

POLITECNICO DI TORINO

Dipartimento di Elettronica e Telecomunicazioni
Corso di Laurea in Ingegneria Elettronica

Tesi di Laurea Magistrale

Signal Processing Industriale con Metodi non Convenzionali e Sistemi Embedded a Basso Costo



Relatore:

prof. Mario Roberto CASU

Correlatore:

dott. ing. Fabrizio RIENTE

Candidato:

Valerio DE FELICE
matricola: 242945

Referenti aziendali
Centro Ricerche FIAT
ing. Giuseppe D'ANGELO
ing. Gianmarco GENCHI

ANNO ACCADEMICO 2018-2019

Sommario

Il seguente elaborato tratta delle motivazioni, della realizzazione e dei risultati del lavoro di tesi di laurea magistrale, svolto in collaborazione con il Centro Ricerche FIAT di Orbassano, Torino, con lo scopo di implementare un sistema di filtraggio di segnali, in ambito industrial-automotive, utilizzando hardware non convenzionale.

Il sistema di filtraggio è basato sulla tecnica di denoising nota come Singular Spectrum Analysis, la quale fa uso della Singular Value Decomposition (SVD), da cui prende il nome, per rimuovere il rumore ambientale da segnali di varia natura. L'algoritmo riceve in ingresso il segnale monodimensionale da filtrare, generando a partire dai suoi campioni la matrice di covarianza, da cui vengono estratti gli autovalori. Sulla base di questi, l'algoritmo discerne i campioni di rumore, li elimina e ricostruisce il vettore iniziale, restituendolo in una versione, appunto, meno rumorosa.

Tutto ciò è stato precedentemente implementato, in linguaggio C++, su hardware National Instruments specializzato e ad alte prestazioni, il cui costo supera i 6500 €. Esso si compone di una scheda NI-9222 che acquisisce e converte i segnali in digitale, a circa 32,7 kHz su due canali, e di un industrial controller NI IC-3173 che processa i dati in real-time e li visualizza attraverso il programma Labview.

L'obiettivo è quello di sostituire la soluzione precedentemente utilizzata con una di tipo embedded a basso costo, che possa essere collocata direttamente su un braccio robotico.

A tal proposito, si è optato per un sistema che combini una CPU, modesta dal punto di vista dei costi e delle performance, con della logica programmabile (FPGA) che consenta di implementare hardware parallelo, garantendo l'elaborazione dei dati in real-time. Allo stesso tempo, il sistema deve essere completo di ADC, con una risoluzione adatta ad acquisire correttamente i segnali, e di periferiche che permettano di visualizzare ed analizzare i segnali in uscita.

Sono state quindi prese in considerazione diverse tipologie di schede di sviluppo con System-on-Chip (SoC) ZYNQ-7000, prodotto da Xilinx, il quale monta una CPU ARM-Cortex A9 dual-core da 667 MHz, un FPGA con 85000 celle riprogrammabili ed un ADC integrato (XADC), da 12 bit di risoluzione, accessibile mediante

pin-header su tre canali, di cui due con campionamento simultaneo.

Tra le possibili alternative disponibili sul mercato è stata selezionata la board di sviluppo nota come Zedboard, prodotta dalla AVNET. Essa si è rivelata il miglior compromesso tra costo (500 €), maneggevolezza (poco ingombrante) e completezza, in termini di periferiche utili alla realizzazione del sistema. Tra queste si sottolineano la memoria SDRAM DDR3, la memoria Flash QSPI, lo slot per SD card, l'UART e le uscite video HDMI e VGA.

L'algoritmo di denoising è stato implementato sul nuovo sistema dapprima su un singolo core. Tuttavia per poter usufruire dell'acquisizione analogica mediante interrupt, senza rallentare l'esecuzione dell'algoritmo, si è fatto uso di un approccio di tipo *pipeline*, dove un core si occupa dell'acquisizione dei segnali e l'altro della loro elaborazione e visualizzazione in uscita.

A seguito dello studio dei tempi di esecuzione delle varie parti dell'algoritmo di denoising e dell'individuazione dei suoi colli di bottiglia, è stata realizzata l'accelerazione hardware delle parti più onerose dell'algoritmo, effettuando sintesi ad alto livello per tradurre il codice C/C++ delle parti dell'algoritmo da accelerare direttamente in linguaggio HDL (Hardware Description Language). Nel farlo, si è ricorso ad un approccio di tipo divide et impera, al fine di non saturare le risorse hardware a disposizione della logica programmabile, a causa dell'alta quantità di dati da gestire per ogni frame.

A ciò è seguita una revisione della parte software, che ha compreso l'eliminazione di alcune inefficienze e la parallelizzazione del codice, laddove possibile. Questo ha portato a raggiungere l'obiettivo dell'esecuzione in real-time dell'algoritmo, creando anche le condizioni per l'implementazione della parte di visualizzazione, in uscita, dei segnali filtrati e non.

L'uscita video utilizzata per quest'ultimo scopo è la VGA (Video Graphics Array), la quale richiede hardware dedicato per poter proiettare i frame, provenienti dalla CPU, su schermo. Per questo motivo è risultata necessaria l'implementazione in Verilog di un controller VGA, che generasse i segnali destinati ai pin del connettore VGA DE-15, tra cui si distinguono quelli dedicati ai colori (RGB) e altri destinati alla sincronizzazione verticale ed orizzontale delle immagini. La visualizzazione dei segnali, sia di ingresso che di uscita, è stata realizzata sul modello di un moderno oscilloscopio digitale, creando uno sfondo con griglia su cui proiettare i segnali, ma garantendo allo stesso tempo la lettura delle informazioni che li riguardano (V/div, s/div e canale).

Per rendere il sistema più agevole da utilizzare, nell'ambito per cui è destinato, è stata realizzata un'immagine BOOT contenente l'hardware per la logica programmabile (bitstream) e le applicazioni di entrambi i core. L'immagine è stata scritta sia su SD Card che su memoria Flash, consentendo al sistema di avviarsi autonomamente (in BOOT mode) ad ogni accensione, riavvio o reset, senza la necessità di PC e cavi USB.

I tool di sviluppo utilizzati fanno tutti parte della Vivado Design Suite, fornita da Xilinx. In particolare, è stato utilizzato Vivado per creare, sintetizzare ed implementare l'hardware destinato alla logica programmabile; Vivado HLS (High Level Synthesis) per effettuare la sintesi ad alto livello in merito all'accelerazione hardware; Xilinx SDK (Software Development Kit) per scrivere ed eseguire le applicazioni riservate al processore ARM.

Infine, sono stati effettuati dei test sperimentali mediante l'utilizzo di un generatore di funzione **Rigol DG1022**, esaminando il comportamento e la robustezza del sistema attraverso diverse tipologie di segnali. Nello specifico, è stato misurato quantitativamente il filtraggio operato dall'algoritmo di denoising, nonché gli errori introdotti dalla conversione analogico-digitale, ed infine, l'eventuale presenza di fenomeni di crosstalk tra i vari canali. I dati sperimentali sono stati raccolti mediante SD card, per poi essere analizzati con **Matlab**.

Indice

Sommario	III
I Premesse	1
1 Introduzione	3
1.1 Obiettivo	3
1.2 Singular Spectrum Analysis	4
1.3 Specifiche	6
2 Hardware e tool di sviluppo	9
2.1 La famiglia ZYNQ-7000 SoC	10
2.1.1 L'XADC	11
2.2 Selezione dell'hardware	12
2.3 La Zedboard	13
2.4 Tool di sviluppo	14
2.4.1 Vivado	15
2.4.2 Xilinx SDK	15
2.4.3 Vivado HLS	16
II Realizzazione	19
3 Sviluppo software	21
3.1 Implementazione dell'algoritmo (fornito)	22
3.2 Acquisizione dei segnali	26
3.2.1 Acquisizione con interrupt	28
3.3 Versione dual-core	30
3.3.1 Ping-pong buffers	31

4	Accelerazione Hardware	35
4.1	Colli di bottiglia	35
4.2	Implementazione dell'hardware parallelo	36
4.2.1	Scomposizione dell'algoritmo: divide et impera	37
4.2.2	Il protocollo AXI4	40
4.2.3	Sintesi ad alto livello ed RTL	42
4.3	Integrazione dell'hardware parallelo	46
4.3.1	AXI DMA	46
4.3.2	Comunicazione tra PS e PL	47
4.3.3	Invocazione dell'hardware in SDK	50
4.3.4	Altre migliorie	52
4.3.5	Programmabilità della lunghezza della finestra	53
5	Visualizzazione	55
5.1	VGA	55
5.1.1	Controller VGA	58
5.2	AXI VDMA ed integrazione	62
5.3	Impostazioni di visualizzazione	65
III	Risultati sperimentali	73
6	Analisi quantitativa del filtraggio	75
6.1	Lunghezza della finestra variabile	79
7	Analisi dell'errore dell'XADC	87
7.1	Errori di linearità, offset e guadagno	87
7.2	Crosstalk	91
8	Immagine BOOT	95
9	Considerazioni finali	97
	Appendice	101
A	Codice SDK	101
A.1	Cpu 1	101
A.1.1	main.cc	101
A.2	Cpu 2	105
A.2.1	main.cc	105
A.2.2	myfilterHW.h	109
A.2.3	myfilterHW.cc	110

A.2.4	dma_interrupt.h	121
A.2.5	dma_interrupt.c	122
A.2.6	vga.h	125
A.2.7	vga.cc	126
B	Codice HLS	131
B.1	myfilterHLS.h	131
B.2	myfilterHLS.cc	131
C	Controller VGA	133
C.1	vga.v	133
	Bibliografia	137

Elenco delle figure

1.1	Flow chart dell'algoritmo di SSA [2].	4
1.2	Scheda di acquisizione NI-9222 (sulla sinistra) e industrial controller NI IC-3173 (sulla destra).	6
2.1	Schema a blocchi del SoC ZYNQ-7000 [5].	10
2.2	Schema a blocchi dell'XADC [6].	11
2.3	Facciata superiore della Zedboard.	14
2.4	Schermata principale di Vivado, in cui si distingue il block diagram (sulla destra) con degli IP interconnessi tra loro.	16
2.5	Flow chart del processo di sintesi ad alto livello.	17
3.1	Block diagram di Vivado con l'IP del Processing System della ZYNQ.	21
3.2	Finestra da cui è possibile modificare le impostazioni del PS.	22
3.3	Tempi di esecuzione delle varie parti dell'algoritmo.	26
3.4	Block diagram di Vivado con l'XADC Wizard collegato all'IP del PS.	26
3.5	Schematico del pin-header della Zedboard con i pin utilizzati in evidenza [9].	27
3.6	Pin dell'XADC Header della Zedboard utilizzati nell'acquisizione dei due segnali.	28
3.7	Suddivisione dei task tra i due core.	30
3.8	Principio di funzionamento dei ping-pong buffers.	31
4.1	Suddivisione dei task tra PS e PL nella prima parte dell'algoritmo.	37
4.2	Transazione di lettura con il protocollo AXI4.	40
4.3	Transazione di scrittura con il protocollo AXI4.	41
4.4	Interfacce assegnate agli ingressi ed alle uscite del futuro IP [8].	42
4.5	Stima post-sintesi C della latenza (in ns) dei vari loop nell'algoritmo.	44
4.6	Stima post-sintesi C del periodo minimo di clock e della latenza (a sinistra) e delle risorse hardware utilizzate (a destra).	45
4.7	Stima post-sintesi RTL delle risorse della PL e del periodo minimo di clock.	45
4.8	Schema a blocchi dell'AXI DMA IP Core [11].	47
4.9	Schema a blocchi della comunicazione PS-PL [12].	48

4.10	Block diagram di Vivado con PS, XADC Wizard e l'hardware parallelo.	49
4.11	Tempi di esecuzione delle varie parti dell'algoritmo dopo l'accelerazione hardware e le migliorie software.	52
5.1	Pin del connettore DE-15 VGA.	56
5.2	Temporizzazione orizzontale e verticale.	57
5.3	Caratteristiche ideali della risoluzione standard 640×480 a 60 Hz.	58
5.4	IP Vivado del controller VGA.	59
5.5	Block diagram di Vivado con gli IP aggiunti per la visualizzazione VGA.	63
5.6	Block diagram di Vivado completo di PS, XADC Wizard, accelerazione hardware e visualizzazione VGA.	64
5.7	Codici in esadecimale di alcuni degli oltre 16 milioni di colori possibili.	66
5.8	Grafica vettoriale adottata.	68
5.9	Principio di funzionamento della proiezione dei segnali con VGA.	69
5.10	Esempio di visualizzazione VGA con mV/div diversi tra i due canali.	70
6.1	Segnale in ingresso (blu) e la sua versione filtrata (rosso).	76
6.2	Potenza spettrale del segnale in ingresso (blu) e del segnale filtrato (rosso).	76
6.3	Zoom della potenza spettrale del segnale in ingresso (blu) e del segnale filtrato (rosso).	77
6.4	Segnale sinusoidale rumoroso in ingresso.	77
6.5	Segnale sinusoidale rumoroso filtrato.	78
6.6	Potenza spettrale del segnale in ingresso (blu) e del segnale filtrato (rosso).	78
6.7	Zoom della potenza spettrale del segnale in ingresso (blu) e del segnale filtrato (rosso).	79
6.8	Denoising per $L = 21$.	80
6.9	Denoising per $L = 21$ nel dominio della frequenza.	80
6.10	Denoising per $L = 18$.	81
6.11	Denoising per $L = 18$ nel dominio della frequenza.	81
6.12	Denoising per $L = 15$.	82
6.13	Denoising per $L = 15$ nel dominio della frequenza.	82
6.14	Denoising per $L = 12$.	83
6.15	Denoising per $L = 12$ nel dominio della frequenza.	83
6.16	Denoising per $L = 9$.	84
6.17	Denoising per $L = 9$ nel dominio della frequenza.	84
6.18	Ultimi campioni del segnale filtrato.	85
7.1	Segnale acquisito (rosso) e segnale ideale (blu).	88
7.2	Retta che meglio approssima la caratteristica del segnale acquisito (rosso) e caratteristica ideale (blu).	88
7.3	Definizione di DNL.	89

7.4	Definizione di INL.	90
7.5	Andamento del DNL (in alto) e dell'INL (in basso) espressi in LSB.	90
7.6	Segnale ad onda quadra di prova in ingresso.	91
7.7	Segnale rumoroso di prova in ingresso.	92
7.8	Acquisizioni sul canale flottante con l'altro canale flottante (blu), con onda quadra (rosso) e con segnale rumoroso (verde).	92
8.1	Jumpers JTAG di configurazione.	95

Parte I

Premesse

Capitolo 1

Introduzione

Oggi giorno l'uso di soluzioni embedded a basso costo e general purpose, affiancato a coprocessori o hardware parallelo, è sempre più diffuso in quegli ambiti dove la capacità elaborativa non è prioritaria, a discapito delle soluzioni specializzate ad alto costo e poco flessibili.

Questa tipologia, oltre all'abbattimento dei costi permette una certa flessibilità, utile laddove si voglia modificare, arricchire o rimodulare il sistema con nuove funzionalità venendo incontro ad altre specifiche.

Non da poco, inoltre, sono i vantaggi relativi al risparmio energetico e alla grande riduzione di spazio occupato. L'hardware embedded, infatti, è perlopiù costituito da board di modeste dimensioni che richiedono alimentatori altrettanto modesti, sia in termini di consumo di potenza che di dimensioni.

1.1 Obiettivo

I vantaggi appena esposti vanno a costituire quelle che sono le motivazioni principali di questo lavoro di tesi.

In particolare si vuole implementare un sistema di filtraggio a basso costo di segnali prevalentemente luminosi, o comunque in ambito industriale, che vada a sostituire hardware già funzionante ad alte prestazioni. I segnali luminosi vengono rilevati da dei fotodiodi disposti in varie angolazioni, in modo tale da raccogliere e processare le saldature laser effettuate da un braccio robotico sulla carrozzeria di veicoli in fabbricazione. Il sistema deve essere quindi abbastanza compatto da poter essere montato sullo stesso braccio robotico.

L'algoritmo di filtraggio viene fornito e fa uso del metodo noto come *Singular Spectrum Analysis* (SSA) per rimuovere le componenti rumorose dei segnali intercettati dai fotodiodi. Esso viene brevemente descritto nel seguente paragrafo.

1.2 Singular Spectrum Analysis

La Singular Spectrum Analysis (SSA) [1] è una tecnica di rimozione del rumore ambientale adatta ad una grande varietà di segnali, non solo luminosi e non necessariamente periodici. Essa consta di alcuni passaggi (fig. 1.1) tra cui la Singular Value Decomposition (SVD), da cui prende il nome.

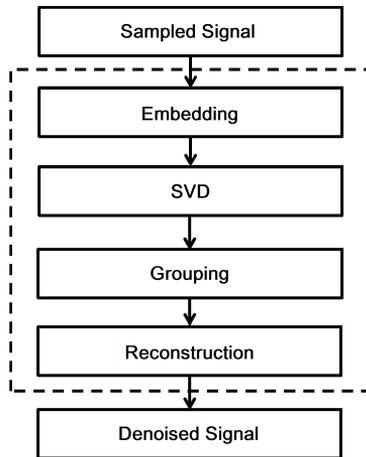


Figura 1.1. Flow chart dell'algoritmo di SSA [2].

- *Embedding*: Si consideri un segnale s monodimensionale di lunghezza N , $s = \{s_n, n = 1 \dots N\}$. Questa parte consiste nel mappare il segnale 1D, composto da N campioni, con una sequenza di $K = N - L + 1$ vettori colonna di lunghezza L . Il risultato è la matrice traiettoria X , che può essere espressa come:

$$X = \begin{pmatrix} s_1 & s_2 & \dots & s_K \\ s_2 & s_3 & \dots & s_{K+1} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ s_L & s_{L+1} & \dots & s_N \end{pmatrix}$$

La matrice traiettoria X di dimensioni $L \times K$ è una matrice di Hankel, ovvero le diagonali con pendenza positiva risultano costanti. L è anche nota come lunghezza della finestra ed è un parametro intero che oscilla tra $[2, N - 1]$. Essa è l'unico parametro in questo algoritmo e va ad influire direttamente sulla qualità del filtraggio, quindi va scelto con cura.

- *SVD*: La Singular Value Decomposition o decomposizione a valori singolari viene eseguita sulla matrice traiettoria X , calcolando prima la matrice di covarianza $S = XX^T$ ed estraendo da essa gli autovalori:

$$S = U\Sigma U^T \quad (1.1)$$

dove $\Sigma = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_L)$ e λ_i sono gli autovalori della matrice S , mentre U è la matrice unitaria. In questo modo la matrice traiettoria X viene decomposta in L matrici elementari:

$$X = \sum_{i=1}^L X_i \quad \text{con} \quad X_i = \sqrt{\lambda_i} u_i v_i^T \quad (1.2)$$

dove u_i sono gli autovettori della matrice S e $v_i = X^T \frac{u_i}{\sqrt{\lambda_i}}$.

- *Grouping*: Questo step consiste nel suddividere la serie di matrici elementari X_i , ottenute dalla decomposizione, in r gruppi I_k disgiunti tra loro, sommando in ognuno di essi proprio le matrici elementari.

$$X_{I_k} = \sum_{i \in I_k} X_i \quad (1.3)$$

Ogni gruppo viene formato utilizzando le informazioni contenute nel vettore dei valori singolari, permettendo di discernere tra campioni di rumore e campioni di segnale. In questo modo la matrice traiettoria può essere espressa come somma di questi gruppi.

$$X = \sum_{k=1}^r X_{I_k} \quad (1.4)$$

- *Ricostruzione*: La ricostruzione è una sorta di "media diagonale" che serve ad eseguire il processo inverso dell'embedding: trasformare la matrice in una sequenza temporale di campioni (1D). Le matrici elementari sono ancora matrici di Hankel e gli elementi di ognuna delle loro diagonali rappresentano un singolo campione.

$$x_n^{(k)} = \begin{cases} \frac{1}{n} \sum_{m=1}^n x_{m, n-m+1} & \text{for } 1 \leq n < L \\ \frac{1}{L} \sum_{m=1}^L x_{m, n-m+1} & \text{for } L \leq n < K \\ \frac{1}{N-n+1} \sum_{m=n-K+1}^L x_{m, n-m+1} & \text{for } K+1 \leq n \leq N \end{cases} \quad (1.5)$$

Eseguendo la media lungo una diagonale di una matrice elementare X_{I_k} si ottiene un campione temporale del vettore $x^{(k)} = \{x_n^{(k)}, n = 1 \dots N\}$. Quindi, usando le equazioni 1.5 viene ricostruito il vettore corrispondente alla versione filtrata del segnale di ingresso.

1.3 Specifiche

L'algoritmo era stato implementato prima di questo lavoro di tesi in linguaggio C++ su hardware **National Instruments** ad alte prestazioni.

Nello specifico, i segnali che vengono acquisiti provengono da due fotodiodi la cui tensione di uscita oscilla nel range $0 \div 1$ V. Questi segnali vengono campionati ciascuno ad una frequenza di 32,768 kHz utilizzando una scheda di acquisizione dati NI-9222 [3]. Essa monta un ADC per ogni canale, con risoluzione a 16 bit ed una frequenza di campionamento massima di 500 kS/s. Il costo della scheda di acquisizione si aggira intorno ai 1300 €.

Una volta acquisiti e convertiti, i segnali vengono elaborati da un industrial controller NI IC-3173 [4], il quale monta un processore Intel Core i7 dual-core da 2.2 GHz. Questa tipologia di unità elaborativa costa almeno 5400 €, nella sua versione più economica, ma permette di eseguire l'algoritmo su ogni gruppo da 32768 campioni (1 secondo di acquisizione) in circa 10/15 ms per canale, con l'ausilio del programma LabView che ne permette anche la visualizzazione.



Figura 1.2. Scheda di acquisizione NI-9222 (sulla sinistra) e industrial controller NI IC-3173 (sulla destra).

In sintesi le specifiche dell'hardware utilizzato prima di questo lavoro di tesi sono:

- Numero di ADC: 2;

- Risoluzione ADC: 16 bit;
- Frequenza di campionamento: 32,768 kHz;
- Tempo di esecuzione dell'algoritmo: 10/15 ms per canale;
- Costo: più di 6500 €.

Capitolo 2

Hardware e tool di sviluppo

Come già accennato, lo scopo è quello di rimpiazzare l'hardware attualmente in uso con una soluzione compatta, a basso costo e low-power.

L'esecuzione di un algoritmo come quello della SSA, comunque abbastanza oneroso in termini di complessità computazionale, in tempi relativamente stretti (bisogna terminare l'elaborazione di un "frame" di 32768 campioni, corrispondenti ad 1 secondo di acquisizione, prima di passare al frame successivo), obbliga a cercare un sistema con un processore abbastanza potente. Tuttavia questo farebbe lievitare i costi, oltre a causare un incremento del consumo di potenza e conseguentemente anche delle dimensioni del sistema stesso (alimentatori ingombranti, ventole e dissipatori). Inoltre la capacità computazionale è richiesta essenzialmente per il solo algoritmo.

Per questi motivi si può optare per soluzioni con CPU più modeste ma che affianchino ad esse delle unità di logica programmabile, FPGA, in maniera tale da accelerare l'esecuzione dell'algoritmo, eseguendolo direttamente in hardware parallelo, laddove è possibile.

Oltre ad elaborare i segnali, il sistema deve essere anche in grado di acquisirli correttamente, va quindi scelta una soluzione che monti uno o più ADC, con almeno 12 bit di risoluzione, in grado di campionare almeno alla stessa frequenza citata nel par. 1.3.

È inoltre utile la presenza di eventuali periferiche utili sia al debug che alla visualizzazione dei risultati, come UART, SD card e VGA.

Infine, anche i tools di sviluppo sono determinanti nella scelta della soluzione hardware appropriata, sia per quanto concerne lo sviluppo software che per la sintesi e l'implementazione dell'hardware parallelo su FPGA.

2.1 La famiglia ZYNQ-7000 SoC

Tra le varie opzioni disponibili sul mercato, la famiglia di System-on-Chip (SoC) ZYNQ-7000 [5], prodotta da Xilinx, risulta essere quella più completa ed adatta al caso preso sotto esame.

Essa presenta una parte di Processing System (PS o CPU), con performance relativamente modeste, combinata ad una parte di Programmable Logic (PL o FPGA) che si interfacciano tra loro con bus AXI ad alte prestazioni e bus AXI General-Purpose. IL PS è connesso al mondo esterno con svariati tipi di interfacce, tra cui SPI, I2C, CAN, UART e GPIO.

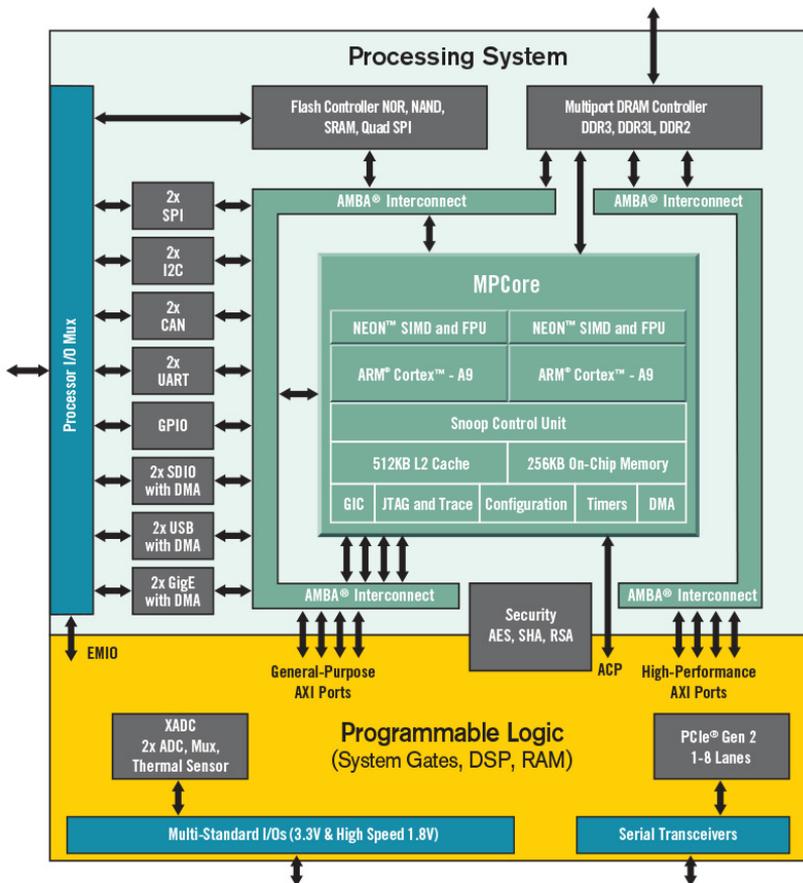


Figura 2.1. Schema a blocchi del SoC ZYNQ-7000 [5].

Nella versione scelta, la XC7Z020, il SoC presenta un processore (PS) ARM Cortex dual-core da 667 MHz mentre la logica programmabile (PL) è costituita da 85000 celle. Nella parte di PS è integrata una On Chip Memory (OCM) di tipo RAM da 256 KB, condivisa dai due core, utile per piccole operazioni di *Inter Process Communication* ad alta velocità.

2.1.1 L'XADC

Ogni tipologia di SoC della famiglia ZYNQ-7000 monta due ADC integrati (XADC) da 12 bit di risoluzione e sample rate massimo di 1 MS/s [6].

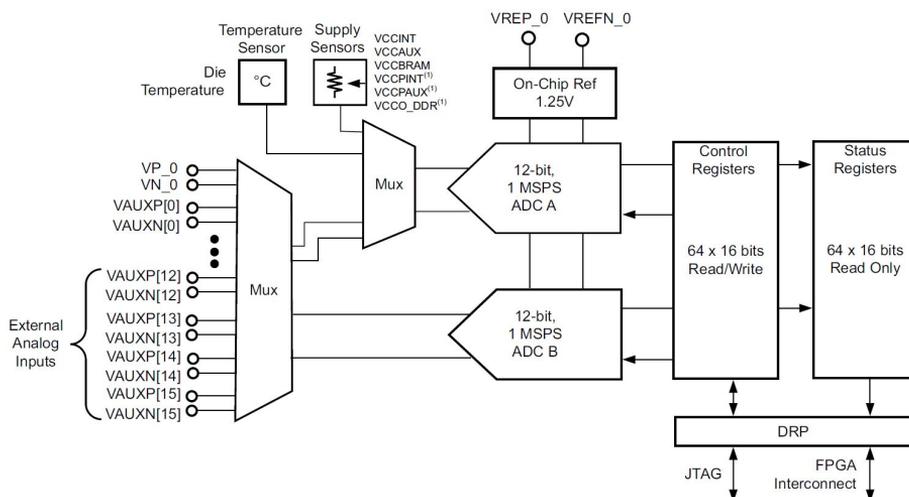


Figura 2.2. Schema a blocchi dell'XADC [6].

I due ADC presentano ingressi di tipo differenziale con range di tensione 0 ÷ 1.25V. Il primo ADC (ADC A) è connesso mediante un MUX a due sensori interni, uno di temperatura e l'altro di tensione, ad un canale analogico dedicato (VP/VN) e ai primi 8 ingressi analogici ausiliari (da VAUXP[0]/VAUXN[0] a VAUXP[7]/VAUXN[7]), mentre l'ADC B è connesso ai restanti 8 ingressi ausiliari (da VAUXP[8]/VAUXN[8] a VAUXP[15]/VAUXN[15]), per un totale di 16 canali analogici ausiliari più uno dedicato.

Tra le varie modalità in cui l'XADC può operare quella *simultaneous sampling* (campionamento simultaneo) risulta essere la più adatta nel caso in cui si vogliono campionare due segnali simultaneamente. Infatti, in questa modalità i due ADC

lavorano indipendentemente l'uno dall'altro. Quindi, abilitando un solo ingresso analogico ausiliare per ADC si ottiene un campionamento simultaneo su due canali.

2.2 Selezione dell'hardware

Esistono diverse soluzioni hardware sul mercato che utilizzano il SoC ZYNQ-7000 XC7Z020, anche molto diverse tra loro. Per scegliere quella più consona sono state confrontate alcune caratteristiche chiave per questo tipo di applicazione, tra cui: costo, ingombro, memorie aggiuntive, periferiche video, possibilità di espansioni, tipo di connettore dell'XADC, eventuali altri ADC on-board (non integrati nel SoC) e tool di sviluppo.

L'obiettivo è selezionare la soluzione più completa possibile, ovvero quella che permette di acquisire i segnali, processarli e visualizzarli senza la necessità di altre espansioni. Questo deve verificarsi con un costo non eccessivo ed un ingombro minimo.

Le opzioni analizzate e le loro caratteristiche sono sintetizzate nella tabella 2.1.

Caratteristiche	Xilinx ZC702	Zedboard	ZYBO Z7-20
Costo	900 €	500 €	300 €
Ingombro	Board	Board	Board
Slot SD Card	Si	Si	Si
Uscite Video	HDMI	HDMI, VGA	HDMI
Espansioni	2xFMC	1xFMC, 6xPMOD	4xPMOD
Accesso XADC	PIN-Header	PIN-Header	PMOD
ADC dedicato	No	No	No
Tool di sviluppo	Vivado	Vivado	Vivado
Caratteristiche	Microzed	NI sbRIO-9627	NI cRIO-9063
Costo	210+60 €	1000 €	1000+600 €
Ingombro	Board compatta	Board	Controller
Slot SD Card	Si	Si	No
Uscite Video	No	No	VGA
Espansioni	No	No	4xSchede NI
Accesso XADC	Breakout (a parte)	MIO	Non accessibile
ADC dedicato	No	Si (16 bit)	No
Tool di sviluppo	Vivado	LabView	LabView

Tabella 2.1. Caratteristiche hardware delle soluzioni ZYNQ-7000 XC7Z020.

Tra queste possibili opzioni la scelta è ricaduta sulla Zedboard (par. 2.3), la

quale rappresenta il miglior compromesso in termini di caratteristiche elencate precedentemente. Ragionando per esclusione, sono state subito scartate le soluzioni National Instruments (NI), sia per il costo elevato che per il tool di sviluppo limitante. La Microzed, seppur molto compatta, non presenta uscite video e richiede un modulo di breakout esterno per accedere all'XADC. La ZYBO, invece, fa uso di un connettore PMOD per accedere all'XADC, ovvero uno standard Digilent poco diffuso sul mercato che rende l'acquisizione analogica poco agevole. Quindi dovendo scegliere tra le rimanenti Zedboard e Xilinx ZC702, boards molto simili dal punto di vista delle caratteristiche, si è optato per la prima avendo essa un costo decisamente inferiore alla soluzione prodotta da Xilinx.

2.3 La Zedboard

Una delle evaluation board, con SoC ZYNQ-7000, più note ed utilizzate nel mondo dell'hardware embedded è senza dubbio la **Zedboard**, prodotta dalla AVNET [7].

