UNIVERSITA' DI PISA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA ELETTRONICA
Nuovo Ordinamento
Anno Accademico 2013 – 2014

Tesi di Laurea Specialistica

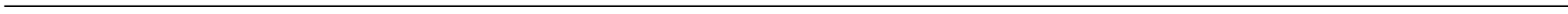
PROGETTAZIONE DI ALGORITMI PER L'ELABORAZIONE DI MAPPE GEORADAR SU PIATTAFORMA FPGA

Relatori:

Prof. Luca FANUCCI

Dott. Marco CONSANI

Candidato:
Marco PAMPANA



A Matilde

Ringraziamenti

Desidero ringraziare i relatori, prof. Luca Fanucci e il dott. Marco Consani amministatore di Tertium Technology.

Ringrazio l'ing. Guido Manacorda e il sig. Mario Miniati di IDS - Ingegneria Dei Sistemi per la collaborazione e l'indispensabile supporto.

Ringrazio Axel, Elisa, Tiziana, Alessandro, Carmine, Michele, Silvia, Andrea e Giorgio per l'aiuto che mi hanno dato e per l'ambiente nel quale ho lavorato.

Prefazione

Questo elaborato è il frutto di una collaborazione con Tertium Technology s.r.l. iniziata già in occasione della tesi per la laurea triennale. Questa azienda ha sede a Pisa e, in quella occasione, mi fece conoscere i sistemi georadar prodotti da IDS - Ingegneria Dei Sistemi, un'altra azienda sempre del territorio. Tertium Technology si occupava, e si occupa tuttora, della produzione e dello sviluppo delle unità di acquisizione dati per questi sistemi georadar (denominate DAD e descritte nel capitolo 2 di questo elaborato).

Pochi giorni dopo l'inizio della mia tesi triennale è iniziata anche la mia esperienza lavorativa presso Tertium Technology come operatore dei collaudi e del controllo qualità delle DAD, situazione che con il tempo è evoluta in termini di coinvolgimento e responsabilità occupandomi attualmente dell'intero processo produttivo.

Per il progetto relativo a questa tesi è stata ovviamente coinvolta anche IDS - Ingegneria Dei Sistemi; per la definizione degli obiettivi ho partecipato a vari incontri con l'ing. Guido Manacorda di IDS Ingegneria Dei Sistemi ed il dott. Marco Consani di Tertium Technology, durante i quali sono emerse le possibili evoluzioni delle unità di acquisizione DAD attualmente in produzione.

Essendo l'architettura della DAD basata su un dispositivo FPGA, è stata valutata l'opportunità di sfruttare tale dispositivo per implementare alcuni tra i principali algorit-

mi di elaborazione dati georadar, dotando quindi la DAD di capacità di elaborazione della quale attualmente è priva. In particolare sono stati considerati gli *algoritmi di visualizzazione* (descritti nel paragrafo 3.3) e gli *algoritmi di migrazione* (descritti nel paragrafo 3.4), risultati i più interessanti rispetto allo sviluppo che interesserà in futuro i sistemi georadar IDS.

In seguito il dott. Axel Penzo di Tertium Technology mi ha illustrato il funzionamento del software *K2 Detector*, da lui sviluppato ed attualmente utilizzato nei sistemi IDS per l'elaborazione dati georadar: l'obiettivo infatti è stato individuato nel dimostrare che grazie alla presenza del dispositivo FPGA nella DAD sarebbe stato possibile svolgere le principali elaborazioni sui dati georadar direttamente nell'unità di acquisizione DAD e non più nell'unità di elaborazione (composta da un PC sul quale è installato il software *K2 Detector*).

L'obiettivo è stato raggiunto quando, una volta progettate le reti per l'esecuzione degli algoritmi sopracitati e descritte in VHDL, è stato dimostrato che con le simulazioni si ottengono gli stessi risultati che con il software *K2 Detector* (come esposto nelle conclusioni di questo elaborato), partendo dagli stessi dati georadar in ingresso.

Riassumendo, l'obiettivo della presente tesi è l'implementazione all'interno di un dispositivo FPGA di alcuni algoritmi per l'elaborazione dei segnali georadar, descritti nel capitolo 3 di questo elaborato. I vantaggi ottenibili sarebbero:

- Migliore utilizzo delle risorse del sistema Georadar IDS con conseguenti riduzioni del carico di lavoro dell'unità di elaborazione e dei tempi di elaborazione.

- Possibilità di realizzazione di dispositivi georadar privi dell'unità di elaborazione ma dotati di interfacce uomo/macchina come ad esempio touchscreen, avvisatori

acustici o luminosi, riducendo dimensioni e costi.

- L'elaborazione dati all'interno della DAD consente anche la possibilità di individuare e trasmettere all'unità di elaborazione solamente le informazioni utili, con conseguente riduzione della banda necessaria alla trasmissione. Ciò consentirebbe la trasmissione dati anche in situazioni critiche nelle quali il canale di comunicazione è intrinsecamente limitato dato il contesto applicativo, come ad esempio la testa di scavo in una macchina per le trivellazioni.

Il presente elaborato è così strutturato:

Nel **capitolo 1** è descritto il sistema georadar nel suo complesso prendendo come riferimento un sistema IDS - Ingegneria Dei Sistemi.

Nel **capitolo 2** è descritta l'unità di acquisizione dati (DAD) che attualmente viene utilizzata su questi dispositivi.

Nel **capitolo 3** sono descritti gli algoritmi che vengono impiegati per l'elaborazione dati georadar, attualmente effettuata dall'unità di elaborazione esterna.

Nel **capitolo 4** sono descritte le strutture progettate affinché tali algoritmi possano essere eseguiti dal dispositivo FPGA presente all'interno della DAD.

Nell'**Appendice A** è riportata la descrizione in linguaggio VHDL delle strutture progettate.

Indice

1 Introduzione ai sistemi georadar **8**

1.1 Descrizione del georadar 8

2 L'unità di controllo DAD **14**

2.1 Funzionamento del georadar e dell'unità di acquisizione 14

2.2 Task di acquisizione e trasmissione 18

3 Gli algoritmi di elaborazione **20**

3.1 Scopo dell'elaborazione 20

3.2 Algoritmi di pre-processing: filtraggi 21

3.3 Algoritmi di processing: visualizzazione 23

3.4 Algoritmi di post-processing: focalizzazione 28

4 Le strutture progettate **34**

4.1 Introduzione 34

4.2 Arhitettura per gli algoritmi di visualizzazione 35

4.3 Arhitettura per gli algoritmi di migrazione 48

5 Conclusioni **57**

A VHDL **63**

A.1 Procedura per le simulazioni 63

A.2 Codice VHDL per gli algoritmi di visualizzazione 64

A.2.1 VHDL Blocco Engine 64

A.2.2 VHDL Blocco Processing 82

A.2.3 VHDL Testbench 91

A.3 Codice VHDL per gli algoritmi di migrazione 94

A.3.1 VHDL Blocco Engine 94

A.3.2 VHDL Blocco Processing 103

A.3.3 VHDL Testbench 104

Capitolo 1

Introduzione ai sistemi georadar

1.1 Descrizione del georadar

I sistemi Ground Penetrating Radar, anche noti come Georadar, Ground Probing Radar, Subsurface Radar o GPR, sfruttano i principi della propagazione elettromagnetica per l'esplorazione di sottosuolo o di strutture edili ottenendo una mappatura riportante tubi, cavi elettrici, griglie metalliche, vuoti o altre disomogeneità eventualmente presenti sotto la superficie in esame. I Georadar trovano quindi applicazioni in molti campi, dalle indagini di massima per la misura della profondità di ghiacciai o fondi stradali e ferroviari, fino alle indagini di precisione nei muri o nelle strutture in cemento armato per la rilevazione di armature o utenze[1].

L'indagine georadar consiste nel provocare degli impulsi costituiti da onde elettromagnetiche a frequenza predeterminata diretti verso la superficie in esame e nell'acquisire l'eco causata dalle discontinuità rilevate sotto la superficie. La frequenza dell'onda emessa varia in base all'indagine da eseguire, tipicamente da qualche decina di MHz per le

indagini di massima e di profondità del substrato del terreno fino a qualche GHz per la ricerca di sottoservizi nelle strutture edili.

I componenti fondamentali del georadar IDS - Ingegneria dei Sistemi sono una o più antenne, un'unità di acquisizione (denominata DAD) e un'unità di elaborazione composta da un PC sul quale è installato il software *K2 Detector*, appositamente sviluppato per l'elaborazione dati georadar. In funzione dell'applicazione per la quale viene utilizzato, il sistema georadar può comprendere anche una batteria, un carrello e una ruota metrica per la georeferenziazione dei dati acquisiti; il sistema georadar riportato a titolo esemplificativo in figura 1.1 è un *Detector*[2], prodotto da IDS Ingegneria Dei Sistemi S.p.A.: è un sistema georadar con antenna singola in grado di rilevare servizi posti sotto la superficie stradale e viene utilizzato prima delle operazioni di scavo in modo da rilevare l'esatta posizione e profondità di eventuali tubi, cavi o rocce. L'operatore, dopo aver impostato i vari parametri di configurazione mediante il software installato sull'unità di elaborazione, avvia il processo di indagine spingendo il georadar lungo la direzione prestabilita sul terreno in esame.

Lo schema a blocchi dell'antenna è riportato in figura 1.2: è possibile individuare i due dipoli per la trasmissione dell'impulso radar e per la ricezione dell'eco generata (figura 1.2a e 1.2b), l'interfaccia di collegamento tramite cavo con la DAD (figura 1.2g e 1.2h) e la ruota metrica (figura 1.2f).

Il segnale captato dall'antenna di ricezione, definito segnale *A-Scan*, ha un andamento tipico come quello in figura 1.3: la prima parte, denominata *Main bang*, è dovuta ai fenomeni di accoppiamento tra le antenne e all'energia riflessa dalla stessa superficie (figura 1.2d), la seconda parte, considerevolmente meno ampia della prima, è il segnale

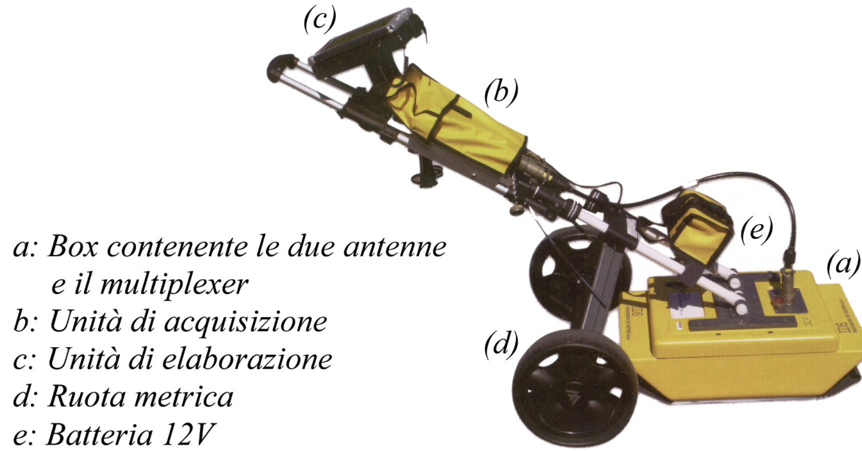


Figura 1.1: Georadar IDS Detector e Detector Duo

utile relativo all'eco provocata dall'onda trasmessa nel mezzo in esame (figura 1.2e).

Durante l'acquisizione, tramite il software *K2 Detector* viene visualizzata sul monitor la mappa radar in tempo reale, come quella in figura 1.4, affinché l'operatore possa individuare i target presenti sotto la superficie: questa mappa riporta sull'asse orizzontale la distanza coperta sul percorso di indagine e sull'asse verticale la rappresentazione dei campioni del segnale *A-Scan* con toni di grigio. I tracciati individuabili in figura 1.4 sono tipicamente dovuti alla presenza di tubi o cavi posti trasversalmente alla direzione di indagine; il principio di formazione dei tracciati è descritto in figura 1.5: un target posto a profondità P_0 e ad una distanza x_0 dall'antenna provoca un'eco che, rispetto all'istante in cui viene rilevato il *Main bang*, raggiunge l'antenna ricevente dopo un tempo t_{x0} pari a

$$t_{x0} = \frac{2\sqrt{P_0^2 + x_0^2}}{\nu_f} \tag{1.1}$$

con ν_f velocità di propagazione dell'onda nel mezzo in esame; quando $x_0 = 0$ (caso C di figura 1.5), il tracciato raggiunge il vertice ed è possibile ricavare la profondità alla

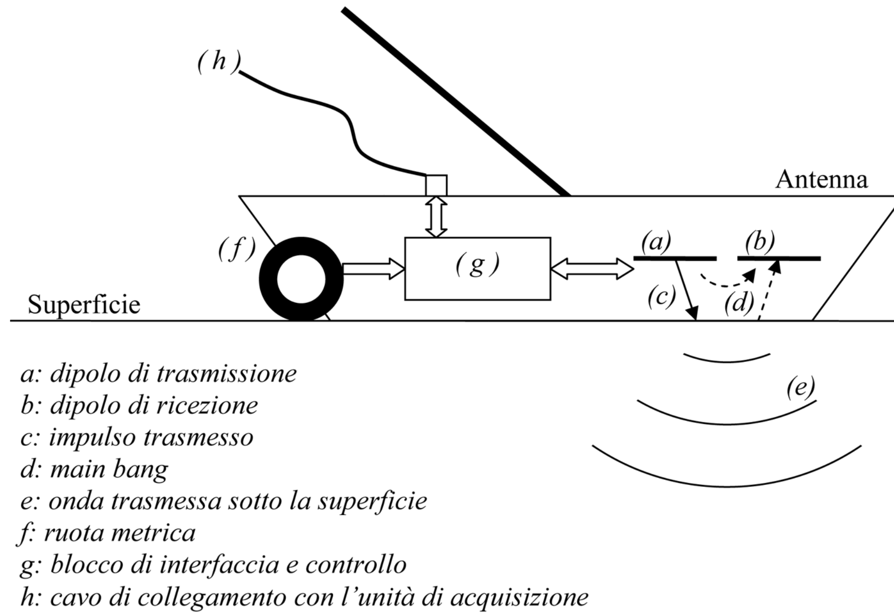


Figura 1.2: Descrizione dell'antenna

quale si trova il target in quanto

$$t_0 = t_{x0}|_{x0=0} = \frac{2P_0}{\nu_f} \Rightarrow P_0 = \frac{t_{tw}\nu_f}{2} \tag{1.2}$$

dove t_{tw} indica il *Two Way Time*, ovvero il tempo che intercorre tra la generazione dell'impulso radar e la ricezione dell'eco relativo. Il valore di ν_f può essere stimato da valori tabulati se è nota la composizione del terreno o ricavato con metodi appositi uno dei quali è descritto nel paragrafo 3.4.

Durante l'indagine la funzione della DAD consiste nel pilotaggio delle antenne, nella conversione A/D del segnale captato e nella trasmissione dei dati acquisiti via ethernet all'unità di elaborazione; le funzioni di questa unità sono la memorizzazione dei dati acquisiti e l'applicazione di specifici algoritmi per la rappresentazione grafica dei dati (ovvero per la formazione del radargramma).

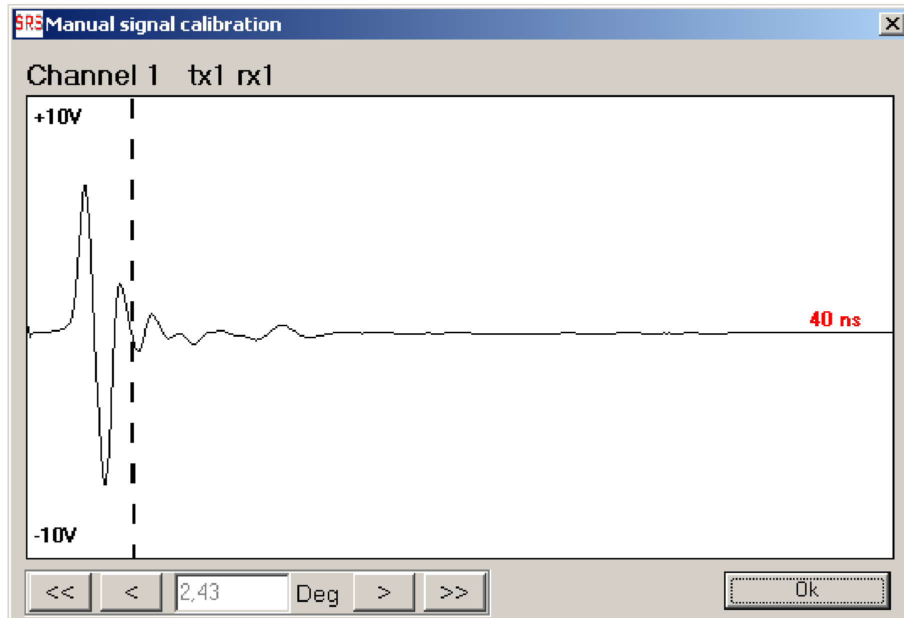


Figura 1.3: Segnale captato dall'antenna di ricezione (segnale A-Scan)

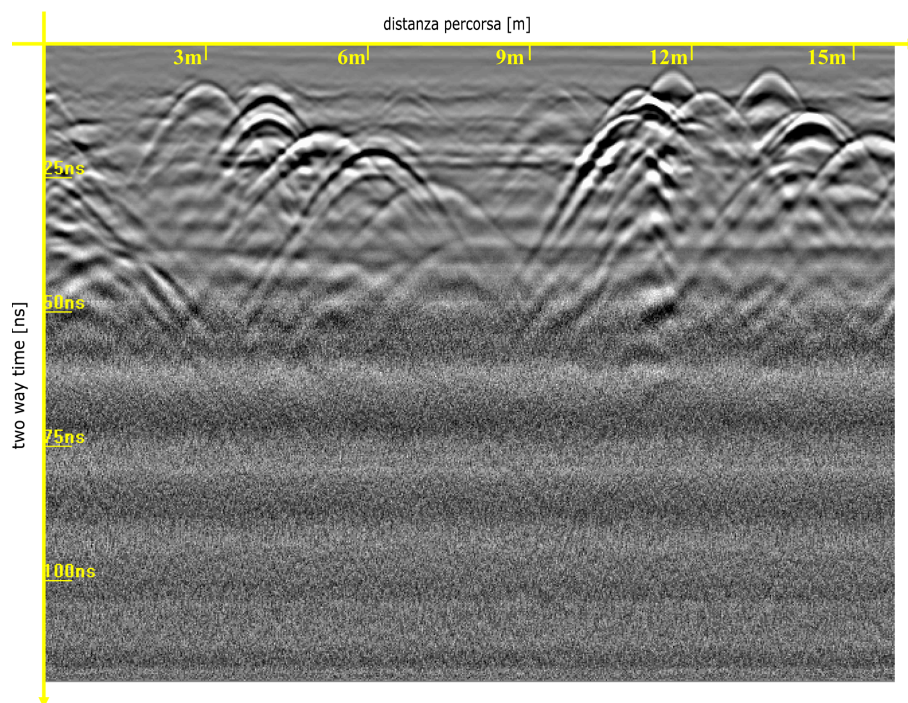


Figura 1.4: Esempio di radargramma

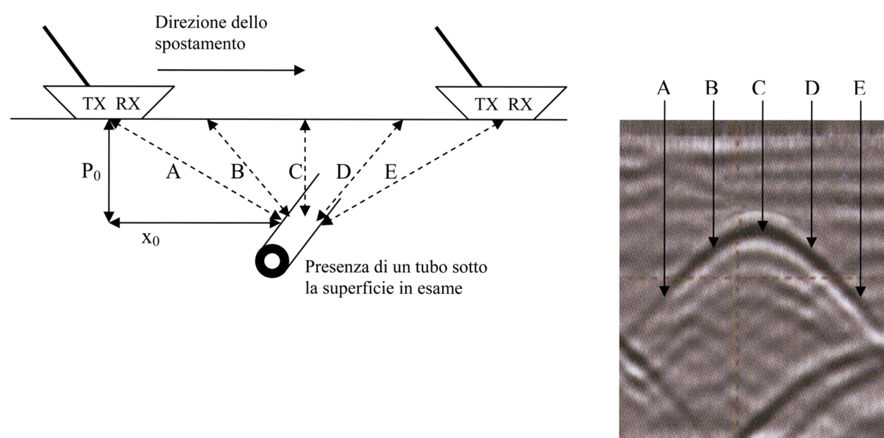


Figura 1.5: Principio di formazione dei tracciati

Capitolo 2

L'unità di controllo DAD

2.1 Funzionamento del georadar e dell'unità di acquisizione

L'unità di acquisizione oggetto di questo lavoro è la DAD Fastwave progettata e realizzata da Tertium Technology S.r.l. per IDS S.p.A.; gli schemi a blocchi del sistema georadar e dell'unità di acquisizione sono riportati in figura 2.1. L'architettura della DAD è basata su un dispositivo FPGA Lattice LFXP2-17E-6QN208C e sul microcontrollore Atmel ATMEGA128-16AU; la FPGA gestisce il flusso dati e il pilotaggio delle antenne mentre il microcontrollore viene utilizzato per la fase di configurazione iniziale dei parametri di funzionamento.

L'acquisizione del segnale *A-Scan* avviene mediante la tecnica del campionamento in tempo equivalente[3]: ciò è possibile in quanto la condizione necessaria di segnale ripetitivo è normalmente rispettata se l'antenna è quasi ferma in quanto, essendo il sottosuolo

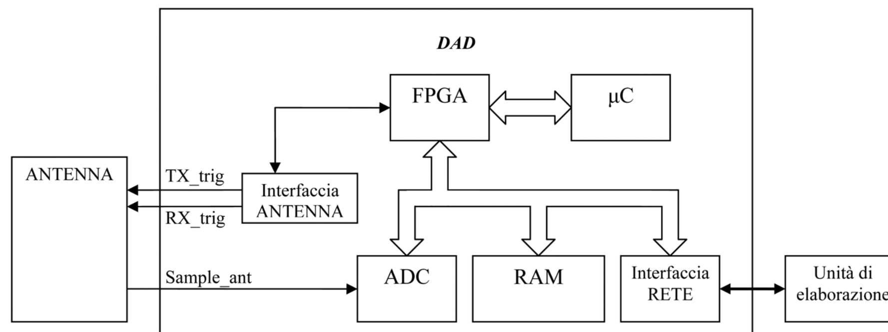


Figura 2.1: Schema a blocchi di una DAD FastWave IDS

immutabile, lo è anche l'eco generata. Come mostrato in figura 2.2, vengono provocati impulsi radar con un periodo T_P mentre il campionamento avviene con periodo $T_C = T_P + t_{C_eq}$, con t_{C_eq} tempo di campionamento equivalente.

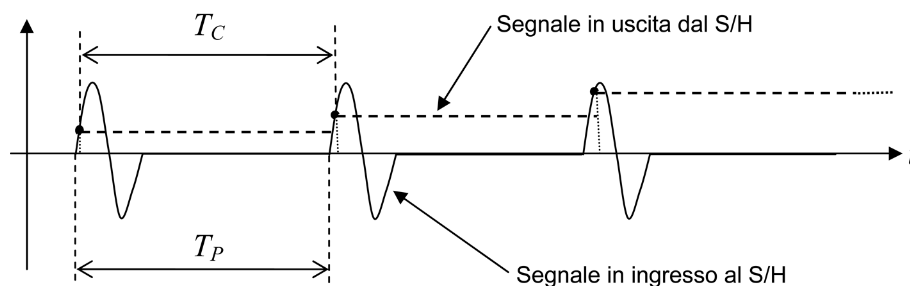


Figura 2.2: Campionamento del segnale A-Scan in tempo equivalente

L'antenna è dotata di un ingresso $TX_{trigger-IN}$, un ingresso $RX_{trigger-IN}$ e un'uscita *Sample_ant*. Un segnale di trigger all'ingresso $TX_{trigger-IN}$ provoca la generazione di un impulso radar e un segnale di trigger sull'ingresso $RX_{trigger-IN}$ comporta il campionamento dell'eco; il campione così ottenuto è mantenuto sull'uscita *Sample_ant* mediante un sample and hold; i segnali $TX_{trigger-IN}$ e $RX_{trigger-IN}$ sono riportati in figura 2.3: ogni fronte in salita del segnale TXTRIG corrisponde ad un impulso radar, mentre un fronte in salita del segnale RXTRIG corrisponde ad un istante di campionamento.

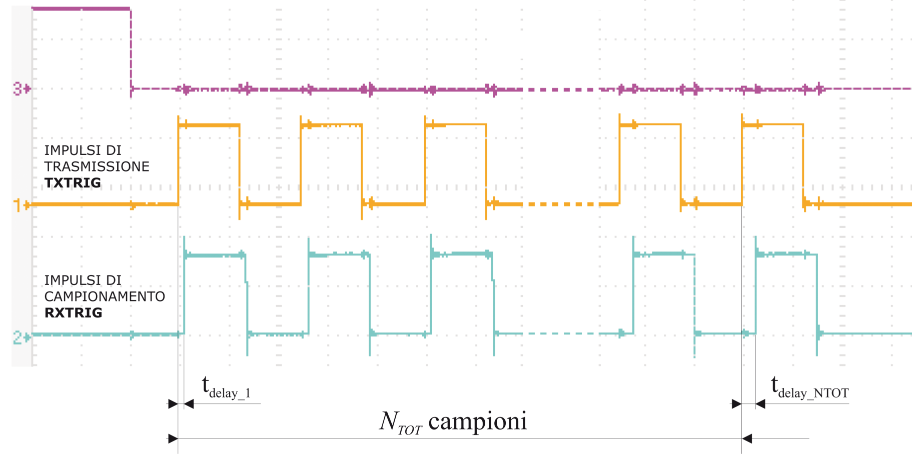


Figura 2.3: Generazione dei trigger per il campionamento del segnale A-Scan

Il ritardo tra l'ultimo trigger di trasmissione e l'ultimo trigger di campionamento, indicato in figura 2.3 con t_{delay_NTOT}, determina la durata t_{fs} del segnale *A-Scan* che viene acquisito: infatti $t_{fs} = t_{delay_NTOT} - t_{delay_1}$, con t_{delay_1} costante preimpostato a 150ns. Se definiamo N_{TOT} come il numero di trigger generati sia di trasmissione che di campionamento è possibile ricavare la frequenza di campionamento equivalente f_{C_eq} data da $f_{C_eq} = \frac{N_{TOT}}{t_{fs}}$; inoltre, definendo $f_{TRIGGER_RX} = \frac{1}{T_C}$ come la frequenza dei trigger di campionamento e $PRF = \frac{1}{T_P}$ come la frequenza dei trigger di trasmissione, si ottiene che:

$$\frac{1}{PRF} = \frac{1}{f_{TRIGGER_RX}} - \frac{t_{fs}}{N_{TOT}} \tag{2.1}$$

La DAD riceve i campioni inviati dall'antenna in ingresso al convertitore A/D; la frequenza di conversione A/D, definita f_{conv}, può essere impostata come un sottomultiplo della frequenza dei trigger di campionamento $f_{TRIGGER_RX}$, tipicamente di un fattore A impostabile da 1 a 4: in questo modo viene convertito solo un campione ogni A. L'insieme dei campioni convertiti relativi al segnale *A-Scan* è definito *Sweep* ed è composto da un

numero N di campioni dato da $N = \frac{N_{TOT}}{A}$.

Una tipica configurazione di lavoro è data dai seguenti parametri (configurazione del georadar *IDS Detector DUO*):

- $N = 512$;
- $t_{fs} = 128ns$;
- $f_{TRIGGER_RX} = 400KHz$;
- $A = 3$;

E' quindi possibile ricavare:

- $f_{conv} = \frac{f_{TRIGGER_RX}}{A} = 133333.333Hz$;
- $N_{TOT} = AN = 1536$;
- Dalla 2.1 si ricava $PRF = 400013Hz$;
- $t_{delay_NTOT} = 150ns + 128ns = 278ns$;
- Tempo di campionamento equivalente $t_{C_eq} = \frac{t_{fs}}{N} = 250ps$;

La frequenza di campionamento equivalente è pari a $f_{C_eq} = \frac{N_{TOT}}{t_{fs}} = 12\ GSample/s$, ma convertendo un campione ogni $A = 3$ si ottiene una frequenza di campionamento equivalente effettiva $f_{C_eq_ef}$ data da $f_{C_eq_ef} = \frac{f_{C_eq}}{A} = 4\ GSample/s$. La frequenza $f_{C_eq_ef}$ deve essere tale da consentire un corretto campionamento del segnale *A-Scan*: l'antenna utilizzata in questa configurazione è accordata ad una frequenza $f_{IR} = 700MHz$ e ha una banda di emissione $B \approx f_{IR}$ centrata su f_{IR}, quindi $f_{IR_MAX} \approx 1GHz$ e $f_{C_eq_ef}$ è

sufficiente per un corretto campionamento. Infine, dato che per ogni sweep viene generato un treno di N_{TOT} impulsi a frequenza PRF, si ottiene che la durata t_{sweep} del processo di acquisizione di un singolo sweep è pari a

$$t_{sweep} = \frac{N_{TOT}}{PRF} = 3.84ms \tag{2.2}$$

In seguito alla conversione lo sweep, in questo caso composto da 512 campioni a 16 bit, viene memorizzato nella RAM interna e successivamente trasmesso all'unità di elaborazione.

2.2 Task di acquisizione e trasmissione

Il processo consistente nella conversione A/D e nella memorizzazione del segnale acquisito è definito *Task di acquisizione.* Successivamente al task di acquisizione viene eseguito il *Task di trasmissione* che consiste nel trasferire lo sweep memorizzato all'interfaccia di rete per la trasmissione all'unità di elaborazione, come descritto in figura 2.4. Attualmente, quindi, non è prevista alcuna elaborazione sui dati acquisiti ma solo la trasmissione all'unità di elaborazione che provvede all'esecuzione degli algoritmi necessari alla rappresentazione grafica.

Questa architettura consente l'introduzione di uno o più task di elaborazione, oltre ai due già presenti, per l'esecuzione di algoritmi quali quelli descritti nel capitolo 3 di questo elaborato.

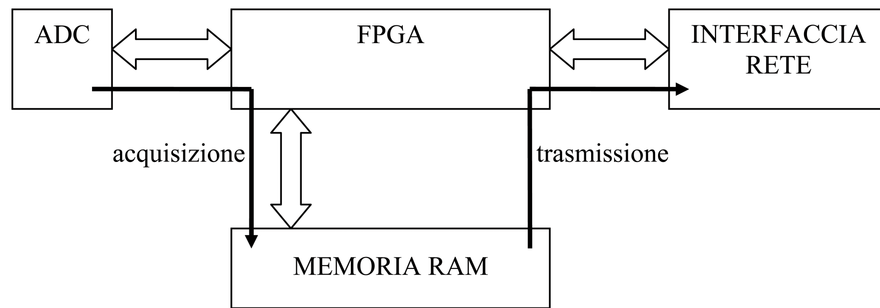


Figura 2.4: Task di acquisizione e Task di trasmissione

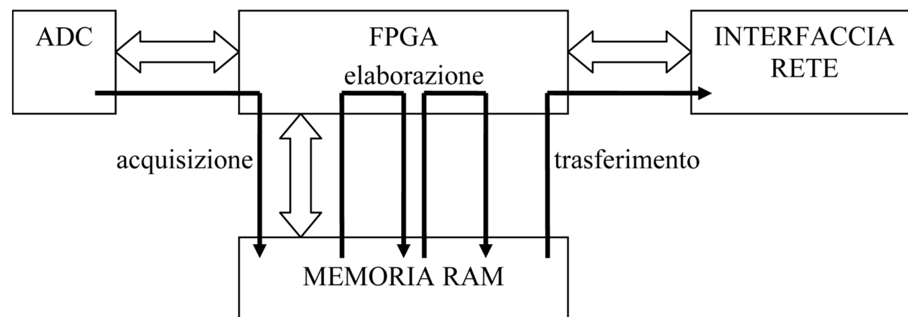


Figura 2.5: Introduzione di uno o più task di elaborazione

Capitolo 3

Gli algoritmi di elaborazione

3.1 Scopo dell'elaborazione

Una rappresentazione grafica dei dati senza alcuna elaborazione dà una mappa come quella riportata in figura 3.1, nella quale i toni più bianchi o più neri corrispondono rispettivamente ai massimi o ai minimi del segnale captato. E' possibile notare come il *Main bang*, essendo un segnale molto ampio e quindi rappresentato con toni dal bianco al nero, risalti nella parte alta della mappa; la parte significativa è invece sottostante alla rappresentazione del *Main bang*, in quanto quest'ultimo coincide con la superficie del terreno in esame, e riporta tracciati non ben distinguibili in quanto gli echi dei target sono molto più piccoli del *Main bang* e non generano apprezzabili variazioni nei toni di grigio.

Per ottenere una mappa come quella in figura 1.4 è necessaria l'applicazione degli algoritmi descritti in questo capitolo.

