University of Cyprus

DEPARTMENT OF ELECTRICAL AND

COMPUTER ENGINEERING

A BIOINSPIRED SYSTEM FOR ACOUSTIC SCENE ANALYSIS

GUILLAUME GARREAU

A Dissertation Submitted to the University of Cyprus in Partial

Fulfillment of the Requirements for the Degree of Doctor of Philosophy

March 2014

VALIDATION PAGE

Doctoral Candidate: Guillaume Garreau

Doctoral Thesis Title: A Bioinspired System for Acoustic Scene Analysis

The present Doctorate Dissertation was submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in the Department of Electrical and Computer Engineering, and was approved on March 10th, 2014 by the members of the Examination Committee.

Examination Committee:

Committee Chair _____
Marios Polycarpou, Ph.D.

Research Supervisor _____
Julius Georgiou, Ph.D.

Committee Member _____
Constantinos Pitris, Ph.D, M.D.

Committee Member _____
Chris Christodoulou, Ph.D.

Committee Member _____
Giacomo Indiveri, Ph.D.

The present Doctoral Dissertation was submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy of the University of Cyprus. It is a product of original work of my own, unless otherwise mentioned through references, notes, or any other statements.

.....

.....

Περίληψη

Η ακουστική ανάλυση χωρικού πεδίου είναι ένα αχανές ερευνητικό πεδίο το οποίο παραδοσιακά επικεντρώνεται στην αναγνώριση του ήχου στο ακουστικό φάσμα καθώς και στον τρόπο με τον οποίο ο εγκέφαλος διαχωρίζει και ταξινομεί τις διάφορες πηγές ήχου σε σημασιολογικά αντικείμενα. Οι προγενέστερες έρευνες επικεντρώθηκαν εύλογα κυρίως στο ακουστικό φάσμα, που είναι και το πιο οικείο στους ανθρώπους. Το ακουστικό φάσμα εμπεριέχει, την ανθρώπινη επικοινωνία, το θόρυβο των αυτοκινήτων, τον άνεμο, τα κελάιδισμα των πουλιών, το γάβγισμα των ζώων, το μουγκρητό, το σφύριγμα... και οι γλώσσες μας είναι πλούσιες στο να τα περιγράφουν όλα αυτά [99, 245].

Ωστόσο, πολλές πληροφορίες περιέχονται και σε άλλα συχνοτικά φάσματα, όπως το φάσμα των υπερήχων και το φάσμα των πολύ χαμηλών συχνοτήτων (σεισμικό φάσμα). Ένα παράδειγμα είναι οι νυχτερίδες και τα δελφίνια που ως γνωστό χαρακτηρίζονται από την ικανότητα τους να εντοπίζουν και να αρπάζουν την τροφή τους αναδίδοντας και λαμβάνοντας υπερήχους. Αρκετά είδη πουλιών, τα ποντίκι, τα ψάρια και τα έντομα με παρόμοιο τρόπο κυνηγούν τη λεία τους, βρίσκουν ταίρι και επικοινωνούν μέσω υπερήχων. Οι δονήσεις χαμηλών συχνοτήτων χρησιμοποιούνται από τους σκορπιούς κατά τη νυκτερινή αναζήτηση σκουληκιών.

Έχουν την ικανότητα να ανιχνεύουν μετατοπίσεις την άμμου της τάξης του $0.1nm$ και να εντοπίζουν την κατεύθυνση του θηράματος τους [218]. Επιπλέων, οι ελέφαντες με το κτύπημα του ποδιού τους στο έδαφος, αναδίδουν και λαμβάνουν σεισμικές δονήσεις στο φάσμα $10Hz$-$40Hz$, καθιστώντας έτσι δυνατή την επικοινωνία τους σε απόσταση μέχρι και $16km$ [181]. Τέλος, οι φάλαινες χρησιμοποιούν χαμηλές συχνότητες της τάξης των $14Hz$ αναδίδοντας με ένταση που ξεπερνά τα $180dB\ re\ 1\mu Pa$ στο $1m$, για την επικοινωνία τους σε μεγάλες αποστάσεις ($1600km$) [246].

Το παρουσιασθέν ερευνητικό έργο χρησιμοποιεί ένα ευρύ ακουστικό φάσμα συχνοτήτων, για την εξαγωγή πληροφοριών που αφορούν ένα χωρικό πεδίο. Μέρος αυτής της εργασίας εμπεριέχει τη δόμηση ενός πρωτότυπου συστήματος συλλογής ακουστικών

δεδομένων αφού δεν υπάρχει εμπορικά διαθέσιμο σύστημα που να έχει τη δυνατότητα να λειτουργεί σε ένα τόσο ευρύ συχνοτικό φάσμα, χρησιμοποιώντας πολλαπλούς διανεμημένους αισθητήρες και διατηρώντας ακριβές συγχρονισμό και υψηλή δυναμική περιοχή. Επιπρόσθετα, το εν λόγω σύστημα σχεδιάστηκε με νευρο-μορφική προσέγγιση καθιστώντας δυνατή τη υλοποίηση βιο-εμπνευσμένων μοντέλων σε επίπεδο υλικού χρησιμοποιώντας διατάξεις πεδιακά προγραμματιζομένων πυλών (*FPGAs*) που βρίσκονται στις θυγατρικές κάρτες που είναι τοποθετημένες στους ακουστικούς κόμβους. Επιπλέων, αναπτύχθηκαν/εφαρμόστηκαν ποικίλοι βιο-εμπνευσμένοι αλγόριθμοι, για την επεξεργασία των δεδομένων που συλλέχθηκαν και την δημιουργία σημασιολογικών ακουστικών αντικειμένων. Τέλος, αυτά τα ακουστικά αντικείμενα ταξινομούνται και διαβαθμίζονται αυτόματα μέσω του συστήματος ακουστικής ανάλυσης χωρικού πεδίου σε πραγματικό χρόνο.

Επισκόπηση Κεφαλαίων

Το Κεφάλαιο 1 εξηγεί το λόγο για τον οποίο η ακουστική ανάλυση χωρικού πεδίου είναι σημαντική καθώς και το λόγο για τον οποίο η νευρο-μορφική τεχνολογία δύναται να καταστήσει εφικτή την ανάπτυξη γρηγορότερων, λιγότερο ενεργοβόρων και ισχυρότερων συστημάτων ακουστικής επεξεργασίας.

Το Κεφάλαιο 2 περιγράφει τη σχεδίαση του βιο-εμπνευσμένου συστήματος συλλογής ακουστικών δεδομένων, παρουσιάζει την λύση που εφαρμόστηκε για την ανάπτυξη του υλικού και αποδεικνύει τις μοναδικές του ικανότητες για υψηλού συγχρονισμού πολυκάναλη συλλογή δεδομένων. Το εν λόγω σύστημα επιτρέπει την υλοποίηση βιο-εμπνευσμένων μοντέλων σε επίπεδο υλικού.

Το Κεφάλαιο 3 δείχνει ότι χρησιμοποιώντας υλικό ακουστικής συλλογής δεδομένων και σεισμικών ήχων, δύναται να εντοπιστεί και να ακολουθηθεί ένα άτομο που βαδίζει. Για το σκοπό αυτό χρησιμοποιείται ένα μοντέλο βασισμένο σε παλμικό νευρωνικό δίκτυο.

Το Κεφάλαιο 4 δείχνει ότι χρησιμοποιώντας υλικό ακουστικής συλλογής δεδομένων και υπερήχους δύναται να ταυτοποιηθούν άτομα, το μέσο διακίνησης τους καθώς και να ταυτοποιηθεί το φύλο τους και να διαβαθμιστούν ποικίλες δράσεις αυτών.

Το Κεφάλαιο 5 παρουσιάζει ένα απλοποιημένο μοντέλο της ακουστικής οδού (βασική μεμβράνη και θαλαμο-φλοιώδεις δίκτυο) και παρουσιάζει τα αποτελέσματα της εφαρμογής του εν λόγω μοντέλου σε μερική υλοποίηση υλικού σε διατάξεις πεδιακά προγραμματιζομένων πυλών (*FPGAs*), λαμβάνοντας τα δεδομένα με συνεχές ροή πραγματικού χρόνου από το εξατομικευμένο υλικό που παρουσιάστηκε.

Το Κεφάλαιο 6 δείχνει την υλοποίηση ενός πολυτροπικού συστήματος που συγχωνεύει ετερογενή ακουστικά δεδομένα, ο οποίος τείνει να αυξάνει την επίδοση των λειτουργιών αναγνώρισης και ταξινόμησης, συγκρινόμενο με μονοτροπικά συστήματα.

Τέλος, το Κεφάλαιο 7 κλίνει τη διατριβή και διερευνά μελλοντικές κατευθύνσεις για την εν λόγω έρευνα.

Abstract

Acoustic scene analysis is a vast field that traditionally focuses on sound recognition in the audible range and on how the brain segregates and classifies different sound sources into meaningful objects. Previous research has mainly been focused on the audible range, since this is the most familiar to humans. It is filled with human communication, cars noises, wind, birds songs, animals barking, roars, whistles... and our languages are rich in words to describe them all [99, 245].

A lot of information however is also contained in other frequency ranges, such as the ultrasonic range and the very-low frequency range (seismic range). For example, bats and dolphins are well known for their ability to locate and catch food through ultrasonic vocalisation and reception. Similarly, several bird species, mice, fish and insects also hunt for prey, find a mate, or communicate through ultrasonic sounds. Low-frequency vibrations are used by sand scorpions to find worms at night. They can detect variations in sand as small as 0.1nm and pinpoint the direction of their prey [218]. Furthermore, elephants communicate up to 16km away [181] by stamping their legs on the ground to 'vocalise' and 'listening' for seismic vibrations through their legs, in the 10Hz-40Hz range. Finally, whales use frequencies as low as 14Hz to communicate over huge distances (1600km) by vocalising at volumes that exceed 180dB re 1μPa at 1m [246].

The presented research utilizes a broad acoustic frequency range in order to extract information about a scene. Since there is no commercially available hardware that can tackle such a broad frequency range from multiple distributed sensors and maintain accurate timing, whilst maintaining a very high dynamic range, part of this work involved building a unique acoustic data collection system. In addition, this system was designed with neuromorphic applications in mind and thus allows hardware implementation of bioinspired models through FPGAs contained on daughterboards mounted on the acoustic nodes. Furthermore, various bioinspired

algorithms were developed/implemented, to process the collected data and to create meaningful auditory objects. Finally, these objects were automatically classified as part of a real-time auditory scene analysis system.

Overview of the chapters

Chapter 1 explains why acoustic scene analysis is important and why neuromorphic technology can enable the development of faster, more energy efficient and powerful acoustic processing systems.

Chapter 2 describes the design of the bioinspired hardware for the acoustic data collection system, presents the solution developed and proves its unique capabilities for highly synchronized multi-channels data collection. This system allows hardware implementation of bioinspired model.

Chapter 3 shows that using the acoustic data collection hardware and seismic sounds one can locate a person walking and follow it. A sand-scorpion based spiking neuron network model is used.

Chapter 4 shows that using the acoustic data collection hardware and ultrasonic sounds one can identify individuals, their mode of transport, discriminate their gender and classify various actions.

Chapter 5 presents a simplified model of the auditory pathway (basilar membrane and thalamocortical network) and show the results of the partial hardware implementation of this model on FPGA, streaming the data in real-time from the custom hardware previously presented.

Chapter 6 shows that multimodal sensory fusion has been implemented on acoustic data collected and is shown to increase the performance of recognition and classification tasks, in comparison to single mode systems.

Finally, Chapter 7 concludes the thesis and future directions of this area are discussed.

Acknowledgement

After four and a half years of hard work under the welcoming sun of Cyprus, it is time to finish my PhD thesis and thank all those who assisted, advised and supported me during this wonderful adventure. I hope I will not forget anyone.

First and foremost I would like to thank my PhD supervisor Julius Georgiou who taught me so much during the last four years, he has been a great teacher, patient guide and a friend too. I am glad I had the opportunity to meet him and work with him.

I would like to especially thank Charalambos Andreou, fellow master student, that invited me to apply for a position in the lab, without him I would have never thought about coming to Cyprus for a PhD program. On several occasions he helped me to debug some of the boards and also helped populate some of the hardware units.

I would like to thank all the researchers (Nicoletta Nicolaou, Horacio Rostro-Gonzalez and Adrian Romiński), PhD candidates (Panayiota Demosthenous, Charalambos Andreou, Evripides Kyriakides, Guillermo Stuarts and Christopher Beck) and undergraduate students/trainees (Eleni Proxenou and Cyrille D'Urbal), who supported me in good and bad times! A special mention to Nicoletta for her endless help in proof reading (papers and PhD thesis) and her help for the data processing with the autoregressive modeling. Horacio Rostro-Gonzalez and Adrian Romiński closely collaborated to the cochlea and the thalamocortical model hardware implementation. Eleni Proxenou participated as her undergraduate project to the initial modifications of the scorpion spiking neuron network model.

I would like to thank also the University of Cyprus and the KIOS research center for providing a pleasant, comfortable, and inspiring work environment. Of special mention our two wonderful Administrative Assistants, Skevi Chrysanthou and Despina Petrou and IT Antonis Antoniou, who I know got mad a few times during my

computer emergencies...

I would like to thank all the partners of the EU FP7 project SCANDLE that provided a lot of support and motivation for many aspects of my PhD. In particular, I would like to acknowledge Salvador Dura-Bernal for his close collaboration in the data collection (treadmill experiment) and the fusion of active and passive acoustic data.

I would also like to thank the members of the Sensory Communication and Microsystems Laboratory, Johns Hopkins University, Baltimore (USA). I learnt a lot from them through a fruitful partnership and I felt especially welcome on both visits to the USA in 2010. In particular I would like to thank Professor Andreas G. Andreou, Dr. Philippe Pouliquen, Dr. Andrew Cassidy, Thomas Murray, Joseph Lin and Sean McVeigh. Philippe provided his valuable expertise and close collaboration in the design of the hardware solution and testing of the synchronization solution. Thomas helped in the testing and tuning of the synchronization solution.

Also my interest in doing research was given during my master internship at the MESA+ Institute for Nanotechnology at the University of Twente in Enschede (NL) and a special thank to my supervisors Professor Niels Tas and Dr. Sandeep Unnikrishnan. As well during my master thesis at the Biomedical Signals and System Lab at the University of Twente with Professor Wim L. C. Rutten and all the members of the team.

I would like to thank as well all my friends, futsal teammates, flatmates, Couchsurfers and InterNations members from all other the world and the places I have been studying and working, for all the good moments, the laughs and the support. Special mention to Ainura, Alina, Anne-Lise, Anton, Benoît, Bogdan, Chloé, Diana, Dimitar, Elsa, François, Frédéric, Guillaume GL, Julien D., Julien G., Lucas, Malick, Marios, Mohamed, Nina, Sadique, Salvador, Seyla, Sina, Sophie, Tanya M., Tanya T., Tatiana, Zhon: Merci!

Last but not least, I want to thank my parents, Marie-Dominique and François, my brother and sisters, Astrid, Hélène, Sophie, Stanislas and Marie-Liesse, and my grand-mother Marie-Thérèse, for their love and support and especially for the time home during the last summer during which they spoiled me while I was initiating the writing of the PhD thesis. I love you!

To all and to those I did not mention MERCI...

Publications

Book chapters

1. S. Dura-Bernal, G. Garreau, C. M. Andreou, A. G. Andreou, J. Georgiou, T. Wennekers, and S. Denham, "Human Action Categorization Using Ultrasound Micro-Doppler Signatures," *2nd International Workshop on Human Behavior Understanding (HBU), Lecture Notes in Computer Science (LNCS) 7065*, 18-28, 2011.

Published journal publications

1. T. M. Böhm, L. Shestopalova, A. Bendixen, A. G. Andreou, J. Georgiou, G. Garreau, P. Pouliquen, A. Cassidy, S. L. Denham and I. Winkler, "The Role of Perceived Source Location in Auditory Stream Segregation: Separation Affects Sound Organization, Common Fate Does Not," *Learning and Bistable Perception special issue of Learning and Perception*, Vol. 5, No. 2, 55-72, June 2013.

2. S. Dura-Bernal, G. Garreau, J. Georgiou, A.G. Andreou, S.L. Denham, and T. Wennekers, "Multimodal Integration of Micro-Doppler Sonar and Auditory Signals for Behaviour Classification with Convolutional Networks," *International Journal of Neural Systems*, Vol. 23, No. 5, October 2013.

3. L. Shestopalova, T. M. Böhm, A. Bendixen, A. G. Andreou, J. Georgiou, G. Garreau, B. Hajdu, S. L. Denham and I. Winkler, "Do Audio-Visual Motion Cues Promote Segregation of Auditory Streams?," *Frontiers in Neuroscience, Auditory Cognitive Neuroscience*,Vol. 8, No. 64, 1-11, April 2014.

Published conference proceedings

1. G. Garreau, Eleni Proxenou, Andreas G. Andreou and Julius Georgiou, "Person Localization Through Ground Vibrations using a Sand-Scorpion Inspired Spiking Neural Network," *Proceedings of the 47th Annual Conference on Information Sciences and Systems (CISS)*, 1-4, 2013.

2. G. Garreau, N. Nicolaou and J. Georgiou, "Individual Classification Through Autoregressive Modelling of Micro-Doppler Signatures," *Proceedings of the IEEE Biomedical Circuits and Systems Conference (BioCAS)*, 312-315, 2012.

3. H. Rostro-Gonzales, G. Garreau, A. G. Andreou, J. Georgiou, J.H. Barron-Zambrano and C. Torres-Huitzil, "An FPGA-Based Approach for Parameter Estimation in Spiking Neural Networks," *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, 2897-2900, 2012.

4. G. Garreau, C. M. Andreou, A. G. Andreou, J Georgiou, S. Dura-Bernal, T. Wennekers and S. Denham, "Gait-Based Person and Gender Recognition Using Micro-Doppler Signatures," *Proceedings of the IEEE Biomedical Circuits and Systems Conference (BioCAS)*, 444-447, 2011.

5. S. Dura-Bernal, G. Garreau, C. M. Andreou, A. G. Andreou, J. Georgiou, T. Wennekers, and S. Denham, "Human Action Categorization Using Ultrasound Micro-Doppler Signatures," *Proceedings of the 2nd International Workshop on Human Behavior Understanding (HBU)*, 2011.

6. G. Garreau, N. Nicolaou, C. M. Andreou, C. D'Urbal, G. Stuarts and J. Georgiou, "Computationally Efficient Classification of Human Transport Mode Using Micro-Doppler Signatures," *Proceedings of the 45th Annual Conference on Information Sciences and Systems (CISS)*, 1-4, 2011.

7. P. O. Pouliquen, A. Cassidy, A. G. Andreou, G. Garreau and J. Georgiou, "A Wireless Architecture for Distributed Sensing/Actuation and Pre-Processing with Microsecond Synchronization," *Proceedings of the 45th Annual Conference on Information Sciences and Systems (CISS)*, 1-6, 2011.

8. J. Georgiou, P. Pouliquen, A. Cassidy, G. Garreau, C. Andreou, G. Stuarts, C. d'Urbal, S. Denham, T. Wennekers, R. Mill, I. Winkler, T. M. Böhm, O. Szalárdy, G. M. Klump, S. Jones, A. Bendixen and A. G. Andreou, "A Multimodal-Corpus Data Collection System for Cognitive Acoustic Scene Analysis," *Proceedings of the 45th Annual Conference on Information Sciences and Systems (CISS)*, 1-6, 2011.

Contents

List of Acronyms and Abbreviations

AASP Audio and Acoustic Signal Processing

ADC Analog-to-Digital Converters

AER Address Event Representation

AM Amplitude Modulation

AR Autoregressive Model

ASA Acoustic or Auditory Scene Analysis

ASIC Application Specific Integrated Circuit

ASIP Application-Specific Instruction-set Processor

ASU Acoustic Surveillance Unit

BCSS Basitarsal Compound Slit Sensilla

BNC Bayonet-Neil-Concelman or British Naval Connector

BRAIN Brain Research through Advancing Innovative Neurotechnologies

CASA Computational Acoustic or Auditory Scene Analysis

CD Contrastive Divergence

CDBN Convolutional Deep Belief Network

CISC Complex Instruction Set Computer

CMOS Complementary Metal Oxide Semiconductor

ConvNets Convolutional Neural Networks

CPU Central Processing Unit

CW Continuous Wave

DAC Digital-to-Analog Converters

DACQ unit Data Acquisition unit

DCM Digital Clock Manager

DCM Dynamic Causal Modeling

DNA Deoxyribose Nucleic Acid or Deoxyribonucleic Acid

DRAM Dynamic Random Access Memory

DSNN Dynamic Synapse Neural Network

DSP Digital Signal Processor

EM Electro-Magnetic wave

EM Expectation-Maximization algorithm

FFT Fast Fourier Transform

FLOPS FLoating-point Operations Per Seconds

FM Frequency Modulation

FM SYNC unit Frequency Modulation Synchronization unit

FP7 Seventh Framework Programme for Research and Technological Development

FPE Final Prediction Error

FPGA Field Programmable Gate Array

GMM Gaussian Mixture Model

GPS Global Positioning System

GPU Graphic Processing Unit

HF High Frequency

I&F Integrate and Fire

IC Integrated Circuit

ICA Independent Component Analysis

ICP Integrated Circuit Piezoelectric

IEEE Institute of Electrical and Electronics Engineers

IFR Instantaneous Frequency Reassigned

IIR Infinite Impulse Response

ILP Instruction Level Parallelism

ITD Interaural Time Delay

kNN k-Nearest Neighbour

LAN Local Area Network

LF Low Frequency

LMS Least Mean-Square

LOO Leave-One-Out

LUT Look-Up Table

MCC Maximum Correlation Coefficient

mD micro-Doppler

MEMS Micro Electro-Mechanical Systems

MFCC Mel-Frequency Cepstral Coefficients

MOS Metal Oxide Semiconductor

MOSFET Metal Oxide Semiconductor Field-Effect Transistor

NTP Network Time Protocol

PAMR Professional Amateur Mobile Radio

PBCH Physical Broadcast Channel

PCB Printed Circuit Board

PLL Phase-Locked Loop

PMR Professional Mobile Radio

PSC Post Synaptic Current

PSD Predictive Sparse Decomposition

PTP Precision-Time-Protocol

RBS Reference-Broadcast Synchronization

RC Resistor Capacitor

RF Radio Frequency

RISC Reduced Instruction Set Computer

ROI Regions of Interest

RSSI Received Signal Strength Indicator

RTSI Real-Time System Integration

SAR Synthetic Aperture RADAR

SCANDLE acoustic SCene Analysis for Detection of Living Entities

SGA Spatial-Gradient Algorithm

SGD Stochastic Gradient Descent

SNN Scorpion Neural Network

SNR Signal to Noise Ratio

SoC System on Chip

SOI Silicon On Insulator

SONAR SOund Navigation And Ranging

SPL Sound Pressure Level

SSD Sensory Substitution Device

STDP Spike Timing Dependent Plasticity

STFT Short-Time Fourier Transform

SVM Support Vector Machine

SyNAPSE Systems of Neuromorphic Adaptive Plastic Scalable Electronics

TC ThalamoCortical

TCXO Temperature Controlled Crystal Oscillator

ToA Time of Arrival

TOMI Thread-Optimized Multiprocessor Instruction

UCF User Constraints File

UCY University of CYprus

UFTRX Ultrasonic Frequency TRansceiver

UNIX Uniplexed Information and Computing System

USB Universal Serial Bus

VHDL VHSIC Hardware Description Language

VHF Very High Frequency

VHSIC Very High Speed Integrated Circuit

VLSI Very-Large-Scale Integration

Wi-fi Wireless-Fidelity

List of Figures

List of Tables

Pour Bon-Papa, repose en paix.

"I hope that posterity will judge me kindly,
not only as to the things which I have explained,
but also to those which I have intentionally omitted
so as to leave to others the pleasure of discovery."
René Descartes

Chapter 1

Introduction

"Le seul véritable voyage, le seul bain de Jouvence, ce ne serait pas d'aller vers de nouveaux paysages, mais d'avoir d'autres yeux, de voir l'univers avec les yeux d'un autre, de cent autres, de voir les cent univers que chacun d'eux voit, que chacun d'eux est."

Proust in La Prisonnière

"The only true voyage of discovery, the only fountain of Eternal Youth, would be not to visit strange lands but to possess other eyes, to behold the universe through the eyes of another, of a hundred others, to behold the hundred universes that each of them beholds, that each of them is." The implications of this are simple: "The real voyage of discovery consists not in seeking new landscapes, but in having new eyes." Finding an innovative solution to a challenging problem does not always involve the discovery of a completely new technology, material or physic law but 'simply' by taking an already existing methodology and applying it in a novel way and sometimes in a different field. The work presented in this thesis follows a bioinspired approach: solutions provided by nature itself after millions of years of evolution are utilized to provide an innovative solution to a particular engineering challenge. The work focuses on the challenging question of acoustic scene analysis, for which a bioinspired system is designed and proposed.

1.1 Why acoustic scene analysis?

1.1.1 Scene Analysis

Scene analysis is the extraction of information from the environment in order to take decisions and act. The ability to answer questions before acting is important.

For example if one wants to 'put the yellow triangle on top of the green square', one must answer multiple questions beforehand: *what is a triangle? what is yellow? where is the yellow triangle? what do I need to do to hold and move it? what is a square? what is green? where is the green square? what is 'top'? what do I need to do to put the triangle on top of the square?* When all these questions are answered, the action can be performed and then the goal is completed. However, the environment may interfere with the action, e.g. some wind, a moving square or even both triangle and square moving, could make the task more difficult. Though humans can adapt to a changing environment in an almost automatic way, Colgate and Hogan showed that getting a robot to perform a similar task is extremely challenging [56].

In order to reproduce a similar situation in the field of robotics, researchers have been working on visual scene analysis, the computer's version of visual perception (human). Wang showed that it involves two basic perceptual processes: the segmentation of the visual scene into a set of objects and the recognition of memorized ones [238, 239].

One approach, which has received recent attention, is to create a grammar of actions to segment the scene and task into a set of basic objects and actions (Ryoo and Aggarwal [199], Wallraven *et al.* [237], Pastra and Aloimonos [185]). Coming back to the previous example of the triangle and the square, the grammar would be composed of colors (yellow, green), objects (triangle, square), and action (put on top). Thus we can construct a tree structure whose branches are the different possible combinations of the basic grammatical elements and achieving the aimed task would be equivalent to climbing up the branches of this tree.

Then, one must choose which technology will be used to capture and identify the different segments of the scene. The most widespread technology used for scene analysis is the video camera. Compared to other sensors, cameras are cheap, offer high spatial definition and provide a lot of information regarding objects in the scene; however this high-dimensionality makes them more difficult to parse. Teixeira *et al.* reported a detailed survey on human sensing [223]. One challenge presented in vision-based problems is the high number of false positives. To solve this, one solution is to use motion detection, which naturally limits false detection compared to background subtraction or pattern matching approaches, and has low processing requirements. However, a drawback of motion detection is that if the objects become immobile they also become invisible to the system. Numerous applications of

scene analysis have been reported in the literature with strong emphasis on security applications such as road traffic (Yguel *et al.* [252], Ellis [75]) or natural actions classification (Wallraven *et al.* [237]). However, the use of cameras for scene analysis is not an option for environments where visibility is impaired, e.g. smoke-filled room, absence of light. In addition, the use of camera-based systems for security monitoring in public spaces raises a host of privacy issues.

1.1.2 Acoustic scene analysis

In contrast to visual-based systems and visual scene analysis, acoustic scene analysis show relatively low computational overhead and resistance to occlusions, and is not affected by illumination (Bregman [24], Wang and Brown [240], Teixeira *et al.* [223]). Sounds are also not affected by smoke or by lack of light, and thus offer an alternative solution in environments where video-cameras cannot be used or are not desired. It has to be noted that acoustic scene analysis is not limited to auditory scene analysis. The audible range is usually referred to as frequencies between 20Hz and 20kHz, but audio range has a much wider frequency bandwidth: from lower frequencies and waves travelling through the ground to higher frequencies such as ultrasounds used in sonars. Benetos *et al.* showed that acoustic scene analysis can be used to classify full scenes such as train station, an outdoor scene, a school, a kitchen or individual acoustic objects (speech recognition or other noises) [12]. The growing interest in acoustic scene analysis motivated the organisation of a competition, the Institute of Electrical and Electronics Engineers (IEEE) Audio and Acoustic Signal Processing (AASP) Challenge, in order to encourage research and development with comparable and repeatable results, and to stimulate new ground-breaking approaches to specific problems in the AASP technical scope. In 2012, the challenge was dedicated to acoustic scene analysis and the evaluation of the performance of systems for the detection and classification of acoustic events and audio scenes [104].

Acoustic scene analysis does not replace camera-based systems, but has access to information that is otherwise inaccessible. Therefore, camera-based and acoustic-based scene analysis should be used as complementary solutions.

1.2 Why bioinspired systems?

Nowadays classical electronics is reaching a "Brick Wall". This wall is composed of three walls (Patterson *et al.* [186]): 1- "Power Wall" - faster computers get very hot, 2- the "Memory Wall" - memory on a Central Processing Unit (CPU) package is limited because of the difficulty of routing too many pins, and 3- the "Instruction Level Parallelism (ILP) Wall" - the deeper the instruction pipeline, the deeper the power hole. In other words, the benefits of the continuously decreasing size of the transistor and Complementary Metal Oxide Semiconductor (CMOS) semiconductor is reaching a point where problems such as power dissipation and leakage outperform the advantages on speed and power consumption. The trade-off between these three areas implies that optimization of one wall will have the opposite effect on the other two walls, therefore computers will stop becoming more efficient.

1.2.1 "Brick Wall"

1.2.1.1 "Moore's Law" and "Dennard Scaling"

For more than 40 years "Moore's Law" has been driving the innovation in the semiconductor world by stating that "the number of transistors on integrated circuits doubles approximately every year" (Moore [177]), which later became two years. This increase in the number of transistors per processor core is illustrated in Fig. 1.1 for the period from 1971 to 2011. Today the two biggest processors are the Xbox One Main System on Chip (SoC), a game station released by Microsoft® in 2013 [253], and the 62-Core Xeon Phi of Intel® released in 2012 in 22nm process [130], with each containing 5 billions of transistors. In addition to the increase in the number of transistors per chip, the increased computational power of the new processors has led to an exponential acceleration of technological achievement (Fig. 1.2). Furthermore, on a technological insight, "Dennard Scaling", which was named after the IBM scientist Robert Dennard, postulates that "Metal Oxide Semiconductor Field-Effect Transistors (MOSFETs) continue to function as voltage-controlled switches while all key figures of merit such as layout density, operating speed, and energy efficiency improve provided geometric dimensions, voltages, and doping concentrations are consistently scaled to maintain the same electric field"(Dennard *et al.* [63]). In other words, the smaller you make the transistors,

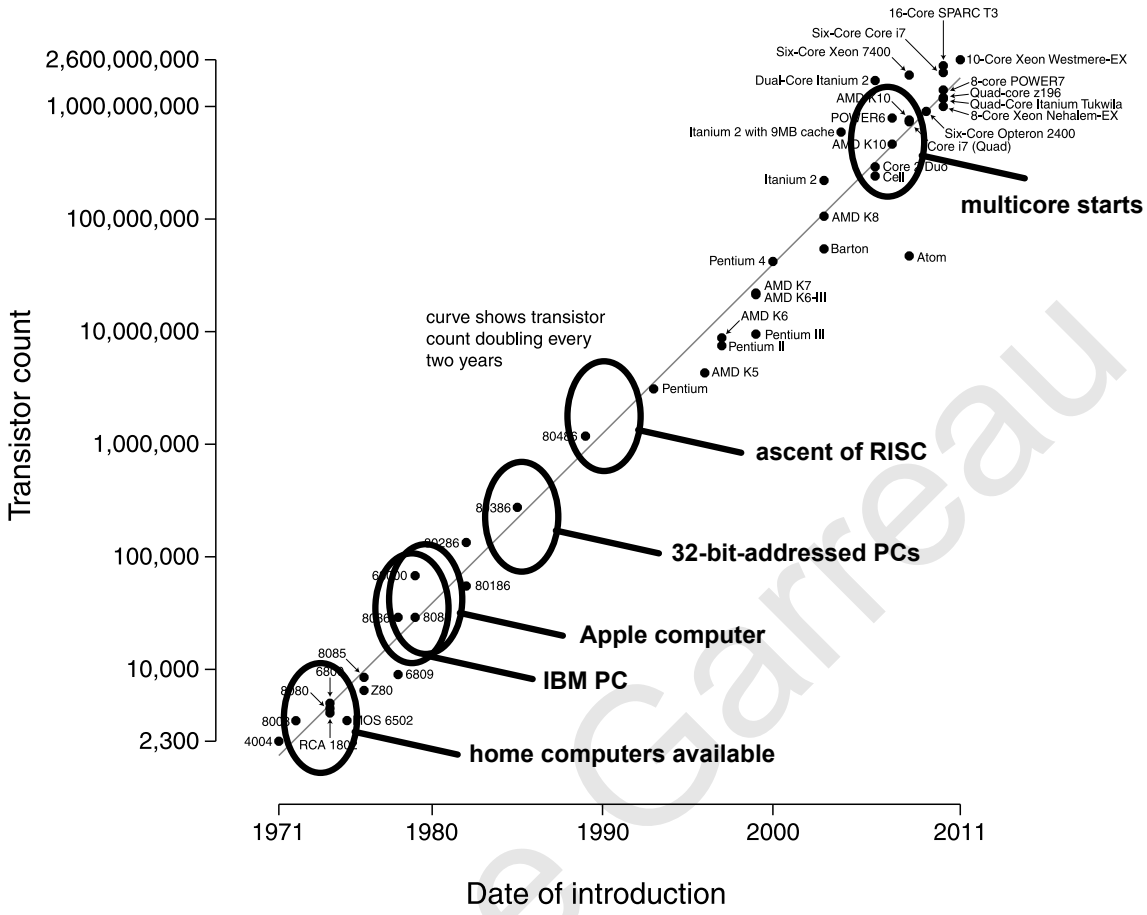


Figure 1.1: Number of transistors per processor core and main innovation milestones (adapted from [130]).

the better they get. As the supply voltage decreases, the power consumption also decreases by the square of the voltage, which means that Dennard predicted that smaller transistors also become faster, cheaper and with less power consumption.

Nowadays, we are approaching the atomic level in the transistors' dimensions and the downscaling is reaching its limits. Issues that were negligible at large scale now have a huge impact on the performance of circuits that are manufactured. Figure 1.3 illustrates this phenomenon: as more and more transistors are placed in commercial microprocessors (exponential increase), the clock speed, the dissipated power and the number of instructions executed per clock cycle have reached a limit and increase much lower than predicted by the "Dennard Scaling".

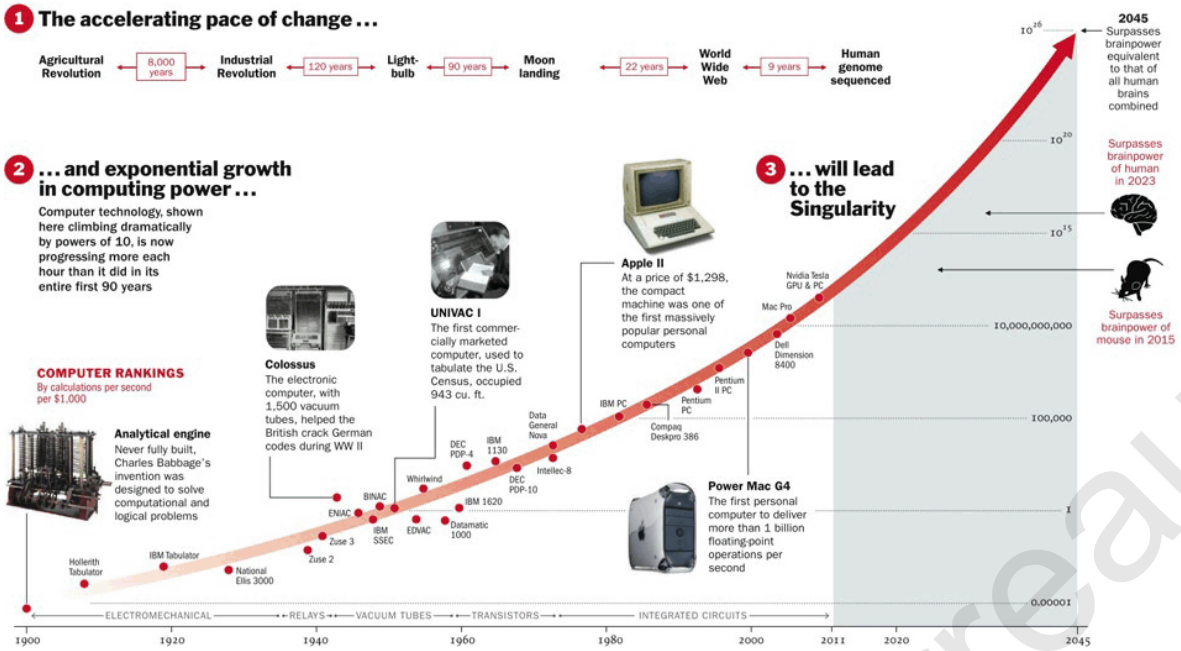


Figure 1.2: Number of transistors per processor core and technological revolution milestones (from [110]).

1.2.1.2 "Power Wall"

The first problem is the energy consumption, or "Power Wall". Energy is the ability to work over a period of time. The power dissipated is divided in two parts, which are the dynamic power and the static power.

The dynamic power (or 'useful' power) P is calculated as $P = \alpha C V_{dd}^2 f$, where α is the activity factor, C the load capacitance, V_{dd} the supply voltage and f the working frequency. To decrease the power, several options are available. Improvements based on a lower α or C are limited by technological processes. Improvements based on the supply voltage V_{dd} have received a lot of attention as it is a squared term in the formula. However as it is decreased, the threshold voltage, voltage necessary to turn the transistor on, is also reduced (now as low as 0.3V), and the closer this voltage gets to zero, the more difficult it gets to turn the transistor completely off. In addition, lower supply voltage means increased current to maintain a constant power level. This translates to the power pins having to supply more current or, the packages must have more power pins (up to 70% in some of them), with the latter being the common option. Increased current may cause voltage droop across the internal power buses and if these droops are high enough, the circuits it connects will stop working. Changing the frequency f is the last option but we are more interested in increasing it rather than lowering it. Unfortunately higher frequency

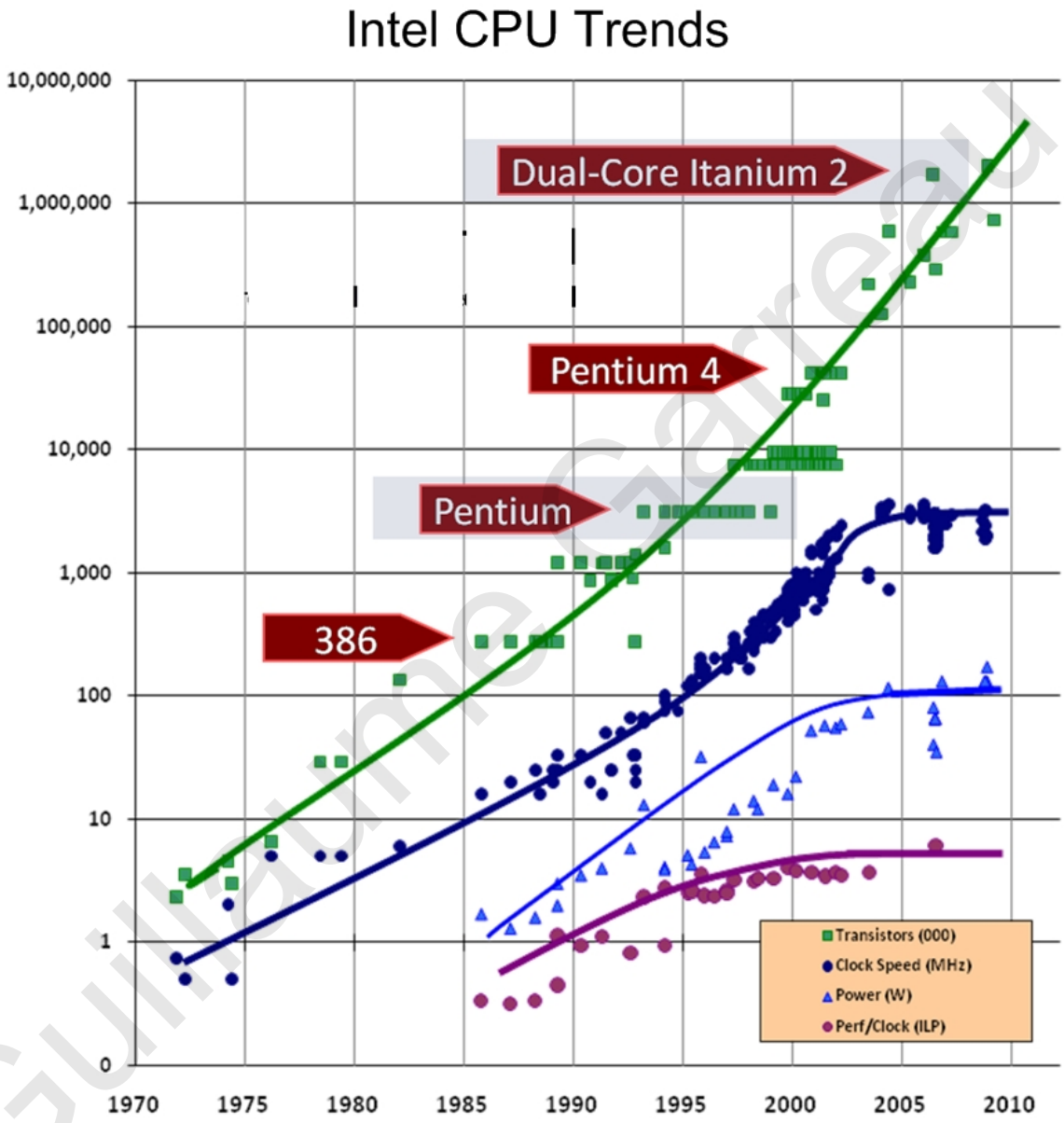


Figure 1.3: History of Intel chip introductions by clock speed and number of transistors (adapted from [222]).

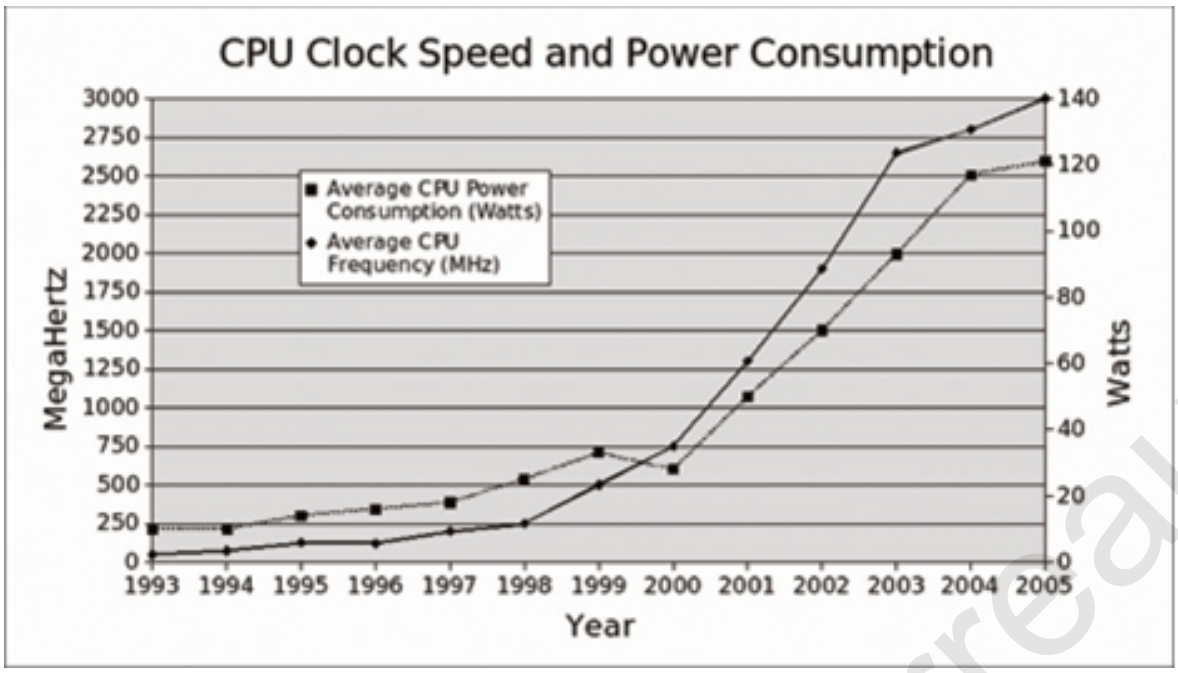


Figure 1.4: Increase of power consumption with frequency (from [71]).

means more heat to dissipate, according to the first Law of Thermodynamics. This is indicated in Fig. 1.4, which plots the average clock speed and heat dissipation of Intel and AMD processors over time and shows that higher frequency requires more power dissipation.

In addition an increasing temperature creates new problems as heat has an impact on both electrical and mechanical characteristics of semiconductor circuits, electrical wiring and packaging. Higher temperature means slower transistors and increased transistor leakage and more power consumed. This causes a vicious circle of increasing heat and power consumption up to breaking point, i.e. it can cause solder connections to melt. Figure 1.5 shows the power density in the CPUs and the importance of cooling them. The Core 2 Extreme QX9775 Yorkfield XE (45nm process) is running at 3.2GHz and consumes 150W of power [130]. If we consider a 1 cm^2 chip, the power density is equivalent to that of a nuclear reactor.

The static power, or 'wasted' power, is the power consumed when the circuit is powered on but not clocking. This power mainly depends on the drain-source's leakage and the insulation layer's leakage (Rabaey *et al.* [190]). The insulation layer is the layer that separates the gate, the electrical connection that controls the Metal Oxide Semiconductor (MOS) switch, from the rest of the transistor. As the layer gets thinner, the performance of the transistors improves, until it becomes so thin that it starts to leak electrons. As gate leakage became an increasing problem, the

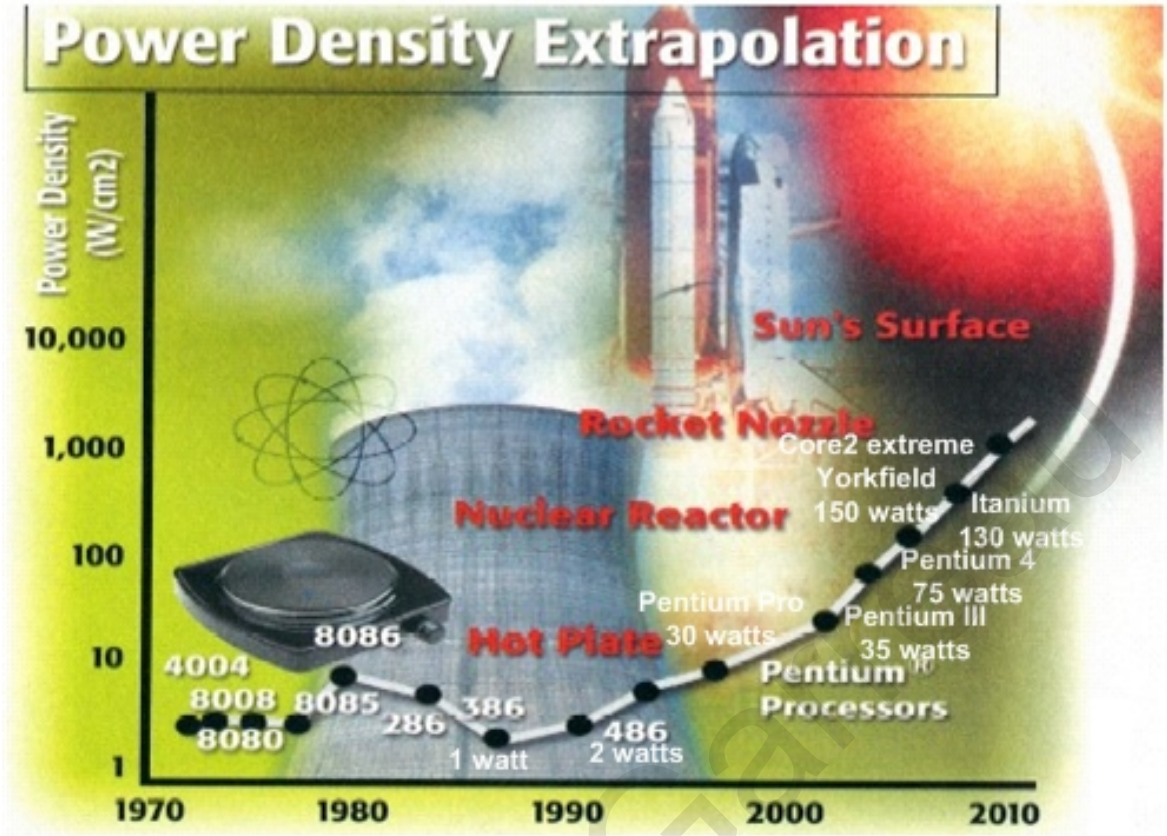


Figure 1.5: Power density trend (adapted from [130]).

silicon dioxide, which was historically used as the insulator, was replaced by other materials. These materials are known as high-k: they are characterised by high dielectric constant and are thus less likely to leak. To get an idea of how important this problem is, current multi-core microprocessor transistors leak 50W of power while in idle mode, with power leakage (Chen *et al.* [49]). Even reaching 100W in the latest multicore Xeon! [130] This static power is wasted and as power is neither infinite nor free, a lot of work is done to minimize it.

All these problems mean that the power consumption improvement has reached a limit and if we want the technology to keep improving we need to find innovative solutions.

1.2.1.3 "ILP Wall"

In first computers programming was simple and instructions were executed serially one after the other. In the early 60s, engineers at Control Data Corporation CDC (Seymour Cray, Dean Roush and Jim Thomton) started to use multiple instructions computers to speed up execution. The Instruction Level Parallelism (ILP), which is the execution of multiple instructions or pieces of instructions at the same time,

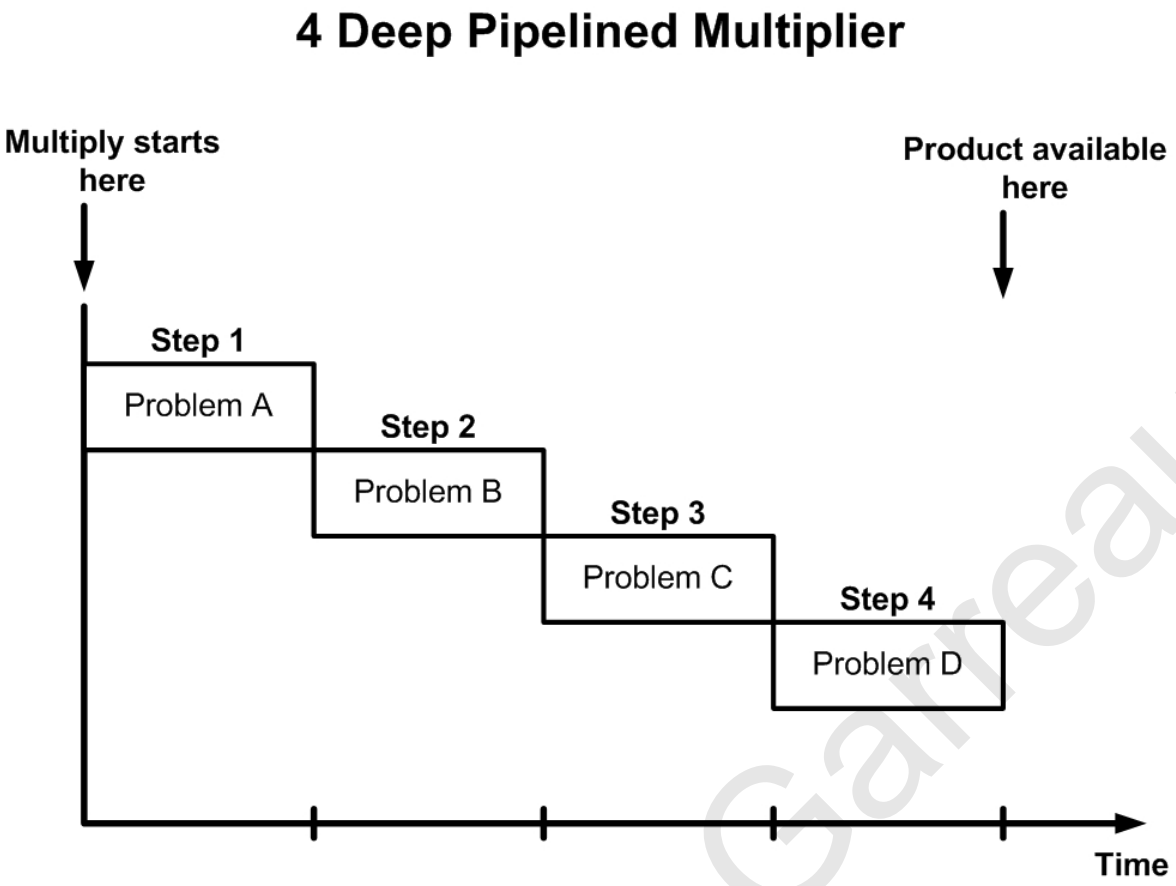


Figure 1.6: Four steps pipelined multiplier (adapted from [88]).

was a great innovation at that time and allowed to speed up computations of the CDC-6600 computer more than ten times compared to their fastest competitors [19]. But the benefits come at a cost of complexity in logics, more transistors, more heat dissipated and harder programming compared with the scalar architectures. The greater the parallelism, the bigger the difficulties in each of these areas. The ILP process works as follows. First, an execution pipeline breaks each instruction in a series of steps. Second, the steps of many instructions are executed simultaneously such that many instructions can be performed in parallel. The more steps, the more instructions executed at the same time and thus the faster the computer can run. For example, an operation that would require n clock cycles to complete, could be broken into n steps. Then n operations can be executed at the same time, producing the first result after n clock cycles and then one result every clock cycles. Figure 1.6 gives an example of a four steps pipelined multiplier.

General purpose processors have to perform many different tasks. In order for the pipeline to accelerate those tasks, all the instructions must be broken into the same number of steps. For that reason David Patterson created the Reduced Instruction

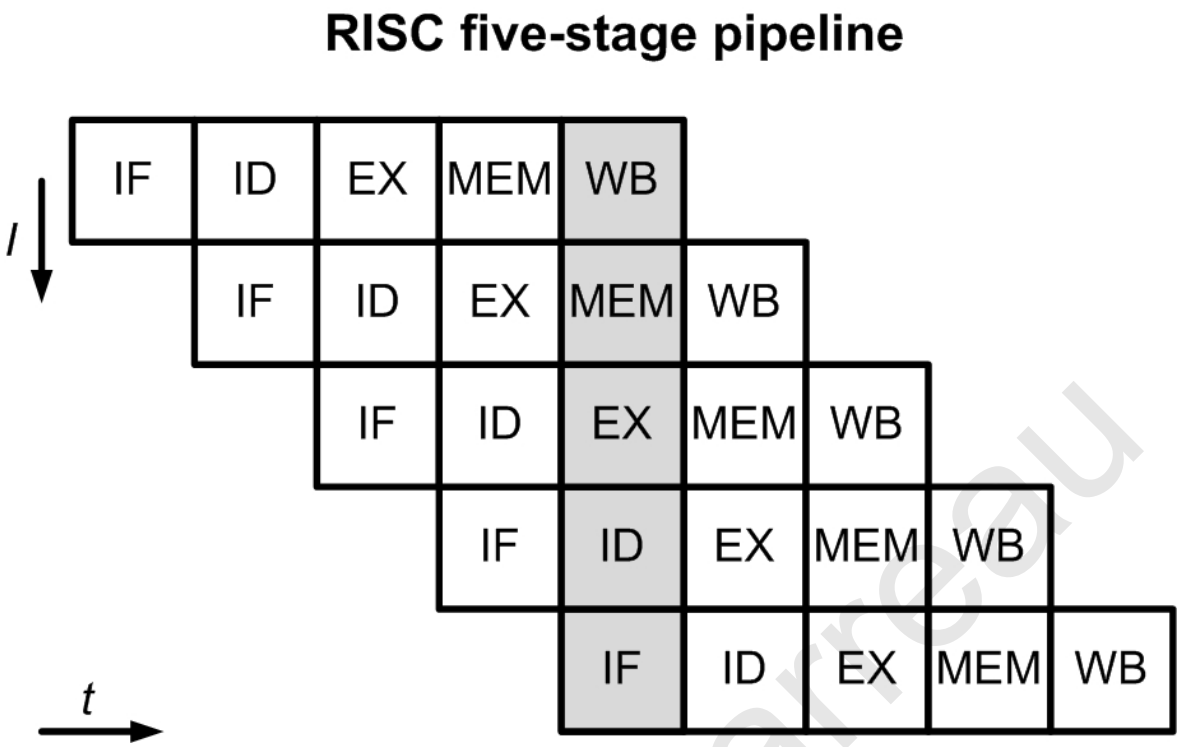


Figure 1.7: RISC five-stage pipeline (adapted from [88]).

Set Computer (RISC), which greatly simplified the pipeline process compared to the variable-length instructions of the Complex Instruction Set Computer (CISC). Theoretically, a five stage pipeline RISC CPU is five times faster than a non-pipelined CPU, (Fig. 1.7), as some instructions may be inter-dependent and require the result of another operation for their execution. There are also branches and loops that make the problem more challenging.

To solve the synchronization problem, several solutions have been proposed. A simple instruction decoder that can identify the problem and slow down one instruction in order to wait for the result but this will slow down the execution time of the program. A more complex instruction decoder can detect the problem and execute during the delay of other instructions' steps. This is called "multi-threading" or "hyper-threading". However, this adds huge complexity to the architecture, requires thousands of transistors and increases power dissipation. Another solution can be achieved through the use of registers: a kind of memory located on the CPU and much faster than ordinary memory. Figure 1.8 shows the speed and latency of access for the different types of memory.

Compilers were designed to use the registers as much as possible to avoid "normal" memory access that would slow down program execution. However, registers

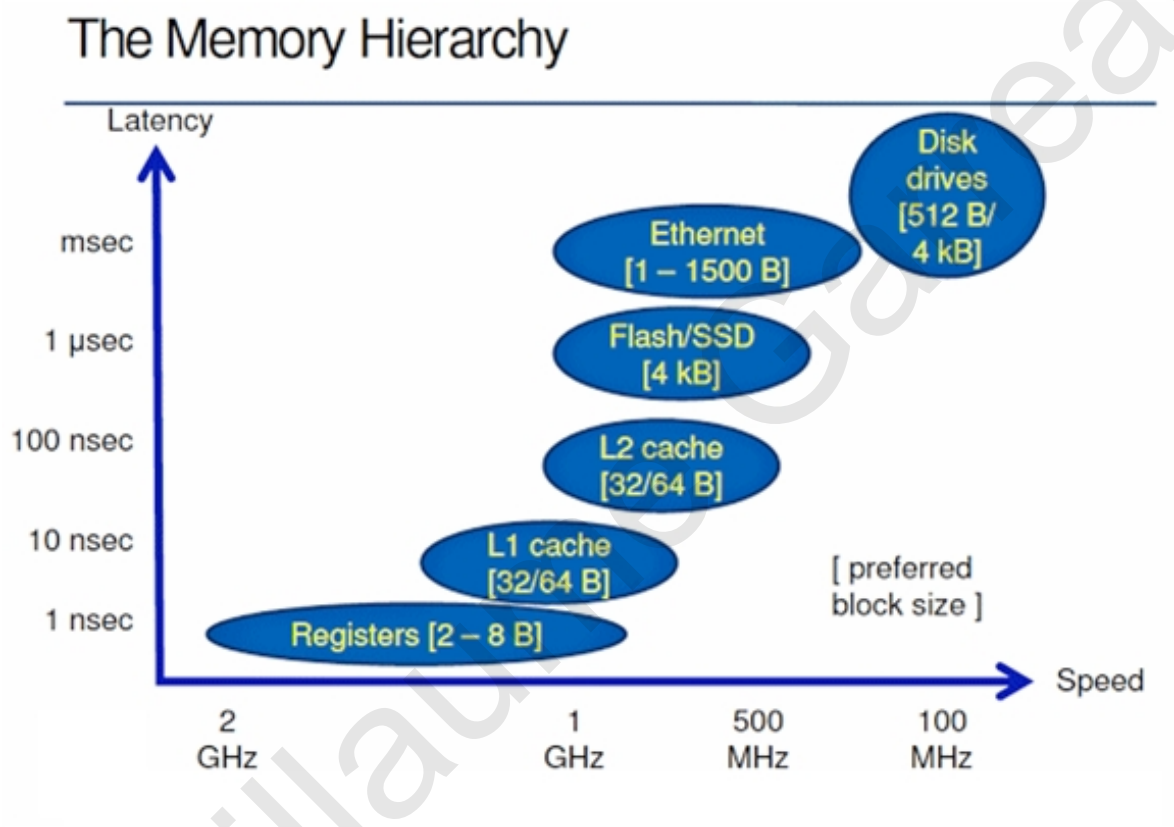


Figure 1.8: Comparison of speed of access of the different types of memory (adapted from [242]).

cost instructions bits to be selected. Five opcode bits are necessary for selection of one through 32 registers. Thus if one needs to specify two operands and a destination, this is already 15 bits used just for the selection of the registers. All the added architectural complexity and logical tricks to optimize the instruction cycle executions have reached the "ILP Wall" due to the induced logical complexity and power consumption. At some point execution speeds actually start to reduce.

1.2.1.4 "Memory Wall"

Another great revolution that started in 2004 was the shift to multi-core scaling to continue the performance growth required by Moore's Law. The idea, as the Instruction Level Parallelism, may seem obvious: instead of increasing the complexity of the CPU on one chip (leading to the the problems presented above), why not just have multiple simple cores on the same chip or a multi-core chip. In a multi-core processor chip, two or more complete microprocessors are built on the same chip and attached to a shared memory bus. However, the positive increase in performance achieved by a multi-core processor is not scalable, i.e. doubling the number of cores does not double the performance increase, which even decreases after some point, as shown in Fig. 1.9. Sandia Labs reported that as the number of cores increased, the processor power increased at substantially less than linear improvement and then decreased at an exponential rate (Murphy *et al.* [180]).

This is mainly due to the lack of memory bandwidth and the contention between processors over the memory bus (also known as the "Memory Wall", Patterson *et al.* [186]) and the fact that software are not yet ready to use the available multi-cores. The one task that multi-cores can perform very efficiently today is the "embarrassingly parallel" problem or "Recognition, Mining, and Synthesis". This task is similar to searching for a phone number in the phone-book. It could take up to several hours to find a phone number if you perform the search on your own. However, if the search is split into smaller tasks and is performed by more people, then it could be mere seconds before the number is found. This technique is also called MapReduce, and it is the same method that runs Google's million server network search engine. Such data mining across enormous datasets seemed to be perfect for the multiple CPU system, until the "Memory Wall" appeared again. One solution to the problem was computer cache. Similarly to the registers that were described above, caches are small dedicated local memories that sit between the CPU core and the main memory.

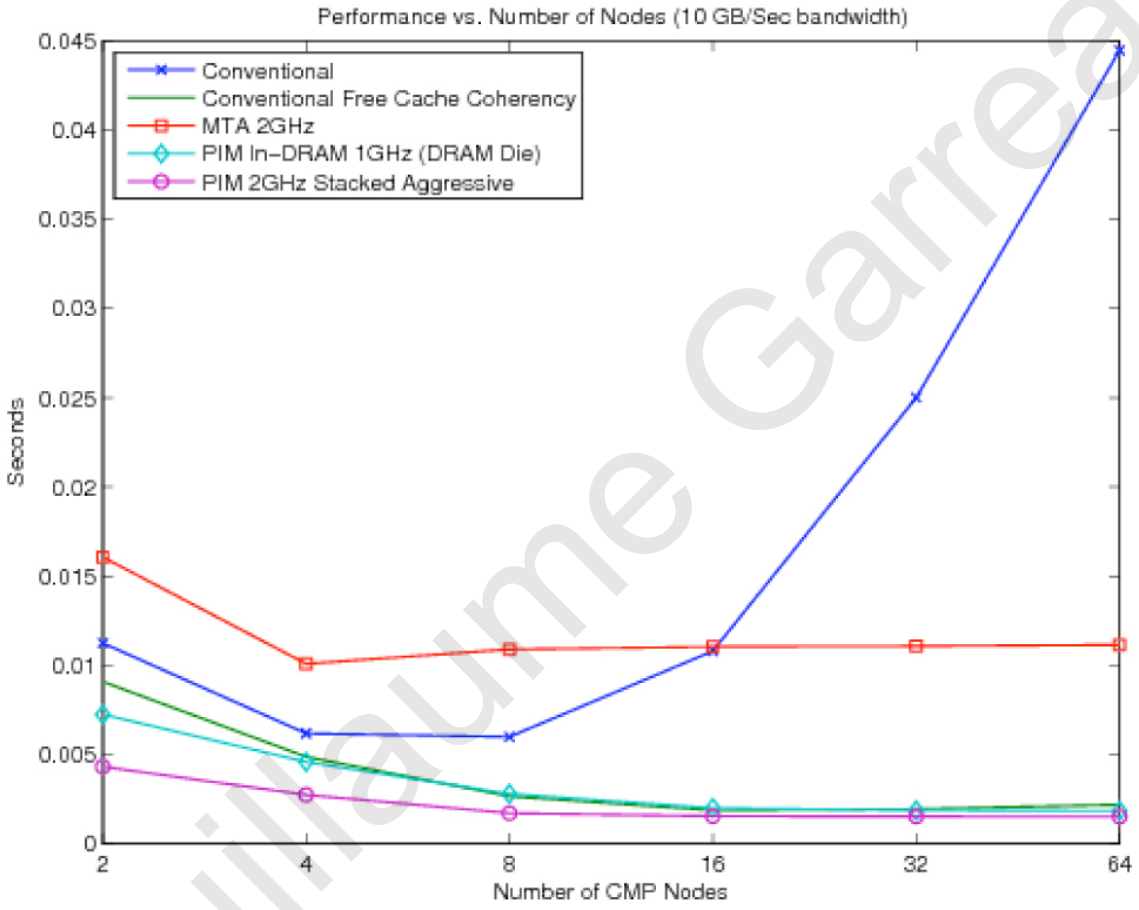


Figure 1.9: Performance versus the number of nodes in the multicore (from [180]).

Caches take advantage of the fact that instructions may need to reuse the same data, thus the core can simply access the data that is already present in the cache, without the need to access the main memory, which is up to 100 times slower than cache access (Fig. 1.8). Going back to the phone number search example, if you are looking for another number, you do not need a new phone-book or to give new pages to your friends, but you can simply give the new name to look for. The problem is that the dataset is exponentially increasing over time and caches already occupy over half of the silicon area of some CPUs and consume much of the power (Huang *et al.* [124]). The cache size being an issue, the solution is to increase the memory bus bandwidth to get data in and out faster. Three ways are possible: increase the memory transfer speed, increase the memory transfer size and finally, move the data closer to the CPU. However, each of those solutions is limited by the three "Walls" encountered before. The memory transfer speed, the clocking or the CPU frequency, is limited by the "Power Wall", as faster means more heat generated. In 2007, Intel designed some processors using high-speed technology called Rambus [72], which was faster but much more expensive than other technologies. The memory transfer size is linked to the "ILP Wall" and even though it has increased over the years, it is still limited to 64 bits in most of the new CPUs, while many CPUs are still 32 bits. Transfer size is also limited by the "Power Wall", as an increased memory bus width implies more pins that change state (charge/discharge) at each clock cycle and thus more power is consumed. The last option, which is to get the memory closer to the CPU to limit the transfer and thus substantially limit the "Memory Wall", will be discussed in more detail later together with other innovative solutions already used by mature technologies. Instead of moving the memory to the CPU, the CPU is placed onto the memory.