La Zedboard monta la versione XC7Z020 della famiglia ZYNQ-7000 SoC, che permette, come citato nel paragrafo 2.1, di affiancare alla CPU (PS), general-purpose, hardware parallelo implementato su FPGA, caratteristica preziosa per garantire l'esecuzione dell'algoritmo di denoising in real-time.

Alla ZYNQ, la board integra una SDRAM DDR3 da 512 MB, utile per memorizzare la grande mole di dati acquisiti, una memoria Flash QSPI da 256 Mb su cui poter scrivere immagini BOOT, e la possibilità di inserire una memoria secondaria, di massa, attraverso uno slot per SD card, molto pratica sia per il debug che per eventuali immagini BOOT.

L'XADC è accessibile mediante un pin-header che permette di collegare dei segnali in ingresso a tre dei diciassette ingressi analogici, tra cui il canale dedicato (VP/VN) e i due ingressi ausiliari (VAUX0P/VAUX0N e VAUX8P/VAUX8N), uno per ognuno dei due ADC (par. 2.1.1). Ciò consentirà di acquisire i due segnali simultaneamente, senza delay dovuti a multiplexer in ingresso.

Oltre all'acquisizione analogica, la Zedboard integra diverse interfacce e periferiche che aprono a svariate possibilità di progetto.

L'UART con USB 2.0 rende possibile la ricezione e la trasmissione di dati da terminale, caratteristica molto utile per il debug e per visionare lo stato di esecuzione delle applicazioni. Sono presenti interfacce video come la VGA e l'HDMI, mentre per le periferiche audio vi sono sia linee d'entrata (per microfoni o sensori di suono) che di uscita (per casse o cuffie).

Volendo ampliare le features della board si possono utilizzare le 5 porte PMOD, l'ingresso USB 2.0 OTG e il connettore FMC che consente di collegare periferiche direttamente alla logica programmabile. Questa possibilità di espansione può rivelarsi utile qualora si vogliono migliorare le caratteristiche di acquisizione analogica.

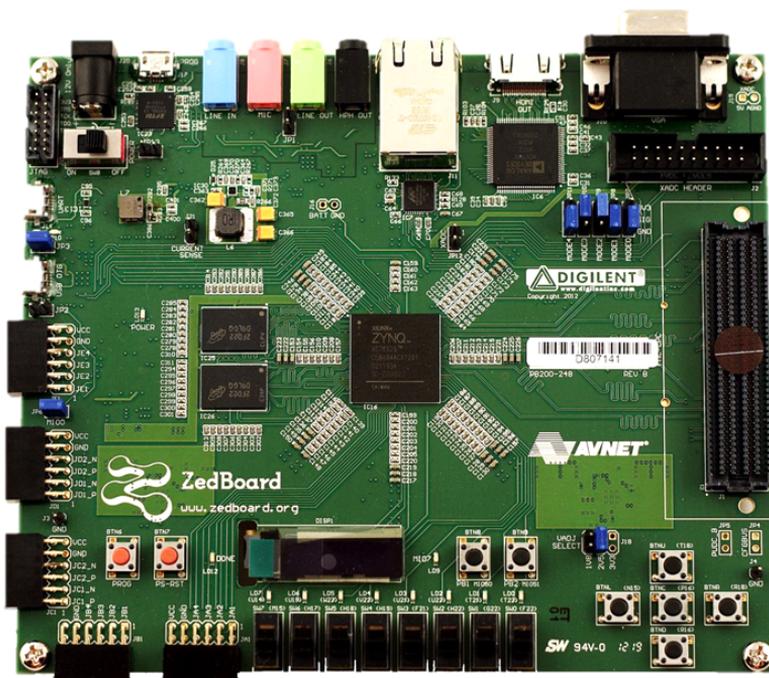


Figura 2.3. Facciata superiore della Zedboard.

Sulla board sono presenti tasti, switch e LED che permettono di interagire attivamente con il sistema.

Il SoC può essere programmato sia con cavo USB 2.0 che, in modalità BOOT, con SD card e memoria Flash QSPI, a seconda della configurazione dei jumpers JTAG. Infine, la board è alimentata mediante un alimentatore da 12 V.

Tutto ciò, alimentatore a parte, è saldato su una board che sul lato più lungo misura appena 16 cm, quindi facilmente collocabile su un braccio robotico.

2.4 Tool di sviluppo

Per la programmazione del PS e della PL, Xilinx mette a disposizione un ecosistema di tools di sviluppo, dove ogni tool risulta comunicante con gli altri ed è specializzato per un singolo scopo. Essi rientrano tutti nella Vivado Design Suite, in particolare si fa uso della versione Webpack Edition 2018.3.

2.4.1 Vivado

Il software principale della suite è **Vivado**. Esso, attraverso un'interfaccia grafica semplice ed intuitiva, permette lo sviluppo di progetti in HDL (*Hardware Description Language*).

L'ambiente di sviluppo si basa sugli IP (*Intellectual Property*), ovvero entità in HDL che possono essere create da zero in Verilog o VHDL, scelti dalla libreria di default di Vivado oppure acquistati ed importati da cataloghi online. Ogni IP precedentemente prodotto può essere personalizzato, a seconda dell'applicazione, modificandone parametri ed impostazioni attraverso un'apposita interfaccia grafica. Quando invece si posseggono i diritti dell'IP è possibile variare direttamente il codice HDL.

Per progetti più complessi si può far uso del *block diagram*, dove è possibile inserire e interconnettere tra di loro tutti gli IP graficamente. Lo stesso PS della ZYNQ può essere inserito, modificato ed interconnesso come un qualunque IP.

Una volta validato il design, Vivado permette di simularlo, attraverso un simulatore interno o uno esterno (come ModelSim), di sintetizzarlo e di implementarlo (*place and route* compresi). In quest'ultimo passo, Vivado effettua il DRC (*Design Rule Checking*) e le verifiche sul timing.

Infine, quando il design è stato correttamente implementato, si può procedere alla generazione del *bitstream* e successivamente alla programmazione dell'hardware. Se si vuole programmare anche il PS, l'hardware (il bitstream) può essere esportato direttamente in **Xilinx SDK** per produrre applicazioni basate su di esso.

2.4.2 Xilinx SDK

Xilinx SDK (Software Development Kit) è un IDE (*Integrated Design Environment*), basato su **Eclipse**, che permette di creare applicazioni embedded per tutti i SoC della famiglia ZYNQ.

SDK fornisce tutti gli strumenti per lo sviluppo software delle applicazioni: librerie, driver dei vari dispositivi, editor, compilatore, builder, debugger, linea di comando per effettuare *scripting* e anche un terminale interno.

Si possono sviluppare applicazioni C o C++ basate su **Linux** oppure di tipo *stand-alone*, quindi in bare-metal, senza l'ausilio di un sistema operativo.

Le applicazioni possono avere come target uno dei core (nel caso bare-metal), qualora ce ne sia più di uno, oppure il microprocessore appartenente alla categoria dei *soft-core* **MicroBlaze**.

L'hardware (bitstream) viene importato direttamente da Vivado in fase di inizializzazione dell'applicazione, evitando allo sviluppatore di andare a modificare manualmente il contenuto di registri di configurazione del PS, cosa comunque possibile qualora lo si voglia fare, anche grazie alle macro fornite dalla librerie.

SDK mette a disposizione degli sviluppatori anche strumenti per il profiling del

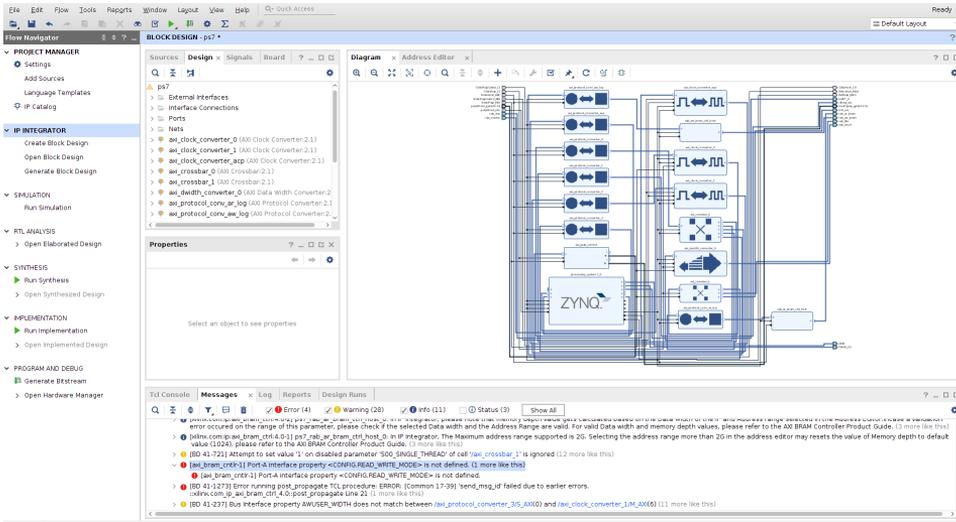


Figura 2.4. Schermata principale di Vivado, in cui si distingue il block diagram (sulla destra) con degli IP interconnessi tra loro.

software, nonchè un sistema di analisi ed ottimizzazione delle performance. Infine è possibile creare immagini BOOT da inserire nella Flash o nella SD card grazie al tool Bootgen.

2.4.3 Vivado HLS

Quando gli algoritmi e gli IP diventano troppo complessi da sviluppare in HDL, risulta particolarmente vantaggioso effettuare sintesi ad alto livello.

Con Vivado HLS è possibile tradurre agevolmente algoritmi in C o C++ direttamente in HDL (Verilog o VHDL), accelerando di molto la creazione di IP da integrare in Vivado.

Il programma permette di simulare la funzione da sintetizzare prima in linguaggio C/C++ attraverso l'uso di un test bench implementato nello stesso linguaggio, dopodiché, una volta impostati il periodo di clock minimo, la *top function* dell'algoritmo e il linguaggio target (Verilog o VHDL), la sintesi può essere eseguita.

Chiaramente l'algoritmo ad alto livello deve soddisfare alcuni requisiti per essere sintetizzabile. La funzione C infatti deve contenere l'intera funzionalità dell'algoritmo, senza l'ausilio di *system calls* che richiedano l'intervento del sistema operativo. Inoltre tutti i costrutti in linguaggio C devono avere dimensioni definite e limitate a tempo di compilazione. Ciò significa che tutte le variabili, i loop e le strutture di

dati utilizzate nel codice sorgente devono avere dimensioni note a priori, dato che l'hardware su FPGA, una volta implementato, è fisso e non può variare a tempo di esecuzione. Non sono quindi sintetizzabili quei costrutti che fanno uso di allocazione/deallocazione dinamica della memoria, come ad esempio i puntatori. Questi ultimi possono essere definiti solo per gli ingressi o le uscite della funzione.

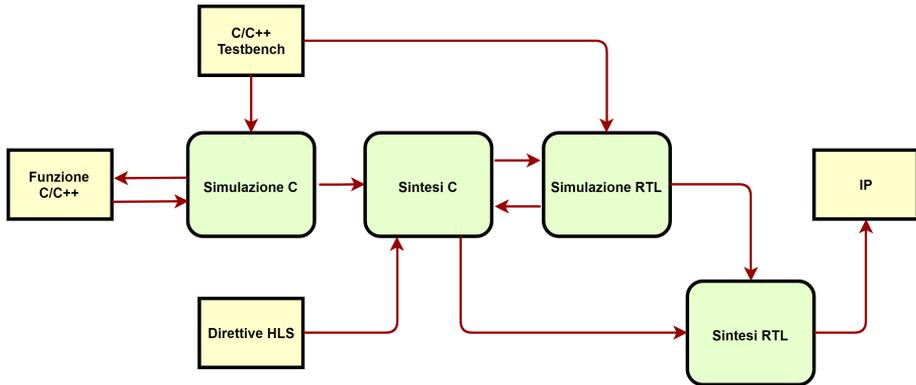


Figura 2.5. Flow chart del processo di sintesi ad alto livello.

La guida del software [8] fornisce alcuni suggerimenti sullo stile di scrittura dell'algoritmo in maniera tale da ottimizzarlo per la fase di sintesi, oltre che renderlo semplicemente sintetizzabile. Questi suggerimenti prevedono anche delle direttive (`#pragma HLS`), da inserire nell'algoritmo, che saranno poi utilizzate in fase di sintesi. Esse possono avere diversi scopi, tra cui: definire il tipo di interfaccia di ingressi ed uscite, applicare tecniche di pipelining, loop unrolling, merging o altro ancora sui loop della funzione, indicare esplicitamente l'uso e la quantità di una particolare risorsa hardware, impostare la latenza massima e così via.

Una volta che la sintesi è andata a buon fine, Vivado HLS mostra delle stime sul timing (periodo di clock minimo e latenza), sulle risorse hardware utilizzate (DSP, LUTs, Flip Flops, BlockRAM) e fa un resoconto dei segnali di ingresso e di uscita con le relative interfacce.

Terminata la fase di sintesi si può ulteriormente simulare l'algoritmo, stavolta in RTL, utilizzando lo stesso test bench della prima fase, o si può procedere direttamente alla fase di esportazione dell'IP, dove viene eseguita una sintesi RTL che fornisce una stima più precisa del timing e delle risorse utilizzate.

Parte II

Realizzazione

Capitolo 3

Sviluppo software

Per poter eseguire l'algoritmo sulla Zedboard ed in generale sul *Processing System* del SoC ZYNQ, bisogna importare l'algoritmo implementato su hardware **National Instruments** (par. 1.3) in Xilinx SDK.

A tal proposito, il primo passo è quello di aggiungere ed impostare l'IP del PS in Vivado.

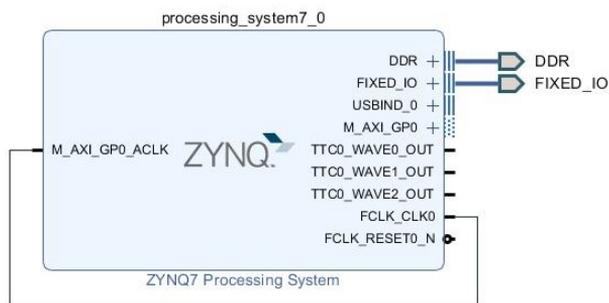


Figura 3.1. Block diagram di Vivado con l'IP del Processing System della ZYNQ.

Nella figura 3.1 è possibile notare alcune delle interfacce abilitate di default nel PS, tra cui DDR e FIXED_IO che hanno dei pin dedicati, essendo esse connesse con le periferiche presenti sulla Zedboard.

Tutte le impostazioni del PS possono essere modificate da un'apposita finestra (figura 3.2), in maniera tale da avere già l'hardware correttamente inizializzato prima di andare ad implementare qualsiasi applicazione. Tra queste impostazioni

si distinguono la frequenza di clock, la comunicazione con la logica programmabile (PL), l'abilitazione di interrupt, la configurazione dei pin delle periferiche ed anche la gestione della memoria DDR.

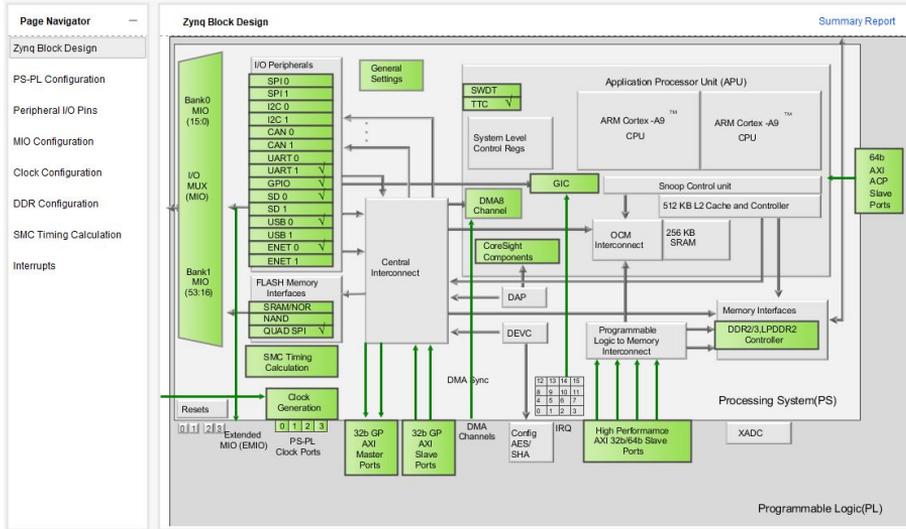


Figura 3.2. Finestra da cui è possibile modificare le impostazioni del PS.

In questa fase di configurazione del PS, Vivado permette di scegliere tra le possibili soluzioni che fanno uso di hardware *Xilinx*, fornendo quindi anche una configurazione di default per la Zedboard.

Una volta effettuate validazione, sintesi ed implementazione si può quindi procedere all'esportazione dell'hardware, con il relativo bitstream, in SDK.

3.1 Implementazione dell'algoritmo (fornito)

Come già detto nel capitolo 1, l'algoritmo è basato sul metodo di rimozione del rumore SSA (par. 1.2) a sua volta basato sulla decomposizione a valori singolari (SVD). Esso viene fornito in linguaggio C++ e fa uso della libreria *Eigen*.

Eigen è una libreria C++, ad alto livello ed open-source che contiene le funzioni matematiche, con i relativi header, utili nell'algebra lineare, nelle operazioni tra vettori e tra matrici, nelle trasformazioni geometriche ed in generale negli algoritmi numerici.

L'algoritmo di filtraggio lavora su 32768 (N) campioni alla volta corrispondenti ad un secondo di acquisizione che vengono poi restituiti in una versione meno

rumorosa di quella iniziale. Il parametro intero noto come *lunghezza della finestra* (L) è invece pari a 21, scelta che garantisce un filtraggio soddisfacente. Ne consegue che il parametro $K = N - L + 1$ è pari a 32748.

Questi campioni vengono immagazzinati in un buffer per essere poi inseriti nella *matrice traiettoria* X le cui dimensioni saranno 21×32748 . Di questa matrice viene ricavata la sua trasposta, per poi ottenere la matrice di covarianza $S = XX^T$ di dimensioni 21×21 , da cui è possibile ricavare gli autovalori e la matrice unitaria U utilizzando due *macro* della libreria Eigen.

```
// Matrix initialization
myMatrix X;
X.setZero(param_1, data_size - param_1 + 1);

for (uint32_t i = 0; i < data_size - param_1 + 1; i++)
{
    for (uint32_t j = 0; j < param_1; j++)
    {
        X(j, i) = in_data[j + i];
    }
}

// SVD calculation
myMatrix Xtranspose(X.transpose());
Eigen::JacobiSVD<myMatrix> svd(X * Xtranspose, Eigen::ComputeThinU);
```

Codice 3.1. Creazione della matrice traiettoria e della matrice di covarianza.

Si sottolinea il fatto che per poter utilizzare queste *macro* occorre definire le matrici con il tipo *Matrix* della stessa libreria Eigen, utilizzando l'allocazione dinamica per entrambe le loro dimensioni, come di seguito riportato.

```
typedef Eigen::Matrix<float, Eigen::Dynamic, Eigen::Dynamic> ←
myMatrix;
```

Codice 3.2. Dichiarazione del tipo Matrix.

Ottenuto il vettore dei valori singolari (di lunghezza 21) della matrice S in ordine decrescente, si può procedere alla loro normalizzazione.

```
// Normalization of eigenvalues
float svd_average = 0;
for (unsigned int i = 0; i < svd.singularValues().size(); i++)
{
    svd_average += svd.singularValues()[i];
}

svd_average = svd_average / (svd.singularValues()[0] * param_1);
```

```
myVector svd_AB(param_1);
for (unsigned int i = 0; i < svd.singularValues().size(); i++)
{
    svd_AB[i] = svd.singularValues()[i] / svd.singularValues()[0] - ←
        svd_average;
}

myVector svd_rms(param_1);

for (unsigned int i = 0; i < svd_AB.size() - 1; i++)
{
    svd_rms[i] = sqrt((svd_AB[i] * svd_AB[i] + svd_AB[i + 1] * svd_AB[i ←
        + 1]) / 2);
}

svd_rms[param_1 - 1] = svd_rms[param_1 - 2];
```

Codice 3.3. Normalizzazione degli autovalori.

Successivamente si ottengono le matrici elementari sulla base del vettore dei valori singolari e viene quindi effettuato il grouping di queste stesse matrici, operazione che prevede altri due prodotti matriciali intermedi.

```
int svd_min_pos;
svd_rms.minCoeff(&svd_min_pos);

uint32_t k = data_size - param_1 + 1;

myVector svd_I;
if (svd_min_pos > 0)
{
    svd_I.resize(svd_min_pos + 1);
    for (int i = 0; i < svd_min_pos + 1; i++)
        svd_I(i) = i;
}
else if (svd_min_pos == 0)
{
    svd_I << 1;
}

myMatrix tmp2(Xtranspose * svd.matrixU().leftCols(svd_I.size()));
myMatrix svd_rca(svd.matrixU().leftCols(svd_I.size()) * tmp2.←
    leftCols(svd_I.size()).transpose());
```

Codice 3.4. Grouping delle matrici elementari.

Infine viene ricostruito il vettore con 32768 campioni filtrati effettuando la media diagonale citata nel paragrafo 1.2 con tre loop distinti.

```
uint32_t svd_Kp, svd_Lp;
if (param_1 < k)
```

```

{
    svd_Lp = param_1;
    svd_Kp = k;
} else
{
    svd_Lp = k;
    svd_Kp = param_1;
}

for (uint32_t i = 0; i < data_size; i++)
    out_data[i] = 0;

// svd
for (uint32_t i = 0; i <= svd_Lp - 2; i++)
{
    for (uint32_t j = 1; j <= i + 1; j++)
    {
        out_data[i] = out_data[i] + (1.0 / (i + 1)) * svd_rca(j - 1, i ←
            - j + 1);
    }
}

for (uint32_t i = svd_Lp - 1; i <= svd_Kp - 1; i++)
{
    for (uint32_t j = 1; j <= svd_Lp; j++)
    {
        out_data[i] = out_data[i] + (1.0 / svd_Lp) * svd_rca(j - 1, i ←
            - j + 1);
    }
}

for (uint32_t i = svd_Kp; i <= data_size; i++)
{
    for (uint32_t j = i - svd_Kp + 2; j <= data_size - svd_Kp + 1; j ←
        ++)
    {
        out_data[i] = out_data[i] + (1.0 / (data_size - i)) * svd_rca(←
            j - 1, i - j + 1);
    }
}

```

Codice 3.5. Ricostruzione del segnale filtrato.

L'algoritmo è stato provato ed eseguito, così come è stato fornito, su uno dei due core della Zedboard, utilizzando come campioni di ingresso dei pattern di prova acquisiti con l'SD card.

Nella figura 3.3 vengono mostrati i tempi di esecuzione delle parti dell'algoritmo più significative in termini di timing (alcune parti non sono state incluse in quanto impiegano un tempo trascurabile), ottenuti grazie alle prove effettuate. L'algoritmo di denoising impiega complessivamente 5,8 secondi per un solo canale. Risulta quindi evidente la necessità di un'accelerazione con hardware parallelo.

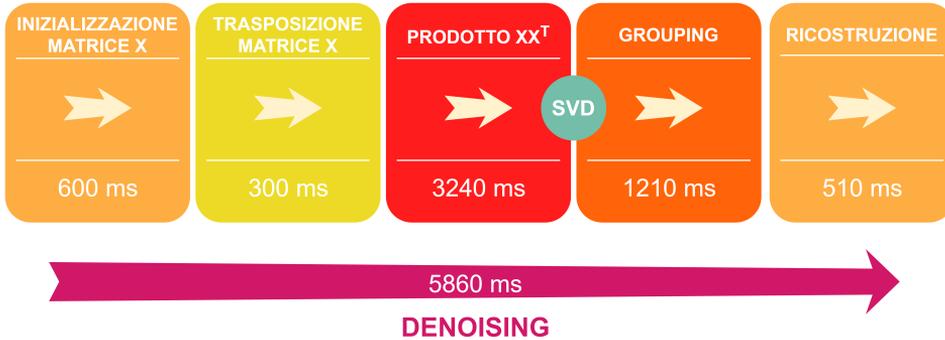


Figura 3.3. Tempi di esecuzione delle varie parti dell'algoritmo.

3.2 Acquisizione dei segnali

Prima di procedere all'accelerazione hardware è necessario acquisire i segnali, per esempio da due fotodiodi, mediante il pin-header della Zedboard.

Ancora una volta bisogna partire da Vivado, dove va inserito e collegato l'*XADC Wizard* nel block diagram del progetto. L'*XADC Wizard* permette di modificare tutti i parametri relativi all'acquisizione analogica, tra cui frequenza di campionamento, canali abilitati e modalità del *sequencer*. Tutti questi parametri sono comunque modificabili anche in SDK.

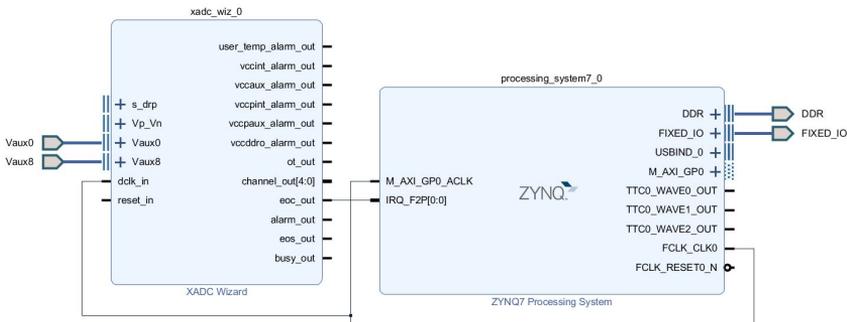


Figura 3.4. Block diagram di Vivado con l'*XADC Wizard* collegato all'IP del PS.

Per poter operare con due canali simultaneamente è stata impostata la modalità *simultaneous sampling* per il sequencer, già citata nel par. 2.1.1, abilitando come canali di ingresso VAUX0 e VAUX8.

La frequenza di campionamento invece non può essere impostata esattamente a 32,768 kHz, essendo essa dipendente dalla frequenza di clock fornita dal PS attraverso dei divisori di frequenza, i quali non permettono di ritoccarla fino all'ordine di grandezza degli Hz. Inoltre la minima frequenza di campionamento che si può inserire è di 39 kHz.

Per questi motivi, e per altri che verranno spiegati nel paragrafo 4.2 relativo all'accelerazione hardware, la frequenza di campionamento scelta è di 65,800 kHz che attraverso un *downsampling* software, eseguito scartando semplicemente i campioni dispari, scende a 32,900 kHz. Se fosse stata impostata una frequenza pari a 32,768 kHz, anziché 32,900 kHz, tra l'acquisizione di un secondo e quello successivo verrebbero persi dei campioni. Questa scelta rispetta comunque le specifiche, avendo che il numero di campioni acquisiti in un secondo è addirittura aumentato, seppur di poco. Da qui in avanti quindi si avrà, $N = 32900$ e $K = N - L + 1 = 32880$ con $L = 21$.

Infine, per poter utilizzare correttamente gli ingressi analogici coinvolti (VAUX0 e VAUX8), bisogna assegnare loro ai rispettivi pin del SoC nei constraints del progetto in Vivado (figura 3.5).

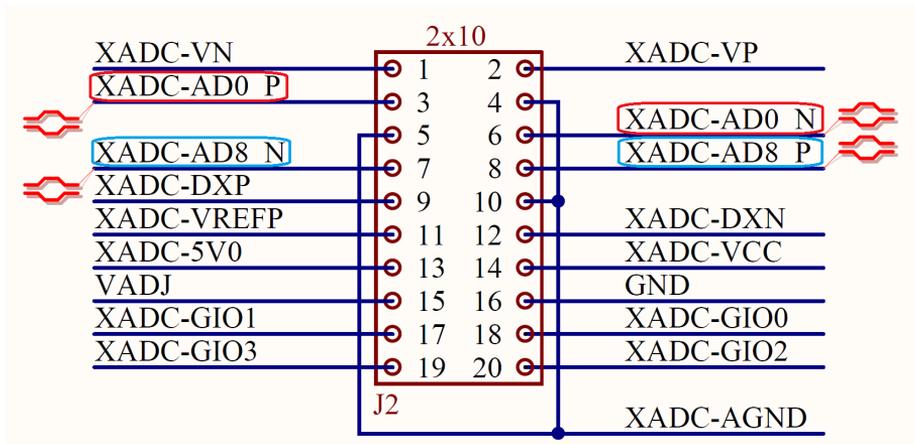


Figura 3.5. Schematico del pin-header della Zedboard con i pin utilizzati in evidenza [9].

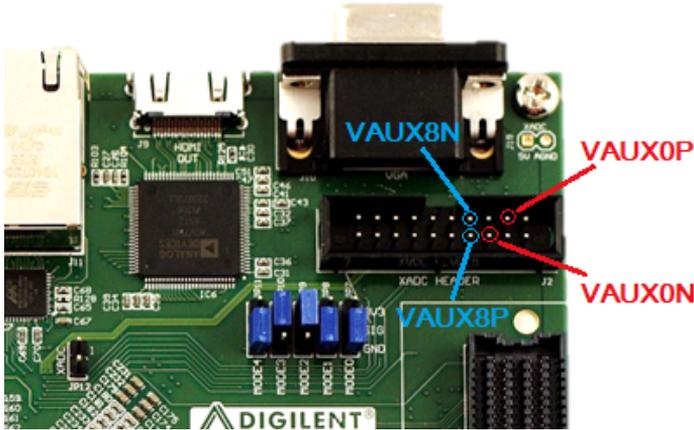


Figura 3.6. Pin dell'XADC Header della Zedboard utilizzati nell'acquisizione dei due segnali.

3.2.1 Acquisizione con interrupt

Nella figura 3.4 si nota un'unica connessione tra XADC Wizard e Processing System, frequenza di clock a parte. Questa connessione va dalla porta *End of Conversion* dell'XADC alla porta dedicata alle *interrupt request* del PS, appositamente abilitata.

L'utilizzo degli interrupt permette di non bloccare la CPU ad ogni conversione effettuata dagli ADC della ZYNQ, anche perché non è noto con precisione il tempo che bisogna aspettare per ognuna di essa.

Con gli interrupt, invece, il segnale EOC si asserisce esattamente quando termina una conversione "triggerando" il PS, il quale avvia una routine grazie ad un *interrupt handler*.

```
void IntrHandler (void *pair)
{
    float data1 = (((float)(XAdcPs_GetAdcData(((std::pair <XAdcPs*, ←
        vector<float >*>)*pair)->first, 16)))) * (1.0f))/65536.0f); //←
    AUX00
    float data2 = (((float)(XAdcPs_GetAdcData(((std::pair <XAdcPs*, ←
        vector<float >*>)*pair)->first, 24)))) * (1.0f))/65536.0f); //←
    AUX08
    // in first entry of data vector store the actual data
    *((std::pair <XAdcPs*, std::vector <float >*>)*pair)->second)←
    [0] = data1;
    *((std::pair <XAdcPs*, std::vector <float >*>)*pair)->second)←
    [1] = data2;
}
```

```

// the second entry is a flag that communicates that the data is ←
new
(*(((std::pair <XAdcPs*, std::vector <float >*>*)pair)->second))←
[2] = 1.;
return ;
}

```

Codice 3.6. Interrupt Handler dell'acquisizione.

L'handler va a trasformare il nuovo campione da numero intero a valore di tensione di tipo *float*, dopodiché comunica al main, con un flag, che un nuovo dato è disponibile. Il main memorizza i soli campioni pari (downsampling) in un buffer in memoria DDR da 32900 elementi e reimposta il flag a zero per indicare che il dato in questione è stato "fetchato".

Tutto ciò avviene per entrambi i canali utilizzando la stessa routine, essendo il campionamento simultaneo, ma con due buffer diversi.

```

while (true)
{
// print data if the buffer is full
if(current_index == data_size*2)
{
//ping-pong
if(*(baseaddr_p+0) == 0) *(baseaddr_p+0) = 1; //first ←
filling
else if(*(baseaddr_p+0) == 2) *(baseaddr_p+0) = 1; //first ←
buffer filled
else if(*(baseaddr_p+0) == 1) //second buffer filled
{
*(baseaddr_p+0) = 2;
k = 1;
}
// start overwriting the buffer from the beginning
current_index = 0;
// set flag to "old data "
(*(mypair.second))[2] = 0.;
}

// fetch the data if the interrupt handler has updated it
if ((*(mypair.second))[2] > 0.)
{
// set the flag to "old data "
(*(mypair.second))[2] = 0.;

//resample (store only even samples)
if((current_index % 2) == 0)
{
*(baseaddr_p+k) = (*(mypair.second))[0]; //CH1
*(baseaddr_p+k+1) = (*(mypair.second))[1]; //CH2
k = k+2;
}
}
}

```

```

    }
    // next element
    ++current_index;
  }
}

```

Codice 3.7. Fetch e downsampling dei campioni acquisiti.

Integrando l'algoritmo alla parte di acquisizione si scopre però che il tempo di esecuzione si allunga ulteriormente fino a 6,5 secondi per canale, anziché 5,8. Questo è causato dall'elevato numero di interrupt che rallenta la CPU, infatti aumentando la frequenza di campionamento, e quindi il numero di interrupt, questo tempo si dilata ulteriormente. Ciò va a motivare l'utilizzo del secondo core del PS.