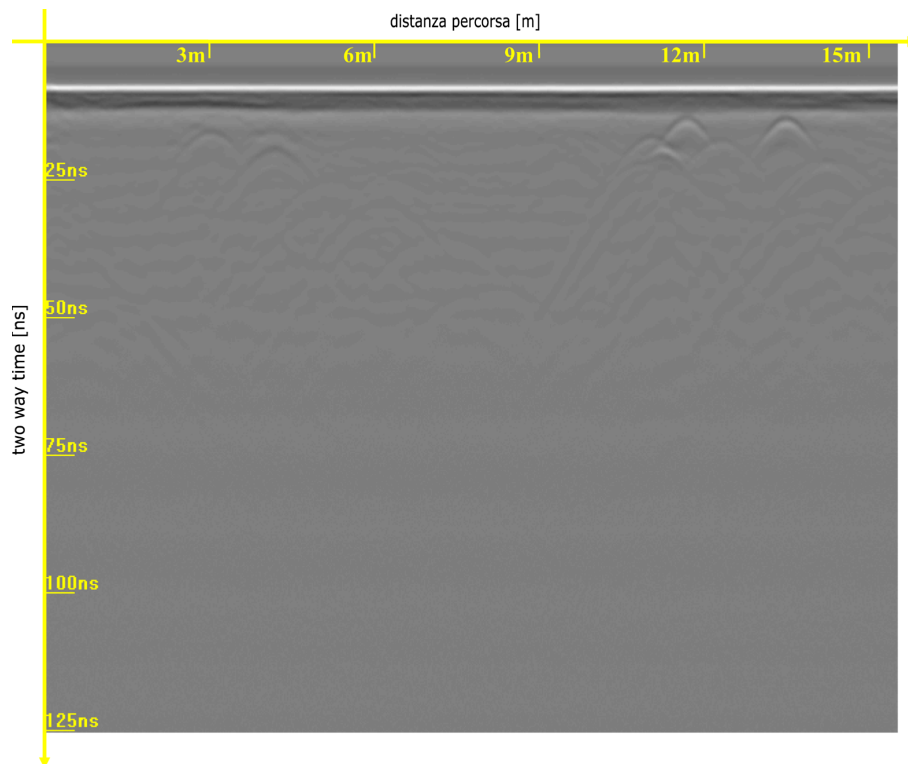


Figura 3.1: Rappresentazione grafica dei dati acquisiti

3.2 Algoritmi di pre-processing: filtraggi

Gli algoritmi di pre-processing vengono eseguiti durante il task di acquisizione e sono costituiti da filtraggi al fine di ridurre i disturbi ed aumentare il SNR.

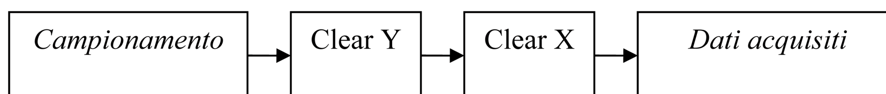


Figura 3.2: Elaborazione pre-processing

Clear Y è un filtraggio passa banda in senso verticale, ovvero lungo l'asse temporale, necessario per eliminare la componente continua introdotta dallo strumento in fase di acquisizione e le frequenze spurie non connesse a bersagli presenti nel mezzo in esame. Le

frequenze di taglio del filtro dipendono dalla frequenza delle antenne utilizzate, ma per la maggior parte dei casi può essere assunta una banda passante compresa fra 100MHz e 1GHz: l'acquisizione di figura 3.1 è stata ottenuta con un'antenna da 200MHz.

Clear X è un'operazione di filtraggio nel dominio dello spazio, quindi in senso orizzontale, e viene applicata durante la formazione della mappa: infatti, come riportato in figura 3.3, il percorso di indagine viene idealmente suddiviso in tratti di lunghezza predefinita Δx_{ant} per ciascuno dei quali viene rappresentato lo sweep Sw' ottenuto effettuando la media tra tutti gli sweep acquisiti lungo il tratto in esame. L'intervallo Δx_{ant} può variare da alcuni centimetri ad alcuni metri, dipendentemente dall'ambito di utilizzo e dalla risoluzione desiderata. Considerando che la durata di uno sweep è pari a t_{sweep} ricavato dalla 2.2, è possibile calcolare il numero M di sweep ricevuti dalla DAD e trasmessi all'unità di elaborazione per ogni tratto Δx_{ant}:

$$M = \frac{\Delta x_{ant}}{V_x \; t_{sweep}}$$

dove V_x è la velocità media con la quale viene percorso Δx_{ant}.

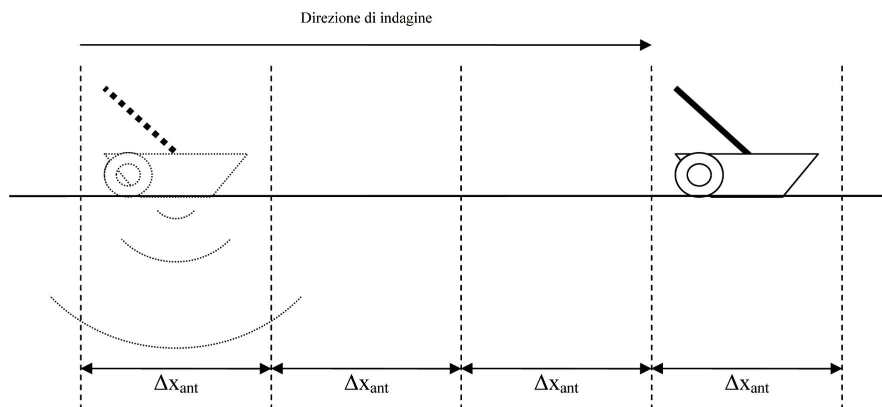


Figura 3.3: Generazione di una serie di sweep ogni intervallo spaziale Δx_{ant}

Quindi sul radargrama ogni colonna di pixel è la rappresentazione dello sweep Sw' calcolato tra gli M sweep acquisiti durante il tratto Δx_{ant} in esame. L'effetto ottenuto è un filtraggio passa basso che riduce il rumore di fondo: il SNR migliora con il diminuire della velocità di indagine, come riportato nel grafico in figura 3.4 nella quale viene confrontata la potenza di rumore nei casi in cui $M = 1$ e $M = 12$.

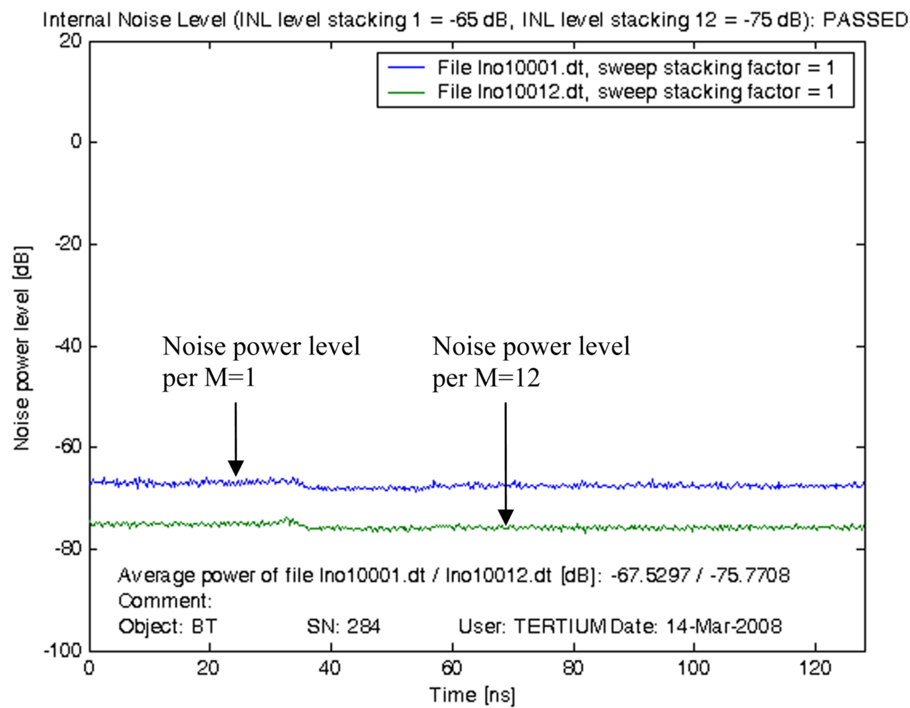


Figura 3.4: Riduzione del rumore impostando la media di 12 sweep ogni tratto Δx_{ant}

3.3 Algoritmi di processing: visualizzazione

Questi algoritmi sono finalizzati alla rappresentazione dei tracciati significativi; necessitano di parametri dipendenti dal tipo di terreno in esame, in quanto le caratteristiche di quest'ultimo influenzano la propagazione delle onde elettromagnetiche: prima di ogni

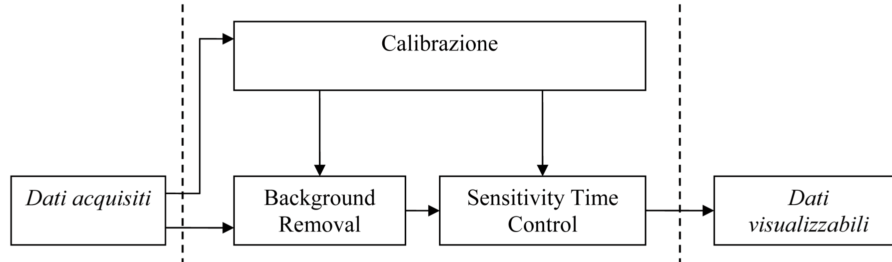


Figura 3.5: Processing per la visualizzazione della mappa

indagine radar è quindi necessario prevedere una *Fase di calibrazione* per ricavare i parametri necessari ad una significativa visualizzazione della mappa. La fase di calibrazione consiste nell'acquisire un buffer contenente un numero di sweep Sw' pari a M_{Buffer}, ovvero consiste nell'effettuare un'indagine della durata di M_{Buffer} tratti Δx_{ant}: una volta terminata, utilizzando il software presente nell'unità di elaborazione, vengono ricavati i parametri necessari per l'applicazione in tempo reale degli algoritmi di visualizzazione della mappa.

Un'indagine georadar completa è quindi suddivisa in due fasi: la fase di calibrazione, nella quale tipicamente vengono acquisiti 128 sweep Sw' (definiti Sw'_{CAL}), e l'indagine effettiva, nella quale vengono acquisiti gli sweep Sw' necessari per la scansione del tragitto voluto (definiti Sw'_{IND}).

I principali algoritmi per l'elaborazione della mappa sono *Background Removal* e *Sensitive Time Control*, applicati secondo lo schema di figura 3.5.

Background Removal consiste nel calcolare lo sweep medio tra gli sweep Sw'_{CAL} e nel sottrarlo da ogni sweep Sw'_{IND}. In particolare definendo:

- Sw'_{CAL_AVG} come lo sweep di calibrazione medio
- S_{CAL_i-j} come l'i-esimo sample del j-esimo sweep di calibrazione

- $S_{CAL_AVG_i}$ come l'i-esimo sample di Sw'_{CAL_AVG}

si calcola

$$S_{CAL_AVG_i} = \frac{\sum_{j=1}^{M_{Buffer}} S_{CAL_i-j}}{M_{Buffer}} \tag{3.1}$$

per $i \in [0; 511]$, ottenendo quindi

$$Sw'_{CAL_AVG} = \{S_{CAL_AVG_0}, S_{CAL_AVG_1} \; ... \; S_{CAL_AVG_511}\} \tag{3.2}$$

Infine, definendo:

- Sw'_{IND_j} come il j-esimo sweep Sw'_{IND}
- Sw'_{BGR_j} come il j-esimo sweep dopo l'operazione di *Background Removal*

l'algoritmo di *Background Removal* consiste nell'eseguire:

$$Sw'_{BGR_j} = Sw'_{IND_j} - Sw'_{CAL_AVG} \tag{3.3}$$

L'operazione di *Background Removal* consente di evidenziare e rappresentare graficamente soltanto le differenze fra il terreno indagato e il terreno di calibrazione: ipotizzando infatti che la composizione del terreno di calibrazione sia simile a quella del terreno indagato, le differenze che vengono rappresentate nel radargramma sono dovute ai target la cui ricerca è lo scopo dell'indagine (utenze, rocce, ecc...). Inoltre, considerando che Sw'_{CAL_AVG} contiene il main bang medio degli sweep Sw'_{CAL} e che questo è approssimativamente uguale a quello degli sweep Sw'_{IND}, questa operazione evita che la mappa radar

riporti la rappresentazione grafica del main bang evidenziando invece i piccoli segnali che risulterebbero mascherati essendo notevolmente più piccoli (figura 3.6).

Figura 3.6: Mappa ottenuta con l'applicazione dell'algoritmo di Background Removal (parte significativa)

Sensitive Time Control è un algoritmo il cui scopo è il recupero della potenza degradata a causa della propagazione dell'onda elettromagnetica nel sottosuolo. Infatti le riflessioni dovute ad oggetti poco profondi sono più ampie di quelle dovuti ad oggetti situati ad una maggiore profondità: nell'esempio di due oggetti identici posti a diverse profondità, nella mappa sarebbe più evidenziato quello a profondità inferiore, mentre sarebbe opportuno che, essendo identici gli oggetti, fossero uguali anche le rappresentazioni grafiche, come in figura 1.4. Quindi per rendere la rappresentazione degli oggetti indipendente dalla profondità alla quale si trovano è necessario stimare l'andamento dell'attenuazione nel terreno in cui avviene l'indagine, in quanto ne è fortemente dipendente: tipicamente la potenza dell'eco captata ha l'andamento caratteristico di figura 3.7.

Durante la fase di calibrazione viene quindi calcolato un vettore composto da N coefficienti di amplificazione che saranno utilizzati durante la fase di indagine per l'equa-

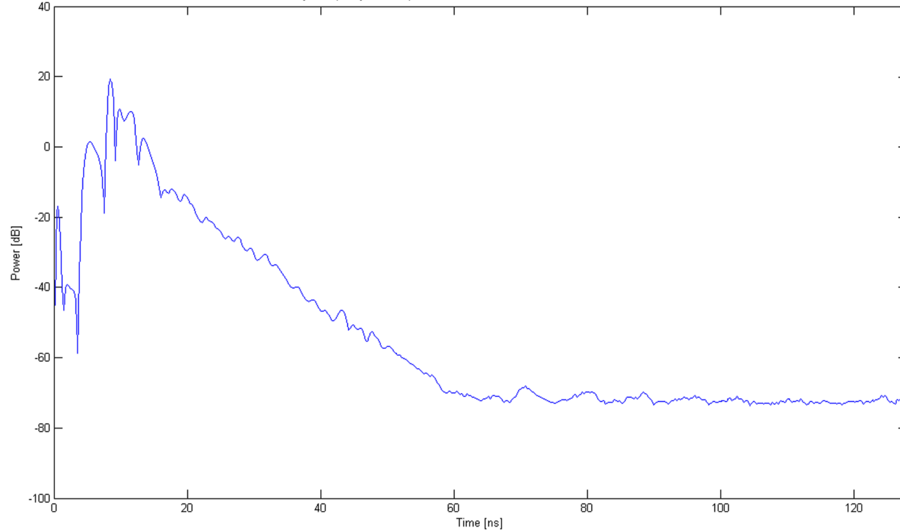


Figura 3.7: Andamento tipico della potenza del segnale A-Scan

lizzazione della potenza; in particolare l'i-esimo coefficiente di amplificazione STC_i, con $i \in [1..N]$, viene definito come

$$STC_i = \frac{2^{15}}{1 + \sqrt{\frac{\sum_{j=1}^{M_{Buffer}} (S_{CAL_i-j})^2}{M_{Buffer}}}} \tag{3.4}$$

dove S_{CAL_i-j} rappresenta l'i-esimo campione del j-esimo sweep Sw'_{CAL}.

Durante l'indagine effettiva l'applicazione dell'algoritmo di *Sensitive Time Control* avviene, per il j-esimo sweep Sw'_{BGR_j}, effettuando la moltiplicazione di ogni campione per il relativo coefficiente STC_i; definendo:

- S_{BGR_i-j} come l'i-esimo campione del j-esimo sweep Sw'_{BGR_j}
- S_{STC_i-j} come l'i-esimo campione del j-esimo sweep dopo l'operazione di *Sensitive Time Control*

viene eseguita:

$$S_{STC_i-j} = S_{BGR_i-j} STC_i \tag{3.5}$$

La rappresentazione grafica di ogni campione S_{STC_i-j} dà la mappa in figura 1.4: in questa i tracciati sono ben più visibili che nella mappa in figura 3.1, soprattutto quelli causati da oggetti profondi.

3.4 Algoritmi di post-processing: focalizzazione

A questa classe appartengono le elaborazioni finalizzate all'estrapolazione delle informazioni riportate nella mappa di figura 1.4. Ad esempio, le elaborazioni più complesse riguardano la costruzione di modelli 3D del sottosuolo confrontando più indagini su diversi percorsi nella stessa porzione di terreno.

Un'elaborazione che, oltre a consistere in un punto di partenza per altre operazioni di post-processing, consente all'operatore una più facile individuazione dei target nella mappa è la *Migrazione geometrica*[4]: questa elaborazione è finalizzata alla visualizzazione di una mappa nella quale vengono evidenziate solamente le zone intorno ai vertici dei tracciati in quanto queste corrispondono alla posizione dei target sepolti[5].

L'algoritmo implementato è riportato dalla 3.6. Questo algoritmo trasforma ogni campione S_{STC_i-j} nel corrispondente campione S_{MIG_i-j} della mappa migrata. La prima operazione è l'individuazione sulla mappa dei tracciati: questo può essere effettuato definendo una soglia S_{th} tale che se $|S_{STC_i-j}| \geq S_{th}$ allora S_{STC_i-j} appartiene ad un tracciato significativo. Per ogni campione della mappa, se $|S_{STC_i-j}| \geq S_{th}$ viene calcolata la media lungo un pattern come quelli in figura 3.9 del quale il campione in esame rappresenta

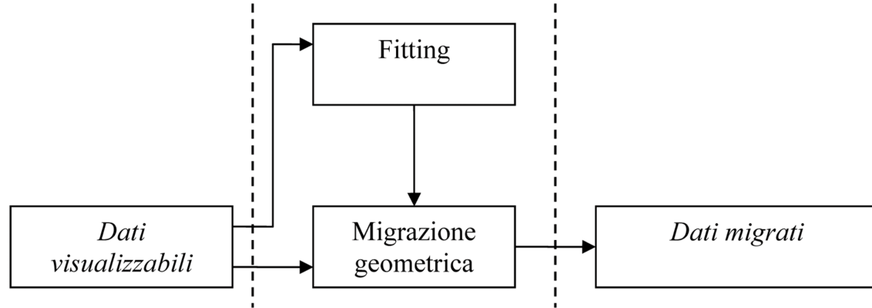


Figura 3.8: Elaborazione post-processing

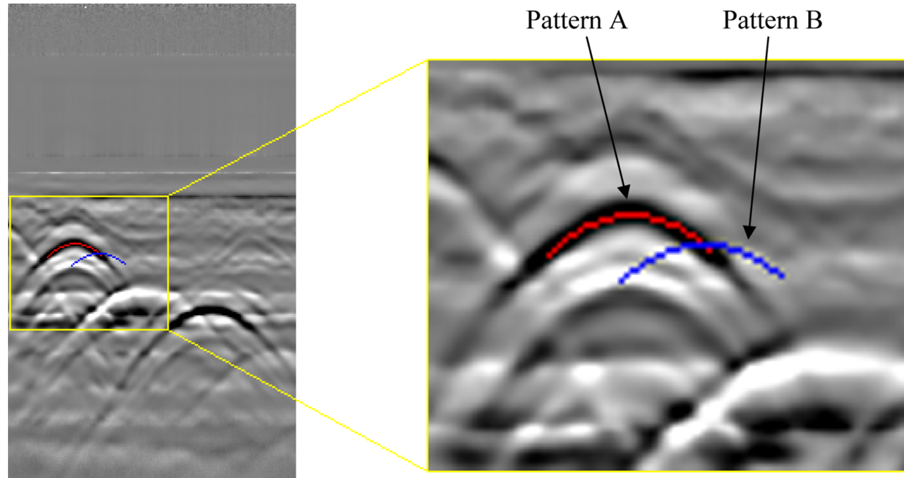


Figura 3.9: Pattern applicati per la migrazione

il vertice, altrimenti viene posto $S_{MIG_i-j} = S_{STC_i-j}$ al fine di lasciare invariate le zone della mappa nelle quali non sono presenti tracciati significativi. In particolare, definendo $N_{Pattern}$ come il numero dei punti costituenti il pattern, viene eseguito:

$$S_{MIG_i-j} = \begin{cases} S_{STC_i-j} & \text{se } |S_{STC_i-j}| < S_{th} \\ \dfrac{\displaystyle\sum_{i,j \in Pattern} S_{STC_i-j}}{N_{Pattern}} & \text{se } |S_{STC_i-j}| \geq S_{th} \end{cases} \tag{3.6}$$

Il risultato grafico di questo algoritmo è riportato in figura 3.10: è possibile distinguere

tre casi:

- se il punto in esame è vertice di un tracciato allora la rappresentazione grafica di S_{MIG_i-j} avrà una tonalità chiara o scura simile a quella di S_{STC_i-j} (fig. 3.10 pattern A), in quanto

$$S_{MIG_i-j} \approx S_{STC_i-j} \forall i, j \in Pattern$$

- se il punto appartiene al tracciato ma non è il vertice allora la rappresentazione grafica di S_{MIG_i-j} avrà una tonalità meno chiara o meno scura rispetto a quella di S_{STC_i-j} (fig. 3.10 pattern B) in quanto, tipicamente, $|S_{MIG_i-j}| < |S_{STC_i-j}|$.

- se $|S_{STC_i-j}| < S_{th}$ la mappa rimane invariata, ovvero $S_{MIG_i-j} = S_{STC_i-j}$. In alternativa è possibile imporre $S_{MIG_i-j} = 0$ in modo da ridurre ulteriormente la presenza di elementi non significativi sulla mappa.

Per l'applicazione di questo algoritmo è necessario determinare l'ampiezza e l'apertura del pattern da utilizzare.

Per quanto riguarda l'ampiezza, un pattern con ampiezza maggiore del tracciato comporterebbe l'inclusione di campioni estranei al tracciato stesso e quindi l'attenuazione di $|S_{MIG_i-j}|$: per la migrazione della mappa in figura 1.4 è possibile impostare l'ampiezza del pattern a 32 pixel, ovvero considerare un intervallo di 32 sweep Sw' per ogni campione individuato come probabile vertice.

L'apertura del pattern dipende invece dalla velocità di propagazione ν_f: infatti, nel caso in cui la dimensione del target sia piccola rispetto alla profondità alla quale si trova, il rapporto fra t_{x0} e t_0 ricavati dalle 1.1 e 1.2 è pari a

$$\frac{t_{x0}}{t_0} = \sqrt{1 + \left(\frac{2x_0}{\nu_f t_0}\right)^2}$$

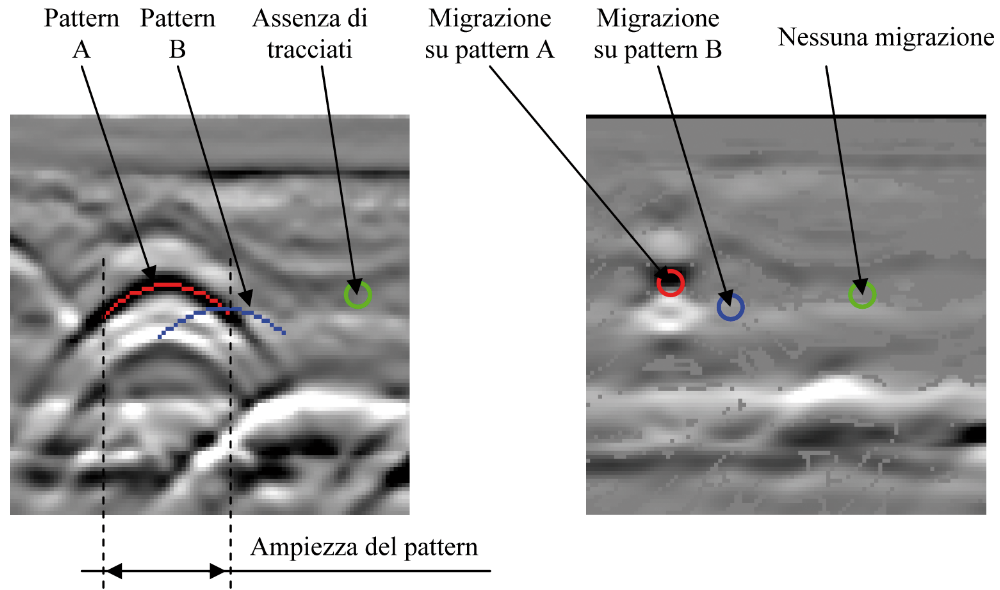


Figura 3.10: Risultato grafico dell'operazione di migrazione (dettaglio)

La stima di ν_f viene effettuata per ogni indagine, a prescindere dall'applicazione dell'algoritmo di migrazione, in quanto consente tramite la 1.2 il calcolo della profondità reale dei target una volta noto il tempo in cui vengono captati gli echi, come visualizzato nel radargramma di figura 1.4.

Un metodo per la stima della velocità ν_f è denominato *Fitting* e consiste nell'individuare sulla mappa un tracciato visibile, il cui vertice si troverà ad una generica ordinata $t_{0\text{-}f}$, e nel confrontarlo con una famiglia di curve note, tutte con il vertice all'ordinata $t_{0\text{-}f}$ ma ricavate con diverse velocità ν_f: la curva appartenente a questa famiglia che meglio coincide con il tracciato rilevato indica una stima della velocità di propagazione ν_f. Un esempio è riportato in figura 3.11: la curva di figura 3.11A coincide con il tracciato rilevato e indica il valore $\nu_f = 105 \cdot 10^6\ m/s$, mentre se la velocità fosse stata di $\nu_f = 75 \cdot 10^6\ m/s$ il tracciato sarebbe coinciso con la curva in figura 3.11B.

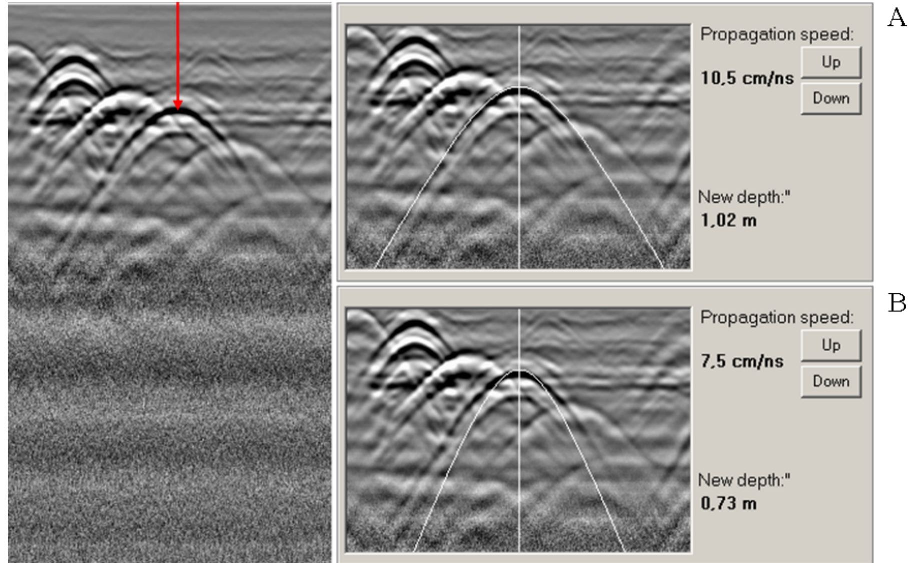


Figura 3.11: Operazione di Fitting

Una volta terminata l'operazione di *Fitting* è possibile visualizzare in tempo reale una mappa come quella riportata in figura 3.12, nella quale le zone più chiare e più scure corrispondono ai target rilevati dei quali è possibile leggere la profondità in metri.

Tale elaborazione dà risultati migliori per tracciati prodotti da oggetti piccoli e superficiali: infatti a parità di profondità del target la curva si allarga al crescere delle dimensioni geometriche in quanto aumenta il rapporto $\frac{t_{x0}}{t_0}$, e ciò può portare a errori di interpretazione. Un'altra causa di errore può essere una non trascurabile differenza del terreno in esame da quello campione, in particolar modo per quanto riguarda la costante dielettrica ϵ_r: ciò porta ad una misura errata della velocità di propagazione e quindi ad una misura falsata della profondità del target. Inoltre in una mappa radar i tracciati come in figura 1.4 non sono gli unici significativi: come si può vedere nell'esempio di figura 3.13 un tubo posto longitudinalmente alla direzione di indagine provoca tracciati orizzontali, che non sarebbero più visibili in seguito all'applicazione dell'algoritmo di migrazione.

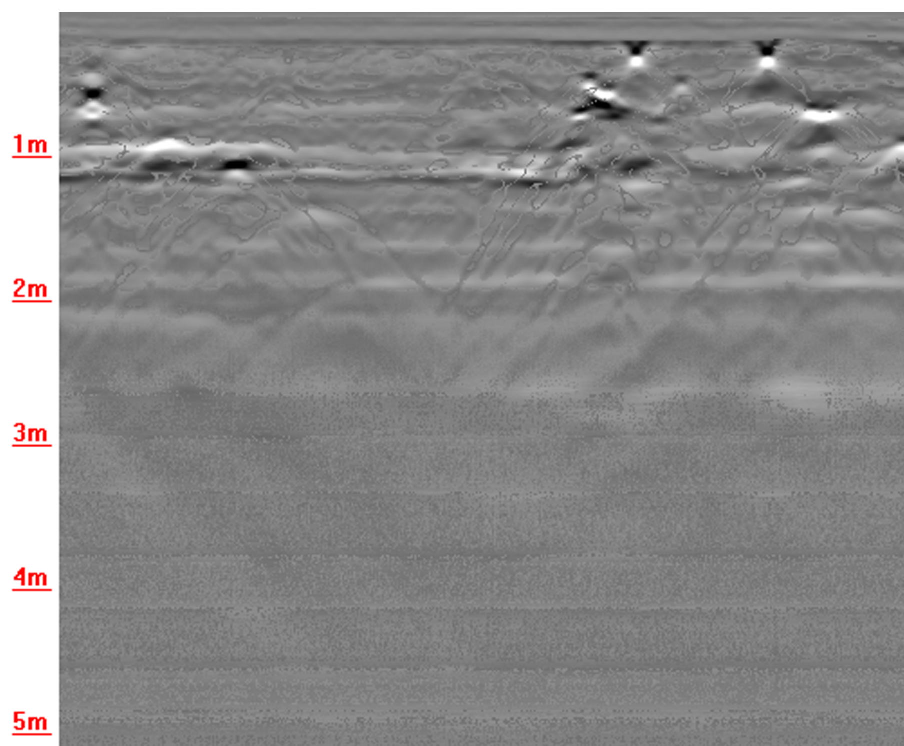


Figura 3.12: Mappa ottenuta con l'applicazione dell'algoritmo di migrazione

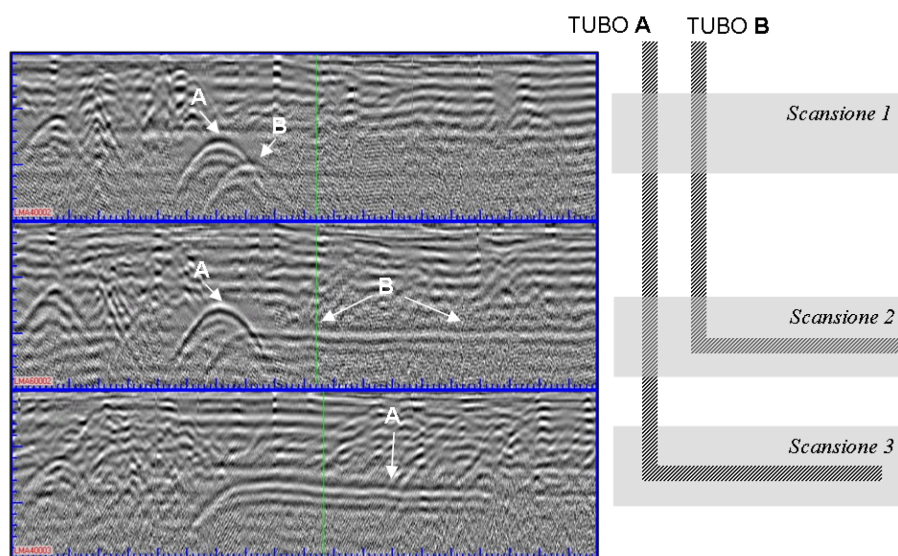


Figura 3.13: Forme tipiche di tracciati georadar

Capitolo 4

Le strutture progettate

4.1 Introduzione

L'architettura della DAD e la sequenza dei task di elaborazione è riportata in figura 4.1; i filtraggi di pre-processing, descritti nel paragrafo 3.2, sono inseriti nel task di acquisizione, mentre gli algoritmi di visualizzazione e migrazione sono inseriti in processi di elaborazione appositi.