1.2.1.5 "Dark silicon" or "Utilisation Wall"

The "Utilisation Wall" or "dark silicon" is a new challenge that appeared with the rise of the multi-core CPU. It refers to the portion of the chip that has to be powered off during runtime and it is linked to the "Power Wall". This is the result of the increase of the number of cores and transistors, while keeping a constant power budget. Esmaeilzadeh *et al.* created three models to project the performances of future multi-cores and evaluate the size of the dark silicon (Fig. 1.10) [80]. They predict that "in just five [technological] generations, at 8nm, the percentage of dark

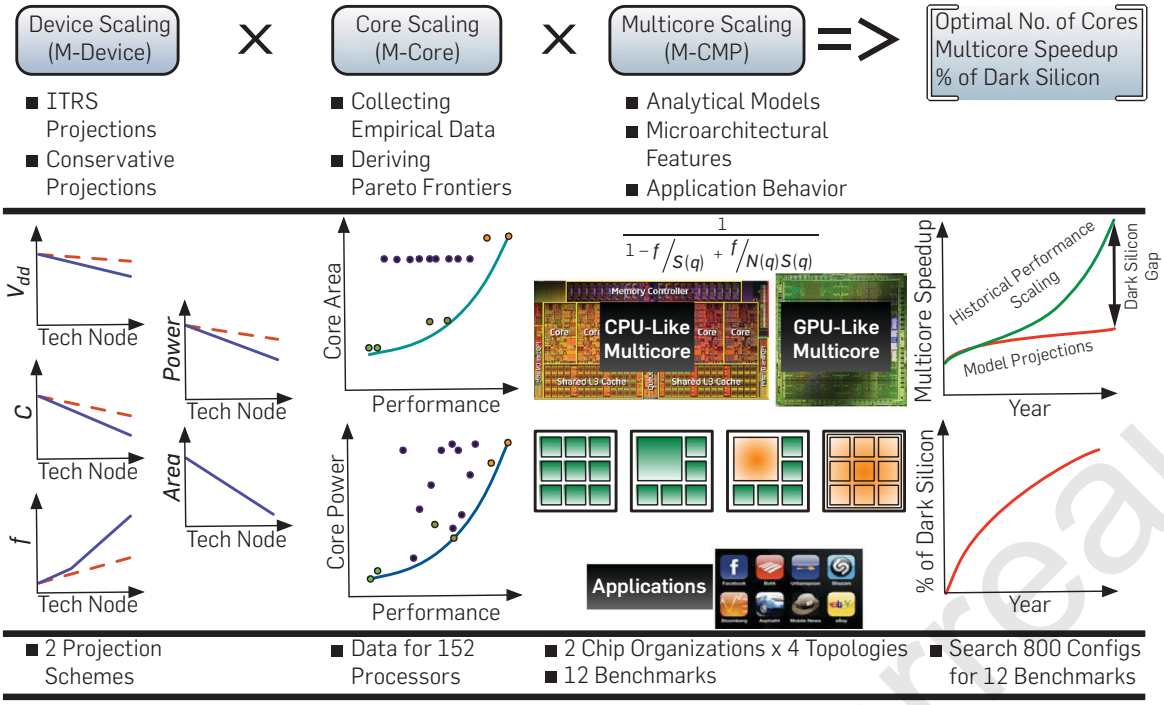


Figure 1.10: Overview of the methodology and the models to project the performances of future multi-cores (from [80]).

silicon in a fixed-size chip may grow to 50%."

The work of Esmaeilzadeh and colleagues was criticised in a study by Ganssle [95], whereby the author points out that Esmaeilzadeh *et al.* have not taken into consideration memory related issues (such as contention, locks, and interprocessor communication) and that prediction should be even worse in terms of performance, especially as the models are used on applications where 75% to 99% of the work can be done in parallel, which is again very optimistic. However, the study by Ganssle ends with the optimistic belief that "some cool and unexpected inventions will continue to drive computer performance on its historical upward trajectory."

1.2.2 Solutions to the "Brick Wall"

Recent years have seen numerous solutions to the "Brick Wall" problem appear, some of which introduce totally new technologies while others use well known technologies in an innovative and ingenuous way.

In the list of the 'new' technologies, two are particularly innovative and challenging: the quantum computer and the bio-integration.

1.2.2.1 Quantum computer

The quantum computer is a computer based on the principles of quantum-mechanics phenomena, such as superposition and entanglement, and that can perform operations on data. Whereas classical electronics and CPU require data to be encoded into binary digits (bits), quantum computation uses quantum properties to represent data and perform operations on these data. This quantum property is called quantum bit or qubit. Commercial applications in cryptology have been released by SwissQuantum, Id Quantique or MagiQ Technologies and large-scale quantum computers are predicted to be able to solve specific problems tremendously faster than any classical computer using the best currently known algorithms (Simon [213]), such as integer factorization using Shor's algorithm or the experimental determination of the Ramsey numbers (Bian *et al.* [16]). Quantum computing technology is still at an early stage, the computers are very large, extremely expensive and require a tremendous amount of power (Hughes and Boshier [127]). In May 2013 Google announced a joint project with NASA for building a $15 million quantum computer, the D-Wave Two, which will be housed in a box the size of a garden shed while cooling the computer's quantum chip to temperatures approaching absolute zero [114].

1.2.2.2 DNA computing

The second technology is bio-integration or Deoxyribose Nucleic Acid (DNA) computing. It is a form of computing which uses DNA, biochemistry and molecular biology, instead of the traditional silicon-based computer technologies. DNA computing, or, more generally, bio-molecular computing, is a fast developing interdisciplinary area (Rotman [196]).

Another way to solve the "Brick Wall" is, as inspired by Proust's quote, "not in seeking new landscapes, but in having new eyes." Instead of ground-breaking and new technologies, try to see how to use technologies that we already master in innovative ways.

1.2.2.3 Dynamic scaling

In order to reduce the power consumption, one innovation was to break the CPU chip in subparts working at different voltage levels or even to have them powered off.

Given that power is proportional to voltage squared, it is easy to see how powering down or switching off the processors that can run slower or are not being used can reduce the power consumption. This is called "dynamic voltage scaling". But this trick has a limited effect now that up to 2/3 of the chip is used by the cache in many microprocessors, as the cache would lose its data if powered down.

A similar effect can be achieved with frequency decrease. It is named "dynamic frequency scaling". However, the frequency decrease induces only a linear decrease in power dissipated. It is less helpful today as with state of the art processes, static power is more important than dynamic power consumed by the microprocessor.

1.2.2.4 CPU on DRAM

As mentioned previously, in order to break the "Memory Wall" one needs to increase the memory bandwidth. One way of achieving this is to bring the memory and CPU closer. We have shown the limitations of moving the memory, cache, closer to the CPU. Several have tried and have actually demonstrated good results, but the cost of the solution was so high than none of was commercialized [87]. Figure 1.11 gives an idea of the cost. The quick explanation is that memory is manufactured with three metal layers process, which keeps the cost of manufacture low; this is in contrast to the CPU, which is manufactured with up to 12 layers and, hence, is expensive (Fig. 1.11).

The other way is to bring the CPU onto the memory. The solution, proposed by Venray engineers, was to simplify the CPU such that it could be manufactured with a three metal layers process and then added to a common Dynamic Random Access Memory (DRAM). The result, the Thread-Optimized Multiprocessor Instruction (TOMI) Borealis chip [158] is a 2.1GHz 98mW 32 bits microprocessor core (excluding the cache) "implemented in 22,000 transistors, including multiplier and barrel shifter. The virtual memory controller adds another 3,500 transistors." 16 of those chips are configured to fit on a 4inch circuit board. The board includes 128 cores, 2Gbyte DRAM, network controller, and a switching power supply. Four rows of 32 of these boards were arranged on a 19inch motherboard and outran an entire 19inch rack of Intel Xeon E5620 multicores running a MapReduce algorithm on 256GByte of dataset [87]. In addition, DRAM transistors are low leakage by design, as leakage would mean loss of information. The capacitances of the buses are very small and the process produces very cheap transistors. This is good news not only for the

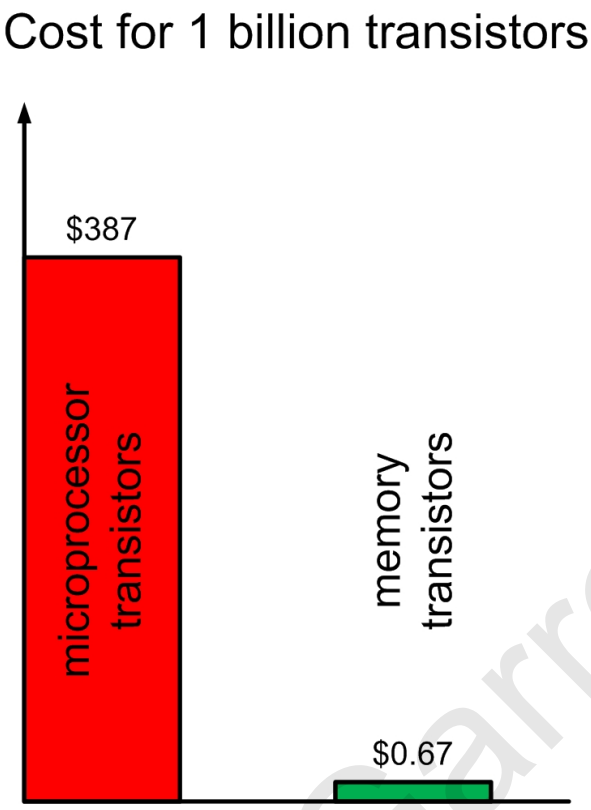


Figure 1.11: Cost of 1 billion transistors built with microprocessor or memory technological process (adapted from [87]).

"Memory Wall", but for the "Power Wall" as well. However, this comes at the cost of slower execution, as DRAM transistors are about 20% slower, and a sensitivity to high current spikes as they are mostly analog devices. Figure 1.12 shows the trade-off between speed and memory capacity for the different types of memory.

1.2.2.5 ASIC, FPGA and GPU

Another optimization was to use a specialized microprocessor, which is extremely efficient and fast on a specific task. A Graphic Processing Unit (GPU) is a specialized processor designed to operate and alter memory to accelerate the creation of images. GPUs are very efficient at manipulating computer graphics and their highly parallel structure makes them more effective than general-purpose CPUs for algorithms where processing of large blocks of data is done in parallel. GPUs are specialized Digital Signal Processors (DSPs). Even more specialized and efficient are the Application Specific Integrated Circuits (ASICs), but their development cost is much higher. As a trade-off between efficiency and specialization, the Field Programmable Gate Arrays (FPGAs), are commonly used. FPGAs have the advantage

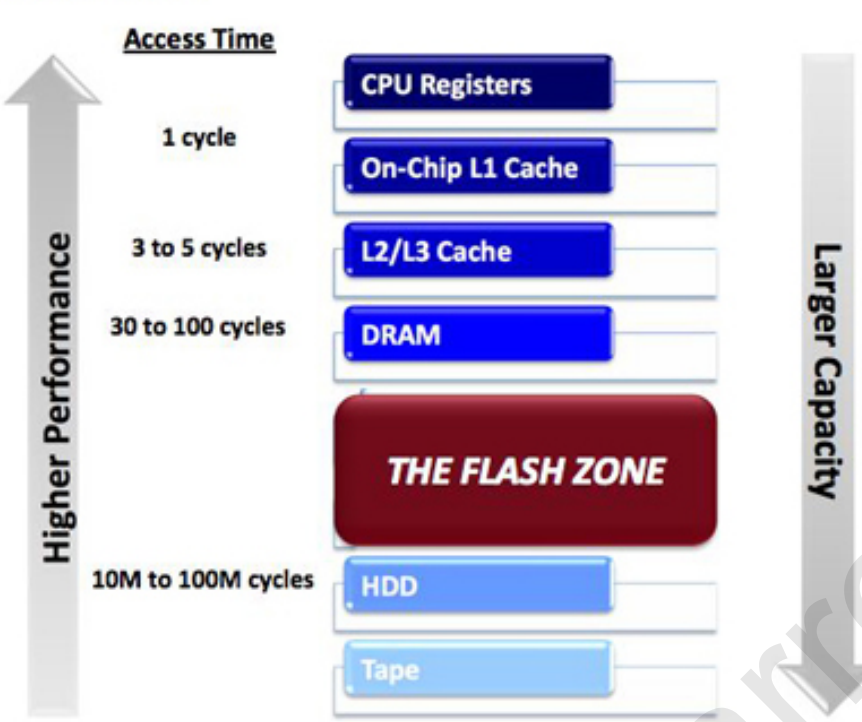


Figure 1.12: Comparison of the characteristics of the different types of memory (adapted from [2]).

of being more efficient than CPU and GPU, with lower development cost than ASIC, as well as being reprogrammable. Figure 1.13 show the trade-off between flexibility and computational efficiency of the different types of processors.

On Fig. 1.14, four parameters are used to compare ASIC, FPGA, Application-Specific Instruction-set Processor (ASIP)/DSP and generic processors. It shows that a flexible system with a quick and low cost development comes at the cost of lower performance.

1.2.2.6 3D chip

A 3D chip is a chip in which two or more layers of active electronic components are integrated both vertically and horizontally into a single circuit. This offers benefits such as: shorter wiring length, higher bandwidth, smaller footprint, heterogeneous integration, lower power and new design possibilities. However some challenges are created: more difficult testing of parts, repairing defects in the inner layers and heat dissipation (Emma and Kursun [78]).

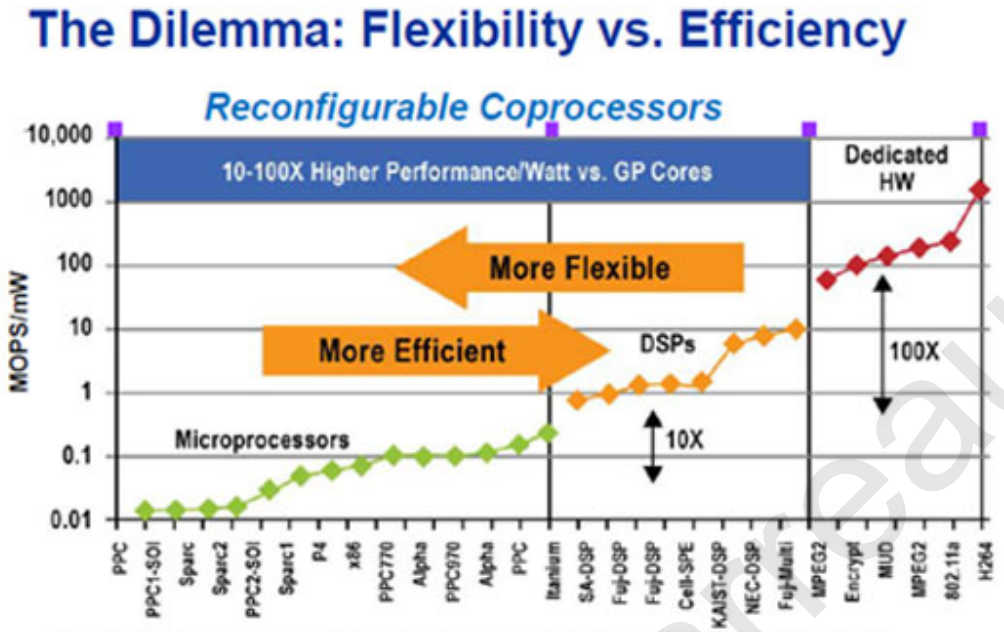


Figure 1.13: Comparison of flexibility versus efficiency of the different types of processors (adapted from [142]).

Technology	Performance/ Cost	Time until running	Time to high performance	Time to change code functionality
ASIC	Very High	Very Long	Very Long	Impossible
FPGA	Medium	Medium	Long	Medium
ASIP/ DSP	High	Long	Long	Long
Generic	Low-Medium	Very Short	Not Attainable	Very Short

Figure 1.14: Comparison of flexibility versus speed of development and cost of the different technologies (from [225]).

1.2.2.7 Carbon nanotube-based Integrated Circuit (IC)

Another innovation is carbon nanotube-based IC. In late 2012, IBM claimed placing more than 10,000 working nanotube transistors on a single device using standard semiconductor processes [234]. Carbon nanotubes are single atomic sheets of carbon rolled up into a tube. The carbon nanotube forms the core of a transistor device that could work in a fashion similar to the current silicon transistor, but with better performance. In September 2013, Shulaker *et al.* reported the first carbon nanotube computer [211]. The computer is rudimentary by modern standards: it contains just 178 carbon nanotube-based transistors. It operates on only 1 bit of information. And it clocks in at just 1 kHz.

1.2.2.8 Neuromorphic computing

The last innovation that is of particular interest to this thesis is the neuromorphic computing [26,43,175]. In general, neuromorphic engineering is the use of a Very-Large-Scale Integration system (VLSI) and an electronic analog circuit to create bioinspired systems that mimic biological neural networks and nervous systems [43]. This field comprises of the development of analog, digital, and mixed-mode analog/digital VLSI and software systems that implement models of neural systems for perception, motor control (Ijspeert [128]), or sensory integration (Liu and Delbruck [154]). An important challenge of neuromorphic computing is to understand how the morphology of individual neurons and group of neurons work in biological nervous systems to represent information and perform computations with robustness to interferences and damages. How do these networks create learning and memory? How can they adapt to change (brain plasticity)? Neuromorphic engineering is an interdisciplinary area that involves fields such as biology, electrical engineering, computer science, physics and mathematics in order to understand natural neural systems and design artificial neural systems that can perform auditory, visual, recognition and learning tasks. The neuromorphic approach has several advantages, particularly considering the problem of the "Brick Wall".

Firstly, parallelism is an inherent characteristic of biological neural systems. Complex tasks, such as image recognition, are performed in a few 100ms by the brain, due to specialized areas working in parallel.

Secondly, the power efficiency of our brains is tremendous compared with mul-

ticore processors or supercomputers. The brain is using as much energy as a 10W light bulb and with its 100 billion neurons and more than 100 trillions synapses can perform about 10kPFLOPS (peta is 10^{15} FLoating-point Operations Per Seconds) [5,148,212] and it is believed to store a rough estimate of 10TByte of memory. All of it fits in about $1200cm^3$ and weights about 1.4kg. The study of spiking axons by Goldberg *et al.* showed that neural communication was developed in a way that maximizes energy efficiency rather than information rate [106]. As of March 2014, the most powerful supercomputer is the Chinese Tianhe-2, which uses 17.8MW of power (plus 24MW for cooling) to reach 33.86PFLOPS [171]. It uses about 1,408TiB of memory, costs about $400million and is the size of a basketball field.

Thirdly, the advantageous computational complexity versus cost of neural network is discussed in [4, 109, 183] and shows a great improvement compared with classical computing solutions. It is still a long way before developing a computer of the computational capabilities of the human brain for the same power and size budget, though similar computational capabilities are expected to be reached the by 2025 (Fig. 1.15).

Fourthly, in neuron networks, there is no dedicated memory space, memory and processing are fused, the neurons doing the calculations and the synapses' weight the 'memory' (Engelbrecht *et al.* [79]). Learning as well can be done using neurons (Brader *et al.* [22]).

Finally in neural systems, preprocessing is delocalized such that not all data is transmitted to the 'end user'. For example in the visual system, preprocessing (filtering) is done at the retina level, and then at the optical nerve level before reaching the visual processing areas (Liu and Delbruck [154]). This has inspired work such as the analog VLSI neuromorphic image acquisition system with pre-processing described in [4]. A methodology for engineering bioinspired hardware systems is proposed in [4], with this work leading to the development of various hardware neural arrays [37, 39] and the first million neuron system design on a single FPGA (Cassidy *et al.* [38]).

Furthermore, neuromorphic design can be use with other technologies such as 3D chip for even greater performances. An example of bioinspired system using 3D Silicon On Insulator (SOI) CMOS is reported by Marwick and Andreou [167]. They reported an increased connectivity of the basic cells and a better energy efficiency compared with a traditional 2D CMOS solution.

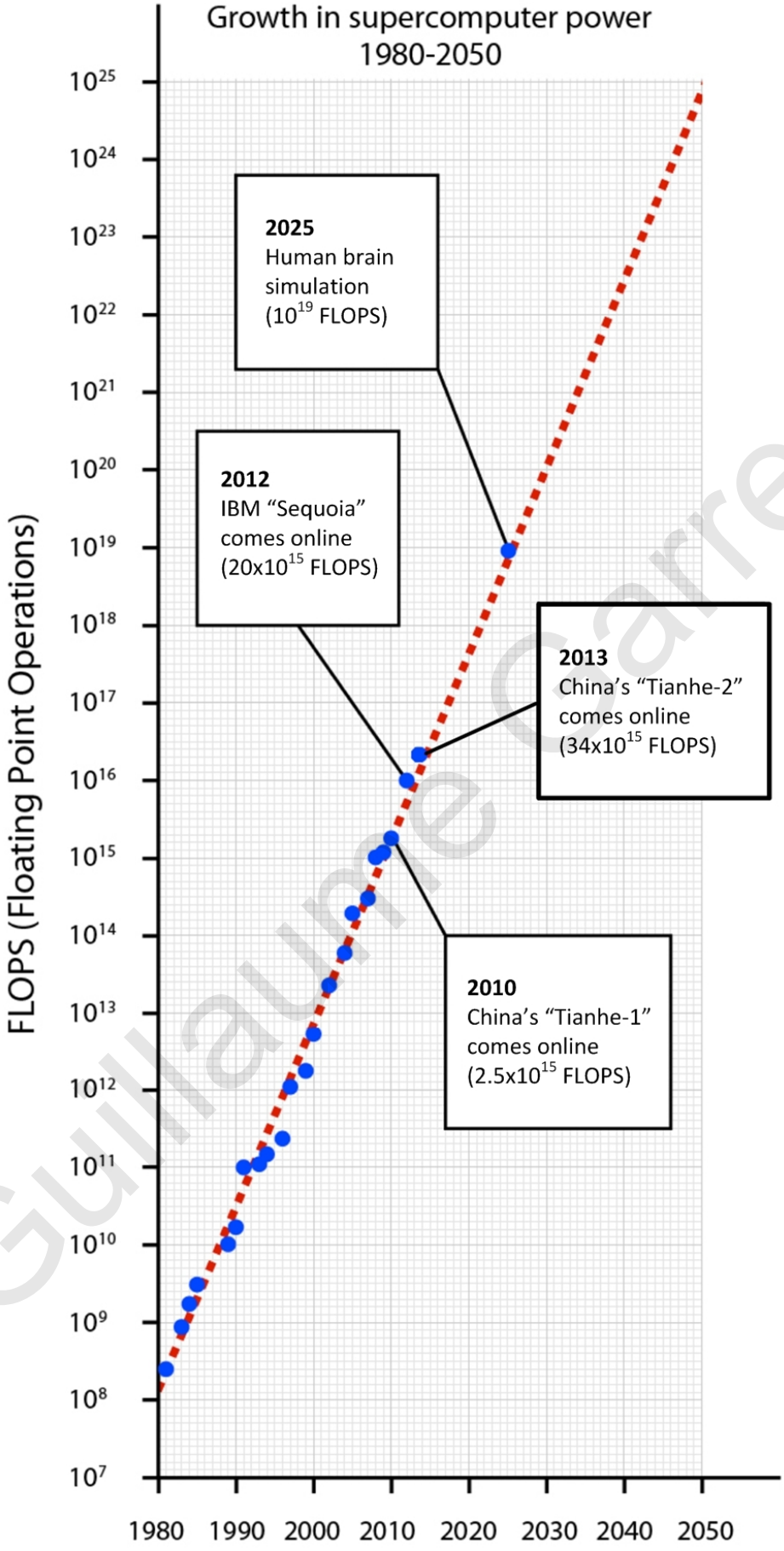


Figure 1.15: Growth in supercomputer power since 1980 (adapted from [89]).

Today large amounts of money are spent in order to get a deeper understanding of the human brain and be able to design artificial systems with comparable capabilities. The most important projects currently running are the Systems of Neuromorphic Adaptive Plastic Scalable Electronics (SyNAPSE) project of IBM that started a few years ago, the human Brain project where the EU invested €1billion over 10 years, and more recently the Brain Research through Advancing Innovative Neurotechnologies (BRAIN) Initiative with an investment of $100million by the American government.

1.2.3 Conclusion

After 40 years of innovation to keep up with Moore's Law and multiple breakthroughs in technological processes (CMOS), hardware (multi-core, GPU, FPGA, ASIC) and software (RISC, multithreading), classical electronics based on transistors have reached a "Brick Wall" that predicts the end of a significant increase in the performance of computers, unless ground-breaking innovations are made. Some researchers have been working on new and totally different technologies and solutions as quantum computing and bio-integration or DNA computing. Others have tried to used existing technologies in new ways, or combined different technologies. One of them is to take inspiration from Nature and biological neural systems.

In Chapter 2 a novel neuromorphic-based instrumentation system for acoustic data acquisition, processing and classification is presented.

Chapter 2

Instrumentation for Acoustic Data Acquisition

2.1 Introduction

The human auditory system processes sounds at almost every level of the auditory pathway in order to enable perception by the brain at a later stage. However, it is not yet perfectly known where and how perception occurs in the brain. Of particular interest here, the role of the auditory system in perception is not clear. Some colleagues at the Magyar Tudományos Akadémia Pszichológiai Kutatóintézet (MTAPI, Budapest, Hungary) wanted to develop psycho-physiological experiments to help understand how perceptual organization of audio stimuli works. However, in order to achieve these experiments, they first needed to collect audio-visual data of multiple mobile and immobile sources of sound. This would later allow to reconstruct an artificial environment based on real-life data and study perceptual construction of acoustic objects in the human brain, and investigate how audio-visual motion cues promote segregation of auditory streams [18, 208].

It has been shown that timing is extremely important for acoustic scene analysis by humans (Shinn-Cunningham [209]). The time can be obtained as time of arrival or, in the case of various sources, as interaural time difference or delay. Interaural time difference also provides information regarding the location of the source(s) (Kandel *et al.* [136]). The human auditory system is sensitive to interaural delays as small as 10μs and displays an accuracy of location of 1 degree to the front and 15 degrees to the side [136, 240]. Animals, just as humans, also rely on interaural time difference for auditory analysis, but it is more important for those that use

echolocation and ultrasounds as a means of communication or prey location, such as bats and dolphins (Moss and Surlykke [178]). Others, such as sand scorpions, measure the vibrations in the ground and use the seismic sounds to locate a prey, using the difference of time of arrival of the sound wave between different legs to evaluate the direction and, at short distances, the exact location of the source of the wave (Brownell and Farley [30]). In addition the human hearing system as a remarkable dynamic range of over 130dB (from threshold of hearing to the threshold of pain, Leather *et al.* [144]). It is capable to discern anything from a quiet murmur to the loudest sounds of jet engine starting up.

As shown by the examples given above, the timing parameter is crucial for the correct behavior of the auditory perceptual system.

When collecting distributed multimodal acoustic data (vibrational, ultrasonic and audible sounds in addition to video stream to establish the ground truth) an estimated 12MB/s of data is generated and processed in realtime. This would not be possible if we were to use single PC or other pre-existing methods, whilst maintaining μsec synchronization. Hence it was decided to develop a hardware solution to fulfil these requirements.

2.2 State of the art acquisition systems

2.2.1 Multi modal data acquisition

In the data collection scenario introduced above, the same single event will be captured by multiple sensors (same mode and/or different modes). In order to be able to identify that the event is the same, in the dataset of each sensor, the time delay between each sensor has to be known. This is called relative time of arrival. One way to do that is to embed time markers in the data, so that the processing computer can time align (*i.e.* synchronize) waveforms obtained from distinct data acquisition units. One way to avoid the tagging would be to connect all the sensors in a given subgroup to the same data acquisition unit. However, this requires a small number of units of data converters (analog-to-digital ADC or digital-to-analog DAC) in locksteps, and located near one another. When a single computer or a standalone device is used for data acquisition, measurements rely on the device's own internal clock, which is typically driven by a crystal oscillator. This crystal

shows a frequency stability of 50ppm in general, which means that a 40kHz sine wave may stretch or shrink by two periods over one second. It is, thus, necessary to tag the data collected in a way that it can be processed coherently. For instance, an ultrasonic transmitter unit and an ultrasonic receiver unit must be synchronized so that Doppler information can be extracted.

Furthermore, we are interested in higher hierarchical levels for the data processing. This means that data from different subgroups is correlated to create a coherent picture of the overall scene; for example, the motion of an individual's foot detected by ultrasound Doppler is correlated with the sound of the footsteps localized by the microphone array. As the complexity of the different scenarios increases, more sensors may be required, reaching the point where too many sensors are necessary for a single data acquisition unit, either because the data acquisition unit becomes too complex to design and build, or because the bandwidth required to send the data to the processing computer becomes prohibitively large. In that case the acquisition system would have data tagged with timestamps linked to different crystals, which means that little by little, they would start to drift away from each other and then the timestamps would be meaningless when re-aligning the data. This means we need one single crystal to serve as the reference to all nodes. If the crystal frequency drift all the timestamps values will drifted of the same amount and thus the relative time delays will stay correct when realigning the data. In conclusion we need a central time reference that is sent to all nodes and timestamps the data collected with a time delays of less than 10μs between reference and each nodes.

2.2.2 Existing synchronization solutions

Numerous synchronization solutions are commercially available, in both wired and wireless form. Even though wired solutions usually have better accuracy of synchronization, they suffer from the requirement of cables that run between the different units of a system.

Existing popular solutions for operating multiple data converters in lockstep involve electrically wiring together multiple data acquisition computer peripheral cards in a single computer. This is exemplified by products from National Instruments Corporation, which make use of a Real-Time System Integration (RTSI) interface connector and a ribbon cable to synchronize data acquisition cards to a

single clock source [129]. This approach works well within a single computer, but becomes more onerous with multiple computers, especially as the distance between them increases.

Another solution is to utilize timing information provided by the ubiquitous Local Area Networks (LANs), which already interconnect most computers. However, in this context, the typical Network Time Protocol (NTP) does not provide an accurate enough clock (Mills *et al.* [173]), and when corrections occur, discontinuities are formed in the data (Minar [174]). The Precision-Time-Protocol (PTP) is much better, but also suffers from abrupt jumps to corrected values [129]. Furthermore, in practice, these protocols only work well on a wired network. Indeed, wireless network latencies have a standard deviation of over 7ms, meaning that offsets of 21ms in simultaneous samples are not unheard of.

Another solution using the Wireless-Fidelity (Wi-fi) network has been presented in (Elson *et al.* [76,77]). This is called RBS for Reference-Broadcast Synchronization, and it is a solution where nodes send reference beacons to their neighbours using physical-layer broadcasts. A reference broadcast does not contain an explicit timestamp; instead, receivers use its arrival time as a point of reference for comparing their clocks. They demonstrate μs performance using off-the-shelf 802.11 wireless Ethernet. The main limitation of this solution is that it requires a network with a Physical Broadcast Channel (PBCH), not available for example in point-to-point networks. It is also a solution where the time between two broadcasts may vary from 10ms to seconds. In addition, it relies on reference broadcast reaching every node at same time. Finally the standard deviation is over 6μs which means that the 95% bound is around 20μs, which is not acceptable for our system.

Global Positioning System (GPS) is currently the only solution that can guarantee a sub-microsecond accuracy (Behrendt and Fodero [11]). Its main drawback is that it operates via line-of-sight to a GPS satellite. Otherwise synchronization will be lost with the typical GPS receiver chip in just a few hours, or even a few seconds for the cheapest chips (Lombardi [157]). In addition, despite the high accuracy, the update rate provided by typical GPS chips is low (on the order of 50ms to 500ms).

Table 2.1 lists the different solutions available for synchronizing networks of sensors.

Solution	Technology	Sync performance	Type	Cost
NI cards	RTSI/PXI	10ns±500ps [129]	wired	high
LAN	NTP	8.2ms±18ms [174]	either	low
LAN	PTP IEEE 1588	25μs±150ns [129]	wired	high
GPS	GPS	35ns (2σ=340ns) [11]	wireless	high
Wi-Fi	Wi-Fi	5ms±7ms [77]	wireless	low
RBS	Wi-Fi	6.29μs±6.45μs [76,77]	wireless	low

Table 2.1: Comparison of available solutions for synchronization.

2.2.3 Additional requirements

As said earlier, the human hearing system has a remarkable dynamic range of over 130dB [144]). This means that, in addition to an efficient synchronization system, the acquisition system needs a dynamic range wide enough to be able to record the full range of data with high precision.

The optimal case is to know the minimum and maximum signal that will be measured and utilise the entire sensitivity range to obtain the best measurement possible. The auditory threshold at 1kHz is $p_0 = 20\mu$Pa (rms), which is used as a reference for the sound pressure level (SPL) measured. This means that a normal conversation SPL of 40 to 60dB (ref p_0 at 1m) is equivalent to 2×10^{-3} to 2×10^{-2}Pa:

$$SPL = 20 * log(\frac{P_{rms}}{p_0}) \tag{2.1}$$

In the following estimation we will consider a pressure of P_{SPL} = 50dB (20μPa). The standard reference input signal for microphone sensitivity measurements is a 1 kHz sine wave at 94dB sound pressure level (SPL), or 1 pascal (Pa). Most of the commercially available MEMS have a sensitivity of S = -42dBV (for the same reference of 0dB = 1V_{rms}/Pa (Fig. 2.1).

The conversion between input (dB SPL) and output (dBV) is given by:

$$Output(dBV) = S - (94 - Input(dBSPL)) \tag{2.2}$$

This means that the input of 50dB (SPL) is equivalent to an output (dBV) of:

$$Output(dBV)\ P = -42 - (94 - 50) = -86 \tag{2.3}$$

or

$$Output(V)\ V_{outP} = 10^{(-86/20)} = 50\mu V \tag{2.4}$$

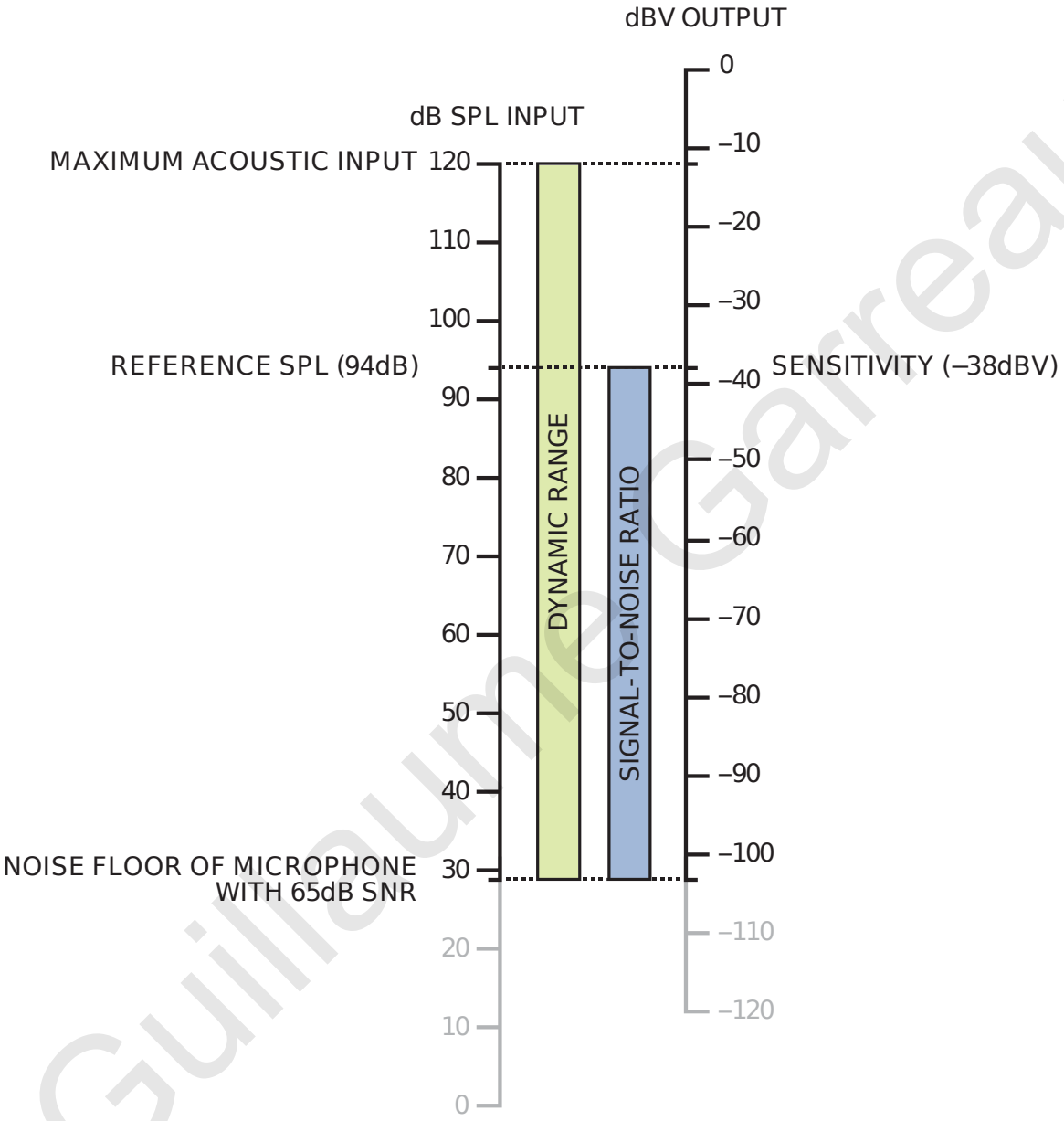


Figure 2.1: Relationship between dB SPL input and dBV output for analog microphones (from [150, 151]).

This means that for an input of a normal conversation (P = 50 dB SPL = -86dBV), we can expect an output voltage of only $V_{outP} = 50\mu V$ (Eq. 2.4).

Another important information to take into account is the Signal to Noise Ratio (SNR). Considering commercially available MEMS microphones, an average SNR value is 59dB, which means a noise floor output of:

$$V_{outSNR} = 10^{\frac{-42-59}{20}} = 8.9\mu V \tag{2.5}$$

Furthermore, if the signal is sampled with an ADC with 16bits precision and 5V reference, then the sampling accuracy is equivalent to:

$$S_{ADC} = 5V/(2^{16} - 1) = 7.63\mu V/bit \tag{2.6}$$

Which is of the same order of magnitude of amplitude than the expected output signal. Therefore pre-amplification will be necessary before sampling our microphone output.

24bits ADCs are also available: these allow higher precision (Eq. 2.7), but additional processing is required as the standard data format nowadays is 8 or 16bits.

$$S_{ADC} = 5V/(2^{24} - 1) = 0.3\mu V/bit \tag{2.7}$$

The last parameter to take into account is the dynamic range of the microphone, which is linked to the maximum output value the microphone gives. It is defined by a maximum acoustic input level of 120dB SPL at the upper end and the SNR at the lower end (Fig. 2.1):

$$dynamic\ range = 120 - 94 + SNR = 120 - 94 + 59 = 85dB \tag{2.8}$$

which is far from the human hearing system capability.

The 85dB dynamic range of the microphone output voltage is equivalent to:

$$dynamic\ range = 10^{85/20} = 158.5mV \tag{2.9}$$

Again, to make a better use of the full range of value offered by the ADC (5V), pre-amplification is required.

2.3 Custom built acoustic data acquisition unit

Section 2.1 has shown how timing is crucial in natural systems, while section 2.2 provided an overview of the state of the art synchronization systems commercially

available. The challenge raised by quantization and dynamic range to exploit the ADC's full capacity for sampling audio signals was also introduced. In this section, the list of the specifications and trade-offs for the acoustic data acquisition system are given.

2.3.1 Specifications and trade-offs

A particular aspect of this work is to obtain as much information as possible about animate entities (with main target being humans) in an area under surveillance. The methods used range from techniques such as source localization and source separation, to source identification through the processing of both passive and active acoustic signatures. For example, source separation can be used to determine the number of speakers in a room and micro-Doppler (mD) to identify the people present in a room through their characteristic walking gate.

To fulfil these tasks a large number of distinct data acquisition units, each of which may be recording and/or emitting sound waves, is used. Furthermore the waveforms of the collected acoustic data are digitized and processed by computer.

Acoustic data cover three different ranges: the first is the audible range, which consists of the range of frequencies that the human acoustic system can sense, which will henceforth be referred to as the "Audible Frequency" range. It is traditionally defined as 20Hz to 20kHz and is usually chosen for the study and simulation of the human auditory system. The second range is the "Low Frequency" (LF) range or seismic range. It contains frequencies range below 20Hz. It will be use for seismic wave measurement to model how sand scorpions use vibrations in the ground to locate and catch their prey. Finally, the third range is the high frequency (HF) range, referred to as "Ultrasonic Frequency" range in this work. It is the frequency range above 20kHz (this work focuses in the 40±1kHz window). The work here is mostly inspired from what bats do and in a lesser extend dolphins.

The specifications listed below are general specifications for an acquisition system, however some may only be relevant to a specific frequency range and it will be indicated. In addition some specifications are related to different parts of the system (software or hardware) or related to legal issues (legislation that may differ from one country to the other).

2.3.1.1 Multichannel recordings

The system must allow capability for multichannel recordings, as data from several sensors will be acquired at the same time and in a synchronous manner (the crucial importance of the synchronization was explained in the previous section). In addition, as the interest is to record various frequency ranges, the channel design must be done bearing in mind that flexibility in implementation must be allowed for easy adaptation to different frequency ranges.

2.3.1.2 Parallel processes

The system must support several parallel processes on-board. This is important as the system will simultaneously record from multiple channels, while monitoring the synchronization process both internally and externally in a network of nodes (made from multiple copies of the system). Furthermore, in the case of the ultrasonic range, the system needs to have the ability to generate and emit a sound wave at a precise frequency and of a chosen periodic pattern (sine, square, triangle...).

2.3.1.3 Dynamic range

The system must be designed in a way that the recorded signal uses as much of the available dynamic range as possible to create the best dataset possible. In order to do so, amplification of the signal at the output of the sensor must be done with the correct gain. A variable gain chain may be necessary in order to allow adaptation to various scenarios, for example a source moving towards or away from the sensor will cause variation in the signal amplitude measure.

A microphone with higher sensitivity, meaning a higher level output for a fixed acoustic input (i.e. a lower absolute value in the dB scale), could be chosen for better performance. However a microphone with higher sensitivity typically has less headroom between the output level under typical conditions, such as normal conversation sound level, and the maximum output level (120dB SPL, Fig. 2.1). A microphone with higher sensitivity is more likely to cause distortion. This distortion often reduces the overall dynamic range of the microphone.

2.3.1.4 Trade-off between performance and sensing range

In the ultrasonic case, a trade-off must be made in the system performance. The maximum sensing range is limited by the attenuation of the sound wave in the air. The attenuation is the sum of several components: the power of the transducer emitting the sound, the scattering of the sound wave in the air (Eq. 2.10), the absorption of the sound in the air (Eq. 2.11), the reflection parameter of the obstacle (i.e. the acoustic object to be detected) and the sensitivity of the receiver.

The scattering of sound waves is governed by an inverse square law :

$$scattering(x) = 20 * log(1/x) \tag{2.10}$$

with x the distance in meter.

The attenuation of the sound in the air is calculated with:

$$I(x) = I_0 * exp(-\mu x) \tag{2.11}$$

with I_0 is the intensity of the sound at the origin (transducer) and $I(x)$ the intensity at distance x, μ is the linear attenuation coefficient. The attenuation coefficient of the sound in the air α is:

$$\alpha = 10 * log(\frac{I(x)}{I_0}) = k * f^2(dB/m) \tag{2.12}$$

with f the frequency in Hz and k a constant.

Hence a lower frequency means a further detection range, but frequency is also linked to the minimum size of the features that can be detected as the minimum detectable feature, when measuring with a wave, is equal to its wavelength:

$$\lambda = c/f \tag{2.13}$$

with c being the speed of sound in air.

Hence the importance of the frequency, if one wants to sense far away, a lower frequency must be used. But lower frequency means a lower precision, i.e. the minimum size of the features that can be detected increases.

One way to decrease this limitations is to use a higher voltage; this follows from the fact that if more energy is send out, potentially more energy can be received back and, thus, smaller features or features that are further away can be measured. However, this requires the supply of more power to the system, which is important if the system has to be deployed outside, far from the main power supply or if it

is receives its power from a usb cable that limits the amount of power that can be transmitted. In case of a battery operated system, increased power usage means a shorter recording ability.

The last two factors, reflection parameter of the acoustic object and sensitivity of receiver, are less dependent of the frequency. The reflection parameter is characteristic of each object. The choice of receiver sensitivity was discussed previously.

2.3.1.5 Safety requirement

In addition, careful attention has to be given to safety issues. In the case of the sound emission, strict safety rules exist and these have to be respected. A maximum sound pressure level of 115dB SPL is permitted [7].

2.3.1.6 Data storage

The system needs to be able to locally store the data collected, which will be subsequently transmitted to a central storage unit that will contain the data collected from all the nodes of the network. Thus, the format of the data collected must take into account local memory, communication bandwidth available, and the dynamic range in order to optimize data transmission and storage. The number of parallel channels, the ADC sampling speed and precision are additional parameters to be taken into account for data storage. In addition, local buffer size and buffer size on the central unit, as well as final file size, have to be carefully chosen.

2.3.1.7 Sampling

In order to limit the communication bandwidth usage and/or the storage required for the data recorded, front end data processing may be necessary. Filtering of the signal before sampling can be performed with passive components (RC filters) in the sampling channel to 'clean' the data and remove unwanted frequencies. In addition, digital filtering or re-sampling may be implemented to sub-sample and, thus, reduce the data set size.

2.3.1.8 Pre-processing

The system must be capable of more advanced local pre-processing of the data, for example a filter bank and an Address Event Representation (AER) coding sim-

ilarly to cochlea operation. Work has been done in this direction, where multiple neural network-like hardware systems are brought together to form a "brain" or "distributed cortex" (Andreou *et al.* [43, 103]).

2.3.1.9 Design technology

The design must allow for certain flexibility in the system, such as adaptation to different kinds of sensors and different frequency ranges, therefore a preconfigurable module should be chosen, as well as VLSI technology, with one or more FPGA. Even though an ASIC design would assure better performance and lower power consumption, the flexibility requirement would not be met and, development time would be longer and at a higher cost.

2.3.2 Actual design

Taking into account the requirements for the acoustic data acquisition system which were outlined in the previous section, the actual hardware-software platform that was designed for the experiments is presented in this section.[1] The hardware platform is based on two types of units: a frequency modulation synchronization unit (FM SYNC unit) and a data acquisition unit (DACQ unit). A distributed sensor network comprises of one FM SYNC unit and multiple DACQ units.

2.3.2.1 Synchronization considerations

In section 2.2, after demonstrating the necessity of a highly synchronized acquisition system, an overview of the different commercially available solutions, for synchronizing networks of sensors was given while showing that none is satisfactory. The GPS solution is not compatible with the indoors scenarios, the RBS solution has limitations such as none regular synchronization period (10ms to several seconds) and a synchronization accuracy too widely spread (95% bound is 20.53μs) and the other solutions are wired, which is a disadvantage when deploying multiple devices to cover medium to large area of surveillance. As a consequence, efforts were focused

[1] The version of the system presented here was fully built at UCY by Guillaume Garreau. The underlying design (FPGA code, PCB layout) was a collaborative effort with Philippe Pouliquen, who was also working on the SCANDLE project. Testing and tuning of the solution was conducted at UCY but also at JHU where Thomas Murray also helped.

on engineering a custom synchronization solution. A wired solution was not an option, as some of the experimental protocols required significant movement of the subjects between the sensors, and arranging the wires to avoid creating unnatural obstacles would be impractical. An optical link solution (similar to what is used for infra-red consumer electronic remote controls) was also rejected as the signals are easily blocked unless the system is operated indoors where the room walls serve as convenient reflectors. Finally, a wireless architecture was selected, employing a single master transmitter that transmits the synchronization information to the slave receivers in each data acquisition unit to coordinate the operation of the digital-to-analog converters (DAC) and analog-to-digital converters (ADC).

Different methods of broadcasting radio waves and choosing the frequency and/or phase modulation (FM) were investigated as it is more robust than amplitude modulation (AM) with respect to interferences and simpler to implement compared to other methods, such as spread-spectrum. To calculate how much bandwidth BW is necessary to use to reliably transmit a signal, FM modulation engineers use a rule of thumb, also called Carson's rule, which states that:

$$BW = 2 * (\beta_{FM} + 1) * f_s \tag{2.14}$$

β_{FM} is the modulation index and f_s the signal frequency. Another way to implement this, is to take a ratio of at least 7 between f_s and BW. In this work, a ratio of 10 is selected. Thus, in order to transmit a 100kHz synchronization signal with FM a radio band of approximately 1MHz in width, on which it was possible to transmit without a license, is required. The radio frequency selected was 72.5MHz, which is the midpoint of the 72 to 73MHz band, a band currently reserved for radio controlled model airplanes (US standard [57]). The used bandwidth, though not reserved for controlled model airplanes in Cyprus (which is the 34.995MHz to 35.225MHz, [182]) is dedicated to Professional and Professional Amateur Mobile Radio (PMR/PAMR), which is mainly used for mobile communication and broadcasting (EU standard [182]). Shifting the main frequency to the Radio-controlled frequencies window in Europe is simply done by updating the FPGA code with different digital clock manager (DCM) input parameters. It is also possible to change the main crystal clock frequency. In addition, the use of low power ensures that the range of synchronization is limited.

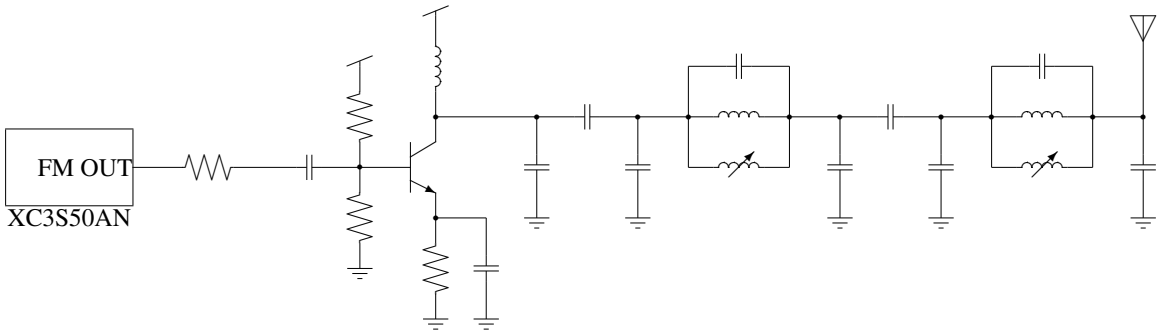


Figure 2.2: Schematic of FM transmitter with all components shown.

2.3.2.2 FM transmitter

The FM transmitter uses a Xilinx XC3S50AN FPGA (in a -5 speed grade) with internal configuration memory to synthesize the transmitted waveform. All that is needed is an external amplifier and optionally additional filtering to reduce the harmonics. In principle, since the FPGA is producing a square wave output, the harmonics of the square wave need to be removed from the signal at the antenna connector. In practice, the FPGA output pins are slew rate limited at the carrier frequency, and no additional filtering was deemed necessary. Nonetheless, the PCB contains mounting holes for the full circuit, as diagrammed in Fig. 2.2.

The actual circuit used is shown in Fig. 2.3. From the FPGA, the FM signal goes through a DC blocking capacitor (100nF) to the base of the RF bipolar transistor (2N3866 or equivalent). The transistor base is biased by the resistive divider formed from the two 3.83kΩ resistors. The transistor emitter is connected to ground via a 10Ω emitter degeneration resistor (to prevent thermal runaway) and an AC bypass capacitor (100nF). The transistor collector is connected to the 5V power supply through an RF choke, which increases the amplitude of the output wave to approximately 10V. The antenna is connected to the collector through a 10pF coupling capacitor.

2.3.2.3 FM receiver

The FM receiver is composed of a SA602A RF mixer, a SA604A FM decoder, an LTC1196 8bit 1Msps ADC and an LT1719 high speed comparator. A block diagram is given in Fig. 2.4 A/. The complete circuit is shown in Fig. 2.5.

The signal from the antenna is coupled to the SA602A input, where it is mixed with the reference signal ($\frac{14}{17}$ 75MHz $\approx$ 61.8MHz) generated by the Xilinx XC3S50AN

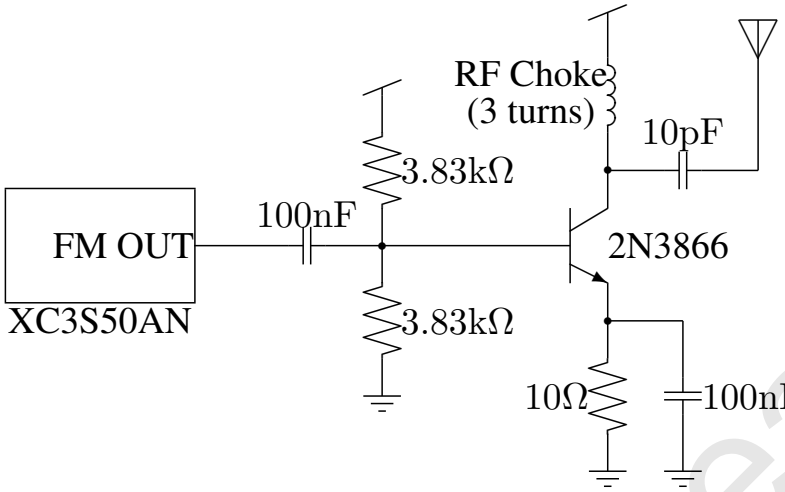


Figure 2.3: Schematic of FM transmitter as actually implemented.

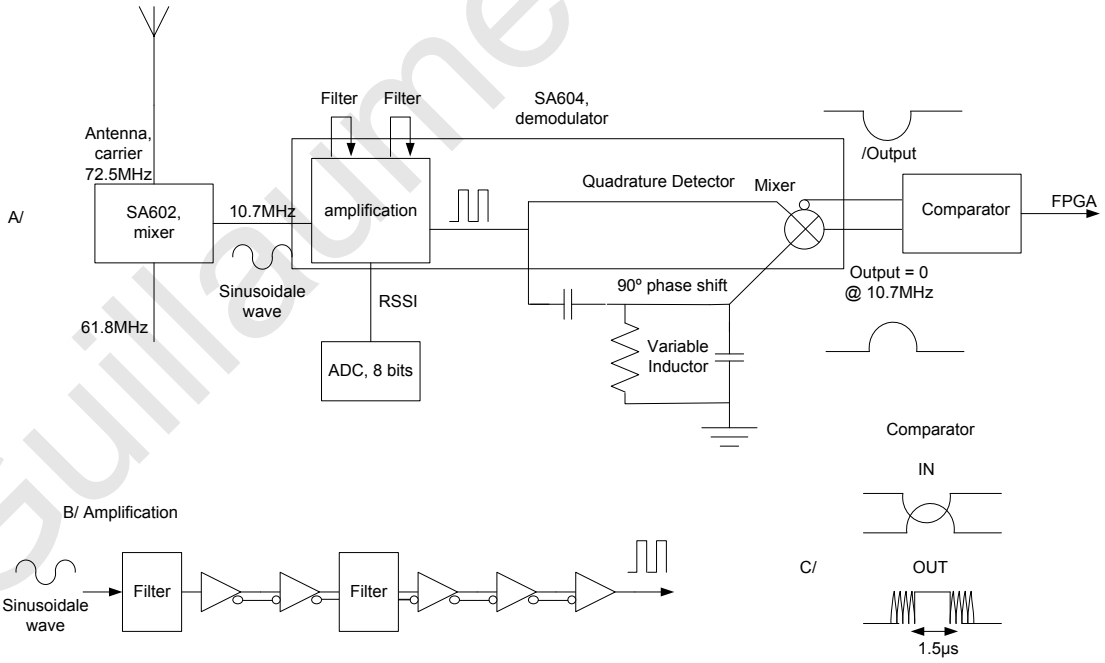


Figure 2.4: FM receiver.

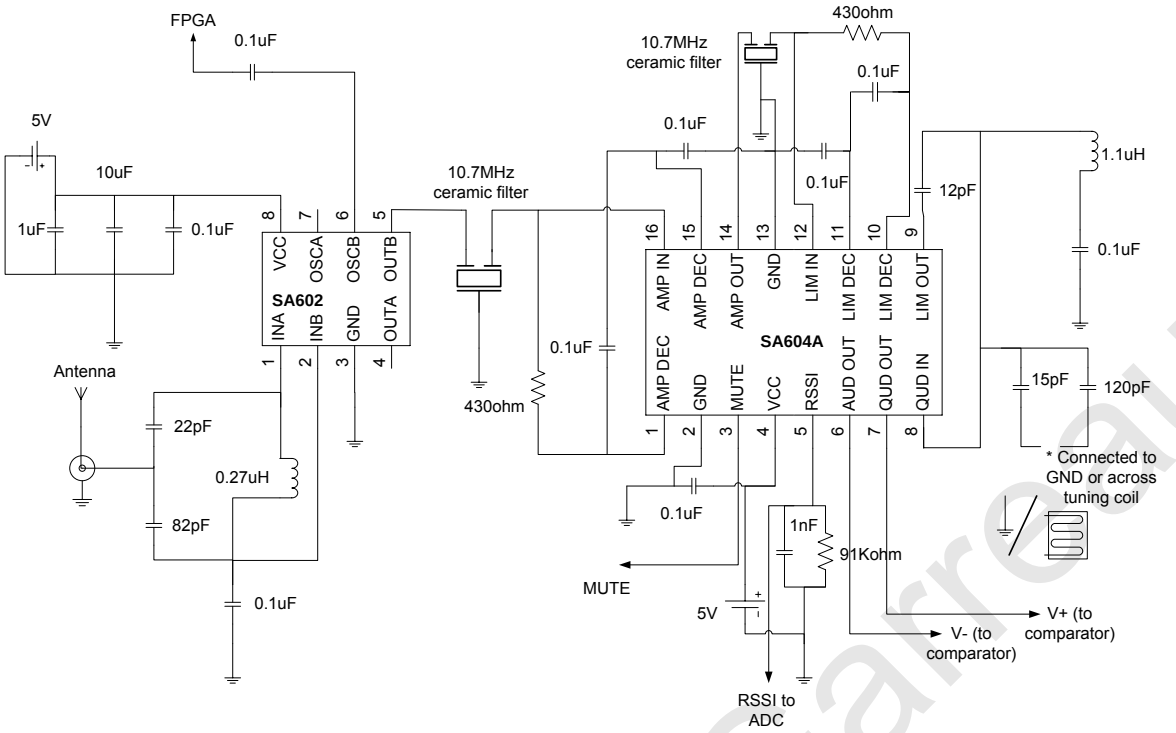


Figure 2.5: Schematic of FM receiver.

FPGA. Since the original signal has a carrier frequency of 72.5MHz, the mixed signal will appear at 72.5 - 61.8 = 10.7MHz. The signal is then filtered by a ceramic filter with a center frequency of 10.7MHz and a bandwidth of 150kHz. The signal is amplified by an amplitude limited amplifier in the SA604A, then filtered and amplified a second time up to saturation. So the current consumption depends on the input signal (Fig. 2.4 B/). The signal is then phase shifted by 90 degrees in the quadrature tank composed of the 1.5μH tunable coil and a 135pF capacitance (formed of the parallel combination of a 15pF and a 120pF capacitors). The phase shifted signal is then mixed with the un-phase shifted signal to produce the decoded output (demodulated). The output of the tank circuit is multiplied by the original (0 degree phased) signal and low-pass filtered (to remove the 21.4MHz component). So the center frequency results in 0Vdc and the off center will produce a positive or negative voltage. The SA604A has complementary decoded outputs at pins 6 and 7 that get the 0 degree phased and 180 degree phased demodulated signal. These outputs are fed to the high speed comparator to create a digital signal for the FPGA to decode further. The length of the pulse is about 1.5μs, It is determined by the components used for the tank and can not be changed. It represents the

time needed by the demodulator to re-adapt itself to a new frequency (Fig. 2.4 C/). The desired waveform for the output is a square wave, although the transitions will necessarily be quite noisy. It is best if the pulse width is somewhat shorter than 5μs. The FM receiver has to be tuned manually, it is very accurately done with the help of an oscilloscope. However, a Matlab® script has been developed to help the tuning (script "tune.m" in AppendixB). The VHDL script and User Constraints File (UCF) file used for the tuning of the FM receiver of each DACQ units can be found in AppendixA. The SA604A also has a Received Signal Strength Indicator (RSSI) output, which is digitized by the 8-bit ADC (controlled by the FPGA).

2.3.2.4 FM SYNC unit

The purpose of the FM SYNC unit is to transmit a signal that can be used by each data acquisition unit to tag its data and to compensate for variations in the unit's local oscillator. The main component of the FM SYNC unit is the FM transmitter presented earlier. Because of the high accuracy desired of the synchronisation signal, the bandwidth of the radio signal is quite large (approximately 1MHz). For this reason, the carrier frequency chosen for its operation is 72.5MHz, which is the midpoint of the 72 to 73MHz band reserved for radio controlled model airplanes (US standard). In addition the crystal oscillator used is a 20MHz temperature controlled crystal oscillator (TCXO) with an accuracy of 0.28ppm. The other units used a 75MHz oscillator (no temperature controlled) with 50ppm accuracy. A block diagram is given in Fig. 2.6 and a picture of the board is shown on Fig. 2.7.

The FM SYNC unit sends raw bits at a fixed rate of 200,000 bits per second (200kbps). A raw bit of one is encoded as a phase shift, while a zero is encoded as no phase shift. An encoded data bit is composed of four raw bits, the encoding is given in Tab. 2.2. The data rate is therefore 50,000 bits per second (50kbps). This gives a modulation index of:

$$\beta_{FM} = 2 * \frac{f_s}{bitrate} = 4 \tag{2.15}$$

The FM SYNC unit transmits a SYNC bit followed by the 60 data bits of the timestamp (least significant bit first), followed by 419 zero data bits. The pattern then repeats, with the timestamp incremented by one. Thus, a new timestamp is sent every 9.6ms, which corresponds to the length of a frame in the DACQ units.

The FM SYNC unit can be connected to a computer running either Microsoft

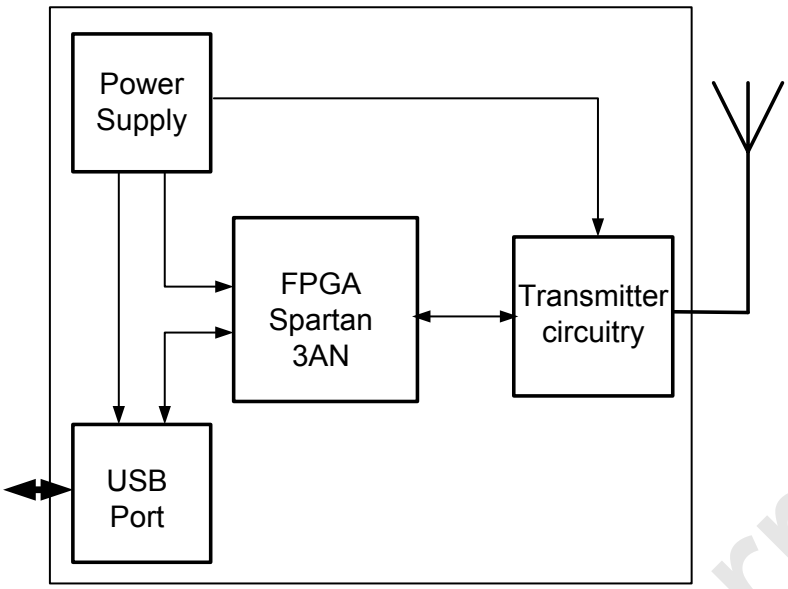


Figure 2.6: Block diagram of the FM SYNC unit.

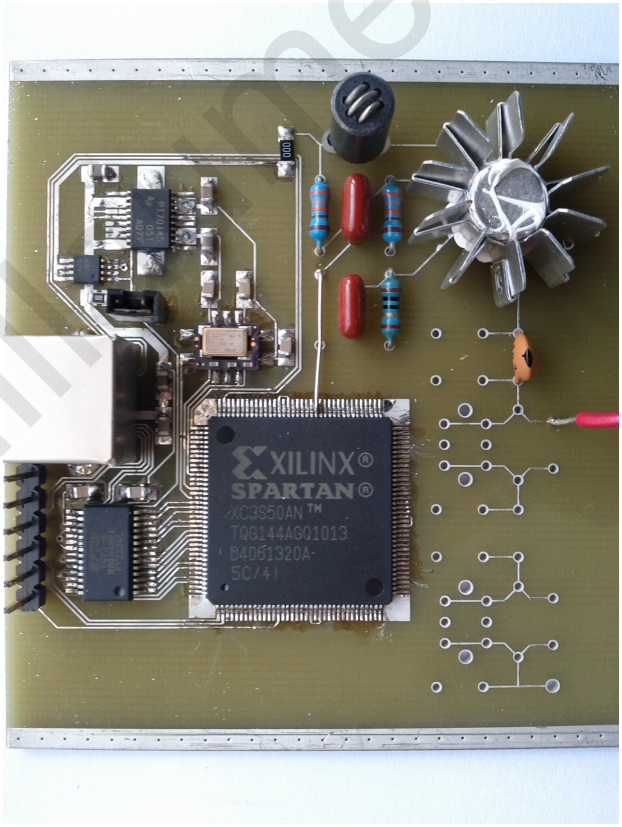


Figure 2.7: Picture of the FM SYNC unit.

Data bit	Encoding
0	1000
1	1010
SYNC	0010

Table 2.2: Data bit encoding.

Windows or UNIX System through USB. The timestamp transmitted by the FM transmitter can be read and altered at any time. The VHDL script and UCF file for the FM SYNC unit programming can be found in Appendix A. Scripts for setting and reading the timestamp can be found in Appendix B.

The FM SYNC unit sends the timestamp to all the other units, so each unit crystal is locked to the same reference signal. If a unit loses the reference, it will keep running normally until it finds again the reference signal and then re-align its clock cycles to the reference.

2.3.2.5 DACQ unit

Multimodality and flexibility of implementation are two of the main characteristics of the DACQ units.

Overview of the DACQ unit Data acquisition on the custom-built boards is controlled by the same FPGA as the FM SYNC unit, but with a lower speed grade. The board uses a 75MHz crystal oscillator (50ppm). It is used to create a 61.8MHz clock, used as reference for the FM receiver, and a 90MHz clock for the FPGA. This FPGA coordinates the operation of the data converters and transmits the data to the host computer through the USB interface IC. The analog input section of the Printed Circuit Board (PCB) consists of four parallel sampling chains. Each chain consists of an AD8656 dual OpAmp, a PGA112 programmable gain amplifier and an AD7686 16bit 500ksps ADC. The reference voltage is supplied by a TLE2425 2.5V reference. The FM receiver section of the PCB is composed of a SA602A Radio Frequency (RF) mixer, a SA604A FM decoder, a LTC1196 8bit 1Msps ADC and a LT1719 high speed comparator. The details of the FM receiver were presented earlier.

A block diagram of the DACQ unit is given in Fig. 2.9. The board was designed to accommodate more powerful FPGAs. However, rather than replacing

the XC3S50AN part with another, it only adds connectors for an Opal Kelly type FPGA daughter board (www.opalkelly.com). This allows to increase the capabilities of the unit by adding some pre-processing and can thus be used for a much wider range of applications, including, standalone and real-time applications. This would permit implementation of an artificial neural network on FPGA as described in [37,39,41,175]. It has be shown that learning of real-world stimuli can be implemented using a spike-driven neural network [22]. In case of several DACQ units used, each for a different modality, a network of communicating neuron arrays can be implemented as shown in [40,42,45] that would allow to simulate a 'brain' like system [103,154].

Decoding the synchronization pattern The signal from the FM receiver section (comparator output) is sampled and if it is high more than half of a 5μs period then it is a '1', else a '0'. This value is later referenced as raw bit. The pulse width from the comparator is expected to be shorter than 5μs (Fig. 2.4 C/), so if the interval ends with the comparator output high, it is assumed that the intervals are straddling bit boundaries, and that the interval phase needs to be adjusted (Fig. 2.8). The direction of the adjustment is based on whether the interval had a majority of ones, in which case the interval is stretched, or a majority of zeros, in which case the interval is shrunk.

