
# Politecnico di Torino

Sistemi Digitali Integrati
A.A. 2019/2020

---

# Fast Fourier Transform

---

Prof. M. Zamboni

*Autori*:

| | |
|---|---|
| Corongiu Francesco | 236180 |
| Mippi Pierpaolo | 237986 |
| Ramella Livrin Mattia | 277897 |

17 febbraio 2020

# Indice

# 1 Introduzione

La seguente relazione spiega come è stata realizzata un'unita di elaborazione che esegue la FFT (Fast Fourier Transform). Essa è un algoritmo ottimizzato per il calcolo della trasformata discreta di Fourier (DFT) portando generalmente la complessità computazionale dall'ordine $O(N^2)$ a $O(Nlog(N))$. Per il progetto è stato richiesto un particolare algoritmo per realizzarla: il più diffuso algoritmo di Cooley-Tukey.

## 1.1 Algoritmo e Butterfly

L'algoritmo di Cooley-Tukey permette di minimizzare il numero di operazioni ripetute, scomponendo ricorsivamente la trasformata come somme di trasformate su insiemi più piccoli di valori di ingresso.
Partendo dalla formula della DFT:

$$X_N(k) = \sum_{n=0}^{N-1} x(n) e^{-j\frac{2\pi}{N}nk} \tag{1}$$

essa viene riscritta:

$$X_N(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk} \tag{2}$$

dove i W sono detti twiddle factor espressi come:

$$W_N^m = cos(\frac{2\pi}{N}m) - jsin(\frac{2\pi}{N}m) \tag{3}$$

La singola operazione necessita di soli tre dati (due ingressi A e B e il twiddle factor W) ed è chiamata butterfly (figura 1). Essa produce due uscite $A'$ e $B'$ date dalle seguenti operazioni:

$$A' = A + B * W^k$$

$$B' = A - B * W^k$$

Nel progetto è stata realizzata inizialmente una butterfly partendo dalla struttura del DFD (Data Flow Diagram) per poi realizzare il Datapath, studiando l'opportuno parallelismo e il miglior protocollo ingresso uscita. E' stato scelto il minor numero di bus come descritto nel sottoparagrafo dedicato. Succesivamente è stata realizzata la *CU (Control Unit)* appoggiandosi sulla *Control ASM* per caricare la memoria con i bit controllo e per studiare il salto di indirizzo che permette alla Butterfly di lavorare in full-speed.
Si è scritto il codice VHDL utilizzando l'ambiente di sviluppo *Quartus* inserendo gli elementi base (multiplexer, registri,sommatori e sottrattori) insieme al register file (di cui si è studiato il funzionamento e semplificato per il progetto) e il particolare moltiplicatore che unisce uno shifter, un moltiplicatore base ed una serie di registri come è spiegato nel sottoparagrafo dedicato. Infine si è verificato il corretto funzionamento della Butterfly utilizzando il simulatore *Modelsim*

dopo aver costruito un *testbench* specifico, modificando i dati in ingresso per individuare gran parte dei possibili casi.

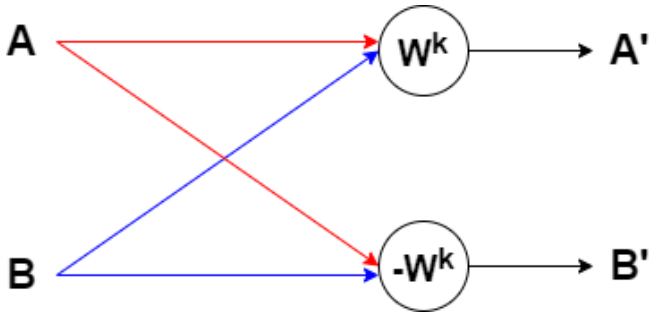


Figura 1: Grafico segnali Buttefly

## 1.2 FFT

La Butterfly è stata succesivamente utilizzata per realizzare una FFT 16x16 collegando opportunamente 32 Butterfly utilizzando *Quartus*. Mediante *Modelsim* si è confrontato il risultato delle uscite con il file contenente esempi di ingressi e uscite di FFT, si lascia la verifica alla sezione dedicata alle simulazioni.
In particolare è stato scelto di rendere la FFT più veloce studiando tramite il DFD la concatenazione migliore degli stadi di Butterfly in modo tale da risparmiare colpi di clock.

# 2 Data Flow Diagram

Il Data Flow Diagram è stato realizzato a partire dalle operazioni ricavate mediante l'algoritmo di Cooley Tukey e a successive semplificazioni per permettere un numero ridotto di calcoli da eseguire. Sapendo che all'uscita di una singola butterfly si hanno i seguenti risultati:

$$A' = (A_R + jA_I) + (W_R + jW_I) * (B_R + jB_I) = (A_R + B_R W_R - B_I W_I) + j(A_I + B_R W_I + B_I W_R) \quad (4)$$

$$B' = (A_R + jA_I) + (-W_R - jW_I) * (B_R + jB_I) = (A_R - B_R W_R + B_I W_I) + j(A_I - B_R W_I - B_I W_R) \quad (5)$$

E' possibile dunque individuare dalle due equazioni 12 operatori:

$$M1 = B_R W_R \quad (6)$$

$$M2 = B_I W_I \quad (7)$$

$$M3 = B_R W_I \quad (8)$$

$$M4 = B_I W_R \quad (9)$$

$$M5 = 2A_R \quad (10)$$

$$M6 = 2A_I \quad (11)$$

$$E1 = A_R + M1 \quad (12)$$

$$E2 = E1 - M2 \quad (13)$$

$$E3 = A_I + M3 \quad (14)$$

$$E4 = E3 + M4 \quad (15)$$

$$E5 = M5 - E2 \quad (16)$$

$$E6 = M6 + E4 \quad (17)$$

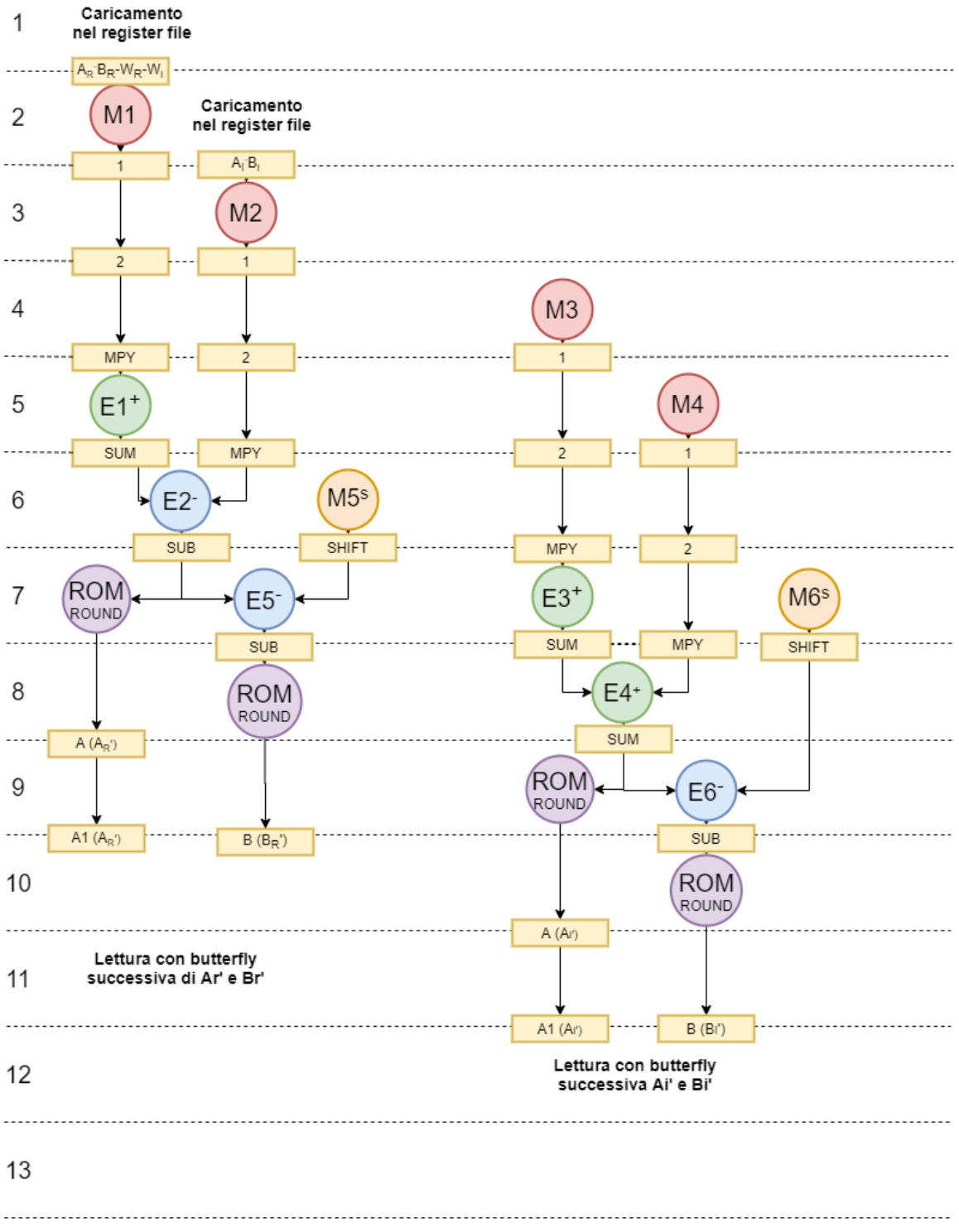


Figura 2: CDFD butterfly

Ordinando questi ultimi e considerando le limitazioni hardware è stata ottenuta una configurazione ottimale per eseguire l'intero algoritmo con un numero di colpi di clock minore possibile.

Come prima operazione, subito dopo aver campionato i valori reali di A e B derivanti dall'esterno e i twiddle factor necessari alla butterfly, viene eseguita la moltiplicazione M1: essa restituirà il risultato solamente dopo due colpi di clock, tempo che può essere sfruttato per eseguire altre operazioni. Insieme alla prima moltiplicazione inoltre vengono campionate le parti immaginarie di A e B, permettendo così di dimezzare il parallelismo degli ingressi siccome sono dati non necessari per M1.
Nei due colpi di clock succesivi vengono fatte partire le moltiplicazioni M2 e M3, e solamente al 4° step viene salvato in un registro il risultato di M1. Disponendo le moltiplicazioni in questo modo si riesce a utilizzare un solo moltiplicatore e si riduce al minimo il tempo di esecuzione sfruttando la pipeline interna ad esso. In uscita al moltiplicatore, sia per quanto riguarda la parte di shift che l'altra sono stati previsti due registri per permettere di memorizzare il dato prima di mandarlo all'interno del sommatore o sottrattore che sono due componenti puramente combinatori.
A partire dal 5° colpo di clock viene eseguita la prima somma E1, la quale non va in conflitto con M4 siccome necessita di operandi differenti; essa viene eseguita solamente in questo istante dato che richiede il risultato di M1, che diventa disponibile proprio ora. In uscita al sommatore e al sottrattore analogamente a quanto visto con il moltiplicatore, sono stati inseriti dei registri per salvare i valori calcolati ed eventualemente mantenerli per più di un colpo di clock.
Nel 6° step invece il moltiplicatore da un lato sta ancora eseguendo M4 e ha terminato M3, dall'altro esegue M5, la quale però trattandosi di un prodotto per una costante ha un percorso combinatorio differente e non richiede 2 colpi di clock per essere svolta. Contemporaneamente E2 calcola il primo dato che sarà uno dei 4 risultati in uscita, ovvero la parte reale di $A^{'}$. Questo risultato viene inviato successivamente al blocco relativo al Rom Rounding, il quale necessita di un colpo di clock per svolgere l'approssimazione, salvando così il valore finale in un ulteriore registro.
Nello step successivo nel blocco sottrattore eseguendo E5 viene calcolata la parte reale di $B^{'}$, eseguendo sempre in parallelo sia la somma E3 che la moltiplicazione M6. Nel 8° colpo di clock viene solamente eseguita l'ultima somma rimanente, cioè E4, che restituisce la parte immaginaria di $A^{'}$, e nel nono step E6 restituisce la parte immaginaria di $B^{'}$, terminando il calcolo di tutti i valori.
Come si può osservare negli ultimi step del CDFD (figura 2) è stata scelta l'opzione di salvare temporaneamente il valore delle parti reali in un registro intermedio (A), il quale ha il compito di permettere al colpo successivo di estrarre dalla butterfly entrambi i valori reali di $A^{'}$ e $B^{'}$. Stessa considerazione è stata fatta per le parti immaginarie, e in questo modo è stato ripreso lo stesso protocollo con il quale vengono inseriti i dati nella butterfly, consentendo di collegarla a un'altra identica senza avere problemi nel campionamento dei valori. Si può notare, infatti, che è possibile disporre in maniera continua tale schema a simulare due butterfly collegate tra loro, una che rende disponibili in uscita i risultati e l'altra che li campiona riducendo al minimo il tempo morto tra le due operazioni.
Per quanto riguarda il numero di colpi di clock necessari per lo svolgimento completo delle operazioni, se si considera come punto di partenza il campionamento delle parti reali e come punto di arrivo la loro uscita dalla butterfly si ha una latenza di 10 colpi di clock, che in questo caso non essendoci stadi di pipeline interni coincide con il throughput.

## 2.1 Modalità full-speed

Per cercare di rendere il throughput inferiore in caso di elaborazione continua di dati, si è cercato di eliminare i tempi di attesa tra un'elaborazione e un'altra all'interno di una butterfly, eseguendo più operazioni possibili in contemporanea evitando dunque di sovrascrivere dati ancora necessari.

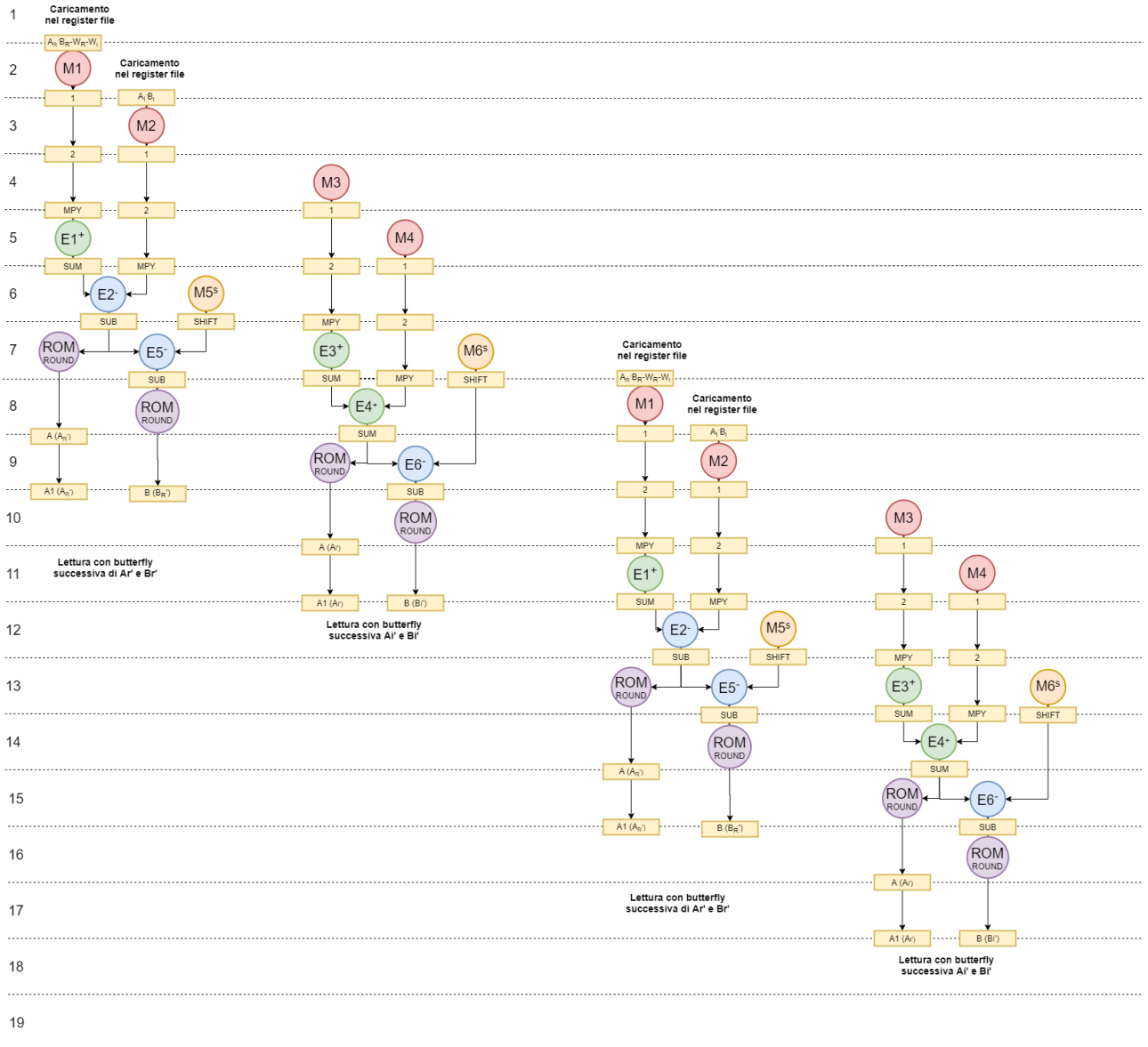


Figura 3: CDFD full speed butterfly

Il problema principale consiste nel moltiplicatore che lavora per ben sei colpi di clock consecutivamente, non permettendo dunque di far partire un nuovo calcolo, poiché la prima operazione della butterfly è rappresentata da M1. E' stato, dunque, necessario far partire la nuova elaborazione all'ottavo colpo di clock, e dunque il campionamento delle parti reali il colpo precedente.