3.3 Versione dual-core

Per ridurre il tempo di esecuzione si possono utilizzare entrambi i core ARM a disposizione. A questo scopo, sono possibili due approcci.

Il primo è un approccio di tipo simmetrico, dove ogni core è dedicato ad un solo canale ed esegue gli stessi identici task dell'altro: acquisizione, denoising e visualizzazione in uscita. Tuttavia questa strada è stata scartata a causa delle problematiche legate alla condivisione di risorse tra i due core, tra cui lo stesso XADC, l'SD card e l'uscita VGA. Inoltre persisterebbe il problema legato agli interrupt dell'acquisizione, i quali andrebbero comunque ad inficiare sulle performance relative al denoising.

Il secondo approccio invece è di tipo asimmetrico, ovvero si ha una sorta di *pipeline* dove i due core eseguono compiti e task differenti senza condividere alcuna risorsa, se non un'area di memoria DDR usata per comunicare.

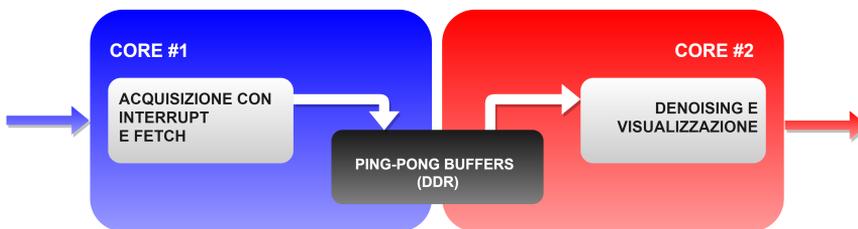


Figura 3.7. Suddivisione dei task tra i due core.

Il primo core si occupa infatti della sola acquisizione mediante interrupt, mentre il secondo core esegue l'algoritmo e manda i dati in uscita. Questo fa sì che gli interrupt non vadano a disturbare l'esecuzione dell'algoritmo, essendo i due core completamente indipendenti, riportando il tempo di esecuzione dell'algoritmo a 5,8

secondi anziché 6,5.

Ogni core ha la propria applicazione e quindi aree di memoria DDR dedicate ad esse, distinte e non sovrapposte. Tuttavia le due unità elaborative hanno bisogno di comunicare.

A tal proposito i due core condividono un'area di memoria DDR allocata esternamente alle aree delle rispettive applicazioni. Quest'area svolge il ruolo di *mailbox*, dove il primo core scrive i campioni di ingresso e il secondo core li legge. Questo meccanismo è innescato correttamente mediante l'uso di *ping-pong buffers*.

3.3.1 Ping-pong buffers

Il primo core dopo aver acquisito un secondo di campioni e dopo averli memorizzati nella locazione di memoria condivisa DDR, deve "svegliare" il secondo core, comunicando ad esso di poter leggere dal buffer e quindi di poter eseguire l'algoritmo. Nel lasso di tempo in cui il core 2 esegue l'algoritmo, però, il core 1 non può fermare l'acquisizione, altrimenti dei campioni verrebbero persi. Allo stesso tempo non può andare a scrivere sullo stesso buffer altrimenti corromperebbe la lettura del core 2. Nasce quindi l'esigenza di utilizzare due buffer diversi, dove i due core si alternano nell'accesso ad essi, da cui quindi il nome *ping-pong buffers*.

L'alternanza viene scandita da un flag, localizzato nell'area di memoria condivisa tra i due core, il quale può assumere tre valori: 0, 1 e 2. Essi stanno rispettivamente per *primo riempimento*, *buffer 1 pronto* e *buffer 2 pronto* e il valore può essere impostato solo dal core 1.

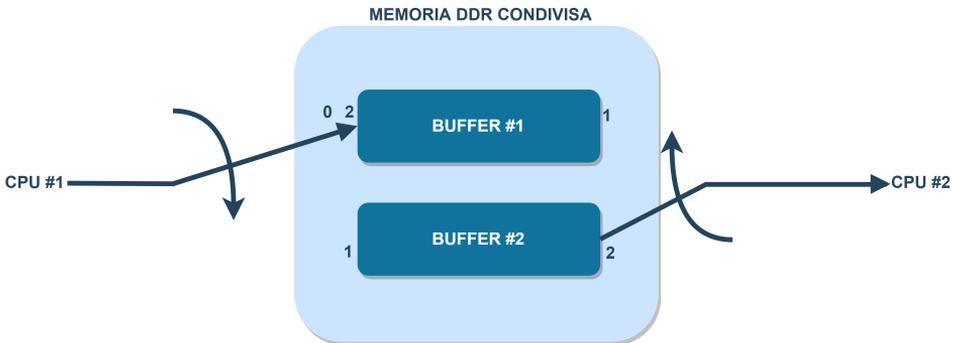


Figura 3.8. Principio di funzionamento dei ping-pong buffers.

Al primo avvio il flag è inizializzato a 0 (primo riempimento), valore che assume solo in questa circostanza. Il core 1 inizia a riempire il buffer 1, mentre il core 2 è in attesa. Appena il buffer 1 viene riempito, il flag viene impostato a 1 (buffer

1 pronto) e il core 1 inizia a riempire il secondo buffer. Ciò consente al core 2 di leggere dal buffer 1, facendo tutte le operazioni annesse (denoising e visualizzazione). Quando il core 2 finisce queste operazioni aspetta che il flag venga impostato a 2 (buffer 2 pronto) dal core 1, cosa che avviene solo quando il buffer 2 è stato riempito. A quel punto il core 2 potrà leggere ed elaborare i dati del buffer 2, mentre il core 1 riempirà di nuovo il buffer 1, sovrascrivendolo. Quando avrà finito il flag verrà di nuovo impostato a 1, consentendo nuovamente al core 2 di leggere ed elaborare i dati del buffer 1, e così via.

Va sottolineato che il core 1 non smette mai di acquisire e riempire i buffer. In particolare, questi ultimi hanno dimensioni doppie, in modo tale da accogliere entrambi i canali. Infatti, ad ogni acquisizione, il campione proveniente dal primo canale viene memorizzato in una posizione dispari del buffer, invece il campione relativo al secondo canale viene memorizzato in una posizione pari, subito successiva a quella precedente. Nella stessa modalità, i dati vengono letti dal core 2 e raggruppati in due vettori distinti, cosa necessaria per il denoising.

```
while (true)
{
    while(*(baseaddr_p+0) != 1){}; //wait for buffer1
    while(*(baseaddr_p+0) == 0){}; //wait for first filling
    while(true)
    {
        in_data1.push_back(*(baseaddr_p+k));
        in_data2.push_back(*(baseaddr_p+k+1));
        k = k+2;

        if(k >= data_size_tot*2)
        {
            cout << "Acquired buf1: " << (in_data1.size()+in_data2.size()←
                ())/2 << "/ch" << endl;

            /******SSA EXECUTION←
                *****/
            vector<float> svd_y1(in_data1.size());
            vector<float> svd_y2(in_data2.size());

            XTime_GetTime(&tStart); //start timer
            /*SSA execution ch1*/
            /*SSA execution ch2*/
            XTime_GetTime(&tEnd); //end timer

            tElapsed = (tEnd-tStart)/(COUNTS_PER_SECOND/1000);
            cout << "SSA executed in " << tElapsed << " ms" << endl;

            //clear buffer
            in_data1.clear();
            in_data2.clear();
            k = 1+(data_size_tot*2);
        }
    }
}
```

```

        svd_y1.clear();
        svd_y2.clear();
        break;
    }
}
while(*(baseaddr_p+0) != 2){}; //wait for buffer2
while(true)
{
    in_data1.push_back(*(baseaddr_p+k));
    in_data2.push_back(*(baseaddr_p+k+1));
    k = k+2;
    if(k >= data_size_tot*2*2)
    {
        cout << "Acquired buf2: " << (in_data1.size()+in_data2.size()←
            ())/2 << "/ch" << endl;

        /*****SSA EXECUTION←
            *****/
        vector<float> svd_y1(in_data1.size());
        vector<float> svd_y2(in_data2.size());

        XTime_GetTime(&tStart); //start timer
        /*SSA execution ch1*/
        /*SSA execution ch2*/
        XTime_GetTime(&tEnd); //end timer

        tElapsed = (tEnd-tStart)/(COUNTS_PER_SECOND/1000);
        cout << "SSA executed in " << tElapsed << " ms" << endl;

        //clear buffer
        in_data1.clear();
        in_data2.clear();
        k = 1;

        svd_y1.clear();
        svd_y2.clear();
        break;
    }
}
}
}

```

Codice 3.8. Lettura dei ping-pong buffers da parte del core 2.

Ovviamente per far sì che non vengano persi campioni, l'elaborazione di ogni buffer (denoising e visualizzazione) e quindi di entrambi i canali, deve avvenire in un tempo inferiore al secondo. Attualmente il tempo di esecuzione dell'algoritmo resta di 5,8 secondi per canale. È necessario quindi accelerare la fase di denoising e di garantirsi un certo margine per consentire la visualizzazione.

Capitolo 4

Accelerazione Hardware

Per poter accelerare efficacemente la fase di denoising, bisogna individuare i colli di bottiglia dell'algoritmo per poi replicare la loro funzionalità in hardware. L'idea è quella di invocare dell'hardware, appositamente implementato nella parte di logica programmabile (PL), ogni qual volta ci siano dei task particolarmente onerosi in termini di complessità computazionale.

Per poter raggiungere un risultato soddisfacente, l'hardware parallelo va combinato ad alcune migliorie software, anche perché non tutto l'algoritmo è riproducibile in hardware. Infatti, alcune parti di esso vengono eseguite in maniera estremamente efficiente, grazie all'utilizzo della libreria **Eigen**. Un esempio è il calcolo degli autovalori della matrice S , il quale viene eseguito in soli 22 ms con delle *macro* apposite. Riprodurre la funzionalità di queste *macro* in hardware significherebbe saturare le risorse della PL a disposizione, a causa del gran numero di iterazioni da eseguire, senza ottenere un effettivo speed-up. Conviene quindi lasciare queste parti alle funzioni Eigen.

4.1 Colli di bottiglia

Nella figura 3.3 erano state individuate, attraverso l'utilizzo del timer del PS, le parti più lente dell'algoritmo. Tra queste spiccano la parte centrale, relativa al prodotto tra la matrice traiettoria e la sua trasposta, e quella successiva riguardante il grouping. Questi due blocchi insieme impiegano oltre 4,4 secondi.

Da un'analisi più approfondita dell'algoritmo, si nota come entrambe queste parti siano essenzialmente rallentate dal prodotto tra matrici di grandi dimensioni. Il blocco centrale, nello specifico, risulta totalmente dedicato al prodotto tra due matrici di dimensioni pari a 21×32880 una, e 32880×21 l'altra (trasposta). Il

blocco di grouping compie più operazioni, tuttavia anche in questo caso la quasi totalità del tempo di esecuzione è impiegato in due prodotti matriciali. Tra l'altro, uno dei moltiplicandi ha una dimensione non nota a tempo di compilazione. Nel caso peggiore però, i due prodotti coinvolgono una matrice 21×32880 ed una 21×21 .

Riassumendo, si ha:

- 1) $S^{21 \times 21} = X^{21 \times 32880} \times X_{tran}^{32880 \times 21}$ (blocco centrale)
- 2) $T^{32880 \times 21} = X_{tran}^{32880 \times 21} \times U_{left}^{21 \times 21}$ (blocco grouping)
- 3) $R^{21 \times 32880} = U_{left}^{21 \times 21} \times T_{tran}^{21 \times 32880}$ (blocco grouping)

Si nota che questi prodotti presentano anche delle data dependency tra loro, infatti il terzo prodotto matriciale fa uso della matrice T_{tran} , trasposta della matrice T , ottenuta dal secondo prodotto matriciale. Questo significa che il terzo prodotto non può essere eseguito finché il secondo prodotto matriciale non è stata portato a termine.

Oltre ai prodotti tra matrici, vanno infine velocizzate le parti di inizializzazione e trasposizione della matrice X (900 ms) e la parte di finale di ricostruzione del vettore dei campioni filtrati (510 ms). Queste parti, come anche i prodotti appena elencati, richiedono un intervento sia hardware che software.

4.2 Implementazione dell'hardware parallelo

I primi tre blocchi dell'algoritmo di denoising della figura 3.3, che insieme impiegano oltre 4 secondi, sono maggiormente adatti ad un'implementazione hardware. Ciò è dovuto al fatto di avere solo operazioni matriciali che non coinvolgono calcoli iterativi particolarmente complessi. Tuttavia servono delle migliorie software.

Questi tre blocchi sono strettamente connessi tra loro. Tutto quello che serve per ottenere la matrice S è la matrice traiettoria X , che viene costruita partendo dal vettore dei campioni iniziali.

Una prima miglioria, dal punto di vista software, è quella di evitare l'uso della libreria Eigen per il calcolo della trasposta di X e per il prodotto $S = XX^T$. Essendo le dimensioni delle matrici coinvolte note a priori, basta eseguire un classico prodotto riga per colonna, tra gli elementi dei due moltiplicandi. L'altra miglioria scaturisce dal fatto che il prodotto in questione può essere eseguito tra i soli elementi della matrice X semplicemente scambiando gli indici all'interno dei loop, senza il bisogno di definire una vera e propria matrice trasposta. Questi accorgimenti software portano ad una prima velocizzazione, ma anche ad un risparmio di memoria particolarmente prezioso per la fase successiva.

Il solo intervento software non è abbastanza. Restano, infatti, i problemi connessi

alla grande mole di dati su cui eseguire il prodotto. Occorre implementare direttamente in hardware il prodotto $S = XX^T$ seguendo le migliori appena citate. Per farlo, conviene utilizzare Vivado HLS, traducendo direttamente in hardware dal linguaggio di alto livello questa parte dell'algoritmo.

4.2.1 Scomposizione dell'algoritmo: divide et impera

L'idea è quella di creare un IP che riceva in ingresso il vettore contenente i campioni acquisiti in un secondo. L'IP deve inizializzare e creare la matrice traiettoria X , eseguire il prodotto riga per colonna $S = XX^T$ (senza definire la matrice traspota) e fornire in uscita direttamente S .

Tuttavia, l'algoritmo così com'è richiederebbe di memorizzare nella logica programmabile (Block-RAM) una matrice X di dimensioni 21×32880 , cosa non possibile. Infatti la PL dedica alla Block-RAM 280 blocchi da 18 kb, ovvero circa 5 Mb di memoria, mentre una matrice X ha 21×32880 elementi di tipo float (32 bit), che portano a 22 Mb di area di memoria necessaria. Per ovviare a questo problema, conviene utilizzare un approccio *divide et impera*, ovvero fornire all'hardware solo una porzione alla volta del vettore di campioni in ingresso, spostando la finestra progressivamente, ed accumulando il risultato di ogni prodotto parziale nella matrice S in memoria DDR.

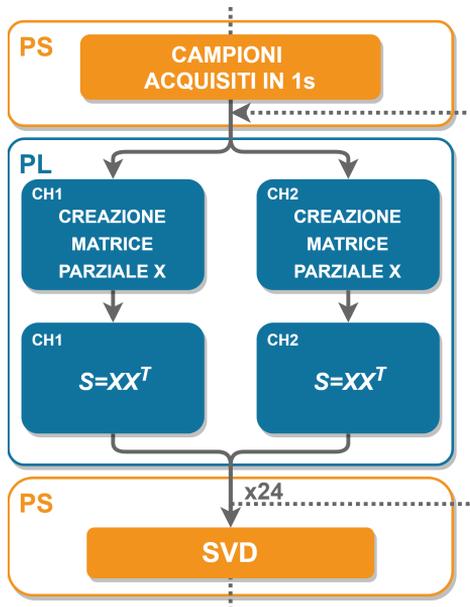


Figura 4.1. Suddivisione dei task tra PS e PL nella prima parte dell'algoritmo.

Ciò permetterebbe di avere ad ogni ciclo una matrice parziale X di dimensioni tali da poter essere memorizzata nella Block-RAM della PL e allo stesso tempo di non modificare l'algoritmo in se. Infatti, gli autovalori saranno poi calcolati sulla matrice S completa, ovvero quella associata ad un secondo di acquisizione.

Riassumendo in maniera analitica, con questo approccio la fase di embedding, citata nel paragrafo 1.2, è affidata interamente all'hardware e verrebbe scomposta nel seguente modo.

Dato il segnale s in ingresso di lunghezza N , $s = \{s_n, n = 1 \dots N\}$, ad ogni ciclo viene prelevato da esso una finestra di lunghezza W , con $L < W < N$. La finestra, composta da W campioni, viene mappata con una sequenza di $K' = W - L + 1$ vettori colonna di lunghezza L . Il risultato è una matrice traiettoria parziale X' di dimensioni $L \times K'$ che, come quella totale, è una matrice di Hankel. Nota quest'ultima, si può ricavare la matrice di covarianza parziale $S' = X'X'^T$, la quale verrà scritta in uscita dall'hardware. Questo procedimento viene ripetuto F volte, dove F è il numero di finestre totali. La matrice di covarianza totale sarà la somma delle F matrici di covarianza parziali.

$$S = \sum_{i=1}^F S'_i \quad (4.1)$$

Da questo punto in poi, il controllo dell'esecuzione ritorna al PS che provvederà ad eseguire la SVD su S per poi completare il processo di denoising, come in precedenza.

Come accennato nel paragrafo 3.2, la frequenza di campionamento, ovvero il numero di campioni N (32900) da acquisire in un secondo, è stata scelta in modo tale da poter essere impostata correttamente nell'XADC Wizard ma anche per altre motivazioni riguardanti l'accelerazione hardware. Queste fanno riferimento alla necessità di usare un numero di finestre F tale da non perdere campioni (K deve essere divisibile per F) e che allo stesso tempo permetta di non saturare la Block-RAM disponibile. A tal proposito, il vettore di ingresso è stato diviso in 24 finestre da 1370 campioni l'una, consentendo di parallelizzare l'hardware e processare i due canali simultaneamente con un unico IP. Questa scelta permette di non saturare nè la Block-RAM della PL, nè il buffer dell'AXI DMA IP (par. 4.3.1), il quale sarà utilizzato per leggere e scrivere dalla memoria DDR.

Si ricorda che $N = 32900$, $L = 21$ e $K = N - L + 1 = 32880$ dove L non va confusa con la lunghezza della finestra $W=K/F$, essendo L la lunghezza della finestra di campioni usata dall'algoritmo internamente, per il calcolo iterativo degli autovalori della matrice S .

```
void X_prod(const float in_data[(data_size+param_1)*2], float ←
    out_data[param_1*param_1*2], bool *out_data_TLAST)
{
```

```

*out_data_TLAST = 0;
float in_buf[data_size+param_1];
float X[param_1][data_size];
float in_buf2[data_size+param_1];
float X2[param_1][data_size];

unsigned int ind_in = 0;
for (unsigned int i = 0; i < (data_size+param_1)*2; i=i+2)
{
    in_buf[ind_in] = in_data[i];
    in_buf2[ind_in] = in_data[i+1];
    ind_in++;
}

for (unsigned int i = 0; i < data_size; i++)
{
    for (unsigned int j = 0; j < param_1; j++)
    {
        X[j][i] = in_buf[j + i];
        X2[j][i] = in_buf2[j + i];
    }
}

float sum = 0;
float sum2 = 0;
unsigned int ind_out = 0;
for (unsigned int i = 0; i < param_1; ++i)
{
    for (unsigned int j = 0; j < param_1; ++j)
    {
        sum = 0;
        sum2 = 0;
        for (unsigned int k = 0; k < data_size; ++k)
        {
            sum += X[i][k] * X[j][k];
            sum2 += X2[i][k] * X2[j][k];
        }
        out_data[ind_out] = sum;
        out_data[ind_out+1] = sum;
        if (i >= param_1-1 && j >= param_1-1 && out_data[ind_out+1] <=
            0.0000) *out_data_TLAST = 1;
        ind_out = ind_out+2;
    }
}
}

```

Codice 4.1. Parte dell'algoritmo accelerato in HLS.

Il blocco IP ha bisogno però di comunicare con l'esterno con un protocollo valido. Esso deve permettere di scrivere e leggere dalla memoria DDR con una latenza trascurabile, ma allo stesso tempo deve essere compatibile con vettori e matrici. La soluzione adatta è il protocollo AXI4-Stream.

4.2.2 Il protocollo AXI4

Il protocollo AXI (*Advanced eXtensible Interface*) [10] fa parte della famiglia di bus per microcontrollori ARM AMBA, introdotta nel 1996. La versione AXI4 è stata rilasciata nel 2010. Esistono tre tipi di interfacce AXI4:

- AXI4, bus ad alte prestazioni di tipo memory-mapped.
- AXI4-Lite, bus per semplici comunicazioni memory-mapped a basso throughput (per esempio, tra un controllo ed un registro di stato).
- AXI4-Stream, bus per lo streaming di dati ad alta velocità.

Il protocollo AXI4-Stream rimuove del tutto la necessità dell'indirizzamento (non è un bus memory-mapped) e permette di avere un data burst con dimensioni illimitate. Le interfacce AXI4 e AXI4-Lite constano di cinque canali diversi:

- Canale Read Address
- Canale Write Address
- Canale Read Data
- Canale Write Data
- Canale Write Response

Supponendo di avere una coppia master-slave, i dati possono andare in entrambe le direzioni, simultaneamente, inoltre la quantità di dati trasferiti può variare. Nel bus AXI4, il limite massimo di dati trasferiti in una sola transazione di tipo burst è di 256 bit, a seconda del parallelismo dei dati e della lunghezza scelta per il burst (numero di trasferimenti consecutivi in una sola transazione). Il bus AXI4-Lite permette invece un solo trasferimento di dati per transazione.

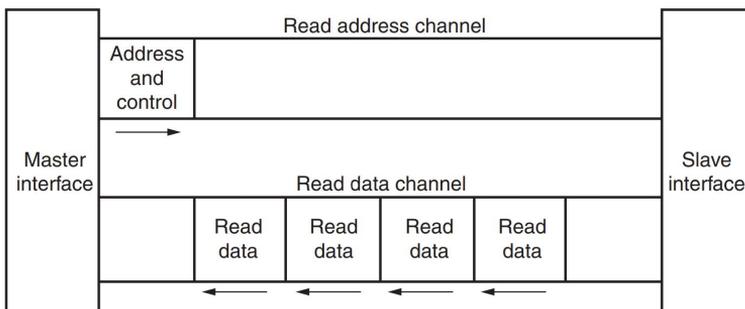


Figura 4.2. Transazione di lettura con il protocollo AXI4.

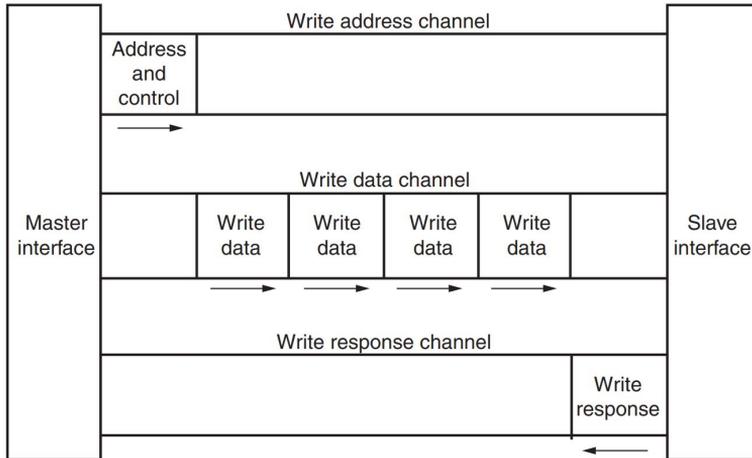


Figura 4.3. Transazione di scrittura con il protocollo AXI4.

Come mostrato nelle figure 4.2 e 4.3, il bus AXI4 fornisce canali separati di dati ed indirizzi tra lettura e scrittura, consentendo il trasferimento dei dati in entrambe le direzioni simultaneamente. Al livello hardware, l'interfaccia AXI4 consente di avere clock differenti per ogni coppia master-slave. Inoltre, in generale, il protocollo AXI permette l'inserimento di registri di pipeline per ridurre la latenza. Il protocollo AXI4-Stream definisce un singolo canale per la trasmissione streaming dei dati (unidirezionalità). Questo canale è modellato sull'interfaccia AXI4. A differenza dell'AXI4, l'interfaccia AXI4-Stream può effettuare il burst su una quantità di dati illimitata. L'unico svantaggio è quello di non avere il riordine dei dati, quindi i dati devono essere letti e scritti in modalità sequenziale. Vivado HLS permette di assegnare gli ingressi e le uscite della funzione da implementare, e quindi del futuro IP, ad una delle tre interfacce AXI o ad altri tipi di interfaccia più generiche, attraverso delle direttive, citate nel paragrafo 2.4.3. In particolare, con la direttiva *axis* si assegna l'ingresso o l'uscita al protocollo AXI4-Stream.

```

void X_prod(const float in_data [(data_size+param_1)*2], float ↔
           out_data[param_1*param_1*2], bool *out_data_TLAST)
{
  #pragma HLS INTERFACE ap_none port=out_data_TLAST
  #pragma HLS INTERFACE axis port=in_data
  #pragma HLS INTERFACE axis port=out_data
  ...

```

Codice 4.2. Aggiunta delle direttive di interfaccia in Vivado HLS.

Interface					
Summary					
RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	X_prod	return value
ap_rst_n	in	1	ap_ctrl_hs	X_prod	return value
ap_start	in	1	ap_ctrl_hs	X_prod	return value
ap_done	out	1	ap_ctrl_hs	X_prod	return value
ap_idle	out	1	ap_ctrl_hs	X_prod	return value
ap_ready	out	1	ap_ctrl_hs	X_prod	return value
in_data_TDATA	in	32	axis	in_data	pointer
in_data_TVALID	in	1	axis	in_data	pointer
in_data_TREADY	out	1	axis	in_data	pointer
out_data_TDATA	out	32	axis	out_data	pointer
out_data_TVALID	out	1	axis	out_data	pointer
out_data_TREADY	in	1	axis	out_data	pointer
out_data_TLAST	out	1	ap_none	out_data_TLAST	pointer

Figura 4.4. Interfacce assegnate agli ingressi ed alle uscite del futuro IP [8].

Questa interfaccia risulta particolarmente adatta al caso in questione, sia perché compatibile con vettori e matrici, sia per le alte prestazioni. L'unico requisito è quello di leggere e scrivere gli elementi degli array di interfaccia in ordine (modalità sequenziale), essendo essi destinati a degli stream.

Nella figura 4.4 si notano i segnali di controllo dell'IP, tra cui anche il clock ed il reset, creati in automatico da Vivado HLS in fase di sintesi con un apposito protocollo *ap_ctrl_hs*.

Nella stessa figura si distinguono gli ingressi e le uscite con il protocollo AXI4-Stream (*axis*). Per ogni ingresso ed uscita vengono creati i segnali TDATA, TVALID e TREADY che stanno rispettivamente (a seconda della direzione) per i canali di Read o Write Data, Read o Write Address (in questo caso Control) e Write Response (fig. 4.3).

Infine, vi è il segnale di uscita TLAST, che è stato aggiunto manualmente per permettere all' AXI DMA IP (par. 4.3.1) di capire quando l'IP connesso al suo ingresso sta scrivendo l'ultimo dato. La direttiva utilizzata per definire l'interfaccia di TLAST è *ap_none*. Essa è utilizzata quando si vuole solo definire un segnale, senza far uso di un vero e proprio protocollo.

4.2.3 Sintesi ad alto livello ed RTL

Una volta definite le interfacce del futuro IP, si può eseguire la sintesi ad alto livello. Per ottenere una latenza bassa vengono impartite delle direttive (citate nel par.

2.4.3) all'interno della parte di algoritmo da sintetizzare. L'algoritmo comprende ora tre loop:

- 1) il primo estrae (in ordine) dal vettore di ingresso la finestra di campioni e li inserisce in due buffer, uno per ogni canale ($1370 + 21$ cicli);
- 2) il secondo crea la matrice traiettoria parziale X per entrambi i canali (1370×21 cicli);
- 3) il terzo esegue il prodotto riga per colonna tra gli stessi elementi della matrice X (per entrambi i canali), inserendo i risultati (in ordine) nell'unico vettore finale ($21 \times 21 \times 1370$ cicli).

I vettori di ingresso ed uscita contengono nelle posizioni pari i campioni del primo canale ed in quelle dispari i campioni del secondo canale, in modo tale da permettere l'accesso sequenziale.

I tre loop appena citati possono essere velocizzati mediante delle direttive HLS PIPELINE, le quali in fase di sintesi andranno a minimizzare la latenza fin dove è possibile. Dei tre loop, il terzo è quello con il maggior numero di cicli, avendo esso tre loop annidati. Per questo motivo, oltre al pipelining, è stata aggiunta una direttiva che imponga di effettuare un ulteriore *loop unrolling* di un fattore 10 nel loop più interno.

```

...
for (unsigned int i = 0; i < (data_size+param_1)*2; i=i+2)
{
    #pragma HLS PIPELINE II=1
    in_buf[ind_in] = in_data[i];
    in_buf2[ind_in] = in_data[i+1];
    ind_in++;
}

for (unsigned int i = 0; i < data_size; i++)
{
    #pragma HLS PIPELINE II=1
    for (unsigned int j = 0; j < param_1; j++)
    {
        X[j][i] = in_buf[j + i];
        X2[j][i] = in_buf2[j + i];
    }
}

...

for(unsigned int i = 0; i < param_1; ++i)
{
    for(unsigned int j = 0; j < param_1; ++j)

```

```

{
  sum = 0;
  sum2 = 0;
  for (unsigned int k = 0; k < data_size; ++k)
  {
    #pragma HLS UNROLL factor=10
    #pragma HLS PIPELINE II=1
    sum += X[i][k] * X[j][k];
    sum2 += X2[i][k] * X2[j][k];
  }
}

```

...

Codice 4.3. Parte dell’algoritmo accelerato con l’aggiunta delle direttive di performance.

Ognuna di queste scelte è stata presa eseguendo più volte la sintesi ad alto livello, al fine di minimizzare il più possibile la latenza, conservando le *data dependency*, ma allo stesso tempo evitando di saturare le risorse hardware della PL. Si sottolinea che prima della sintesi C, il codice ad alto livello è stato testato con diverse simulazioni, mediante un test-bench C/C++.

Nella figura 4.5 si evince come il terzo loop resta quello dominante, ma con una latenza accettabile. Nella stessa figura è possibile esaminare l’intervallo in cui effettivamente parte la pipeline per ognuno dei loop. Le data dependency obbligano a posticipare di diversi intervalli l’avvio della pipeline. Questo perché si sta utilizzando per ogni canale una sola matrice X, la quale avrà un numero limitato di porte attraverso cui vengono effettuate sia operazioni di lettura che di scrittura.

Loop

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- Loop 1	2782	2782	2	2	1	1391	yes
- Loop 2	15070	15070	12	11	1	1370	yes
- Loop 3	616077	616077	1397	-	-	441	no
+ Loop 3.1	1392	1392	33	10	1	137	yes

Figura 4.5. Stima post-sintesi C della latenza (in ns) dei vari loop nell’algoritmo.

Nella figura 4.6 si può ottenere una prima stima della velocità dell’hardware nell’eseguire la parte iniziale dell’algoritmo di denoising. Moltiplicando il periodo di clock minimo per la latenza si ottiene 9,81 ms che moltiplicati per il numero di finestre (24) diventano circa 236 ms. Risultato più che accettabile.

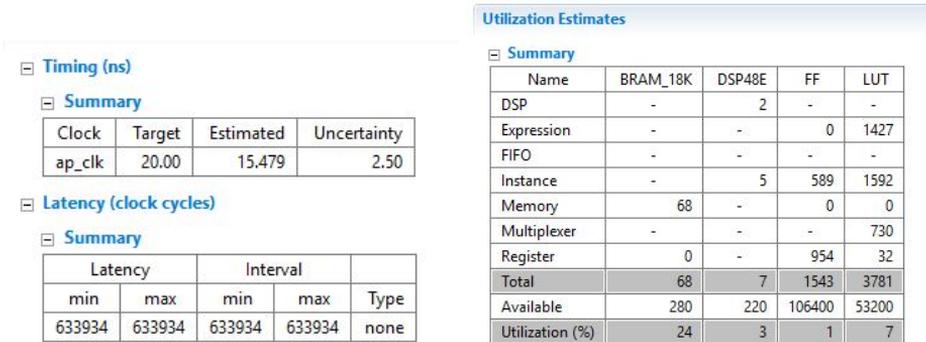


Figura 4.6. Stima post-sintesi C del periodo minimo di clock e della latenza (a sinistra) e delle risorse hardware utilizzate (a destra).

Nella stessa figura vengono riportate le risorse della PL utilizzate per raggiungere questo risultato. Si nota come esse siano molto contenute tranne che per la Block-RAM, la quale ospita le due matrici traiettoria.

