22 gennaio 2018

Esercitazione finale di progetto sulle architetture integrate

Sistemi Digitali Integrati

Prof. M. Zamboni

M. Montagna, A. Malacrino, V. Emanuele

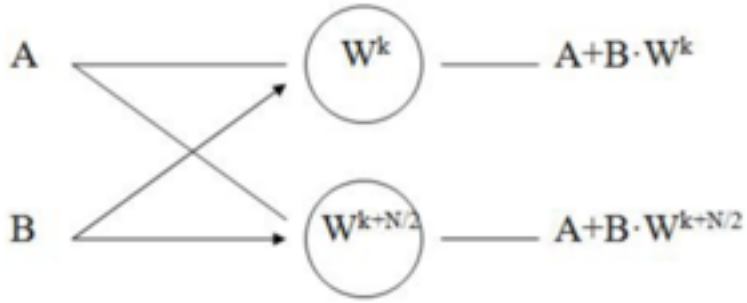
Anno Accademico 2017-2018

Contenuti

1. Introduzione Butterfly

L'obiettivo di questo progetto è quello di realizzare un'unità di elaborazione che esegua la Butterfly, un elemento base per la realizzazione di una FFT classica. Per FFT si intende la Fast Fourier Transform che è un algoritmo ottimizzato per calcolare la trasformata discreta di Fourier e la sua inversa. L'algoritmo più diffuso per la risoluzione della FFT è l'algoritmo di Cooley-Tukey, che consiste nello spezzare ricorsoivamentre una DFT di una qualsiasi dimensione in DFT più piccole.

La Butterfly viene eseguita tramite moltiplicazioni e somme sui numeri complessi A, B e W^k sulla base dello schema riportato in seguito:



Le operazione da eseguire con la Butterfly sono:

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

Dove A = Ar+ j Ai, B = Br + jBi e W^k = Wr + jWi. Supponiamo che A, B e W^k siano definititi in forma frazionaria (sia la parte reale che quella immaginaria) cioè il dato è strettamente compreso tra -1 e 1; lavoreremo in complemento a 2 (C2) su 24 bit tramite il metodo "Uncoditional Block Floating Scaling" che non prevede l'uso di bit di guardia in ingresso per evitare l'overflow. Lavoriamo quindi, come da specifiche, alla massima precisione per tutto l'algoritmo, ma ,al termine del processo, effettuiamo un'operazione di rounding e uno scalamento di due posizioni per poter rientrare nell'intervallo -1, +1 ed essere compatibili quindi con lo stadio successivo.

Il vantaggio di questo metodo rispetto all'inserimento dei bit di guardia è che con i guard bits riduco la dinamica di ingresso poiché questi non vengono inseriti in aggiunta, ma prima di iniziare; in questo modo posso evitare il fenomeno di overflow, ma perdo poi nella dinamica dei dati. Con il metodo che abbiamo utilizzato invece abbiamo un range maggiore, nonostante si perda un po' in precisione.

2. Progetto Butterfly

Il primo passo per ogni progetto è stabilire quali siano le specifiche da seguire: in questo caso particolare non abbiamo particolari condizioni su velocità del processo, performance, dimensioni fisiche, etc. L'unica restrizione è che abbiamo a disposizione un moltiplicatore e due sommatori.

I segnali che arrivano dall'esterno della nostra macchina sono il segnale di RESET asincrono, il segnale di START, il CLOCK e gli ingressi; in uscita invece avremo i dati in uscita e il segnale di DONE che ci avverte che il primo dato uscirà l colpo di clock dopo il suo invio e gli altri dati usciranno in serie ogni step.

2.1 DFD e scheduling

Poiché abbiamo molti gradi di libertà, per questa macchina le soluzioni saranno molteplici; il nostro obiettivo è quello di esplorare questo spazio delle soluzioni per poter trovare quella che ottimizzi il nostro sistema in base alle nostre scelte di progetto. Uno dei punti fondamentali per l'evoluzione del nostro lavoro è quello di derivare un'architettura ottimizzata, ma per farlo abbiamo prima di tutto bisogno di elaborare un data flow diagram (DFD) che ci permetta di modellizzare il nostro sistema e, anche senza un segnale di temporizzazione, osservare le data dependencies. È perciò fondamentale l'ordine e lo scheduling con il quale effettuiamo le nostre operazioni: in questo modo riusciamo a stabilire delle relazioni temporali nel nostro progetto.

Il DFD può facilmente essere disegnato se si decidono inizialmente le operazioni necessarie per lo svolgimento dell'algoritmo e il loro ordine. Calcoliamo prima di tutto A' e B' che saranno i risultati da fornire in uscita dalla Butterfly:

$$B' = (Ar + jAi) + (-Wr - jWi)*(Br + Bi) = (Ar - BrWr + BiWi) + j(Ai - BrWi - BiWr);$$

trovato $B' = B'r + jB'i$ posso poi trovare $A' = A'r + jA'i$ in questo modo:

$$A' = -B'r + 2Ar + j(2Ai - B'i).$$

A questo punto possiamo definire tutti i nostri operatori:

M1 = Br*Wr;

M2 = Bi*Wi;

M3 = Br*Wi;

M4 = Bi*Wr;

∑1 = Ar - M1;

∑2 = ∑1 + M2 = B'r;

∑3 = Ai - M4;

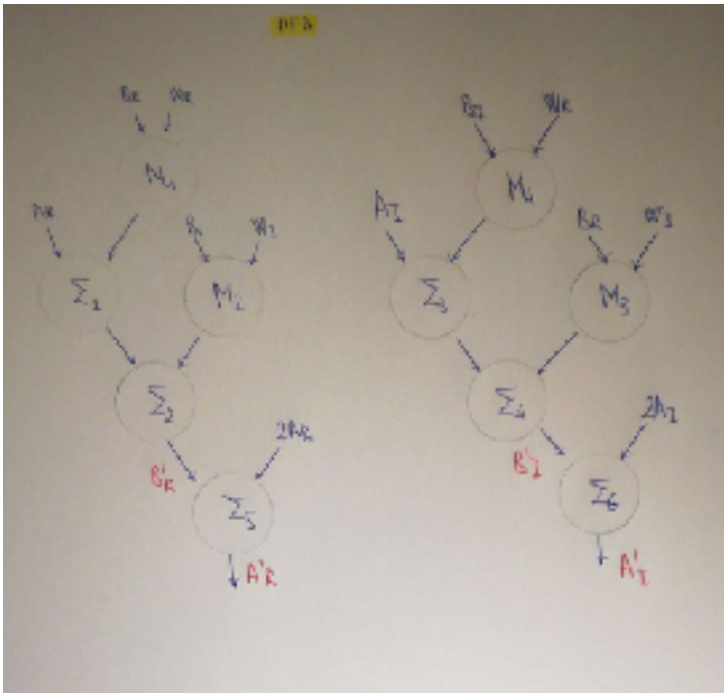
∑4 = ∑3 - M3 = B'i;

∑5 = 2Ar - B'r = A'r;

∑6 = 2Ai - B'i = A'i;

(Il numero delle operazioni non è legato all'ordine in cui verrano eseguite)

Riportiamo in seguito il DFD dell'algoritmo:



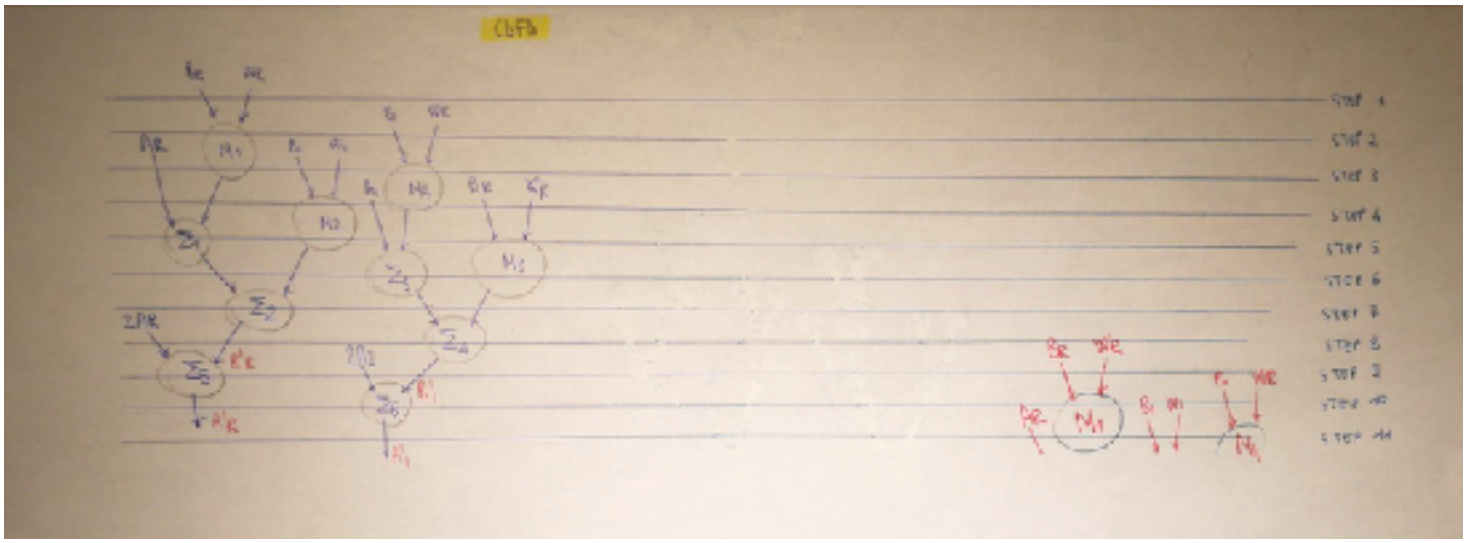
Vediamo che in questa rappresentazione non c'è nessun riferimento al clock, ma solamente al funzionamento della macchina e il flusso delle nostre operazioni.

Per le moltiplicazioni *2 utilizziamo un blocco combinatorio esterno che agisca come uno shifter aritmetico (aggiungiamo uno '0' a destra del numero).

Il moltiplicatore ha due livelli di pipeline, mentre il sommatore ne ha uno; i livelli di pipeline sono da tenere in considerazione quando si rappresenta il DFG in quanto sono una parte importante del timing della mia macchina. Con due livelli di pipeline (moliplicatore) ad ogni colpo di clock posso iniziare una nuova operazione, ma il risultato sarà disponibile non

nel colpo di clock successivo, ma nel colpo di clock ancora dopo.
In rosso sono rappresentate le uscite della Butterfly. In questa rappresentazione temporale non è presente il colpo di clock in cui avviene l'approssimazione dell'ultimo dato, qui abbiamo solamente rappresentato il nostro algoritmo.

In questi passi iniziali del progetto abbiamo dovuto scegliere le caratteristiche principali del nostro sistema: non essendoci delle specifiche particolari abbiamo deciso di optare per una soluzione a basso costo ed ad area ridotta, sacrificando un po' la velocità nella modalità in cui la Butterfly è utilizzata in continua - la butterfly è posizionata in cascata insieme ad altre butterfly e lavora senza intervalli, prendendo appena può i dati in ingresso ed iniziandoli ad elaborare. Questa scelta ci ha permesso di risparmiare un sommatore ed alcuni registri come vedremo successivamente. Riportiamo in seguito la nostra soluzione rappresentata tramite il CDFG:



Ogni step rappresenta un colpo di clock, il moltiplicatore essendo pipelinato due volte occupa due step, mentre il sommatore solo uno. A sinistra abbiamo il funzionamento normale della nostra macchina (one-shot), a destra invece sono rappresentate le operazioni da svolgere se stessi lavorando in modalità continua, in modo da non lasciare tempi morti tra un ciclo e l'altro. Ar e Ai entrano un colpo di clock prima rispetto a quando sono realmente necessarie per fare in modo che la dinamica di ingresso dei dati sia uguale alla dinamica d'uscita.

