N}}$.

Per quanto riguarda l'indice `i`, va precisato che nell'implementazione VHDL è stato sostituito con `rc`, da (Codice 3.10) si deduce chiaramente che evolve secondo la formula $rc = rc + count \cdot 2$ e continua ad essere incrementato fino a quando il suo valore non risulta maggiore o uguale ad N, cioè del numero di campioni.
Infine l'indice `k` è un semplice contatore che viene incrementato finché risulta essere minore di `count`.

L'intero blocco, contenuto all'interno del file *MSF_counter.vhd*, è strutturato nel seguente modo: due processi gestiscono la generazione degli indici `k` ed `rc` in risposta ad un segnale di incremento, mentre un altro processo costituisce una piccola macchina a stati che genera i segnali di incremento e controlla che gli indici non oltrepassino i limiti.
Lo schema degli ingressi e delle uscite è perciò il seguente:

- `N` segnale che proviene dall'**unità di controllo globale** e costituisce l'indice del numero di stadio che si sta eseguendo ($0 \leq \texttt{N} \leq log_2 N - 1$);

- `clk` segnale di clock;
- `inc` segnale di incremento degli indici proveniente dal **blocco di generazione dei segnali di controllo**; se tale segnale viene portato a livello alto, gli indici `k` ed `rc` vengono incrementati in accordo con l'evoluzione generale dell'algoritmo;
- `start` segnale proveniente dall'unità di controllo globale che, se portato a livello alto, avvia l'esecuzione di uno stadio;
- `rc_out` segnale in uscita che coincide con il valore dell'indice `rc`;
- `k_out` segnale in uscita che coincide con il valore dell'indice `k`;
- `count_out` segnale in uscita che coincide con il valore dell'indice `count`;
- `eos` segnale in uscita che indica il termine di ognuno dei $log_2 N$ stadi.

Per quanto riguarda `count`, come descritto prima, la sua generazione risulta molto semplice e consiste semplicemente in uno shift a sinistra dei bit della parola corrispondente al valore 1, di tanti bit quanto è il numero (intero senza segno) espresso da `N` (ad esempio se `N` $= 3$ allora `count` $= 2^3$, in codice VHDL `conv_std_logic_vector(2**N,addr_bits)`).

I due processi che generano `k` ed `rc` sono strutturati come un contatore: ad ogni ciclo di clock leggono lo stato di un segnale interno (ad essi riservato, diverso per ognuno dei due processi), se tale segnale è a livello logico alto allora viene incrementato il valore del corrispondente indice. I due processi leggono inoltre un segnale di reset in grado di azzerare il corrispondente indice in qualunque condizione.
Per rendere efficiente il calcolo di `rc` è stato utilizzato il seguente metodo: ad ogni incremento `rc` risulta `rc + shl(count,1)`; in sostanza quella che era una moltiplicazione per due è stata effettuata attraverso un semplice shift logico.

Listing 3.11: Processo che genera l'indice rc

```
-- il processo controlla l'incremento ed il reset di rc
rc_ctrl : process(clk,MSF_out)
begin
  if clk = '1' and clk'event then
    if MSF_out(2) = '1' then -- reset sincrono
      rc <= (others => '0');
    else
      if MSF_out(3) = '1' then
        rc <= rc + shl(count,"1");
      else
        rc <= rc + conv_std_logic_vector(0,addr_bits+1);
```

```
100         end if;
        end if;
102   end if;
end process;
```

Macchina a Stati

La macchina a stati (Figura 3.5), che ha il compito effettivo di portare all'uscita dell'unità di generazione degli indici i valori `k`,`rc` e `count`, è comandata a sua volta da `start` ed `inc` e comprende tre stati:

1. **first**: costituisce lo stato di partenza della macchina. Durante lo stato **first** gli indici sono in condizioni di reset, quindi risultano azzerati, inoltre il segnale di fine stadio `eos` è a livello logico alto. Se il segnale `start` è portato a livello logico alto, la macchina passa allo stato **increment**, altrimenti resta nello stato **first**.

2. **increment**: in questo stato vengono incrementati i valori di `k` ed `rc` in accordo con lo svolgimento dell'algoritmo. Per consentire l'incremento, vengono portati a livello logico alto i segnali interni che controllano i due processi relativi a `k` ed `rc`. Lo stato **increment** ha inoltre il compito di verificare che i due indici `k` ed `rc` non eccedano i limiti (`k` $<$ `count` ed `rc` $< N$), a tale scopo vengono abilitati o inibiti i segnali di reset relativi ai processsi che controllano `k` ed `rc`. Dallo stato **increment** è possibile passare allo stato **first** se è stata raggiunta la condizione limite per entrambi gli indici, altrimenti la macchina passa allo stato **idle**.

3. **idle**: questo stato ha il compito di attendere che il segnale di incremento degli indici torni a livello alto, nell'attesa mantiene costanti i valori alle uscite `k_out` ed `rc_out`. Se il segnale `inc` è portato a livello alto allora passa allo stato **increment**, altrimenti resta su **idle**.

Listing 3.12: Particolare della gestione dello stato **increment**

```
122 when increment =>
        eos_int <= '0';
124     rc_rst <= '0';
        case MSF_in is
126       when "01" => -- situazione in cui inc = '1' e start = '0'
            if conv_integer(rc) = (2**addr_bits)-(2*conv_integer(count)) then
128           k_rst <= '0';
              if conv_integer(k) = conv_integer(count)-1 then
130             k_inc <= '0';
                rc_inc <= '0';
132             stato_fut <= first;
              else
```

```
134                 k_inc <= '1';
                    rc_inc <= '0';
136                 stato_fut <= idle;
                  end if;
```