Si è studiato, inoltre, il metodo con il quale far capire alla butterfly che sta lavorando in modalita continua: se il segnale di start è asserito nel colpo di clock n°6 allora può partire la memorizzazione, altrimenti si sta lavorando in modalità singola e i dati terminano di essere processati secondo quanto spiegato precedentemente. Anche nel caso in cui una butterfly debba comunicare alla successiva di elaborare in dati in full speed, il ragionamento è ancora valido, ed è solamente necessario asserire un segnale che identifichi tale richiesta al 6° step, analogamente a come viene fatto per i dati provenienti dall'esterno.
Disponendo in quest'ordine le operazioni da svolgere i dati inizialmente memorizzati relativi a parte reale e immaginaria non sono più necessari e possono essere rimpiazzati da quelli nuovi; stesso ragionamento può essere fatto per l'uso del sommatore e del sottrattore, i quali terminano di elaborare i dati prima che siano richiesti per svolgere i calcoli successivi. Anche il Rom Rounding e i relativi registri non vanno a ostacolare la modalità full speed.
Concatenando in questo modo le operazioni da svolgere la latenza dei dati all'interno di una singola butterfly rimane la stessa, mentre il throughput viene ridotto notevolmente, passando da 10 colpi di clock a 6.

# 3 Datapath Butterfly

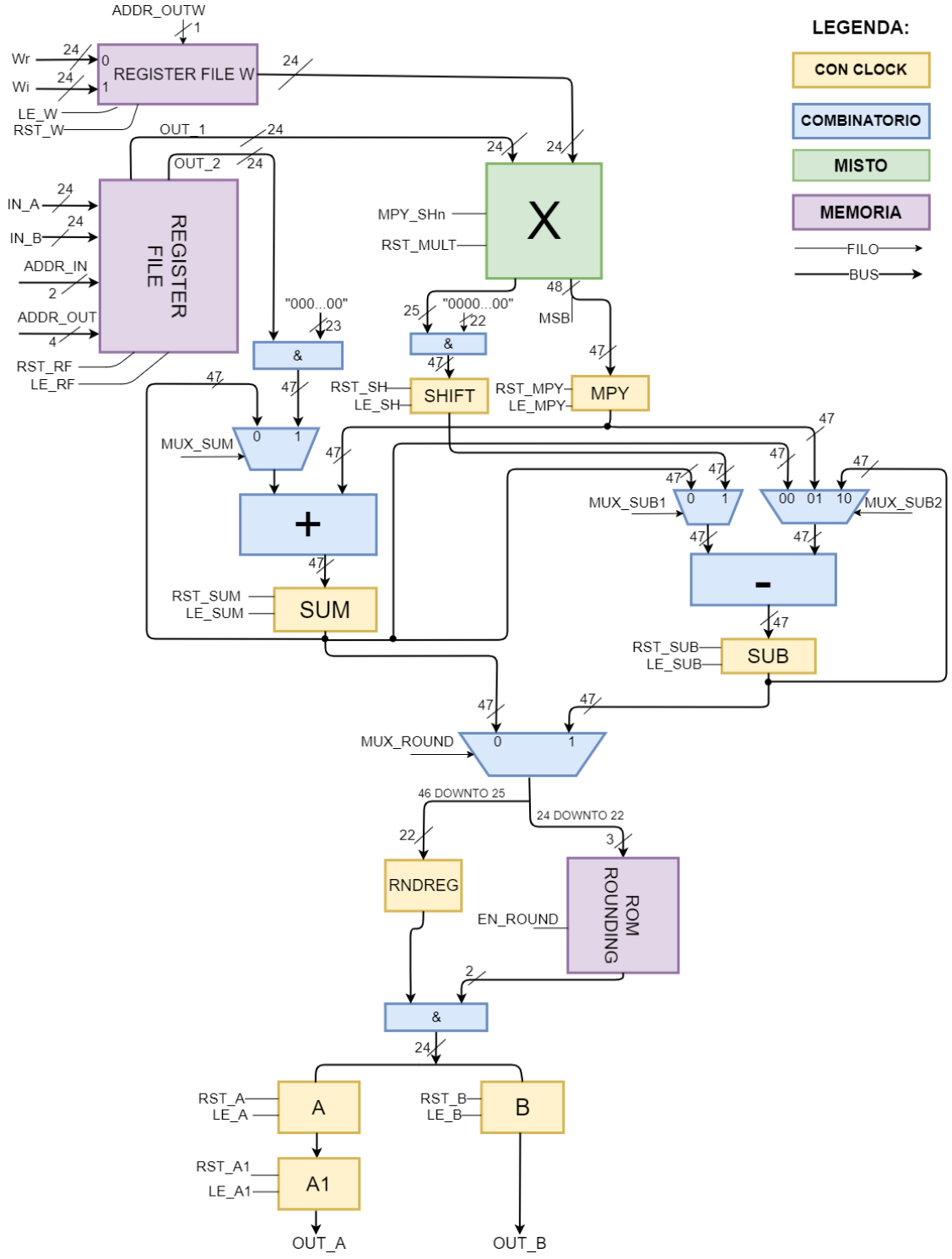


Figura 4: Datapath Butterfly

Il datapath mostrato in figura 1 rappresenta il modello strutturale ottimizzato che realizza il componente alla base della Fast Fourier Transform: il processore Butterfly.
I segnali in ingresso sono salvati all'interno di due diversi register file: da una parte i campioni e dall'altra i twiddle factor, entrambi caratterizzati da una parte reale e una parte immaginaria. La soluzione di separare i due tipi di ingresso ha permesso una più semplice gestione dei dati dal punto di vista circuitale: la presenza di una unica struttura di memoria ne avrebbe, infatti, complicato sia il salvataggio sia l'uso rispetto al protocollo di interfacciamento scelto. In questo modo, inoltre, si gestiscono separatamente dati che arrivano da luoghi differenti: mentre i campioni possono arrivare dall'esterno o da una precedente butterfly, i twiddle factor sono forniti da registri esterni alle butterfly.
I dati in uscita dai register file sono forniti a due blocchetti: un "moltiplicatore" e un sommatore.
Il primo, in realtà, esegue due operazioni: la moltiplicazione per 2, tramite shift di una posizione a sinistra di uno dei due dati in ingresso, e la normale moltiplicazione degli ingressi forniti. Il risultato di quest'ultima, effettuando moltiplicazioni frazionarie in CA2 con numeri minori di 1, è disponibile su 2n-1 bit. Poiché la moltiplicazione è descritta in modo comportamentale in linguaggio VHDL, l'uscita viene comunque fornita su 2n bit con il raddoppio del bit di segno. Per correggere il valore, in uscita il dato è troncato del Most Significant Bit.
Mentre, l'uscita dello shift è disponibile immediatamente, la moltiplicazione presenta due livelli di pipeline che rendono il risultato disponibile dopo due colpi di clock.
Entrambe le uscite del blocchetto "moltiplicatore" sono salvate all'interno di due registri, SHIFT e MPY, prima di essere fornite ai successivi componenti del datapath. Per rispettare il parallelismo interno, pari a 47 bit, il risultato dello shifter aritmetico, prima di essere salvato nel corrispondente registro, viene esteso aggiungendo ventidue '0' (zero fill) di seguito al LSB.
Gli ingressi del blocchetto, invece, provengono dai due register file: le operazioni di moltiplicazione, come si può notare dal Data Flow Diagram, coinvolgono, infatti, solamente gli ingressi A e B e i twiddle factor.
Il sommatore, puramente combinatorio, presenta due ingressi: l'uscita del registro MPY contenente il risultato della moltiplicazione e l'uscita di un multiplexer a due vie. Quest'ultimo permette di selezionare, in base all'operazione da effettuare, tra un dato contenuto nel register file, previamente esteso su 47 bit tramite inserimento di '0' di seguito al LSB, e il risultato stesso della somma salvata nel registro SUM.
Nel datapath è, inoltre, presente un blocchetto sottrattore puramente combinatorio. Per svolgere tutte le operazioni di sottrazione richieste dalla Butterfly, sono posti in ingressi al blocchetto due multiplexer.
Per la scelta del minuendo, è presente un multiplexer a due vie, i cui ingressi provengono dai registri SUM e SHIFT contenenti i risultati delle rispettive operazioni.
Il sottraendo è scelto, invece, attraverso un multiplexer a tre vie: gli ingressi provengono dai registri SUM, MPY e dal registro di uscita della sottrazione stessa SUB.
Una volta terminate le operazioni aritmetiche, per rientrare su 24 bit i dati in uscita dalla Butterfly vengono troncati tramite tecnica del ROM ROUNDING attraverso l'omonimo blocchetto. I dati da troncare provengono, in modo alternato, dai registri SUM e SUB; per questo motivo è presente un multiplexer a due vie che ha il compito di selezionare il risultato opportuno da

troncare.
Il blocchetto di troncamento è una memoria ROM (read-only memory) che contiene al suo interno i valori da arrotondare. I 3 bit del dato ricevuti in ingressi (i due bit meno significativi dei 24 più il successivo extra bit) fungono da indirizzo per la scelta dei giusti bit di arrotondamento in uscita.
Questa operazione avviene attraverso una memoria, perciò è necessario un colpo di clock per generare il valore arrotondato; per questo motivo i restanti 22 bit del dato sono ritardati di un colpo di clock tramite il registro RNDREG. I bit in uscita da quest'ultimo registro e quelli provenienti dal ROM ROUNDING sono concatenati in modo da generare il dato in uscita. Come si può notare dal DFD, figura 2, le uscite sono disponibili nel seguente ordine:

- $A'_r$;
- $B'_r$;
- $A'_i$;
- $B'_i$.

Poiché il protocollo di interfacciamento con l'esterno prevede di fornire ad un colpo di clock le parti reali di $A'$ e $B'$ e al successivo le parti immaginarie, il dato $A'_r$ viene ritardato tramite il registro intermedio A. In questo modo, le due uscite $A'_r$ e $B'_r$ vengono campionate dai registri di uscita A1 e B nello stesso colpo di clock. Stesso comportamento si riscontra con le rispettive parti immaginarie.

## 3.1 Rom Rounding

Il processore Butterfly svolge al suo interno operazioni di moltiplicazione frazionarie in CA2 su 24 bit che forniscono un'uscita su 47 bit.
Il datapath è stato studiato affinché le operazioni al suo interno avvenissero senza alcun tipo di troncamento.
Poiché, però, in uscita parte reale e immaginaria di $A'$ e $B'$ devono essere comunque forniti su 24 bit, alla fine delle operazioni viene effettuato un arrotondamento tramite la tecnica del ROM ROUNDING.
Questa utilizza una LUT, realizzata tramite una memoria ROM, che possiede all'interno i valori da arrotondare e riceve in ingresso un sottoinsieme di bit della parte significativa del dato da mantenere e dei bit della parte da tagliare.
Nel caso del progetto in esame, si è studiata una soluzione che permettesse di avere sia una ROM non troppo grande e perciò prestazionale sia un numero di casi che non rendesse eccessivo l'errore di approssimazione.
Come mostrato in figura 5 si è deciso di prendere in considerazione un bit della parte da troncare (d) e due bit della parte significativa del dato (L).
La ROM, perciò, presenta 8 locazioni di memoria indirizzate da 3 bit in ingresso. In tabella 1 sono riportati i valori arrotondati nella LUT in base agli ingressi ricevuti e l'errore che si commette nell'approssimazione.

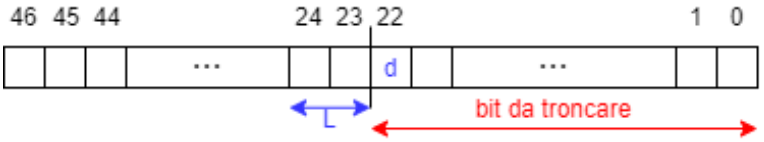


Figura 5: Bit considerati per il troncamento

Il troncamento del dato porta, ovviamente, ad una diminuzione della precisione numerica; in questo caso, l'errore di troncamento massimo che si può realizzare è pari a 1/2 LSB.
Inoltre, si può calcolare l'errore di bias dato da:

$$bias = \frac{1}{2}[(\frac{1}{2})^d - (\frac{1}{2})^L] \tag{18}$$

Sostituendo i valori di L e d si ottiene un errore medio commesso pari ad 1/8.
Come si può notare dalla formula 18 l'errore medio che si commette è legato al numero di bit che vengono presi in considerazione: maggiore è il numero di bit e minore è l'errore.
Nel caso della Butterfly, poiché il peso dei bit da approssimare non hanno un impatto determinante sul valore complessivo del dato (essendo il dato espresso in fixed point su 24 bit), si è deciso di considerare 3 bit in totale in modo da avere una ROM piccola, risparmiando risorse, e veloce.

| INGRESSO | ROM | ERRORE |
|---|---|---|
| X00.0 | X00 | 0 |
| X00.1 | X01 | +1/2 |
| X01.0 | X01 | 0 |
| X01.1 | X10 | +1/2 |
| X10.0 | X10 | 0 |
| X10.1 | X11 | +1/2 |
| X11.0 | X11 | 0 |
| X11.1 | X11 | -1/2 |

Tabella 1: Valori arrotondati in base agli ingressi e errore commesso

## 3.2 Moltiplicatore

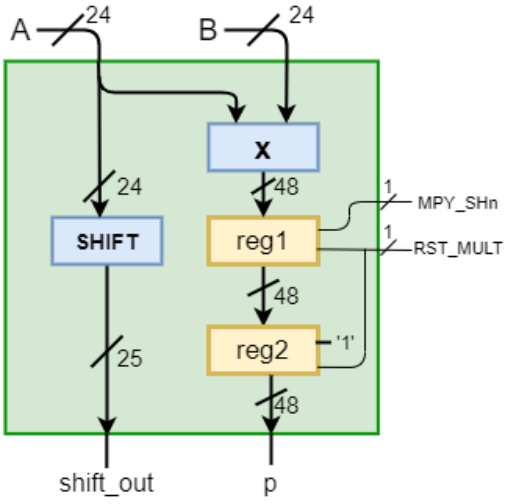


Figura 6: Schema moltiplicatore

Il blocchetto "moltiplicatore" presentato in figura 6 è caratterizzato da due ingressi: il primo è in comune sia all'operazione di shift (moltiplicazione per 2) che a quella di moltiplicazione tra due dati, dove rappresenta il moltiplicando; il secondo ingresso, invece, rappresenta il moltiplicatore. L'operazione di shift è puramente combinatoria, perciò non presenta latenza. L'operazione di moltiplicazione, invece, presenta due livelli di pipeline: i due registri, reg1 e reg2, rendono disponibile il risultato all'esterno dopo due colpi di clock (latenza di due cicli). Il primo registro di PIPE è abilitato tramite il segnale MPY_SHn che arriva dalla Control Unit: è asserito solamente nel momento in cui è richiesta una operazione di moltiplicazione tra due dati. Nel caso di shift, invece, il segnale è negato ('0') e il registro SHIFT esterno al componente viene abilitato. Il secondo registro di PIPE, invece, è sempre abilitato e campiona ad ogni colpo di clock il dato contenuto in reg1. I reset di reg1 e reg2 sono in comune tramite il segnale RST_MULT.
L'operazione di moltiplicazione per due (shift aritmetico) consiste nello scalamento a sinistra del dato in ingresso: il MSB viene troncato e di seguito al LSB vengono inseriti due '0'. Dal punto di vista di descrizione in VHDL:

$$shift\_out <= A(N-2\ DOWNTO\ 0)\ \&\ \text{“00”}; \tag{19}$$

L'operazione di moltiplicazione tra due dati, invece, è stata descritta a livello comportamentale.

## 3.3 Struttura di memorizzazione

Le variabili in ingresso al processore Butterfly, i dati e il twiddle factor, sono salvati all'interno di due diversi register file; questi fungono da "barriera" di registri permettendo la sincronizzazione degli ingressi che diventano, così, scorrelati da eventuali ritardi esterni.

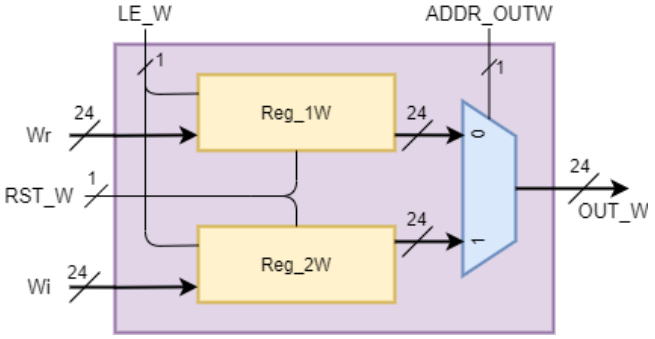


Figura 7: Schema register file twiddle factor

Il register file del twiddle factor, figura 7, caratterizzato da due porte in ingresso e una in uscita, contiene due registri: nel primo, Reg_1W, viene salvata la parte reale del dato mentre, nel secondo, Reg_2W, la relativa parte immaginaria.
Il salvataggio, per entrambi le parti, avviene nello stesso colpo di clock. Per questo motivo sia il segnale di abilitazione, LE_W, sia il segnale di reset, RST_W, collegati alla C.U., sono comuni ad entrambi i registri.
Il dato in uscita, invece, è fornito attraverso un multiplexer a due vie il cui segnale di selezione, ADDR_OUTW, collegato alla C.U., permette di scegliere tra la parte reale o immaginaria del twiddle factor.