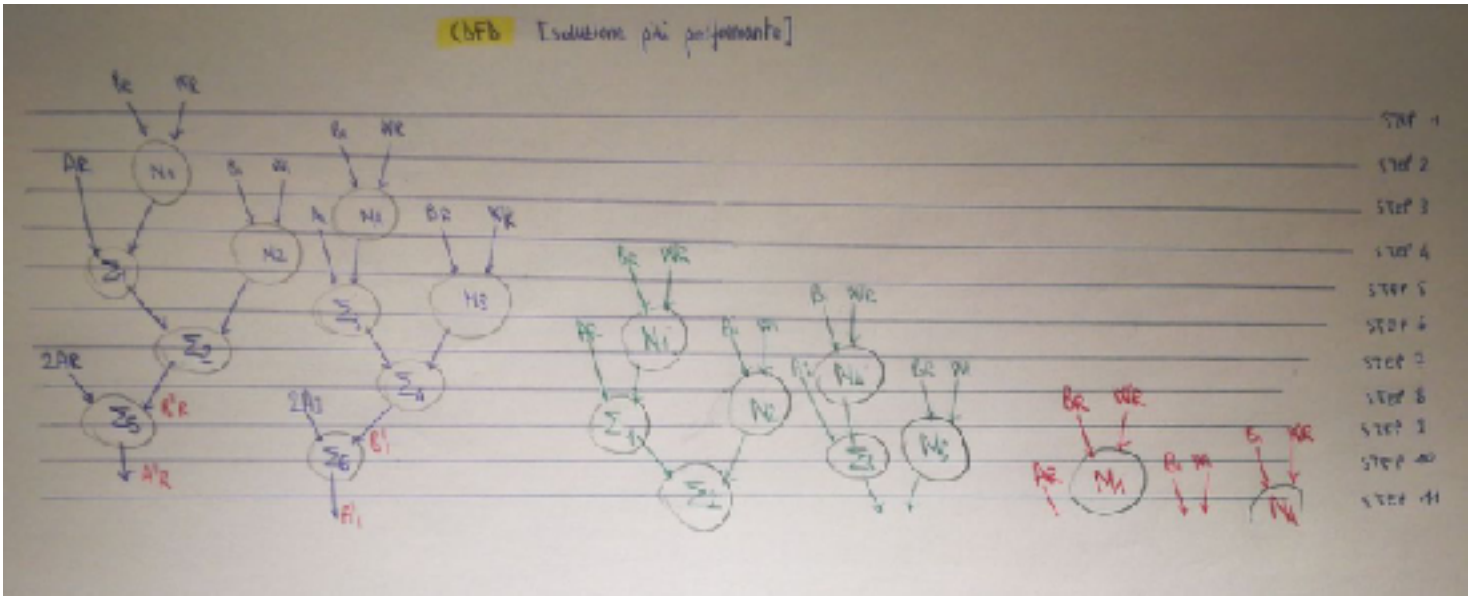
Per quanto riguarda il funzionamento in modalità continua sappiamo che non sono presenti segnali esterni che ci indichino in che situazione stiamo lavorando perciò al settimo colpo di clock andremo a fare una verifica sul segnale START per vedere se sta iniziando un

nuovo ciclo di operazioni - lavoreremo quindi in modalità continua - oppure la macchina sta lavorando una volta sola (modalità one-shot).

La macchina dopo essere stata resettata - resettiamo il registro del μAR perciò il nostro primo indirizzo è quello di reset - entra in uno stato di IDLE, dove aspetta l'arrivo di un segnale di START dall'esterno; quando viene campionato, il colpo di clock successivo vengono inviati i primi dati (Br e Wr) e vengono abilitati i relativi registri all'interno del RF per salvare l'informazione. Da questo punto in avanti i passi del processo sono tutti rappresentati sul CDFG.

La soluzione che abbiamo adottato per lo scheduling è l'ALAP (as late as possible) che consiste nell'aspettare di eseguire l'operazione in questione fino a quando non è strettamente necessario. Come possiamo vedere nell'immagine successiva la Butterfly che abbiamo sviluppato utilizza solamente un sommatore e un moltiplicatore, riducendo le dimensione ed i costi del nostro dispositivo; a partire dal segnale esterno di START la macchina impiega 13 colpi di clock a far uscire l'ultimo dato approssimato (quindi a concludere completamente un ciclo di calcoli, ma le operazioni da sole sono 11 step). La dinamica dei dati è la stessa sia in ingresso che in uscita: i dati delle operazioni entrano ed escono dalla macchina in serie, ogni colpo di clock; prima Br e Bi e dopo Ar e Ai.

Riportiamo in seguito il CDFD della soluzione più performante, in modo da poter vedere le differenze che verranno man mano spiegate all'interno del report:



Per quanto riguarda le approssimazioni per rispettare le specifiche abbiamo deciso di operare in questo modo: dovendo effettuare una moltiplicazione e due somme il parallelismo dei dati diventa da 24 bit iniziali, a 49 bit in uscita; l'obiettivo è quello di tornare al parallelismo di ingresso, in modo da poter riutilizzare i dati in una butterfly uguale successiva. Il numero che ci troviamo ora in uscita potrebbe essere compreso tra 2 e 3 in unità decimali (minimo tra -2 e -3); siamo sempre in fixed point, ma la codifica non è più Q1.23, ma bensì Q3.46. Le specifiche ci richiedono di utilizzare il metodo di round half-up: si arrotonda quindi all'infinito positivo; in hardware si tratta di sommare metà LSB (ovvero sommare 1 bit immediatamente dopo quello si vuole troncare, in questo caso si somma 2^-24). Il primo vantaggio di questa soluzione è che l'errore massimo viene dimezzato (0.5 LSB dopo il troncamento); il bias non è 0, quindi questa tecnica è leggermente sbilanciata verso il positivo. Arrivati a questo punto è necessario ancora scalare il risultato di due posizioni in modo da rientrare nella dinamica -1 +1; ad ogni ciclo di butterfly quindi lo scale factor viene moltiplicato per 4: SF = SF precedente * 4. Per trattare i numeri all'interno della nostra Butterfly abbiamo utilizzato la libreria signed, quindi il VHDL lavora con numeri interi, siamo noi che, sapendo dove si trova la virgola, li interpretiamo in un certo modo.
Fondamentale in questo passaggio è quello di definire correttamente il parallelismo dei dati in maniera da poi riuscire ad ottimizzare anche i bus utilizzati(lo vedremo successivamente).

2.2 Tempo di vita variabili

Il passo successivo è quello di ottimizzare il datapath riducendo al minimo il numero di variabili temporanee senza introdurre nuovi step algoritmici; per farlo utilizziamo il timelife diagram dei registri: facciamo un diagramma temporale dove riportiamo per ogni registro per quanto tempo questo deve mantenere memorizzata l'informazione.

Riportiamo in seguito la tabella della soluzione da noi adottata:

Operazioni	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8	Step 9	Step 10	Step 11
Br	X	X	X	X					O	O	O
Bi		X	X							O	O
Ar			X	X	X	X	X	X			O
Ai				X	X	X	X	X	X		
Wr	X	X							O	O	
Wi			X	X							O
M1				X							
M2						X					
M3							X				
M4					X						
∑1						X					
∑2								X			
∑3							X				
∑4									X		
∑5										X	
∑6											X

Le X rappresentano le variabili in modalità one-shot, mentre i O rappresentano i registri occupati durante l'esecuzione in modalità continua, sono quindi i dati dell'elaborazione successiva (in modalità continua bisogna perciò tenere in considerazione sia le X che i O che rappresentano un nuovo ciclo contemporaneo a quello in elaborazione). Come possiamo vedere il numero massimo di registri occupati contemporaneamente è 5, ma

per una gestione più semplice del datapath e dei bus abbiamo deciso di inserire un registro aggiuntivo all'uscita del sommatore, in modo da non dover tornare all'interno del RF per salvare la somma; in questo modo abbiamo risparmiato un bus globale ed alcuni multiplexer rendendo così meno complesso il controllo della macchina. Dovremmo poi considerare il parallelismo dei dati in ingresso ai registri che non sarà lo stesso. Se spostassimo ad un colpo di clock prima il check dello start avremmo dei problemi nel riutilizzo dei registri all'interno del RF a meno di non cambiare l'algoritmo (nel momento in cui avevamo bisogno di utilizzare Ai cambiavamo il dato nel registro inserendo il nuovo Bi).
In modalità continua, dopo l'undicesimo colpo di clock non salto più allo step 1, ma bensì allo step 5, in modo da non lasciare tempi morti tra un ciclo e quello successivo.

Per poter apprezzare maggiormente i vantaggi in termini di costi e dimensioni della nostra scelta progettuale riportiamo in seguito il timelife diagram della soluzione più prestante, che impiega però un sommatore e 3 registri in più rispetto alla nostra scelta progettuale (quattro in realtà se tenessimo solamente conto del timelife diagram):

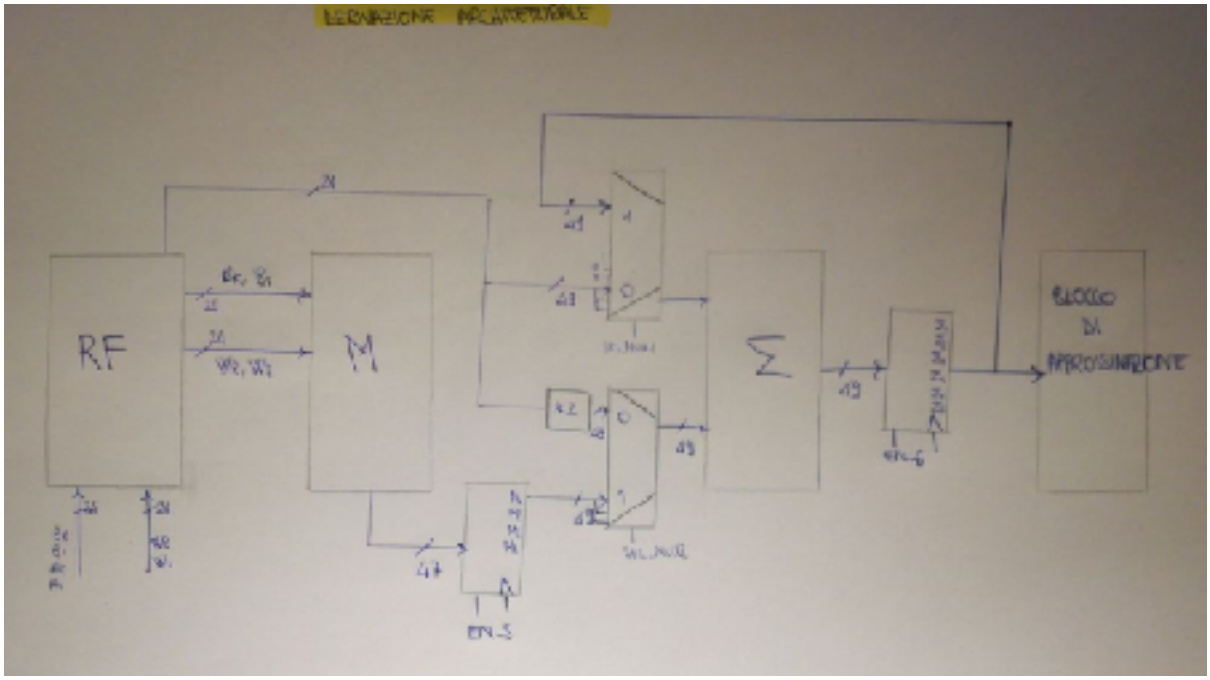
Operazioni	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8	Step 9	Step 10	Step 11
Br	X	X	X	X	O	O	O	O	Z	Z	Z
Bi		X	X			O	O			Z	Z
Ar			X	X	X	X	X O	X O	O	O	O Z
Ai				X	X	X	X	X O	X O	O	O
Wr	X	X			O	O			Z	Z	
Wi			X	X			O	O			Z
M1				X				O			
M2						X				O	
M3							X				O
M4					X				O		
∑1						X				O	
∑2								X			
∑3							X				O
∑4									X		
∑5										X	
∑6											X

Questa configurazione, per quanto riguarda la modalità one-shot, è identica alla nostra in termini di prestazioni, ma è in modalità continua che si vedono molto le differenze. Questo progetto in modalità continua impiega due sommatori e 9 registri, ma finito l'undicesimo step riprende dall'ottavo, risparmiando quattro colpi di clock ogni elaborazione in continua rispetto alla nostra soluzione. I dati in ingresso entrano in serie, non ci sono pause tra un'elaborazione e l'altra; in questo modo viene massimizzato il throughput, lasciano invariata la latenza.