Una volta testato il comportamento dell'hardware con una simulazione RTL, si può procedere alla sintesi RTL e quindi all'esportazione dell'IP.

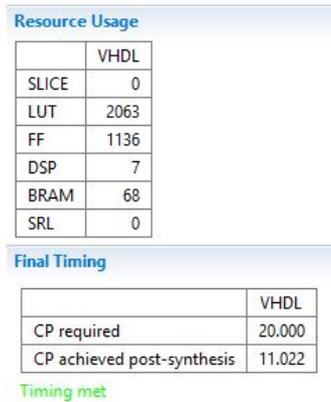


Figura 4.7. Stima post-sintesi RTL delle risorse della PL e del periodo minimo di clock.

Dalla figura 4.7 è possibile vedere il numero effettivo di risorse hardware utilizzate. Si nota, rispetto alla sintesi C, una riduzione del numero di Flip-Flop ed un aumento delle Look-Up Table. Dalla stessa figura si evince come, in fase di sintesi RTL, il timing non solo sia stato rispettato ma addirittura migliorato, con il periodo minimo di clock che è diminuito di circa 4 ns rispetto alla sintesi C.

Ora che l'IP è stato esportato, è possibile passare a Vivado per integrarlo con il PS e la memoria DDR.

4.3 Integrazione dell'hardware parallelo

Una volta importato in Vivado l'IP dedicato ai primi tre blocchi dell'algoritmo di denoising, riportati nella figura 3.3, è necessario integrarlo con il PS e l'XADC al fine di ricostruire l'algoritmo per intero.

Serve un meccanismo di comunicazione ad alta velocità tra memoria DDR e l'IP prodotto da Vivado HLS, che sia compatibile con il protocollo AXI4-Stream. Questo meccanismo deve permettere il trasferimento dei dati dalla DDR verso la PL e viceversa. A tal proposito, l'IP AXI DMA fornito da Vivado risulta particolarmente utile.

4.3.1 AXI DMA

L'AXI Direct Memory Access (AXI DMA) IP core [11] è un *Intellectual Property* fornito dalla suite di Vivado che permette l'accesso diretto e a banda larga tra memoria e periferiche di tipo AXI4-Stream della logica programmabile.

L'AXI DMA si può comportare, a seconda degli interlocutori e della fase elaborativa, sia da master che da slave. In merito a ciò, si distinguono due tipi di canali di comunicazione, distinti ed indipendenti tra loro:

- il canale MM2S *Memory Mapped to Slave*;
- il canale S2MM *Slave to Memory Mapped*.

In figura 4.8 sono riportati i blocchi interni all'AXI DMA. Nella modalità *Simple DMA* l'interfaccia AXI Read MM2S legge i dati da una memoria esterna di tipo master, dopodiché l'AXI Data Mover trasmette questi dati ad una periferica esterna di tipo slave, attraverso la porta AXI Stream MM2S. Analogamente, una periferica master può inviare dati alla porta AXI Write S2MM, quindi gli stessi dati vengono poi scritti in una memoria esterna di tipo slave, attraverso la porta AXI Stream S2MM. Le operazioni in uscita vengono segnalate con degli interrupt (MM2S_IntrOut e S2MM_IntrOut) anch'essi indipendenti.

Entrambi i canali fanno uso di un apposito controller che va a regolare i rispettivi Data Mover, nonché a gestire i segnali di controllo in uscita (MM2S) e quelli di stato in ingresso (S2MM). Il modulo SG viene utilizzato nella modalità *Scatter Gather*, non di interesse per questa applicazione.

Infine, vi è un'interfaccia AXI4-Lite utile per configurare i registri delle possibili modalità di funzionamento.

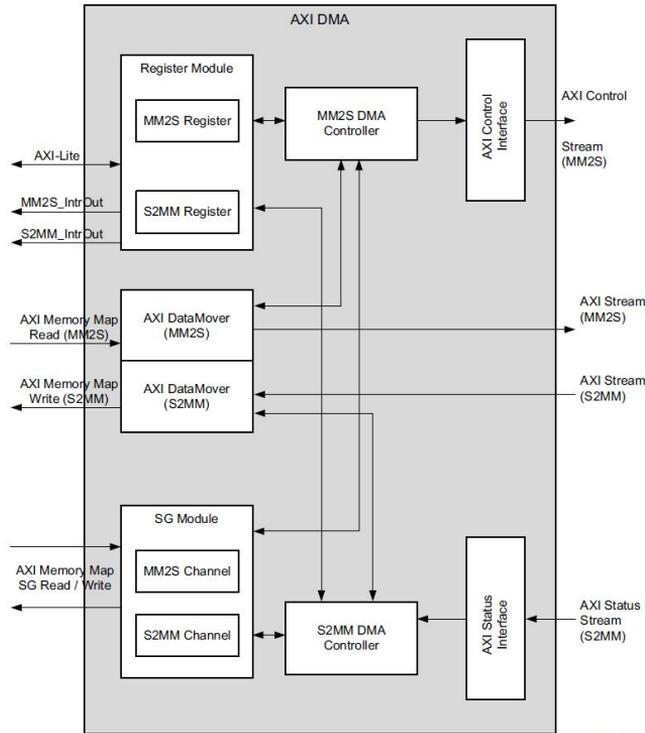


Figura 4.8. Schema a blocchi dell'AXI DMA IP Core [11].

4.3.2 Comunicazione tra PS e PL

Una volta aggiunto l'IP importato da Vivado HLS e l'AXI DMA al block diagram con il PS e l'XADC Wizard, si può procedere al loro collegamento seguendo lo schema di principio della figura 4.9.

Le porte utilizzate dal PS per comunicare con la logica programmabile sono la GP0 e l'ACP. La prima è una porta *General Purpose* di tipo AXI che serve al PS, che fa da master, per inviare i comandi all'AXI DMA (slave in questo caso) attraverso l'interfaccia AXI4-Lite.

L'ACP (*Accelerator Coherency Port*) è invece un'interfaccia AXI da 64 bit che implementa un punto di accesso diretto tra PL, quindi AXI DMA (master in questo caso), e PS (slave), preservando la coerenza della memoria cache. Questo collegamento è a bassa latenza, infatti i master presenti nella PL possono accedere alla cache esattamente come fanno i due processori, incrementando così le prestazioni. Tuttavia per questo caso particolare, può essere utilizzata anche la porta HP (*High*

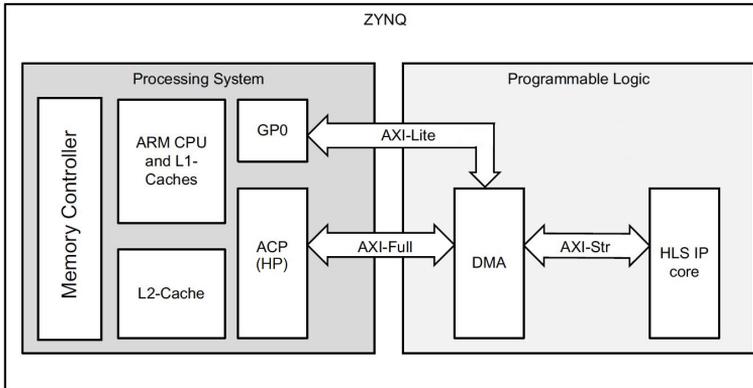


Figura 4.9. Schema a blocchi della comunicazione PS-PL [12].

Performance Port), anch'essa di tipo AXI fino a 64 bit ma con il vantaggio di avere due porte dedicate alla memoria DDR, nonché un collegamento con l'OCM. Essa fa uso di alcune FIFO sia in fase di scrittura che di lettura.

Nella figura 4.3.2 è possibile vedere il block diagram di Vivado completo. In esso si distinguono alcuni IP ausiliari atti a completare l'hardware da implementare su FPGA. Vengono utilizzati due IP AXI Interconnect, i quali servono a connettere uno o più master ad uno o più slave. In questo caso, infatti, essi si interpongono tra AXI DMA e PS che si comportano entrambi da master in uscita e da slave in ingresso.

I segnali di interrupt generati dall'AXI DMA vengono concatenati con un IP apposito e collegati alla porta *Interrupt Request* del PS, dove prima era collegato l'EOC proveniente dall'XADC Wizard. Quest'ultimo, per evitare conflitti con i due nuovi segnali di interrupt, è stato collegato ad una porta privata di *Interrupt Request* utilizzata solo dal core 1, ovvero il core dedicato all'acquisizione dei segnali analogici.

All'ingresso `ap_start` dell'IP generato da Vivado HLS, viene collegato una costante pari ad 1, in modo tale da avviare ad ogni reset l'hardware dedito all'accelerazione della prima parte dell'algoritmo di denoising. Questo stesso IP, oltre ai dati in uscita, produce il segnale `TLAST`, citato nel paragrafo 4.2.2, il quale viene connesso al corrispondente ingresso dell'AXI DMA.

Il PS genera due frequenze di clock distinte, una per l'XADC (frequenza di campionamento) e l'altra per l'hardware parallelo. Quest'ultima viene impostata in base al risultato ottenuto nella fase di sintesi RTL (par. 4.2.3) in merito al periodo di clock minimo. Si nota infine (sulla sinistra), l'IP che si occupa di resettare l'hardware parallelo su comando del PS.

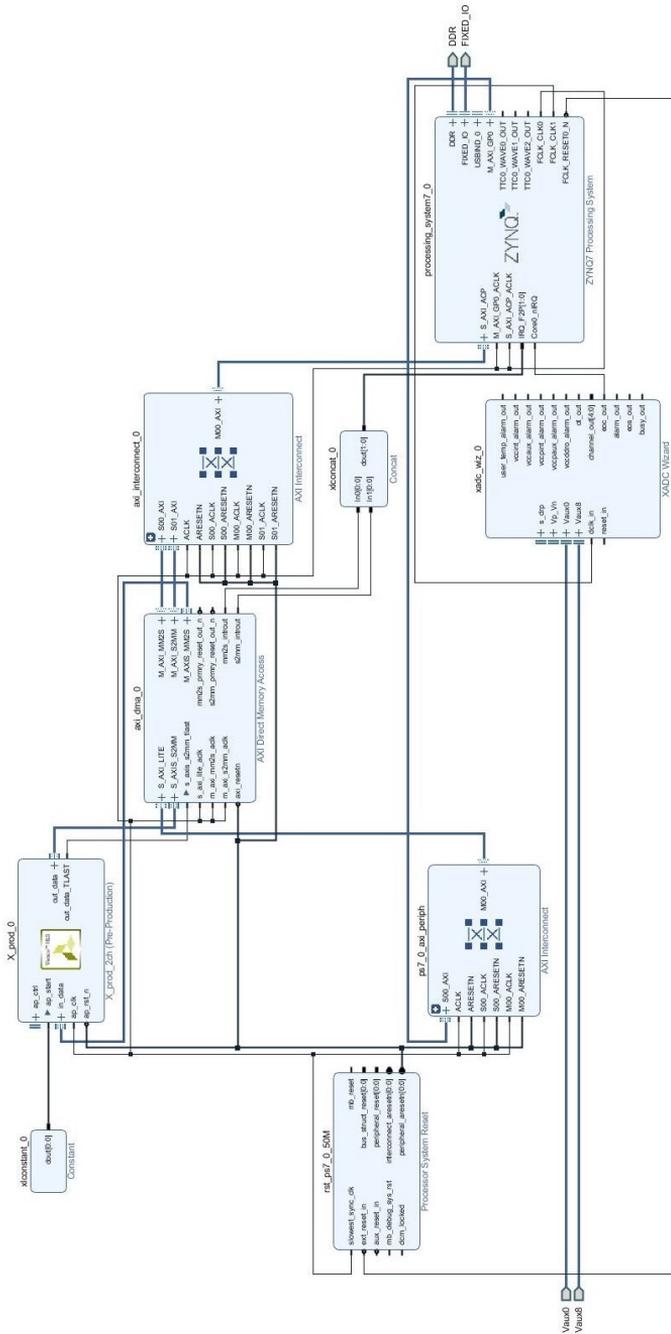


Figura 4.10. Block diagram di Vivado con PS, XADC Wizard e l'hardware parallelo.

Una volta completato e validato il design nel block diagram, si possono effettuare nell'ordine: sintesi, implementazione e generazione del bitstream. In quest'ultima fase viene verificata l'effettiva frequenza di clock massima che si può impostare all'hardware parallelo, che in questo caso risulta essere circa 76,93 MHz. Ciò vuol dire che l'effettivo periodo di clock minimo è di 13 ns.

A questo punto, l'hardware può essere esportato ed invocato in SDK.

4.3.3 Invocazione dell'hardware in SDK

Per rendere eseguibile l'accelerazione hardware è necessario modificare il codice sorgente del core 2. Le funzioni Eigen utilizzate per le prime tre parti dell'algoritmo di denoising, mostrate in figura 3.3, devono essere sostituite da funzioni che invocano l'hardware.

Per farlo, ed in generale, per poter richiamare le funzionalità relative all'IP AXI DMA serve configurare ed inizializzare i registri inerenti ad esso. Inoltre, bisogna abilitare gli interrupt anche per il core 2, i quali, come visto nel paragrafo 4.3.1, servono a richiamare l'attenzione del PS, interrompendolo ogni qual volta vengono prodotte delle uscite dai canali MM2S e S2MM dello stesso AXI DMA.

L'intervento dell'hardware viene invocato mediante due interrupt handler ed una *macro* predefinita dalle API di SDK relativa al DMA. La funzione in questione prende il nome di *XAxiDma_SimpleTransfer* e va semplicemente ad inviare o ricevere stream di dati, di lunghezza definita, verso o dall'AXI DMA. Ciò vuol dire che la funzione *XAxiDma_SimpleTransfer* viene chiamata una prima volta per inviare la finestra di campioni in ingresso all'hardware ed una seconda volta per ricevere i risultati parziali della matrice S e del relativo prodotto ($S = XX^T$). I due interrupt handler servono invece a segnalare al main quando realmente la trasmissione o la ricezione (verso o dall'AXI DMA) è stata completata, attraverso l'utilizzo di due flag.

```
for(unsigned int i = 0; i < (tot_size-param_1+1)/data_size; i++) //↔
    invoke HW (24 times)
{
    //input stream assembling
    for (unsigned int i = 0; i < (data_size+param_1)*2; i=i+2)
    {
        in[i] = in_data1[indi];
        in[i+1] = in_data2[indi];
        indi++;
    }
    indi=indi-param_1;
    ...
}
```

```
/* Flush the SrcBuffer before the DMA transfer , in case the Data ←
Cache is enabled */
Xil_DCacheFlushRange((u32)in.data(), (data_size+param_1)*2*←
sizeof(float));

//input stream transfer
Status = XAxiDma_SimpleTransfer(&AxiDma, (u32)in.data(), 2*(←
data_size+param_1)*sizeof(float), XAXIDMA_DMA_TO_DEVICE);

//output stream transfer
Status = XAxiDma_SimpleTransfer(&AxiDma, (u32)out.data(), 2*(←
param_1*param_1)*sizeof(float), XAXIDMA_DEVICE_TO_DMA);

...

/* Wait TX done and RX done */
while (!TxDone){

/* Invalidate the DestBuffer before checking the data, in case ←
the Data Cache is enabled */
Xil_DCacheInvalidateRange((u32)out.data(), 2*(param_1*param_1)*←
sizeof(float));

while (!RxDone){

...

//output stream disassembling
for (unsigned int i = 0; i < param_1; i++)
{
    for (unsigned int j = 0; j < param_1; j++)
    {
        X_mul(i, j) += out[ind_out];
        X_mul2(i, j) += out[ind_out+1];
        ind_out = ind_out+2;
    }
}
}
```

Codice 4.4. Loop con l'invocazione dell'hardware.

Le operazioni citate vengono ripetute 24 volte (24 finestre), fornendo progressivamente all'hardware tutti i campioni ed accumulando i risultati parziali su un'unica matrice S . Essa dopo i 24 cicli conterrà il risultato del prodotto $S = XX^T$ relativo ad un secondo di acquisizione (cioè a 32900 campioni). In questo modo, dopo circa 215 ms dall'acquisizione di un secondo di campioni su entrambi i canali, si può procedere al calcolo degli autovalori della matrice S ed all'esecuzione delle parti restanti dell'algoritmo di denoising, anziché aspettare 4,1 secondi per ogni canale come in precedenza.

4.3.4 Altre migliorie

La sola accelerazione hardware dell’inizializzazione e trasposizione della matrice traiettoria X e del prodotto $S = XX^T$ non è abbastanza. La parte finale dell’algoritmo di denoising, che comprende grouping e ricostruzione, impiega ancora circa 1,7 secondi per essere eseguita su ogni canale. Tuttavia, in questo caso, si possono ottenere risultati soddisfacenti eliminando alcune inefficienze all’interno del codice sorgente, senza ovviamente modificare l’algoritmo in sè.

Un esempio particolarmente significativo riguarda i due prodotti matriciali rimanenti, ovvero i prodotti 2 e 3 indicati nel paragrafo 4.1. Questi venivano eseguiti utilizzando la libreria Eigen richiamando in essi il calcolo della matrice unitaria U ad ogni iterazione, cosa poco efficiente. Il calcolo della matrice unitaria indicata è relativo alla sola matrice S e lo si può anticipare a prima dei due prodotti, eseguendolo una sola volta. Inoltre, può essere evitato anche l’uso della libreria Eigen per le due moltiplicazioni citate, sostituendo esse con semplici loop atti ad implementare i prodotti riga per colonna tra due matrici. Infine, in questi stessi loop può essere applicato un loop unrolling per velocizzare ulteriormente l’esecuzione dei prodotti in questione.

Queste migliorie unite ad altre analoghe per la parte di ricostruzione, ed alla parallelizzazione del codice (laddove possibile), permettendo così l’esecuzione quasi contemporanea dell’algoritmo su entrambi i canali, portano ad una cospicua riduzione del tempo di esecuzione delle parti di grouping e ricostruzione. Il tempo impiegato scende infatti da 1,7 secondi per ogni singolo canale a circa 600 ms per entrambi i canali, che portano il tempo di esecuzione complessivo dell’algoritmo di denoising a circa 815 ms, tempo molto più basso rispetto ai 5,8 secondi originari per singolo canale e che quindi abilita il real-time.

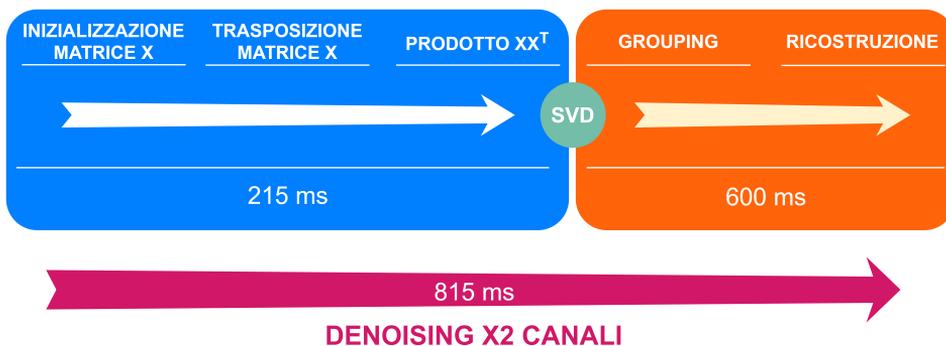


Figura 4.11. Tempi di esecuzione delle varie parti dell’algoritmo dopo l’accelerazione hardware e le migliorie software.

Questo risultato non solo permette di evitare di perdere campioni tra un filtraggio e l'altro ma anche di avere un discreto margine (oltre 180 ms) per implementare la visualizzazione dei segnali coinvolti.

4.3.5 Programmabilità della lunghezza della finestra

Un modo ulteriore di velocizzare l'algoritmo di denoising è quello di diminuire la lunghezza della finestra L , dato che tutti i calcoli sono influenzati direttamente da questo parametro.

Tuttavia, la programmabilità di L può essere implementata soltanto nell'ultima parte dell'algoritmo, ovvero quella correlata alla ricostruzione del vettore iniziale. Ciò è dovuto all'uso dell'accelerazione hardware per le fasi iniziali della SSA. L'hardware, infatti, una volta scritto sulla PL risulta permanente, ovviamente fino al prossimo reset o spegnimento.

Ne consegue che conviene lasciare $L = 21$ come limite superiore, andando poi a diminuirla soltanto prima della ricostruzione. In questo modo l'SVD, la normalizzazione e la prima parte del grouping fanno comunque uso delle informazioni raccolte nella matrice di covarianza S per $L = 21$, ma nella parte finale si andrà a ricostruire il segnale monodimensionale utilizzando una finestra più piccola.

Chiaramente una riduzione della finestra L porta ad una diminuzione dell'efficacia del filtraggio, ma se scelto con consapevolezza, anche in base al tipo di segnale in ingresso, permette di ottenere un altro significativo speed-up, garantendo comunque un filtraggio soddisfacente.

Nella tabella 4.1 è riportato il tempo di esecuzione totale dell'algoritmo di denoising al variare della lunghezza della finestra L .

L	Tempo di esecuzione
21	820 ms
18	785 ms
15	718 ms
12	660 ms
9	590 ms

Tabella 4.1. Tempo di esecuzione totale al variare di L .

Questa velocizzazione è dovuta perlopiù allo snellimento dell'ultimo prodotto matriciale, il terzo indicato nel paragrafo 4.1, l'unico per cui non è stato possibile utilizzare il loop unrolling.

Capitolo 5

Visualizzazione

Lo scopo dell'ultima parte del lavoro di tesi è quello di rendere fruibili su schermo i due segnali acquisiti in ingresso e la loro versione filtrata.

A questo scopo possono essere d'aiuto le due tipologie di interfacce video presenti sulla Zedboard, ovvero la VGA e l'HDMI. La prima ha una risoluzione video massima nettamente inferiore alla seconda (480p contro i 1080p dell'HDMI), tuttavia la sua gestione è immediata. L'interfaccia VGA, infatti, necessita di un semplice controller dedicato (hardware minimo) e di poche configurazioni (software), mentre l'HDMI può essere gestito solo mediante il transceiver ADV7511, presente sulla board, attraverso l'interfaccia I^2C .

In questo tipo di applicazione non si andranno ad utilizzare schermi particolarmente grandi, inoltre i segnali coinvolti richiedono di essere visualizzati in maniera qualitativa. Non avendo elevate esigenze risolutive è quindi preferibile utilizzare l'interfaccia video VGA.

5.1 VGA

Lo standard *Video Graphics Array* (VGA) è uno standard video analogico introdotto nel 1987. Esso consente di visualizzare immagini video con risoluzione massima a 640×480 pixel, formato comunemente noto con la stessa dicitura VGA. È anche possibile visualizzare immagini video a risoluzioni minori (fino a 320×200 pixel), mentre per risoluzioni più alte esistono versioni più evolute di questo standard, tra cui l'SVGA (800×600 pixel), l'XGA (1024×768 pixel) e l'SXGA (1280×1024 pixel).

Nel suo formato classico (DE-15), il connettore VGA utilizza 15 pin descritti nella tabella 5.1.

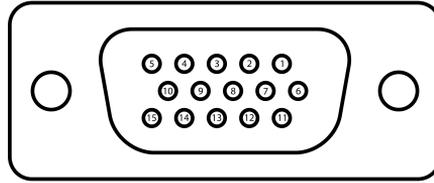


Figura 5.1. Pin del connettore DE-15 VGA.

Di questi 15 pin, 6 (Rosso, Verde, Blu) sono addetti alla generazione della palette colori RGB, formata in questo caso da 256 tonalità diverse per ognuno dei tre colori, e 4 sono necessari alla temporizzazione verticale (VSync) ed orizzontale (HSync). I restanti sono obsoleti o non di interesse per questa applicazione.

Pin	Descrizione
1	Rosso (video)
2	Verde (video)
3	Blu (video)
4	Ex bit 2 (monitor)
5	Massa (HSync)
6	Rosso (massa)
7	Verde (massa)
8	Blu (massa)
9	5V (EEPROM)
10	Massa (VSync)
11	Ex bit 0 (monitor)
12	Ex bit 1 (monitor)
13	HSync
14	VSync
15	Ex bit 3 (monitor)

Tabella 5.1. Descrizione dei pin del connettore DE-15 VGA.

I colori sono trasmessi in segnali analogici attraverso i pin RGB con tensioni comprese tra 0 V e 0,7 V, in cui il valore più alto corrisponde al massimo della luminosità. Più livelli si hanno in questo intervallo, ottenuti grazie a delle reti resistive o a dei DAC, maggiore sarà il numero di combinazioni cromatiche. I segnali di sincronismo, invece, sono dei segnali digitali ad impulsi attivi bassi. I dati relativi all'immagine sono inviati con una scansione orizzontale, da sinistra a destra, ed una verticale, dall'alto verso il basso. Dopo la fine di ogni linea (insieme di pixel) segue un impulso di sincronismo orizzontale (HSync), mentre dopo la fine

di ogni frame o quadro (insieme di linee) segue un impulso di sincronismo verticale (VSync). Il segnale verticale quindi è molto più lento (figura 5.2).

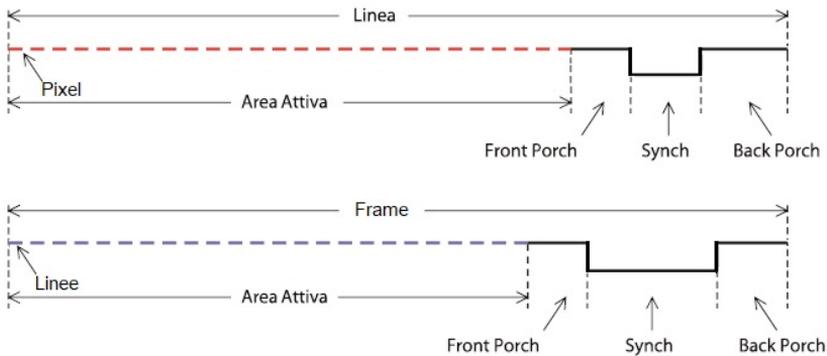


Figura 5.2. Temporizzazione orizzontale e verticale.

Per entrambe le temporizzazioni è presente un'area attiva, in cui sono presentati alle uscite i valori di colore associati a ciascun pixel, o alle varie linee. Terminata la zona attiva si ha un intervallo chiamato "pedistallo anteriore" (*Front Porch*), poi l'impulso di sincronismo, ed un "pedistallo posteriore" (*Back Porch*). Dall'inizio del front porch al nuovo inizio dell'area attiva i segnali di colore devono essere spenti (portati a 0V). Questo è necessario perché in questo tempo, chiamato intervallo di *horizontal blank*, il pennello elettronico (nei monitor CRT) deve spostarsi dal bordo destro a quello sinistro per ricominciare la scansione della linea successiva. Molti dispositivi video, anche non basati su CRT, non funzionano correttamente se non si spengono i segnali di colore durante i blank. Per le temporizzazioni verticali si ha una situazione simile, solo che in questo caso si contano le linee tracciate e non i pixel. Dopo un certo numero di linee (dipendente dalla risoluzione utilizzata) si avrà un intervallo di *vertical blank*, che comprenderà i due porch e l'impulso di sincronismo.

Le ampiezze delle aree attive, dei porch e degli impulsi di temporizzazione sono espresse in numero di pixel per le grandezze orizzontali ed in numero di linee per le grandezze verticali, e dipendono dallo standard di risoluzione utilizzato. Quello di riferimento da ora in poi sarà lo standard VGA 640×480 con frequenza di aggiornamento delle immagini pari a 60 Hz (numero di frame al secondo).

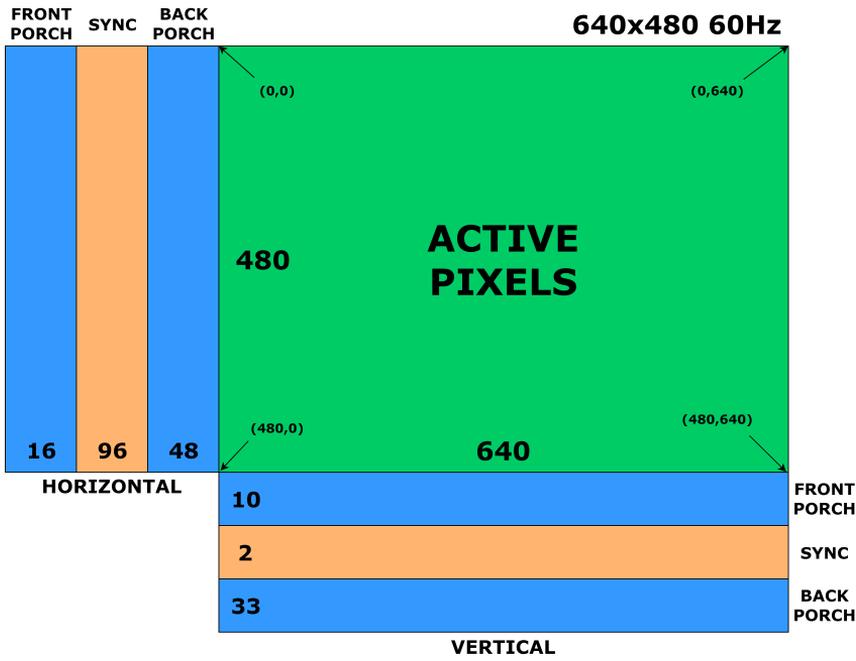


Figura 5.3. Caratteristiche ideali della risoluzione standard 640×480 a 60 Hz.

La banda caratteristica dei segnali RGB può essere calcolata approssimativamente moltiplicando il numero di pixel orizzontali per quelli verticali, per il numero di frame al secondo e per il tempo aggiuntivo impiegato per il *retrace* (ritraccia), che conta per un fattore compreso tra 1,2 e 1,4.

$$640 \times 480 \times 60 \times 1.3 \approx 25 \text{ MHz}$$

Questa frequenza è il cosiddetto *pixel clock*, cioè la velocità con cui i dati di colore relativi ad ogni pixel devono essere presentati alle uscite. Si intuisce quindi che la generazione di un segnale VGA richiede hardware dedicato.

5.1.1 Controller VGA

La generazione dei segnali di colore (RGB) e di quelli di temporizzazione necessita di un componente hardware (IP) dedicato, noto comunemente come controller VGA. Per realizzare questo nuovo IP in Vivado, essendo esso non particolarmente complesso, lo si può descrivere direttamente in HDL (Verilog in questo caso). Il controller VGA (figura 5.4) riceve in ingresso i dati relativi ai pixel provenienti dal Video-DMA (par. 5.2) con un parallelismo di 32 bit, il reset asincrono (attivo

basso) ed il clock con un valore ideale di 25,175 MHz. Tuttavia, il controller deve poter comunicare con gli IP a monte che fanno uso del protocollo AXI4-Stream. Per questo motivo, il controller ha una porta TVALID che permette di discernere i dati in ingresso (pixel) validi ed un'uscita TREADY per segnalare la generazione di pixel relativi all'area attiva. Come già accennato, l'IP in uscita deve fornire i segnali di temporizzazione (VSync e HSync) e quelli di colore (RGB). Questi ultimi constano di 3 segnali (rosso, verde e blu) ognuno con un parallelismo di 4 bit (4 livelli), in maniera tale da generare 256 possibili combinazioni per ognuno dei tre colori, per un totale di più di 16 milioni di colori (256^3).

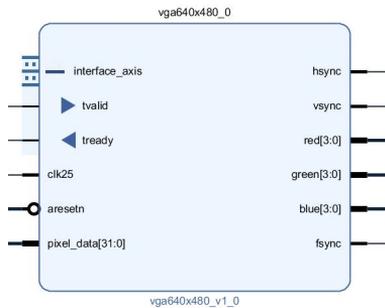


Figura 5.4. IP Vivado del controller VGA.

Sia i segnali RGB che quelli di temporizzazione vengono assegnati, mediante i constraint, ai PIN corrispondenti della logica programmabile, ognuno alimentato con una tensione di 3,3 V.

Infine si nota l'uscita fsync, la quale andrà ad essere collegata alla porta MM2S Frame Sync Input (`mm2s_fsync`) del VDMA. Questa porta, quando abilitata, obbliga il VDMA a triggerare le proprie operazioni sul fronte di discesa del segnale connesso ad essa (fsync), il quale deve essere asserito per un solo ciclo di clock del canale MM2S.

Il funzionamento del controller VGA può essere racchiuso in tre semplici *process* che fanno uso dei segnali citati e di diverse costanti che vanno a definire lo standard di risoluzione utilizzato. Esempi di questi parametri sono il numero di pixel per linea orizzontale, il numero di linee verticali per frame e la lunghezza degli impulsi di temporizzazione. Questi parametri vengono riportati nella tabella 5.2.

Il primo process, sincronizzato sul fronte di salita del clock, gestisce i due contatori utilizzati per tenere traccia di dove si è sullo schermo. Uno è dedicato ai pixel e si azzerà alla fine di ogni linea, mentre l'altro conta le linee e si azzerà alla fine di ogni frame. In base ai due contatori, viene assegnato 0 o 1 ai segnali di temporizzazione HSync e VSync.