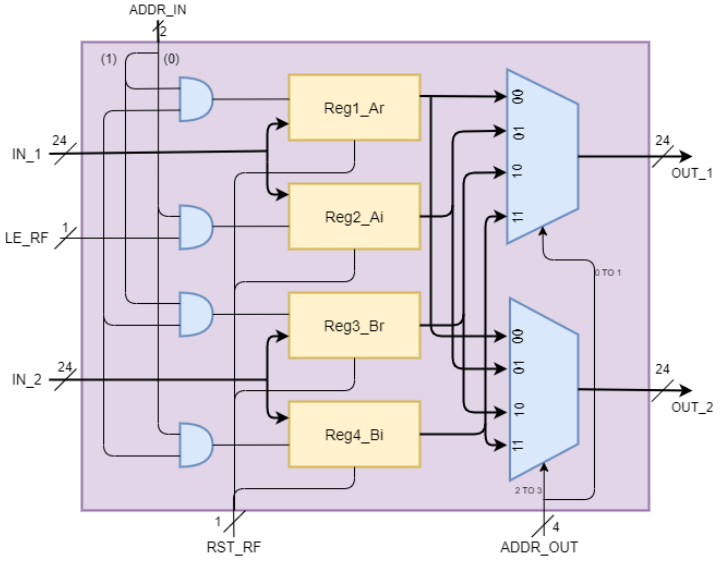


Figura 8: Schema register file campioni in ingresso

Il register file nel quale sono salvate le parti reali e immaginarie di A e B, figura 8, è caratterizzato da due porte sia in ingresso e sia in uscita e al suo interno sono presenti quattro registri.
La prima porta di ingresso, IN_1, è collegata a due registri, Reg1_Ar e Reg2_Ai, che contengono rispettivamente la parte reale e la parte immaginaria del dato A.
La seconda porta di ingresso, invece, lavora con i restanti due registri, Reg3_Br e Reg4_Bi, che contengono il dato B.
Poiché, in base al protocollo di interfacciamento con l'esterno, vengono fornite prima le parti reali dei due dati e al successivo colpo di clock le parti immaginarie, i registri vengono abilitati a coppie di due: ad un colpo di clock Reg1_Ar e Reg3_Br e al successivo colpo Reg2_Ai e Reg4_Bi.
Questo tipo di abilitazione avviene attraverso due segnali provenienti dalla C.U.: LE_RF e ADDR_IN. Il primo, con parallelismo 1 bit, è asserito ogni qual volta è necessario salvare i dati all'interno del register file; il secondo, invece, con parallelismo 2 bit, permette il salvataggio o nei registri della parte reale o in quelli della parte immaginaria.
Gli enable di ogni registro sono, infatti, collegati all'uscita di una porta AND che compie l'operazione logica tra il segnale LE_RF e uno dei due bit del segnale ADDR_IN. Nello specifico il bit 0 di quest'ultimo è collegato alle porte AND relative ai registri Reg2_Ai e Reg4_Bi mentre il bit 1 Reg1_Ar e Reg3_Br.
Con LE_RF asserito e ADDR_IN="01" vengono abilitati al salvataggio i registri che contengono le parti immaginarie; nel caso in cui, invece, ADDR_IN="10" sono abilitati i registri che contengono le parti reali.
I dati sono forniti all'esterno attraverso due multiplexer a 4 vie. Ogni registro è collegato ad entrambi i multiplexer: in questo modo i dati sono disponibili su entrambe le porte di uscita OUT_1 E OUT_2.
La selezione delle uscite avviene attraverso il segnale ADDR_OUT, caratterizzato da un parallelismo di 4 bit: i due bit meno significativi sono collegati al selettore del multiplexer relativo all'uscita OUT_2, mentre i due bit piu significativi al selettore del restante multiplexer.

## 3.4 Parallelismo

Per il calcolo del parallelismo interno sono stati studiati la dimensione dei dati in ingresso A e B e i valori possibili che il twiddle factor potesse assumere.
I primi, definiti in forma frazionaria in CA2 su 24 bit, presentano almeno due bit di guardia per preservare la precisione di calcolo e non avere overflow nei dati in uscita; il loro modulo, cioè, è minore di 0.25.
I twiddle factor, invece, non sono scalati e il loro valore è, perciò, compreso tra -1 e 1.
Dalla loro definizione (espressione 3) si può considerare come caso peggiore la condizione per la quale $W_r$=1 e $W_i$=0; si è considerato, inoltre, il caso peggiore degli ingressi che prevede $A_r$=$B_r$=$A_i$=$B_i$=0.25 .
In base a questi valori si sono calcolati i risultati di $A'_r$, $B'_r$, $A'_i$ e $B'_i$ controllando che questi fossero comunque esprimibile in accordo con la rappresentazione fixed point considerata, cioè il loro modulo fosse minore di 1.
Poiché dai calcoli effettuati questa condizione è sempre verificata, si è deciso di far lavorare il datapath con un parallelismo pari ai bit in uscita dalla moltiplicazione, cioè 47 bit.

In alcuni casi, come si può notare dal datapath (figura 4), è stato necessario una estensione del dato tramite l'inserimento di un numero opportuno di '0' a seguito del LSB.

## 3.5 Ottimizzazione bus

Il progetto del processore Butterfly si è soffermato ampiamente sull'ottimizzazione del numero di bus globali e di connessioni tra i componenti del datapath in modo da non penalizzare le prestazioni.
Nelle prime fasi si è ipotizzato di lavorare con risorse infinite: all'interno del register file principale, figura 8, venivano salvati, oltre ai dati A e B in ingresso, tutti i risultati delle operazioni aritmetiche svolte dal datapath. In quest'ottica, erano necessarie 5 porte in uscita, ognuna collegata agli ingressi dei blocchetti aritmetici (una per il moltiplicatore, due per il sommatore e due per il sottrattore) e 3 porte in ingresso per salvare i relativi risultati. Questa soluzione avrebbe implicato l'uso di un register file a 8 porte, complicato sia dal punto di vista di gestione sia di implementazione.
Osservando il Data Flow Diagram si è notato che i risultati delle operazioni aritmetiche sono immediatamente necessarie in successivi componenti; il salvataggio all'interno del register file diventa perciò inutile ed inefficiente.
Si è deciso, quindi, di lavorare il più possibile in locale riducendo i problemi connessi al trasporto dei dati: all'uscita di ogni blocchetto aritmetico è stato inserito un registro intermedio. Questa soluzione ha portato ad una notevole semplificazione del register file sia in termini di porte sia in termini di locazioni necessarie; grazie, infatti, alla gestione locale dei risultati delle operazioni e al loro collegamento diretto con i componenti, si sono potute eliminare 7 porte.
Il register file, così ottimizzato, presenta 4 registri per salvare solamente le parti reali e immaginarie di A e B e 4 porte, 2 in uscita e 2 in ingresso. Queste ultime sono necessarie per salvare a coppie le parti reali e al successivo colpo di clock le parti immaginarie dei dati.
Lo stessa linea progettuale è stata utilizzata per ottimizzare i bus in uscita dal register file del twiddle factor: dal DFD si è notato come le parti reali ed immaginarie del dato fossero coinvolte solamente in operazioni di moltiplicazione. Per questo motivo, il register file è caratterizzato da una sola porta in uscita, collegata direttamente ad un ingresso del moltiplicatore, e due porte in ingresso.
I componenti aritmetici, però, presentano ingressi sia fissi, come nel caso del sommatore dove il risultato della moltiplicazione tra due dati è sempre presente, sia variabili. Nel sottrattore, ad esempio, il sottraendo e il minuendo variano ad ogni operazione; per questo motivo sono stati inseriti dei multiplexer agli ingressi dei blocchi aritmetici, ove necessario, controllati dalla C.U. Essi in base all'operazione da svolgere selezionano il registro intermedio che contiene il dato necessario. L'ottimizzazione apportata al circuito non ha arrecato alcun danno alle prestazioni; lavorando, anzi, in locale, la riduzione dei bus globali sia in termini di numero che in termini di lunghezza ha permesso di diminuire l'area del register file, i ritardi ed eventuali costi di realizzazione.

# 4 Tempo di vita delle variabili

Per comprendere quanti registri utilizzare e cercare di ottimizzare il loro numero è stato svolto uno studio riguardo il tempo di vita delle variabili, prendendo in considerazione i risultati dei dodici operatori descritti in precedenza e dei valori reali e immaginari in ingresso necessari per i calcoli.

| Tempo di vita delle variabili | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| Ar | X | X | X | X | X | X | | | | | | |
| Ai | | X | X | X | X | X | X | | | | | |
| Br | X | X | | | | | | | | | | |
| Bi | | X | X | X | X | | | | | | | |
| Wr | X | X | X | X | X | | | | | | | |
| Wi | X | X | X | X | | | | | | | | |
| M1 | | | | X | X | | | | | | | |
| M2 | | | | | X | X | | | | | | |
| M3 | | | | | | X | X | | | | | |
| M4 | | | | | | | X | X | | | | |
| M5 | | | | | | | X | | | | | |
| M6 | | | | | | | | X | | | | |
| Σ1 | | | | | | X | | | | | | |
| Σ2 | | | | | | | X | X | X | X | X | |
| Σ3 | | | | | | | | X | | | | |
| Σ4 | | | | | | | | | X | X | X | X |
| Σ5 | | | | | | | | X | X | X | X | |
| Σ6 | | | | | | | | | | X | X | X |

Figura 9: Tempo di vita delle variabili

Come si può osservare dalla tabella il numero massimo di registri richiesto per un singolo colpo di clock è pari a 6; dunque da questo risultato si potrebbero istanziare solamente sei registri dove memorizzare tutti i dati. Nel datapath precedentemente descritto, all'interno sono presenti in uscita dal sommatore, dal sottrattore e dal moltiplicatore un totale di quattro registri. Ne sarebbero dunque necessari solamente altri due per permettere la memorizzazione dei campioni in ingresso e dei twiddle factor. In realtà è stata scelta una soluzione intermedia, che punta da un lato ad ottimizzare il numero di registri, e dall'altra di ottenere una velocità di esecuzione elevata, specialmente per quanto riguarda la modalità full speed.
Infatti, in ingresso, è stato posto un register file che contiene 4 locazioni dove memorizzare A e B, e un altro per salvare parte reale e immaginaria di W. Questa scelta permette di non dover durante l'esecuzione sovrascrivere Ar, Br, Ai e Bi, in quanto i valori calcolati passano da un blocco all'altro del datapath, intervallati solamente dai 4 registri interni.
Questa scelta si rivela efficiente se si lavora in modalità full speed, nella quale è possibile cambiare i valori del register file con nuovi campioni dopo 6 colpi di clock, senza andare ad alterare i calcoli già effettuati o i dati necessari per quelli ancora rimanenti.

# 5 Protocollo di interfacciamento con l'esterno

Il protocollo di l'interfacciamento con il mondo esterno è stato scelto per ottimizzare il numero di colpi di clock necessari per l'ingresso dei dati, evitando però di avere un parallelismo troppo elevato.
Per quanto riguarda l'ingresso, ci si aspetta che come primo segnale asserito lo START, condizione che permette alla macchina di partire. Successivamente, devono essere inviate tutte le parti reali dei campioni, e solamente al colpo di clock successivo le parti immaginarie. Questo protocollo risulta valido sia per quanto riguarda la singola butterfly che la FFT completa, siccome si tratta solamente di una concatenazione.
Per i dati in ingresso alla FFT è previsto che siano forniti con 5 bit di guardia per ognuno dei 16 campioni sui 24 bit complessivi per parte reale e immaginaria, in modo da evitare che all'interno durante i calcoli nei quattro stadi avvenga overflow compromettendo i risultati.
In uscita si ha la stessa sequenza: infatti, dopo che il segnale di DONE diventa attivo, si ricevono prima le parti reali del campioni calcolati e poi le parti immaginarie. Il fatto di aver scelto un protocollo di ingresso identico a quello di uscita permette di concatenare senza difficoltà le butterfly tra loro, e di poterle utilizzare per realizzare in seguito una Fast Fourier Transform di qualsiasi dimensione.
Anche se si lavora in modalità full speed il protocollo rimane il medesimo, ma è richiesto solamente di rispettare il timing per inviare i nuovi campioni nel momento opportuno.

# 6 Unità di controllo

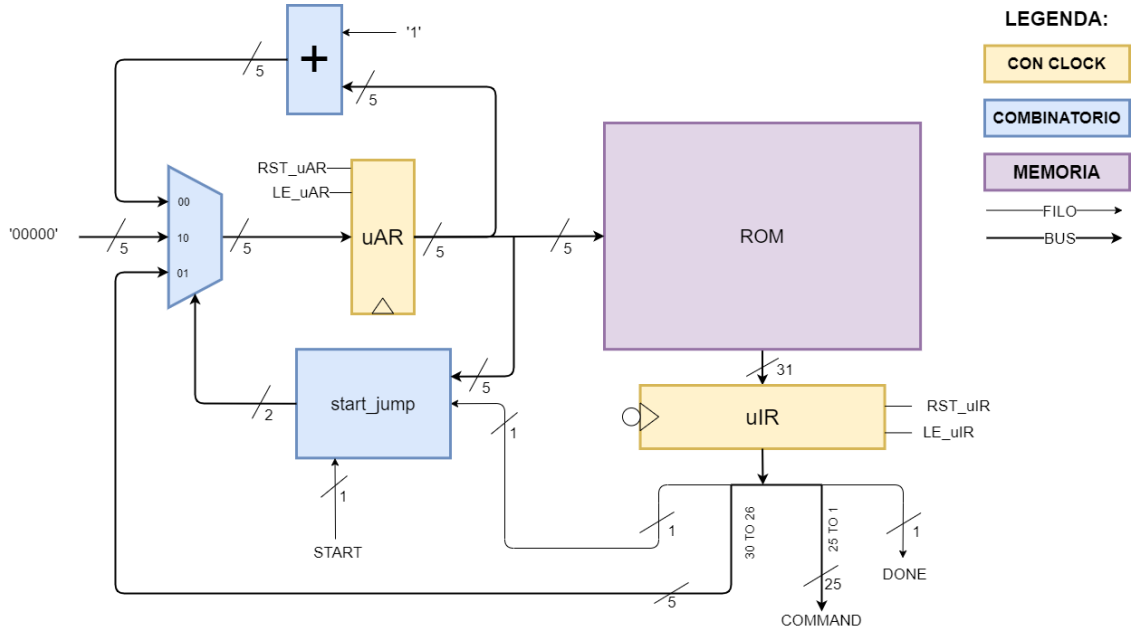


Figura 10: Control Unit Butterfly

L'unità di controllo della Butterfly è stata progettata secondo la tecnica della microprogrammazione.
Lo schema, figura 10, è caratterizzato da due registri: il $\mu AR$ (*Micro Address Register*), che contiene l'indirizzo, da fornire alla $\mu ROM$, dello stato corrente, e il $\mu IR$ (*Micro Instruction Register*) che contiene i comandi, in uscita della $\mu ROM$, da fornire al datapath.
La C.U. progettata presenta un totale di 19 stati (la $\mu ROM$ contiene, quindi, altrettante locazioni di memoria); per questo motivo il parallelismo del $\mu AR$ è pari a 5 bit.
Il $\mu IR$, invece, è suddiviso in diversi campi con un parallelismo totale di 32 bit: un bit, denominato condition code validation (cc), evidenzia la presenza di un eventuale salto (condizionato dal segnale di START), 5 bit rappresentano l'indirizzo dello stato sul quale saltare (jump address), 25 bit contengono i comandi da fornire al datapath e 1 bit indica il segnale di DONE.

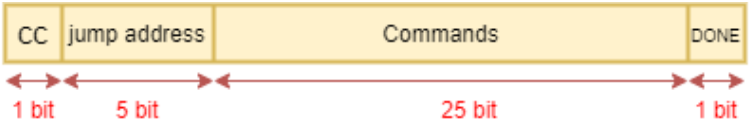


Figura 11: Campi presenti all'interno del $\mu IR$

Il sequenziatore, che ha il compito di inviare l'informazione dello stato presente al command generator, presenta un algoritmo di indirizzamento implicito. Questo meccanismo permette di generare il next state aggiungendo '1', mediante un sommatore, all'indirizzo del current state.
La modalità di funzionamento Full Speed del processore Butterfly, però, implica anche la presenza di salti condizionati a due vie (2-way branches) che devono essere gestiti dalla Control

Unit. Come si può notare, infatti, dall'ASM chart (figura 12), sono presenti due salti dipendenti dal segnale di START: nel primo, dopo lo stato M5_E2, la macchina deve decidere se svolgere una esecuzione isolata (ramo di sinistra, START='0') oppure continua (ramo di destra, START='1'). Nel secondo, invece, al termine dell'esecuzione continua la macchina deve decidere se continuare a lavorare nello stesso ramo o passare al ramo di esecuzione singola. La gestione dei salti è affidata ad un blocchetto denominato *start_jump* il cui segnale di uscita rappresenta il selettore di un multiplexer che discrimina il contenuto del $\mu AR$.
Il multiplexer è caratterizzato da 3 ingressi: l'uscita del sommatore +1, l'indirizzo del primo stato della $\mu ROM$ ("00000", IDLE), e l'indirizzo dello stato al quale puntare in caso di salto, contenuto in un campo specifico del $\mu IR$.
Il blocchetto start_jump riceve in ingresso il segnale di condition code validation (cc), il segnale di START e l'indirizzo contenuto nel $\mu AR$.
Quando la macchina è nello stato di IDLE e il segnale di START è asserito inizia la modalità di lavoro sequenziale e, perciò, il selettore fornisce la codifica "00" al multiplexer, connettendo il registro $\mu AR$ all'uscita del sommatore.
Nel caso in cui, invece, il bit cc è asserito, il blocchetto di *start_jump* discrimina tra due situazioni: se START è '1' deve essere effettuato un salto e, perciò, la codifica del selettore è "01"; se START è, invece, pari a '0' la macchina continua a lavorare in modo sequenziale (codifica selettore "00").
Infine, quando entrambi i bit cc e START sono pari a '0': se si è nello stato di IDLE, ovvero l'indirizzo contenuto nel $\mu AR$ è "00000", il selettore del multiplexer è codificato per mantenere la macchina nello stesso stato ("10"); se, al contrario, lo stato corrente è OUT_IM (indirizzo "01100"), cioè l'ultimo stato del ramo di esecuzione singola, la macchina torna in IDLE (selettore "10"). Se non si è in nessuno dei due stati, la C.U. continua a lavorare in modo sequenziale (selettore "00").

| START | cc | ADDRESS | RESET | SEL_MUX |
|---|---|---|---|---|
| X | X | X | 1 | 10 |
| 0 | 0 | 00000 | 0 | 10 |
| 0 | 0 | 01100 | 0 | 10 |
| 0 | 0 | X | 0 | 00 |
| 1 | 1 | X | 0 | 01 |
| 0 | 1 | X | 0 | 00 |

Tabella 2: Codifica del selettore del multiplexer in funzione degli ingressi

Il meccanismo di funzionamento della C.U., cioè il passaggio da next state a present state, deve essere completato all'interno di un ciclo così da poter fornire i comandi al datapath ad ogni colpo di clock.
Per questo motivo, si è deciso di far lavorare i due registri su fronti di clock diversi: il $\mu AR$ sul fronte di salita e il $\mu IR$ sul fronte di discesa.
Questa soluzione è, inoltre, compatibile con il protocollo di interfacciamento con l'esterno scelto che prevede la disponibilità dei dati in ingresso nel colpo di clock successivo al quale il segnale di START è asserito.