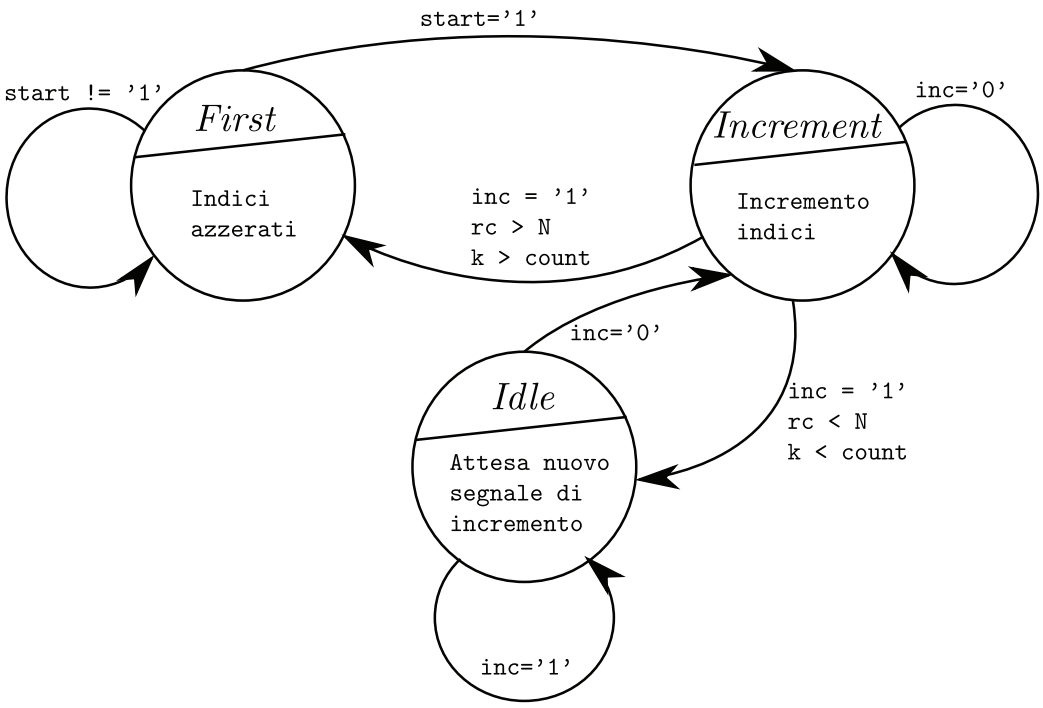


Figura 3.5: Stati della macchina

3.4.2 Blocco di generazione dei segnali di controllo

Prima di passare alla descrizione del **blocco di generazione dei segnali di controllo**, alla luce di quanto descritto finora è bene riprendere in considerazione lo schema esposto all'inizio del capitolo (Figura 3.1).
Risulta importante comprendere che gli ingressi e le uscite dei vari blocchi sono connessi tra loro direttamente: le uscite della *butterfly* vanno direttamente agli ingressi della RAM mentre le uscite della RAM e delle ROM vanno direttamente agli ingressi della *butterfly*. Non è stato previsto alcun blocco intermedio (tra un'uscita ed il rispettivo ingresso) poiché il flusso dei dati è direttamente controllato abilitando o inibendo i segnali di clock di ognuno dei diversi blocchi.

L'operazione svolta dal **blocco di generazione dei segnali di controllo** è proprio quella di controllare il flusso di dati all'interno dell'intero *core*

oltre che calcolare gli indirizzi di memoria (RAM e ROM) da cui prelevare e su cui memorizzare i dati. L'intera struttura è implementata nel file *SCU.vhd* e presenta i seguenti segnali in ingresso/uscita:

- `N`: come nel caso del **blocco di generazione degli indici** esso costituisce l'indice del numero di stadio che si sta eseguendo;
- `clk`: segnale di clock;
- `start`: segnale proveniente dall'**unità di controllo globale** che avvia l'esecuzione di ogni singolo stadio;
- `rst`: segnale di reset, portando a livello logico alto tale segnale è possibile riportare alle condizioni iniziali l'intero *core*;
- `eos`: segnale che indica all'**unità di controllo globale** il termine dell'intero algoritmo;
- `WE`: segnale che abilita la scrittura della RAM se portato a livello logico alto;
- `en_dout`: segnale che permette di abilitare la lettura della RAM dall'esterno del *core* se portato a livello logico alto;
- `addr_in`: se `en_dout` è a livello alto, è possibile scrivere su `addr_in` l'indirizzo della locazione di memoria da leggere;
- `CE_ROM`: segnale che abilita il clock della *ROM*;
- `CE_RAM`: segnale che abilita il clock della *RAM*;
- `CE_BUF`: segnale che abilita il clock della *butterfly*;
- `ROM_addr`: segnale che indica l'indirizzo della ROM che contiene il coefficiente W_N^k (equivalente a `k*(N/count*2)` nel (Codice 3.10));
- `data1_addr`: segnale che indica l'indirizzo della RAM che contiene il primo operando della *butterfly* (equivalente a `i+k` nel (Codice 3.10));
- `data2_addr`: segnale che indica l'indirizzo della RAM che contiene il secondo operando della *butterfly* (equivalente a `i+k+count` nel (Codice 3.10));

Per gestire l'intera operazione è stata utilizzata una particolare macchina a stati: i vari stati infatti si susseguono direttamente l'uno con l'altro e servono solamente per scandire nel tempo le diverse operazioni che l'algoritmo deve compiere. Più precisamente il compito svolto da questa unità è di prelevare gli operandi da porre all'ingresso della *butterfly* dalle corrette locazioni di memoria (RAM e ROM); successivamente di permettere alla *butterfly* di calcolare il risultato ed infine di salvare nelle apposite locazioni della RAM i valori ottenuti. Le operazioni appena descritte vanno chiaramente reiterate fino a concludere lo svolgimento dell'intero algoritmo.
Gli stati della macchina sono sei:

1. **addr_gen**: in questo stato la macchina genera, a partire da `k, rc` e `count` (provenienti dal **blocco di generazione degli indici**), gli indirizzi (`data1_addr, data2_addr` e `ROM_addr`), abilita poi il clock della RAM e delle ROM in modo che gli operandi della *butterfly* vengano inizializzati con i valori corretti. Al successivo ciclo di clock la macchina passa allo stato **addr_read** a meno che il segnale `eos_int` (proveniente dal **blocco di generazione degli indici**) non sia a livello alto, in tal caso la macchina passa allo stato **end_of_stage**.