Nel sistema in questione la DAD è dotata di una memoria RAM con un bus dati di 8 bit. Le reti descritte nei prossimi paragrafi eseguono i task *Elaborazione 1* ed *Elaborazione 2*

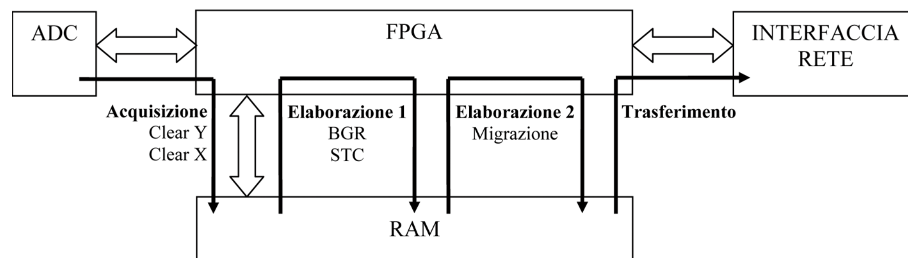


Figura 4.1: Introduzione dei task di elaborazione

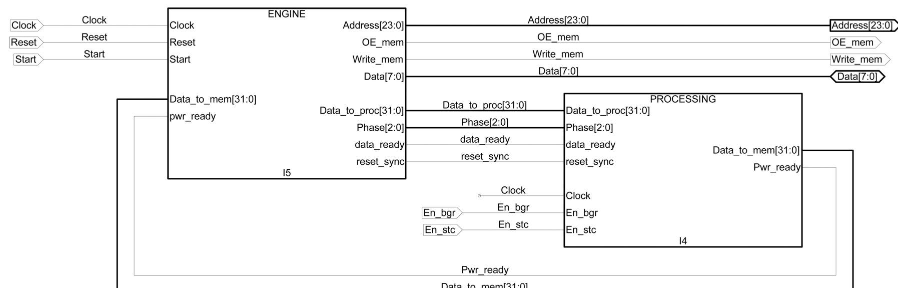


Figura 4.2: Architettura per gli algoritmi Background Removal e Sensitive Time Control

di figura 4.1.

4.2 Arhitettura per gli algoritmi di visualizzazione

La struttura progettata è composta dal blocco *Engine* e dal blocco *Processing*, come riportato in figura 4.2; la rappresentazione dei dati è in virgola fissa con una precisione di 16 bit.

Il blocco *Engine* rappresenta l'interfaccia con la RAM e gestisce i flussi dei dati in ingresso e in uscita; il blocco *Processing* provvede alle elaborazioni.

La sequenza del processo di lettura e scrittura è gestita dal blocco Engine, che legge i dati a 8 bit dalla memoria e li pone in uscita sul bus a 16 bit *Data_to_proc*, che va in ingresso al blocco *Processing*; quando il dato sul bus *Data_to_proc* è pronto viene impostato il segnale *data_ready* che abilita il blocco *Processing* ad effettuare l'elaborazione. Il blocco *Engine* genera anche il vettore *Phase* che indica la fase di elaborazione in corso; in particolare per il calcolo dei vettori *Background Removal* e *Sensitive Time Control* sono previste 3 fasi:

- Fase *Fill_buffer*: Riempimento del buffer composto da 128 sweep

- Fase *Accumulator*: Calcolo dei vettori AVG e STC
- Fase *Write_AVG_and_STC*: Scrittura dei vettori AVG e STC in memoria

Durante la fase *Fill_buffer* viene effettuata la copia dei primi 128 sweep dell'acquisizione nel buffer di calibrazione, ovvero nella zona di memoria compresa tra 0x760000 e 0x79FFFF. Durante questa fase l'uscita *Data_to_mem* del blocco *Processing* è la replica dell'ingresso *Data_to_proc*.

Terminata la fase *Fill_buffer* iniziano le fasi *Accumulator* e *Write_AVG_and_STC* durante le quali vengono calcolati e scritti in memoria vettori AVG e STC come descritto nelle formule 3.1, 3.2 e 3.4. Durante la fase *Accumulator* tramite le reti nelle figure 4.3 e 4.4 vengono calcolati:

$$S_{CAL_AVG_i} = \frac{\sum_{j=1}^{M_{Buffer}} S_{CAL_i-j}}{M_{Buffer}} \tag{4.1}$$

e

$$\overline{P_i} = \frac{\sum_{j=1}^{M_{Buffer}} (S_{CAL_i-j})^2}{M_{Buffer}} \tag{4.2}$$

Quindi alla fine della fase *Accumulator* è già calcolato il valore di $S_{CAL_AVG_i}$ che può essere posto sul bus *Data_to_mem* e scritto in memoria. Il valore di STC_i deve ancora essere calcolato in quanto corrisponde a

$$STC_i = \frac{2^{15}}{1 + \sqrt{\overline{P_i}}}$$

La fase *Write_AVG_and_STC* inizia con la scrittura in memoria di $S_{CAL_AVG_i}$, poi il blocco *Engine* resta in attesa che il blocco *Processing* calcoli STC_i: quando ciò avviene

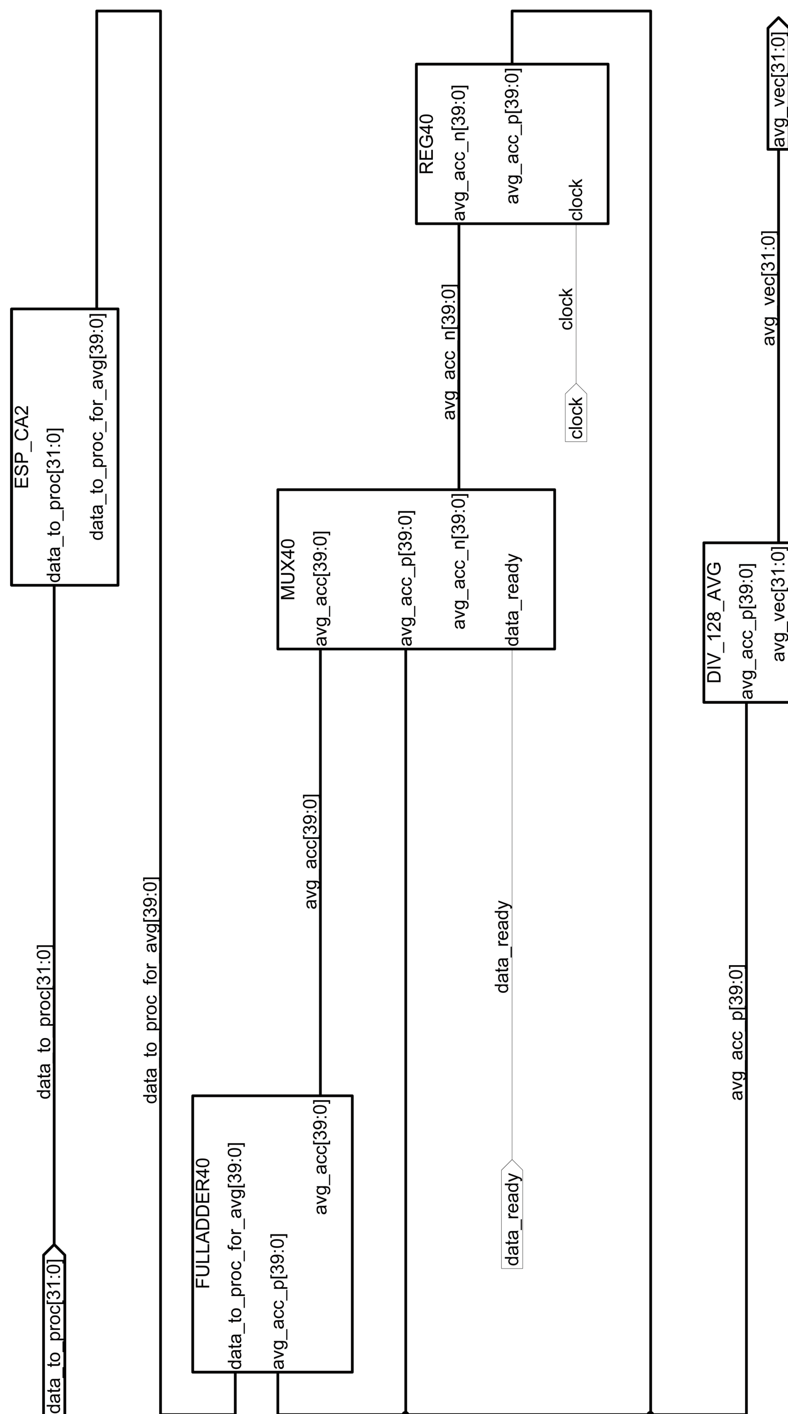


Figura 4.3: Schematico - Rete per il calcolo di $S_{CAL_AVG_i}$

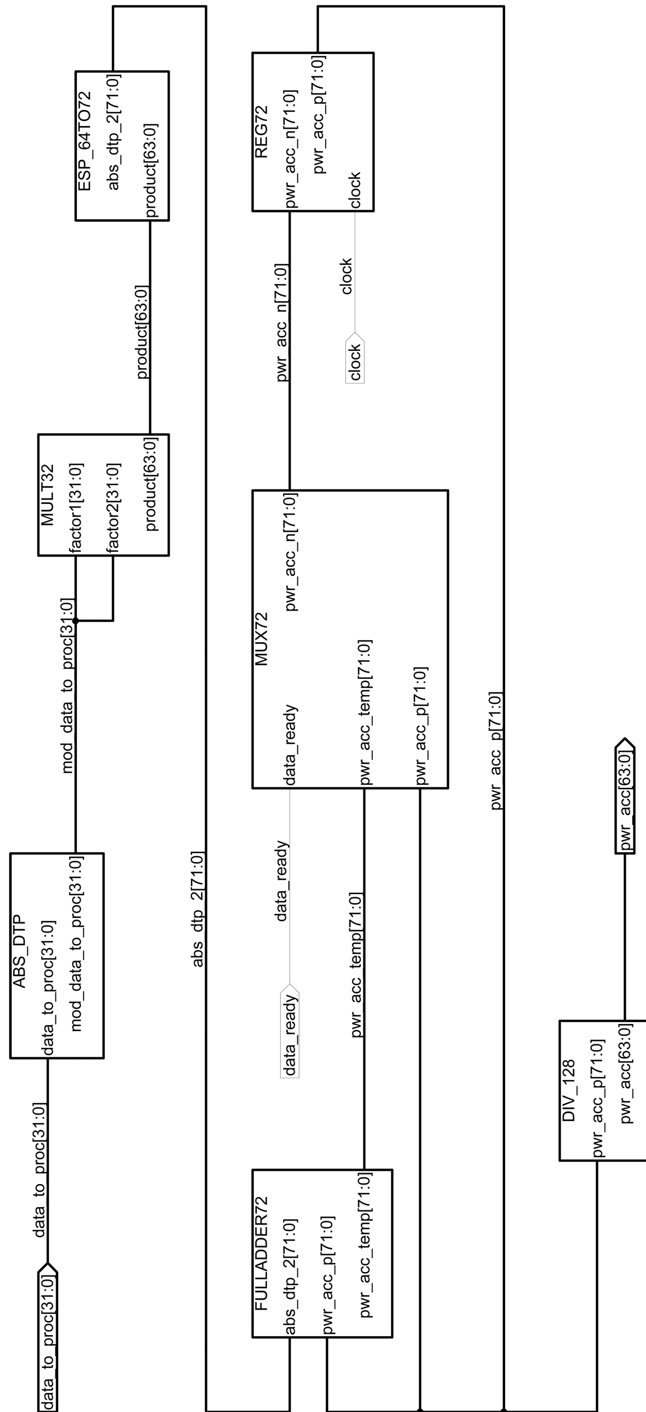


Figura 4.4: Schematico - Rete per il calcolo di $\overline{P_i}$

il segnale *STC_ready* va alto ed il blocco *Engine* scrive STC_i in memoria terminando la fase *Write_AVG_and_STC*.

Il calcolo di STC_i viene completato mediante le reti riportate nelle figure 4.5 e 4.6.

La rete in figura 4.5 realizza

$$Sp_i = \sqrt{\overline{P_i}}$$

utilizzando il metodo della bisezione: il ciclo inizia imponendo sul bus *STC_rad_n* il valore $0x80000000$, ovvero impostando il bit più significativo a '1'; dopo il fronte di clock il bus *Product* contiene $0x4000000000000000$, in quanto $MULT32$ è un moltiplicatore combinatorio già implementato all'interno del FPGA[6]. Il comparatore confronta *pwr_acc* con *Product* e pone l'uscita alta se *Product* > *pwr_acc*: questo caso indica che il risultato della radice quadrata è minore del valore impostato su *pwr_rad_n*, quindi il bit più significativo di *pwr_rad_n* non può essere '1' e viene posto a '0', altrimenti rimane '1'; contemporaneamente il secondo bit più significativo viene posto a '1' ed al colpo di clock successivo viene ripetuto il confronto. Ripetendo questo procedimento per ogni bit di *pwr_rad_n* si ottiene il valore Sp_i sul bus *pwr_rad_p*.

La rete in figura 4.6 realizza

$$STC_i = \frac{2^{15}}{1 + Sp_i}$$

La struttura è analoga alla rete in figura 4.5 sostituendo $\overline{P_i}$ con il valore 2^{15} e ponendo il valore $(1+Sp_i)$ sull'ingresso *factor2* del moltiplicatore. Ciò consente l'utilizzo dello stesso moltiplicatore in quanto il calcolo di STC_i viene effettuato successivamente al calcolo di Sp_i.

Una volta ricavati e scritti in memoria il vettore Sw'_{CAL_AVG} e gli N valori STC_i è possibile calcolare S_{STC_i-j} con la 3.5. La rete utilizzata è riportata in figura 4.7: durante

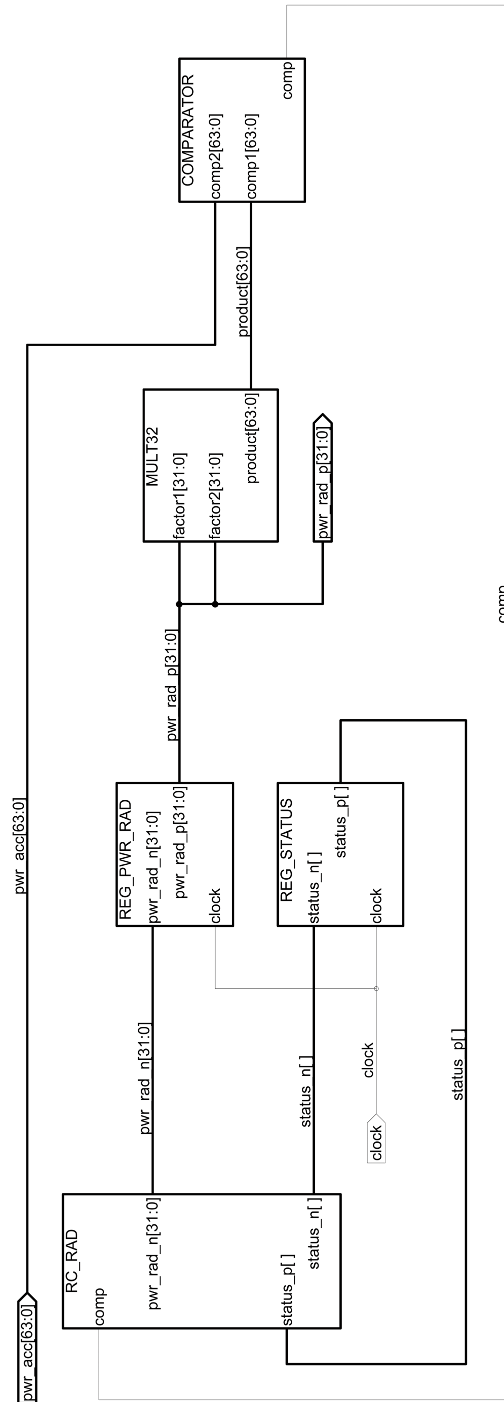


Figura 4.5: Schematico - Rete per il calcolo di Sp_i

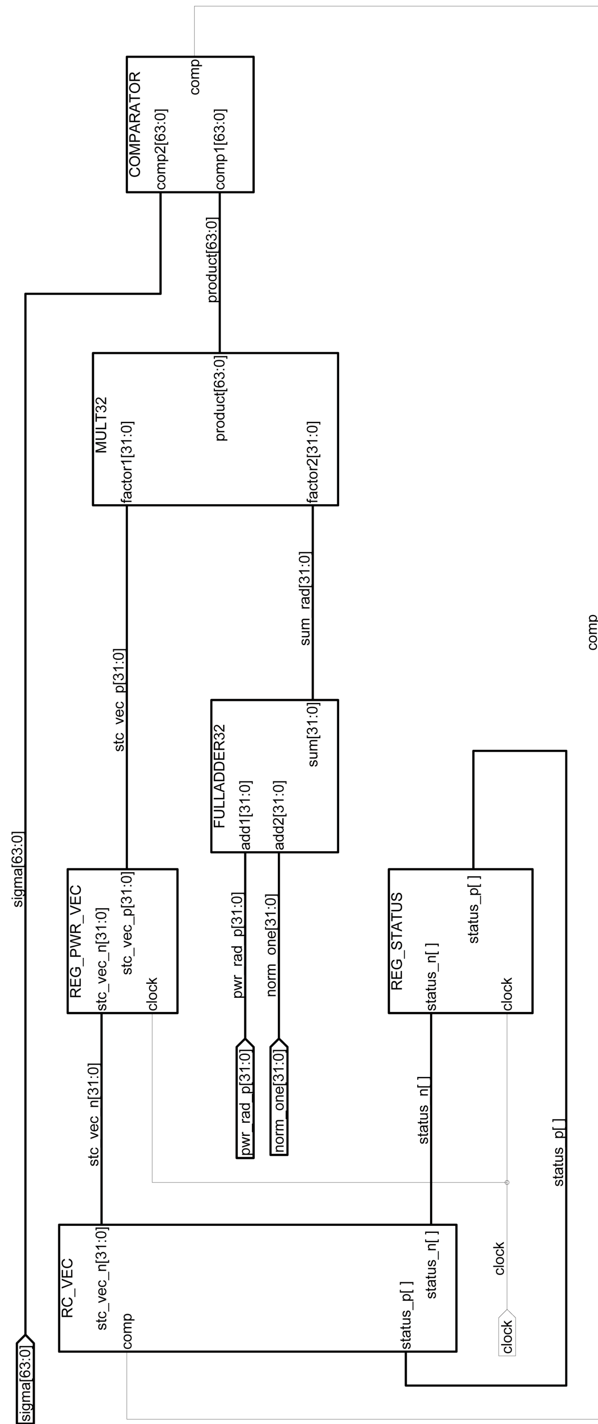


Figura 4.6: Schematico - Rete per il calcolo di STC_i

le fasi *load_avg* e *load_stc* vengono letti dalla memoria i valori $S_{CAL_AVG_i}$ e STC_i e posti rispettivamente agli ingressi AVG_VEC e STC_VEC; durante la fase *load_and_write_sample* viene letto dalla memoria il corrispondente S'_{IND_j-i} e posto all'igresso *Data_to_proc*: essendo la rete combinatoria il dato in uscita S_{STC_i-j} è subito disponibile e può essere scritto in memoria, processo che avviene durante la stessa fase.

Quindi una volta calcolati il vettore Sw'_{CAL_AVG} e gli N valori STC_i inizia il ciclo composto dall'esecuzione delle 3 fasi *load_avg*, *load_stc* e *load_and_write_sample* secondo il diagramma di flusso riportato in figura 4.8.

Le reti nelle figure 4.3, 4.4, 4.5, 4.6 e 4.7 possono essere implementate in un'unica rete, riportata nelle figure 4.9 e 4.10, nella quale il moltiplicatore viene utilizzato, mediante un multiplexer, sia per il calcolo dei coefficienti STC_i sia per l'elaborazione in tempo reale.

L'organizzazione della memoria è riportata in figura 4.11; nella prima parte della memoria viene riportata l'intera acquisizione come output del task di acquisizione, mentre nell'ultima parte viene scritto il risultato dell'elaborazione. A partire dall'indirizzo 0x760000 è presente lo spazio di memoria riservato ai primi 128 sweep dai quali verranno ricavati i due vettori per gli algoritmi *Background Removal* e *Sensitive Time Control*: essendo ciascuno sweep composto da 512 campioni a 32 bit deve avere un'ampiezza di 256 KB. Di seguito vengono memorizzati il vettore Sw'_{CAL_AVG} e gli N coefficienti di amplificazione STC_i ottenuti dagli algoritmi descritti nel paragrafo 3.3: l'ampiezza di questo spazio di memoria è pari a 4 KB, dovendo contenere 2 vettori di 512 campioni a 32 bit.

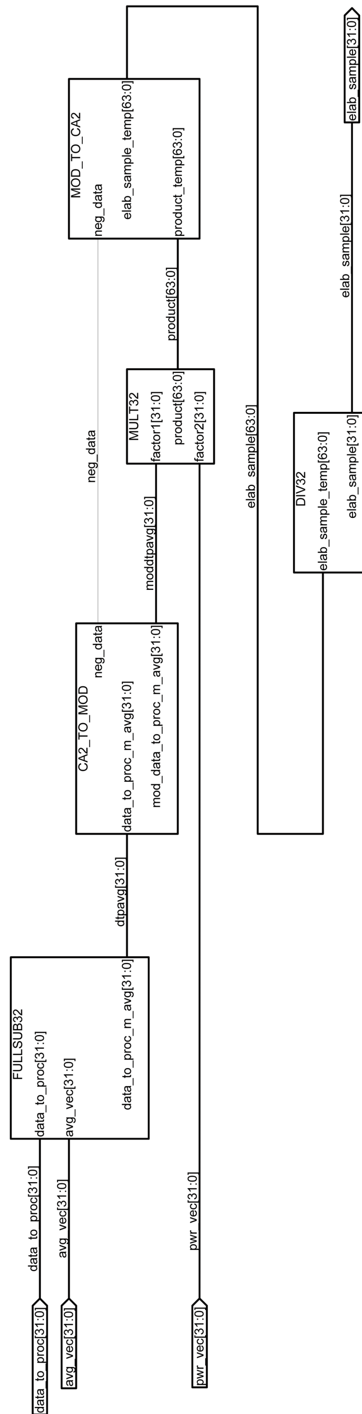


Figura 4.7: Schematico - Rete per il calcolo di S_{STC_i-j}

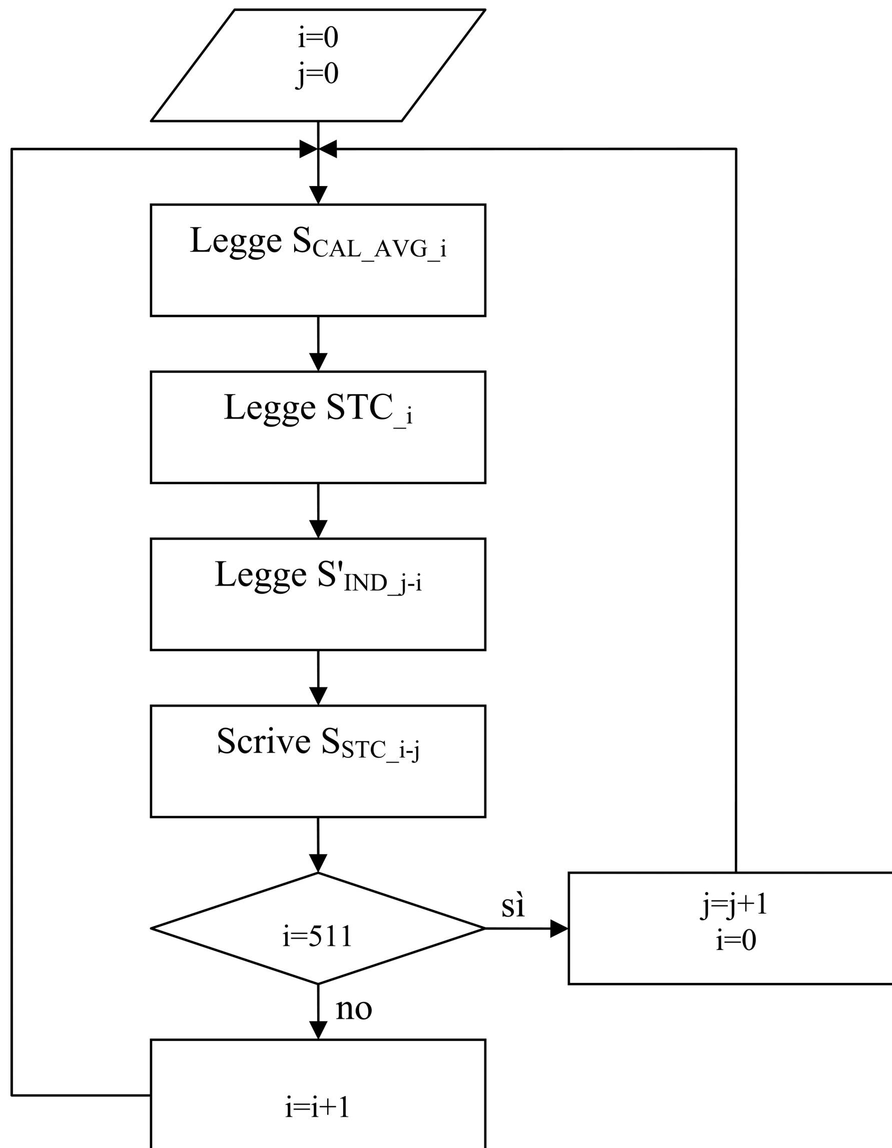


Figura 4.8: Flusso realtime

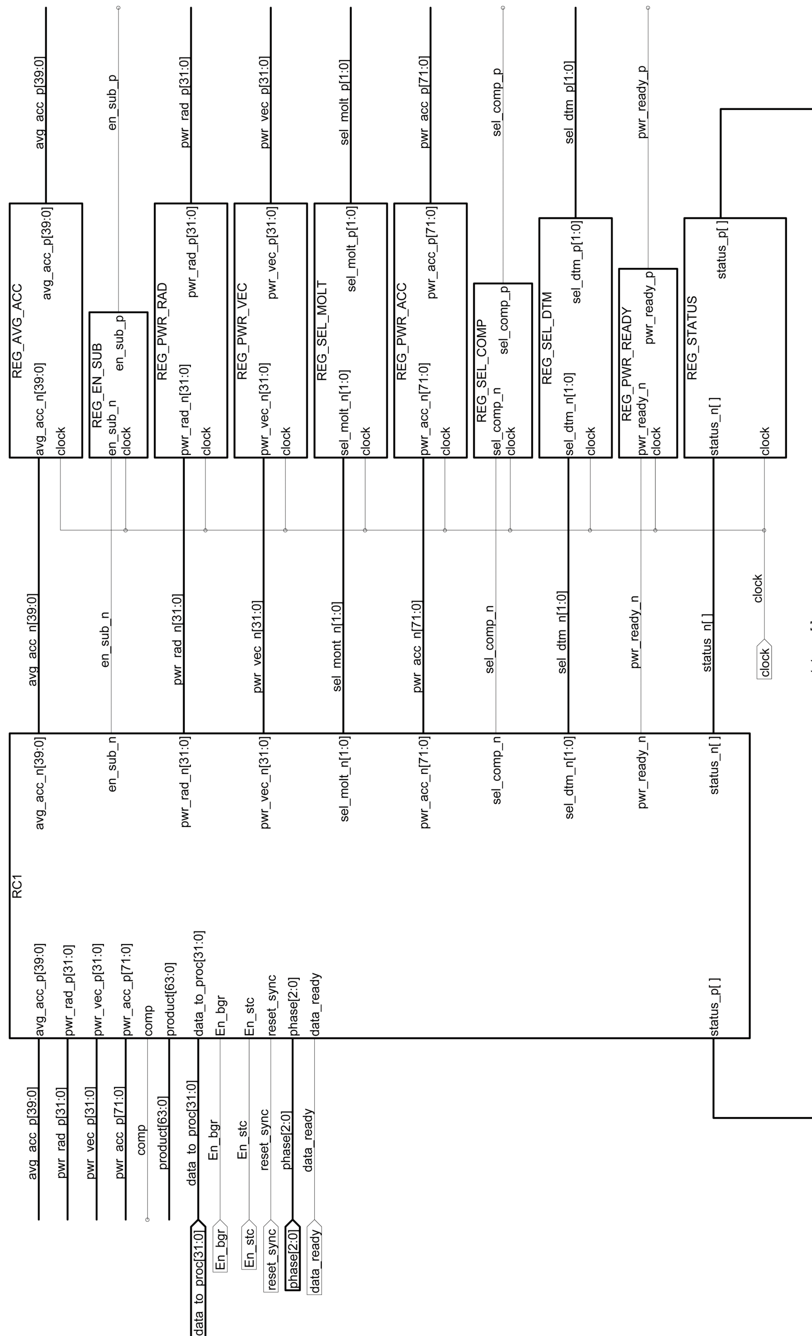


Figura 4.9: *Schematico - Rete globale per gli algoritmi* Background Removal *e* Sensitive Time Control *(prima parte)*

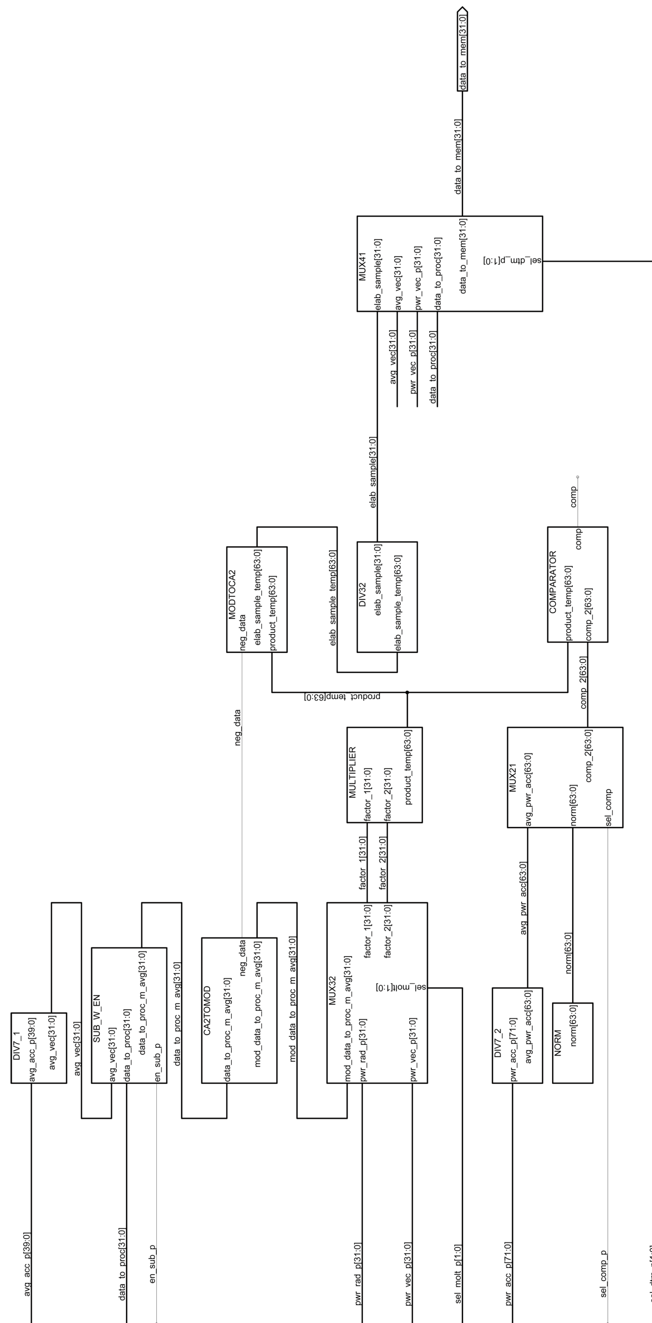


Figura 4.10: Schematico - Rete globale per gli algoritmi *Background Removal* e *Sensitive Time Control* (seconda parte)

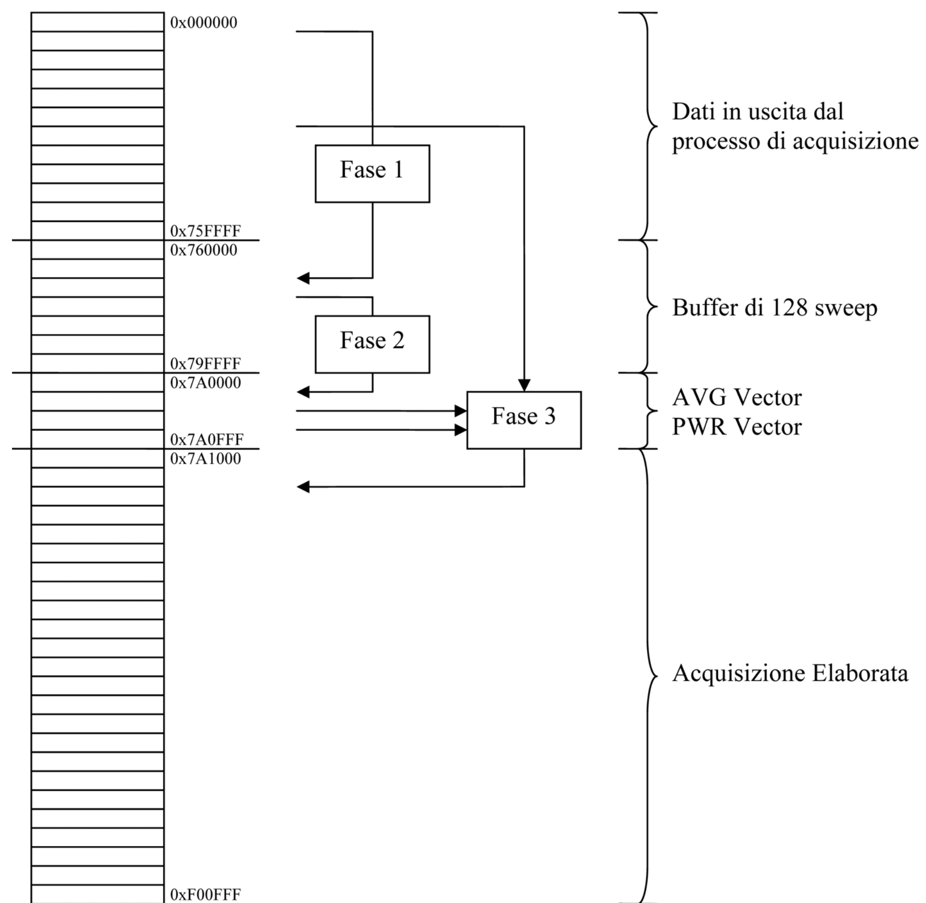


Figura 4.11: Organizzazione della memoria per gli algoritmi Background Removal e Sensitive Time Control

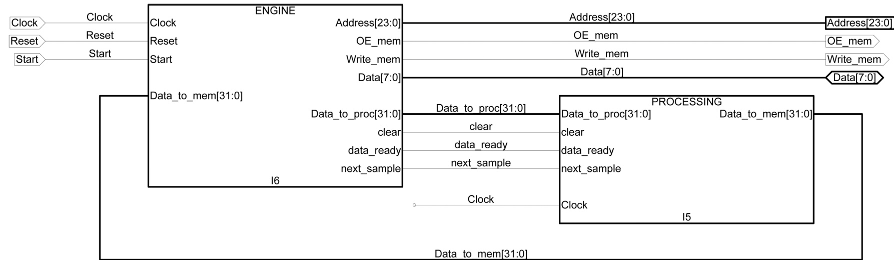


Figura 4.12: Architettura per l'algoritmo Migration

4.3 Arhitettura per gli algoritmi di migrazione

Anche per l'esecuzione degli algoritmi di migrazione è stata progettata una struttura basata sul blocco *Engine* che interfaccia la memoria con il blocco *Processing* che esegue gli algoritmi (figura 4.12).