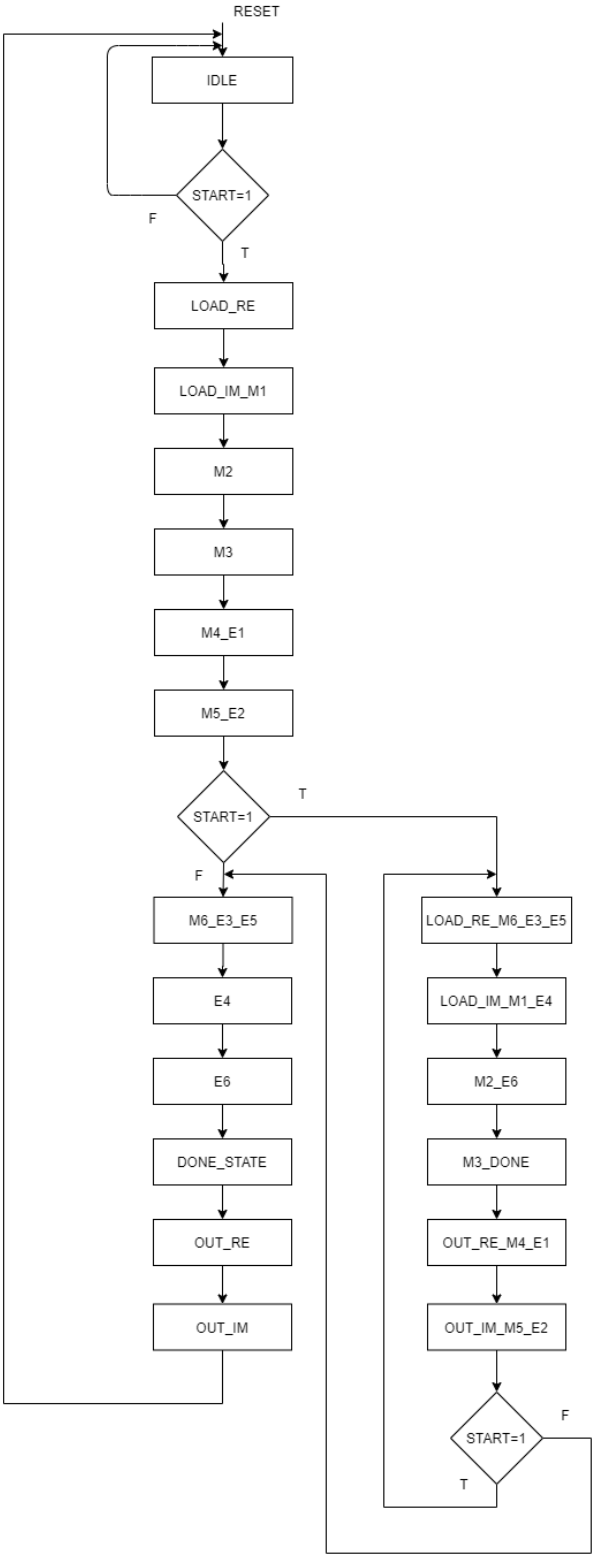


Figura 12: ASM chart Butterfly

# 7 FFT 16x16

La FFT 16x16 è stata realizzata, come si può vedere in figura 13, utilizzando quattro stadi da 8 butterfly ciascuno. Il primo stadio di Butterfly ha 16 ingressi chiamati $X$, ognuno dei quali è la concatenazione degli ingressi $A$ e $B$ della singola Butterfly. Perciò se il parallelismo in ingresso nella singola era di 24 bit sono stati inseriti in ingresso alla FFT 384 bit (24 bit x 16 ingressi). Per l'inserimento dei "twiddle factor" sono stati inseriti 16 registri: i primi 8 contengono la parte reale; i restanti 8 la parte immaginaria. Ogni registro ha un parallelismo di 24 bit cosicchè ogni Butterfly utilizzi in totale 48 bit tra parte reale e immaginaria.
Gli ulteriori stadi sono stati opportunamente connessi rispettando l'ordine dettato dai "twiddle factor" per un totale di 32 Butterfly istanziate.
Tutte le Butterfly hanno lo stesso segnale di Clock. Contestualmente il segnale di DONE di ogni Butterfly di ogni stadio è stato collegato in un unico DONE per un totale di 4, dove solo l'ultimo è l'effettivo DONE finale della FFT. Mentre il segnale di START dipende dal segnale di DONE della Butterfly dello stadio precedente tranne nel primo stadio in cui lo START è l'ingresso della FFT.
Infine in modo tale da rispettare il protocollo ingresso-uscita le uscite sono identiche agli ingressi.

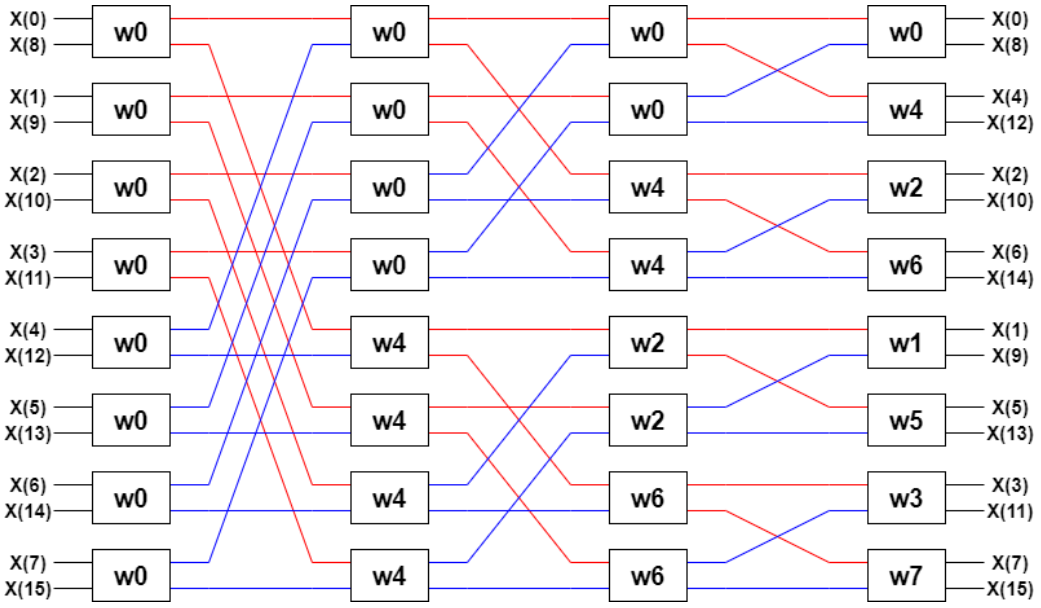


Figura 13: Grafico segnali FFT 16x16

# 8 Simulazioni

## 8.1 Butterfly

Per verificare il corretto funzionamento della Butterfly è stato utilizzato il software *Modelsim*. Utilizzando il testbench sono stati inseriti in ingresso $Ar = 000000000000010000011001$ (0.004 in decimale), $Br = 111111111111101011100001$ (-0.005) e $Wr = 011111111111111111111111$ (1) e $Wi = 000000000000000000000000$ (0) che entrano dopo 40 ns da quando è stato abilitato lo start. Dopo 40 ns vengono inviati gli immaginari $Ai = 000000000000000100000110$ (0.001) e $Bi = 000000000000011100101011$ (0.007). I twiddle factor Wr e Wi sono senza segno.
Come anticipato nella sezione relativa al CDFD la latenza è di 10 colpi di clock a partire dal campionamento dei reali sino alla loro uscita. In figura 14 si possono leggere, nella parte sinistra indicate dal cursore giallo, le uscite reali $A'r = 111111111111111011111010$ (-0.001) e $B'r = 000000000000100100110111$ (0.009); mentre in figura 15 si leggono le uscite immaginarie $A'i = 000000000000100000110001$ (0.008) e $B'i = 111111111111100111011011$ (-0.006).
i risultati coincidono con quelli teorici. Sono stati provati ulteriori casi con deversi ingressi ottenendo un risultato coerente con quello teorico.

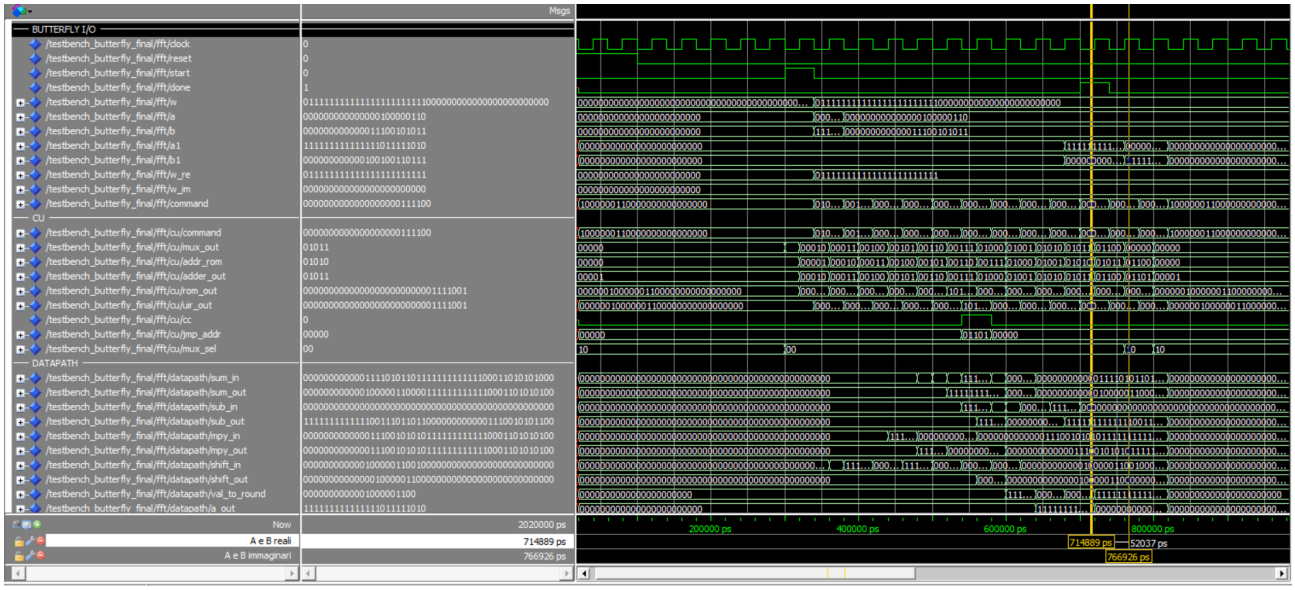


Figura 14: Butterfly: cursore che indica uscite reali

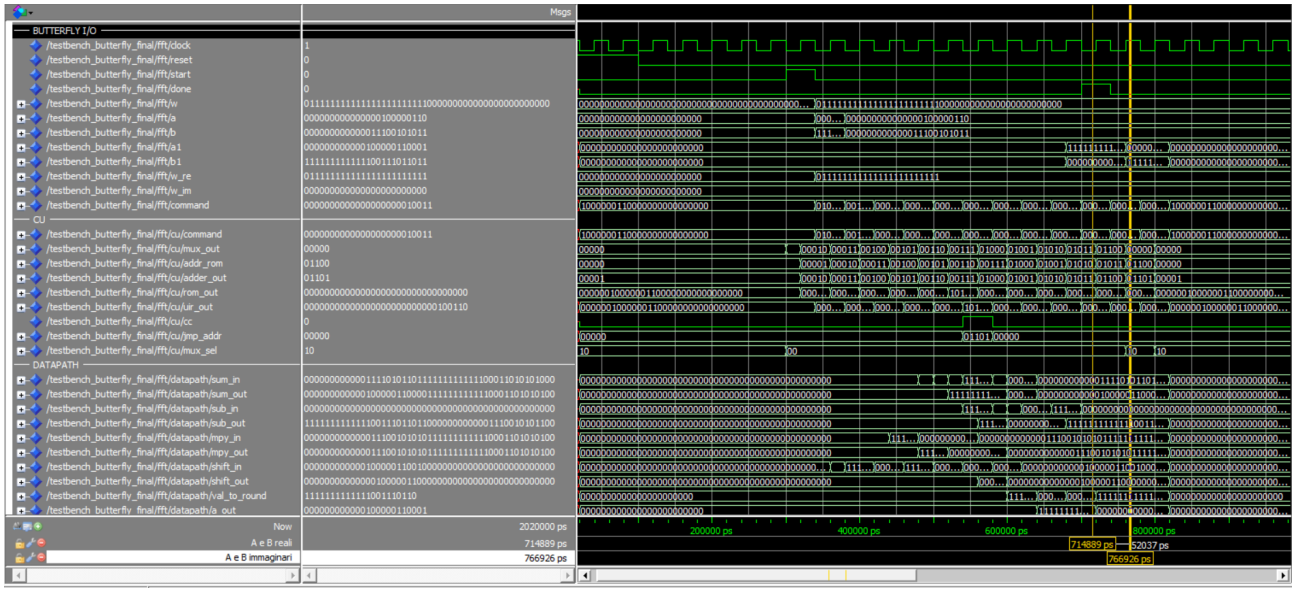


Figura 15: Butterfly: cursore che indica uscite immaginarie

## 8.2 Butterfly full-speed

Nella simulazione della Butterfly in modalità full speed sono stati inviati consecutivamente due dati in ingresso si può notare, dalle seguenti immagini 16 e 17, che la latenza (dal primo dato reale campionato alla sua uscita reale) rimane sempre di 10 colpi di clock ma il throughput passa da 10 a 6 colpi di clock tra l'uscita di un dato e la successiva uscita del secondo dato.

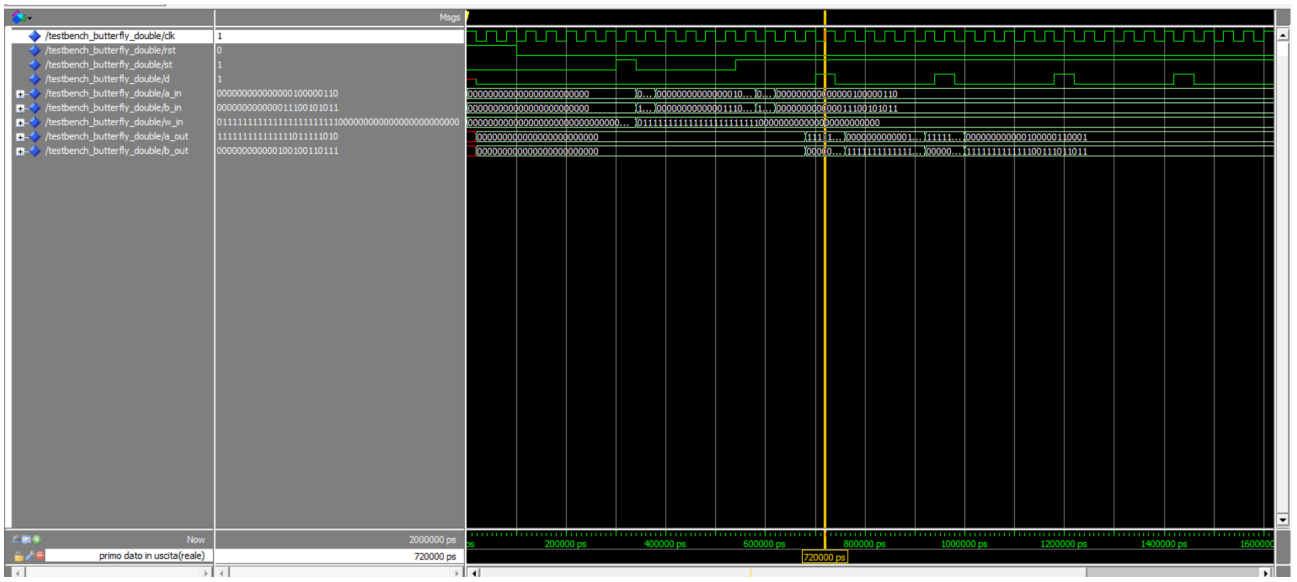


Figura 16: Butterfly full speed: cursore che indica primo dato reale in uscita

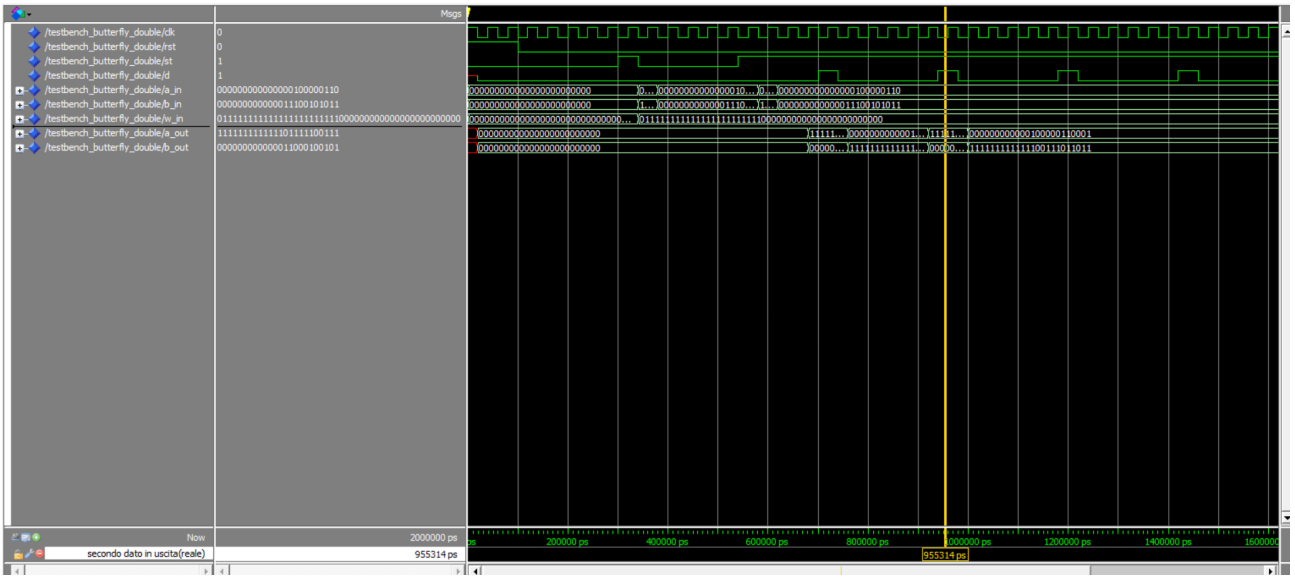


Figura 17: Butterfly full speed: cursore che indica secondo dato reale in uscita

## 8.3 FFT 16x16

Per verificare la FFT è stato utilizzato il file *FFT_16x16_Examples.pdf* dove si è constatato che ogni ingresso coincide con l'uscita perciò la FFT funziona perfettamente.
La latenza dal primo dato campionato alla sua uscita è di 40 colpi di clock poichè la FFT è composta, come detto in precedenza, da 4 stadi di butterfly con latenza di 10 colpi di clock.
Di seguito nella figura 18 si leggono gli ingressi reali (gli immaginari valgono 0) di uno dei dodici esempi in particolare X = [-1 -1 1 1 -1 -1 1 1 -1 -1 1 1 -1 -1 1 1].