2. **addr_read**: in questo stato vengono "letti" i risultati in uscita dalla *butterfly* abilitando il clock di quest'ultima. Questo tipo di operazione ha l'effetto di portare sui bus all'ingresso della RAM il risultato del calcolo della *butterfly*. Durante questa operazione, i segnali di clock di RAM e ROM sono disabilitati. Anche in questo caso, se il segnale `eos_int` è a livello alto, la macchina si porta allo stato **end_of_stage** altrimenti prosegue con lo stato successivo ovvero **butterfly_exec**.

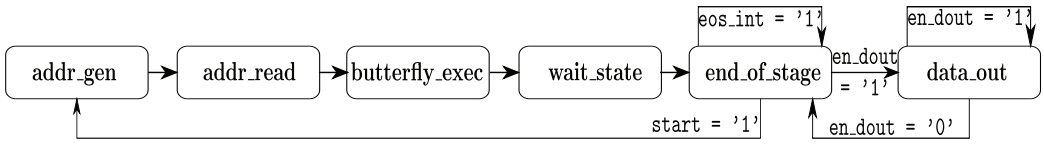
3. **butterfly_exec**: in questo stato vengono rigenerati gli indirizzi `data1_addr, data2_addr` e `ROM_addr`, inoltre vengono abilitati il clock ed il segnale di scrittura della RAM (`WE`) in modo tale da consentire la scrittura dei risultati del calcolo compiuto dalla *butterfly* nella RAM. Un'ulteriore operazione svolta da questo stato è quella di incrementare gli indici del **blocco di generazione degli indici** in modo da prepararli per il ciclo successivo. Una volta terminate le operazioni, la macchina passa allo stato **wait_state**.

4. **end_of_stage**: questo costituisce lo stato iniziale della macchina, nonché il più articolato degli stati. In tale stato tutti i segnali di controllo e gli indirizzi (`CE_ROM, CE_BUF, CE_RAM, WE, data1_addr, data2_addr, ROM_addr`) sono azzerati. Il compito di questo stato è di avviare un nuovo ciclo solamente nel caso in cui l'algoritmo non sia

terminato. Se il segnale `en_dout` è portato a livello logico alto allora la macchina passa allo stato **data_out** altrimenti viene controllato il segnale `start`. Nel caso in cui esso sia a livello logico alto la macchina passa allo stato **addr_gen** altrimenti viene controllato il livello del segnale `eos_int`, nel caso in cui sia a livello logico alto la macchina resta nello stato **end_of_stage** altrimenti passa allo stato **addr_gen**.

5. **wait_state**: questo stato non svolge in particolare alcuna operazione, ma serve poiché il **blocco di generazione degli indici** offre una latenza di un ciclo di clock nell'incremento degli indici e quindi lo stato **wait_state** ha il compito di attendere che l'incremento degli indici sia completato. Al suo termine la macchina passa allo stato **end_of_stage**.

6. **data_out**: questo stato è stato introdotto per motivi legati alla simulazione e permette di abilitare la lettura della RAM attraverso i due segnali `en_dout` e `addr_in`. Ciò permette di leggere il contenuto della memoria una volta terminato l'intero algoritmo.



Listing 3.13: Generazione degli indirizzi di memoria a partire da `count,k,rc`

```
data1_addr_int <=  k + rc; -- indirizzo del primo dato da mandare
                           -- alla butterfly
data2_addr_int <=  k + rc + count; -- indirizzo del secondo dato da mandare
                                   -- alla butterfly
ROM_addr_int <= shl(k(addr_bits-2 downto 0),conv_std_logic_vector(
                addr_bits-1-conv_integer(N),4)); -- indirizzo della posizione
                -- di seno e coseno da prelevare dalla rom
                --> k*(2^(addr_bits-1-N))
```

Listing 3.14: Esempio di sequenzializzazione dei dispositivi attraverso il controllo del segnale di clock

```
CE_ROM_int <= '1';
CE_RAM_int <= '1';
CE_buf_int  <= '0';
```

Capitolo 4

Simulazioni e funzionamento generale

4.1 Simulazioni dei blocchi

Verranno ora prese in esame le simulazioni relative alle principali strutture che compongono il *core*.

4.1.1 *Butterfly*

Il funzionamento della *butterfly* è stato simulato facendo in modo che ad intervalli regolari i segnali in ingresso presentassero dei valori scelti in modo arbitrario. Tali segnali sono stati inseriti direttamente, come valore numerico, nel file di testbench.
Il simulatore fornito nel pacchetto ISE (Software di sviluppo per i sistemi *Xilinx*) non prevede la possibilità di visualizzare i risultati della simulazione secondo una codifica personalizzata, ma solamente in modalità: binaria, esadecimale, decimale intera con segno e senza segno, ottale e ASCII. Per poter generare con facilità i valori numerici da inserire nel file di testbench e per poter verificare il corretto funzionamento della *butterfly* è stato predisposto uno script in MATLAB (contenuto nel file *test_butterfly.m*) in grado di simulare il funzionamento della *butterfly* e di generare, a partire da una serie di segnali in ingresso (limitati tra -1 e 1), i rispettivi valori in rappresentazione decimale intera con segno, relativi ai segnali in ingresso ed ai risultati in uscita. Per effettuare una simulazione è quindi sufficiente lanciare lo script *test_butterfly.m*, inserire gli operandi della *butterfly* ed attendere che lo script generi i risultati. Con i risultati ottenuti è quindi possibile inizializzare il file di testbench e verificare la correttezza del risultato ottenuto.