Il blocco *Engine* legge i dati a 8 bit dalla memoria e li pone in uscita sul bus a 16 bit *Data_to_proc*, che va in ingresso al blocco *Processing*; quando il dato sul bus *Data_to_proc* è pronto viene impostato il segnale *data_ready* che abilita il blocco *Processing* ad effettuare l'elaborazione. Per la gestione dei casi in cui l'algoritmo di migrazione non deve essere eseguito, Il blocco *Engine* genera anche il segnale *Next_sample* che viene attivato quando non deve essere applicato l'algoritmo di migrazione sul pattern.

I parametri necessari per l'algoritmo di migrazione sono la velocità di propagazione del terreno ν_f e l'ordinata alla quale si trova la rappresentazione del *Main bang*, che corrisponde alla soglia: a tal fine l'operatore effettuerà una breve indagine come descritto nel paragrafo 3.4. Una volta ricavati questi parametri viene definito un pattern per ogni campione dello sweep Sw'. La figura 4.13 riporta il pattern relativo ad un campione con ordinata $NP0$: considerando che ogni pixel corrisponde in orizzontale al passo Δx_{ant} e in

verticale a $2t_{C_eq}\nu_f$ è possibile calcolare l'ordinata di ogni punto del pattern:

$$NP_k = \left\lfloor 2\sqrt{(k\,\Delta x_{ant})^2 + \left(\frac{(NP_0 - NP_{th})\,\nu_f\,t_{C_eq}}{2}\right)^2}\right\rfloor \tag{4.3}$$

dove $k \in [-\frac{N_{Pattern}}{2}; \frac{N_{Pattern}}{2} - 1]$; ad esempio, per ricavare NP_k relativo all'estremità sinistra del pattern in figura 4.13 è necessario impostare:

- $NP_0 = 115$: ovvero considerare il 115° sample dello sweep, partendo dall'estremità superiore della mappa
- $k = -16$: l'ampiezza del pattern è di 32 pixel centrato sullo sweep in esame, quindi le estremità si ottengono imponendo $k = -16$ e $k = 15$ mentre il vertice imponendo $k = 0$

e considerare:

- $\nu_f = 105 \cdot 10^6\ m/s$: velocità di propagazione dell'onda nel materiale in esame;
- $NP_{th} = 90$: il *Main bang* in questa mappa è stato rilevato al 90° sample dello sweep, quindi per ogni sweep i primi 90 sample sono privi di significato
- $x_0 = 24mm$: ogni sweep rappresenta un tratto dell'indagine di 24mm
- $t_0 = 25ns$: tempo di campionamento equivalente del segnale radar

Ai fini dell'applicazione dell'algoritmo di migrazione realtime, un metodo più utile per la definizione dei pattern si basa sul ricavare, per ciascuno di essi, quanti campioni separano il vertice SP_0 dal generico punto sul pattern SP_k: definendo questa quantità

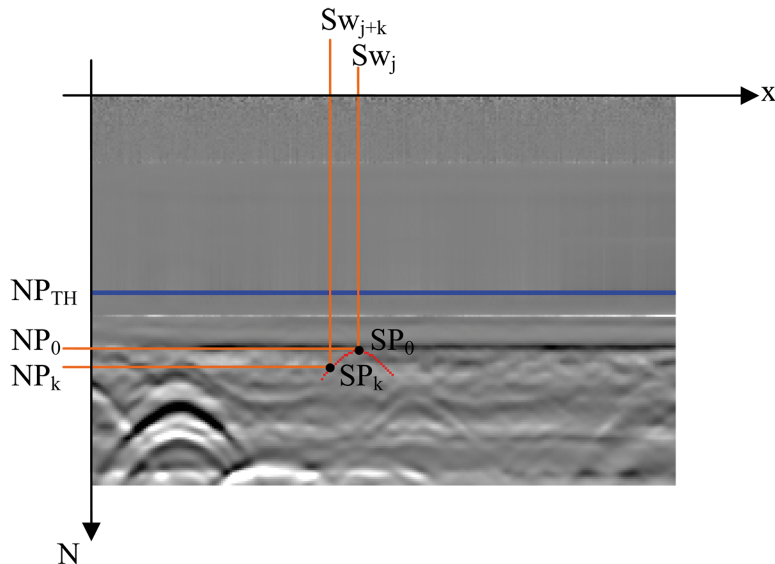


Figura 4.13: Rappresentazione dei pattern sul radargramma

$jump_{n_k}$ si ottiene

$$jump_{n_k} = \begin{cases} -((N - NP_k) + N(-k-1) + NP_0) & \text{se } k < 0 \\ (N - NP_0) + N(k-1) + NP_k & \text{se } k > 0 \end{cases} \tag{4.4}$$

Ad esempio, per il pattern in figura 4.13, applicando le formule 4.3 e 4.4 per $k \in [-16; 15]$ si ottengono i valori $jump_{n_k}$ riportati nella tabella 4.1.

I valori $jump_{n_k}$ vengono calcolati per ogni k e per ogni campione significativo dello sweep (ovvero i $N - NP_{th}$ campioni rappresentanti il terreno sotto la superficie); questa operazione, che deve essere eseguita una sola volta all'inizio dell'indagine, può essere effettuata dal software nell'unità di elaborazione o dal microcontrollore comunque presente nella DAD.

E' quindi possibile realizzare all'interno del blocco *Engine* una struttura per la generazione degli indirizzi basata su 2 registri e un sommatore, come riportata in figura 4.14:

n	k	$jump_{n_k}$	$jump_{n_k}$ in complemento a 2
115	-32	-8187	0xE00D
115	-31	-7669	0xE20B
115	...	...	...
115	-1	-512	0xFE00
115	0	0	0x0000
115	1	512	0x00200
115	...	...	...
115	30	7178	0x1C0A
115	31	7691	0x1E0B

Tabella 4.1: Tabella valori $jump_{n_k}$ (esempio)

inizialmente nel registro indirizzi *REG_ADDRESS_R_MAP* è presente l'indirizzo del campione in esame e nel registro *REG_JUMP* il valore $jump_{n_0} = 0x0000$, quindi il segnale *Address_jump* corrisponde al registro *REG_ADDRESS_R_MAP*; se il campione in esame rispetta i criteri della 3.6 allora nel registro *REG_JUMP* vengono caricati, uno dopo l'altro, gli altri valori $jump_{n_k}$ che, una volta moltiplicati per 4 in quanto ogni campione è composto da 4 byte, vengono sommati all'indirizzo nel registro *REG_ADDRESS_R_MAP* in modo da leggere l'intero pattern.

Dato che la media lungo il pattern viene effettuata solo nel caso descritto dalla 3.6, i valori $jump_{n_0} = 0x0000$ vengono scritti come primo elemento dell'array: quando il registro *REG_JUMP* contiene il valore $0x0000$ viene letto il campione sul vertice, quindi deve essere effettuato il controllo della soglia ed eventualmente deve essere eseguita la media lungo il pattern. Infine, l'algoritmo di migrazione non deve essere applicato né ai campioni sopra la superficie né ai campioni per i quali il pattern corrispondente supera il limite inferiore della mappa: per questi campioni non è necessario valutare la soglia in quanto non devono essere processati. Per consentire questa eccezione, $jump_{n_0}$ per questi campioni viene impostato a $0x7FFF$: quando il registro *REG_JUMP* contiene $0x7FFF$ non viene applicato l'algoritmo di migrazione e viene processato il campione successivo. Queste funzioni vengono svolte dal blocco *Int_next_sample_generator* che genera il segnale *Next_sample_generator*.

L'organizzazione della memoria è riportata in figura 4.15. Anche in questo caso la simulazione avviene ipotizzando che la memoria contenga già l'intera mappa da elaborare; la differenza sta nel fatto che per la migrazione la mappa da elaborare è quella risultante dagli algoritmi di visualizzazione. Nella prima parte della memoria vengono scritti i valori

Figura 4.14: Schematico - Blocco Engine *per l'algoritmo* Migration

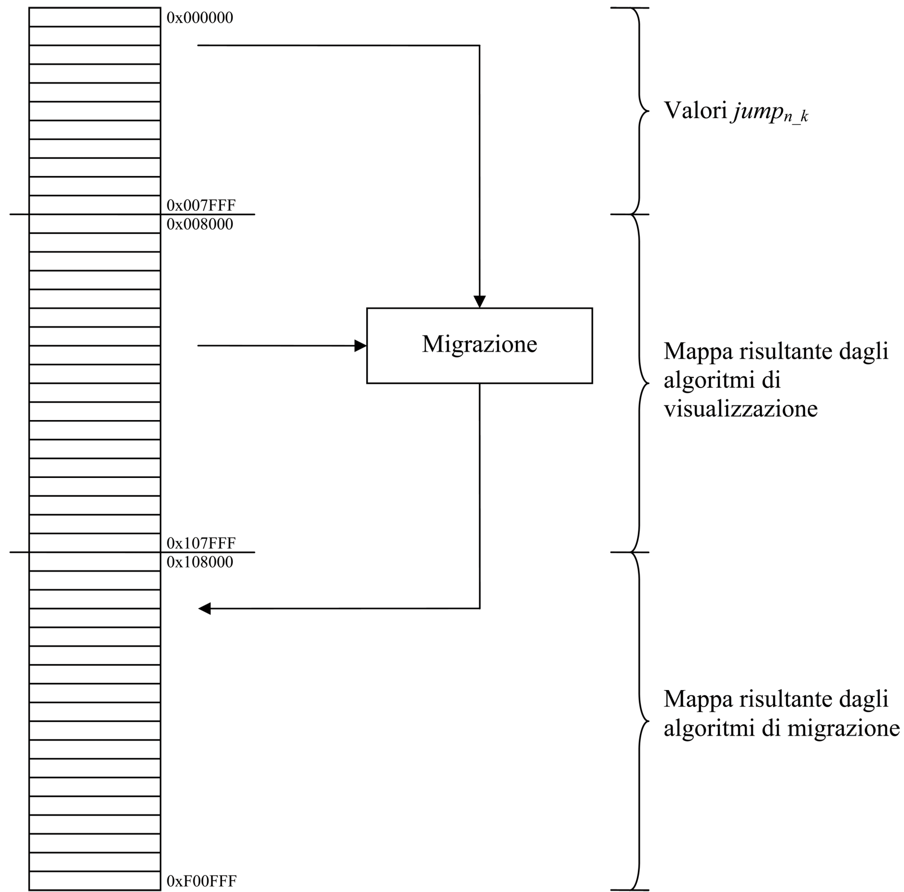


Figura 4.15: Organizzazione della memoria per l'algoritmo Migration

$jump_{n_k}$ (in questo caso essendo $N = 512$, $k = 32$ e i dati a 16 bit lo spazio necessario è pari a 32 KB); successivamente è riportata l'intera acquisizione come output degli algoritmi di visualizzazione mentre nell'ultima parte viene scritto il risultato della migrazione.

Il diagramma di flusso per l'algoritmo di migrazione è riportato in figura 4.16, nella quale $S_{STC_P(k)}$ indica il k-esimo campione del pattern di migrazione. Il ciclo di lettura inizia, per ogni campione, con la lettura del valore $jump_{n_0}$ e del campione S_{STC_i-j}: possono verificarsi tre casi:

1. $jump_{n_0} = 0x7FFF$

2. $jump_{n_0} = 0x0000$ e $|S_{STC_i-j}| < S_{th}$

3. $jump_{n_0} = 0x0000$ e $|S_{STC_i-j}| \geq S_{th}$

Nei primi due casi non deve essere applicato l'algoritmo di migrazione, quindi *Next_sample* viene attivato ed il blocco *Processing* fornisce come dato in uscita $0x0000$, in accordo con la 3.6; l'operazione di scrittura in memoria avviene quindi di seguito e viene elaborato il campione successivo. Nel terzo caso invece vengono letti in successione gli altri valori $jump_{n_k}$ con $k = [-16; -1]$ e $k = [1; 15]$: ciascuno di questi valori, come riportato in figura 4.14, viene sommato all'indirizzo del campione in esame affinché possano essere letti tutti i campioni appartenenti al pattern; in questo caso il blocco *Processing* esegue la media e restituisce il risultato sul bus *Data_to_mem* affinché possa avvenire la scrittura in memoria.

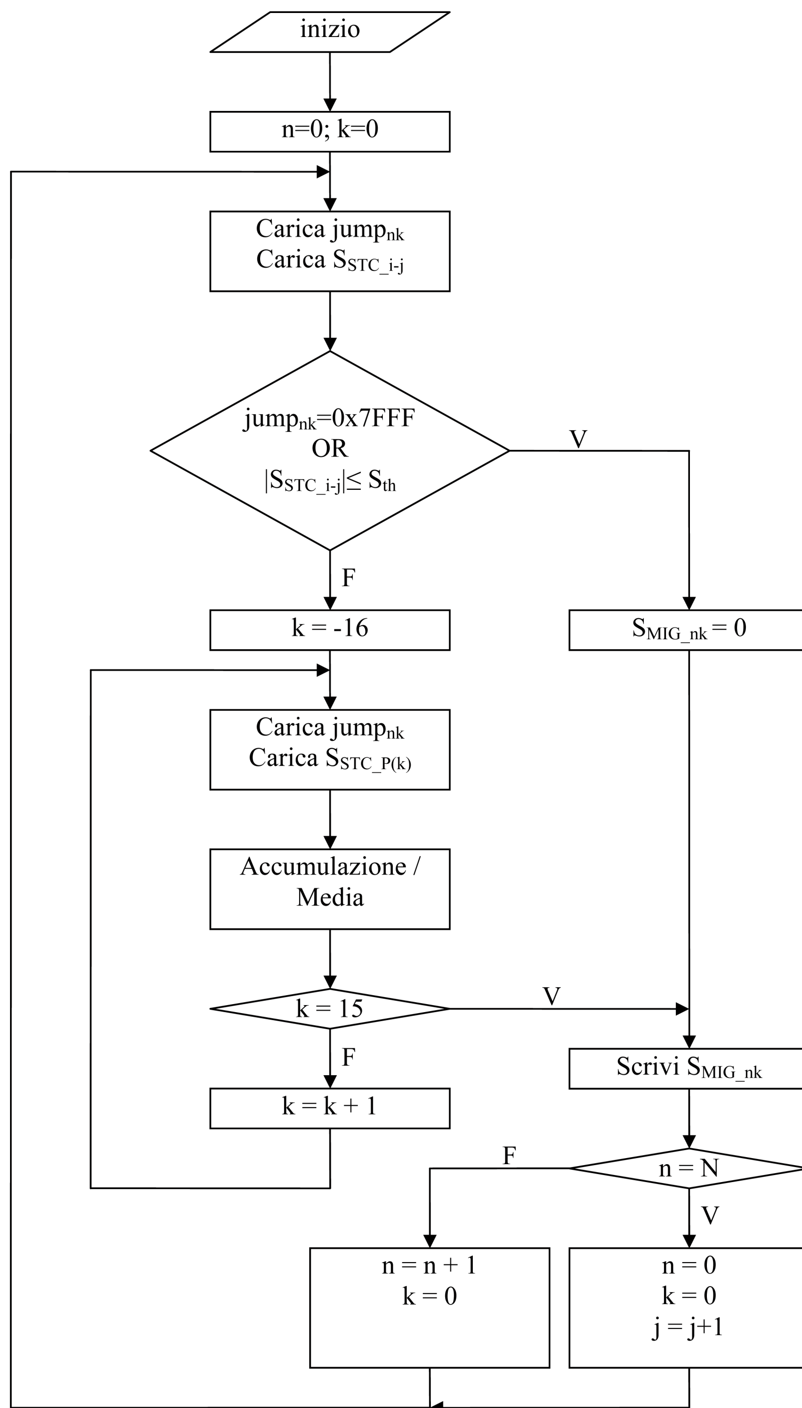


Figura 4.16: Diagramma di flusso per l'algoritmo Migration

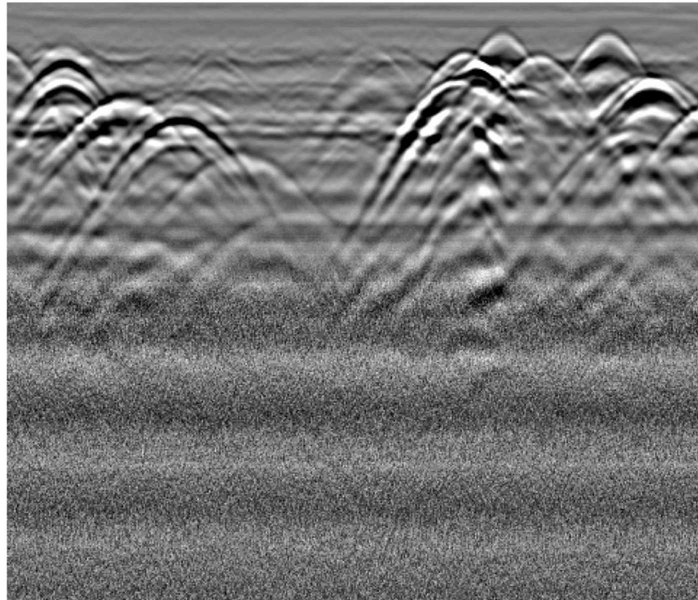
Capitolo 5

Conclusioni

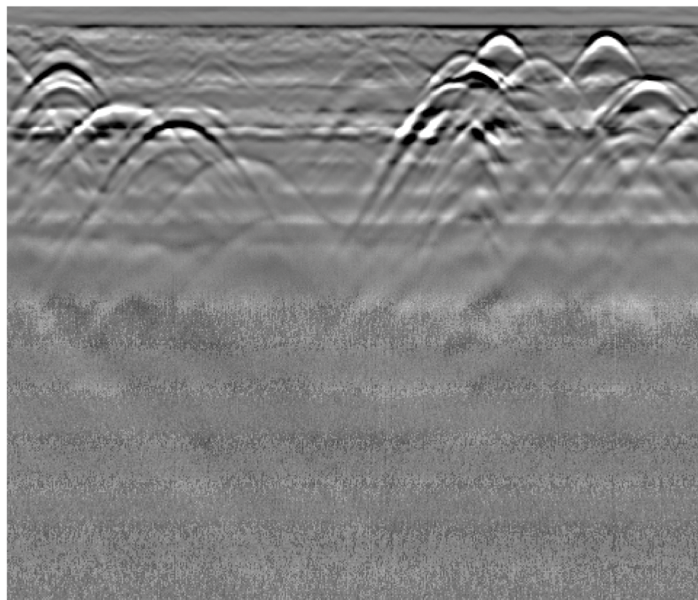
Lo scopo di questo elaborato è stato la progettazione di algoritmi per l'elaborazione dati georadar da implementare all'interno di un dispositivo FPGA. Gli algoritmi considerati (*Background Removal*, *Sensitive Time Control* e *Migration*) sono quelli che attualmente vengono svolti per la visualizzazione delle mappe radar sul sistema *Detector* prodotto da I.D.S. Spa. L'attuale architettura della DAD FastWave consente quindi l'introduzione di soluzioni embedded per l'elaborazione dati al fine di ridurre o eliminare i tempi di elaborazione dell'unità esterna. Per quanto riguarda l'applicazione degli *algoritmi di visualizzazione* descritti nel paragrafo 3.3, la figura 5.1 mostra il confronto tra l'immagine radar ottenuta con il software *K2 Detector* e quella ottenuta simulando il codice VHDL riportato nel'appendice A.2; analogamente la figura 5.2 mostra il confronto per gli *algoritmi di focalizzazione*.

Attualmente è in previsione lo sviluppo di una diversa versione della DAD, mantenendo comunque l'attuale architettura basata sull'utilizzo di un microcontrollore per la configurazione dei parametri di lavoro e di un dispositivo FPGA per la gestione del flusso dati;

una possibile evoluzione può consistere nell'utilizzo di un dispositivo All Programmable SoC[7] al posto degli attuali FPGA e microcontrollore separati, come ad esempio un dispositivo della famiglia Xilinx®Zynq™-7000 che integrano un processore ARM®Cortex™-A9, presentati anche a Firenze durante un workshop il 31 Gennaio 2013 al quale ho partecipato insieme al dott. Marco Consani di Tertium Technology. L'utilizzo di un dispositivo di questa famiglia consentirebbe anche una semplificazione della BOM e delle dimensioni dell'attuale DAD, diminuendo i costi di produzione e facilitando ulteriormente lo sviluppo di dispositivi portatili o l'integrazione della DAD direttamente nei macchinari per le escavazioni.



a)



b)

Figura 5.1: Algoritmi di visualizzazione: confronto fra le mappe radar ottenute con il software K2 Detector (figura a) e con la simulazione del codice VHDL (figura b)

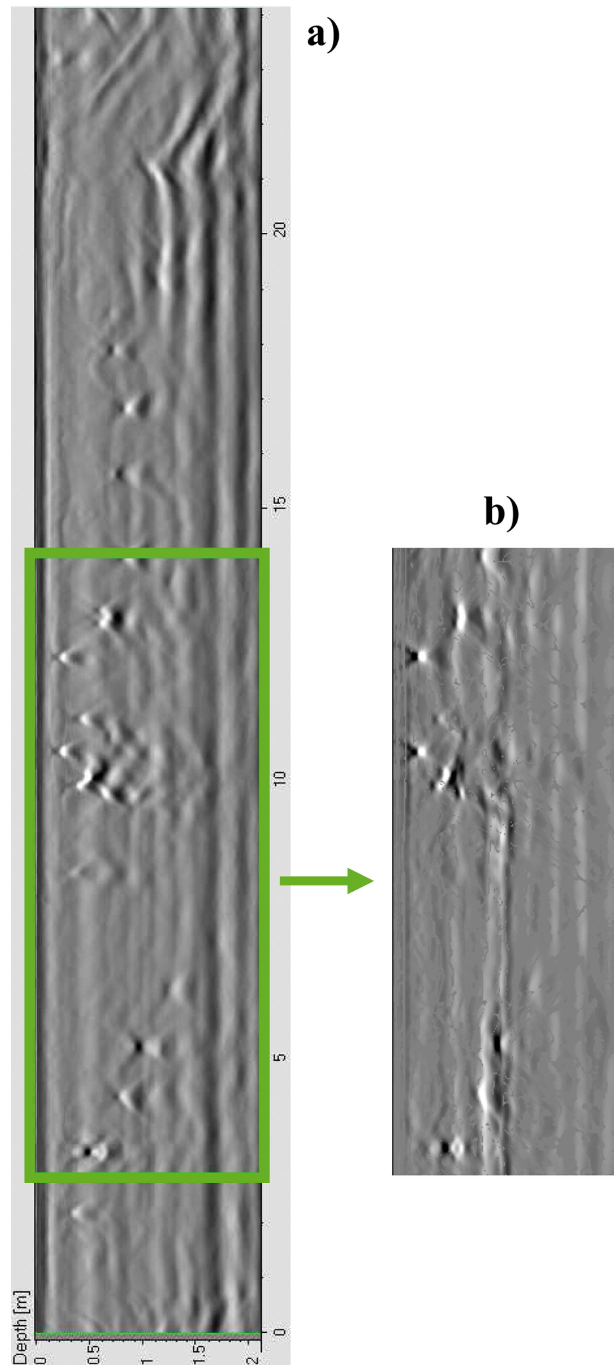


Figura 5.2: Algoritmi di focalizzazione: confronto fra le mappe radar ottenute con il software K2 Detector (figura a) e con la simulazione del codice VHDL (figura b)

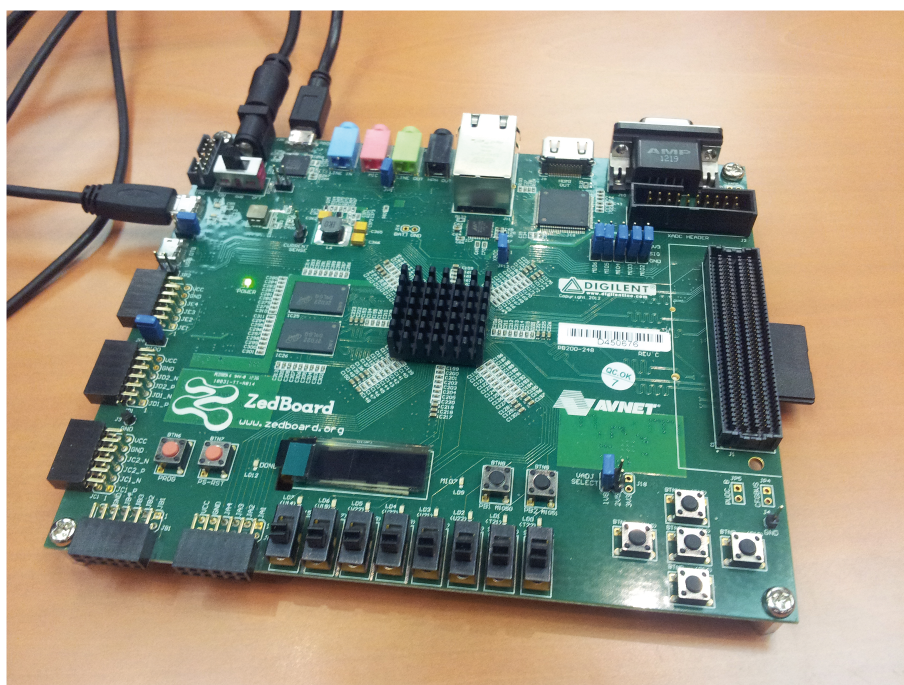


Figura 5.3: La scheda di valutazione Zedboard presentata durante il workshop del 31/01/2013

Figura 5.4: Partecipazione al workshop

Appendice A

VHDL

A.1 Procedura per le simulazioni

In questa appendice viene riportato il codice VHDL[8] delle reti descritte nel capitolo 4.

Per la simulazione del codice VHDL relativo agli *Algoritmi di visualizzazione* sono stati utilizzati gli stessi dati impiegati per lo sviluppo del software *K2 Detector* citato nel paragrafo 1.1: questi dati sono costituiti dai campioni del segnale *A-scan* di figura 1.3 scritti in formato proprietario, sono stati quindi convertiti in formato esadecimale a 16 bit all'interno di un file di testo ed importati all'interno dei *testbench*. Anche i dati in uscita dalle simulazioni (composti da valori in formato esadecimale) vengono scritti in un file di testo e successivamente convertiti in formato grafico per la visualizzazione (ogni valore esadecimale scritto nel file di testo generato dalle simulazioni viene convertito in un pixel di un'immagine bitmap); per simulare l'intera architettura della DAD, nel testbench viene descritta anche la memoria RAM nella quale vengono scritti sia i dati in ingresso che quelli di uscita, secondo lo schema in figura A.1.