Parametro	Valore
Pixel per linea	800
Linee per frame	521
Lunghezza dell'impulso HSync	96
Lunghezza dell'impulso VSync	2
Fine del back porch orizzontale	144
Inizio del front porch orizzontale	784
Fine del back porch verticale	31
Inizio del front porch verticale	511

Tabella 5.2. Parametri ideali del controller VGA 640 × 480.

```

// registers for storing the horizontal & vertical counters
reg [9:0] hc;
reg [9:0] vc;

always @(posedge clk25)
begin
    // reset condition
    if (~aresetn)
    begin
        hc <= 0;
        vc <= 0;
    end
    else
    begin
        // keep counting until the end of the line
        if (hc < hpixels - 1)
            hc <= hc + 1;
        else
            // When we hit the end of the line, reset the horizontal ←
            counter and increment the vertical counter.
            // If vertical counter is at the end of the frame, then reset ←
            that one too.
            begin
                hc <= 0;
                if (vc < vlines - 1)
                    vc <= vc + 1;
                else
                    vc <= 0;
            end
    end
end
assign hsync = (hc < hpulse) ? 0:1;
assign vsync = (vc < vpulse) ? 0:1;

```

Codice 5.1. Primo processo del controller VGA.

Il secondo process, la cui sensitivity list comprende i due contatori appena citati, genera i tre segnali RGB assegnando ad ognuno di essi 4 dei 32 bit in ingresso. Questo avviene solo quando si è nell'area attiva (TREADY = 1), altrimenti i segnali RGB in uscita vengono impostati a zero (TREADY = 0).

```
always @(*)
begin
  // first check if we're within vertical active video range
  if (vc >= vbp && vc < vfp)
  begin
    // now display different colors every 80 pixels
    // while we're within the active horizontal range
    // -----
    // display
    if (hc >= hbp && hc < (hbp+640))
    begin
      red = pixel_data[23:20];
      green = pixel_data[15:12];
      blue = pixel_data[7:4];
      tready = 1;
    end
    // we're outside active horizontal range so display black
  else
  begin
    red = 0;
    green = 0;
    blue = 0;
    tready = 0;
  end
end
// we're outside active vertical range so display black
else
begin
  red = 0;
  green = 0;
  blue = 0;
  tready = 0;
end
end
```

Codice 5.2. Secondo processo del controller VGA.

L'ultimo process, sincronizzato sul fronte di salita del clock, si occupa di asserire il segnale fsync, cosa che avviene solo quando il segnale di temporizzazione verticale è a 1 dopo esser stato precedentemente asserito (attivo basso), condizione che indica l'inizio di un nuovo frame.

```
reg vsync_last;
```

```
always@(posedge clk25)
begin
  if(~aresetn)
  begin
    vsync_last <= 0;
    fsync <= 0;
  end
  else
  begin
    vsync_last <= vsync;
    if (~vsync_last & vsync)
    begin
      fsync <= 1;
    end
    else
    begin
      fsync <= 0;
    end
  end
end
end
```

Codice 5.3. Terzo processo del controller VGA.

5.2 AXI VDMA ed integrazione

Il controller VGA non è in grado di prelevare autonomamente i dati dalla memoria DDR. Serve un meccanismo analogo a quello dell'AXI DMA ma più performante, adatto ai segnali video.

La soluzione adatta è l'IP predefinito di Vivado noto come AXI VDMA (*Video Direct Memory Access*) [13], il cui funzionamento è del tutto analogo a quello dell'AXI DMA, visto nel paragrafo 4.3.1. La differenza è che questo IP è pensato per trattare dati di tipo AXI4-Stream Video e quindi a banda più larga, cosa che implica maggiori risorse hardware. Rispetto all'AXI DMA utilizzato per l'accelerazione dell'algoritmo di denoising, non è necessario abilitare il canale di scrittura nella DDR, in quanto l'AXI VDMA deve solo leggere il frame buffer dalla memoria per poi inviarlo all'uscita VGA. Dalle impostazioni dell'IP si può inoltre scegliere il numero di frame buffer da utilizzare (1 per questa applicazione).

In figura 5.5 è riportato il block diagram di Vivado con l'hardware dedicato alla visualizzazione VGA. Ancora una volta sono necessari alcuni IP ausiliari per collegare al meglio gli IP principali (AXI VDMA, PS e controller VGA). Come per l'AXI DMA, anche il VDMA riceve in ingresso i segnali di controllo provenienti dal PS in formato AXI4-Lite mediante l'IP AXI Interconnect.

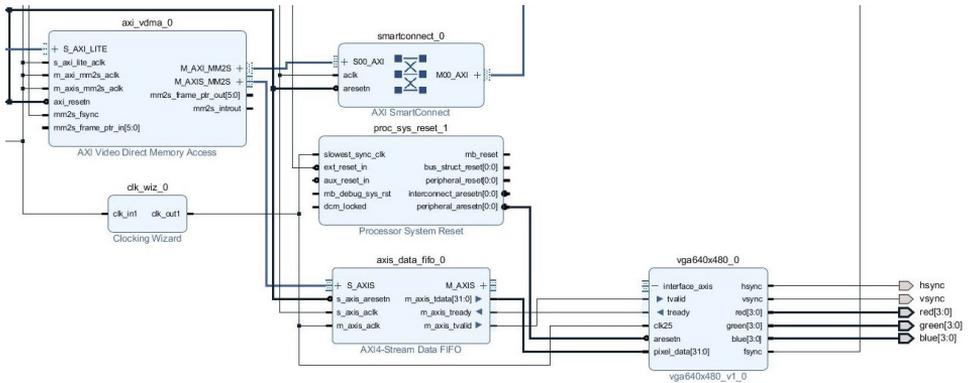


Figura 5.5. Block diagram di Vivado con gli IP aggiunti per la visualizzazione VGA.

I frame buffer in uscita dalla memoria DDR e quindi dalla porta HP0 (*High Performance Port*) del PS, entrano invece nel canale MM2S dell'AXI VDMA mediante un IP AXI SmartConnect. Quest'ultimo a differenza del semplice IP AXI Interconnect, ha la peculiarità di adattare autonomamente, senza l'intervento dell'utente, il bus master a quello slave e viceversa, modificando a seconda degli interlocutori il parallelismo. L'IP Clocking Wizard genera il pixel clock destinato al controller VGA, convertendo il clock generato dal PS da 100 MHz in una frequenza di clock da 25,173 MHz (valore più vicino a quello ideale). Per questa ragione, all'interno del block diagram si notano due diversi IP relativi al reset, uno dedicato a gestire tutte le periferiche che lavorano alla frequenza di clock da 100 MHz (come in precedenza) e un altro dedicato al controller VGA. Questa differenza in termini di frequenza di lavoro viene gestita attraverso l'IP AXI4-Stream Data FIFO. Esso regola il flusso di dati tra AXI VDMA e controller VGA, ricevendo in ingresso i frame buffer provenienti dal canale MM2S dell'AXI VDMA, ad una frequenza di 100 MHz, e generando gli stream AXI da 32 bit, a 25 MHz, destinati all'ingresso dei dati (pixel) del controller VGA.

In figura 5.2 viene illustrato il block diagram di Vivado con la parte di visualizzazione integrata all'hardware precedentemente implementato. L'integrazione dei nuovi IP con quelli già presenti è possibile grazie all'uso di due distinte porte in ingresso al PS, una per ognuno dei due task: l'ACP per l'algoritmo di denoising e l'HP0 per la visualizzazione. In uscita dal PS vi è però un'unica porta, la GP0. Per trasmettere i controlli AXI4-Lite alle due parti hardware si può quindi utilizzare un AXI IP Interconnect con un unico ingresso connesso alla GP0 del PS (master) e due uscite (slave) connesse all'AXI DMA e all'AXI VDMA. La crossbar (sistema di commutazione) presente in questo IP eviterà conflitti tra i due canali.

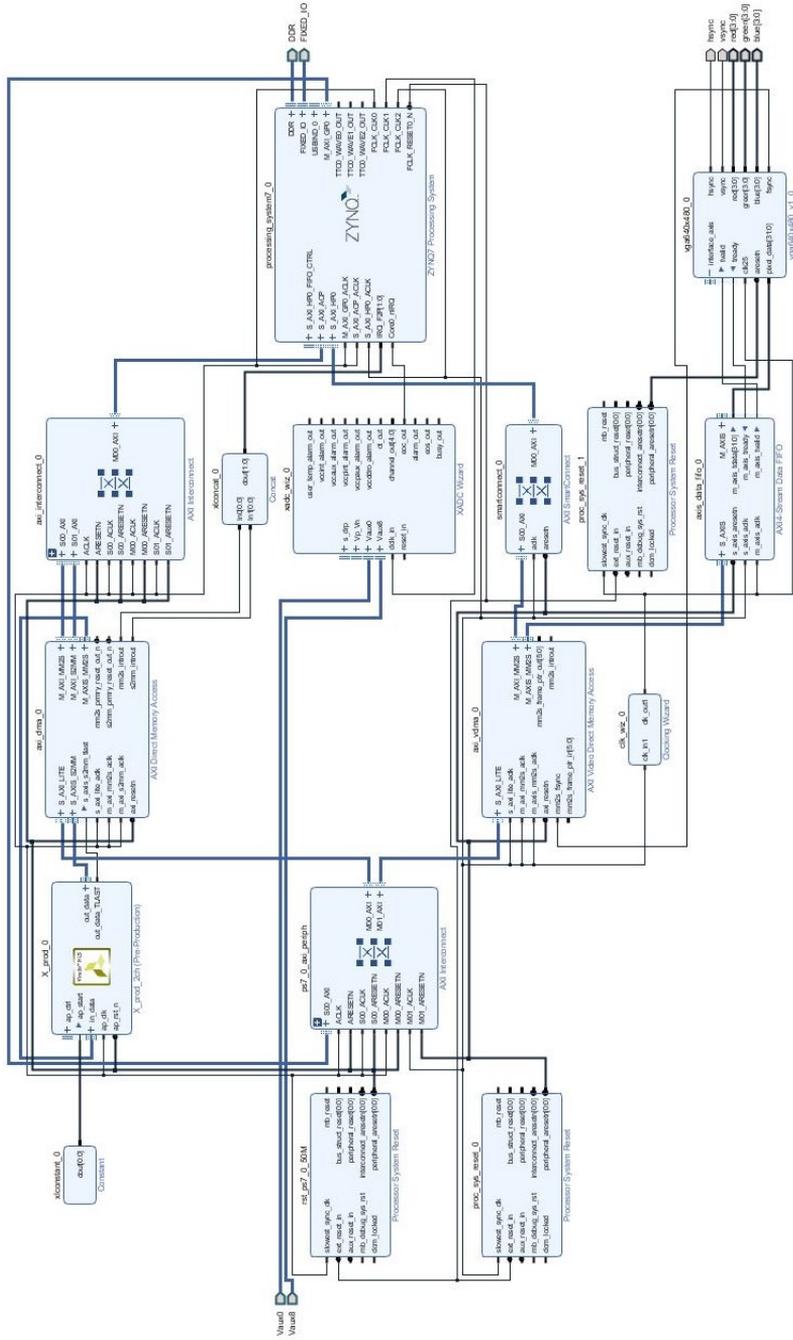


Figura 5.6. Block diagram di Vivado completo di PS, XADC Wizard, accelerazione hardware e visualizzazione VGA.

Il nuovo schematico della PL può essere a questo punto validato, sintetizzato, implementato ed esportato in SDK.

5.3 Impostazioni di visualizzazione

Come già accennato nel paragrafo 3.3, la visualizzazione è gestita dal core 2, il quale ha a disposizione i segnali di ingresso e di uscita di entrambi i canali, raggruppati in quattro diversi vettori da 32900 campioni l'uno (1 secondo di acquisizione). Una volta importato l'hardware in SDK, si va ad inizializzare il VDMA impostando alcuni parametri, tra cui il base address del frame buffer ed il numero di righe e di colonne del display. Questa prima configurazione verrà eseguita una sola volta dall'applicazione, a differenza delle prossime operazioni che si andranno ad effettuare su VGA.

```

*(VDMA_VGA_CR) = 0x1; // Enable VDMA
*(VDMA_VGA_AD) = 0x10000000; // The base address of the frame↵
buffer
*(VDMA_VGA_ST) = 4*DISPLAY_COLUMNS; // Line size (stride value)
*(VDMA_VGA_HS) = 4*DISPLAY_COLUMNS; // Horizontal Pixel Count * 4↵
B/pixel
*(VDMA_VGA_VS) = DISPLAY_ROWS; // Vertical Line Count

```

Codice 5.4. Inizializzazione VDMA.

A questo punto, eseguendo l'applicazione comparirebbe su schermo la classica schermata grigia dei vecchi CRT. Per garantire una visualizzazione chiara e coerente con i segnali coinvolti ci si può rifare al modello di schermata utilizzato sui moderni oscilloscopi digitali, ovvero creare uno sfondo, di solito scuro, con una griglia che permetta di identificare i V/div ed i s/div. La differenza principale sarà quella di avere quattro quadranti, uno per ogni buffer, che andranno a dividere lo schermo in parti uguali, invece di avere un unico quadrante dove sovrapporre i vari canali. Ciò risulta attuabile viste le dimensioni medie di uno schermo con VGA, tendenzialmente molto maggiori degli schermi che montano gli oscilloscopi. Un'altra sostanziale differenza sta nel refresh dei segnali a video, i quali saranno aggiornati solo alla fine dell'esecuzione dell'algorithm di denoising. Questo porta ad un refresh della schermata ogni secondo, tempo molto più alto di quello degli oscilloscopi, dove è necessario visualizzare i segnali pressoché in tempo reale. Questo tipo di struttura video conduce a ripartire la visualizzazione in tre funzioni C, eseguite in ordine dopo ogni filtraggio:

- visualizzazione dello sfondo, dei quadranti, della griglia e dei caratteri;
- visualizzazione dei due canali di ingresso;
- visualizzazione dei due canali di uscita.

Ognuna di queste funzioni fa uso di alcune costanti, espresse in esadecimale, che indicano una particolare tonalità (figura 5.7). Per visualizzare un colore su un pixel dello schermo, basta assegnarlo ad un certo indirizzo costruito partendo dal base address del frame buffer, sommando ad esso due indici per identificare una particolare posizione sullo schermo, come se si trattasse di una semplice matrice espressa con l'aritmetica dei puntatori.

NAVY #001f3f	BLUE #0074D9	AQUA #7FDBFF	TEAL #39CCCC	OLIVE #3D9970	GREEN #2ECC40	LIME #01FF70	YELLOW #FFDC00
ORANGE #FF851B	RED #FF4136	MAROON #85144b	FUCHSIA #F012BE	PURPLE #B10DC9	BLACK #111111	GRAY #AAAAAA	SILVER #DDDDDD
WHITE #FFFFFF							

Figura 5.7. Codici in esadecimale di alcuni degli oltre 16 milioni di colori possibili.

La prima funzione si occupa di creare lo sfondo sul quale verranno visualizzati i segnali, dando modo a chi effettua una lettura di uno di essi di capire qual è il canale in questione, se si tratta di un ingresso o di un'uscita e quali valori di tensione assume nei vari istanti di tempo. Per farlo, viene generata una schermata completamente nera su cui vengono proiettati i due assi principali (in bianco), uno verticale e l'altro orizzontale per dividere lo schermo in quattro quadranti, nonché la griglia (grigio chiaro) che va a definire le divisioni per ogni quadrante. Inoltre, vengono proiettati i caratteri (in bianco) che indicano i mV/div, i s/div, il canale e la direzione (ingresso o uscita). Questa schermata viene ridefinita ad ogni filtraggio, pulendola dai segnali proiettati in precedenza che altrimenti si andrebbero ad accumulare rendendo impossibile la corretta visualizzazione. Tuttavia questa operazione, ripetuta ogni secondo, andrebbe a creare un fastidioso effetto "flicker" delle varie componenti proiettate sullo sfondo nero (griglia, assi e caratteri). Per evitarlo vengono proiettate prima le varie componenti "chiare", dopodiché si va a generare lo sfondo nero che riempirà solo i pixel non coinvolti nel passaggio precedente. In questo modo si dà la percezione all'utente di avere una griglia e dei quadranti permanenti, mentre i segnali in sovrapposizione variano continuamente.

```
volatile unsigned int *pStorageMem = (unsigned int *)0x10000000; //↔
    frame buffer init
```

```

//horizontal grid lines
for(unsigned int r = 24; r < 480; r=r+24)
{
    if(r != 240 )
    {
        for(unsigned int c = 3; c < 639; c=c+8)
        {
            *(pStorageMem+(r*640)+c) = SOFT_GRAY;
            *(pStorageMem+(r*640)+c+1) = SOFT_GRAY;
            *(pStorageMem+(r*640)+c+2) = SOFT_GRAY;
            *(pStorageMem+(r*640)+c+3) = SOFT_GRAY;
            if (*(pStorageMem+(r*640)+c+4) != WHITE) *(pStorageMem+(r←
                *640)+c+4) = BLACK;
            if (*(pStorageMem+(r*640)+c+5) != WHITE) *(pStorageMem+(r←
                *640)+c+5) = BLACK;
            if (*(pStorageMem+(r*640)+c+6) != WHITE) *(pStorageMem+(r←
                *640)+c+6) = BLACK;
            if (*(pStorageMem+(r*640)+c+7) != WHITE) *(pStorageMem+(r←
                *640)+c+7) = BLACK;
        }
    }
}

//vertical grid lines

...

//characters
char_fix(); //permanent characters
char_var1(); //variable characters ch1
char_var2(); //variable characters ch2

//blank screen and axes
for(unsigned int r = 0; r < 480; r++)
{
    for(unsigned int c = 0; c < 640; c++)
    {
        if(r == 240 || r == 0 || r == 479 || c == 320 || c == 0 || c ←
            == 639) *(pStorageMem+(r*640)+c) = WHITE;
        else {if(*(pStorageMem+(r*640)+c) != WHITE && *(pStorageMem+(r←
            *640)+c) != SOFT_GRAY && *(pStorageMem+(r*640)+c) != ←
            DARK_WHITE) *(pStorageMem+(r*640)+c) = BLACK;}
    }
}

```

Codice 5.5. Generazione dello sfondo.

I caratteri vengono proiettati utilizzando un approccio di tipo vettoriale, dove ogni lettera o numero corrisponde ad una precisa configurazione di pixel accesi (in bianco) e altri spenti (in nero) (fig. 5.8).

I caratteri generati vanno ad indicare:

- mV/div (variabile);

- ms/div (fisso);
- canale (fisso);
- ingresso-uscita (fisso);
- valore centrale in mV (variabile);
- valore centrale in s (fisso).



Figura 5.8. Grafica vettoriale adottata.

La funzione di visualizzazione dei due segnali di ingresso sovrascrive questi ultimi ai pixel relativi allo sfondo appena generato. Tra un canale e l'altro cambierà l'offset di partenza che va sommato al base address e che consente di posizionare il segnale nel quadrante corretto. Chiaramente, non è possibile proiettare un solo campione acquisito su ogni singola colonna di pixel del quadrante. Infatti, i campioni sono 32900 mentre il numero di colonne di pixel per ogni quadrante è pari a 320 ($640 \div 2$). Questo vuol dire che ogni segnale deve essere proiettato scrivendo 103 campioni ($32900 \div 320$) per colonna. Ciò viene attuato prelevando dal buffer dei segnali un campione ogni 103, proiettandolo nella prima colonna di pixel vuota, così fino alla fine del buffer. Una volta arrivati in fondo al quadrante si riparte da sinistra prelevando sempre un campione ogni 103 ma "shiftato" di una posizione verso destra rispetto a prima, e così via (figura 5.9). In questo modo si ottiene una visualizzazione abbastanza fedele dei segnali pur non avendo a disposizione una colonna di pixel per ogni campione.

Per le righe di pixel, invece, il problema non sussiste perché esse bastano a rappresentare ogni valore di tensione distinguendo essi fino all'ordine dei mV. Questo perché per convertire il valore di tensione in indice di riga di pixel basta effettuare una proporzione del tipo $120 : 0,400 = in : x$, ovvero 120 righe di pixel (mezza altezza di un quadrante) stanno a 400 mV (se il range è $0 \div 800$ mV) come il valore di tensione del campione sta all'indice di riga di pixel corrispondente. Ad x viene

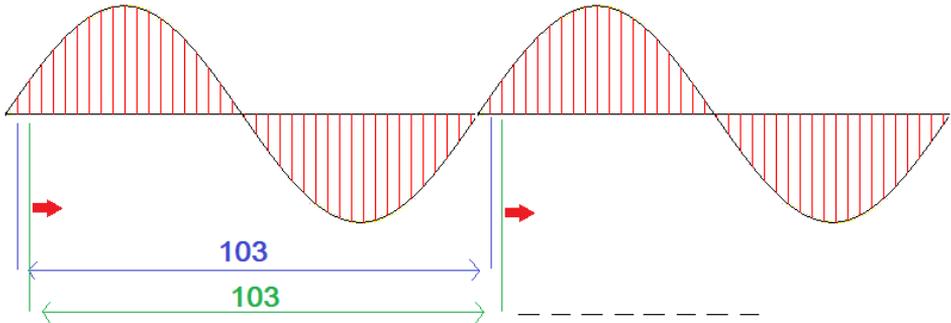


Figura 5.9. Principio di funzionamento della proiezione dei segnali con VGA.

poi sommato l'eventuale offset per posizionare il segnale nel quadrante corretto alla giusta altezza.

```

void plot_input(float *in_data, unsigned int size, float *in_data2, ←
               unsigned int size2)
{
    int center = DISPLAY_ROWS/4;
    //voltage middle point
    float mid1 = (float)((MAX1-MIN1)/2)/1000;
    float mid2 = (float)((MAX2-MIN2)/2)/1000;
    //pixel middle point
    int level1 = center/mid1;
    int level2 = center/mid2;
    //pixel offset
    int offset_n1 = ((float)MIN1/1000) * level1;
    int offset_n2 = ((float)MIN2/1000) * level2;

    volatile unsigned int *pStorageMem = (unsigned int *)0x10000000; ←
        //frame buffer init
    pStorageMem = pStorageMem + (640*480); //decreasing counter

    //print waveform
    for(unsigned int ind_up = 0; ind_up < 104; ind_up++)
    {
        unsigned int cnt = 638;
        for(unsigned int i = ind_up; i < size; i = i+104)
        {
            if ( (((int)(in_data[i]*level1-offset_n1) +240)*640)+cnt >←
                153600) *(pStorageMem-((( (int)(in_data[i]*level1-←
                offset_n1) +240)*640)+cnt)) = YELLOW; //CHI
        }
    }
}

```

```

if ((( (int)(in_data2[i]*level2-offset_n2) *640)+cnt) < ←
    153600) *(pStorageMem-((( (int)(in_data2[i]*level2-←
        offset_n2) *640)+cnt)) = FUCHSIA; //CH2
    cnt--;
}
}
}

```

Codice 5.6. Proiezione dei segnali di ingresso.

Tutta questa parte, compresa la generazione dei caratteri, è resa parametrica in termini di mV/div attraverso l'uso di due parametri per canale: valore massimo e minimo di tensione visualizzato. Questo permette di andare a visualizzare meglio i segnali che oscillano in range di tensione più ristretti, in maniera indipendente tra i due canali. I valori possibili che si possono inserire vanno da 0 a 1000 mV con un passo di 50 mV. Se vengono inseriti valori di tensione diversi da questi, oppure valori per i minimi superiori a quelli dei massimi, la visualizzazione con VGA non avviene. La restante parte dell'applicazione va comunque avanti ma con in più una segnalazione del problema in tempo reale, attraverso l'UART.

Tutto ciò è ripetuto in maniera analoga per la visualizzazione delle uscite.

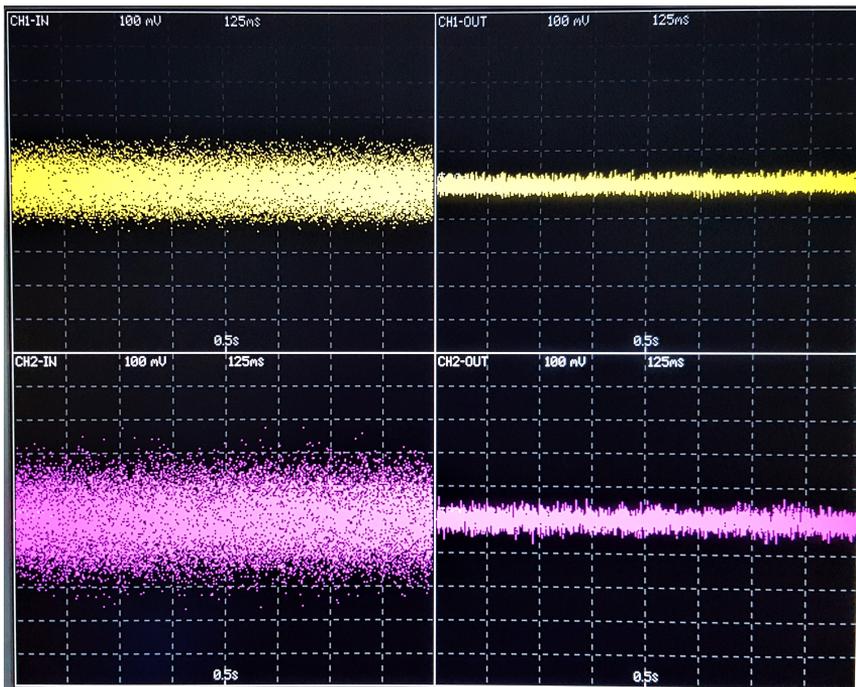


Figura 5.10. Esempio di visualizzazione VGA con mV/div diversi tra i due canali.

Le funzioni di visualizzazione viste (sfondo, ingressi e uscite) vengono richiamate, nell'ordine, dal main del core 2 dopo ogni filtraggio. Esse impiegano complessivamente circa 55 ms, perlopiù dovuti alla generazione dello sfondo ed alla parametrizzazione di alcuni dei caratteri citati (mV).

Il risultato è mostrato in figura [5.10](#).

Parte III

Risultati sperimentali

Capitolo 6

Analisi quantitativa del filtraggio

Al fine di mostrare come effettivamente l'algoritmo lavora sul nuovo sistema viene effettuata una piccola analisi quantitativa del filtraggio. Per farlo sono stati presi sotto esame due possibili casi:

- un segnale rumoroso completamente aleatorio;
- segnale noto (sinusoidale) con rumore sovrapposto.

Per generare i due segnali è stato utilizzato un generatore di funzione `Rigol DG1022` ed in entrambi i casi si è evitato di andare troppo vicini al range massimo di tensione dell'XADC ($0 \div 1V$), al fine di scongiurare qualsiasi danneggiamento dello stesso, data l'aleatorietà dei segnali in questione.

I segnali sono stati acquisiti in digitale mediante SD Card, scrivendo in due file separati (.txt) il frame in ingresso e quello in uscita, in modalità one shot. I dati memorizzati sono poi stati analizzati con l'ausilio di `Matlab`, generando i plot di seguito riportati.

Il primo caso è quello di un segnale puramente rumoroso (figura 6.1). Già nel dominio del tempo si nota una notevole "pulizia" visiva del segnale in uscita, che oscilla in un range di tensioni di ($0.4 \div 0.6V$), invece degli ($0.1 \div 0.8V$) dell'ingresso. Tuttavia, per avere un'idea più quantitativa del filtraggio conviene passare al dominio della frequenza, analizzando la densità spettrale di potenza in dB.

In `Matlab` è stato generato lo spettro di potenza facendo uso della FFT (Fast Fourier Transform), la quale è stata applicata su ogni segnale per poi proiettare il risultato in dB, in funzione della frequenza.

Nel grafico in figura 6.2 risulta evidente il grande abbattimento delle componenti spettrali a frequenze più elevate, arrivando a picchi di oltre 60 dB in termini di

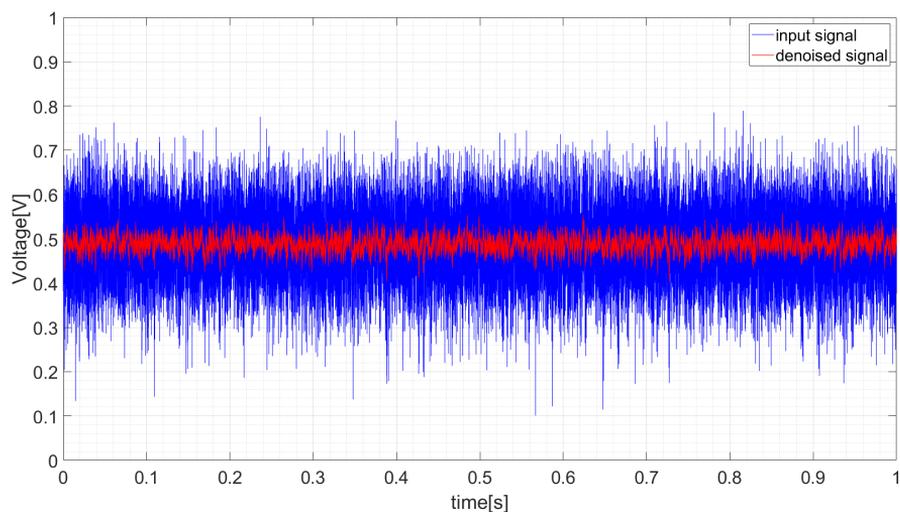


Figura 6.1. Segnale in ingresso (blu) e la sua versione filtrata (rosso).

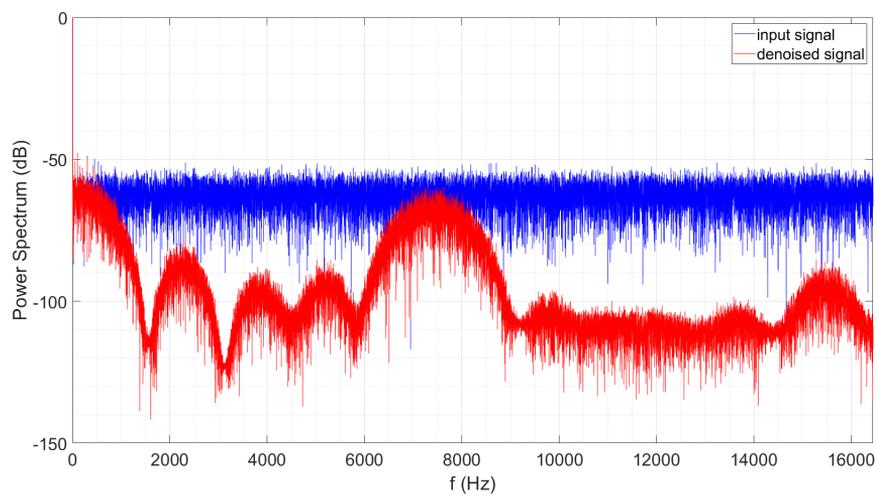


Figura 6.2. Potenza spettrale del segnale in ingresso (blu) e del segnale filtrato (rosso).

riduzione della potenza spettrale per alcune frequenze.

In figura 6.3 è possibile vedere la densità spettrale di potenza per le componenti in

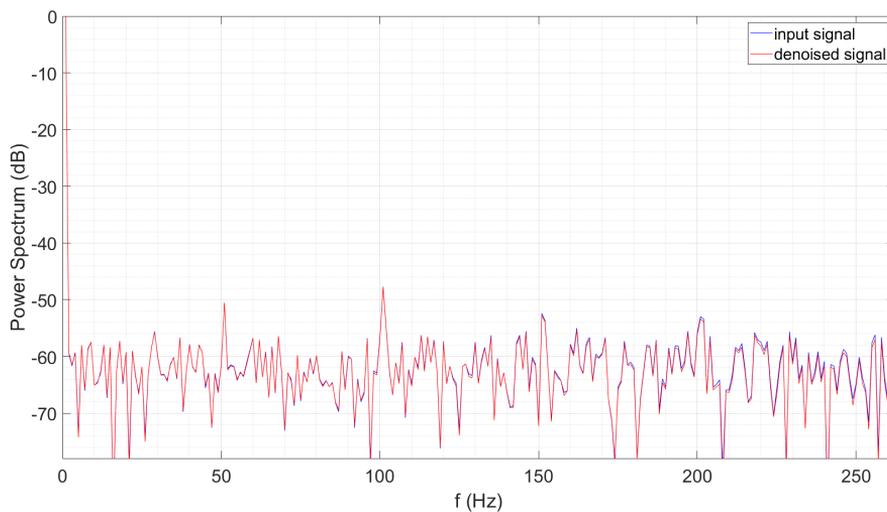


Figura 6.3. Zoom della potenza spettrale del segnale in ingresso (blu) e del segnale filtrato (rosso).

frequenza più basse e più vicine alla componente continua. Focalizzando l'attenzione su queste componenti, si nota come l'abbattimento cominci ad essere evidente da circa 200 Hz in poi.