Come spiegato precedentemente entrambi i progetti hanno dei vantaggi e degli svantaggi. Noi abbiamo scelto di agire in questo modo in quanto, in assenza di specifiche riguardo alle prestazioni, abbiamo pensato che fosse meglio risparmiare in costi e area; un fattore che avrebbe potuto influenzare molto la scelta sarebbe stato ad esempio sapere in quale modalità venisse maggiormente utilizzata questa Butterfly (one-shot oppure continua). Volendo si potrebbe anche trovare il compromesso più adatto alla situazione spostando l'inizio dell'elaborazione in continua.

Un altro vantaggio in termini di costi riguarda l'ottimizzazione dei bus e la dimensione della ROM del sequenziatore e del generatore di comandi, ma ne parleremo successivamente.

2.3 Derivazione architettura e ottimizzazione dei bus

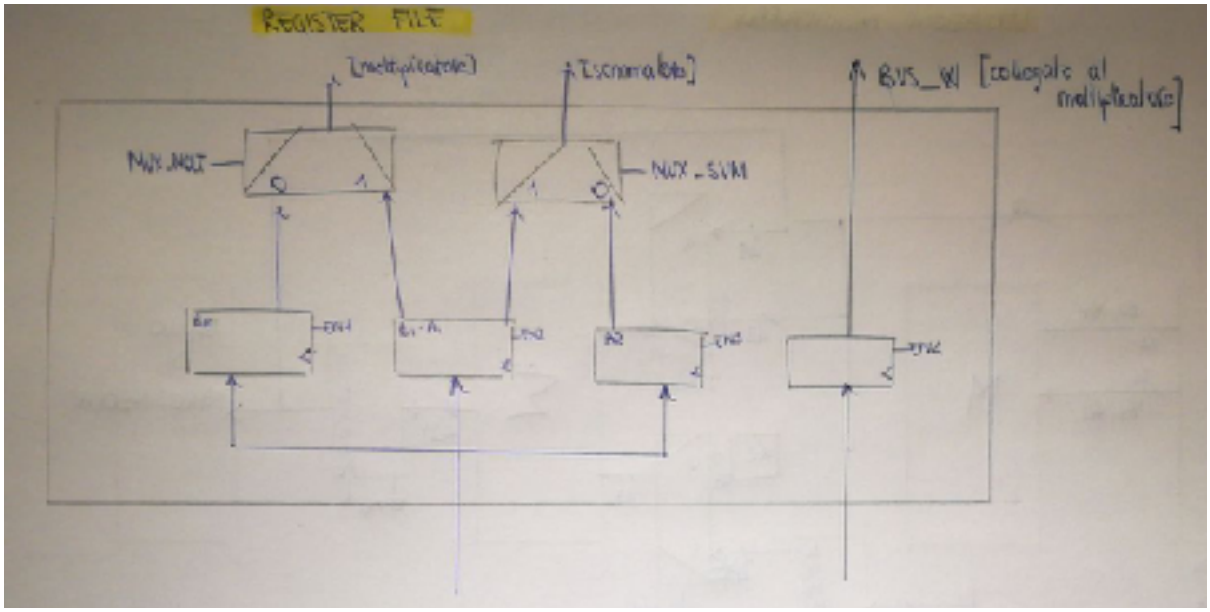


Arrivati a questo punto dobbiamo derivare l'architettura della nostra macchina; la butterfly che stiamo progettando è composta da quattro blocchi base; il primo di questi è il register file (RF) all'interno del quale abbiamo 4 registri e rappresenta il nostro sistema di memoria delle variabili principali ricevute dall'esterno. Dal RF verrano inviati i dati a moltiplicatore e sommatore (altri due blocchi della nostra architettura) attraverso dei bus locali che collegano direttamente i blocchi. Per come abbiamo pensato la macchina non necessitiamo di alcun bus globale; con bus globale si intende un sistema di interconnessione tra due o più elementi mentre il bus locale collega direttamente due blocchi. Se diminuissimo il numero di bus locali uscenti dal RF e mettessimo al posto un bus globale collegando sommatore e moltiplicatore, avremmo un calo delle prestazioni (un colpo di clock in più in one-shot e due in modalità continua).

Gli altri due registri esterni al RF sono registri in cui salvo le variabili temporanee e continuo a fare operazioni su di esse. Ad esempio l'uscita del moltiplicatore andrà sempre verso il sommatore, per questo motivo abbiamo deciso di inserire un registro e effettuare un collegamento diretto, in modo da non dover mandare i dati al RF e poi recuperarli per riutilizzarli nel sommatore. In questo modo abbiamo un vantaggio in termini di esecuzione. Stesso discorso per i dati che, in uscita dal sommatore, entrano all'interno del blocco combinatorio dell'approssimazione e in loop nel sommatore stesso.

Oltretutto i bus locali costano molto meno perché hanno un solo driver e non hanno nessun bisogno di controlli per l'abilitazione. Nel progetto più performante, essendoci più blocchi in gioco avremmo avuto bisogno di più bus globali rispetto a questo progetto.

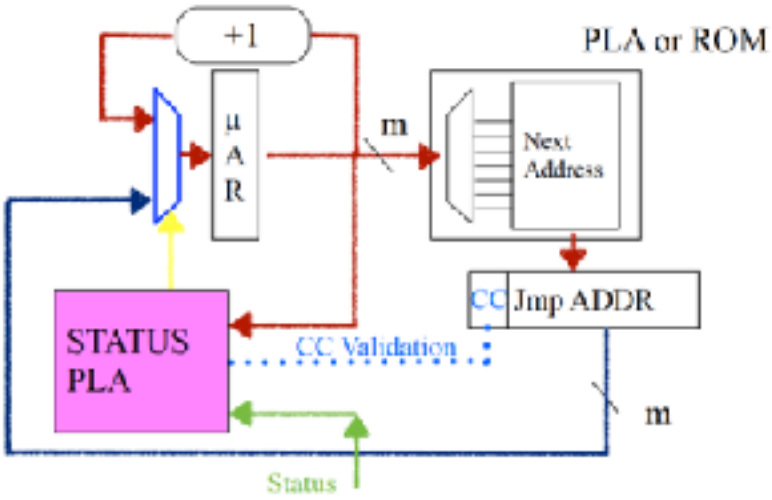
Per quanto concerne il parallelismo dei bus abbiamo lavorato in questo modo: in ingresso al RF entreranno i dati su 24 bit e usciranno su 24 bit; per il moltiplicatore i dati entrano su 24 bit ed escono su 47 bit, in quanto la moltiplicazione tra due numeri in fractional point non darà mai overflow; la moltiplicazione tra due numeri su N bit può quindi essere rappresentata su 2N-1. In ingresso al sommatore avremmo 49 bit, quindi è necessario aumentare il parallelismo dei dati in ingresso ai multiplexer :come possiamo vedere nel codice VHDL abbiamo semplicemente aggiunto dei bit di segno oppure degli zeri a sinistra del numero in base alla necessità; in uscita dal sommatore abbiamo perciò 49 bit, il bus della somma porterà il dato in loop in ingresso nuovamente al sommatore e al blocco di approssimazione che, con le tecniche spiegate precedentemente, riporta il parallelismo da 49 bit a 24 bit, in modo da favorire una nuova elaborazione.



Descriviamo brevemente i blocchi del nostro datapath: il RF è composto da quattro registri e due multiplexer, un registro mi servirà per salvare Br, nel secondo salverò sia Bi che Ai (come possiamo vedere dalla tabelle riguardante tempo di vita delle variabili queste due non saranno mai salvate contemporaneamente), nel terzo registro salviamo Ar mentre nel quarto i fattori moltiplicativi Wr e Wi. I due multiplexer gestiranno Br e Bi per quanto riguarda l'ingresso al moltiplicatore; Ar e Ai per l'ingresso al sommatore.

Il mio moltiplicatore è descritto in maniera comportamentale nel VHDL, la particolarità è che contiene due registri dopo le operazioni combinatorie in modo da risultare pipelinato.
Il sommatore, come il moltiplicatore, è descritto in maniera comportamentale e contiene un registro di pipelining e ha, oltre al segnale sum_sub_n, un segnale per dire se si deve effettuare la sottrazione del primo dato meno il secondo o il contrario.

2.4 Progettazione unità di controllo



Teoricamente la parte legate alla derivazione del datapath è terminata, ora dobbiamo passare all'unità di controllo. Il blocco della control unit può essere diviso in due sezioni: la prima è il sequenziatore che ha il compito di definire lo stato futuro in base allo stato attuale ed agli ingressi; la seconda è il command generator che si occupa di generare le istruzioni da eseguire in base allo stato in cui si trova la macchina.

Definiamo la CU mediante la tecnica della microprogrammazione utilizzando un sequenzatore con indirizzamento implicito, tramite la tecnica del conditional sequencing pla, in particolare abbiamo utilizzato quello a due vie per la gestione degli indirizzi di salto in quanto, completata la FSM del mio progetto, vediamo che i salti effettuati dalla nostra macchina sono sempre tra massimo due stati; quando la macchina si trova a dover decidere se effettuare un salto oppure no. Dovrà decidere in base agli ingressi "status" (che sarebbe il segnale di START esterno) se proseguire in sequenza oppure saltare nello stato descritto nella ROM.

All'interno della ROM sono salvati gli indirizzi dei salti in base al contenuto del μAR che contiene lo stato presente, in uscita dalla ROM invece abbiamo l'indirizzo di salto e il CC validation che è un ingresso della STATUS PLA; quest'ultima in base al codice di validazione e ai segnali di stato pilota il MUX per decidere se saltare oppure proseguire in sequenza. La ROM sarà μprogrammata con 0 e 1, la dimensione sarà legata al numero degli stati. La STATUS PLA conterrà il risultato di un'elaborazione attraverso la quale deciderà se saltare oppure no (in VHDL abbiamo deciso di implementarla con un process e l'algoritmo del when

- case). I segnali di stato è il segnale di START; all'interno della status pla entra anche il valore dell'indirizzo corrente. Volendo il contenuto della ROM potrebbe essere scritto su un file e poi caricato all'accensione della macchina, noi abbiamo deciso di optare per una ROM costante pre-caricata. Il µAR possiede anche un segnale di RESET proveniente dall'esterno che è quello che mi permette di partire con il funzionamento normale della macchina.