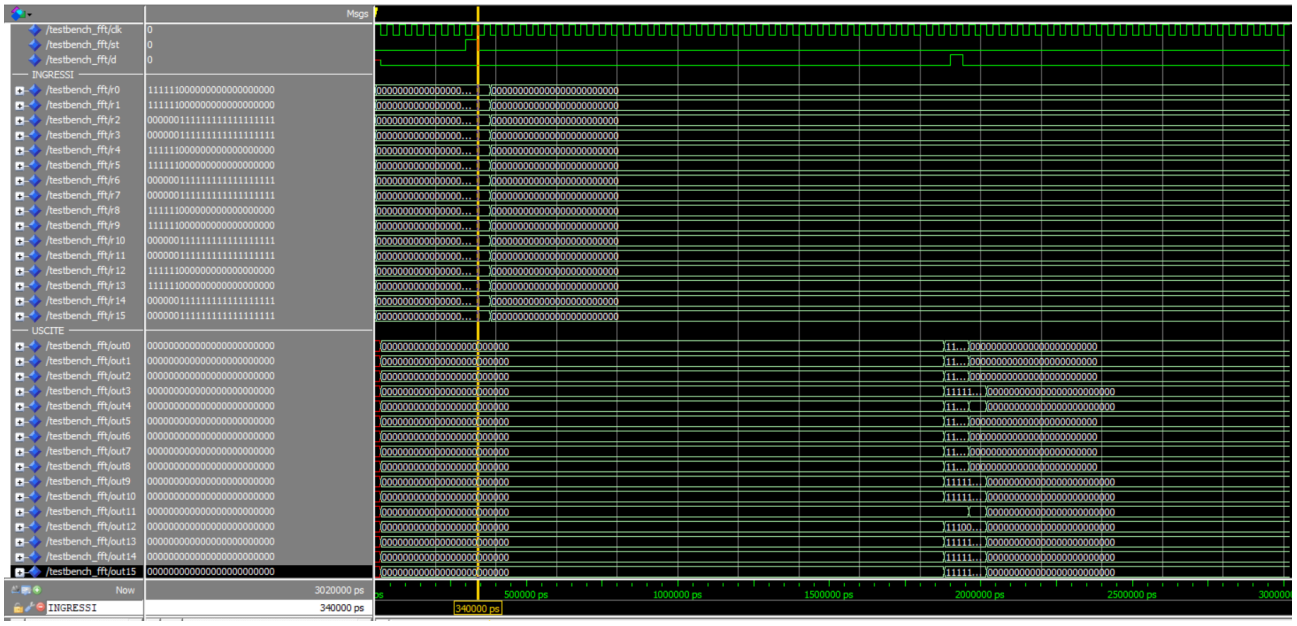


Figura 18: FFT: cursore che indica ingressi reali

Le uscite reali sono rispettate come si evince dalla figura 19 con un -8 in quinta posizione e in tredicesima posizione.

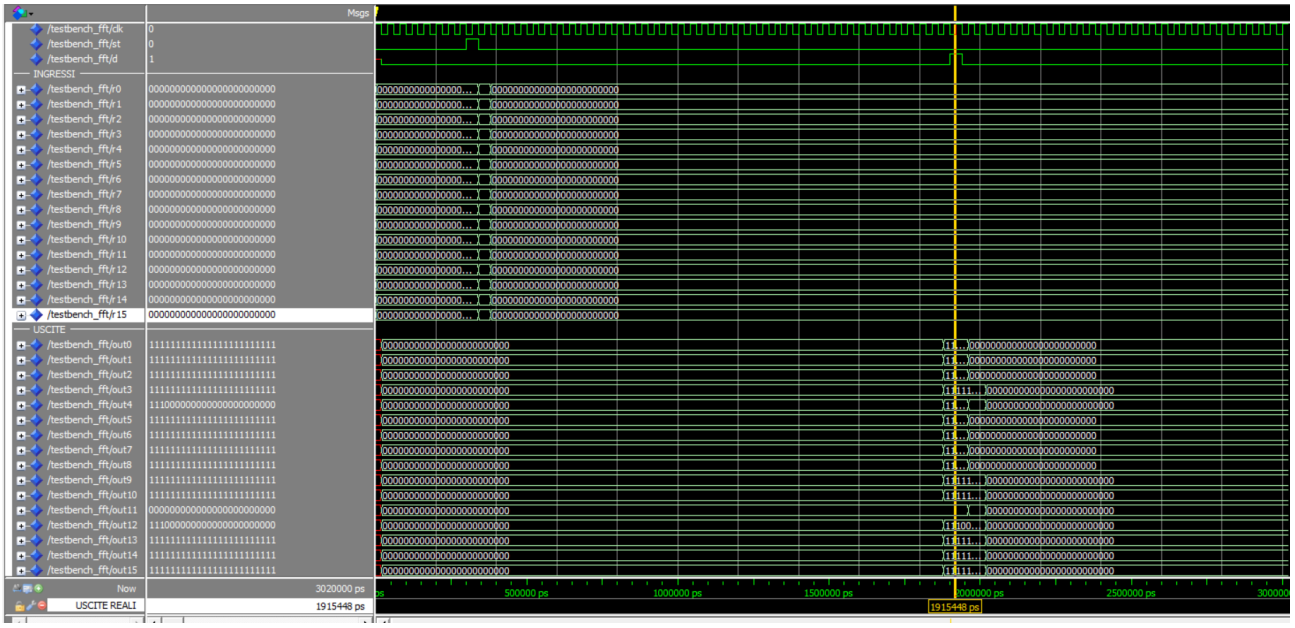


Figura 19: FFT: cursore che indica uscite reali

In figura 20 si leggono i risultati corretti della parte immaginaria: un 8 in quinta posizione e un -8 in tredicesima posizione.

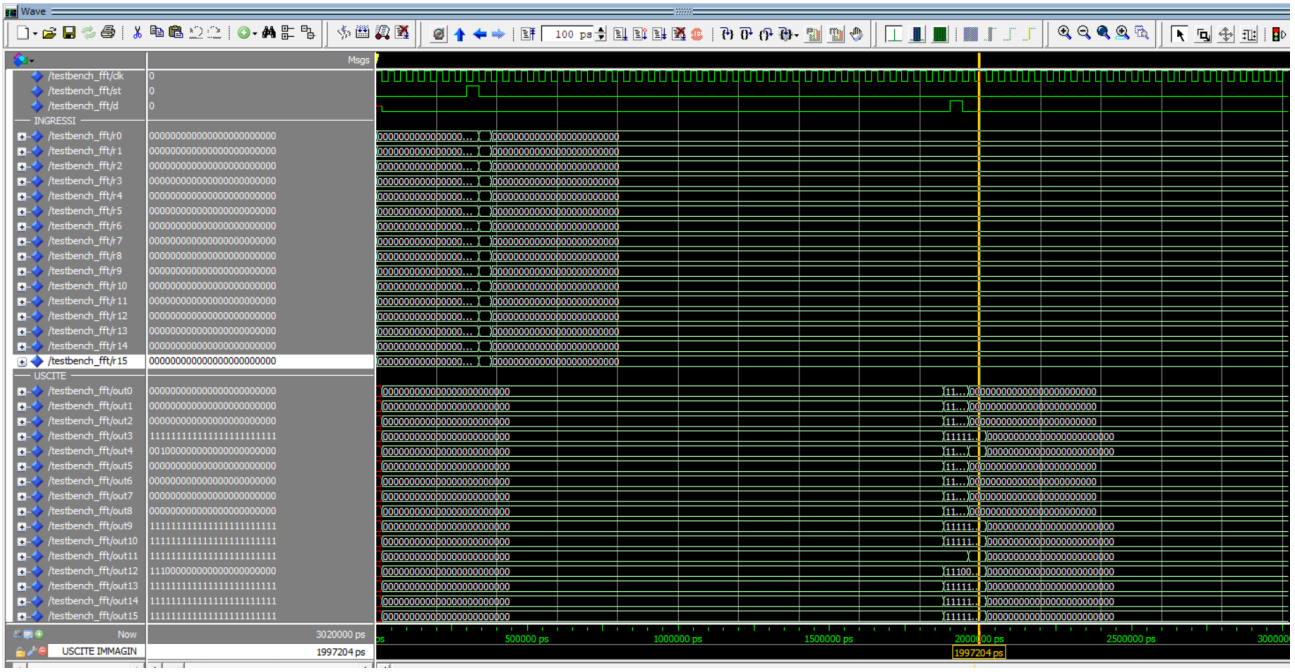


Figura 20: FFT: cursore che indica uscite immaginarie

# 9 Appendice

In questa sezione vengono riportati i codici VHDL.

## 9.1 Moltiplicatore

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY multiplier IS
GENERIC(N : integer:=8);
PORT(A,B: IN SIGNED(N-1 DOWNTO 0);
     clock, reset, MPY_SHn: IN STD_LOGIC;
     p: OUT SIGNED(2*N-1 DOWNTO 0);
     shift_out: OUT SIGNED(N DOWNTO 0));
END multiplier;

ARCHITECTURE Structure OF multiplier IS

COMPONENT rise_register
  GENERIC (N : INTEGER:=10);
    PORT ( clock : IN STD_LOGIC;
          data_in : IN SIGNED(N-1 DOWNTO 0);
          clear, reset, LE: IN STD_LOGIC;
          data_out : OUT SIGNED(N-1 DOWNTO 0));
END COMPONENT;

SIGNAL outreg1,outproduct: SIGNED (2*N-1 DOWNTO 0);

BEGIN
reg1: rise_register GENERIC MAP (N=>2*N)
              PORT MAP (clock=>clock, data_in=>outproduct, clear=>'0', reset=>reset, LE=>MPY_SHn, data_out=>
    outreg1);

reg2: rise_register GENERIC MAP (N=>2*N)
              PORT MAP (clock=>clock, data_in=>outreg1, clear=>'0', reset=>reset, LE=>'1', data_out=>p);

shift_out<= A(N-2 downto 0) & "00";
outproduct<=A*B;

END Structure;
```

## 9.2 Register File

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity reg_file is
  generic (
    N     : positive := 24        --Numero bit riga
   );
  port(
    outA          : out SIGNED(N-1 downto 0);
    outB          : out SIGNED(N-1 downto 0);
    inputA        : in  SIGNED(N-1 downto 0);
   inputB        : in  SIGNED(N-1 downto 0);
   inSel       : in  STD_LOGIC_VECTOR(1 downto 0);
    outSel       : in  STD_LOGIC_VECTOR(3 downto 0);
   load_en      : in STD_LOGIC;
   reset         : in  std_logic;
   Clk           : in std_logic
    );
end reg_file;


architecture behavioral of reg_file is

COMPONENT rise_register
  GENERIC (N : INTEGER:=10);
    PORT ( clock : IN STD_LOGIC;
          data_in : IN SIGNED(N-1 DOWNTO 0);
          clear, reset, LE: IN STD_LOGIC;  --Enable = load enable
          data_out : OUT SIGNED(N-1 DOWNTO 0));
END COMPONENT;

COMPONENT mux4to1 IS
GENERIC(N : positive :=24);
PORT (w0, w1, w2, w3 : IN SIGNED(N-1 DOWNTO 0);
              s : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
              f : OUT SIGNED(N-1 DOWNTO 0));
```

```
38 END COMPONENT;
39
40   SIGNAL en1, en2, en3, en4: STD_LOGIC;
41   SIGNAL out1, out2, out3, out4: SIGNED(N-1 DOWNTO 0);
42
43 BEGIN
44
45 Reg_1: rise_register    generic map (N=>N)
46                       port map (data_in=>inputA, clock=>Clk, reset=>reset, LE=>en1, data_out=>out1, clear=>'0');
47 Reg_2: rise_register    generic map (N=>N)
48                       port map (data_in=>inputA, clock=>Clk, reset=>reset, LE=>en2, data_out=>out2, clear=>'0');
49 Reg_3: rise_register    generic map (N=>N)
50                       port map (data_in=>inputB, clock=>Clk, reset=>reset, LE=>en3, data_out=>out3, clear=>'0');
51 Reg_4: rise_register    generic map (N=>N)
52                       port map (data_in=>inputB, clock=>Clk, reset=>reset, LE=>en4, data_out=>out4, clear=>'0');
53
54 mux1: mux4to1 generic map (N=>N)
55                  port map (w0=>out1, w1=>out2, w2=>out3, w3=>out4, s=>outSel(3 DOWNTO 2), f=>outA);
56 mux2: mux4to1 generic map (N=>N)
57                  port map (w0=>out1, w1=>out2, w2=>out3, w3=>out4, s=>outSel(1 DOWNTO 0), f=>outB);
58
59 en1<=load_en AND inSel(1);
60 en2<=load_en AND inSel(0);
61 en3<=load_en AND inSel(1);
62 en4<=load_en AND inSel(0);
63
64
65 end behavioral;
```

## 9.3 Register File "twiddle factor"

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  USE ieee.numeric_std.all;
4
5  ENTITY Reg_file_W IS
6  GENERIC (
7      N      : POSITIVE := 24 --num bit riga
8     );
9  PORT        (ADDR_OUT              : IN STD_LOGIC;
10          INPUT1        :IN SIGNED(N-1 DOWNTO 0);
11          INPUT2        :IN SIGNED(N-1 DOWNTO 0);
12          CLOCK          : IN STD_LOGIC;
13          RESET          : IN STD_LOGIC;
14               LOAD_EN           : IN STD_LOGIC;
15          REG_OUT          : OUT SIGNED(N-1 DOWNTO 0));
16 END Reg_file_W;
17
18 ARCHITECTURE Behavior OF Reg_file_W IS
19
20   COMPONENT mux2to1 IS
21   GENERIC (N : INTEGER:=10);
22     PORT( input1 : IN SIGNED (N-1 DOWNTO 0);
23           input2 : IN SIGNED (N-1 DOWNTO 0);
24           sel   : IN STD_LOGIC;
25           output : OUT SIGNED (N-1 DOWNTO 0));
26   END COMPONENT;
27
28   COMPONENT rise_register
29   GENERIC (N : INTEGER:=10);
30     PORT ( clock : IN STD_LOGIC;
31           data_in : IN SIGNED(N-1 DOWNTO 0);
32           clear, reset, LE: IN STD_LOGIC;  --Enable = load enable
33           data_out : OUT SIGNED(N-1 DOWNTO 0));
34   END COMPONENT;
35
36   --SIGNAL out_reg1,out_reg2: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
37   SIGNAL uout1, uout2: SIGNED(N-1 DOWNTO 0);
38
39 BEGIN
40
41 Reg_1W: rise_register    generic map (N=>N)
42                       port map (data_in=>INPUT1, clock=>CLOCK, reset=>RESET, LE=>LOAD_EN, data_out=>uout1, clear
       =>'0');
43 Reg_2W: rise_register    generic map (N=>N)
44                       port map (data_in=>INPUT2, clock=>CLOCK, reset=>RESET, LE=>LOAD_EN, data_out=>uout2, clear
       =>'0');
45
46 mux: mux2to1 generic map(N=>N)
47                    port map(input1=>uout1,input2=>uout2,sel=>ADDR_OUT,output=>REG_OUT);
48
49 END Behavior;
```

## 9.4 ROM rounding

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY ROM_rounding IS
GENERIC (
    N      : POSITIVE := 2 --num bit riga
   );
PORT        (ADDR              : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
         CLOCK          : IN STD_LOGIC;
              EN             : IN STD_LOGIC;
         ROM_OUT          : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
END ROM_rounding;

ARCHITECTURE Behavior OF ROM_rounding IS



TYPE Rom_type IS ARRAY (0 TO 7) OF STD_LOGIC_VECTOR(N-1 DOWNTO 0);
constant rom8: Rom_type:=(      "00", --000           --8 elementi della rom da 2 bit ciascuno
                        "01", --001
                        "01", --010
                        "10", --011
                        "10", --100
                        "11", --101
                        "11", --110
                        "11" --111
                    );

BEGIN
PROCESS(CLOCK)
  BEGIN
  IF(CLOCK'EVENT AND CLOCK='1')THEN
    IF(EN='1')THEN
     ROM_OUT<=rom8(to_integer(unsigned(ADDR)));
    END IF;
  END IF;
END PROCESS;

END Behavior;
```