Ad esempio, nel caso simulato, ponendo gli ingressi della *butterfly* pari a:

$$\begin{cases} gr = 0.015 \\ gi = -0.008 \\ hr = 0.015 \\ hi = -0.008 \\ cos = 0.015 \\ sin = -0.008 \end{cases}$$

lo script *test_ butterfly.m* restituisce:

$$\begin{cases} gr = 3932 \\ gi = -2097 \\ hr = 3932 \\ hi = -2097 \\ cos = 3932 \\ sin = -2097 \\ Xr = 3932 \\ Xi = -2097 \\ Yr = 3932 \\ Yi = -2097 \end{cases}$$

cui corrisponde il codice:

Listing 4.1: Valori di simulazione per la *butterfly*

```
124 gr <= conv_std_logic_vector(3932,18); -- 15/1000
    gi <= conv_std_logic_vector(-2097,18); -- -8/1000
126 hr <= conv_std_logic_vector(3932,18); -- 15/1000
    hi <= conv_std_logic_vector(-2097,18); -- -8/1000
128 cos <= conv_std_logic_vector(3932,18); -- 15/1000
    sin <= conv_std_logic_vector(-2097,18); -- -8/1000
130 -- Risultato Xr =1985 ; Xi = -1049; Yr = 1956; Yi = -1049
```

Oltre a verificare che la *butterfly* esegua correttamente i calcoli, come si può vedere dalla (Figura 4.1), è stato anche verificato il corretto funzionamento del segnale di enable del clock.

Simulazioni non riportate

Nel corso dello sviluppo dell'intera *butterfly* sono state effettuate ulteriori simulazioni: sul moltiplicatore complesso e sui sommatori con controllo di overflow. Questo tipo di verifiche non vengono riportate in quanto si ritiene siano superflue e appesantiscano l'esposizione dell'argomento. É chiaro che tali simulazioni sono risultate necessarie per la corretta implementazione della

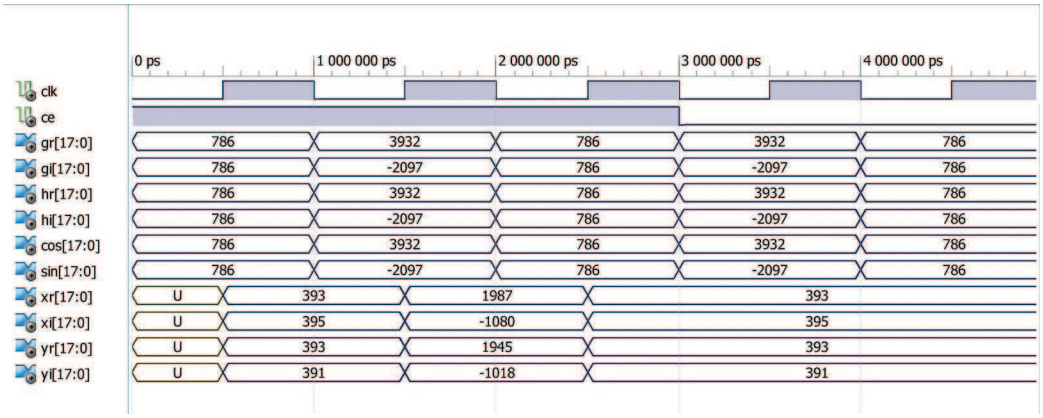


Figura 4.1: Simulazione del funzionamento della *butterfly*

butterfly, ma si considera che una volta dimostrato l'esatto funzionamento del dispositivo, non siano necessarie ulteriori verifiche. In ogni caso i file di testbench utilizzati per queste simulazioni sono compresi nel progetto e quindi restano a disposizione di chiunque desideri utilizzarli.

4.1.2 Unità di generazione degli indici

Per testare il funzionamento dell'unità di generazione degli indici, è stato costruito un file di testbench in grado di simulare il comportamento dell'algoritmo e quindi di generare i segnali di incremento degli indici per ognuno dei $log_2 N$ stadi.
La simulazione prevede che venga avviata l'esecuzione del primo stadio portando a livello logico alto il segnale di `start` e inviando all'unità il valore `N=0`; successivamente vengono incrementati ad intervalli regolari gli indici `k` ed `rc` agendo sul segnale `inc`. Se il segnale `eos`, in uscita dall'unità, è a livello logico alto ciò significa che è terminato uno stadio e quindi la simulazione riparte dall'inizio abilitando il segnale di `start`, ma questa volta, ponendo `N=1`. L'intera simulazione prosegue quindi incrementando `N`, stadio per stadio fino a quando $\texttt{N} = log_2 N - 1$, nel caso implementato $\texttt{N} = 9$. Per ottenere una simulazione quanto più veritiera possibile, i segnali `start` ed `inc`, sono stati mantenuti a livello logico alto per più di un ciclo di clock, in questo modo è stato possibile verificare il corretto funzionamento della macchina a stati interna all'unità. Può succedere infatti che (se il controllo degli stati della macchina non è gestito correttamente) abilitando i segnali per un unico ciclo di clock il dispositivo, funzioni nel modo giusto, ma mantenendoli a livello alto per più di un ciclo di clock vengano generati dei valori in uscita errati.
Infatti, durante le simulazioni, sono stati riscontrati problemi dovuti all'ef-

fetto del segnale `inc` sulle uscite: se esso veniva abilitato per un unico ciclo di clock, il dispositivo funzionava correttamente; al contrario se veniva lasciato a livello logico alto per più di un ciclo di clock, la macchina continuava ad incrementare i valori `k` ed `rc`; tale situazione generava un malfunzionamento dell'intero dispositivo.

Per confermare il corretto funzionamento del dispositivo, gli stessi indici `k` ed `rc`, sono stati generati tramite un programma in codice C (*indici.c*) che simula il funzionamento dell'unità implementata. Chiaramente non sono state confrontate una per una tutte le iterazioni; l'utilità del programma di generazione degli indici è quella di poter fornire un andamento generale dei segnali `k` ed `rc` in modo da poter fare un confronto in ogni momento della simulazione.
La verifica del corretto andamento degli indici è stata fatta principalmente basandosi sulle espressioni che li generano: $\texttt{k} = \texttt{k} + 1$ $(0 \leq \texttt{k} \leq \log_2 N - 1)$ ed $\texttt{rc} = \texttt{rc} + \texttt{count} \cdot 2$ $(0 \leq \texttt{rc} < N)$, è stata posta particolare attenzione alle situazioni limite, che corrispondono al passaggio tra uno stadio e il successivo e alla condizione di termine degli incrementi.