L'uscita del µAR entra, oltre che nella µROM contenente gli indirizzi di salto, anche nella µROM del command generator. Il command generator è quel blocco che, dato in ingresso l'indirizzo dello stato attuale, restituisce in uscita le istruzioni da eseguire; l'uscita sarà salvata nel µIR. È necessario fare attenzione al campionamento dello stato in uscita dalla µROM del sequenziatore: una soluzione sarebbe potuta essere quella di campionare gli stati sul fronte di discesa in maniera che la status pla scegliesse correttamente per lo stato attuale e non quello futuro (se non si facesse così ci potrebbero essere problemi nel timing e la macchina potrebbe non funzionare correttamente). Noi per ovviare a questo problema abbiamo deciso di inserire lo stato di salto e il CC validation allo stato precedente rispetto a quello in cui avremmo dovuto saltare, in questo modo la macchina riesce a funzionare correttamente per quanto riguarda l'evoluzione degli stati.

Il blocco command generator è una memoria ROM che dovrà essere µprogrammata in maniera che possa interagire correttamente con il modo esterno. In questo frangente possiamo notare un'altro risparmio di costi e dimensioni dovuti alla nostra scelta progettuale: il parallelismo della nostra ROM è 18 mentre, se avessimo adottato la soluzione più prestante, avremmo avuto come minimo i bit di controllo dei registri e del sommatore in più, aumentando la dimensione di questa ROM.

Per quanto riguarda l'interpretazione dei dati del µIR abbiamo deciso di optare per una codifica diretta in quanto è più veloce e, per un numero così limitato di comandi (18), è poco produttivo implementare un sistema di codifica e decodifica.

L'ordine dei bit dei comandi del µIR è riportato e commentato all'interno del VHDL, in modo da capire quale bit equivale a quale comando.