## 9.5 Start jump

```
library IEEE;
use IEEE.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY start_jump IS
  PORT( reset: IN STD_LOGIC;
       addr: IN STD_LOGIC_VECTOR(4 DOWNTO 0);
       cc: IN STD_LOGIC;
       start: IN STD_LOGIC;
       sel_addr: OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
       );
END start_jump;


ARCHITECTURE Behavior OF start_jump IS
  SIGNAL sel: STD_LOGIC_VECTOR(1 DOWNTO 0);
   SIGNAL idlestate: STD_LOGIC;

  BEGIN

  idlestate<=NOT(addr(4)) AND NOT(addr(3)) AND NOT(addr(2)) AND NOT(addr(1)) AND NOT(addr(0));

  PROCESS(reset,start,cc,addr,idlestate)
    BEGIN
    IF (reset='1') THEN
      sel<="10";
    ELSE
      IF(start='1' AND cc='0')THEN --inizio
       sel<="00";
      ELSIF(start='0' AND cc='0') THEN --run
         IF(idlestate='0')THEN
          IF(addr="01100")THEN
           sel<="10";
          ELSE
            sel<="00";
          END IF;
        ELSE
            sel<="10";
        END IF;
      ELSIF(start='1' AND cc='1') THEN --jump
       sel<="01";
      ELSIF(start='0' AND cc='1') THEN --no jump
       sel<="00";
      ELSE
```

```
45          sel<="10";
46        END IF;
47      END IF;
48    END PROCESS;
49
50 sel_addr<=sel;
51
52 END Behavior;
```

## 9.6 Datapath Butterfly

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  USE ieee.numeric_std.all;
4
5  ENTITY butterfly_dp IS
6
7    PORT( CLOCK : IN STD_LOGIC;
8         IN_A, IN_B: IN SIGNED (23 DOWNTO 0); --REGISTER FILE AB
9         ADDR_IN: IN STD_LOGIC_VECTOR (1 DOWNTO 0); --REGISTER FILE AB  --addr reg1 e reg2 + addr reg3 e reg4
10        ADDR_OUT : IN STD_LOGIC_VECTOR(3 DOWNTO 0); --REGISTER FILE AB --addr uscita1 + addr uscita2
11        RST_RF, LE_RF   : IN STD_LOGIC; --REGISTER FILE AB
12        Wi, Wr : IN SIGNED (23 DOWNTO 0); --REGISTER FILE W
13        LE_W, RST_W : IN STD_LOGIC; --REGISTER FILE W
14        ADDR_OUTW : IN STD_LOGIC; --REGISTER FILE W
15        MPY_SHn, RST_MULT: IN STD_LOGIC; --MOLTIPLICATORE
16        RST_SH, LE_SH : IN STD_LOGIC; --REGISTRO USCITA SHIFT
17        RST_MPY, LE_MPY: IN STD_LOGIC;--REGISTRO USCITA MOLTIPLICATORE
18        MUX_SUM: IN STD_LOGIC; --MUX INGRESSO SOMMATORE
19        MUX_SUB1: IN STD_LOGIC; --MUX INGRESSO1 SOTTRATTORE
20        MUX_SUB2: IN STD_LOGIC_VECTOR(1 DOWNTO 0); --MUX INGRESSO2 SOTTRATTORE
21        RST_SUB, LE_SUB: IN STD_LOGIC; --REGISTRO USCITA SOTTRATTORE
22        RST_SUM, LE_SUM: IN STD_LOGIC; --REGISTRO USCITA SOMMATORE
23        MUX_ROUND, EN_ROUND: IN STD_LOGIC; --ROM ROUNDING + MUX INGRESSO
24        RST_RNDREG, LE_RNDREG: IN STD_LOGIC; --REG PER ROUNDING
25        RST_A,LE_A: IN STD_LOGIC;
26        RST_A1,LE_A1: IN STD_LOGIC;
27        RST_B,LE_B: IN STD_LOGIC;
28        OUT_A, OUT_B : OUT SIGNED (23 DOWNTO 0));
29 END butterfly_dp;
30
31 ARCHITECTURE Behaviour OF butterfly_dp IS
32
33 --componenti
34 COMPONENT multiplier
35 GENERIC(N : integer:=8);
36 PORT(A,B: IN SIGNED(N-1 DOWNTO 0);
37      clock, reset, MPY_SHn: IN STD_LOGIC;
38      p: OUT SIGNED(2*N-1 DOWNTO 0);
39      shift_out: OUT SIGNED(N DOWNTO 0));
40 END COMPONENT;
41
42 COMPONENT adder
43 GENERIC(N : integer:=8);
44 PORT(a,b: IN SIGNED(N-1 DOWNTO 0);
45      s: OUT SIGNED(N-1 DOWNTO 0)
46      );
47 END COMPONENT;
48
49 COMPONENT rise_register
50   GENERIC (N : INTEGER:=10);
51     PORT ( clock : IN STD_LOGIC;
52           data_in : IN SIGNED(N-1 DOWNTO 0);
53           clear, reset, LE: IN STD_LOGIC;  --Enable = load enable
54           data_out : OUT SIGNED(N-1 DOWNTO 0));
55 END COMPONENT;
56
57 COMPONENT mux2to1
58   GENERIC (N : INTEGER:=10);
59     PORT( input1 : IN SIGNED (N-1 DOWNTO 0);
60          input2 : IN SIGNED (N-1 DOWNTO 0);
61          sel   : IN STD_LOGIC;
62          output : OUT SIGNED (N-1 DOWNTO 0));
63 END COMPONENT;
64
65 COMPONENT mux3to1
66 GENERIC (N : INTEGER:=10);
67   PORT( input1 : IN SIGNED (N-1 DOWNTO 0);
68        input2 : IN SIGNED (N-1 DOWNTO 0);
69        input3 : IN SIGNED (N-1 DOWNTO 0);
70        sel   : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
71        output : OUT SIGNED (N-1 DOWNTO 0));
72 END COMPONENT;
73
74 COMPONENT subtractor
75 GENERIC(N : integer:=8);
76 PORT(a,b: IN SIGNED(N-1 DOWNTO 0);
```

```
77      s: OUT SIGNED(N-1 DOWNTO 0));
78  END COMPONENT;
79
80  COMPONENT ROM_rounding
81  GENERIC ( N       : POSITIVE := 2);
82  PORT        (ADDR              : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
83           CLOCK          : IN STD_LOGIC;
84                EN              : IN STD_LOGIC;
85           ROM_OUT           : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
86  END COMPONENT;
87
88  COMPONENT reg_file IS
89    GENERIC (
90      N       : POSITIVE := 24         --Numero bit riga
91     );
92    PORT(
93      outA           : OUT SIGNED(N-1 DOWNTO 0);
94      outB           : OUT SIGNED(N-1 DOWNTO 0);
95      inputA         : IN  SIGNED(N-1 DOWNTO 0);
96     inputB          : IN  SIGNED(N-1 DOWNTO 0);
97     inSel           : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
98      outSel         : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
99     load_en         : IN STD_LOGIC;
100    reset           : IN  STD_LOGIC;
101    Clk             : IN STD_LOGIC
102     );
103 END COMPONENT;
104
105 COMPONENT Reg_file_W IS
106   GENERIC (
107     N       : POSITIVE := 24 --num bit riga
108     );
109   PORT     (ADDR_OUT          : IN STD_LOGIC;
110          INPUT1        :IN SIGNED(N-1 DOWNTO 0);
111          INPUT2        :IN SIGNED(N-1 DOWNTO 0);
112          CLOCK         : IN STD_LOGIC;
113          RESET         : IN STD_LOGIC;
114               LOAD_EN       : IN STD_LOGIC;
115          REG_OUT           : OUT SIGNED(N-1 DOWNTO 0));
116 END COMPONENT;
117
118 SIGNAL out_w, out_1, out_2: SIGNED(23 DOWNTO 0);
119 SIGNAL mpy_in,shift_in: SIGNED(48 DOWNTO 0);
120 SIGNAL p_out: SIGNED(47 DOWNTO 0);
121 SIGNAL s_out: SIGNED(24 DOWNTO 0);
122 SIGNAL shift_out, mpy_out: SIGNED(48 DOWNTO 0);
123 SIGNAL mux_sum_out, mux_sum_in1: SIGNED(48 DOWNTO 0);
124 SIGNAL sum_in, sum_out: SIGNED(48 DOWNTO 0);
125 SIGNAL mux_sub1_out, mux_sub2_out: SIGNED(48 DOWNTO 0);
126 SIGNAL sub_in, sub_out: SIGNED(48 DOWNTO 0);
127 SIGNAL mux_rnd_out: SIGNED(48 DOWNTO 0);
128 SIGNAL mux_rnd_out_std: STD_LOGIC_VECTOR(48 DOWNTO 0);
129 SIGNAL rnd_out: STD_LOGIC_VECTOR(1 DOWNTO 0);
130 SIGNAL val_rounded: STD_LOGIC_VECTOR(23 DOWNTO 0);
131 SIGNAL val_to_round: SIGNED(21 DOWNTO 0);
132 SIGNAL reg_out_in: SIGNED(23 DOWNTO 0);
133 SIGNAL a_out: SIGNED(23 DOWNTO 0);
134
135   BEGIN
136
137   --register file Ar Ai Br Bi
138   register_file: reg_file GENERIC MAP(N=>24)
139     PORT MAP(outA=>out_1 , outB=>out_2 , inputA=>IN_A , inputB=>IN_B , inSel=>ADDR_IN , outSel=>ADDR_OUT ,
       load_en=>LE_RF, reset=>RST_RF, Clk=>CLOCK);
140
141   --register file Wr Wi
142   register_file_W: Reg_file_W GENERIC MAP(N=>24)
143     PORT MAP(ADDR_OUT=>ADDR_OUTW , INPUT1=>Wr , INPUT2=>Wi , CLOCK=>CLOCK , RESET=>RST_W , LOAD_EN=>LE_W ,
       REG_OUT=>out_w);
144
145   --moltiplicatore
146   multip: multiplier GENERIC MAP (N=>24)
147     PORT MAP (A=>out_1, B=>out_w, clock=>CLOCK, reset=>RST_MULT, MPY_SHn=>MPY_SHn, p=>p_out, shift_out=>s_out)
       ;
148
149   mpy_in<= p_out(46 DOWNTO 0) & "00";
150   shift_in<=s_out & "00000000000000000000000";
151
152   shift_reg: rise_register GENERIC MAP (N=>49)
153     PORT MAP (clock=>CLOCK, data_in=>shift_in, clear=> '0', reset=>RST_SH, LE=>LE_SH, data_out=>shift_out);
154
155   mpy_reg: rise_register GENERIC MAP (N=>49)
156     PORT MAP (clock=>CLOCK, data_in=>mpy_in, clear=> '0', reset=>RST_MPY, LE=>LE_MPY, data_out=>mpy_out);
157
158   --sommatore
159   mux_sum_in1<=out_2 & "000000000000000000000000"; --estensione di segno
160
161   mux_add: mux2to1 GENERIC MAP (N=>49)
162     PORT MAP (input1=>sum_out, input2=>mux_sum_in1, sel=>MUX_SUM, output=>mux_sum_out);
```

```
163
164   add: adder GENERIC MAP (N=>49)
165     PORT MAP (a=>mpy_out, b=>mux_sum_out, s=>sum_in);
166
167   sum_reg: rise_register GENERIC MAP (N=>49)
168     PORT MAP (clock=>CLOCK, data_in=>sum_in, clear=> '0', reset=>RST_SUM, LE=>LE_SUM, data_out=>sum_out);
169
170   --sottrattore
171   muxA_sub: mux2to1 GENERIC MAP (N=>49)
172     PORT MAP (input1=>sum_out, input2=>shift_out, sel=>MUX_SUB1, output=>mux_sub1_out);
173
174   muxB_sub: mux3to1 GENERIC MAP (N=>49)
175     PORT MAP (input1=>sum_out, input2=>mpy_out, input3=>sub_out, sel=>MUX_SUB2, output=>mux_sub2_out );
176
177   sub: subtractor GENERIC MAP (N=>49)
178     PORT MAP (a=>mux_sub1_out, b=>mux_sub2_out, s=>sub_in);
179
180   sub_reg: rise_register GENERIC MAP (N=>49)
181     PORT MAP (clock=>CLOCK, data_in=>sub_in, clear=> '0', reset=>RST_SUB, LE=>LE_SUB, data_out=>sub_out);
182
183   --ROM rounding
184   mux_rom:  mux2to1 GENERIC MAP (N=>49)
185     PORT MAP (input1=>sum_out, input2=>sub_out, sel=>MUX_ROUND, output=>mux_rnd_out);
186
187   mux_rnd_out_std<=STD_LOGIC_VECTOR(mux_rnd_out);
188
189   rnd_reg: rise_register GENERIC MAP (N=>22)
190     PORT MAP (clock=>CLOCK, data_in=>SIGNED(mux_rnd_out_std(48 DOWNTO 27)), clear=> '0', reset=>RST_RNDREG, LE
       =>LE_RNDREG, data_out=>val_to_round);
191
192   rom_round: ROM_rounding GENERIC MAP (N=>2)
193     PORT MAP (ADDR=>mux_rnd_out_std(26 DOWNTO 24), CLOCK=>CLOCK, EN=>EN_ROUND, ROMOUT=>rnd_out);
194
195   val_rounded<=STD_LOGIC_VECTOR(val_to_round) & rnd_out;
196   reg_out_in<=SIGNED(val_rounded);
197
198   --uscite
199   A_reg: rise_register GENERIC MAP (N=>24)
200     PORT MAP (clock=>CLOCK, data_in=>reg_out_in, clear=> '0', reset=>RST_A, LE=>LE_A, data_out=>a_out);
201
202   B_reg: rise_register GENERIC MAP (N=>24)
203     PORT MAP (clock=>CLOCK, data_in=>reg_out_in, clear=> '0', reset=>RST_B, LE=>LE_B, data_out=>OUT_B);
204
205   A1_reg: rise_register GENERIC MAP (N=>24)
206     PORT MAP (clock=>CLOCK, data_in=>a_out, clear=> '0', reset=>RST_A1, LE=>LE_A1, data_out=>OUT_A);
207
208 END Behaviour;
```

## 9.7 CU Butterfly

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  USE ieee.numeric_std.all;
4
5  ENTITY CU_Butterfly IS
6  PORT        (START              : IN STD_LOGIC;
7           RESET          : IN STD_LOGIC;
8           DONE           : OUT STD_LOGIC;
9           CLOCK          :IN STD_LOGIC;
10          COMMAND           : OUT STD_LOGIC_VECTOR(24 DOWNTO 0));
11 END CU_Butterfly;
12
13 ARCHITECTURE Behavior OF CU_Butterfly IS
14
15   COMPONENT reg IS
16     GENERIC (N : INTEGER:=10);
17       PORT ( clock : IN STD_LOGIC;
18              data_in : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
19              clear, reset, LE: IN STD_LOGIC;  --Enable = load enable
20              data_out : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
21   END COMPONENT;
22
23   COMPONENT fall_register IS
24    GENERIC (N : INTEGER:=10);
25     PORT ( clock : IN STD_LOGIC;
26           data_in : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
27           clear, reset, LE: IN STD_LOGIC;  --Enable = load enable
28           data_out : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
29    END COMPONENT;
30
31   COMPONENT mux3to1_std IS
32     GENERIC (N : INTEGER:=10);
33       PORT( input1 : IN STD_LOGIC_VECTOR (N-1 DOWNTO 0);
34             input2 : IN STD_LOGIC_VECTOR (N-1 DOWNTO 0);
35             input3 : IN STD_LOGIC_VECTOR (N-1 DOWNTO 0);
36             sel   : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
37             output : OUT STD_LOGIC_VECTOR (N-1 DOWNTO 0));
```

```
38   END COMPONENT;
39
40   COMPONENT Rom_async IS
41     PORT        (ADDR             : IN STD_LOGIC_VECTOR(4 DOWNTO 0);
42              OUTPUT           : OUT STD_LOGIC_VECTOR(31 DOWNTO 0));
43   END COMPONENT;
44
45   COMPONENT adder_u IS
46   GENERIC(N : integer:=8);
47     PORT(a,b: IN UNSIGNED(N-1 DOWNTO 0);
48          s: OUT UNSIGNED(N-1 DOWNTO 0)
49          );
50   END COMPONENT;
51
52   COMPONENT start_jump IS
53         PORT(
54            reset: IN STD_LOGIC;
55            addr: IN STD_LOGIC_VECTOR(4 DOWNTO 0);
56            cc: IN STD_LOGIC;
57            start: IN STD_LOGIC;
58            sel_addr: OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
59            );
60   END COMPONENT;
61
62   SIGNAL mux_out, addr_rom: STD_LOGIC_VECTOR(4 DOWNTO 0);
63   SIGNAL adder_out: UNSIGNED(4 DOWNTO 0);
64   SIGNAL rom_out: STD_LOGIC_VECTOR(31 DOWNTO 0);
65   SIGNAL cc: STD_LOGIC;
66   SIGNAL jmp_addr: STD_LOGIC_VECTOR(4 DOWNTO 0);
67   SIGNAL uir_out: STD_LOGIC_VECTOR(31 DOWNTO 0);
68   SIGNAL mux_sel: STD_LOGIC_VECTOR(1 DOWNTO 0);
69
70 BEGIN
71
72 uAR: reg GENERIC MAP( N=>5)
73     PORT MAP(clock=>CLOCK, data_in=>mux_out, clear=>'0', reset=>'0', LE=>'1', data_out=>addr_rom);
74
75 ROM: Rom_async PORT MAP(ADDR=>addr_rom, OUTPUT=>rom_out);
76
77 uIR: fall_register GENERIC MAP( N=>32)
78     PORT MAP(clock=>CLOCK, data_in=>rom_out, clear=>'0', reset=>'0', LE=>'1', data_out=>uir_out);
79
80 cc<=uir_out(31);
81 jmp_addr<=uir_out(30 DOWNTO 26);
82 COMMAND<=uir_out(25 DOWNTO 1);
83 DONE<=uir_out(0);
84
85
86 mux: mux3to1_std GENERIC MAP(N=>5)
87     PORT MAP (input1=>STD_LOGIC_VECTOR(adder_out), input2=>jmp_addr, input3=>"00000", sel=>mux_sel, output=>
       mux_out);
88
89 sommatore: adder_u GENERIC MAP(N=>5)
90       PORT MAP(a=>UNSIGNED(addr_rom), b=>"00001",s=>adder_out);
91
92 st_jmp: start_jump PORT MAP ( reset=>RESET, addr=>addr_rom, cc=>cc, start=>START, sel_addr=>mux_sel);
93
94 END Behavior;
```

