ALMA MATER STUDIORUM – UNIVERSITA' DI BOLOGNA
CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA IN INGEGNERIA ELETTRONICA, INFORMATICA E TELECOMUNICAZIONI

Progetto VHDL di un nucleo di calcolo per misure di impedenza basate su segnali pseudo-random

Elaborato in :
Elettronica dei Sistemi Digitali

Relatore
Prof.Ing. Aldo Romani

Presentata da
Marco Raspanti

Correlatore
Dr. Marco Crescentini

II appello I sessione
Anno Accademico 2014-2015

INDICE

Introduzione....3
CAPITOLO 1 - PRESENTAZIONE SISTEMA....5
1.1 Teoria dei segnali per il calcolo della risposta impulsiva....5
1.2 Schema a blocchi del sistema....6
1.3 Specifiche dei componenti....6
CAPITOLO 2 - GENERATORE PSEUDO-RANDOM....11
2.1 Introduzione ai generatori random....11
2.2 Lfsr....11
2.3 Polinomi degli Lfsr....12
2.4 Software....13
2.5 Caratteristiche della sequenza pseudo-random....14
CAPITOLO 3 – CONVOLUZIONE....17
3.1 Introduzione alla convoluzione....17
3.2 Filtro Fir....17
3.3 Overlap and add....18
3.4 Software....20
CAPITOLO 4 – NUCLEO DI CALCOLO....25
4.1 Introduzione al nucleo di calcolo....25
4.2 Nucleo di calcolo....26
4.3 Macchina a stati del nucleo di calcolo....29
CAPITOLO 5 – SIMULAZIONE....37
5.1 Sintesi logica....37
5.2 Simulazione....37
Conclusioni....41
Bibliografia/sitografia....43
Ringraziamenti....45

INTRODUZIONE

In questo elaborato si affronta il progetto di un nucleo di calcolo per misure d'impedenza sulla pelle tramite l'utilizzo di segnali pseudo-random. Esistono infatti studi che legano l'impedenza cutanea, misurata a frequenza tra 1 kHz e 1 Mhz, alla presenza o assenza del carcinoma della pelle. Per maggiori informazioni sull'applicazione fare riferimento alla tesi di laurea di Giulia Luciani (riferimento [9] bibliografia). La misura viene effettuata applicando il segnale casuale all'impedenza per ottenere la risposta impulsiva, successivamente se ne calcola la trasformata di Fourier per ricavarne lo spettro e con gli opportuni calcoli si ottiene il valore dell'impedenza. In questo progetto ci si concentrerà sul calcolo della risposta impulsiva mentre le successive operazioni non verranno affrontate. La sequenza casuale dovrà avere una periodicità superiore ai 10 ms in modo da poter svolgere misure d' impedenze con un tempo di risposta fino a 10 ms e quindi minima frequenza di misura 100 Hz. Se questo requisito non fosse rispettato, durante il calcolo dello spettro si noterebbe la periodicità della sequenza. Essa dovrà avere una frequenza di 10 MHz per garantire corrette misure d'impedenza fino ad almeno 1 MHz. La fase di applicazione del segnale random e la memorizzazione dei dati non deve avere una durata superiore a qualche secondo così da essere facilmente applicabile sul paziente, mentre la fase di elaborazione dei dati non deve durare più di qualche minuto. In questo progetto è stata scelta una sequenza pseudo-random binaria e si è diviso il progetto in due fasi: una fase in real-time dove viene generata la sequenza casuale e vengono memorizzati tutti dati e una off-line dove viene calcolata la risposta impulsiva.

Nel capitolo 1 viene mostrata la teoria su cui si basa il metodo per il calcolo della risposta impulsiva e si presentano i dispositivi sui quali è basato il progetto.

Nel capitolo 2 si affronta il progetto del generatore pseudo-random e si mostra l'implementazione in VHDL.

Nel capitolo 3 si affronta il progetto del package per il calcolo della convoluzione e si mostra l'implementazione VHDL.

Nel capitolo 4 viene spiegato come è stato implementato il nucleo di calcolo.

Nel capitolo 5 si affronta la fase di simulazione del sistema.

CAPITOLO 1

1.1 TEORIA DEI SEGNALI PER IL CALCOLO DELLA RISPOSTA IMPULSIVA

In questo paragrafo dimostreremo come calcolare la risposta impulsiva di un sistema svolgendo un'operazione di convoluzione tra l'uscita e l'ingresso invertito. Il segnale di ingresso deve essere un segnale random. Sapendo che la risposta di un sistema LTI è uguale alla convoluzione tra il segnale in ingresso e la risposta impulsiva possiamo scrivere:

$y(n)*x(-n)=x(n)*h(n)*x(-n)$

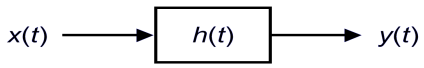


Figura 1.1 Schema a blocchi di un sistema LTI

dove x(n) rappresenta la sequenza casuale in ingresso al sistema LTI, y(n) è la sua risposta e h(n) è la risposta impulsiva del sistema LTI in esame.
Utilizzando la proprietà commutativa, si può ricavare la funzione di autocorrelazione di x(n). Essendo definita come $Rx(n)=x(n)*\bar{x}(n)$ e ricordandoci che x(n) è reale allora $\bar{x}(n)=x(-n)$ si ricava:

$y(n)*x(-n)=h(n)*x(n)*x(-n)=h(n)*Rx(n)$

se x(n) è casuale allora $Rx(n)=\delta(n)$, dove δ(n) rappresenta la Delta di Dirac, quindi semplificando si ottiene:

$y(n)*x(-n)=h(n)*\delta(n)=h(n)$

In questo modo abbiamo dimostrato che calcolando la convoluzione tra la risposta di un sistema LTI a un ingresso casuale e lo stesso ingresso casuale ma invertito, si ottiene la risposta impulsiva di tale sistema. Ovviamente dato che x(n) non è random ma pseudo-random la sua funzione di autocorrelazione non sarà una Delta di Dirac ma sarà triangolare. Tanto più la forma triangolare tenderà ad una Delta di Dirac, ovvero la base del triangolo sarà stretta, tanto più il comportamento sarà vicino a quello ideale e quindi migliore sarà la sequenza.

1.2 SCHEMA A BLOCCHI DEL SISTEMA

Il sistema dovrà quindi generare una sequenza pseudo-random, e dovrà calcolare la convoluzione tra la stessa e l'uscita del sistema LTI. In questo caso si è deciso di utilizzare un FPGA poiché consente la diminuzione dei tempi di progettazione, di verifica mediante simulazioni e di prova sul campo dell'applicazione. Il grande vantaggio che permette è quello di apportare eventuali modifiche o correggere errori semplicemente riprogrammando il dispositivo in qualsiasi momento. Oltre all' FPGA sarà necessaria una memoria per memorizzare i dati su cui calcolare la convoluzione e per salvarne il risultato. E' necessario anche un convertitore corrente tensione, poiché in questo caso il sistema LTI in esame è un bipolo, e un convertitore analogico digitale per convertire il segnale di uscita del sistema LTI. La Figura 1.2 mostra lo schema a blocchi del sistema completo.

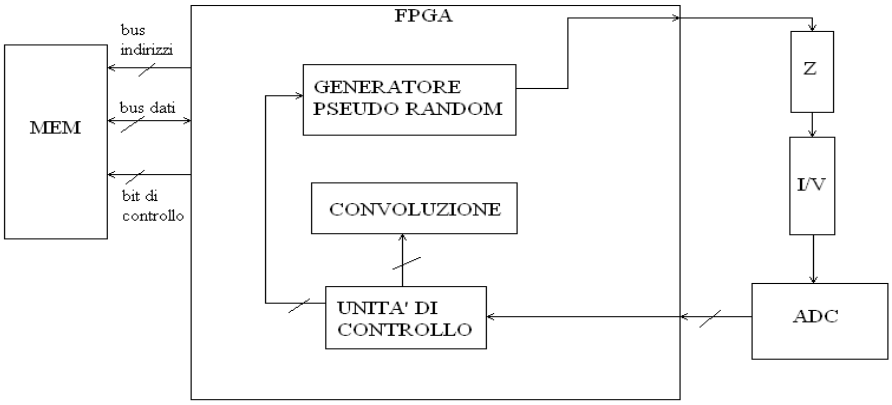


Figura 1.2 Schema a blocchi dei componenti per la misura dell'impedenza

1.2 SPECIFICHE DEI COMPONENTI

Il modello di FPGA utilizzato per il progetto è Cyclone II EP2C20F484C7, presente sulla development board Altera “Cyclone II Starter Development Kit”.
Le caratteristiche principali della scheda sono mostrate in Figura 1.3.

- Altera Cyclone® II EP2C20 FPGA device
- Altera EPCS4 Serial Configuration device
- USB-Blaster controller chip set for programming and user API control, supporting both JTAG and Active Serial (AS) programming modes
- 512-KByte SRAM
- 8-MByte SDRAM
- 4-MByte Flash memory
- SD Flash Card socket
- 4 Push button switches
- 10 Toggle switches
- 10 Red user LEDs
- 8 Green user LEDs
- 50 MHz, 27 MHz, and 24 MHz oscillators for clock sources
- 24-bit CD-quality audio CODEC with line-in, line-out, and microphone-in jacks
- VGA DAC (4-bit resistor network) with VGA-out connector
- RS-232 transceiver and 9-pin connector
- PS/2 mouse/keyboard connector
- Two 40-pin expansion headers with resistor protection
- 7.5V DC adapter or a USB cable (provided in the kit) for power

Figura 1.3 Principali caratteristiche della development board (riferimento [3] bibliografia)

Come si nota sulla scheda è presente una SRAM. Il progetto si è basato sulle caratteristiche di quest'ultima anche se successivamente si dimostrerà che servirà una memoria con una capacità maggiore. La famiglia a cui appartiene la SRAM è la IS61C25616AL/AS. Le caratteristiche utili a questo progetto sono:

- celle di memoria con word a 16bit.
- tempi caratteristici della SRAM per i cicli di lettura e scrittura (illustrati in seguito).

A0-A17	Address Inputs
I/O0-I/O15	Data Inputs/Outputs
$\overline{CE}$	Chip Enable Input
$\overline{OE}$	Output Enable Input
$\overline{WE}$	Write Enable Input

$\overline{LB}$	Lower-byte Control (I/O0-I/O7)
$\overline{UB}$	Upper-byte Control (I/O8-I/O15)
NC	No Connection
V_{DD}	Power
GND	Ground

Figura 1.4 Pin della memoria (riferimento [4] bibliografia)

Con riferimento alla Figura 1.4 e 1.5 analizzeremo un ciclo di lettura. Per comandare la memoria in modo che possa essere letta dovremmo abilitare il CE, OE, LB, UB mentre dovremmo tenere disabilitato il WE. Essendo segnali attivi bassi, per abilitarli devono essere posti uguali a zero. Una volta fornito l'indirizzo della cella, la memoria risponde con il dato dopo 25 ns e appena l'indirizzo cambia il dato non sarà più valido. Per i tempi caratteristici si fa riferimento alla Figura 1.6.

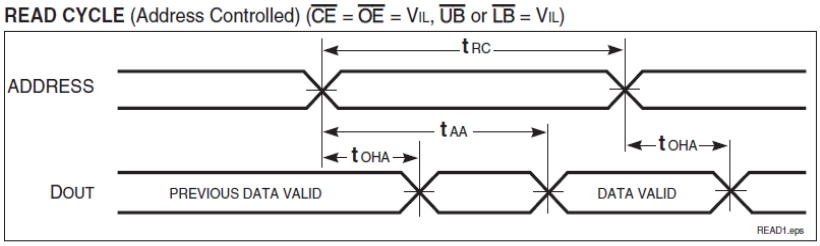


Figura 1.5 Forme d'onda relative al ciclo di lettura della memoria (riferimento [4] bibliografia)

READ CYCLE SWITCHING CHARACTERISTICS (Over Operating Range)

		-10		-12		-25		
Symbol	**Parameter**	**Min.**	**Max.**	**Min.**	**Max.**	**Min.**	**Max.**	**Unit**
tRC	Read Cycle Time	10	—	12	—	25	—	ns
tAA	Address Access Time	—	10	—	12	—	25	ns
tOHA	Output Hold Time	3	—	3	—	3	—	ns

Figura 1.6 Tempi caratteristici della memoria per il ciclo di lettura (riferimento [4] bibliografia)

Durante il ciclo di scrittura invece bisogna abilitare il CE, LB, UB e disabilitare l'OE. La scrittura sarà comandata con il WE. Una volta applicato l'indirizzo e abilitato il WE, il dato deve rimanere valido per 15 ns per memorizzarlo correttamente. Prima di poter disattivare la memoria, il dato deve rimanere stabile per altri 15 ns. Una volta disabilitato il WE, si può cambiare l'indirizzo. Per le forme d'onda e i tempi si faccia riferimento alle Figure 1.7 e 1.8.