3. Codice VHDL

Dopo aver progettato la butterfly nel suo insieme possiamo ora passare alla sua descrizione in VHDL; abbiamo descritto i component in maniera comportamentale.
Riportiamo in seguito l VHDL del lavoro svolto:

```
-- butterfly
-- progetto butterfly 2017/2018
-- corso sistemi digitali integrati, prof. Zamboni
-- Montagna Marco, Malacrino Andrea, Valpreda Emanuele

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

entity butterfly is
  port (
                    START, CLOCK, RESET: in std_logic;
                    input_btf_1, input_btf_2: in signed (23 downto 0); -
                    DONE: out std_logic;
                    output_btf: out signed (23 downto 0));
end butterfly;

architecture behavior of butterfly is

component control_unit
  port (
                    clk, reset, start: in std_logic;
                    verso_sommatore, done, reset_datapath, sum_sub_n, mux1, mux2,
                    reg_molt1, reg_molt2, reg_sum, en1, en2, en3, en4, en5, en6,
                    en_reg_approx, mux_RF_sum, mux_RF_molt: out std_logic);
end component;

component datapath_butterfly
  port (
                    clk, verso_sommatore, done,
                    reset_datapath, sum_sub_n, mux1, mux2: in std_logic;
                    reg_molt1, reg_molt2, reg_sum: in std_logic;
                    en1, en2, en3, en4, en5, en6: in std_logic;
                    en_reg_approx, mux_RF_sum, mux_RF_molt : in std_logic;
                    B_A, W: in signed (23 downto 0);
                    uscita: out  signed (23 downto 0));
end component;

signal verso_sommatore_btf, done_datapath_btf, reset_datapath_btf,
          sum_sub_n_btf, mux1_btf, mux2_btf, reg_molt1_btf, reg_molt2_btf,
          reg_sum_btf, en1_btf, en2_btf, en3_btf, en4_btf, en5_btf, en6_btf,
          en_reg_approx_btf, mux_RF_sum_btf, mux_RF_molt_btf: std_logic;
signal uscita_btf: signed (23 downto 0);
```

```
begin

CU: control_unit
  PORT MAP
        (CLOCK, RESET, START,
        verso_sommatore_btf, done_datapath_btf, reset_datapath_btf,
        sum_sub_n_btf, mux1_btf, mux2_btf,
        reg_molt1_btf, reg_molt2_btf, reg_sum_btf,
        en1_btf, en2_btf, en3_btf, en4_btf, en5_btf, en6_btf,
        en_reg_approx_btf, mux_RF_sum_btf, mux_RF_molt_btf);

DONE <= done_datapath_btf;

DP: datapath_butterfly
  PORT MAP
        (CLOCK,
        verso_sommatore_btf, done_datapath_btf, reset_datapath_btf,
        sum_sub_n_btf, mux1_btf, mux2_btf,
        reg_molt1_btf, reg_molt2_btf, reg_sum_btf,
        en1_btf, en2_btf, en3_btf, en4_btf, en5_btf, en6_btf,
        en_reg_approx_btf, mux_RF_sum_btf, mux_RF_molt_btf,
        input_btf_1, input_btf_2, uscita_btf);

output_btf <= uscita_btf;
end behavior;
```

```
-- progetto butterfly 2017/2018
-- corso sistemi digitali integrati, prof. Zamboni
-- Montagna Marco, Malacrino Andrea, Valpreda Emanuele
-- control unit

library ieee;
use ieee.std_logic_1164.all;

entity control_unit is
  port (
                clk, reset, start: in std_logic;
                verso_sommatore, done, reset_datapath, sum_sub_n, mux1, mux2,
                reg_molt1, reg_molt2, reg_sum, en1, en2, en3, en4, en5, en6,
                en_reg_approx, mux_RF_sum, mux_RF_molt: out std_logic);
end control_unit;

architecture behavior of control_unit is

component command_generator
  port (
                clk, reset: in std_logic;
                input_microAR: in std_logic_vector (4 downto 0);
                output_microIR: out std_logic_vector(17 downto 0));
end component;

component conditional_sequential_pla
  port (
                clk, reset, start: in std_logic;
                output_microAR: out std_logic_vector( 4 downto 0));
end component;

signal microAR: std_logic_vector( 4 downto 0);
signal microIR: std_logic_vector( 17 downto 0);

begin
  conditional_pla: conditional_sequential_pla PORT MAP (clk, reset, start, microAR);
  generatore_comandi: command_generator PORT MAP(clk, reset, microAR, microIR);

  verso_sommatore <= microIR(17);
  done <= microIR(16);
  reset_datapath <= microIR(15);
  sum_sub_n <= microIR(14);
  mux1 <= microIR(13);
  mux2 <= microIR(12);
  reg_molt1 <= microIR(11);
  reg_molt2 <= microIR(10);
  reg_sum <= microIR(9);
  en1 <= microIR(8);
  en2 <= microIR(7);
  en3 <= microIR(6);
  en4 <= microIR(5);
  en5 <= microIR(4);
  en6 <= microIR(3);
  en_reg_approx <= microIR(2);
  mux_RF_sum <= microIR(1);
  mux_RF_molt <= microIR(0);

end behavior;
```

```
-- progetto butterfly 2017/2018
-- corso sistemi digitali integrati, prof. Zamboni
-- Montagna Marco, Malacrino Andrea, Valpreda Emanuele
-- conditional sequential pla

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity conditional_sequential_pla is
  port (
                clk, reset, start: in std_logic;
                output_microAR: out std_logic_vector( 4 downto 0));
end conditional_sequential_pla;

architecture behavior of conditional_sequential_pla is

component vector_register
GENERIC (N:INTEGER);
PORT(
EN_REG,CLK,RESET     : IN STD_LOGIC;          --segnale di enable, reset e clock
DATA_REG_IN    :     IN std_logic_vector((N-1) DOWNTO 0);          --dato in ingresso
DATA_REG_OUT :       OUT std_logic_vector((N-1) DOWNTO 0) --dato in uscita
);
END component;

component vector_mux
GENERIC (N:INTEGER);
PORT(
SEL : IN STD_LOGIC;      --segnale di enable, reset e clock
DATA_1, DATA_2          : IN std_logic_vector((N-1) DOWNTO 0);        --dato in ingresso
DATA_OUT : OUT std_logic_vector((N-1) DOWNTO 0)); --dato in uscita

END component;


component ROM_JUMP
port (
                ADDR : in std_logic_vector (4 downto 0); -- address input
                DOUT : out std_logic_vector (5 downto 0)); -- data output
end component;

signal address: std_logic_vector (4 downto 0);
signal CC_validation: std_logic;
signal cc_next_address, cc_next_address_out: std_logic_vector (5 downto 0);
signal jmp_address, seq_address, mux_address: std_logic_vector (4 downto 0);
signal sel_status_pla: std_logic;
signal temp: std_logic;

begin

ROM_salti: ROM_JUMP PORT MAP (address, cc_next_address);
reg_cc_jmp: vector_register
  GENERIC MAP (N=>6)
  PORT MAP ('1', clk, reset, cc_next_address, cc_next_address_out);
microAR: vector_register
  GENERIC MAP (N=>5)
  PORT MAP ('1', clk, reset, mux_address, address);
```

```
	cc_validation <= cc_next_address_out(5);
	jmp_address <= cc_next_address_out(4 downto 0);
	mux_seq_jmp: vector_mux
	  GENERIC MAP (N=>5)
	  PORT MAP (sel_status_pla, seq_address, jmp_address, mux_address);
	output_microAR <= address;

	seq_address <= "00001" + address; -- somma 1 per indirizzo successivo
	-- status pla è un segnale che si abilita in certe condizioni

	status_pla: process(cc_validation, address, start)
	  begin
		if (cc_validation = '1') then
			case address is
				when "00001" =>
					if (start = '0') then
						temp <= '1';
				-- se sono in idle e non ho ricevuto lo start allora salto di nuovo
in idle
					else
						temp <= '0';
				-- se no vado in sequenziale e comincio elaborazione
					end if;
				when "01000" =>
					if (start = '1') then
						temp <= '1';
					else
						temp <= '0';
					end if;
				when "01101" =>
					temp <= '1';
				when "10001" =>
					temp <= '1';
				when others =>
					temp <= '0';
			end case;
		else
			temp <= '0';
		end if;
	  end process;

	  sel_status_pla <= temp;
	  end behavior;
```

```
-- progetto butterfly 2017/2018
-- corso sistemi digitali integrati, prof. Zamboni
-- Montagna Marco, Malacrino Andrea, Valpreda Emanuele
-- command generator

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE ieee.numeric_std.ALL;

entity command_generator is
  port (
                clk, reset: in std_logic;
                input_microAR: in std_logic_vector (4 downto 0);
                output_microIR: out std_logic_vector(17 downto 0));
end command_generator;

architecture behavior of command_generator is

component vector_register
  GENERIC (N:INTEGER);
  PORT(
        EN_REG,CLK,RESET  : IN STD_LOGIC; --segnale di enable, reset e clock
        DATA_REG_IN :     IN std_logic_vector((N-1) DOWNTO 0);      --dato in
ingresso
        DATA_REG_OUT      :     OUT std_logic_vector((N-1) DOWNTO 0)); --dato
in uscita
end component;

component ROM_istruzioni
  port (
                ADDR : in std_logic_vector (4 downto 0); -- address input microAR
                DOUT : out std_logic_vector (17 downto 0)); -- data output istruzioni
microIR
end component;

signal istruzione: std_logic_vector (17 downto 0);

begin

  ROM_generatore_comandi: ROM_istruzioni PORT MAP (input_microAR, istruzione);
  reg_microIR: vector_register
  GENERIC MAP (N=>18)
  PORT MAP ('1', clk, reset, istruzione, output_microIR);

end behavior;
```

```
-- ROM jump
-- progetto butterfly 2017/2018
-- corso sistemi digitali integrati, prof. Zamboni
-- Montagna Marco, Malacrino Andrea, Valpreda Emanuele

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

entity ROM_JUMP is
  port (
                ADDR : in std_logic_vector (4 downto 0); -- address input
                DOUT : out std_logic_vector (5 downto 0)); -- data output
end ROM_JUMP;

architecture BEHAVIOR of ROM_JUMP is

type ROMTABLE is array (0 to 17) of std_logic_vector (5 downto 0);
-- internal table
  constant romdata : romtable := (
        -- ho spostato tutti i cc validation indietro di uno perchè il controllo io ce l'ho
sull'address che è l'uscita di un registro e anche i comandi sono spostati di uno perchè devo
contare anche il clock
        "100001", -- data for address 0, reset
        "100001", -- data for address 1, salto in idle se non arriva start, idle
        "000000", --step 1
        “000000", — step 2
        “000000", — step 3
        "000000",
        "000000",
        "000000",
        "101110", --step 7
        "000000", -- salto nello stato 9.1, step 8
        "000000", -- step 9
        "000000",
        "100001",
        "100001", -- step 12, salto nell'idle perchè ero in one shot e ho finito
        "000000", -- step 9.1, sono in modalità continua e sono saltato qui
        "000000",
        "100110",
        "000000"); -- ho finito giro in modalità continua, salto allo step 5 e riprendo
elaborazione da lì

begin -- BEHAVIOR
-- purpose: Main process
-- type : combinational
-- inputs : ADDR
-- outputs: DOUT
process (ADDR)
begin -- process
        DOUT <= romdata(to_integer(unsigned(ADDR)));
  end process;
end BEHAVIOR;




-- progetto butterfly 2017/2018
```

```
	-- corso sistemi digitali integrati, prof. Zamboni
	-- Montagna Marco, Malacrino Andrea, Valpreda Emanuele
	-- ROM istruzioni

	library ieee;
	use ieee.std_logic_1164.all;
	USE ieee.numeric_std.ALL;

	entity ROM_istruzioni is
	  port (
				ADDR : in std_logic_vector (4 downto 0); -- address input
				DOUT : out std_logic_vector (17 downto 0)); -- data output
	end ROM_istruzioni;

	architecture BEHAVIOR of ROM_istruzioni is

	type ROMTABLE is array (0 to 17) of std_logic_vector (17 downto 0);
	-- internal table
	  constant romdata : romtable := (
	  -- istruzioni per ogni stato
	  -- verso_sommatore, done, reset, sum/sub, mux1, mux2, reg_molt1, reg_molt2,
reg_sum
	  -- en1, en2, en3, en4, en5, en6, en_reg_approx, mux_RF_sum, mux_RF_molt

				"000000000000000000", -- reset
		"001000000000000000", -- idle
		"001000000100100000", -- step 1, abilito reg br e wr
		"001000100010000000", -- step 2
		"001000110001100001", -- step 3
		"001000110010010001", -- step 4
		"001001111000010000", -- step 5, corretto
		"001001011000011010", -- step 6
		"001111001000011000", -- step 7.
		"001011001000001000", -- step 8
		"111010001000001100", -- step 9
		"101010001000001110", -- step 10
		"001000000000001100", -- step 11
		"001000000000000100", -- step 12
		"111010001100101100", -- step 9.1
		"101010101010001110", -- step 10.1
		"001000110001101101", -- step 11.1
		"001000110010010101"); -- step 12.1

	begin -- BEHAVIOR
	-- purpose: Main process
	-- type : combinational
	-- inputs : ADDR
	-- outputs: DOUT
	process (ADDR)
	begin -- process
			DOUT <= romdata(to_integer(unsigned(ADDR)));
	  end process;
	end BEHAVIOR;



	-- progetto butterfly 2017/2018
	-- corso sistemi digitali integrati, prof. Zamboni
```

```
-- Montagna Marco, Malacrino Andrea, Valpreda Emanuele
-- datapath_butterfly

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

entity datapath_butterfly is
  port (
                clk, verso_sommatore, done, reset_datapath, sum_sub_n, mux1, mux2:
in std_logic;
                reg_molt1, reg_molt2, reg_sum: in std_logic;
                en1, en2, en3, en4, en5, en6: in std_logic;
                en_reg_approx, mux_RF_sum, mux_RF_molt : in std_logic;
                B_A, W: in signed (23 downto 0);
                uscita: out  signed (23 downto 0));
end datapath_butterfly;


architecture behavior of datapath_butterfly is

component register_file is
  port (
                clk, reset: in std_logic;
                en1, en2, en3, en4: in std_logic;
                sel_muxB, sel_muxA: in std_logic;
                input_B_A, input_W: in signed (23 downto 0); -- che cosa sono i due
ingressi? Che formato?
                outputB, outputA, outputW: out signed (23 downto 0)); -- di nuovo devo
capire il formato
end component;

component DATA_REGISTER
GENERIC (N:INTEGER);
PORT(
EN_REG,CLK,RESET     : IN STD_LOGIC;          --segnale di enable, reset e clock
DATA_REG_IN   :          IN SIGNED((N-1) DOWNTO 0);          --dato in ingresso
DATA_REG_OUT :          OUT SIGNED((N-1) DOWNTO 0) --dato in uscita
);
end component;

component MUX_NBIT
GENERIC (N:INTEGER);
PORT(
SEL       : IN STD_LOGIC;          --segnale di enable, reset e clock
DATA_1, DATA_2          :          IN SIGNED((N-1) DOWNTO 0);          --dato in ingresso
DATA_OUT        :          OUT SIGNED((N-1) DOWNTO 0)); --dato in uscita
END component;

component molt_pipe2
GENERIC ( N : integer :=24 );
port
  (  en1_molt,en2_molt,rst,clock: in std_logic;
    a : in signed (N-1 downto 0);
    b : in signed  (N-1 downto 0);
    result : out signed  (2*N-2 downto 0)
  );
end component;
```

```
component round_D
GENERIC ( N : integer :=49 );
port (
en,rst,clock: in std_logic;
input: in signed(N-1 downto 0);
output: out signed(23 downto 0));
end component;

component sum_pipe
GENERIC ( N : integer :=49 );
port
  (
    a : in signed (N-1 downto 0);
    b : in signed  (N-1 downto 0);
            w,en,rst,clock, verso_sommatore: in std_logic;
    result : out signed  (N-1 downto 0)
  );
end component;

signal busB, busA, busW: signed (23 downto 0);
signal bus_molt, moltiplicazione: signed (46 downto 0);
signal A_49bit, out_mux1, out_mux2, Aper2_49bit, molt_49bit: signed (48 downto 0);
signal bus_somma, somma: signed (48 downto 0);
signal t : signed (22 downto 0);

begin

RF: register_file PORT MAP (clk, reset_datapath, en1, en2, en3, en4, mux_RF_molt,
mux_RF_sum, B_A, W, busB, busA, busW);
multiplier: molt_pipe2 PORT MAP (reg_molt1, reg_molt2, reset_datapath, clk, busB,
busW, moltiplicazione);
reg_moltiplicatore: DATA_REGISTER
  GENERIC MAP (N=>47)
  PORT MAP (en5, clk, reset_datapath, moltiplicazione, bus_molt);
t <= (OTHERS => '0');
A_49bit <= busA(23)&busA(23)&busA&t;-- come faccio a far diventare A su 49 bit?
multiplexer1: MUX_NBIT
  GENERIC MAP (N=>49)
  PORT MAP (mux1, A_49bit, bus_somma, out_mux1);

Aper2_49bit <= busA(23)&busA&t&'0';-- vedere come fare moltipicazione per 2 e shift
molt_49bit <= bus_molt(46)&bus_molt(46)&bus_molt;
multiplexer2: MUX_NBIT
  GENERIC MAP (N=>49)
  PORT MAP (mux2, Aper2_49bit, molt_49bit, out_mux2);
sommatore: sum_pipe PORT MAP (out_mux1, out_mux2, sum_sub_n, reg_sum,
reset_datapath, clk, verso_sommatore, somma);
reg_multiplier: DATA_REGISTER
  GENERIC MAP (N=>49)
  PORT MAP (en6, clk, reset_datapath, somma, bus_somma);
blocco_approssimazione: round_D
  PORT MAP (en_reg_approx, reset_datapath, clk, bus_somma, uscita);

end behavior;

-- register file
-- progetto butterfly 2017/2018
```

```
-- corso sistemi digitali integrati, prof. Zamboni
-- Montagna Marco, Malacrino Andrea, Valpreda Emanuele

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


entity register_file is
  port (
                clk, reset: in std_logic;
                en1, en2, en3, en4: in std_logic;
                sel_muxB, sel_muxA: in std_logic;
                input_B_A, input_W: in signed (23 downto 0); -- che cosa sono i due
ingressi? Che formato?
                outputB, outputA, outputW: out signed (23 downto 0)); -- di nuovo devo
capire il formato
end register_file;

architecture behavior of register_file is

component DATA_REGISTER
  GENERIC (N:INTEGER);
  PORT(
        EN_REG,CLK,RESET  : IN STD_LOGIC;       --segnale di enable, reset e clock
        DATA_REG_IN :       IN SIGNED((N-1) DOWNTO 0);      --dato in ingresso
        DATA_REG_OUT      :       OUT SIGNED((N-1) DOWNTO 0)); --dato in uscita
end component;

component MUX_NBIT
                GENERIC (N:INTEGER);
                PORT(
                        SEL     : IN STD_LOGIC;         --segnale di enable, reset e clock
                        DATA_1, DATA_2      :       IN SIGNED((N-1) DOWNTO 0);
--dato in ingresso
                        DATA_OUT      :       OUT SIGNED((N-1) DOWNTO 0)); --dato
in uscita
end component;

signal Br, Bi_Ai, Ar: signed (23 downto 0);

begin

  reg_Br: DATA_REGISTER
                GENERIC MAP (N=>24)
                PORT MAP (en1, clk, reset, input_B_A, Br);

  reg_Bi_Ai: DATA_REGISTER
                GENERIC MAP (N=>24)
                PORT MAP (en2, clk, reset, input_B_A, Bi_Ai);

  reg_Ar: DATA_REGISTER
                GENERIC MAP (N=>24)
                PORT MAP (en3, clk, reset, input_B_A, Ar);

  reg_Wr_Wi: DATA_REGISTER
                GENERIC MAP (N=>24)
                PORT MAP (en4, clk, reset, input_W, outputW);
```

```
    mux_B: MUX_NBIT
            GENERIC MAP (N=>24)
            PORT MAP (sel_muxB, Br, Bi_Ai, outputB);

    mux_A: MUX_NBIT
            GENERIC MAP (N=>24)
            PORT MAP (sel_muxA, Ar, Bi_Ai, outputA);

end behavior;
```

```
-- progetto butterfly 2017/2018
-- corso sistemi digitali integrati, prof. Zamboni
-- Montagna Marco, Malacrino Andrea, Valpreda Emanuele

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

entity round_D is
  GENERIC ( N : integer :=49 );
  port (
        en,rst,clock: in std_logic;
        input: in signed(N-1 downto 0);
        output: out signed(23 downto 0));
end round_d;

architecture rtl of round_D is

component DATA_REGISTER IS
  GENERIC (N:INTEGER);
  PORT(
        EN_REG,CLK,RESET  : IN STD_LOGIC;          --segnale di enable, reset e clock
        DATA_REG_IN :         IN signed((N-1) DOWNTO 0);  --dato in ingresso
        DATA_REG_OUT        :         OUT signed((N-1) DOWNTO 0)
); --dato in uscita
END component;

signal rs :signed(N-1 downto 0);
signal bs: std_logic_vector(N-1 downto 0):= (OTHERS => '0');
signal bs1 :signed(N-1 downto 0);
signal c :signed(23 downto 0);

begin
   bs1 <= signed(bs);
   bs(22) <= '1';
  rs <= input+bs1;
   c <= rs(48 downto 25);

Registro : DATA_REGISTER generic map(N=>24) port map(en,clock,rst,c,output);
end rtl;
```

```
-- progetto butterfly 2017/2018
-- corso sistemi digitali integrati, prof. Zamboni
-- Montagna Marco, Malacrino Andrea, Valpreda Emanuele

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


entity sum_pipe is
GENERIC ( N : integer :=49 );
port
  (
    a : in signed (N-1 downto 0);
    b : in signed  (N-1 downto 0);
          w,en,rst,clock, verso_sommatore: in std_logic;
    result : out signed  (N-1 downto 0)
  );

end sum_pipe;

architecture rtl of sum_pipe is
component DATA_REGISTER IS
  GENERIC (N:INTEGER);
  PORT(
          EN_REG,CLK,RESET   : IN STD_LOGIC;          --segnale di enable, reset e clock
          DATA_REG_IN :          IN signed((N-1) DOWNTO 0);  --dato in ingresso
          DATA_REG_OUT         :          OUT signed((N-1) DOWNTO 0)
); --dato in uscita
END component;

signal p :signed (N-1 downto 0);

begin

  process(a,b,w, verso_sommatore)
  begin
  if(w='1') then
  p <= a + b;
  else
          if (verso_sommatore = '0') then
                  p<= a-b;
          elsif (verso_sommatore = '1') then
                  p <= b-a;
          end if;
  end if;
  end process;
pipe : DATA_REGISTER generic map(N=>49) port map(en,clock,rst,p,result);

end rtl;
```

```
-- progetto butterfly 2017/2018
-- corso sistemi digitali integrati, prof. Zamboni
-- Montagna Marco, Malacrino Andrea, Valpreda Emanuele

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


entity molt_pipe2 is
GENERIC ( N : integer :=24 );
port
  (  en1_molt,en2_molt,rst,clock: in std_logic;
     a : in signed (N-1 downto 0);
     b : in signed  (N-1 downto 0);
     result : out signed  (2*N-2 downto 0)
  );

end molt_pipe2;

architecture rtl of molt_pipe2 is
component DATA_REGISTER IS
GENERIC (N:INTEGER);
PORT(
EN_REG,CLK,RESET     : IN STD_LOGIC;          --segnale di enable, reset e clock
DATA_REG_IN    :     IN signed((N-1) DOWNTO 0);  --dato in ingresso
DATA_REG_OUT :       OUT signed((N-1) DOWNTO 0)
); --dato in uscita
END component;

signal rs :signed(2*N-1 downto 0);
signal t,p :signed(2*N-2 downto 0);

begin

  rs <= a * b;
   t<=rs(2*N-2 downto 0);

pipe1 : DATA_REGISTER generic map(N=>47) port map(en1_molt,clock,rst,t,p);
pipe2 : DATA_REGISTER generic map(N=>47) port map(en2_molt,clock,rst,p,result);
end rtl;
```

```
-- progetto butterfly 2017/2018
-- corso sistemi digitali integrati, prof. Zamboni
-- Montagna Marco, Malacrino Andrea, Valpreda Emanuele

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
use ieee.numeric_std.all;

ENTITY DATA_REGISTER IS
GENERIC (N:INTEGER);
PORT(
EN_REG,CLK,RESET     : IN STD_LOGIC;          --segnale di enable, reset e clock
DATA_REG_IN    :     IN SIGNED((N-1) DOWNTO 0);          --dato in ingresso
DATA_REG_OUT :       OUT SIGNED((N-1) DOWNTO 0) --dato in uscita
);
END ENTITY DATA_REGISTER;
ARCHITECTURE BEHAVIOR OF DATA_REGISTER IS
BEGIN
REG_PROCESS: PROCESS(CLK)
BEGIN
IF (CLK'EVENT AND CLK='1') THEN
IF RESET='0' THEN
DATA_REG_OUT<=(OTHERS=>'0');
ELSIF (EN_REG='1') THEN
DATA_REG_OUT((N-1) DOWNTO 0)<=DATA_REG_IN((N-1) DOWNTO 0);
END IF;
END IF;
END PROCESS;
END BEHAVIOR;
```

```
-- progetto butterfly 2017/2018
-- corso sistemi digitali integrati, prof. Zamboni
-- Montagna Marco, Malacrino Andrea, Valpreda Emanuele

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
use ieee.numeric_std.all;

ENTITY MUX_NBIT IS
GENERIC (N:INTEGER);
PORT(
SEL	: IN STD_LOGIC;		--segnale di enable, reset e clock
DATA_1, DATA_2		:		IN SIGNED((N-1) DOWNTO 0);		--dato in ingresso
DATA_OUT		:		OUT SIGNED((N-1) DOWNTO 0)); --dato in uscita

END ENTITY MUX_NBIT;
ARCHITECTURE BEHAVIOR OF MUX_NBIT IS
BEGIN
WITH SEL SELECT DATA_OUT<=
DATA_1 WHEN '0',
DATA_2 WHEN '1',
DATA_1 WHEN OTHERS;
END BEHAVIOR;
```