## 9.8 Butterfly

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  USE ieee.numeric_std.all;
4
5  ENTITY Butterfly_final IS
6
7    PORT( CLOCK : IN STD_LOGIC;
8         START: IN STD_LOGIC;
9         RESET: IN STD_LOGIC;
10        W: IN STD_LOGIC_VECTOR(47 DOWNTO 0);
11        A: IN SIGNED(23 DOWNTO 0);
12        B: IN SIGNED(23 DOWNTO 0);
13        A1:OUT SIGNED(23 DOWNTO 0);
14        B1: OUT SIGNED(23 DOWNTO 0);
15        DONE: OUT STD_LOGIC
16        );
17 END Butterfly_final;
18
19 ARCHITECTURE Behaviour OF Butterfly_final IS
20
21 COMPONENT butterfly_dp IS
22
23   PORT( CLOCK : IN STD_LOGIC;
24        IN_A, IN_B: IN SIGNED (23 DOWNTO 0); --REGISTER FILE AB
25        ADDR_IN: IN STD_LOGIC_VECTOR (1 DOWNTO 0); --REGISTER FILE AB  --addr reg1 e reg2 + addr reg3 e reg4
26        ADDR_OUT : IN STD_LOGIC_VECTOR(3 DOWNTO 0); --REGISTER FILE AB --addr uscita1 + addr uscita2
```

```
27        RST_RF, LE_RF   : IN STD_LOGIC; --REGISTER FILE AB
28        Wi, Wr : IN SIGNED (23 DOWNTO 0); --REGISTER FILE W
29        LE_W, RST_W : IN STD_LOGIC; --REGISTER FILE W
30        ADDR_OUTW : IN STD_LOGIC; --REGISTER FILE W
31        MPY_SHn, RST_MULT: IN STD_LOGIC; --MOLTIPLICATORE
32        RST_SH, LE_SH : IN STD_LOGIC; --REGISTRO USCITA SHIFT
33        RST_MPY, LE_MPY: IN STD_LOGIC;--REGISTRO USCITA MOLTIPLICATORE
34        MUX_SUM: IN STD_LOGIC; --MUX INGRESSO SOMMATORE
35        MUX_SUB1: IN STD_LOGIC; --MUX INGRESSO1 SOTTRATTORE
36        MUX_SUB2: IN STD_LOGIC_VECTOR(1 DOWNTO 0); --MUX INGRESSO2 SOTTRATTORE
37        RST_SUB, LE_SUB: IN STD_LOGIC; --REGISTRO USCITA SOTTRATTORE
38        RST_SUM, LE_SUM: IN STD_LOGIC; --REGISTRO USCITA SOMMATORE
39        RST_RNDREG, LE_RNDREG: IN STD_LOGIC; --REG PER ROUNDING
40        MUX_ROUND, EN_ROUND: IN STD_LOGIC; --ROM ROUNDING + MUX INGRESSO
41        RST_A,LE_A: IN STD_LOGIC;
42        RST_A1,LE_A1: IN STD_LOGIC;
43        RST_B,LE_B: IN STD_LOGIC;
44        OUT_A, OUT_B : OUT SIGNED (23 DOWNTO 0));
45  END COMPONENT;
46
47  COMPONENT CU_Butterfly IS
48  PORT        (START              : IN STD_LOGIC;
49           RESET              : IN STD_LOGIC;
50           DONE          : OUT STD_LOGIC;
51           CLOCK         :IN STD_LOGIC;
52           COMMAND          : OUT STD_LOGIC_VECTOR(24 DOWNTO 0));
53  END COMPONENT;
54
55  SIGNAL w_re,w_im: SIGNED(23 DOWNTO 0);
56  SIGNAL command: STD_LOGIC_VECTOR(24 DOWNTO 0);
57
58  BEGIN
59  w_re<=SIGNED(W(47 DOWNTO 24));
60  w_im<=SIGNED(W(23 DOWNTO 0));
61
62  datapath: butterfly_dp PORT MAP(CLOCK=>CLOCK, IN_A=>A, IN_B=>B, ADDR_IN=>command(23 DOWNTO 22), ADDR_OUT=>
       command(21 DOWNTO 18),
63                          RST_RF=>command(24), LE_RF=>command(17), Wi=>w_im  , Wr=>w_re  , LE_W=>command(16), RST_W
       =>command(24),
64                          ADDR_OUTW=>command(15), MPY_SHn=>command(14), RST_MULT=>command(24), RST_SH=>command(24)
       ,
65                          LE_SH=>command(13), RST_MPY=>command(24), LE_MPY=>command(12), MUX_SUM=>command(11),
       MUX_SUB1=>command(10),
66                          MUX_SUB2=>command(9 DOWNTO 8), RST_SUB=>command(24), LE_SUB=>command(7), RST_SUM=>
       command(24), LE_SUM=>command(6),
67                          MUX_ROUND=>command(5), EN_ROUND=>command(4), RST_RNDREG=>command(24), LE_RNDREG=>command
       (3),
68                          RST_A=>command(24), LE_A=>command(2), RST_A1=>command(24),
69                          LE_A1=>command(1), RST_B=>command(24), LE_B=>command(0), OUT_A=>A1, OUT_B=>B1);
70
71  cu : CU_Butterfly PORT MAP(START=>START, RESET=>RESET, CLOCK=>CLOCK, DONE=>DONE, COMMAND=>command);
72
73  END Behaviour;
```

## 9.9 FFT

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  USE ieee.numeric_std.all;
4
5  ENTITY FFT IS
6
7    PORT( CLOCK : IN STD_LOGIC;
8         START: IN STD_LOGIC;
9         RESET: IN STD_LOGIC;
10        Xt: IN STD_LOGIC_VECTOR(383 DOWNTO 0);
11        --A: IN STD_LOGIC_VECTOR(191 DOWNTO 0);  --24 bit * 8
12        --B: IN STD_LOGIC_VECTOR(191 DOWNTO 0);
13        Xf: OUT STD_LOGIC_VECTOR(383 DOWNTO 0);
14        --A1:OUT STD_LOGIC_VECTOR(191 DOWNTO 0);
15        --B1: OUT STD_LOGIC_VECTOR(191 DOWNTO 0);
16        DONE: OUT STD_LOGIC
17         );
18 END FFT;
19
20 ARCHITECTURE Behaviour OF FFT IS
21
22 COMPONENT Butterfly_final IS
23
24   PORT( CLOCK : IN STD_LOGIC;
25        START: IN STD_LOGIC;
26        RESET: IN STD_LOGIC;
27        W: IN STD_LOGIC_VECTOR(47 DOWNTO 0);
28        A: IN SIGNED(23 DOWNTO 0);
29        B: IN SIGNED(23 DOWNTO 0);
30        A1:OUT SIGNED(23 DOWNTO 0);
```

```
31        B1: OUT SIGNED(23 DOWNTO 0);
32        DONE: OUT STD_LOGIC
33        );
34  END COMPONENT;
35
36  COMPONENT reg IS
37    GENERIC (N : INTEGER:=10);
38      PORT ( clock : IN STD_LOGIC;
39           data_in : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
40           clear, reset, LE: IN STD_LOGIC;  --Enable = load enable
41           data_out : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
42  END COMPONENT;
43
44  TYPE W_type IS ARRAY (0 TO 7) of STD_LOGIC_VECTOR(23 DOWNTO 0);
45  TYPE Sample_type IS ARRAY (0 TO 7) of SIGNED(23 DOWNTO 0);
46  TYPE DONE_type IS ARRAY (0 TO 31) of STD_LOGIC;
47  TYPE Twiddle IS ARRAY (0 TO 7) of STD_LOGIC_VECTOR(47 DOWNTO 0);
48  SIGNAL w_re_coeff: W_type;
49  SIGNAL w_im_coeff: W_type;
50  SIGNAL DONE_sgn1,DONE_sgn2,DONE_sgn3,DONE_sgn4: DONE_type;
51  SIGNAL DS1,DS2,DS3,DS4: STD_LOGIC;
52  SIGNAL twidf: Twiddle;
53  SIGNAL A_sample1: Sample_type;
54  SIGNAL B_sample1: Sample_type;
55  SIGNAL A_sample2: Sample_type;
56  SIGNAL B_sample2: Sample_type;
57  SIGNAL A_sample3: Sample_type;
58  SIGNAL B_sample3: Sample_type;
59  SIGNAL A_sample4: Sample_type;
60  SIGNAL B_sample4: Sample_type;
61
62  BEGIN
63
64  --registri per i coefficienti W
65  w_0re : reg GENERIC MAP(N=>24)
66      PORT MAP (clock=>CLOCK, data_in=>"011111111111111111111111", clear=>'0', reset=>'0', LE=>'1', data_out=>
        w_re_coeff(0)); --1
67
68  w_1re : reg GENERIC MAP(N=>24)
69      PORT MAP (clock=>CLOCK, data_in=>"011101100100000110101111", clear=>'0', reset=>'0', LE=>'1', data_out=>
        w_re_coeff(1)); --0.92387953251129
70
71  w_2re : reg GENERIC MAP(N=>24)
72      PORT MAP (clock=>CLOCK, data_in=>"010110101000001001111010", clear=>'0', reset=>'0', LE=>'1', data_out=>
        w_re_coeff(2)); --0.70710678118655
73
74  w_3re : reg GENERIC MAP(N=>24)
75      PORT MAP (clock=>CLOCK, data_in=>"001100001111101111000101", clear=>'0', reset=>'0', LE=>'1', data_out=>
        w_re_coeff(3)); --0.38268343236509
76
77  w_4re : reg GENERIC MAP(N=>24)
78      PORT MAP (clock=>CLOCK, data_in=>"000000000000000000000000", clear=>'0', reset=>'0', LE=>'1', data_out=>
        w_re_coeff(4)); --0
79
80  w_5re : reg GENERIC MAP(N=>24)
81      PORT MAP (clock=>CLOCK, data_in=>"110011110000010000111011", clear=>'0', reset=>'0', LE=>'1', data_out=>
        w_re_coeff(5)); --(-0.38268343236509)
82
83  w_6re : reg GENERIC MAP(N=>24)
84      PORT MAP (clock=>CLOCK, data_in=>"101001010111110110000110", clear=>'0', reset=>'0', LE=>'1', data_out=>
        w_re_coeff(6)); --(-0.70710678118655)
85
86  w_7re : reg GENERIC MAP(N=>24)
87      PORT MAP (clock=>CLOCK, data_in=>"100010011011111001010001", clear=>'0', reset=>'0', LE=>'1', data_out=>
        w_re_coeff(7)); --(-0.92387953251129)
88
89  w_0im : reg GENERIC MAP(N=>24)
90      PORT MAP (clock=>CLOCK, data_in=>"000000000000000000000000", clear=>'0', reset=>'0', LE=>'1', data_out=>
        w_im_coeff(0)); --0
91
92  w_1im : reg GENERIC MAP(N=>24)
93      PORT MAP (clock=>CLOCK, data_in=>"110011110000010000111011", clear=>'0', reset=>'0', LE=>'1', data_out=>
        w_im_coeff(1)); --(-0.38268343236509)
94
95  w_2im : reg GENERIC MAP(N=>24)
96      PORT MAP (clock=>CLOCK, data_in=>"101001010111110110000110", clear=>'0', reset=>'0', LE=>'1', data_out=>
        w_im_coeff(2)); --(-0.70710678118655)
97
98  w_3im : reg GENERIC MAP(N=>24)
99      PORT MAP (clock=>CLOCK, data_in=>"100010011011111001010001", clear=>'0', reset=>'0', LE=>'1', data_out=>
        w_im_coeff(3));  --(-0.92387953251129)
100
101 w_4im : reg GENERIC MAP(N=>24)
102     PORT MAP (clock=>CLOCK, data_in=>"100000000000000000000000", clear=>'0', reset=>'0', LE=>'1', data_out=>
        w_im_coeff(4)); --(-1)
103
104 w_5im : reg GENERIC MAP(N=>24)
105     PORT MAP (clock=>CLOCK, data_in=>"100010011011111001010001", clear=>'0', reset=>'0', LE=>'1', data_out=>
        w_im_coeff(5)); --(-0.92387953251129)
```

```
106
107 w_6im: reg GENERIC MAP(N=>24)
108     PORT MAP (clock=>CLOCK, data_in=>"101001010111110110000110", clear=>'0', reset=>'0', LE=>'1', data_out=>
        w_im_coeff(6)); --(-0.70710678118655)
109
110 w_7im: reg GENERIC MAP(N=>24)
111     PORT MAP (clock=>CLOCK, data_in=>"110011110000010000111011", clear=>'0', reset=>'0', LE=>'1', data_out=>
        w_im_coeff(7)); --(-0.38268343236509)
112
113 twidf(0)<=w_re_coeff(0) & w_im_coeff(0);
114 twidf(1)<=w_re_coeff(1) & w_im_coeff(1);
115 twidf(2)<=w_re_coeff(2) & w_im_coeff(2);
116 twidf(3)<=w_re_coeff(3) & w_im_coeff(3);
117 twidf(4)<=w_re_coeff(4) & w_im_coeff(4);
118 twidf(5)<=w_re_coeff(5) & w_im_coeff(5);
119 twidf(6)<=w_re_coeff(6) & w_im_coeff(6);
120 twidf(7)<=w_re_coeff(7) & w_im_coeff(7);
121
122 --PRIMO STADIO
123
124 btf_0: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>START, RESET=>RESET, W=>twidf(0), A=>SIGNED(Xt(383
        DOWNTO 360)),
125                         B=>SIGNED(Xt(191 DOWNTO 168)), A1=>A_sample1(0), B1=>B_sample1(0), DONE=>DONE_sgn1(0));
126
127 btf_1: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>START, RESET=>RESET, W=>twidf(0), A=>SIGNED(Xt(359
        DOWNTO 336)),
128                         B=>SIGNED(Xt(167 DOWNTO 144)), A1=>A_sample1(1), B1=>B_sample1(1), DONE=>DONE_sgn1(1));
129
130 btf_2: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>START, RESET=>RESET, W=>twidf(0), A=>SIGNED(Xt(335
        DOWNTO 312)),
131                         B=>SIGNED(Xt(143 DOWNTO 120)), A1=>A_sample1(2), B1=>B_sample1(2), DONE=>DONE_sgn1(2));
132
133 btf_3: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>START, RESET=>RESET, W=>twidf(0), A=>SIGNED(Xt(311
        DOWNTO 288)),
134                         B=>SIGNED(Xt(119 DOWNTO 96)), A1=>A_sample1(3), B1=>B_sample1(3), DONE=>DONE_sgn1(3));
135
136 btf_4: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>START, RESET=>RESET, W=>twidf(0), A=>SIGNED(Xt(287
        DOWNTO 264)),
137                         B=>SIGNED(Xt(95 DOWNTO 72)), A1=>A_sample1(4), B1=>B_sample1(4), DONE=>DONE_sgn1(4));
138
139 btf_5: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>START, RESET=>RESET, W=>twidf(0), A=>SIGNED(Xt(263
        DOWNTO 240)),
140                         B=>SIGNED(Xt(71 DOWNTO 48)), A1=>A_sample1(5), B1=>B_sample1(5), DONE=>DONE_sgn1(5));
141
142 btf_6: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>START, RESET=>RESET, W=>twidf(0), A=>SIGNED(Xt(239
        DOWNTO 216)),
143                         B=>SIGNED(Xt(47 DOWNTO 24)), A1=>A_sample1(6), B1=>B_sample1(6), DONE=>DONE_sgn1(6));
144
145 btf_7: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>START, RESET=>RESET, W=>twidf(0), A=>SIGNED(Xt(215
        DOWNTO 192)),
146                         B=>SIGNED(Xt(23 DOWNTO 0)), A1=>A_sample1(7), B1=>B_sample1(7), DONE=>DONE_sgn1(7));
147
148 DS1<=DONE_sgn1(0) AND DONE_sgn1(1) AND DONE_sgn1(2) AND DONE_sgn1(3) AND DONE_sgn1(4) AND DONE_sgn1(5) AND
        DONE_sgn1(6) AND DONE_sgn1(7);
149
150 --SECONDO STADIO
151
152 btf_8: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>DS1, RESET=>RESET, W=>twidf(0), A=>A_sample1(0),
153                         B=>A_sample1(4), A1=>A_sample2(0), B1=>B_sample2(0), DONE=>DONE_sgn2(0));
154
155 btf_9: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>DS1, RESET=>RESET, W=>twidf(0), A=>A_sample1(1),
156                         B=>A_sample1(5), A1=>A_sample2(1), B1=>B_sample2(1), DONE=>DONE_sgn2(1));
157
158 btf_10: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>DS1, RESET=>RESET, W=>twidf(0), A=>A_sample1(2),
159                         B=>A_sample1(6), A1=>A_sample2(2), B1=>B_sample2(2), DONE=>DONE_sgn2(2));
160
161 btf_11: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>DS1, RESET=>RESET, W=>twidf(0), A=>A_sample1(3),
162                         B=>A_sample1(7), A1=>A_sample2(3), B1=>B_sample2(3), DONE=>DONE_sgn2(3));
163
164 btf_12: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>DS1, RESET=>RESET, W=>twidf(4), A=>B_sample1(0),
165                         B=>B_sample1(4), A1=>A_sample2(4), B1=>B_sample2(4), DONE=>DONE_sgn2(4));
166
167 btf_13: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>DS1, RESET=>RESET, W=>twidf(4), A=>B_sample1(1),
168                         B=>B_sample1(5), A1=>A_sample2(5), B1=>B_sample2(5), DONE=>DONE_sgn2(5));
169
170 btf_14: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>DS1, RESET=>RESET, W=>twidf(4), A=>B_sample1(2),
171                         B=>B_sample1(6), A1=>A_sample2(6), B1=>B_sample2(6), DONE=>DONE_sgn2(6));
172
173 btf_15: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>DS1, RESET=>RESET, W=>twidf(4), A=>B_sample1(3),
174                         B=>B_sample1(7), A1=>A_sample2(7), B1=>B_sample2(7), DONE=>DONE_sgn2(7));
175
176 DS2<=DONE_sgn2(0) AND DONE_sgn2(1) AND DONE_sgn2(2) AND DONE_sgn2(3) AND DONE_sgn2(4) AND DONE_sgn2(5) AND
        DONE_sgn2(6) AND DONE_sgn2(7);
177
178 --TERZO STADIO
179
180 btf_16: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>DS2, RESET=>RESET, W=>twidf(0), A=>A_sample2(0),
181                         B=>A_sample2(2), A1=>A_sample3(0), B1=>B_sample3(0), DONE=>DONE_sgn3(0));
182
```

```
183 btf_17: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>DS2, RESET=>RESET, W=>twidf(0), A=>A_sample2(1),
184                         B=>A_sample2(3), A1=>A_sample3(1), B1=>B_sample3(1), DONE=>DONE_sgn3(1));
185
186 btf_18: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>DS2, RESET=>RESET, W=>twidf(4), A=>B_sample2(0),
187                         B=>B_sample2(2), A1=>A_sample3(2), B1=>B_sample3(2), DONE=>DONE_sgn3(2));
188
189 btf_19: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>DS2, RESET=>RESET, W=>twidf(4), A=>B_sample2(1),
190                         B=>B_sample2(3), A1=>A_sample3(3), B1=>B_sample3(3), DONE=>DONE_sgn3(3));
191
192 btf_20: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>DS2, RESET=>RESET, W=>twidf(2), A=>A_sample2(4),
193                         B=>A_sample2(6), A1=>A_sample3(4), B1=>B_sample3(4), DONE=>DONE_sgn3(4));
194
195 btf_21: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>DS2, RESET=>RESET, W=>twidf(2), A=>A_sample2(5),
196                         B=>A_sample2(7), A1=>A_sample3(5), B1=>B_sample3(5), DONE=>DONE_sgn3(5));
197
198 btf_22: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>DS2, RESET=>RESET, W=>twidf(6), A=>B_sample2(4),
199                         B=>B_sample2(6), A1=>A_sample3(6), B1=>B_sample3(6), DONE=>DONE_sgn3(6));
200
201 btf_23: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>DS2, RESET=>RESET, W=>twidf(6), A=>B_sample2(5),
202                         B=>B_sample2(7), A1=>A_sample3(7), B1=>B_sample3(7), DONE=>DONE_sgn3(7));
203
204 DS3<=DONE_sgn3(0) AND DONE_sgn3(1) AND DONE_sgn3(2) AND DONE_sgn3(3) AND DONE_sgn3(4) AND DONE_sgn3(5) AND
      DONE_sgn3(6) AND DONE_sgn3(7);
205
206 --QUARTO STADIO
207
208 btf_24: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>DS3, RESET=>RESET, W=>twidf(0), A=>A_sample3(0),
209                         B=>A_sample3(1), A1=>A_sample4(0), B1=>B_sample4(0), DONE=>DONE_sgn4(0));
210
211 btf_25: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>DS3, RESET=>RESET, W=>twidf(4), A=>B_sample3(0),
212                         B=>B_sample3(1), A1=>A_sample4(1), B1=>B_sample4(1), DONE=>DONE_sgn4(1));
213
214 btf_26: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>DS3, RESET=>RESET, W=>twidf(2), A=>A_sample3(2),
215                         B=>A_sample3(3), A1=>A_sample4(2), B1=>B_sample4(2), DONE=>DONE_sgn4(2));
216
217 btf_27: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>DS3, RESET=>RESET, W=>twidf(6), A=>B_sample3(2),
218                         B=>B_sample3(3), A1=>A_sample4(3), B1=>B_sample4(3), DONE=>DONE_sgn4(3));
219
220 btf_28: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>DS3, RESET=>RESET, W=>twidf(1), A=>A_sample3(4),
221                         B=>A_sample3(5), A1=>A_sample4(4), B1=>B_sample4(4), DONE=>DONE_sgn4(4));
222
223 btf_29: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>DS3, RESET=>RESET, W=>twidf(5), A=>B_sample3(4),
224                         B=>B_sample3(5), A1=>A_sample4(5), B1=>B_sample4(5), DONE=>DONE_sgn4(5));
225
226 btf_30: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>DS3, RESET=>RESET, W=>twidf(3), A=>A_sample3(6),
227                         B=>A_sample3(7), A1=>A_sample4(6), B1=>B_sample4(6), DONE=>DONE_sgn4(6));
228
229 btf_31: Butterfly_final PORT MAP (CLOCK=>CLOCK, START=>DS3, RESET=>RESET, W=>twidf(7), A=>B_sample3(6),
230                         B=>B_sample3(7), A1=>A_sample4(7), B1=>B_sample4(7), DONE=>DONE_sgn4(7));
231
232 DS4<=DONE_sgn4(0) AND DONE_sgn4(1) AND DONE_sgn4(2) AND DONE_sgn4(3) AND DONE_sgn4(4) AND DONE_sgn4(5) AND
      DONE_sgn4(6) AND DONE_sgn4(7);
233
234 DONE<=DS4;
235
236 Xf<=STD_LOGIC_VECTOR(A_sample4(0)) & STD_LOGIC_VECTOR(A_sample4(4)) & STD_LOGIC_VECTOR(A_sample4(2)) &
      STD_LOGIC_VECTOR(A_sample4(6)) &
237     STD_LOGIC_VECTOR(A_sample4(1)) & STD_LOGIC_VECTOR(A_sample4(5)) & STD_LOGIC_VECTOR(A_sample4(3)) &
      STD_LOGIC_VECTOR(A_sample4(7)) &
238     STD_LOGIC_VECTOR(B_sample4(0)) & STD_LOGIC_VECTOR(B_sample4(4)) & STD_LOGIC_VECTOR(B_sample4(2)) &
      STD_LOGIC_VECTOR(B_sample4(6)) &
239     STD_LOGIC_VECTOR(B_sample4(1)) & STD_LOGIC_VECTOR(B_sample4(5)) & STD_LOGIC_VECTOR(B_sample4(3)) &
      STD_LOGIC_VECTOR(B_sample4(7));
240
241 END Behaviour;
```