Every 5μs, a new raw bit is stored. Four raw bits will make a data bit, using encoding given in Tab. 2.2. When a series of "00SYNC" data bits is detected the reference sync pulse is set to '1' for one clock cycle. Then there are 60 additional cycles of $4 \times 5\mu$s to read the timestamp value, followed by 419 cycles of zero data bits. The FPGA has also its own sync pulse, named local sync pulse, that goes high for one clock cycle every 9.6ms. A counter counts the number of clock cycles between the two sync pulses. Another counter, the sequencer counter, is used as reference for all the periodic processes that are controlled by the FPGA (ADCs, PGA, ADC, serial connection, buffers, data storage). The sequencer counter period needs to be adjustable so that the DACQ unit clock is phase-locked to the FM transmitter clock. The increment adjustment is phs_inc (1 integer bit, 18 fractional bits) and it is done by integrating the error signal (delay between the two sync pulses). phs_inc is updated with the delay between the sync pulses in number of clock cycles when the local sync pulse is 1 and the reference pulse sync is 0. The code limits the value of phs_inc

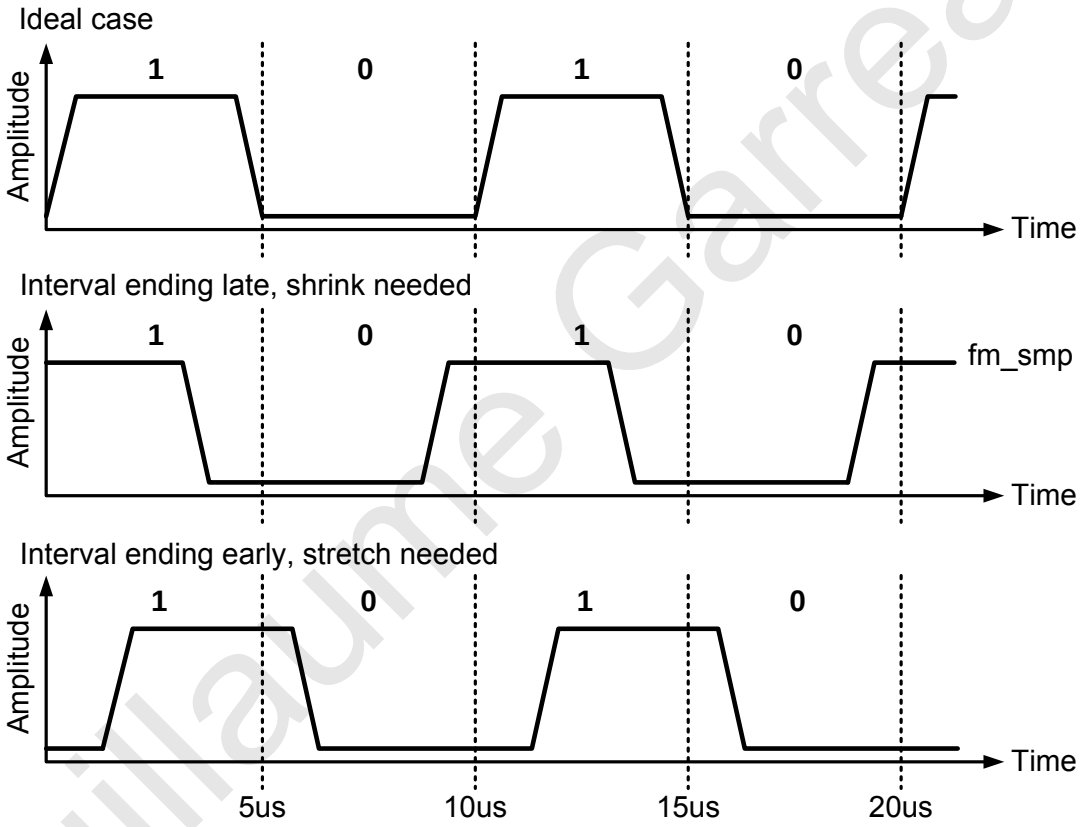


Figure 2.8: FM comparator output signal decoding.

Unit type	Analog inputs	Analog outputs	Sampling rate
Ultrasound	1	2	100kHz
ASU (4 microphones)	4	0	25kHz
Geophones	3	0	33kHz
Mannequin (2 biased microphones)	2	0	50kHz

Table 2.3: List of the different types of data acquisition units used.

to be between 0 and 2 (non-inclusive), nominally 1.

The phase-locking of the DACQ crystal unit is updated every 180 clock cycles or 2μs by 0 or ±1 clock cycle. However, the sync pulses have a period of 9.6ms (frames length). The phase-locking accuracy is embodied in the number of bits in phs_inc (1 part in 2^{18}) not by the 1 part in 180. For additional details, refer to the AppendixD.

Sampling lines Each unit is design to have up to four analog inputs and two analog outputs. The actual configuration depends on the components installed on the PCB (printed circuit board) and the firmware programmed into the FPGA. Note, however, that physical space constraints limit the number of analog connectors to a total of four, so only three or two analog inputs are available if the analog output(s) are used.

The analog inputs and outputs are connected to 16bits data converters. The digital data is transferred between a data acquisition unit and a host computer via a USB cable. The USB interface IC (integrated circuit) used in the data acquisition units limits the total bandwidth to 100,000 samples per second (100ksps). This in turn limits the analog sampling to 100ksps for a single channel data acquisition unit, 50ksps for a two channel unit, 33.3ksps for a three channel unit and 25ksps for a four channel unit. Table 2.3 summarizes these characteristics. A picture of an example of implementation is given on Fig. 2.10. It is the Acoustic Surveillance Unit (ASU) implementation which is a unit with four microphones, it is used to monitor an area, based on acoustic data.

Note that the ultrasonic units are listed as using two analog outputs as the ultrasonic transmitter transducer element is driven differentially to increase output power. The connector type is typically a BNC connector, unless the sensor requires a power supply (such as microphones). It is also possible to use Integrated Circuit

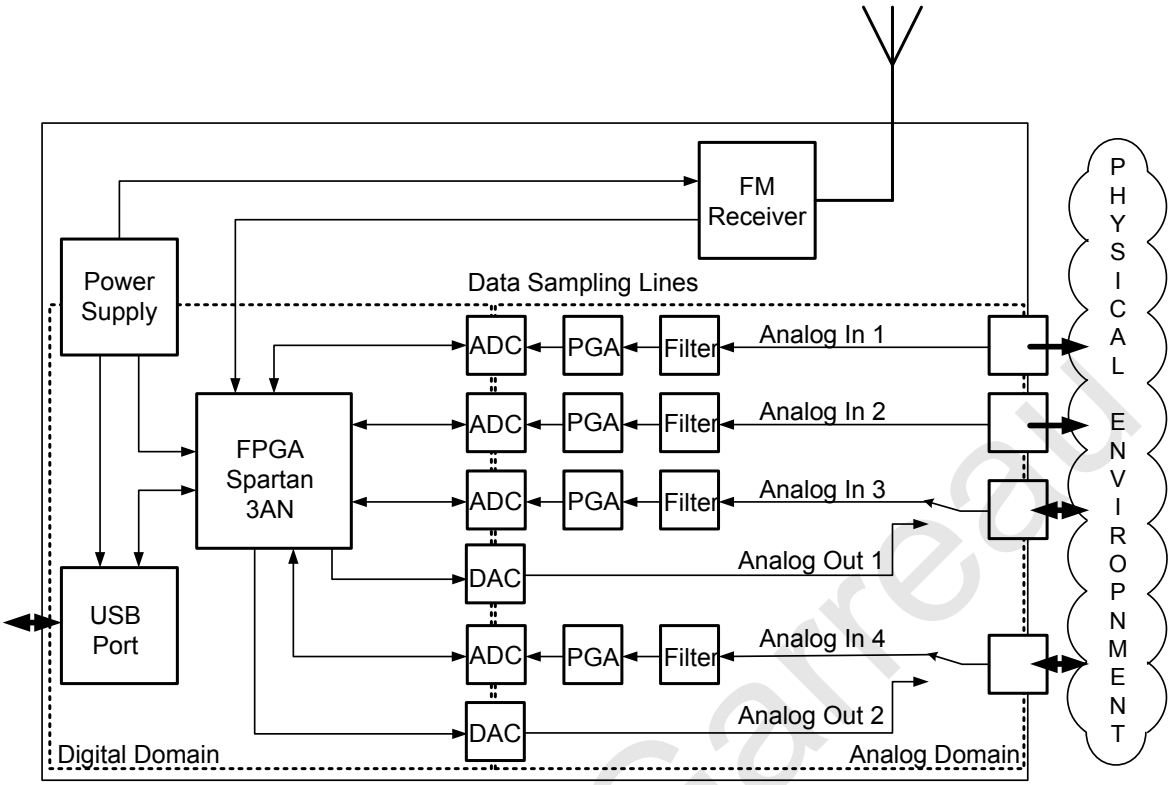


Figure 2.9: Block diagram of the data acquisition unit.

Piezoelectric (ICP) type microphones where the power is supplied using the same wire as the returned audio signal.

The components on the PCB are further customized according to the data acquisition units intended application. This customization includes adjustments to the gain and bandwidth of the analog channels, as well as the type of filters used (*e.g.* Butterworth versus conventional). The PCB design allows a large amount of flexibility in this regard, and a programmable gain amplifier is also included to extend the range of the ADCs (analog-to-digital converters).

The data acquisition units can be connected to a computer running either Microsoft Windows or UNIX through USB. The data acquisition units send their information in frames. Each frame is composed of a header of 12 16bit words followed by 960 samples. The arrangement of the frame data is shown in Tab. 2.4. Note that the sample words are interleaved, so that their arrangement depends on the number of input channels that the data acquisition unit has. The detailed content of the header is shown in Tab. 2.5. Additional informations are given in Appendix E.

The VHDL script and UCF file for the DACQ units programming can be found in Appendix A.

location	1 analog input	2 analog inputs	3 analog inputs	4 analog inputs
word 0	sync	sync	sync	sync
word 1	sync	sync	sync	sync
word 2	module ID	module ID	module ID	module ID
word 3	module ID	module ID	module ID	module ID
word 4	frame index	frame index	frame index	frame index
word 5	timestamp	timestamp	timestamp	timestamp
word 6	timestamp	timestamp	timestamp	timestamp
word 7	timestamp	timestamp	timestamp	timestamp
word 8	timestamp	timestamp	timestamp	timestamp
word 9	PGA and RSSI	PGA and RSSI	PGA and RSSI	PGA and RSSI
word 10	frame minimum	frame minimum	frame minimum	frame minimum
word 11	frame maximum	frame maximum	frame maximum	frame maximum
word 12	sample 0 channel 0	sample 0 channel 0	sample 0 channel 0	sample 0 channel 0
word 13	sample 1 channel 0	sample 0 channel 1	sample 0 channel 1	sample 0 channel 1
word 14	sample 2 channel 0	sample 1 channel 0	sample 0 channel 2	sample 0 channel 2
word 15	sample 3 channel 0	sample 1 channel 1	sample 1 channel 0	sample 0 channel 3
word 16	sample 4 channel 0	sample 2 channel 0	sample 1 channel 1	sample 1 channel 0
word 17	sample 5 channel 0	sample 2 channel 1	sample 1 channel 2	sample 1 channel 1
word 18	sample 6 channel 0	sample 3 channel 0	sample 2 channel 0	sample 1 channel 2
word 19	sample 7 channel 0	sample 3 channel 1	sample 2 channel 1	sample 1 channel 3
word 20	sample 8 channel 0	sample 4 channel 0	sample 2 channel 2	sample 2 channel 0
word 21	sample 9 channel 0	sample 4 channel 1	sample 3 channel 0	sample 2 channel 1
word 22	sample 10 channel 0	sample 5 channel 0	sample 3 channel 1	sample 2 channel 2
word 23	sample 11 channel 0	sample 5 channel 1	sample 3 channel 2	sample 2 channel 3
		...		
word 971	sample 959 channel 0	sample 479 channel 1	sample 319 channel 2	sample 239 channel 3

Table 2.4: Format of data acquisition frame.

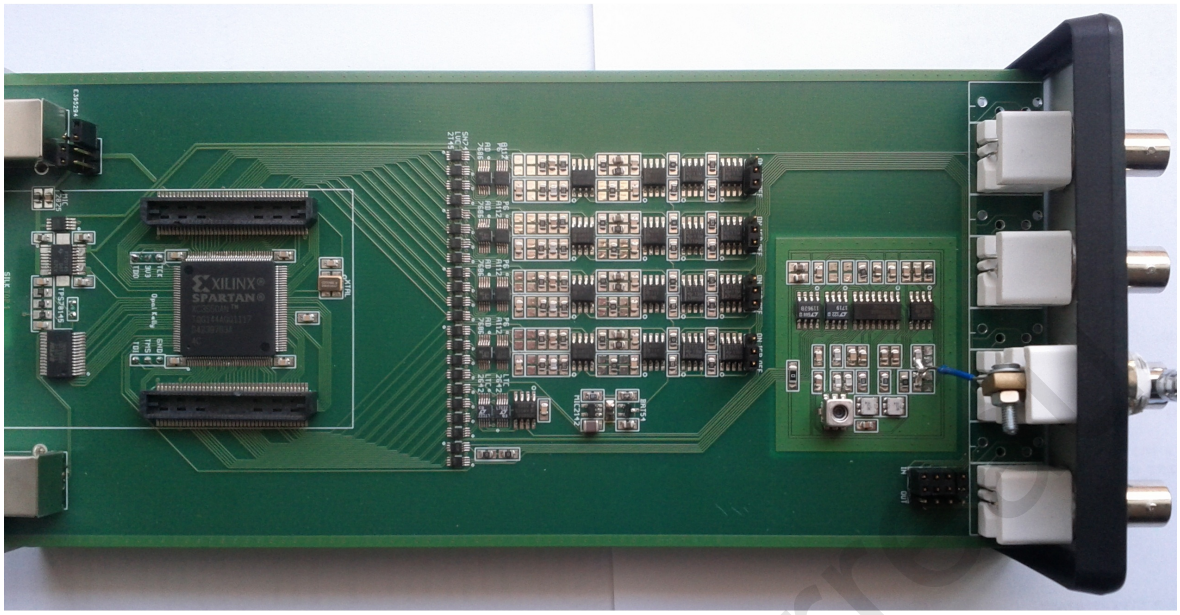


Figure 2.10: Picture of an ASU data acquisition unit.

	Most significant byte								Least significant byte							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
word 0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
word 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
word 2	module ID, byte 1								module ID, byte 0							
word 3	module ID, byte 3								module ID, byte 2							
word 4	frame index															
word 5	1	timestamp, bits 14 to 0														
word 6	1	timestamp, bits 29 to 15														
word 7	1	timestamp, bits 44 to 30														
word 8	1	timestamp, bits 59 to 45														
word 9	PGA								RSSI							
word 10	frame minimum															
word 11	frame maximum															

Table 2.5: Format of data acquisition frame header.

2.3.3 Hardware testing

This section provides evidence in support of the claimed synchronization performance of the FM SYNC unit described above. Initially, a Tektronix AFG 3252 signal generator was used to generate a 1kHz sync pulse every 100ms (10Hz). It was connected to a resistive attenuator and then through BNC cable extension to 3 DACQ units (each one connected to a computer). The FM SYNC unit was connected to a fourth computer. However, the first set of data collected was too noisy, because the resistive attenuator at the function generator output made the cabling more susceptible to noise coupling. Furthermore, one of the DACQ was programmed as a ultrasonic unit and the output DAC was adding noise (a sine wave) to the ADC data. Additional noise was also noticed coupling in from the USB cables when data collection was active. As a result, the DACQ units were reprogrammed to disable the DAC and the attenuation was moved to each unit input. Then another data set was collected. However, there were a few problems still. The main problem was that the sync pulse was too short (only about 10 samples of the ASU units, which sample every 40μs) and the amplitude was a bit low. So the amplitude was increased and the frequency lowered to 100Hz (to stretch out the sync) while keeping the repetition rate of 100ms. In addition all units were reprogrammed with the ultrasound bit code (and DAC disabled), so that they only sample on one channel, resulting in a 10μs sampling rate for all modules. But the results were still not as good as desired.

In the mean time, the idea came up that a different waveform might work better for measuring jitter. After a bit of trial and error, the optimal waveform was found (Fig. 2.11). Essentially, the long upward sloping section is used to measure the shift amount. The procedure is to find the lower (blue) and upper (green) plateaus which bound the sloping section (red line) by starting from the min and max and adding point to each set whose value is close enough to the min/max. The threshold is currently 1% of the overall amplitude (max-min). The min/max plateaus were introduced because each module has a slightly different voltage gain and offset (due to the components mismatch in the ADC channels). The two plateaus essentially allow a two-point correction of the data for these variations. The sloping portions before and after are there only to make the waveform symmetric.

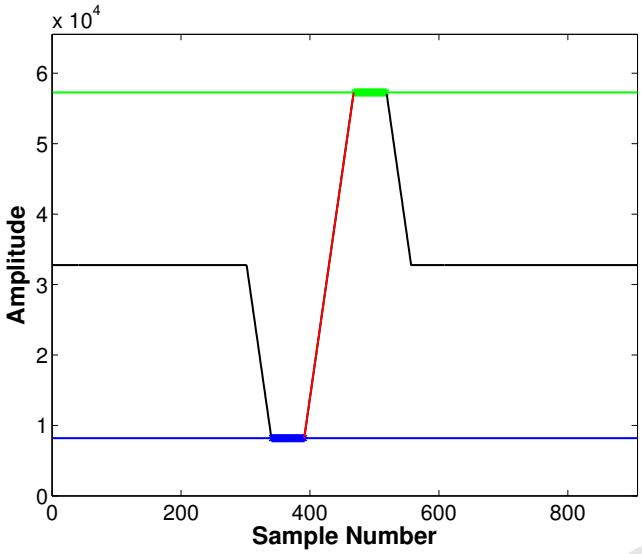


Figure 2.11: Optimal waveform used for synchronization precision measurement.

2.3.3.1 Analog pattern synchronization test

In the analog pattern test, four DACQ units and the FM SYNC unit are all controlled by different computers (five in total). The setup is shown schematically in Fig. 2.12. One of the DACQ units is programmed to generate a specific waveform through its DACs (with a period of 9.6ms and a sampling rate of 1MHz, unit noted waveform generator in Fig. 2.12) which is sent to three other DACQ units (noted US25, US33, US40 in Fig. 2.12) that digitize the waveform with their ADCs (at a sampling rate of 100kHz). The digitized waveform is then sent back to each host computer for analysis. Note that in order to reduce the noise in the measurement the ADCs, on the waveform generating DACQ, and the DACs, on the waveform receiving DACQ units, are disabled. Furthermore, attenuators were installed at the input of the waveform receiving DACQ units, so that the waveform generating DACQ unit could produce signals using the full dynamic range of its DACs without saturating the ADCs in the other units.

This experiment is designed to test the end-to-end synchronization performance of the DACQ units. Indeed, any measured time shift between the original waveform and the digitized waveform incorporates the effect of the phase-locked loop (PLL) phase error in the waveform receiving DACQ unit as well as the waveform generating DACQ unit, and also any phase delay caused by the analog signal chains within the DACQ units. By connecting each DACQ unit to a different computer, we are emulating the usual scenario for the intended use of the system.

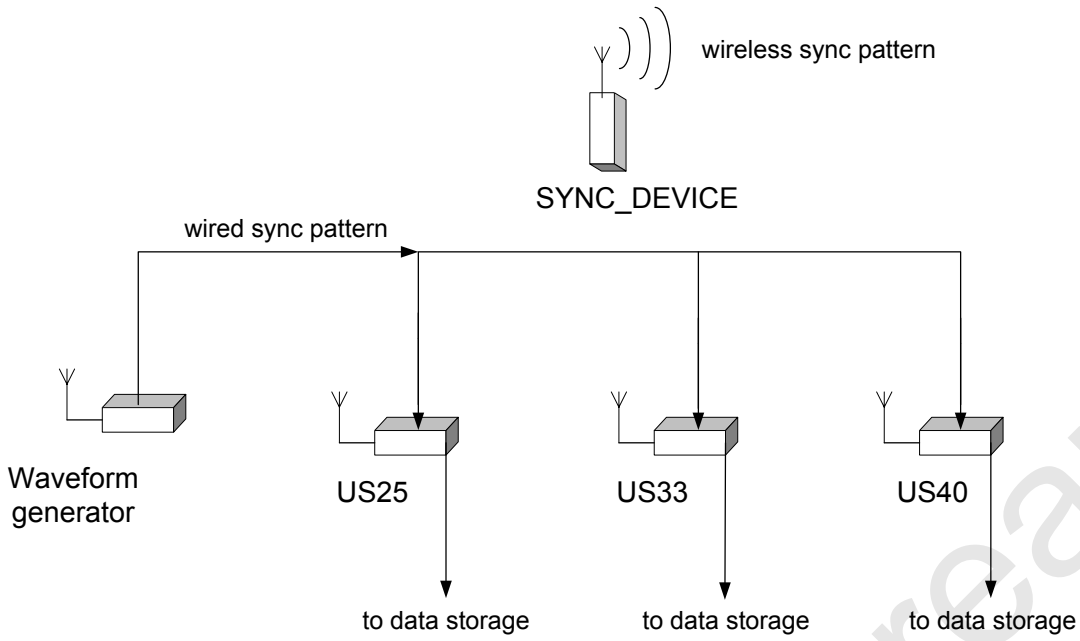


Figure 2.12: Schematic of the setup used for the analog synchronization precision measurement.

The generated waveform was designed to simplify the calculation of the time shift. A sample of the captured waveform is shown in Fig. 2.13. The waveform is composed of a central upward sloping section (approximately 100 samples at 100kHz), bounded by two plateaus (approximately 50 samples at 100kHz). These minimum and maximum plateaus are present to allow the calculation of the actual gain and offset of analog channel, which may differ from the nominal values due to manufacturing variations, drift and temperature. The central sloping section, after correction for gain and offset, and after fitting to a line, allows the calculation of the actual time shift (zero-crossing time). The short downward sloping sections (approximately 50 samples at 100kHz) on either side of the plateaus match the slope of the central section to avoid introducing additional frequency components in the signal spectrum and ensure that the waveform has a zero overall mean.

The histogram of the time shift measurements collected continuously over a period of approximately 30 hours (approximately 11×10^6 measurements) are shown in Fig. 2.14, and the mean and standard deviation are summarized in Tab. 2.6.

The result obtained (max delay of less than 5μs, average 0.4μs and standard deviation of 0.35μs) are very good. However, the measurements do include signal

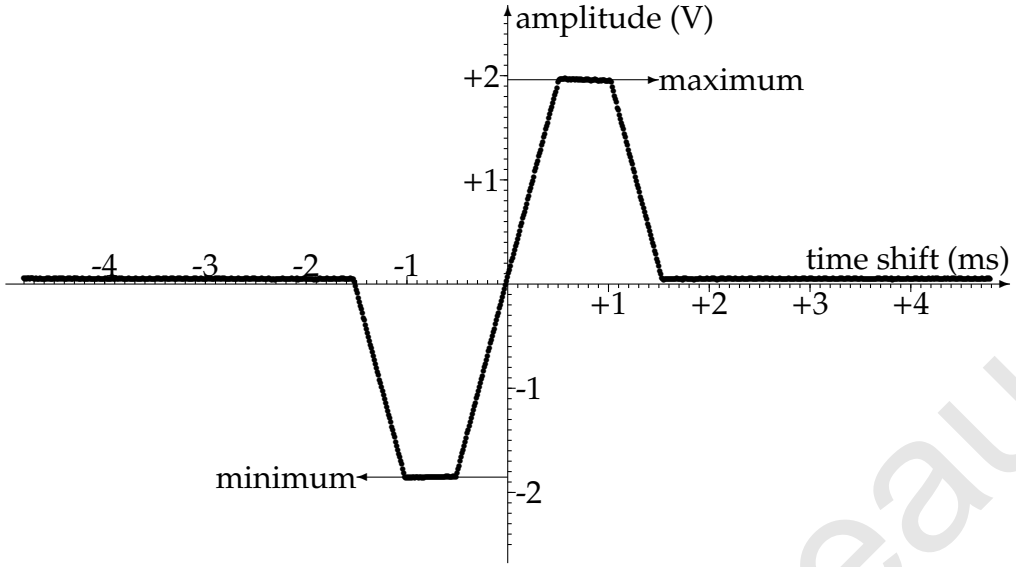


Figure 2.13: Waveform used for the analog synchronization precision measurement.

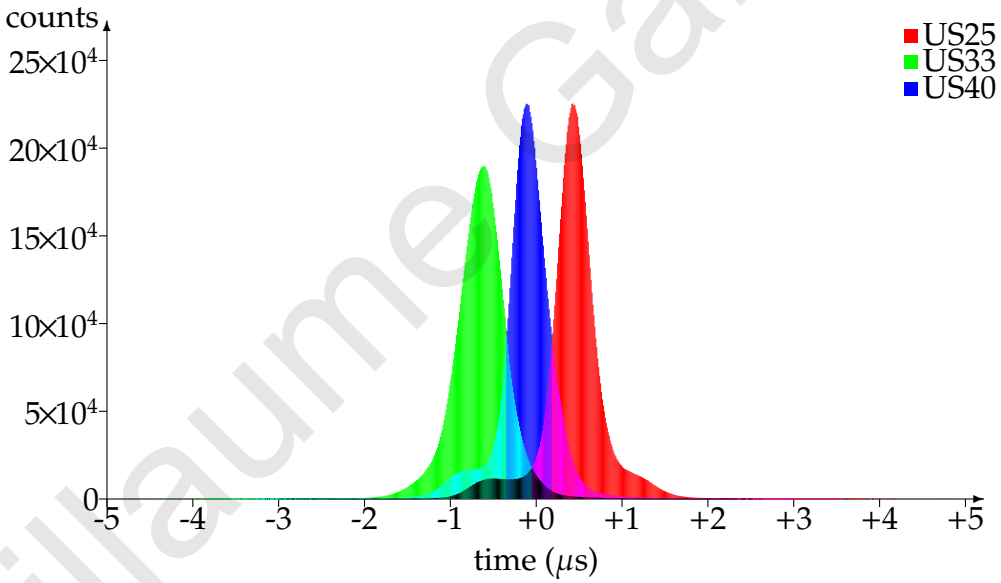


Figure 2.14: Histograms of the synchronization pulse delay on the 3 acquisition units for end-to-end measurement.

Unit	Minimum	Maximum	Mean	Std.
US25	−3.522μs	+4.756μs	+0.422μs	±0.380μs
US33	−4.444μs	+2.622μs	−0.650μs	±0.328μs
US40	−3.856μs	+3.911μs	−0.118μs	±0.339μs

Table 2.6: Statistics for the analog test.

Test	Source of error/delay	Solution
Digital	supply chain noise/mismatch	bypass sampling line
Analog	attenuator noise	moving attenuator from signal generator to input sampling line
Analog	sync pattern	optimized waveform
Analog & Digital	delay in cables	assumed negligeable
Analog & Digital	cross-talk	disable ADC/DAC
Analog & Digital	jitter noise	5 samples averaging
Analog & Digital	crystal drift	FM SYNC (what we are interested in measuring)

Table 2.7: Sources of error/delay and their solutions.

chain noise and jitter, although the line fit is an average of about 100 samples and each sample is the average of 5 ADC samples, so that should attenuate the effects of jitter and noise from the signal chain. Thus it was decided to do a digital sync test were a sync pattern would be directly connected to the FPGA in order to bypass the sampling lines. In Tab. 2.7 is given the different sources of error/delay and how they were fixed/diminished, it is also specified which test (analog/digital they are applied to).

2.3.3.2 Digital pattern synchronization test

In the digital pattern test, three DACQ units (GEO1, GEO2, GEO3) and the FM SYNC unit are connected to the same computer. The computer is used to initialize the timestamp, power the units and read out the data from the DACQ units. Additionally, the FM transmitter is programmed to generate a reference digital square wave (at 100kHz) that is synchronous to the transmitted wireless signal and which is connected directly to a spare pin of the FPGA on each DACQ unit. The setup is shown schematically in Fig. 2.15.

With this setup, the DACQ's FPGA can directly compare the synchronization pulse received wirelessly with the reference signal received digitally from the FM transmitter. The DACQ therefore records the clock cycle number at which a rising edge on the reference signal was observed. Since the clock cycle counter is phase-locked to the wireless FM signal, this measurement allows us to calculate the jitter and the absolute timing error for each DACQ unit. Note that in order to reduce the

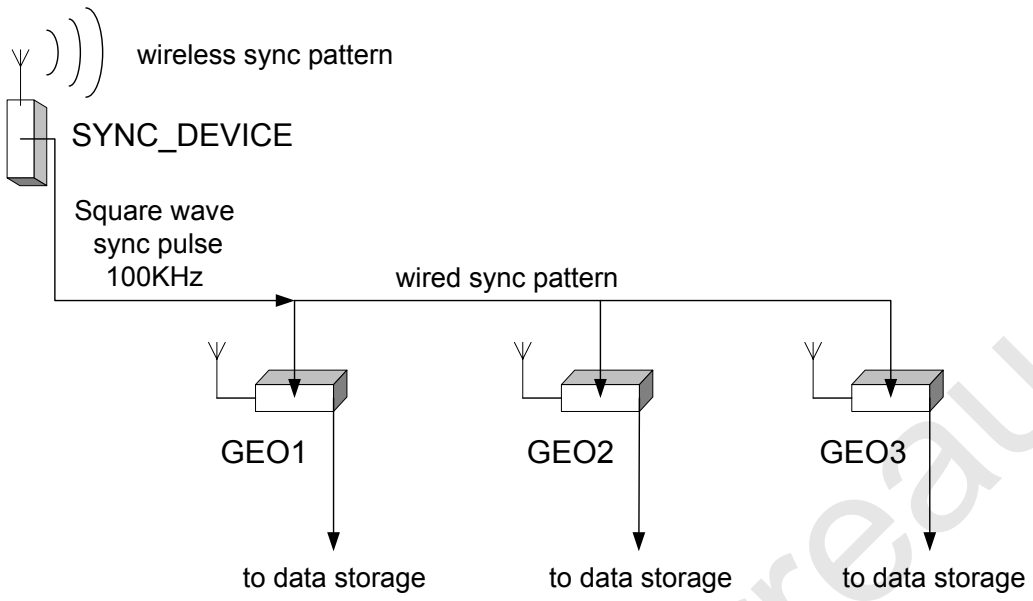


Figure 2.15: Schematic of the setup used for the digital synchronization precision measurement.

Unit	Mininimum	Maximum	Mean	Std.
GEO1	+1.244μs	+1.522μs	+1.378μs	±0.049μs
GEO2	+0.900μs	+1.167μs	+1.022μs	±0.047μs
GEO3	+2.500μs	+2.811μs	+2.630μs	±0.065μs

Table 2.8: Statistics for the direct digital test.

noise in the measurement (from digital lines cross-talk), the DACs and ADCs are disabled in this setup.

The histogram of these measurements collected continuously over a period of approximately 44 hours (approximately 16×10^9 measurements) are shown in Fig. 2.16, and the mean and standard deviation are summarized in Table 2.8. Note that in each case, the maximum spread of the measurement distribution is no greater than $\frac{1}{3}\mu$s, but each distribution has a distinct mean. Note that the standard deviation of the results is 7 times better than the one obtained in the analog synchronization test, this shows the high precision of the phase locking.

The reason for the distinct mean is that each DACQ unit implements a software PLL locked to the wireless FM signal. In a PLL, the phase error between the local

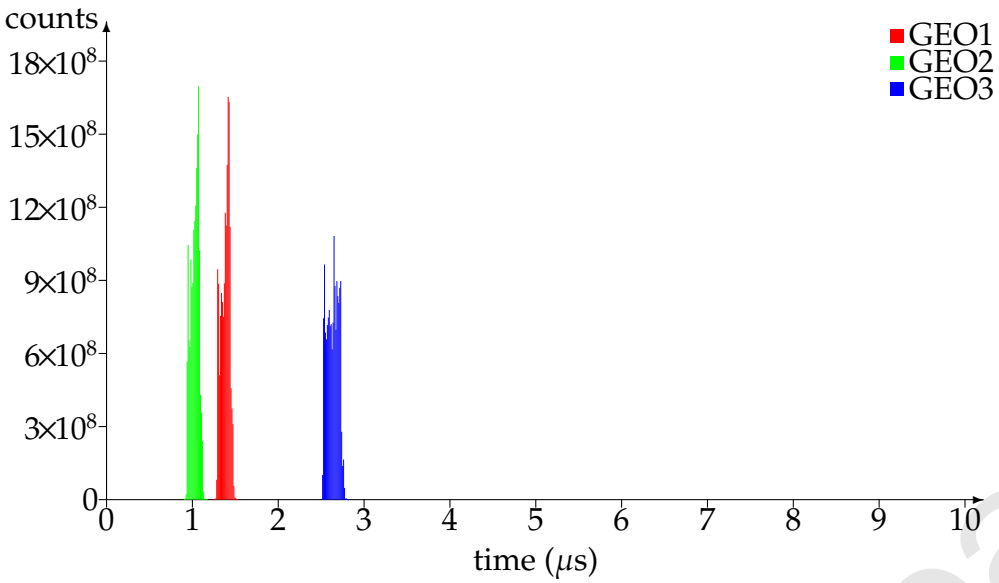


Figure 2.16: Histograms of the synchronization pulse delay on the 3 acquisition units for direct digital input.

oscillator and the external signal is used to drive the local oscillator frequency. Hence, unless the local oscillator has a natural frequency exactly equal to the external signal, a non-zero phase error will be required to drive the local oscillator to a matching frequency. Since each DACQ unit has its own local crystal oscillator module, the distinct mean is simply a reflection of the manufacturing variability in the actual frequencies of these modules.

These performances assure that the system can be used for synchronization with recordings of signals from the audio (up to 16kHz) and seismic range (in the <2kHz). For the ultrasonic range, though the working frequency is typically higher than that (40kHz or more, which means period of 25μs or less) the targeted information is spanned between 0 to 2kHz around it [96, 133, 255].

2.4 Conclusion

After a short introduction on the importance of precise timing in natural acoustic system, the state of the art synchronization systems were presented. Then the requirements for a bioinspired hardware platform for acoustic scene analysis were listed. The hardware platform is a custom distributed wireless data acquisition system for passive and active multimodal sensing. As commercially available synchronization modules have particular limitations (Tab. 2.9), the details of the design of a unique synchronization module (the FM SYNC unit) and a data acquisition module (DACQ unit) were given.

Solution	**Technology**	**Sync performance**	**Limitation**
NI cards	RTSI/PXI	10ns±500ps [129]	wired
LAN	PTP IEEE 1588	25μs±150ns [129]	wired
LAN	NTP	8.2ms±18ms [174]	sync >10μs
Wi-Fi	Wi-Fi	5ms±7ms [77]	sync > 10μs
RBS	Wi-Fi	6.29μs±6.45μs [76,77]	requires PBCH irregular sync period
GPS	GPS	35ns (2σ=340ns) [11]	outdoor only
This work	FMDASYNC	0.40μs±0.35μs	–

Table 2.9: Comparison of available solutions for synchronization.

The designed system relies on Very High Frequency (VHF) radio operated in the 72MHz band to achieve fine resolution in synchronization as well as functionality over distances exceeding 100m. It can be deployed indoors and outdoors. The system is capable of synchronized signal sampling with a precision of 1μs and an update rate of 2μs. This is demonstrated in two data acquisition tests, one using a digital synchronization signal and the other using a custom analog synchronization signal.

In the following chapters, the custom built system is used to collect data for various categorization tasks using the three frequency ranges defined in section 2.3.

Chapter 3

Scorpion-inspired LF Acoustic Wave Analysis

3.1 Introduction

In this chapter, an implementation of a scorpion-inspired neural network model is presented. The acoustic range of interest is the low frequency range, i.e. the range of frequencies below the human hearing threshold, range below 20Hz. Low-frequency energy traveling through the ground is usually produced by earthquakes (seismic activity) or explosions. Thus seismic sensors are used to detect these vibrations and predict future activity.

In nature, many animals use low frequencies for multiple purposes [246]. For communication purposes several species of whales use frequencies as low as 14Hz. They communicate over huge distances (1600km) [58,214] by vocalising at volumes that exceed 180dB with a reference at 1μPa at 1m [241]. These high powered broadcasts are used mostly to find a mate as it is not rare that the whale is the only one of its specie in an area with a radius of several hundreds of kilometres. On a smaller scale, elephants communicate up to 16km away [181] by stamping their legs on the ground to 'vocalise' and 'listen' for seismic vibrations through their legs, in the 10Hz-40Hz range. They not only communicate but are also able to identify precisely who is the individual they are talking to. Finally, on an even smaller scale, of the order of a few tens of centimetres, low-frequency vibrations are used by sand scorpions to find worms at night. They can detect variations in sand as small as 0.1nm and pinpoint the direction of their prey [30,218].

A recent trend in the marketing-related field is the personalisation of advertise-

ments according to the customer. This creates the need of a quick and efficient way to identify gender and even individuals in order to offer services that fit the needs and desires of the clients as much as possible. In the online world it is easy to provide such service via the use of cookies, which are stored by the web browser, and/or registered accounts, but this is more complicated in the 'real world'. Studies were conducted to offer cost effective systems to achieve this need. It has been shown that it is possible to identify gender and individuals from their footsteps (Sudo *et al.* [219], Ekimov and Sabatier [73]). It is also possible to envision applications in security (banks, home) or monitoring (toilettes in public facilities) without any privacy concerns. For example, an implementation of a footstep-based indoor location system using the traditional Japanese GETA sandals, equipped with force, ultrasonic, orientation, RFID sensors and an accelerometer to produce a wearable location tracking system that demands little infrastructure in the deployed environment is presented by Yeh *et al.* [251]. Another example is a system based on footstep vibration signals measured by a MEMS accelerometer for the estimation of indoor physical activity level for personal health care under smart home environments (Lee *et al.* [147]).

In the work presented here, a prototype spiking-neural-network localisation system is used, inspired by the way that sand-scorpions locate their prey. The system utilizes custom-built hardware, designed for accurate timing of concurrent data samples. Seismic sensors are used to detect the ground vibrations, which are fed to a spiking neural model, capable of working with real data.

3.2 Scorpion bioinspired model literature

A spiking neural model is used to calculate the subject direction from the ground vibration created by the footsteps. It is inspired from sand scorpions and how they locate their prey at night in the desert using the vibration traveling in the ground.

The first researchers to investigate the theory behind arachnid prey localization were Brownell and Farley [28–30]. Arachnids locate their prey through mechanoreceptors (sensilla) located at the tips of their eight legs. Each mechano-receptor can be modelled by a noisy leaky integrate-and-fire (I&F) neuron, which responds to vibrations with stimulus-locked action potentials, thus encoding the prey direction. When a sensilla receives an excitatory input the corresponding neuron will send an

inhibitory signal to the triads of neurons opposite it. The neurons with the strongest activity give the direction of the prey. This model was further developed by Stürzl *et al.* who showed that time-coding through spiking neurons is key to this process [218]. Brownell and van Hemmen then compared biological measurements to the theoretical model [31] mentioned above. Kim investigated different excitatory and inhibitory combinations (monad, dyad, pentad) [137], whilst Adams demonstrated, through simulations, the possibility of using this model in robotic applications [1]. Furthermore, the model of Stürzl *et al.* has been implemented on a python spiking neural network simulator called Brian (Brette and Goodman [26, 107, 108]), and tested using artificially generated data (Brette and Goodman [25, 107]). Finally, implementation of a simplified version of the model has been proposed for 6 legged robots by Wallander [236] and Kim [138].

The following section provides an overview of the original model and the improvements made at the University of Cyprus (UCY) for adaptation to real data.[1]

3.2.1 Original neural model

The model created by Stürzl *et al.* has 8 command neurons and 8 inhibitory neurons [218]. Each of the 8 legs of the arachnid is connected to one command neuron and three inhibitory partner neurons (shown in Fig. 3.1 in black and grey respectively). The sense organ of each leg of the arachnid, the basitarsal compound slit sensilla (BCSS), located just above the joint of the foot (tarsus) is compressed as a transverse wave passes along (Rayleigh waves, approximately 50m/s). The reaction mechanism is sensitive to movements smaller than 0.1nm and creates an excitatory spike at the command neuron corresponding to this leg. It also creates an excitatory spike at the 3 inhibitory partner neurons connected to the leg and projecting their inhibitory spike to the command neurons of the opposite legs (Fig. 3.1). Spike generation is modeled by an inhomogeneous Poisson process. Leg j makes an angle θ_j with respect to a reference (arbitrary selected as the head to tail axis of the scorpion) and spikes with a rate a_j. The leg closest to the source spikes at a higher rate whilst inhibiting others from spiking as fast. Thus we can compute the direction of the source. The polar position of the prey, $\mathbf{x}$, can be estimated from the angle, θ_j, of leg j (Eq. 3.1). The

[1] The initial work on the adaptation of the model was started by Eleni Proxenou as part of an "Undergraduate Research Opportunities Program" project under the direct supervision of Guillaume Garreau.

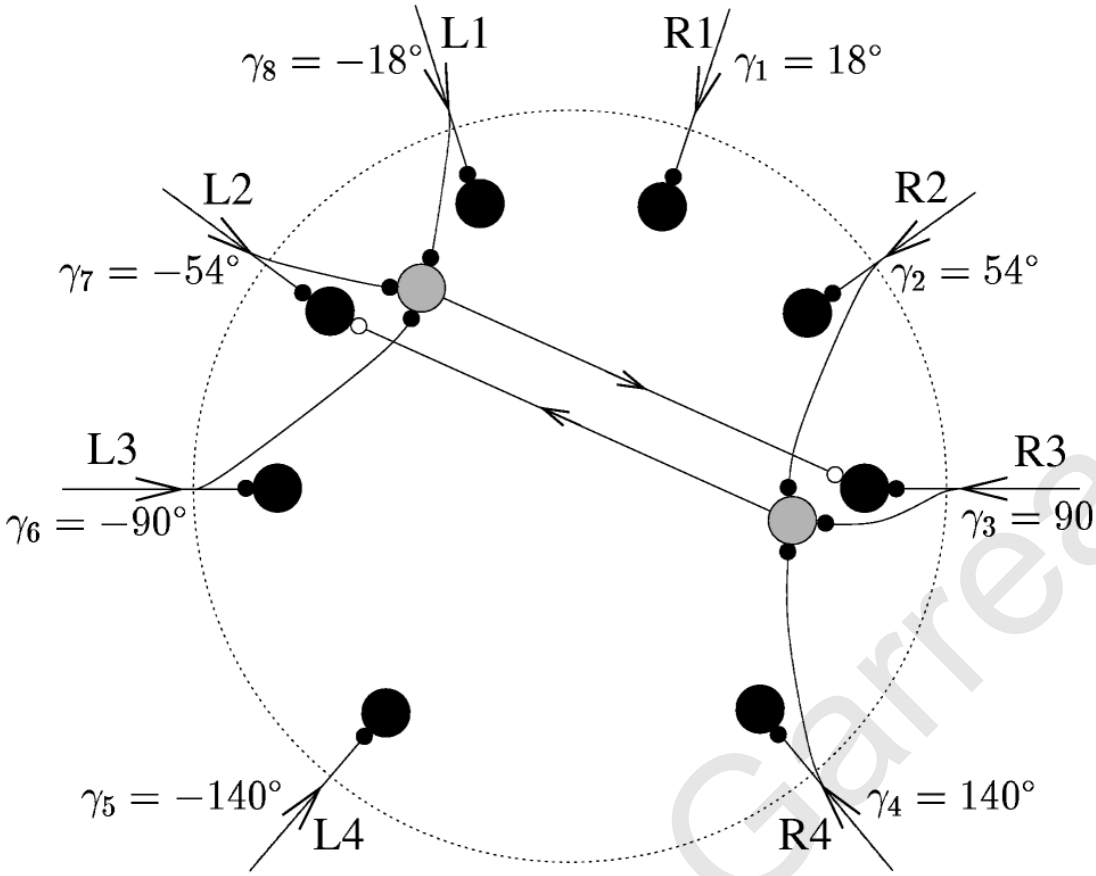


Figure 3.1: Original spiking neuron model developed for the sand-scorpion, from Stürzl *et al.* [218]. The eight command neurons are colored in black. For two of them, k = 3 and k = 7 = $\bar{3}$ corresponding to R3 and L2, respectively, the inhibitory partner neurons are shown as well in grey. The triad of R3 consists of L1, L2, and L3.

computed source angle ϕ can then be calculated from the polar position of the prey via Eq. 3.2.

$$\mathbf{x} = \sum_{j=1}^{8} \left(a_j * exp(i\theta_j) \right) \tag{3.1}$$

$$\phi = arg(\mathbf{x}) \tag{3.2}$$

This model has been implemented on a spiking neural simulator Brian using the following equations [25, 107]. The spiking rates at the leg mechanical receptors can be estimated as:

$$\frac{dv}{dt} = \frac{(1 + rates(t - \delta) - v)}{\tau_{legs}} + \sigma * \sqrt{\frac{2}{\tau_{legs}}} * xi \tag{3.3}$$

where v is the neuronal potential of the receptor neuron, t is the time, *rates* gives the value of the front wave at time $t - \delta$, δ is the wave delay between the legs, τ_{legs}=1ms, σ=0.01 and xi is the added white noise. The neuron activity is calculated

assuming a threshold equal to 1, a reset value (after spiking) of 0, and a refractory period of 1ms. In other words, when v reaches 1, a spike is generated and then v is reset to 0 and stays null for the next 1ms.

The spiking rates at the command neurons are estimated as::

$$\frac{dv}{dt} = \frac{(x - v)}{\tau} \tag{3.4}$$

$$\frac{dx}{dt} = \frac{(y - x)}{\tau_s} \tag{3.5}$$

$$\frac{dy}{dt} = \frac{-y}{\tau_s} \tag{3.6}$$

where v is the neuronal potential of the command neuron, t is the time, x is the ratio between excitatory and inhibitory synaptic conductances, and τ and τ_s are equal to 1ms. The neuron activity is calculated assuming a threshold equal to 1, a reset value (after spiking) of 0 and no refractory period. The excitatory synapses weight, w_{ex}, is equal to 7 and the inhibitory synapses weight, w_{inh}, is equal to -2. In addition, the inhibitory synapses have an inter-neuron synaptic delay, Δ_I, of 1ms. The post synaptic current (PSC) y is modeled as an alpha function.

The Python script used can be found in Appendix C. However, this implementation, by Brette *et al.*, has certain limitations. The main limitation is that it uses a single simulated Rayleigh wave that is fed to each leg with a certain delay, which depends on the location of source of the wave and the position of the leg. The same method was applied when noise was added to the simulated wave, thus exposed each leg to the same (but delayed) noise. This is different to the case of real data, where each sensor/leg has a different characteristic response and completely different noise contributions. Finally, the above model was not tested on moving sources.

3.2.2 Adaptations of the Brette implementation

First, several factors needed to be taken into account, in order to adapt the Brette implementation to our dataset.

Input Signals: The Brette implementation generates one simulated Rayleigh wave and add delays, calculated according to source location, for the different legs. The modified model is rewritten so as to accommodate the five unique input signals collected by each one of the seismic sensors.

Model	Mininimum	Maximum	Mean	Std.
Original (8 legs - triad)	42.1°	106.0°	71.7°	9.3°
Ours (5 legs - dyad)	40.1°	107.2°	72.1°	11.9°

Table 3.1: Test of the original Brian model with only 5 legs, prey angle is 68°.

Number of Legs and Inhibitory Partners: Given that we have only 5 seismic sensors at our disposal, the triad (three partners) inhibitory network is replaced by a dyad (two partners) instead to avoid creating an asymmetrical inhibitory system. The original spiking neuron network was modified and tested with only the number of legs changed and with a dyad inhibitory system. The modified model showed no difficulty in locating the source of the transversal wave (Table 3.1). The average was calculated on 200 simulation runs.

Moving Source: The code is designed for a fixed source, thus it was modified for a moving source, i.e a loop was added to run the simulation with a different set of data at each loop.

3.2.3 Real implementation issues of the Stürzl *et al.* Model

Additional adaptations had to be made to take into account the real implementation issues linked to the Stürzl *et al.* model.

In order to collect and process *real data* for location computation several key parameters needed to be taken into account, which were initially explored through a modified version of the Brette implementation. A noisy artificial wave source located at 68° was used, whilst different parameters were varied. In each case, 200 different simulation runs were performed, thus indicating the effect of key parameters on the reliability of the results. Furthermore, as shown in Fig. 3.2, different locations were used to run data collections. Those different trials helped to fully understand the model and the impact of various parameters on the final results. These are as follows:

Scorpion Radius: The size of the scorpion, i.e. the maximum delay that a Rayleigh wave will need to travel between two legs. This parameter is critical as it determines the refractory period of the neurons and the time constants of the equations that

control the neurons spiking activity. Similarly to the case of the ultrasonic system and the trade-off on sensing range and precision, which is mainly driven by the frequency used, the optimal spacing between the legs is linked to the wavelength of the sonic waves traveling in the ground with the relation:

$$\lambda = c/f \tag{3.7}$$

with λ the wavelength, c the speed of sound in the ground (depends on the type of material, cf Tab. 3.2) and f the frequency of the wave. It is usually advisable to use a $\lambda/4$ spacing in-between the sensors to get the best measurement possible. This is a similar problem to that encountered in choosing spacing of sensors in phased arrays (Harput and Bozkurt [115]). The original model uses sand and Rayleigh waves are measured at a velocity of 50m/s, a frequency of 250Hz and a radius of 2.5cm. For our setup, the velocity of the surface acoustic wave is measured using the time delay at each seismic sensor for the peaks created by the tapping in position 1 (Fig. 3.6) at the beginning of the recording. Several measurements in different environment were made to evaluate the speed of the sonic wave in the ground and evaluate its frequency in order to determine the optimal locations of the sensors. Tab. 3.3 shows the source of sound direction angle for an immobile source when assuming different velocity of the sonic wave and having different radius. For the data collection in the room with marble tiles, the wave velocity obtained is 315m/s. Thus, the optimal radius for placing the sensors was determined to be 20cm. This corresponds to a wave frequency of 197Hz. However, when doing the frequency analysis of the seismic data, the strongest energy response is found for a window of 3 to 4Hz (Fig. 3.3), which is expected in the case of multiple footsteps (Ekimov and Sabatier [73]). A footstep can be assimilated to a Dirac, i.e. many frequency are generated and most of them will be attenuated by the seismic sensors internal filters. The main reason of this discrepancy is that the seismic sensors have a frequency response range of 4.5 to 10Hz [55]. But this does not cancel the validity of the results obtained from the measurements. The theory would require a spacing of several meters between the seismic sensors (11.5m radius for a velocity of 315m/s and a frequency of 3.5Hz), however, Cauwenberghs *et al.* have shown that it is possible to do sub-wavelength beam-forming for source localization using array of sensors [44, 216, 217]. They used a technique named gradient flow, where time delays have been converted to relative amplitude by taking derivatives, thus allowing to use sensor arrays of

Material	Velocity surface acoustic wave
Sand	50m/s
Concrete	2500m/s
Brick	4170m/s
Wood	3300-3600m/s
Rubber	40-150m/s
Data collection ground	315.0m/s

Table 3.2: Velocity of surface acoustic wave in different material (adapted from [227]). For the experiment the value is calculated based on the data collected.

smaller dimension than the wavelength of the signal. Their work is based on studies of parasitoid flies made by Robert et al. [194, 195].

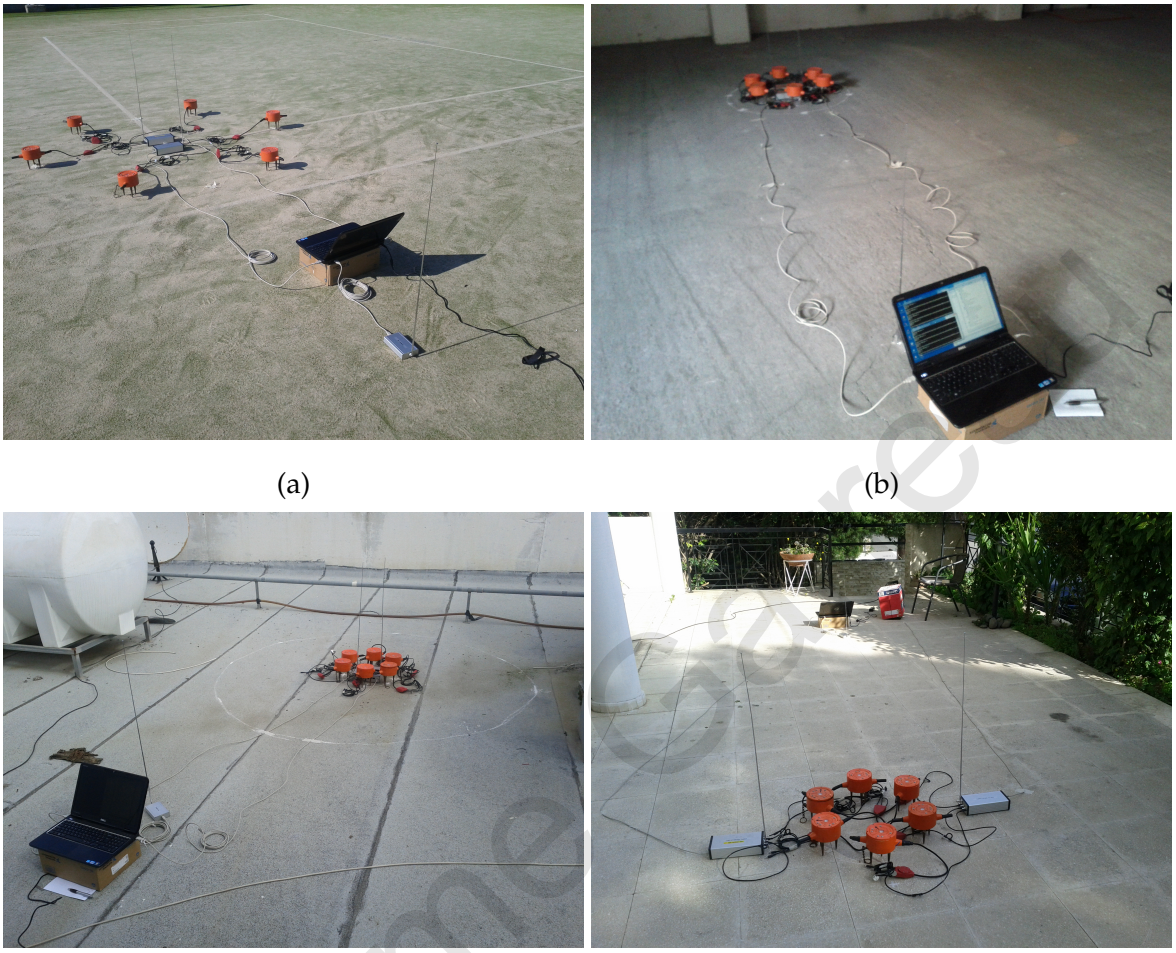
Sampling Frequency: The original model uses 10kHz sampling rate. Our custom made hardware has a sampling rate of 33kHz, which was then downsampled by a ratio of 10 to speed up calculations. The sampling rate parameter did not change the performance of the model (Table 3.4).

Additional features: A number of parameters were kept as in the Brette implementation: a) the refractory index was set to equal the maximum interleg delay. This insures that a neuron does not fire multiple times before the acoustic wavefront reaches the opposite side of the neural network. It would inhibit any spiking activity in the other neurons of the network, which would disturb the correct behavior of the model. b) the inter-neurons delay was kept at 70% of the maximum interleg delay. This is to simulate the intra neuron delay and the time-delay needed for a spike to travel from one neuron to the next. c) the excitatory and inhibitory weights were kept at 7 and -2 respectively.

3.3 Data collection

3.3.1 Data collection setup

In this section a physical system is presented, which builds upon the model of Stürzl *et al.* [218] in order to work with real data in real time, via the addition of the ability to



(a) (b)

(c) (d)

(e)

Figure 3.2: Pictures of the acquisition setup in various locations: (a) tennis court, (b) parking lot, (c) building roof, (d) terrace, (e) university hall.

Radius	Wave velocity	Mininimum	Maximum	Mean	Std.
2.5cm	50m/s	42.1°	106.0°	71.7°	9.3°
5cm	50m/s	1.5°	244.9°	64.5°	64.5°
7.5cm	50m/s	3.3°	247.6°	73.2°	83.3°
10cm	50m/s	1.4°	245.0°	63.1°	97.3°
2.5cm	100m/s	60.0°	75.7°	67.2°	2.5°
5cm	100m/s	45.8°	92.8°	72.0°	8.6°
7.5cm	100m/s	9.1°	247.5°	71.2°	82.1°
10cm	100m/s	3.8°	237.7°	78.1.1°	69.7°
2.5cm	200m/s	53.6°	77.2°	67.2°	4.3°
5cm	200m/s	61.5°	74.0°	67.5°	2.5°
7.5cm	200m/s	56.3°	75.9°	67.4°	3.5°
10cm	200m/s	46.4°	93.0°	71.2°	8.8°
15cm	200m/s	7.3°	247.7°	65.9°	74.7°
20cm	200m/s	3.3°	243.5°	63.2°	64.0°

Table 3.3: Average result of the model according to velocity of surface acoustic wave and radius, prey angle is 68°.

Model	Mininimum	Maximum	Mean	Std.
Original (1e4Hz)	42.1°	106.0°	71.7°	9.3°
Ours ((1e5/3)Hz)	21.7°	97.5°	67.7°	11.0°
Ours ((1e5/30)Hz)	46.9°	87.0°	67.1°	7.1°

Table 3.4: Test of the original Brian model with different frequency, prey angle is 68°.

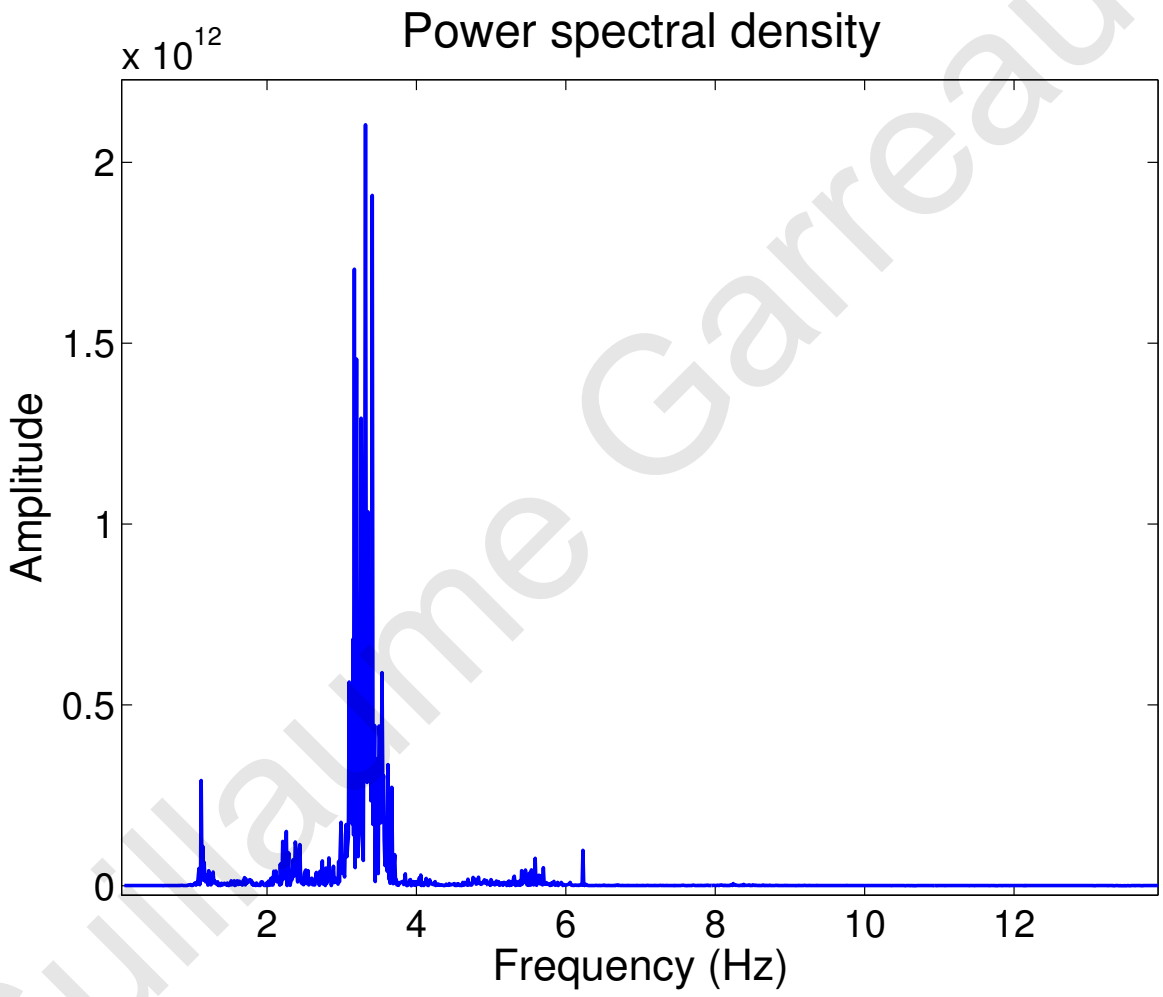


Figure 3.3: Power density of the seismic signal.

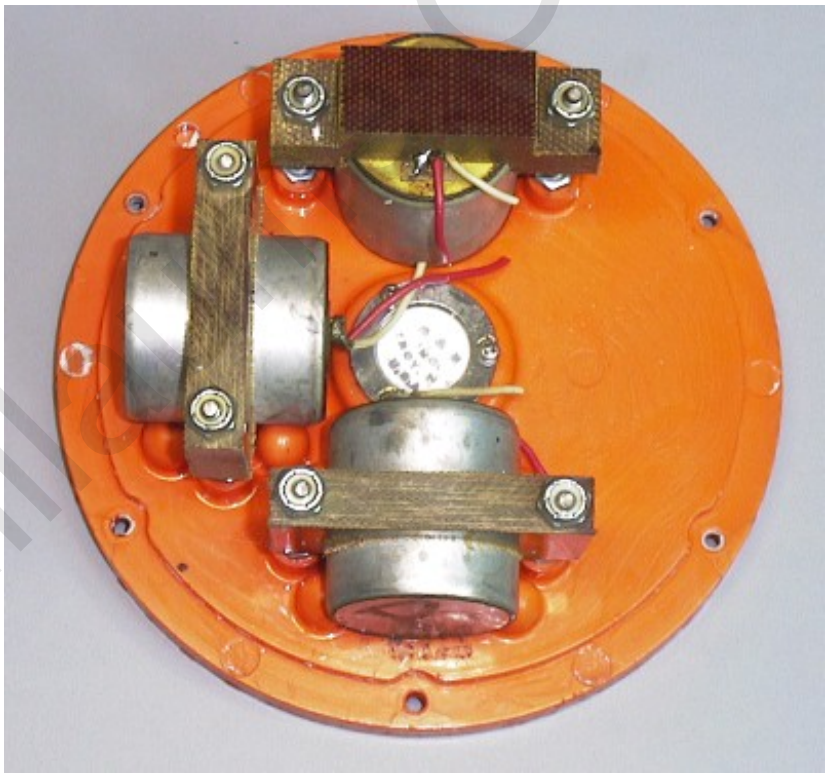
process vibrations produced by a person walking on circle on a marble floor. To the best of my knowledge this is the first time that such a system has been implemented. The vibration data was collected from five seismic sensors (Mark Product L-15B 4.5hz x,y,z geophones, Fig. 3.4) positioned at the center of the circle. To ensure correct timing the sensors were connected to the acoustic data acquisition system (presented in the Chapter 2), which time-stamped the incoming data. The data was fed to the modified version of the spiking neuron model, which determined the direction of the subject.

The acquisition setup is based on three of the custom-built units described previously (section 2.3): one that is used for the synchronization (FM SYNC unit) and the other two for the data acquisition (DACQ unit). The setup, which is shown in Fig. 3.5, consisted of five seismic sensors (Mark Product L-15B 4.5hz x,y,z geophones, marked 1 to 5 on Fig. 3.6), two DACQ units (marked GEO1 & GEO2 on Fig. 3.6), a laptop for data storage and processing and the synchronization device (marked as sync in Fig. 3.6). The five seismic sensors were positioned on a 20cm radius circle with 72° between each sensor, on a marble floor in a large room. The area was not protected from external vibrations and was less than 50m from a busy road. The subject walked on a circle of radius 1.2m. The subject was asked to start from position 1 (marked pos1 in Fig. 3.6, aligned with sensor 1) and tapped the ground 10 times with his/her heel before starting to walk. The tapping data was used to calculated the velocity of the surface acoustic wave in the ground. The subject proceeded to walk two circles counter-clockwise, and then turned back to walk two additional circles clockwise.

3.3.2 Data processing

The data recorded by the seismic sensor network is first pre-processed before being fed into the model. Firstly, the data is aligned by using the timestamps placed in the data by the custom-built data acquisition system (presented in the Chapter 2). Secondly, the data stamps are removed from the data. Thirdly, the data is segmented into smaller windows in order to reduce the data file size to be loaded in the memory for processing. Then, for each sensor, the signal is normalised by removing the mean and dividing by the standard deviation of the data (removes DC offsets and gain mismatch). The normalization was done assuming that the amplitude of the actual

(a)



(b)

Figure 3.4: Geophone Mark Product L-15B 4.5hz x,y,z (a) with inside view (b).

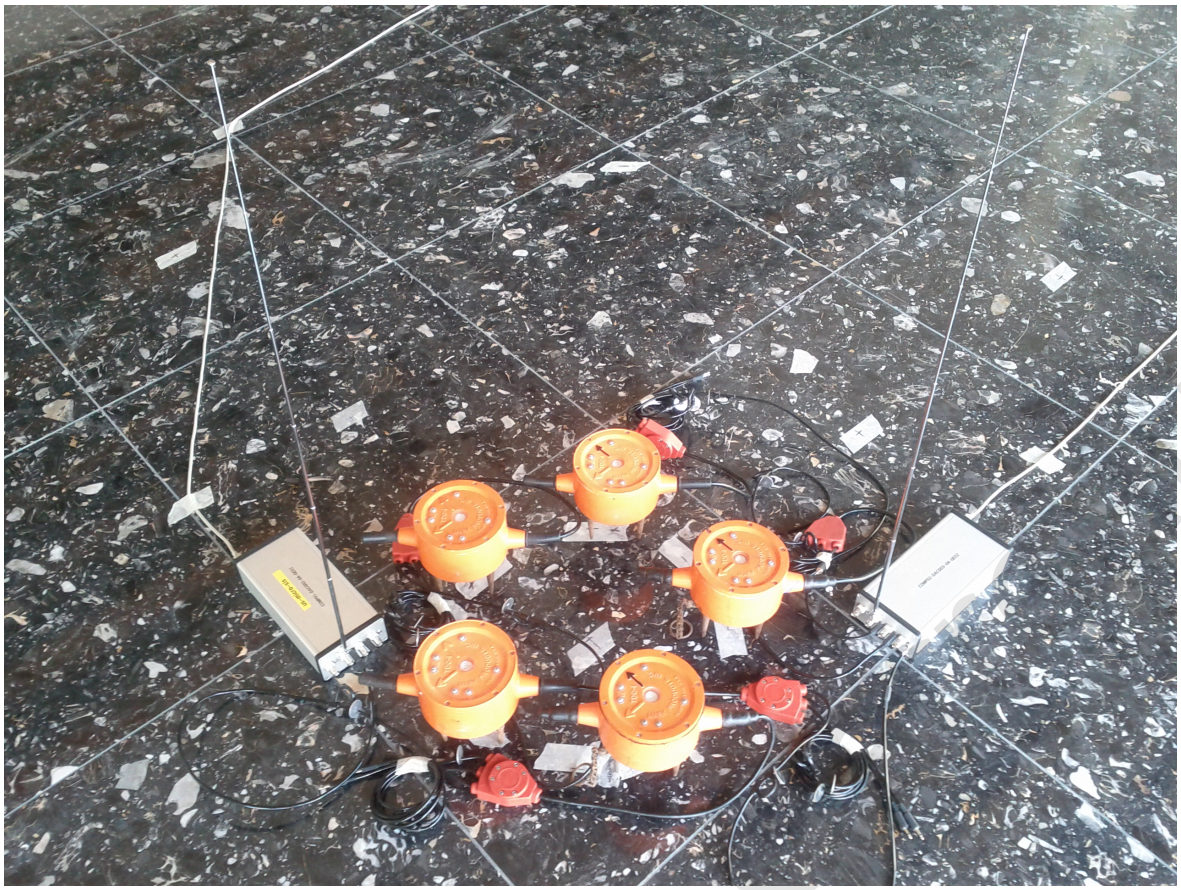


Figure 3.5: Picture of the data acquisition setup.