Gli ultimi due componenti *vector_register* e *vector_mux* sono uguali agli ultimo due, con la differenza che i dati in ingresso non sono SIGNED, ma STD_LOGIC_VECTOR.

4. Test

Scritto il codice VHDL, è stato necessario testarlo per vedere il corretto funzionamento; per farlo abbiamo utilizzato Modelsim e, tramite delle testbench, abbiamo ricreato situazioni di lavoro in modalità continua o modalità one-shot in modo da analizzare se il comportamento della macchina fosse quello da noi concepito.

La cosa fondamentale durante il test è, come dovrebbe poi avvenire nell'utilizzo reale della Butterfly, mandare con la giusta tempistica i segnali di ingresso: dopo che la mia macchina campiona il segnale di start devo far passare un tempo pari al tempo di clock per inviare i primi due dati in ingresso. Da quel momento in poi bisogna inviare gli altri dati ogni colpo di clock fino a quando non ce n'è più bisogno.
Per quanto riguarda il test della modalità continua daremo lo start in modo che venga campionato il settimo colpo di clock per poi dare i dati il colpo di clock successivo. Il test è stato effettuato con l'utilizzo di Matlab, cercando di ricreare il funzionamento della macchina con il relativo meccanismo di approssimazione. I risultati sono già quelli corretti cioè riscaldati di un fattore 4 per rientrare nella dinamica corretta -1, +1.

4.1 Modalità One-shot

Essendo difficile riportare in una schermata il funzionamento completo della macchina, riporteremo i risultati all'interno di una tabella, accompagnandoli con alcuni screen-shot di Modelsim rappresentanti le fasi più importanti del funzionamento.

Riportiamo prima di tutto una tabella con alcuni test della modalità one-shot ed alcuni screen-shot in cui possiamo vedere alcuni dei risultati rappresentati nella tabella; in queste immagini possiamo vedere ad esempio come il segnale di DONE si alzi il colpo di clock precedente all'invio del primo dato in uscita, come previsto durante la progettazione del sistema. Il funzionamento interno della macchina non è stato mostrato in quanto è semplicemente un'esecuzione dei comandi descritti in VHDL. Nei test 2, 3 e 4 abbiamo provato a testare le criticità del nostro sitema, ad esempio arrivando al massimo valore di uno dei risultati o andando in underflow (B'r test 3) per vedere che il risultato fosse quello atteso.

Qui sotto riportiamo immagini e tabelle:

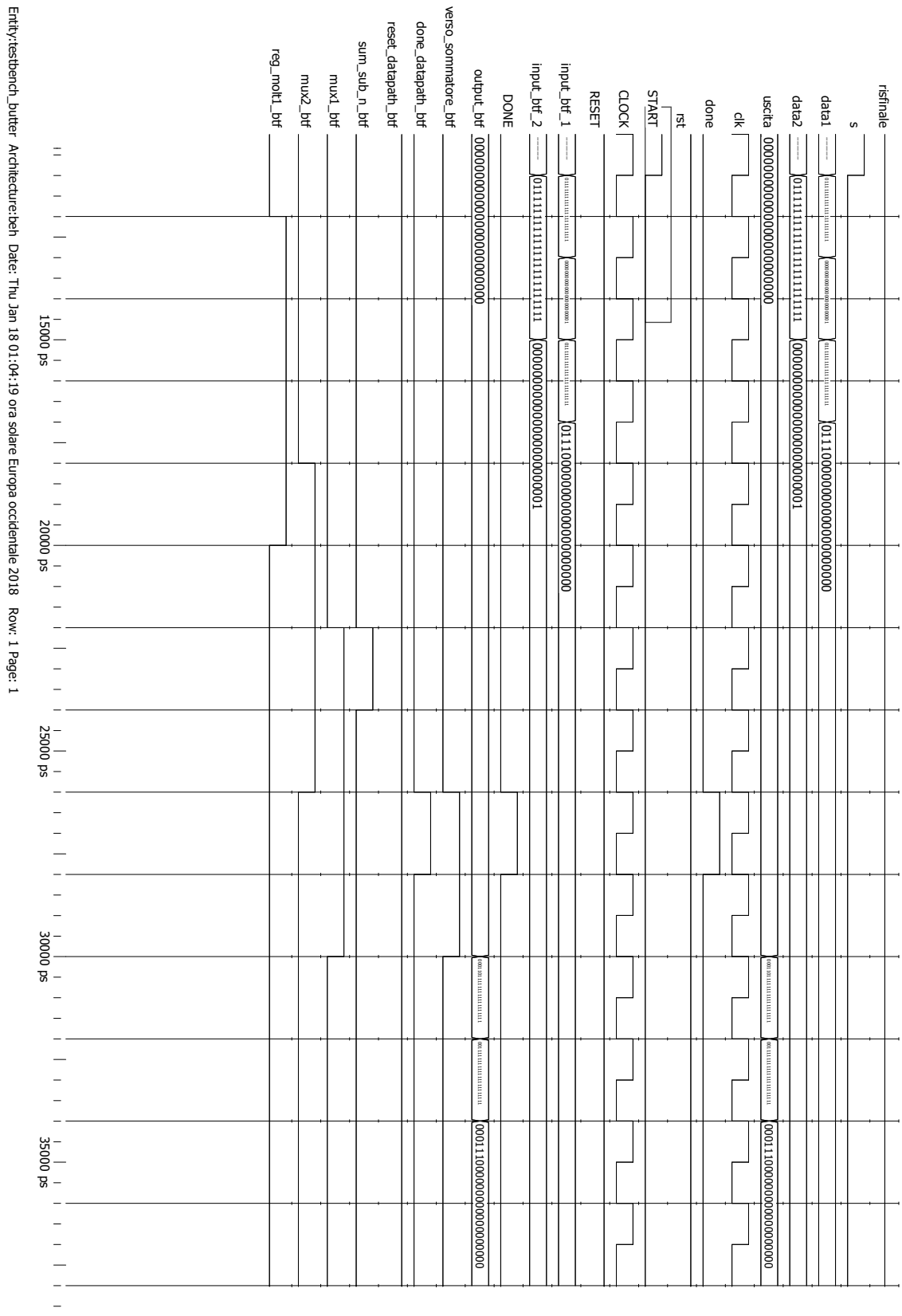
Tabella test Butterfly in modalità one-shot