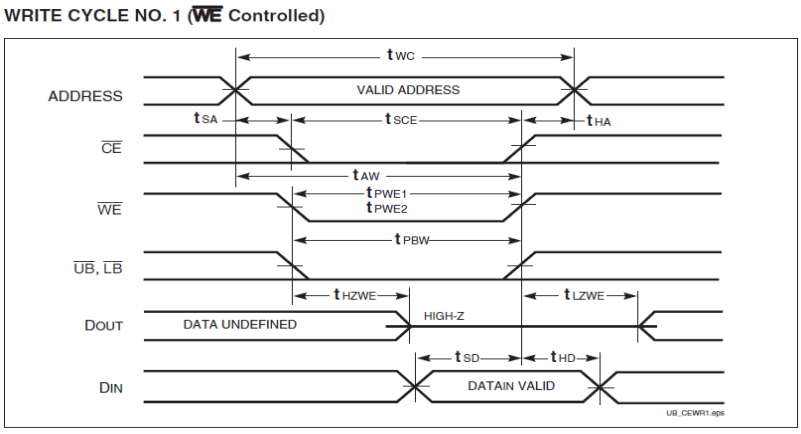


Figura 1.7 Forme d'onda relative al ciclo di scrittura della memoria (riferimento [4] bibliografia)

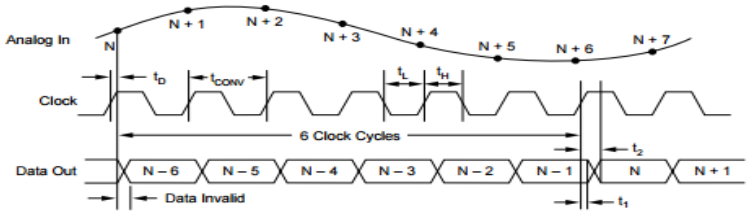
WRITE CYCLE SWITCHING CHARACTERISTICS (Over Operating Range)

Symbol	Parameter	-10 Min.	-10 Max.	-12 Min.	-12 Max.	-25 Min.	-25 Max.	Unit
t_{WC}	Write Cycle Time	10	—	12	—	25	—	ns
t_{SCE}	$\overline{CE}$ to Write End	7	—	9	—	18	—	ns
t_{AW}	Address Setup Time to Write End	7	—	9	—	18	—	ns
t_{HA}	Address Hold from Write End	0	—	0	—	0	—	ns
t_{SA}	Address Setup Time	0	—	0	—	0	—	ns
t_{PWB}	$\overline{LB}$, $\overline{UB}$ Valid to End of Write	7	—	9	—	18	—	ns
t_{PWE1}	$\overline{WE}$ Pulse Width ($\overline{OE}$ =High)	7	—	9	—	15	—	ns
t_{PWE2}	$\overline{WE}$ Pulse Width ($\overline{OE}$=Low)	7	—	9	—	17	—	ns
t_{SD}	Data Setup to Write End	6	—	6	—	15	—	ns
t_{HD}	Data Hold from Write End	0	—	0	—	0	—	ns
t_{HZWE}	$\overline{WE}$ LOW to High-Z Output	—	6	—	6	—	15	ns
t_{LZWE}	$\overline{WE}$ HIGH to Low-Z Output	3	—	3	—	5	—	ns

Figura 1.8 Tempi caratteristici della memoria per il ciclo di scrittura (riferimento [4] bibliografia)

Nel caso si voglia cambiare famiglia di memorie, si dovrà optare per una con tempi di accesso e di risposta inferiori o uguali a quella scelta per il progetto.

Il convertitore analogico digitale (ADC) utilizzato è l'ADS804, il quale raggiunge una frequenza di campionamento di 10 MHz ed ha una risoluzione di 12 bit. Questo ADC va comandato tramite un segnale di clock che sarà generato dall' FPGA. Tale segnale dovrà essere sincronizzato con quello del generatore random in modo da campionare il dato ogni volta che il generatore ne genera uno nuovo. Unico dettaglio da tenere in considerazione è lo sfasamento tra il campione e la sua conversione poiché la conversione relativa al campione corrente sarà disponibile in uscita dopo 6 cicli di clock.



SYMBOL	DESCRIPTION	MIN	TYP	MAX	UNITS
t_{CONV}	Convert Clock Period	100		100µs	ns
t_L	Clock Pulse LOW	48	49		ns
t_H	Clock Pulse HIGH	48	49		ns
t_D	Aperture Delay		2		ns
t_1	Data Hold Time, C_L = 0pF	3.9			ns
t_2	New Data Delay Time, C_L = 15pF max			12	ns

Figura 1.9 Forme d'onda relative all'adc (riferimento [4] bibliografia)

CAPITOLO 2

2.1 INTRODUZIONE AI GENERATORI RANDOM

Pensare di produrre una sequenza random utilizzando un circuito logico può sembrare strano dato che il risultato di qualsiasi circuito logico è per sua natura deterministico; di conseguenza anche la sequenza random deve essere deterministica, questo però non elimina la possibilità di generare numeri pseudo-random. E' necessario precisare che nessun generatore random può produrre una sequenza di lunghezza arbitraria, quindi la sequenza sarà periodica. La caratteristica dei generatori di numeri random è quella di produrre sequenze deterministiche tali che ogni sequenza, con lunghezza inferiore a quella del periodo, abbia le stesse proprietà statistiche di una sequenza di numeri veramente random o comunque ideale. Per sottolineare questo aspetto spesso le sequenze di numeri generati da un circuito logico sono chiamate pseudo-random.

2.2 LFSR

La prima struttura da implementare è il generatore pseudo-random: in questo progetto si utilizzerà una sequenza binaria, in inglese “pseudo random binary sequence” (PRBS). Si è scelta una sequenza binaria per la facilità con cui può essere implementata tramite un circuito logico. La struttura del generatore random può essere implementata tramite un “linear feedback shift register” (lfsr), che consiste in una sequenza di registri caricati con un seed iniziale e retroazionati da una porta xor. Nella Figura 2.1 vine mostrato l'esempio di un “lfsr-4”, cioè composto da quattro registri, e la tabella che ne spiega il funzionamento. Si noti che l'xor genera il nuovo bit il quale verrà caricato nel primo registro, mentre l'uscita dell'ultimo registro è il bit utilizzato per la sequenza random. Come precedentemente affermato, tale sequenza è periodica. La lunghezza della stessa dipende dal numero di registri che si utilizzano nella struttura secondo la formula $N = 2^L - 1$ dove L è il numero di registri ed N è la lunghezza della sequenza. Questa struttura non permette che tutti i registri siano a 0, in questo caso l'xor genererebbe sempre uno 0 e non porterebbe a nessun cambiamento nei registri e quindi l'uscita del generatore sarebbe sempre nulla.

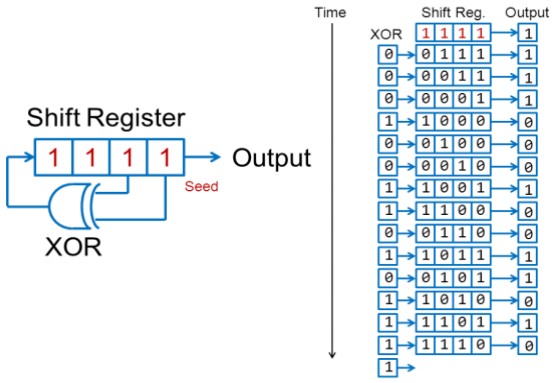


Figura 2.1 Lfsr-4 (riferimento [7] bibliografia)

2.3 POLINOMI DEGLI LFSR

I registri utilizzati come ingresso per l'xor non sono scelti in modo casuale, ma sono espressi da un polinomio modulo 2, cioè i coefficienti di quest'ultimo devono essere uno o zero. Questo prende il nome di polinomio caratteristico o polinomio di retroazione. Per quanto riguarda l'esempio di Figura 1.1, il polinomio caratteristico è x^4+x^3+1 , nel quale gli esponenti indicano i registri da utilizzare come ingressi per l'xor e il numero 1 indica che la sua uscita corrisponde all'ingresso del primo registro. In Figura 2.2 la tabella mostra i polinomi caratteristici di alcuni casi.

Bit	Polinomio caratteristico	Periodo(n° di bit che compongono la sequenza)	Bit	Polinomio caratteristico	Periodo(n° di bit che compongono la sequenza)
L		2^L-1	L		2^L-1
2	x^2+x+1	3	11	$x^{11}+x^9+1$	2˙047
3	x^3+x^2+1	7	12	$x^{12}+x^{11}+x^{10}+x^4+1$	4˙095
4	x^4+x^2+1	15	13	$x^{13}+x^{12}+x^{11}+x^8+1$	8˙191
5	x^5+x^3+1	31	14	$x^{14}+x^{13}+x^{12}+x^2+1$	16˙383
6	x^6+x^5+1	63	15	$x^{15}+x^{14}+1$	32˙762
7	x^7+x^6+1	127	16	$x^{16}+x^{14}+x^{13}+x^{11}+1$	65˙535
8	$x^8+x^6+x^5+x^4+1$	255	17	$x^{17}+x^{14}+1$	131˙071
9	x^9+x^5+1	511	18	$x^{18}+x^{11}+1$	262˙143
10	$x^{10}+x^7+1$	1˙023	19	$x^{19}+x^{18}+x^{17}+x^{14}+1$	524˙287

Figura 2.2 Tabella contente alcuni polinomi generatori

2.4 SOFTWARE

Per realizzare il generatore pseudo-random dobbiamo quindi implementare in VHDL dei registri e un xor. Per specifiche di progetto, la sequenza deve avere una periodicità superiore ai 10 ms e i bit devono avere una frequenza di 10MHz. Ogni 100 ns si avrà un nuovo dato quindi, per soddisfare la periodicità, la sequenza dovrà avere almeno 100˙000 bit. Come detto prima, la lunghezza della sequenza è data dalla formula $N = 2^L - 1$, di conseguenza il numero dei registri dovrà essere diciassette e, consultando la tabella di Figura 2.2, possiamo ricavare i registri da utilizzare come ingressi dell'xor. Quindi in questo caso la sequenza sarà periodica ogni 131˙071 bit e gli ingressi dell'xor sono le uscite del registro diciassette e quattordici.

```
entity prbsgen is
   generic( n : positive );
   port (  en, res, ck : in std_logic;
        u : out std_logic);
end prbsgen;

architecture a of prbsgen is

signal q : std_logic_vector(n downto 0);

begin
   r : for k in 0 to (n-1) generate
      r_k:process(ck,res,en)
      begin
         if res='1' then
            q(k+1)<='1';
         elsif ck'event and ck='1'and en='1' then
            q(k+1)<=q(k);
         end if;
      end process;
   end generate;

   b0:u<=q(n);

   or0:q(0)<=q(17) xor q(14);
end a;
```

Figura 2.3 File VHDL del generatore pseudo-random

La Figura 2.3 mostra l'implementazione del generatore pseudo-random. Di seguito sarà specificato il significato dei segnali di ingresso ed uscita:

- n: costante positiva che rappresenta il numero di registri della struttura; il codice VHDL è stato parametrizzato con questa costante per rendere scalabile la struttura, andrà specificata durante l'inizializzazione del componente con l'utilizzo del comando “generic map”.
- ck: segnale di clock; questo segnale corrisponderà al clock dell' FPGA.
- res: segnale di reset; se è a uno porterà a zero le uscite di tutti i registri che compongo la struttura.
- en: segnale di enable dei registri; per generare bit con una frequenza di 10MHz si è deciso di agire sul segnale di enable dei registri e non di comandare questi ultimi con un clock a 10

MHz. Questo è stato fatto perché generando un segnale di clock tramite una rete combinatoria si potrebbero verificare dei glitch, ciò porterebbe ad un segnale di clock instabile. I glitch sono causati del ritardo di propagazione del segnale lungo la rete combinatoria, il componente a valle dal quale verrà generato il segnale di clock, per alcuni instanti avrà i suoi ingressi instabili e non al valore di regime. Questa instabilità si propagherà in uscita e causerà l'instabilità del segnale di clock. Un'altra possibile soluzione è quella di porre un registro in cascata alla rete combinatoria che genera il clock in modo da non propagare il glitch, ma in questo progetto si è preferito comandare la struttura con il segnale di enable.

- u: segnale di uscita del generatore.

Il processo r_k è il classico processo che descrive un registro con reset asincrono. Come si nota dalla Figura 2.3 è stato parametrizzato con la costante k in modo da generare automaticamente la struttura di 17 registri tramite il comando “generate”. Si noti che quando il reset è attivo si inizializza l'uscita dei registri a 1 e non a 0 per non avere un'uscita nulla come spiegato precedentemente. Quindi la sequenza composta da tutti 1 è il seed iniziale, con questa struttura non è possibile utilizzare un altro seed. Un possibile miglioramento potrebbe essere quello di poter caricare seed differenti nella struttura. Il processo b0 è un semplice buffer di uscita mentre or0 è il processo per descrivere l'xor. Per rendere l'intero generatore scalabile bisognerebbe rendere gli ingressi dell' xor dipendenti da “n” in modo da soddisfare tutti i polinomi caratteristici.

2.5 CARATTERISTICHE DELLA SEQUENZA PSEUDO-RANDOM

Successivamente il generatore è stato simulato e la sequenza di uscita importata in Matlab per poterne calcolare la funzione di autocorrelazione e la sua trasformata. Di seguito sono riportati i grafici.