Nelle immagini sottostanti è possibile visualizzare il passaggio tra il primo e il secondo stadio (Figura 4.2) e il termine delle iterazioni (Figura 4.3). Si ponga particolare attenzione a `eos` che indica la fine dello stadio, al segnale di `start`, ai valori di `k` ed `rc` e ai segnali `k_inc`, `rc_inc`, `k_rst` e `rc_rst` che costituiscono i segnali di incremento e di reset dei singoli indici.

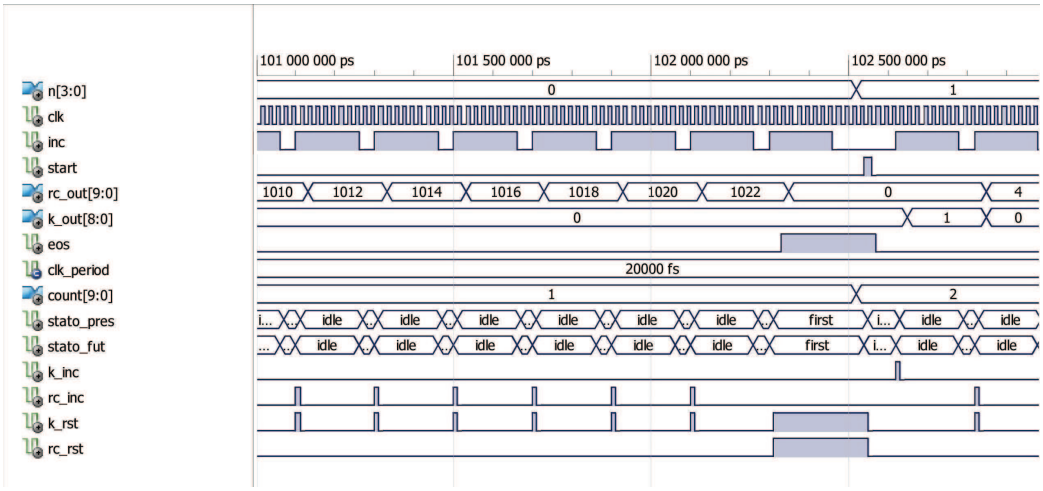


Figura 4.2: Particolare del passaggio tra primo e secondo stadio

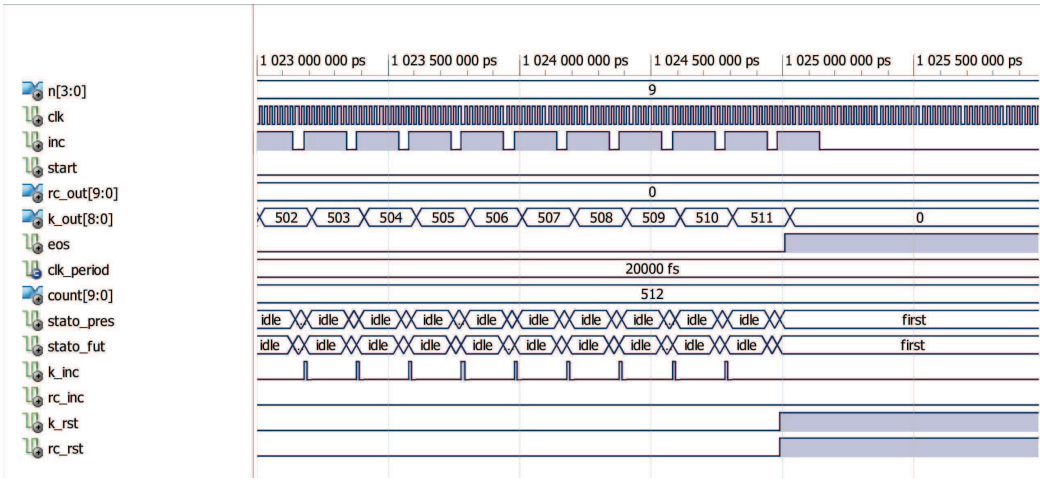
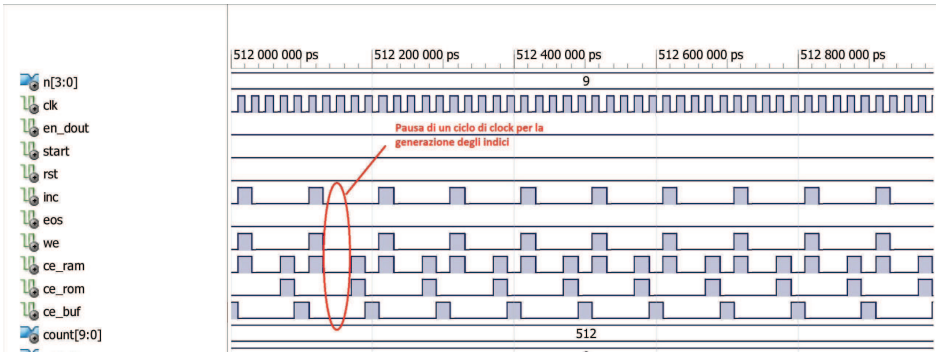