Tabella 1

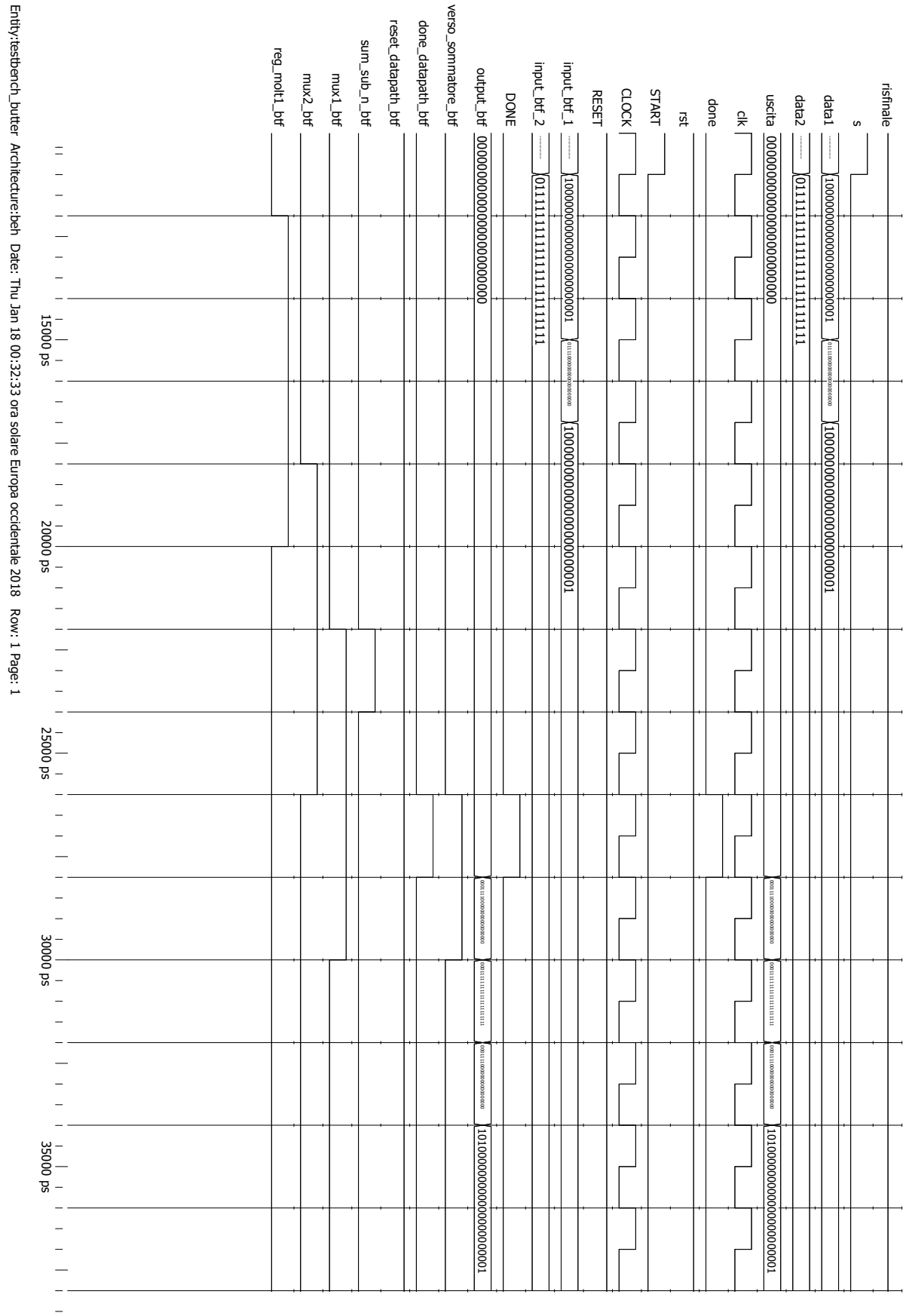
TEST	Br	Bi	Ar	Ai	Wr	Wi
TEST 1	0110000000000000000000000000	1100000000000000000000000000	1110000000000000000000000000	0111000000000000000000000000	0100000000000000000000000000	0100000000000000000000000000
TEST 2	0111111111111111111111111111	0111111111111111111111111111	0111111111111111111111111111	0111000000000000000000000000	0111111111111111111111111111	0111111111111111111111111111
TEST 3	0111111111111111111111111111	0111111111111111111111111111	0111111111111111111111111111	0111000000000000000000000000	0111111111111111111111111111	0000000000000000000000000001
TEST 4	1000000000000000000000000001	1000000000000000000000000001	0111100000000000000000000000	1000000000000000000000000001	0111111111111111111111111111	0111111111111111111111111111

B'r	B'i	A'r	A'i
1110010000000000000000000000	0001100000000000000000000000	0000110000000000000000000000	0010000000000000000000000000
1110000000000000000000000000	0001110000000000000000000000	0101111111111111111111111110	0001110000000000000000000000
0000000000000000000000000000	0001101111111111111111111111	0011111111111111111111111111	0001110000000000000000000000
0001111000000000000000000000	0001111111111111111111111111	0001111000000000000000000000	1010000000000000000000000001

Schermata Modelsim TEST 3:



Schermata Modelsim TEST 4:

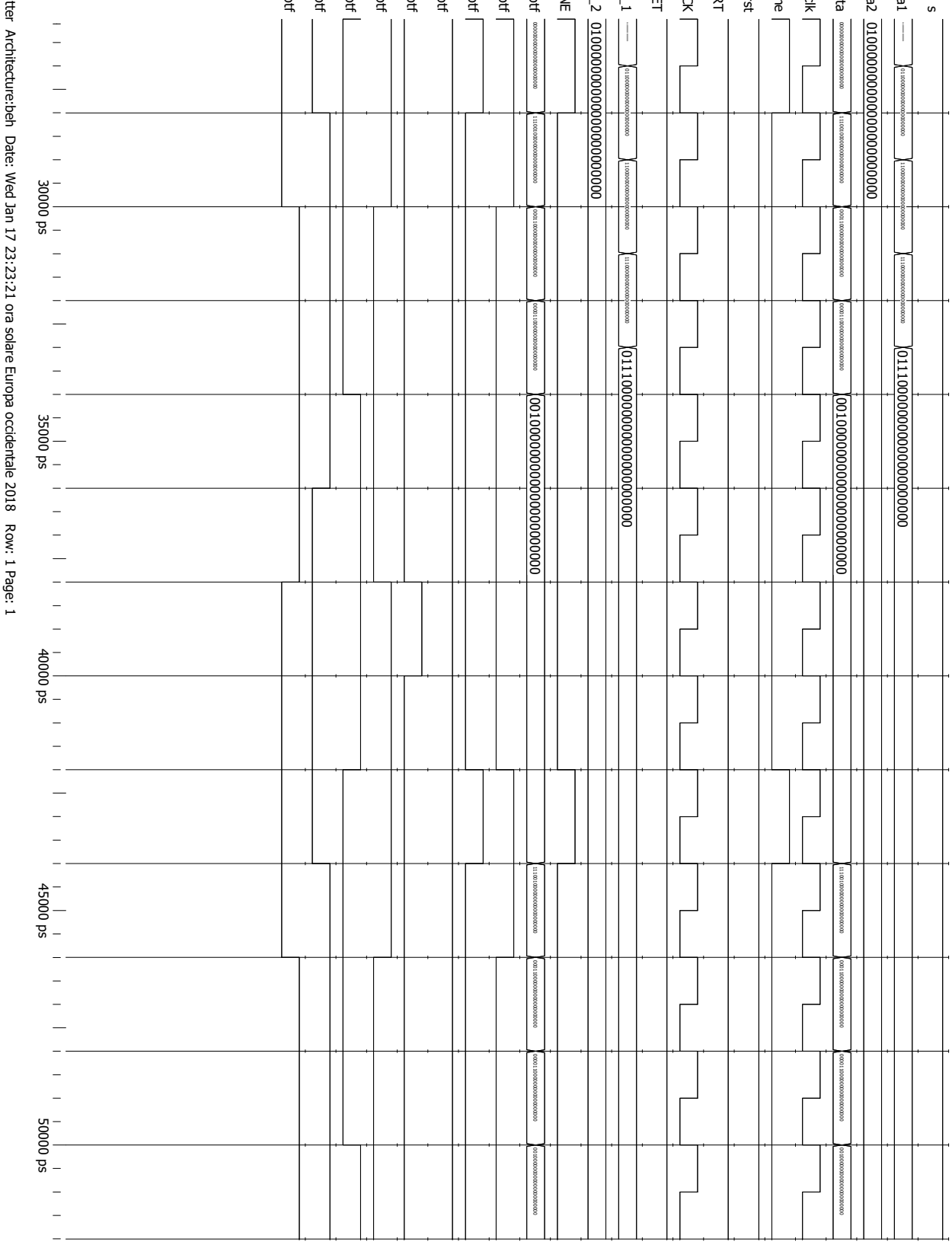


4.2 Modalità continua

La modalità continua, come spiegato precedentemente, consiste nell'utilizzo sequenziale della butterfly in modo da non lasciare tempi morti tra un'elaborazione ed un'altra. Per testarla inseriamo dei dati in ingresso come se la stessimo utilizzando in modalità one-shot, poi, al settimo colpo di clock, facciamo in modo di campionare il segnale di START in modo da saltare dello step 9.1 della µROM che definisce il funzionamento in questa seconda modalità. Invieremo quindi il nuovo ciclo di dati a partire dal colpo di clock successivo a quello in cui abbiamo campionato il segnale di START. Riporteremo in seguito uno screen-shot che mostri il contenuto della µROM e quindi il corretto funzionamento dei salti tra gli stati. Per come la nostra macchina era stata pensata, dopo il primo ciclo, in mdalità continua la nostra macchina avrebbe dovuto far uscire i dati in sequenza ogni quattro colpi di clock (tempo calcolato dall'ultimo dato del ciclo precedente al primo del ciclo attuale). Inizialmente per testare abbiamo deciso di riutilizzare i dati i dati del TEST 1 della modalità one-shot inviandoli in ingresso sia al primo che al secondo ciclo: come mostrato dallo screen-shot successivo i dati sono uguali tra loro e corretti.

[I risultati si vedranno molto meglio nel .pdf dove si può zoommare]

Riportiamo nella prima immagine i risultati del test inserendo come valori di ingressi quelli del TEST 1 della modalità one-shot; nella seconda immagine possiamo vedere il salto del µAR che va dallo step 8 allo step 9.1, iniziando così il funzionamento in continua della macchina (da 01001 a 01110). Vediamo infatti come il CC validation sia attivo e questo segnale - insieme al segnale di start che non si vede nella schermata - mi permette di spostare il multiplexer in modo da saltare correttamente nello stato futuro.


s
data1
data2
uscita
clk
done
rst
START
CLOCK
RESET
input_btf_1
input_btf_2
DONE
output_btf
verso_sommatore_btf
done_datapath_btf
reset_datapath_btf
sum_sub_n_btf
mux1_btf
mux2_btf
reg_molt1_btf
reg_molt2_btf
30000 ps
35000 ps
40000 ps
45000 ps
50000 ps
Entity:testbench_butter Architecture:beh Date: Wed Jan 17 23:23:21 ora solare Europa occidentale 2018 Row: 1 Page: 1

Entity:testbench_butter Architecture:beh Date: Wed Jan 17 23:18:11 ora solare Europa occidentale 2018 Row: 1 Page: 1

Per fare in modo che i dati fossero campionati in maniera corretta li inviamo a metà del colpo di clock precedente in modo da essere sicuri che siano stabili. All'interno del file della butterfly [allegato all'e-mail del progetto] abbiamo inserito anche lo script matlab con cui abbiamo testato i nostri valori.