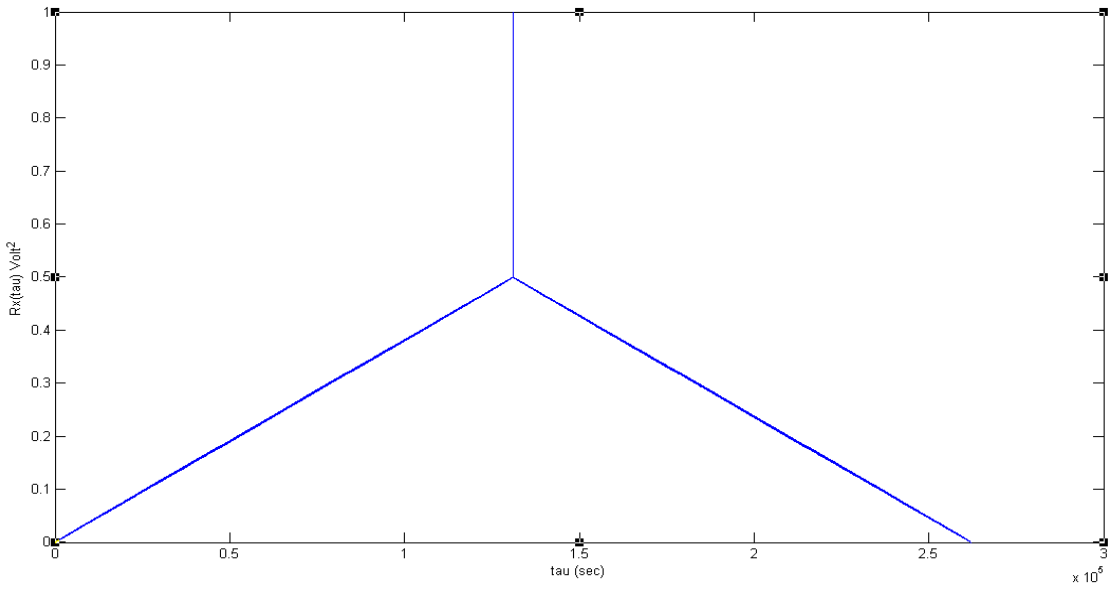


Figura 2.4 Funzione di autocorrelazione

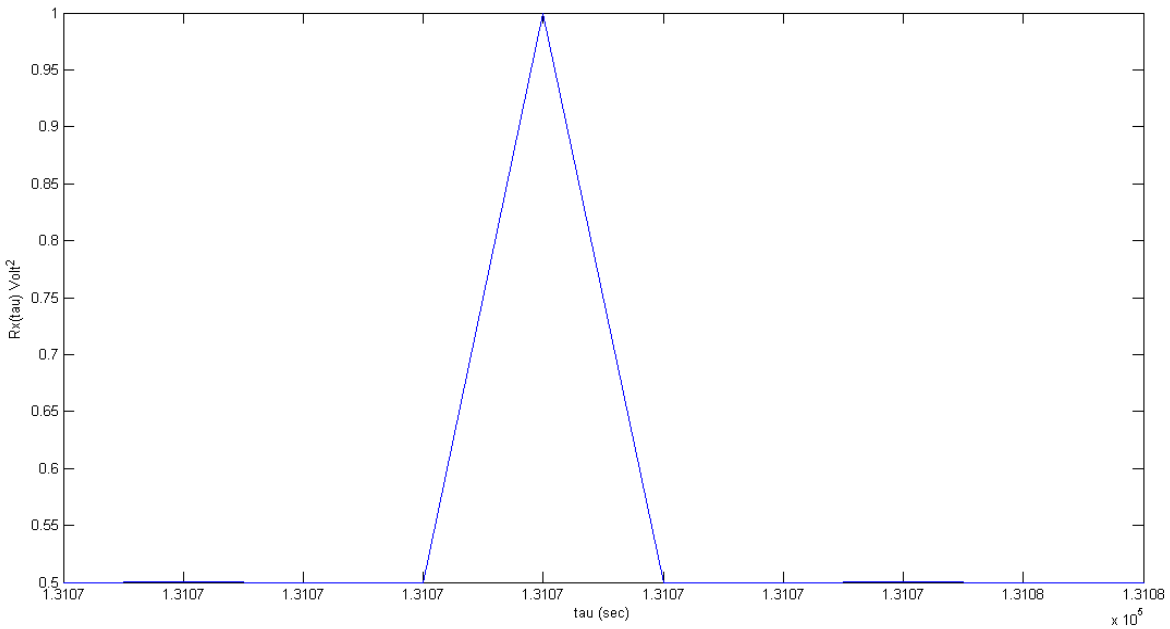


Figura 2.5 Zoom sul picco della funzione di autocorrelazione

Come detto in precedenza, la funzione di autocorrelazione deve tendere ad una Delta di Dirac, quindi la bontà dipende da quanto questa forma triangolare è stretta. La Figura 2.5 mostra che questa specifica è rispettata poiché la differenza tra il primo e il secondo campione nell'ascissa è nulla. Per avere una differenza bisogna considerare anche le decine. Per quanto riguarda l'ordinata il secondo campione dovrebbe essere a zero, il valore di 0.5 e la pendenza lenta tra 0.5 e 0 dipendono dal fatto che la sequenza random è binaria e non numerica. Questo significa che c'è una correlazione non nulla tra un campione e quello successivo.

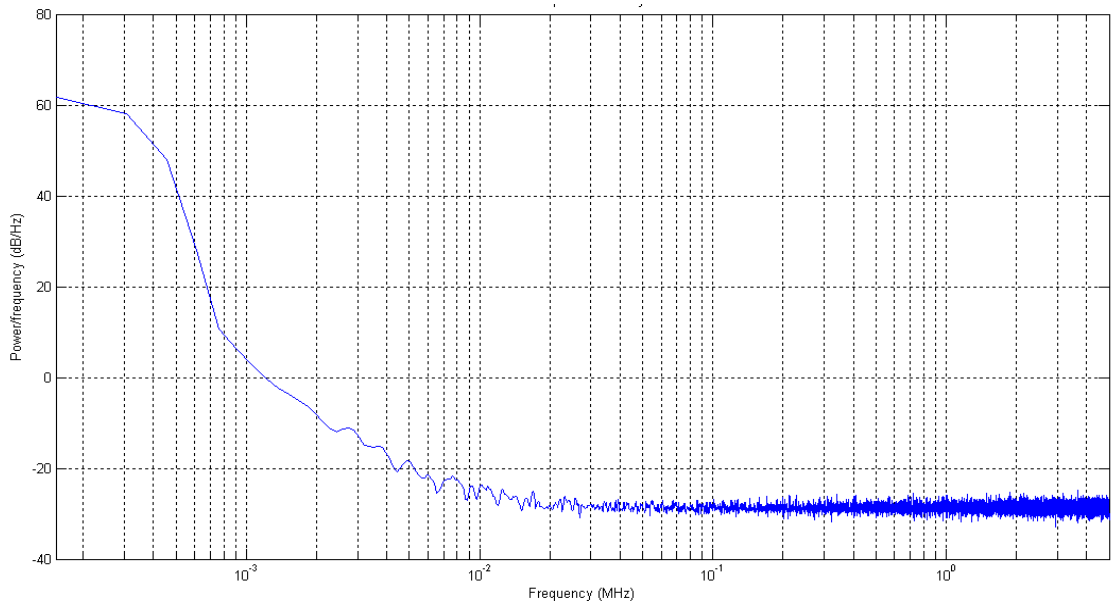


Figura 2.6 Trasformata della funzione di autocorrelazione

Come si nota dal grafico di Figura 2.6 la trasformata può essere considerata un rumore bianco tra 10 kHz e 1 Mhz; per aumentare la banda si può ripetere la sequenza pseudo-random per creare una sequenza più lunga. La Figura 2.5 mostra la trasformata con la sequenza ripetuta dieci volte e si nota un aumento della banda da 1kHz a 1MHz.

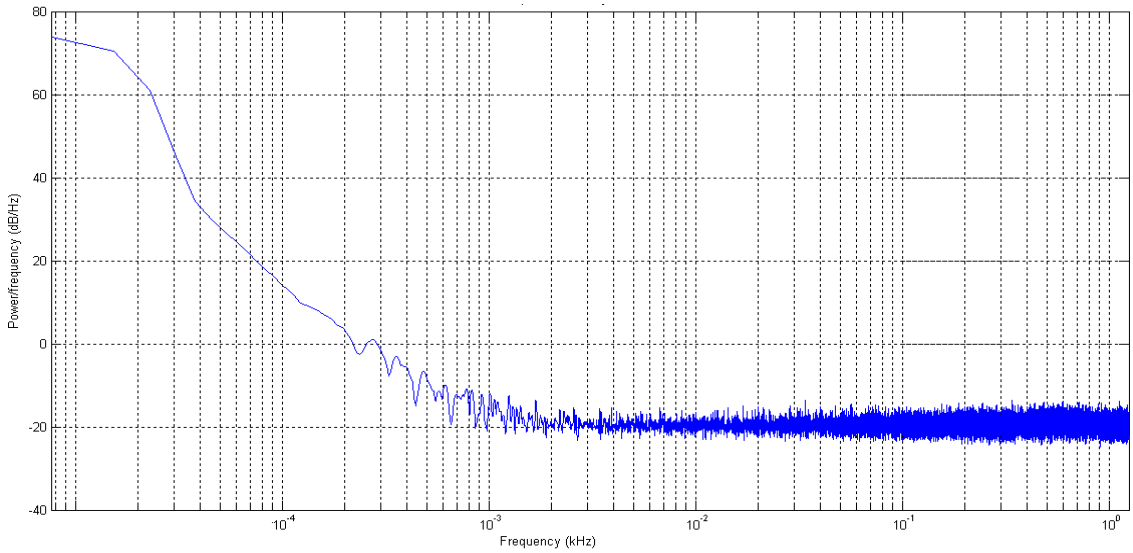


Figura 2.7 Trasformata della funzione di autocorrelazione della sequenza ripetuta

Da specifiche la trasformata della funzione di autocorrelazione deve essere bianca da 1 KHz a 1 Mhz: più la banda è ampia, più il sistema sarà versatile.

CAPITOLO 3

3.1 INTRODUZIONE ALLA CONVOLUZIONE

In matematica, date due funzioni f(t) e g(t) si dice prodotto di convoluzione (o integrale di convoluzione, o in assoluto convoluzione) $\int_{-\infty}^{+\infty} f(\tau)\cdot g(t-\tau)\,d\tau$, e si indica con $f(t)*g(t)$.

La convoluzione mette in relazione l'uscita di un sistema lineare tempo invariante con il suo ingresso e con la sua risposta impulsiva. Per sequenze discrete è definita così:

$$y(n)=x(n)*ht(n)=\sum_{n=-\infty}^{+\infty} ht(m)\cdot x(n-m)$$

dove y(n) rappresenta l'uscita, x(n) l'ingresso e ht(n) la risposta impulsiva del sistema. La lunghezza di y sarà uguale alla lunghezza di x sommata a quella di ht, il tutto diminuito di uno.

3.2 FILTRO FIR

Una possibile struttura per implementare la convoluzione può essere ricavata dall'applicazione diretta dell'equazione in un'architettura a linea di ritardo nota come FIR (finite impulse response) mostrata in Figura 3.1 .

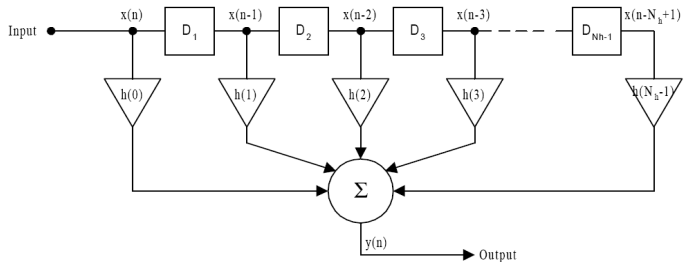


Figura 3.1 Architettura per l'implementazione del processo di convoluzione nella sua forma tempo-discreta per sequenze finite (riferimento [8] bibliografia)

La sequenza pseudo-random è composta da 131˙071 bit, questo comporta un equivalente numero di registri per memorizzarli poiché devono essere tutti disponibili per calcolare la moltiplicazione e altrettanti registri per implementare la linea di ritardo. Per l' FPGA scelta, questa struttura non può essere implementata perché verrebbe superato il numero massimo di logic elements (LE) disponibili nell' FPGA. L'unico modo per realizzarla è in forma ridotta, quindi si deve ricorrere ad un algoritmo

che permette di svolgere la convoluzione su partizioni della sequenza iniziale e di conseguenza avere una struttura più ridotta.

3.2 OVERLAP AND ADD

L'algoritmo utilizzato prende il nome di overlap and add, in questo esempio andremo a dimostrare che l'operazione di convoluzione può essere compiuta anche partizionando l'ingresso e utilizzando particolari accorgimenti (riferimento [8] bibliografia). Si consideri l'esempio in Figura 3.2, la convoluzione tra x(n) "figura a" e h(n) "figura b" è mostrata in "figura c". Le "figure d-f" illustrano invece le convoluzioni tre serie di partizioni consecutive di x(n) e h(n). Ogni convoluzione avrà lunghezza $N_{xi}+N_h-1$, dove con N_{xi} si indica la lunghezza della partizione x_i e N_h la lunghezza di "h". Dovranno essere aggiunti N_h-1 zeri alla sequenza in ingresso per poter calcolare la convoluzione. Nel caso specifico illustrato tale lunghezza è 5, la loro successione dà luogo ad una sequenza di 15 campioni anziché 11 del caso tradizionale. Per raggiungere la lunghezza adeguata è logico assumere che ciascuna sequenza successiva debba sovrapporsi alla precedente di un numero pari a (15-11)/2=2 campioni, come mostrato in figura. Questo valore coincide con N_h-1 . Se si procede alla somma dei campioni che si sovrappongono si ottiene una sequenza identica a quella della convoluzione tradizionale.

