# POLITECNICO DI TORINO


**PROGETTO**

**DI**

## SISTEMI DIGITALI INTEGRATI

# *BUTTERFLY*

**Data:**
**16/01/2015**

**Partecipanti:**
***Marco Coletta s231067***
***Francesco Condemi s231127***
***Nicolangelo Piscitelli s231005***

# Indice

# Capitolo 1

# Specifiche del sistema

Scopo di questa esercitazione è quello di realizzare una unità di elaborazione che esegua la Butterfly, la quale è l'elemento base di una FFT classica. Le specifiche assegnate sono le seguenti:

- Uso delle tecniche di microprogrammazione
- Utilizzo di un solo moltiplicatore che agisce sia come tale che da *shifter* aritmetico con un livello di *pipe* al suo interno
- Utilizzo di un solo sommatore che agisce anche da sottrattore puramente combinatorio
- Dati di ingresso con parallelismo a 16 bit con due bit di "guardia"
- Dati di uscita con parallelismo a 16 bit
- Segnale di START progettato in modo tale da discriminare una operazione singola da una operazione continua
- Segnale di DONE progettato in modo tale da interfacciarsi adeguatamente con un'altra Butterfly e decidere se quest'ultima debba lavorare in modalità isolata o continua
- Ottimizzazione dei bus e delle connessioni senza perdita di prestazioni
- Sequenziatore con indirizzamento esplicito
- $\mu$PC e $\mu$IR campionati su fasi opposte
- Uso del tecnica di approssimazione denominata *rounding to nearest even*. Tale processo deve, inoltre, occupare un intero ciclo di clock

# Capitolo 2

# Software utilizzati

Per la progettazione e la simulazione della "Butterfly" sono stati utilizzati i seguenti software:

- Quartus II 9.1 sp2 Web Edition: per la stesura dei codici VHDL
- ModelSim - Altera 6.5b (Quartus II 9.1 sp2) Starter Edition: per la simulazione del progetto mediante *test bench*
- MatLab R2014a: per la generazione dei dati da fornire alla simulazione e calcolo della relative uscite, usate come riferimento
- Inkscape 0.91: per la realizzazione dei grafici in forma vettoriale.
- TeXworks: per la stesura della relazione

# Capitolo 3

# Progetto

## 3.1 Data Flow Diagram

Il primo passo della fase progettuale è stato quello di derivare il *data flow diagram* dell'algoritmo.

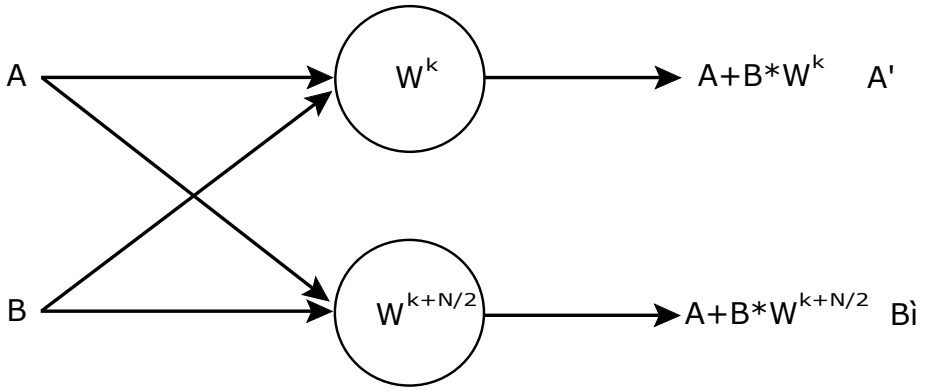


Figura 3.1: Schema grafico dell'algoritmo butterfly

Per questo motivo si è partiti dalle relazioni matematiche da cui si ricavano i valori di A' e B':

B' = Ar - BrWr + BiWi + j(Ai - BrWi - BiWr)
A' = Ar + BrWr - BiWi + j(Ai + BrWi + BiWr)

Tali relazioni possono essere divise scrivendo la relazione che permette di ottenere la parte immaginaria e quella reale.

B'r = Ar - BrWr + BiWi
B'i = Ai - BrWi - BiWr
A'r = Ar + BrWr - BiWi
A'i = Ai + BrWi + BiWr

Siccome in tutte le espressioni si ritrovano i parametri Br e Bi si è deciso di effettuare prima le operazioni che producono la parte reale e immaginaria di B' ed in

un secondo momento quella di A', sfruttando il fatto che esse sono in funzione di B'r e B'i secondo tali relazioni:

A'r = 2Ar - B'r
A'i = 2Ai - B'i

Per quanto riguarda il moltiplicatore un'altra specifica non marginale è quella sullo stadio di pipeline che ha obbligato a disporre le operazioni di somma due colpi di clock dopo l'entrata dei dati nel moltiplicatore. Questo ha reso impossibile l'utilizzo del sommatore già dai primi cicli di clock e ne ha posticipato l'utilizzo dal sesto ciclo in poi.
Dal momento che è possibile avere un dato solo in uscita per ogni ciclo di clock e per non complicare il meccanismo con cui i dati in uscita da una "Butterfly" entrano nella successiva, si è stabilito che il *register file* abbia un solo ingresso. Ciò comporta che i primi tre cicli di clock vengano spesi unicamente per caricare i primi dati in memoria.
In particolar modo i dati vengono acquisiti nell'ordine Wi, Wr, Br, Bi, Ar ed Ai (si noti come, quello di questi ultimi quattro dati, è anche l'ordine delle uscite della macchina).
Come da specifica lo START oltre a far partire la macchina discrimina i due casi di funzionamento. In particolare il segnale rimane attivo per 5 colpi di clock se si vuol far lavorare la macchina "in continua" altrimenti l'algoritmo verrà eseguito una sola volta.
Si deve tenere in considerazione che se due "Butterfly" sono collegate tra loro il segnale di DONE della prima è interpretato come START dalla seconda, perciò il DONE dovrà essere asserito nello "stato" 5a fino alla fine dell'esecuzione in modalità continua, oppure solo all'ultimo "stato" (12b) in caso di modalità singola (si veda la figura 3.8).

Per quanto concerne il DFD, sono stati considerati due possibili approcci che differiscono per la tempistica con cui si effettuano le varie moltiplicazioni e somme intermedie.
Approccio A:

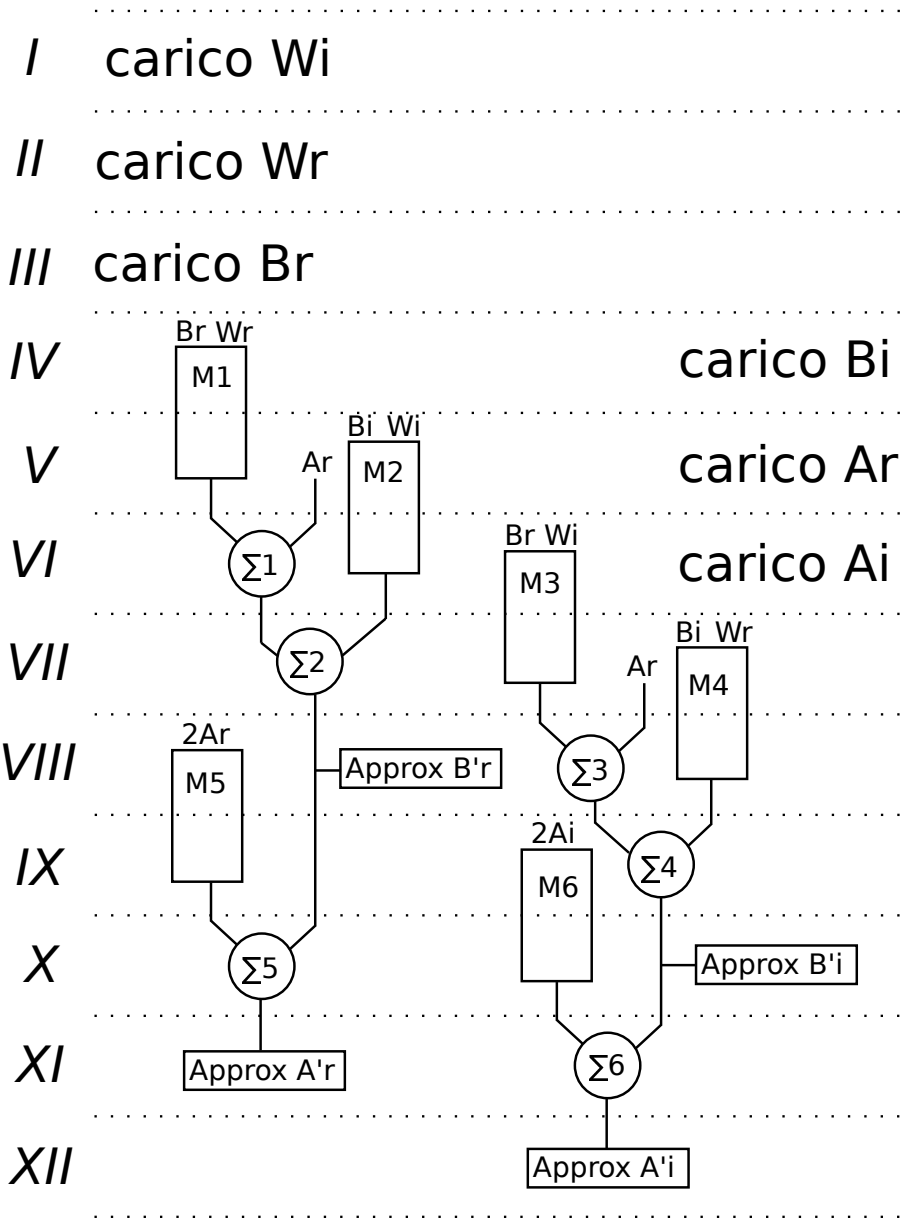


Figura 3.2: DFD approccio A

Approccio B:

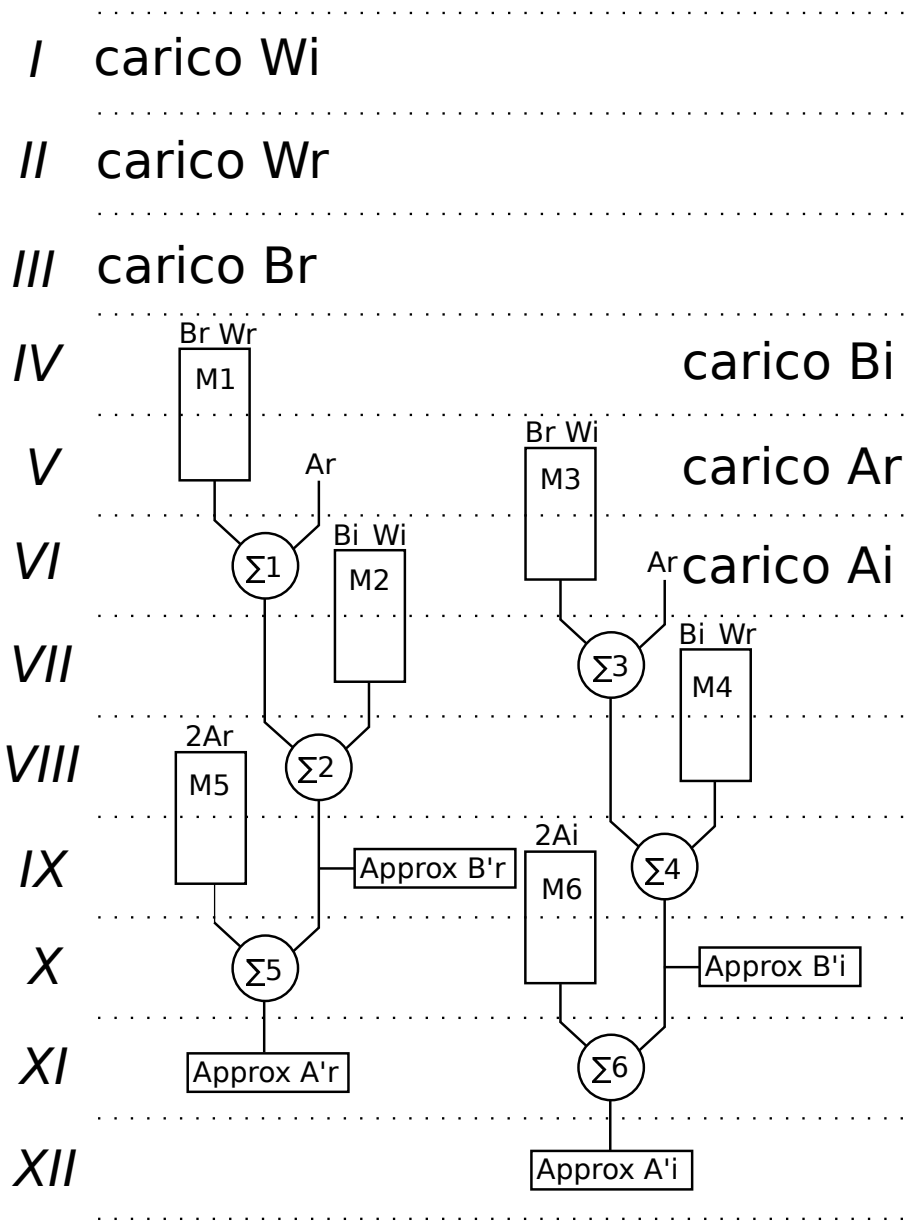


Figura 3.3: DFD approccio B

Ambedue le scelte permettono di avere l'ultimo dato in uscita dopo 12 colpi di clock e tutte le uscite nello stesso ordine in cui un'eventuale "Butterfly" successiva si aspetterebbe gli ingressi. L'approccio B, però, risulta essere migliore in quanto i nuovi valori di Ar, Ai, Br e Bi sono disponibili in sequenza, a distanza di un colpo di clock uno dall'altro, esattamente come vengono introdotti gli ingressi, mentre nell'approccio A, B'i viene generato 2 colpi di clock dopo B'r, il che richiederebbe l'aggiunta di un registro per ritardare il primo dato da inviare in uscita.
Per tale ragione la scelta è ricaduta sulla soluzione B.
Analizzando il tempo di vita delle variabili (riportato in figura 3.4), si è deciso di istanziare un *register file* da cinque registri, perché questo è il massimo numero di variabili utilizzato contemporaneamente (cicli di clock numero 5 e 6).

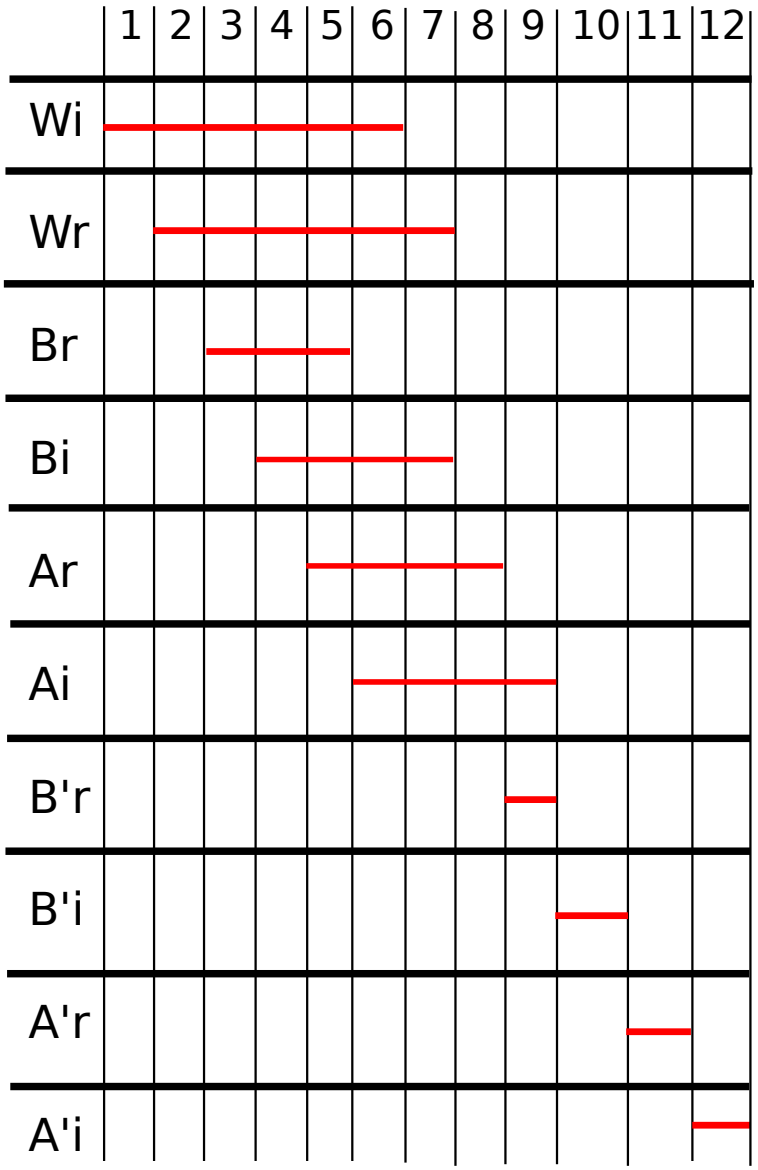


Figura 3.4: Diagramma del tempo di vita delle variabili

## 3.2 Data Path

Il *data path* comprende tutti i blocchi che servono per la corretta esecuzione dell'algoritmo.
Il *register file* contiene i dati che vengono caricati dall'esterno. Esso è collegato a un moltiplicatore che è in grado di effettuare sia la moltiplicazione di due dati sia la moltiplicazione di un dato per 2, agendo come uno *shifter* aritmetico. Un segnale di controllo permette di scegliere tra i due tipi di operazione. Il moltiplicatore secondo le specifiche ha un livello di *pipe*, ovvero che ad ogni ciclo di clock possa iniziare una nuova operazione, ma il risultato sarà disponibile non nel ciclo corrente ma dopo un ciclo di clock. Poichè i moltiplicandi sono numeri con formato fractional point, il risultato se non modificato opportunamente, presenterebbe due bit prima della virgola. Per cui si è deciso di scartare il bit più significativo e di aggiungere uno zero in posizione meno significativa. Per quanto riguarda il funzionamento da shifter si conserva il bit più significativo, si traslano di una posizione i restanti bit verso sinistra (eliminando il bit che segue l'MSB) e poi si estende il numero aggiungendo zeri nelle posizioni meno significative. L'uscita di tale blocco, poiché il risultato va sempre nel sommatore, viene salvata in un registro intermedio senza tornare nel *register file*.
Il sommatore, i cui dati in entrata sono selezionati dai tre multiplexer in modo da soddisfare tutte le possibili combinazioni previste dall'algoritmo, è in grado di effettuare anche sottrazioni in base al valore del segnale di controllo "sub_add". Il dato elaborato entra in un registro il quale a sua volta è collegato sia ad un altro registro che riporta il dato all'ingresso del sommatore che all'approssimatore, il cui scopo è quello di garantire che il parallelismo d'uscita sia coerente con le specifiche.
L'aggiunta di due registri in cascata (Reg2 e Reg3) si è resa necessaria poiché, così come indicato nel DFD, le uscite dei sommatori che generano i segnali S1, S2, S3 ed S4 devono essere salvate per due cicli di clock per poi poter essere riutilizzate.
Per collegare i vari componenti si è deciso di utilizzare non bus globali ma locali, point to point opportunamente dimensionati. Infatti in nessun caso più risorse richiedono lo stesso dato nello stesso momento.
Siccome il massimo numero di risorse che il *data path* richiede contemporaneamente al *register file* è pari a tre, si è stabilito che quest'ultimo avesse tre porte d'uscita.