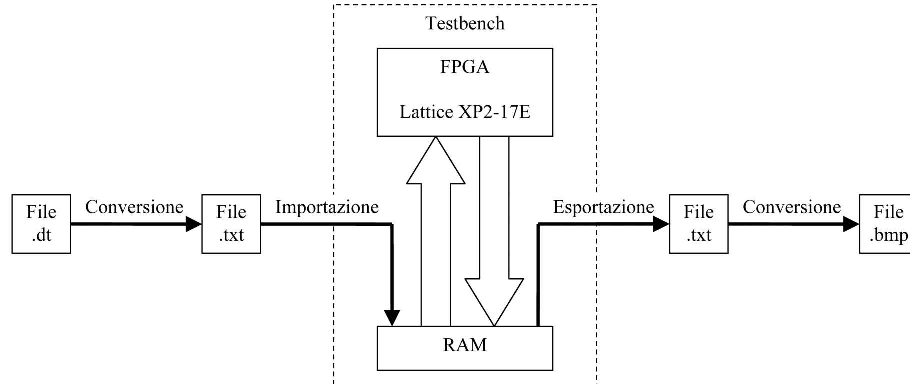


Figura A.1: Procedimento per la simulazione degli algoritmi di visualizzazione

Analoga struttura viene utilizzata anche per la simulazione del VHDL relativo agli *Algoritmi di migrazione*, con la differenza che in questo caso nel relativo *testbench* viene effettuata l'importazione dei dati in uscita dalla simulazione degli *algoritmi di visualizzazione.*

Per la descrizione in VHDL delle strutture progettate e il disegno degli schematici è stato utilizzato l'ambiente di sviluppo *Lattice ISPLever 7.0 SP2*, nel quale è integrato il software di simulazione *ModelSim LATTICE 6.2g.*

A.2 VHDL per gli algoritmi di visualizzazione

A.2.1 VHDL Blocco Engine

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity Engine is
generic(buffer_lenght : integer := 128;
        sample_in_sweep : integer := 512;
        num_sweep_in_acq:   integer:= 512);
-- La costante "num_sweep_in_acq" è inserita per operazioni di debug:
-- la simulazione infatti si interrompe dopo l'elaborazione di un numero
-- di sweep pari alla costante "num_sweep_in_acq"
port(Clock: in std_logic;
     Reset: in std_logic;
     Start: in std_logic;
-- Uscita del bus indirizzi
     Address: out std_logic_vector(23 downto 0);
-- Uscite per pilotaggio della RAM
```

```
      OE_mem: out std_logic;
      Write_mem: out std_logic;
-- Bus dati interni
      Data_to_proc: out std_logic_vector(31 downto 0);
      Data_to_mem: in std_logic_vector(31 downto 0);
-- Segnali di controllo e pilotaggio del blocco Engine
      Phase: out std_logic_vector(2 downto 0);
      data_ready: out std_logic;
      STC_ready: in std_logic;
      reset_sync: out std_logic;
-- Bus dati verso la RAM
      Data: inout std_logic_vector(7 downto 0));
end;
architecture Behav of Engine is
-- definisco gli indirizzi di memoria corrispondenti
-- all'inizio delle zone dove sono scritti i dati utili
constant offset_acq:        std_logic_vector(23 downto 0) := X"000000";
constant offset_buffer:     std_logic_vector(23 downto 0) := X"760000";
constant offset_vectors:    std_logic_vector(23 downto 0) := X"7A0000";
constant offset_elab_sweep: std_logic_vector(23 downto 0) := X"7A1000";
-- definisco la codifica del segnale Phase_n:
constant fill_buffer:            std_logic_vector(2 downto 0) := "000";
constant accumulator:            std_logic_vector(2 downto 0) := "001";
constant write_avg_and_stc:      std_logic_vector(2 downto 0) := "010";
constant load_avg:               std_logic_vector(2 downto 0) := "011";
constant load_stc:               std_logic_vector(2 downto 0) := "100";
constant load_and_write_sample:  std_logic_vector(2 downto 0) := "101";
-- Definizione dei tipi enumerati (stati della macchina a stati
-- e selettore degli indirizzi)
type state_engine_type is (
     s0,
     read_acq_to_buffer_i1_a, wait1,
     read_acq_to_buffer_i0_a, wait2,
     write_acq_in_buffer_i1_a, write_acq_in_buffer_i1_b, wait3,
     write_acq_in_buffer_i0_a, write_acq_in_buffer_i0_b, wait4,
     write_acq_in_buffer_d0_a, write_acq_in_buffer_d0_b, wait5,
     write_acq_in_buffer_d1_a, write_acq_in_buffer_d1_b, wait6,
     read_buffer_i1_a, wait9,
     read_buffer_i0_a, wait10,
     read_buffer_d0_a, wait11,
     read_buffer_d1_a, wait12,
     write_avg_i1_a, write_avg_i1_b, wait15,
     write_avg_i0_a, write_avg_i0_b, wait16,
     write_avg_d0_a, write_avg_d0_b, wait17,
     write_avg_d1_a, write_avg_d1_b, wait18,
     write_stc_i1_a, write_stc_i1_b, wait21,
     write_stc_i0_a, write_stc_i0_b, wait22,
     write_stc_d0_a, write_stc_d0_b, wait23,
     write_stc_d1_a, write_stc_d1_b, wait24,
     read_avg_i1_a, wait27,
     read_avg_i0_a, wait28,
     read_avg_d0_a, wait29,
     read_avg_d1_a, wait30,
     read_stc_i1_a, wait33,
     read_stc_i0_a, wait34,
     read_stc_d0_a, wait35,
     read_stc_d1_a, wait36,
     read_smp_i1_a, wait39,
     read_smp_i0_a, wait40,
     write_elab_sweep_i1_a, write_elab_sweep_i1_b, wait41,
     write_elab_sweep_i0_a, write_elab_sweep_i0_b, wait42,
     write_elab_sweep_d0_a, write_elab_sweep_d0_b, wait43,
     write_elab_sweep_d1_a, write_elab_sweep_d1_b, wait44);
type sel_address_type is (
sel_addr_r_vec, sel_addr_r_acq, sel_addr_w);
signal addr_r_vec_p, addr_r_vec_n: std_logic_vector(23 downto 0);
signal addr_r_acq_p, addr_r_acq_n: std_logic_vector(23 downto 0);
signal addr_w_p, addr_w_n: std_logic_vector(23 downto 0);
signal sel_address_p, sel_address_n : sel_address_type;
signal OE_mem_p, OE_mem_n : std_logic;
signal Write_mem_p, Write_mem_n : std_logic;
signal data_to_proc_p, data_to_proc_n : std_logic_vector(31 downto 0);
signal Phase_p, Phase_n : std_logic_vector(2 downto 0);
signal data_ready_p, data_ready_n : std_logic;
signal reset_sync_p, reset_sync_n : std_logic;
signal start_sync_p, start_sync_n : std_logic;
signal state_engine_p, state_engine_n : state_engine_type;
signal data_p, data_n : std_logic_vector(7 downto 0);
begin
-- Processi sequenziale: le 3 macchine aggiornano gli stati; impongo il reset sincrono e
-- quindi lo descrivo nel processo combinatorio
seq: process (clock) begin
  if (clock'event and clock='1') then
       addr_r_vec_p <= addr_r_vec_n;
       addr_r_acq_p <= addr_r_acq_n;
```

```
        addr_w_p <= addr_w_n;
        sel_address_p <= sel_address_n;
        OE_mem_p <= OE_mem_n;
        Write_mem_p <= Write_mem_n;
        data_to_proc_p <= data_to_proc_n;
        Phase_p <= Phase_n;
        data_ready_p <= data_ready_n;
        state_engine_p <= state_engine_n;
        data_p <= data_n;
        reset_sync_p <= reset_sync_n;
        start_sync_p <= start_sync_n;
   end if;
end process;
-- processo combinatorio
com: process (
      -- Segnali in ingresso al blocco engine
      reset_sync_p, start_sync_p, stc_ready, data_to_mem,
      -- stato
      state_engine_p,
      addr_r_acq_p, addr_r_vec_p,
      addr_w_p, data, data_to_proc_p)
      begin
      if reset_sync_p = '0' then
        OE_mem_n <= '0';
        Write_mem_n <= '1';
        addr_r_acq_n <= offset_acq;
        addr_r_vec_n <= offset_vectors;
        addr_w_n <= offset_buffer;
        sel_address_n <=  sel_addr_r_acq;
        data_n <= data;
        data_to_proc_n(31 downto 24) <= (others => '0') ;
        data_to_proc_n(23 downto 16) <= (others => '0') ;
        data_to_proc_n(15 downto 8) <= (others => '0') ;
        data_to_proc_n(7 downto 0) <= (others => '0') ;
        Phase_n <= fill_buffer;
        data_ready_n <= '0';
        data_n <= data;
        state_engine_n <= s0;
       else
         case state_engine_p is
-- In questa fase vengono copiati i primi 128 sweep dalla prima
-- zona di memoria al buffer: i dati letti vengono memorizzati
-- uno alla volta nei 16 bit più significativi del registro
-- a 32 bit "data_to_proc" che viene letto dal blocco processing
-- per il calcolo di AVG_SWEEP e di STC_VECTOR
          when s0 =>
             OE_mem_n <= '0';
             Write_mem_n <= '1';
             addr_r_acq_n <= offset_acq;
             addr_r_vec_n <= offset_vectors;
             addr_w_n <= offset_buffer;
             sel_address_n <=  sel_addr_r_acq;
             data_n <= data;
             data_to_proc_n(31 downto 24) <= (others => '0') ;
             data_to_proc_n(23 downto 16) <= (others => '0') ;
             data_to_proc_n(15 downto 8) <= (others => '0') ;
             data_to_proc_n(7 downto 0) <= (others => '0') ;
             Phase_n <= fill_buffer;
             data_ready_n <= '0';
             data_n <= (others => '0');
             if start_sync_p = '1' then
               state_engine_n <= read_acq_to_buffer_i1_a;
              else
               state_engine_n <= s0;
             end if;
          when read_acq_to_buffer_i1_a =>
             OE_mem_n <= '0';
             Write_mem_n <= '1';
             state_engine_n <= wait1;
             addr_r_acq_n <= addr_r_acq_p;
             addr_r_vec_n <= addr_r_vec_p;
             addr_w_n <= addr_w_p;
             sel_address_n <= sel_addr_r_acq;
             data_n <= data;
             data_to_proc_n(31 downto 24) <= data;
             data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
             data_to_proc_n(15 downto 8) <= (others => '0') ;
             data_to_proc_n(7 downto 0) <= (others => '0') ;
             Phase_n <= fill_buffer;
             data_ready_n <= '0';
          when wait1 =>
             OE_mem_n <= '0';
             Write_mem_n <= '1';
             state_engine_n <= read_acq_to_buffer_i0_a;
             addr_r_acq_n <= addr_r_acq_p + 1;
             addr_r_vec_n <= addr_r_vec_p;
             addr_w_n <= addr_w_p;
             sel_address_n <= sel_addr_r_acq;
             data_n <= data;
             data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
```

```
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= (others => '0') ;
        data_to_proc_n(7 downto 0) <= (others => '0') ;
        Phase_n <= fill_buffer;
        data_ready_n <= '0';
    when read_acq_to_buffer_i0_a =>
        OE_mem_n <= '0';
        Write_mem_n <= '1';
        state_engine_n <= wait2;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p;
        sel_address_n <= sel_addr_r_acq;
        data_n <= data;
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data;
        data_to_proc_n(15 downto 8) <= (others => '0') ;
        data_to_proc_n(7 downto 0) <= (others => '0') ;
        Phase_n <= fill_buffer;
        data_ready_n <= '1';
    when wait2 =>
        OE_mem_n <= '1';
        Write_mem_n <= '1';
        state_engine_n <= write_acq_in_buffer_i1_a;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p;
        sel_address_n <= sel_addr_w;
        data_n <= data_to_mem(31 downto 24);
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= (others => '0') ;
        data_to_proc_n(7 downto 0) <= (others => '0') ;
        Phase_n <= fill_buffer;
        data_ready_n <= '0';
    when write_acq_in_buffer_i1_a =>
        OE_mem_n <= '1';
        Write_mem_n <= '0';
        state_engine_n <= write_acq_in_buffer_i1_b;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p;
        sel_address_n <= sel_addr_w;
        data_n <= data_to_mem(31 downto 24);
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= fill_buffer;
        data_ready_n <= '0';
    when write_acq_in_buffer_i1_b =>
        OE_mem_n <= '1';
        Write_mem_n <= '1';
        state_engine_n <= wait3;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p;
        sel_address_n <= sel_addr_w;
        data_n <= data_to_mem(31 downto 24);
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= fill_buffer;
        data_ready_n <= '0';
    when wait3 =>
        OE_mem_n <= '1';
        Write_mem_n <= '1';
        state_engine_n <= write_acq_in_buffer_i0_a;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p + 1;
        sel_address_n <= sel_addr_w;
        data_n <= data_to_mem(23 downto 16);
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= fill_buffer;
        data_ready_n <= '0';
    when write_acq_in_buffer_i0_a =>
        OE_mem_n <= '1';
        Write_mem_n <= '0';
        state_engine_n <= write_acq_in_buffer_i0_b;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p;
        sel_address_n <= sel_addr_w;
        data_n <= data_to_mem(23 downto 16);
```

```
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= fill_buffer;
        data_ready_n <= '0';
    when write_acq_in_buffer_i0_b =>
        OE_mem_n <= '1';
        Write_mem_n <= '1';
        state_engine_n <= wait4;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p;
        sel_address_n <= sel_addr_w;
        data_n <= data_to_mem(23 downto 16);
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= fill_buffer;
        data_ready_n <= '0';
    when wait4 =>
        OE_mem_n <= '1';
        Write_mem_n <= '1';
        state_engine_n <= write_acq_in_buffer_d0_a;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p + 1;
        sel_address_n <= sel_addr_w;
        data_n <= data_to_mem(15 downto 8);
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= fill_buffer;
        data_ready_n <= '0';
    when write_acq_in_buffer_d0_a =>
        OE_mem_n <= '1';
        Write_mem_n <= '0';
        state_engine_n <= write_acq_in_buffer_d0_b;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p;
        sel_address_n <= sel_addr_w;
        data_n <= data_to_mem(15 downto 8);
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= fill_buffer;
        data_ready_n <= '0';
    when write_acq_in_buffer_d0_b =>
        OE_mem_n <= '1';
        Write_mem_n <= '1';
        state_engine_n <= wait5;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p;
        sel_address_n <= sel_addr_w;
        data_n <= data_to_mem(15 downto 8);
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= fill_buffer;
        data_ready_n <= '0';
    when wait5 =>
        OE_mem_n <= '1';
        Write_mem_n <= '1';
        state_engine_n <= write_acq_in_buffer_d1_a;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p + 1;
        sel_address_n <= sel_addr_w;
        data_n <= data_to_mem(7 downto 0);
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= fill_buffer;
        data_ready_n <= '0';
    when write_acq_in_buffer_d1_a =>
        OE_mem_n <= '1';
        Write_mem_n <= '0';
        state_engine_n <= write_acq_in_buffer_d1_b;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p;
        sel_address_n <= sel_addr_w;
```

```
            data_n <= data_to_mem(7 downto 0);
            data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
            data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
            data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
            data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
            Phase_n <= fill_buffer;
            data_ready_n <= '0';
         when write_acq_in_buffer_d1_b =>
            OE_mem_n <= '1';
            Write_mem_n <= '1';
            state_engine_n <= wait6;
            addr_r_acq_n <= addr_r_acq_p;
            addr_r_vec_n <= addr_r_vec_p;
            addr_w_n <= addr_w_p;
            sel_address_n <= sel_addr_w;
            data_n <= data_to_mem(7 downto 0);
            data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
            data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
            data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
            data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
            Phase_n <= fill_buffer;
            data_ready_n <= '0';
         when wait6 =>
            OE_mem_n <= '0';
            Write_mem_n <= '1';
-- Condizione per l'individuazione dell'ultima cella di memoria della zona riservata al buffer:
-- se viene verificata è necessario procedere alla lettura degli sweep dell'acquisizione
            if addr_w_p = offset_vectors - 1 then
              state_engine_n <= read_buffer_i1_a;
              addr_r_acq_n <= offset_buffer;
              addr_w_n <= offset_vectors;
              Phase_n <= accumulator;
              sel_address_n <= sel_addr_r_acq;
             else
              state_engine_n <= read_acq_to_buffer_i1_a;
              addr_r_acq_n <= addr_r_acq_p + 1;
              addr_w_n <= addr_w_p + 1;
              Phase_n <= fill_buffer;
              sel_address_n <= sel_addr_r_acq;
            end if;
            addr_r_vec_n <= addr_w_p;
            data_n <= data;
            data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
            data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
            data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
            data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
            data_ready_n <= '0';
-- Inizio della lettura del buffer
         when read_buffer_i1_a =>
            OE_mem_n <= '0';
            Write_mem_n <= '1';
            state_engine_n <= wait9;
            addr_r_acq_n <= addr_r_acq_p;
            addr_r_vec_n <= addr_r_vec_p;
            addr_w_n <= addr_w_p;
            sel_address_n <= sel_addr_r_acq;
            data_n <= data;
            data_to_proc_n(31 downto 24) <= data;
            data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
            data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
            data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
            Phase_n <= accumulator;
            data_ready_n <= '0';
         when wait9 =>
            OE_mem_n <= '0';
            Write_mem_n <= '1';
            state_engine_n <= read_buffer_i0_a;
            addr_r_acq_n <= addr_r_acq_p + 1;
            addr_r_vec_n <= addr_r_vec_p;
            addr_w_n <= addr_w_p;
            sel_address_n <= sel_addr_r_acq;
            data_n <= data;
            data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
            data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
            data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
            data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
            Phase_n <= accumulator;
            data_ready_n <= '0';
         when read_buffer_i0_a =>
            OE_mem_n <= '0';
            Write_mem_n <= '1';
            state_engine_n <= wait10;
            addr_r_acq_n <= addr_r_acq_p;
            addr_r_vec_n <= addr_r_vec_p;
            addr_w_n <= addr_w_p;
            sel_address_n <= sel_addr_r_acq;
            data_n <= data;
            data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
            data_to_proc_n(23 downto 16) <= data;
            data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
```

```
          data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
          Phase_n <= accumulator;
          data_ready_n <= '0';
       when wait10 =>
          OE_mem_n <= '0';
          Write_mem_n <= '1';
          state_engine_n <= read_buffer_d0_a;
          addr_r_acq_n <= addr_r_acq_p + 1;
          addr_r_vec_n <= addr_r_vec_p;
          addr_w_n <= addr_w_p;
          sel_address_n <= sel_addr_r_acq;
          data_n <= data;
          data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
          data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
          data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
          data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
          Phase_n <= accumulator;
          data_ready_n <= '0';
       when read_buffer_d0_a =>
          OE_mem_n <= '0';
          Write_mem_n <= '1';
          state_engine_n <= wait11;
          addr_r_acq_n <= addr_r_acq_p;
          addr_r_vec_n <= addr_r_vec_p;
          addr_w_n <= addr_w_p;
          sel_address_n <= sel_addr_r_acq;
          data_n <= data;
          data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
          data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
          data_to_proc_n(15 downto 8) <= data;
          data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
          Phase_n <= accumulator;
          data_ready_n <= '0';
       when wait11 =>
          OE_mem_n <= '0';
          Write_mem_n <= '1';
          state_engine_n <= read_buffer_d1_a;
          addr_r_acq_n <= addr_r_acq_p + 1;
          addr_r_vec_n <= addr_r_vec_p;
          addr_w_n <= addr_w_p;
          sel_address_n <= sel_addr_r_acq;
          data_n <= data;
          data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
          data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
          data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
          data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
          Phase_n <= accumulator;
          data_ready_n <= '0';
       when read_buffer_d1_a =>
          OE_mem_n <= '0';
          Write_mem_n <= '1';
          state_engine_n <= wait12;
          addr_r_acq_n <= addr_r_acq_p;
          addr_r_vec_n <= addr_r_vec_p;
          addr_w_n <= addr_w_p;
          sel_address_n <= sel_addr_r_acq;
          data_n <= data;
          data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
          data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
          data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
          data_to_proc_n(7 downto 0) <= data;
          Phase_n <= accumulator;
          data_ready_n <= '1';
-- Nel prossimo stato la condizione si avvera quando viene superato l'indirizzo dell'ultimo campione
-- del penultimo sweep del buffer, ciò significa che il campione appena letto appartiene all'ultimo
-- sweep e quindi devono essere calcolati i corrispondenti coefficienti AVG e STC; per abilitare
-- il blocco "Processing" al calcolo dei due valori viene impostato il registro "Phase" dal valore
-- "accumulator" al valore "write_avg_and_stc".
       when wait12 =>
          Write_mem_n <= '1';
          if addr_r_acq_p > offset_buffer+(buffer_lenght-1)*sample_in_sweep*4-1 then
            OE_mem_n <= '1';
            addr_r_acq_n <= addr_r_acq_p + 1;
                        addr_r_vec_n <= addr_r_vec_p;
            state_engine_n <= write_avg_i1_a;
            Phase_n <= write_avg_and_stc;
            sel_address_n <= sel_addr_w;
            data_n <= data_to_mem(31 downto 24);
           else
-- Se la condizione non è rispettata continua la lettura nel buffer
            OE_mem_n <= '0';
            addr_r_acq_n <= addr_r_acq_p + 4 * sample_in_sweep - 3;
            addr_r_vec_n <= addr_r_vec_p;
            state_engine_n <= read_buffer_i1_a;
            Phase_n <= accumulator;
            sel_address_n <= sel_addr_r_acq;
            data_n <= data;
          end if;
          addr_w_n <= addr_w_p;
          data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
```

```
            data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
            data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
            data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
            data_ready_n <= '0';
-- Vengono ora scritti i valori AVG e STC calcolari dal blocco "Processing". Viene scritto prima il
-- valore AVG in quanto il calcolo è immediato, poi rimane in attesa del segnale "STC_ready" che
-- significa l'avvenuto calcolo del valore STC
        when write_avg_i1_a =>
            OE_mem_n <= '1';
            Write_mem_n <= '0';
            state_engine_n <= write_avg_i1_b;
            addr_r_acq_n <= addr_r_acq_p;
            addr_r_vec_n <= addr_r_vec_p;
            addr_w_n <= addr_w_p;
            sel_address_n <= sel_addr_w;
            data_n <= data_to_mem(31 downto 24);
            data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
            data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
            data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
            data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
            Phase_n <= write_avg_and_stc;
            data_ready_n <= '0';
        when write_avg_i1_b =>
            OE_mem_n <= '1';
            Write_mem_n <= '1';
            state_engine_n <= wait15;
            addr_r_acq_n <= addr_r_acq_p;
            addr_r_vec_n <= addr_r_vec_p;
            addr_w_n <= addr_w_p;
            sel_address_n <= sel_addr_w;
            data_n <= data_to_mem(31 downto 24);
            data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
            data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
            data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
            data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
            Phase_n <= write_avg_and_stc;
            data_ready_n <= '0';
        when wait15 =>
            OE_mem_n <= '1';
            Write_mem_n <= '1';
            state_engine_n <= write_avg_i0_a;
            addr_r_acq_n <= addr_r_acq_p;
            addr_r_vec_n <= addr_r_vec_p;
            addr_w_n <= addr_w_p + 1;
            sel_address_n <= sel_addr_w;
            data_n <= data_to_mem(23 downto 16);
            data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
            data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
            data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
            data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
            Phase_n <= write_avg_and_stc;
            data_ready_n <= '0';
        when write_avg_i0_a =>
            OE_mem_n <= '1';
            Write_mem_n <= '0';
            state_engine_n <= write_avg_i0_b;
            addr_r_acq_n <= addr_r_acq_p;
            addr_r_vec_n <= addr_r_vec_p;
            addr_w_n <= addr_w_p;
            sel_address_n <= sel_addr_w;
            data_n <= data_to_mem(23 downto 16);
            data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
            data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
            data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
            data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
            Phase_n <= write_avg_and_stc;
            data_ready_n <= '0';
        when write_avg_i0_b =>
            OE_mem_n <= '1';
            Write_mem_n <= '1';
            state_engine_n <= wait16;
            addr_r_acq_n <= addr_r_acq_p;
            addr_r_vec_n <= addr_r_vec_p;
            addr_w_n <= addr_w_p;
            sel_address_n <= sel_addr_w;
            data_n <= data_to_mem(23 downto 16);
            data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
            data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
            data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
            data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
            Phase_n <= write_avg_and_stc;
            data_ready_n <= '0';
        when wait16 =>
            OE_mem_n <= '1';
            Write_mem_n <= '1';
            state_engine_n <= write_avg_d0_a;
            addr_r_acq_n <= addr_r_acq_p;
            addr_r_vec_n <= addr_r_vec_p;
            addr_w_n <= addr_w_p + 1;
```

```
        sel_address_n <= sel_addr_w;
        data_n <= data_to_mem(15 downto 8);
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= write_avg_and_stc;
        data_ready_n <= '0';
    when write_avg_d0_a =>
        OE_mem_n <= '1';
        Write_mem_n <= '0';
        state_engine_n <= write_avg_d0_b;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p;
        sel_address_n <= sel_addr_w;
        data_n <= data_to_mem(15 downto 8);
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= write_avg_and_stc;
        data_ready_n <= '0';
    when write_avg_d0_b =>
        OE_mem_n <= '1';
        Write_mem_n <= '1';
        state_engine_n <= wait17;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p;
        sel_address_n <= sel_addr_w;
        data_n <= data_to_mem(15 downto 8);
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= write_avg_and_stc;
        data_ready_n <= '0';
    when wait17 =>
        OE_mem_n <= '1';
        Write_mem_n <= '1';
        state_engine_n <= write_avg_d1_a;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p + 1;
        sel_address_n <= sel_addr_w;
        data_n <= data_to_mem(7 downto 0);
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= write_avg_and_stc;
        data_ready_n <= '0';
    when write_avg_d1_a =>
        OE_mem_n <= '1';
        Write_mem_n <= '0';
        state_engine_n <= write_avg_d1_b;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p;
        sel_address_n <= sel_addr_w;
        data_n <= data_to_mem(7 downto 0);
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= write_avg_and_stc;
        data_ready_n <= '0';
    when write_avg_d1_b =>
        OE_mem_n <= '1';
        Write_mem_n <= '1';
        state_engine_n <= wait18;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p;
        sel_address_n <= sel_addr_w;
        data_n <= data_to_mem(7 downto 0);
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= write_avg_and_stc;
        data_ready_n <= '0';
    when wait18 =>
        OE_mem_n <= '1';
        Write_mem_n <= '1';
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p;
```

```
    sel_address_n <= sel_addr_w;
    data_n <= data_to_mem(31 downto 24);
    data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
    data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
    data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
    data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
    Phase_n <= write_avg_and_stc;
    if stc_ready = '0' then
      state_engine_n <= wait18;
      addr_w_n <= addr_w_p;
     else
      state_engine_n <= write_stc_i1_a;
      addr_w_n <= addr_w_p + 1;
 end if;
    data_ready_n <= '0';
when write_stc_i1_a =>
    OE_mem_n <= '1';
    Write_mem_n <= '0';
    state_engine_n <= write_stc_i1_b;
    addr_r_acq_n <= addr_r_acq_p;
    addr_r_vec_n <= addr_r_vec_p;
    addr_w_n <= addr_w_p;
    sel_address_n <= sel_addr_w;
    data_n <= data_to_mem(31 downto 24);
    data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
    data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
    data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
    data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
    Phase_n <= write_avg_and_stc;
    data_ready_n <= '0';
when write_stc_i1_b =>
    OE_mem_n <= '1';
    Write_mem_n <= '1';
    state_engine_n <= wait21;
    addr_r_acq_n <= addr_r_acq_p;
    addr_r_vec_n <= addr_r_vec_p;
    addr_w_n <= addr_w_p;
    sel_address_n <= sel_addr_w;
    data_n <= data_to_mem(31 downto 24);
    data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
    data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
    data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
    data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
    Phase_n <= write_avg_and_stc;
    data_ready_n <= '0';
when wait21 =>
    OE_mem_n <= '1';
    Write_mem_n <= '1';
    state_engine_n <= write_stc_i0_a;
    addr_r_acq_n <= addr_r_acq_p;
    addr_r_vec_n <= addr_r_vec_p;
    addr_w_n <= addr_w_p + 1;
    sel_address_n <= sel_addr_w;
    data_n <= data_to_mem(23 downto 16);
    data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
    data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
    data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
    data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
    Phase_n <= write_avg_and_stc;
    data_ready_n <= '0';
when write_stc_i0_a =>
    OE_mem_n <= '1';
    Write_mem_n <= '0';
    state_engine_n <= write_stc_i0_b;
    addr_r_acq_n <= addr_r_acq_p;
    addr_r_vec_n <= addr_r_vec_p;
    addr_w_n <= addr_w_p;
    sel_address_n <= sel_addr_w;
    data_n <= data_to_mem(23 downto 16);
    data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
    data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
    data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
    data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
    Phase_n <= write_avg_and_stc;
    data_ready_n <= '0';
when write_stc_i0_b =>
    OE_mem_n <= '1';
    Write_mem_n <= '1';
    state_engine_n <= wait22;
    addr_r_acq_n <= addr_r_acq_p;
    addr_r_vec_n <= addr_r_vec_p;
    addr_w_n <= addr_w_p;
    sel_address_n <= sel_addr_w;
    data_n <= data_to_mem(23 downto 16);
    data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
    data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
    data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
    data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
    Phase_n <= write_avg_and_stc;
```

```
        data_ready_n <= '0';
    when wait22 =>
        OE_mem_n <= '1';
        Write_mem_n <= '1';
        state_engine_n <= write_stc_d0_a;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p + 1;
        sel_address_n <= sel_addr_w;
        data_n <= data_to_mem(15 downto 8);
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= write_avg_and_stc;
        data_ready_n <= '0';
    when write_stc_d0_a =>
        OE_mem_n <= '1';
        Write_mem_n <= '0';
        state_engine_n <= write_stc_d0_b;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p;
        sel_address_n <= sel_addr_w;
        data_n <= data_to_mem(15 downto 8);
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= write_avg_and_stc;
        data_ready_n <= '0';
    when write_stc_d0_b =>
        OE_mem_n <= '1';
        Write_mem_n <= '1';
        state_engine_n <= wait23;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p;
        sel_address_n <= sel_addr_w;
        data_n <= data_to_mem(15 downto 8);
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= write_avg_and_stc;
        data_ready_n <= '0';
    when wait23 =>
        OE_mem_n <= '1';
        Write_mem_n <= '1';
        state_engine_n <= write_stc_d1_a;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p + 1;
        sel_address_n <= sel_addr_w;
        data_n <= data_to_mem(7 downto 0);
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= write_avg_and_stc;
        data_ready_n <= '0';
    when write_stc_d1_a =>
        OE_mem_n <= '1';
        Write_mem_n <= '0';
        state_engine_n <= write_stc_d1_b;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p;
        sel_address_n <= sel_addr_w;
        data_n <= data_to_mem(7 downto 0);
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= write_avg_and_stc;
        data_ready_n <= '0';
    when write_stc_d1_b =>
        OE_mem_n <= '1';
        Write_mem_n <= '1';
        state_engine_n <= wait24;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p;
        sel_address_n <= sel_addr_w;
        data_n <= data_to_mem(7 downto 0);
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
```

```
            Phase_n <= write_avg_and_stc;
            data_ready_n <= '0';
-- Nel prossimo stato viene effettuato il controllo dell'avvenuta scrittura di tutti gli N valori
-- AVG e STC. Se ciò è verificato inizia la fase di elaborazione della mappa, altrimenti viene
-- ripetuto il ciclo di lettura e calcolo dei successivi coefficienti AVG e STC
         when wait24 =>
            OE_mem_n <= '0';
            if addr_w_p = offset_elab_sweep - 1 then
              state_engine_n <= read_avg_i1_a;
              addr_r_acq_n <= offset_acq + buffer_lenght * sample_in_sweep * 2;
              addr_r_vec_n <= offset_vectors;
              addr_w_n <= offset_elab_sweep;
              sel_address_n <= sel_addr_r_vec;
              Phase_n <= load_avg;
             else
              state_engine_n <= read_buffer_i1_a;
              addr_r_acq_n <= addr_r_acq_p  - (4 * (buffer_lenght - 1) * sample_in_sweep);
              addr_r_vec_n <= addr_r_vec_p;
              addr_w_n <= addr_w_p + 1;
              sel_address_n <=  sel_addr_r_acq;
              Phase_n <= accumulator;
            end if;
            Write_mem_n <= '1';
            data_n <= data;
            data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
            data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
            data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
            data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
            data_ready_n <= '0';
-- La fase di elaborazione inizia rileggendo i valori AVG e STC ed il corrispondente campione
-- della mappa da elaborare
         when read_avg_i1_a =>
            OE_mem_n <= '0';
            Write_mem_n <= '1';
            state_engine_n <= wait27;
            addr_r_acq_n <= addr_r_acq_p;
            addr_r_vec_n <= addr_r_vec_p;
            addr_w_n <= addr_w_p;
            sel_address_n <= sel_addr_r_vec;
            data_n <= data;
            data_to_proc_n(31 downto 24) <= data;
            data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
            data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
            data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
            Phase_n <= load_avg;
            data_ready_n <= '0';
         when wait27 =>
            OE_mem_n <= '0';
            Write_mem_n <= '1';
            state_engine_n <= read_avg_i0_a;
            addr_r_acq_n <= addr_r_acq_p;
            addr_r_vec_n <= addr_r_vec_p + 1;
            addr_w_n <= addr_w_p;
            sel_address_n <= sel_addr_r_vec;
            data_n <= data;
            data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
            data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
            data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
            data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
            Phase_n <= load_avg;
            data_ready_n <= '0';
         when read_avg_i0_a =>
            OE_mem_n <= '0';
            Write_mem_n <= '1';
            state_engine_n <= wait28;
            addr_r_acq_n <= addr_r_acq_p;
            addr_r_vec_n <= addr_r_vec_p;
            addr_w_n <= addr_w_p;
            sel_address_n <= sel_addr_r_vec;
            data_n <= data;
            data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
            data_to_proc_n(23 downto 16) <= data;
            data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
            data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
            Phase_n <= load_avg;
            data_ready_n <= '0';
         when wait28 =>
            OE_mem_n <= '0';
            Write_mem_n <= '1';
            state_engine_n <= read_avg_d0_a;
            addr_r_acq_n <= addr_r_acq_p;
            addr_r_vec_n <= addr_r_vec_p + 1;
            addr_w_n <= addr_w_p;
            sel_address_n <= sel_addr_r_vec;
            data_n <= data;
            data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
            data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
            data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
            data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
            Phase_n <= load_avg;
```

```
    data_ready_n <= '0';
when read_avg_d0_a =>
    OE_mem_n <= '0';
    Write_mem_n <= '1';
    state_engine_n <= wait29;
    addr_r_acq_n <= addr_r_acq_p;
    addr_r_vec_n <= addr_r_vec_p;
    addr_w_n <= addr_w_p;
    sel_address_n <= sel_addr_r_vec;
    data_n <= data;
    data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
    data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
    data_to_proc_n(15 downto 8) <= data;
    data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
    Phase_n <= load_avg;
    data_ready_n <= '0';
when wait29 =>
    OE_mem_n <= '0';
    Write_mem_n <= '1';
    state_engine_n <= read_avg_d1_a;
    addr_r_acq_n <= addr_r_acq_p;
    addr_r_vec_n <= addr_r_vec_p + 1;
    addr_w_n <= addr_w_p;
    sel_address_n <= sel_addr_r_vec;
    data_n <= data;
    data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
    data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
    data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
    data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
    Phase_n <= load_avg;
    data_ready_n <= '0';
when read_avg_d1_a =>
    OE_mem_n <= '0';
    Write_mem_n <= '1';
    state_engine_n <= wait30;
    addr_r_acq_n <= addr_r_acq_p;
    addr_r_vec_n <= addr_r_vec_p;
    addr_w_n <= addr_w_p;
    sel_address_n <= sel_addr_r_vec;
    data_n <= data;
    data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
    data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
    data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
    data_to_proc_n(7 downto 0) <= data;
    Phase_n <= load_avg;
    data_ready_n <= '1';
when wait30 =>
    OE_mem_n <= '0';
    Write_mem_n <= '1';
    state_engine_n <= read_stc_i1_a;
    addr_r_acq_n <= addr_r_acq_p;
    addr_r_vec_n <= addr_r_vec_p + 1;
    addr_w_n <= addr_w_p;
    sel_address_n <= sel_addr_r_vec;
    data_n <= data;
    data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
    data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
    data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
    data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
    Phase_n <= load_stc;
    data_ready_n <= '0';
when read_stc_i1_a =>
    OE_mem_n <= '0';
    Write_mem_n <= '1';
    state_engine_n <= wait33;
    addr_r_acq_n <= addr_r_acq_p;
    addr_r_vec_n <= addr_r_vec_p;
    addr_w_n <= addr_w_p;
    sel_address_n <= sel_addr_r_vec;
    data_n <= data;
    data_to_proc_n(31 downto 24) <= data;
    data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
    data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
    data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
    Phase_n <= load_stc;
    data_ready_n <= '0';
when wait33 =>
    OE_mem_n <= '0';
    Write_mem_n <= '1';
    state_engine_n <= read_stc_i0_a;
    addr_r_acq_n <= addr_r_acq_p;
    addr_r_vec_n <= addr_r_vec_p + 1;
    addr_w_n <= addr_w_p;
    sel_address_n <= sel_addr_r_vec;
    data_n <= data;
    data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
    data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
    data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
    data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
```