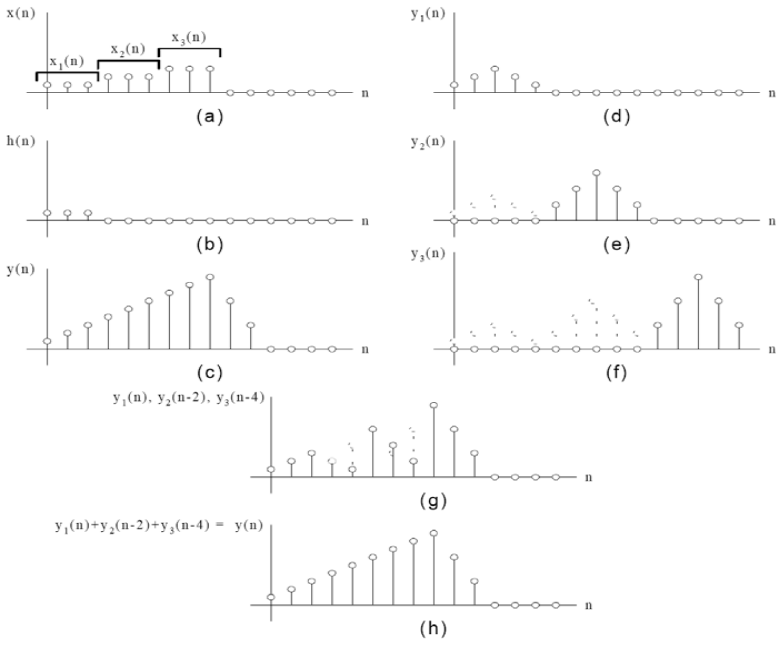


Figura 3.2 Metodo standard nel dominio del tempo (a_c) confrontato con il metodo overlap and add (d-h) (riferimento [8] bibliografia)

Matematicamente la situazione può essere descritta così:

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

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

Cioè, riassumendo quanto detto:

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

$$x_2=\{x(N_{xi}),x(N_{xi}+1),\ldots,x(2\cdot N_{xi}-1),0,0,\ldots,0\}$$

$$x_3=\{x(2\cdot N_{xi}),x(2\cdot N_{xi}+1),\ldots,x(3\cdot N_{xi}-1),0,0,\ldots,0\}$$

$$y_n=\{y_1(0),\ y_1(1),\ldots,\ y_1(N_{xi}-1),\ y_1(N_{xi})+y_2(0),\ y_1(N_{xi}+1)+y_2(1),\ldots,$$
$$y_1(N_y-1)+y_2(N_h-1),\ y_2(N_h)\}$$

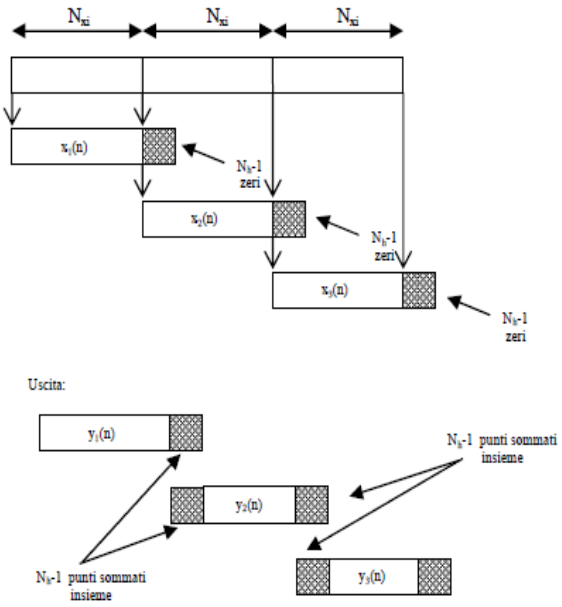


Figura 3.3 Ingressi e uscite del metodo overlap and add (riferimento [8] bibliografia)

3.3 SOFTWARE

Per quanto riguarda questo progetto si ha una sequenza di ingresso con word a 12 bit. La sommatoria è stata implementata sommando due a due i campioni e ripetendo questo metodo con i risultati ottenuti fino ad arrivare ad un unico dato. Ogni livello di somma nel caso peggiore aumenta la lunghezza della word di un bit, quindi si è scelto di avere in uscita una word a 16 bit che corrisponde alla lunghezza delle celle della memoria in modo da avere un risultato per cella. Questo comporta una dimensione della FIR pari a 15 registri e 16 moltiplicatori.

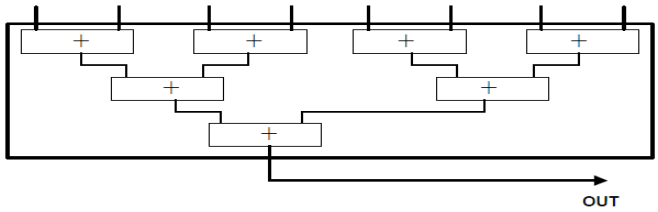


Figura 3.4 Struttura utilizzata per effettuare la somma

Quindi ricordandoci dell'algoritmo overlap and add avremo partizioni della sequenza pseudo-random di 16 bit che andranno memorizzati nel package per il calcolo della convoluzione, successivamente andranno caricati sequenzialmente le word dei dati provenienti dall'ADC aggiungendo sedici zeri in coda alla sequenza per il corretto calcolo della convoluzione. In questo modo si avranno convoluzioni parziali lunghe (131˙071+16)-1=131˙086. I campioni corretti saranno quindi 131˙086-131070=16, gli altri sono tutti campioni da sovrapporre e sommare a quelli della convoluzione successiva. Questo procedimento si ripeterà fino al caricamento di tutti i bit della sequenza pseudo-random.

```
entity conv is
   generic( n : positive :=12; m : positive:= 16);
   port (   res,res_0,res_x, ck,x,en_y : in std_logic;
         u_prec : in std_logic_vector(15 downto 0);
         en_x : in std_logic_vector((m-1) downto 0);
         y : in std_logic_vector((n-1) downto 0);
         u : out std_logic_vector((n+3) downto 0));
end conv;
```

Figura 3.5 File VHDL relativo all'entity del package della convoluzione

La Figura 3.5 mostra l'implementazione della convoluzione. Di seguito sarà specificato il significato dei segnali di ingresso ed uscita:

- n: costante positiva che indica il numero di bit che compongo la word di ingresso; il codice VHDL è stato parametrizzato con questa costante per rendere scalabile la struttura, essa andrà specificata durante l'inizializzazione del componete.

- m: costante positiva che indica il numero di registri e quindi la lunghezza della struttura.
- ck: segnale di clock; questo segnale corrisponderà al clock dell'FPGA.
- res: segnale di reset; se è a uno porterà a zero le uscite di tutti i registri che compongo la struttura.
- res_0: segnale di reset; se è ad uno porterà a zero il primo registri della linea di ritardo.
- en_y: segnale di enable; questo segnale abilita esclusivamente i registri della linea di ritardo.
- x: ingresso per i bit della sequenza pseudo-random; questo segnale è collegato a tutti i registri utilizzati per memorizzare le partizioni da 16 bit della sequenza pseudo-random, verrà utilizzato un segnale di enable per caricare il valore nel giusto registro.
- en_x: vettore di 16 bit utilizzato per abilitare i registri della sequenza pseudo-random; ogni bit corrisponde ad un registro: il bit zero abilita il registro zero, il bit uno abilita il registro uno. Sarà l'unità di controllo che dovrà generare il giusto vettore per salvare il bit nel registro corretto.
- y: ingresso del segnale proveniente dall'ADC;
- u_prec: ingresso utilizzato per caricare il campione della precedente convoluzione che va sommato con quello successivo per implementare l'algoritmo overlap and add. Ovviamente per avere il risultato corretto quando non ci sono campioni da sommare tale segnale dovrà essere nullo.
- u: risultato della convoluzione;

```
signal xm : std_logic_vector((m-1) downto 0);
signal q,z : std_logic_vector(((n*m)-1) downto 0);
signal z_0 : std_logic_vector(103 downto 0);
signal z_1 : std_logic_vector(55 downto 0);
signal z_2 : std_logic_vector(29 downto 0);
signal z_3 : std_logic_vector(15 downto 0);
signal ress_0,ress_x :std_logic;
```

Figura 3.6 File VHDL relativo ai segnali interni utilizzati nel package della convoluzione

Per quanto riguarda i segnali interni:

- xm: vettore di 16 bit che contiene le partizioni della sequenza pseudo-random caricate nella struttura.
- q: vettore contenente le uscite dei registri della linea di ritardo.
- z: vettore contenente il risultato della moltiplicazione del bit con la word.
- z_0,z_1,z_2,z_3: vettori contenti i risultati delle rispettive linee di somme.

- ress_0, ress_x: segnale di reset utilizzato rispettivamente per il primo registro e per quelli utilizzati per salvare i bit della sequenza random, il loro utilizzo sarà spiegato successivamente.

```
or0: ress_0<=res or res_0;

r0:process(ck,ress_0,en_y)
begin
   if ress_0='1' then
      q((n-1) downto 0)<=std_logic_vector(to_unsigned(0,n));
   elsif ck'event and ck='1' and en_y='1' then
      q((n-1) downto 0)<=y;
   end if;
end process;

r : for k in 1 to (m-1) generate
   r_k:process(ck,res,en_y)
   begin
      if res='1' then
         q(((n*(k+1))-1) downto (n*k))<=std_logic_vector(to_unsigned(0,n));
      elsif ck'event and ck='1' and en_y='1' then
         q(((n*(k+1))-1) downto (n*k))<=q(((n*k)-1) downto (n*(k-1)));
      end if;
   end process;
end generate;

mx : for k in 0 to (m-1) generate
      mb_k:process(xm,q)
      begin
         if xm(k)='1' then
            z(((n*(k+1))-1) downto (n*k))<=q(((n*(k+1))-1) downto (n*k));
         else
            z(((n*(k+1))-1) downto (n*k))<=std_logic_vector(to_unsigned(0,n));
         end if;
      end process;
   end generate;

or1: ress_x<=res or res_x;


rx : for k in 0 to (m-1) generate
   rx_k:process(ck,ress_x,en_x)
   begin
      if ress_x='1' then
         xm(k)<='0';
      elsif ck'event and ck='1' and en_x(k)='1' then
         xm(k)<=x;
      end if;
   end process;
end generate;

sum0: for k in 0 to 7 generate
   sum0_k:z_0((k*(n+1))+n downto (k*(n+1)))<=std_logic_vector(resize(unsigned(z((k*(n*2))+(n-1) downto k*(n*2))),n+1)+
                                                              resize(unsigned(z((k*(n*2))+(n+(n-1)) downto (k*(n*2))+n)),n+1));
end generate;

sum1: for k in 0 to 3 generate
   sum1_k:z_1((k*(n+2))+(n+1) downto (k*(n+2)))<=std_logic_vector(resize(unsigned(z_0((k*((n+1)*2))+n downto k*((n+1)*2))),n+2)+
                                                                  resize(unsigned(z_0((k*((n+1)*2))+(n+(n+1)) downto (k*((n+1)*2))+(n+1))),n+2))
end generate;

sum2: for k in 0 to 1 generate
   sum2_k:z_2((k*(n+3))+(n+2) downto (k*(n+3)))<=std_logic_vector(resize(unsigned(z_1((k*((n+2)*2))+(n+1) downto k*((n+2)*2))),n+3)+
                                                                  resize(unsigned(z_1((k*((n+2)*2))+(n+(n+3)) downto (k*((n+2)*2))+(n+2))),n+3))
end generate;

sum3: z_3((n+3) downto 0)<=std_logic_vector(resize(unsigned(z_2((n+2) downto 0)),n+4)+resize(unsigned(z_2(((n+3)*2)-1 downto (n+3))),n+4));

s0:u<=z_3+u_prec;
```

Figura 3.7 File VHDL relativo all'implementazione del package della convoluzione

Il processo or0 implementa un or tra il segnale res e res_0 in modo da resettare il registro zero sia con il reset di sistema sia con il reset dedicato. Lo stesso ragionamento è stato fatto per l'or1 ma in questo caso riguarda i registri della sequenza pseudo-random. R0 implementa un registro, il quale è stato aggiunto a quelli necessari alla struttura del FIR in modo da avere sempre disponibile la word una volta caricata dalla memoria ed eventualmente per aggiungere degli zeri in coda alla sequenza, utilizzando il segnale res_0. Il processo r genera i registri della linea di ritardo in modo del tutto

analogo a quello visto per il generatore pseudo-random, lo stesso ragionamento è stato utilizzato anche per rx il quale genera i registri dove memorizzare la partizione della sequenza pseudo-random. Mx svolge l'operazione di moltiplicazione che si riduce ad una semplice scelta tra la word e un vettore nullo in base al valore del bit della sequenza pseudo-random. Sum0, sum1, sum2, sum3 implementano i vari livelli di somma. Per questi processi va fatta una precisazione: per come è definita l'operazione di somma nella libreria "numeric.std" i due addendi devono avere già le dimensioni del risultato anche se composti da un numero di bit inferiore, per questo è stata utilizzata la funzione "resize" che ridimensiona i vettori degli addendi. A questo punto bisogna effettuare una conversione perché l'argomento della funzione "resize" deve essere un "unsigned", mentre i vettori in ingresso sono degli "std_logic_vector". Il risultato viene poi riconvertito ad uno "std_logic_vector". Infine, il processo s0 svolge la somma tra il risultato della attuale convoluzione con il quello della precedente. La struttura è parzialmente parametrizzata da "n" e "m", per renderla totalmente scalabile bisogna rendere i vari livelli di somma parametrizzati in base alla lunghezza della struttura "m".

CAPITOLO 4

4.1 INTRODUZIONE AL NUCLEO DI CALCOLO

Il sistema da implementare dovrà quindi generare la sequenza pseudo-random, generare il clock per comandare l'ADC e acquisire i dati provenienti dallo stesso. Tutto questo viene fatto in real-time. Successivamente off-line verranno svolti tutti i calcoli e memorizzati i risultati. In Figura 4.1 viene mostrata l'entity che è stata creata per svolgere tutto questo.

```
entity syst is
   port (  start,reset,ck : in std_logic;
      y : in std_logic_vector(11 downto 0);
      data : inout std_logic_vector (15 downto 0);
      add : out std_logic_vector(18 downto 0);
      lb,ub,ce,oe,we,ck_adc,busy,x  : out std_logic);
end syst;
```

Figura 4.1 Codice VHDL relativa all'entity del sistema

Di seguito verranno spiegati i segnali di ingresso e uscita:

- reset: segnale di reset; se è ad uno resetta l'intero sistema, può essere associato ad un pulsante.
- ck: clock di sistema; corrisponderà al clock di funzionamento dell'FPGA.
- start: segnale di inizio dell'elaborazione, se è ad uno il sistema passa dallo stato di standby a quello di funzionamento. Anche questo può essere associato ad un pulsante.
- y: segnale di ingresso associato ai dati provenienti dall'ADC.
- data: segnale associato al bus dati della memoria.
- add: segnale associato ai bit di indirizzamento della memoria.
- Lb, ub, ce, oe, we: bit di comando della memoria.
- ck_adc: segnale associato al clock generato per comandare l'ADC.
- busy: se ad uno indica che il sistema sta elaborando i dati, può essere associato ad un led.
- x: segnale di uscita associato alla sequenza pseudo-random generata.