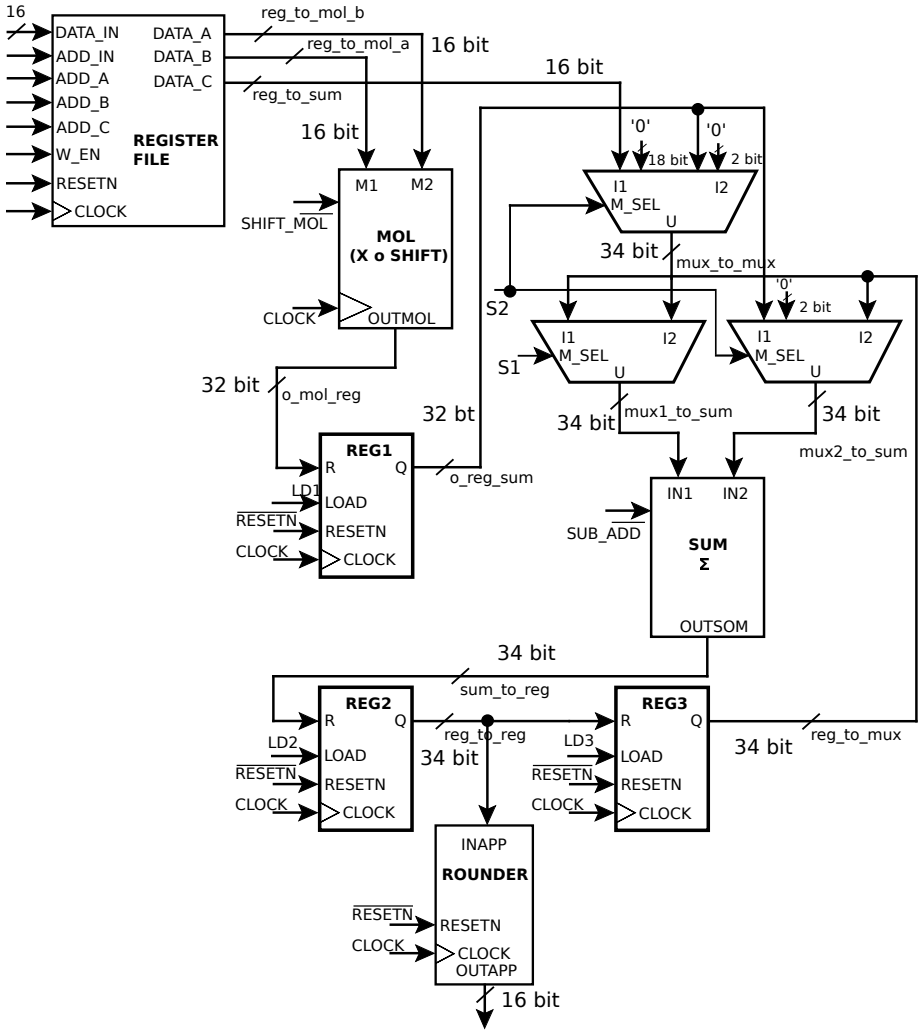


Figura 3.5: Data Path della Butterfly

### 3.2.1 Approssimatore

Considerando gli ingressi da 16 bit, si è calcolato che il parallelismo interno della macchina risulta essere su 34 bit (2*16 all'uscita del moltiplicatore + 2 dovuti alle somme successive). Le specifiche però impongono uscite della stessa grandezza degli input, perciò si è usato come metodo di approssimazione il *rounding to nearest even* (dettato da specifica). Esso prevede di sommare mezzo LSB al dato da approssimare e poi troncarlo, tranne nel caso in cui ci si trovi a metà della dinamica. In questo caso si deve approssimare il numero al valore pari più vicino eliminando cosi il BIAS error.
Per implementare tale algoritmo si è utilizzato un sommatore che riceve in ingresso i 17 bit più significativi e "00000000000000001"(mezzo LSB).
Per discriminare il caso di metà dinamica si è deciso di mettere in "AND" i 16 bit meno significativi del dato negati e il diciassettesimo. Se non si è a metà dell'intervallo, in uscita dal MUX è presente il risultato del sommatore, in caso contrario l'uscita sarà composta sempre dal dato proveniente dal sommatore ma con il secondo

bit meno significativo forzato a ’0’. In questo modo si ottiene esattamente l’approssimazione al pari più vicino. Il bit meno significativo viene scartato e il dato su 16 bit viene salvato in un registro.

```
                          parte significativa  |  parte da scartare
numero da approssimare 0001111110101000|100000000000000000
somma mezzo LSB        0001111110101001|000000000000000000
si porta l'LSB della parte significativa a 0 e la si salva nel registro

                    numero salvato

                    0001111110101000
```

```
                          parte significativa  |  parte da scartare
numero da approssimare 0001111110101001|100000000000000000
somma mezzo LSB        0001111110101010|000000000000000000
si porta il LSB della parte significativa a 0 e la si salva nel registro

                    numero salvato

                    0001111110101010
```

Figura 3.6: Esempi di casi a metà dinamica

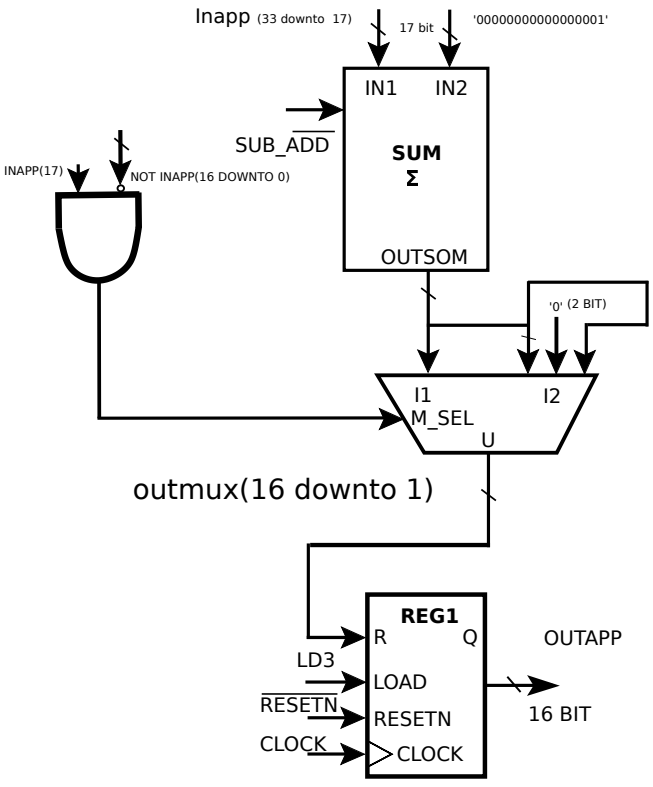


Figura 3.7: Schema del blocco approssimatore

## 3.3 Control Data Flow Diagram

Derivato il *data path*, è stato progettaro il *control data flow diagram* (mostrata in fig. 3.8) in modo da ricavare i comandi, che poi verranno inseriti nella $\mu$ROM, da passare ai vari blocchi nei vari cicli di clock.
Come si può vedere dal grafico sottostante, a partire dallo stato 5, vengono inviati controlli diversi a seconda se si lavori in modalità continua o isolata.
In particolar modo esse si diversificano per l'asserimento del DONE in momenti diversi, nonchè per il fatto che dopo il "salto", in modalità continua, il $\mu$PC punta sempre ad indirizzi dispari, mentre in modalità isolata agli indirizzi pari.

| cicli di clock | modalità singola (b) | modalità continua (a) |
|---|---|---|
| 0 | CC=1;N_ADD=00000;ARA=ARB=ARC=000;WE=0;AW=000; SHIFT_MOL=0;SUB_ADD=0;SEL1=SEL2=0;DONE=0; | |
| 1 | CC=0;N_ADD=00010;ARA=ARB=ARC=000;WE=1;AW=000; SHIFT_MOL=0;SUB_ADD=0;SEL1=SEL2=0;DONE=0; | |
| 2 | CC=0;N_ADD=00011;ARA=ARB=ARC=000;WE=1;AW=001; SHIFT_MOL=0;SUB_ADD=0;SEL1=SEL2=0;DONE=0; | |
| 3 | CC=0;N_ADD=00100;ARA=ARB=ARC=000;WE=1;AW=010; SHIFT_MOL=0;SUB_ADD=0;SEL1=SEL2=0;DONE=0; | |
| 4 | CC=1;N_ADD=00111;ARA=001;ARB=010;ARC=000; WE=1;AW=011;SHIFT_MOL=0;SUB_ADD=0; SEL1=SEL2=0;DONE=0; | |
| 5 | CC=0;N_ADD=01000;ARA=010;ARB=000; ARC=000;WE=1;AW=100;SHIFT_MOL=0; SUB_ADD=0;SEL1=SEL2=0;DONE=0; | CC=0;N_ADD=00101;ARA=001;ARB=000; ARC=001;WE=1;AW=100;SHIFT_MOL=0; SUB_ADD=0;SEL1=SEL2=0;DONE=1; |
| 6 | CC=0;N_ADD=01010;ARA=011;ARB=000; ARC=100;WE=1;AW=010;SHIFT_MOL=0; SUB_ADD=1;SEL1=1;SEL2=0;DONE=0; | CC=0;N_ADD=01001;ARA=011;ARB=000; ARC=100;WE=1;AW=010;SHIFT_MOL=0; SUB_ADD=1;SEL1=1;SEL2=0;DONE=1; |
| 7 | CC=0;N_ADD=01100;ARA=011;ARB=001; ARC=010;WE=0;AW=000;SHIFT_MOL=0; SUB_ADD=1;SEL1=1;SEL2=0;DONE=0; | CC=0;N_ADD=01011;ARA=011;ARB=001; ARC=010;WE=0;AW=000;SHIFT_MOL=0; SUB_ADD=1;SEL1=1;SEL2=0;DONE=1; |
| 8 | CC=0;N_ADD=01110;ARA=000;ARB=100; ARC=000;WE=0;AW=000;SHIFT_MOL=1; SUB_ADD=0;SEL1=1;SEL2=1;DONE=0; | CC=0;N_ADD=01101;ARA=000;ARB=100; ARC=000;WE=0;AW=000;SHIFT_MOL=1; SUB_ADD=0;SEL1=1;SEL2=1;DONE=1; |
| 9 | CC=0;N_ADD=10000;ARA=000;ARB=010; ARC=000;WE=0;AW=000;SHIFT_MOL=1; SUB_ADD=1;SEL1=0;SEL2=0;DONE=0; | CC=0;N_ADD=01111;ARA=000;ARB=010; ARC=000;WE=0;AW=000;SHIFT_MOL=1; SUB_ADD=1;SEL1=0;SEL2=0;DONE=1; |
| 10 | CC=0;N_ADD=10010;ARA=000;ARB=000; ARC=000;WE=0;AW=000;SHIFT_MOL=0; SUB_ADD=1;SEL1=1;SEL2=1;DONE=0; | CC=0;N_ADD=10001;ARA=000;ARB=000; ARC=000;WE=0;AW=000;SHIFT_MOL=0; SUB_ADD=1;SEL1=1;SEL2=1;DONE=1; |
| 11 | CC=0;N_ADD=10100;ARA=000;ARB=000; ARC=000;WE=0;AW=000;SHIFT_MOL=0; SUB_ADD=1;SEL1=1;SEL2=1;DONE=0; | CC=0;N_ADD=10011;ARA=000;ARB=000; ARC=000;WE=0;AW=000;SHIFT_MOL=0; SUB_ADD=1;SEL1=1;SEL2=1;DONE=1; |
| 12 | CC=1;N_ADD=00000;ARA=000;ARB=000; ARC=000;WE=0;AW=000;SHIFT_MOL=0; SUB_ADD=0;SEL1=0;SEL2=0;DONE=1; | CC=1;N_ADD=00000;ARA=000;ARB=000; ARC=000;WE=0;AW=000;SHIFT_MOL=0; SUB_ADD=0;SEL1=0;SEL2=0;DONE=1; |

Figura 3.8: Control Data Flow Diagram

## 3.4 $\mu$PC e $\mu$IR

La struttura della CU di una macchina microprogrammata prevede un registro che contiene l'indirizzo dell'istruzione che deve essere eseguita ($\mu$PC) e da una memoria in cui sono contenuti tutti i comandi. Questi ultimi vengono salvati in un altro registro ($\mu$IR) che contiene, inoltre, l'indirizzo dello stato futuro e altri segnali di controllo che serviranno alla Status PLA per decidere, in caso di salto, il percorso da seguire. Il $\mu$PC è dimensionato su 5 bit poichè sono presenti 21 possibili "stati" ed è sensibile al fronte di salita del clock.
Il $\mu$IR è composto da 24 bit ed è sensibile al fronte di discesa del clock. Al suo interno sono presenti campi fissi (*fixed format*) che consentono una codifica diretta in modo da evitare un ulteriore step di decodifica che potrebbe rallentare la macchina e aumentarne il percorso critico.