```
        Phase_n <= load_stc;
        data_ready_n <= '0';
    when read_stc_i0_a =>
        OE_mem_n <= '0';
        Write_mem_n <= '1';
        state_engine_n <= wait34;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p;
        sel_address_n <= sel_addr_r_vec;
        data_n <= data;
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data;
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= load_stc;
        data_ready_n <= '0';
    when wait34 =>
        OE_mem_n <= '0';
        Write_mem_n <= '1';
        state_engine_n <= read_stc_d0_a;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p + 1;
        addr_w_n <= addr_w_p;
        sel_address_n <= sel_addr_r_vec;
        data_n <= data;
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= load_stc;
        data_ready_n <= '0';
    when read_stc_d0_a =>
        OE_mem_n <= '0';
        Write_mem_n <= '1';
        state_engine_n <= wait35;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p;
        sel_address_n <= sel_addr_r_vec;
        data_n <= data;
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data;
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= load_stc;
        data_ready_n <= '0';
    when wait35 =>
        OE_mem_n <= '0';
        Write_mem_n <= '1';
        state_engine_n <= read_stc_d1_a;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p + 1;
        addr_w_n <= addr_w_p;
        sel_address_n <= sel_addr_r_vec;
        data_n <= data;
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= load_stc;
        data_ready_n <= '0';
    when read_stc_d1_a =>
        OE_mem_n <= '0';
        Write_mem_n <= '1';
        state_engine_n <= wait36;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p;
        sel_address_n <= sel_addr_r_vec;
        data_n <= data;
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data;
        Phase_n <= load_stc;
        data_ready_n <= '1';
    when wait36 =>
        OE_mem_n <= '0';
        Write_mem_n <= '1';
        state_engine_n <= read_smp_i1_a;
        addr_r_acq_n <= addr_r_acq_p;
        if addr_r_vec_p = offset_elab_sweep - 1 then
          addr_r_vec_n <= offset_vectors;
         else
          addr_r_vec_n <= addr_r_vec_p + 1;
        end if;
        addr_w_n <= addr_w_p;
        sel_address_n <= sel_addr_r_acq;
        data_n <= data;
```

```
            data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
            data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
            data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
            data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
            Phase_n <= load_and_write_sample;
            data_ready_n <= '0';
         when read_smp_i1_a =>
            OE_mem_n <= '0';
            Write_mem_n <= '1';
            state_engine_n <= wait39;
            addr_r_acq_n <= addr_r_acq_p;
            addr_r_vec_n <= addr_r_vec_p;
            addr_w_n <= addr_w_p;
            sel_address_n <= sel_addr_r_acq;
            data_n <= data;
            data_to_proc_n(31 downto 24) <= data;
            data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
            data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
            data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
            Phase_n <= load_and_write_sample;
            data_ready_n <= '0';
         when wait39 =>
            OE_mem_n <= '0';
            Write_mem_n <= '1';
            state_engine_n <= read_smp_i0_a;
            addr_r_acq_n <= addr_r_acq_p + 1;
            addr_r_vec_n <= addr_r_vec_p;
            addr_w_n <= addr_w_p;
            sel_address_n <= sel_addr_r_acq;
            data_n <= data;
            data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
            data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
            data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
            data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
            Phase_n <= load_and_write_sample;
            data_ready_n <= '0';
         when read_smp_i0_a =>
            OE_mem_n <= '0';
            Write_mem_n <= '1';
            state_engine_n <= wait40;
            addr_r_acq_n <= addr_r_acq_p;
            addr_r_vec_n <= addr_r_vec_p;
            addr_w_n <= addr_w_p;
            sel_address_n <= sel_addr_r_acq;
            data_n <= data;
            data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
            data_to_proc_n(23 downto 16) <= data;
            data_to_proc_n(15 downto 8) <= (others => '0') ;
            data_to_proc_n(7 downto 0) <= (others => '0') ;
            Phase_n <= load_and_write_sample;
            data_ready_n <= '0';
         when wait40 =>
            OE_mem_n <= '1';
            Write_mem_n <= '1';
            state_engine_n <= write_elab_sweep_i1_a;
            addr_r_acq_n <= addr_r_acq_p + 1;
            addr_r_vec_n <= addr_r_vec_p;
            addr_w_n <= addr_w_p;
            sel_address_n <= sel_addr_w;
            data_n <= data_to_mem(31 downto 24);
            data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
            data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
            data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
            data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
            Phase_n <= load_and_write_sample;
            data_ready_n <= '0';
-- Scrittura dei campioni elaborati:
         when write_elab_sweep_i1_a =>
            OE_mem_n <= '1';
            Write_mem_n <= '0';
            state_engine_n <= write_elab_sweep_i1_b;
            addr_r_acq_n <= addr_r_acq_p;
            addr_r_vec_n <= addr_r_vec_p;
            addr_w_n <= addr_w_p;
            sel_address_n <= sel_addr_w;
            data_n <= data_to_mem(31 downto 24);
            data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
            data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
            data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
            data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
            Phase_n <= load_and_write_sample;
            data_ready_n <= '0';
         when write_elab_sweep_i1_b =>
            OE_mem_n <= '1';
            Write_mem_n <= '1';
            state_engine_n <= wait41;
            addr_r_acq_n <= addr_r_acq_p;
            addr_r_vec_n <= addr_r_vec_p;
            addr_w_n <= addr_w_p;
```

```
        sel_address_n <= sel_addr_w;
        data_n <= data_to_mem(31 downto 24);
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= load_and_write_sample;
        data_ready_n <= '0';
    when wait41 =>
        OE_mem_n <= '1';
        Write_mem_n <= '1';
        state_engine_n <= write_elab_sweep_i0_a;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p + 1;
        sel_address_n <= sel_addr_w;
        data_n <= data_to_mem(23 downto 16);
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= load_and_write_sample;
        data_ready_n <= '0';
    when write_elab_sweep_i0_a =>
        OE_mem_n <= '1';
        Write_mem_n <= '0';
        state_engine_n <= write_elab_sweep_i0_b;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p;
        sel_address_n <= sel_addr_w;
        data_n <= data_to_mem(23 downto 16);
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= load_and_write_sample;
        data_ready_n <= '0';
    when write_elab_sweep_i0_b =>
        OE_mem_n <= '1';
        Write_mem_n <= '1';
        state_engine_n <= wait42;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p;
        sel_address_n <= sel_addr_w;
        data_n <= data_to_mem(23 downto 16);
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= load_and_write_sample;
        data_ready_n <= '0';
    when wait42 =>
        OE_mem_n <= '1';
        Write_mem_n <= '1';
        state_engine_n <= write_elab_sweep_d0_a;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p + 1;
        sel_address_n <= sel_addr_w;
        data_n <= data_to_mem(15 downto 8);
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= load_and_write_sample;
        data_ready_n <= '0';
    when write_elab_sweep_d0_a =>
        OE_mem_n <= '1';
        Write_mem_n <= '0';
        state_engine_n <= write_elab_sweep_d0_b;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
        addr_w_n <= addr_w_p;
        sel_address_n <= sel_addr_w;
        data_n <= data_to_mem(15 downto 8);
        data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
        data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
        data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
        data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
        Phase_n <= load_and_write_sample;
        data_ready_n <= '0';
    when write_elab_sweep_d0_b =>
        OE_mem_n <= '1';
        Write_mem_n <= '1';
        state_engine_n <= wait43;
        addr_r_acq_n <= addr_r_acq_p;
        addr_r_vec_n <= addr_r_vec_p;
```

```
      addr_w_n <= addr_w_p;
      sel_address_n <= sel_addr_w;
      data_n <= data_to_mem(15 downto 8);
      data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
      data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
      data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
      data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
      Phase_n <= load_and_write_sample;
      data_ready_n <= '0';
   when wait43 =>
      OE_mem_n <= '1';
      Write_mem_n <= '1';
      state_engine_n <= write_elab_sweep_d1_a;
      addr_r_acq_n <= addr_r_acq_p;
      addr_r_vec_n <= addr_r_vec_p;
      addr_w_n <= addr_w_p + 1;
      sel_address_n <= sel_addr_w;
      data_n <= data_to_mem(7 downto 0);
      data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
      data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
      data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
      data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
      Phase_n <= load_and_write_sample;
      data_ready_n <= '0';
   when write_elab_sweep_d1_a =>
      OE_mem_n <= '1';
      Write_mem_n <= '0';
      state_engine_n <= write_elab_sweep_d1_b;
      addr_r_acq_n <= addr_r_acq_p;
      addr_r_vec_n <= addr_r_vec_p;
      addr_w_n <= addr_w_p;
      sel_address_n <= sel_addr_w;
      data_n <= data_to_mem(7 downto 0);
      data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
      data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
      data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
      data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
      Phase_n <= load_and_write_sample;
      data_ready_n <= '0';
   when write_elab_sweep_d1_b =>
      OE_mem_n <= '1';
      Write_mem_n <= '1';
      state_engine_n <= wait44;
      addr_r_acq_n <= addr_r_acq_p;
      addr_r_vec_n <= addr_r_vec_p;
      addr_w_n <= addr_w_p;
      sel_address_n <= sel_addr_w;
      data_n <= data_to_mem(7 downto 0);
      data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
      data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
      data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
      data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
      Phase_n <= load_and_write_sample;
      data_ready_n <= '0';
   when wait44 =>
      Write_mem_n <= '1';
      if addr_r_acq_p = (num_sweep_in_acq+buffer_lenght)*sample_in_sweep*2 then
        OE_mem_n <= '1';
        state_engine_n <= s0;
        sel_address_n <= sel_addr_w;
        Phase_n <= load_and_write_sample;
       else
        OE_mem_n <= '0';
        state_engine_n <= read_avg_i1_a;
        sel_address_n <= sel_addr_r_vec;
        Phase_n <= load_avg;
      end if;
      addr_r_acq_n <= addr_r_acq_p;
      addr_r_vec_n <= addr_r_vec_p;
      addr_w_n <= addr_w_p + 1;
      data_n <= data;
      data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
      data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
      data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
      data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
      data_ready_n <= '1';
   when others =>
      OE_mem_n <= '0';
      Write_mem_n <= '1';
      addr_r_acq_n <= offset_acq;
      addr_r_vec_n <= offset_vectors;
      addr_w_n <= offset_buffer;
      sel_address_n <=  sel_addr_r_acq;
      data_n <= data;
      data_to_proc_n(31 downto 24) <= (others => '0') ;
      data_to_proc_n(23 downto 16) <= (others => '0') ;
      data_to_proc_n(15 downto 8) <= (others => '0') ;
      data_to_proc_n(7 downto 0) <= (others => '0') ;
      Phase_n <= load_and_write_sample;
```

```
            data_ready_n <= '0';
            data_n <= (others => '0');
            state_engine_n <= s0;
        end case;
      end if;
end process;
process (addr_r_vec_p, addr_r_acq_p, addr_w_p, sel_address_p)
begin
  case sel_address_p is
    when sel_addr_r_acq =>
      address <= addr_r_acq_p;
    when sel_addr_w =>
      address <= addr_w_p;
    when sel_addr_r_vec =>
      address <= addr_r_vec_p;
    when others =>
      address <= addr_r_acq_p;
  end case;
end process;
-- Collego gli ingressi e le uscite dei registri alle uscite del blocco Engine
reset_sync_n <= reset;
start_sync_n <= start;
reset_sync <= reset_sync_p;
OE_mem <= OE_mem_p;
Write_mem <= Write_mem_p;
Data_to_proc <= data_to_proc_p;
Phase <= Phase_p;
Data_ready <= Data_ready_p;
data <= data_p when OE_mem_p = '1' else "ZZZZZZZZ";
end Behav;
```

A.2.2 VHDL Blocco Processing

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity processing is
port(
  Clock: in std_logic;
  reset_sync: in std_logic;
  Phase: in std_logic_vector(2 downto 0);
  data_ready: in std_logic;
  Data_to_proc: in std_logic_vector(31 downto 0);
  Data_to_mem: out std_logic_vector(31 downto 0);
  En_bgr: in std_logic;
  En_stc: in std_logic;
  STC_ready: out std_logic );
end;
architecture behav of processing is
-- Codifica del segnale Phase come nel blocco Engine:
constant fill_buffer:             std_logic_vector(2 downto 0) := "000";
constant accumulator:             std_logic_vector(2 downto 0) := "001";
constant write_avg_and_stc:       std_logic_vector(2 downto 0) := "010";
constant load_avg:                std_logic_vector(2 downto 0) := "011";
constant load_stc:                std_logic_vector(2 downto 0) := "100";
constant load_and_write_sample:   std_logic_vector(2 downto 0) := "101";
-- La costante sigma viene utilizzata nel calcolo del vettore STC.
constant sigma: std_logic_vector(63 downto 0) := x"0000800000000000";
type state_processing_type is (
  s0, s_accumulator,
  s_calc_pwr_rad0,
  s_calc_pwr_rad1,  s_calc_pwr_rad2,  s_calc_pwr_rad3,  s_calc_pwr_rad4,
  s_calc_pwr_rad5,  s_calc_pwr_rad6,  s_calc_pwr_rad7,  s_calc_pwr_rad8,
  s_calc_pwr_rad9,  s_calc_pwr_rad10, s_calc_pwr_rad11, s_calc_pwr_rad12,
  s_calc_pwr_rad13, s_calc_pwr_rad14, s_calc_pwr_rad15, s_calc_pwr_rad16,
  s_calc_pwr_rad17, s_calc_pwr_rad18, s_calc_pwr_rad19, s_calc_pwr_rad20,
  s_calc_pwr_rad21, s_calc_pwr_rad22, s_calc_pwr_rad23, s_calc_pwr_rad24,
  s_calc_pwr_rad25, s_calc_pwr_rad26, s_calc_pwr_rad27, s_calc_pwr_rad28,
  s_calc_pwr_rad29, s_calc_pwr_rad30, s_calc_pwr_rad31, s_calc_pwr_rad32,
  s_calc_stc_vec0,
  s_calc_stc_vec1,  s_calc_stc_vec2,  s_calc_stc_vec3,  s_calc_stc_vec4,
  s_calc_stc_vec5,  s_calc_stc_vec6,  s_calc_stc_vec7,  s_calc_stc_vec8,
  s_calc_stc_vec9,  s_calc_stc_vec10, s_calc_stc_vec11, s_calc_stc_vec12,
  s_calc_stc_vec13, s_calc_stc_vec14, s_calc_stc_vec15, s_calc_stc_vec16,
  s_calc_stc_vec17, s_calc_stc_vec18, s_calc_stc_vec19, s_calc_stc_vec20,
  s_calc_stc_vec21, s_calc_stc_vec22, s_calc_stc_vec23, s_calc_stc_vec24,
  s_calc_stc_vec25, s_calc_stc_vec26, s_calc_stc_vec27, s_calc_stc_vec28,
  s_calc_stc_vec29, s_calc_stc_vec30, s_calc_stc_vec31, s_calc_stc_vec32,
  s_wait_write_stc,
  s_load_avg, s_load_stc, s_load_and_write_smp);
signal state_processing_p, state_processing_n : state_processing_type;
-- Definizione dei registri:
-- Per il processo di calcolo:
signal avg_acc_n, avg_acc_p : std_logic_vector(39 downto 0);
signal en_sub_n, en_sub_p : std_logic;
signal pwr_rad_n, pwr_rad_p : std_logic_vector(31 downto 0);
signal stc_vec_n, stc_vec_p : std_logic_vector(31 downto 0);
signal pwr_acc_n, pwr_acc_p : std_logic_vector(71 downto 0);
type sel_comp_type is (c_pwracc, c_sigma);
signal sel_comp_n, sel_comp_p : sel_comp_type;
type sel_molt_type is (m_accumulator, m_root, m_stcvec, m_elab);
signal sel_molt_n, sel_molt_p : sel_molt_type;
-- Per il processo di uscita:
type sel_dtm_type is (d_fillbuff, d_writeavg, d_writestc, d_elab);
signal sel_dtm_n, sel_dtm_p : sel_dtm_type;
signal stc_ready_n, stc_ready_p : std_logic;
-- Definizione dei signal di uscita del processo di elaborazione e di ingresso del processo di uscita
signal avg_vec : std_logic_vector(31 downto 0);
signal elab_sample : std_logic_vector(63 downto 0);
signal product : std_logic_vector(63 downto 0);
signal comp : std_logic;
-- Definizione delle funzioni per il calcolo dei vettori
function root (
  i: integer;
  comp: std_logic;
  pwr_rad_p : std_logic_vector(31 downto 0))
  return std_logic_vector
  is begin
    if comp = '1' then
      return pwr_rad_p(31 downto i + 1)&"01"&pwr_rad_p(i-2 downto 0);
     else
      return pwr_rad_p(31 downto i + 1)&"11"&pwr_rad_p(i-2 downto 0);
    end if;
end root;
function division (
  j: integer;
```

```
  comp: std_logic;
  stc_vec_p : std_logic_vector(31 downto 0))
  return std_logic_vector
  is begin
    if  comp = '1' then
      return stc_vec_p(31 downto j + 1)&"01"&stc_vec_p(j-2 downto 0);
     else
      return stc_vec_p(31 downto j + 1)&"11"&stc_vec_p(j-2 downto 0);
    end if;
end division;
begin
-- Processo sequenziale
process (clock) begin
  if (clock'event and clock = '1') then
    state_processing_p <= state_processing_n;
    avg_acc_p <= avg_acc_n;
    pwr_acc_p <= pwr_acc_n;
    pwr_rad_p <= pwr_rad_n;
    stc_vec_p <= stc_vec_n;
    sel_comp_p <= sel_comp_n;
    sel_molt_p <= sel_molt_n;
    sel_dtm_p <= sel_dtm_n;
    stc_ready_p <= stc_ready_n;
    en_sub_p <= en_sub_n;
  end if;
end process;
process (
  reset_sync, phase,  data_ready, data_to_proc,
  state_processing_p,
  avg_acc_p, pwr_acc_p, pwr_rad_p, stc_vec_p,
  comp, product, En_bgr, En_stc)
variable data_to_proc_for_avg: std_logic_vector(39 downto 0);
begin
-- Condizionamento di data_to_proc per il calcolo di AVG_ACC su 40 bit (32+8)
-- (espansione complemento a 2)
data_to_proc_for_avg(39 downto 32) := (others => data_to_proc(31));
data_to_proc_for_avg(31 downto 0) := data_to_proc;
if (reset_sync = '0') then
  avg_acc_n <= (others => '0');
  pwr_acc_n <= (others => '0');
  pwr_rad_n <= (others => '0');
  stc_vec_n <= (others => '0');
  en_sub_n <= '0';
  sel_comp_n <= c_pwracc;
  sel_molt_n <= m_accumulator;
  sel_dtm_n <= d_writestc;
  stc_ready_n <= '0';
  state_processing_n <= s0;
 else
  case state_processing_p is
-- Riempimento del buffer
    when s0 =>
      avg_acc_n <= (others => '0');
      pwr_acc_n <= (others => '0');
      pwr_rad_n <= (others => '0');
      stc_vec_n <= (others => '0');
      en_sub_n <= '0';
      sel_comp_n <= c_pwracc;
      sel_molt_n <= m_accumulator;
      sel_dtm_n <= d_fillbuff;
      stc_ready_n <= '0';
      if phase = fill_buffer then
        state_processing_n <= s0;
       else
        state_processing_n <= s_accumulator;
      end if;
-- In questo stato avviene il calcolo della 4.1 e della 4.2
    when s_accumulator =>
      if data_ready = '0' then
        avg_acc_n <= avg_acc_p;
        pwr_acc_n <= pwr_acc_p;
       else
        avg_acc_n <= data_to_proc_for_avg + avg_acc_p;
        pwr_acc_n <= product + pwr_acc_p;
      end if;
      pwr_rad_n <= pwr_rad_p;
      stc_vec_n <= stc_vec_p;
      en_sub_n <= '0';
      sel_comp_n <= c_pwracc;
      sel_molt_n <= m_accumulator;
      sel_dtm_n <= d_writeavg;
      stc_ready_n <= '0';
      if phase = accumulator then
        state_processing_n <= s_accumulator;
       else
        state_processing_n <= s_calc_pwr_rad32;
      end if;
-- Calcolo della radice presente nella 3.4 con il metodo della bisezione
    when s_calc_pwr_rad32 =>
      avg_acc_n <= avg_acc_p;
```

```
    pwr_acc_n <= pwr_acc_p;
    pwr_rad_n <= '1'&pwr_rad_p(30 downto 0);
    stc_vec_n <= stc_vec_p;
    en_sub_n <= '0';
    sel_comp_n <= c_pwracc;
    sel_molt_n <= m_root;
    sel_dtm_n <= d_writeavg;
    stc_ready_n <= '0';
    state_processing_n <= s_calc_pwr_rad31;
  when s_calc_pwr_rad31 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
    if comp = '1' then
      pwr_rad_n <= "01"&pwr_rad_p(29 downto 0);
     else
      pwr_rad_n <= "11"&pwr_rad_p(29 downto 0);
    end if;
    stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
    sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_pwr_rad30;
  when s_calc_pwr_rad30 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
    if  comp = '1' then
      pwr_rad_n <= pwr_rad_p(31)&"01"&pwr_rad_p(28 downto 0);
     else
      pwr_rad_n <= pwr_rad_p(31)&"11"&pwr_rad_p(28 downto 0);
    end if;
    stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
    sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_pwr_rad29;
  when s_calc_pwr_rad29 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
    pwr_rad_n <= root(29, comp, pwr_rad_p);
    stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
    sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_pwr_rad28;
  when s_calc_pwr_rad28 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
    pwr_rad_n <= root(28, comp, pwr_rad_p);
    stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
    sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_pwr_rad27;
  when s_calc_pwr_rad27 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
    pwr_rad_n <= root(27, comp, pwr_rad_p);
    stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
    sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_pwr_rad26;
  when s_calc_pwr_rad26 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
    pwr_rad_n <= root(26, comp, pwr_rad_p);
    stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
    sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_pwr_rad25;
  when s_calc_pwr_rad25 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
    pwr_rad_n <= root(25, comp, pwr_rad_p);
    stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
    sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_pwr_rad24;
  when s_calc_pwr_rad24 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
    pwr_rad_n <= root(24, comp, pwr_rad_p);
    stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
    sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_pwr_rad23;
  when s_calc_pwr_rad23 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
    pwr_rad_n <= root(23, comp, pwr_rad_p);
    stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
    sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_pwr_rad22;
  when s_calc_pwr_rad22 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
    pwr_rad_n <= root(22, comp, pwr_rad_p);
    stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
    sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_pwr_rad21;
  when s_calc_pwr_rad21 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
    pwr_rad_n <= root(21, comp, pwr_rad_p);
    stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
    sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_pwr_rad20;
  when s_calc_pwr_rad20 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
    pwr_rad_n <= root(20, comp, pwr_rad_p);
    stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
    sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_pwr_rad19;
  when s_calc_pwr_rad19 =>
```

```
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
    pwr_rad_n <= root(19, comp, pwr_rad_p);
    stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
    sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_pwr_rad18;
  when s_calc_pwr_rad18 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
    pwr_rad_n <= root(18, comp, pwr_rad_p);
    stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
    sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_pwr_rad17;
  when s_calc_pwr_rad17 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
    pwr_rad_n <= root(17, comp, pwr_rad_p);
    stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
    sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_pwr_rad16;
  when s_calc_pwr_rad16 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
    pwr_rad_n <= root(16, comp, pwr_rad_p);
    stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
    sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_pwr_rad15;
  when s_calc_pwr_rad15 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
    pwr_rad_n <= root(15, comp, pwr_rad_p);
    stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
    sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_pwr_rad14;
  when s_calc_pwr_rad14 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
    pwr_rad_n <= root(14, comp, pwr_rad_p);
    stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
    sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_pwr_rad13;
  when s_calc_pwr_rad13 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
    pwr_rad_n <= root(13, comp, pwr_rad_p);
    stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
    sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_pwr_rad12;
  when s_calc_pwr_rad12 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
    pwr_rad_n <= root(12, comp, pwr_rad_p);
    stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
    sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_pwr_rad11;
  when s_calc_pwr_rad11 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
    pwr_rad_n <= root(11, comp, pwr_rad_p);
    stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
    sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_pwr_rad10;
  when s_calc_pwr_rad10 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
    pwr_rad_n <= root(10, comp, pwr_rad_p);
    stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
    sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_pwr_rad9;
  when s_calc_pwr_rad9 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
    pwr_rad_n <= root(9, comp, pwr_rad_p);
    stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
    sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_pwr_rad8;
  when s_calc_pwr_rad8 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
    pwr_rad_n <= root(8, comp, pwr_rad_p);
    stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
    sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_pwr_rad7;
  when s_calc_pwr_rad7 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
    pwr_rad_n <= root(7, comp, pwr_rad_p);
    stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
    sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_pwr_rad6;
  when s_calc_pwr_rad6 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
    pwr_rad_n <= root(6, comp, pwr_rad_p);
    stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
    sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_pwr_rad5;
  when s_calc_pwr_rad5 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
    pwr_rad_n <= root(5, comp, pwr_rad_p);
    stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
    sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_pwr_rad4;
  when s_calc_pwr_rad4 =>
```

```
      avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
      pwr_rad_n <= root(4, comp, pwr_rad_p);
      stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
      sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
      state_processing_n <= s_calc_pwr_rad3;
    when s_calc_pwr_rad3 =>
      avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
      pwr_rad_n <= root(3, comp, pwr_rad_p);
      stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
      sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
      state_processing_n <= s_calc_pwr_rad2;
    when s_calc_pwr_rad2 =>
      avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
      if comp = '1' then
        pwr_rad_n <= pwr_rad_p(31 downto 3)&"01"&pwr_rad_p(0);
       else
        pwr_rad_n <= pwr_rad_p(31 downto 3)&"11"&pwr_rad_p(0);
      end if;
      stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
      sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
      state_processing_n <= s_calc_pwr_rad1;
    when s_calc_pwr_rad1 =>
      avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
      if comp = '1' then
        pwr_rad_n <= pwr_rad_p(31 downto 2)&"01";
       else
        pwr_rad_n <= pwr_rad_p(31 downto 2)&"11";
      end if;
      stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
      sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
      state_processing_n <= s_calc_pwr_rad0;
    when s_calc_pwr_rad0 =>
      avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p;
      if comp = '1' then
        pwr_rad_n <= pwr_rad_p(31 downto 1)&'0';
       else
        pwr_rad_n <= pwr_rad_p(31 downto 1)&'1';
      end if;
      stc_vec_n <= stc_vec_p; en_sub_n <= '0'; sel_comp_n <= c_pwracc;
      sel_molt_n <= m_root; sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
      state_processing_n <= s_calc_stc_vec32;
-- Calcolo della 3.4 con il metodo della bisezione
    when s_calc_stc_vec32 =>
      avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
      stc_vec_n <= '1'&stc_vec_p(30 downto 0);
      en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
      sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
      state_processing_n <= s_calc_stc_vec31;
    when s_calc_stc_vec31 =>
      avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
      if comp = '1' then
        stc_vec_n <= "01"&stc_vec_p(29 downto 0);
       else
        stc_vec_n <= "11"&stc_vec_p(29 downto 0);
      end if;
      en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
      sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
      state_processing_n <= s_calc_stc_vec30;
    when s_calc_stc_vec30 =>
      avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
      if comp = '1' then
        stc_vec_n <= stc_vec_p(31)&"01"&stc_vec_p(28 downto 0);
       else
        stc_vec_n <= stc_vec_p(31)&"11"&stc_vec_p(28 downto 0);
      end if;
      en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
      sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
      state_processing_n <= s_calc_stc_vec29;
    when s_calc_stc_vec29 =>
      avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
      stc_vec_n <= division(29, comp, stc_vec_p);
      en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
      sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
      state_processing_n <= s_calc_stc_vec28;
    when s_calc_stc_vec28 =>
      avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
      stc_vec_n <= division(28, comp, stc_vec_p);
      en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
      sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
      state_processing_n <= s_calc_stc_vec27;
    when s_calc_stc_vec27 =>
      avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
      stc_vec_n <= division(27, comp, stc_vec_p);
      en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
      sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
      state_processing_n <= s_calc_stc_vec26;
    when s_calc_stc_vec26 =>
      avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
      stc_vec_n <= division(26, comp, stc_vec_p);
      en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
```

```
    sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_stc_vec25;
  when s_calc_stc_vec25 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
    stc_vec_n <= division(25, comp, stc_vec_p);
    en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
    sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_stc_vec24;
  when s_calc_stc_vec24 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
    stc_vec_n <= division(24, comp, stc_vec_p);
    en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
    sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_stc_vec23;
  when s_calc_stc_vec23 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
    stc_vec_n <= division(23, comp, stc_vec_p);
    en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
    sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_stc_vec22;
  when s_calc_stc_vec22 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
    stc_vec_n <= division(22, comp, stc_vec_p);
    en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
    sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_stc_vec21;
  when s_calc_stc_vec21 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
    stc_vec_n <= division(21, comp, stc_vec_p);
    en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
    sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_stc_vec20;
  when s_calc_stc_vec20 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
    stc_vec_n <= division(20, comp, stc_vec_p);
    en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
    sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_stc_vec19;
  when s_calc_stc_vec19 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
    stc_vec_n <= division(19, comp, stc_vec_p);
    en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
    sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_stc_vec18;
  when s_calc_stc_vec18 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
    stc_vec_n <= division(18, comp, stc_vec_p);
    en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
    sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_stc_vec17;
  when s_calc_stc_vec17 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
    stc_vec_n <= division(17, comp, stc_vec_p);
    en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
    sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_stc_vec16;
  when s_calc_stc_vec16 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
    stc_vec_n <= division(16, comp, stc_vec_p);
    en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
    sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_stc_vec15;
  when s_calc_stc_vec15 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
    stc_vec_n <= division(15, comp, stc_vec_p);
    en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
    sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_stc_vec14;
  when s_calc_stc_vec14 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
    stc_vec_n <= division(14, comp, stc_vec_p);
    en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
    sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_stc_vec13;
  when s_calc_stc_vec13 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
    stc_vec_n <= division(13, comp, stc_vec_p);
    en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
    sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_stc_vec12;
  when s_calc_stc_vec12 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
    stc_vec_n <= division(12, comp, stc_vec_p);
    en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
    sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_stc_vec11;
  when s_calc_stc_vec11 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
    stc_vec_n <= division(11, comp, stc_vec_p);
    en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
```

```
    sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_stc_vec10;
  when s_calc_stc_vec10 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
    stc_vec_n <= division(10, comp, stc_vec_p);
    en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
    sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_stc_vec9;
  when s_calc_stc_vec9 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
    stc_vec_n <= division(9, comp, stc_vec_p);
    en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
    sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_stc_vec8;
  when s_calc_stc_vec8 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
    stc_vec_n <= division(8, comp, stc_vec_p);
    en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
    sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_stc_vec7;
  when s_calc_stc_vec7 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
    stc_vec_n <= division(7, comp, stc_vec_p);
    en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
    sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_stc_vec6;
  when s_calc_stc_vec6 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
    stc_vec_n <= division(6, comp, stc_vec_p);
    en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
    sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_stc_vec5;
  when s_calc_stc_vec5 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
    stc_vec_n <= division(5, comp, stc_vec_p);
    en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
    sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_stc_vec4;
  when s_calc_stc_vec4 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
    stc_vec_n <= division(4, comp, stc_vec_p);
    en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
    sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_stc_vec3;
  when s_calc_stc_vec3 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
    stc_vec_n <= division(3, comp, stc_vec_p);
    en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
    sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_stc_vec2;
  when s_calc_stc_vec2 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
    if comp = '1' then
      stc_vec_n <= stc_vec_p(31 downto 3)&"01"&stc_vec_p(0);
     else
      stc_vec_n <= stc_vec_p(31 downto 3)&"11"&stc_vec_p(0);
    end if;
    en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
    sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_stc_vec1;
  when s_calc_stc_vec1 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
    if comp = '1' then
      stc_vec_n <= stc_vec_p(31 downto 2)&"01";
     else
      stc_vec_n <= stc_vec_p(31 downto 2)&"11";
    end if;
    en_sub_n <= '0'; sel_comp_n <= c_sigma; sel_molt_n <= m_stcvec;
    sel_dtm_n <= d_writeavg; stc_ready_n <= '0';
    state_processing_n <= s_calc_stc_vec0;
  when s_calc_stc_vec0 =>
    avg_acc_n <= avg_acc_p; pwr_acc_n <= pwr_acc_p; pwr_rad_n <= pwr_rad_p;
    if comp = '1' then
      stc_vec_n <= stc_vec_p(31 downto 1)&'0';
     else
      stc_vec_n <= stc_vec_p(31 downto 1)&'1';
    end if;
    en_sub_n <= '1';
    sel_comp_n <= c_sigma;
    sel_molt_n <= m_stcvec;
    sel_dtm_n <= d_writestc;
    stc_ready_n <= '1';
    state_processing_n <= s_wait_write_STC;
  when s_wait_write_STC =>
    sel_comp_n <= c_sigma;
    sel_molt_n <= m_stcvec;
    sel_dtm_n <= d_writestc;
    stc_ready_n <= '0';
    en_sub_n <= '0';
    case phase is
```

```
            when accumulator =>
              state_processing_n <= s_accumulator;
              avg_acc_n <= (others => '0');
              pwr_acc_n <= (others => '0');
              pwr_rad_n <= (others => '0');
              stc_vec_n <= (others => '0');
            when write_avg_and_STC =>
              state_processing_n <= s_wait_write_STC;
              avg_acc_n <= avg_acc_p;
              pwr_acc_n <= pwr_acc_p;
              pwr_rad_n <= pwr_rad_p;
              stc_vec_n <= stc_vec_p;
            when load_avg =>
              state_processing_n <= s_load_avg;
              avg_acc_n <= avg_acc_p;
              pwr_acc_n <= pwr_acc_p;
              pwr_rad_n <= pwr_rad_p;
              stc_vec_n <= stc_vec_p;
            when others =>
              state_processing_n <= s0;
              avg_acc_n <= avg_acc_p;
              pwr_acc_n <= pwr_acc_p;
              pwr_rad_n <= pwr_rad_p;
              stc_vec_n <= stc_vec_p;
          end case;
-- Terminata la fase di calcolo dei vettori AVG e PWR inizia la fase di elaborazione:
-- viene configurata la rete viene configurata affinché una volta calcolati i valori
-- AVG, PWR e il campione in esame, si presenti in uscita il valore del campione elaborato
        when s_load_avg =>
          if data_ready = '0' then
            avg_acc_n <= avg_acc_p;
            state_processing_n <= s_load_avg;
           else
            avg_acc_n(39) <= '0';
            if En_bgr = '1' then
              avg_acc_n(38 downto 7) <= data_to_proc;
             else
              avg_acc_n(38 downto 7) <= (others => '0');
            end if;
            avg_acc_n(6 downto 0) <= (others => '0');
            state_processing_n <= s_load_STC;
          end if;
          pwr_acc_n <= pwr_acc_p;
          pwr_rad_n <= pwr_rad_p;
          stc_vec_n <= stc_vec_p;
          en_sub_n <= '1';
          sel_comp_n <= c_sigma;
          sel_molt_n <= m_elab;
          sel_dtm_n <= d_elab;
          stc_ready_n <= '0';
        when s_load_STC =>
          avg_acc_n <= avg_acc_p;
          pwr_acc_n <= pwr_acc_p;
          pwr_rad_n <= pwr_rad_p;
          if data_ready = '0' then
            stc_vec_n <= stc_vec_p;
            state_processing_n <= s_load_STC;
           else
            if En_stc = '1' then
              stc_vec_n <= data_to_proc;
             else
              stc_vec_n <= x"00040000";
            end if;
            state_processing_n <= s_load_and_write_smp;
          end if;
          en_sub_n <= '1';
          sel_comp_n <= c_sigma;
          sel_molt_n <= m_elab;
          sel_dtm_n <= d_elab;
          stc_ready_n <= '0';
        when s_load_and_write_smp =>
          avg_acc_n <= avg_acc_p;
          pwr_acc_n <= pwr_acc_p;
          pwr_rad_n <= pwr_rad_p;
          stc_vec_n <= stc_vec_p;
          en_sub_n <= '1';
          sel_comp_n <= c_sigma;
          sel_molt_n <= m_elab;
          sel_dtm_n <= d_elab;
          stc_ready_n <= '1';
          if data_ready = '0' then
            state_processing_n <= s_load_and_write_smp;
           else
            state_processing_n <= s_load_avg;
          end if;
        when others =>
          avg_acc_n <= avg_acc_p;
          pwr_acc_n <= pwr_acc_p;
          pwr_rad_n <= pwr_rad_p;
          stc_vec_n <= stc_vec_p;
          en_sub_n <= '0';
          sel_comp_n <= c_pwracc;
          sel_molt_n <= m_elab;
          sel_dtm_n <= d_elab;
```

```
      stc_ready_n <= '0';
      state_processing_n <= s0;
  end case;
end if;
end process;
-- I seguenti due processi descrivono la rete in figura 4.11:
process (
  data_to_proc,
  avg_acc_p, en_sub_p, pwr_rad_p, stc_vec_p, pwr_acc_p,
  sel_comp_p, sel_molt_p)
variable data_to_proc_m_avg : std_logic_vector(31 downto 0);
variable mod_data_to_proc_m_avg : std_logic_vector(31 downto 0);
variable neg_data : std_logic;
variable factor_1 : std_logic_vector(31 downto 0);
variable factor_2 : std_logic_vector(31 downto 0);
variable comp_2 : std_logic_vector(63 downto 0);
variable product_temp : std_logic_vector(63 downto 0);
variable avg_vec_temp : std_logic_vector(31 downto 0);
variable avg_pwr_acc : std_logic_vector(63 downto 0);
begin
  avg_vec_temp := avg_acc_p(38 downto 7);
  avg_pwr_acc := pwr_acc_p(70 downto 7);
  if en_sub_p = '0' then
    data_to_proc_m_avg := data_to_proc;
   else
    data_to_proc_m_avg := data_to_proc - avg_vec_temp;
  end if;
  if data_to_proc_m_avg(31) = '0' then
    mod_data_to_proc_m_avg := data_to_proc_m_avg;
    neg_data := '0';
   else
    mod_data_to_proc_m_avg := not(data_to_proc_m_avg) + '1';
    neg_data := '1';
  end if;
  case sel_molt_p is
    when m_accumulator =>
      factor_1 := mod_data_to_proc_m_avg;
      factor_2 := mod_data_to_proc_m_avg;
    when m_root =>
      factor_1 := pwr_rad_p;
      factor_2 := pwr_rad_p;
    when m_stcvec =>
      factor_1 := stc_vec_p;
      factor_2 := pwr_rad_p + x"10000";
    when m_elab =>
      factor_1 := mod_data_to_proc_m_avg;
      factor_2 := stc_vec_p;
    when others =>
      factor_1 := mod_data_to_proc_m_avg;
      factor_2 := stc_vec_p;
  end case;
  case sel_comp_p is
    when c_pwracc =>
      comp_2 := avg_pwr_acc;
    when c_sigma =>
      comp_2 := sigma;
    when others =>
      comp_2 := sigma;
  end case;
  product_temp := factor_1 * factor_2;
  if product_temp > comp_2 then
    comp <= '1';
   else
    comp <= '0';
  end if;
  if neg_data = '0' then
    elab_sample <= product_temp;
   else
    elab_sample <= not(product_temp) + '1';
  end if;
  product <= product_temp;
  avg_vec <= avg_vec_temp;
end process;
process (data_to_proc, avg_vec, stc_vec_p, elab_sample, sel_dtm_p)
begin
  case sel_dtm_p is
    when d_fillbuff =>
      data_to_mem <= data_to_proc;
    when d_writeavg =>
      data_to_mem <= avg_vec;
    when d_writestc =>
      data_to_mem <= stc_vec_p;
    when d_elab =>
      data_to_mem <= elab_sample(63 downto 32);
    when others =>
      data_to_mem <= data_to_proc;
  end case;
end process;
stc_ready <= stc_ready_p;
end behav;
```