4.2 NUCLEO DI CALCOLO

Per implementare il sistema completo, oltre al generatore pseudo-random e al blocco che svolge la convoluzione, sono stati utilizzati altri componenti. In Figura 4.2 è rappresentato lo schema a blocchi del sistema e di seguito sono spiegati i componenti aggiuntivi.

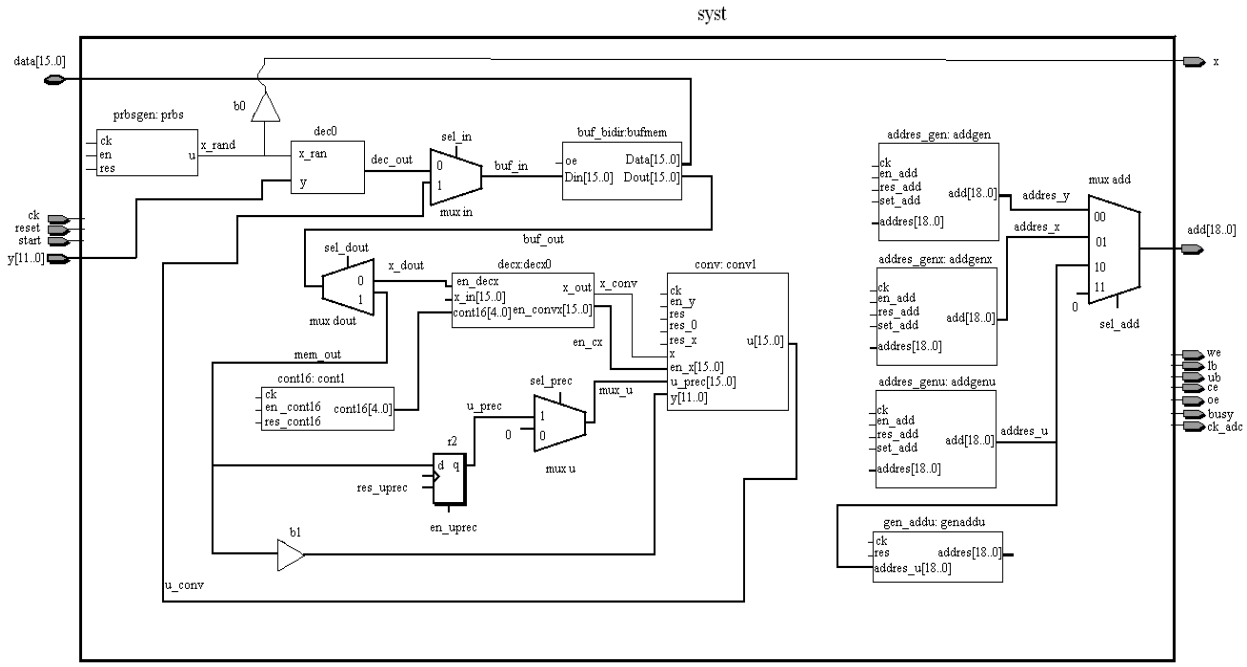


Figura 4.2 Schema a blocchi del nucleo di calcolo.

I segnali di Figura 4.2 che non sono collegati appartengono alla macchina a stati, la quale è stata omessa per non appesantire la figura. Il componente “addres_gen” è un semplice contatore utilizzato per generare gli indirizzi della memoria, può essere resettato tramite il segnale “res_add” ed è abilitato dal segnale “en_add”. Il contatore può essere settato ad un indirizzo specifico utilizzando il segnale “set_add” e applicando l'indirizzo voluto all'ingresso “addset_y”. Per un corretto funzionamento bisogna applicare al segnale “addset_y” l'indirizzo precedente a quello voluto e portare ad uno sia “set_add” che “en_add”, in questo modo nel contatore viene caricato l'indirizzo corretto.

```
entity addres_gen is
   port (   res_add, ck, en_add,set_add: in std_logic;
         addres: in std_logic_vector(18 downto 0);
         add : out std_logic_vector(18 downto 0));
end addres_gen;
```

Figura 4.3 Codice VHDL relativo all'entity di addres_gen

Questa struttura è stata utilizzata anche nei componenti “addres_genx” e “addres_genu” con alcune piccole differenze. “Addres-genx” è un contatore all'indietro: quando resettato, l'indirizzo di partenza è il 131˙070. Questo contatore è utilizzato esclusivamente per generare gli indirizzi nella fase di caricamento dei bit random nella struttura della convoluzione. E' un contatore all'indietro perché va caricato x(-n), come spiegato in precedenza, quindi se la sequenza è lunga 131˙071 e memorizziamo un bit per cella, partendo da quella con indirizzo zero, il primo indirizzo da utilizzare per caricare le partizioni sarà appunto 131˙070. “Addres_genu” è un contatore in avanti utilizzato per generare gli indirizzi dove salvare il risultato o caricare i campioni delle convoluzioni precedenti da sommare ai campioni correnti. Unica differenza: se resettato parte dall'indirizzo 131˙077, il quale rappresenta il primo indirizzo utile dopo quelli utilizzati per l'acquisizione, tenendo conto dello sfasamento dei dati provocato dall'ADC. Tramite un “mux” viene scelto quale generatore di indirizzi sarà collegato alla memoria. Si è scelto di utilizzare tre generatori di indirizzi per avere una macchina a stati più semplice dato che, abilitando o meno i contatori, si riesce ad avere un buon controllo sugli indirizzi senza doverli impostare tramite gli appositi ingressi. “Gen_addu” è un componente che genera l'indirizzo dal quale si dovranno sommare i campioni della convoluzione. Questi indirizzi andranno settati nel contatore “add_genu”. L'indirizzo viene calcolato come l'indirizzo corrente meno 131˙070, questo valore deriva dall'algoritmo dell' “overlap and add” e rappresenta il numero di campioni da sommare. É stato inserito un registro che campiona l'indirizzo corrente in modo da avere un valore stabile su cui calcolare il nuovo indirizzo.

```
entity gen_addu is
   port (   res,ck : in std_logic;
         addres_u : in std_logic_vector(18 downto 0);
         addres : out std_logic_vector(18 downto 0));
end gen_addu;

architecture a of gen_addu is

signal addres_in : std_logic_vector(18 downto 0);

begin
   r0:process(addres_u,res,ck)
   begin
         if res='1' then
            addres_in<=conv_std_logic_vector(131077,19);
         elsif ck'event and ck='1' then
            addres_in<=addres_u;
         end if;

   end process;

   r1:process(addres_in)
   begin
      if addres_in=conv_std_logic_vector(131077,19) then
         addres<=conv_std_logic_vector(131077,19);
      else
         addres<=conv_std_logic_vector((conv_integer(addres_in)-131072),19);
      end if;
   end process;
end a;
```

Figura 4.4 Codice VHDL relativo all'entity di “gen_addu”

Il componente “buf_bidir” è un'interfaccia per la memoria, dato che il segnale “data” è bidirezionale questo buffer lo scompone in due segnali: uno di ingresso “buf_in”, utilizzato quando si legge la memoria, uno di uscita “buf_out” usato quando si scrive in memoria. I due segnali vengono selezionati tramite un bit OE (“buf_oe”). Al segnale “buf_in” è collegato un mux in modo da scegliere se i dati da scrivere sono quelli provenienti dall'ADC e dalla sequenza pseudo-random(“dec_out”) o quelli che risultano dalla convoluzione(“u_conv”). “Dec0” è un semplice processo che unisce in un unico vettore i dati dell'ADC al bit della sequenza pseudo-random. Esso viene utilizzato nella fase di acquisizione per sfruttare al meglio la memoria poiché si è scelto di memorizzare nei primi dodici bit della cella di memoria i dati dell'ADC e nel MSB il bit della sequenza pseudo-random. Anche il segnale “buf_out” è collegato ad un mux per scegliere se i dati da leggere sono quelli della sequenza random (“x_dout”) o i dati dell'ADC e i risultati delle convoluzioni (mem_out). “Decx” è un componente, che oltre a selezionare solo i MSB dal dato della memoria per ricavare il bit random, genera anche il segnale di enable per memorizzare nella posizione corretta tale bit. Questo viene calcolato in base al valore del segnale “cont_16”.

```
entity decx is
   port (x_in : in std_logic_vector(15 downto 0);
      en_decx : in std_logic;
      cont16: in std_logic_vector(4 downto 0);
      x_out : out std_logic;
      en_convx : out std_logic_vector(15 downto 0));

end decx;

architecture a of decx is

   r0:process(cont16)
      begin
         case cont16 is
            when conv_std_logic_vector(0,5)  => en_convx1<="0000000000000001";
            when conv_std_logic_vector(1,5)  => en_convx1<="0000000000000010";
            when conv_std_logic_vector(2,5)  => en_convx1<="0000000000000100";
            when conv_std_logic_vector(3,5)  => en_convx1<="0000000000001000";
            when conv_std_logic_vector(4,5)  => en_convx1<="0000000000010000";
            when conv_std_logic_vector(5,5)  => en_convx1<="0000000000100000";
            when conv_std_logic_vector(6,5)  => en_convx1<="0000000001000000";
            when conv_std_logic_vector(7,5)  => en_convx1<="0000000010000000";
            when conv_std_logic_vector(8,5)  => en_convx1<="0000000100000000";
            when conv_std_logic_vector(9,5)  => en_convx1<="0000001000000000";
            when conv_std_logic_vector(10,5) => en_convx1<="0000010000000000";
            when conv_std_logic_vector(11,5) => en_convx1<="0000100000000000";
            when conv_std_logic_vector(12,5) => en_convx1<="0001000000000000";
            when conv_std_logic_vector(13,5) => en_convx1<="0010000000000000";
            when conv_std_logic_vector(14,5) => en_convx1<="0100000000000000";
            when conv_std_logic_vector(15,5) => en_convx1<="1000000000000000";
            when others => en_convx1<=conv_std_logic_vector(0,16);
         end case;
      end process;
```

Figura 4.5 Codice VHDL relativo all'entity di “gen_addu” e ad una parte dell'implementazione

Il componente cont16 è un contatore che viene utilizzato dal componete “decx” per generare il corretto segnale di enable e dalla macchina a stati nella fase di caricamento dei bit, per capire

quando la struttura della convoluzione è stata completamente caricata. Il segnale “mem_out” corrisponde ai dati dell'ADC o ai campioni da sommare per la convoluzione quindi è collegato tramite un buffer (b1) all'ingresso “y” o all'ingresso “u_prec” del package della convoluzione. L'ingresso “u_prec” non è direttamente collegato a “mem_out” ma ad un mux per selezionare il dato precedente o lo zero in caso non ci siano campioni da sovrapporre. Tra il mux e “mem-out” c'è un registro per rendere stabile il campione da sommare anche in caso la memoria non sia più indirizzata correttamente o comunque si voglia utilizzare per altri dati.

4.3 MACCHINA A STATI DEL NUCLEO DI CALCOLO

La macchina a stati del nucleo di calcolo è costituita da un registro e da una rete logica che genera lo stato futuro e tutti i segnali di controllo. Gli stati sono divisi in stato iniziale “idle”, stati di acquisizione, stati di caricamento della sequenza random e stati per l'elaborazione dei dati. La Figura 4.5 mostra la fase di acquisizione, ora spiegheremo stato per stato il funzionamento:

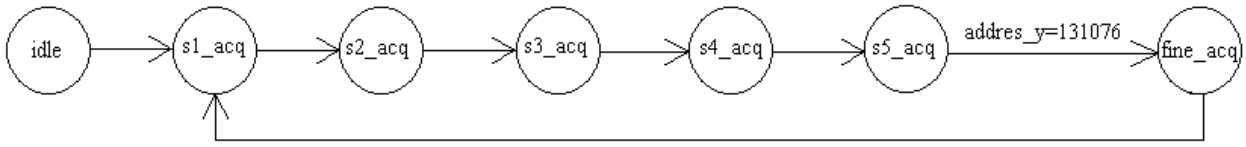


Figura 4.6 Diagramma degli stati relativi all'acquisizione

```
when s1_acq =>en_gen<='0';res_gen<='0';buf_oe<='1';sel_in<='0';res_add<='0';en_add<='0';set_add<='0';
      addset_y<=conv_std_logic_vector(0,19);lb<='0';ub<='0';ce<='0';oe<='1';we<='1';ck_adc<='0';busy<='0';
      res_16<='1';en_16<='0';sel_add<="00";res_addx<='1';en_addx<='0';set_addx<='0';sel_dout<='0';
      en_decx<='0';res_addu<='1';en_addu<='0';res_conv<='1';res_c0<='0';res_x<='0';en_cy<='0';sel_prec<='0';
      set_addu<='0';en_uprec<='0'; res_uprec<='1';res_ugen<='1';
     ns<=s2_acq;
when s2_acq =>en_gen<='0';res_gen<='0';buf_oe<='1';sel_in<='0';res_add<='0';en_add<='0';set_add<='0';
      addset_y<=conv_std_logic_vector(0,19);lb<='0';ub<='0';ce<='0';oe<='1';we<='1';ck_adc<='0';busy<='0';
      res_16<='1';en_16<='0';sel_add<="00";res_addx<='1';en_addx<='0';set_addx<='0';sel_dout<='0';
      en_decx<='0';res_addu<='1';en_addu<='0';res_conv<='1';res_c0<='0';res_x<='0';en_cy<='0';sel_prec<='0';
      set_addu<='0';en_uprec<='0'; res_uprec<='1';res_ugen<='1';
     ns<=s3_acq;
when s3_acq =>en_gen<='0';res_gen<='0';buf_oe<='1';sel_in<='0';res_add<='0';en_add<='0';set_add<='0';
      addset_y<=conv_std_logic_vector(0,19);lb<='0';ub<='0';ce<='0';oe<='1';we<='1';ck_adc<='0';busy<='0';
      res_16<='1';en_16<='0';sel_add<="00";res_addx<='1';en_addx<='0';set_addx<='0';sel_dout<='0';
      en_decx<='0';res_addu<='1';en_addu<='0';res_conv<='1';res_c0<='0';res_x<='0';en_cy<='0';sel_prec<='0';
      set_addu<='0';en_uprec<='0'; res_uprec<='1';res_ugen<='1';
     ns<=s4_acq;
when s4_acq =>en_gen<='0';res_gen<='0';buf_oe<='1';sel_in<='0';res_add<='0';en_add<='0';set_add<='0';
      addset_y<=conv_std_logic_vector(0,19);lb<='0';ub<='0';ce<='0';oe<='1';we<='1';ck_adc<='1';busy<='0';
      res_16<='1';en_16<='0';sel_add<="00";res_addx<='1';en_addx<='0';set_addx<='0';sel_dout<='0';
      en_decx<='0';res_addu<='1';en_addu<='0';res_conv<='1';res_c0<='0';res_x<='0';en_cy<='0';sel_prec<='0';
      set_addu<='0';en_uprec<='0'; res_uprec<='1';res_ugen<='1';
     ns<=s5_acq;
when s5_acq =>en_gen<='1';res_gen<='0';buf_oe<='1';sel_in<='0';res_add<='0';en_add<='1';set_add<='0';
      addset_y<=conv_std_logic_vector(0,19);lb<='0';ub<='0';ce<='0';oe<='1';we<='0';ck_adc<='1';busy<='0';
      res_16<='1';en_16<='0';sel_add<="00";res_addx<='1';en_addx<='0';set_addx<='0';sel_dout<='0';
      en_decx<='0';res_addu<='1';en_addu<='0';res_conv<='1';res_c0<='0';res_x<='0';en_cy<='0';sel_prec<='0';
      set_addu<='0';en_uprec<='0'; res_uprec<='1';res_ugen<='1';
```