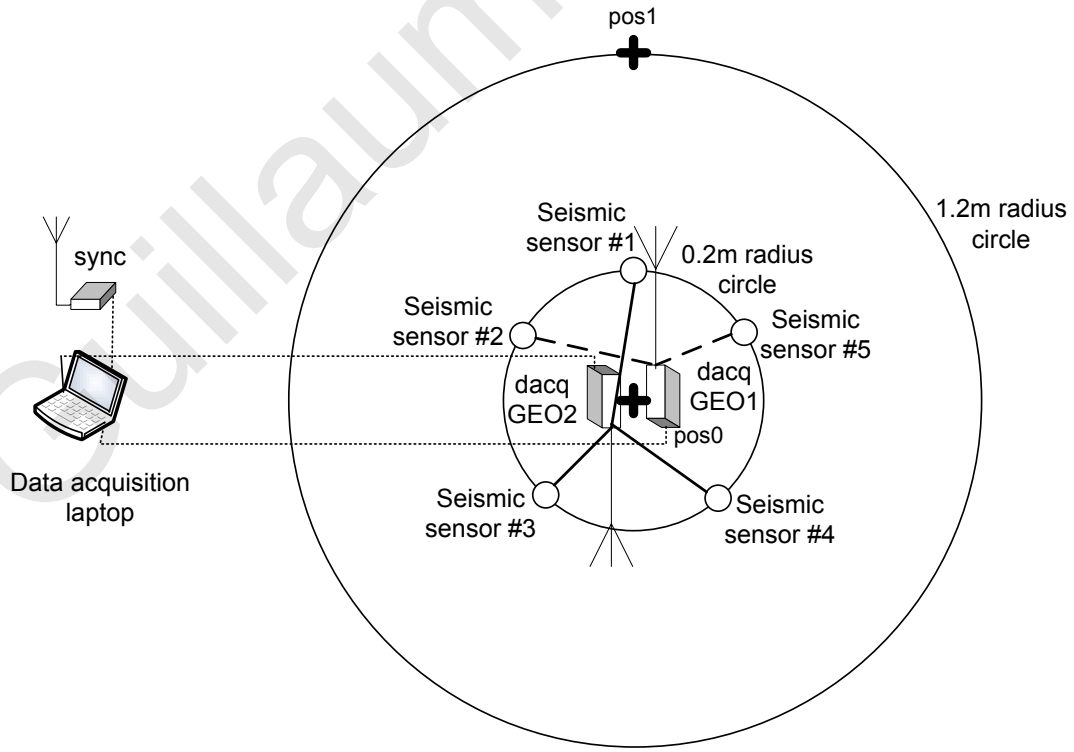


Figure 3.6: Schematic of the data acquisition setup.

sonic wave does not decay when passing from one leg to the other. This is true to a certain extend, as in Nature, scorpions use this to determine the exact location of the prey (direction and distance), but they can do that only up to short distances, further away only direction is perceived [30]. Several trials of normalization were applied to the data sets (removing mean value and divide by max/min amplitude, normalization to 1, and combinations of using those different parameters) but none of these attempts gave satisfactory results. This is due to the fact that a minimum amount of similarity between the different data threads is necessary in order to get positive results. That is why using matched response sensors and amplifiers would improve the results. Perfect matching is not necessary, as it has been shown that not only neural networks are quite resistant to noise (as seen here in the different simulations) but noise could be 'used' to perform optimal performance at lower computational and energetic cost (Salinas [201]). The processing is performed with Matlab®2012. Finally, the processed signal is fed to the model, each sensor matching one leg. The scorpion inspired model is implemented in the Brian spiking neuron network simulator [25,108] with Spyder2© on Python2.7®. The Python script used can be found in Appendix C.

3.3.3 Data collection results

By feeding a modified spiking-neural-network scorpion model with normalized raw data, it is possible to get a reasonable estimation of the subject location, as shown in Fig. 3.7. In that case, the velocity of the subject is assumed constant for each circle. In the first 10 seconds the subject is stationary at about 0° (pos1 in Fig. 3.6). The subject then walks along the circle twice with increasing angle values, turns back and walks two circles in the opposite direction. As mentioned before, no special precautions were taken to avoid unwanted external vibrations from reaching the system, which may explain the larger deviations in the computed position. As well if the subject does a short pause, then the calculated position is based only on the external vibrations. It should be noted that the response of the seismic sensors is not perfectly matched in frequency and gain. Also the component properties used in the signal chain of the acquisition boxes are susceptible to manufacturing variations and consequently also introduce mismatch, thus contributing to errors in angle estimation. The impact of such mismatches on the performance of the sound

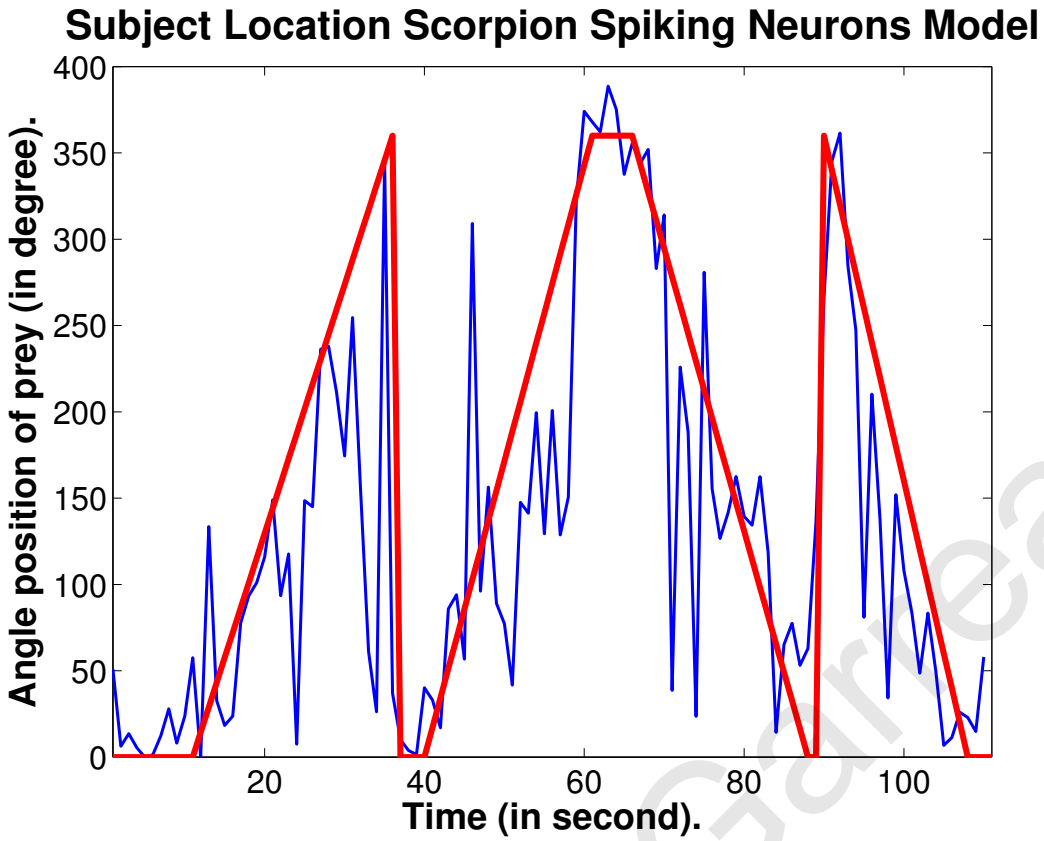


Figure 3.7: Plot of the subject position on the circle (in red actual position assuming constant subject velocity, in blue Scorpion Neural Network (SNN) model result).

localization in air has been discussed by Julian *et al.* [132]. In nature, the scorpion has a 13° to 15° error on the localization of its prey, an error which is compensated by the size of the pedipalps [30].

In order to evaluate the performance of the model, a ±13/15° error margin was used. On the raw data, see Table 3.5, the performance obtained was very low, less than a third of the data points are in the acceptable error range. However, considering a time delay of 1s (equivalent to 1 time window) in the system response, then 50% of the data points are within the acceptable margin. Furthermore, if a wider margin is used, which is acceptable considering we have 5 legs instead of 8, for a error margin of 24°, then a performance of 60% is obtained. 24° is chosen as with 5 legs, the distance between the legs is 72°, to compare with the 45° of a 8 legs system (which means 15° is an error equal to one third of the interleg distance).

These results are obtained when considering constant velocity of the subjects walking on the circles. If during the data collection, the real trajectory would be measured (camera based or ultrasonic sonar system) then one could expect greater precision and performance.

error margin (°)	raw data (%)	with 1point/1s delay window (%)
13	24	45
15	32	50
24	40	60

Table 3.5: Table showing the error analysis of the SNN model result for the raw dataset. In Nature, a 13° to 15° error on the direction of the source is observed [30].

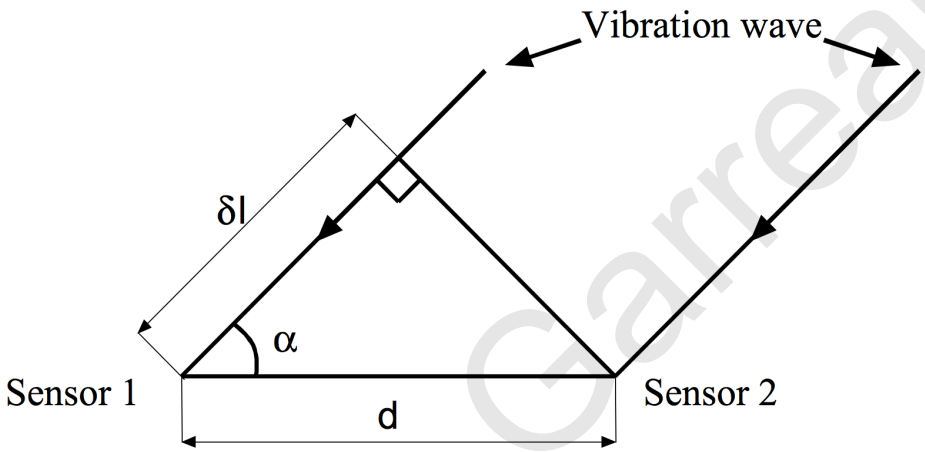


Figure 3.8: Definition of the Time of Arrival (ToA) calculation, time that the vibration wave needs to travel the distance δl.

3.3.4 Comparison with a time of arrival approach

Given that the problem seems to be a classical time of arrival (ToA, Fig 3.8) process, the use of a neural network implementation does not seem necessary. In terms of performance, the results reported by Wallander *et al.* [236], using a ToA approach with a 6 legged robot, are 75% of the point within 20° (one third of the interleg angle) of the correct direction for a source at 15cm from the nearest leg, the standard deviation is 46.7 and goes up to 60.9 when the source is at 30cm. This is a 30% increase of error with a doubling of the distance. One point is calculated every 500ms. For comparison, as shown in Table 3.5, our system reaches 60% of the points within 24° at 1m (almost 7 times further away). Furthermore, the use of a neural network allows to extend the implementation to more complex application while keeping the same computational infrastructure. For example, using an array of microphones, additional layers of neurons are capable of identification of acoustic object while the first layer is used only for finding the direction of the source of sound.

3.4 Conclusion

This chapter presented a system that performs person localization through 5 closely spaced seismic sensors and a bioinspired spiking neuron model, that is capable of providing an additional information source for performing acoustic scene analysis. This is the first time where a real working system using the SNN has been shown to track a vibration source. In addition to the different improvements detailed in section 3.2, the data collection on the field gave the opportunity to discover the link between the size of the artificial scorpion and the properties of the environment. It would be interesting to see if this can be matched with observation of the size of real scorpions in their natural environment or if scorpions would change their leg geometry according to changes in environment properties. The noise in the calculated location could be improved by using matched Micro Electro-Mechanical Systems (MEMS) based ground vibration sensors.

Furthermore, as an extension of this work, it seems to be possible to design a low-cost arachnoid robot that would have the ability to follow a 'prey'. This could be done using 3D-printed parts as the T8 of Robugtix™ (`http://www.robugtix.com/t8/`) and an FPGA. By giving reconfigurable abilities to the robot, one can think of having software updates for more efficient algorithm implementations or new and more complex behaviors. The T8 is basically a radio-commanded spider (Fig. 3.9). For example after an earthquake, a search and rescue team could use such a spider robot to explore a destroyed building and find the location of persons trapped inside. The robot would use acoustic information generated from the victim to locate and navigate to the victim. A thermal imager could be added to use body heat to locate someone and a camera and microphone would allow communication between the rescue team and the victim. Such system could also be used as a kit for children to experience engineering in a fun way. In a high security area, they could be used for surveillance, by wandering in a building or outdoors, as shown in [73, 147, 219, 251]. Footsteps can be used to locate, track, identify a person and characterize their activity.

Figure 3.9: Picture of the arachnid robot T8 of Robugtix^{TM}.

Chapter 4

Ultrasonic Range Acoustic Scene Analysis

4.1 Introduction

In nature ultrasounds have been used by some animals for millions of years to locate and identify a prey and also to communicate. Well known examples are bats and dolphins, but different species of birds, mice, fish or insects also use ultrasounds. Ultrasounds are pressure waves in the 18khz to 200kHz frequency range and are outside the audible human range. Sound waves are reflected by objects in the environment and echoes are analyzed to extract useful information such as position or identification of a prey. The principle behind this is the reflection of sound waves by objects and the Doppler effect.

4.1.1 Doppler effect

The Doppler effect is named after the Austrian physicist Christian Doppler, who discovered the effect in 1842 [66]. He discovered that sound waves have a higher frequency when the source is moving toward the observer and a lower frequency when the source is moving away from the observer. He tested this idea by standing next to a rail track while a train full of musicians passed by the rail track. This test confirmed that sound pitch is higher when the sound source is approaching, and lower when the sound source moves away. The same is true with sirens on police cars and the engines of race cars. One way to visualize the Doppler effect is to think of sound waves as pulses emitted at regular intervals, Fig. 4.1. Imagine that each

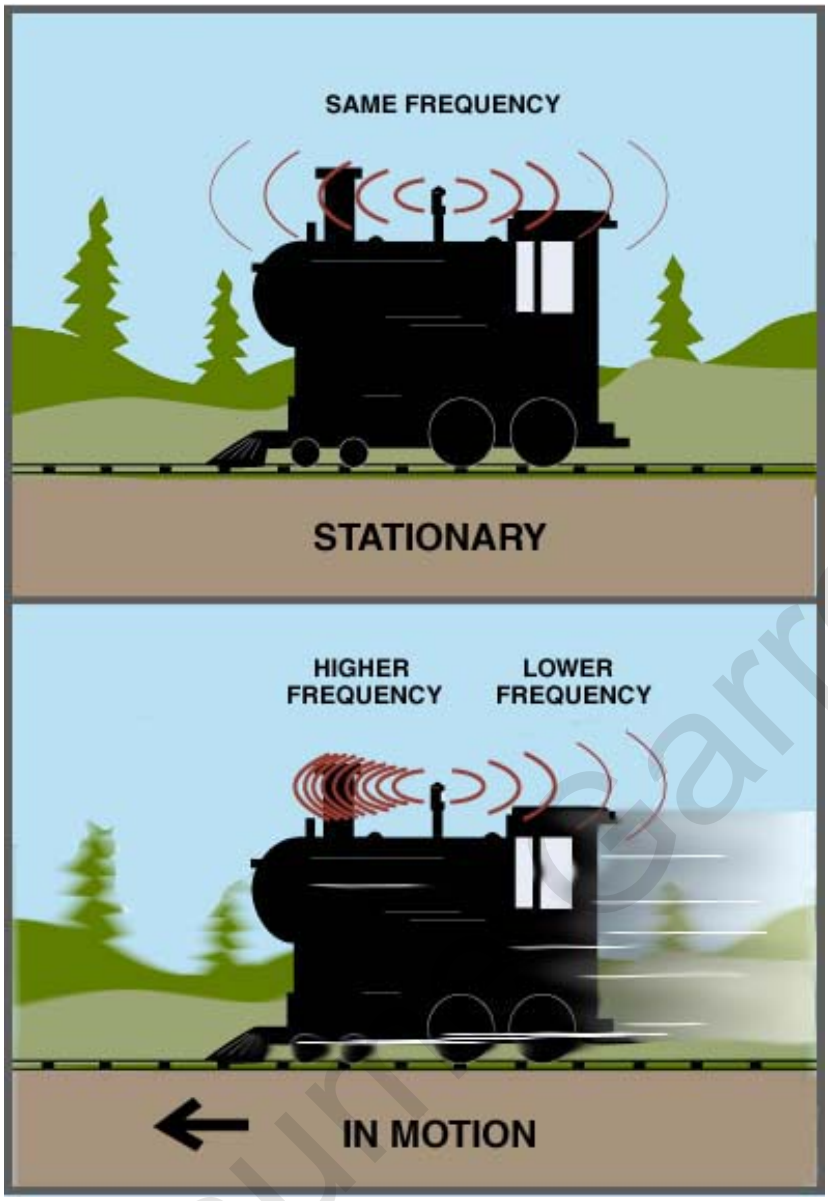


Figure 4.1: Doppler effect: sound waves that move toward you are compressed while those moving away are stretched [116].

time you take a step, you emit a pulse. Each pulse in front of you would be a step closer than if you were standing still and each pulse behind you would be a step further away. In other words, the frequency of the pulses in front of you is higher and the frequency of the pulses behind you is lower. The Doppler effect does not just apply to sound. It applies to all types of waves, including light. It is widely used in astronomy to study the speed of stars and galaxies.

The Doppler effect can be modeled by a simple equation:

$$f = \frac{v + v_r}{v + v_s} * f_0 \tag{4.1}$$

where v is the velocity of waves in the medium; v_r is the velocity of the receiver

relative to the medium, and is positive if the receiver is moving towards the source; v_s is the velocity of the source relative to the medium, and is positive if the source is moving away from the receiver; f_0 is the frequency of the source and f is the observed frequency.

Here, we have a stationary source and moving receiver, the equation can be simplified as:

$$f = f_0 * \left(1 + \frac{v_r}{c}\right) \tag{4.2}$$

with the same notations as in Eq. 4.1 and c is the speed of sound in air. It is assumed that the receiver is reflecting the sound wave without changing its frequency, thus a linear relation between the speed of motion of the receiver and the variation of wave frequency is obtained:

$$\Delta f = f - f_0 = f_0 * \frac{v_r}{c} \tag{4.3}$$

4.1.2 A little bit of history

Human technologies using this principle are quite recent, despite the first report of using sounds travelling in water to detect approaching boats dates back to Leonardo Da Vinci in 1490 (Fahy and Walker [81]), who reported using a tube inserted into the water to listen for, and detect, approaching vessels. It was not until the 19th century that another application was reported, when an underwater bell was used as an ancillary to lighthouses to provide warning of hazards. The first patents for underwater echo location system were filed (Hill [120]) in 1912 at the British Patent Office by the English meteorologist Lewis Richardson, and in 1913 by the German physicist Alexander Behm. It is described as the "use of sound to 'echo locate' underwater in the same way as bats use sound for aerial navigation" [120] and it is believed to have been prompted by the recent tragedy of the Titanic disaster a few months earlier. The need to detect submarines during World War I led to the discovery and development of hydrophones and, later on, the SONAR; SONAR is an acronym for SOund Navigation And Ranging. Sonar uses sound propagation (usually underwater, as in submarine navigation) to navigate, communicate with (underwater Morse code [120]) or detect objects on or under the surface of the water, such as other boats and submarines. In 1915, the French physicist Paul Langevin and the Russian electrical engineer Constantin Chilowski worked on the development of active sound devices for detecting submarines using quartz.

Sonar is classed in two types of technologies: passive sonar is essentially listening for sound made by vessels; active sonar is emitting pulses of sounds and listening for echoes. The acoustic frequencies used in sonar systems vary from very low (infrasonic) to extremely high (ultrasonic). In recent years the major military development led to an increasing interest in low frequency active systems. Sonar is also used in biomedical application, and more specifically in bio-imaging, such as diagnostic sonography (or ultrasonography), a technique used for visualizing subcutaneous body structures including tendons, muscles, joints, blood vessels and internal organs for possible pathology or lesions. Obstetric sonography (or echography) is commonly used during pregnancy. The frequencies used in diagnostic ultrasound are typically between 2 and 18MHz. The work presented here is using a frequency of 40KHz and traveling in air, as opposed to traveling through liquids and body tissue.

4.1.3 Previous work with ultrasound

In any physical human action, several body parts, whose loci are constrained with respect to one another, move in a coordinated manner in order to generate the action. The velocity of each of these moving parts can be used to modulate and reflect incoming waves. Hence, the micro-Doppler (mD) signature is linked both to a particular action, as well as to the particular physical constraints of an individual's body part (related to size, flexibility, coordination etc.).

In recent years the use of mD gait signatures has been growingly reported in various safety and surveillance applications where the ability to identify an individual quickly and accurately is critical. Some examples are objects localization and identification [33,34,141], passability through a door [126], presence detection [101,184,200], presence and tracking [117,191], individual identification [8,68,133,134,161,172,255], gender classification [133], behavior/actions classification [32,82,135,139,140] or aged persons safekeeping [153,250].

A number of features set the different systems apart, such as the transmission mode of the wave, the type of wave used for micro-Doppler (acoustic or electromagnetic) and the data analysis methods.

The radar and synthetic aperture radar (SAR) studies focus on electro-magnetic (EM) wave radars and the MHz and GHz frequency range. There are two modes

of transmission: the continuous wave (CW) sonar RF at 10.5GHz [101, 184] and the Pulse Doppler RF [153, 250] with a carrier at 5.8GHz and pulse modulated at a lower frequency. However, there is an increasing interest in exploration of the acoustic range (20kHz to 100kHz) and the use of a CW sonar at a fixed frequency (40kHz [9, 133–135, 255–257] or 80kHz [8]) or chirps, sweeping frequencies from 25kHz to 100kHz [68, 172].

Most of mD analysis focuses on time-frequency features [101, 200, 255] and starts with a Fast Fourier Transform (FFT) or a Short-Time Fourier Transform (STFT) to process the data and then a classifier is used for pattern recognition. Various classifiers have been used: linear [184], neural network [68], Bayesian [133, 134], Gaussian Mixture Model (GMM, [135]), Maximum Correlation Coefficient (MCC, [172]), Support Vector Machine (SVM, [153]), and k-Nearest Neighbour (kNN, [153]).

This work focuses on the use of a module, the ultrasonic frequency transceiver (UFTRX), that creates a continuous ultrasonic wave at 40kHz, records the amplitude of the echo that is processed and displayed. This module is a version of the DACQ unit presented in Chapter 2, configured specifically for transmitting and receiving ultrasound. It was successfully used to: distinguish different means of transport for human beings (pedestrians, inline skaters and cyclists) based on their mD time-frequency signatures with accuracies as high as 97% [97]; to identify an individual, with accuracy as high as 100% for some individuals [96,98], recognise an individual's gender with accuracy as high as 92% [96]; and to identify different actions performed by the individual [70].

4.2 Bioinspired model literature

4.2.1 Evidence of spectro-temporal brain processes

A computational model of auditory object processing based on spectro-temporal features, is described by Kumar *et al.* [143], a study that employed Dynamic Causal Modeling (DCM) to evaluate the functional connectivity between three different regions of the auditory system. It gives convergent evidence that some of the principles of auditory processing in the cortex make use of spectrotemporal features and a hierarchical organization. Regarding mD sonar processing, evidence suggests that the auditory system in bats is also organized hierarchically (Suga [220]). Similar

time-frequency and hierarchical organization is found in dolphins (Cauwenberghs *et al.* [46]). Even though echolocation is not a common human ability, some examples have been reported of early and even late blind people developing the ability to navigate and precisely identify their environment through echolocation via mouth clicks and listening to the returning echoes [67]. A neuroimaging study showed that processing of click-echoes recruits brain regions typically devoted to vision rather than audition in both early and late blind echolocation experts (Thaler *et al.* [224]). In this section the algorithms that have been investigated in this thesis, which are based on spectro-temporal processes, are presented [69, 70, 96–98].

4.2.2 Mode of transport classification

Initial investigations involved the discrimination between different mode of transport using a linear classifier and features calculated as the standard deviation of the energy spread on a spectrogram of the echo of the sound wave [97].

The important of mode of transport classification is evident considering that pedestrian-vehicle collisions represent 15% of the deaths in road accidents in developed countries, and up to 80% in the developing countries. This corresponds to 1.2 million deaths and 50 million injuries annually according to the World Health Organization [187]. Prevention of such collisions has been the focus of a number of studies recently [94, 162]. Furthermore cyclists or inline skaters sometimes ignore legislation and use pavements instead of their reserved lanes, consequently leading to injuries every year [119]. A potential way of reducing the number of injuries is to have dedicated lanes for each mode of transport and to make sure that no violations are incurred. If most of the systems used to police traffic segmentation on the road, utilize one or more visible light cameras [111, 205, 248, 252], infra-red technologies [15,83] or cellular technology [60]; However, existing technologies have some drawbacks as explained in Chapter 1.

The physical differences between these modes of transport can be used to explain particular characteristics that are key for discriminating between the different signatures. During running and walking, if the torso moves roughly at speed v, then the speed of the legs ranges between 0 (foot on floor) and an average forward recovery speed of $2v$. In addition the arms swing to counteract the leg motion. During cycling or inline skating, the subject torso is sliding on wheels with a velocity V, whilst the

leg cyclic movements are slower in relation to the absolute torso speed. The arm motion during inline skating can be used to differentiate between inline skating and cycling, in addition to the more pronounced leg movements of skating. The different body part speeds are captured in the spectrogram according to Eq. 4.4.

$$V_T = -V_S * (F - F_{mD})/F_{mD} \tag{4.4}$$

where V_S is the speed of sound (348 m/s), F is the average of the frequency components (Hz) and F_{mD} is the frequency of the UFTRX module (40kHz). Visual examination of the obtained spectrograms (Fig. 4.5), reveals that each mode of transport is characterized by different spread of energy across both time and frequency. This spread of energy can be estimated as the standard deviation of the spectral power across all frequencies at a given point in time with respect to the torso velocity. As the torso is much larger than the limbs one can expect that at any point in time, the frequency component with the maximum amplitude will correspond to the torso velocity, hence one can find the maximum value in each column (after smoothing) and store the corresponding frequency. The average value of these frequencies over a short time frame can accurately determine the torso velocity. Once the torso velocity is found one can then proceed to calculate the amplitude-weighted standard deviation of the constituent frequency components at each point in time. In the case of modes with pronounced limb motions, such as running, the spread of energy around the torso will be larger. Thus, one can expect a larger standard deviation for modes with significant limb motions and lower standard deviation for the modes with small limb motions, such as cycling.

4.2.3 Gender and individual recognition

In [96], the mD signatures of walking subjects were analyzed in order to perform gender and individual recognition based on well-known algorithms, such as FFT, time and frequency averaging, k-means clustering and nearest neighbour classification. Individual recognition has obvious application in security. Gender classification can be of interest for security purposes at single sex facilities, e.g. at a gym or bathrooms. Furthermore this could be useful for dynamic marketing on flat screen displays, e.g in a shop, for the personalisation of advertisements according to the customer. This creates the need of a quick and efficient way to identify gender and even individuals in order to offer services that fit the needs and desires of the clients as much as

possible. Another case is when statistics are created in order to learn more about clients, customers or users of public or private services.

4.2.4 Action classification

Additional investigations involved the discrimination between different set of actions [70]. These included walking (unknown speed), walking slowly (3km/h), walking fast (6km/h) , running (unknown speed), running slowly (9km/h), running fast (12km/h), inline skating, slow cycling (approximately 11km/h), fast cycling (approximately 22km/h), clapping hands, calling"help me" while moving their arms up in the air and, calling"come her" while beckoning with their right arm.

4.2.5 Individual recognition with AR model

Whilst there are many algorithms that could have been tested for individual recognition, such as the spectrogram-based methods used above, we decided to investigate the use of the autoregressive model (AR) and its time-varying coefficients, since these coefficients can be linked to the changing frequency content of the micro-Doppler signatures [3]. AR models are linear filters whereby current samples of a time series are modelled as a weighted sum of p past values. This segmentation is similar to the segmentation produced by the cochlea where complex sound mixtures are distributed through a cascade of low pass filters and differences between consecutive filters are examined. First it absorbs the high frequency components at the beginning of the cochlea and then progressively the low frequencies are propagated until the end of the cochlea. Let us consider a given time series, $\mathbf{X}(t) = [x(1), x(2), ..., x(T)]$, with T time samples. An AR model can then be defined as:

$$x(t) = \sum_{p=1}^{P} a_p x(t-p) + \varepsilon(t) \tag{4.5}$$

where a_p are the estimated AR coefficients at time lag p, and $\varepsilon(t)$ is normal noise. There are various ways of estimating the AR coefficients, such as least squares, forward-backward method and Yule-Walker method (Söderström and Stoica [215]). AR models were fitted to the raw mD data collected. The estimated AR coefficients capture individual movement characteristics. Such features can be used to identify different subjects.

4.3 Data collection

In the previous section the models used for mode of transport classification, gender and individual recognition and actions categorization have been detailed. Now the data collection set-ups will be described, the data processing and the results of the various experiments given.

The data was collected using the UFTRX module, which is a version of the DACQ unit, presented in Chapter 2, dedicated to ultrasonic data collection (Fig. 4.2). It has one analog output that powers a 40kHz ultrasonic emitter and one analog input connected to an ultrasonic receiver. The receiver convert the acoustic wave returned (mechanical energy) into a voltage (electrical energy) with a piezoelectric sensor. Then the signal is filtered at 50kHz with a passive Resistor Capacitor (RC) filter, amplified with a PGA and finally sampled at 500kHz by the ADC. The FPGA subsamples the data collected to 100kHz and sends the data to a laptop connected to the DACQ unit. The ultrasonic transceiver hardware is connected to a computer via a usb cable, through which the raw data is transmitted to the PC, for subsequent analysis with Matlab®.

4.3.1 Mode of transport classification

4.3.1.1 Data collection setup

For the first dataset (dataset 1) [97], six subjects (two females and four males) participated in the study. They were of the same age range of 25 to 35 years old. The UFTRX module was placed on a table at a height of 0.8m and the subjects were required to move in front of the UFTRX module in one of the following modes of transport: (1) walking, (2) running, (3) inline skating, (4) slow cycling (approximately 11km/h), and (5) fast cycling (approximately 22km/h). Data was collected from several repetitions of each action. Specifically, 74 successful trials were collected for modes 1, 2 and 4. Furthermore, 54 successful trials were collected for mode 3 and 80 trials for mode 5. Figure 4.3 shows the data collection setup.

4.3.1.2 Data processing

The mD signatures are visualized using a spectrogram, which can be obtained as the squared magnitude of the short-time Fourier Transform (STFT) of the data [17].

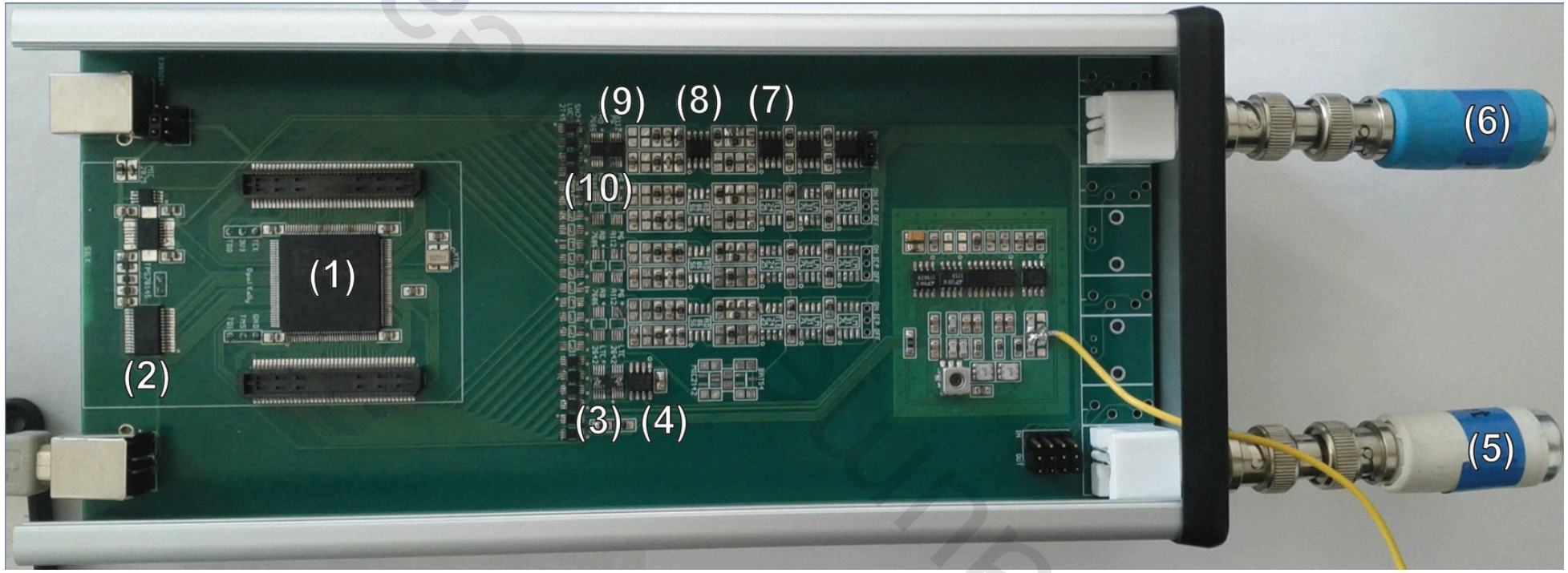
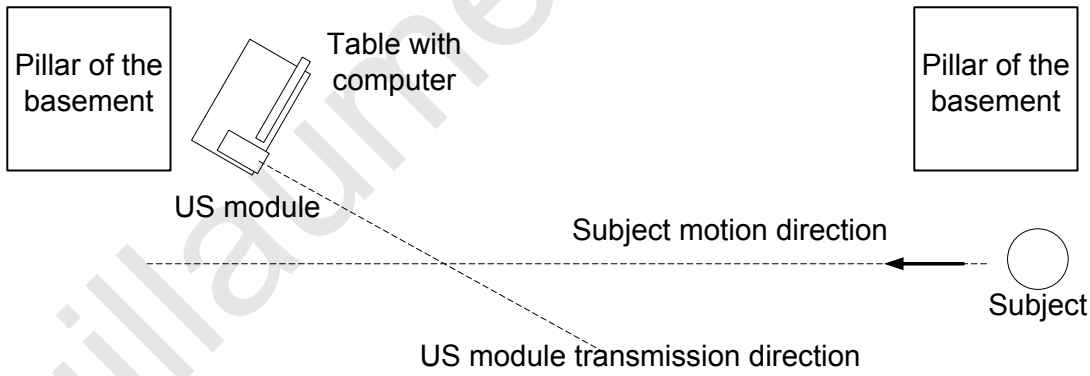


Figure 4.2: The UFTRX is composed of the controller part (1&2), the ultrasound emitter channel (3 to 5) and the ultrasound receiver and sampling channel (6 to 10). More specifically: (1) a high-performance Spartan3 FPGA XC3S50AN at 90MHz; (2) a RS-232 serial to usb controller FT232R; (3) 2 programmable DAC LTC2642; (4) a low noise rail-to-rail output 28MHz dual amplifier AD8656; (5) a transmitter transducer 400ET180; (6) a receiver transducer 400ER180; (7) a precision virtual ground TLE2425 (8) a low noise rail-to-rail output 28MHz dual amplifier AD8656; (9) a zero drift programmable gain amplifier PGA112; (10) a 16bit A/D converter at 500kSps AD7686.

(a)



(b)

Figure 4.3: The pictures show: (a) a subject on inline skates passing in front of the UFTRX module and, (b) the top-view of the experimental set-up.

A Hamming window was used in calculating the STFT. The obtained spectrograms were segmented into regions of interest (ROIs) containing the mD signatures of each mode of transport (see Fig. 4.5 for some examples). The frequency range of each ROI's frequency range spanned between 40kHz (the frequency of the transmitted ultrasound) and 41kHz (determined by the maximum object speed of interest). The length of the ROIs was 130 samples (2.6s) for walking, 65 samples (1.3s) for running, 90 samples (1.8s) for inline skating, 135 samples (2.7s) for slow cycling, and 55 samples (1.1s) for fast cycling. The length of the ROIs depends on the time when the subjects were in the detection zone of the UFTRX module; the faster a subject moves the shorter the duration spent in the detection zone.

Once the ROIs have been determined, it is necessary to extract the velocity of the subject. The velocity captured by the sensor is the orthogonal projection of the actual velocity of the subject (Fig. 4.4). As the subject passes by the ultrasonic module, this component tends to zero due to the incidence angle and the directional response of the transducers (Fig. 4.5). Consequently, the estimation of V_T is done only on the first half of the spectrogram where the velocity is constant. Finally, the classification feature, $C_{feature}$, for a spectrogram S_{ij} of length N is given by relation (4.6):

$$C_{feature} = V_T * \frac{1}{N} \sum_{j=1}^{N} \left(\frac{\sum_i (a_i f_i - \mu_i)^2}{\sum_i (a_i f_i)} \right) \tag{4.6}$$

where f_i is the frequency component, a_i the amplitude and μ_i the frequency component with maximum amplitude. Based on the obtained spectrograms (Fig. 4.5), it is expected that each mode of transport will have different energy characteristics, thus allowing discrimination between them. This algorithm will only work for single object observations. A more sophisticated algorithm is needed for simultaneous multiple object classification, e.g. Independent Component Analysis (ICA).

Average classification performance is estimated over 100 bootstrap cycles (Fig. 4.6). In each cycle 60% of data from each transport mode is randomly chosen for training, while the remaining 40% of the data is used as a test set. The classification features are estimated for each mode in the training set. This gives a threshold for each mode. Then for each sample of the test set the features are computed and the sample is classified as the condition with the nearest feature value. A simple linear classifier was used to show the potential of this method. It is expected that there is room for improvement by using more advanced classifiers to be able to discriminate

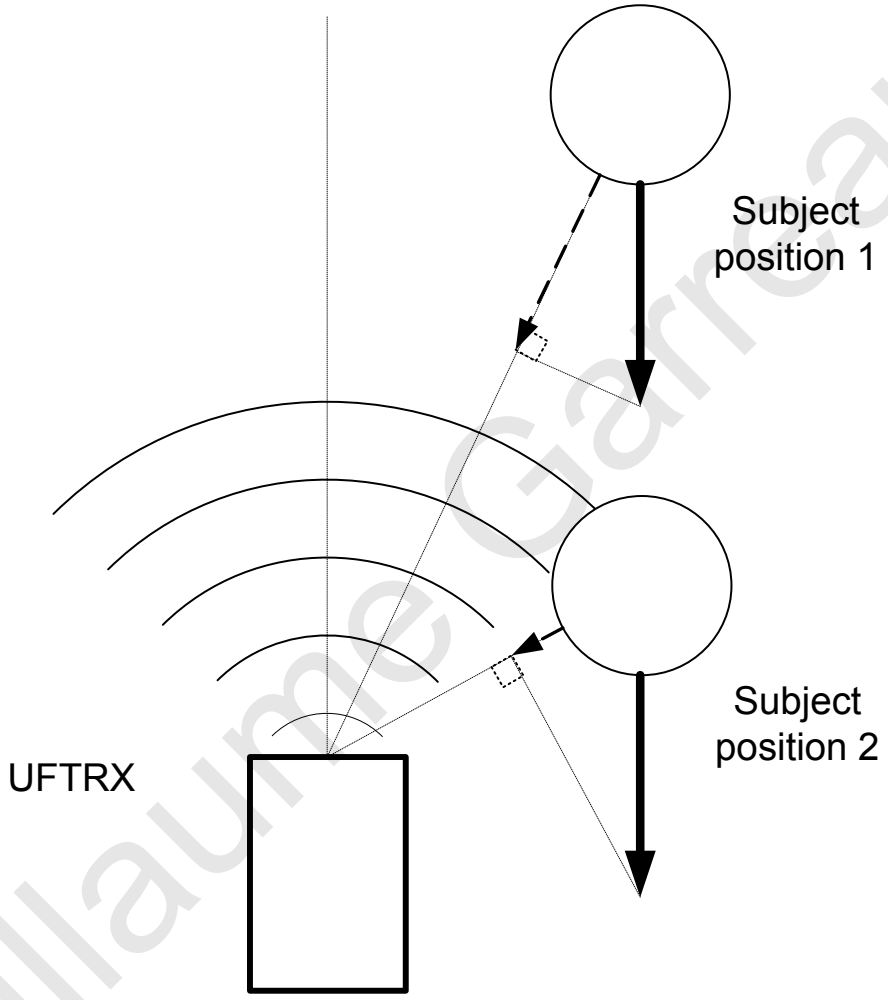


Figure 4.4: Mode of transport data collection setup. Spectrograms show a relative velocity that tends to zero as the velocity is the orthogonal projection of the velocity of the subject relative to the UFTRX module and the component value changes with the angle between the motion direction and the UFTRX module transmission direction.

between significantly larger populations, as is done in face recognition software.

4.3.1.3 Results

Figure 4.5 shows example ROIs for each of the five activities for a specific subject. The spectrograms show the frequency of the echo as a function of time. The straight line present in all spectrograms is the ultrasound transmitted from the module (at 40kHz). Deflections in frequencies above 40kHz denote a target moving towards the transmitter. The frequency deflection is proportional to the speed of travel: the faster the speed, the higher the deflection. In addition, the spread of the energy is dependent on the motion of the limbs. Characteristic mD signatures are obtained for each type of activity, which are similar for all subjects.

As expected a larger standard deviation is found for modes with significant limb motions and lower standard deviation for the modes with small limb motions, such as cycling. The average value of standard deviation, σ_A, is equal to 20.7 for fast cycling mode and 25.7 for running.

Figure 4.6 shows the average performance for classification of means of transport (top) and the categorization of errors (bottom). The average threshold values for the classification feature are: 2.42 for running, 1.10 for walking, 3.14 for fast cycling, 1.63 for slow cycling and 1.82 for inline skating. Average performances, expressed as mean±(standard deviation), were obtained as 96±3.2%, 95±3.9%, 81±5.7%, 62±6.9%, and 66±8.5% for running, walking, fast cycling, slow cycling and inline skating respectively. These performances can be explained by 2 effects: 1-the average standard deviation of the spectrogram and 2- the average velocity of the subject torso. The walking and running modes have high standard deviation due to the motions of the arms and legs. They are easily recognizable from the other conditions, and the average speed differentiates them from each other. For fast cycling, even though the subjects' limbs perform small motions, the effect of the average speed of the torso still allows a good recognition. The slow cycling and inline skate modes shows similar standard deviation and speed and consequently are more prone to error.

The false positive classifications can also be seen in Fig. 4.6 (bottom). An interesting observation is that all false negatives of walking are a result of misclassification as slow cycling. This could be a result of some walkers displaying small arm movements, resulting into signatures similar to those of cycling. Fast cycling is mostly misclassified as running (80%). Slow cycling is misclassified as walking (53%) and

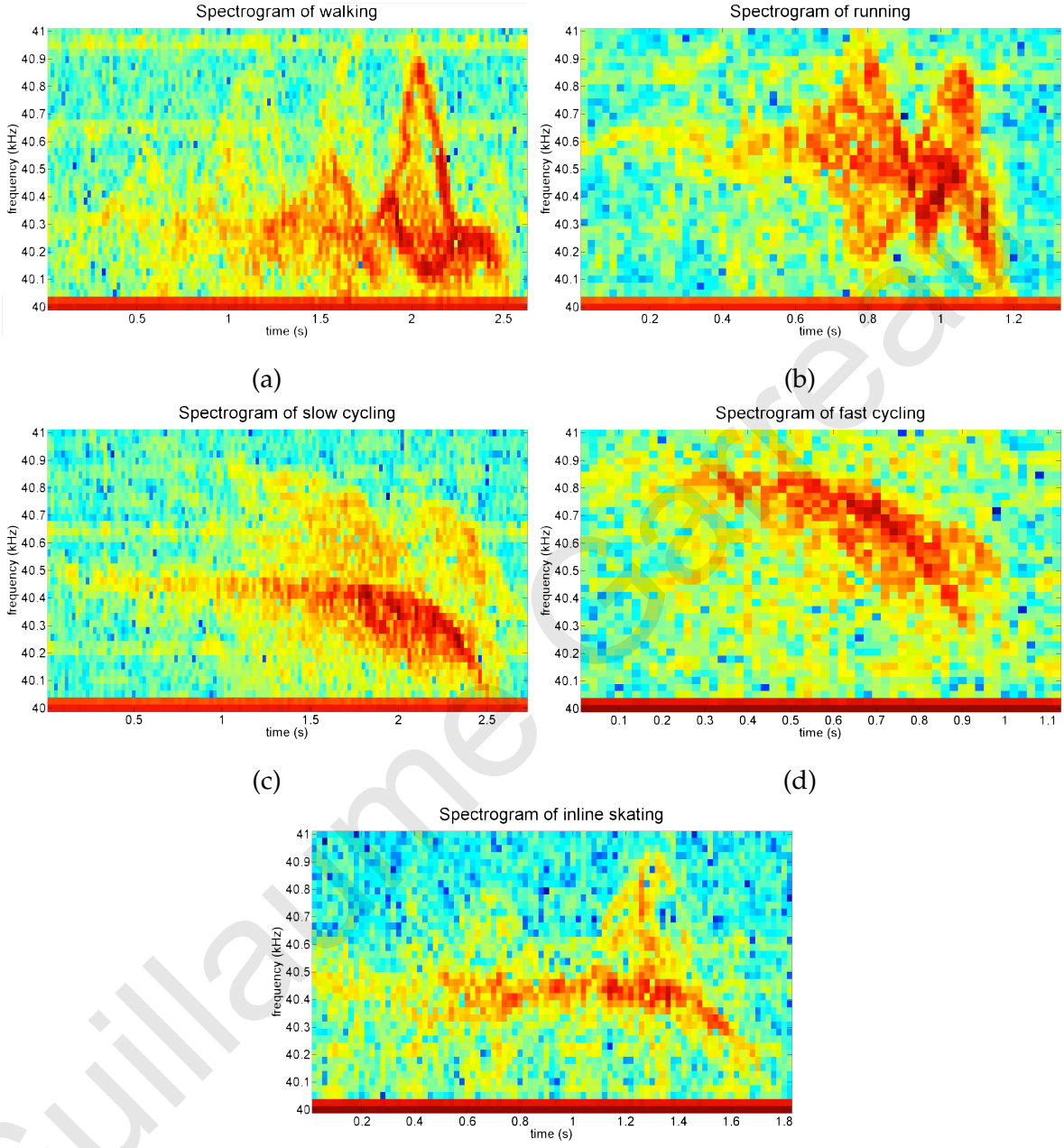


Figure 4.5: Characteristic spectrograms for each motion are shown. The x-axis is time in seconds. The y scale shows frequencies from 40kHz to 41kHz. Spectrograms shown are: (a) a walking subject, (b) a running subject, (c) a subject cycling slowly, (d) a subject cycling fast and, (e) a subject on inline skates.

as inline skating (26%). Finally inline skating is misclassified at 63% as slow cycling and 26% as running. Overall, misclassifications are almost equally spread between walking, running and slow cycling (about 25%) and less likely for fast cycling (9%). This seems contradictory, as fast cycling seems to be the less confusing, with a much higher threshold value, one would expect a better recognition performance. A greater variation of the standard deviation could explain the poorer recognition rate.

Finally the effect of the training set size on classification performance is shown in Table 4.1. It is shown that size of the training set has only a small effect on the classification performance. An improvement of 5% in performance was observed between training set size with the lowest performance (training set size of 10%) and the training set size with the best performance (training set size of 80%). It is shown that even with 10% training set size correct recognition for walking and running is as high as 91%.

4.3.2 Gender and individual recognition

4.3.2.1 Data collection setup

For the second dataset (dataset 2) [96], 13 subjects (six females and seven males) participated in the study. They were of the same age range of 25 to 35 years old. The UFTRX module was placed at a height of 1.5m and the subjects were required to walk at two different speeds (3 and 6km/h) on a treadmill positioned 2.5m away from the UFTRX module. The subjects were facing the UFTRX module. Data was collected from 5 repetitions of each action (each of a duration of 10s).[1]

4.3.2.2 Data processing

The mD signatures are visualized using a spectrogram, which can be obtained from the fast Fourier Transform (FFT) of the data. The processing models consist of seven steps (Fig. 4.7) that can be grouped into 2 stages: the segmentation stage (steps 1-5), which obtains a set of features or events from each micro-sonar file; and the categorization stage (steps 6-7), which calculates representative templates from these events that are used as category event templates. The segmentation consists of the following steps: (1) the spectrogram is first calculated and (2) then centred on

[1]The data collection was done at UCY in collaboration with Salvador Dura-Bernal.

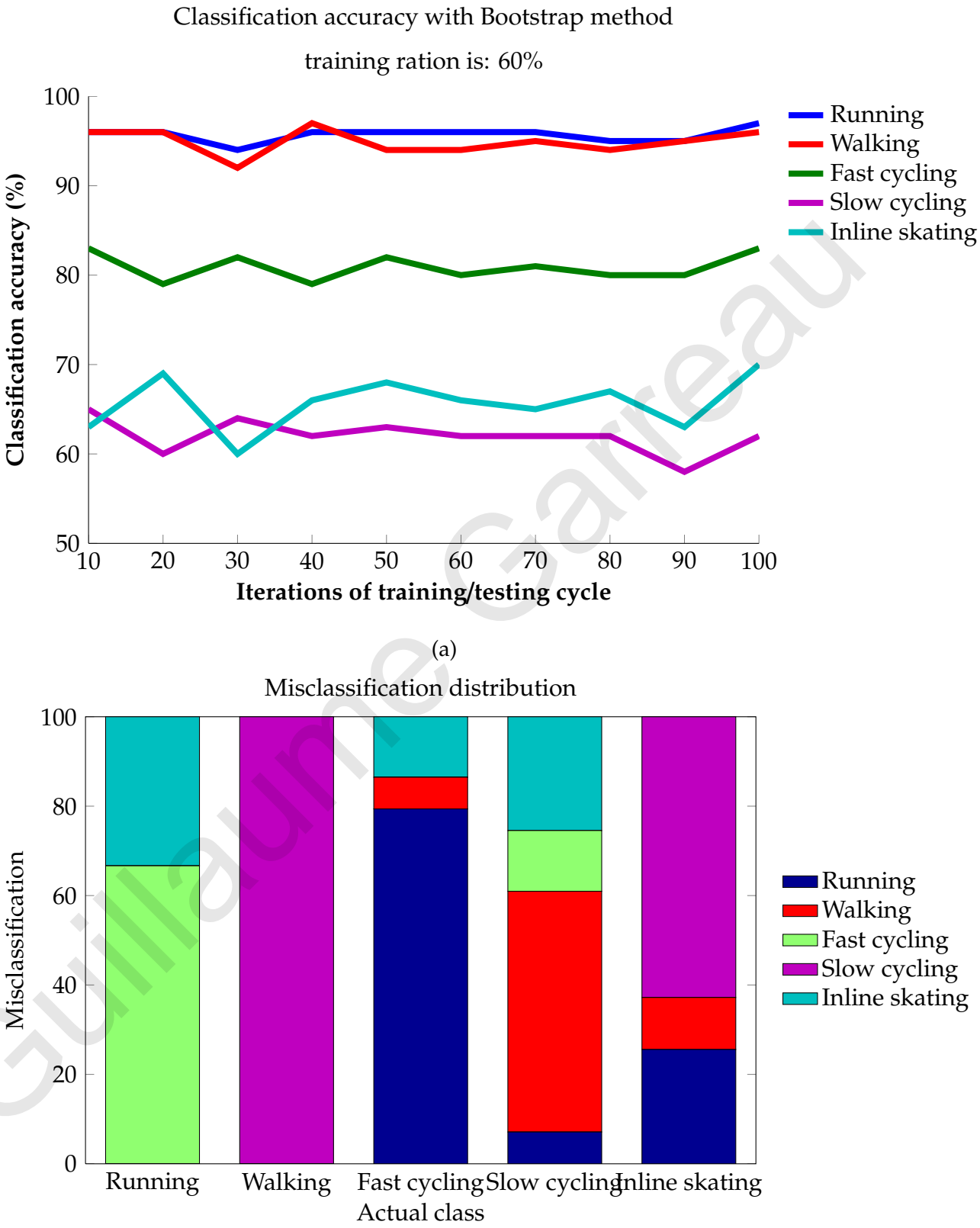


Figure 4.6: (a) Average classification performance for the 5 transport modes. (b) Categorization of misclassifications.

training set size	10%	20%	30%	40%	50%	60%	70%	80%	90%
running	91.4	93.3	95.0	95.2	95.7	95.8	96.3	96.5	96.4
walking	92.8	94.5	94.8	95.1	95.6	95.2	95.8	95.9	96.8
fast cycling	81.2	82.1	81.9	81.8	80.9	80.8	80.6	82.1	78.4
slow cycling	57.0	58.0	61.1	62.8	63.2	62.8	61.9	63.0	60.1
inline skating	61.7	60.2	63.8	64.3	64.7	66.8	64.7	66.9	64.5

Table 4.1: Effect of training set size

40kHz (the frequency of the transmitted ultrasound) with a window of ±620Hz. The obtained spectrograms are (3) scaled to a logarithmic scale and normalised to keep amplitude values higher than half of the maximum amplitude of the spectrograms. They are finally segmented (4) into regions of interest (named events). The length of the events is set to 134ms i.e about 75 events per file and the distance between 2 consecutive events is 74ms. (5) The amplitudes for each frequency are summed over time during one event, in order to obtain a single amplitude value for each frequency component for each event. This collapses the event from a 3D representation into a 2D representation and forms the basis for the event templates.

The categorization stage is implemented using k-means clustering (MacQueen [163]) to learn a set of event templates from the data, followed by a nearest neighbour approach to determine the closest event template (Euclidean distance) for each test event.[2] This is achieved by randomly dividing the full set of data in 2 equal parts, one being the training set and the other one the testing set. (6) The k-means clustering algorithm is applied to all the events of each category separately. This ensures that there are a set of event templates strongly representative of each category and that, within each category, the event templates are relatively uncorrelated. Given that the number of clusters (n) must be fixed before hand, the clustering procedure is repeated for different numbers of clusters (n from 5 to 70 in steps of 5), in order to find the optimum value. (7) During testing the Euclidean distance of the test event feature vector and the centroid of each cluster is estimated. Thus, for each incoming event we obtain N values, representing the distance to each of the N event templates, where N is equal to the number of clusters per category multiplied by the number of categories. Then the average distance over the event templates of each category is calculated for all incoming events belonging to the same trial, such that the minimum distance indicates the winning category. This has the advantage of showing some invariance with respect to the temporal position of the test event. However, it is also easier to obtain a false positive as there is a higher probability of finding a similar event in the wrong category when comparing to all events. This is repeated 30 times and the average success rate is estimated.

[2] A simple linear classifier was used to show the potential of this method. It is expected that there is room for improvement by using more advanced classifiers to be able to discriminate between significantly larger populations.

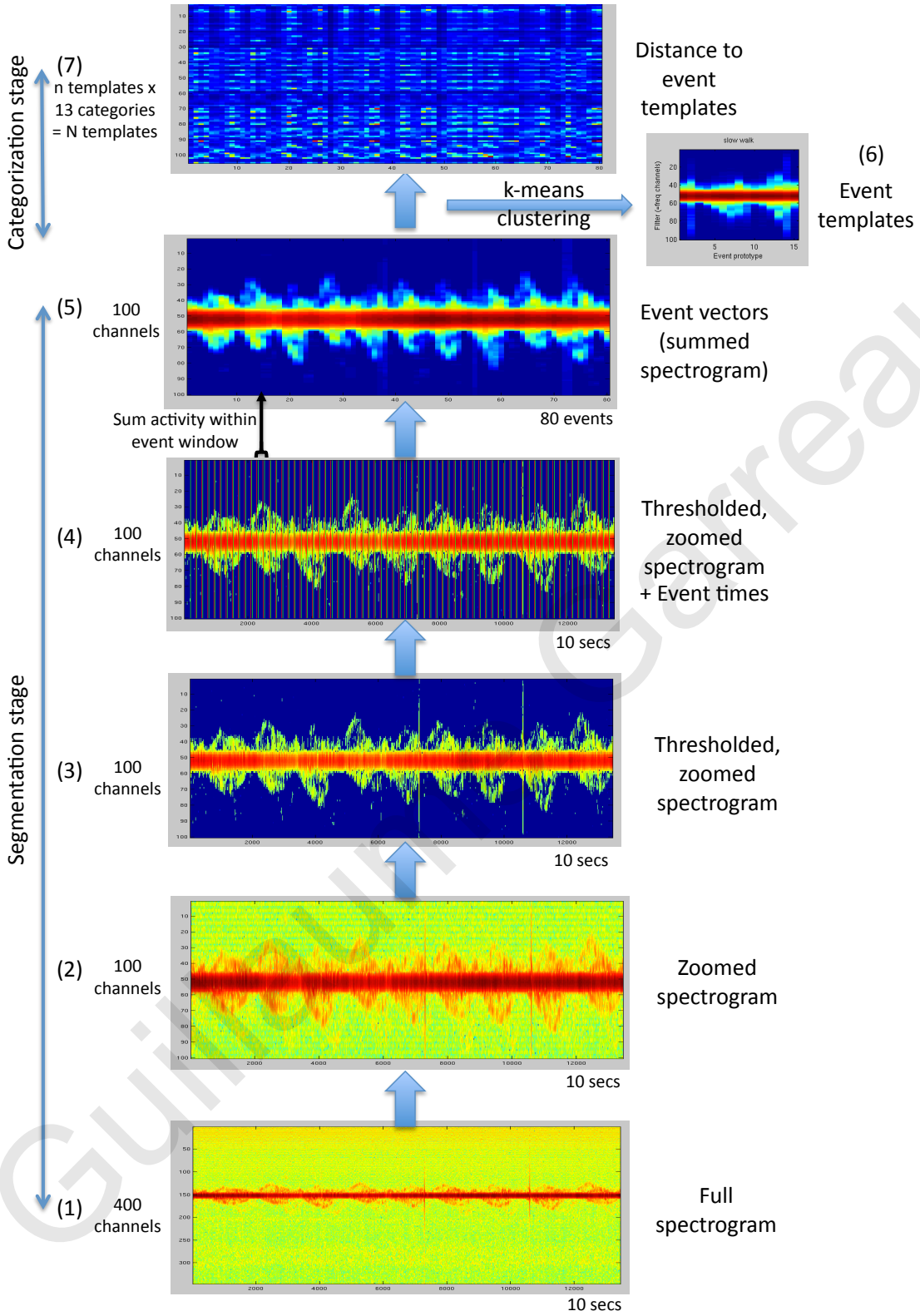


Figure 4.7: The figure shows the step by step of the processing of the data from spectrogram determination to the event template calculation and then classification.

4.3.2.3 Results

Investigations concentrated on the effect of the number of clusters for one event template on individual and gender classification performance. Even though other parameters, such as events size, interval variation between events or precision of the spectrogram, have potentially interesting effects on the performance, these remain the subject of future work. Figure 4.8 shows the effect of cluster number on individual identification; the number of clusters investigated ranged from 5 to 70, in steps of 5. Average performances, expressed as mean±(standard deviation), are 72.1±19%, 81.6±12.5%, 81.6±12.6%, 84.1±11.6%, 85.4±11.0%, 87.1±9.3%, 83.6±10.3%, 84.2±10.5%, 84.4±9.4%, 85.1±10.3%, 85.0±10.1%, 81.1±13.2%, 85.3±11.0% and 83.7±11.2% for cluster number of 5 to 70 by step of 5. The overall average recognition accuracy is 83.2%. The best average recognition rate is 87.1% for a cluster number of 30 for 13 subjects and 88.7% for 12 subjects. This is similar to results reported by Zhang and Andreou [255] (90%, 8 subjects) and higher than those reported by Kalgaonkar and Raj [133] (72%, 30 subjects). For 1 subject, performance as high as 100% was obtained for almost all clusters number (12 over 14) and an average of 99.7% over all cluster number. As expected, the average performance varies with the cluster number. A very low cluster number as well as a high cluster number results in worse performance. If the number of clusters is too high, the algorithm cannot generalize to basic event patterns larger than the event template size (Fig. 4.7, top right) as it does not have enough data to extract a useful pattern. At the other extreme if the cluster number is too low, the performance in classification is worse as the event pattern contains too many repetitions of the identical basic pattern. The size of the event template (number of clusters) has to be chosen relative to the events size (in our study about 134ms). Figure 4.9 shows the event templates for each subject for a cluster number of 50.

We can see in Fig. 4.10 that gender is characterized by distinct event templates (top). Even though one example can be quite different from the event template (bottom), the classification between gender is straightforward. For the same number of clusters, male event template contains more repetition of the same curvy pattern (similar to inverted S) than female one. In addition, the female basic pattern is flattened compared with male one.

Figure 4.11 shows the effect of cluster number on gender identification. The

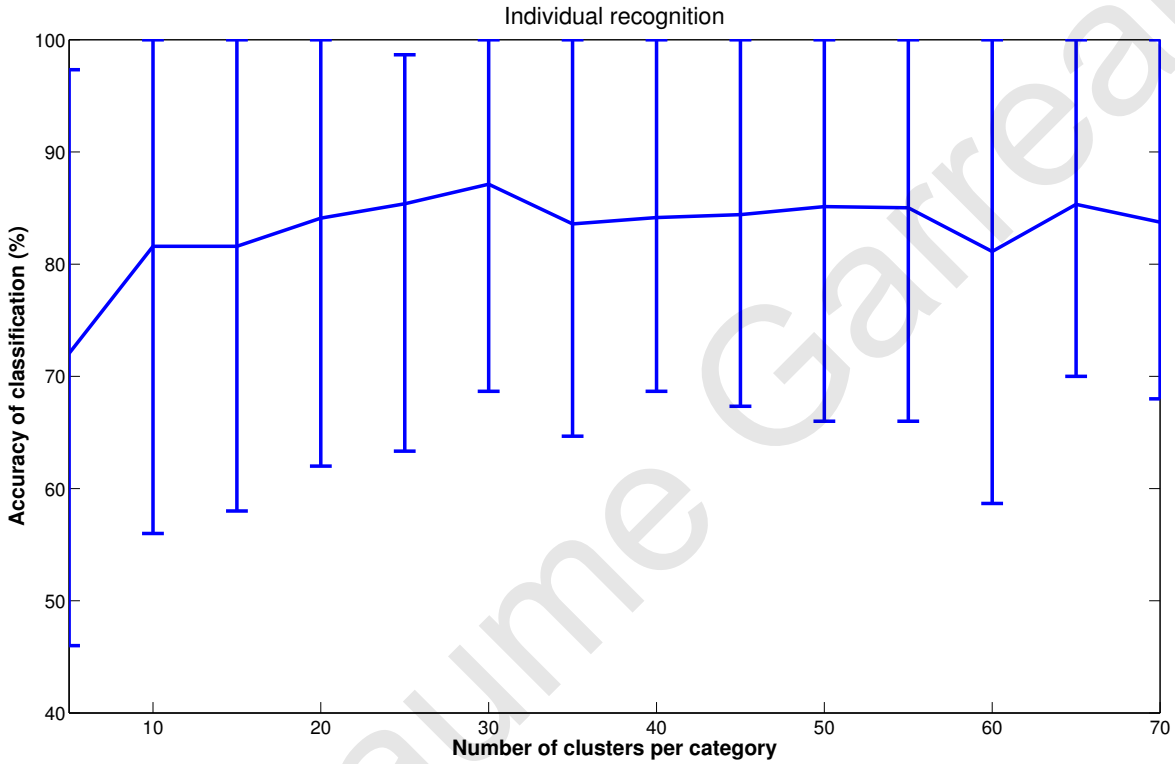


Figure 4.8: The effect of the cluster number on individual recognition is shown. The average over all subjects is plotted and reaches a maximum of 87.1%. The error bars represent the subjects variability. For one subject, the average is equal to 99.7% over all event template size.

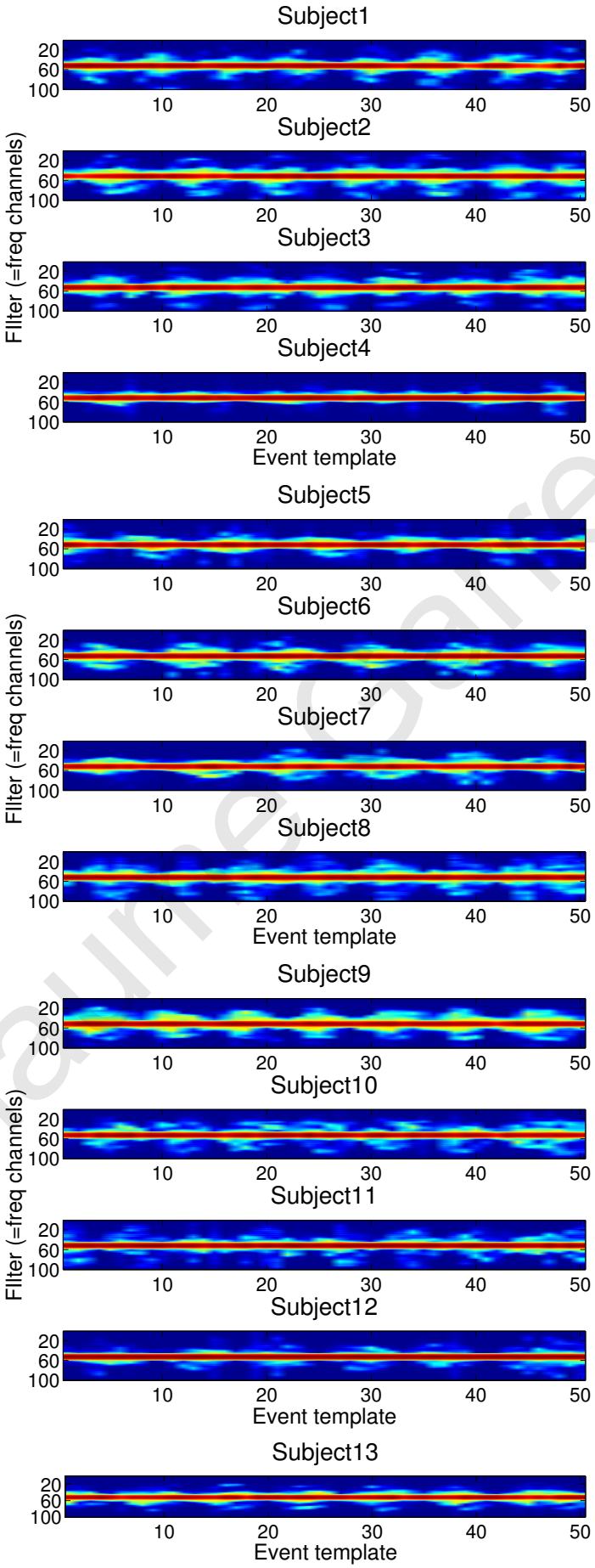


Figure 4.9: Event templates for the 13 subjects for number of clusters equal to 50.

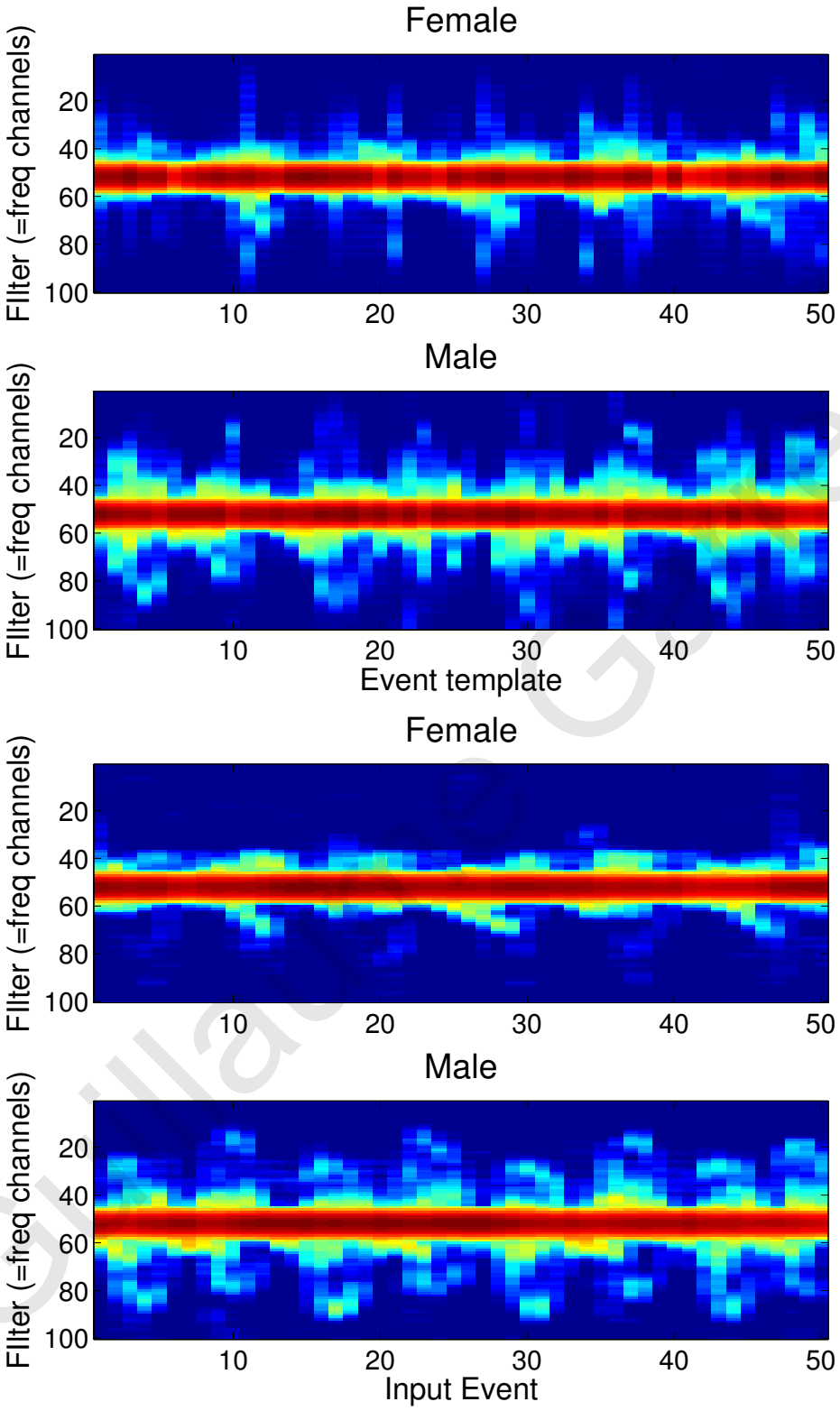


Figure 4.10: Gender classification: the top spectrograms are the event templates for female and male categories for cluster number of 50 and the bottom spectrograms are one example for each gender.