A.2.3 VHDL Testbench

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
USE IEEE.STD_LOGIC_TEXTIO.ALL;
USE STD.TEXTIO.ALL;
library xp2;
use xp2.components.all;
ENTITY testbench IS
generic( buffer_lenght : integer := 128;
         sample_in_sweep : integer := 512;
         byte_per_sample : integer := 2;
         num_sweep_in_acq :   integer:= 512);
END testbench;
 Architecture behavior of testbench is
  signal Data : std_logic_vector (7 downto 0);
  signal Write_mem : std_logic;
  signal OE_mem : std_logic;
  signal Address : std_logic_vector (23 downto 0);
  signal Start : std_logic;
  signal Reset : std_logic;
  signal En_bgr : std_logic;
  signal En_stc : std_logic;
  signal Clock : std_logic := '0';
  component SYSTEM
   Port(
    Data : InOut std_logic_vector (7 downto 0);
    Write_mem : Out std_logic;
    OE_mem : Out std_logic;
    Address : Out std_logic_vector (23 downto 0);
    Start : In std_logic;
    Reset : In std_logic;
    En_bgr : In std_logic;
    En_stc : In std_logic;
    Clock : In std_logic);
  end component;
-- Segnali per abilitare la scrittura su file dei vettori AVG, STC e della mappa elaborata
  signal write_in_file_avg: std_logic;
  signal write_in_file_stc: std_logic;
  signal write_in_file_map: std_logic;
-- Definizione del tipo RAM
  type ram_type is array (0 to 2**24-1) of std_logic_vector(7 downto 0);
-- Dichiarazione dei tipi file per lettura e scrittura dei dati:
-- File contenente la mappa da elaborare
  file preelabmap: text;
-- File per il salvataggio del vettore AVG
  file avg_sweep: text;
-- File per il salvataggio del vettore AVG in formato grafico
  file avg_sweep_bmp: text;
-- File per il salvataggio del vettore STC
  file pow_vec: text;
-- File per il salvataggio della mappa elaborata
  file elabmap: text;
-- File per il salvataggio della mappa elaborata in formato grafico
  file elabmap_bmp: text;
-- Definizione del tipo array per la scrittura su file della mappa elaborata
  type acq_type is array (0 to num_sweep_in_acq * sample_in_sweep - 1) of std_logic_vector(31 downto 0);
-- Costanti per la mappatura della memoria
  constant offset_acq : std_logic_vector(23 downto 0) := X"000000";
  constant offset_buffer : std_logic_vector(23 downto 0) := X"760000";
  constant offset_vectors : std_logic_vector(23 downto 0) := X"7A0000";
  constant offset_elab_sweep : std_logic_vector(23 downto 0) := X"7A1000";
-- Definizione della funzione "visualizza" per la conversione in formato grafico della
-- mappa elaborata. La funzione riceve in ingresso il campione da rappresentare in
-- formato grafico su scala di grigio e il parametro "sigma" che rappresenta il contrasto
-- dell'immagine. L'aumento del contrasto è ottenuto effettuando lo shift a sinistra di
-- "sigma" posizioni del campione da visualizzare, considerando anche i casi in cui
-- avviene la saturazione, positiva o negativa.
-- Quindi per ogni campione viene verificato se l'operazione di shift provoca una
-- saturazione; in caso negativo viene effetuato lo shift e viene letto il byte più
-- significativo. Essendo il byte rappresentato in complemento a 2, affinché questo
-- valore rappresenti un livello di grigio in una scala in cui il valore x"00" sia il nero
-- e x"FF" il bianco, viene considerato solo il byte più significativo a cui viene
-- sommato il valore x"80"
  function visualizza (
   sigma: integer;
   elab_sample: std_logic_vector(31 downto 0))
   return std_logic_vector is
    variable saturato_neg : boolean := FALSE;
    variable saturato_pos : boolean := FALSE;
    variable elab_sample_temp : std_logic_vector(31 downto 0);
    variable bmp_temp : std_logic_vector(7 downto 0);
    constant all1 : std_logic_vector(31 downto 0) := (others => '1');
    constant all0 : std_logic_vector(31 downto 0) := (others => '0');
    begin
-- Verifica delle saturazioni
```

```
      elab_sample_temp(31 downto 31-sigma) := elab_sample(31 downto 31-sigma);
      if (elab_sample(31) = '1') then
        elab_sample_temp(31-sigma-1 downto 0) := (others => '1');
        if elab_sample_temp = all1 then saturato_neg := FALSE; else saturato_neg := TRUE; end if;
       else
        elab_sample_temp(31-sigma-1 downto 0) := (others => '0');
        if elab_sample_temp = all0 then saturato_pos := FALSE; else saturato_pos := TRUE; end if;
      end if;
      if (saturato_neg = FALSE AND saturato_pos = FALSE)  then
-- Scorrimento a sinistra
        elab_sample_temp(31 downto sigma) := elab_sample(31 - sigma downto 0);
        elab_sample_temp(sigma - 1 downto 0) := (others => '0');
        bmp_temp := elab_sample_temp(31 downto 24) + x"80";
       elsif saturato_pos = TRUE then
         bmp_temp := x"FF";
        else
         bmp_temp := x"00";
      end if;
  return bmp_temp;
 end visualizza;
BEGIN
UUT : SYSTEM
  Port Map( Address=>Address, Clock=>Clock, Data=>Data,
            OE_mem=>OE_mem, Reset=>Reset, Start=>Start,
            En_bgr => En_bgr, En_stc => En_stc,
            Write_mem=>Write_mem);
-- Processo che simula la RAM e gestisce le operazioni di lettura e scrittura su file
process (write_mem, OE_mem, data, write_in_file_avg, write_in_file_stc, write_in_file_map, address)
  Variable tmp_ram : ram_type;
  Variable in_line : LINE;
  Variable out_line : LINE;
  Variable sample : std_logic_vector(15 downto 0);
  Variable acq_elab : acq_type;
  Variable first_process : boolean := TRUE;
  Variable bmp : std_logic_vector(7 downto 0);
  begin
-- La prima volta che viene eseguito il process viene copiato il file "acq_preelab.txt"
-- in memoria; il file "acq_preelab.txt" è un file di testo così composto:
-- - ogni riga contiene i campioni di uno sweep separati da uno spazio;
--   La prima riga rappresenta il primo sweep della mappa, il primo campione è il primo
--   dello sweep
-- - ogni campione è rappresentato con 16 bit in formato esadecimale complemento a 2
    if first_process = TRUE then
      file_open(preelabmap, "acq_preelab.txt", READ_MODE);
      for i in 0 to (num_sweep_in_acq + buffer_lenght - 1) loop
-- lettura di un intero sweep dal file (una riga)
        READLINE(preelabmap, in_line);
        for j in 0 to (sample_in_sweep - 1) loop
-- lettura di un sample dalla riga
          HREAD(in_line, sample);
-- scrittura in memoria
          tmp_ram((i * byte_per_sample * sample_in_sweep) +
                          (j * byte_per_sample)) := sample(15 downto 8);
          tmp_ram((i * byte_per_sample * sample_in_sweep) +
                          (j * byte_per_sample) + 1) := sample(7 downto 0);
        end loop;
      end loop;
      file_close(preelabmap);
      first_process := FALSE;
    end if;
-- Simulazione della RAM:
-- Condizione di scrittura
    if write_mem ='0' then
      tmp_ram(conv_integer(Address)) := Data;
    end if;
-- Condizione di lettura
    if OE_mem='0' then
      Data <= tmp_ram(conv_integer(Address));
     else
-- Se Output_enable = 0 il bus viene posto in alta impedenza
      Data <= (others => 'Z');
    end if;
-- Scrittura su file del vettore AVG, sia in formato numerico che grafico BMP
-- La scrittura avviene in seguito a un fronte sul segnale "write_in_file_avg"
    if (write_in_file_avg'event and write_in_file_avg='1') then
      file_open(avg_sweep, "avg.txt", WRITE_MODE);
      for j in 0 to (sample_in_sweep - 1) loop
        hwrite(out_line, tmp_ram((8 * j) + conv_integer(offset_vectors)));
        hwrite(out_line, tmp_ram((8 * j) + conv_integer(offset_vectors) + 1));
        hwrite(out_line, tmp_ram((8 * j) + conv_integer(offset_vectors) + 2));
        hwrite(out_line, tmp_ram((8 * j) + conv_integer(offset_vectors) + 3));
        writeline(avg_sweep, out_line);
      end loop;
      file_close(avg_sweep);
          file_open(avg_sweep_bmp, "avg_bmp.txt", WRITE_MODE);
      for j in 0 to (sample_in_sweep - 1) loop
        bmp := tmp_ram((8 * j) + conv_integer(offset_vectors)) + x"80";
        hwrite(out_line, bmp);
```

```
        write(out_line, string'(" "));
        hwrite(out_line, bmp);
        write(out_line, string'(" "));
        hwrite(out_line, bmp);
        write(out_line, string'(" "));
      end loop;
      writeline(avg_sweep_bmp, out_line);
      file_close(avg_sweep_bmp);
    end if;
-- Scrittura su file del vettore STC
-- La scrittura avviene in seguito a un fronte sul segnale "write_in_file_stc"
    if (write_in_file_stc'event and write_in_file_stc='1') then
      file_open(pow_vec, "pow_vec.txt", WRITE_MODE);
      for j in 0 to (sample_in_sweep - 1) loop
        hwrite(out_line, tmp_ram((8 * j) + conv_integer(offset_vectors) + 4));
        hwrite(out_line, tmp_ram((8 * j) + conv_integer(offset_vectors) + 5));
        hwrite(out_line, tmp_ram((8 * j) + conv_integer(offset_vectors) + 6));
        hwrite(out_line, tmp_ram((8 * j) + conv_integer(offset_vectors) + 7));
        writeline(pow_vec, out_line);
      end loop;
      file_close(pow_vec);
    end if;
-- Scrittura su file della mappa elaborata, sia in formato numerico che grafico BMP
-- La scrittura avviene in seguito a un fronte sul segnale "write_in_file_map"
    if (write_in_file_map'event and write_in_file_map='1') then
-- Viene prima creato un array che contiene i "num_sweep_in_acq * sample_in_sweep"
-- elementi della mappa elaborata
      for i in 0 to num_sweep_in_acq * sample_in_sweep - 1 loop
        acq_elab(i) := tmp_ram((4 * i) + conv_integer(offset_elab_sweep) + 0)&
                       tmp_ram((4 * i) + conv_integer(offset_elab_sweep) + 1)&
                       tmp_ram((4 * i) + conv_integer(offset_elab_sweep) + 2)&
                       tmp_ram((4 * i) + conv_integer(offset_elab_sweep) + 3);
      end loop;
-- Scrittura in formato numerico su file di testo
      file_open(elabmap, "elabmap.txt", WRITE_MODE);
      for i in 0 to num_sweep_in_acq - 1 loop
        for j in 0 to sample_in_sweep - 1 loop
          hwrite(out_line, acq_elab(i * 512 + j));
          write(out_line, string'(" "));
        end loop;
        writeline(elabmap, out_line);
      end loop;
      file_close(elabmap);
-- Scrittura in formato grafico
      file_open(elabmap_bmp, "elabmap_bmp.txt", WRITE_MODE);
      for j in 0 to num_sweep_in_acq * sample_in_sweep - 1 loop
-- viene chiamata la funzione visualizza con il parametro "sigma"=14 scelto in seguito
-- a prove empiriche
        bmp := visualizza(14, acq_elab(j));
        hwrite(out_line, bmp);
        write(out_line, string'(" "));
        hwrite(out_line, bmp);
        write(out_line, string'(" "));
        hwrite(out_line, bmp);
        write(out_line, string'(" "));
        writeline(elabmap_bmp, out_line);
      end loop;
      file_close(elabmap_bmp);
    end if;
end process;
-- Generazione del clock a 40 MHz e dei segnali di ingresso per il testbench
clk : PROCESS (clock)
BEGIN
  clock <= not clock after 12500 ps;
END PROCESS;
tb : PROCESS
BEGIN
  reset <= '0';
  start <= '0';
  En_bgr <= '1';
  En_stc <= '1';
  write_in_file_avg <= '0';
  write_in_file_stc <= '0';
  write_in_file_map <= '0';
  wait for 90 ns;
  reset <= '1';
  wait for 60 ns;
  start <= '1';
  wait for 90 ns;
  start <= '0';
  wait for 251 ms;
  write_in_file_avg <= '1';
  write_in_file_stc <= '1';
  write_in_file_map <= '1';
  wait;
END PROCESS;
END;
```

A.3 VHDL per gli algoritmi di migrazione

A.3.1 VHDL Blocco Engine

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity Engine_mig is
-- Definizione delle costanti
generic( jumps_per_pattern : integer := 32;
         sample_in_sweep : integer := 512;
         sweep_in_acq:   integer:= 512);
port(    Clock: in std_logic;
         Reset: in std_logic;
         Start: in std_logic;
         -- Uscita bus indirizzi
         Address: out std_logic_vector(23 downto 0);
         -- Interfaccia RAM
         OE_mem: out std_logic;
         Write_mem: out std_logic;
         Data: inout std_logic_vector(7 downto 0);
         -- Bus e segnali interni
         Data_to_proc: out std_logic_vector(31 downto 0);
         Data_to_mem: in std_logic_vector(31 downto 0);
         data_ready: out std_logic;
         clear: out std_logic;
         next_sample: out std_logic
    );
end;
architecture Behav of Engine_mig is
-- Segnali per la sincronizzazione degli ingressi asincroni
signal reset_sync_p, reset_sync_n : std_logic;
signal start_sync_p, start_sync_n : std_logic;
-- definizione degli indirizzi di memoria
constant offset_jump : std_logic_vector(23 downto 0)    := X"000000";
constant offset_map : std_logic_vector(23 downto 0)     := X"008000";
constant offset_migrmap : std_logic_vector(23 downto 0) := X"108000";
-- definizione di un sottoinsieme della mappa per sul quale viene applicato l'algoritmo di migrazione
-- (vengono esclusi i primi 16 sweep e gli ultimi 16 sweep)
constant margin_start_migration : std_logic_vector(23 downto 0) := X"008800";
constant margin_stop_migration : std_logic_vector(23 downto 0)  := X"0F7800";
-- derfinizione delle soglie Sth oltre le quali viene applicato l'algoritmo di migrazione su pattern
constant pos_threshold : std_logic_vector(31 downto 0) := X"00002000";
constant neg_threshold : std_logic_vector(31 downto 0) := X"FFFFDFFF";
-- Definizione dei tipi enumerati (stati della macchina a stati e selettore degli indirizzi)
type state_engine_type is (
  s0,
  read_jump_i1_a, wait1,
  read_jump_i0_a, wait2,
  read_map_i1_a, wait3,
  read_map_i0_a, wait4,
  read_map_d0_a, wait5,
  read_map_d1_a, wait6,
  write_migrmap_i1_a, write_migrmap_i1_b, wait7,
  write_migrmap_i0_a, write_migrmap_i0_b, wait8,
  write_migrmap_d0_a, write_migrmap_d0_b, wait9,
  write_migrmap_d1_a, write_migrmap_d1_b, wait10);
type Sel_address_type is (
 Sel_address_r_map, Sel_address_r_jump, Sel_address_w);
-- Indirizzamento della memoria
signal Address_r_map_p, Address_r_map_n : std_logic_vector(23 downto 0);
signal Address_r_jump_p, Address_r_jump_n : std_logic_vector(23 downto 0);
signal Address_w_p, Address_w_n : std_logic_vector(23 downto 0);
signal Sel_address_p, Sel_address_n : Sel_address_type;
-- Segnali di ingresso e uscita dei registri della macchina a stati
signal jump_p, jump_n : std_logic_vector(15 downto 0);
signal cont_jump_p, cont_jump_n : std_logic_vector(4 downto 0);
signal OE_mem_p, OE_mem_n : std_logic;
signal Write_mem_p, Write_mem_n : std_logic;
signal data_p, data_n : std_logic_vector(7 downto 0);
signal data_to_proc_p, data_to_proc_n : std_logic_vector(31 downto 0);
signal data_ready_p, data_ready_n : std_logic;
signal clear_p, clear_n : std_logic;
signal int_next_sample : std_logic;
-- Definizione degli stati
signal state_engine_p, state_engine_n : state_engine_type;
begin
-- Processo sequenziale:
seq: process (clock) begin
  if (clock'event and clock='1') then
       reset_sync_p <= reset_sync_n;
```

```
        start_sync_p <= start_sync_n;
        Address_r_map_p <= Address_r_map_n;
        Address_r_jump_p <= Address_r_jump_n;
        Address_w_p <= Address_w_n;
        Sel_address_p <= Sel_address_n;
        jump_p <= jump_n;
        cont_jump_p <= cont_jump_n;
        OE_mem_p <= OE_mem_n;
        Write_mem_p <= Write_mem_n;
        data_p <= data_n;
        data_to_proc_p <= data_to_proc_n;
        data_ready_p <= data_ready_n;
        clear_p <= clear_n;
        state_engine_p <= state_engine_n;
  end if;
end process;
-- Processo combinatorio:
com: process (
reset_sync_p, start_sync_p,
data_to_mem, int_next_sample,
state_engine_p,
Address_r_map_p, Address_r_jump_p, Address_w_p,
jump_p, cont_jump_p, data, data_to_proc_p)
begin
-- Condizione per il reset
if reset_sync_p = '0' then
  Address_r_map_n <= offset_map;
  Address_r_jump_n <= offset_jump;
  Address_w_n <= offset_migrmap;
  Sel_address_n <= Sel_address_r_jump;
  jump_n(15 downto 8) <= (others => '0') ;
  jump_n(7 downto 0) <= (others => '0') ;
  cont_jump_n <= (others => '0');
  OE_mem_n <= '1';
  Write_mem_n <= '1';
  data_n <= (others => '0') ;
  data_to_proc_n(31 downto 24) <= (others => '0') ;
  data_to_proc_n(23 downto 16) <= (others => '0') ;
  data_to_proc_n(15 downto 8) <= (others => '0') ;
  data_to_proc_n(7 downto 0) <= (others => '0') ;
  data_ready_n <= '0';
  clear_n <= '1';
  state_engine_n <= s0;
 else
-- Condizione operativa della macchina a stati
  case state_engine_p is
   when s0 =>
    Address_r_map_n <= offset_map;
    Address_r_jump_n <= offset_jump;
    Address_w_n <= offset_migrmap;
    Sel_address_n <= Sel_address_r_jump;
    jump_n(15 downto 8) <= (others => '0') ;
    jump_n(7 downto 0) <= (others => '0') ;
    cont_jump_n <= (others => '0');
    OE_mem_n <= '1';
    Write_mem_n <= '1';
    data_n <= (others => '0') ;
    data_to_proc_n(31 downto 24) <= (others => '0') ;
    data_to_proc_n(23 downto 16) <= (others => '0') ;
    data_to_proc_n(15 downto 8) <= (others => '0') ;
    data_to_proc_n(7 downto 0) <= (others => '0') ;
    data_ready_n <= '0';
    clear_n <= '1';
-- Attesa del segnale di start (generato dal testbench)
    if start_sync_p = '1' then
      state_engine_n <= read_jump_i1_a;
     else
      state_engine_n <= s0;
    end if;
-- Lettura del valore "jump"
   when read_jump_i1_a =>
    Address_r_map_n <= Address_r_map_p;
    Address_r_jump_n <= Address_r_jump_p;
    Address_w_n <= Address_w_p;
    Sel_address_n <= Sel_address_r_jump;
-- Con questa condizione viene imposto il valore zero alla parte alta del vettore "jump" se viene
-- letta una zona della mappa nella quale non deve essere eseguito l'algoritmo di migrazione
    if ((Address_r_map_p < offset_map + margin_start_migration) or
            (Address_r_map_p > offset_map + margin_stop_migration - 1)) then
      jump_n(15 downto 8) <= (others => '0');
     else
      jump_n(15 downto 8) <= data;
    end if;
    jump_n(7 downto 0) <= jump_p(7 downto 0);
    cont_jump_n <= cont_jump_p;
    OE_mem_n <= '0';
    Write_mem_n <= '1';
    data_n <= data;
    data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
    data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
```

```
      data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
      data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
      data_ready_n <= '0';
      clear_n <= '0';
      state_engine_n <= wait1;
    when wait1 =>
      Address_r_map_n <= Address_r_map_p;
      Address_r_jump_n <= Address_r_jump_p + 1;
      Address_w_n <= Address_w_p;
      Sel_address_n <= Sel_address_r_jump;
      jump_n(15 downto 8) <= jump_p(15 downto 8);
      jump_n(7 downto 0) <= jump_p(7 downto 0);
      cont_jump_n <= cont_jump_p;
      OE_mem_n <= '0';
      Write_mem_n <= '1';
      data_n <= data;
      data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
      data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
      data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
      data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
      data_ready_n <= '0';
      clear_n <= '0';
      state_engine_n <= read_jump_i0_a;
    when read_jump_i0_a =>
      Address_r_map_n <= Address_r_map_p;
      Address_r_jump_n <= Address_r_jump_p;
      Address_w_n <= Address_w_p;
      Sel_address_n <= Sel_address_r_jump;
      jump_n(15 downto 8) <= jump_p(15 downto 8);
-- Con questa condizione viene imposto il valore zero alla parte bassa del vettore "jump" se viene
-- letta una zona della mappa nella quale non deve essere eseguito l'algoritmo di migrazione
      if ((Address_r_map_p < offset_map + margin_start_migration) or
          (Address_r_map_p > offset_map + margin_stop_migration - 1)) then
        jump_n(7 downto 0) <= (others => '0');
       else
        jump_n(7 downto 0) <= data;
      end if;
      cont_jump_n <= cont_jump_p;
      OE_mem_n <= '0';
      Write_mem_n <= '1';
      data_n <= data;
      data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
      data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
      data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
      data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
      data_ready_n <= '0';
      clear_n <= '0';
      state_engine_n <= wait2;
    when wait2 =>
      Address_r_map_n <= Address_r_map_p;
-- Condizione per cui se viene letto l'ultimo valore del vettore "jump" relativo
-- all'ultimo campione dello sweep inizia nuovamenre il ciclo di lettura
      if Address_r_jump_p = offset_jump + sample_in_sweep*jumps_per_pattern*2 - 1 then
        Address_r_jump_n <= offset_jump;
       else
        Address_r_jump_n <= Address_r_jump_p + 1;
      end if;
      Address_w_n <= Address_w_p;
 -- In questo stato il valore del vettore "jump" è stato letto, nel prossimo deve essere letto
 -- il valore del campione in esame, quindi viene impostato il multiplexer  sul bus indirizzi
 -- affinché venga letto il campione corrispondente
      Sel_address_n <= Sel_address_r_map;
      jump_n(15 downto 8) <= jump_p(15 downto 8);
      jump_n(7 downto 0) <= jump_p(7 downto 0);
      cont_jump_n <= cont_jump_p + 1;
      OE_mem_n <= '0';
      Write_mem_n <= '1';
      data_n <= data;
      data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
      data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
      data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
      data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
      data_ready_n <= '0';
      clear_n <= '0';
      state_engine_n <= read_map_i1_a;
    when read_map_i1_a =>
      Address_r_map_n <= Address_r_map_p;
      Address_r_jump_n <= Address_r_jump_p;
      Address_w_n <= Address_w_p;
      Sel_address_n <= Sel_address_r_map;
      jump_n(15 downto 8) <= jump_p(15 downto 8);
      jump_n(7 downto 0) <= jump_p(7 downto 0);
      cont_jump_n <= cont_jump_p;
      OE_mem_n <= '0';
      Write_mem_n <= '1';
      data_n <= data;
      data_to_proc_n(31 downto 24) <= data;
      data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
      data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
```

```
  data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
  data_ready_n <= '0';
  clear_n <= '0';
  state_engine_n <= wait3;
 when wait3 =>
  Address_r_map_n <= Address_r_map_p + 1;
  Address_r_jump_n <= Address_r_jump_p;
  Address_w_n <= Address_w_p;
  Sel_address_n <= Sel_address_r_map;
  jump_n(15 downto 8) <= jump_p(15 downto 8);
  jump_n(7 downto 0) <= jump_p(7 downto 0);
  cont_jump_n <= cont_jump_p;
  OE_mem_n <= '0';
  Write_mem_n <= '1';
  data_n <= data;
  data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
  data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
  data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
  data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
  data_ready_n <= '0';
  clear_n <= '0';
  state_engine_n <= read_map_i0_a;
 when read_map_i0_a =>
  Address_r_map_n <= Address_r_map_p;
  Address_r_jump_n <= Address_r_jump_p;
  Address_w_n <= Address_w_p;
  Sel_address_n <= Sel_address_r_map;
  jump_n(15 downto 8) <= jump_p(15 downto 8);
  jump_n(7 downto 0) <= jump_p(7 downto 0);
  cont_jump_n <= cont_jump_p;
  OE_mem_n <= '0';
  Write_mem_n <= '1';
  data_n <= data;
  data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
  data_to_proc_n(23 downto 16) <= data;
  data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
  data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
  data_ready_n <= '0';
  clear_n <= '0';
  state_engine_n <= wait4;
 when wait4 =>
  Address_r_map_n <= Address_r_map_p + 1;
  Address_r_jump_n <= Address_r_jump_p;
  Address_w_n <= Address_w_p;
  Sel_address_n <= Sel_address_r_map;
  jump_n(15 downto 8) <= jump_p(15 downto 8);
  jump_n(7 downto 0) <= jump_p(7 downto 0);
  cont_jump_n <= cont_jump_p;
  OE_mem_n <= '0';
  Write_mem_n <= '1';
  data_n <= data;
  data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
  data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
  data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
  data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
  data_ready_n <= '0';
  clear_n <= '0';
  state_engine_n <= read_map_d1_a;
 when read_map_d1_a =>
  Address_r_map_n <= Address_r_map_p;
  Address_r_jump_n <= Address_r_jump_p;
  Address_w_n <= Address_w_p;
  Sel_address_n <= Sel_address_r_map;
  jump_n(15 downto 8) <= jump_p(15 downto 8);
  jump_n(7 downto 0) <= jump_p(7 downto 0);
  cont_jump_n <= cont_jump_p;
  OE_mem_n <= '0';
  Write_mem_n <= '1';
  data_n <= data;
  data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
  data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
  data_to_proc_n(15 downto 8) <= data;
  data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
  data_ready_n <= '0';
  clear_n <= '0';
  state_engine_n <= wait5;
 when wait5 =>
  Address_r_map_n <= Address_r_map_p + 1;
  Address_r_jump_n <= Address_r_jump_p;
  Address_w_n <= Address_w_p;
  Sel_address_n <= Sel_address_r_map;
  jump_n(15 downto 8) <= jump_p(15 downto 8);
  jump_n(7 downto 0) <= jump_p(7 downto 0);
  cont_jump_n <= cont_jump_p;
  OE_mem_n <= '0';
  Write_mem_n <= '1';
  data_n <= data;
  data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
  data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
```

```
    data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
    data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
    data_ready_n <= '0';
    clear_n <= '0';
    state_engine_n <= read_map_d0_a;
  when read_map_d0_a =>
    Address_r_map_n <= Address_r_map_p;
    Address_r_jump_n <= Address_r_jump_p;
    Address_w_n <= Address_w_p;
    Sel_address_n <= Sel_address_r_map;
    jump_n(15 downto 8) <= jump_p(15 downto 8);
    jump_n(7 downto 0) <= jump_p(7 downto 0);
    cont_jump_n <= cont_jump_p;
    OE_mem_n <= '0';
    Write_mem_n <= '1';
    data_n <= data;
    data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
    data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
    data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
    data_to_proc_n(7 downto 0) <= data;
    data_ready_n <= '1';
    clear_n <= '0';
    state_engine_n <= wait6;
  when wait6 =>
    Address_w_n <= Address_w_p;
    jump_n(15 downto 8) <= jump_p(15 downto 8);
    jump_n(7 downto 0) <= jump_p(7 downto 0);
    Write_mem_n <= '1';
    data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
    data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
    data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
    data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
    data_ready_n <= '0';
    clear_n <= '0';
-- Il valore del registro "address_r_jump" dipende dalle varie condizioni:
-- - Se ha raggiunto il valore della costante "offset_map" significa che
--   è stata raggiunta la fine della zona di memoria riservata ai valori "jump"
--   quindi deve essere impostato al valore "offset_jump"
-- - Nel caso in cui non sia raggiunta la precedente condizione deve essere
--   letto il segnale "int_next_sample": se questo vale "0" significa che deve
--   essere applicato l'algoritmo di mgrazione leggendo uno dopo l'altro i valori
--   "jump", quindi il registro non deve essere modificato perché viene
--   incrementato di una unità negli stati "wait1" e "wait2"
-- - Nel caso in cui il segnale "int_next_sample" sia uguale a '1' è necessario
--   valutare se il registro stesso contiene l'indirizzo del primo "jump" relativo
--   all'ultimo campione dello sweep (in questo caso il 511-esimo). Se questa
--   condizione è vera il valore del "jump" successivo è il valore del primo "jump"
--   per il primo campione dello sweep (quindi quello scritto a partire dall'indirizzo
--   "offset_jump)", altrimenti il valore del "jump" per il campione successivo
--   si trova aumentando di 62 il valore del registro "Address_r_jump"
    if address_r_jump_p = offset_map then
      Address_r_jump_n <= offset_jump;
     elsif int_next_sample = '0' then
       address_r_jump_n <= address_r_jump_p;
        elsif address_r_jump_p = X"7FC2" then
          Address_r_jump_n <= offset_jump;
         else
          Address_r_jump_n <= address_r_jump_p + 62;
    end if;
-- In questo stato viene valutata la condizione per l'analisi del campione successivo:
-- la condizione si verifica quando il segnale "int_next_sample" è attivo, ovvero
-- il valore assoluto del campione in esame risulta essere sotto la soglia Sth,
-- o quando sono state effettuate la lettura dei 32 campioni sul pattern (quindi
-- cont_jump_p torna al valore "00000"). Nel caso in cui la condizione non
-- sia verificata il registro "Address_r_map" viene ogni volta decrementato
-- di 3 in quanto durante gli stati "wait_3", "wait_4", "wait_5" viene incrementato di 1
-- per leggere il dato su 32 bit
    if (int_next_sample = '1' OR cont_jump_p = "00000") then
      Address_r_map_n <= Address_r_map_p + 1;
      OE_mem_n <= '1';
      Sel_address_n <= Sel_address_w;
      state_engine_n <= write_migrmap_i1_a;
      data_n <= data_to_mem(31 downto 24);
      cont_jump_n <= "00000";
     else
      Address_r_map_n <= Address_r_map_p - 3;
      OE_mem_n <= '0';
      Sel_address_n <= Sel_address_r_jump;
      state_engine_n <= read_jump_i1_a;
      data_n <= data;
      cont_jump_n <= cont_jump_p;
    end if;
-- Inizio scrittura
  when write_migrmap_i1_a =>
    Address_r_map_n <= Address_r_map_p;
    Address_r_jump_n <= Address_r_jump_p;
    Address_w_n <= Address_w_p;
    Sel_address_n <=     Sel_address_w;
    jump_n(15 downto 8) <= jump_p(15 downto 8);
    jump_n(7 downto 0) <= jump_p(7 downto 0);
```

```
  cont_jump_n <= cont_jump_p;
  OE_mem_n <= '1';
  Write_mem_n <= '0';
  data_n <= data_to_mem(31 downto 24);
  data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
  data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
  data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
  data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
  data_ready_n <= '0';
  clear_n <= '0';
  state_engine_n <= write_migrmap_i1_b;
when write_migrmap_i1_b =>
  Address_r_map_n <= Address_r_map_p;
  Address_r_jump_n <= Address_r_jump_p;
  Address_w_n <= Address_w_p;
  Sel_address_n <=    Sel_address_w;
  jump_n(15 downto 8) <= jump_p(15 downto 8);
  jump_n(7 downto 0) <= jump_p(7 downto 0);
  cont_jump_n <= cont_jump_p;
  OE_mem_n <= '1';
  Write_mem_n <= '1';
  data_n <= data_to_mem(31 downto 24);
  data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
  data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
  data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
  data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
  data_ready_n <= '0';
  clear_n <= '0';
  state_engine_n <= wait7;
when wait7 =>
  Address_r_map_n <= Address_r_map_p;
  Address_r_jump_n <= Address_r_jump_p;
  Address_w_n <= Address_w_p + 1;
  Sel_address_n <=    Sel_address_w;
  jump_n(15 downto 8) <= jump_p(15 downto 8);
  jump_n(7 downto 0) <= jump_p(7 downto 0);
  cont_jump_n <= cont_jump_p;
  OE_mem_n <= '1';
  Write_mem_n <= '1';
  data_n <= data_to_mem(23 downto 16);
  data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
  data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
  data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
  data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
  data_ready_n <= '0';
  clear_n <= '0';
  state_engine_n <= write_migrmap_i0_a;
when write_migrmap_i0_a =>
  Address_r_map_n <= Address_r_map_p;
  Address_r_jump_n <= Address_r_jump_p;
  Address_w_n <= Address_w_p;
  Sel_address_n <=    Sel_address_w;
  jump_n(15 downto 8) <= jump_p(15 downto 8);
  jump_n(7 downto 0) <= jump_p(7 downto 0);
  cont_jump_n <= cont_jump_p;
  OE_mem_n <= '1';
  Write_mem_n <= '0';
  data_n <= data_to_mem(23 downto 16);
  data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
  data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
  data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
  data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
  data_ready_n <= '0';
  clear_n <= '0';
  state_engine_n <= write_migrmap_i0_b;
when write_migrmap_i0_b =>
  Address_r_map_n <= Address_r_map_p;
  Address_r_jump_n <= Address_r_jump_p;
  Address_w_n <= Address_w_p;
  Sel_address_n <=    Sel_address_w;
  jump_n(15 downto 8) <= jump_p(15 downto 8);
  jump_n(7 downto 0) <= jump_p(7 downto 0);
  cont_jump_n <= cont_jump_p;
  OE_mem_n <= '1';
  Write_mem_n <= '1';
  data_n <= data_to_mem(23 downto 16);
  data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
  data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
  data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
  data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
  data_ready_n <= '0';
  clear_n <= '0';
  state_engine_n <= wait8;
when wait8 =>
  Address_r_map_n <= Address_r_map_p;
  Address_r_jump_n <= Address_r_jump_p;
  Address_w_n <= Address_w_p + 1;
  Sel_address_n <=    Sel_address_w;
```

```
  jump_n(15 downto 8) <= jump_p(15 downto 8);
  jump_n(7 downto 0) <= jump_p(7 downto 0);
  cont_jump_n <= cont_jump_p;
  OE_mem_n <= '1';
  Write_mem_n <= '1';
  data_n <= data_to_mem(15 downto 1*8);
  data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
  data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
  data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
  data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
  data_ready_n <= '0';
  clear_n <= '0';
  state_engine_n <= write_migrmap_d1_a;
 when write_migrmap_d1_a =>
  Address_r_map_n <= Address_r_map_p;
  Address_r_jump_n <= Address_r_jump_p;
  Address_w_n <= Address_w_p;
  Sel_address_n <=    Sel_address_w;
  jump_n(15 downto 8) <= jump_p(15 downto 8);
  jump_n(7 downto 0) <= jump_p(7 downto 0);
  cont_jump_n <= cont_jump_p;
  OE_mem_n <= '1';
  Write_mem_n <= '0';
  data_n <= data_to_mem(15 downto 1*8);
  data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
  data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
  data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
  data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
  data_ready_n <= '0';
  clear_n <= '0';
  state_engine_n <= write_migrmap_d1_b;
 when write_migrmap_d1_b =>
  Address_r_map_n <= Address_r_map_p;
  Address_r_jump_n <= Address_r_jump_p;
  Address_w_n <= Address_w_p;
  Sel_address_n <=    Sel_address_w;
  jump_n(15 downto 8) <= jump_p(15 downto 8);
  jump_n(7 downto 0) <= jump_p(7 downto 0);
  cont_jump_n <= cont_jump_p;
  OE_mem_n <= '1';
  Write_mem_n <= '1';
  data_n <= data_to_mem(15 downto 1*8);
  data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
  data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
  data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
  data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
  data_ready_n <= '0';
  clear_n <= '0';
  state_engine_n <= wait9;
 when wait9 =>
  Address_r_map_n <= Address_r_map_p;
  Address_r_jump_n <= Address_r_jump_p;
  Address_w_n <= Address_w_p + 1;
  Sel_address_n <=    Sel_address_w;
  jump_n(15 downto 8) <= jump_p(15 downto 8);
  jump_n(7 downto 0) <= jump_p(7 downto 0);
  cont_jump_n <= cont_jump_p;
  OE_mem_n <= '1';
  Write_mem_n <= '1';
  data_n <= data_to_mem(7 downto 0);
  data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
  data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
  data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
  data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
  data_ready_n <= '0';
  clear_n <= '0';
  state_engine_n <= write_migrmap_d0_a;
 when write_migrmap_d0_a =>
  Address_r_map_n <= Address_r_map_p;
  Address_r_jump_n <= Address_r_jump_p;
  Address_w_n <= Address_w_p;
  Sel_address_n <=    Sel_address_w;
  jump_n(15 downto 8) <= jump_p(15 downto 8);
  jump_n(7 downto 0) <= jump_p(7 downto 0);
  cont_jump_n <= cont_jump_p;
  OE_mem_n <= '1';
  Write_mem_n <= '0';
  data_n <= data_to_mem(7 downto 0);
  data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
  data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
  data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
  data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
  data_ready_n <= '0';
  clear_n <= '0';
  state_engine_n <= write_migrmap_d0_b;
 when write_migrmap_d0_b =>
  Address_r_map_n <= Address_r_map_p;
  Address_r_jump_n <= Address_r_jump_p;
```

```
      Address_w_n <= Address_w_p;
      Sel_address_n <=     Sel_address_w;
      jump_n(15 downto 8) <= jump_p(15 downto 8);
      jump_n(7 downto 0) <= jump_p(7 downto 0);
      cont_jump_n <= cont_jump_p;
      OE_mem_n <= '1';
      Write_mem_n <= '1';
      data_n <= data_to_mem(7 downto 0);
      data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
      data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
      data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
      data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
      data_ready_n <= '0';
      clear_n <= '0';
      state_engine_n <= wait10;
    when wait10 =>
      Address_r_map_n <= Address_r_map_p;
      Address_r_jump_n <= Address_r_jump_p;
      Address_w_n <= Address_w_p + 1;
      Sel_address_n <= Sel_address_r_jump;
      jump_n(15 downto 8) <= jump_p(15 downto 8);
      jump_n(7 downto 0) <= jump_p(7 downto 0);
      cont_jump_n <= cont_jump_p;
      OE_mem_n <= '0';
      Write_mem_n <= '1';
      data_n <= data;
      data_to_proc_n(31 downto 24) <= data_to_proc_p(31 downto 24);
      data_to_proc_n(23 downto 16) <= data_to_proc_p(23 downto 16);
      data_to_proc_n(15 downto 8) <= data_to_proc_p(15 downto 8);
      data_to_proc_n(7 downto 0) <= data_to_proc_p(7 downto 0);
      data_ready_n <= '0';
      clear_n <= '1';
-- Terminata la scrittura viene controllato il valore del registro REG_ADDRESS_R_MAP: se non ha
-- raggiunto il valore della costante "offset_migrmap" il processo deve essere ripetuto per il
-- campione successivo
      if (Address_r_map_p > offset_migrmap - 1) then
        state_engine_n <= s0;
       else
        state_engine_n <= read_jump_i1_a;
      end if;
    when others =>
      Address_r_map_n <= Address_r_map_p;
      Address_r_jump_n <= Address_r_jump_p;
      Address_w_n <= Address_w_p;
      Sel_address_n <= Sel_address_r_map;
      jump_n(15 downto 8) <= jump_p(15 downto 8);
      jump_n(7 downto 0) <= jump_p(7 downto 0);
      cont_jump_n <= cont_jump_p;
      OE_mem_n <= '1';
      Write_mem_n <= '1';
      data_n <= (others => '0') ;
      data_to_proc_n(31 downto 24) <= (others => '0') ;
      data_to_proc_n(23 downto 16) <= (others => '0') ;
      data_to_proc_n(15 downto 8) <= (others => '0') ;
      data_to_proc_n(7 downto 0) <= (others => '0') ;
      data_ready_n <= '0';
      clear_n <= '1';
      state_engine_n <= s0;
  end case;
end if;
end process;
process (Address_r_jump_p, Address_r_map_p, Address_w_p, Sel_address_p, jump_p)
variable jump_ca2 : std_logic_vector(23 downto 0);
    begin
-- Viene effettuata la moltiplicazione per 4 del valore nel registro "jump" mediante lo
-- shift a sinistra di due bit e l'espansione della dinamica sulla variabile "jump_ca2"
        jump_ca2(23 downto 18) := (others => jump_p(15));
        jump_ca2(17 downto 2) := jump_p(15 downto 0);
        jump_ca2(1 downto 0) := "00";
        case Sel_address_p is
            when Sel_address_r_map =>
                Address <= Address_r_map_p + jump_ca2;
            when Sel_address_r_jump =>
                Address <= Address_r_jump_p;
            when Sel_address_w =>
                Address <= Address_w_p;
            when others =>
                Address <= Address_r_map_p;
        end case;
end process;
-- Viene descritto il blocco "NEXT_SAMPLE_GENERATOR" che genera il segnale
-- "int_next_sample"
process(jump_p, data_to_proc_p)
  begin
    if (jump_p = X"0000" AND (Data_to_proc_p < pos_threshold OR Data_to_proc_p > neg_threshold))
          OR jump_p = X"7FFF" then
      int_next_sample <= '1';
     else
      int_next_sample <= '0';
```

```
    end if;
end process;
-- Collego gli ingressi e le uscite dei registri alle uscite del blocco Engine
start_sync_n <= start;
reset_sync_n <= reset;
OE_mem <= OE_mem_p;
Write_mem <= Write_mem_p;
Data_to_proc <= data_to_proc_p;
Data_ready <= Data_ready_p;
clear <= clear_p;
data <= data_p when OE_mem_p = '1' else "ZZZZZZZZ";
next_sample <= int_next_sample;
end Behav;
```