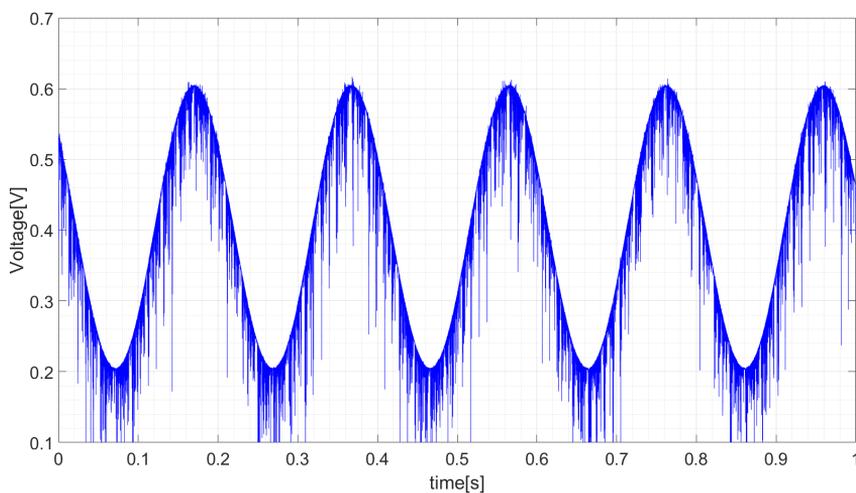


Figura 6.4. Segnale sinusoidale rumoroso in ingresso.

La stessa analisi è stata effettuata per un segnale sinusoidale in ingresso all'XA-DC con frequenza di 5 Hz e ampiezza picco-picco di 400 mV, a cui è sovrapposto del rumore (figura 6.4). Il risultato del filtraggio è mostrato in figura 6.5.

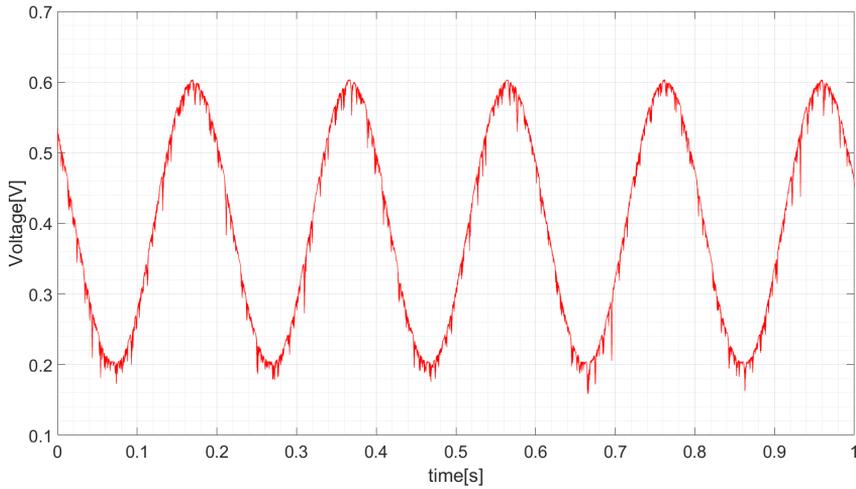


Figura 6.5. Segnale sinusoidale rumoroso filtrato.

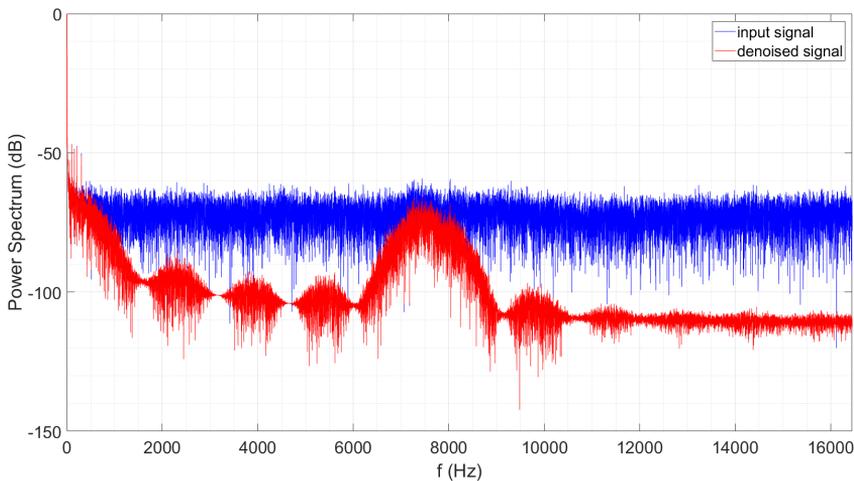


Figura 6.6. Potenza spettrale del segnale in ingresso (blu) e del segnale filtrato (rosso).

Anche in questo caso, si nota visivamente, già nel dominio del tempo, una notevole riduzione delle componenti diverse da quella fondamentale. Ciò viene evidenziato dal confronto ingresso-uscita delle densità spettrale di potenza, in figura 6.6. Il comportamento è del tutto analogo a quello del caso precedente, sia per le frequenze più elevate che per quelle vicine alla fondamentale, in questo caso.

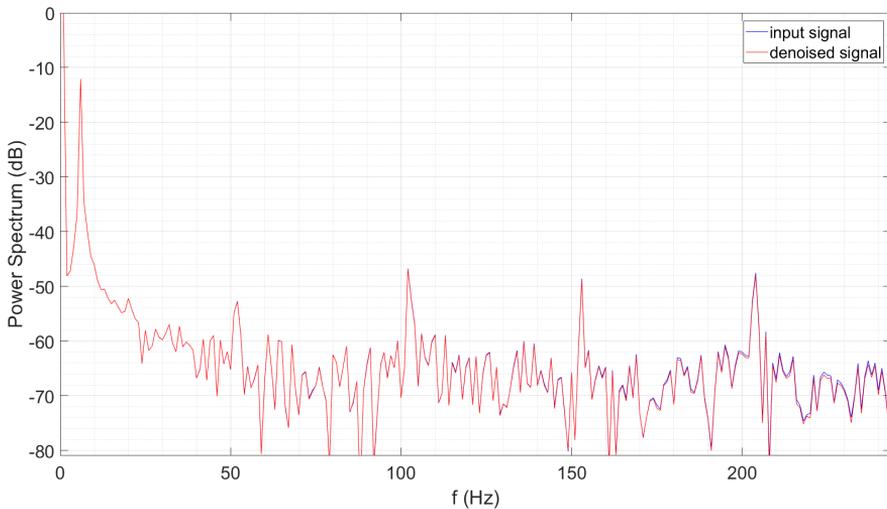


Figura 6.7. Zoom della potenza spettrale del segnale in ingresso (blu) e del segnale filtrato (rosso).

6.1 Lunghezza della finestra variabile

L'analisi è stata ripetuta per osservare l'effetto sul filtraggio provocato dalla diminuzione della lunghezza della finestra L in fase di ricostruzione del segnale monodimensionale (par. 4.3.5).

Ancora una volta si è fatto uso di un segnale rumoroso in ingresso, al fine di testare l'efficacia del filtraggio con la diminuzione di L .

Nei grafici seguenti sono riportati al variare di L i frame nel dominio del tempo, sia in ingresso che in uscita, e le densità spettrali di potenza degli stessi. Da queste ultime si osserva la progressiva riduzione dell'abbattimento delle componenti a frequenze più alte, causata dalla diminuzione della finestra L , come prevedibile.

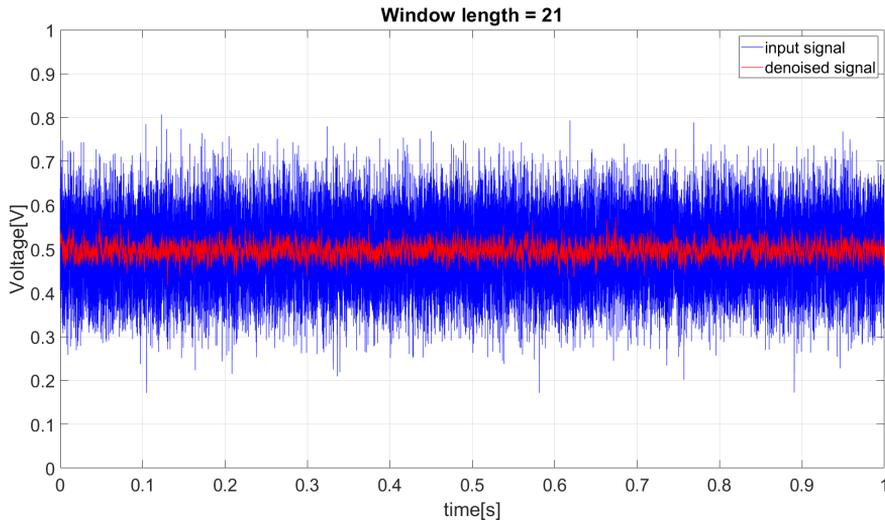


Figura 6.8. Denoising per $L = 21$.

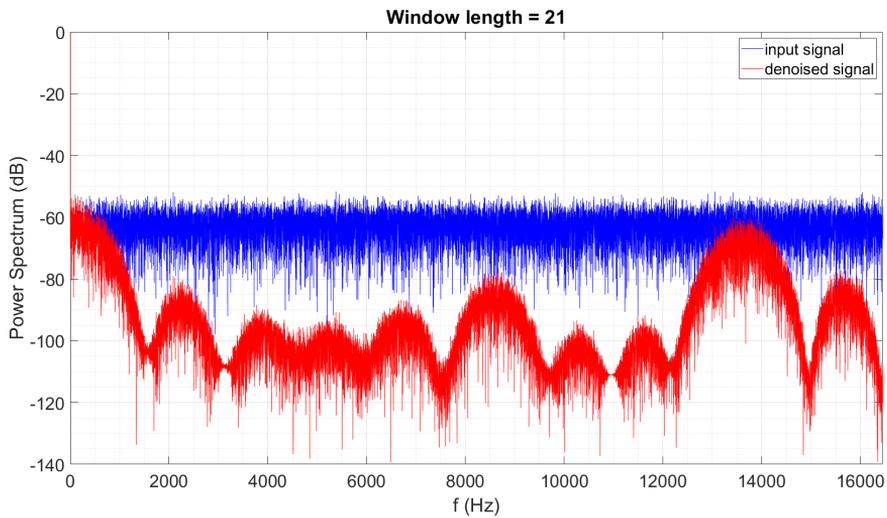


Figura 6.9. Denoising per $L = 21$ nel dominio della frequenza.

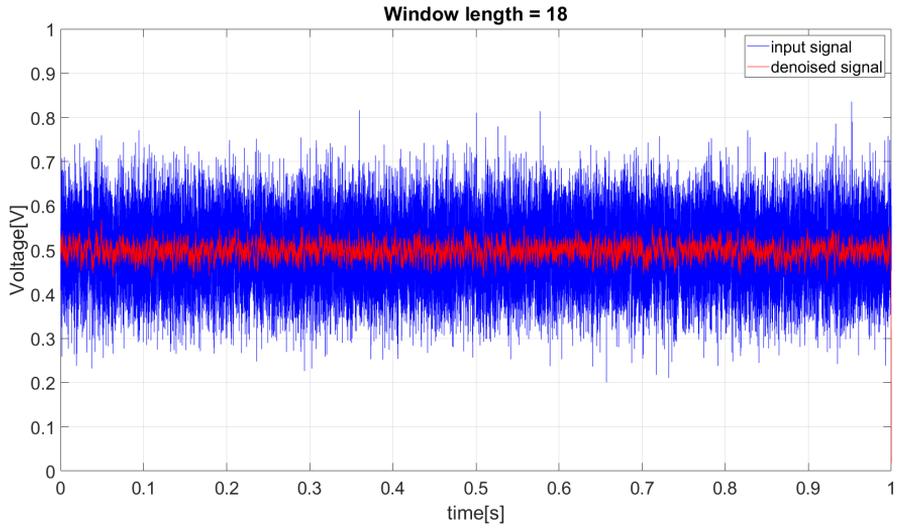


Figura 6.10. Denoising per $L = 18$.

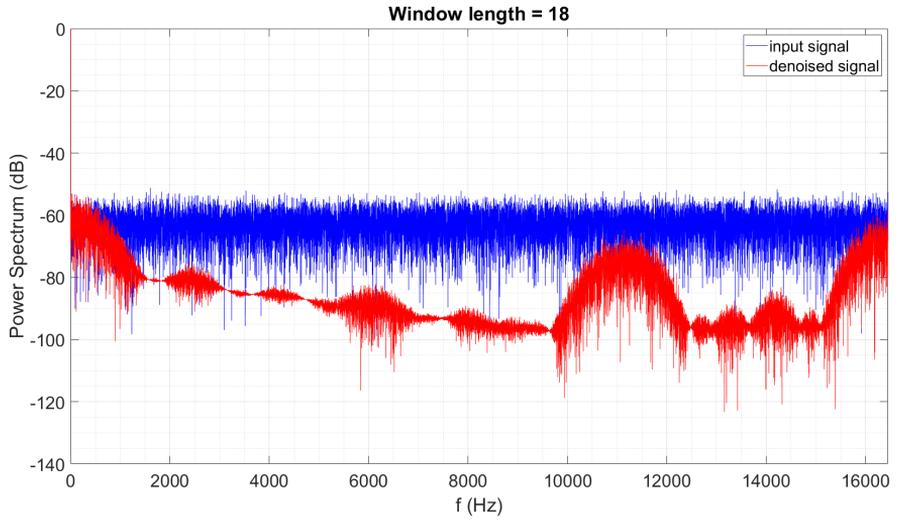


Figura 6.11. Denoising per $L = 18$ nel dominio della frequenza.

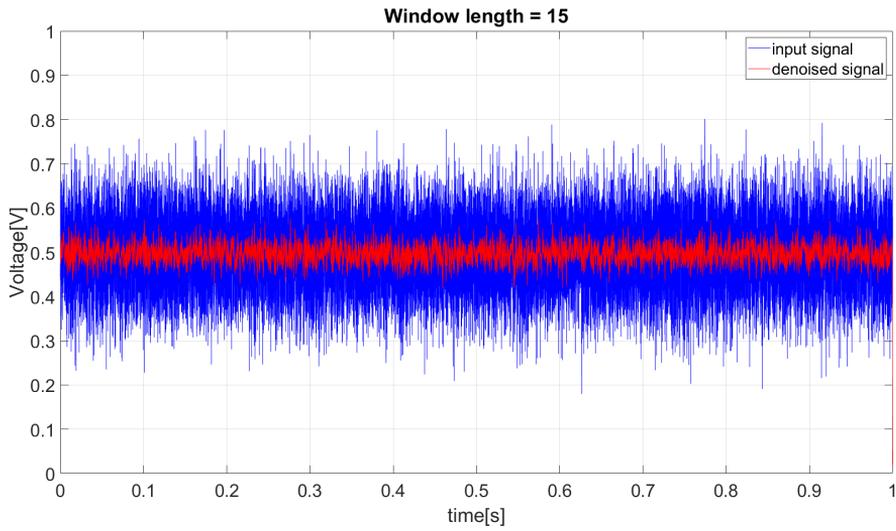


Figura 6.12. Denoising per $L = 15$.

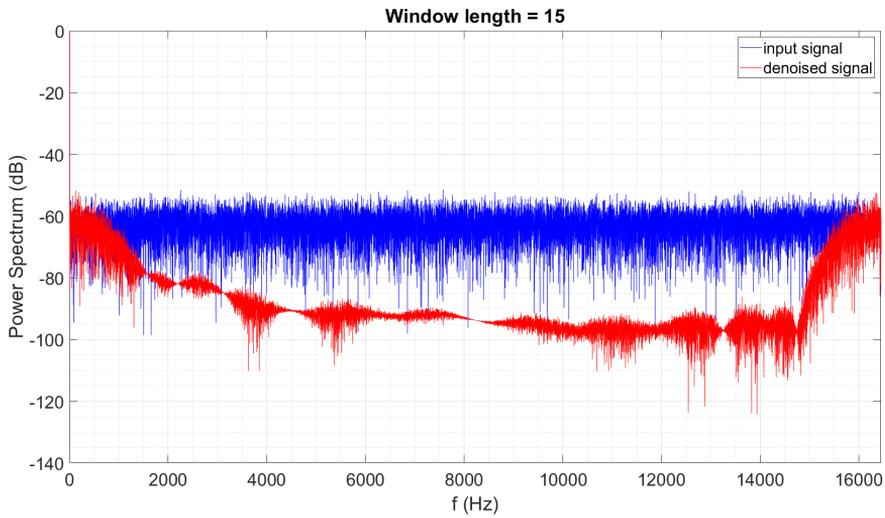


Figura 6.13. Denoising per $L = 15$ nel dominio della frequenza.

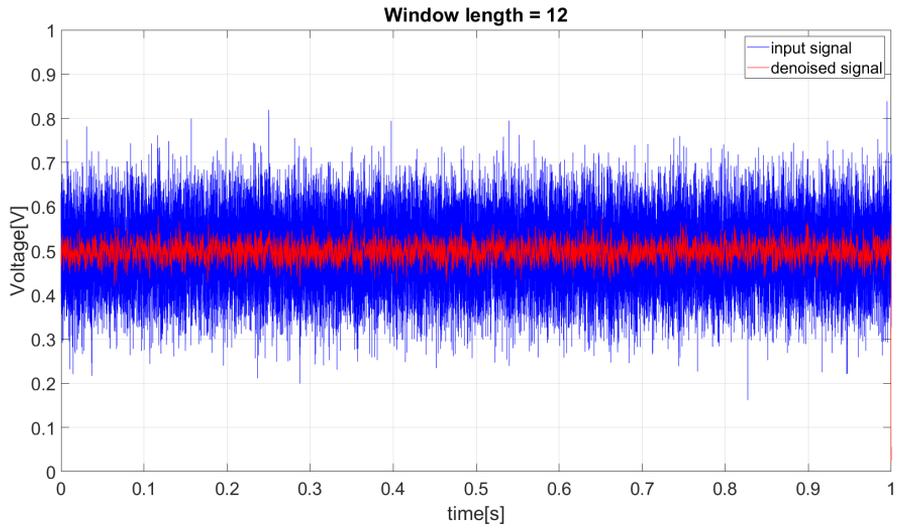


Figura 6.14. Denoising per $L = 12$.

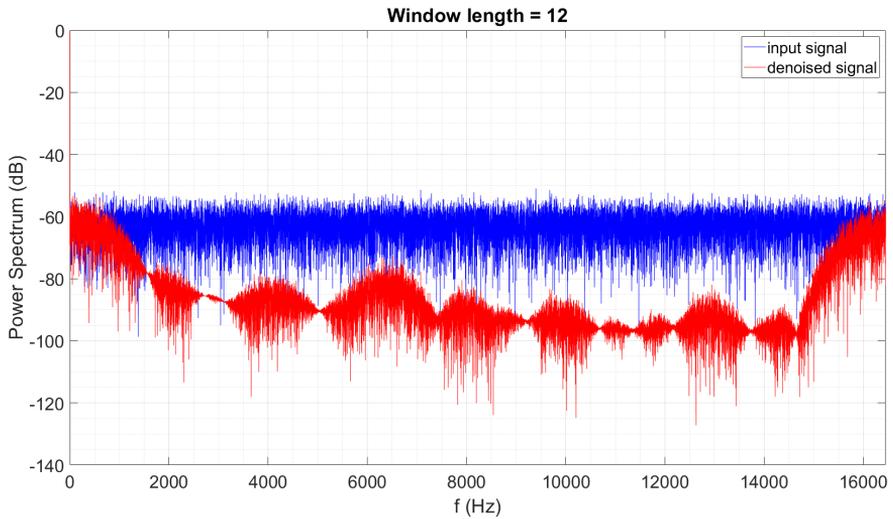


Figura 6.15. Denoising per $L = 12$ nel dominio della frequenza.

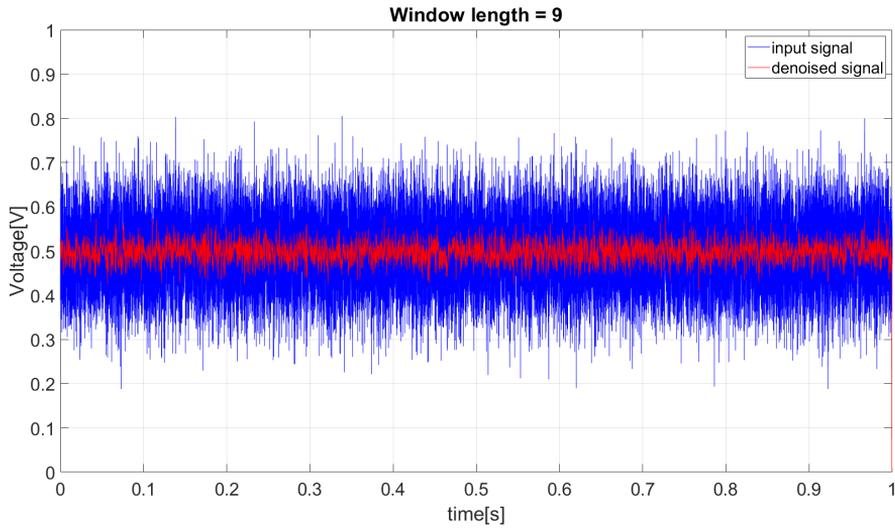


Figura 6.16. Denoising per $L = 9$.

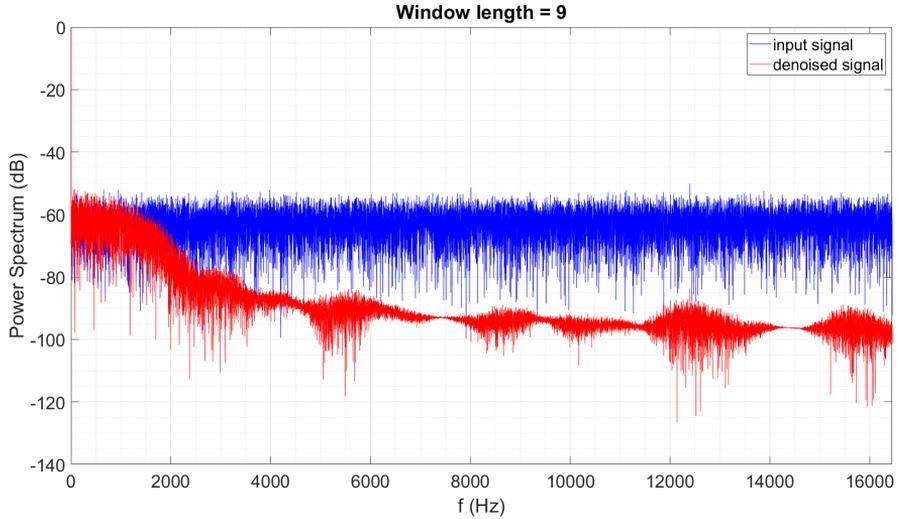


Figura 6.17. Denoising per $L = 9$ nel dominio della frequenza.

Analizzando l'ultima parte dei frame filtrati, nel dominio del tempo, si nota un effetto di bordo che causa l'azzeramento degli ultimissimi campioni (figura 6.18).

Questo fenomeno coinvolge 3 o 4 campioni nel caso con $L = 18$, ed aumenta con il diminuire di L , fino ad una decina di campioni azzerati nel caso peggiore ($L = 9$).

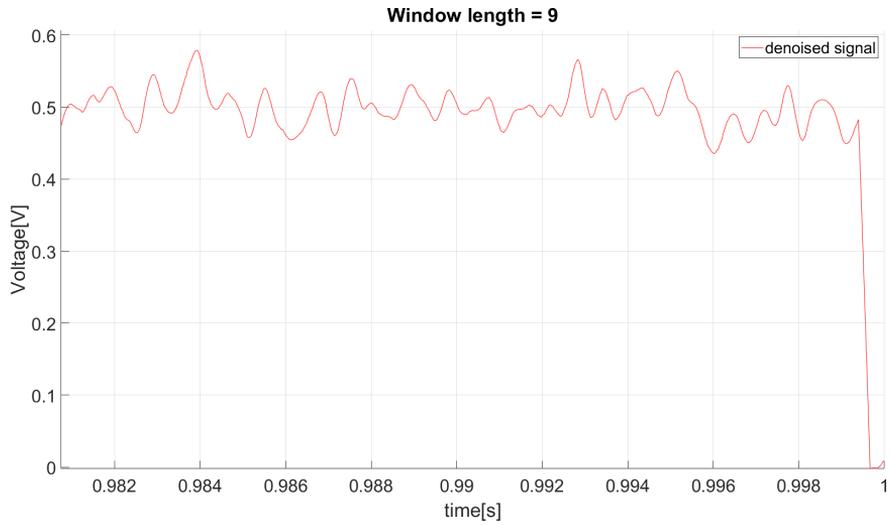


Figura 6.18. Ultimi campioni del segnale filtrato.

Capitolo 7

Analisi dell'errore dell'XADC

In questa sezione vengono analizzati gli errori causati dalla conversione analogico-digitale, da non trascurare vista la risoluzione più bassa rispetto all'hardware National Instruments. Tra essi troviamo:

- errore di linearità;
- errore di guadagno;
- errore di offset;
- crosstalk.

7.1 Errori di linearità, offset e guadagno

Per valutare la linearità dell'XADC, si può utilizzare un segnale noto di tipo sinusoidale che ricopra tutta la dinamica di ingresso dell'XADC, confrontandolo con la sua versione ideale (figura 7.1).

L'errore di linearità, ma anche quelli di guadagno e di offset, possono essere valutati costruendo la caratteristica di trasferimento ingresso-uscita e confrontandola con quella del caso ideale. Per farlo occorre lasciare i campioni acquisiti in valori interi, senza convertirli in Volt. Dopodiché si costruisce la retta che meglio approssima la caratteristica di trasferimento, la quale va poi confrontata con quella ideale, con pendenza unitaria.

Le caratteristiche sono mostrate in figura 7.2 e ad un primo sguardo risultano del tutto sovrapponibili.

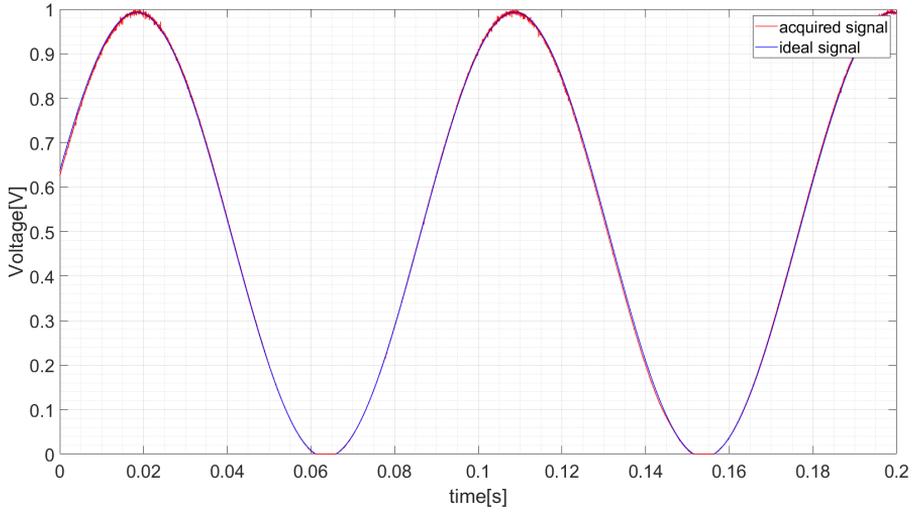


Figura 7.1. Segnale acquisito (rosso) e segnale ideale (blu).

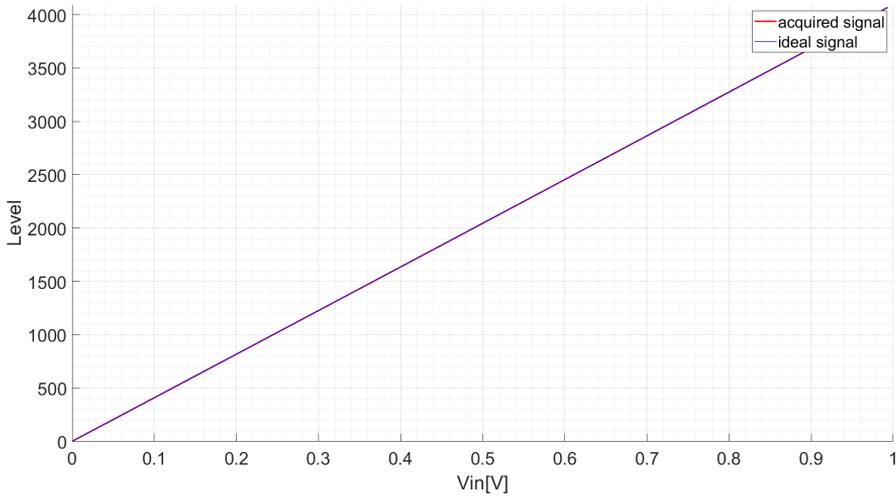


Figura 7.2. Retta che meglio approssima la caratteristica del segnale acquisito (rosso) e caratteristica ideale (blu).

Facendo una valutazione numerica si ha che il guadagno si ottiene come la pendenza della retta che approssima la caratteristica ingresso-uscita del segnale acquisito: $\alpha_{ADC} = 1,0025$. Questo valore è leggermente superiore a quello ideale, ovvero 1.

L'errore di guadagno è quindi pari a $err_{gain} = 1 - \alpha_{ADC} = -0,0025$. In valore percentuale $err_{gain\%} = -0,25\%$.

L'errore di offset viene valutato calcolando i valori medi del segnale reale e di quello ideale: $offset_{ADC} = 0,5297V$; $offset_{ideale} = 0,5299V$. Vi è quindi un errore di offset pari a $200\mu V$.

Sia l'errore di guadagno che di offset sono errori di tipo sistematico che possono essere corretti a livello software. La correzione dell'errore di guadagno può essere effettuata moltiplicando al valore campionato il reciproco dell'errore di guadagno. L'errore di offset viene invece corretto sommando (o sottraendo) l'errore al campione convertito.

La valutazione quantitativa degli errori di linearità avviene attraverso due parametri: DNL ed INL.

Il DNL (Differential Non-Linearity) è definito come lo scostamento massimo dei centri di due intervalli adiacenti rispetto allo stesso scostamento teorico, dove un intervallo corrisponde ad un livello della caratteristica ingresso-uscita dell'ADC (figura 7.3).

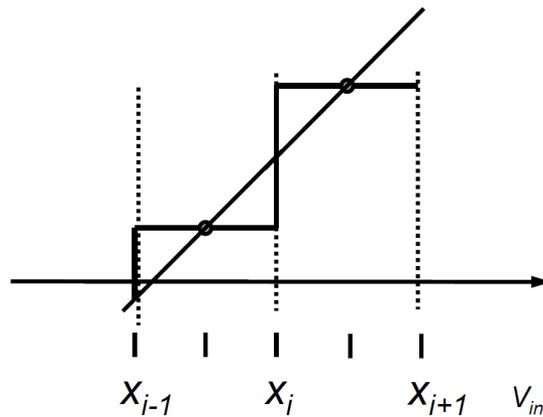


Figura 7.3. Definizione di DNL.

Esso è un indice di misura della non linearità locale e può essere espresso come:

$$Max_i \left(\left(\frac{X_{i-1} + X_i}{2} - iq \right) - \left(\frac{X_{i+1} + X_i}{2} - (i+1)q \right) \right) \quad (7.1)$$

dove iq è il centro teorico dell'intervallo misurato e le X_i le soglie di conversione dell'ADC.

Per avere una misura della non linearità globale si fa uso dell'INL (Integral Non-Linearity), definito come lo scostamento massimo del centro intervallo misurato dal centro intervallo teorico (figura 7.4).

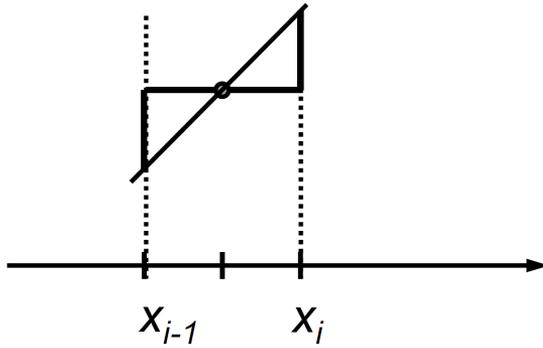


Figura 7.4. Definizione di INL.

Esso può essere espresso come:

$$\text{Max}_i \left(\frac{X_{i-1} + X_i}{2} - iq \right) \quad (7.2)$$

Confrontando le definizioni è chiaro che questi due parametri sono strettamente correlati tra loro.

Partendo dai segnali in figura 7.1 si può ricavare l'andamento del DNL e dell'INL (figura 7.5), questo considerando che nel caso sotto esame gli intervalli risultano pari a $2^{12} = 4096$, dove 12 è il numero di bit di risoluzione dell'XADC.