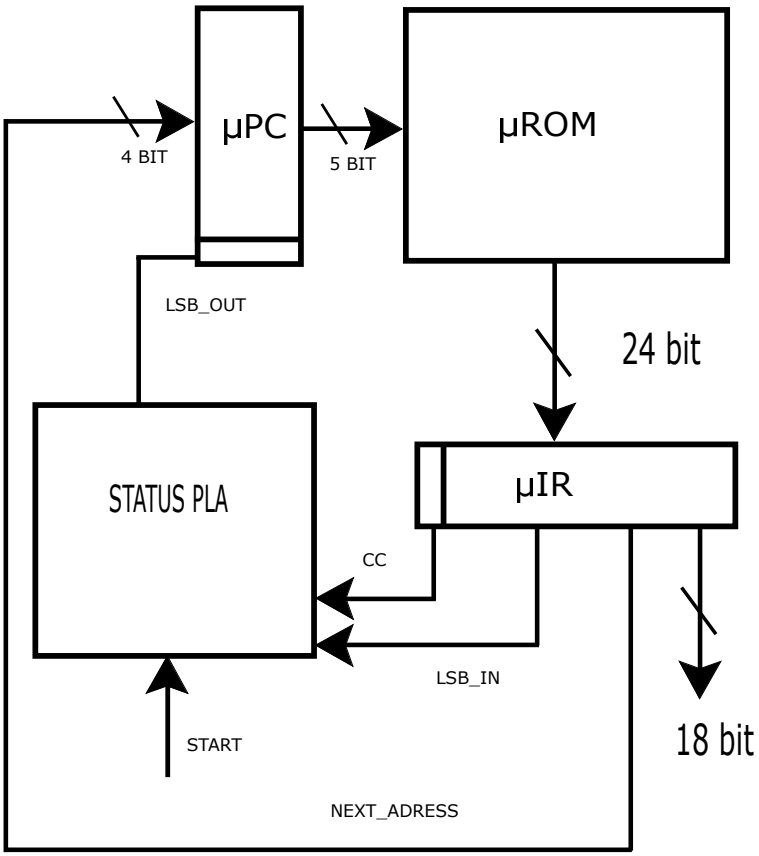


Figura 3.9: Struttura macchina microprogrammata con indirizzamento esplicito

## 3.5 $\mu$ROM

La $\mu$ROM contiene il codice in grado di gestire le operazioni svolte nel *data path*.
Vi è poi una colonna relativa al Condition Code usata per la gestione della status PLA ed in particolar modo per la gestione dei salti.
Si è scelto di inserire nella $\mu$ROM le seguenti colonne:

- CC (Condition Code) utilizzato per la gestione dei salti. Se a 1 ci può essere un salto 0 viceversa.
- NEXT ADDR(5 bit) sono cinque colonne che contengono l'indirizzo della $\mu$ROM nelle quali sono contenuti i comandi per le operazioni successive.
- ARA (3 bit) indirizzo A del register file utilizzato per la lettura del dato DATA_A
- ARB (3 bit) indirizzo B del register file utilizzato per la lettura del dato DATA_B
- ARC (3 bit) indirizzo C del register file utilizzato per la lettura del dato DATA_C
- WE Bit utilizzato per abilitare la scrittura del register file
- AW (3 bit) indirizzo del *register file* in cui deve essere scritto il dato presente su DATA_IN
- S_M segnale utilizzato per decidere tra shift aritmetico e moltiplicazione
- S_A segnale utilizzato per discriminare l'operazione di somma da differenza
- S1 bit utilizzato per la gestione dei mux 1 e 3
- S2 bit utilizzato per la gestione del mux 2
- DONE bit utilizzato per la generazione del segnale di DONE propriamente generato per discernere una esecuzione isolata da una continua

## 3.6 Status PLA

La Status PLA è un blocco interamente combinatorio che decide in base allo START e al CC se si deve procedere in sequenza o eseguire un salto. La tecnica utilizzata per la gestione dei salti è la two-way branch in cui il *next adress* è uno tra una coppia di indirizzi che differiscono solo per l'LSB. In questo modo la PLA modifica un solo bit e decide l'indirizzo dello stato futuro. In particolare, se si deve proseguire in maniera sequenziale, l'LSB rimane inalterato in uscita da tale blocco.
L'espressione che permette di ricavare l'"LSB_OUT" è la seguente:

LSB_OUT = (CC AND STATUS) OR (NOT(CC) OR LSB_IN).
Tale espressione è stata ricavata mediante l'utilizzo di una mappa di Karnaugh.

# Capitolo 4

# Test Bench

In questo capitolo verrà presentata la simulazione effettuata con il software *ModelSim-Altera 6.5b* per documentare l'effettivo funzionamento della "Butterfly". A tal proposito è stato progettato uno *script* MatLab la cui funzione è quella di generare in maniera casuale i dati da fornire all'ingresso della macchina (aventi valori compresi tra -0.25 e 0.25) e di simularne il funzionamento fornendo le relative quattro uscite B'r, B'i, A'r, A'i (si veda il capitolo 6).
Tali dati sono stati riprodotti su un file .dat il quale viene letto automaticamente dal codice VHDL progettato per eseguire il *test bench*, (si veda il capitolo 5).
Esso, infine, è stato programmato in modo che, nella simulazione riportata, la "Butterfly" funzioni in modalità isolata.
Nella tabella sottostante vengono riportati sia in forma binaria che in base in 10 i dati di ingresso forniti alla "Butterfly".

| Data In | | |
|---|---|---|
| Wi | 1110001000001001 | -0.234100341796875 |
| Wr | 0000110100101111 | 0.102996826171875 |
| Br | 1110011000110111 | -0.201446533203125 |
| Bi | 0001010010110011 | 0.161712646484375 |
| Ar | 1111000110111001 | -0.111541748046875 |
| Ai | 1110001011110100 | -0.226928710937500 |

Nella tabella che segue, invece, vengono riportati i valori in base 10 dei segnali M1, S1 e di tutte le uscite della "Butterfly" fornite da MatLab.

| Segnale Interni | |
|---|---|
| M1 | -0.020748353563249 |
| S1 | -0.090793394483626 |
| Data Out | |
| B'r | -0.128650380298495 |
| B'i | -0.290743302553892 |
| A'r | -0.094433115795255 |
| A'i | -0.163114119321108 |

In tutte le simulazioni viene mostrato il dato di uscita del $\mu$PC, utile a fornire una scansione temporanea dei processi all'interno della macchina.

## 4.1 Register File

In figura 4.1 viene proposto il risultato della simulazione per quanto riguarda la fase di caricamento del *Register File*
Per avere un riscontro grafico più immediato, i vari segnali sono stati presentati in formato decimale.

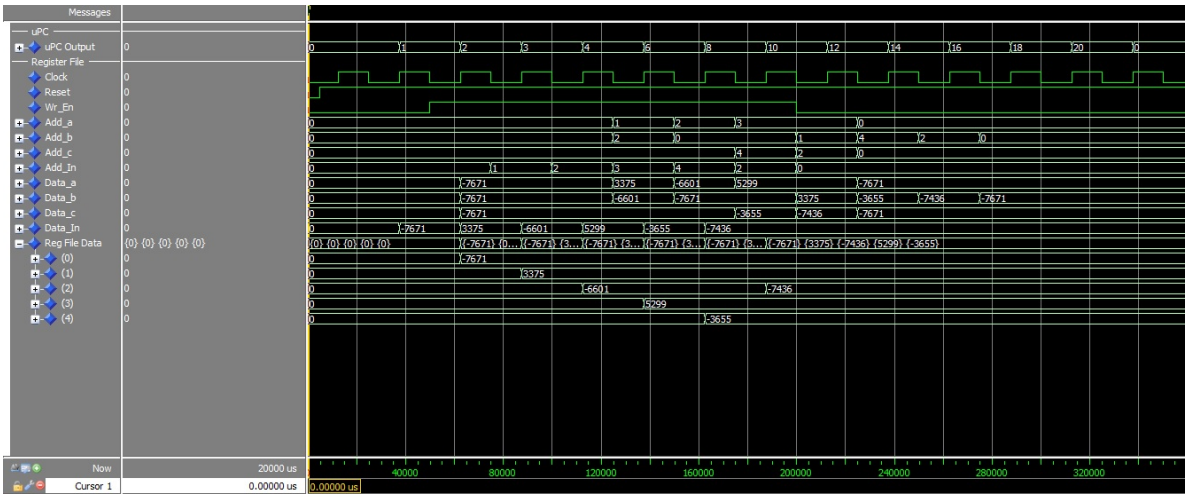


Figura 4.1: Simulazione della fase di caricamento del Register File

E' possibile notare che i dati vengono forniti alla "Butterfly" nel seguente ordine Wi, Wr, Br, Bi, Ar ed Ai. I primi cinque vanno a riempire il *register file* nell'ordine con cui entrano nella macchina, mentre l'ultimo segnale, Ai, va a sostituire il dato Br nella locazione di memoria puntata dall'indirizzo "010" (indicata con 2 nell'immagine) in quanto quest'ultimo, da quel momento in poi, non deve essere più utilizzato dal *data path*.

## 4.2 Moltiplicatore

In figura 4.2 vengono rappresentati i segnali che, durante tutto il funzionamento della "Butterfly", entrano ed escono dal moltiplicatore.