Figura 4.7 Codice VHDL relativo agli stati di acquisizione

- s1_acq: in questo stato il generatore random non viene più resettato in modo che una volta abilitato il segnale “en_gen” possa funzionare correttamente senza rimanere nella condizione iniziale. Si comanda il “buf_bidir” in modo che si possa scrivere il dato sulla memoria tramite “buf_oe”=1 e si sceglie di salvare la sequenza random e i dati dell'ADC con “sel_in”=0. Si comanda la memoria in modo che sia pronta ad essere scritta e si genera l'indirizzo scegliendo “addres_gen” tramite “sel_add”=00. Il clock dell'ADC è tenuto a zero per generare il semi periodo basso. Tutti i componenti che non servono sono resettati.
- s2_acq, s3_acq: in questi stati rimane tutto inalterato in modo da tenere stabile l'indirizzo per la memoria.
- s4_acq: rimane tutto invariato tranne il clock dell'ADC che viene portato ad uno per generare il semi periodo alto.
- s5_acq: qui si abilita il segnale di enable del generatore random (en_gen=1) in modo da avere un nuovo bit, e si abilita l'enable del generatore di indirizzi (en_add=1) per avere l'indirizzo incrementato al ciclo successivo. Inoltre si abilita anche il write enable della memoria (we=0 attivo basso) in modo che il dato venga memorizzato.

Si noti che abilitando il generatore random nell'ultimo stato si genera un dato ogni 10 MHz, come da specifiche. Nello stato “s5_acq” si controlla il vettore degli indirizzi e, se sono stati memorizzati tutti i dati, si passa all'elaborazione di questi ultimi oppure si ripetono ciclicamente questi stati fino al raggiungimento della giusta cella di memoria.

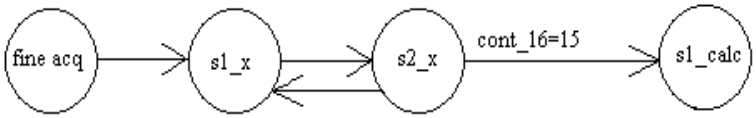


Figura 4.8 Diagramma degli stati relativi al caricamento della sequenza pseudo-random

```
when fine_acq=>en_gen<='0';res_gen<='1';buf_oe<='0';sel_in<='1';res_add<='1';en_add<='0';set_add<='0';
      addset_y<=conv_std_logic_vector(0,19);lb<='0';ub<='0';ce<='0';oe<='0';we<='1';ck_adc<='0';busy<='1';
      res_16<='1';en_16<='0';sel_add<="01";res_addx<='0';en_addx<='0';set_addx<='0';sel_dout<='0';
      en_decx<='0';res_addu<='0';en_addu<='0';res_conv<='1';res_c0<='0';res_x<='0';en_cy<='0';sel_prec<='0'
      set_addu<='0';en_uprec<='0'; res_uprec<='0';res_ugen<='0';
   ns<=s1_x;
when s1_x =>en_gen<='0';res_gen<='1';buf_oe<='0';sel_in<='1';res_add<='1';en_add<='0';set_add<='0';
      addset_y<=conv_std_logic_vector(0,19);lb<='0';ub<='0';ce<='0';oe<='0';we<='1';ck_adc<='0';busy<='1';
      res_16<='0';en_16<='0';sel_add<="01";res_addx<='0';en_addx<='0';set_addx<='0';sel_dout<='0';
      en_decx<='0';res_addu<='0';en_addu<='0';res_conv<='0';res_c0<='0';res_x<='0';en_cy<='0';sel_prec<='0'
      set_addu<='0';en_uprec<='0'; res_uprec<='0';res_ugen<='0';
   ns<=s2_x;
   when s2_x=>en_gen<='0';res_gen<='1';buf_oe<='0';sel_in<='1';res_add<='1';en_add<='0';set_add<='0';
      addset_y<=conv_std_logic_vector(0,19);lb<='0';ub<='0';ce<='0';oe<='0';we<='1';ck_adc<='0';busy<='1';
      res_16<='0';en_16<='1';sel_add<="01";res_addx<='0';en_addx<='1';set_addx<='0';sel_dout<='0';
      en_decx<='1';res_addu<='0';en_addu<='0';res_conv<='0';res_c0<='0';res_x<='0';en_cy<='0';sel_prec<='0'
      set_addu<='0';en_uprec<='0'; res_uprec<='0';res_ugen<='0';
```

Figura 4.9 Codice VHDL relativo agli stati per il caricamento della sequenza random nella struttura per il calcolo della convoluzione

Ricordando che la struttura della convoluzione è composta da sedici registri, dovremmo caricare in tale struttura gli ultimi sedici bit della sequenza random.

- fine_acq: in questo stato il segnale “buf_bidir” viene settato per leggere la memoria (buf_oe=0) inoltre si comanda la memoria per essere letta. Viene scelto il dato da leggere (sel_dout=0). Si seleziona l' “addres_genx” (sel_add=01) per generare gli indirizzi delle celle dove sono contenuti i bit della sequenza pseudo-random e viene disabilitato “res_addx”=0. Il segnale “busy” viene abilitato per indicare che è iniziata l'elaborazione dei dati.
- s1_acq: in questo stato si aspetta che la memoria restituisca il dato contenuto nella cella e si disabilita il reset di “cont16” (res_16=0).
- s2_acq: si abilita “cont16” (en_16=1) per contare quanti bit vengono caricati e viene abilitato anche il generatore di indirizzi (en_addx=1) per avere un nuovo indirizzo al ciclo successivo. Viene abilitato il “decx” (en_decx=1) in modo da generare il segnale di enable corretto per memorizzare il bit della sequenza pseudo-random nel registro corretto della struttura per la convoluzione.

Ora si controlla il valore di “cont16”, se sono stati caricati 16 dati si passa all'elaborazione, altrimenti si ripetono ciclicamente questi ultimi due stati. Si noti che utilizzando un diverso generatore di indirizzi per i bit della sequenza random, al ciclo successivo quando dovranno essere caricati altri sedici bit il contatore sarà già all'indirizzo corretto senza dover utilizzare degli stati

aggiuntivi per il caricamento dell'indirizzo corretto.

La fase di elaborazione si divide in due: la prima parte dove vengono caricati i dati dell'ADC e viene memorizzato il risultato della convoluzione, la seconda parte dove oltre a caricare i dati dell'ADC dovrà essere caricato anche il valore del campione precedentemente calcolato e successivamente si potrà salvare il risultato della convoluzione.

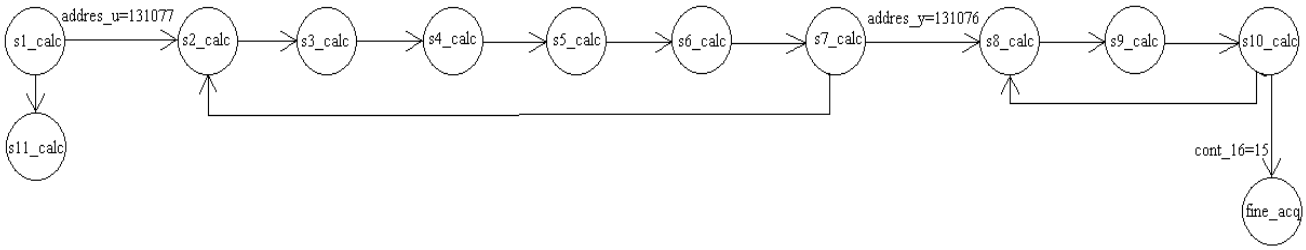


Figura 4.10 Diagramma degli stati relativi al calcolo della convoluzione nel primo ciclo

```
when s1_calc=>en_gen<='0';res_gen<='1';buf_oe<='0';sel_in<='1';res_add<='0';en_add<='1';set_add<='1';
      addset_y<=conv_std_logic_vector(5,19);lb<='0';ub<='0';ce<='0';oe<='0';we<='1';ck_adc<='0';busy<='1';
      res_16<='1';en_16<='0';sel_add<="00";res_addx<='0';en_addx<='0';set_addx<='0';sel_dout<='1';
      en_decx<='0';res_addu<='0';en_addu<='0';res_conv<='0';res_c0<='0';res_x<='0';en_cy<='0';sel_prec<='0';
      set_addu<='0';en_uprec<='0'; res_uprec<='0';res_ugen<='0';
   if addres_u=conv_std_logic_vector(131077,19) then
      ns<=s2_calc;
   else
      ns<=s11_calc;
   end if;
when s2_calc=>en_gen<='0';res_gen<='1';buf_oe<='0';sel_in<='1';res_add<='0';en_add<='0';set_add<='0';
      addset_y<=conv_std_logic_vector(0,19);lb<='0';ub<='0';ce<='0';oe<='0';we<='1';ck_adc<='0';busy<='1';
      res_16<='1';en_16<='0';sel_add<="00";res_addx<='0';en_addx<='0';set_addx<='0';sel_dout<='1';
      en_decx<='0';res_addu<='0';en_addu<='0';res_conv<='0';res_c0<='0';res_x<='0';en_cy<='0';sel_prec<='0';
      set_addu<='0';en_uprec<='0'; res_uprec<='0';res_ugen<='0';
   ns<=s3_calc;
when s3_calc=>en_gen<='0';res_gen<='1';buf_oe<='0';sel_in<='1';res_add<='0';en_add<='0';set_add<='0';
      addset_y<=conv_std_logic_vector(0,19);lb<='0';ub<='0';ce<='0';oe<='0';we<='1';ck_adc<='0';busy<='1';
      res_16<='1';en_16<='0';sel_add<="00";res_addx<='0';en_addx<='0';set_addx<='0';sel_dout<='1';
      en_decx<='0';res_addu<='0';en_addu<='0';res_conv<='0';res_c0<='0';res_x<='0';en_cy<='1';sel_prec<='0';
      set_addu<='0';en_uprec<='0'; res_uprec<='0';res_ugen<='0';
   ns<=s4_calc;
when s4_calc=>en_gen<='0';res_gen<='1';buf_oe<='0';sel_in<='1';res_add<='0';en_add<='0';set_add<='0';
      addset_y<=conv_std_logic_vector(0,19);lb<='0';ub<='0';ce<='0';oe<='0';we<='1';ck_adc<='0';busy<='1';
      res_16<='1';en_16<='0';sel_add<="00";res_addx<='0';en_addx<='0';set_addx<='0';sel_dout<='1';
      en_decx<='0';res_addu<='0';en_addu<='0';res_conv<='0';res_c0<='0';res_x<='0';en_cy<='0';sel_prec<='0';
      set_addu<='0';en_uprec<='0'; res_uprec<='0';res_ugen<='0';
   ns<=s5_calc;
when s5_calc=>en_gen<='0';res_gen<='1';buf_oe<='1';sel_in<='1';res_add<='0';en_add<='0';set_add<='0';
      addset_y<=conv_std_logic_vector(0,19);lb<='0';ub<='0';ce<='0';oe<='1';we<='1';ck_adc<='0';busy<='1';
      res_16<='1';en_16<='0';sel_add<="10";res_addx<='0';en_addx<='0';set_addx<='0';sel_dout<='1';
      en_decx<='0';res_addu<='0';en_addu<='0';res_conv<='0';res_c0<='0';res_x<='0';en_cy<='0';sel_prec<='0';
      set_addu<='0';en_uprec<='0'; res_uprec<='0';res_ugen<='0';
   ns<=s6_calc;
when s6_calc=>en_gen<='0';res_gen<='1';buf_oe<='1';sel_in<='1';res_add<='0';en_add<='0';set_add<='0';
      addset_y<=conv_std_logic_vector(0,19);lb<='0';ub<='0';ce<='0';oe<='1';we<='0';ck_adc<='0';busy<='1';
      res_16<='1';en_16<='0';sel_add<="10";res_addx<='0';en_addx<='0';set_addx<='0';sel_dout<='1';
      en_decx<='0';res_addu<='0';en_addu<='0';res_conv<='0';res_c0<='0';res_x<='0';en_cy<='0';sel_prec<='0';
      set_addu<='0';en_uprec<='0'; res_uprec<='0';res_ugen<='0';
   ns<=s7_calc;
when s7_calc=>en_gen<='0';res_gen<='1';buf_oe<='1';sel_in<='1';res_add<='0';en_add<='1';set_add<='0';
      addset_y<=conv_std_logic_vector(0,19);lb<='0';ub<='0';ce<='0';oe<='1';we<='1';ck_adc<='0';busy<='1';
      res_16<='1';en_16<='0';sel_add<="10";res_addx<='0';en_addx<='0';set_addx<='0';sel_dout<='1';
      en_decx<='0';res_addu<='0';en_addu<='1';res_conv<='0';res_c0<='0';res_x<='0';en_cy<='0';sel_prec<='0';
      set_addu<='0';en_uprec<='0'; res_uprec<='0';res_ugen<='0';
```

Figura 4.11 Codice VHDL relativo agli stati per il calcolo della convoluzione senza campioni sovrapposti

- s1_calc: la memoria è ancora comandata per essere letta e tutti i bit di selezione sono già settati in modo corretto dagli stati precedenti a parte “sel_dout” che va settato a uno. Viene caricato l'indirizzo in cui sono memorizzati i dati dell'ADC, quindi tenendo conto dello sfasamento introdotto da quest'ultimo, esso sarà 6. Come detto in precedenza, il generatore di indirizzi va settato all'indirizzo precedente e abilitato per avere il corretto indirizzo per tanto “addset_y”=5, ”en_add”=1

Si noti come “res_addx” non viene abilitato in modo da avere il corretto indirizzo per il successivo caricamento dei bit della sequenza pseudo-random. Questo vale per tutti gli stati di elaborazione dei dati. A questo punto si controlla se l'indirizzo in cui verrà salvato il risultato è il primo, in modo da capire se siamo nel caso di calcolo della convoluzione con campioni sovrapposti o senza. Se siamo in quest'ultimo caso, lo stato successivo è s2_cal.