A.3.2 VHDL Blocco Processing

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity Processing_mig is
port(    Clock: in std_logic;
         clear: in std_logic;
         data_ready: in std_logic;
         next_sample: in std_logic;
         Data_to_proc: in std_logic_vector(31 downto 0);
         Data_to_mem: out std_logic_vector(31 downto 0));
end;
architecture behav of Processing_mig is
-- Definisco i registri per il calcolo della sommatoria su pattern
signal acc_n, acc_p : std_logic_vector(36 downto 0);
begin
-- Processo sequenziale
process (clock) begin
  if (clock'event and clock = '1') then
    acc_p <= acc_n;
  end if;
end process;
-- Processo combinatorio
process (clear, data_ready, data_to_proc, next_sample,
         acc_p)
-- Definizione della variabile "data_to_proc_ca2" per l'espansione della dinamica
-- del bus "data_to_proc"
  variable data_to_proc_ca2 : std_logic_vector(36 downto 0);
  begin
    data_to_proc_ca2(36 downto 32) := (others => data_to_proc(31));
    data_to_proc_ca2(31 downto 0) := data_to_proc;
    if (clear = '1') then
      acc_n <= (others => '0');
     else
      if data_ready = '1' then
        if (next_sample = '1') then
-- Con l'impostazione "acc_n <= (others => '0')" nel caso in cui l'ingresso "next_sample = '1'"
-- si ottiene la cancellazione dalla mappa di tutte le zone in cui non sussistano le condizioni
-- per effettuare la migrazione sul partttern: in questo modo la mappa risultante riporterà
-- solamente le zone di interesse
          acc_n <= (others => '0');
         else
-- Nel caso in cui deve essere effettuata la migrazione lungo il pattern viene effettuata
-- l'accumulazione
          acc_n <= acc_p + data_to_proc_ca2;
        end if;
       else
        acc_n <= acc_p;
      end if;
    end if;
end process;
-- L'uscita del blocco "Processing" è la media dei 32 campioni lungo il pattern,
-- ottenuta eliminando i 5 bit meno significativi del registro accumulatore "acc"
data_to_mem <= acc_p(36 downto 5);
end behav;
```

A.3.3 VHDL Testbench

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
USE IEEE.STD_LOGIC_TEXTIO.ALL;
USE STD.TEXTIO.ALL;
library xp2;
use xp2.components.all;
ENTITY testbench IS
generic( sample_in_sweep : integer := 512;
         byte_per_sample : integer := 4;
         byte_per_jump : integer := 2;
         jumps_for_sample : integer := 32;
         sweep_in_acq :   integer:= 512);
END testbench;
Architecture behavior of testbench is
 signal Data : std_logic_vector (7 downto 0);
 signal Write_mem : std_logic;
 signal OE_mem : std_logic;
 signal Address : std_logic_vector (23 downto 0);
 signal Start : std_logic;
 signal Reset : std_logic;
 signal Clock : std_logic := '0';
 component SYSTEM_MIG
  Port (
   Data : InOut std_logic_vector (7 downto 0);
   Write_mem : Out   std_logic;
   OE_mem : Out   std_logic;
   Address : Out   std_logic_vector (23 downto 0);
   Start : In     std_logic;
   Reset : In     std_logic;
   Clock : In     std_logic);
 end component;
-- Segnali per abilitare la scrittura su file della mappa migrata
 signal write_in_file_migrmap: std_logic;
-- Definizione del tipo RAM
 type ram_type is array (0 to 2**24-1) of std_logic_vector(7 downto 0);
-- Dichiarazione dei tipi file per lettura e scrittura dei dati:
-- File "jumps"
 file jumps: text;
-- File contenente la mappa elaborata con gli algoritmi di visualizzazione
 file elabmap: text;
-- File per il salvataggio della mappa migrata
 file migrmap: text;
-- File per il salvataggio della mappa migrata in formato grafico
 file migrmap_bmp: text;
-- Definizione del tipo array per la scrittura su file della mappa migrata
 type acq_type is array (0 to sweep_in_acq * sample_in_sweep - 1) of std_logic_vector(31 downto 0);
-- Costanti per la mappatura della memoria
 constant offset_jump : std_logic_vector(23 downto 0) := X"000000";
 constant offset_map : std_logic_vector(23 downto 0) := X"008000";
 constant offset_migrmap : std_logic_vector(23 downto 0) := X"108000";
-- Definizione della funzione "visualizza" per la conversione in formato grafico della
-- mappa migrata, analoga a quella definita nel testbench per gli algoritmi di
-- visualizzazione
 function visualizza (
  sigma: integer;
  elab_sample: std_logic_vector(31 downto 0))
  return std_logic_vector is
    variable saturato_neg : boolean := FALSE;
    variable saturato_pos : boolean := FALSE;
    variable elab_sample_temp : std_logic_vector(31 downto 0);
    variable bmp_temp : std_logic_vector(7 downto 0);
    constant all1 : std_logic_vector(31 downto 0) := (others => '1');
    constant all0 : std_logic_vector(31 downto 0) := (others => '0');
    begin
      elab_sample_temp(31 downto 31-sigma) := elab_sample(31 downto 31-sigma);
      if (elab_sample(31) = '1') then
        elab_sample_temp(31-sigma-1 downto 0) := (others => '1');
        if elab_sample_temp = all1 then saturato_neg := FALSE; else saturato_neg := TRUE; end if;
       else
        elab_sample_temp(31-sigma-1 downto 0) := (others => '0');
        if elab_sample_temp = all0 then saturato_pos := FALSE; else saturato_pos := TRUE; end if;
      end if;
      if (saturato_neg = FALSE AND saturato_pos = FALSE)  then
        elab_sample_temp(31 downto sigma) := elab_sample(31 - sigma downto 0);
        elab_sample_temp(sigma - 1 downto 0) := (others => '0');
        bmp_temp := elab_sample_temp(31 downto 24) + x"80";
       elsif saturato_pos = TRUE then
        bmp_temp := x"FF";
       else
        bmp_temp := x"00";
      end if;
   return bmp_temp;
 end visualizza;
BEGIN
```

```
UUT : SYSTEM_MIG
  Port Map ( Address=>Address, Clock=>Clock, Data=>Data,
             OE_mem=>OE_mem, Reset=>Reset, Start=>Start,
             Write_mem=>Write_mem);
-- Processo che simula la RAM, legge il file di acquisizione e scrive i file dei risultati
process (write_mem, OE_mem, data, write_in_file_migrmap, address)
  Variable tmp_ram : ram_type;
  Variable in_line : LINE;
  Variable out_line : LINE;
  Variable sample : std_logic_vector(31 downto 0);
  Variable jump : std_logic_vector(15 downto 0);
  Variable acq_migr : acq_type;
  Variable first_process : boolean := TRUE;
  Variable bmp : std_logic_vector(7 downto 0);
  begin
-- La prima volta che viene eseguito il process vengono copiati i file "jumps.txt" e
-- "elabmap.txt" in memoria; il file "jumps.txt" è un file di testo così composto:
-- - ogni riga contiene i 32 salti separati da uno spazio; La prima riga rappresenta il
--   pattern del primo campione di uno sweep
-- - ogni salto è rappresentato con 16 bit in formato esadecimale complemento a 2
-- il file "elabmap.txt" è un file di testo così composto:
-- - ogni riga contiene i campioni di uno sweep separati da uno spazio; La prima riga
-- rappresenta il primo sweep della mappa, il primo campione è il primo dello sweep
-- - ogni campione è rappresentato con 32 bit in formato esadecimale complemento a 2
    if first_process = TRUE then
      file_open(jumps, "jumps.txt", READ_MODE);
      for i in 0 to (sample_in_sweep - 1) loop
-- lettura di un'intera riga
        READLINE(jumps, in_line);
        for j in 0 to (jumps_for_sample - 1) loop
-- lettura di un valore di jump dalla riga
          HREAD(in_line, jump);
-- scrittura in memoria
          tmp_ram((i * byte_per_jump * jumps_for_sample) +
                  (j * byte_per_jump) + conv_integer(offset_jump)) := jump(15 downto 8);
          tmp_ram((i * byte_per_jump * jumps_for_sample) +
                           (j * byte_per_jump) + conv_integer(offset_jump) + 1) := jump(7 downto 0);
        end loop;
      end loop;
      file_close(jumps);
      file_open(elabmap, "elabmap.txt", READ_MODE);
      for i in 0 to (sweep_in_acq - 1) loop
-- lettura di un'intera riga
        READLINE(elabmap, in_line);
        for j in 0 to (sample_in_sweep - 1) loop
-- lettura di un campione dalla riga
          HREAD(in_line, sample);
-- scrittura in memoria (32 bit)
          tmp_ram((i * byte_per_sample * sample_in_sweep) +
                  (j * byte_per_sample) + conv_integer(offset_map)) := sample(31 downto 24);
          tmp_ram((i * byte_per_sample * sample_in_sweep) +
                  (j * byte_per_sample) + conv_integer(offset_map) + 1) := sample(23 downto 16);
          tmp_ram((i * byte_per_sample * sample_in_sweep) +
                  (j * byte_per_sample) + conv_integer(offset_map) + 2) := sample(15 downto 8);
          tmp_ram((i * byte_per_sample * sample_in_sweep) +
                  (j * byte_per_sample) + conv_integer(offset_map) + 3) := sample(7 downto 0);
        end loop;
      end loop;
      file_close(elabmap);
      first_process := FALSE;
    end if;
-- Simulazione della RAM:
-- Condizione di scrittura
    if write_mem ='0' then
      tmp_ram(conv_integer(Address)) := Data;
    end if;
-- Condizione di lettura
    if OE_mem='0' then
      Data <= tmp_ram(conv_integer(Address));
    else
-- Se Output_enable = 0 il bus viene posto in alta impedenza
      Data <= (others => 'Z');
    end if;
-- Scrittura su file della mappa migrata, sia in formato numerico che grafico BMP
-- La scrittura avviene in seguito a un fronte sul segnale "write_in_file_migrmap"
    if (write_in_file_migrmap'event and write_in_file_migrmap='1') then
-- Viene prima creato un array che contiene i "num_sweep_in_acq * sample_in_sweep"
-- elementi della mappa migrata
      for i in 0 to sweep_in_acq * sample_in_sweep - 1 loop
        acq_migr(i) := tmp_ram((i * byte_per_sample) + conv_integer(offset_migrmap) + 0)&
                       tmp_ram((i * byte_per_sample) + conv_integer(offset_migrmap) + 1)&
                       tmp_ram((i * byte_per_sample) + conv_integer(offset_migrmap) + 2)&
                       tmp_ram((i * byte_per_sample) + conv_integer(offset_migrmap) + 3);
      end loop;
-- Scrittura in formato numerico su file di testo
      file_open(migrmap, "migrmap.txt", WRITE_MODE);
      for i in 0 to sweep_in_acq - 1 loop
```

```
        for j in 0 to sample_in_sweep - 1 loop
          hwrite(out_line, acq_migr(i * 512 + j));
          write(out_line, string'(" "));
        end loop;
        writeline(migrmap, out_line);
      end loop;
      file_close(migrmap);
-- Scrittura in formato grafico
      file_open(migrmap_bmp, "migrmap_bmp.txt", WRITE_MODE);
      for i in 0 to sweep_in_acq * sample_in_sweep - 1 loop
-- viene chiamata la funzione visualizza con il parametro "sigma"=14 scelto in seguito
-- a prove empiriche
        bmp := visualizza(14, acq_migr(i));
        hwrite(out_line, bmp);
        write(out_line, string'(" "));
        hwrite(out_line, bmp);
        write(out_line, string'(" "));
        hwrite(out_line, bmp);
        write(out_line, string'(" "));
        writeline(migrmap_bmp, out_line);
      end loop;
      file_close(migrmap_bmp);
    end if;
end process;
-- Generazione del clock a 40 MHz e dei segnali di ingresso per il testbench
clk : PROCESS (clock)
BEGIN
  clock <= not clock after 12500 ps;
END PROCESS;
tb : PROCESS
BEGIN
  reset <= '0';
  start <= '0';
  write_in_file_migrmap <= '0';
  wait for 90 ns;
  reset <= '1';
  wait for 60 ns;
  start <= '1';
  wait for 90 ns;
  start <= '0';
  wait for 1311 ms;
  write_in_file_migrmap <= '1';
  wait;
END PROCESS;
END;
```

Elenco delle figure

1.1 Georadar IDS Detector e Detector Duo 10
1.2 Descrizione dell'antenna 11
1.3 Segnale captato dall'antenna di ricezione (segnale *A-Scan*) 12
1.4 Esempio di radargramma 12
1.5 Principio di formazione dei tracciati 13

2.1 Schema a blocchi di una DAD FastWave IDS 15
2.2 Campionamento del segnale *A-Scan* in tempo equivalente 15
2.3 Generazione dei trigger per il campionamento del segnale *A-Scan* 16
2.4 Task di acquisizione e Task di trasmissione 19
2.5 Introduzione di uno o più task di elaborazione 19

3.1 Rappresentazione grafica dei dati acquisiti 21
3.2 Elaborazione pre-processing 21
3.3 Generazione di una serie di sweep ogni intervallo spaziale Δx_{ant} 22
3.4 Riduzione del rumore impostando la media di 12 sweep ogni tratto Δx_{ant} . 23
3.5 Processing per la visualizzazione della mappa 24
3.6 Mappa ottenuta con l'applicazione dell'algoritmo di Background Removal (parte significativa) 26
3.7 Andamento tipico della potenza del segnale *A-Scan* 27
3.8 Elaborazione post-processing 29

3.9 Pattern applicati per la migrazione 29
3.10 Risultato grafico dell'operazione di migrazione (dettaglio) 31
3.11 Operazione di Fitting 32
3.12 Mappa ottenuta con l'applicazione dell'algoritmo di migrazione 33
3.13 Forme tipiche di tracciati georadar 33

4.1 Introduzione dei task di elaborazione 34
4.2 Architettura per gli algoritmi *Background Removal* e *Sensitive Time Control* 35
4.3 Schematico - Rete per il calcolo di $S_{CAL_AVG_i}$ 37
4.4 Schematico - Rete per il calcolo di $\overline{P_i}$ 38
4.5 Schematico - Rete per il calcolo di Sp_i 40
4.6 Schematico - Rete per il calcolo di STC_i 41
4.7 Schematico - Rete per il calcolo di S_{STC_i-j} 43
4.8 Flusso realtime 44
4.9 Schematico - Rete globale per gli algoritmi *Background Removal* e *Sensitive Time Control* (prima parte) 45
4.10 Schematico - Rete globale per gli algoritmi *Background Removal* e *Sensitive Time Control* (seconda parte) 46
4.11 Organizzazione della memoria per gli algoritmi *Background Removal* e *Sensitive Time Control* 47
4.12 Architettura per l'algoritmo *Migration* 48
4.13 Rappresentazione dei pattern sul radargramma 50
4.14 Schematico - Blocco *Engine* per l'algoritmo *Migration* 53
4.15 Organizzazione della memoria per l'algoritmo *Migration* 54
4.16 Diagramma di flusso per l'algoritmo *Migration* 56

5.1 Algoritmi di visualizzazione: confronto fra le mappe radar ottenute con il software *K2 Detector* (figura a) e con la simulazione del codice VHDL (figura b) 59
5.2 Algoritmi di focalizzazione: confronto fra le mappe radar ottenute con il software *K2 Detector* (figura a) e con la simulazione del codice VHDL (figura b) 60
5.3 La scheda di valutazione Zedboard presentata durante il workshop del 31/01/2013 61
5.4 Partecipazione al workshop 62

A.1 Procedimento per la simulazione degli algoritmi di visualizzazione 64

Bibliografia

[1] Radar Geoservizi, *GPR - Metodologia e applicazioni*, www.georadar.it/pdf/GPR.pdf

[2] IDS Georadar Division *Detector DUO brochure*, http://www.idscorporation.com

[3] M. Macucci, *Appunti di Misure elettroniche II*, Dispense del corso di Misure elettroniche II - Ingegneria elettronica - Università degli Studi di Pisa - Anno accademico 2003-2004

[4] David J. Daniels, *Ground Penetrating Radar - 2nd Edition*, Edited by David J. Daniels

[5] IDS Georadar Division *Fondamenti del G.P.R.*, Rapporto tecnico interno

[6] Lattice Semiconductor *LatticeXP2 Family Handbook*, http://www.latticesemi.com

[7] Xilinx, Inc. *Zynq-7000 All Programmable SoC Overview*, http://www.xilinx.com/

[8] Mark Zwolinski, *Progetto di sistemi digitali*, Pearson Education