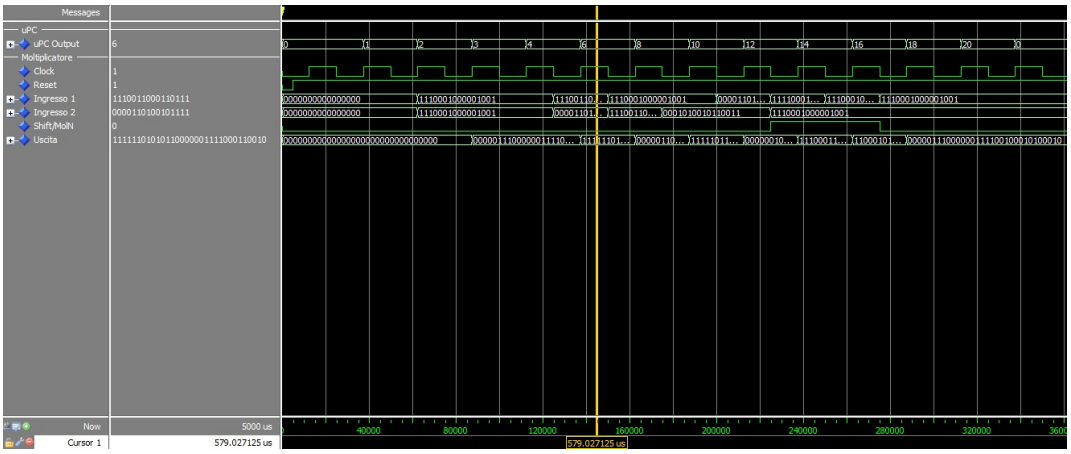


Figura 4.2: Simulazione del Moltiplicatore

Dalla simulazione si nota che, quando il $\mu$PC punta all'indirizzo 4 (quarto ciclo di clock), il moltiplicatore computa i dati Br (1110011000110111) e Wr (0000110100101111). Il risultato M1, in uscita al colpo di clock successivo, risulta essere "11111101010110000001111000110010", ovvero -0.020748353563249, in accordo con quanto calcolato mediante MatLab.
Si noti infine che quando il $\mu$PC punta agli indirizzi 12 e 14 il $\mu$IR seleziona il moltiplicatore affinché moltiplichi per due il valore presente sul primo ingresso, in accordo con quanto progettato e mostrato attraverso il *data flow diagram*, generando i segnali M5 ed M6.
Analizzando l'intera simulazione si evince che sono stati rispettati tutti i passaggi mostrati nel DFD.

## 4.3 Sommatore

In figura 4.3 vengono rappresentati i segnali che, durante tutto il funzionamento della "Butterfly", entrano ed escono dal sommatore.

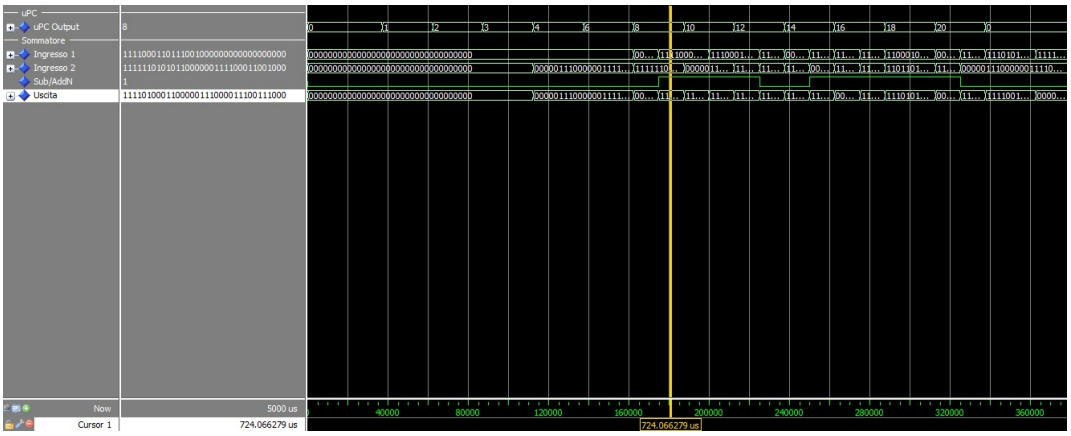


Figura 4.3: Simulazione del Sommatore

Durante il ciclo di clock numero sei, in cui il $\mu$PC punta all'indirizzo 8, all'ingresso del sommatore si hanno i segnali Ar ed M1 e il risultato della loro sottrazione, indicato con S1, è pari a "11110100011000001110000111001110000" ovvero -0.090793394483626.
Si noti che il sommatore è correttamente selezionato per operare la sottrazione tra i due ingressi per tutti i cicli di clock nei quali ne è richiesto l'uso ad eccezione dell'ottavo, ovvero quando produce il segnale S2.
Come per il moltiplicatore, si può concludere che sono stati rispettati tutti i passaggi mostrati nel DFD.

## 4.4 Approssimatore

In figura 4.4 viene mostrata la simulazione della fase finale della macchina, ovvero quella nella quale i segnali S2, S4, S5 ed S6 vengono approssimati e quindi riportati in uscita alla "Butterfly".

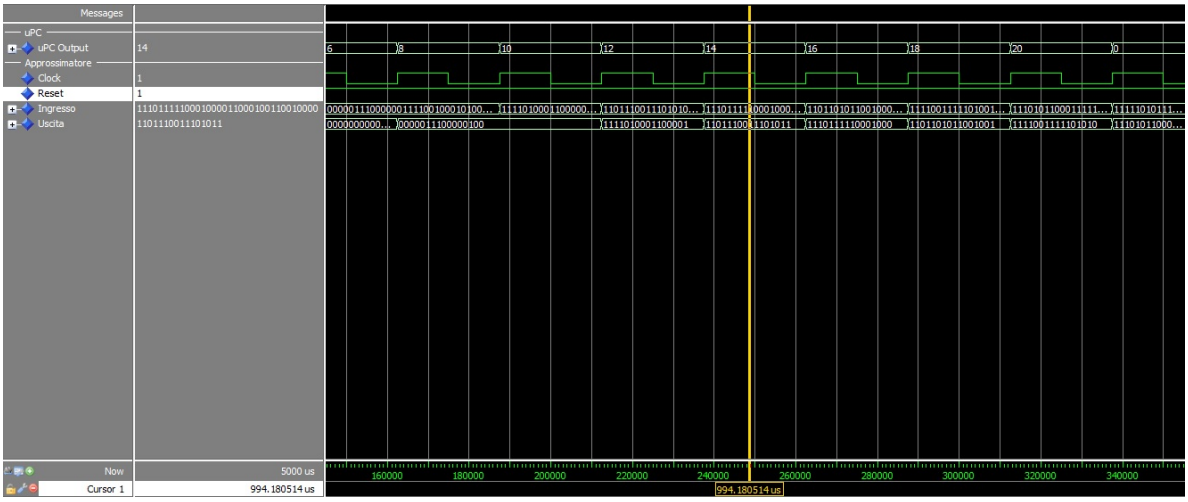


Figura 4.4: Simulazione dell' approssimatore

Si può subito evincere che, come richiesto dalle specifiche, il processo di approssimazione richiede un intero ciclo di clock. In corrispondenza dell'indirizzo 14, all'ingresso di tale circuito c'è il segnale S2 la cui codifica binaria è 11101111100010000110001001100 10000. La sua approssimazione, secondo la regola del *round to nearest even*, è pari a 1110111110001000 la quale risulta uguale a quella fornita dalla simulazione. Analizzando i passi successivi si può osservare che anche le restati tre approssimazioni sono state eseguite correttamente.

## 4.5 Butterlfy modalità isolata

Viene, infine, mostrata in figura 4.5 la simulazione della "Butterfly" vista come *black box* all'ingresso della quale sono stati posti i medesimi segnali presentati nelle sezioni precedenti.

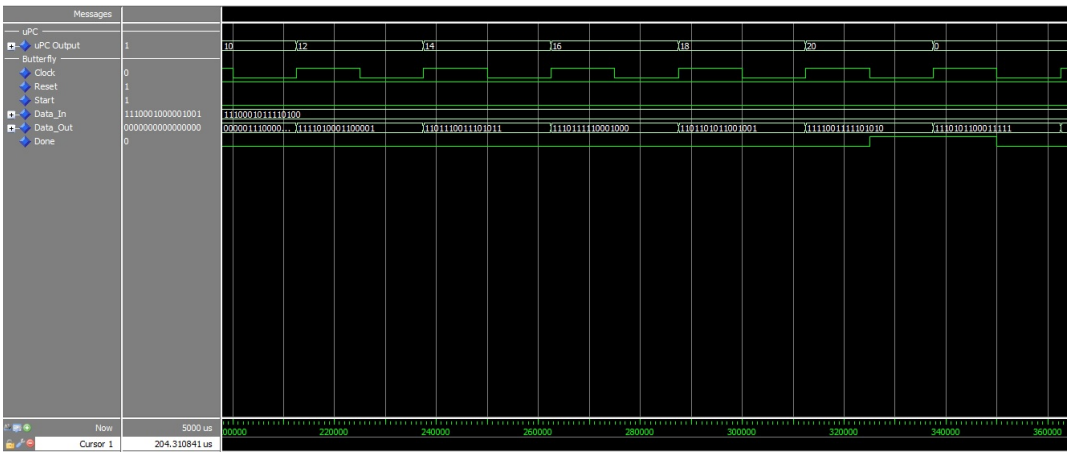


Figura 4.5: Simulazione della Butterfly vista come black box

Dalla simulazione, in uscita dalla macchina, risultano i seguenti valori:

B'r = "1110111110001000", ovvero -0.128662109
B'i = "1101101011001001", ovvero -0.290740966
A'r = "1111001111101010", ovvero -0.094421386
A'i = "1110101100011111", ovvero -0.163116455

E' immediato riscontrare come le uscite valide fornite dalla "Butterfly" (il $\mu$PC punta agli indirizzi 16, 18, 20 e 0) differiscano di pochissimo da quelle fornite dallo *script* MatLab. Ciò è plausibile in quanto questi segnali sono codificati su 16 bit, quindi, a differenza delle uscite del moltiplicatore e del sommatore che erano codificate su molti più bit, hanno una precisione inferiore a quella con cui MatLab esegue le sue operazioni. In particolar modo, si noti, che il risultato fornito dalla "Butterfly" è esatto fino alla quarta cifra decimale.

## 4.6 Butterfly modalità continua

In questa sezione viene mostrata la simulazione di un sistema costituito da due "Butterfly" poste l'una in cascata all'altra, quindi impostate in modalità continua.