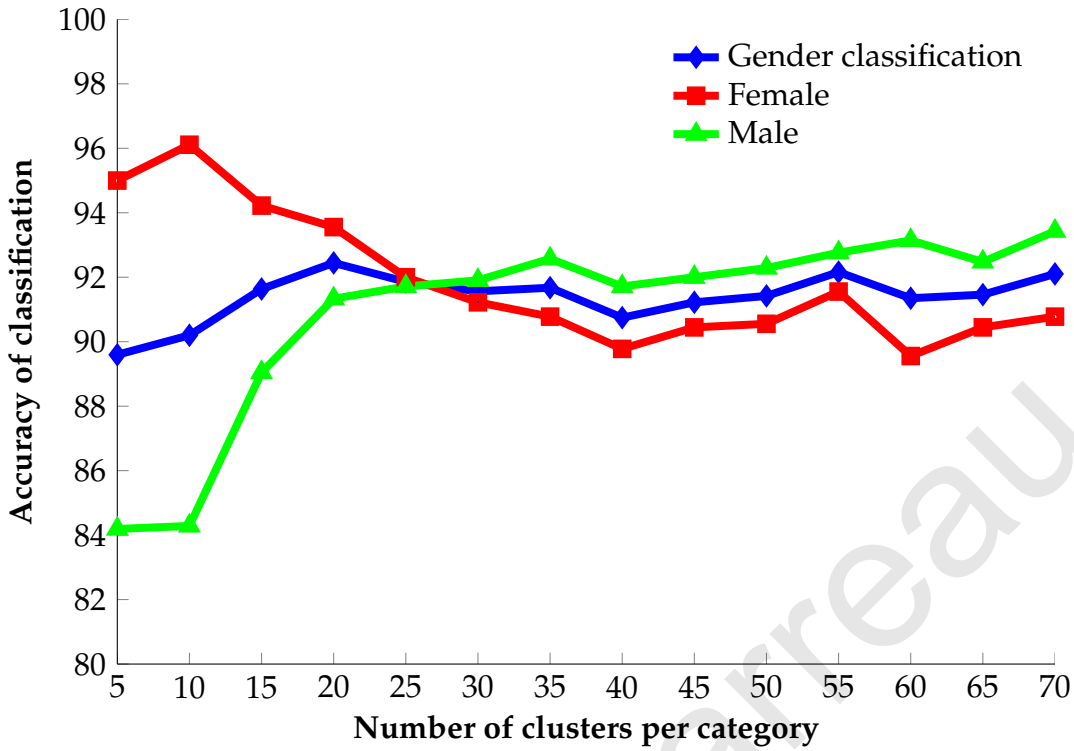


Figure 4.11: Gender classification as a function of the number of clusters. Average accuracy as high as 92.4% is obtained for a cluster number of 20.

averages are 89.6±7.6%, 90.2±8.4%, 91.7±3.7%, 92.4±1.6%, 91.9±0.2%, 91.6±0.5%, 91.7±1.3%, 90.8±1.4%, 91.2±1.1%, 91.4±1.2%, 92.2±0.9%, 91.4±2.5%, 91.5±1.4%, 92.1±1.9%, for cluster number of 5 to 70 in steps of 5. The best recognition rate is 92.4% which is much higher than the 68.7% reported by Kalgaonkar and Raj [133]. It is interesting to point out that gender classification gives better performance than individual recognition, whereas Kalgaonkar and Raj [133] showed better individual recognition than gender classification. This suggests that the proposed methodology can recognize general features. It would be interesting to test this on other characteristics, such as size of subject and speed of motion. It must also be mentioned that the effect of event template size is different for each gender (Fig. 4.11). Males obtain the best performance for a number of clusters of 70, with 93.4%, while best performance for female is 96.1% for a number of clusters of 10. This can be explained considering that the number of repetitions of the curvy pattern in the event template for female is lower than the one for male (Fig. 4.10). 8 patterns can be seen for male and only 6 for female, thus the female event template can be recognized faster.

Table 4.2 shows the best average performance obtained and the performance reported in other studies [133,255]. It is shown that regarding individual recognition

Source	Kalgaonkar et al., 2007	Zhang et al., 2008	Garreau et al., 2011
Subjects (%)	72	90	87
Gender (%)	69	no data	92

Table 4.2: Average performance in individual identification and gender classification.

the proposed system has similar accuracy than the one of Zhang and Andreou and better than the one of Kalgaonkar and Raj. The proposed system shows the best accuracy for gender classification.

4.3.3 Action classification

4.3.3.1 Data collection setup

Datasets 1 and 2 were also analysed in terms of actions classification (these investigations are described in [70]). In total 13 different actions were selected from datasets 1 and 2: walking (unknown speed), walking slowly (3km/h), walking fast (6km/h), running (unknown speed), running slowly (9km/h), running fast (12km/h), inline skating, slow cycling (approximately 11km/h), fast cycling (approximately 22km/h), clapping hands, calling "help me" while waving arms up in the air and, calling "come here" while beckoning with the right arm.[3]

4.3.3.2 Data processing

The processing is the same than in previous subsection 4.3.2.2.

4.3.3.3 Results

Examples of the event templates obtained for each action category are shown in Fig. 4.12 and Fig. 4.15 and action categorization results for datasets 1 and 2 as a function of the number of k-means clusters (event templates) per category are shown in Fig. 4.13 and Fig. 4.16 respectively. The optimum number of clusters for dataset 1 is 17, reaching 95.5% accuracy; and for dataset 2, 55 clusters with 95.3% accuracy.

[3]The data collection was done at UCY for dataset 1. The data collection was done at UCY in collaboration with Salvador Dura-Bernal for dataset 2. The data processing methods was a collaborative effort with Salvador Dura-Bernal conducted both at UCY and University of Plymouth.

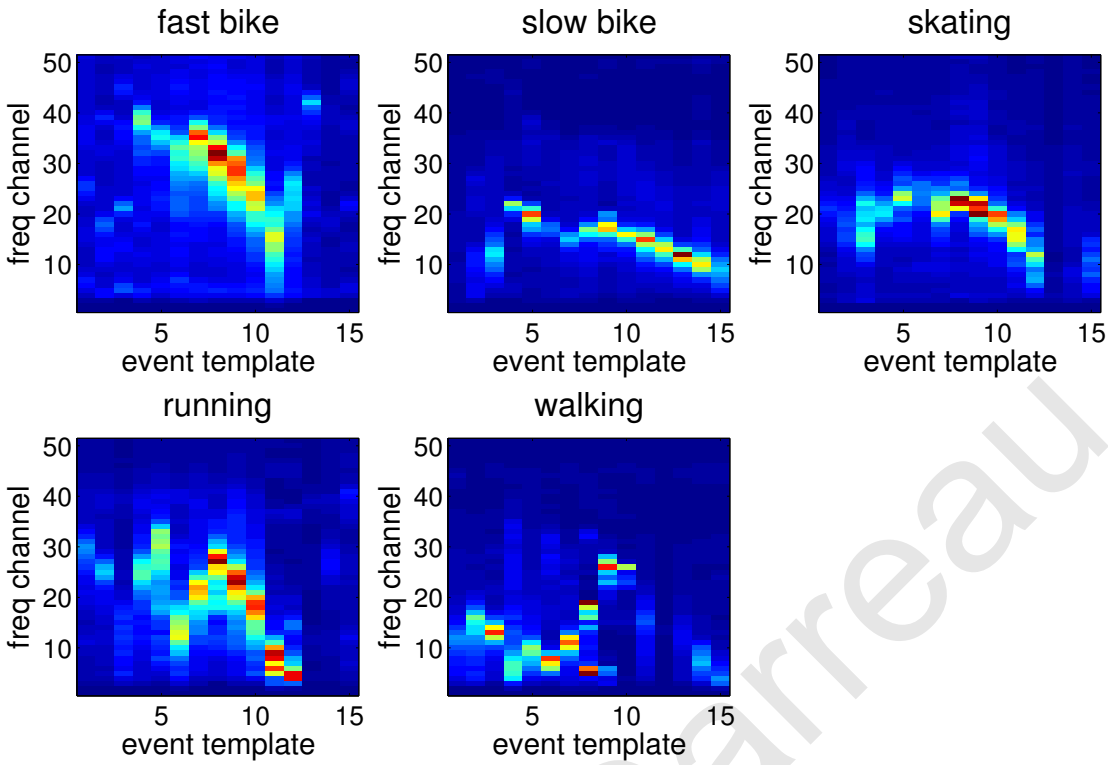


Figure 4.12: Some action templates for dataset 1.

The action categorization results for datasets 1 and 2 as a function of the number of test events used per action are shown in Fig. 4.14 and Fig. 4.17 respectively. An accuracy of approximately 90% or above was achieved with only 4 out of 17 incoming events for dataset 1, and only 20 out of 70 events for dataset 2.

The results above confirms the excellent performance of the categorization model. In previous investigations, a performance of 87% for individual recognition on all subjects (as high as 100% for one subject) and 92.4% for gender classification were reported [96]. Here an accuracy greater than 95% for two different datasets composed of mD signatures of human actions was achieved.

This outperforms the results for dataset 1, with categorization accuracy of 81% [97], as well as similar action categorization studies previously published, such as [135], which reported 88.4% for an 8-category dataset. Importantly, all results are obtained using a 50% training ratio, which means the model is able to generalize to new data well. For dataset 2 the accuracy of all categories was close to 100% except for 'clapping' (85%). This can be explained, by considering that the ultrasonic reflected wave was least affected by the relatively small clapping motion, and that the particular action had the highest inter-trial variability as people clapped at many different frequencies and with different hand positions.

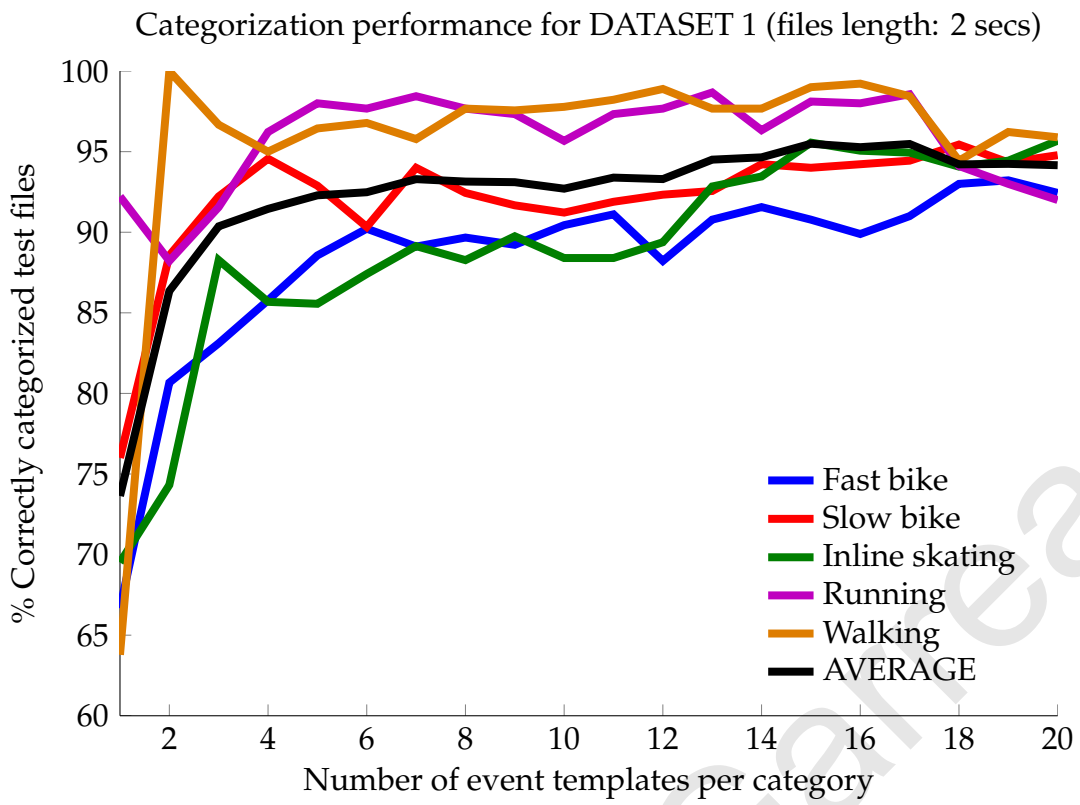


Figure 4.13: Action categorization for dataset 1 as a function of the number of event templates per category.

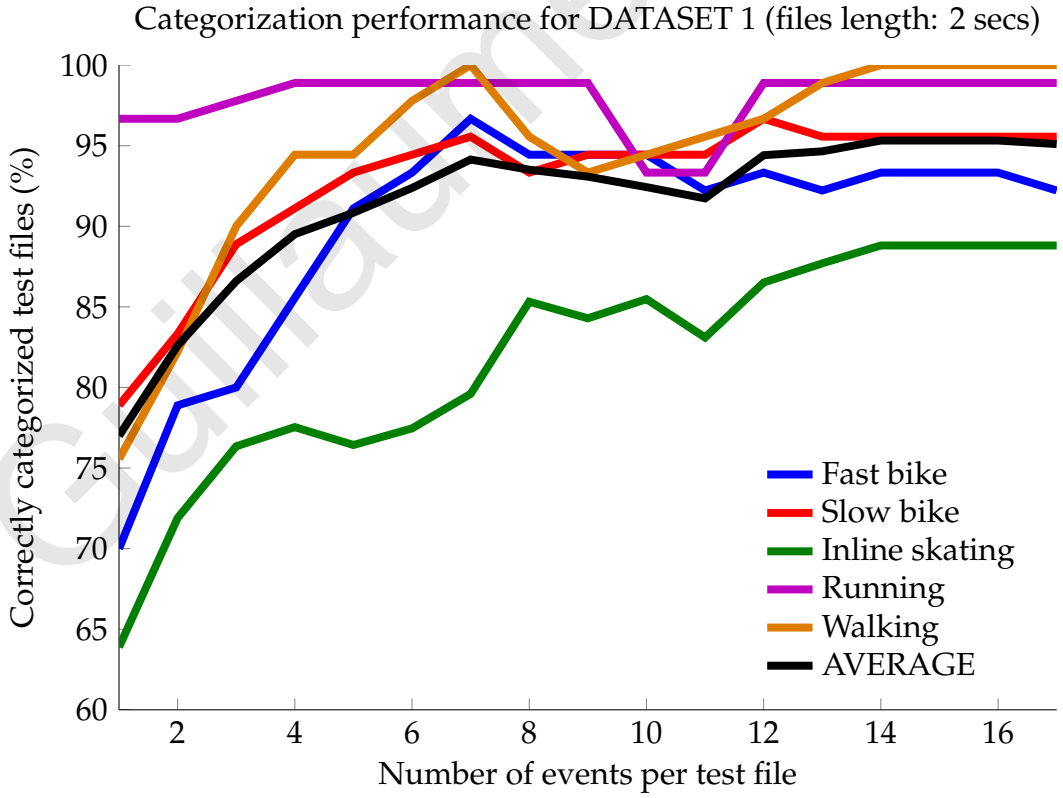


Figure 4.14: Action categorization for dataset 1 as a function of the number of test events used per action. Longer actions improve recognition.

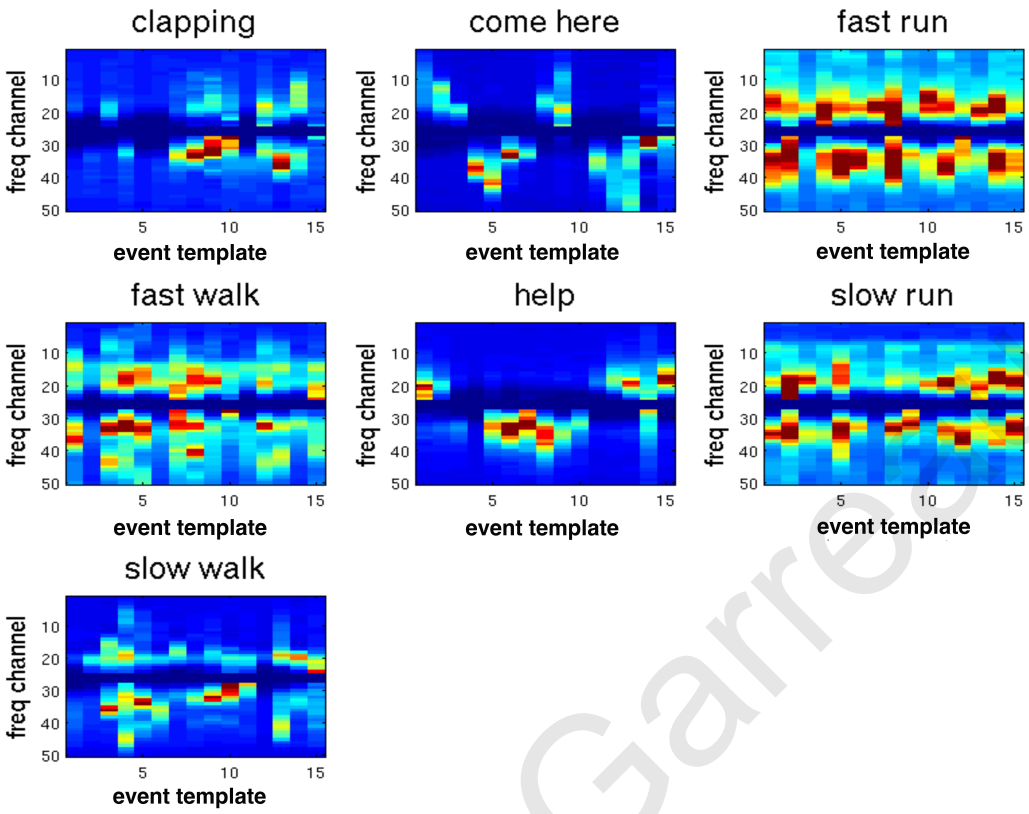


Figure 4.15: Some action event templates for dataset 2.

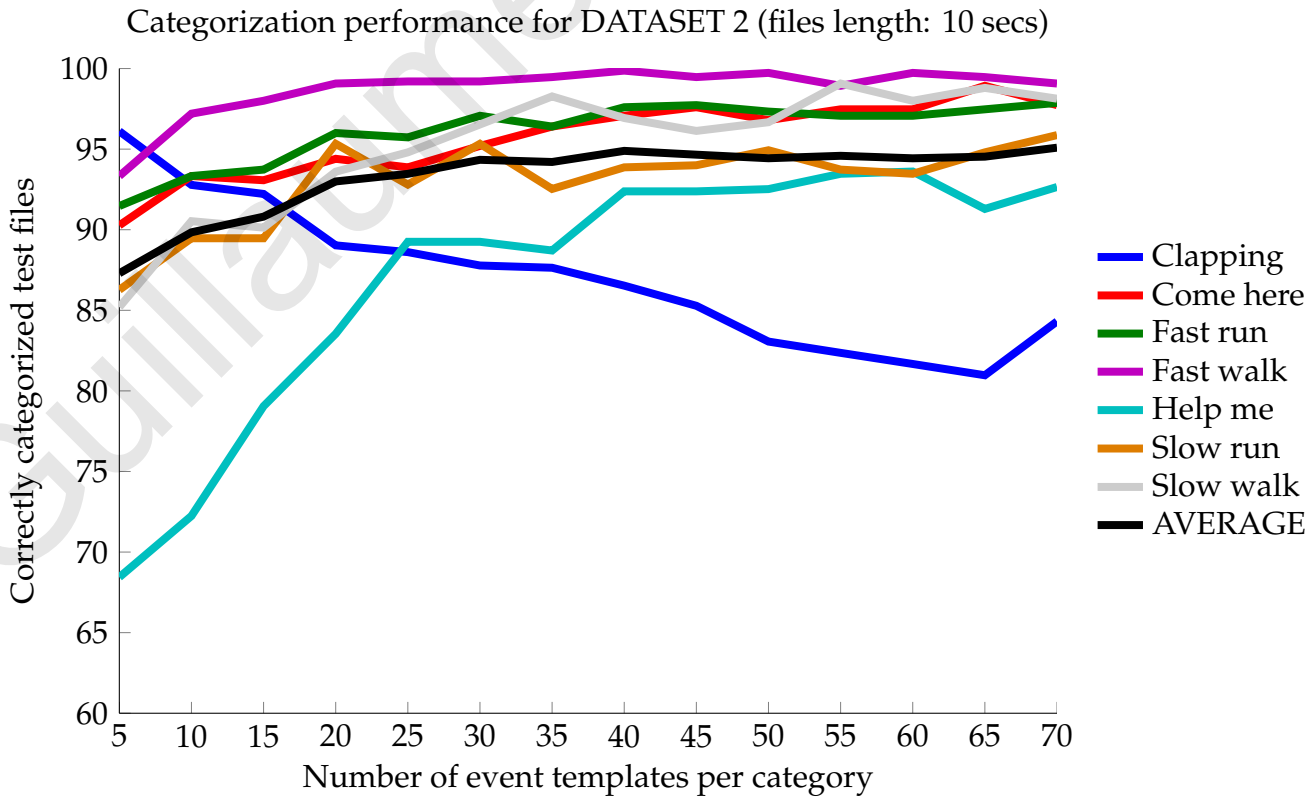


Figure 4.16: Action categorization for dataset 2 as a function of the number of event templates per category.

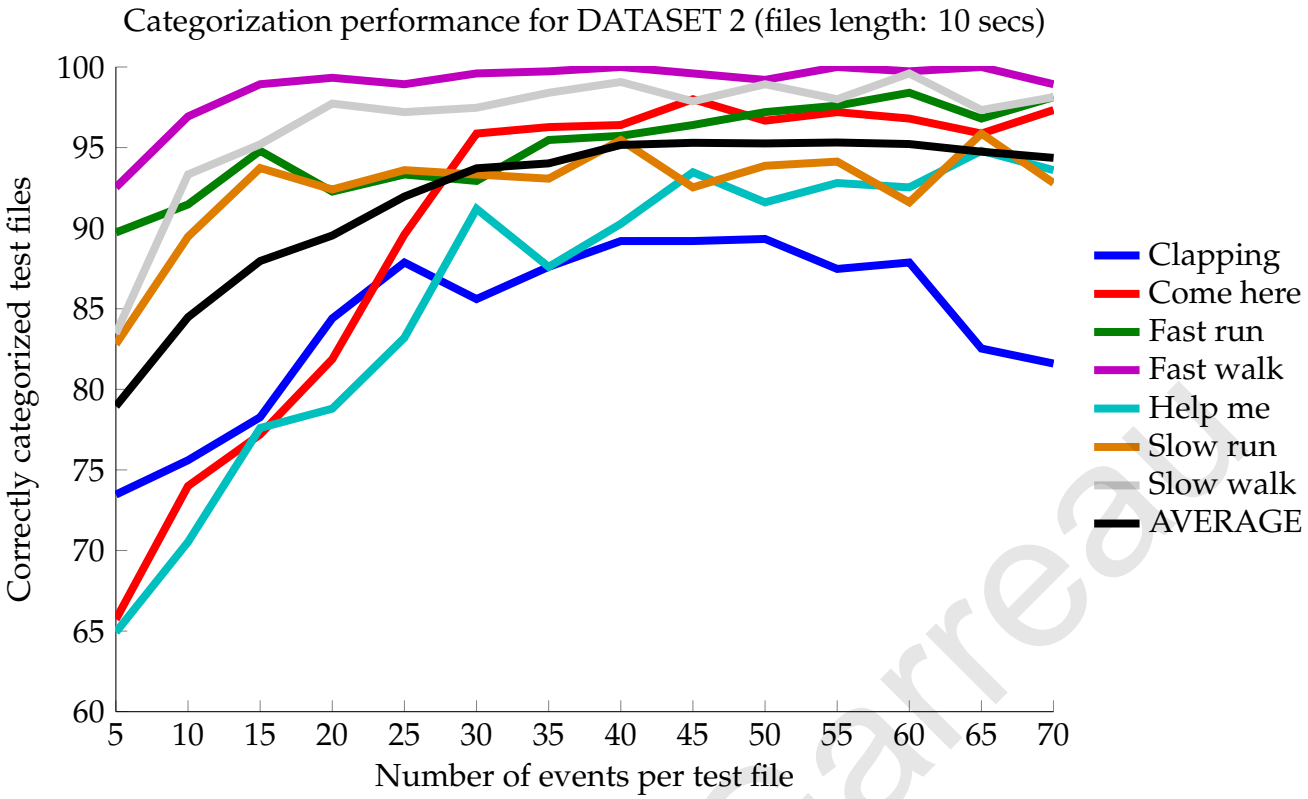


Figure 4.17: Action categorization for dataset 2 as a function of the number of test events used per action. Longer actions improve recognition.

It was also noticed that the number of event templates can be significantly reduced compared to the number of events per action. It means that it is possible to keep high categorization performance, > 90% with only 3 event templates in dataset 1, and suggests that the individual periodic components of each action, like steps in the walking condition, are captured by the event templates. Therefore only a small number, corresponding to single cycles of an action, is required. The length of the event templates, approximately 200ms, is consistent with this interpretation.

It was noticed that if the number of test events per action-file is reduced it does not have a significant effect on the performance (higher than 90% accuracy with less than 25% of the test events). It confirms that only a small number of events, corresponding to one cycle, are required to characterize each action. This is interesting as it suggests that a real time implementation of the model is possible. Along with the onset of the events, the events contribute to form a belief of the current action, by reinforcing or weakening the previous hypothesis.

Similarly, the model also showed a strong robustness to variations of the event length (< 3% variation for values between 140ms and 500ms) and event interval

(< 3% variation for values between 140ms and 500ms). The optimum value of the event length and interval provides a compromise between the model's selectivity and invariance in the temporal domain.

4.3.4 Individual recognition with AR model

4.3.4.1 Data collection setup

For the third dataset [98], subjects are asked to walk towards the UFTRX from a distance of six meters. They each repeated the task 10 times. Six subjects (two females and four males) participated in the data collection. They were of the same age range of 25 to 35 years old.

4.3.4.2 Data processing

The first processing step is to isolate the ROI containing the subjects mD signatures. Segmentation of the ROIs is performed in real-time during continuous data collection from the UFTRX unit. About 10 ROIs of three to four seconds were collected for each subject (2 females and 4 males). The second step is to create a model or template for each subject, which will be used for the classification. The template is created in 2 steps. Firstly, a predictive model for each individual trial is created using an AR model. The estimated AR coefficients obtained from each trial are aligned in time using the 'icoshift' function. Finally, a template for each individual is created by averaging the estimated model coefficients of each trial of the same class. Individual classification is obtained using Leave-One-Out (LOO) cross-validation.[4]

Once the ROIs have been isolated, a model of each trial is computed using AR models. The optimum AR order was estimated using the Final Prediction Error (FPE) [215]. AR models of orders p=2,3,...,19 were fitted on the raw data and the FPE was obtained for each p, using the System Identification toolbox for Matlab®. Based on the FPE AR models of order 10 were chosen. Figure 4.18 shows an example of the decreasing FPE for a randomly chosen trial and subject.

The raw mD data were split into windows of 10000 samples (0.1s), overlapping by 8000 samples (0.08s). On each of these segments 10^{th} order AR models were

[4] A simple linear classifier was used to show the potential of this method. It is expected that there is room for improvement by using more advanced classifiers to be able to discriminate between significantly larger populations.

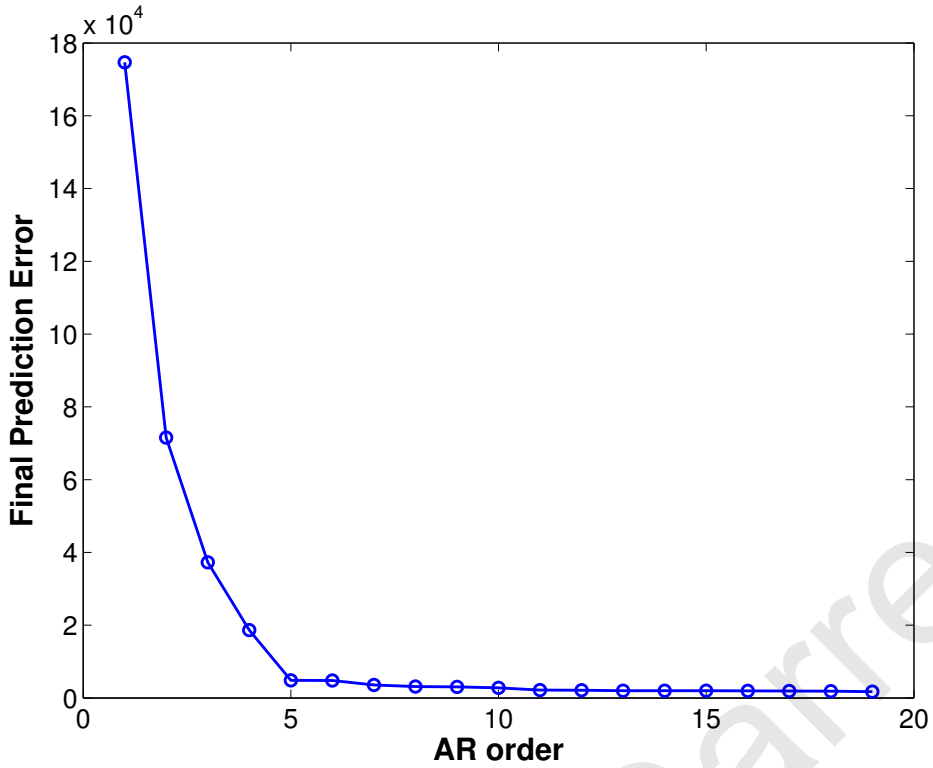


Figure 4.18: Example of estimated FPE over different AR model orders.

fitted using the forward-backward algorithm. The individual gait characteristics for a particular movement sequence and subject were identified by tracking the estimated AR coefficients at different time delays.

A template for each category is obtained using the average of the estimated AR coefficients of all trials of the category. The first step is to align all trials using the '*icoshift*' function. This Matlab® function is specifically designed for solving signal alignment problems. Alignment is performed with reference to a template, which can be estimated as the average, maximum or median of the dataset. The peaks in the spectra are used for the alignment. This function was originally implemented for spectra alignment, but can also be successfully applied for alignment in the time domain (for more details see [202, 226]).

Finally the performance for individual recognition is obtained using Leave-One-Out (LOO) cross-validation. At each LOO validation, the template for the particular subject and movement type is created from all trials but one (test trial). The correlation between the test trial and all templates created for all subjects is then obtained ('*xcorr*' function in Matlab®). The test trial is classified as belonging to the subject of the corresponding template with highest correlation.

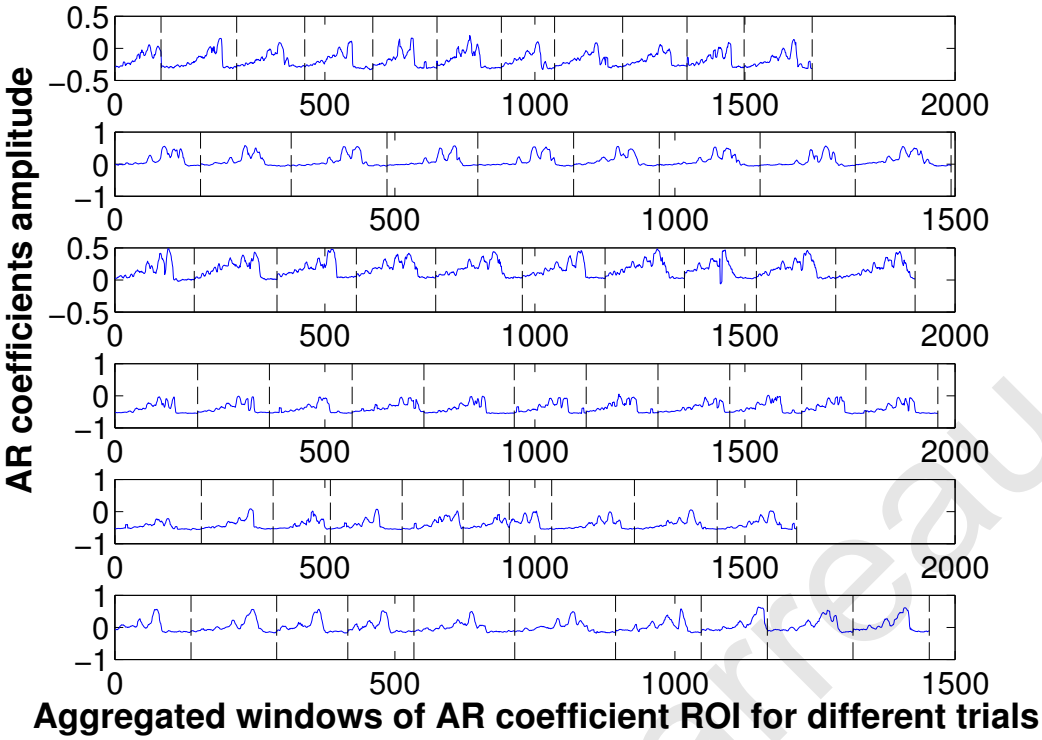


Figure 4.19: Estimated coefficient a_{10} for walking towards the UFTRX unit. Each row corresponds to one person and many events concatenated together.

4.3.4.3 Results

An AR model of order 10 was fitted to the data and the estimated AR coefficients a_p at different delays p were tracked over time. Figures 4.19 and 4.20 show the 10^{th} order AR coefficient for walking towards the ultrasonic transceiver module for all trials of all subjects; an AR model order of 10 corresponds to 0.1ms.[5] The following can be deduced: individual gait characteristics can be identified from the AR coefficients, with some delays being more appropriate than others – e.g. the coefficient estimated at a delay of 0.1ms Fig. 4.19; see Fig. 4.20 for an example of AR coefficient that do not capture individual gait characteristics). This implies that gait characteristics are related to the time lag. This is explained by the fact that if each limb was modeled as an oscillator, each one would have a different characteristic frequency. The torso will have a lower oscillatory frequency than the arm and thus the time lag needed to model it (i.e. predict the next limb position) will be shorter.

As seen in Fig. 4.19, there is a consistency between different trials of the same subject. This property is interesting and suggests that the AR coefficients at particular

[5]Please note that there was a mistake in the related publication, where the delay was stated as 10ms [98].

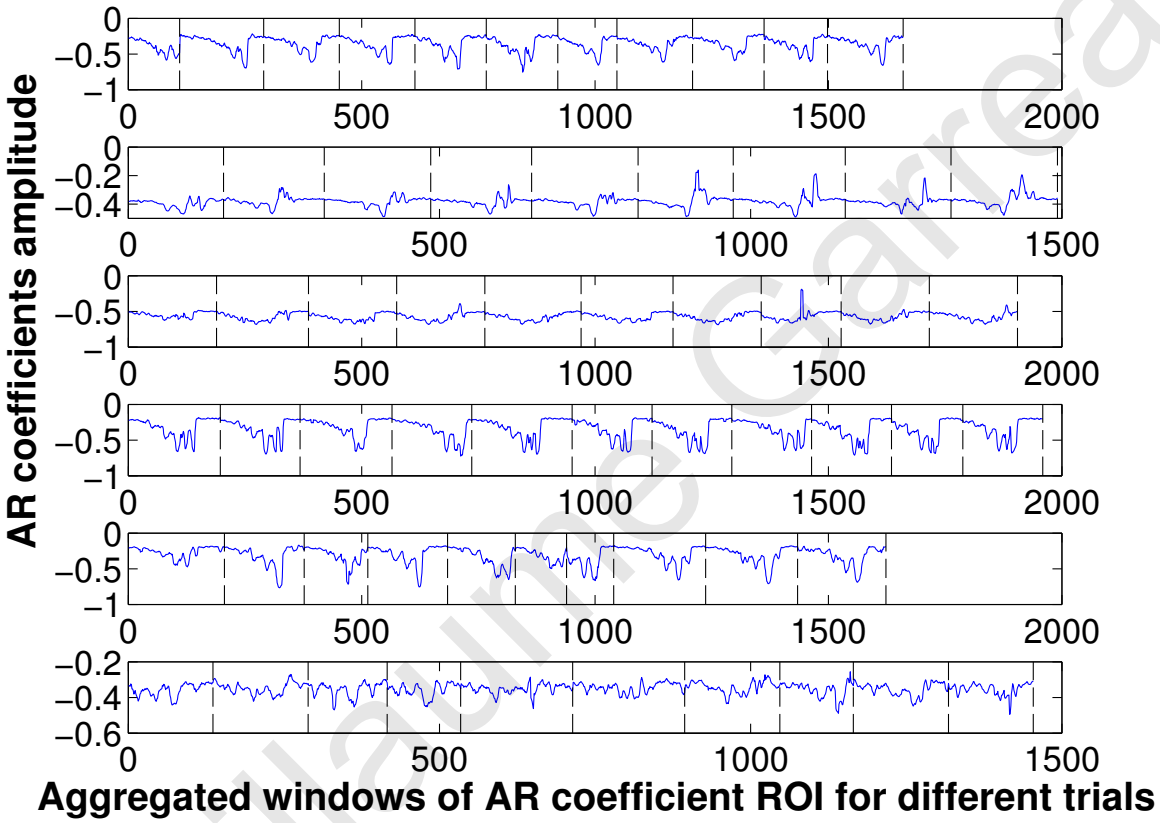


Figure 4.20: Estimated coefficient a_2 AR coefficient for walking towards the UFTRX unit. Each row corresponds to one person and many events concatenated together.

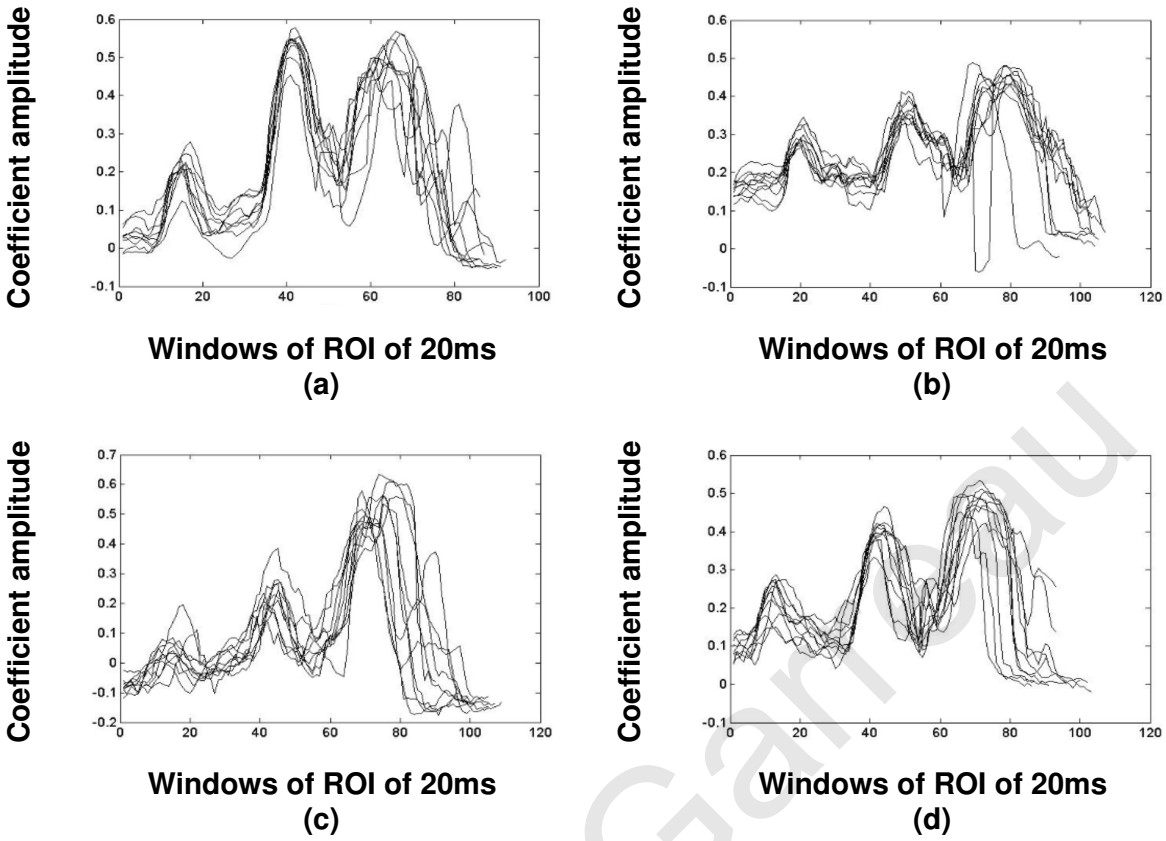


Figure 4.21: Estimated coefficient a_{10} for different occurrences of walking towards the sensor for 4 randomly chosen subjects ((a)-(d) respectively).

time delays can track the movement of individual subjects over time. Alignment of the different trials for each subjects (Fig. 4.21) reveals a pattern that remains robust with respect to time and different occurrences of the same event. Furthermore, these patterns are characteristic for each subject.

The '*icoshift*' function was then used to align the trials for each subject. An example of such alignment is shown in Fig. 4.22 for a randomly chosen subject. The top part of the figure shows the different trials for that subject. The bold line shows the reference chosen for the alignment estimated using the max of the dataset. The bottom part shows the trials after alignment. The template used for the Leave-One-Out cross-validation is obtained using the average of the trials after the alignment is done. Figure 4.23 shows the corresponding template estimated from the data shown on Fig. 4.22.

Recognition performance was obtained for each subject and AR coefficients, estimated at delays p=4,9,10.

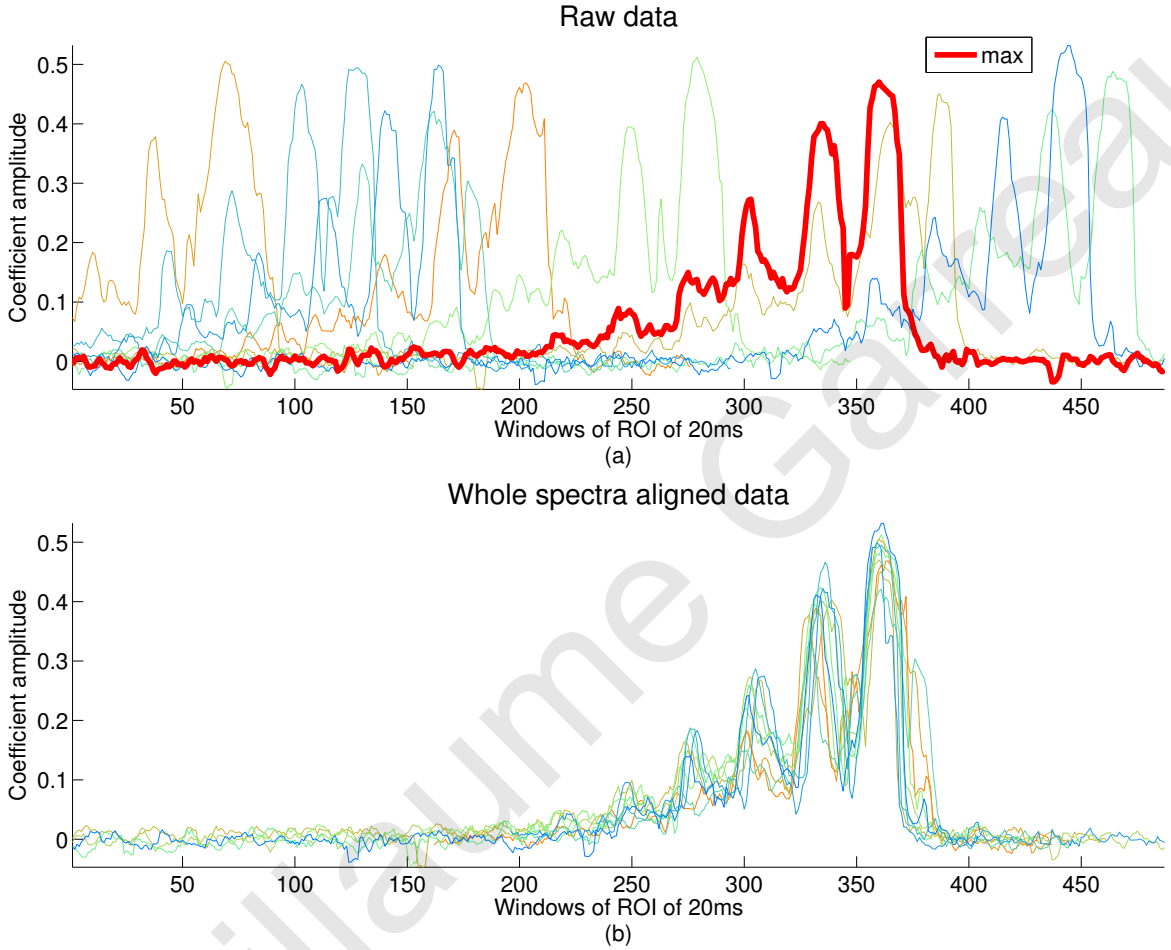


Figure 4.22: Top figure shows the a_{10} AR coefficient estimated for different occurrences of walking towards the sensor for one subject as well as the maximum (in red) of the a_{10} AR coefficient. The bottom shows the different occurrences shifted after using the *'icoshift'* function.

Template reference	max			median			average		
AR coefficient	10	9	4	10	9	4	10	9	4
Subject 1 (%)	100	100	63.6	100	100	60	100	100	60
Subject 2 (%)	88.9	88.9	44.4	100	100	44.4	100	100	44.4
Subject 3 (%)	80	90	80	90	80	80	90	80	80
Subject 4 (%)	100	100	18.2	100	100	0	100	100	0
Subject 5 (%)	100	90	0	100	100	0	100	100	0
Subject 6 (%)	70	20	20	100	20	20	90	20	20
Average (%)	89.8±12.7	81.5±30.5	37.7±30.4	98.3±4.1	83.3±32.0	34.1±32.9	96.7±5.2	83.3±32.0	34.1±32.9

Table 4.3: Average performance in individual identification Leave-One-Out classification.

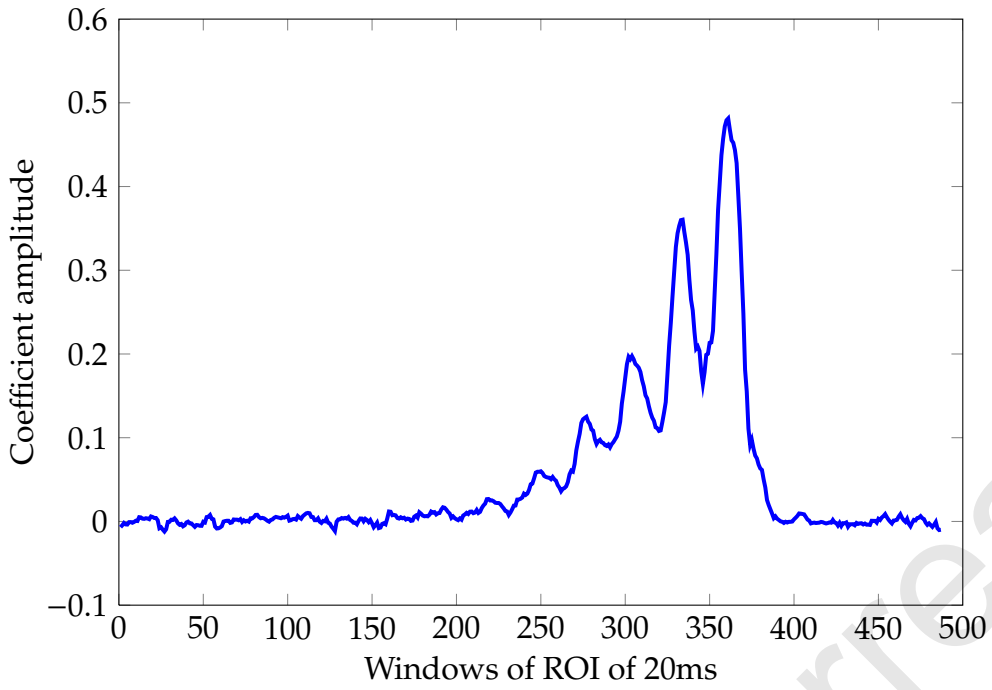


Figure 4.23: The template obtained after averaging the different a_{10} AR coefficient shifted by the '*icoshift*' function for the same set of data as in Fig. 4.22.

Performance was assessed as the average recognition accuracy (4.7):

$$P(\%) = 100 * \frac{T_{cc}}{T_{tt}} \tag{4.7}$$

where T_{tt} is the total number of 'ground truth' examples and T_{cc} is the number of 'ground truth' examples correctly classified. Performance was evaluated using the 3 different types of alignment reference for the '*icoshift*' function (maximum, median and average) and with 3 different coefficients a_p of the 10^{th} order AR model. The particular delays were chosen to illustrate best, good and worst performance. Some coefficients result in high performance (>80%, e.g. a_9 and a_{10}) while others give poor recognition (<40%, e.g. for a_4). It should be noted (Tab. 4.3) that the recognition rate shows large variations (from 0% to 100%) from subject to subject for the same parameters configuration (alignment reference and estimated coefficient of the AR model). The best result (98.3±4.1%) is obtained for a median reference for the '*icoshift*' alignment and a_{10}. When using the 9^{th} coefficient (a_9) of the 10^{th} order AR, the average performance is 83±32% and when using the 4^{th} coefficient (a_4) only 34.1±32.9% is obtained.

To explain the good performance of this method, one has to remember that during walking several body parts, whose loci are constrained with respect to one another, move in a coordinated manner in order to generate the action. The velocity if each

one of these moving parts can be used to modulate and reflect incoming ultrasonic waves. Hence, the mD signature is linked to a particular individual's body part physical constraints (related to size, flexibility, coordination etc.). Another way of viewing this is as the coupling of various oscillating body parts, thus revealing a unique time-frequency signature for each individual. Fitting an AR model to the raw mD data, over a sliding window, is equivalent to applying an adaptive low-pass filter whose coefficients change depending on the frequencies present in the signal. Thus, by tracking the estimated autoregressive coefficients over time, one can obtain a unique signature describing an individual's motion.

4.4 Conclusion

In this chapter, it has been shown that using the DACQ unit described in Chapter 2, adapted to an ultrasonic use and with different algorithms, described in section 4.2, it is possible to classify individuals, gender, actions or mode of transport using the reflected sonic wave and time-frequency representations of mD signatures.

More specifically, individuals were recognised with 87% accuracy using short temporal events learned through k-means clustering and gender was recognised with 92% accuracy, outperforming other methods reported in the literature [133,255]. Individuals were also recognized using autoregressive coefficients for the first time, with an accuracy ranging between 90% and 100%. For the first time, mode of transport was correctly classified with an 81% rate using the standard deviation of the energy spread on the spectrogram of the mD signatures and with a linear classifier. Actions were recognized with average accuracy greater than 95% using a spectrogram segmentation method that facilitates the learning of representative event templates and permits the continuous categorization of temporally short incoming events. It is the first reported used of this technique with ultrasonic data.

The above results were obtained with a relatively low number of subjects (from six to thirteen) and with a close age range (25 to 35 years old). In applications where a larger number of subjects and greater diversity is required, it may be necessary to implement more complex classifiers and use further sensory modalities. Another possible way of increasing the performance is to create a skeleton model of the human body, then generate artificial mD signatures and match artificial data with real data.

In addition, such identification can be achieved with a low-cost, low-power compact system. The algorithms and the results presented are based on evidence of that some of the principles of auditory processing in the cortex make use of spectrotemporal features and a hierarchical organization. They encourage a full hardware implementation targeting real-time application. Real-time demonstration of learning new simple actions (clapping, punching) has been successfully done on a public event during the EU Seventh Framework Programme for Research and Technological Development (FP7) funded acoustic SCene Analysis for Detection of Living Entities (SCANDLE) project workshop at the University of Plymouth in February 2012 (Making Sense of Sound 2012).

One recent trend has been to investigate and commercialize ultrasonic navigation system to help blind people [21, 100, 115, 131, 166, 176, 206]. Furthermore, work has been already done fusing neuromorphic hardware and sonar data with applications in robotics [10, 46].

Overall, such systems may have a wide range of applications in the context of robotics, security, surveillance, human behavior monitoring and even gaming.

Chapter 5

Towards an FPGA-based Auditory Pathway Implementation

5.1 Introduction

In the previous chapters, it has been shown that it is possible to extract information from the environment, such as localization of source of sound, identification of individuals or recognition of actions, solely based on acoustic waves. In primitive biological organisms, such as flies, there is evidence of acoustic scene analysis done with simple neural networks (Robert *et al.* [194,195]). Understanding and modelling these processes will allow to develop robotic systems that can perform acoustic scene analysis with better performance than current solutions.

In humans, the acoustic scene analysis is performed by the auditory system. It is composed of the peripheral auditory system and the central auditory system. If the auditory periphery does not form part of the nervous system, its components feed directly into the nervous system, performing mechano-electrical transduction of sound pressure-waves into neural action potentials (Fig. 5.1).

The auditory periphery starts with the outer ear (in green in Fig. 5.1). The pinna reflects and attenuates the incoming sound waves into the auditory canal that amplifies the 3 to 12kHz sounds. Then comes the middle ear (in red in Fig. 5.1) with the tympanic membrane. Also called eardrum, it is composed of three delicate bones that convert the lower-pressure eardrum sound vibrations into higher-pressure sound vibrations at another, smaller membrane called the oval window. At this stage occurs the first transduction of sounds from an air filled medium to a liquid filled medium of the inner ear. The inner ear consists of the cochlea and several

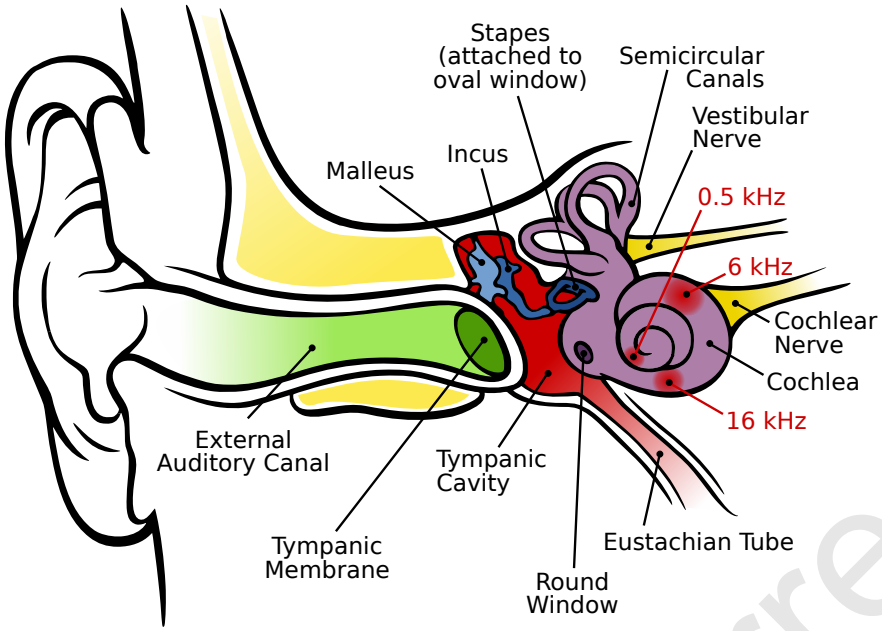


Figure 5.1: The human ear and frequency mapping in the cochlea. Information from the cochlear receptor cells is transmitted to the cochlear nuclei via the 8th cranial nerve, which then flows through the midbrain to the cortex (from [50]).

non-auditory structures (in purple in Fig. 5.1). The cochlea has three liquid-filled chambers, and supports a fluid wave driven by pressure across the basilar membrane separating two of the chambers, the scala vestibuli and scala tympani (Fig. 5.2). The organ of Corti forms a ribbon of sensory epithelium which runs lengthwise down the cochlea's entire scala media. Its hair cells transform the fluid waves into nerve signals. This is the second stage of transduction of sounds from mechanical waves to electric signals in neurons. Because of the mechanical properties of the basilar membrane within the snail-shaped cochlea, high frequencies will produce a vibration peak near the oval window, whereas low frequencies will stimulate receptors near the apex of the cochlea (locations for three frequencies indicated schematically on Fig. 5.1).

The central auditory system re-encodes the sound information and sends it to the vestibulo-cochlear nerve, through intermediate stations such as the cochlear nuclei and superior olivary complex of the brainstem and the inferior colliculus of the midbrain, being further processed at each waypoint. The information eventually reaches the thalamus, and from there it is relayed to the cortex (Fig 5.3).

Although the cochlea has been extensively studied, there are still mechanisms

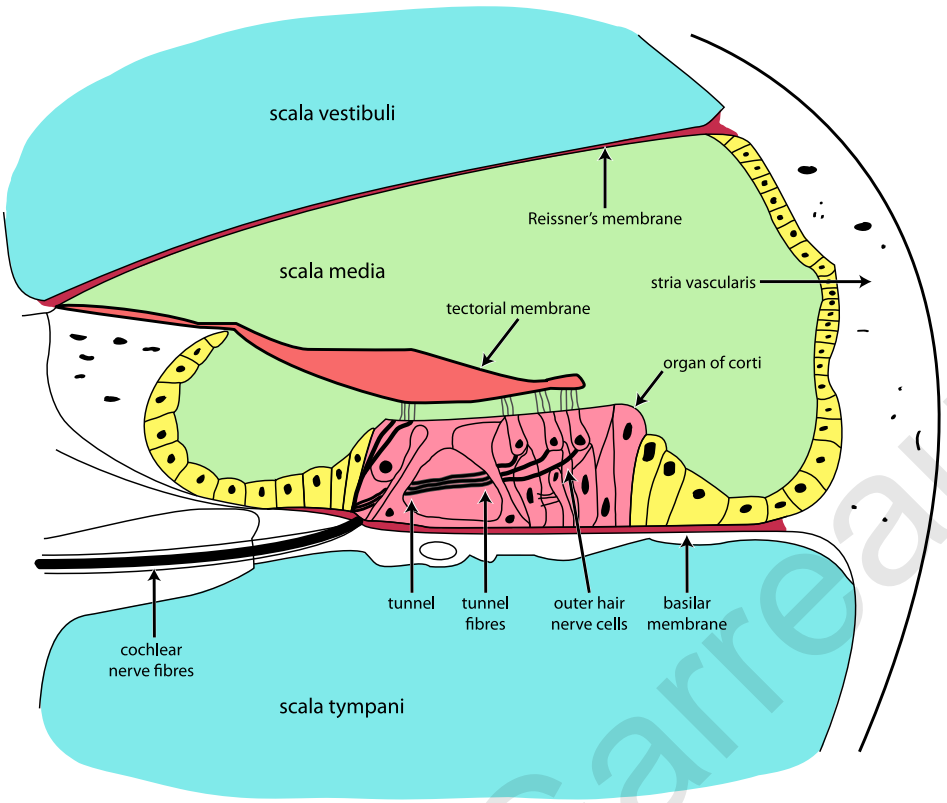


Figure 5.2: Cross section of the cochlea (from [244]).

that have yet to be fully understood. For example, the precise location of the cochlear amplifier in living sensitive ears has only been recently demonstrated experimentally (2011). The cochlea amplifier resides at a small longitudinal region basal to the response peak in the sensitive cochlea (Ren *et al.* [192]). This data provides critical information for advancing our knowledge on cochlear mechanisms responsible for the remarkable hearing sensitivity, frequency selectivity and dynamic range. Moreover, there is no commonly accepted theory of Interaural Time Delay (ITD) based localization that explains both the behavioral and neural data for the processing of fine temporal differences between the incoming signals at both ears for azimuthal sound source localization. A unifying theory was proposed by Benichoux *et al.* in 2011 for the barn owl [13] and in 2013 for mammals [14].

A lot of work has been published on acoustic scene analysis; some representative examples are presented, in order to put the contributions of this thesis in context with prior work. For a more detailed review see a historical review by Leong *et al.* [149].

The first and oldest group of studies is the one using classical or bioinspired algorithms for signal processing. The second and more recent group is using neural network processing with software or hardware processing.

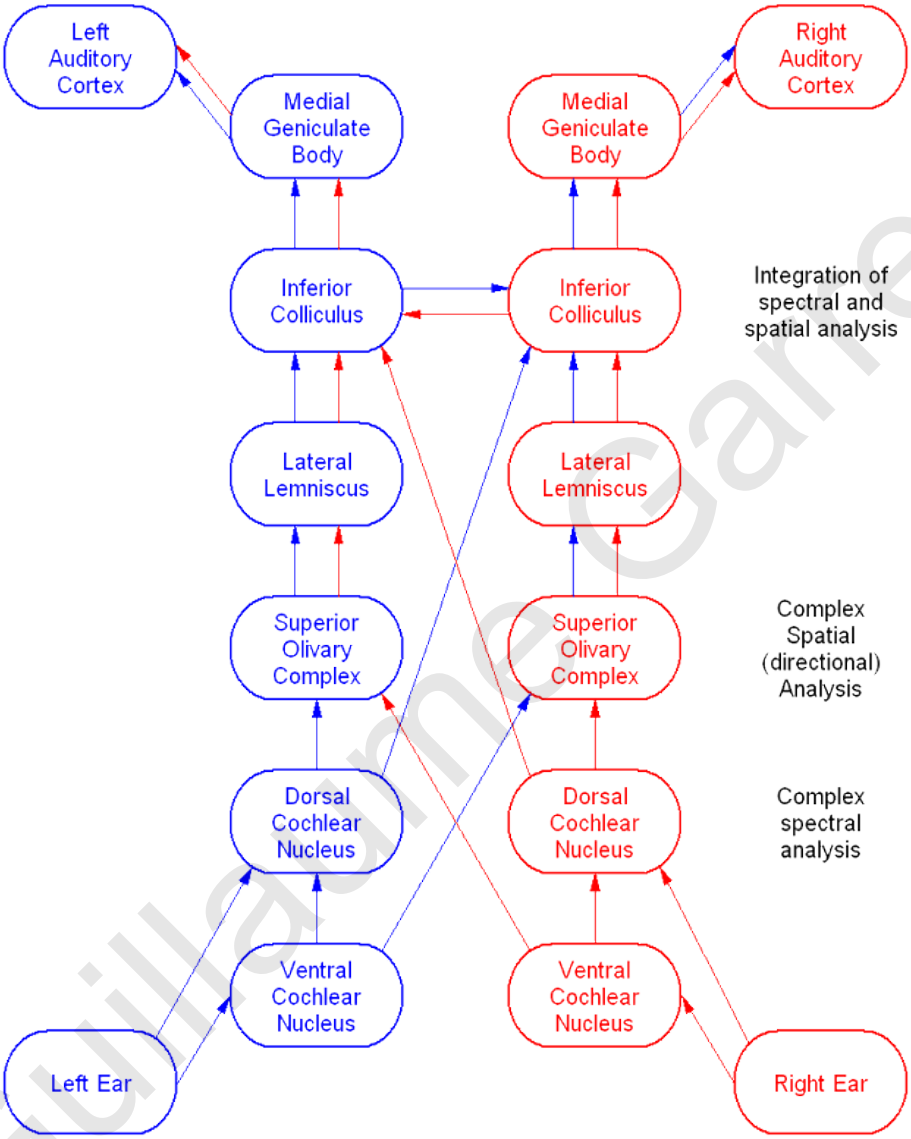


Figure 5.3: An overview of afferent ipsilateral and contralateral interactions in the auditory brainstem (from [165]).

In the first group, some studies focus on single sound source localisation. Julian *et al.* compared four different algorithms, for sound localization, using MEMS microphones and signals recorded in a natural environment [132]. The spatial-gradient algorithm (SGA) gave the best results. Its implementation requires a sampled data analog architecture able to adaptively solve a standard least mean-square (LMS) problem. Masson *et al.* used a data fusion algorithm to perform the estimation of the position based on measurements from five nodes, each having four MEMS microphones [168]. The measurement is made using a unique fixed source emitting a 1kHz signal. Zhang *et al.* used cross correlation of the signals received and a zero crossing point to estimate the bearing angles of moving vehicles [254]. The hardware was an ASU with four MEMS microphones, that required an acoustic horn.

Other studies used techniques for separation of multiple sound sources. Prior work for bioinspired (from flies) acoustic surveillance units, such as that of Cauwenberghs *et al.*, used spatial and temporal derivatives of the field over a sensor array of MEMS microphones, power series expansion and Independent Component Analysis (ICA) for localizing and separating mixtures of delayed sound sources [44]. It showed that the number of sources that can be extracted, depends strongly on the number of resolvable terms in the series. Similar work was also done by Sawada *et al.* using ICA for estimating the number of source of sound [121] and localization of multiple sources of sound [121, 203, 204].

Finally, the most interesting application of acoustic scene analysis is the detection and classification of real-world acoustic events and audio scenes or soundscape (for instance a train station or a classroom). Andringa *et al.* addressed the problem of recognizing acoustic events present in an unconstrained input. The authors proposed a novel approach to the processing of audio data, which combined bottom-up hypothesis generation with top-down expectations, which unlike standard pattern recognition techniques, can ensure a physically realizable representation of the input sound [6]. This was used in verbal aggression detection in complex social environments [228] and with the Sensor City Sound project in the Dutch city of Groningen (`www.sensorcity.nl`). Brown *et al.* reviewed the principles underlying Acoustic or Auditory Scene Analysis (ASA) and showed how they can be implemented in computational ASA (CASA) systems [27]. They linked CASA and automatic speech recognition, and drew distinctions between the CASA and ICA approaches.

The reported algorithms have demonstrated good results at the expense of high

computational cost in comparison to using a neural network approach. The advantages of neuron network computation compared to other digital algorithms are discussed in [109, 183].

The second group of studies works on the same applications, but adding the constraint of neural network processing. Identification of acoustic signals in noisy environments remains one of the most difficult signal processing problems and Dibazar *et al.* proposed a hardware implementation of a novel biologically based Dynamic Synapse Neural Networks (DSNN) for Acoustic Sound Recognition [64]. The hardware, named acoustic SENTRI is based on Texas instrument TMS320C6713 DSK and a custom designed data acquisition board. An array of four microphones is attached to the board for sound input.

Other solutions have an artificial cochlea, which is used as first stage of the signal processing. Hardware implementations of electronic cochlea models have traditionally used analog VLSI as the implementation medium due to their small area, high speed, and low power consumption. The first analog electronic cochlea was designed by Lyon and Mead [159, 160]. Van Schaik *et al.* proposed a silicon cochlea using compatible lateral bipolar transistors [232] and a pseudo-voltage domain implementation of a 2-dimensional silicon cochlea [229]. They were used for a neuromorphic sound localizer with two MEMS microphones are connected to two analog neuromorphic cochlea. The system is low power and the average standard deviation of the estimated azimuth is 4.5° [231, 233]. The system bases its azimuth estimation on the interaural delay between the two 'ears'. An AER module was also added to the cochlea for sound processing for identification [155, 230] or source localization using ITD [47].

Note that groups have also been working on implantable systems to help hearing-impaired patients, such as the 126μW cochlear chip for speech processor/stimulator developed by Georgiou and Toumazou [102].

Other groups proposed digital cochlea on FPGAs for several reasons: FPGA-based implementations offer shorter design time, improved dynamic range, higher accuracy, and a simpler computer interface. Leong *et al.* presented a module generator that can produce an FPGA-based implementation of an electronic cochlea filter with arbitrary precision [149]. The authors show the cochleagram resulting from an implementation example with 88 infinite impulse response (IIR) filters covering a frequency window of 0Hz to 16khz. Clarke *et al.* presented an implementation of

neuromimetic cochlea for a bionic bat head using a virtex II FPGA [51]. Gambin *et al.* proposed a cochlea with a cascade of 24 filters covering the frequency range 20Hz to 20kHz on a Spartan3 FPGA [93]. Mugliette *et al.* extended the previous work with automatic gain control [179].

Currently a full auditory pathway hardware implementation of a cochlea with advanced signal processing and learning capabilities is a target avidly pursued. The work presented here is a first step towards the implementation of such a system based on a compact neural network solution using a single FPGA. An FPGA approach is chosen as it has a shorter development time and higher flexibility compared with an analog VLSI approach. In addition, it was demonstrated that very large neural population can be implemented on FPGA (Cassidy *et al.* demonstrated that 1 million neurons can be implemented on a single Spartan Virtex5 FPGA [38]). In the next section, a model of the cochlea is presented, it was developed by Liu *et al.* [156]. Then a simplified model of the auditory thalamocortical (TC) network developed by Coath *et al.* [53] is presented. It is followed by the latest progress towards a full implementation of the cochlea (32 channels, covering the 500Hz to 16kHz) and the simplified TC model on a single FPGA. Finally, the latest progress in implementing the same model using an hybrid software and analog/digital VLSI implementation [54,207] is presented and compared with our partial implementation of the TC model.

5.2 Bioinspired model literature

The hardware implementation of the auditory pathway presented here is composed of two main parts. The first part is the filter bank simulating the cochlea and the second part is the thalamocortical (TC) network.