Figura 4.3: Fine delle iterazioni degli indici

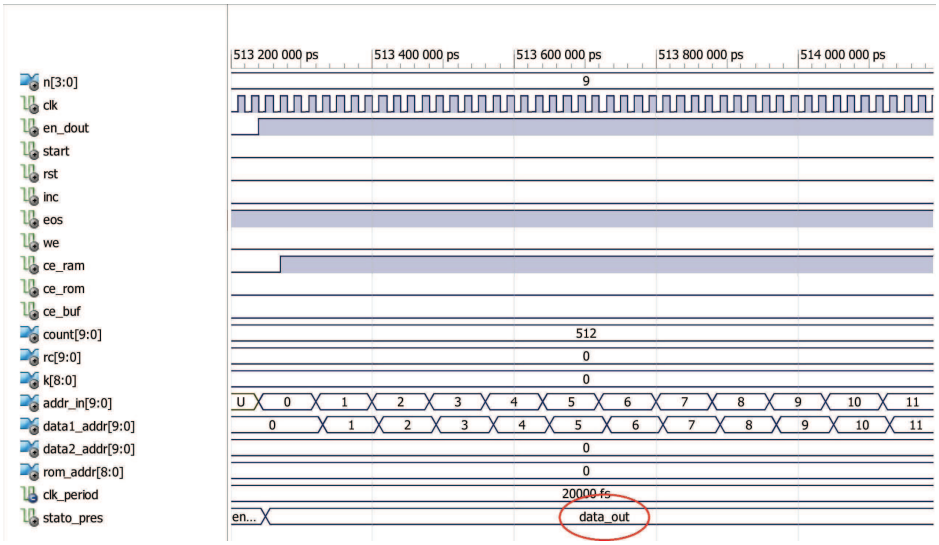
4.1.3 Unità di controllo dello stage

Il funzionamento dell'**unità di controllo dello stage** è stato simulato in maniera molto simile a quella del blocco precedente. Anche in questo caso è stato predisposto un file di testbench in grado di fornire i medesimi segnali di controllo utilizzati per l'**unità di generazione degli indici**, la cosa non dovrebbe stupire in quanto l'**unità di generazione degli indici** è contenuta all'interno dell'**unità di controllo dello stage** quindi è chiaro che le due unità condividono alcuni segnali.
Partendo dal presupposto che gli indici vengano generati correttamente (quindi che il funzionamento dell'**unità di generazione degli indici** sia esente da errori), in questa simulazione si è posta particolare attenzione all'andamento dei segnali di controllo ovvero ai diversi segnali di abilitazione del clock (`CE_RAM, CE_BUF, CE_ROM`), al segnale `WE` di abilitazione della scrittura sulla RAM, al segnale `inc` e agli indirizzi (`data1_addr, data2_addr, ROM_addr`).
Il risultato è corretto se dalla simulazione è possibile riscontrare che vengono abilitati: prima il segnale di clock delle ROM e della RAM, successivamente solo quello della *butterfly* e infine vengono abilitati i segnali di scrittura (`WE`) e di clock della RAM. Come è possibile vedere nella (Figura 4.4a) l'evoluzione dei segnali corrisponde alla descrizione precedente. Si noti in particolar modo la condizione di "pausa" di un ciclo di clock che si verifica in seguito all'abilitazione del segnale `inc`; come è stato spiegato nel paragrafo relativo alla descrizione dell'unità di controllo dello stage, ciò è dovuto al fatto che l'unità di generazione degli indici offre una latenza di un ciclo di clock nella generazione del risultato.

Per poter verificare la correttezza degli indirizzi generati è possibile utilizzare anche in questo caso il file *indici.c* che, oltre a generare gli indici `k` ed `rc`, calcola anche gli indirizzi relativi ad ognuna delle iterazioni.
Un'ulteriore simulazione condotta è stata quella relativa al caso in cui l'unità entri nello stato **data_out** in seguito all'abilitazione del segnale `en_dout`. In questo particolare stato il file di testbench scansiona tutte le locazioni di memoria della RAM, dalla 0 alla 1023 (che corrisponde al lavoro che dovrebbe effettuare l'**unità di controllo globale** al termine dell'esecuzione dell'algoritmo).
L'immagine (Figura 4.4b) riporta l'esito corretto di tale simulazione in cui è possibile vedere che l'indirizzo `data1_addr` corrisponde a `addr_in`.



(a) Simulazione del controllo sui segnali di clock



(b) Scansionamento della RAM

Figura 4.4

4.2 Simulazione generale del *core*

Il file che genera i segnali per la simulazione del *core* svolge il compito che dovrebbe compiere l'**unità di controllo globale** e quindi segue l'evolvere dell'algoritmo generando, stadio per stadio i vari segnali di avvio (`start`,`N`) e leggendo quelli di controllo (`eos`). Questa simulazione può risultare simile alle precedenti, ma in questo caso è da considerare che i vari blocchi che costituiscono lo stage sono interconnessi tra loro, non viene quindi simulata la singola unità, ma l'intero dispositivo. Per fare un esempio, se viene abilitato il segnale di clock della RAM, allora all'uscita della porta A della RAM risulterà esserci il contenuto della locazione di indice `data1_addr` il quale costituirà uno dei due operandi della *butterfly* e così via. In sostanza la simulazione è in grado di generare risultati numerici e quindi di decretare il corretto funzionamento del dispositivo stesso.
Come è stato detto nel paragrafo relativo all'implementazione della memoria RAM, essa risulta essere già inizializzata correttamente (ordinamento *bitreverse* dei campioni) con un segnale costituito da un periodo di una sinusoide. É stato utilizzato tale segnale poiché risulta essere il più chiaro ai fini della simulazione infatti il suo modulo in freqenza è una riga verticale, facile quindi da riconoscere.

Dovendo estrapolare i dati contenuti nella RAM per poterli poi utilizzare nella costruzione di un grafico in grado di riportare appunto l'andamento in frequenza del segnale, è stato necessario fare in modo che durante la simulazione venisse generato un file di testo in cui viene stampato il contenuto della RAM. La RAM è costituita da locazioni di 36 bit di cui, i primi 18 sono la parte reale del campione e i restanti 18 sono quella immaginaria. Sfruttando alcune funzioni presenti nella libreria `std_logic_textio` è stato possibile generare un file di testo costituito di 1024 righe ognuna delle quali contenenti 36 caratteri binari (0 e 1).