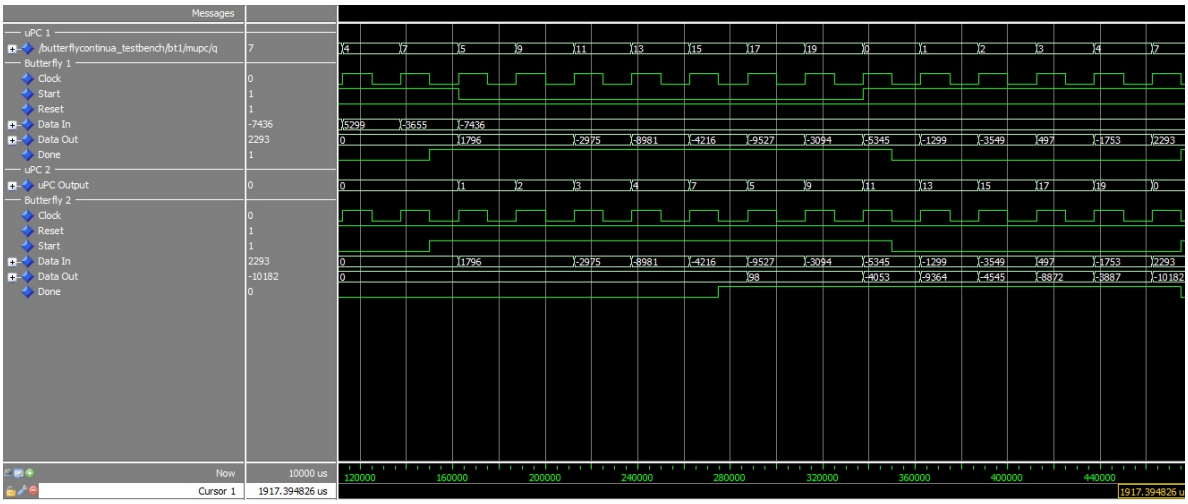


Figura 4.6: Simulazione della Butterfly in modalità continua

Si notino come le uscite valide della prima "Butterfly" (-4216, -9527, -3094, -5345) divengono gli ingressi della seconda, così come il "Done" della prima è lo "Start" della seconda.
Il dimensionamento della lunghezza del "Done" risulta essere corretta, in quanto è sufficiente a far funzionare anche la seconda macchina in modalità continua. Ciò è dimostrato dal fatto che anche il $\mu$PC di quest'ultima, dopo il *branch*, punta sempre ad indirizzi dispari.

# Capitolo 5

# Listati VHDL

## 5.1 Butterfly

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

entity butterfly is
port(       DATA_IN: IN SIGNED(15 DOWNTO 0);
            START,CLK,RESETN : IN STD_LOGIC;
            DATA_OUT: OUT SIGNED(15 DOWNTO 0);
            DONE : OUT STD_LOGIC
           );
end  butterfly ;

architecture behavior of  butterfly is

SIGNAL ADDR_UROM: STD_LOGIC_VECTOR(4 DOWNTO 0);
SIGNAL ROM_TO_UIR: STD_LOGIC_VECTOR(23 DOWNTO 0);
SIGNAL UIR_OUT: STD_LOGIC_VECTOR(23 DOWNTO 0);
SIGNAL PLA_TO_SEQ: STD_LOGIC;


COMPONENT microrom is
port(
            ADDR  : in std_logic_vector(4 downto 0);
            DATA_OUT: out STD_LOGIC_VECTOR(23 downto 0));
END COMPONENT;

COMPONENT REGN_STD IS
GENERIC (N: POSITIVE := 33);
PORT (R : IN STD_LOGIC_VECTOR(N DOWNTO 0);
LOAD: IN STD_LOGIC;
ClOCK, RESETN : IN STD_LOGIC;
```

```
Q : OUT STD_LOGIC_VECTOR(N DOWNTO 0));
END COMPONENT;

COMPONENT STATUS_PLA IS
PORT (
CC,STATUS,LSB_IN : IN STD_LOGIC;
LSB_OUT : OUT STD_LOGIC);
END COMPONENT;

COMPONENT REGN_F_STD IS
GENERIC (N: POSITIVE := 33);
PORT (R : IN STD_LOGIC_VECTOR(N DOWNTO 0);
LOAD: IN STD_LOGIC;
ClOCK, RESETN : IN STD_LOGIC;
Q : OUT STD_LOGIC_VECTOR(N DOWNTO 0));
END COMPONENT;

COMPONENT DATAPATH is
port( COM :IN STD_LOGIC_VECTOR(17 DOWNTO 0);
          ING:IN SIGNED(0 TO 15);
          CLOCK: IN STD_LOGIC;
       RESET: IN STD_LOGIC;
          DATA_OUT: OUT SIGNED(15 DOWNTO 0);
          DONE: OUT STD_LOGIC);
end  COMPONENT ;

BEGIN

MUROM: microrom PORT MAP(ADDR=>ADDR_UROM,DATA_OUT=>ROM_TO_UIR);

MUIR    :REGN_F_STD GENERIC MAP(N=>23)
                    PORT MAP(R=>ROM_TO_UIR,LOAD=>'1',ClOCK=>CLK,
                    RESETN=>RESETN,Q=>UIR_OUT);

MUPC    :REGN_STD GENERIC MAP(N=>4)
                    PORT MAP(R(4 DOWNTO 1)=>UIR_OUT(22 DOWNTO 19),
                    R(0)=>PLA_TO_SEQ,LOAD=>'1',ClOCK=>CLK,
                    RESETN=>RESETN,Q=>ADDR_UROM);

S_PLA   :STATUS_PLA  PORT MAP (CC=>UIR_OUT(23),STATUS=>START,
                    LSB_IN=>UIR_OUT(18),LSB_OUT=>PLA_TO_SEQ);

DP      : DATAPATH PORT MAP(COM=>UIR_OUT(17 DOWNTO 0),ING=>DATA_IN,
                    CLOCK=>CLK,RESET=>RESETN,DATA_OUT=>DATA_OUT,
                    DONE=>DONE);

end behavior;
```

### 5.1.1 Data Path

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


entity DATAPATH is
port( COM :IN STD_LOGIC_VECTOR(17 DOWNTO 0);
          ING:IN SIGNED(0 TO 15);
          CLOCK: IN STD_LOGIC;
      RESET: IN STD_LOGIC;
          DATA_OUT: OUT SIGNED(15 DOWNTO 0);
          DONE: OUT STD_LOGIC);
end  DATAPATH ;



architecture behavior of  DATAPATH is

SIGNAL REG_TO_MOL_A,REG_TO_MOL_B,REG_TO_SUM: SIGNED(15 DOWNTO 0);
SIGNAL O_MOL_REG,O_REG_SUM: SIGNED(31 DOWNTO 0);
SIGNAL MUX1_TO_SUM,MUX2_TO_SUM,SUM_TO_REG,REG_TO_REG,
                  REG_TO_MUX2,MUX_TO_MUX:SIGNED(33 DOWNTO 0);


COMPONENT reg_file IS
PORT (  CLOCK: IN STD_LOGIC;
                  RESET: IN STD_LOGIC;
                  WR_EN: IN STD_LOGIC;
                  ADD_A: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
                  ADD_B: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
                  ADD_C: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
                  ADD_IN: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
                  DATA_A,DATA_B,DATA_C:OUT SIGNED(0 TO 15);
                  DATA_IN:IN SIGNED(0 TO 15));
END COMPONENT;


COMPONENT REGN IS
GENERIC (N: POSITIVE := 33);
PORT (R : IN SIGNED(N DOWNTO 0);
          LOAD: IN STD_LOGIC;
          ClOCK, RESETN : IN STD_LOGIC;
          Q : OUT SIGNED(N DOWNTO 0));
END COMPONENT;
```

```
COMPONENT SUMM IS
GENERIC (N: POSITIVE := 16);
PORT ( IN1: IN SIGNED(N DOWNTO 0);
       IN2: IN SIGNED(N DOWNTO 0);
SUB_ADD: IN STD_LOGIC;
OUTSOM : OUT SIGNED(N DOWNTO 0));
END COMPONENT;

COMPONENT MUX2TO1 IS
GENERIC( N: POSITIVE :=33);
PORT( I1,I2: IN SIGNED( N DOWNTO 0);
      U: OUT SIGNED (N DOWNTO 0);
      M_SEL: IN STD_LOGIC);
END COMPONENT;


COMPONENT MOLT IS
GENERIC (N: POSITIVE := 15);
PORT ( M1: IN SIGNED(N DOWNTO 0);
       M2: IN SIGNED(N DOWNTO 0);
       SHIFT_MOL: IN STD_LOGIC;
       ClOCK, RESETN ,LDM: IN STD_LOGIC;
       OUTMOL : OUT SIGNED(31 DOWNTO 0));
END COMPONENT;

COMPONENT APPROX IS
PORT (INAPP : IN SIGNED(33 DOWNTO 0);
ClOCK, RESETN ,LD3: IN STD_LOGIC;
OUTAPP: OUT SIGNED(15 DOWNTO 0));
END COMPONENT;

BEGIN

RF      :reg_file PORT MAP(CLOCK=>CLOCK,RESET=>RESET,
WR_EN=>COM(8),ADD_A=>COM(17 DOWNTO 15),
        ADD_B=>COM(14 DOWNTO 12),ADD_C=>COM(11 DOWNTO 9),
        ADD_IN=>COM(7 DOWNTO 5),
        DATA_A=>REG_TO_MOL_B,DATA_B=>REG_TO_MOL_A,
        DATA_C=>REG_TO_SUM,DATA_IN=>ING);
REG1   :REGN GENERIC MAP(N=>31)
        PORT MAP( R=>O_MOL_REG,LOAD=>'1',CLOCK=>CLOCK,
        RESETN=>RESET,Q=>O_REG_SUM);
MOL    :MOLT GENERIC MAP( N=>15)
        PORT MAP ( M1=>REG_TO_MOL_A,M2=>REG_TO_MOL_B,SHIFT_MOL=>COM(4),
        CLOCK=>CLOCK,RESETN=>RESET,LDM=>'1',OUTMOL=>O_MOL_REG);
SUM    :SUMM GENERIC MAP (N=>33)
        PORT MAP ( IN1=>MUX1_TO_SUM,IN2=>MUX2_TO_SUM,
```

```
        SUB_ADD=>COM(3),OUTSOM=>SUM_TO_REG);
MUX3    : MUX2TO1 GENERIC MAP (N=>33)
        PORT MAP (I2(0)=>'0',I2(1)=>'0',
        I2(33 DOWNTO 2)=>O_REG_SUM,
        I1(33 DOWNTO 18)=>REG_TO_SUM,I1(17 DOWNTO 0)=>(OTHERS=>'0'),
        U=>MUX_TO_MUX,M_SEL=>COM(1));
MUX1    : MUX2TO1 GENERIC MAP (N=>33)
        PORT MAP (I2=>MUX_TO_MUX,I1=>REG_TO_MUX2,
        U=>MUX1_TO_SUM,M_SEL=>COM(2));
MUX2    :MUX2TO1 GENERIC MAP (N=>33)
         PORT MAP (I1(1)=>'0',I1(0)=>'0',
         I1(33 DOWNTO 2)=>O_REG_SUM,I2=>REG_TO_MUX2,
         U=>MUX2_TO_SUM,M_SEL=>COM(1));
REG2    :REGN GENERIC MAP(N=>33)
        PORT MAP( R=>SUM_TO_REG,LOAD=>'1',CLOCK=>CLOCK,
        RESETN=>RESET,Q=>REG_TO_REG);
REG3    :REGN GENERIC MAP(N=>33)
        PORT MAP( R=>REG_TO_REG,LOAD=>'1',CLOCK=>CLOCK,
        RESETN=>RESET,Q=>REG_TO_MUX2);
ROUNDER :APPROX PORT MAP(INAPP=>REG_TO_REG,CLOCK=>CLOCK,
RESETN=>RESET,LD3=>'1',OUTAPP=>DATA_OUT);
DONE<=COM(0);
end behavior;
```