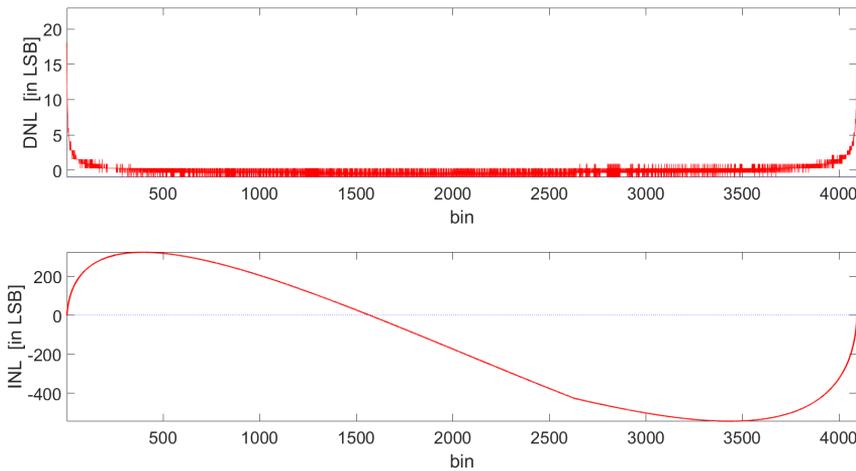


Figura 7.5. Andamento del DNL (in alto) e dell'INL (in basso) espressi in LSB.

Prendendo i valori massimi (in valore assoluto) dagli andamenti in figura 7.5 e moltiplicandoli per il LSB in valore di tensione, pari a $V_{LSB} = \frac{V_{FS}}{2^{12}} = \frac{1}{4096} = 244\mu V$, si ottengono il DNL massimo e l'INL massimo espressi in valori di tensione.

$$V_{DNL_{max}} = 0.0052V$$

$$V_{INL_{max}} = 0.1326V$$

7.2 Crosstalk

Il sistema realizzato acquisisce simultaneamente due segnali su canali adiacenti attraverso un semplice pin-header. Questo può causare interferenze tra i due canali, inoltre vanno valutati eventuali accoppiamenti capacitivi on-chip.

Per valutare la presenza di crosstalk vengono utilizzati due tipologie di segnali, applicati uno alla volta su un singolo canale, mentre l'altro resta flottante. Il primo segnale (figura 7.6) è un segnale ad onda quadra con frequenza a 10 Hz ed ampiezza picco-picco pari ad 1 V, ovvero l'intero range di ingresso dell'XADC.

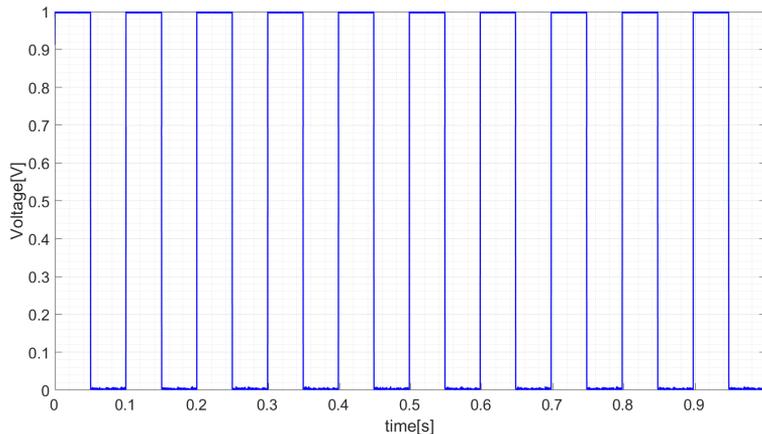


Figura 7.6. Segnale ad onda quadra di prova in ingresso.

Questo segnale risulta particolarmente adatto per la valutazione di fenomeni di crosstalk, avendo esso la massima variazione possibile di tensione all'interno del range in questione.

L'altro tipo di ingresso utilizzato è un segnale fortemente rumoroso (figura 7.7), condizione tipica dei segnali che verranno filtrati dal sistema, seppur estremizzata in maniera tale da garantire una grande variazione di tensione, al fine di valutare eventuali accoppiamenti capacitivi.

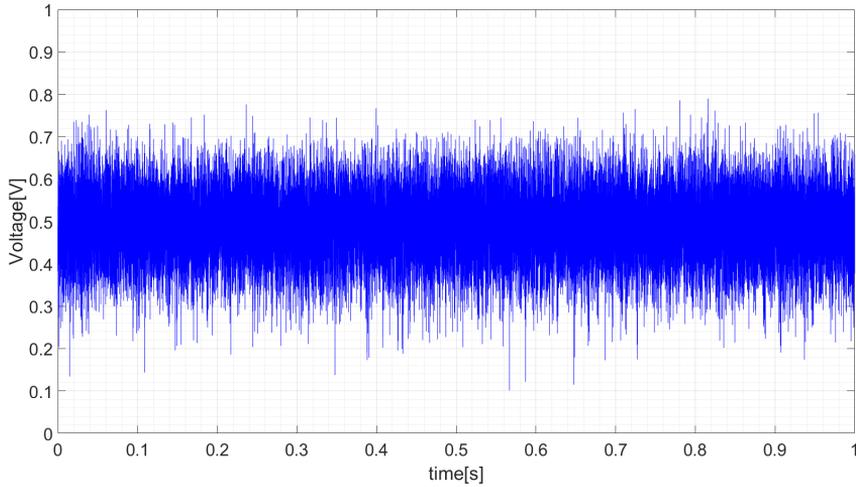


Figura 7.7. Segnale rumoroso di prova in ingresso.

Come già accennato, i due segnali vengono collegati uno alla volta su un canale mentre l'altro, lasciato flottante, viene acquisito ed analizzato e comparato al caso in cui entrambi i canali risultano flottanti. Il risultato è mostrato in figura 7.8.

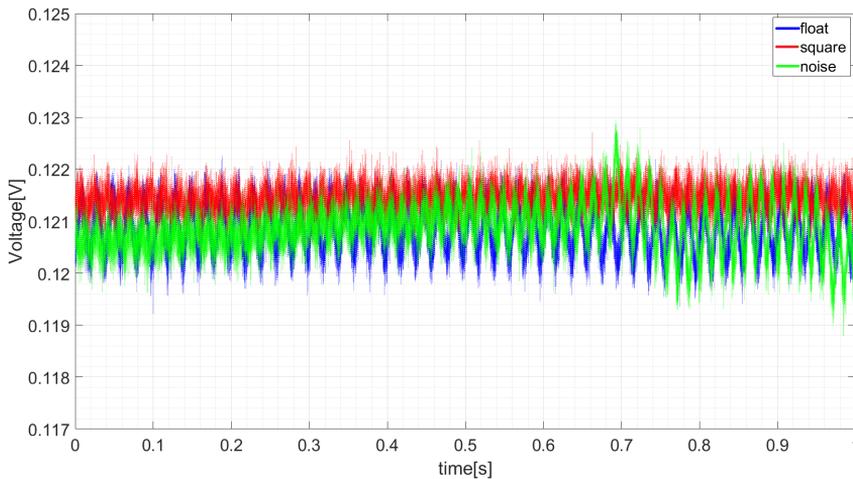


Figura 7.8. Acquisizioni sul canale flottante con l'altro canale flottante (blu), con onda quadra (rosso) e con segnale rumoroso (verde).

Il grafico mostra innanzitutto la presenza di un offset di circa 120 mV di default

per tutti i casi, e un comportamento più o meno sovrapponibile tra loro. Tuttavia calcolando il valore medio per ogni segnale risulta che esiste effettivamente un fenomeno di crosstalk nel caso del segnale ad onda quadra.

$$V_{avg_{float}} = 0,1209V$$

$$V_{avg_{square}} = 0,1214V$$

$$V_{avg_{noise}} = 0,1209V$$

Questo vuol dire che nel caso peggiore il canale flottante ha un offset di circa $500\mu V$ causato dall'altro canale. Considerando che il LSB in valore di tensione è pari a $V_{LSB} = 244\mu V$, l'errore di offset causato dal crosstalk, nel caso peggiore, sarà di circa 2 LSB.

Capitolo 8

Immagine BOOT

Il sistema realizzato può essere avviato in diversi modi a seconda della posizione dei jumpers JTAG sulla Zedboard (figura 8.1). La modalità standard (tutti i jumpers su GND) richiede l'utilizzo di SDK per programmare l'FPGA ed avviare le applicazioni. Questo implica l'utilizzo di un PC con SDK in esecuzione e di un cavo USB collegato tra la Zedboard ed il PC stesso. Ciò non è molto agevole visto l'ambito per cui è destinata la board.



Figura 8.1. Jumpers JTAG di configurazione.

Per rendere il sistema più maneggevole è stata realizzata un'immagine BOOT da scrivere sia sulla memoria Flash QSPI, sia sulla SD card. L'immagine viene generata attraverso il tool di Xilinx SDK noto come *Bootgen*. Essa è in formato binario e contiene all'interno le istruzioni provenienti da 4 file diversi, inseriti nel seguente ordine:

- FSBL (First Start Boot Loader) (.elf);
- bitstream dell'hardware implementato (.bit);

- applicazione core 1 (.elf);
- applicazione core 2 (.elf).

Il FSBL è il file che contiene le prime istruzioni eseguite dal PS della ZYNQ-7000 al fine di inizializzare il sistema. Esso inoltre scrive il bitstream presente nell'immagine sulla PL ed invoca l'applicazione del primo core. Quest'applicazione a sua volta "sveglia" il secondo core avviando la seconda applicazione.

```
ps7_post_config();
Xil_SetTlbAttributes(0xFFFF0000,0x14de2);
Xil_Out32(0xFFFFFFFF,0x2000000);
dmb();
__asm__( "sev" );
```

Codice 8.1. Istruzioni eseguite dal core 1 (main) per avviare l'applicazione 2.

Una volta generata l'immagine essa può essere scritta sia sulla Flash che sulla SD card. Impostando la modalità BOOT mode (MIO3 su VCC per la Flash, MIO3 e MIO2 su VCC per l'SD card) il PS estrae dall'immagine BOOT le informazioni necessarie all'avvio dell'intero sistema. In questo modo, il sistema si avvia autonomamente ad ogni accensione, riavvio o reset, senza l'utilizzo del PC e di nessun cavo, oltre all'alimentazione e alla VGA.

Capitolo 9

Considerazioni finali

In questo lavoro di tesi, in collaborazione con il Centro Ricerche FIAT di Orbassano, Torino, è stato realizzato un sistema di filtraggio di segnali luminosi, e non, con una soluzione hardware non convenzionale.

Il sistema doveva essere adatto all'ambito industriale in misura tale da poter essere montato su un braccio robotico. Allo stesso tempo doveva essere in grado di sostituire l'industrial controller specializzato National Instruments che già implementava l'algoritmo di filtraggio real-time.

La soluzione hardware utilizzata, la Zedboard, basata sul System-on-Chip ZYNQ-7000, ha permesso di implementare l'algoritmo di filtraggio fornito, il quale fa uso della tecnica di denoising nota come Singular Spectrum Analysis (SSA). L'esecuzione real-time è stata resa possibile grazie alla combinazione di CPU, seppur con prestazioni modeste rispetto all'hardware NI, e di logica programmabile presente sulla ZYNQ.

La flessibilità della Zedboard ha permesso inoltre di implementare la visualizzazione VGA dei segnali coinvolti, sul modello oscilloscopio, e di fare uso della BOOT mode per avviare autonomamente il sistema, senza la necessità di un PC.

Possibili sviluppi futuri potrebbero prevedere la possibilità di adattare a tempo di esecuzione, a livello software, la lunghezza della finestra L in base al tipo di segnale in ingresso. Oltre alla semplice programmabilità in fase di ricostruzione, già implementata, si potrebbe sviluppare un metodo per prevedere la lunghezza L ottimale, facendo uno studio preliminare sul segnale acquisito. Ciò potrebbe essere affiancato da un algoritmo che permetta di eliminare gli effetti di bordo, tra un frame e l'altro, osservati nella diminuzione di L .

Un'altra applicazione potrebbe essere anche il filtraggio, non in real-time, di immagini. In questo caso si avrebbero in ingresso due segnali correlati tra loro, uno per ogni dimensione.

Infine la soluzione hardware scelta potrebbe essere a sua volta sostituita da hardware ad hoc, creato appositamente per il contesto automotive. La Zedboard infatti è una board di sviluppo con tante periferiche superflue per questo tipo di applicazioni. Essa potrebbe essere rimpiazzata con una board ancora più compatta con le sole periferiche necessarie al caso in questione e magari con un DAC (Digital-Analog Converter) per riconvertire i segnali in analogico.

Appendice

Appendice A

Codice SDK

A.1 Cpu 1

A.1.1 main.cc

```
#include <iostream>
#include <string>
#include <vector>
#include "xparameters.h" /* SDK generated parameters */
#include "xstatus.h"
#include "xadcps.h"
#include "xscugic.h"
#include "xil_exception.h"
#include "xil_mmu.h"
#include "sleep.h"
//#include "ps7_init.c" //(BOOT MODE)

#define XADC_DEVICE_ID XPAR_PS7_XADC_0_DEVICE_ID
#define data_size 32900 // Fc = 65800 Samples/s

float *baseaddr_p = (float *) 0x11000000; //shared DDR space

using namespace std;

void IntrHandler (void *pair)
{
float data1 = (((float)(XAdcPs_GetAdcData(((std::pair <XAdcPs*, ↵
vector<float >*>*)pair)->first, 16)))) * (1.0f))/65536.0f); //AUX00
//float data1 = XAdcPs_RawToTemperature(((XAdcPs_GetAdcData(((std::↵
pair <XAdcPs*, vector<float >*>*)pair)->first , XADCPS_CH_TEMP)));↵
//temperature test
```

```

float data2 = (((float)(XAdcPs_GetAdcData(((std::pair <XAdcPs*, ↵
    vector<float>*>*)pair)->first, 24))) * (1.0f)) / 65536.0f); //AUX08
// in first entry of data vector store the actual data
*(((std::pair <XAdcPs*, std::vector <float>*>*)pair)->second)[0] ↵↵
    data1;
*(((std::pair <XAdcPs*, std::vector <float>*>*)pair)->second)[1] ↵↵
    data2;
// the second entry is a flag that communicates that the data is new
*(((std::pair <XAdcPs*, std::vector <float>*>*)pair)->second)[2] ↵↵
    1.;
return ;
}

int main(int argc, char* argv[])
{
sleep(1);

vector<float> data_storage1(data_size, 0.);
vector<float> data_storage2(data_size, 0.);
unsigned current_index = 0;

//XADC Initialization
XAdcPs XADC_Driver_Instance;
XAdcPs_Config* cfg = XAdcPs_LookupConfig(XADC_DEVICE_ID);
if(cfg == NULL)
{
    cout << " Config Lookup Failed " << endl;
    return 0;
}
XAdcPs_CfgInitialize(&XADC_Driver_Instance, cfg, cfg->BaseAddress);

//Self-Test
int Status = XAdcPs_SelfTest(&XADC_Driver_Instance);
if(Status != XST_SUCCESS)
{
    cout << " Self Test Failed !" << endl;
    return 0;
}

//Sequencer settings (only if different from Vivado)
//XAdcPs_SetSequencerMode (&XADC_Driver_Instance, ↵
    XADCPS_SEQ_MODE_SAFE);
//XAdcPs_SetSeqChEnables (&XADC_Driver_Instance, XADCPS_SEQ_CH_TEMP)↵
    ; //Channels enabling
//XAdcPs_SetSeqInputMode (&XADC_Driver_Instance, 0); //All channels ↵
    set to unipolar mode
//XAdcPs_SetAvg (&XADC_Driver_Instance , XADCPS_AVG_16_SAMPLES); //↵
    Perform Average
//XAdcPs_SetSequencerMode (&XADC_Driver_Instance, ↵
    XADCPS_SEQ_MODE_SIMUL_SAMPLING); //Sequencer mode

```

```

/*****ACQUISITION WITH INTERRUPT*****/
vector<float> data (3,0.);
pair<XAdcPs*, vector<float>*> mypair (&XADC_Driver_Instance , &data);

//Interrupt configuration
XScuGic InterruptController ; // instance of the GIC driver
XScuGic_Config* GicConfig ; // pointer to the GIC config
int Status_Gic ; // setup status

// setup GIC
Xil_ExceptionInit();
GicConfig = XScuGic_LookupConfig(XPAR_PS7_SCUGIC_0_DEVICE_ID); //←
    fetch config
if(GicConfig == nullptr) // check for failure
{
    cout << " Config Loading Failed !" << endl;
    return 0;
}
Status_Gic = XScuGic_CfgInitialize(&InterruptController , GicConfig , ←
    GicConfig -> CpuBaseAddress); // initialize config
if(Status_Gic != XST_SUCCESS) // check for failure
{
    cout << " Config Initialization Failed !" << endl;
    return 0;
}

// connect GIC interrupt handler which processes all interrupts and ←
    then calls the correct handlers
Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_IRQ_INT , (←
    Xil_ExceptionHandler)IntrHandler/*XScuGic_InterruptHandler*/, (←
    void*)&mypair/*&InterruptController*/);

// Connect our custom handler . This is done with the ←
    XScuGic_Connectfunction . To disconnect handlers , use ←
    XScuGic_Disconnect .
// The first arguments are : Pointer to GIC driver , interrupt port ←
    we target , interrupt handler ( cast to Xil_ExceptionHandler ) , ←
    argument of interrupt handler
Status_Gic = XScuGic_Connect(&InterruptController , 31 , (←
    Xil_ExceptionHandler)IntrHandler , (void*) &mypair);
if( Status_Gic != XST_SUCCESS )
{
    cout << " Handler Connection Failed !" << endl;
    return 0;
}

// Enable interrupts on port 61
XScuGic_Enable(&InterruptController , 31);
// Enable interrupts in ARM
Xil_ExceptionEnable();

```

```
// Set the priority and trigger type for interrupts on port 61. I'm ←
    not sure how the last two arguments work , but they set the ←
    trigger type (rising ,level , falling ,..) and priority
XScuGic_SetPriorityTriggerType (&InterruptController , 31, 0xa0 , 3);

unsigned int k = 1;
*(baseaddr_p+0) = 0;

//Wake up CPU1 (ONLY BOOT MODE)
//ps7_post_config();
//Xil_SetTlbAttributes(0xFFFF0000,0x14de2);
//Xil_Out32(0xFFFFFFFF,0x2000000);
//dmb();
//__asm__("sev");

while (true)
{
// print data if the buffer is full
if(current_index == data_size*2)
{
    //ping-pong
    if(*(baseaddr_p+0) == 0) *(baseaddr_p+0) = 1; //first filling
    else if(*(baseaddr_p+0) == 2) *(baseaddr_p+0) = 1; //first ←
        buffer filled
    else if(*(baseaddr_p+0) == 1) //second buffer filled
    {
        *(baseaddr_p+0) = 2;
        k = 1;
    }
    // start overwriting the buffer from the beginning
    current_index = 0;
    // set flag to "old data "
    (*(mypair.second))[2] = 0.;
}

// fetch the data if the interrupt handler has updated it
if ((*(mypair.second))[2] > 0.)
{
    // set the flag to "old data "
    (*(mypair.second))[2] = 0.;

    //resample (stores only even samples)
    if((current_index % 2) == 0)
    {
        *(baseaddr_p+k) = (*(mypair.second))[0]; //CH1
        *(baseaddr_p+k+1) = (*(mypair.second))[1]; //CH2
        k = k+2;
    }
    // next element
    ++current_index;
}
}
```

```
return 0;
}
```

A.2 Cpu 2

A.2.1 main.cc

```
#include "myfilterHW.h"
#include "xparameters.h" /* SDK generated parameters */
#include "xsdps.h" /* SD device driver */
#include "xil_printf.h"
#include "ff.h"
#include "xil_cache.h"
#include "xplatform_info.h"
#include "xtime_l.h"
#include "xil_mmu.h"
#include "vga.h"

#define data_size_tot 32900 // Fc = 65800 Samples/s
float *baseaddr_p = (float *) 0x11000000; //shared DDR space

/***** SdCard Variable Definitions ←
      *****/
static FIL fil;
static FATFS fatfs;
static char FileName_dest[32] = "in_test.txt";
static char FileName_dest2[32] = "out_test.txt";
static char *SD_File;
/*←
      *****/
using namespace std;

static const int kParam_1 = 21;

int main(int argc, char* argv[])
{
    vector<float> in_data1;
    vector<float> in_data2;
    unsigned int k = 1;
    int sd_cnt = 0;
    vector<float> svd_y_in;
    vector<float> svd_y_out;

    FRESULT Res;
    TCHAR const *Path = "0:/";
```

```

SD_File = (char *)FileName_dest;
string out_data;

XTime tStart, tEnd, tElapsed;

SetupVGA();

while (true)
{
    while(*(baseaddr_p+0) != 1){}; //wait for buffer1
    while(*(baseaddr_p+0) == 0){}; //wait for first filling
    while(true)
    {
        in_data1.push_back(*(baseaddr_p+k));
        in_data2.push_back(*(baseaddr_p+k+1));
        k = k+2;

        if(k >= data_size_tot*2)
        {
            cout << "Acquired buf1: " << (in_data1.size()+in_data2.size())
            << "/ch" << endl;

            /*****SSA EXECUTION*****/
            vector<float> svd_y1(in_data1.size());
            vector<float> svd_y2(in_data2.size());

            XTime_GetTime(&tStart); //start timer
            svdHW(in_data1.data(), in_data2.data(), svd_y1.data(), svd_y2.data(),
            in_data1.size());
            XTime_GetTime(&tEnd); //end timer

            tElapsed = (tEnd-tStart)/(COUNTS_PER_SECOND/1000);
            cout << "SSA executed in " << tElapsed << " ms" << endl;
            ;

            if(MAX1 <= 1000 && MAX2 <= 1000 && MIN1 >= 0 && MIN2 >= 0
            && MAX1 > MIN1 && MAX2 > MIN2 && (MAX1 % 50) == 0
            && (MIN1 % 50) == 0 && (MAX2 % 50) == 0 && (MIN2 % 50) == 0)
            {
                cout << "Start VGA" << endl << endl;
                blank_screen();
                plot_input(in_data1.data(), in_data1.size(), in_data2.data(),
                in_data2.size());
                plot_output(svd_y1.data(), svd_y1.size(), svd_y2.data(),
                svd_y2.size());

                Xil_DCacheDisable();
                Xil_DCacheEnable();
            }
        }
    }
}

```

```

else cout << "Edge value not valid: VGA turned OFF" << ←
      endl << endl;

//SD card input test
//svd_y_in.insert(svd_y_in.end(), in_data1.begin(), ←
in_data1.end());

//clear buffer
in_data1.clear();
in_data2.clear();
k = 1+(data_size_tot*2);

//SD card output test
//svd_y_out.insert(svd_y_out.end(), svd_y1.begin(), ←
svd_y1.end());
//sd_cnt++;

svd_y1.clear();
svd_y2.clear();
break;
}
}
while(*(baseaddr_p+0) != 2){}; //wait for buffer2
while(true)
{
in_data1.push_back(*(baseaddr_p+k));
in_data2.push_back(*(baseaddr_p+k+1));
k = k+2;
if(k >= data_size_tot*2*2)
{
cout << "Acquired buf2: " << (in_data1.size()+in_data2.←
size())/2 << "/ch" << endl;

/*****SSA EXECUTION←
*****/
vector<float> svd_y1(in_data1.size());
vector<float> svd_y2(in_data2.size());

XTime_GetTime(&tStart); //start timer
svdHW(in_data1.data(), in_data2.data(), svd_y1.data(), ←
svd_y2.data(), in_data1.size());
XTime_GetTime(&tEnd); //end timer

tElapsed = (tEnd-tStart)/(COUNTS_PER_SECOND/1000);
cout << "SSA executed in " << tElapsed << " ms" << endl←
;

if(MAX1 <= 1000 && MAX2 <= 1000 && MIN1 >= 0 && MIN2 >= ←
0 && MAX1 > MIN1 && MAX2 > MIN2 && (MAX1 % 50) == 0 ←
&&
(MIN1 % 50) == 0 && (MAX2 % 50) == 0 && (MIN2 % ←
50) == 0)

```

```

    {
        cout << "Start VGA" << endl << endl;
        blank_screen();
        plot_input(in_data1.data(), in_data1.size(), in_data2←
            .data(), in_data2.size());
        plot_output(svd_y1.data(), svd_y1.size(), svd_y2.data←
            (), svd_y2.size());

        Xil_DCacheDisable();
        Xil_DCacheEnable();
    }
    else cout << "Edge value not valid: VGA turned OFF" << ←
        endl << endl;

    //SD card input test
    //svd_y_in.insert(svd_y_in.end(), in_data1.begin(), ←
        in_data1.end());

    //clear buffer
    in_data1.clear();
    in_data2.clear();
    k = 1;

    //SD card output test
    //svd_y_out.insert(svd_y_out.end(), svd_y1.begin(), ←
        svd_y1.end());
    //sd_cnt++;

    svd_y1.clear();
    svd_y2.clear();
    break;
}
}
//if(sd_cnt >= 4) break; //SD card test after 4 seconds
}

/*****SD CARD FILE TRANSMISSION←
*****/
//SD Card Mounting
Res = f_mount(&fatfs, Path, 0);
if (Res != FR_OK) {
    cout << "MOUNTING FAILED" << endl;
    return XST_FAILURE;
}
/*****INPUT*****/
//File opening
Res = f_open(&fil, SD_File, FA_CREATE_ALWAYS | FA_WRITE);
if (Res) {
    cout << "OPENING FAILED" << endl;
    return XST_FAILURE;
}
}

```

```
//SD Card Writing
for(unsigned long int i = 0; i < svd_y_in.size(); i++)
{
    out_data = to_string(svd_y_in[i]);
    out_data.append("\n");
    //cout << out_data;
    f_printf (&fil, (TCHAR*)out_data.c_str());
}

//File Closing
Res = f_close(&fil);
if (Res) {return XST_FAILURE;}

/*****OUTPUT*****/
SD_File = (char *)FileName_dest2;

//File opening
Res = f_open(&fil, SD_File, FA_CREATE_ALWAYS | FA_WRITE);
if (Res) {
    cout << "OPENING FAILED" << endl;
    return XST_FAILURE;
}

//SD Card Writing
for(unsigned long int i = 0; i < svd_y_out.size(); i++)
{
    out_data = to_string(svd_y_out[i]);
    out_data.append("\n");
    f_printf (&fil, (TCHAR*)out_data.c_str());
}

//File Closing
Res = f_close(&fil);
if (Res) {return XST_FAILURE;}

cout << "Writing complete" << endl;

return 0;
}
```

A.2.2 myfilterHW.h

```
#ifndef SRC_MYFILTERHW_H_
#define SRC_MYFILTERHW_H_

#include <Eigen>
#include <Dense>
#include <iostream>
#include <string>
```

```
#include <fstream>
#include <vector>

#include "dma_interrupt.h"
#include "xaxidma.h"
#include "xparameters.h"
#include "xil_exception.h"
#include "xdebug.h"
#include "xscugic.h"

/***** Variable Definitions ←
*****/
#define data_size 1370
#define param_l 21
#define param_l_mod 21

typedef Eigen::Matrix<float, Eigen::Dynamic, Eigen::Dynamic> ←
    myMatrix;
typedef Eigen::Matrix<float, Eigen::Dynamic, 1> myVector;

using namespace std;
/*←
*****←
*/

int svdHW (const float *in_data1, const float *in_data2, float *←
    out_data1, float *out_data2, unsigned int size);

#endif /* SRC_MYFILTERHW_H_ */
```

A.2.3 myfilterHW.cc

```
#include "myfilterHW.h"

/*
 * Device instance definitions
 */
static XAxiDma AxiDma; /* Instance of the XAxiDma */
static INTC Intc; /* Instance of the Interrupt Controller */

/*
 * Flags interrupt handlers use to notify the application context ←
 * the events.
 */
volatile int TxDone;
volatile int RxDone;
volatile int Error;
```

```

int svdHW (const float *in_data1, const float *in_data2, float *←
out_data, float *out_data2, unsigned int size)
{
    int Status;
    XAxidma_Config *Config;

    Config = XAxidma_LookupConfig(DMA_DEV_ID);
    if (!Config) {
        xil_printf("No config found for %d\r\n", DMA_DEV_ID);

        return XST_FAILURE;
    }

    unsigned int tot_size = size;
    vector <float> in((data_size+param_1)*2);
    vector <float> out(param_1*param_1*2);
    float X[param_1][size-param_1+1];
    float X2[param_1][size-param_1+1];
    myMatrix X_mul, X_mul2;

    X_mul.setZero(param_1, param_1);
    X_mul2.setZero(param_1, param_1);

    //X matrix filling
    for (unsigned int i = 0; i < (tot_size - param_1 + 1); i++)
    {
        X[0][i] = in_data1[0 + i];
        X[1][i] = in_data1[1 + i];
        X[2][i] = in_data1[2 + i];
        X[3][i] = in_data1[3 + i];
        X[4][i] = in_data1[4 + i];
        X[5][i] = in_data1[5 + i];
        X[6][i] = in_data1[6 + i];
        X[7][i] = in_data1[7 + i];
        X[8][i] = in_data1[8 + i];
        X[9][i] = in_data1[9 + i];
        X[10][i] = in_data1[10 + i];
        X[11][i] = in_data1[11 + i];
        X[12][i] = in_data1[12 + i];
        X[13][i] = in_data1[13 + i];
        X[14][i] = in_data1[14 + i];
        X[15][i] = in_data1[15 + i];
        X[16][i] = in_data1[16 + i];
        X[17][i] = in_data1[17 + i];
        X[18][i] = in_data1[18 + i];
        X[19][i] = in_data1[19 + i];
        X[20][i] = in_data1[20 + i];
        X2[0][i] = in_data2[0 + i];
        X2[1][i] = in_data2[1 + i];
    }
}

```

```

X2[2][i] = in_data2[2 + i];
X2[3][i] = in_data2[3 + i];
X2[4][i] = in_data2[4 + i];
X2[5][i] = in_data2[5 + i];
X2[6][i] = in_data2[6 + i];
X2[7][i] = in_data2[7 + i];
X2[8][i] = in_data2[8 + i];
X2[9][i] = in_data2[9 + i];
X2[10][i] = in_data2[10 + i];
X2[11][i] = in_data2[11 + i];
X2[12][i] = in_data2[12 + i];
X2[13][i] = in_data2[13 + i];
X2[14][i] = in_data2[14 + i];
X2[15][i] = in_data2[15 + i];
X2[16][i] = in_data2[16 + i];
X2[17][i] = in_data2[17 + i];
X2[18][i] = in_data2[18 + i];
X2[19][i] = in_data2[19 + i];
X2[20][i] = in_data2[20 + i];
}

unsigned int indi=0;
for(unsigned int i = 0; i < (tot_size-param_1+1)/data_size; i++) ←
    //invoke HW (24 times)
{
    //input stream assembling
    for (unsigned int i = 0; i < (data_size+param_1)*2; i=i+2)
    {
        in[i] = in_data1[indi];
        in[i+1] = in_data2[indi];
        indi++;
    }
    indi=indi-param_1;

    /* Initialize DMA engine */
    Status = XAxiDma_CfgInitialize(&AxiDma, Config);
    if (Status != XST_SUCCESS)
    {
        xil_printf("Initialization failed %d\r\n", Status);
        return XST_FAILURE;
    }
    if(XAxiDma_HasSg(&AxiDma))
    {
        xil_printf("Device configured as SG mode \r\n");
        return XST_FAILURE;
    }
    /* Set up Interrupt system */
    Status = SetupIntrSystem(&Intc, &AxiDma, TX_INTR_ID, ←
        RX_INTR_ID);
    if (Status != XST_SUCCESS)
    {
        xil_printf("Failed intr setup\r\n");
    }
}

```

```

    return XST_FAILURE;
}
/* Disable all interrupts before setup */
XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK, ←
    XAXIDMA_DMA_TO_DEVICE);
XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK, ←
    XAXIDMA_DEVICE_TO_DMA);
/* Enable all interrupts */
XAxiDma_IntrEnable(&AxiDma, XAXIDMA_IRQ_ALL_MASK, ←
    XAXIDMA_DMA_TO_DEVICE);
XAxiDma_IntrEnable(&AxiDma, XAXIDMA_IRQ_ALL_MASK, ←
    XAXIDMA_DEVICE_TO_DMA);

/* Initialize flags before start transfer test */
TxDone = 0;
RxDone = 0;
Error = 0;

/* Flush the SrcBuffer before the DMA transfer, in case the ←
Data Cache is enabled */
Xil_DCacheFlushRange((u32)in.data(), (data_size+param_1)*2*←
    sizeof(float));

//input stream transfer
Status = XAxiDma_SimpleTransfer(&AxiDma, (u32)in.data(), 2*(←
    data_size+param_1)*sizeof(float), XAXIDMA_DMA_TO_DEVICE);
if (Status != XST_SUCCESS) {return XST_FAILURE;}

//output stream transfer
Status = XAxiDma_SimpleTransfer(&AxiDma, (u32)out.data(), 2*(←
    param_1*param_1)*sizeof(float), XAXIDMA_DEVICE_TO_DMA);
if (Status != XST_SUCCESS) {return XST_FAILURE;}

if (Error) {xil_printf("Failed test transmit %s done, receive ←
    %s done\r\n", TxDone? "" : "not", RxDone? "" : "not");}

/* Wait TX done and RX done */
while (!TxDone){}

/* Invalidate the DestBuffer before checking the data, in case←
the Data Cache is enabled */
Xil_DCacheInvalidateRange((u32)out.data(), 2*(param_1*param_1)←
    *sizeof(float));

while (!RxDone){}

/* Disable TX and RX Ring interrupts and return success */
DisableIntrSystem(&Intc, TX_INTR_ID, RX_INTR_ID);

unsigned int ind_out = 0;
//output stream disassembling
for (unsigned int i = 0; i < param_1; i++)