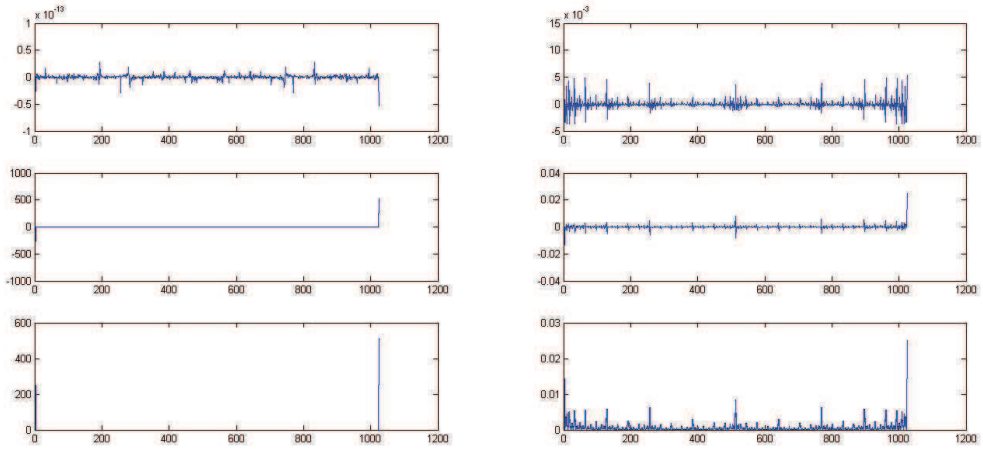
Listing 4.2: Scrittura del contenuto della RAM nel file `data_out.txt`

```
122 file_open(file_out, "../matlab/data_out.txt", write_mode);
    en_dout <= '1';
124 for i in 0 to 1023 loop
      addr_in <= conv_std_logic_vector(i,10);
126   wait for clk_period*2;
      x:= to_bitvector(dout);
128   write(oline,x,right,36);
      writeline(file_out,oline);
130   wait for clk_period*2;
    end loop;
132 file_close(file_out);
```

Per poter "leggere" ed interpretare con facilità il contenuto di questo file è stato predisposto uno script MATLAB (*plot_ data.m*) che si occupa di estrapolare le singole righe di 36 caratteri binari copiandole all'interno di una matrice. Successivamente le righe della matrice vengono scisse in parte reale e parte immaginaria e poi ognuna delle due componenti viene convertita da binario a decimale con segno. Infine i valori decimali vengono scalati in base alla codifica utilizzata.
Attraverso i dati così ottenuti viene costituito un grafico che comprende: (dall'alto) parte reale, parte immaginaria e modulo del segnale in frequenza.

In questo paragrafo non viene riportata alcuna immagine relativa all'evoluzione dei segnali poiché le precedenti simulazioni dovrebbero essere sufficienti a dimostrare la corretta evoluzione dell'algoritmo. Si preferisce invece mostrare i risultati numerici ottenuti dalla simulazione, nelle tre immagini sottostanti è possibile confrontare i risultati ottenuti con il medesimo segnale di ingresso da:

- (4.5a) Simulazione ottenuta utilizzando la funzione `fft()` di MATLAB;
- (4.5b) Simulazione ottenuta tramite la funzione `fft()` implementata in C al fine di comprendere il funzionamento dell'algoritmo;
- (4.6) Risultato della simulazione ottenuta dal dispositivo implementato.



(a) Simulazione in MATLAB (b) Simulazione in C

Figura 4.5: Simulazioni

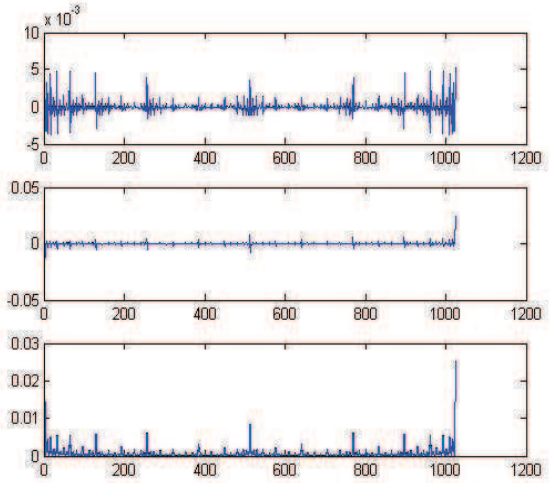


Figura 4.6: Risultato ottenuto dal dispositivo

4.3 Sintesi del dispositivo

In questo paragrafo vengono riportati i dati di sintesi del dispositivo. L'analisi dei risultati ottenuti dalla sintesi è utile alla comprensione della quantità di risorse utilizzate.

FFT Project Status (05/24/2010 - 17:04:53)			
Project File:	FFT.ise	**Implementation State:**	Synthesized
Module Name:	stage	• **Errors:**	
Target Device:	xc3s200-4ft256	• **Warnings:**	
Product Version:	ISE 11.1	• **Routing Results:**	
Design Goal:	Balanced	• **Timing Constraints:**	
Design Strategy:	Xilinx Default (unlocked)	• **Final Timing Score:**	

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	214	1920	11%
Number of Slice Flip Flops	183	3840	4%
Number of 4 input LUTs	405	3840	10%
Number of bonded IOBs	55	173	31%
Number of BRAMs	4	12	33%
Number of MULT18X18s	3	12	25%
Number of GCLKs	1	8	12%

Risulta evidente dai dati sopra riportati che il dispositivo implementato ha un'occupazione piuttosto limitata (uno degli obiettivi che si volevano raggiungere). Tra i diversi parametri c'è da porre particolare attenzione a **Number of MULT18x18s** e a **Number of BRAMs** che indicano rispettivamente la quantità di moltiplicatori embedded utilizzati e la quantità di BLOCK RAM (RAM embedded) utilizzata.

Capitolo 5

Conclusioni

5.1 Obiettivi raggiunti

Ricordando che l'obiettivo che si voleva raggiungere era la costruzione del *core* di un dispositivo in grado di svolgere l'algoritmo FFT, più precisamente l'intero dispositivo meno l'**unità di controllo globale**, si può affermare che tale meta è stata raggiunta. In particolar modo le specifiche iniziali richiedevano di implementare una struttura che:

- occupasse un'area limitata;
- fosse parametrizzata nelle sue principali grandezze (numero di campioni da elaborare, n° di bit per parola);