## 9.10 Testbench butterfly

```
1  library ieee;
2  USE ieee.std_logic_1164.all;
3  USE ieee.numeric_std.all;
4
5  ENTITY Testbench_butterfly_final IS
6    END Testbench_butterfly_final;
7
8  ARCHITECTURE Behavior OF Testbench_butterfly_final IS
9
10 SIGNAL clk, rst, st, d: STD_LOGIC;
11 SIGNAL a_in,b_in : SIGNED (23 DOWNTO 0);
12 SIGNAL w_in: STD_LOGIC_VECTOR (47 DOWNTO 0);
13 SIGNAL a_out, b_out: SIGNED (23 DOWNTO 0);
14
15
16 COMPONENT Butterfly_final IS
17
18   PORT( CLOCK : IN STD_LOGIC;
19         START: IN STD_LOGIC;
```

```
20        RESET: IN STD_LOGIC;
21        W: IN STD_LOGIC_VECTOR(47 DOWNTO 0);
22        A: IN SIGNED(23 DOWNTO 0);
23        B: IN SIGNED(23 DOWNTO 0);
24        A1:OUT SIGNED(23 DOWNTO 0);
25        B1: OUT SIGNED(23 DOWNTO 0);
26        DONE: OUT STD_LOGIC
27        );
28 END COMPONENT;
29
30
31 BEGIN
32
33  -- CLK: clk_gen port map(clock,RESET);
34 clck : PROCESS
35   BEGIN
36   clk <= '1';
37   wait for 20 ns;
38   clk <= '0';
39   wait for 20 ns;
40
41
42    END PROCESS;
43
44 rset : PROCESS
45   BEGIN
46   rst <= '1';
47   wait for 100 ns;
48   rst <= '0';
49   wait;
50    END PROCESS;
51
52
53 run : PROCESS
54   BEGIN
55
56   a_in<="00000000000000000000000";
57   b_in<="00000000000000000000000";
58   w_in<="000000000000000000000000000000000000000000000000";
59   st<='0';
60
61   wait for 300 ns;
62
63   st<='1';
64   wait for 40 ns;
65
66   a_in<="00000000000001000011001"; --Ar
67   b_in<="11111111111101011100001"; --Br
68   w_in<="011111111111111111111110000000000000000000000000"; --Wr Wi
69   st<='0';
70
71   wait for 40 ns;
72   a_in<="00000000000000100000110"; --Ai
73   b_in<="00000000000011100101011"; --Bi
74
75
76
77   wait;
78 END PROCESS;
79
80 FFT: Butterfly_final PORT MAP( CLOCK=>clk, RESET=>rst, START=>st, DONE=>d, A=>a_in, B=>b_in, W=>w_in, A1=>
       a_out, B1=>b_out);
81
82
83 END Behavior;
```

## 9.11 Testbench butterfly "full speed"

```
1  library ieee;
2  USE ieee.std_logic_1164.all;
3  USE ieee.numeric_std.all;
4
5  ENTITY Testbench_butterfly_double IS
6    END Testbench_butterfly_double;
7
8  ARCHITECTURE Behavior OF Testbench_butterfly_double IS
9
10 SIGNAL clk, rst, st, d: STD_LOGIC;
11 SIGNAL a_in,b_in : SIGNED (23 DOWNTO 0);
12 SIGNAL w_in: STD_LOGIC_VECTOR (47 DOWNTO 0);
13 SIGNAL a_out, b_out: SIGNED (23 DOWNTO 0);
14
15
16 COMPONENT Butterfly_final IS
17
18   PORT( CLOCK : IN STD_LOGIC;
19       START: IN STD_LOGIC;
```

```
20        RESET: IN STD_LOGIC;
21        W: IN STD_LOGIC_VECTOR(47 DOWNTO 0);
22        A: IN SIGNED(23 DOWNTO 0);
23        B: IN SIGNED(23 DOWNTO 0);
24        A1:OUT SIGNED(23 DOWNTO 0);
25        B1: OUT SIGNED(23 DOWNTO 0);
26        DONE: OUT STD_LOGIC
27        );
28 END COMPONENT;
29
30
31 BEGIN
32
33  -- CLK: clk_gen port map(clock,RESET);
34 clck : PROCESS
35   BEGIN
36   clk <= '1';
37   wait for 20 ns;
38   clk <= '0';
39   wait for 20 ns;
40
41
42    END PROCESS;
43
44 rset : PROCESS
45   BEGIN
46   rst <= '1';
47   wait for 100 ns;
48   rst <= '0';
49   wait;
50    END PROCESS;
51
52
53 run : PROCESS
54   BEGIN
55
56   a_in<="00000000000000000000000";
57   b_in<="00000000000000000000000";
58   w_in<="000000000000000000000000000000000000000000000000";
59   st<='0';
60
61   wait for 300 ns;
62
63   st<='1';
64   wait for 40 ns;
65
66   a_in<="00000000000010000011001"; --Ar
67   b_in<="11111111111101011100001"; --Br
68   w_in<="011111111111111111111110000000000000000000000000"; --Wr Wi
69   st<='0';
70
71   wait for 40 ns;
72   a_in<="00000000000000100000110"; --Ai
73   b_in<="00000000000011100101011"; --Bi
74
75   wait for 160 ns;
76   st<='1';
77   wait for 40 ns;
78
79   a_in<="00000000000010000011001"; --Ar
80   b_in<="11111111111101011100001"; --Br
81   w_in<="011111111111111111111110000000000000000000000000"; --Wr Wi
82  --st<='0';
83
84   wait for 40 ns;
85   a_in<="00000000000000100000110"; --Ai
86   b_in<="00000000000011100101011"; --Bi
87
88
89
90   wait;
91 END PROCESS;
92
93 FFT: Butterfly_final PORT MAP( CLOCK=>clk, RESET=>rst, START=>st, DONE=>d, A=>a_in, B=>b_in, W=>w_in, A1=>
       a_out, B1=>b_out);
94
95
96 END Behavior;
```

## 9.12 Testbench FFT

```
1 library ieee;
2 USE ieee.std_logic_1164.all;
3 USE ieee.numeric_std.all;
4
5 ENTITY Testbench_FFT IS
6   END Testbench_FFT;
```

```
7
8  ARCHITECTURE Behavior OF Testbench_FFT IS
9
10 SIGNAL clk, rst, st, d: STD_LOGIC;
11 SIGNAL OUTf, INt: STD_LOGIC_VECTOR(383 DOWNTO 0);
12 SIGNAL r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15 : STD_LOGIC_VECTOR (23 DOWNTO 0);
13 SIGNAL out0, out1, out2, out3, out4, out5, out6, out7, out8, out9, out10, out11, out12, out13, out14, out15:
       STD_LOGIC_VECTOR (23 DOWNTO 0);
14
15
16 COMPONENT FFT IS
17
18   PORT( CLOCK : IN STD_LOGIC;
19         START: IN STD_LOGIC;
20         RESET: IN STD_LOGIC;
21         Xt: IN STD_LOGIC_VECTOR(383 DOWNTO 0);
22         --A: IN STD_LOGIC_VECTOR(191 DOWNTO 0);  --24 bit * 8
23         --B: IN STD_LOGIC_VECTOR(191 DOWNTO 0);
24         Xf: OUT STD_LOGIC_VECTOR(383 DOWNTO 0);
25         --A1:OUT STD_LOGIC_VECTOR(191 DOWNTO 0);
26         --B1: OUT STD_LOGIC_VECTOR(191 DOWNTO 0);
27         DONE: OUT STD_LOGIC
28         );
29 END COMPONENT;
30
31
32 BEGIN
33
34  -- CLK: clk_gen port map(clock,RESET);
35
36 clck : PROCESS
37   BEGIN
38   clk <= '1';
39   wait for 20 ns;
40   clk <= '0';
41   wait for 20 ns;
42
43
44    END PROCESS;
45
46 rset : PROCESS
47   BEGIN
48   rst <= '1';
49   wait for 100 ns;
50   rst <= '0';
51   wait;
52    END PROCESS;
53
54
55 run : PROCESS
56   BEGIN
57
58   r0<="000000000000000000000000";
59   r1<="000000000000000000000000";
60   r2<="000000000000000000000000";
61   r3<="000000000000000000000000";
62   r4<="000000000000000000000000";
63   r5<="000000000000000000000000";
64   r6<="000000000000000000000000";
65   r7<="000000000000000000000000";
66   r8<="000000000000000000000000";
67   r9<="000000000000000000000000";
68   r10<="000000000000000000000000";
69   r11<="000000000000000000000000";
70   r12<="000000000000000000000000";
71   r13<="000000000000000000000000";
72   r14<="000000000000000000000000";
73   r15<="000000000000000000000000";
74
75   st<='0';
76
77   wait for 300 ns;
78
79   st<='1';
80   wait for 40 ns;
81
82   --REALI
83   r0<="111111000000000000000000";
84   r1<="111111000000000000000000";
85   r2<="000000111111111111111111";
86   r3<="000000111111111111111111";
87   r4<="111111000000000000000000";
88   r5<="111111000000000000000000";
89   r6<="000000111111111111111111";
90   r7<="000000111111111111111111";
91   r8<="111111000000000000000000";
92   r9<="111111000000000000000000";
93   r10<="000000111111111111111111";
94   r11<="000000111111111111111111";
```

```
95    r12<="11111100000000000000000";
96    r13<="11111100000000000000000";
97    r14<="00000011111111111111111";
98    r15<="00000011111111111111111";
99
100   st<='0';
101
102   wait for 40 ns;
103
104   --IMMAGINARI
105   r0<="00000000000000000000000";
106   r1<="00000000000000000000000";
107   r2<="00000000000000000000000";
108   r3<="00000000000000000000000";
109   r4<="00000000000000000000000";
110   r5<="00000000000000000000000";
111   r6<="00000000000000000000000";
112   r7<="00000000000000000000000";
113   r8<="00000000000000000000000";
114   r9<="00000000000000000000000";
115   r10<="00000000000000000000000";
116   r11<="00000000000000000000000";
117   r12<="00000000000000000000000";
118   r13<="00000000000000000000000";
119   r14<="00000000000000000000000";
120   r15<="00000000000000000000000";
121
122  wait;
123 END PROCESS;
124
125 FFT1: FFT PORT MAP( CLOCK=>clk, START=>st, RESET=>rst, Xt=>INt, Xf=>OUTf , DONE=>d);
126
127 INt<=r0&r1&r2&r3&r4&r5&r6&r7&r8&r9&r10&r11&r12&r13&r14&r15;
128
129 out15<=OUTf(23 DOWNTO 0);
130 out14<=OUTf(47 DOWNTO 24);
131 out13<=OUTf(71 DOWNTO 48);
132 out12<=OUTf(95 DOWNTO 72);
133 out11<=OUTf(119 DOWNTO 96);
134 out10<=OUTf(143 DOWNTO 120);
135 out9<=OUTf(167 DOWNTO 144);
136 out8<=OUTf(191 DOWNTO 168);
137 out7<=OUTf(215 DOWNTO 192);
138 out6<=OUTf(239 DOWNTO 216);
139 out5<=OUTf(263 DOWNTO 240);
140 out4<=OUTf(287 DOWNTO 264);
141 out3<=OUTf(311 DOWNTO 288);
142 out2<=OUTf(335 DOWNTO 312);
143 out1<=OUTf(359 DOWNTO 336);
144 out0<=OUTf(383 DOWNTO 360);
145
146 END Behavior;
```