- s2_calc: rimane tutto invariato perché si aspetta che la memoria renda disponibile il dato.
- s3_calc: viene caricato il dato nella struttura per la convoluzione (en_y=1).
- s4_calc: si aspetta il risultato della convoluzione.
- s5_calc: viene comandata la memoria e il buffer, per poter scrivere il dato (buf_oe=1). Si seleziona il generatore di indirizzi per il risultato (sel_add=10).
- s6_calc: si scrive il dato (we=0).
- s7_calc: si disabilita il write enable (we=1).

Anche in questo caso si fa notare la comodità di avere due generatore di indirizzi uno per il caricamento del dato e uno per il salvataggio del risultato: in questo modo non si deve settare l'indirizzo ma basta scegliere quale usare per generare l'indirizzo tramite “sel_add”. Nello stato s7_calc si controlla l'indirizzo dei dati dell'ADC per capire se sono stati caricati tutti. In caso affermativo si passa allo stato s8_calc, in caso contrario si ripetono gli stati da s2_calc. Negli stati successivi (s8_calc - s10_calc) verranno aggiunti sedici zeri in coda alla struttura per calcolare gli ultimi campioni della convoluzione.

- s8_calc: viene disabilitato il reset di cont16 perché vanno contati sedici zeri e viene abilitato “en_y” per far scorrere i dati dell'ADC nella struttura della convoluzione in modo da poter aggiungere uno zero.
- s9_calc: si resetta il registro zero della convoluzione (res_c0=1) in modo da inserire gli zeri in coda
- s10_calc: viene salvato il risultato della convoluzione (we=0).

Se il contatore è arrivato a quindici, quindi sono stati contati sedici cicli (ciclo zero compreso), tutti i campioni sono stati memorizzati, altrimenti si ripetono gli stati da s8_calc per aggiungere altri

zeri. Se sono stati caricati tutti gli zeri si caricano sedici nuovi bit della sequenza pseudo-random.

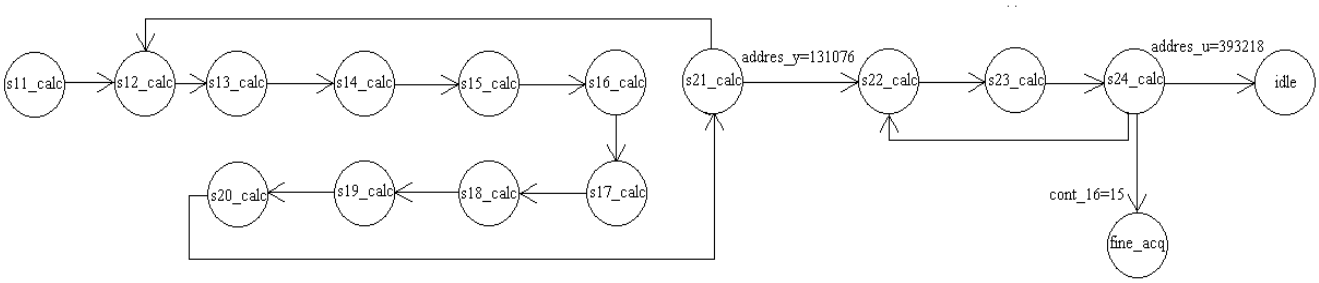


Figura 4.12 Diagramma degli stati relativi al calcolo della convoluzione

```
when s11_calc=>en_gen<='0';res_gen<='1';buf_oe<='0';sel_in<='1';res_add<='0';en_add<='1';set_add<='1';
      addset_y<=conv_std_logic_vector(5,19);lb<='0';ub<='0';ce<='0';oe<='0';we<='1';ck_adc<='0';busy<='1';
      res_16<='1';en_16<='0';sel_add<="00";res_addx<='0';en_addx<='0';set_addx<='0';sel_dout<='1';
      en_decx<='0';res_addu<='0';en_addu<='1';res_conv<='0';res_c0<='0';res_x<='0';en_cy<='0';sel_prec<='0';
      set_addu<='1';en_uprec<='0'; res_uprec<='0';res_ugen<='0';
   ns<=s12_calc;
when s12_calc=>en_gen<='0';res_gen<='1';buf_oe<='0';sel_in<='1';res_add<='0';en_add<='0';set_add<='0';
      addset_y<=conv_std_logic_vector(0,19);lb<='0';ub<='0';ce<='0';oe<='0';we<='1';ck_adc<='0';busy<='1';
      res_16<='1';en_16<='0';sel_add<="00";res_addx<='0';en_addx<='0';set_addx<='0';sel_dout<='1';
      en_decx<='0';res_addu<='0';en_addu<='0';res_conv<='0';res_c0<='0';res_x<='0';en_cy<='0';sel_prec<='0';
      set_addu<='0';en_uprec<='0'; res_uprec<='0';res_ugen<='0';
   ns<=s13_calc;
when s13_calc=>en_gen<='0';res_gen<='1';buf_oe<='0';sel_in<='1';res_add<='0';en_add<='0';set_add<='0';
      addset_y<=conv_std_logic_vector(0,19);lb<='0';ub<='0';ce<='0';oe<='0';we<='1';ck_adc<='0';busy<='1';
      res_16<='1';en_16<='0';sel_add<="00";res_addx<='0';en_addx<='0';set_addx<='0';sel_dout<='1';
      en_decx<='0';res_addu<='0';en_addu<='0';res_conv<='0';res_c0<='0';res_x<='0';en_cy<='1';sel_prec<='0';
      set_addu<='0';en_uprec<='0'; res_uprec<='0';res_ugen<='0';
   ns<=s14_calc;
when s14_calc=>en_gen<='0';res_gen<='1';buf_oe<='0';sel_in<='1';res_add<='0';en_add<='0';set_add<='0';
      addset_y<=conv_std_logic_vector(0,19);lb<='0';ub<='0';ce<='0';oe<='0';we<='1';ck_adc<='0';busy<='1';
      res_16<='1';en_16<='0';sel_add<="00";res_addx<='0';en_addx<='0';set_addx<='0';sel_dout<='1';
      en_decx<='0';res_addu<='0';en_addu<='0';res_conv<='0';res_c0<='0';res_x<='0';en_cy<='0';sel_prec<='0';
      set_addu<='0';en_uprec<='0'; res_uprec<='0';res_ugen<='0';
   ns<=s15_calc;
when s15_calc=>en_gen<='0';res_gen<='1';buf_oe<='0';sel_in<='1';res_add<='0';en_add<='0';set_add<='0';
      addset_y<=conv_std_logic_vector(0,19);lb<='0';ub<='0';ce<='0';oe<='0';we<='1';ck_adc<='0';busy<='1';
      res_16<='1';en_16<='0';sel_add<="10";res_addx<='0';en_addx<='0';set_addx<='0';sel_dout<='1';
      en_decx<='0';res_addu<='0';en_addu<='0';res_conv<='0';res_c0<='0';res_x<='0';en_cy<='0';sel_prec<='1';
      set_addu<='0';en_uprec<='0'; res_uprec<='0';res_ugen<='0';
   ns<=s16_calc;
when s16_calc=>en_gen<='0';res_gen<='1';buf_oe<='0';sel_in<='1';res_add<='0';en_add<='0';set_add<='0';
      addset_y<=conv_std_logic_vector(0,19);lb<='0';ub<='0';ce<='0';oe<='0';we<='1';ck_adc<='0';busy<='1';
      res_16<='1';en_16<='0';sel_add<="10";res_addx<='0';en_addx<='0';set_addx<='0';sel_dout<='1';
      en_decx<='0';res_addu<='0';en_addu<='0';res_conv<='0';res_c0<='0';res_x<='0';en_cy<='0';sel_prec<='1';
      set_addu<='0';en_uprec<='0'; res_uprec<='0';res_ugen<='0';
   ns<=s17_calc;
when s17_calc=>en_gen<='0';res_gen<='1';buf_oe<='0';sel_in<='1';res_add<='0';en_add<='0';set_add<='0';
      addset_y<=conv_std_logic_vector(0,19);lb<='0';ub<='0';ce<='0';oe<='0';we<='1';ck_adc<='0';busy<='1';
      res_16<='1';en_16<='0';sel_add<="10";res_addx<='0';en_addx<='0';set_addx<='0';sel_dout<='1';
      en_decx<='0';res_addu<='0';en_addu<='0';res_conv<='0';res_c0<='0';res_x<='0';en_cy<='0';sel_prec<='1';
      set_addu<='0';en_uprec<='1'; res_uprec<='0';res_ugen<='0';
```

Figura 4.13 Codice VHDL relativo agli stati per il calcolo della convoluzione con campioni sovrapposti

Gli stati per il calco dei campioni, caricando anche quelli precedenti, vengono descritti di seguito:

- s11_calc: vengono settati gli indirizzi in “addres_gen” (set_add=1, addset_y=5) e in

“addres_genu”(set_addu=1), ovviamente sono anche abilitati (en_add=1, en_addu=1).

- s12_calc, s13_cal, s14_calc: del tutto uguali a s2_calc, s3_calc, s4_calc.
- s15_calc: viene selezionato il campione precedente (sel_prec=1) e si seleziona “addgen_u” (sel_add=10).
- s16_calc: si aspetta che la memoria fornisca il dato.
- s17_calc: viene salvato il campione precedente nel registro “r2” (en_uprec=1).
- s18_calc: si aspetta il risultato della convoluzione.
- s19_calc: viene comandata la memoria e il buffer per poter scrivere il dato (buf_oe=1).
- s20_calc: viene scritto il dato (we=0).
- s21_calc: si disabilita il write enable.

Anche in questo caso si controlla l'indirizzo dei dati dell'ADC per capire se sono stati caricati tutti i dati. Gli stati s22_calc, s23_calc, s24_calc sono del tutto uguali agli stati s8_calc, s9_calc, s10_calc. Se si raggiunge l'indirizzo 393˙218 significa che si è calcolata tutta la convoluzione quindi si può tornare allo stato di standby, altrimenti vale lo stesso ragionamento fatto per s10_calc. Tale indirizzo è stato calcolato considerando che i primi 131˙076 indirizzi vengono utilizzati per memorizzare la sequenza random e la sequenza generata dall'ADC, mentre per la convoluzione servono 262˙141 indirizzi. Quindi sommando la lunghezza della convoluzione al primo indirizzo utilizzato per memorizzare il risultato (131˙077) si arriva all'indirizzo finale. La memoria presente sulla development board Altera non è sufficiente, si dovrà utilizzare una memoria più ampia, ma comunque per il progetto è possibile utilizzare le tempistiche relative a quella presente sulla scheda.

CAPITOLO 5

5.1 SINTESI LOGICA

Utilizzando il programma Quartus è stato creato un progetto basato sull' FPGA scelta in precedenza. E' stata fatta la sintesi logica del codice VHDL per verificarne la funzionalità e per capire quante risorse dell' FPGA vengono utilizzate. Queste informazioni sono disponibili nel "compilation report" prodotto da Quartus.

Flow Summary

Flow Status	Successful - Tue Jun 30 11:11:31 2015
Quartus II 32-bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	syst
Top-level Entity Name	syst
Family	Cyclone II
Device	EP2C20F484C7
Timing Models	Final
Total logic elements	743 / 18,752 (4 %)
Total combinational functions	698 / 18,752 (4 %)
Dedicated logic registers	355 / 18,752 (2 %)
Total registers	355
Total pins	58 / 315 (18 %)
Total virtual pins	0
Total memory bits	0 / 239,616 (0 %)
Embedded Multiplier 9-bit elements	0 / 52 (0 %)
Total PLLs	0 / 4 (0 %)

Slow Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	181.75 MHz	181.75 MHz	ck	

Figura 5.1 Estratto del compilation report di Quartus

Osservando la Figura 5.1 si nota che sono utilizzati 734 "logic elements" totali rispetto a 18˙752 disponibili. Questo comporta la possibilità di aggiungere altre strutture per migliorare e incrementare le capacità di calcolo del progetto o la possibilità di utilizzare un' FPGA meno capiente e quindi meno costosa. Si nota anche come la frequenza massima ammissibile di funzionamento del circuito sia 181.75 MHz. Questo vuol dire che possiamo incrementare la velocità di calcolo del sistema, ciò comporterà un miglioramento delle prestazioni del sistema solo se si sostituisce la SRAM scelta con un modello più veloce.