Entrambe le specifiche sono state rispettate: per quanto riguarda l'area l'ottimizzazione è stata ottenuta sfruttando un numero minimo di risorse (3 moltiplicatori per effettuare la moltiplicazione complessa) e cercando di evitare parallelizzazioni delle strutture (piuttosto che implementare $N/2$ *butterfly* all'interno dello stadio si è preferito, reiterare $N/2$ volte il calcolo all'interno di un'unica *butterfly*).
Per quel che riguarda la parametrizzazione, il codice è tale da permettere in ogni momento di cambiare i coefficienti che determinano: il numero di campioni utilizzati (agendo sul valore `addr_bits` che corrisponde al $log_2 N$ con N pari al numero di campioni), il numero di bit per ogni campione (agendo sul valore `data_bits`) e il numero di bit utilizzati per esprimere la parte intera di ogni singolo campione (agendo sul valore `n_int_bits`). In seguito all'utilizzo di dispositivi embedded (RAM) tali valori sono stati bloccati, è possibile quindi elaborare un maggior o minor numero di campioni solamente andando a modificare la dimensione delle ROM e della RAM. Anche per

quanto riguarda il numero di bit per campione è necessario apportare delle modifiche alla moltiplicazione se si vuole incrementare o diminuire tale valore. Il numero di bit utilizzati per esprimere la parte intera invece può essere modificato a piacimento (entro i limiti fisici del dispositivo) senza pregiudicare il corretto funzionamento della macchina.

5.2 Problemi esistenti e possibili soluzioni

Il dispositivo implementato ovviamente non è esente da alcune problematiche che sono principalmente legate al calcolo. Per prima cosa l'overflow: benché siano state adottate misure in grado di limitare il verificarsi di tale problema, è ancora possibile che in fase di calcolo si generi overflow. Una possibile soluzione a tale problema potrebbe essere quella di evitare di dividere per 2 i segnali all'ingresso della *butterfly* e utilizzare invece sommatori(e sottrattori) che restituiscono un risultato a $b+1$ bit, di cui vengono prelevati solo i b bit più alti (più vicini all'**MSB**). Con questo metodo è possibile ottenere, all'uscita dei sommatori, un risultato che è pari alla metà della somma dei due addendi, il risultato inoltre non può mai generare overflow. Dato che la struttura della *butterfly* risulta essere molto complessa, per poter sfruttare questo tipo di soluzione è necessaria un'analisi approfondita del flusso dei dati attraverso la struttura stessa. L'approccio appena descritto deve essere infatti tale per cui vengono sommati o moltiplicati tra loro numeri che sono stati scalati (divisi per due) lo stesso numero di volte.

Un ulteriore problema che si verifica in fase di calcolo si ha in seguito ad una moltiplicazione. Nel momento in cui si esegue la moltiplicazione $1 \cdot 1$, cui corrisponde (in binario) $01\ldots1 * 01\ldots1$ il risultato restituito dal moltiplicatore non risulta essere pari a 1 quindi $01\ldots1$ come ci si aspetterebbe, ma risulta invece pari a $\frac{2^{b-n\_int\_bits}-2}{2^{b-n\_int\_bits}-1}$ cui corrisponde la parola binaria $01\ldots10$. Tale errore, pur essendo di piccola entità, a mano a mano che b cresce, pregiudica la precisione del risultato. Questo genere di problema è legato al tipo di codifica utilizzata, una possibile soluzione potrebbe essere quella di aumentare il numero di bit dedicati alla rappresentazione della parte intera del campione, pur continuando ad esprimere numeri compresi tra -1 e 1.

5.3 Ulteriore sviluppo

Volendo migliorare il dispositivo per renderlo più performante sia in termini di rapidità di esecuzione del calcolo sia per quanto riguarda l'occupazione d'area è possibile agire principalmente sulle memorie ROM. Nel dispositivo implementato sono state utilizzate due ROM, una per memorizzare i valori del seno e l'altra per i valori del coseno, risulta evidente che è possibile evitare di utilizzare una delle due ROM sapendo che $cos(\alpha) = sin\left(\alpha + \frac{\pi}{2}\right)$. In base a tale espressione è possibile utilizzare un'unica ROM, per esempio dual port, dalla quale vengono letti i valori semplicemente traslando l'indirizzo di base di un fattore pari a $\frac{N}{4}$.

Un'ulteriore soluzione può essere quella di eliminare le ROM e sostituirle con un'unità in grado di svolgere l'algoritmo CORDIC, utilizzato appunto per generare i valori di seno e coseno senza l'ausilio di memorie ROM (o lookup table).

Bibliografia

[1] L. Benettazzo and C. Narduzzi. *Dispense di Misure per l'automazione e la produzione industriale*. Edizioni Libreria Progetto Padova, 2008.

[2] Gianfranco Cariolaro, Gianfranco Pierobon, and Giancarlo Calvagno. *Segnali e Sistemi*. McGraw-Hill, 2004.

[3] A. A. Ghouwayel and Y. Louët. Fpga implementation of a reconfigurable fft for multi-standard systems in software radio context. *IEEE*, 2009.

[4] Giada Giorgi. Dispensa del corso di: misure per l'automazione e la produzione industriale. Technical report, Università degli studi di Padova, Facoltà di Ingegneria, 2010.

[5] Brian W. Kernighan and Dennis W. Ritchie. *Il linguaggio C*. Pearson Prentice Hall, 2008.

[6] Gianantonio Mian. *Appunti di Elaborazione Numerica dei Segnali*. Edizioni Libreria Progetto Padova, 1998.

[7] *Spartan-3 FPGA Family Data Sheet*, 2008. http://www.xilinx.com/support/documentation/data_sheets/ds099.pdf.

[8] *Spartan-3 Generation FPGA User Guide*, 2009. http://www.xilinx.com/support/documentation/user_guides/ug331.pdf.

[9] Daniele Vogrig. Dispensa del corso di: laboratorio di elettronica digitale. Technical report, Università degli studi di Padova, Facoltà di Ingegneria, 2009.

[10] Mark Zwolinski. *VHDL Progetto di sistemi digitali*. Pearson Prentice Hall, 2007.

Ringraziamenti

Desidero ringraziare la mia famiglia, gli amici e tutti coloro i quali mi hanno sempre sostenuto e spronato ad impegnarmi e a dare il meglio di me stesso.

Un particolare ringraziamento va al prof. Vogrig per la sua professionalità e disponibilità, entrambe sempre presenti in questi mesi di tesi.