```

```

    {
        for (unsigned int j = 0; j < param_1; j++)
        {
            X_mul(i, j) += out[ind_out];
            X_mul2(i, j) += out[ind_out+1];
            ind_out = ind_out+2;
        }
    }
}

//SVD (compute U matrix)
Eigen::JacobiSVD<myMatrix> svd(X_mul, Eigen::ComputeThinU);
Eigen::JacobiSVD<myMatrix> svd2(X_mul2, Eigen::ComputeThinU);

// Normalization of eigenvalues
float svd_average = 0, svd_average2 = 0;
for (unsigned int i = 0; i < (unsigned) svd.singularValues().size()
    (); i++)
{
    svd_average += svd.singularValues()[i];
    svd_average2 += svd2.singularValues()[i];
}
svd_average = svd_average / (svd.singularValues()[0] * param_1);
svd_average2 = svd_average2 / (svd2.singularValues()[0] * param_1);

myVector svd_AB(param_1), svd_AB2(param_1);
for (unsigned int i = 0; i < (unsigned) svd.singularValues().size()
    (); i++)
{
    svd_AB[i] = svd.singularValues()[i] / svd.singularValues()[0] -
        svd_average;
    svd_AB2[i] = svd2.singularValues()[i] / svd2.singularValues()
        [0] - svd_average2;
}

myVector svd_rms(param_1), svd_rms2(param_1);
for (unsigned int i = 0; i < (unsigned) svd_AB.size() - 1; i++)
{
    svd_rms[i] = sqrt((svd_AB[i] * svd_AB[i] + svd_AB[i + 1] *
        svd_AB[i + 1]) / 2);
    svd_rms2[i] = sqrt((svd_AB2[i] * svd_AB2[i] + svd_AB2[i + 1] *
        svd_AB2[i + 1]) / 2);
}
svd_rms[param_1 - 1] = svd_rms[param_1 - 2];
svd_rms2[param_1 - 1] = svd_rms2[param_1 - 2];

//Grouping
int svd_min_pos, svd_min_pos2;
svd_rms.minCoeff(&svd_min_pos);
svd_rms2.minCoeff(&svd_min_pos2);

```

```

myVector svd_I, svd_I2;
if (svd_min_pos > 0)
{
    svd_I.resize(svd_min_pos + 1);
    for (int i = 0; i < svd_min_pos + 1; i++)
        svd_I(i) = i;
}
else if (svd_min_pos == 0)
{
    svd_I << 1;
}
unsigned int svd_i_size = svd_I.size();

if (svd_min_pos2 > 0)
{
    svd_I2.resize(svd_min_pos2 + 1);
    for (int i = 0; i < svd_min_pos2 + 1; i++)
        svd_I2(i) = i;
}
else if (svd_min_pos2 == 0)
{
    svd_I2 << 1;
}
unsigned int svd_i_size2 = svd_I2.size();

float U[param_1][param_1], U2[param_1][param_1];
for(unsigned int i = 0; i < param_1; ++i)
{
    for(unsigned int j = 0; j < param_1; ++j)
    {
        U[i][j] = svd.matrixU()(i, j);
        U2[i][j] = svd2.matrixU()(i, j);
    }
}

float tmp2[tot_size-param_1+1][param_1];
float tmp2_2[tot_size-param_1+1][param_1];
float sum2 = 0, sum2_2 = 0;
for(unsigned int i = 0; i < tot_size - param_1 + 1; ++i)
{
    for(unsigned int j = 0; j < svd_i_size; ++j)
    {
        sum2 = 0;
        sum2 += X[0][i] * U[0][j];
        sum2 += X[1][i] * U[1][j];
        sum2 += X[2][i] * U[2][j];
        sum2 += X[3][i] * U[3][j];
        sum2 += X[4][i] * U[4][j];
        sum2 += X[5][i] * U[5][j];
        sum2 += X[6][i] * U[6][j];
        sum2 += X[7][i] * U[7][j];
        sum2 += X[8][i] * U[8][j];
    }
}

```

```

sum2 += X[9][i] * U[9][j];
sum2 += X[10][i] * U[10][j];
sum2 += X[11][i] * U[11][j];
sum2 += X[12][i] * U[12][j];
sum2 += X[13][i] * U[13][j];
sum2 += X[14][i] * U[14][j];
sum2 += X[15][i] * U[15][j];
sum2 += X[16][i] * U[16][j];
sum2 += X[17][i] * U[17][j];
sum2 += X[18][i] * U[18][j];
sum2 += X[19][i] * U[19][j];
sum2 += X[20][i] * U[20][j];
tmp2[i][j] = sum2;
}
for(unsigned int j = 0; j < svd_i_size2; ++j)
{
    sum2_2 = 0;
    sum2_2 += X2[0][i] * U2[0][j];
    sum2_2 += X2[1][i] * U2[1][j];
    sum2_2 += X2[2][i] * U2[2][j];
    sum2_2 += X2[3][i] * U2[3][j];
    sum2_2 += X2[4][i] * U2[4][j];
    sum2_2 += X2[5][i] * U2[5][j];
    sum2_2 += X2[6][i] * U2[6][j];
    sum2_2 += X2[7][i] * U2[7][j];
    sum2_2 += X2[8][i] * U2[8][j];
    sum2_2 += X2[9][i] * U2[9][j];
    sum2_2 += X2[10][i] * U2[10][j];
    sum2_2 += X2[11][i] * U2[11][j];
    sum2_2 += X2[12][i] * U2[12][j];
    sum2_2 += X2[13][i] * U2[13][j];
    sum2_2 += X2[14][i] * U2[14][j];
    sum2_2 += X2[15][i] * U2[15][j];
    sum2_2 += X2[16][i] * U2[16][j];
    sum2_2 += X2[17][i] * U2[17][j];
    sum2_2 += X2[18][i] * U2[18][j];
    sum2_2 += X2[19][i] * U2[19][j];
    sum2_2 += X2[20][i] * U2[20][j];
    tmp2_2[i][j] = sum2_2;
}
}
//param_l_mod instead of param_l

unsigned int k = tot_size - param_l_mod + 1;

float svd_rca [param_l_mod][tot_size-param_l_mod+1];
float svd_rca2 [param_l_mod][tot_size-param_l_mod+1];
float sum3 = 0, sum3_2 = 0;
for(unsigned int i = 0; i < param_l_mod; ++i)
{
    for(unsigned int j = 0; j < tot_size-param_l_mod+1; ++j)

```

```

{
    sum3 = 0;
    for(unsigned int k = 0; k < svd_i_size; ++k)
    {
        sum3 += U[i][k] * tmp2[j][k];
    }
    svd_rca[i][j] = sum3;
    sum3_2 = 0;
    for(unsigned int k = 0; k < svd_i_size2; ++k)
    {
        sum3_2 += U2[i][k] * tmp2_2[j][k];
    }
    svd_rca2[i][j] = sum3_2;
}
}

//reconstruction
unsigned int svd_Kp, svd_Lp;
svd_Lp = param_l_mod;
svd_Kp = k;

for (unsigned int i = 0; i < tot_size; i++)
{
    out_data[i] = 0;
    out_data2[i] = 0;
}

for (unsigned int i = 0; i <= svd_Lp - 2; i++)
{
    for (unsigned int j = 1; j <= i + 1; j++)
    {
        out_data[i] = out_data[i] + (1.0 / (i + 1)) * svd_rca[j-1][i-←
        -j+1];
        out_data2[i] = out_data2[i] + (1.0 / (i + 1)) * svd_rca2[j-←
        -1][i-j+1];
    }
}

for (unsigned int i = svd_Lp - 1; i <= svd_Kp - 1; i++)
{
    for (unsigned int j = 1; j <= svd_Lp; j++)
    {
        out_data[i] = out_data[i] + (1.0 / svd_Lp) * svd_rca[j-1][i-←
        j+1];
        out_data2[i] = out_data2[i] + (1.0 / svd_Lp) * svd_rca2[j-←
        -1][i-j+1];
    }
}

for (unsigned int i = svd_Kp; i <= tot_size; i++)
{

```

```

    for (unsigned int j = i - svd_Kp + 2; j <= tot_size - svd_Kp + 1; j++)
    {
        out_data[i] = out_data[i] + (1.0 / (tot_size - i)) * svd_rca[j-1][i-j+1];
        out_data2[i] = out_data2[i] + (1.0 / (tot_size - i)) * svd_rca2[j-1][i-j+1];
    }

    return XST_SUCCESS;
}

/*
 *
 * This is the DMA TX Interrupt handler function.
 *
 * It gets the interrupt status from the hardware, acknowledges it, and if any
 * error happens, it resets the hardware. Otherwise, if a completion interrupt
 * is present, then sets the TxDone.flag
 *
 * @param Callback is a pointer to TX channel of the DMA engine.
 *
 * @return None.
 *
 * @note None.
 *
 */
void TxIntrHandler(void *Callback)
{
    u32 IrqStatus;
    int TimeOut;
    XAxiDma *AxiDmaInst = (XAxiDma *)Callback;

    /* Read pending interrupts */
    IrqStatus = XAxiDma_IntrGetIrq(AxiDmaInst, XAXIDMA_DMA_TO_DEVICE);

    /* Acknowledge pending interrupts */

    XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DMA_TO_DEVICE);

    /*
     * If no interrupt is asserted, we do not do anything

```

```

* error happens, it resets the hardware. Otherwise, if a completion ←
  interrupt
* is present, then it sets the RxDone flag.
*
* @param Callback is a pointer to RX channel of the DMA engine.
*
* @return None.
*
* @note None.
*
*****←
  */
void RxIntrHandler(void *Callback)
{
    u32 IrqStatus;
    int TimeOut;
    XAxiDma *AxiDmaInst = (XAxiDma *)Callback;

    /* Read pending interrupts */
    IrqStatus = XAxiDma_IntrGetIrq(AxiDmaInst, XAXIDMA_DEVICE_TO_DMA)←
        ;

    /* Acknowledge pending interrupts */
    XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DEVICE_TO_DMA);

    /*
     * If no interrupt is asserted, we do not do anything
     */
    if (!(IrqStatus & XAXIDMA_IRQ_ALL_MASK)) {
        return;
    }

    /*
     * If error interrupt is asserted, raise error flag, reset the
     * hardware to recover from the error, and return with no further
     * processing.
     */
    if ((IrqStatus & XAXIDMA_IRQ_ERROR_MASK)) {

        Error = 1;

        /* Reset could fail and hang
         * NEED a way to handle this or do not call it??
         */
        XAxiDma_Reset(AxiDmaInst);

        TimeOut = RESET_TIMEOUT_COUNTER;

        while (TimeOut) {
            if(XAxiDma_ResetIsDone(AxiDmaInst)) {
                break;
            }
        }
    }
}

```

```
        TimeOut -= 1;
    }

    return;
}

/*
 * If completion interrupt is asserted, then set RxDone flag
 */
if ((IrqStatus & XAXIDMA_IRQ_IOC_MASK)) {

    RxDone = 1;
}
}
```

A.2.4 dma_interrupt.h

```
#ifndef SRC_DMA_INTERRUPT_
#define SRC_DMA_INTERRUPT_

#include "xaxidma.h"
#include "xparameters.h"
#include "xil_exception.h"
#include "xdebug.h"
#include "xscugic.h"
/***** Constant Definitions ←
******/

/* Device hardware build related constants. */

#define DMA_DEV_ID        XPAR_AXIDMA_0_DEVICE_ID

#define RX_INTR_ID        XPAR_FABRIC_AXI_DMA_0_S2MM_INTROUT_INTR
#define TX_INTR_ID        XPAR_FABRIC_AXI_DMA_0_MM2S_INTROUT_INTR

#define INTC_DEVICE_ID    XPAR_SCUGIC_SINGLE_DEVICE_ID

#define INTC                XScuGic
#define INTC_HANDLER        XScuGic_InterruptHandler

/* Timeout loop counter for reset */
#define RESET_TIMEOUT_COUNTER 10000

/***** Function Prototypes ←
******/
#ifndef DEBUG
extern void xil_printf(const char *format, ...);
```

```
#endif

void TxIntrHandler(void *Callback);
void RxIntrHandler(void *Callback);

int SetupIntrSystem(INTC * IntcInstancePtr,
                    XAxiDma * AxiDmaPtr, u16 TxIntrId, u16 RxIntrId);
void DisableIntrSystem(INTC * IntcInstancePtr,
                       u16 TxIntrId, u16 RxIntrId);

#endif /* SRC_DMA_INTERRUPT_ */
```

A.2.5 dma_interrupt.c

```
#include "dma_interrupt.h"
/*****
/*
* This function setups the interrupt system so interrupts can occur ←
for the
* DMA, it assumes INTC component exists in the hardware system.
*
* @param IntcInstancePtr is a pointer to the instance of the INTC.
* @param AxiDmaPtr is a pointer to the instance of the DMA engine
* @param TxIntrId is the TX channel Interrupt ID.
* @param RxIntrId is the RX channel Interrupt ID.
*
* @return
* - XST_SUCCESS if successful,
* - XST_FAILURE if not succesful
*
* @note None.
*/
*****
int SetupIntrSystem(INTC * IntcInstancePtr,
                    XAxiDma * AxiDmaPtr, u16 TxIntrId, u16 RxIntrId)
{
    int Status;

#ifdef XPAR_INTC_0_DEVICE_ID

    /* Initialize the interrupt controller and connect the ISRs */
    Status = XIntc_Initialize(IntcInstancePtr, INTC_DEVICE_ID);
    if (Status != XST_SUCCESS) {

        xil_printf("Failed init intc\r\n");
        return XST_FAILURE;
    }
#endif
}
```

```

Status = XIntc_Connect(IntcInstancePtr, TxIntrId,
                      (XInterruptHandler) TxIntrHandler, AxiDmaPtr);
if (Status != XST_SUCCESS) {

    xil_printf("Failed tx connect intc\r\n");
    return XST_FAILURE;
}

Status = XIntc_Connect(IntcInstancePtr, RxIntrId,
                      (XInterruptHandler) RxIntrHandler, AxiDmaPtr);
if (Status != XST_SUCCESS) {

    xil_printf("Failed rx connect intc\r\n");
    return XST_FAILURE;
}

/* Start the interrupt controller */
Status = XIntc_Start(IntcInstancePtr, XIN_REAL_MODE);
if (Status != XST_SUCCESS) {

    xil_printf("Failed to start intc\r\n");
    return XST_FAILURE;
}

XIntc_Enable(IntcInstancePtr, TxIntrId);
XIntc_Enable(IntcInstancePtr, RxIntrId);

#else

XScuGic_Config *IntcConfig;

/*
 * Initialize the interrupt controller driver so that it is ready↔
 * to
 * use.
 */
IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);
if (NULL == IntcConfig) {
    return XST_FAILURE;
}

Status = XScuGic_CfgInitialize(IntcInstancePtr, IntcConfig,
                               IntcConfig->CpuBaseAddress);
if (Status != XST_SUCCESS) {
    return XST_FAILURE;
}

XScuGic_SetPriorityTriggerType(IntcInstancePtr, TxIntrId, 0xA0, 0↔
                               x3);

```

```

XScuGic_SetPriorityTriggerType(IntcInstancePtr, RxIntrId, 0xA0, 0←
    x3);
/*
 * Connect the device driver handler that will be called when an
 * interrupt for the device occurs, the handler defined above ←
 * performs
 * the specific interrupt processing for the device.
 */
Status = XScuGic_Connect(IntcInstancePtr, TxIntrId,
    (Xil_InterruptHandler)TxIntrHandler,
    AxiDmaPtr);
if (Status != XST_SUCCESS) {
    return Status;
}

Status = XScuGic_Connect(IntcInstancePtr, RxIntrId,
    (Xil_InterruptHandler)RxIntrHandler,
    AxiDmaPtr);
if (Status != XST_SUCCESS) {
    return Status;
}

XScuGic_Enable(IntcInstancePtr, TxIntrId);
XScuGic_Enable(IntcInstancePtr, RxIntrId);

#endif

/* Enable interrupts from the hardware */

Xil_ExceptionInit();
Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
    (Xil_ExceptionHandler)INTC_HANDLER,
    (void *)IntcInstancePtr);

Xil_ExceptionEnable();

return XST_SUCCESS;
}

/*****
**
* This function disables the interrupts for DMA engine.
*
* @param IntcInstancePtr is the pointer to the INTC component ←
* instance
* @param TxIntrId is interrupt ID associated w/ DMA TX channel
* @param RxIntrId is interrupt ID associated w/ DMA RX channel
*
* @return None.
*
**
*/

```

```

* @note      None.
*
*****/
void DisableIntrSystem(INTC * IntcInstancePtr,
                       u16 TxIntrId, u16 RxIntrId)
{
#ifdef XPAR_INTC_0_DEVICE_ID
    /* Disconnect the interrupts for the DMA TX and RX channels */
    XIntc_Disconnect(IntcInstancePtr, TxIntrId);
    XIntc_Disconnect(IntcInstancePtr, RxIntrId);
#else
    XScuGic_Disconnect(IntcInstancePtr, TxIntrId);
    XScuGic_Disconnect(IntcInstancePtr, RxIntrId);
#endif
}

```

A.2.6 vga.h

```

#ifndef SRC_VGA_H_
#define SRC_VGA_H_

#include "xparameters.h"
#include <string>

//colors definition
#define YELLOW    0xFFFFF0
#define WHITE     0FFFFFFF
#define BLACK     0x000000
#define GRAY      0x707B7C
#define SOFT_GRAY 0x515A5A
#define DARK_WHITE 0xB3B6B7
#define MY_BLUE   0x000099
#define MY_RED    0xB71C1C
#define FUCHSIA   0xFF00FF

//resolution
#define DISPLAY_COLUMNS 640
#define DISPLAY_ROWS    480

/*Maximum and minimum definition
Acceptable values (max > min) : ←→
1000–950–900–850–800–750–700–650–600–550–500–450–400–350–300–250–200–150–100–
*/
#define MAX1 1000 //maximum value ch1(mV)
#define MIN1 0 //minimum value ch1(mV)
#define MAX2 1000 //maximum value ch2(mV)
#define MIN2 0 //minimum value ch2(mV)

void SetupVGA();

```

```
void numbers(char num, int r, int c, char col);
void char_fix();
void char_var1();
void char_var2();
void blank_screen();
void plot_input(float *in_data, unsigned int size, float *in_data2, ←
    unsigned int size2);
void plot_output(float *in_data, unsigned int size, float *in_data2, ←
    unsigned int size2);

#endif /* SRC_VGA_H */
```

A.2.7 vga.cc

```
#include "vga.h"

// VDMA (VGA)
volatile unsigned int * VDMA_VGA = (unsigned int *)0x43000000;
volatile unsigned int * VDMA_VGA_CR = (unsigned int *)0x43000000;
volatile unsigned int * VDMA_VGA_AD = (unsigned int *)0x4300005C;
volatile unsigned int * VDMA_VGA_ST = (unsigned int *)0x43000058;
volatile unsigned int * VDMA_VGA_HS = (unsigned int *)0x43000054;
volatile unsigned int * VDMA_VGA_VS = (unsigned int *)0x43000050;

void SetupVGA()
{
    *(VDMA_VGA_CR) = 0x1; // Enable VDMA
    *(VDMA_VGA_AD) = 0x10000000; // The base address of the ←
    frame buffer
    *(VDMA_VGA_ST) = 4*DISPLAY_COLUMNS; // Line size (stride value←
    )
    *(VDMA_VGA_HS) = 4*DISPLAY_COLUMNS; // Horizontal Pixel Count ←
    * 4B/pixel
    *(VDMA_VGA_VS) = DISPLAY_ROWS; // Vertical Line Count
}

void blank_screen()
{
    volatile unsigned int *pStorageMem = (unsigned int *)0x10000000; ←
    //frame buffer init

    //horizontal grid lines
    for(unsigned int r = 24; r < 480; r=r+24)
    {
        if(r != 240 )
        {
            for(unsigned int c = 3; c < 639; c=c+8)
            {
```

```

        *(pStorageMem+(r*640)+c) = SOFT_GRAY;
        *(pStorageMem+(r*640)+c+1) = SOFT_GRAY;
        *(pStorageMem+(r*640)+c+2) = SOFT_GRAY;
        *(pStorageMem+(r*640)+c+3) = SOFT_GRAY;
        if (*(pStorageMem+(r*640)+c+4) != WHITE) *(pStorageMem+(←
            r*640)+c+4) = BLACK;
        if (*(pStorageMem+(r*640)+c+5) != WHITE) *(pStorageMem+(←
            r*640)+c+5) = BLACK;
        if (*(pStorageMem+(r*640)+c+6) != WHITE) *(pStorageMem+(←
            r*640)+c+6) = BLACK;
        if (*(pStorageMem+(r*640)+c+7) != WHITE) *(pStorageMem+(←
            r*640)+c+7) = BLACK;
    }
}

//vertical grid lines
for(unsigned int c = 40; c < 640; c=c+40)
{
    if(c != 320)
    {
        for(unsigned int r = 1; r < 479; r=r+8)
        {
            *(pStorageMem+(r*640)+c) = SOFT_GRAY;
            *(pStorageMem+((r+1)*640)+c) = SOFT_GRAY;
            *(pStorageMem+((r+2)*640)+c) = SOFT_GRAY;
            *(pStorageMem+((r+3)*640)+c) = SOFT_GRAY;
            if (*(pStorageMem+((r+4)*640)+c) != WHITE) *(pStorageMem←
                +((r+4)*640)+c) = BLACK;
            if (*(pStorageMem+((r+5)*640)+c) != WHITE) *(pStorageMem←
                +((r+5)*640)+c) = BLACK;
            if (*(pStorageMem+((r+6)*640)+c) != WHITE) *(pStorageMem←
                +((r+6)*640)+c) = BLACK;
            if (*(pStorageMem+((r+7)*640)+c) != WHITE) *(pStorageMem←
                +((r+7)*640)+c) = BLACK;
        }
    }
}

//characters
char_fix(); //permanent characters
char_var1(); //variable characters ch1
char_var2(); //variable characters ch2

//blank screen and axes
for(unsigned int r = 0; r < 480; r++)
{
    for(unsigned int c = 0; c < 640; c++)
    {
        if(r == 240 || r == 0 || r == 479 || c == 320 || c == 0 || ←
            c == 639) *(pStorageMem+(r*640)+c) = WHITE;
    }
}

```

```

        else {if(*(pStorageMem+(r*640)+c) != WHITE && *(pStorageMem←
+(r*640)+c) != SOFT_GRAY && *(pStorageMem+(r*640)+c) !=←
DARK_WHITE) *(pStorageMem+(r*640)+c) = BLACK;}
    }
}
}

void plot_input(float *in_data, unsigned int size, float *in_data2, ←
unsigned int size2)
{
    int center = DISPLAY_ROWS/4;
    //voltage middle point
    float mid1 = (float)((MAX1-MIN1)/2)/1000;
    float mid2 = (float)((MAX2-MIN2)/2)/1000;
    //pixel middle point
    int level1 = center/mid1;
    int level2 = center/mid2;
    //pixel offset
    int offset_n1 = ((float)MIN1/1000) * level1;
    int offset_n2 = ((float)MIN2/1000) * level2;

    volatile unsigned int *pStorageMem = (unsigned int *)0x10000000; ←
    //frame buffer init
    pStorageMem = pStorageMem + (640*480); //decreasing counter

    //print waveform
    for(unsigned int ind_up = 0; ind_up < 104; ind_up++)
    {
        unsigned int cnt = 638;
        for(unsigned int i = ind_up; i < size; i = i+104)
        {
            if ( (((int)(in_data[i]*level1-offset_n1) +240)*640)+cnt >←
153600) *(pStorageMem-((( (int)(in_data[i]*level1←
offset_n1) +240)*640)+cnt)) = YELLOW; //CH1
            if ((( (int)(in_data2[i]*level2-offset_n2) *640)+cnt) < ←
153600) *(pStorageMem-(( (int)(in_data2[i]*level2←
offset_n2) *640)+cnt)) = FUCHSIA; //CH2
            cnt--;
        }
    }
}

void plot_output(float *in_data, unsigned int size, float *in_data2,←
unsigned int size2)
{
    int center = DISPLAY_ROWS/4;
    //voltage middle point
    float mid1 = (float)((MAX1-MIN1)/2)/1000;
    float mid2 = (float)((MAX2-MIN2)/2)/1000;
    //pixel middle point
    int level1 = center/mid1;
    int level2 = center/mid2;

```

```
//pixel offset
int offset_n1 = ((float)MIN1/1000) * level1;
int offset_n2 = ((float)MIN2/1000) * level2;

volatile unsigned int *pStorageMem = (unsigned int *)0x10000000; ←
//frame buffer init
pStorageMem = pStorageMem + (640*480); //decreasing counter

//print waveform
for(unsigned int ind_up = 0; ind_up < 104; ind_up++)
{
    unsigned int cnt = 318;
    for(unsigned int i = ind_up; i < size; i = i+104)
    {
        if ( (((int)(in_data[i]*level1-offset_n1) +240)*640)+cnt >←
            153600) *(pStorageMem-((( (int)(in_data[i]*level1-←
            offset_n1) +240)*640)+cnt)) = YELLOW; //CH1
        if ((( (int)(in_data2[i]*level2-offset_n2) *640)+cnt) < ←
            153600) *(pStorageMem-(( (int)(in_data2[i]*level2-←
            offset_n2) *640)+cnt)) = FUCHSIA; //CH2
        cnt--;
    }
}
}
```

Appendice B

Codice HLS

B.1 myfilterHLS.h

```
#ifndef SRC_MYFILTER_HLS_
#define SRC_MYFILTER_HLS_

#define data_size 1370
#define param_l 21

#include <math.h>

void X_prod(const float in_data[(data_size+param_l)*2], float ←
            out_data[param_l*param_l*2], bool *out_data_TLAST);

#endif /* SRC_MYFILTER_HLS_ */
```

B.2 myfilterHLS.cc

```
#include "myfilterHLS.h"

void X_prod(const float in_data[(data_size+param_l)*2], float ←
            out_data[param_l*param_l*2], bool *out_data_TLAST)
{
    #pragma HLS INTERFACE ap_none port=out_data_TLAST
    #pragma HLS INTERFACE axis port=in_data
    #pragma HLS INTERFACE axis port=out_data
```

```

*out_data_TLAST = 0;
float in_buf[data_size+param_1];
float X[param_1][data_size];
float in_buf2[data_size+param_1];
float X2[param_1][data_size];

unsigned int ind_in = 0;
for (unsigned int i = 0; i < (data_size+param_1)*2; i=i+2)
{
    #pragma HLS PIPELINE II=1
    in_buf[ind_in] = in_data[i];
    in_buf2[ind_in] = in_data[i+1];
    ind_in++;
}

for (unsigned int i = 0; i < data_size; i++)
{
    #pragma HLS PIPELINE II=1
    for (unsigned int j = 0; j < param_1; j++)
    {
        X[j][i] = in_buf[j + i];
        X2[j][i] = in_buf2[j + i];
    }
}

float sum = 0;
float sum2 = 0;
unsigned int ind_out = 0;
for(unsigned int i = 0; i < param_1; ++i)
{
    for(unsigned int j = 0; j < param_1; ++j)
    {
        sum = 0;
        sum2 = 0;
        for(unsigned int k = 0; k < data_size; ++k)
        {
            #pragma HLS UNROLL factor=10
            #pragma HLS PIPELINE II=1
            sum += X[i][k] * X[j][k];
            sum2 += X2[i][k] * X2[j][k];
        }
        out_data[ind_out] = sum;
        out_data[ind_out+1] = sum;
        if(i >= param_1-1 && j >= param_1-1 && out_data[ind_out+1] <=
            0.0000) *out_data_TLAST = 1;
        ind_out = ind_out+2;
    }
}
}

```

Appendice C

Controller VGA

C.1 vga.v

```
`timescale 1ns / 1ps

module vga640x480(
    input wire clk25,          //pixel clock: 25MHz
    input wire aresetn,       //asynchronous reset
    output wire hsync,        //horizontal sync out
    output wire vsync,        //vertical sync out
    output reg [3:0] red,     //red vga output
    output reg [3:0] green,   //green vga output
    output reg [3:0] blue,    //blue vga output
    input wire [31:0] pixel_data,
        input wire tvalid,
        output reg tready,
        output reg fsync
    );

// video structure constants
parameter hpixels = 800; // horizontal pixels per line
parameter vlines = 521; // vertical lines per frame
parameter hpulse = 96;  // hsync pulse length
parameter vpulse = 2;   // vsync pulse length
parameter hbp = 144;    // end of horizontal back porch
parameter hfp = 784;    // beginning of horizontal front porch
parameter vbp = 31;     // end of vertical back porch
parameter vfp = 511;    // beginning of vertical front porch
// active horizontal video is therefore: 784 - 144 = 640
// active vertical video is therefore: 511 - 31 = 480

// registers for storing the horizontal & vertical counters
```

```
reg [9:0] hc;
reg [9:0] vc;
// Horizontal & vertical counters --
// this is how we keep track of where we are on the screen.

always @(posedge clk25)
begin
    // reset condition
    if (~aresetn)
    begin
        hc <= 0;
        vc <= 0;
    end
    else
    begin
        // keep counting until the end of the line
        if (hc < hpixels - 1)
            hc <= hc + 1;
        else
        // When we hit the end of the line, reset the horizontal
        // counter and increment the vertical counter.
        // If vertical counter is at the end of the frame, then
        // reset that one too.
        begin
            hc <= 0;
            if (vc < vlines - 1)
                vc <= vc + 1;
            else
                vc <= 0;
        end
    end
end

// generate sync pulses (active low)

assign hsync = (hc < hpulse) ? 0:1;
assign vsync = (vc < vpulse) ? 0:1;

// display 100% saturation colorbars

always @(*)
begin
    // first check if we're within vertical active video range
    if (vc >= vbp && vc < vfp)
    begin
        // now display different colors every 80 pixels
        // while we're within the active horizontal range
        // -----
        // display
    end
end
```

```
    if (hc >= hbp && hc < (hbp+640))
    begin
        red = pixel_data[23:20];
        green = pixel_data[15:12];
        blue = pixel_data[7:4];
        tready = 1;
    end
    // we're outside active horizontal range so display black
    else
    begin
        red = 0;
        green = 0;
        blue = 0;
        tready = 0;
    end
end
// we're outside active vertical range so display black
else
begin
    red = 0;
    green = 0;
    blue = 0;
    tready = 0;
end
end

reg vsync_last;
always@(posedge clk25)
begin
    if(~aresetn) begin
        vsync_last <= 0;
        fsync <= 0;
    end else begin
        vsync_last <= vsync;
        if (~vsync_last & vsync) begin
            fsync <= 1;
        end else begin
            fsync <= 0;
        end
    end
end
end

endmodule
```

Bibliografia

- [1] Bojja Venkata Hemambhar, Sheeba Rani J, *Denoising of ECG signals using Fuzzy based Singular Spectrum Analysis*, Department of Avionics, Indian Institute of Space Science and Technology Thiruvananthapuram, India, 2018.
- [2] Tao Zeng, JiaLi Ma, MingChui Dong, *Environmental Noise Elimination of Heart Sound Based on Singular Spectrum Analysis*, Department of Electrical and Computer Engineering, Faculty of Science and Technology, University of Macau, China, 2014.
- [3] *NI-9222 Datasheet*, http://www.ni.com/pdf/manuals/374210a_02.pdf, National Instruments.
- [4] *NI IC-3173 Datasheet*, <http://www.ni.com/pdf/manuals/375284d.pdf>, National Instruments.
- [5] *ZYNQ-7000 SoC Datasheet*, https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf, Xilinx.
- [6] *XADC Datasheet*, https://www.xilinx.com/support/documentation/user_guides/ug480_7Series_XADC.pdf, Xilinx.
- [7] *Zedboard User Guide*, http://www.zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf, Avnet.
- [8] *Vivado Design Suite User Guide, High-Level Synthesis*, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug902-vivado-high-level-synthesis.pdf, Xilinx.
- [9] *Zedboard Schematic*, https://www.xilinx.com/support/documentation/university/XUP%20Boards/XUPZedBoard/documentation/ZedBoard_RevC.1_Schematic_130129.pdf, Avnet.
- [10] *AXI Reference Guide*, https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf, Xilinx.
- [11] *AXI DMA v7.1 LogiCORE IP Product Guide*, https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf, Xilinx.
- [12] *XAPP1170*, https://www.xilinx.com/support/documentation/application_notes/xapp1170-zynq-hls.pdf, Xilinx.

- [13] *AXI VDMA v6.2 LogiCORE IP Product Guide*, https://www.xilinx.com/support/documentation/ip_documentation/axi_vdma/v6_2/pg020_axi_vdma.pdf, Xilinx.