#### 5.1.1.1 MOLT

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

ENTITY MOLT IS
GENERIC (N: POSITIVE := 15);
PORT ( M1: IN SIGNED(N DOWNTO 0);
       M2: IN SIGNED(N DOWNTO 0);
       SHIFT_MOL: IN STD_LOGIC;
       ClOCK, RESETN ,LDM: IN STD_LOGIC;
       OUTMOL : OUT SIGNED(31 DOWNTO 0));
END MOLT;

ARCHITECTURE BEHAV OF MOLT IS

COMPONENT REGN IS
GENERIC (N: POSITIVE := 33);
PORT (R : IN SIGNED(N DOWNTO 0);
LOAD: IN STD_LOGIC;
ClOCK, RESETN : IN STD_LOGIC;
Q : OUT SIGNED(N DOWNTO 0));
END COMPONENT;

SIGNAL TEMP_M: SIGNED(31 DOWNTO 0);
SIGNAL OUTM: SIGNED(31 DOWNTO 0);


BEGIN
PROCESS (M1,M2,SHIFT_MOL, TEMP_M)
BEGIN
IF (SHIFT_MOL= '0') THEN
TEMP_M <= M1*M2;
OUTM(31 DOWNTO 1) <= TEMP_M(30 DOWNTO 0);
OUTM(0) <= '0';
ELSE

OUTM(31) <= M1(15);
OUTM(30 DOWNTO 17) <= M1(13 downto 0);
OUTM(16 downto 0)<=( others => '0');

END IF;
END PROCESS;

REG31:REGN GENERIC MAP (N=>31)
       PORT MAP(R=>OUTM,LOAD=>LDM,RESETN=>RESETN,
```

```
        CLOCK=>CLOCK,Q=>OUTMOL);
 END BEHAV;
```

#### 5.1.1.2 MUX2TO1

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY MUX2TO1 IS
GENERIC( N: POSITIVE :=33);
PORT(I1,I2: IN SIGNED( N DOWNTO 0);
      U: OUT SIGNED (N DOWNTO 0);
      M_SEL: IN STD_LOGIC);
END MUX2TO1;

ARCHITECTURE BEHAV OF MUX2TO1 IS
BEGIN
MUXPROCESS: PROCESS(I1,I2,M_SEL)
BEGIN
IF M_SEL = '0' THEN
         U <= I1;
ELSE
         U <= I2;
END IF;
END PROCESS;
END BEHAV;
```

#### 5.1.1.3 REGN

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
ENTITY REGN IS
GENERIC (N: POSITIVE := 33);
PORT (R : IN SIGNED(N DOWNTO 0);
LOAD: IN STD_LOGIC;
ClOCK, RESETN : IN STD_LOGIC;
Q : OUT SIGNED(N DOWNTO 0));
END REGN;
ARCHITECTURE Behavior OF REGN IS
BEGIN
PROCESS (CLOCK, RESETN)
BEGIN
IF (RESETN = '0') THEN
Q <= (OTHERS => '0');
ELSIF (CLOCK'EVENT AND CLOCK = '1') THEN
IF LOAD='1' THEN
Q <= R;
END IF;
END IF;
END PROCESS;
END Behavior;
```

#### 5.1.1.4 reg_file

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
ENTITY reg_file IS
PORT (  CLOCK: IN STD_LOGIC;
        RESET: IN STD_LOGIC;
        WR_EN: IN STD_LOGIC;
        ADD_A,ADD_B,ADD_C,ADD_IN: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
        DATA_A,DATA_B,DATA_C:OUT SIGNED(0 TO 15);
        DATA_IN:IN SIGNED(0 TO 15));
END reg_file;

ARCHITECTURE BEHAVIOR OF reg_file IS

TYPE REG IS ARRAY (0 TO 4) OF SIGNED(15 DOWNTO 0);
SIGNAL DATA: REG;
BEGIN
MEM: PROCESS(RESET,CLOCK)
BEGIN
IF RESET = '0' THEN
          DATA<=(OTHERS => "0000000000000000");
ELSIF CLOCK' EVENT AND CLOCK='1' THEN
          IF WR_EN = '1'THEN
                DATA(to_integer(unsigned(ADD_IN)))<=DATA_IN;

          END IF;
END IF;
END PROCESS;

DATA_A<=DATA(to_integer(unsigned(ADD_A)));
DATA_B<=DATA(to_integer(unsigned(ADD_B)));
DATA_C<=DATA(to_integer(unsigned(ADD_C)));
END ARCHITECTURE;
```

#### 5.1.1.5 APPROX

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

ENTITY APPROX IS

PORT (INAPP : IN SIGNED(33 DOWNTO 0);
ClOCK, RESETN ,LD3: IN STD_LOGIC;
OUTAPP: OUT SIGNED(15 DOWNTO 0));
END APPROX;

ARCHITECTURE BEHAV OF APPROX IS

COMPONENT REGN IS
GENERIC (N: POSITIVE := 33);
PORT (R : IN SIGNED(N DOWNTO 0);
LOAD: IN STD_LOGIC;
ClOCK, RESETN : IN STD_LOGIC;
Q : OUT SIGNED(N DOWNTO 0));
END COMPONENT;

COMPONENT MUX2TO1 IS
GENERIC( N: POSITIVE :=33);
PORT( I1 ,I2: IN SIGNED( N DOWNTO 0);
      U: OUT SIGNED (N DOWNTO 0);
      M_SEL: IN STD_LOGIC);
END COMPONENT;

COMPONENT SUMM IS
GENERIC (N: POSITIVE := 16);
PORT ( IN1: IN SIGNED(N DOWNTO 0);
       IN2: IN SIGNED(N DOWNTO 0);
SUB_ADD: IN STD_LOGIC;
OUTSOM : OUT SIGNED(N DOWNTO 0));
END COMPONENT;

SIGNAL OUTAND: STD_LOGIC ;
SIGNAL OUTMUX1: SIGNED(16 DOWNTO 0);
SIGNAL OUTS: SIGNED(16 DOWNTO 0);

BEGIN

OUTAND<=INAPP(17) AND NOT INAPP(16) AND NOT INAPP(15)
        AND NOT INAPP(14) AND NOT INAPP(13) AND NOT INAPP(12)
        AND NOT INAPP(11) AND NOT INAPP(10) AND NOT INAPP(9)
```

```
          AND NOT INAPP(8) AND NOT INAPP(7) AND NOT INAPP(6)
          AND NOT INAPP(5) AND NOT INAPP(4) AND NOT INAPP(3)
          AND NOT INAPP(2) AND NOT INAPP(1) AND NOT INAPP(0);

SUM1:SUMM GENERIC MAP (N=>16)
      PORT MAP(IN1=>INAPP(33 DOWNTO 17),
      IN2=>"00000000000000001", SUB_ADD=>'0',OUTSOM=>OUTS);

MUX1: MUX2TO1 GENERIC MAP (N=>16)
       PORT MAP(I1=>OUTS, I2(16 DOWNTO 2)=>OUTS(16 DOWNTO 2),
       I2(1)=>'0',I2(0)=>OUTS(0),M_SEL=>OUTAND,U=>OUTMUX1);

REG16:REGN GENERIC MAP (N=>15)
        PORT MAP(R=>OUTMUX1(16 DOWNTO 1),LOAD=>LD3,
        RESETN=>RESETN,CLOCK=>CLOCK,Q=>OUTAPP);
END BEHAV;
```

#### 5.1.1.6 SUMM

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

ENTITY SUMM IS
GENERIC (N: POSITIVE := 16);
PORT ( IN1: IN SIGNED(N DOWNTO 0);
       IN2: IN SIGNED(N DOWNTO 0);
SUB_ADD: IN STD_LOGIC;
OUTSOM : OUT SIGNED(N DOWNTO 0));
END SUMM;

ARCHITECTURE BEHAV OF SUMM IS
BEGIN
PROCESS (IN1,IN2,SUB_ADD )
BEGIN
IF (SUB_ADD= '0') THEN
OUTSOM<= IN1+IN2;
ELSE
OUTSOM<= IN1-IN2;
END IF;
END PROCESS;
END BEHAV;
```

#### 5.1.1.7 REGN_F_STD

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
ENTITY REGN_F_STD IS
GENERIC (N: POSITIVE := 33);
PORT (R : IN STD_LOGIC_VECTOR(N DOWNTO 0);
LOAD: IN STD_LOGIC;
ClOCK, RESETN : IN STD_LOGIC;
Q : OUT STD_LOGIC_VECTOR(N DOWNTO 0));
END REGN_F_STD ;
ARCHITECTURE Behavior OF REGN_F_STD  IS
BEGIN
PROCESS (CLOCK, RESETN)
BEGIN
IF (RESETN = '0') THEN
Q <= (OTHERS => '0');
ELSIF (CLOCK'EVENT AND CLOCK = '0') THEN
IF LOAD='1' THEN
Q <= R;
END IF;
END IF;
END PROCESS;
END Behavior;
```

#### 5.1.1.8 STATUS_PLA

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

ENTITY STATUS_PLA IS
PORT (
CC,STATUS,LSB_IN : IN STD_LOGIC;
LSB_OUT : OUT STD_LOGIC);
END STATUS_PLA;
ARCHITECTURE Behavior OF STATUS_PLA IS
BEGIN
LSB_OUT<=( CC AND STATUS) OR ( NOT CC AND LSB_IN);
END Behavior;
```