5.2.1 Cochlea filter bank

Several models of cochlea were evaluated, they gave similar results and thus it was decided to implement the cochlea model of Liu *et al.* [156]. This model gives a good trade-off between complexity and accuracy. It consists of a cascade of first-order, low-pass IIR filters, the majority of which branch into cascades of two second-order, band-pass IIR filters. A diagram of the model is provided in Fig. 5.4. The inputs are

fed at the left-hand side of the diagram, which represents the base of the cochlea; the outputs are read from the taps along the bottom of the diagram, which represent points spaced along the cochlea between its base and apex. Note that, as in the natural cochlea, these series of low-pass filters start with a high bandwidth that gets progressively lower.

The core of cochlea is composed of 32 sensory channels. Each channel is composed of one first-order low-pass IIR filter (P). It branches out into the next channel P filter and a cascade of two second-order band-pass or biquadratic IIR filters (Q and R). The output of the R filter is then rectified and used as an input to the first stage of the TC model.

The core is preceded by a cascade of three first-order low-pass IIR filters (S) used for pre-emphasis of the signal. The cochlea is specified using six parameters: the channel number n_{ch}, the sampling interval d_t, the resonator frequency at the base f_{base}, the resonator tuning at the base q_{base}, the resonator frequency at the apex f_{apex} and the resonator tuning at the apex q_{apex}. The last four parameters are illustrated in Fig. 5.5.

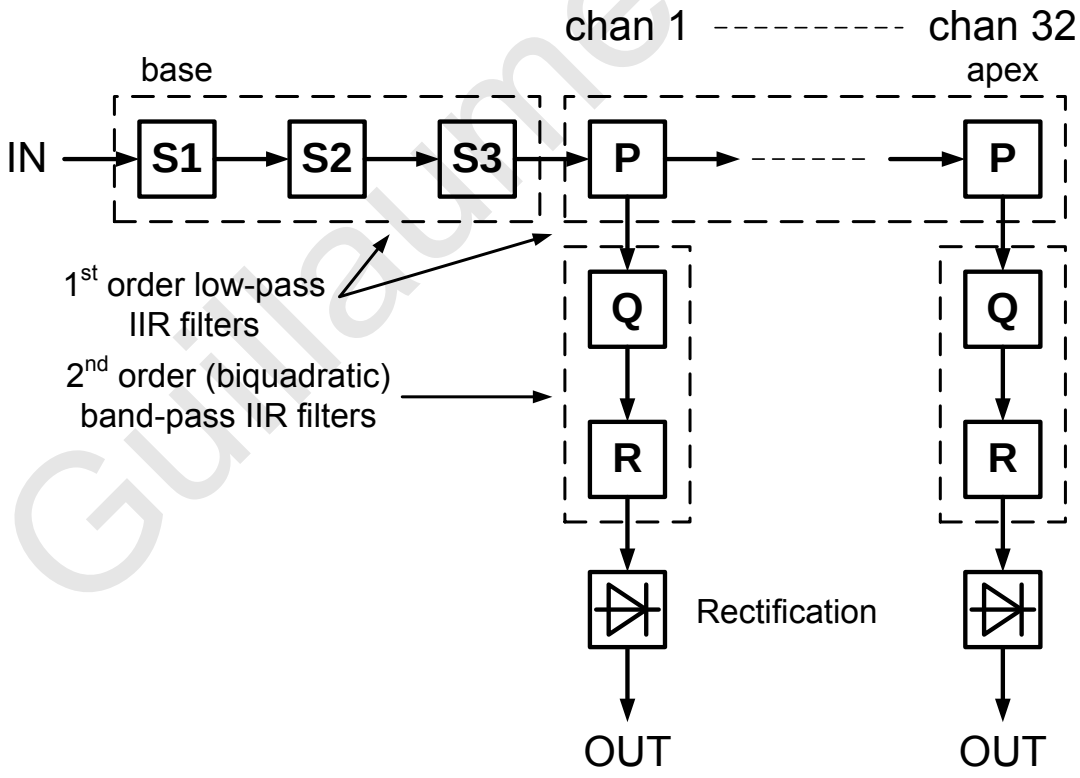


Figure 5.4: Diagram of the model of the cochlea.

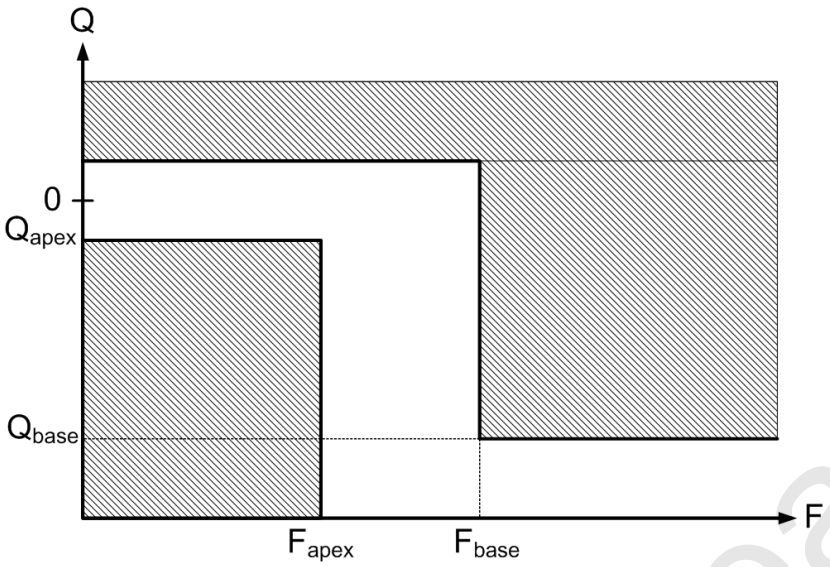


Figure 5.5: Filter amplitude response requirements.

For each channel $n_{ch} = 1,...,32$ the center frequency f and the attenuation coefficient Q are calculated as follows:

$$\omega_{n_{ch}} = \omega_{n_{ch}-1} * \delta f_{inc} \tag{5.1}$$

$$\delta f_{inc} = \left(\frac{f_{apex}}{f_{base}}\right)^{\frac{1}{n_{ch}-1}} \tag{5.2}$$

$$q_{n_{ch}} = q_{n_{ch}-1} + \delta q_{inc} \tag{5.3}$$

$$\delta q_{inc} = \frac{q_{apex} - q_{base}}{n_{ch} - 1} \tag{5.4}$$

The initialization values for the the first channel are:

$$\omega_0 = 2 * \pi * f_{base} \tag{5.5}$$

$$q_0 = q_{base} \tag{5.6}$$

The next stage of the auditory pathway, after the cochlea, is the thalamocortical model.

5.2.2 Thalamocortical model

The thalamocortical model used in this work is a simplified version of the auditory thalamocortical system model developed by Coath *et al.*. A detailed description of the simplified version is given in this chapter. However, for more details on the full model, please refer to [52,53]. The simplified version is the result of discussions with the original model inventors. The simplified version allows a faster implementation and gives good results in accordance with the prediction given by the full TC model.

The simpler version of the thalamocortical network was developed to target hardware implementation, it is shown in Fig. 5.7. The network has 3 layers of neurons, labelled A, B_1 and B_2. The TC model includes excitatory projections from B_1 to B_2 which are plastic and exhibit Spike Timing Dependent Plasticity (STDP), a spike-based Hebbian learning rule first reported by Caporale *et al.* [35]. This rule potentiates the synapses where an apparent cause-effect relationship predominates over time, in other words it learns the temporal correlations in the time-varying stimulus. A modified version of the STDP rule is implemented here, it was developed by Brader *et al.* [22]. The synaptic update rule adjusts the synaptic weight X upon arrival of a pre-synaptic spike, depending on the instantaneous membrane potential and the internal state of the post-synaptic neuron [22, 91]. The internal state $C(t)$ is identified with the post-synaptic neuron calcium concentration, driven by firing of the neuron (Shouval *et al.* [210]). The synaptic efficacy X is altered according to:

$$X = X + a; \text{ if } V(t) > \theta \text{ and } \theta^l_{up} < C(t) < \theta^h_{up} \tag{5.7}$$

$$X = X - b; \text{ if } V(t) \leqslant \theta \text{ and } \theta^l_{down} < C(t) < \theta^h_{down} \tag{5.8}$$

with $V(t)$ is the post-synaptic membrane voltage, θ the learning threshold. If neither of the conditions are satisfied, then X drifts to one of two stable states 0 or 1, depending on whether $X(t)$ is above or below the threshold, θ_X:

$$\frac{dX(t)}{dt} = \alpha; \text{ if } X(t) > \theta_X \tag{5.9}$$

$$\frac{dX(t)}{dt} = -\beta; \text{ if } X(t) \leqslant \theta_X \tag{5.10}$$

The internal state $C(t)$ is driven by firing of the neuron and is governed by:

$$\frac{dC(t)}{dt} = -\frac{C(t)}{\tau_C} + J_C \sum_i \delta(t - t_i) \tag{5.11}$$

where J_C is the amount of calcium contributed by a single spike. The synaptic dynamics are illustrated in Fig. 5.6.

The network behavior is as follow. For each one of the 32 channels of the cochlea, the output of the channel is fed into the neuron in layer A. The I&F neurons of layer A spike depending on their input. The spiking activity results in the synaptic conductance being modified for synapse A-B_1 and synapse A-B_2. The synaptic responses are modelled by an alpha profile. Also note that synapse A-B_1 weights are slightly higher than those of synapse A-B_2. This is to allow initial inhibition

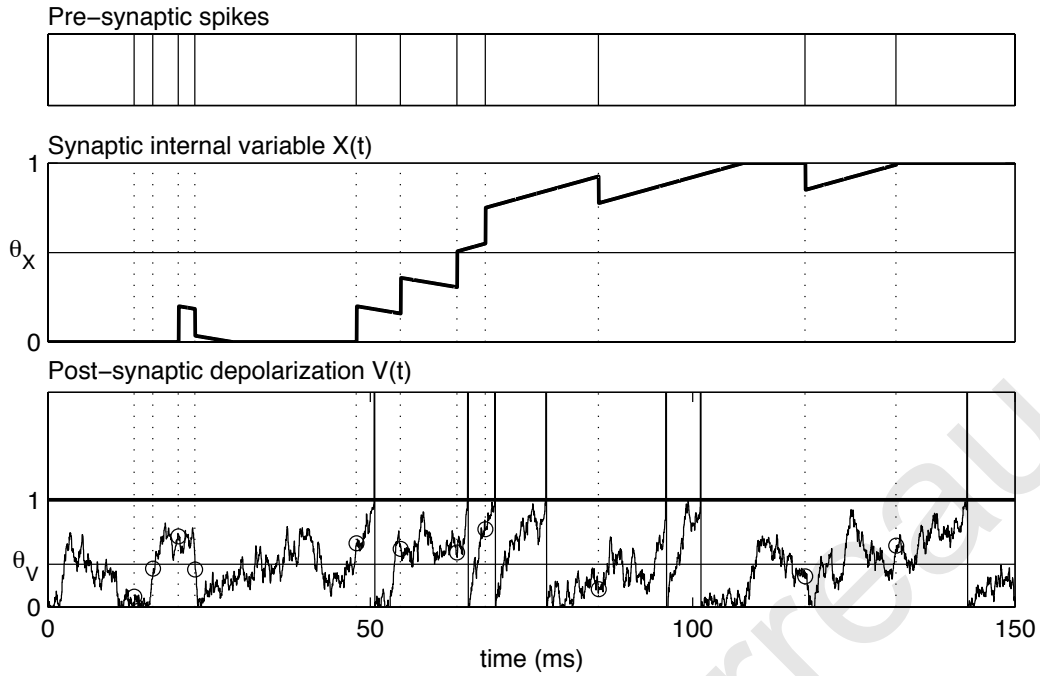


Figure 5.6: Synaptic dynamics: time evolution of the simulated internal variable $X(t)$ describing the synaptic state (center), pre-synaptic spikes (top) and depolarization $V(t)$ of post-synaptic neuron (bottom, from [92]).

of the B_2 neurons, until the delayed projections via the excitatory STDP potentiate the B_2 neurons. The activation of the synapses A-B_1 leads to the change in the B_1 neuron membrane potential, which causes them to spike. This leads to the inhibitory synapse between B_1 and B_2 to be active thus providing an inhibitory input to the B_2 neuron within the same column, but also excitatory STDP inputs to other columns, via delayed projection B_1-B_2. Columns that are close by have their membrane potential increased since the combination of the input from synapse A-B_2 and the excitatory STDP input from B_1-B_2 makes the B_2 neuron membrane potential increase and spike, hence increasing the excitatory efficacy of the STDP input. Where no compound activity of the synapse A-B_2 was present the STDP input's inhibitory efficacy increased. Finally the acoustic stimuli causes adaptation of the synaptic weights following the Fusi-Brader learning rule [22, 91, 92]. The stimulus-driven modifications of the network connectivity result from the interaction between the stimulus itself and the spike-based plasticity (STDP) rule adopted for the delayed feed-back connections. After learning, the firing patterns of the neurons reflect the emergent connectivity, which means that neurons will fire more during presentation of learned stimuli, i.e. the network is tuned to the stimulus properties.

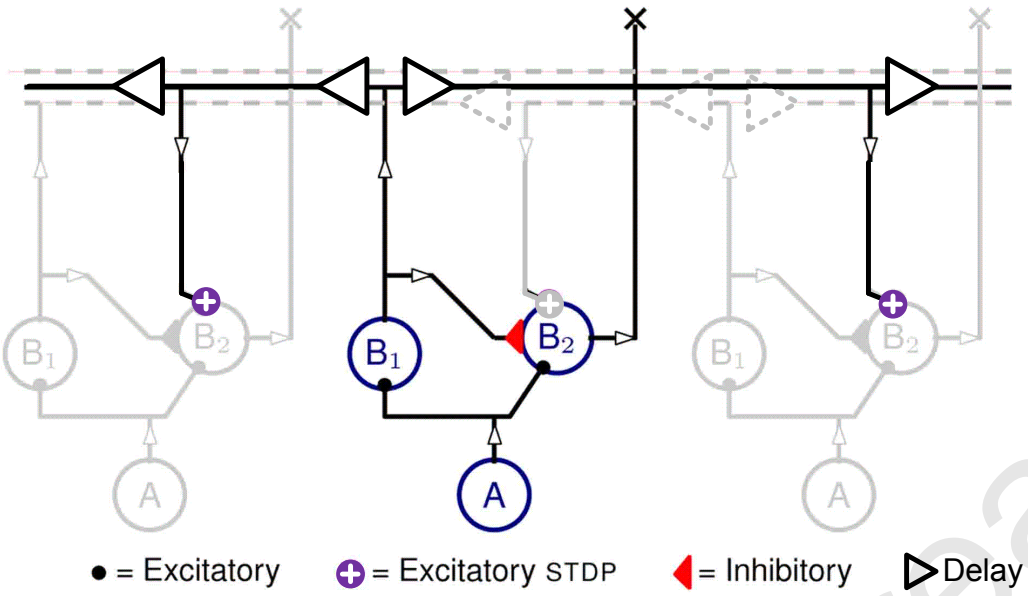


Figure 5.7: The figure shows a simplified Thalamocortical Neural Network that is fed with voltage spikes coming from a cochlea model. Each "A" neuron gets stimulated by tonotopic inputs. Only 3 columns are shown, however the targeted final implementation has 32 columns.

5.3 Data collection

The cochlea and the TC model of Coath *et al.* have already been fully implemented in software (in C with an interface to Matlab®) by the developers, the idea here is to extend this implementation to the field of the FPGAs. For this, it is needed to bridge the gap between the behavior of the mathematical models in the auditory periphery system and the limitations in hardware. To be more specific, mathematical models allow one to describe the behavior of the real world at different levels of realism. However, such realism implies a high complexity in the design of hardware architectures. For this reason the perfect balance between the models that will be used and the specific hardware (i.e. FPGAs) needs to be found.

The first step towards the full implementation on FPGA was to use Handel-C, a high level programming language which targets low-level hardware, such as FPGAs. Handel-C allows to write code in a language very close to C and then compile it into VHDL (to be later implemented on FPGA, however it is not an optimal implementation). It can be used as well to simulate the hardware implementation and thus accelerate the development of an FPGA-based full implementation of the cochlea and simplified TC model.

This section will give the details of the latest progress towards the full hardware implementation.[1] Firstly, the results of the hardware simulation of the full implementation of the cochlea and the simplified TC model using Handel-C are presented. Secondly, the cochlea hardware implementation will be compared with the software version of the model and the hardware simulation. Thirdly, the latest progress towards the full hardware implementation of the simplified TC model will be shown. Finally, the partial FPGA implementation will be compared with the analog VLSI implementation reported by Sheik *et al.* in [207] and Coath *et al.* in [54].

5.3.1 Hardware simulation of full implementation with Handel-C

For the hardware simulation of the full auditory pathway model the number of channels for the cochlea and the number of columns for the simplified TC model is 30. The scripts used are given in Appendix F. The following cochleagram and plots illustrate the simulation with Handel-C of an FPGA based implementation of the full simplified auditory pathway (cochlea and simplified TC model). The simplified TC model neural network is initialised with transmission/propagation delays between each neuron B_1 and the neurons B_2 of every other column using the delay matrix shown in Fig. 5.8, where 0ms correspond to the case of no connection between B_1 and B_2 (i.e. neurons in the same column). The assumption is that it takes more time to travel further away. Also, the STDP synapse weights are all initialised to the same value (Fig. 5.9). The aim here is to show the modification of the weights of the STDP synapses after presentation of a particular stimulus in order to validate that the simplified TC model neural network is working as expected and can be implemented in hardware.

The cochlea, presented in the previous section, is stimulated with two different mixtures of frequencies, which are non-overlapping in time. The resulting cochleagram is shown in Fig. 5.10. This is fed into the simplified TC model, presented in the previous section, through the layer A, whose membrane potential is modified as shown in Fig. 5.11. This layer contains I&F neurons which are driven by the acoustic power in each frequency band from the cochlea and spike depending on their input to produce a raster plot as shown in Fig. 5.12. The spiking activity results

[1] The work presented here was done at UCY in collaboration with Adrian Romiński (cochlea) and Horacio Rostro-Gonzalez (cochlea and TC model).

in the synaptic conductance being modified for synapse A-B_1 and synapse A-B_2 as shown in Fig. 5.13. The activation of the synapses A-B_1 leads to the change in the B_1 neuron membrane potential Fig. 5.14. The incoming inputs from A are integrated by the integrate-and-fire (I&F) neuron equation, which causes the B_1 neuron to spike as shown on the raster plot of Fig. 5.15. The spikes are projected through inhibitory synapses to the post-synaptic layer B_2. The spikes are also projected to B_2 in other locations through plastic synapses with transmission/propagation delays (Fig. 5.8). This leads to the inhibitory synapse between B_1 and B_2 to be active (Fig. 5.16) thus providing an inhibitory input to the B_2 neuron within the same column, but also excitatory STDP inputs to other columns. Columns that are close by have their membrane potential increased since the combination of the input from synapse A-B_2 and the excitatory STDP input from B_1-B_2 makes the B_2 neuron membrane potential increase (Fig. 5.17) and spike (Fig. 5.18), hence increasing the excitatory efficacy of the STDP input. Where no compound activity of the synapse A-B_2 was present the STDP input's inhibitory efficacy increased as shown in Fig. 5.19. Finally in Fig. 5.20 one can see how the two acoustic stimuli caused adaptation of the synaptic weights following the Fusi-Brader learning rule.

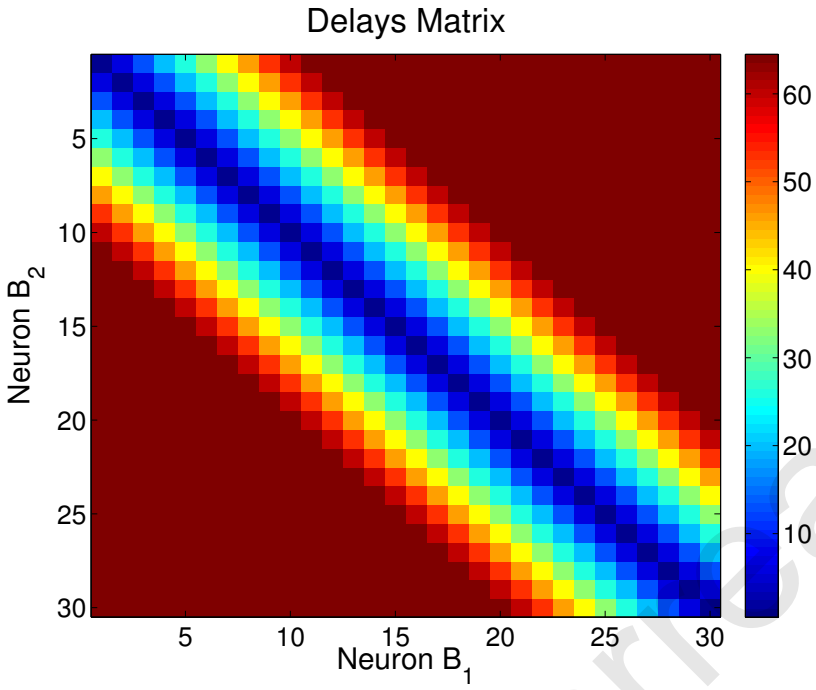


Figure 5.8: This figure shows the delayed projections from neurons in layer B_1 to neurons in layer B_2. The scale of the delays is represented in milliseconds, where 0ms (dark blue) correspond to the case of no connection between B_1 and B_2 (i.e. neurons in the same column).

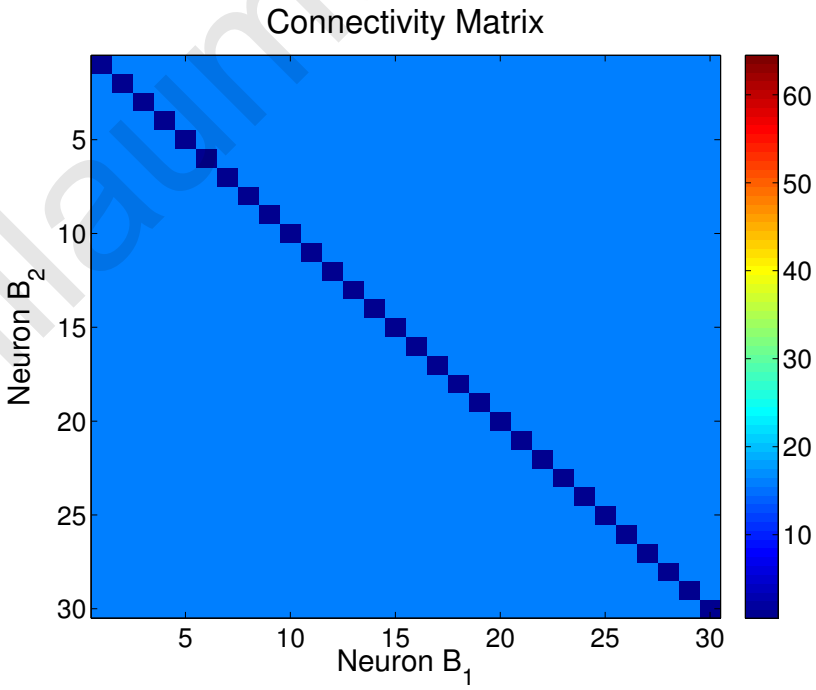


Figure 5.9: Connectivity matrix. This figure shows the initial strength of synapses between B_1 and B_2. The initial lack of connection within each particular column is presented in dark blue. All the other connections are assumed to be of same weight.

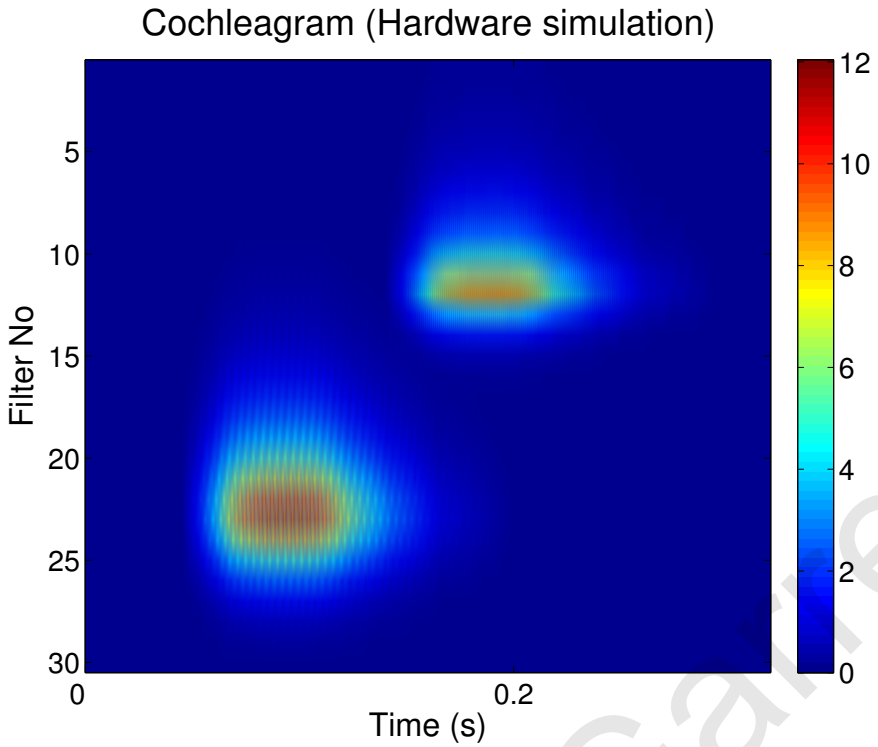


Figure 5.10: This figure shows the power output of a 30 channels cochlear model [156], which is used as a stimulus for the 30 columns of the simplified TC model.

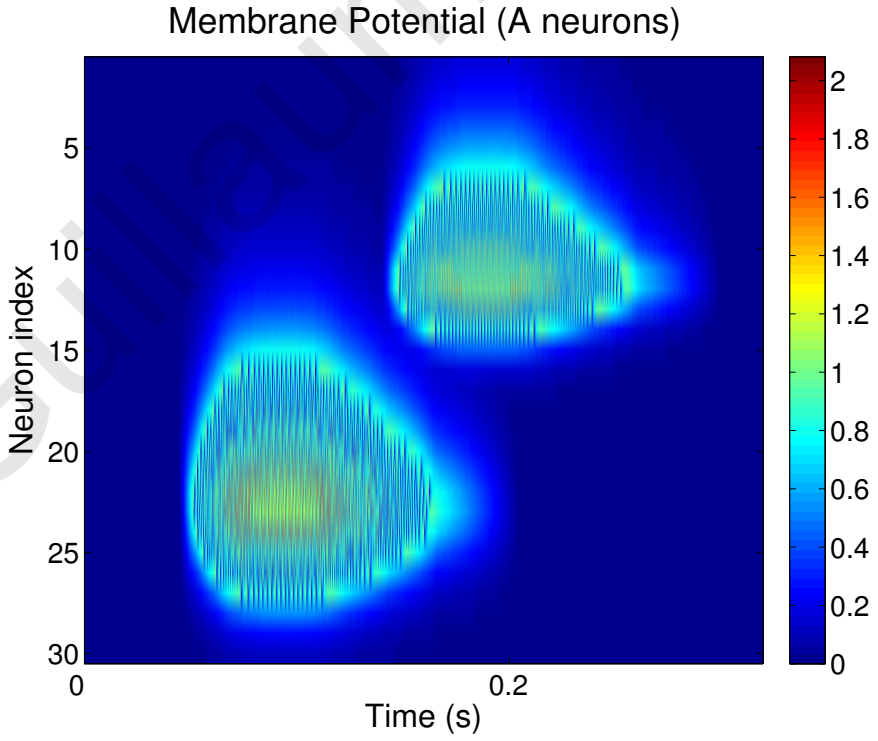


Figure 5.11: Membrane potential in layer A.

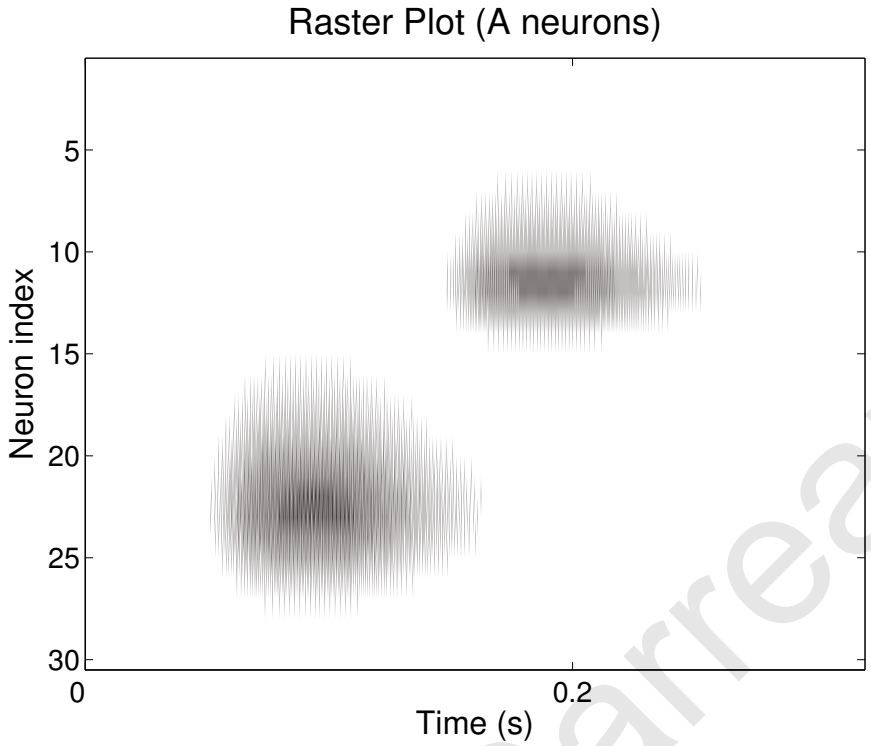


Figure 5.12: Spiking activity in layer A. This figure shows the response (represented as spikes) of neurons in layer A, which are propagated through excitatory synapses to layers B_1 and B_2.

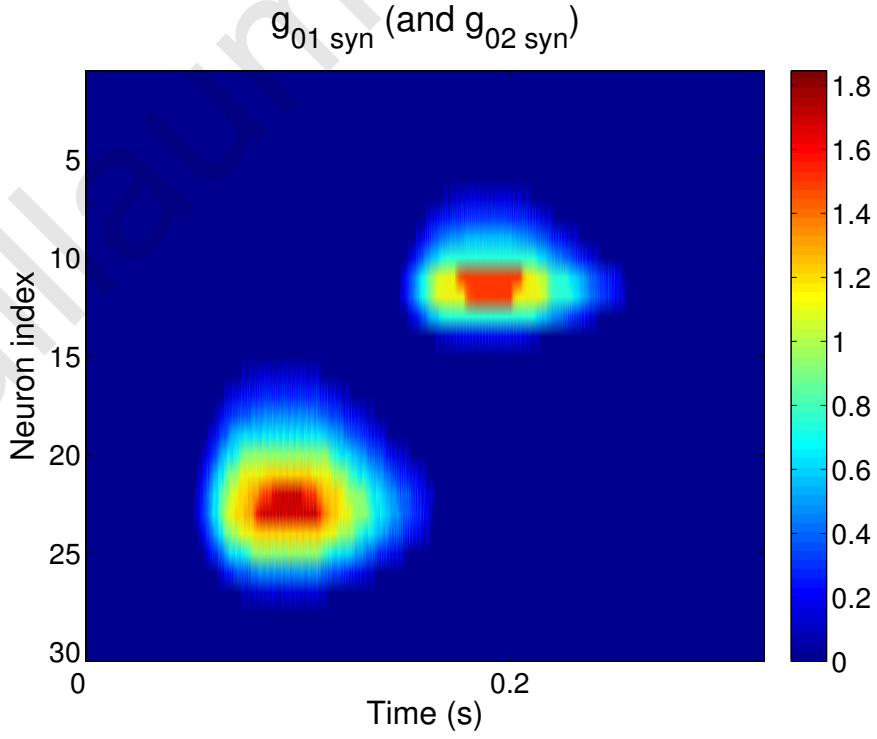


Figure 5.13: Synaptic response in A-B_1 and A-B_2. Here, the synapses are modeled by an alpha profile.

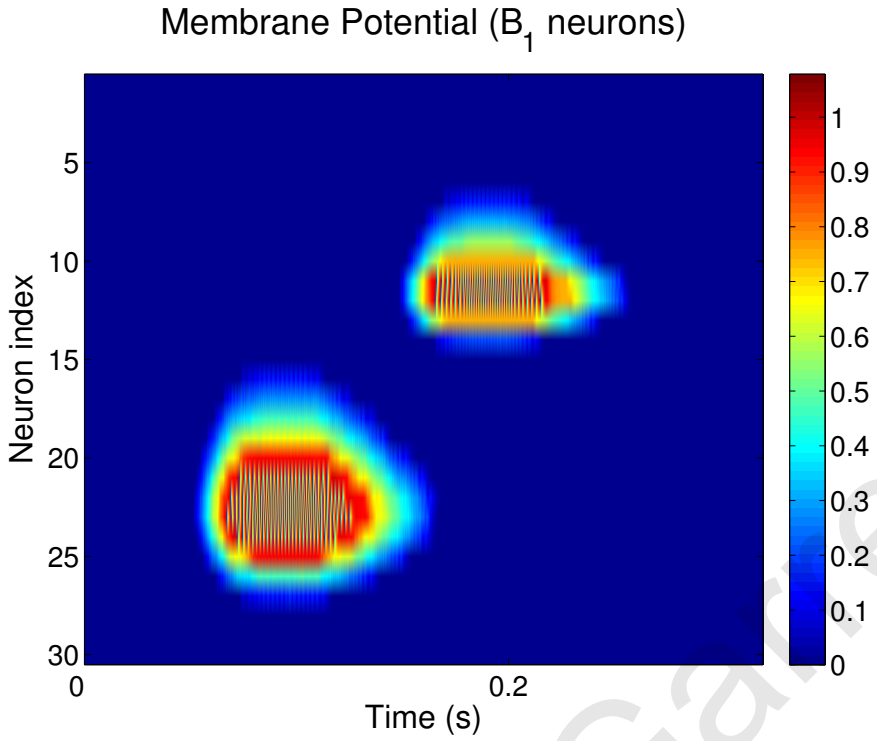


Figure 5.14: Membrane potential in B_1.

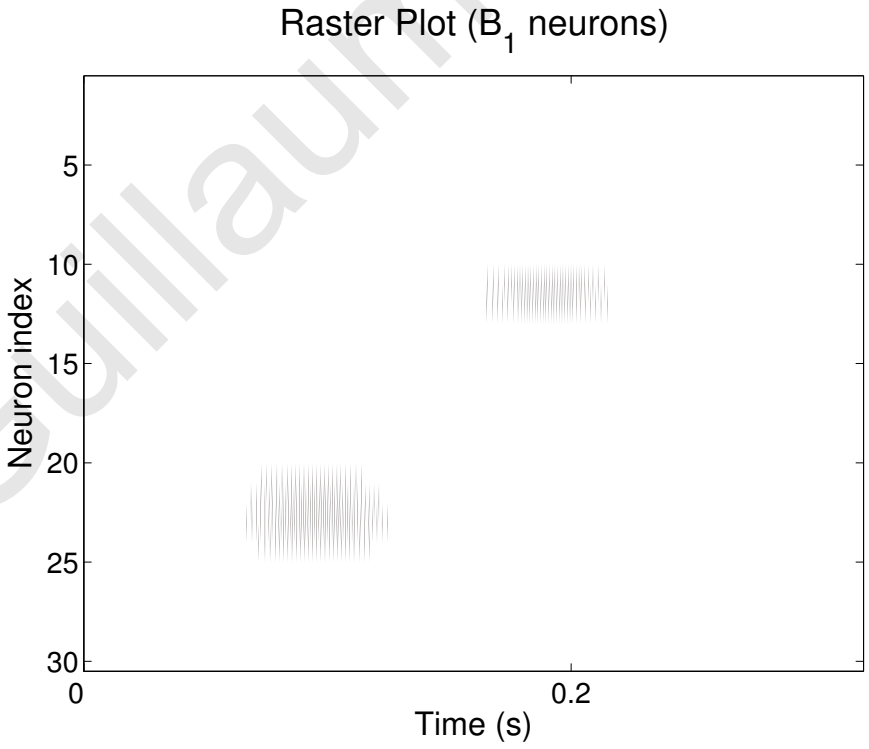


Figure 5.15: Spiking response in B_1.

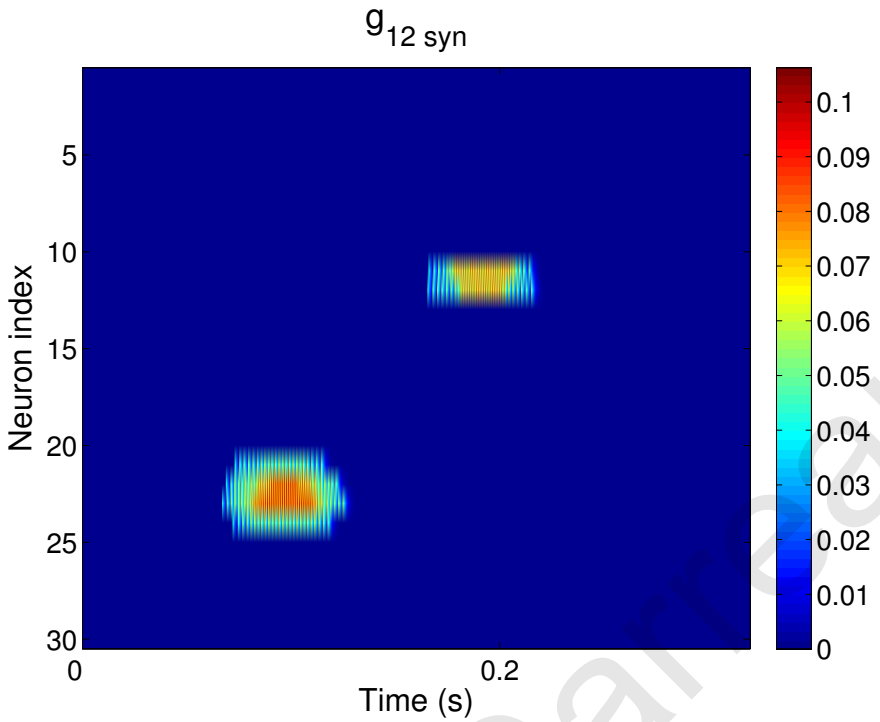


Figure 5.16: Synaptic response in B_1-B_2, which shows the response of the synapses with an alpha profile, which inhibits neurons in layer B_2.

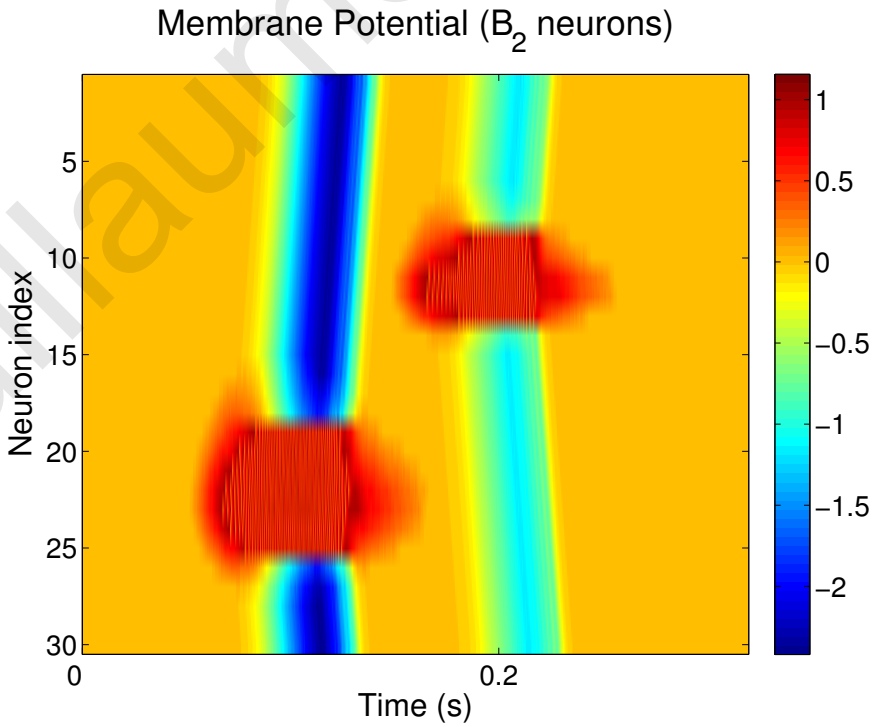


Figure 5.17: Membrane potential in B_2, which shows the response of neurons in layer B_2 from the integration of incoming inputs from A, B_1 and recurrent connections.

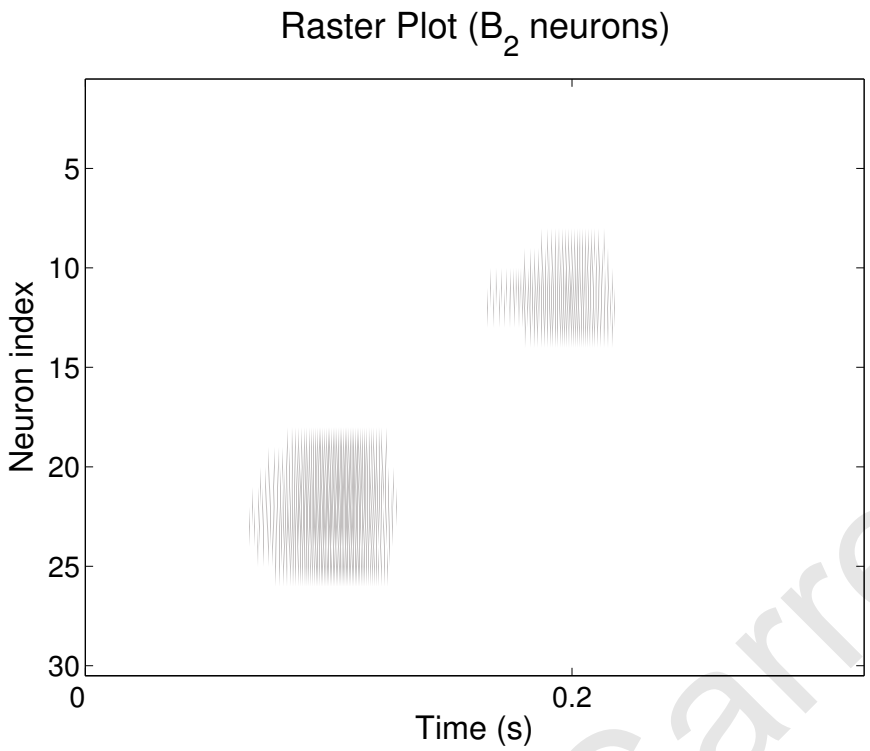


Figure 5.18: Spiking response in B_2. This figure shows the resulting dynamics of the 30 neurons in the thalamocortical network.

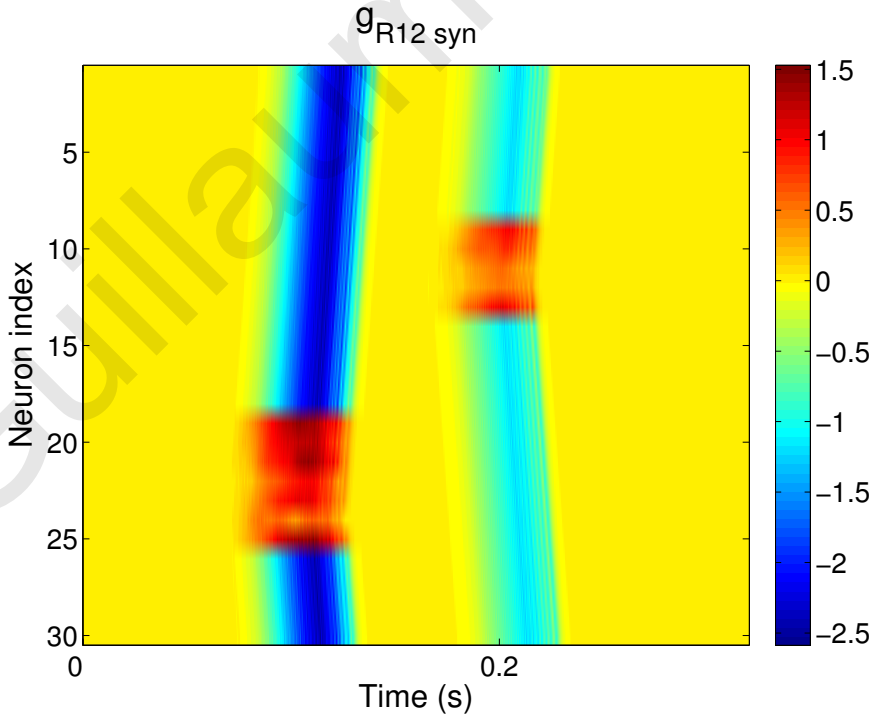


Figure 5.19: Synaptic response among the recurrent connections B_1-B_2. Here, we can observe the effects of the transmission delays in the network.

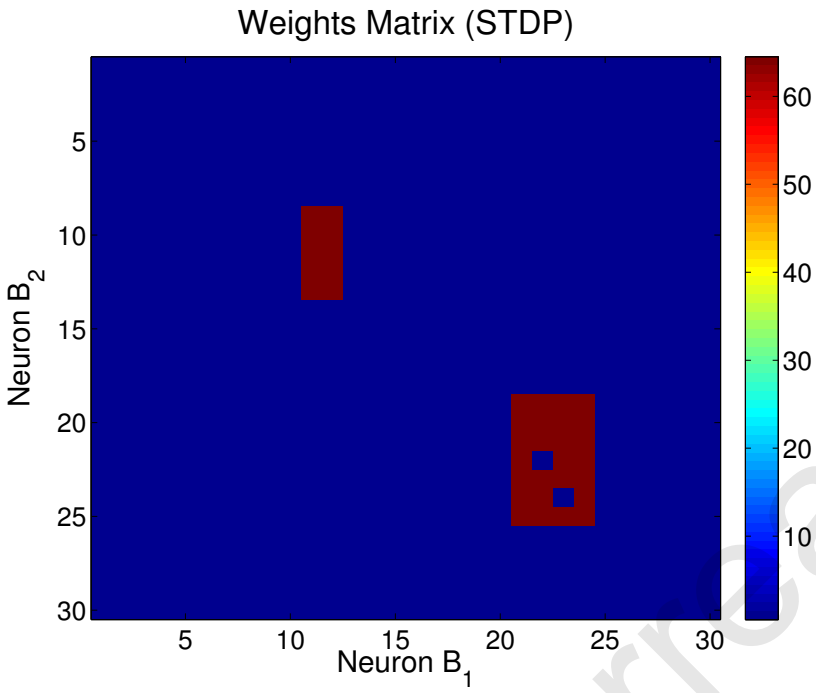


Figure 5.20: Synaptic weights after STDP. The weights are adapted in the network following the Fusi-Brader learning rule.

It is now proven that the simplified model is able to adapt the STDP synapses to a particular input stimulus. As a result, differences in the response of the network induced by learning can be used to distinguish similar stimuli that have parametrically different dynamic properties. The next step is to do the actual hardware implementation of the simplified model. First the hardware implementation of the cochlea is given and its correct behavior illustrated, then the latest progress in implementing the simplified TC model are given.

5.3.2 Hardware implementation of the cochlea filter bank

To implement the cochlea on the FPGA, it is necessary to add additional modules for the streaming of the data in and out of the FPGA. Two versions were developed. The first one uses an Opal Kelly® (OK) board and a laptop running Matlab® to send data to the OK board and feed the model output back in to Matlab®. It will be later referred to as version 1 (OK alone). The second version uses a DACQ unit with the OK board plugged onto it. An acoustic sensor or a signal generator is connected to the DACQ. The data sampled by the acquisition unit is streamed to the OK board in real-time. Then a laptop running Matlab® reads out the output of the cochlea. It will be later referred to as version 2 (DACQ and OK).

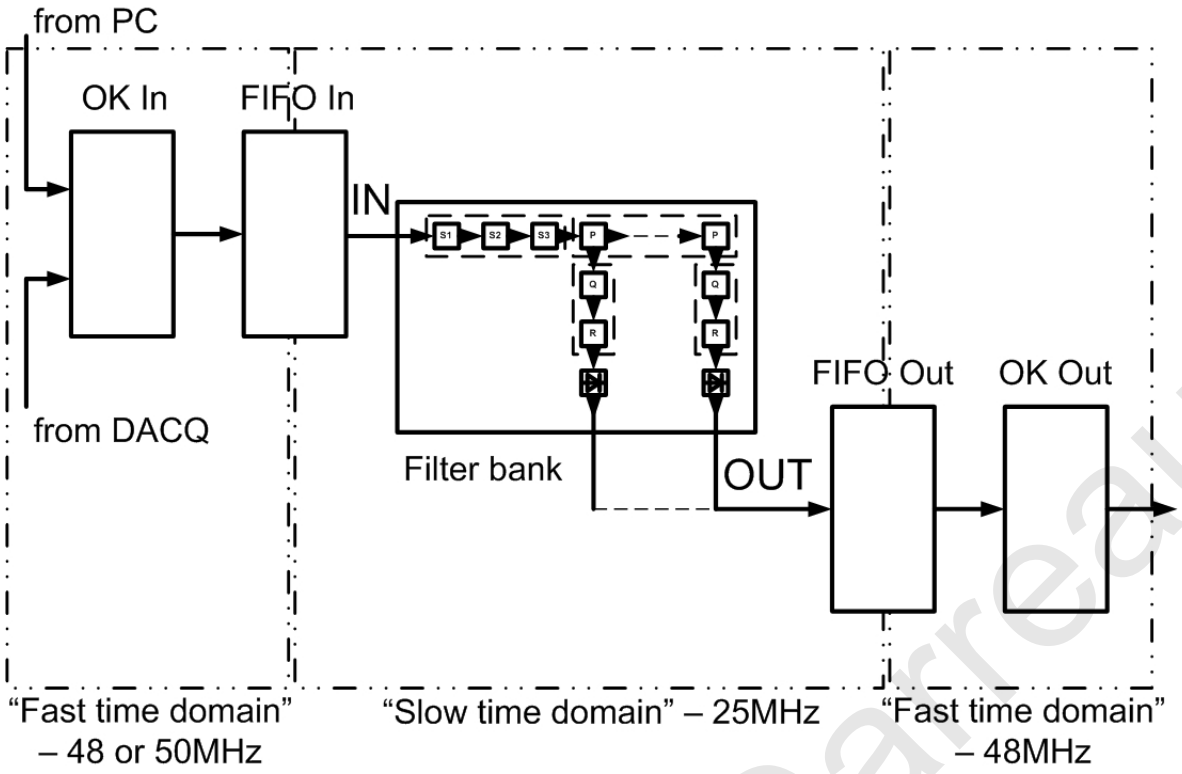


Figure 5.21: Diagram of the cochlea model.

The modules implemented on the Spartan3 of the OK board are: the input interface (USB to FPGA) that controls the data stream coming into the OK board, an input FIFO for slowing down the input data to the cochlea speed, the cochlea itself, an output FIFO to do the time domain conversion from the 'slow' cochlea to the fast output USB2.0 interface and finally the output interface (FPGA to USB) that controls the data stream out of the OK board (cochleagram). A block diagram of the different modules is shown in Fig. 5.21.

Table 5.1 gives the main parameters used to generate the filter bank mimicking the cochlea. Table 5.2 gives the filter parameter values for the three pre-emphasis first order IIR filters (S1 to S3 in Fig. 5.4). Table 5.3 gives the filters' parameter values for each one of the 32 channels comprising the cochlea. C is the filter coefficient corresponding to the first order IIR filter (P in Fig. 5.4). C_1 and C_2 are the coefficients for the identical cascaded second order IIR filters, noted Q and R in fig. 5.4.

The equations for the first order IIR filters (pre-emphasis and P) are:

$$tmp = dp_{cp}(0) \times dp_{zp}(t) + d_p \tag{5.12}$$

$$d_p = dp_{cp}(1) \times (dp_{zp}(t) + tmp) \tag{5.13}$$

where dp_{cp} are the filter coefficients and dp_{zp} are the values of the signal to be

Parameter	Description	Value
n_ch	Number of channels	32
d_t	Sampling interval	3.125×10^{-5}s
f_{base}	Resonator frequency at the base	16kHz
f_{apex}	Resonator frequency at the apex	500Hz
q_{base}	Low-pass filter and resonator tuning at the base	4
q_{apex}	Low-pass filter and resonator tuning at the apex	0.5

Table 5.1: Parameters used for the cochlea implementation.

Filter	Q	Center F (Hz)	C
1	4	17892.20	0.569
2	4	20008.96	0.509
3	4	22375.59	0.455

Table 5.2: Frequency and coefficient parameters of the pre-emphasis filters.

filtered at the current and previous time steps. d_p is the output of the previous channel. In the case of the first cochlea channel the previous stage is the last one of the three S filters. For S1, the input is the audio signal.

The equations for the 2nd order bandpass IIR filters (Q) are:

$$tmp = dp_{cq}(0) \times dp_{zq}(t) + dp_{cq}(1) \times dp_{zq}(t-1) + d_p \tag{5.14}$$

$$d_q = dp_{cq}(3) \times dp_{zq}(t-1) + dp_{cq}(2) \times tmp \tag{5.15}$$

where dp_{cq} are the filter coefficients and dp_{zq} are the values of the signal to be filtered at the current and previous time steps. d_p is the output of the previous stage. For the R filter, the same equations are used, replacing q indices by r and p indices by q.

Figure 5.22 gives the block diagram of the generic implementation of second order band-pass IIR filters.

Channel ID	Q	Center F (Hz)	C	C_1 ($\times 10^8$)	C_2 ($\times 10^9$)
1	4 .000	16000.00	0.637	16.085	15.811
2	3.887	14307.55	0.712	14.801	13.658
3	3.774	12794.15	0.796	13.632	11.921
4	3.661	11440.85	0.89	12.566	10.52
5	3.548	10230.64	0.996	11.594	9.388
6	3.435	9148.54	1.113	10.708	8.471
7	3.323	8180.88	1.245	9.901	7.728
8	3.210	7315.56	1.392	9.165	7.125
9	3.097	6541.75	1.557	8.495	6.635
10	2.984	5849.74	1.741	7.884	6.235
11	2.87	5232.38	1.947	7.327	5.909
12	2.758	4677.72	2.178	6.82	5.642
13	2.645	4182.91	2.435	6.359	5.423
14	2.532	3740.46	2.435	6.359	5.423
15	2.419	3344.80	3.045	5.559	5.094
16	2.307	2991.00	3.405	5.215	4.971
17	2.194	2674.60	3.808	4.903	4.869
18	2.081	2391.78	4.259	4.623	4.784
19	1.968	2138.72	4.763	4.371	4.714
20	1.855	1912.56	5.326	4.146	4.655
21	1.742	1710.28	5.956	3.948	4.606
22	1.629	1529.32	6.696	3.775	4.566
23	1.516	1367.62	7.448	3.627	4.533
24	1.403	1222.95	8.329	3.505	4.505
25	1.290	1093.55	9.314	3.408	4.484
26	1.177	977.85	10.416	3.34	4.468
27	1.065	874.40	11.648	3.303	4.457
28	0.9516	667.34	13.026	3.304	4.451
29	0.8387	699.17	14.567	3.353	4.451
30	0.7258	625.16	16.29	3.464	4.458
31	0.6129	559.11	18.217	3.669	4.475
32	0.500	500.00	20.372	4.021	4.508

Table 5.3: Frequency and coefficient parameters of the cochlea filter bank.

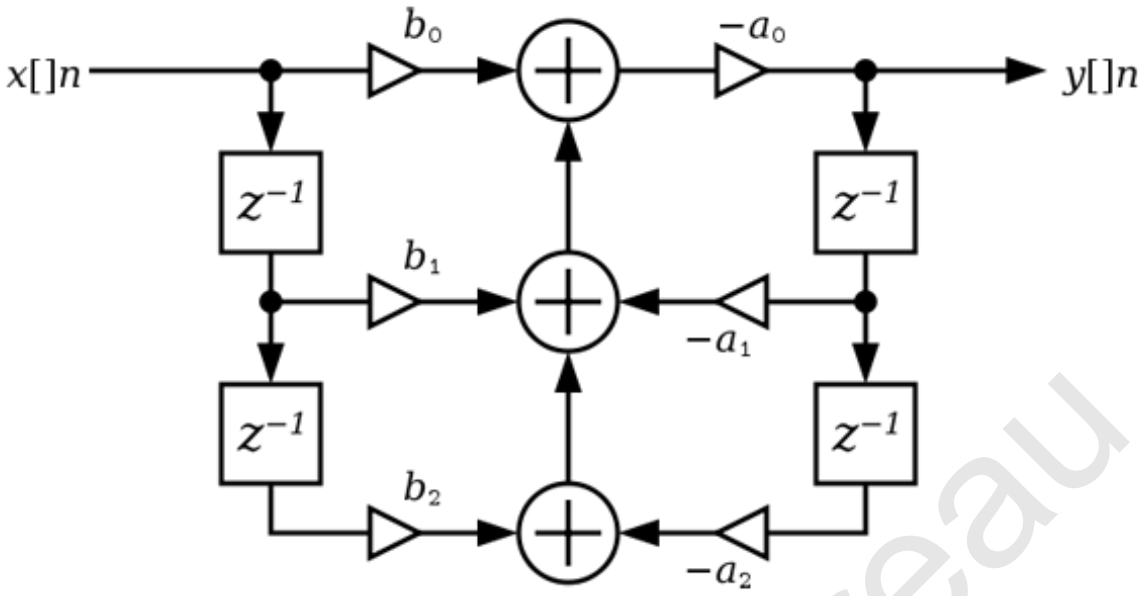


Figure 5.22: Diagram of a second order band-pass IIR filter.

5.3.3 Comparison of hardware and software implementation of the cochlea filter bank

The cochlea network model presented by Liu et al. in [156] was fully implemented in software (C language) and simulated in hardware using Handel-C. A VHDL version of the cochlea was written and implemented on the OK board XEM3010. The utilisation table of the FPGA by the cochlea is given in Table 5.4 for version 1 (OK alone) and in Table 5.5 for version 2 (DACQ and OK).

Figure 5.23 shows the output of the hardware implementation of the cochlea model on the OK board and the expected output as given by the software model, using the same input signal and parameters for the filter bank. The signal used is a 1s chirp, i.e. a sine wave sweeping from 512Hz to 15.5kHz. The two outputs are almost identical. The small differences are due to the available precision for the parameter values in the software and hardware implementations.

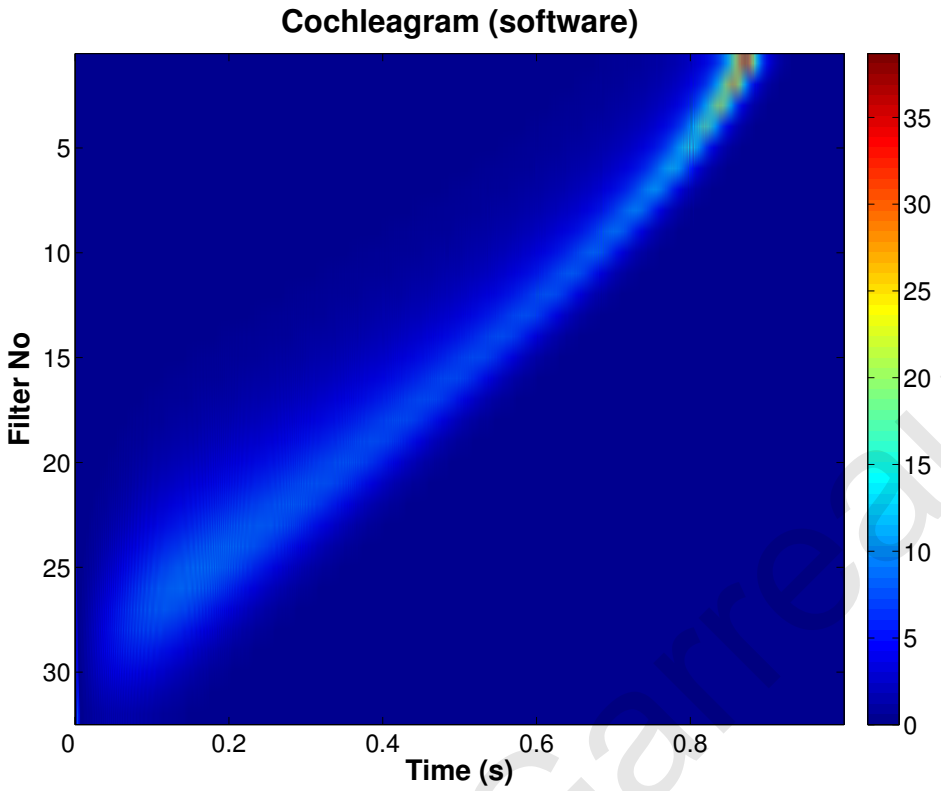
The previous plots show the test using version 1 of the model (OK alone). For version 2, the set-up is equivalent to the whole outer and inner ear, the DACQ unit is sampling a signal and sending it to the cochlea on the XEM3010 of Opal Kelly board connected to it. The cochleagram is then display on the laptop. For lack of anechoic chamber, it was decided to generate a signal equivalent to that output from a microphone. The signal generator is connected to the DACQ unit. A picture of the set-up is given in Fig. 5.25. However as audio data has been recorded with

Logic Utilization	Available	Utilization (%)
Number of occupied slices	13,312	8
Number of slice Flip Flops	26,624	2
Number of 4 input LUTs	26,624	6
LUTs as logic	-	84.7
LUTs as 32×1 RAMs	-	9.1
Number of bonded IOBs	221	17
Number of RAMB16s	32	6
Number of MULT18X18s	32	31
Number of BUFGMUXs	8	25
Number of DCMs	4	25

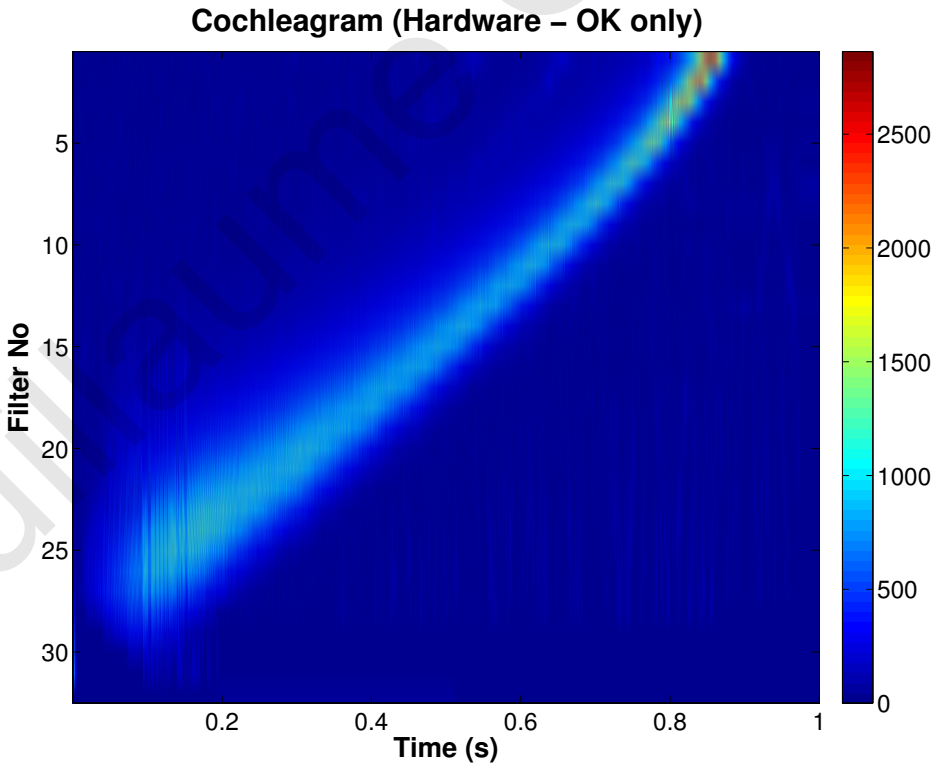
Table 5.4: Utilisation table of the cochlea implementation on the XEM3010, version 1 (OK alone).

Logic Utilization	Available	Utilization (%)
Number of occupied slices	13,312	8
Number of slice Flip Flops	26,624	2
Number of 4 input LUTs	26,624	6
LUTs as logic	-	84.4
LUTs as 32×1 RAMs	-	9.1
Number of bonded IOBs	221	44
Number of RAMB16s	32	6
Number of MULT18X18s	32	31
Number of BUFGMUXs	8	25
Number of DCMs	4	25

Table 5.5: Utilisation table of the cochlea implementation on the XEM3010, version 2 (DACQ and OK).



(a)



(b)

Figure 5.23: (a) Output of the software implementation of the cochlea model. (b) Output of the hardware implementation on the Opal Kelly board (version 1). The input signal is a 1s chirp of increasing frequency sine wave.

the DACQ unit, the 'real' scenario of a microphone recording an acoustic scene that is then streamed to the cochlea is possible in the current state of development of the system. Figure 5.24 shows the cochleagram obtained for a pure sine wave at 2.5kHz. The result confirms what we expected with a linear response over time around channels 17-18 that correspond to frequency of 2.67 to 2.39kHz.

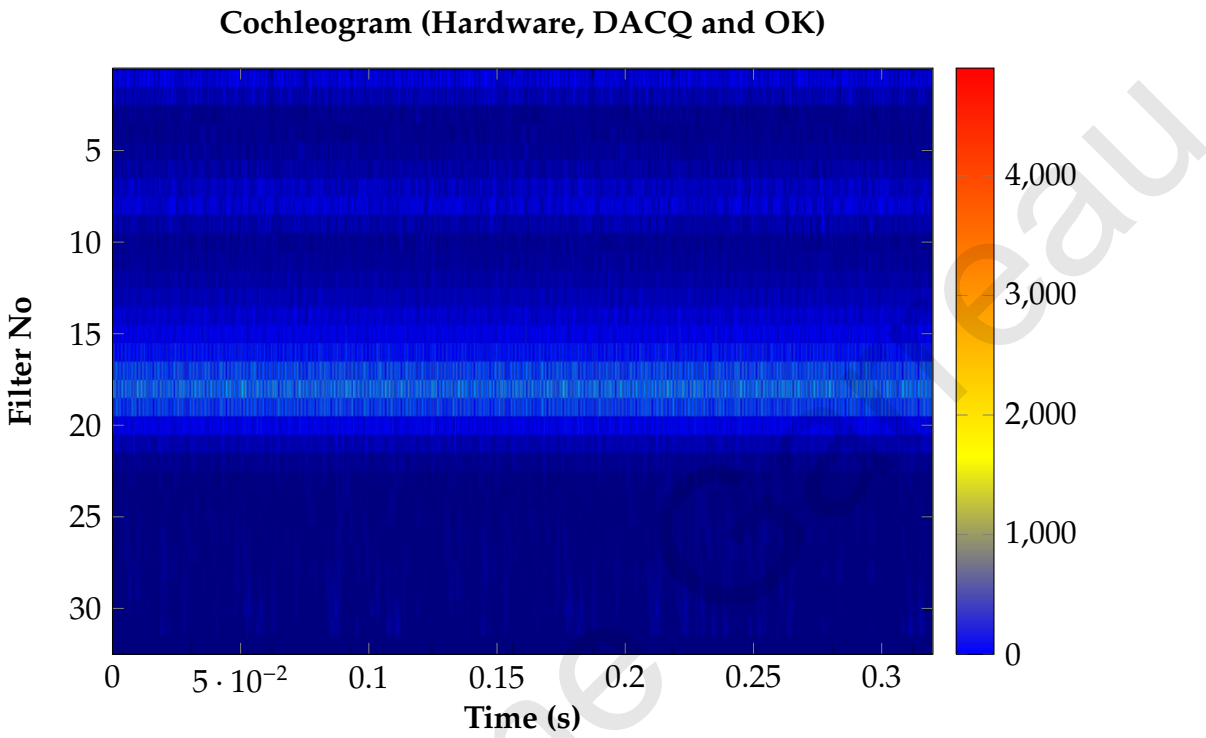


Figure 5.24: Output of the hardware implementation on the Opal Kelly board (version 2). The input signal is a pure sine wave.

5.3.4 Hardware implementation of the TC model

The Handel-C simulation demonstrated that the simplified auditory pathway (cochlea and simplified TC model) can be implemented on hardware. However the compilation of the VHDL script generated by the Handel-C software did not fit on the targeted FPGA (a XC3S1500), thus it was decided to optimized the code by doing an implementation step by step of the different components of the model. For now, the full implementation of the simplified model is not done yet. The time delay between B_1 and B_2 and the STDP synapses responsible for the learning and memory capability (Fig. 5.7), are not yet implemented. The utilisation table of the FPGA by the cochlea is given in Table 5.6. Here, the latest progress towards the full implementation are presented.