Riportiamo qui sotto la tabella con i risultati dei test in continua in cui abbiamo eseguito due cicli concatenati:

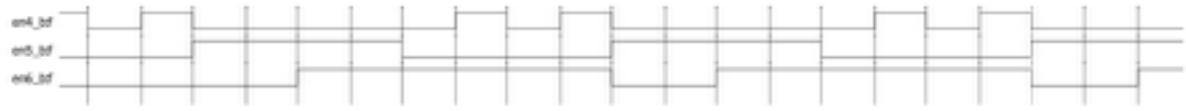
Tabella 1

TEST	Br	Wr	Bi	Ar	Wi	Ai
TEST 1 (primo ciclo)	0001000000000000000000000000	0000000000000000000000000001	1000010010001000100000000000	0100000000000000000000000000	0011000000000000000000000000	1001000000000000000000000000
TEST 1 (secondo ciclo)	1000000000010000001000000000	0111111111111111111111111111	0110110000000000000000000000	0000000100000000000000000001	0110000000000000000000000000	1000000010001000000000000000
TEST 2 (primo ciclo)	0011000000000000000000000000	0001000000000000000000000000	0100000000000000000000000000	0000000010000000000000000000	1001000000000000000000000000	1000010010001000100000000000
TEST 2 (secondo ciclo)	0110000000000000000000000000	1000000000010000001000000000	0000001000000000000000000001	0111111111111111111111111111	1000000010001000000000000000	0110110000000000000000000000

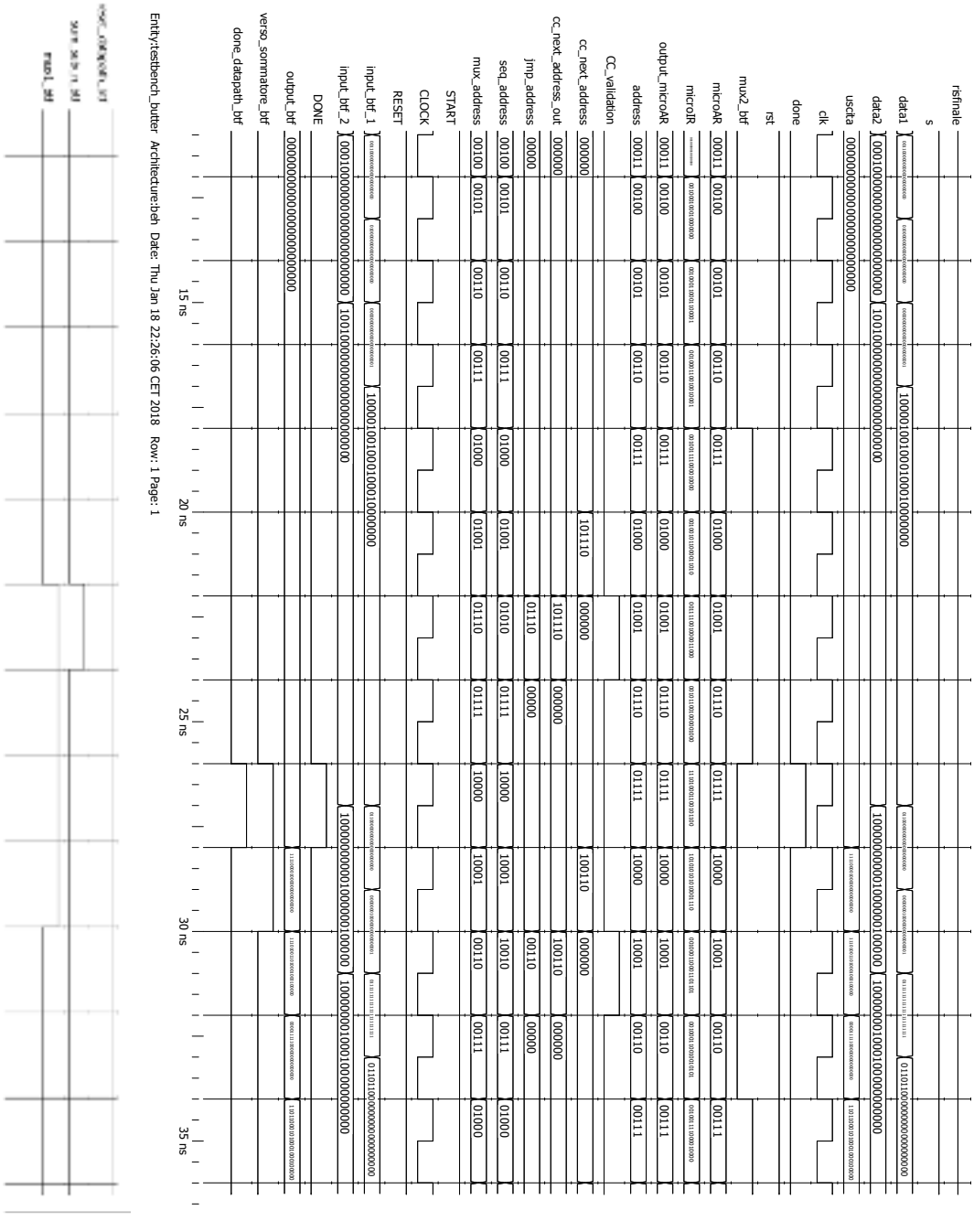
B'r	B'i	A'r	A'i
00000100011011001100100	11100010100000000000000000	00011011100100110010100	11100101011111111111111
00101000111011111111000	11011010001110111111010	11001100000001000000001000	11100011001001010000101
11110000100000000000000000	11101001101000100010000000	00001111100000000000000000	11011000101000100010000000
00110111011110100111101	00110011001001100111111000	00001000010000101100010	00000010110110011000111

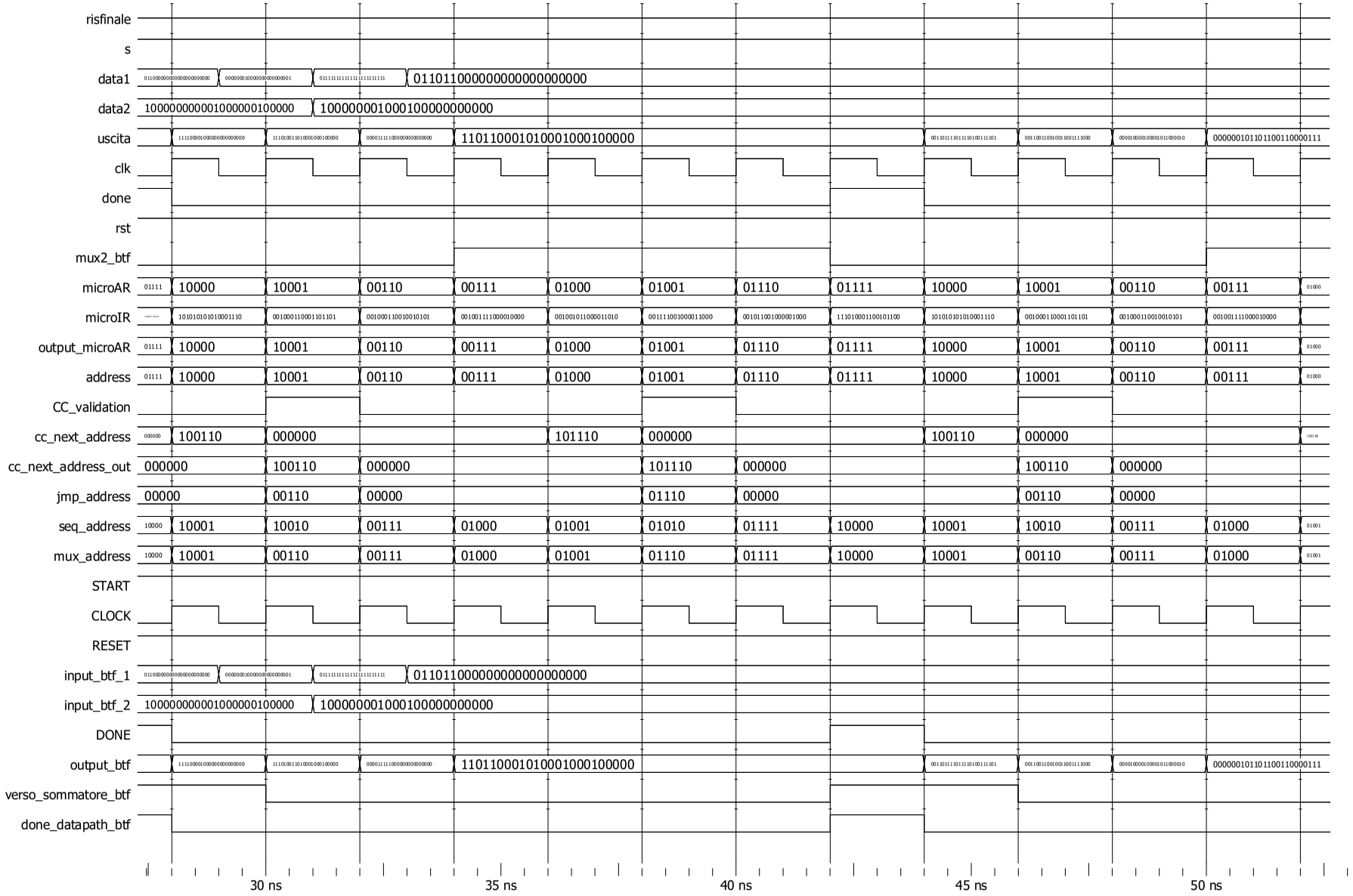
Entity:testbench_butter Architecture:beh Date: Thu Jan 18 21:59:00 CET 2018 Row: 1 Page: 1

Nel timing diagram presente nella pagina precedente è mostrata una sezione del ciclo in modalità continua. In questo modo si possono vedere i segnali e la sequenza degli ingressi con i relativi salti dovuti alla modalità continua.
[Qui sotto riportiamo gli ultimi segnali, fanno parte del timing diagram precedente].

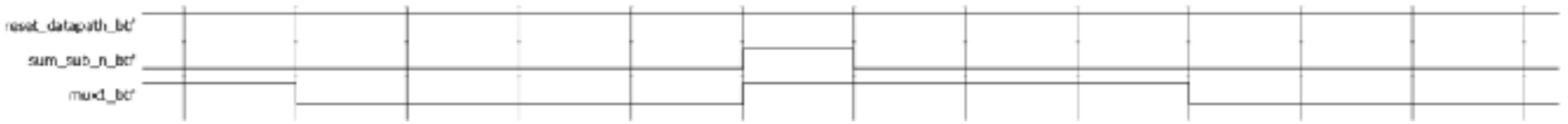


Nelle pagine successive riportiamo ancora qualche timing diagram di altri test eseguiti, i risultati sono presenti nella seconda tabella (modalità continua).





Entity:testbench_butter Architecture:beh Date: Thu Jan 18 22:27:05 CET 2018 Row: 1 Page: 1



Fine della relazione Butterfly, corso di sistemi digitali integrati, anno accademico 2017/2018.

Marco Montagna

Andrea Malacrino

Emanuele Valpreda