### 5.1.2 $\mu$ROM

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

entity microrom is
port(
                ADDR    : in std_logic_vector(4 downto 0);
                DATA_OUT: out STD_LOGIC_VECTOR(23 downto 0)
);
end  microrom ;

architecture behavior of  microrom  is

    type microrom_data is array (0 to 31)
        of STD_LOGIC_VECTOR(23 downto 0);

    constant ctrl_data: microrom_data := (
--                      NEXT
--                      AAAAA
--                      DDDDDAAAAAAAAA         D
--                      DDDDDRRRRRRRRR AAASS   O
--                     CRRRRRAAABBBCCCWWWW//SSN
--  addr               C432102102102l0E210MA12E       stato
                0 => "10000000000000000000000", -- idle
                1 => "00001000000000100000000", -- c_wi
                2 => "00001100000000100100000", -- c_wr
                3 => "00010000000000101000000", -- c_br
                4 => "10011100101000010110000", -- c_bi
                5 => "00100101100010010100l101", -- c_ai_1
                6 => "00100001000000110000000", -- c_ar_0
```

```
                    7 => "000101010000000110000001", -- c_ar_1
                    8 => "001010011000100101001100", -- c_ai_0
                    9 => "001011011001010000001101", -- sum_3_1
                    10 =>"001100011001010000001100", -- sum_3_0
                    11 =>"001101000100000000010111", -- sum_2_1
--                          NEXT
--                          AAAAA
--                          DDDDDAAAAAAAAA          D
--                          DDDDDRRRRRRRRR AAASS   O
--                         CRRRRRAAABBBCCCWWWW//SSN
--   addr                  C43210210210210E210MA12E        stato
                    12 =>"001110000100000000010110", -- sum_2_0
                    13 =>"001111000010000000011001", -- app_br_1
                    14 =>"010000000010000000011000", -- app_br_0
                    15 =>"010001000000000000001111", -- app_bi_1
                    16 =>"010010000000000000001110", -- app_bi_0
                    17 =>"010011000000000000001111", -- app_ar_1
                    18 =>"010100000000000000001110", -- app_ar_0
                    19 =>"100000000000000000000001", -- app_ai_1
                    20 =>"100000000000000000000001", -- app_ai_0
                    OTHERS=>"000000000000000000000000"
                    );
begin
    process(ADDR)
    begin
        Data_out <= ctrl_data(to_integer(unsigned(ADDR)));
    end process;
end behavior;
```

## 5.2 Test Bench

### 5.2.1 Modalità singola

```
LIBRARY IEEE;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
USE STD.TEXTIO.ALL;

ENTITY Butterfly_TestBench IS

END Butterfly_TestBench;

ARCHITECTURE structure OF Butterfly_TestBench IS

COMPONENT butterfly is
port(DATA_IN: IN SIGNED(15 DOWNTO 0);
```

```
            START,CLK,RESETN : IN STD_LOGIC;
            DATA_OUT: OUT SIGNED(15 DOWNTO 0);
            DONE : OUT STD_LOGIC
           );
END COMPONENT;

SIGNAL DATA_IN: SIGNED (15 DOWNTO 0) := (Others => '0');
SIGNAL DATA_OUT: SIGNED (15 DOWNTO 0);
SIGNAL DONE, CLK, RESETN, START, prova: STD_LOGIC;


BEGIN

CLKSYST: PROCESS
BEGIN
CLK <= '0';
WAIT FOR 50 us;
CLK <= '1';
WAIT FOR 50 us;
END PROCESS;

Reset: PROCESS
BEGIN
RESETN <= '0';
WAIT FOR 20 us;
RESETN <= '1';
WAIT;
END PROCESS;

LETTURADAIIN: PROCESS(CLK,RESETN)

FILE FILE_OPENED: TEXT IS IN "DATIIN.dat";
VARIABLE LINE_READ: LINE;
VARIABLE DAT : integer;
VARIABLE CONT:INTEGER;
begin

          IF RESETN='0' THEN
                    START<='0';
                    CONT:=0;
          ELSIF CLK' EVENT AND CLK='1' THEN
                    CONT:=CONT+1;
                    if cont = 1 then
                              START<='1';
                    end if;
                    if cont>12 then
                              cont:=0;
```

```
                    else
                    IF CONT>1 AND CONT<8 THEN
                              IF endfile( FILE_OPENED ) then
                              ELSE
                               READLINE( FILE_OPENED, LINE_READ);
                               READ( LINE_READ, DAT );
                               DATA_IN <= to_signed(DAT ,16);
                    END IF;
                    IF CONT>=7 THEN
                              START<='0';
                    END IF;
                    END IF;
          END IF;
END IF;

END PROCESS;

BT: butterfly PORT MAP(CLK => CLK, RESETN => RESETN,
DATA_IN => DATA_IN,
DATA_OUT => DATA_OUT, START => START, DONE => DONE);
END STRUCTURE;
```

### 5.2.2 Modalità continua

```
LIBRARY IEEE;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
USE STD.TEXTIO.ALL;

ENTITY ButterflyContinua_TestBench IS

END ButterflyContinua_TestBench;

ARCHITECTURE structure OF ButterflyContinua_TestBench IS

COMPONENT butterfly is
 port(DATA_IN: IN SIGNED(15 DOWNTO 0);
           START,CLK,RESETN : IN STD_LOGIC;
           DATA_OUT: OUT SIGNED(15 DOWNTO 0);
           DONE : OUT STD_LOGIC
          );
END COMPONENT;

SIGNAL DATA_IN, OUT_TO_IN, DATA_OUT, Checker: SIGNED (15 DOWNTO 0);
SIGNAL DONE1, DONE2, CLK, RESETN,  START: STD_LOGIC;
```

```
BEGIN

CLKSYST: PROCESS
BEGIN
CLK <= '0';
WAIT FOR 50 us;
CLK <= '1';
WAIT FOR 50 us;
END PROCESS;

Reset: PROCESS
BEGIN
RESETN <= '0';
WAIT FOR 20 us;
RESETN <= '1';
WAIT;
END PROCESS;

LETTURADAIIN: PROCESS(CLK,RESETN)

FILE FILE_OPENED: TEXT IS IN "DATIIN.dat";
VARIABLE LINE_READ: LINE;
VARIABLE DAT : integer;
VARIABLE CONT:INTEGER;

begin

IF RESETN='0' THEN
        START<='0';
        CONT:=0;
ELSIF CLK' EVENT AND CLK='1' THEN
        CONT:=CONT+1;
        if cont = 1 then
          START<='1';
        end if;
        if cont>12 then
          cont:=0
        else
                IF CONT>1 AND CONT<8 THEN
                        IF endfile( FILE_OPENED ) then
                        ELSE
                                READLINE( FILE_OPENED, LINE_READ);
                                READ( LINE_READ, DAT );
                                DATA_IN <= to_signed(DAT ,16);
                                END IF;
                                IF CONT>=7 THEN
                                        START<='0';
```

```
                                    END IF;
                                END IF;
                            END IF;
END IF;

END PROCESS;

BT1: butterfly PORT MAP(CLK => CLK, RESETN => RESETN, DATA_IN =>
DATA_IN, DATA_OUT => OUT_TO_IN, START => START, DONE => DONE1);
BT2: butterfly PORT MAP(CLK => CLK, RESETN => RESETN, DATA_IN =>
OUT_TO_IN, DATA_OUT => DATA_OUT, START => DONE1, DONE => DONE2);

END STRUCTURE;
```

# Capitolo 6

# Codice MatLab

## 6.1 Script

```
% Test Butterlfy versione 4

close all
clear all
clc

text1 = fopen('Data_In.txt', 'w');
text2 = fopen('Data_Out.txt', 'w');

format long

%%%%% Input %%%%%
Nbit_Input = 16;
Bit_Guardia = 2; % Numero di bit di guarda che si desidera inserire
Nbit = Nbit_Input - Bit_Guardia;
M = 1; % Numeri di codici che si desidera generare

%%%% Generazione Input %%%%
MaxValue = 2^(Nbit) - 1;

for a=1:M
%Wr
DataGeneratorWr = randi([0, MaxValue], 1, 1);
InDecimale_Wr = (DataGeneratorWr - 2^(Nbit_Input-1-Bit_Guardia));
InFrazionario_Wr = InDecimale_Wr/2^15
%Wi
DataGeneratorWi = randi([0, MaxValue], 1, 1);
InDecimale_Wi = (DataGeneratorWi - 2^(Nbit_Input-1-Bit_Guardia));
InFrazionario_Wi = InDecimale_Wi/2^15
%Ar
```

```
DataGeneratorAr = randi([0, MaxValue], 1, 1);
InDecimale_Ar = (DataGeneratorAr - 2^(Nbit_Input-1-Bit_Guardia));
InFrazionario_Ar = InDecimale_Ar/2^15
%Ai
DataGeneratorAi = randi([0, MaxValue], 1, 1);
InDecimale_Ai = (DataGeneratorAi - 2^(Nbit_Input-1-Bit_Guardia));
InFrazionario_Ai = InDecimale_Ai/2^15
%Br
DataGeneratorBr = randi([0, MaxValue], 1, 1);
InDecimale_Br = (DataGeneratorBr - 2^(Nbit_Input-1-Bit_Guardia));
InFrazionario_Br = InDecimale_Br/2^15
%Bi
DataGeneratorBi = randi([0, MaxValue], 1, 1);
InDecimale_Bi = (DataGeneratorBi - 2^(Nbit_Input-1-Bit_Guardia));
InFrazionario_Bi = InDecimale_Bi/2^15

end

InBinario_Br(a,:) = dec2bin_n(InDecimale_Br, Nbit_Input);
InBinario_Bi(a,:) = dec2bin_n(InDecimale_Bi, Nbit_Input);
InBinario_Ar(a,:) = dec2bin_n(InDecimale_Ar, Nbit_Input);
InBinario_Ai(a,:) = dec2bin_n(InDecimale_Ai, Nbit_Input);
InBinario_Wr(a,:) = dec2bin_n(InDecimale_Wr, Nbit_Input);
InBinario_Wi(a,:) = dec2bin_n(InDecimale_Wi, Nbit_Input);

M1 = InFrazioario_Wr * InFrazionario_Br;
S1 = InFrazionario_Ar - M1;

[Ar_o, Ai_o, Br_o, Bi_o] = Butterfly(InFrazionario_Wr,
InFrazionario_Wi, InFrazionario_Ar, InFrazionario_Ai,
InFrazionario_Br, InFrazionario_Bi)
```

## 6.2 Function

```
function [Ar_O, Ai_O, Br_O, Bi_O] = Butterfly(Wr, Wi, Ar, Ai, Br, Bi)
    Br_O = Ar - Br.*Wr + Bi.*Wi;
    Bi_O = Ai - Br.*Wi - Bi.*Wr;
    Ar_O = 2*Ar - Br_O;
    Ai_O = 2*Ai - Bi_O;
end
```