5.2 SIMULAZIONE

Per la simulazione funzionale si è utilizzato il tool Questasim. Per poter simulare il sistema è stato creato un file VHDL non sintetizzabile in cui il nucleo di calcolo è stato inserito come componente

insieme ad un package che simula la SRAM ed a uno che simula l'impedenza. Il package della SRAM contiene tutti i segnali di controllo della SRAM in più sono presenti anche tutti i tempi caratteristici della stessa. Questi sono utilizzati per generare degli errori se durante la simulazione vengono violati tali tempi. Sono presenti anche comandi per inizializzare la SRAM caricando il contenuto di un file di testo nella stessa. Altri comandi (dump) sono utilizzati per scaricare in un file di testo il contenuto della SRAM in modo da controllarne il corretto utilizzo.

L'impedenza scelta per testare il corretto funzionamento è una semplice resistenza. Tale impedenza è stata implementata generando una word costante se il bit della sequenza pseudo-random è uno, mentre ne è stata generata una nulla se il bit era zero. La word è stata anche ritardata di 6 cicli di clock per simulare il ritardo inserito dall'ADC tramite l'utilizzo di sei registri. La Figura 5.3 mostra il codice relativo all'impedenza.

```
entity impedenza is
   port (x,ck_bit,reset : in std_logic;
      y : out std_logic_vector(11 downto 0));
end impedenza;

architecture a of impedenza is

signal y_adc : std_logic_vector(11 downto 0);
signal q : std_logic_vector(83 downto 0);

begin
   r0 : process(x)
   begin
      if x='1' then
         y_adc<=conv_std_logic_vector(5,12);
      else
         y_adc<=conv_std_logic_vector(0,12);
      end if;
   end process;

   r1 : process(reset,ck_bit,y_adc)
   begin
      if reset='1' then
         q(11 downto 0)<=conv_std_logic_vector(0,12);
      elsif ck_bit'event and ck_bit='1' then
         q(11 downto 0)<=y_adc;
      end if;
   end process;

   rk: for k in 1 to 6 generate
      r_k:process(reset,ck_bit,y_adc)
      begin
         if reset='1' then
            q(((12*(k+1))-1) downto (12*k))<=conv_std_logic_vector(0,12);
         elsif ck_bit'event and ck_bit='1' then
            q(((12*(k+1))-1) downto (12*k))<=q(((12*k)-1) downto (12*(k-1)));
         end if;
      end process;
   end generate;

   b0:y<=q(83 downto 72);
end a;
```

Figura 5.2 Codice VHDL per l'implementazione dell'impedenza

Una volta avviata la simulazione si è controllato il funzionamento del circuito nella varie fasi.

Per controllare la fase di acquisizione si è utilizzato il comando “dump” in modo da controllare lo

stato della memoria al termine di essa. Confrontando la sequenza pseudo-random generata con quella memorizzata e verificatane l'uguaglianza con l'ausilio di Matlab si è stabilito il corretto funzionamento di questa fase.

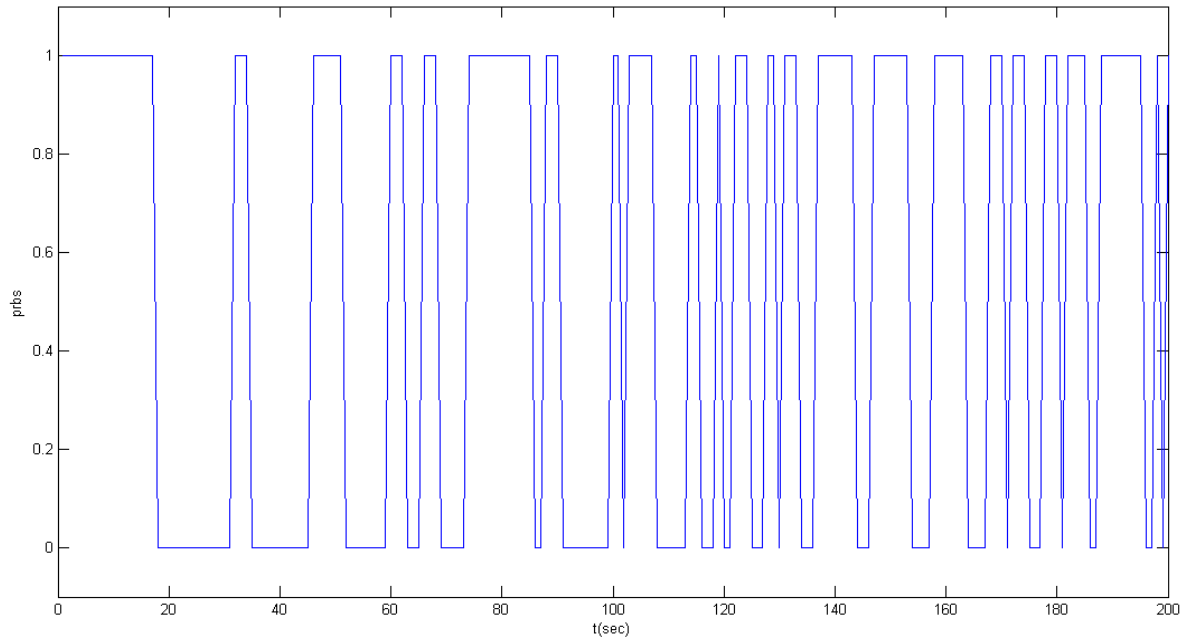


Figura 5.3 Confronto tra la sequenza pseudo-random generata e quella memorizzata

La Figura 5.3 mostra il confronto tra la sequenza pseudo-random memorizzata (traccia blu) e quella generata (traccia rossa), essendo sovrapposte la traccia blu copre quella rossa, questo dimostra l'uguaglianza tra le sequenze.

Per quanto riguarda la fase del caricamento dei bit pseudo-random nella struttura per la convoluzione si è confrontato il segnale relativo al package della convoluzione (xm) contenente il vettore dei bit caricati, con i bit della sequenza generata. Questo controllo è stato fatto per 4-5 cicli di funzionamento. Anche in questo caso il funzionamento si è rivelato corretto. Per la fase di elaborazione dei dati si è confrontata la sequenza memorizzata, con la sequenza ricavata calcolando la convoluzione tramite Matlab. L'impedenza utilizzata per la simulazione è una resistenza con valore costante di 5 Ω.

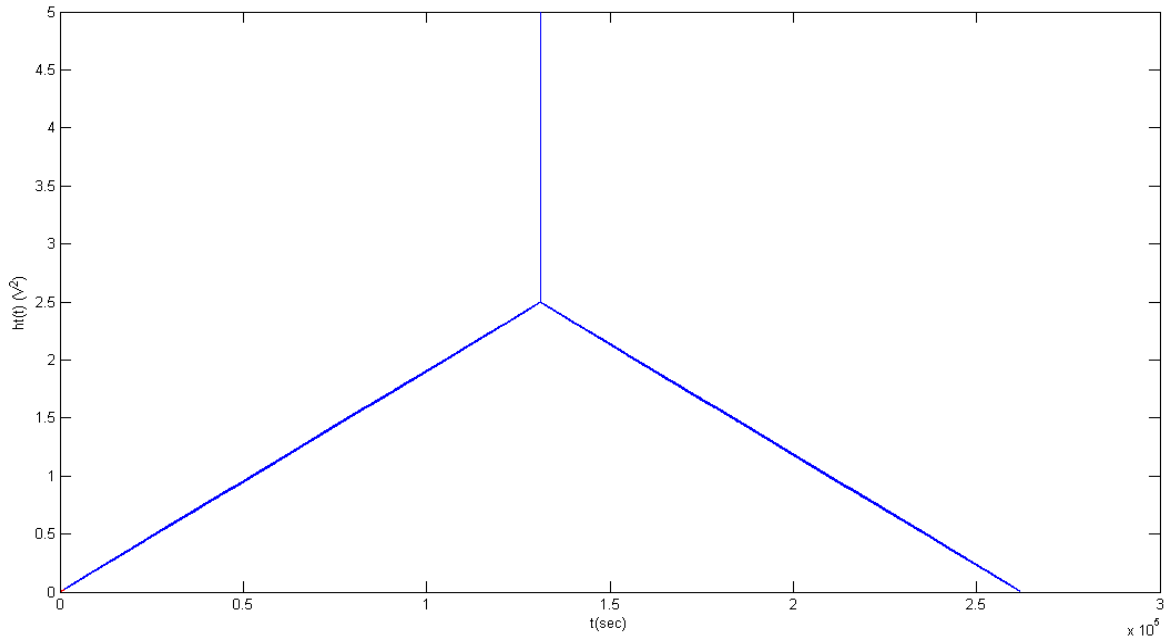


Figura 5.4 Confronto tra la risposta impulsiva teorica e quella pratica

La Figura 5.4 mostra il confronto tra la risposta impulsiva teorica (traccia blu) e quella calcolata durante la simulazione (traccia rossa). La sequenza ricavata dalla simulazione non è però completa, perché la simulazione per un tempo maggiore al secondo risulta molto onerosa. Il numero di dati ricavato dalla simulazione è molto ridotto infatti la traccia rossa è quasi assente dal grafico. Si ritiene comunque corretto il funzionamento dato che il sistema ripete ciclicamente determinati stati fino al completamento del calcolo e se questi stati portano ad un risultato corretto per i primi cicli non ci sono motivi per pensare che i risultati siano errati nei cicli successivi.

Il tempo totale di elaborazione dei dati è di circa 3 minuti. Questo valore è stato ottenuto moltiplicando il tempo necessario per un ciclo (circa 26 ms) per il numero di cicli (circa 8192). Il numero di cicli si può ottenere dividendo la lunghezza della sequenza per la lunghezza delle partizioni. L'acquisizione invece ha una durata di 13 ms.

CONCLUSIONI

Questo sistema ha raggiunto l'obbiettivo prefissato cioè calcolare la risposta impulsiva dell'impedenza al segnale pseudo-random. Per ricavare il valore dell'impedenza si deve effettuare un'ulteriore elaborazione dei dati. Il primo passo è quello di calcolare la trasformata di Fourier della risposta impulsiva ottenuta. Dopo averne calcolata la trasformata si può ricavare la parte reale e la parte immaginaria dell'impedenza dalla fase e dall'ampiezza.

Per verificare il corretto funzionamento del sistema nella realtà, oltre che caricare il sistema implementato sull' FPGA, si dovrebbe implementare un modulo per la comunicazione tra l'FPGA e il pc in modo da poter importare i risultati contenuti nella SRAM. Successivamente andranno svolti tramite pc tutti i calcoli per ricavare il valore dell'impedenza.

Per quanto riguarda la velocità del sistema, si è raggiunto un buon tempo di misura e un tempo di elaborazione dei dati ancora accettabile. Se si volesse diminuire il tempo di calcolo come prima soluzione si consiglia di aumentare il clock di sistema, utilizzando però, una memoria più rapida.

BIBLIOGRAFIA/SITOGRAFIA

[1] FPGA: Field Programmable Gate Array http://it.wikipedia.org/wiki/Field_Programmable_Gate_Array

[2] L. Calandrino, M. Chiani, Lezioni di Comunicazioni Elettriche, Pitagora Editrice, Bologna.

[3] Altera cyclone II development board https://www.altera.com/en_US/pdfs/literature/manual/mnl_cii_starter_board_rm.pdf

[4] SRAM: datasheet IS61C25616AL http://www.mouser.com/ds/2/198/61-64C25616AL-AS-258641.pdf

[5] ADC: datasheet ADS804 http://www.ti.com/lit/ds/symlink/ads804.pdf

[6] LFSR: Linear feedback shift register https://en.wikipedia.org/wiki/Linear_feedback_shift_register

[7] LFSR: Hideo Okawara's Mixed signal lecture Series https://www.advantest.com/cs/groups/public/documents/document/zhzw/mda3/~edisp/advp007355.pdf

[8] FIR, Overlap and add: Tommaso Cecchini, “Studio e progettazione vlsi di un acceleratore hardware per la ricostruzione di ambienti sonori”, Tesi Laurea Specialistica, Università di Pisa, Facoltà di Ingegneria, corso di laurea specialistica in Ingegneria Elettronica, indirizzo Sistemi Elettronici, anno accademico 2005-2006.

[9] G. Luciani, "Strumento per la diagnosi di carcinoma BCC basato su spettroscopia impedenziometrica", tesi di laurea magistrale , Scuola di Architettura e Ingegneria, Università di Bologna, Cesena, 2015.

RINGRAZIAMENTI

Ringrazio il professore Aldo Romani per avermi dato la possibilità di svolgere questa tesi e per averlo avuto come relatore. Inoltre, insieme al professor Romani, ringrazio il correlatore Crescentini: la loro grande disponibilità e i loro preziosi consigli mi hanno permesso di risolvere i dubbi e i problemi durante l'intero progetto.

Ringrazio la mia famiglia per avermi sempre sostenuto e per aver creduto in me anche nei momenti difficili. In particolar modo ringrazio i miei genitori per avermi dato la possibilità di affrontare questo percorso nonostante tutti i sacrifici fatti per darmi questa opportunità.

Ringrazio la mia ragazza Eleonora per essermi stata sempre vicino, per non aver mai dubitato delle mie capacità e per essere stata il mio appiglio nei momenti più difficili, senza di lei non sarei riuscito ad affrontare le difficoltà di questo percorso.

Ringrazio la famiglia della mia ragazza per avermi sempre sostenuto, in particolar modo ringrazio Angela per avermi trattato come un figlio e per avermi fatto sentire parte della famiglia.

Ringrazio i miei amici universitari per aver condiviso con me gioie e difficoltà dell'università; ringrazio soprattutto Francesca per aver sempre creduto in me, a volte anche più di me stesso, in lei ho trovato una grande amica, e Matteo inseparabile amico con il quale ho condiviso quasi dieci anni di studio e altrettanti anni di divertimenti fuori dall'università.

Ringrazio il mio amico Gio per avermi fatto staccare dagli studi con le nostre “uscite” e per essermi stato sempre vicino anche se a volte l'ho trascurato per dedicarmi all'università.