(a)

(b)

Figure 5.25: (a) Picture of the DACQ board connected to a signal generator, the data sampled is streamed to the Opal Kelly board, processed by the cochlea and then, the output cochleagram is sent to a computer. (b) Zoom on the DACQ unit with the OK board on top of it.

In addition to the modules used for the cochlea implementation, several more modules are implemented on the Spartan3 of the OK board XEM3010. These are: an intermediate FIFO between the cochlea and the simplified TC model input (neurons A) and the TC model itself with the different neurons and the connections between them. The output communication is changed from a FIFO to a bRAM. A block diagram of the different modules is given in Fig. 5.26.

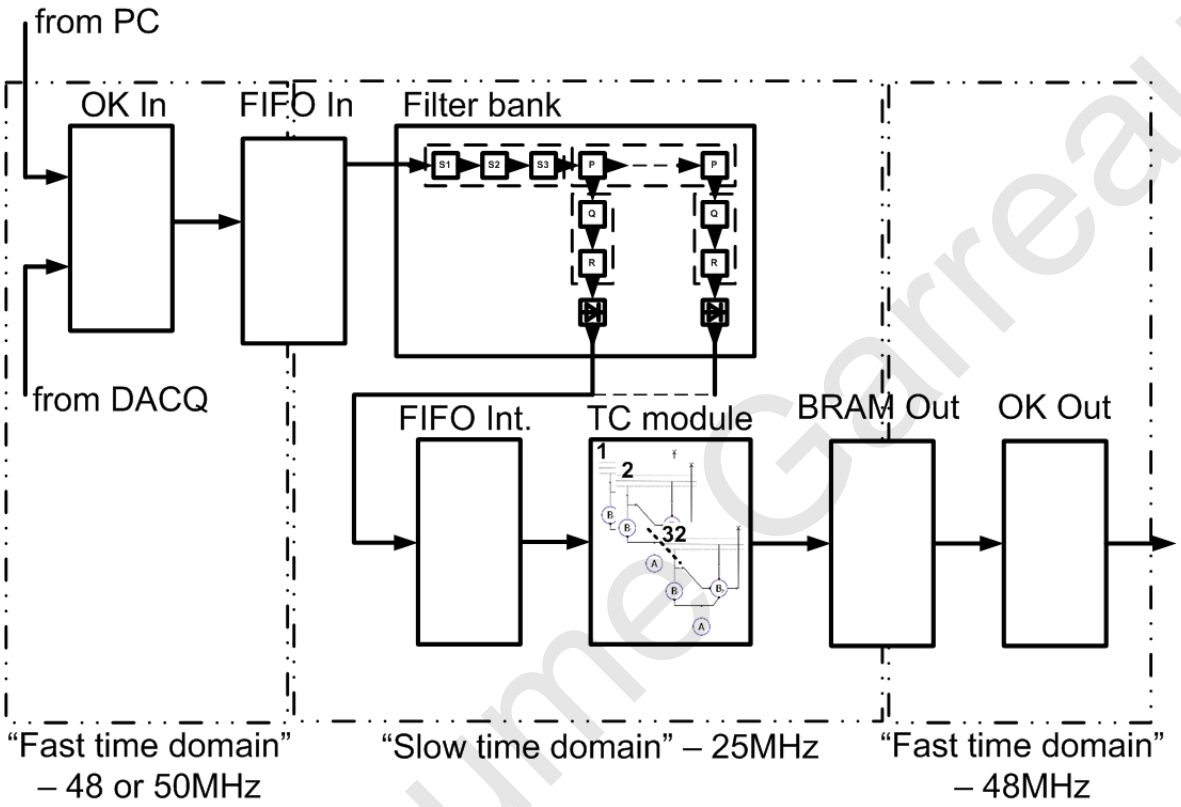


Figure 5.26: Diagram of the model of the thalamocordical model.

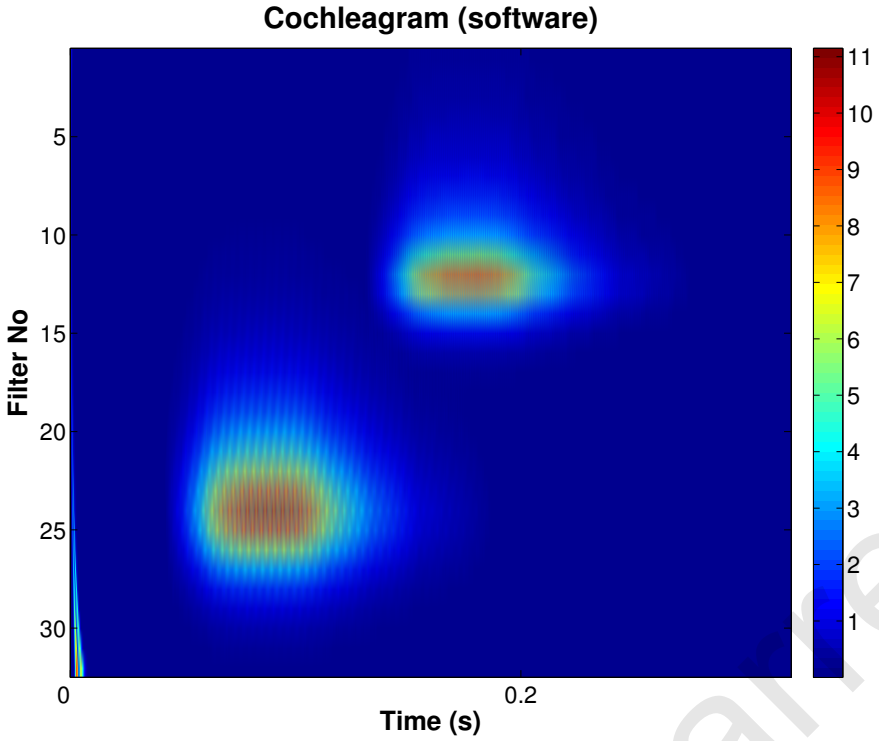
The following cochleagrams and plots illustrate the response of a FPGA based implementation of various auditory pathway components. The cochlea is stimulated with two different mixtures of frequencies, which are non-overlapping in time. It is the same stimulus than the one used in the hardware simulation with Handel-C. The comparison between hardware simulation and actual hardware implementation is thus straightforward. The resulting cochleagram is shown in Fig. 5.27. This is fed into layer A, whose membrane potential is modified as shown in Fig. 5.28a. This layer contains I&F neurons which are driven by the acoustic power in each frequency band from the cochlea and spike depending on their input to produce a raster plot as shown in Fig. 5.28b. The spiking activity results in the synaptic conductance being modified for synapse A-B_1 and synapse A-B_2 as shown in Fig. 5.29. The activation

Logic Utilization	Available	Utilization (%)
Number of occupied slices	13,312	66
Number of slice Flip Flops	26,624	36
Number of 4 input LUTs	26,624	32
LUTs as logic	-	96.5
LUTs as 32×1 RAMs	-	1.8
Number of bonded IOBs	221	17
Number of RAMB16s	32	9
Number of MULT18X18s	32	31
Number of BUFGMUXs	8	25
Number of DCMs	4	25

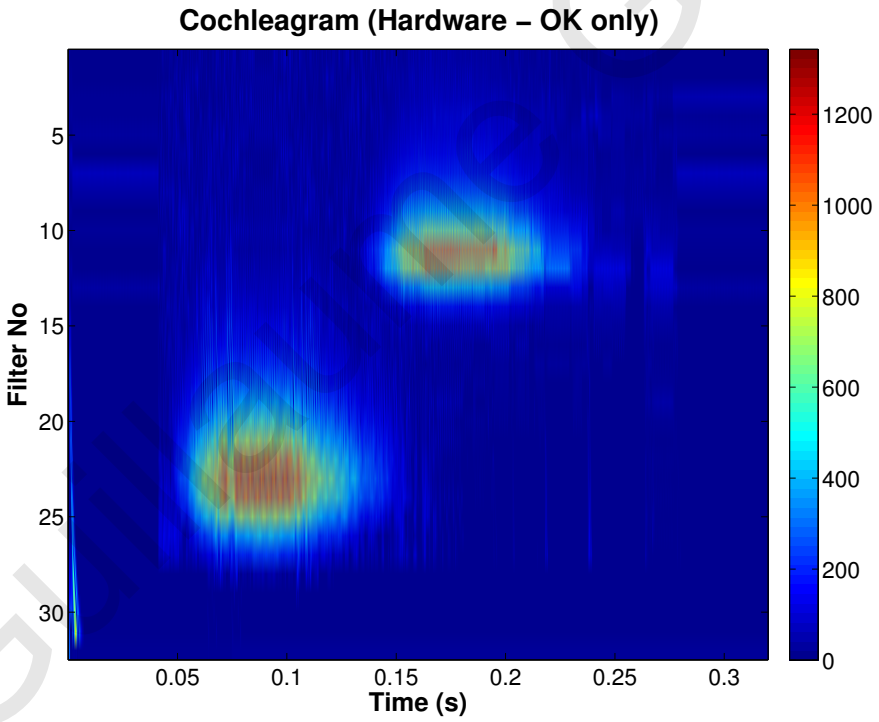
Table 5.6: Utilisation table of the TC network implementation on the XEM3010.

of the synapses A-B_1 leads to the change in the B_1 neuron membrane potential Fig. 5.30a. The incoming inputs from A are integrated by the I&F neuron equation, which causes the B_1 neuron to spike as shown on the raster plot of Fig. 5.30b. The spikes are projected through inhibitory synapses to the post-synaptic layer B_2. The spikes are also projected to B_2 in other locations through plastic synapses with transmission delays. This leads to the inhibitory synapse between B_1 and B_2 to be active (Fig. 5.31) thus providing an inhibitory input to the B_2 neuron within the same column, but also excitatory inputs to other columns. Columns that are close by have their membrane potential increased since the combination of the input from synapse A-B_2 and the excitatory input from B_1-B_2 makes the B_2 neuron membrane potential increase (Fig. 5.32a) and spike (Fig. 5.32b).

Note that the plots obtained with the hardware implementation and the ones obtained with the hardware simulation (Handel-C) are almost identical for the layer A and B_1 (membrane potential, spike activity and synapses) but different for layer B_2 due to the missing STDP synapses and delays. This confirms that a full implementation is possible and that it should give the expected output.

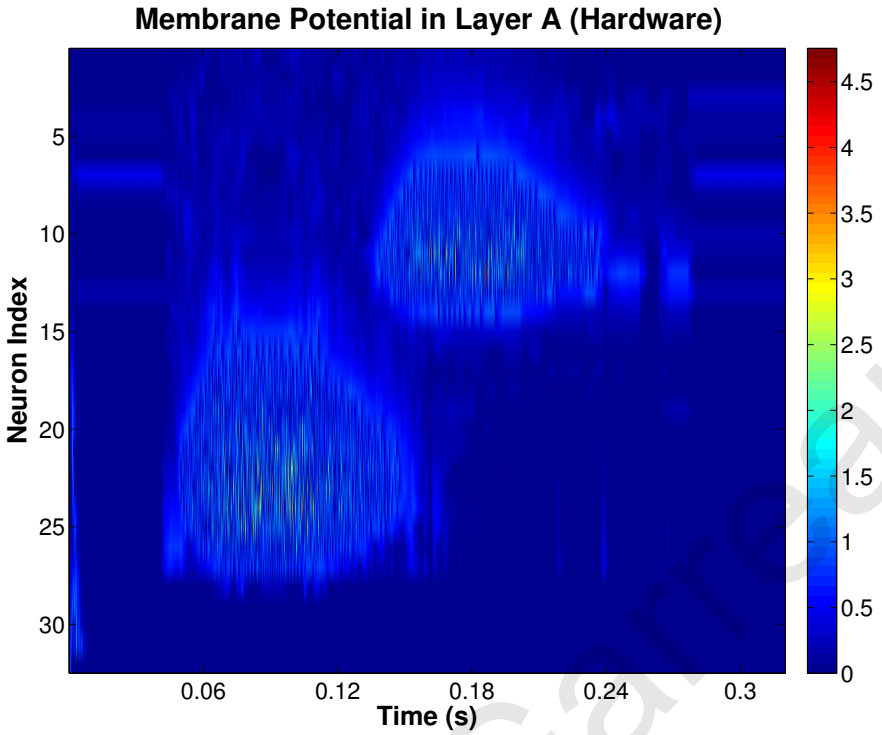


(a)

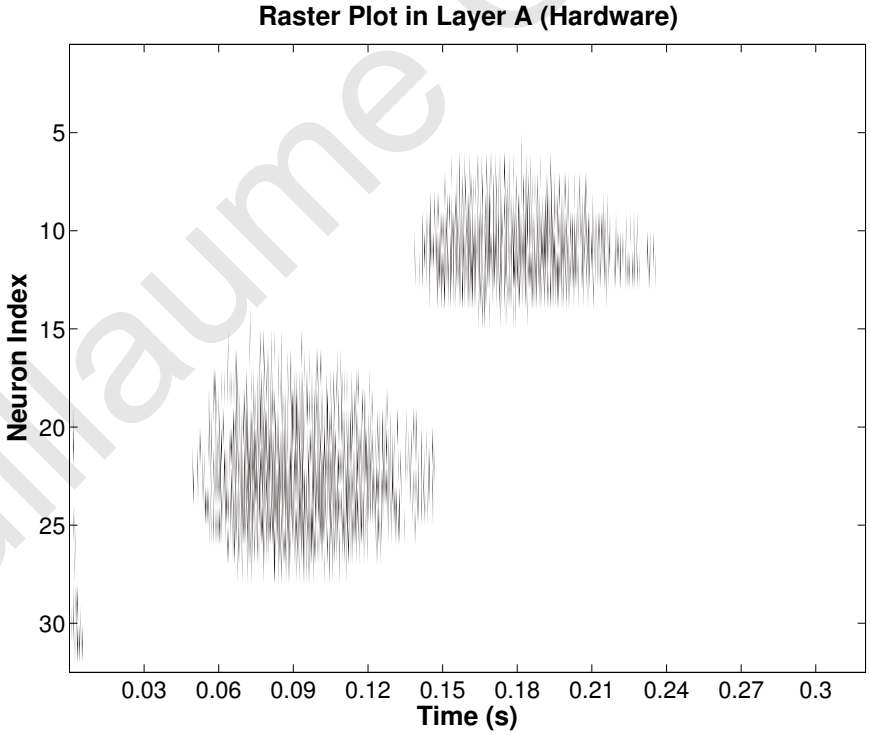


(b)

Figure 5.27: This figure shows the power output of a 32 channel cochlear model [156], which is used as a stimulus for the 32 columns of the simplified thalamocortical model. (a) Output of the software implementation of the cochlea model. (b) Output of the hardware implementation on the Opal Kelly board (version 1). The input signal is two different mixtures of frequencies, which are non-overlapping in time.



(a)



(b)

Figure 5.28: This figure shows the response (represented as spikes) of neurons in layer A, which are propagated through excitatory synapses to layers B_1 and B_2. (a) Membrane potential in layer A. (b) Spiking activity in layer A.

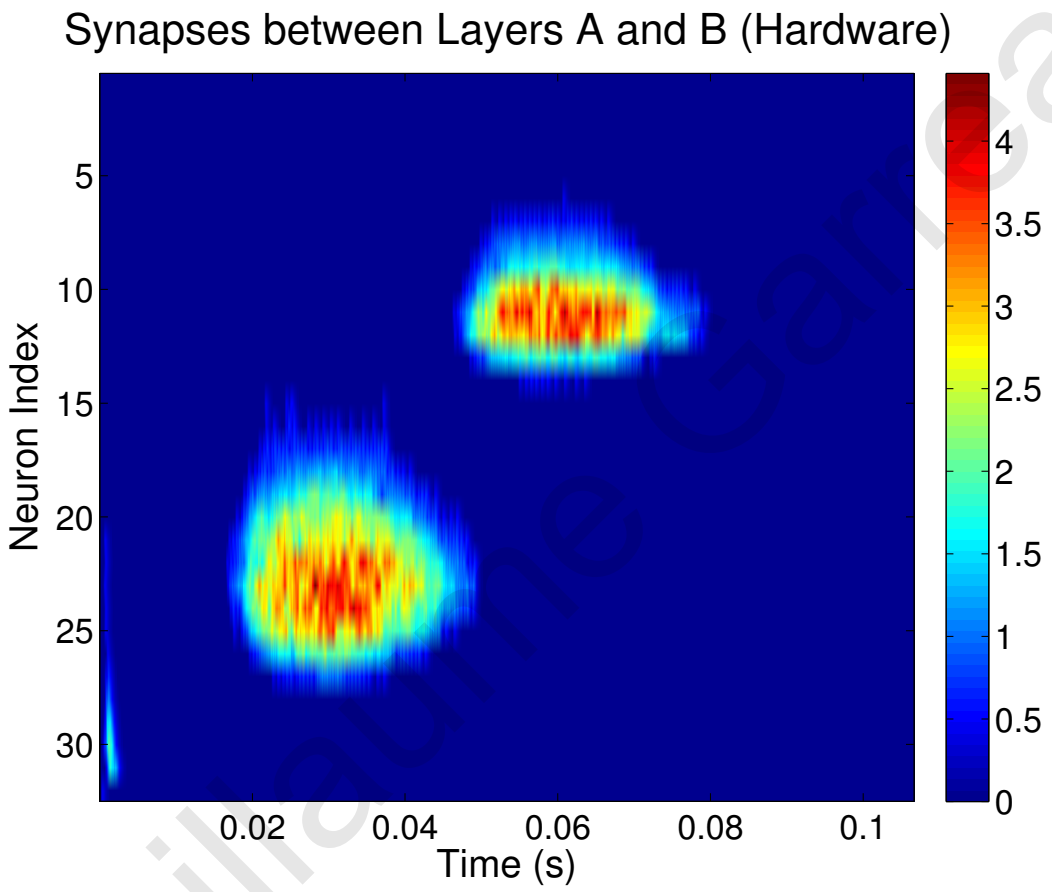
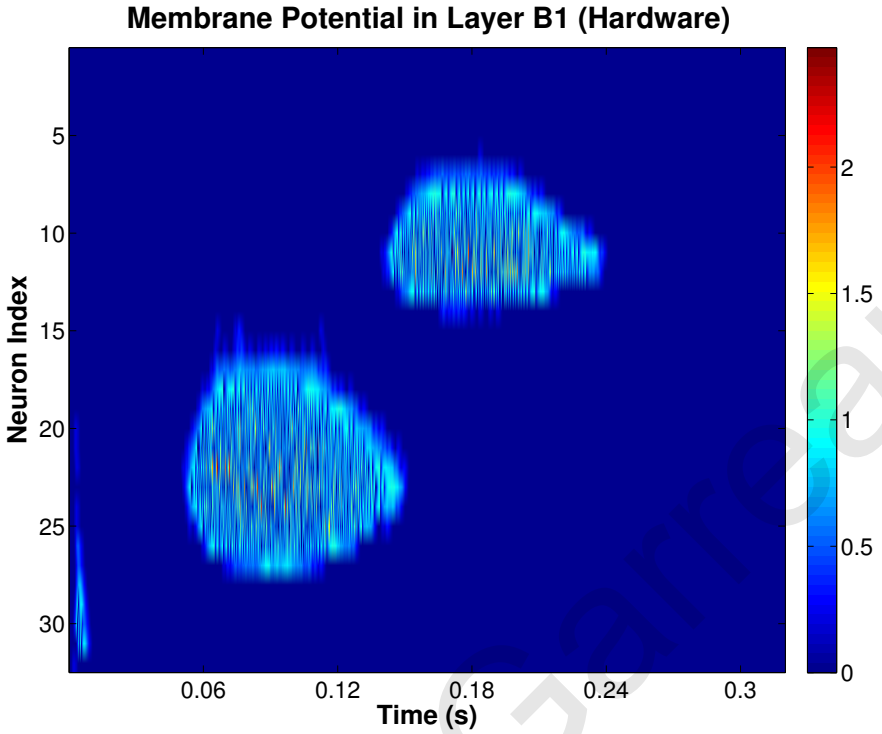
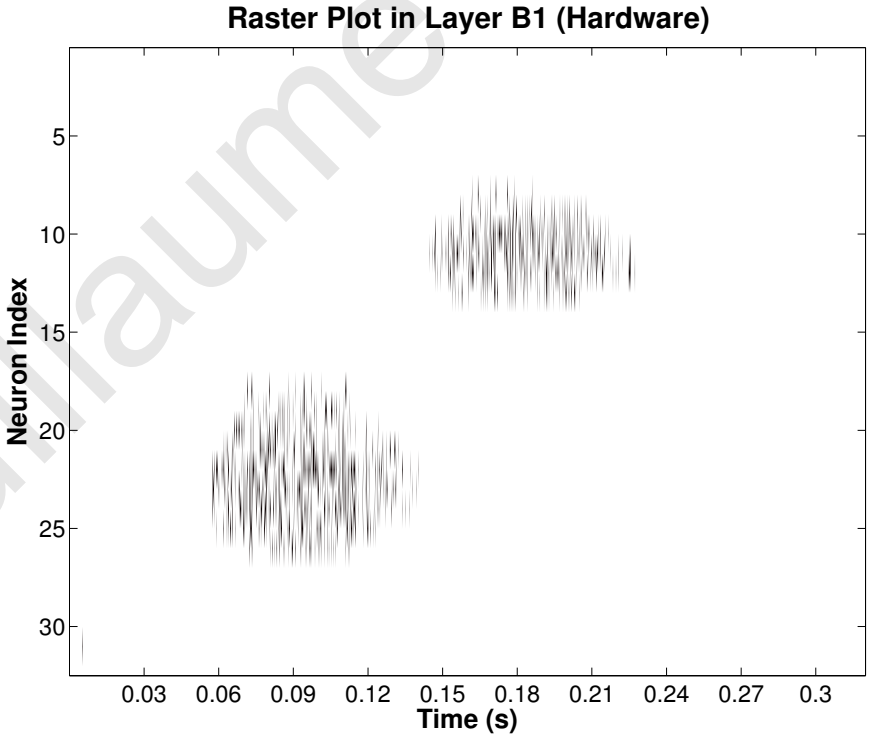


Figure 5.29: Synaptic response in A-B_1 and A-B_2. Here, the synapses are modelled by an alpha profile.



(a)



(b)

Figure 5.30: (a) Membrane potential in B_1. (b) Spiking response in B_1.

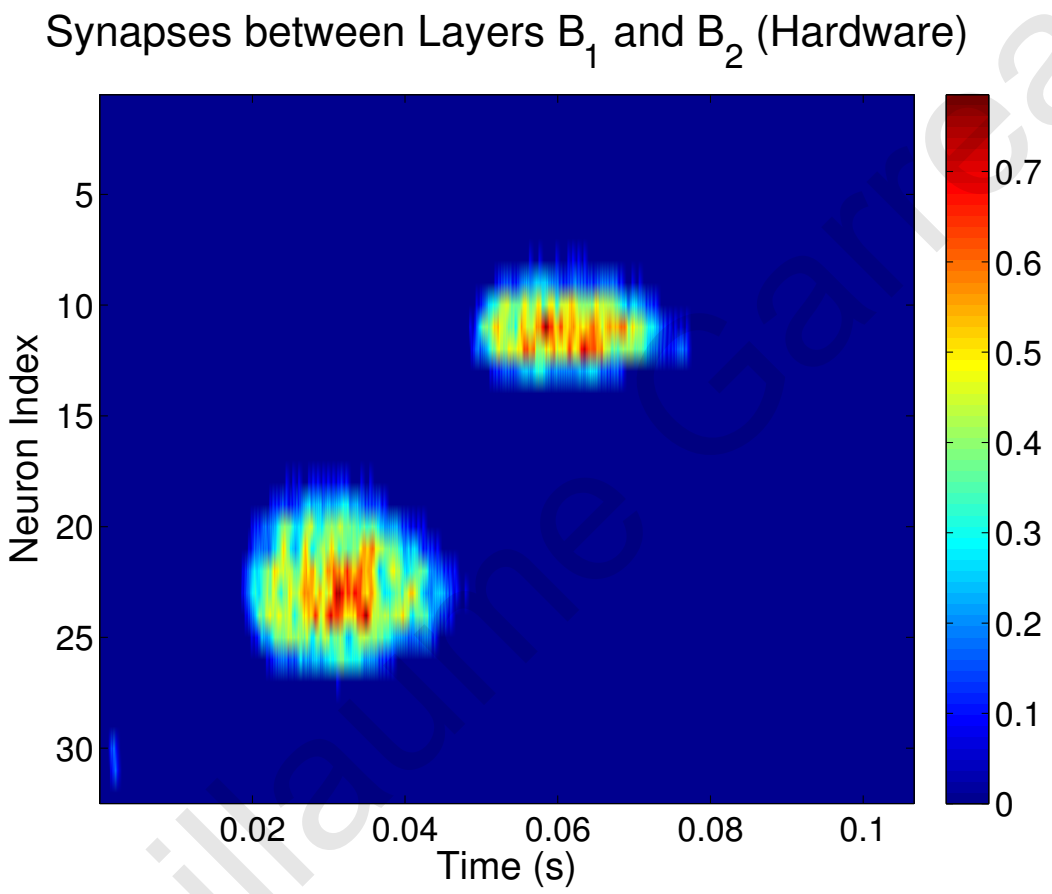
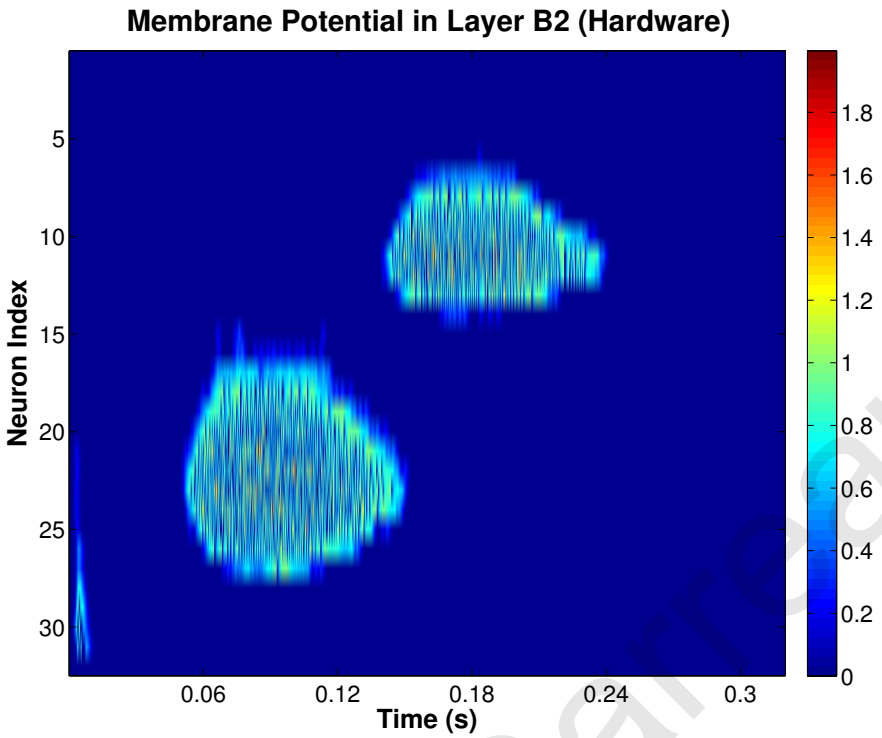
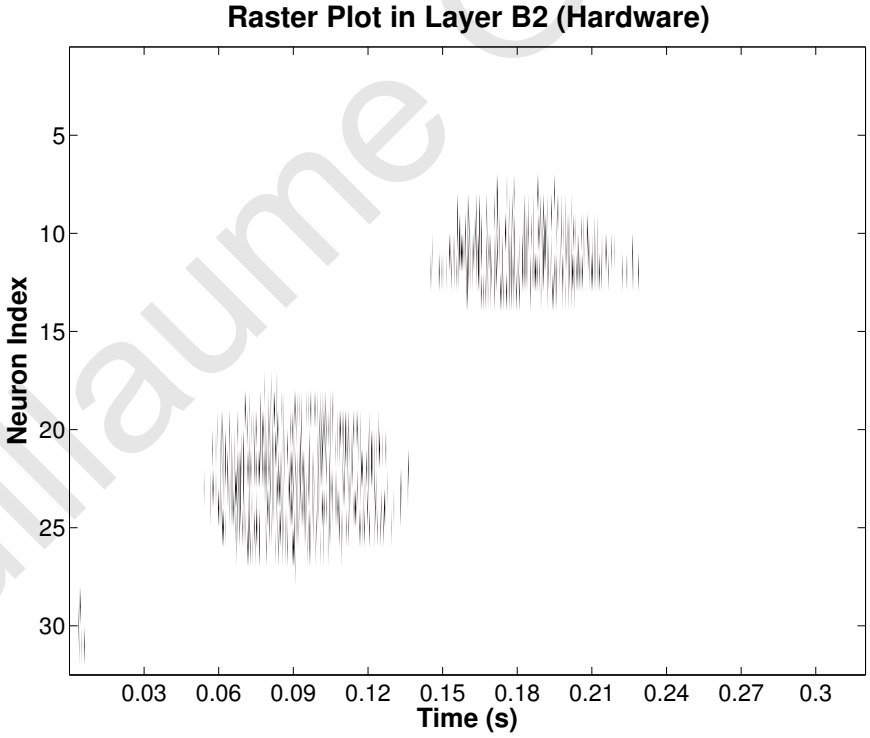


Figure 5.31: Synaptic response in B_1-B_2, which shows the response of the synapses with an alpha profile, which inhibits neurons in layer B_2.



(a)



(b)

Figure 5.32: This figure shows the resulting dynamics of the 32 neurons in the thalamocortical network. (a) Membrane potential in B_2, which shows the response of neurons in layer B_2 from the integration of incoming inputs from A, B_1 and recurrent connections. (b) Spiking response in B_2.

	XEM3010 (partial TC)		XEM3050 (full TC)		XEM6010 (full TC)	
	Avail.	Used (%,meas.)	Avail.	Used (%,est.)	Avail.	Used (%,est.)
Flip Flops	26,624	36	55296	80	184304	36
LUTs	26,624	32	55296	41	92152	55

Table 5.7: Estimation of the FPGA utilization ratio. The XEM3050 has a full TC model using the simplest STDP implementation. The XEM6010 has a full TC model using the most complex STDP implementation. These estimations are based on the partial implementation reported here and the work reported in [41].

The utilisation ratio of the FPGA for the partial implementation of the model is quite low (Tab. 5.6) and confirm that a compact implementation of the full model on a single FPGA is possible. More particularly only 1/3 of the Flip Flops and of the 4 input Look-Up Tables (LUTs) are used. Cassidy *et al.* demonstrated very efficient implementations of various STDP encoding targeting a XC3S1500 [41]. According to their results, the most efficient implementation for an STDP synapse is 35 slice Flip Flops and 14 4 input LUTs. Implementing 31 STDP synapses $\times$ 32 neurons (each B_1 neurons projecting on every B_2 neurons not in the same column) would require 34720 Flip Flops and 13888 4 input LUTs. Our current FPGA would have enough LUTs but would lack of Flip Flops. By upgrading our Opal Kelly board from the XEM3010 (with XC3S1500 FPGA) to a XEM3050 (with XC3S4000), the full implementation of the TC model using the simplest STDP implementation is possible and upgrading to the more powerful XEM6010 would allow implementing the full TC model with the more complex implementation of STDP. Tab. 5.7 gives the utilization estimation of the partial and full TC model implementation with more or less complex STDP implementations. Note that the utilization estimation of the most powerful FPGA encourages the use of FPGA to implement even more complex models with more columns, more neurons per columns and more connection between neurons.

5.3.5 Comparison of FPGA-based and hybrid analog/digital VLSI-based hardware implementation of the TC model

During the SCANDLE project, partners at the University of Zurich implemented the simplified version of the model by Coath *et al.* [53] using hybrid analog/digital VLSI chips. The implementation is shown in Fig. 5.33. The set-up comprises 3

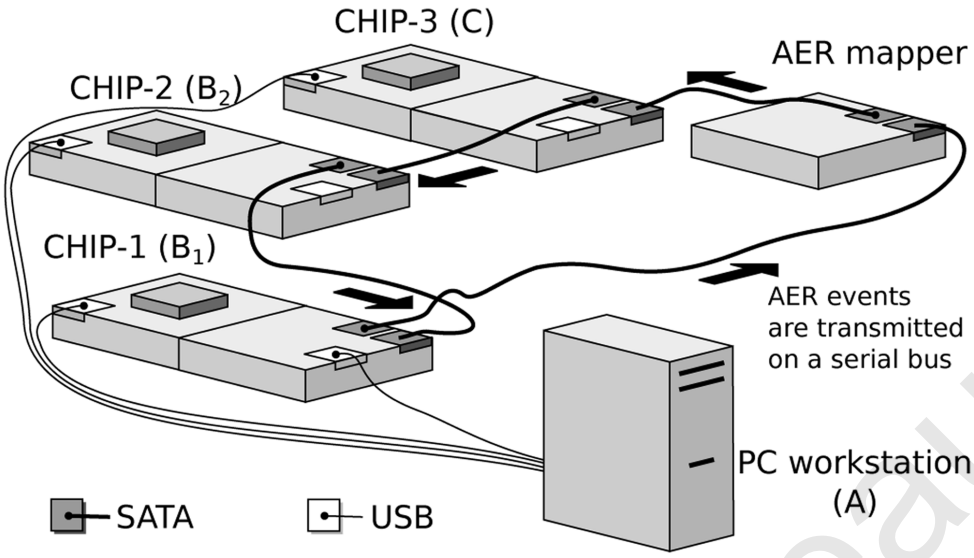


Figure 5.33: The chips are mounted on custom PCBs (AMDA) which supply bias voltages to the chips. These biases can be configured via a USB interface connected to the PC workstation. The AER events are handled by dedicated PCBs equipped with FPGAs (AEX, [86]). These events are transmitted from one board to the other over SATA cables in a serial loop. Events can also be sent and monitored from a PC workstation via a USB interface (from [207]).

hybrid analog/digital VLSI chips (each has a XC3E500 FPGA [86]), 1 AER mapper (1 XC3S1500 FPGA, a dedicated Serializer-Deserializer chip and a PC motherboard [85]) and a computer [207].

The cochlea and the neuron layer A are simulated on software on the computer. Then the A spikes are sent through USB to chips 1 and 2 (implementing neuron layers B_1 and B_2). As the current hardware does not directly support propagation delays between neurons. Every projection in the model that requires a delay is passed through an additional neuron, referred to as a delay neuron (Fig. 5.34). The third chip implements the delay neuron layer, labeled C. The AER mapper and the chips are connected to each other with SATA connections. Chips 1 and 2 are identical and comprise of an array of 128 linear I&F neurons and 128×32 synaptic circuits. Each neuron in the chip is connected to 2 excitatory, 2 inhibitory, and 28 excitatory plastic synapse circuits. The third multi-neuron chip comprises of a two-dimensional array of 32-by-64 neurons. Each neuron in the chip is connected to 3 synaptic circuits (2 excitatory, 1 inhibitory). The results presented demonstrate, in hardware, how an implementation of a recurrently connected spiking network is able to learn and

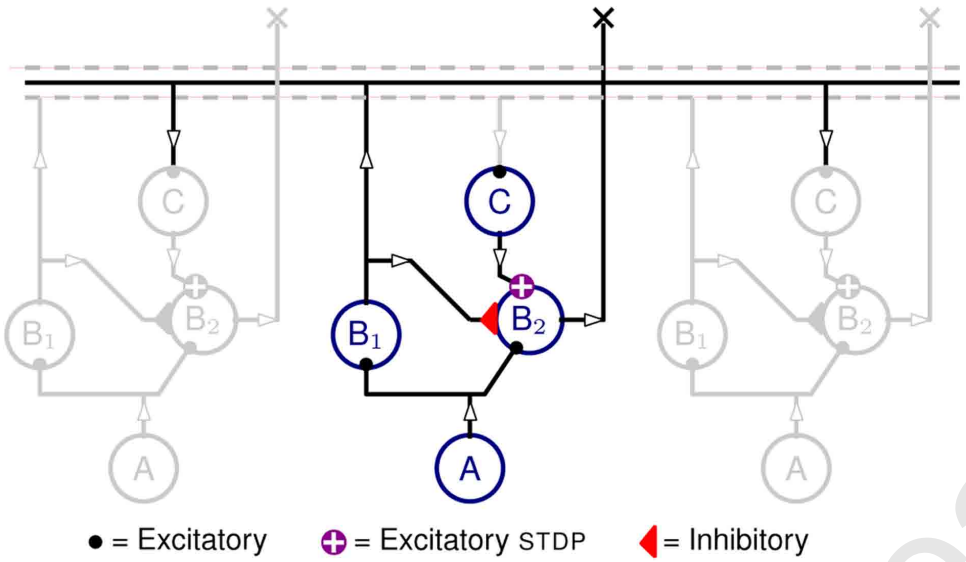


Figure 5.34: Neural network diagram, with one column highlighted and two neighboring ones in gray, included to indicate lateral connections. Circles represent neurons from neuronal populations A, B_1, B_2, and C. The output of the network is represented by the activity of population B_2 (from [207]).

selectively respond to the dynamic spectro-temporal features of stimuli (Fig. 5.35). The model relies on delays, which might arise from a number of processes including axonal propagation and spike interaction via intermediate neurons. These results were published in [54, 207]. It is the first report of hardware implementation of a model of auditory pathway comprising a cochlea, processing and learning neural networks.

As a comparison the implementation presented here has, on a single XC3S1500 FPGA, cochlea and neurons layers A, B_1 and B_2 but lacks of the STDP synapses and the delays. It is much smaller in size, consumes less power and is more versatile than the analog solution. Using the DACQ unit to provide data to the input of the cochlea adds another small XC3S50AN FPGA and mimic the full auditory pathway starting from the outer ear. Table 5.8 summarize the different characteristics of each implementation. Note that the hybrid analog/digital hardware solution of Sheik *et al.* was implemented with pre-existing hardware and was not a full custom analog ASIC targeting this TC model implementation. An interesting comparison in the future could entail the benchmarking between a custom analog ASIC and a custom digital ASIC of the TC model.

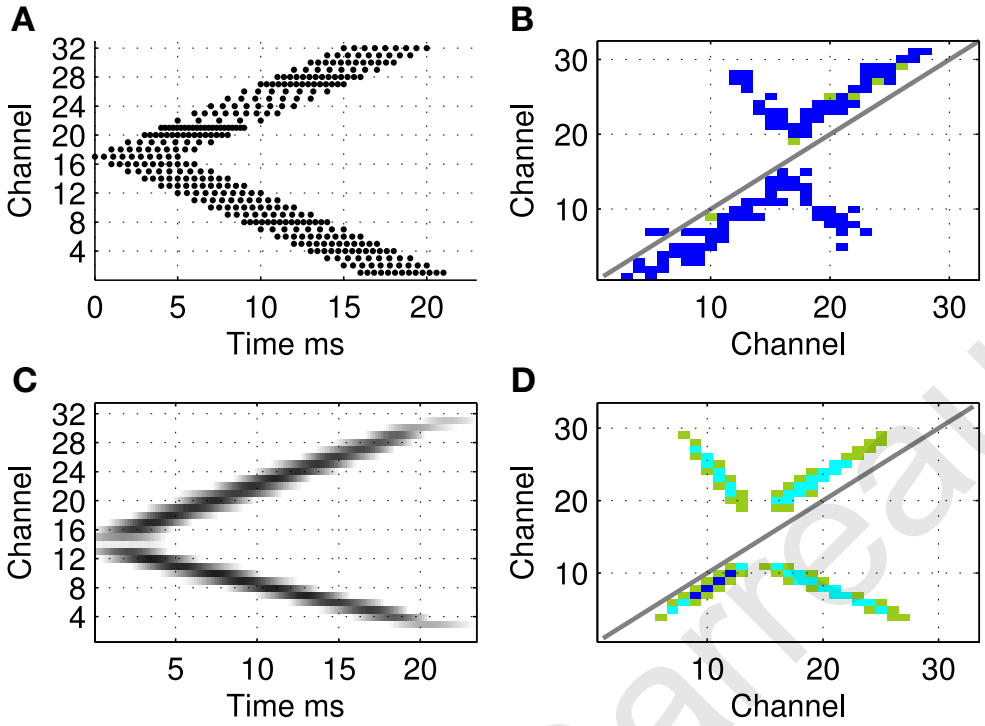


Figure 5.35: Comparison between hardware result using a synthetic stimulus pattern (A,B) and learning prediction using a real sound file (C,D). Top row shows raster of synthetic exposure stimulus (A) and resulting network connectivity after exposure (B) for hardware network, these figures are taken from [207]. Bottom row shows spectrogram of comparable sound file (C) and the analytically predicted pattern of connectivity (D) based on correlations in the stimulus representation (from [54]).

Parameters	Sheik *et al.*, 2012 [207]	This work
Technology	Hybrid analog/digital VLSI	FPGA
Number of FPGAs	4	1
Number of neurons per layers	128	32
Layers implemented in hardware	B_1, B_2 and C	cochlea, A, B_1 and B_2
Missing on hardware	cochlea and A	STDP synapses & delays
Development time	long	short
Power consumption	high	low
Scaling up	hard	easy

Table 5.8: Main characteristics of the 2 hardware implementations of the thalamocortical model by Coath et al. [53].

5.4 Conclusion

In this chapter, a very compact partial hardware implementation of the auditory pathway is presented, it is inspired from the cochlea of Liu *et al.* [156] and the model of Coath *et al.* [53]. The hardware implementation was also tested on data recorded with the DACQ unit and streamed in real-time to the Opal Kelly® board. It was compared with a hardware implementation on hybrid analog/digital VLSI chips, reported by Sheik *et al.* [207] and Coath *et al.* [54]. The results of the hardware simulation in Handel-C reported here, added to the work reported by Cassidy *et al.* in [38, 41] on optimized STDP FPGA implementations and a 1 million neurons array on a single FPGA, are a promising way to the compact and full implementation of an auditory pathway model. The FPGA thalamocortical network has a number of applications, including the development of more refined auditory pathway models. An FPGA cochlea is particularly suited as a testbed for algorithms that involve concurrent processing across multiple neuron processing channels. In addition, our DACQ unit coupled with an additional FPGA platform (such as the Opal Kelly® board) provides an avenue for developing, simulating, and studying auditory processing in more complicated, but realistic acoustic environments, that involve multiple sound sources. The signal processing required to simulate such realistic environments is computationally intensive and some of this preprocessing can be incorporated into the FPGA platform enabling real-time studies of auditory processing under realistic acoustic conditions. In future work, the aim is to create a sound surveillance unit that can be used for surveying an emergency site remotely, e.g. after an earthquake, so that the emergency response team can localise sources of sound and optimally utilize available resources or for supplying information to assist medical triage. Another application is the localization of a speaker during a conference or video conference.

Chapter 6

Multimodal Sensory Fusion

6.1 Introduction

The three previous chapters, describe and demonstrate the opportunities and/or potential that the three ranges of frequencies (LF, HF and audio) offer when it comes to localization of individuals, identification of acoustic object or individuals, gender classification, actions/behavior recognitions and modes of transport classification. These applications were demonstrated in simple scenarios, with one 'sense' used, but it may happen that in a real-life environment one technique could be more appropriate for a specific situation, and vice versa. Let us say that a building requires a security system to keep the entrance safe. By daylight, a video camera and a security guard may be the best solution; however, in a foggy day the camera becomes 'blind', and an ultrasonic system or seismic sensors are more appropriate to do the task. Another example is action recognition, where some actions could be easily recognized using sound recordings and other motion detectors. That is why a system that uses multimodal fusion of the data collected by one or many sensors of different type and frequency range is necessary in order to obtain the optimum solution.

Multimodal fusion is not something new for animals and humans, as one can easily realise how we use several of our senses (vision, hearing, smell, taste, and touch) to accomplish tasks such as identification of food, communication, memory or moving in our environment (Pegna *et al.* [188], Gelder [61]). It is also quite interesting to see the interaction between those senses in a way that may seem strange. For example if you ask someone to identify the perfume or taste of a liquid in a glass in which a colorant was previously added, the color tricks the

mind into thinking it is tasting a flavour that corresponds to the colour and not the actual flavour of the liquid (Hoegg and Alba [122]). Another example is the famous McGurk effect [169], where repeated utterances of the syllable [ba] had been dubbed onto lip movements for [ga], where normal adults reported hearing [da] instead of [ba]. This indicates an influence of vision upon speech perception. Furthermore, in case of blindness, it is amazing to observe what the brain is capable of. It has already been mentioned in Chapter 4 that blind people can use echolocation [67,224] but blind-sight has also been reported for navigation without echolocation (Gelder *et al.* [62]) and emotions recognition (Pegna *et al.* [189]). The brain imaging studies showed that the amygdala, the emotional center of the brain, is strongly involved.

Multimodal fusion is a very timely topic, that is of great interest to the military for the detection, location and identification of enemy vessels in water, underwater, in the air or on ground (Hall and Llinas [113]). For example, Yang *et al.* presented a system for target recognition and tracking based on radar and infrared image sensors [249]. Another example is a neural model of a Barn Owl created for bimodal (microphones and camera) orientation of a robotic system (Rucci *et al.* [197, 198]). More recently some work has been done for emotion recognition using a bimodal (faces and speech analysis, Datcu and Rothkrantz [59]) or trimodal (facial, vocal, and gestural modalities, Wagner Datcu, Dragoş and Rothkrantz [235]) system, as well as more advanced biometric-based identification process (Manchula and Arumugam [164]). Finally the work by Hong *et al.* has similar goals to the work proposed here [123], where evidential contextual information is represented, analysed and merged to achieve a consensus in automatically inferring activities of daily living for inhabitants in Smart Homes. In addition, Ekimov and Sabatier presented a system where a bimodal system using a sonar and microphones to record footsteps and show the time synchronization between events recorded by the different acoustic sensors [74]. Another related field is the one of sensory substitution devices (SSD), where one sense is used to compensate the loss of another one and a lot of work is done with visual-to-auditory SSD (e.g. see Haigh *et al.* [112]).

In this chapter, a new real-time acoustic system for human action and behavior recognition that integrates passive audio and active micro-Doppler sonar signatures over multiple time scales is reported. The system uses the hardware presented in Chapter 2 and its architecture is based on a six-layer convolutional neural network, trained and evaluated using a dataset of 10 subjects performing seven different

behaviors. Probabilistic combination of system output through time for each modality separately yields 94% (passive audio) and 91% (micro-Doppler sonar) correct behavior classification; probabilistic multimodal integration increases classification performance to 98%. This study supports the efficacy of micro-Doppler sonar systems in characterizing human actions, which can then be efficiently classified using Convolutional Neural Networks (ConvNets, LeCun *et al.* [146]). It also demonstrates that the integration of multiple sources of acoustic information can significantly improve system performance. This work has been published in [69].[1]

6.2 Bioinspired model literature

The system presented is capable of forming composite representations of behavior exclusively through the use of information derived from sounds. Firstly, it is able to identify and classify moving individuals in the environment by active sound emission and the detection of micro-Doppler frequency shifts in sonar returns (as shown in Chapter 4). Secondly, it can identify the behavioral consequences of actions by passive detection and classification of sounds emitted by the individuals themselves as they interact with their environment (e.g. foot-steps, hand-claps).

6.2.1 ConvNets: overview

The system architecture is based on a six-layer convolutional neural network. Convolutional Neural Networks or ConvNets [146] are based on biophysiological principles discovered from the study of primary visual cortex and reported by Hubel and Wiesel [125]. Additional support for a computational model of auditory object processing based on spectro-temporal features is found in a study that employed Dynamic Causal Modeling (DCM) to evaluate the functional connectivity between three different regions of the auditory system (Kumar *et al.* [143]). Regarding micro-Doppler sonar processing, evidence suggests that the auditory system in bats (Suga [220]), and dolphins (Cauwenberghs *et al.* [46]) is organized hierarchically. The ConvNets architecture is flexible and trainable, and the network is able to learn selective and invariant features for classification. ConvNets have been

[1]The data collection was done at UCY in collaboration with Salvador Dura-Bernal. The data processing was conducted in collaboration with Salvador Dura-Bernal.

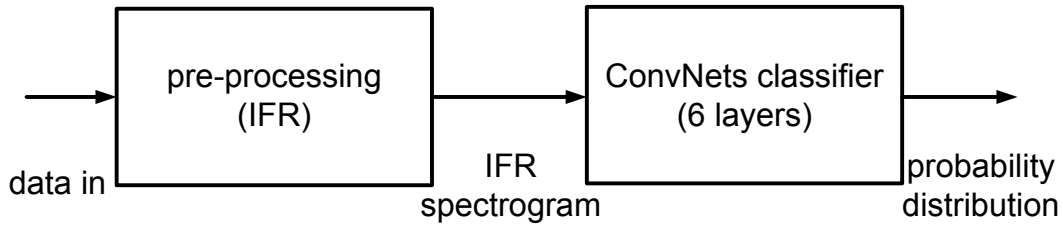


Figure 6.1: The ConvNets architecture is composed of 2 main stages: pre-processing and classifier.

known for many years and have been reported working robustly for different applications, such as character recognition, time-series prediction and speech (LeCun and Bengio [145]). This suggests that ConvNets may capture some of the principles of auditory processing in cortex, such as the use of spectrotemporal features and a hierarchical organization. Other similar bioinspired architectures have been used for object (Barshan *et al.* [10]), movement (Giese and Poggio [105]) and face (Wi *et al.* [243]) recognition. In addition the simplicity of the ConvNets architecture makes it an ideal candidate for custom hardware implementations (Boser *et al.* [20]) and a stream processor for ConvNets has also been built (Farabet *et al.* [84]).

Only a few previous studies have employed ConvNets to model auditory processing [65,118,152,221] (a comparison of the results are given in Tab. 6.3) but so far no study has used ConvNets for micro-Doppler sonar processing. Here, ConvNets are used to model both passive sound (microphones data) and active sound (micro-Doppler sonar data) processing systems with application to behavior classification.

6.2.2 ConvNets: stage by stage

The ConvNets architecture is composed of 2 main stages (pre-processing and classifier) that are presented in details below. A block diagram of the ConvNets architecture is given in Fig. 6.1.

6.2.2.1 Pre-processing

The first stage involves pre-processing of the raw data to compute a time-corrected instantaneous frequency reassigned (IFR) spectrogram representation of each recorded micro-Doppler signature; this is done using the method of Nelson (Fulopa and Fitz [90]). The first step of the method is to mix the ultrasonic signal such that

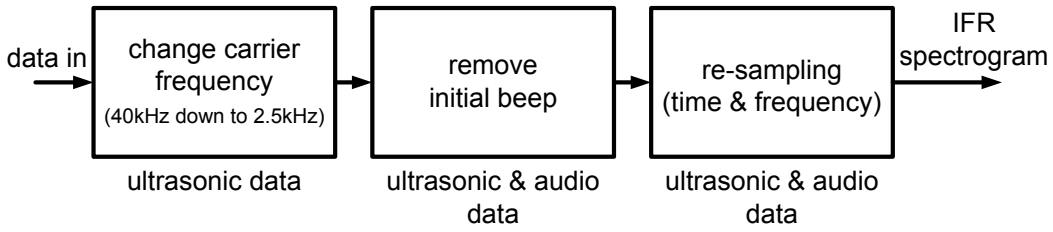


Figure 6.2: Pre-processing block diagram, first stage of the ConvNets.

the emitter carrier frequency is reduced from 40kHz to 2.5kHz. Thus, the frequency range of the IFR for ultrasonic data is set from 1.85kHz to 3.15kHz, in order to capture the frequency shifts generated by the moving body parts. In addition, the IFR channels carrying information corresponding to the ultrasonic carrier frequency (2.5kHz after down-sampling) are set to zero, in order to emphasize the micro-Doppler frequency shifts. For the passive acoustic data the IFR frequency range is set between 100Hz and 12kHz, which is contained in the human audible frequency range. To eliminate a beep sound that signals the start of the trial, the IFR channels corresponding to this frequency are set to zero. Other parameters of the IFR spectrogram function are the number of channels, which determines the frequency resolution (set to 50) and the bandwidth, which determines the number of samples or temporal resolution (set to 95). These parameter values are derived from the work done in [70], presented in Chapter 4, and provided a good compromise between reducing the size of the input and keeping enough information to maximize classification. However, it was found that varying these values did not have a significant effect on performance, as long as the rest of the ConvNet parameters were tuned afterwards (see section 6.3.3). A block diagram of the pre-processing is given in Fig. 6.2.

6.2.2.2 Classifier

Once the calculation of the IFR spectrogram of the data is finished, the second stage involves feeding these features into the ConvNets classifier. The second stage is composed of several modules or building blocks that can be arranged in various configurations according to the task requirements (for example the type of data audio or ultrasonic). These building blocks are described below and the nomenclature and definition are similar to the ones proposed by LeCun *et al.* in [146]. For each building block the variable x denotes the input 3D array composed of a_f 2D feature maps of size $a_1 \times a_2$; and y denotes the output 3D array composed of b_f 2D feature maps of

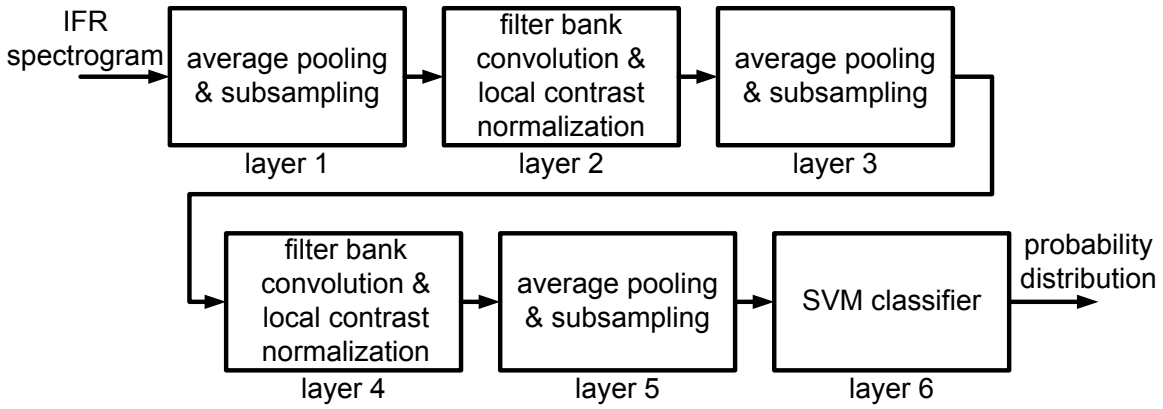


Figure 6.3: Classifier block diagram, second stage of the ConvNets.

size $b_1 \times b_2$. The i^{th} feature map is denoted as x_i, and a specific component at location (j,k) within that feature map is denoted as x_{ijk}. A block diagram of the classifier is given in Fig. 6.3.

Filter Bank Convolution The first block is the Filter Bank Convolution and it is denoted as $n.F^{f_1,f_2}_{CTA}$ where n is the number of filters; f_1,f_2 define the filter bandwidth; and CTA represents the use of the convolution (C) operation, the tanh (T) non-linearity and the absolute (A) value function. This module computes the convolution of the input signal with a set of pre-learned filters and applies the tanh and absolute function non-linearities to the output, according to the following equation 6.1:

$$y_j = \left| \tanh\left(\sum_i w_{ij} * x_i \right) \right| \tag{6.1}$$

where tanh is the hyperbolic tangent non-linearity; $*$ is the 2D discrete convolution operator; w_{ij} is a filter of size $f1 \times f2$ that connects input feature map x_i to output feature map y_j. Taking into account the border effects, the size of the output feature maps is: $b_1 = a_1 - f_1 + 1$ and $b_2 = a_2 - f_2 + 1$.

Local Contrast Normalization The second block is the Local Contrast Normalization and it is denoted as N_r, where r is the radius of a Gaussian window. This module performs local subtractive and divisive normalization enforcing competition between the spatially surrounding units in the same feature map and between units coding different features at the same spatial location. The subtractive normalization operation for a given location x_{ijk} is implemented as:

$$v_{ijk} = x_{ijk} - \sum_{ipq} g_{pq} * x_{i,j+p,k+q} \tag{6.2}$$

v_{ijk} is the subtractively normalized output and g_{pq} a Gaussian window of radius r, sum-normalized to 1. Divisive normalization then computes:

$$y_{ijk} = \frac{v_{ijk}}{max(c, \sigma_{jk})}, \tag{6.3}$$

$$\sigma^2_{jk} = \left(\sum_{ipq} g_{pq} * v^2_{i,j+p,k+q} \right) \tag{6.4}$$

where y_{ijk} is the divisively normalized output; $c = mean(\sigma_{jk})$; and σ_{jk} is the weighted standard deviation of all features within the surrounding region of radius r. The normalization operation is inspired by computational models of primary visual cortex (Carandini and Heeger [36]).

Average Pooling and Subsampling The third block is the Average Pooling and Subsampling and it is denoted as $P^{f_1,f_2}_{s_1,s_2}$, where f_1, f_2 are pooling sizes and s_1, s_2 are the subsampling step in each dimension:

$$y_{ijk} = \sum_{p=1...f1,q=1...f2} \frac{x_{i,j+p,k+q}}{f_1 * f_2} \tag{6.5}$$

Each feature map y_i is then spatially subsampled by a factor of s_1 and s_2 in the horizontal and vertical dimension, respectively, such that $b_1 = a_1/s_1$ and $b_2 = a_2/s_2$.

Classifier The fourth and last block is the Classifier and it is the module that employs the output of the previous layer to train a classifier using supervised learning and multinomial logistic regression or a linear support vector machine (SVM). In the simulations the latter was employed (implemented using the publicly available '*libsvm*' library, Chang and Lin [48]) and thus this top module is noted as SVM. The SVM module is trained using labeled data from the layer immediately below, and generates multi-class probability distributions over the trained classes using pairwise coupling (Wu *et al.* [247]).

6.2.3 ConvNets: parameters values

6.2.3.1 Micro-Doppler data

The ConvNet processing the micro-Doppler data consists of six layers, three of which are pooling layers, two convolution layers and one is the classifier; these are shown in Fig. 6.4. Only the convolution and pooling modules constitute independent model

layers, whereas the normalization module is included within the convolution layer. The size of the output of each layer, which takes into account the border effect of the convolution operation (the pooling operation has no border effect), is also shown in Fig. 6.4.

The configuration and parameters of the ConvNet were chosen as a trade-off between maximizing classification performance and minimizing simulation time. The parameters are shown in Tab. 6.1 and section 6.3.3 includes a robustness analysis for most of them. Following the nomenclature presented above, a network with selected parameters can be written in short form as:

$$P_{1,20}^{1,30} \rightarrow 8.F_{CTA}^{10,6}/N_3 \rightarrow P_{1,2}^{4,4} \rightarrow 64.F_{CTA}^{4,4}/N_0 \rightarrow P_{2,2}^{4,8} \rightarrow SVM \tag{6.6}$$

Any size of IFR spectrogram can be used as input to the ConvNets described above and it will generate a fixed number of output probability distributions corresponding to different time steps. Nevertheless a minimum number of spectrogram frames of 530 for ultrasonic (mD) data (or 2.57s of input data) is required to generate an output probability distribution at the top layer given the parameters of the network.

All receptive field sizes are shown in Tab. 6.1. These have been calculated taking into account the large overlap in the receptive fields (e.g. 10/30 for Layer 1). The high overlap in the network leads to a temporal step between top layer events of only 80 spectrogram frames (0.39 s of signal). Thus, for the example shown in Fig. 6.4, with an input spectrogram of 1025 frames (5s of signal), the number of output probability distributions at the top classifier layer should be 5/0.39 = 12.8, but the resulting number of output events is only 11 due to the border effect of the convolution layers.

6.2.3.2 Auditory data

The ConvNet processing the auditory data has exactly the same parameters as the one for the micro-Doppler data except for the first pooling layer, which is responsible for the temporal sub-sampling of the IFR spectrogram. This is due to the fact that the microphone data sampling frequency is higher (48kSps) than the one of the micro-Doppler data (12,5kSps) and in addition, the recording times of both devices were not always identical, the spectrograms corresponding to each modality have different lengths. To synchronize the outputs of the mD and audio data, the ratio of the length of the respective spectrograms was calculated and an average scaling

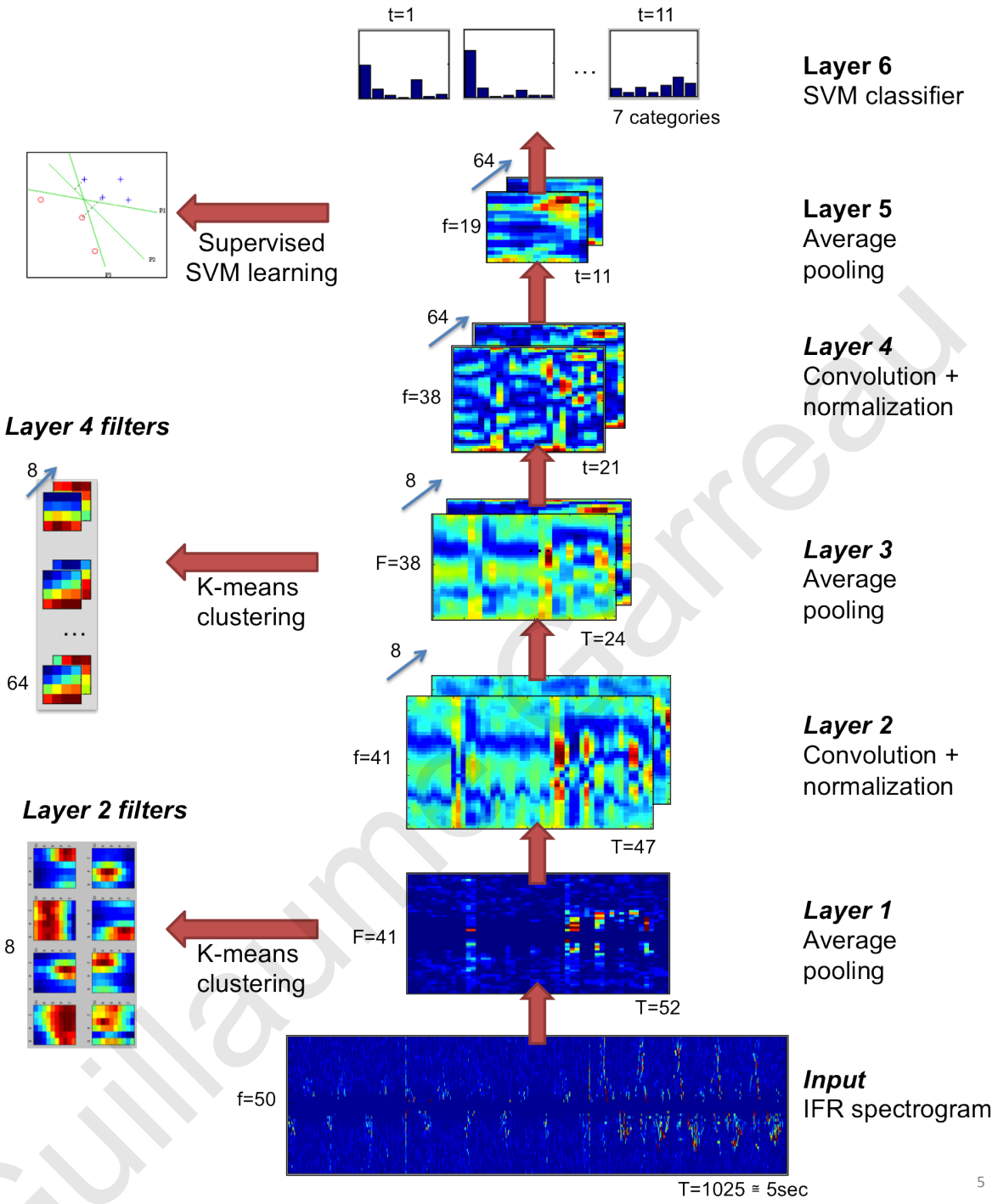


Figure 6.4: Convolutional Neural Network for ultrasonic data classification. The output of each of the six layers in the network is shown schematically for an input signal of 5s. The input is first pre-processed by calculating its IFR spectrogram, which serves as input to the model. Along the left-hand side, the different learning stages and resulting filters are also represented schematically. The output of the model consists of a probability distribution over the classes for each top layer time step, obtained using an SVM classifier (from [69]).

Parameter	Ultrasonic	Auditory
Layer 1		
Type	Average pooling	
Pooling size (freq), f_1	1	1
Pooling size (time), f_2	30 (0.15s)	72 (0.15s)
Step size (freq), s_1	1	1
Step size (time), s_2	20 (0.10s)	48 (0.10s)
Layer 2		
Type	Convolution and normalization	
Number of filters, n	8	
Filter size (freq), f_1	10	
Filter size (time), f_2	6 (0.63s)	
Normalization radius, r	3	
Layer 3		
Type	Average pooling	
Pooling size (freq), f_1	4	
Pooling size (time), f_2	4 (0.92s)	
Step size (freq), s_1	1	
Step size (time), s_2	2 (0.19s)	
Layer 4		
Type	Convolution and normalization	
Number of filters, n	64	
Filter size (freq), f_1	4	
Filter size (time), f_2	4 (1.21s)	
Normalization radius, r	0	
Layer 5		
Type	Average pooling	
Pooling size (freq), f_1	4	
Pooling size (time), f_2	8 (2.57s)	
Step size (freq), s_1	2	
Step size (time), s_2	2 (0.39s)	
Layer 6		
Type	SVM linear classifier (*libsvm*)	
SVM options	t = 0 (linear kernel), c = 1 (cost) b=1 (probability estimates)	

Table 6.1: Parameters of the ConvNet for ultrasonic (middle) and auditory (right) data processing, [69].

factor of 2.4 was obtained. This value is used to modulate the pooling parameters of the audio ConvNets such that the output of Layer 1 has the same length as for the mD ConvNets. The resulting architecture and parameters of the ConvNets for auditory data are described in Tab. 6.1 and indicated below using the short form notation:

$$P_{1,48}^{1,72} \rightarrow 8.F_{CTA}^{10,6}/N_3 \rightarrow P_{1,2}^{4,4} \rightarrow 64.F_{CTA}^{4,4}/N_0 \rightarrow P_{2,2}^{4,8} \rightarrow SVM \tag{6.7}$$

6.2.3.3 Learning

Following the processing step, which is specific to each data type set, is the identical step of learning. The collected dataset is randomly divided into a training set containing 50% of the subjects and a testing set containing the remaining 50%. Thus, each set is composed of 175 files of approximately 10s, corresponding to 7 actions $\times$ 5 subjects $\times$ 5 trials. The random division into training and test sets is repeated 50 times and the results are averaged (cross-validation). K-means clustering (see section 4.3.2.2) is used to calculate the filters of the convolution layers. First all 3D patches of size $f_0 \times f_1 \times f_2$ are computed from the previous layer output, where f_0 is the number of features maps in the previous layer, f_1 is the frequency dimension of the spectrogram and f_2 the time dimension of the spectrogram. These patches are fed into the k-means clustering algorithm to obtain n cluster centres, which are used as filters after having sum-normalized them to one. Given the sensitivity of the k-means to initial starting conditions, a procedure for computing a refined starting condition was implemented for improved solutions (Bradley and Fayyad [23]). The learning process, similar to the one used in section 4.3.2.2 is illustrated in Fig. 6.4, which includes examples of the filters learned by the mD processing ConvNets. In this case, the eight Layer 2 filters of size $f_0 = 1 \times f_1 = 10 \times f_2 = 6$ are learned from the output of Layer 1. Note that $f_0 = 1$ as the number of features maps in Layer 1 is just one, i.e. the pooled IFR spectrogram. The eight filters learned show similar characteristics to the spectro-temporal receptive fields found in the primary auditory cortex (Mesgarani *et al.* [170]). The output of Layer 1 is then convolved with each of these filters to obtain the eight output feature maps of Layer 2. Similarly, the 64 Layer 4 filters of size $f_0 = 8 \times f_1 = 4 \times f_2 = 4$ are learned from the output of Layer 3. In this case $f_0 = 8$ because Layer 3 is composed of eight feature maps. The output of Layer 3 is then convolved with each of the 64 learned filters, as described by Eq. 6.1, yielding the 64 feature maps of Layer 5.

The linear SVM coefficients of the Layer 6 classifier were learned using the '*libsvm*' library. The output of Layer 5, x, has dimensions $a_f \times a_1 \times a_2$, where a_f is the number of features maps, a_1 represents the frequency dimension, and a_2 the temporal dimension. Then the vector of data points for the SVM are computed. Each data point corresponds to one time step of x such that there are a_2 data points of size $a_f \times a_1$. As learning is supervised at this stage, a vector containing the class corresponding to each of the a_2 data points is fed to the SVM training function. The default cost value, $C = 1$, is used and the probability estimate option is switched on to allow for multi-class probability outputs during testing.

The previous steps are performed at every time step, therefore it is necessary to combine the probability distributions over several time steps in order to have a better estimation of the output probability distributions of the ConvNet. $P(C|X_t)$ is the class probability given X_t, the input feature vector to Layer 6 at time step t. Assuming conditional independence of the input feature vectors given the class, i.e. $P(X_t, X_{t-1}|C) = P(X_t|C) * P(X_{t-1}|C)$, and a uniform prior distribution, $P(C)$ can be written as:

$$P(C|X_{t-n_{win}}, ..., X_t) = \alpha * \prod_{d=1...n_{win}} P(C|X_{t-d}) \tag{6.8}$$

where α is a normalizing constant that ensures $\sum_k P(C_k|X_{t-n_{win}}, ..., X_t) = 1$; n_{win} is the window size and represents the number of probability distributions combined over time (in Layer 6 time steps); and $P(C|X_{t-n_{win}}, ..., X_t)$ represents the posterior distribution over classes given the input feature vectors between time steps $t - n_{win}$ and t.

6.2.3.4 Multimodal integration

Now that $P(C)$ is known for each one of the two modalities (audio and ultrasonic), it is possible to use multimodal integration to test whether combining information from both modalities improves the final results. The indices u and a are used to specify ultrasonic or audio modality, respectively. Let $X_{t,u}$ and $X_{t,a}$ be the input feature vectors to Layer 6 at time step t for the ConvNets.

Importantly, these outputs, $P(C|X_{t,u})$ and $P(C|X_{t,a})$, are aligned in time thanks to the adaptation of the Layer 1 pooling parameters. Assuming conditional independence of $X_{t,u}$ and $X_{t,a}$ given the class, and a flat prior probability, $P(C)$, both sources of

information can be combined:

$$P(C|X_{t,u}, X_{t,a}) = \alpha * P(C|X_{t,u}) * P(C|X_{t,a}) \tag{6.9}$$

where α is a normalizing constant that ensures $\sum_k P(C_k|X_{t,u}, X_{t,a}) = 1$. $P(C|X_{t,u}, X_{t,a})$ represents the posterior probability over classes given the input feature vectors at time step t for both modalities.

6.2.4 ConvNets: real-time

The final stage of the work is to implement the real-time processing. It is done by employing a C application that was developed to continuously read data blocks of fixed size from the mD DACQ unit and feed them to Matlab®. In order not to suffer any data loss while other operations are carried out in real time, a pipe is used for the transmission. For each incoming block of data, the IFR spectrogram is calculated first and then passed through the six-layer ConvNet that outputs a probability distribution over the classes. An efficient implementation of the model using Matlab's Image Processing toolbox, ensures that the ConvNet requires just 40ms of computation time for each second of data. Additionally, the receptive field size in the top layer was reduced from 8 to 4 such that the system only requires 1.8s of data to provide the first output probability distribution, which is then updated every 400ms. The real-time demo can be run in three different modes: (1) learning, to record the IFR spectrogram of a micro-Doppler signal, (2) training, to use stored IFR spectrograms for each action class to train the ConvNet model, and (3) testing, to compute the output probability distribution over different action classes for a given real-time input signal. A graphical user interface was developed to visualize the input IFR spectrogram, the output of the model layers, and the output probability distribution. The GUI allows the user to record new classes and train the model parameters to newly recorded data. Training is done offline very quickly. This allows users to experiment with different sets of actions. For example, training the six-layer model with 60s recordings of four different actions takes approximately three seconds. Figure 6.5 shows the GUI developed. The top-left panel allows the user to choose whether to learn new categories, train the network with the current data files or just test the model. The bottom-left panel indicates the labels of the categories and the training files currently stored for each one. The middle panel shows the output of Layers 1 to 5 of the ConvNet model. The right panel shows

the probability distribution over categories for the last 10 events (top), the current probability distribution over categories (middle) and the input IFR spectrogram (bottom). All panels were updated approximately every 500 ms.

6.3 Data collection

The model described in the previous section was tested on the dataset 2. The dataset was preprocessed and then fed to the ConvNet architecture, the classification performance and multimodal integration results are presented below.

6.3.1 Performance

The classification performance of the ConvNets for micro-Doppler and auditory data as well as their combined performance after integration, are shown in Fig. 6.6.

Results are shown for each class individually and as an average over all classes. The error bars indicate the standard deviation over 50 cross-validation repetitions. Results are averaged over all classes and plotted as a function of the temporal integration window, n_{win}, where a window of one time step corresponds to approximately 2.57s and a window of 13 time steps corresponds to approximately 7.25s. The average classification accuracy for mD data ranges between 83.4% (n_{win} = 1) and 91.6% (n_{win} = 13); for auditory data between 88.2% (n_{win} = 1) and 92.2% (n_{win} = 13); and for multimodal integration between 95.3% (n_{win} = 1) and 97.6% (n_{win} = 13). With respect to individual classes, assuming n_{win} = 13, the classification accuracy for mD data is lowest for the "come here" condition (83.0%) and highest for the "fast run" condition (97.4%); for auditory data the lowest accuracy is also for the "come here" condition (78.2%) and the highest for the "slow walk" (97.1%) condition. The multimodal integration results, for n_{win} = 13, show an accuracy above 90% for all conditions, reaching almost perfect categorization for the "fast run" (99.1%), "fast walk" (99.3%) and "slow walk" (100.0%) conditions. After a quick look at the plots, it is interesting to note that the "come here" condition, which is the condition with the lowest recognition accuracy, is greatly improved by multimodal integration and the multimodal integration always improves the recognition performance.

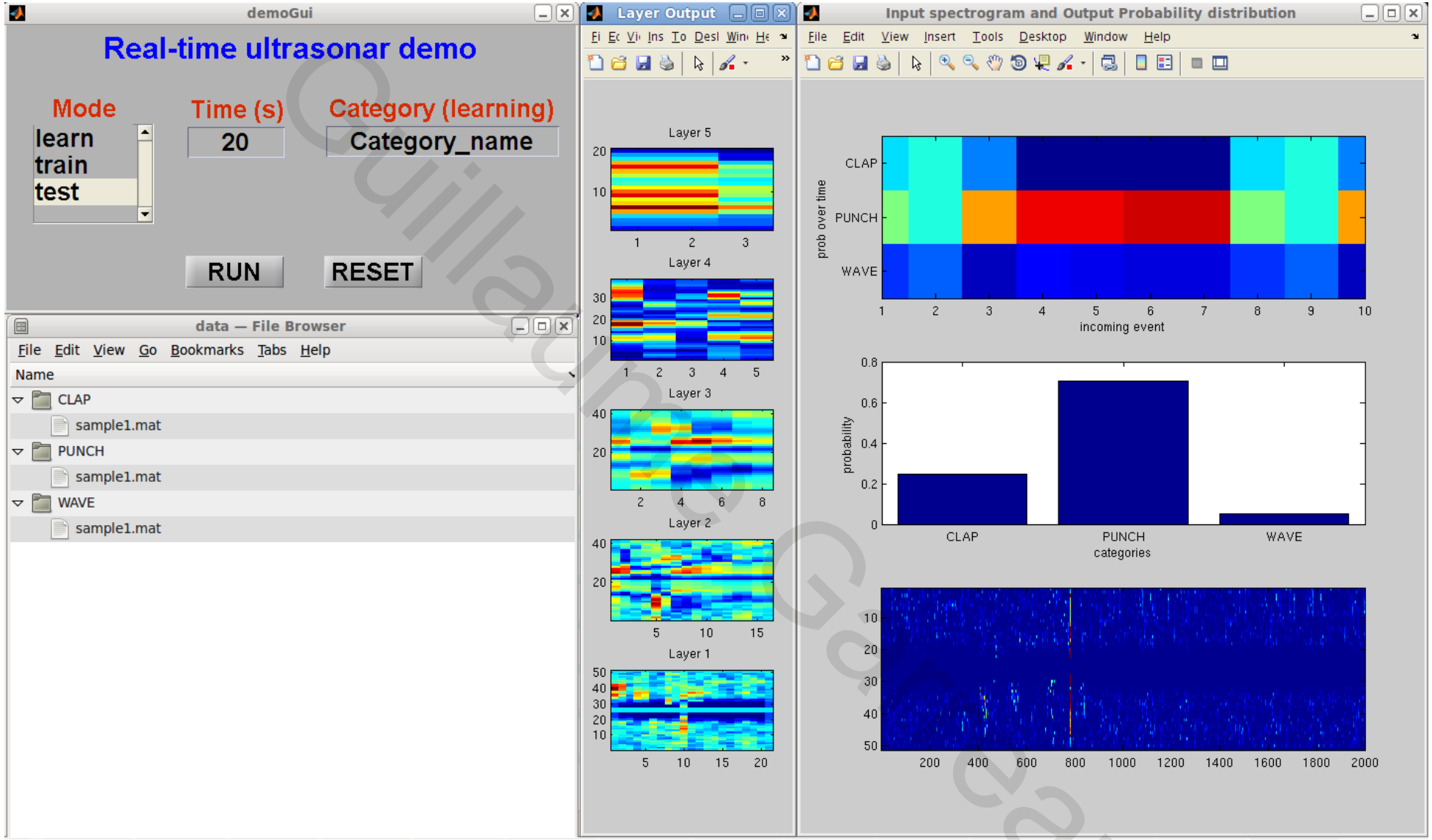


Figure 6.5: The top-left panel is for the mode selection: learn new categories, train the network with the current data files or test the model. The bottom-left panel gives the labels of the categories and the training files currently stored for each one. The middle panel shows the output of Layers 1 to 5 of the ConvNet model. The right panel shows the probability distribution over categories for the last 10 events (top), the current probability distribution over categories (middle) and the input IFR spectrogram (bottom).

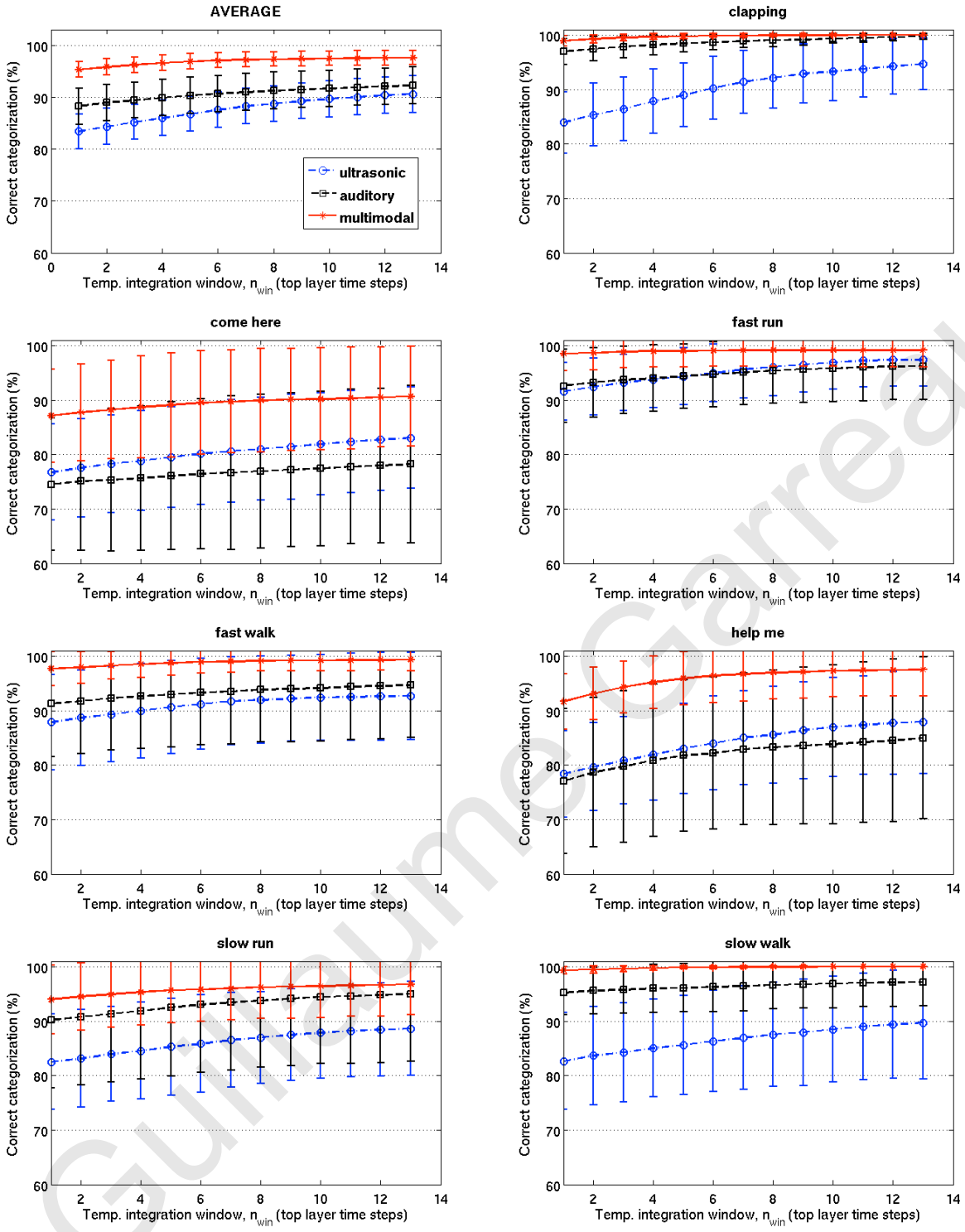


Figure 6.6: Classification results for the ConvNet models of ultrasonic and auditory processing, as well as for multimodal integration. Results are shown for the overall average over classes and for all individual classes. The error bars indicate the standard deviation over 50 cross-validation repetitions. The first top layer time step represents 2.57s of data and each additional time step adds 0.39s of new data, such that 13 time steps represent 7.25s.

6.3.2 Discussion of results

A more detailed study shows that the classification accuracy of the ConvNet models always increases with the size of the cumulative probability window. For the ultrasonic data it ranges from 84% to 91%; for the auditory data, from 88% to 94%; and for multimodal integration from 95% to 98%. This outperforms previous models on the same (Chapter 4 and [70,96]) and similar (Chapter 4 and [97,135]) ultrasonic data, and constitutes a novel benchmark for auditory and multimodal data. Furthermore, the fact that multimodal integration improves the results for all of the classes suggests that the information contained in the ultrasonic and auditory modalities complement each other. For example, for the classes with a great variability in the sounds, i.e. the "come here" and "help me" classes, the ultrasonic data provides strong cues that compensate the decreased performance of the auditory model. Similarly, for classes such as "slow walking" where the ultrasonic performance is not high, the auditory model can recognize the clear step sounds to compensate for this and achieve an overall high multimodal performance. It is also possible that the high classification results are partly due to the regularity in the step sounds and movements of the conditions that were performed using the fixed speeds of the treadmill. However, this does not seem to be the main contributing factor given that for the "clapping" condition the rate and intervals of the claps were highly heterogeneous amongst subjects and the classification performance is still very high. Interestingly, the model copes well with actions with short periodic cycles, such as "fast running", and those with longer periodic cycles, such as "help me". The hierarchical nature of the model implies that features learned at different layers will have different time scales. This suggests that for composite actions like the ultrasonic signature of "slow walking", lower level features may encode short components of the action, e.g. the forward movement of the right leg, whereas the higher level features may encode longer periods of the action, such as a complete walking cycle. Similar composite actions in the auditory domain include the "come here" and "help me" classes, which can be hierarchically organized into phonemes and words, for example. Notably, the ConvNet is also able to categorize actions with very short cycles, such as "clapping" or the walking and running conditions in the auditory domain, suggesting that the higher level features can also be composed of several periodic cycles of the same action. The model results have been averaged over 50 different trials with

randomized subject subsets for cross-validation. This prevents model over-fitting and allows the model to generalize well to new subjects. The error bars also indicate that there is high variability across trials, particularly for the "come here" and "help me" conditions. This is consistent with the experimental design, given that during these two conditions the subjects were free to decide the frequency and duration of the gestures performed and the accompanying utterances. Part of the variability is also due to initialization of k-means clusters. By running 50 trials with the same subject subset the average standard deviation due to k-means initialization was estimated as 1.2% for the ultrasonic modality and 3.3% for the auditory modality. For the combination of probability distributions over time and the integration of both modalities it is possible to use different methods as proposed here. For example, the method of averaging the probability distributions was also investigated, which yielded only slightly worse classification results than the introduced joint probability distribution method: an average of 0.1% less for the combination of probability distributions over time and 0.9% less for multimodal integration.

6.3.3 Robustness

To assess the robustness of the ConvNets model to variations in parameters the classification accuracy was computed, with $n_{win} = 1$, for different values of these parameters. The analysis was performed only on the mD data as this was the most challenging modality and, thus, the ConvNets parameters were mainly tuned to optimize its performance. The results reflect the average accuracy over 10 repetitions of random subsets of subjects, 50% for training and 50% testing. The number of repetitions was 10 instead of 50 in order to reduce the simulation time required to test the 16 different parameters analysed. These parameters were: temporal and frequency pooling size of Layers 1, 3 and 5; temporal and frequency step size of Layers 1, 3 and 5; filter size (time and frequency) of Layers 2 and 4; and the number of filters in Layers 2 and 4. Figure 6.7 shows the filled 2D contour plots of the categorization accuracy as a function of the following parameters: temporal pooling size of Layers 1, 3 and 5; temporal step size of Layers 1, 3 and 5; frequency pooling size of Layers 1, 3 and 5; frequency step size of Layers 1, 3 and 5; filter size of Layers 2 and 4; and number of filters in Layers 2 and 4. The range of values for each parameter was selected in order to provide a reasonable model output based on preliminary

results. Thus, the X and Y axes do not necessarily have a linear scale. The scale bar of each graph indicates the minimum and maximum correct classification percentage, which measures the model robustness to variations of those parameters. Out of the 16 parameters analyzed, 14 exhibit a maximum classification accuracy variation of 10–15% (typically between 70% and 80%), for the range of values tested. The two exceptions are the Layer 3 frequency step size, s1, reaching an accuracy below 20% for values above 4; and the Layer 3 step size, s2, reaching an accuracy below 60% for values above 6.

The results shown in Fig. 6.7 provide an indication of the robustness of the model to variations in the networks parameters. However, these are not intended to provide an exhaustive robustness analysis, as they are limited in a number of ways. First, the analysis was only performed for the ultrasonic model and not for the auditory model or the multimodal integration. Second, the results only take into account a single top layer time step, whereas the classification performance after integrating several time steps is likely to yield improved robustness patterns. Finally, the parameter space was analyzed by varying the values of two parameters at a time, while keeping the rest of parameters fixed. Despite these constraints, the results suggest that the model is robust to moderate variations of most of the architecture parameters (pooling size, step size and number of features). Two exceptions are the temporal and frequency step sizes in Layer 3 (subsampling), where higher values drastically reduce the model performance. This is reasonable given that increasing the subsampling step is directly proportional to the amount of information lost, such that doubling the step size means losing 50% of the information. This can represent a considerable loss given that the input to the model has already been significantly subsampled in the frequency domain by the IFR spectrogram computation and in the temporal domain by the Layer 1 operations.

6.3.4 Comparison with other classification algorithms

For comparison purposes, the same set of data was processed using a standard auditory classification method: computing a set of features based on the Mel-frequency cepstral coefficients (MFCC) of the signal, building a Gaussian Mixture Model (GMM) for each class and using a Bayesian classifier. This methodology has proven to be effective for sound-based speaker identification (Reynolds and Rose [193]) and

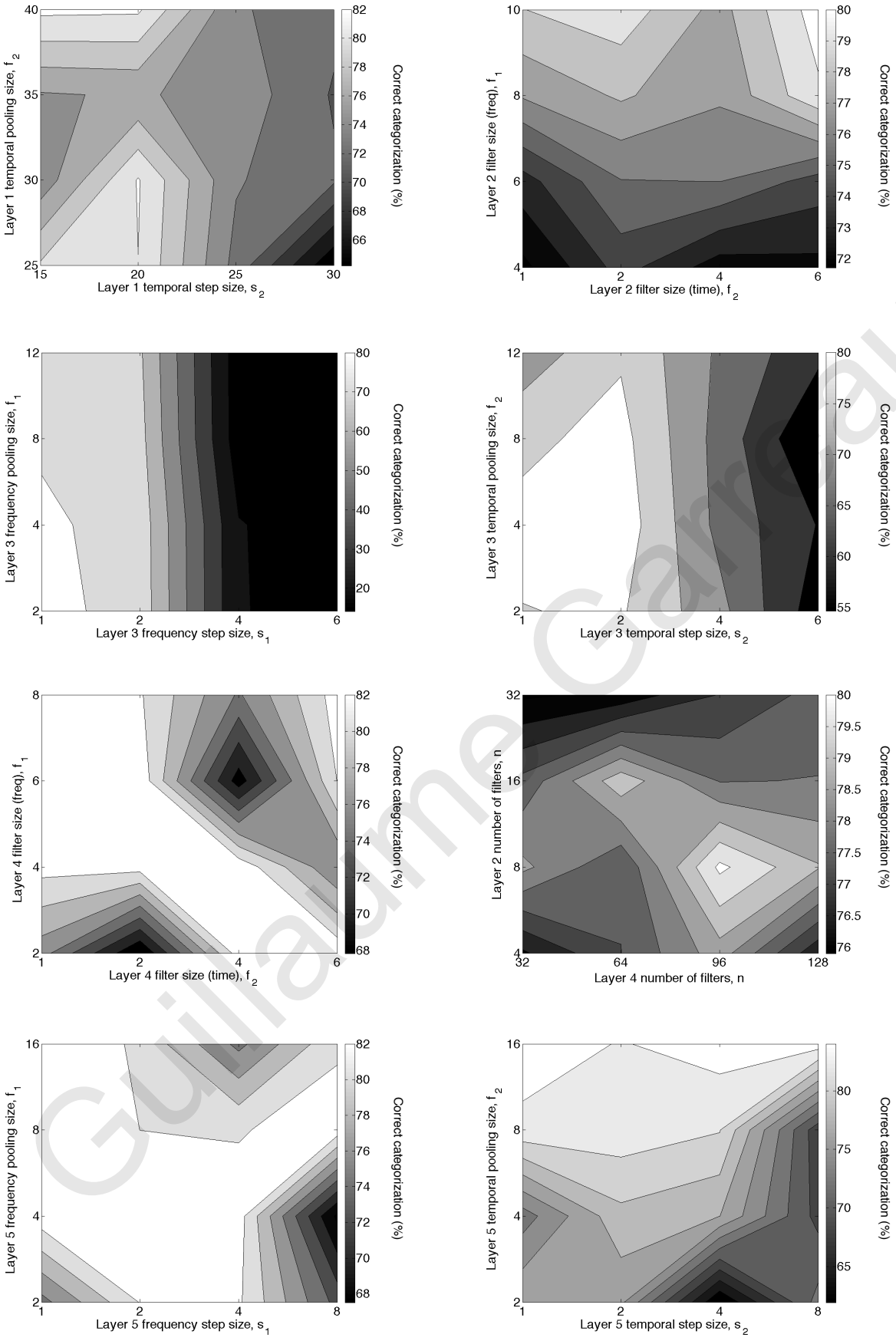


Figure 6.7: Classification performance of the micro-Doppler model as a function of the ConvNet architecture parameters. The scale bar of each graph indicates the minimum and maximum correct classification percentage, providing a measure of the model robustness to variations of those specific parameters.

Classification (%)	Micro-Doppler	Auditory
MFCC + GMM	68.9±2.9	86.0±1.8
IFR + ConvNet	83.4±3.1	88.2±3.1

Table 6.2: Classification performance of GMM versus ConvNet, [69].

walker identification (Kalgaonkar and Raf [133], Zhang and Andreou [255]) from micro-Doppler signatures.

The parameters of the model were based on these previous studies, but were optimized for this dataset by tuning them independently for each modality. The frequency range of the MFCC was set to 1.5–3.5kHz (ultrasonic) and 0–15kHz (auditory), and the number of filters in the MFCC filter-bank was set to 128 filters (ultrasonic) and 512 filters (auditory). The remaining parameters, i.e. number of dimensions in the MFCC feature vector (20, 40, 80 or 120), number of frames per 10s file (10, 20 or 30) and the number of mixtures in the GMM (5, 10 or 20) were also tuned independently for each modality but resulted in the same optimum values for both, i.e. 80, 20 and 10, respectively. The 80-dimensional feature vector was composed of the first 40 MFCC and the 40 first-order differential MFCC, in order to include temporal information. Reducing the number of dimensions to 20 or 40 reduced the performance, but increasing it to 120, did not show significant improvements; this was consistent with previous studies [255]. The feature vector was computed for each frame of the signal, where each 10s file had 20 frames, similar to the Layer 5 output of the ConvNet. The parameters of the Gaussian mixture models were initialized using the k-means clustering algorithm (with initial clusters from centroids and variances of random partitions of the dataset) and learned using the expectation-maximization (EM) algorithm. A simple Bayesian classifier was used to compute the probability that the input frames from the test dataset belonged to each of the seven classes. The training and testing datasets and the random subsets of subjects were kept the same as for the ConvNet experiments. The classification results are summarized in Tab. 6.2.

The classification performance was significantly lower for the ultrasonic data (69%) and slightly lower for the auditory data (86%) as compared to the ConvNet model. In both cases the same training and testing data subsets were employed and the 10s files were segmented into the same number of frames or events before the

classification stage. This suggests the hierarchical structure of the ConvNet might have advantages over the GMM in coping with the variability and generalizing to the new data.

6.3.5 Comparison with other work

Despite the recent resurgence and success of ConvNets for object recognition, only a few studies have applied this methodology to the auditory domain. Here, four of these models were analyzed by comparing their input, architecture parameters and learning methods as shown in Tab. 6.3. The model proposed in this work can be applied to the classification of any sound, whereas previous models focus on speech detection [221] or music classification [65,118,152]. All models employ some kind of spectro-temporal representation of the sound signal, applied over frames of a fixed length and typically with some overlap.

With respect to architecture, previous models have either two [118,152,221] or three [65] feature convolution layers, and only one of them has separate pooling/subsampling layers [65]. The rest of the models include a temporal subsampling step within the feature convolution layer [118,152,221]. This means more information is lost in the temporal dimension as compared to models, such as the one proposed here, which average over neighboring units before subsampling and thus preserve some of the contextual information. Additionally, average pooling and subsampling of the frequency domain was performed in order to increase invariance to transformations and distortions in that domain. Previous learning algorithms include unsupervised methods that learn features useful for classification, such as Predictive Sparse Decomposition (PSD) [118] or Contrastive Divergence (CD) using an equivalent Convolutional Deep Belief Network (CDBN) [65], and supervised methods, such as Stochastic Gradient Descent (SGD [65,152,221]), that learn to associate the top layer output to the different classes and refine the intermediate layer features. In this work, k-means clustering was employed to learn the intermediate layer features in an unsupervised manner, and supervised learning was subsequently used to train a linear SVM that can assign the Layer 5 output to different classes. Regarding ultrasonic processing, this is the first model, to the best of my knowledge, that applies ConvNets to the classification of micro-Doppler signatures obtained using an ultrasonic device.

Model	Input	Architecture parameters	Learning
Speech detection [221]	20 features (log SNR of mel-frequency bands per 16ms-frame	$25.F^{20,1} \rightarrow 25.F^{1,3} \rightarrow$ classifier	Stochastic Gradient Descent
Music classification [152]	13 features (MFCCs) per 190 frames of 27 ms (50% overlap)	$3.F^{10,13}_{1,4} \rightarrow 15.F^{10,1}_{1,4} \rightarrow 65.F^{10,1}_{1,4} \rightarrow$ classifier	Stochastic Gradient Descent
Music classification [118]	96 features (constant-Q transform) per 46 ms-frame (50% overlap)	$N_r \rightarrow 512.F^{96,1\ frame}$ or $4 \times 128.F^{24,1\ octave} \rightarrow$ SVM	Predictive Sparse Decomposition
Music classification [65]	2 parallel streams: 12 chroma features (pitch) and 12 timbre features per beat	$2 \times 100.F^{12,8\ beats} \rightarrow P^{1,4\ beats} \rightarrow 100.F^{200,8\ bars} \rightarrow P^{1,4\ bars} \rightarrow$ logistic regression	Unsupervised feature learning with CDBN and Stochastic Gradient Descent

Table 6.3: Comparison of existing auditory models based on ConvNets, [69].

6.4 Conclusion

In this chapter, a mD sonar system, a data collection experiment and a ConvNet model that processes mD and auditory data for human action and behavior classification were presented. The biologically-inspired ConvNets model can use the mD data for classification of human actions with high efficiency (91%) and can be coupled with a parallel ConvNet architecture for passive sound recognition to further improve its performance (98%). The model shows robustness to parameter variations and runs in real time, as demonstrated by the demo already implemented (Fig. 6.5). This is partly thanks to homogeneity of the network (weight sharing), which also enhances the generalization ability of the model, even with limited training data. These properties make the system suitable for a wide range of applications in the context of security, surveillance and human behavior monitoring. The main novelties of the proposed integrated system are, first, performing action classification using mD sonar signals derived from a custom made compact hardware system of low power consumption and cost. Second, the use of ConvNets to process mD ultrasonic data based on an efficient computational implementation that can be run in real time. And third, the probabilistic integration of two complementary sources of acoustic information, ultrasonic and auditory, that significantly improves system performance.

Future lines of research are intended to explore the applicability of the sensor to real-life scenarios. In this sense, experiments will be developed to evaluate aspects such as the distance limits of the system, particularly in outdoor conditions, and the effects on accuracy of the angle of incidence between the ultrasonic module and the target object. One key aspect here is the potential active control of the mD sonar for interrogating the scene: unlike audio, which comes from all directions without control, the sonar device can be activated intermittently and directed towards the desired targets. Another interesting extension would be the implementation of the whole model on a FPGA, in order to obtain a compact and fast system that can be run in real time. A recent study demonstrates the feasibility of running ConvNets on FPGAs and the excellent performance that can be achieved [84]. Although this approach has been typically limited to vision systems, preliminary results from work still in progress shows the successful implementation on FPGA of the proposed ConvNet for ultrasonic and auditory action classification.

Chapter 7

Conclusions

7.1 Summary

Acoustic scene analysis is a vast field that traditionally focuses on sound recognition in the audible range and on how the brain segregates and classifies the different sound sources into meaningful objects. Previous research has mainly focused on the audible range, since this is the most familiar one to humans.

In this thesis, a system that can collect data from three different frequency ranges (defined as LF or seismic range, audible range and HF or ultrasonic range) was developed.

The first step towards the development of this new system was to recognize and solve the issue of the synchronization of the different data sources. A unique solution was designed and implemented. The proposed solution (FM SYNC unit) relies on VHF radio operated in the 72.5MHz band to achieve $< 1\mu$s resolution in synchronization, as well as functionality over distances exceeding 100m. The system is capable of synchronized signal sampling with an accuracy of 1μs, whilst correcting the sampling clock frequency every 2μs.

Furthermore, a flexible bioinspired data acquisition unit (DACQ unit) is presented. The hardware platform is a custom distributed wireless data acquisition system for passive and active multimodal sensing. In addition, it is achieved with a low-cost, low-power compact solution, encouraging a full hardware implementation targeting real-time application of bioinspired models.

A network of FM SYNC unit and DACQ unit or units have been used to implement a sand scorpion-based neural network model for person localization using seismic waves. The system was used also to classify individuals, gender, actions or

mode of transport using the reflected ultrasonic (with the UFTRX unit) wave and time-frequency representations of mD signatures. Thanks to an additional FPGA on the DACQ system, a model of the auditory pathway (basilar membrane and thalamocortical network) was partially implemented. The compact implementation processes in real-time the data collected by the DACQ unit and streamed to the second FPGA before displaying the results on a laptop screen. Finally, it was shown by using the acoustic data collection hardware and multimodal sensors (audio and ultrasonic), multimodal sensory fusion can be performed, in order to increase the performance in recognition and classification tasks. This was demonstrated using another bioinspired model: the ConvNet. The model shows robustness to parameter variations and runs in real time. This is partly thanks to homogeneity of the network (weight sharing), which also enhances the generalization ability of the model, even with limited training data.

Overall, the highly synchronized multimodal acoustic data acquisition hardware system is unique and has the potential to support a wide range of applications in the context of robotics, security, surveillance, human behavior monitoring and even gaming. Some examples of potential applications include: systems that can transparently and non-invasively provide situation awareness (warning signals), for example at busy road crossings or policing bicycle lanes; smart motion detection for buildings (PIR substitute for light control or bell ring); detecting movement in a smoke-filled room where cameras are unsuitable (firemen and rescue teams); surveillance unit for surveying an emergency site; recognition of normal/abnormal behavior in a crowd (for example panic on a platform train or stadium); monitoring of elderly at home; human-computer interface for simulator/video games; or autonomous video conference system. The proposed system allows compact and full hardware implementation of bioinspired models, which can be used for the development of more refined bioinspired models and increasing our knowledge and understanding of the brain. Last, it is a custom made compact hardware system characterized by low power consumption and cost.

7.2 Future work

As an extension of the work presented here, the acoustic data acquisition system developed seems an excellent base to study our auditory sense and implement bioinspired models. This will allow a better understanding of how the auditory pathway in the human brain works and will permit the creation of more precise models. Some examples of applications that can be envisioned, in the next years, based on the developed auditory pathway hardware implementation are: identification of the number sound sources, multiple source localization, acoustic object or soundscape recognition, and speech recognition. This would be possible with a low cost and energy efficient system. On a longer time scale, a chip implanted in the brain could contribute towards 'fixing' brain areas by replacing damaged ones with an electronic implementation of their functions. Furthermore, other kind of models can be implemented on the hardware developed, such as those simulating sand scorpion localization ability or bats and dolphins with the use of ultrasonic waves. With a shift of the 'useful' frequency span of the ultrasonic return into the audible frequency window, we expect that the auditory pathway model can process the ultrasonic data with the same performance than the regular audible sounds. Another way of improving recognition is the creation of a skeleton model of the human body in order to generate artificial mD signatures and match artificial data with real data.

7.3 Contributions

1. The design of a unique multimodal data acquisition hardware system with high synchronization under the μs was presented and its performance demonstrated. It is composed of 2 type of units: the FM SYNC unit that provides the FM synchronization reference and the DACQ units that collect data. A network of 1 FM SYNC unit and multiple DACQ units was used in various experiments.

2. The demonstration of various bioinspired methods of sound processing for analysing the environment, i.e. acoustic object detection, location of a person walking, action or mode of transport recognition, gender classification and individuals identification. Those methods where inspired from sand scorpions

(low frequency domain), bats and dolphins (ultrasonic frequency domain) and the human auditory pathway (audio frequency domain).

3. The sensory fusion of active and passive acoustic data for action recognition in realtime using convolutional networks (ConvNets).

4. The implementation in realtime using FPGA and digital hardware of a partial model of the auditory pathway starting from the outer ear with a cochlea and partial thalamocortical network.

Bibliography

[1] S. Adams, T. Wennekers, G. Bugmann, S. Denham, and P. Culverhouse, "Application of arachnid prey localisation theory for a robot sensorimotor controller," *Neurocomputing*, vol. 74, no. 17, pp. 3335–3342, October 2011.

[2] AGIGATECH, "Non-volatile RAM," AGIGA TECH company, March 2014. [Online]. Available: http://www.agigatech.com/nvram.php

[3] T. W. Anderson, "Estimation for autoregressive moving average models in the time and frequency domains," *The Annals of Statistics*, vol. 5, no. 5, pp. 842–865, September 1977.

[4] A. G. Andreou, R. C. Meitzler, K. Strohbehn, and K. A. Boahen, "Analog VLSI neuromorphic image acquisition and pre-processing systems," *Neural Networks*, vol. 8, no. 7-8, pp. 1323–1347, July 1995.

[5] A. Andreou, "Energy and information processing in biological and silicon sensory systems," in *Proceedings of the 7th International Conference on Microelectronics for Neural, Fuzzy and Bio-Inspired Systems (MicroNeuro)*, July 1999, pp. 21–25.

[6] T. C. Andringa and M. E. Niessen, "Real-world sound recognition: A recipe," *Proceedings of 1st Workshop on Learning Semantics in Audio Signals (LSAS)*, pp. 106–118, 2006.

[7] A. auf der Maur, "Limits of exposure to airborne ultrasound," *Annual American Conference Industrial Hygiene*, vol. 12, pp. 177–181, 1985.

[8] A. Balleri, K. Chetty, and K. Woodbridge, "Classification of personnel targets by acoustic micro-Doppler signatures," *The Institution of Engineering and Technology (IET) Radar, Sonar Navigation*, vol. 5, no. 9, pp. 943–951, December 2011.

[9] A. Balleri, K. Woodbridge, and K. Chetty, "Frequency-agile non-coherent ultrasound radar for collection of micro-Doppler signatures," in *Proceedings of the IEEE Radar Conference (RADAR)*, 2011, pp. 45–48.

[10] B. Barshan, B. Ayrulu, and S. Utete, "Neural network-based target differentiation using sonar for robotics applications," *IEEE Transactions on Robotics and Automation*, vol. 16, no. 4, pp. 435–442, August 2000.

[11] K. Behrendt and K. Fodero, "The perfect time: An examination of time synchronization techniques," in *Proceedings of DistribuTECH*, Schweitzer Engineering Laboratories, Inc. PennWell, February 2006.

[12] E. Benetos, M. Lagrange, and S. Dixon, "Characterisation of acoustic scenes using a temporally-constrained shift-invariant model," in *Proceedings of the International Conference on Digital Audio Effects*, September 2012.

[13] V. Benichoux and R. Brette, "A functional spiking model of the ITD processing pathway of the barn owl," *BioMed Central (BMC) Neuroscience*, vol. 12, no. Suppl 1, p. 20, 2011.

[14] V. Benichoux, M. Stimberg, B. Fontaine, and R. Brette, "A unifying theory of ITD-based sound azimuth localization at the behavioral and neural levels," *BioMed Central (BMC) Neuroscience*, vol. 14, no. Suppl 1, pp. 39–40, 2013.

[15] L. Bi, O. Tsimhoni, and Y. Liu, "Using image-based metrics to model pedestrian detection performance with night-vision systems," *IEEE Transactions on Intelligent Transportation Systems*, vol. 10, no. 1, pp. 155–164, March 2009.

[16] Z. Bian, F. Chudak, W. G. Macready, L. Clark, and F. Gaitan, "Experimental determination of ramsey numbers with quantum annealing," *arXiv, Quantum Physics*, vol. 1842, no. 2, p. 6, January 2012.

[17] B. Boashash, *Time Frequency Signal Analysis and Processing*, B. Boashash, Ed. Elsevier, 2003.

[18] T. M. Böhm, L. Shestopalova, A. Bendixen, A. G. Andreou, J. Georgiou, G. Garreau, P. Pouliquen, A. Cassidy, S. L. Denham, and I. Winkler, "The role of perceived source location in auditory stream segregation: Separation affects sound organization, common fate does not," *Learning & Perception*, vol. 5, no. 2, pp. 55–72, June 2013.

[19] T.-L. Books, *Speed and Power (Understanding Computers)*, 2nd ed., ser. Understanding Computers. Time Life Education, February 1990.

[20] B. Boser, E. Sackinger, J. Bromley, Y. Le Cun, and L. Jackel, "An analog neural network processor with programmable topology," *IEEE Journal of Solid State Circuits*, vol. 26, no. 12, pp. 2017–2025, 1991.

[21] M. Bousbia-Salah, A. Redjati, M. Fezari, and M. Bettayeb, "An ultrasonic navigation system for blind people," in *Proceedings of the IEEE International Conference on Signal Processing and Communications (ICSPC)*, November 2007, pp. 1003–1006.

[22] J. M. Brader, W. Senn, and S. Fusi, "Learning real-world stimuli in a neural network with spike-driven synaptic dynamics," *Neural Computation*, vol. 19, no. 11, pp. 2881–2912, November 2007.

[23] P. S. Bradley and U. M. Fayyad, "Refining initial points for k-means clustering," in *Proceedings of the 15th International Conference on Machine Learning (ICML)*. Morgan Kaufmann Publishers Inc., 1998, pp. 91–99.

[24] A. S. Bregman, *Auditory Scene Analysis: The Perceptual Organization of Sound*. MIT Press, 1990.

[25] R. Brette and D. Goodman, "Brian: a simple and flexible simulator for spiking neural networks," *The Neuromorphic Engineer*, July 2009.

[26] R. Brette, M. Rudolph, T. Carnevale, M. Hines, D. Beeman, J. M. Bower, M. Diesmann, A. Morrison, P. H. Goodman, A. P. Davison, S. El Boustani, and A. Destexhe, "Simulation of networks of spiking neurons: A review of tools and strategies," *Journal of Computational Neuroscience*, vol. 23, no. 3, pp. 349–398, November 2007.

[27] G. J. Brown and D. Wang, "Separation of speech by computational auditory scene analysis," in *Speech Enhancement*, ser. Signals and communication Technology, S. M. J. Benesty and J. Chen, Eds. Springer Berlin Heidelberg, 2005, ch. 16, pp. 371–402.

[28] P. Brownell and R. D. Farley, "Detection of vibrations in sand by tarsal sense organs of the nocturnal scorpion Paruroctonus Mesaensis," *Journal of Comparative Physiology A: Neuroethology, Sensory, Neural, and Behavioral Physiology*, vol. 131, no. 1, pp. 23–30, 1979.

[29] ——, "Orientation to vibrations in sand by the nocturnal scorpion Paruroctonus Mesaensis: Mechanism of target localization," *Journal of Comparative Physiology A: Neuroethology, Sensory, Neural, and Behavioral Physiology*, vol. 131, no. 1, pp. 31–38, 1979.

[30] ——, "Prey-localizing behaviour of the nocturnal desert scorpion Paruroctonus Mesaensis: Orientation to substrate vibrations," *Animal Behaviour*, vol. 27, pp. 185–193, February 1979.

[31] P. H. Brownell and J. L. van Hemmen, "Vibration sensitivity and a computational theory for prey-localizing behavior in sand scorpions," *American Zoologist*, vol. 41, no. 5, pp. 1229–1240, 2001.

[32] J. Bryan and Y. Kim, "Classification of human activities on UWB radar using a support vector machine," in *Proceedings of the IEEE Antennas and Propagation Society International Symposium (APSURSI)*, July 2010, pp. 1–4.

[33] C. Cai, W. Liu, J. Fu, and L. Lu, "Empirical mode decomposition of micro-Doppler signature," in *Proceedings of the IEEE International Radar Conference*, May 2005, pp. 895–899.

[34] C. Cai, W. Liu, J. Fu, and Y. Lu, "Radar micro-Doppler signature analysis with hht," in *IEEE Transactions on Aerospace and Electronic Systems*, vol. 46, no. 2, April 2010, pp. 929–938.

[35] N. Caporale and Y. Dan, "Spike timing-dependent plasticity: A hebbian learning rule," *Annual Review of Neuroscience*, vol. 31, no. 1, pp. 25–46, February 2008.

[36] M. Carandini and D. J. Heeger, "Normalization as a canonical neural computation," *Nature Reviews Neuroscience*, vol. 13, no. 1, pp. 51–62, January 2012.

[37] A. Cassidy and A. Andreou, "Dynamical digital silicon neurons," in *Proceedings of the IEEE Biomedical Circuits and Systems Conference (BioCAS)*, November 2008, pp. 289–292.

[38] A. Cassidy, A. Andreou, and J. Georgiou, "Design of a one million neuron single fpga neuromorphic system for real-time multimodal scene analysis," in

45th Annual Conference on Information Sciences and Systems (CISS), March 2011, pp. 1–6.

[39] A. Cassidy, S. Denham, P. Kanold, and A. Andreou, "Fpga based silicon spiking neural array," in *Proceedings of the IEEE Biomedical Circuits and Systems Conference (BIOCAS)*, November 2007, pp. 75–78.

[40] A. Cassidy, Z. Zhang, and A. Andreou, "Neuromorphic interconnects using ultra wideband radio," in *Proceedings of the IEEE Biomedical Circuits and Systems Conference (BioCAS)*, November 2008, pp. 297–300.

[41] A. Cassidy, A. G. Andreou, and J. Georgiou, "A combinational digital logic approach to stdp." in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2011, pp. 673–676.

[42] A. Cassidy, T. Murray, A. G. Andreou, and J. Georgiou, "Evaluating on-chip interconnects for low operating frequency silicon neuron arrays." in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2011, pp. 2437–2440.

[43] A. S. Cassidy, J. Georgiou, and A. G. Andreou, "Design of silicon brains in the nano-CMOS era: Spiking neurons, learning synapses and neural architecture optimization," *Neural Networks*, vol. 45, pp. 4–26, April 2013.

[44] G. Cauwenberghs, M. Stanacevic, and G. Zweig, "Blind broadband source localization and separation in miniature sensor arrays," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 3, May 2001, pp. 193–196.

[45] G. Cauwenberghs, A. Andreou, J. West, M. Stanacevic, A. Celik, P. Julian, T. Teixeira, C. Diehl, and L. Riddle, "A miniature low-power intelligent sensor node for persistent acoustic surveillance," *Proceedings of SPIE 5796, Unattended Ground Sensor Technologies and Applications*, vol. 7, pp. 294–305, June 2005.

[46] G. Cauwenberghs, R. T. Edwards, Y. Deng, R. Genov, and D. Lemonds, "Neuromorphic processor for real-time biosonar object detection," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 4, May 2002, pp. 3984–3987.

[47] V. Chan, S.-C. Liu, and A. van Schaik, "AER EAR: A matched silicon cochlea pair with address event representation interface," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 54, no. 1, pp. 48–59, 2007.

[48] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011.

[49] M. Chen, X. Wang, and X. Li, "Coordinating processor and main memory for efficient server power control," in *Proceedings of the International Conference on Supercomputing (ICS)*, 2011, pp. 130–140.

[50] L. Chittka and A. Brockmann, "Perception space - the final frontier," *PLoS Biology*, vol. 3, no. 4, pp. 0564–0568, April 2005.

[51] C. T. Clarke, L. Qiang, H. Peremans, and A. Hernandez, "FPGA implementation of a neuromimetic cochlea for a bionic bat head," in *Proceedings of the 14th International Conference on Field Programmable Logic and Application (FPL)*, ser. Lecture Notes in Computer Science. Springer, 2004, vol. 3203, pp. 1073–1075.

[52] M. Coath, E. Balaguer-Ballester, S. Denham, and M. Denham, "The linearity of emergent spectro-temporal receptive fields in a model of auditory cortex," *Biosystems*, vol. 94, no. 1–2, pp. 60–67, October 2008.

[53] M. Coath, R. Mill, S. Denham, and T. Wennekers, "Emergent feature sensitivity in a model of the auditory thalamocortical system," in *From Brains to Systems*, ser. Advances in Experimental Medicine and Biology, C. Hernández, R. Sanz, J. Gómez-Ramirez, L. S. Smith, A. Hussain, A. Chella, and I. Aleksander, Eds. Springer New York, 2011, vol. 718, pp. 7–17.

[54] M. Coath, S. Sheik, E. Chicca, G. Indiveri, S. Denham, and T. Wennekers, "A robust sound perception model suitable for neuromorphic implementation," *Frontiers in Neuroscience*, vol. 7, no. 278, January 2014.

[55] L. Cochrane, "Mark products 4.5 hz 3-component l15b sensor," Data sheet, March 2014. [Online]. Available: http://seismicnet.com/geophone/index.html\#L15B

[56] J. Colgate and N. Hogan, "Robust control of dynamically interacting systems." *International Journal of Control*, vol. 48, no. 1, pp. 65–88, 1988.

[57] F. C. Commission, "Radio spectrum allocations 101," Federal Communications Commission, April 2013.

[58] W. C. Cummings and P. O. Thompson, "Underwater sounds from the blue whale, Balaenoptera Musculus," *Journal of the Acoustical Society of America*, vol. 50, no. 4b, pp. 1193–1198, May 1971.

[59] D. Datcu and L. J. M. Rothkrantz, "Emotion recognition using bimodal data fusion," in *Proceedings of the 12th International Conference on Computer Systems and Technologies*, ser. CompSysTech '11. New York, NY, USA: ACM, 2011, pp. 122–128.

[60] K. David and A. Flach, "CAR-2-X and pedestrian safety," *IEEE Vehicular Technology Magazine*, vol. 5, no. 1, pp. 70–76, March 2010.

[61] B. de Gelder, "Uncanny sight in the blind," *Scientific American*, vol. 302, pp. 60–65, May 2010.

[62] B. de Gelder, M. Tamietto, G. van Boxtel, R. Goebel, A. Sahraie, J. van den Stock, B. M. Stienen, L. Weiskrantz, and A. Pegna, "Intact navigation skills after bilateral loss of striate cortex," *Current Biology*, vol. 18, no. 24, pp. 1128–1129, December 2008.

[63] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[64] A. Dibazar, A. Bangalore, H. ook Park, S. George, W. Yamada, and T. Berger, "Hardware implementation of dynamic synapse neural networks for acoustic sound recognition," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, 2006, pp. 2015–2022.

[65] S. Dieleman, P. Brakel, and B. Schrauwen, "Audio-based music classification with a pretrained convolutional network," in *Proceedings of the International Symposium on Music Information Retrieval (ISMIR)*, A. Klapuri and C. Leider, Eds. University of Miami, 2011, pp. 669–674.

[66] C. J. Doppler, "Ueber das farbige licht der doppelsterne und einiger anderer gestirne des himmels [on the coloured light of the binary stars and some other stars of the heavens]," in *Proceedings of the Royal Bohemian Society of Sciences*, vol. 2, no. 5, 1842, pp. 465–482.

[67] G. Downey, "Getting around by sound: Human echolocation," Plos, June 2011. [Online]. Available: http://blogs.plos.org/neuroanthropology/2011/06/14/getting-around-by-sound-human-echolocation/

[68] I. E. Dror, F. L. Florer, D. Rios, and M. Zagaeski, "Using artificial bat sonar neural networks for complex pattern recognition: Recognizing faces and the speed of a moving target." *Biological Cybernetics*, vol. 74, pp. 331–338, April 1996.

[69] S. Dura-Bernal, G. Garreau, A. G. Andreou, J. Georgiou, S. Denham, and T. Wennekers, "Multimodal integration of micro-Doppler sonar and auditory signals for behavior classification with convolutional networks," *International Journal of Neural Systems*, vol. 23, no. 5, October 2013.

[70] S. Dura-Bernal, G. Garreau, C. Andreou, A. Andreou, J. Georgiou, T. Wennekers, and S. Denham, "Human action categorization using ultrasound micro-Doppler signatures," in *Proceedings of the Second international conference on Human Behavior Unterstanding*, ser. HBU'11. Berlin, Heidelberg: Springer-Verlag, September 2011, pp. 18–28.

[71] D. Eadline, "Preparing for the revolution : Dual-core technology for HPC clusters," SYS-CON Media, March 2006. [Online]. Available: http://linux.sys-con.com/node/193382

[72] T. Economist, "Rambo Rambus (Intel and Rambus develop new faster dynamic random-access memory computer chip)." *The Economist (US)*, February 1997. [Online]. Available: http://www.highbeam.com/doc/1G1-19104133.html

[73] A. Ekimov and J. M. Sabatier, "Vibration and sound signatures of human footsteps in buildings," *Journal of the Acoustical Society of America*, vol. 120, no. 2, pp. 762–768, June 2006.

[74] ——, "Human motion analyses using footstep ultrasound and Doppler ultrasound," *Journal of the Acoustical Society of America*, vol. 123, no. 6, pp. 149–154, May 2008.

[75] T. Ellis, "Multi-camera video surveillance," in *Proceedings of the 36th Annual International Carnahan Conference on Security Technology*, 2002, pp. 228–233.

[76] J. Elson, "Time synchronization in wireless sensor networks," Ph.D. dissertation, University of California, Los Angeles, May 2003.

[77] J. Elson, L. Girod, and D. Estrin, "Fine-grained network time synchronization using reference broadcasts," *ACM Special Interest Group on Operating Systems (SIGOPS) Operating Systems Review (OSR)*, vol. 36, no. SI, pp. 147–163, December 2002.

[78] P. Emma and E. Kursun, "Opportunities and challenges for 3D systems and their design," *IEEE Design & Test of Computers*, vol. 26, no. 5, pp. 6–14, October 2009.

[79] P. C. Engelbrecht, T. Makany, K. Meadmore, R. Dudley, and I. E. Dror, "It is not worth learning if it is not remembered: designing e-learning to increase memory," in *Proceedings of the International Technology, Education and Development Conference (INTED)*, March 2007.

[80] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Power challenges may end the multicore era," *Communications of the Association for Computing Machinery (ACM)*, vol. 56, no. 2, pp. 93–102, February 2013.

[81] F. Fahy and J. Walker, *Fundamentals of Noise and Vibration*, E. . F. Spon, Ed. Routledge, 1998.

[82] D. P. Fairchild and R. M. Narayanan, "Micro-Doppler radar classification of human motions under various training scenarios," in *Proceedings of Society of Photo-Optical Instrumentation Engineers (SPIE), Active and Passive Signatures IV*, vol. 8734, May 2013.

[83] Y. Fang, K. Yamada, Y. Ninomiya, B. Horn, and I. Masaki, "A shape-independent method for pedestrian detection with far-infrared images," *IEEE Transactions on Vehicular Technology*, vol. 53, no. 6, pp. 1679–1697, November 2004.

[84] C. Farabet, Y. LeCun, K. Kavukcuoglu, E. Culurciello, B. Martini, P. Akselrod, and S. Talay, "Large-scale FPGA-based convolutional networks," in *Scaling up Machine Learning: Parallel and Distributed Approaches*, R. Bekkerman, M. Bilenko, and J. Langford, Eds. Cambridge University Press, 2011, pp. 399–419.

[85] D. Fasnacht and G. Indiveri, "A PCI based high-fanout AER mapper with 2 GiB RAM look-up table, 0.8μs latency and 66MHz output event-rate," in *45th Annual Conference on Information Sciences and Systems (CISS)*, Johns Hopkins University, March 2011, pp. 1–6.

[86] D. Fasnacht, A. Whatley, and G. Indiveri, "A serial communication infrastructure for multi-chip address event systems," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, 2008, pp. 648–651.

[87] R. Fish, "The future of computers - part 1: Multicore and the memory wall," EDN Network, November 2011. [Online]. Available: http://www.edn.com/design/systems-design/4368705/The-future-of-computers--Part-1-Multicore-and-the-Memory-Wall

[88] ——, "Future of computing - part 3: The ilp wall and pipelines," EDN Network, February 2012. [Online]. Available: http://www.edn.com/design/systems-design/4368983/Future-of-computing--Part-3-The-ILP-Wall-and-pipelines

[89] W. Fox, "Computers and the internet," June 2011. [Online]. Available: http://www.futuretimeline.net/subject/computers-internet.htm

[90] S. A. Fulopa and K. Fitz, "Algorithms for computing the time-corrected instantaneous frequency (reassigned) spectrogram, with applications," *Journal of The Acoustical Society of America*, vol. 119, pp. 360–71, 2006.

[91] S. Fusi, "Spike-driven synaptic plasticity for learning correlated patterns of mean firing rates," *Reviews in the Neurosciences*, vol. 14, no. 1-2, pp. 73–84, January 2003.

[92] S. Fusi, M. Annunziato, D. Badoni, A. Salamon, and D. J. Amit, "Spike-driven synaptic plasticity: Theory, simulation, VLSI implementation," *Neural Computation*, vol. 12, no. 10, pp. 2227–2258, October 2000.

[93] I. Gambin, I. Grech, O. Casha, E. Gatt, and J. Micallef, "Digital cochlea model implementation using Xilinx XC3S500E Spartan-3E FPGA," in *Proceedings of the 17th IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, 2010, pp. 946–949.

[94] T. Gandhi and M. Trivedi, "Pedestrian protection systems: Issues, survey, and challenges," *IEEE Transactions on Intelligent Transportation Systems*, vol. 8, no. 3, pp. 413–430, September 2007.

[95] J. Ganssle, "Will Moore's Law doom multicore?" March 2013. [Online]. Available: http://www.embedded.com/electrical-engineer-community/industry-blog/4409679/1/Multicore-challenges

[96] G. Garreau, C. Andreou, A. Andreou, J. Georgiou, S. Dura-Bernal, T. Wennekers, and S. Denham, "Gait-based person and gender recognition using micro-Doppler signatures," in *IEEE Biomedical Circuits and Systems Conference (BioCAS)*, November 2011, pp. 444–447.

[97] G. Garreau, N. Nicolaou, C. Andreou, C. D'Urbal, G. Stuarts, and J. Georgiou, "Computationally efficient classification of human computationally efficient classification of human transport mode using micro-Doppler signatures," in *45th Annual Conference on Information Sciences and Systems (CISS)*, March 2011, pp. 1–4.

[98] G. Garreau, N. Nicolaou, and J. Georgiou, "Individual classification through autoregressive modelling of micro-Doppler signatures," in *IEEE Biomedical Circuits and Systems Conference (BioCAS)*, November 2012, pp. 312–315.

[99] P. Garrett, "descripwords," August 2013. [Online]. Available: http://www.msgarrettonline.com/Descriptive_Words.html

[100] C. Gearhart, A. Herold, B. Self, C. Birdsong, and L. Slivovsky, "Use of ultrasonic sensors in the development of an electronic travel aid," in *IEEE Sensors Applications Symposium (SAS)*, February 2009, pp. 275–280.

[101] J. Geisheimer, W. Marshall, and E. Greneker, "A continuous-wave (CW) radar for gait analysis," in *Proceedings of the Conference Record of the 35th Asilomar Conference on Signals, Systems and Computers*, vol. 1, November 2001, pp. 834–838.

[102] J. Georgiou and C. Toumazou, "A 126 μW cochlear chip for a totally implantable system," *IEEE Journal of Solid-State Circuits*, vol. 40, no. 2, pp. 430–443, February 2005.

[103] J. Georgiou, P. Pouliquen, A. Cassidy, G. Garreau, C. Andreou, G. Stuarts, C. d'Urbal, A. G. Andreou, S. Denham, T. Wennekers, R. Mill, I. Winkler, T. Bohm, O. Szalardy, G. M. Klump, S. Jones, and A. Bendixen, "A multimodal-corpus data collection system for cognitive acoustic scene analysis," in *45th Annual Conference on Information Sciences and Systems (CISS)*, March 2011, pp. 1–6.

[104] D. Giannoulis, E. Benetos, D. Stowell, M. Rossignol, M. Lagrange, and M. Plumbley, "Detection and classification of acoustic scenes and events," IEEE AASP Challenges, 2012.

[105] M. A. Giese and T. Poggio, "Neural mechanisms for the recognition of biological movements," *Nature Reviews Neuroscience*, vol. 4, no. 3, pp. 179–192, 2003.

[106] D. H. Goldberg, A. P. Sripati, and A. G. Andreou, "Energy efficiency in a channel model for the spiking axon," *Neurocomputing*, vol. 52-54, pp. 39–44, 2003.

[107] D. F. Goodman and R. Brette, "The Brian simulator," *Frontiers in Neuroscience*, vol. 3, no. 2, pp. 192–197, July 2009.

[108] D. F. M. Goodman and R. Brette, "Brian: a simulator for spiking neural networks in python," *Frontiers in Neuroinformatics*, vol. 2, pp. 1–10, November 2008.

[109] C. Goutte, "Lag space estimation in time series modelling," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 4, April 1997, pp. 3313–3316.

[110] L. Grossman, "2045: The year man becomes immortal," TIME Magazine, February 2011. [Online]. Available: http://content.time.com/time/magazine/article/0,9171,2048299-1,00.html

[111] G. Grubb, A. Zelinsky, L. Nilsson, and M. Rilbe, "3D vision sensing for improved pedestrian safety," in *Proceedings of the IEEE Intelligent Vehicles Symposium*, june 2004, pp. 19–24.

[112] A. Haigh, D. J. Brown, P. Meijer, and M. J. Proulx, "How well do you see what you hear? the acuity of visual-to-auditory sensory substitution," *Frontiers in Psychology*, vol. 4, no. 330, June 2013.

[113] D. Hall and J. Llinas, "An introduction to multisensor data fusion," *Proceedings of the IEEE*, vol. 85, no. 1, pp. 6–23, 1997.

[114] Q. Hardy, "Google buys a quantum computer," The New York Times, May 2013. [Online]. Available: http://bits.blogs.nytimes.com/2013/05/16/google-buys-a-quantum-computer/?_r=0

[115] S. Harput and A. Bozkurt, "Ultrasonic phased array device for acoustic imaging in air," *IEEE Sensors Journal*, vol. 8, no. 11, pp. 1755–1762, November 2008.

[116] B. Hatheway, "The doppler effect," June 2011. [Online]. Available: http://www.windows2universe.org/earth/Atmosphere/tornado/doppler_effect.html

[117] M. Hazas and A. Ward, "A novel broadband ultrasonic location system," in *Proceedings of the ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*, vol. 2498, September 2002, pp. 264–280.

[118] M. Henaff, K. Jarrett, K. Kavukcuoglu, and Y. LeCun, "Unsupervised learning of sparse features for scalable audio classification," in *Proceedings of International Symposium on Music Information Retrieval (ISMIR)*, A. Klapuri and C. Leider, Eds. University of Miami, 2011, pp. 681–686.

[119] M. Herman, C. Komanoff, J. Orcutt, and D. Perry, *The Electronic Bicycle Blueprint*, 2nd ed., C. Komanoff, Ed. Bicycling Magazine, January 1999.

[120] M. N. Hill, *The Sea, Volume 1: Physical Oceanography*, ser. The Sea: Ideas and Observations on Progress in the Study of the Seas. M. N. Hill, January 1962.

[121] S. A. Hiroshi Sawada, Ryo Mukai and S. Makino, "Estimating the number of sources using independent component analysis," in *Proceedings of the Acoustical Science and Technology*, vol. 26, no. 5, May 2005, pp. 450–452.

[122] J. Hoegg and J. W. Alba, "Taste perception: More than meets the tongue," *Journal of Consumer Research*, vol. 33, no. 4, pp. 490–498, March 2007.

[123] X. Hong, C. Nugent, M. Mulvenna, S. McClean, B. Scotney, and S. Devlin, "Evidential fusion of sensor data for activity recognition in smart homes," *Pervasive and Mobile Computing*, vol. 5, no. 3, pp. 236 – 252, 2009.

[124] W. Huang, M. R. Stant, K. Sankaranarayanan, R. J. Ribando, and K. Skadron, "Many-core design from a thermal perspective," in *Proceedings of the 45th Annual Design Automation Conference (DAC)*, June 2008, pp. 746–749.

[125] D. H. Hubel and T. N. Wiesel, "Receptive fields and functional architecture in two nonstriate visual areas (18 and 19) of the cat," *Journal of Neurophysiology*, vol. 28, pp. 229–289, 1965.

[126] B. Hughes, "Active artificial echolocation and the nonvisual perception of aperture passability," *Human Movement Science*, vol. 20, no. 4-5, pp. 371–400, 2001.

[127] R. Hughes and M. Boshier, "Quantum computation roadmap," Los Alamos National Security, LLC, April 2004. [Online]. Available: http://qist.lanl.gov/qcomp_map.shtml

[128] A. J. Ijspeert, "Central pattern generators for locomotion control in animals and robots: a review," *Neural Networks*, vol. 21, no. 4, pp. 642–653, March 2008.

[129] N. Instrument, *NI PXI-6682 Series User Manual*, National Instrument, March 2009.

[130] Intel, "Intel," Intel press release. [Online]. Available: http://www.intel.com/content/www/us/en/homepage.html

[131] A. Iqbal, U. Farooq, H. Mahmood, and M. Asad, "A low cost artificial vision system for visually impaired people," in *Proceedings of the 2nd International Conference on Computer and Electrical Engineering (ICCEE)*, vol. 2, December 2009, pp. 474–479.

[132] P. Julian, A. Andreou, L. Riddle, S. Shamma, D. Goldberg, and G. Cauwenberghs, "A comparative study of sound localization algorithms for energy aware sensor network nodes," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 51, no. 4, pp. 640–648, April 2004.

[133] K. Kalgaonkar and B. Raj, "Acoustic Doppler sonar for gait recognition," in *Proceedings of the IEEE Conference on Advanced Video and Signal Based Surveillance (AVSS)*, September 2007, pp. 27–32.

[134] ——, "Recognizing talking faces from acoustic Doppler reflections," in *Proceedings of the 8th IEEE International Conference on Automatic Face Gesture Recognition*, September 2008, pp. 1–6.

[135] ——, "One-handed gesture recognition using ultrasonic Doppler sonar," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, April 2009, pp. 1889–1892.

[136] E. R. Kandel, J. H. Schwartz, and T. M. Jessell, *Principles of Neural Science*, 4th ed. McGraw-Hill Medical, Jul. 2000.

[137] D. Kim, "Neural network mechanism for the orientation behavior of sand scorpions towards prey," *IEEE Transactions on Neural Networks*, vol. 17, no. 4, pp. 1070–1076, July 2006.

[138] ——, "Tactile information processing for the orientation behaviour of sand scorpions," in *Sensors: Focus on Tactile Force and Stress Sensors*, J. G. Rocha and S. Lanceros-Mendez, Eds. InTech, December 2008, ch. 24, pp. 431–444.

[139] Y. Kim and H. Ling, "Human activity classification based on micro-Doppler signatures using an artificial neural network," in *Proceedings of the IEEE Antennas and Propagation Society International Symposium*, July 2008, pp. 1–4.

[140] ——, "Human activity classification based on micro-Doppler signatures using a support vector machine," in *IEEE Transactions on Geoscience and Remote Sensing*, vol. 47, no. 5, May 2009, pp. 1328–1337.

[141] H. Kiragi and A. Ersak, "Object recognition and localization with ultrasonic scanning," in *Proceedings of the 7th Mediterranean Electrotechnical Conference*, vol. 3, April 1994, pp. 1185–1188.

[142] R. Krishnamurthy, "High-performance energy-efficient reconfigurable accelerators/co-processors for tera-scale multi-core microprocessors," in *Proceedings of the 6th international conference on Reconfigurable Computing: Architectures, Tools and Applications (ARC)*, ser. Lecture Notes in Computer Science, vol. 5992, 2010, p. 1.

[143] S. Kumar, K. E. Stephan, J. D. Warren, K. J. Friston, and T. D. Griffiths, "Hierarchical processing of auditory objects in humans," *PLoS Computational Biology*, vol. 3, no. 6, 2007.

[144] P. Leather, D. Beale, and L. Sullivan, "Noise, psychosocial stress and their interaction in the workplace," *Journal of Environmental Psychology*, vol. 23, pp. 213–222, 2003.

[145] Y. LeCun and Y. Bengio, "Convolutional networks for images, speech and time-series," in *The Handbook of Brain Theory and Neural Networks*, M. A. Arbib, Ed. MIT Press, 1995, pp. 255–258.

[146] Y. LeCun, K. Kavukcuoglu, and C. Farabet, "Convolutional networks and applications in vision," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2010, pp. 253–256.

[147] H. Lee, J. W. Park, and A. Helal, "Estimation of indoor physical activity level based on footstep vibration signal measured by MEMS accelerometer in smart home environments," in *Proceedings of the 2nd international conference on Mobile entity localization and tracking in GPS-less environments*, ser. MELT'09, June 2009, pp. 148–162.

[148] P. Lennie, "The cost of cortical computation," *Current Biology*, vol. 13, no. 6, pp. 493–497, March 2003.

[149] M. Leong, C. T. Jin, and P. H. Leong, "An FPGA-based electronic cochlea," *EURASIP Journal on Advances in Signal Processing*, vol. 2003, no. 7, pp. 629–638, 2003.

[150] J. Lewis, "Microphone specifications explained," Analog Devices, Application Note 1112, 2011.

[151] ——, "Microphone specifications explained," Analog Devices, Analog Dialogue 46, May 2012.

[152] T. L. H. Li, A. B. Chan, and A. H. W. Chun, "Automatic musical pattern feature extraction using convolutional neural network," in *Proceedings of the International MultiConference of Engineers and Computer Scientists*, S. I. Ao, O. Castillo, C. Douglas, D. D. Feng, and J.-A. Lee, Eds., vol. I. Newswood Limited, 2010, pp. 546–550.

[153] L. Liang, P. Mihail, R. Marylin, S. Marjorie, C. Paul, and Y. Tarik, "Automatic fall detection based on Doppler radar motion signature," in *5th International Conference on Pervasive Computing Technologies for Healthcare*, February 2011.

[154] S.-C. Liu and T. Delbruck, "Neuromorphic sensory systems," *Current Opinion in Neurobiology*, vol. 20, no. 3, pp. 288–295, June 2010.

[155] S.-C. Liu, A. Van Schaik, B. Minch, and T. Delbruck, "Event-based 64-channel binaural silicon cochlea with Q enhancement mechanisms," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, 2010, pp. 2027–2030.

[156] W. Liu, A. Andreou, and J. Goldstein, M.H., "Voiced-speech representation by an analog silicon model of the auditory periphery," *IEEE Transactions on Neural Networks*, vol. 3, no. 3, pp. 477–487, 1992.

[157] M. Lombardi, "Microsecond accuracy at multiple sites: is it possible without GPS?" *IEEE Instrumentation Measurement Magazine*, vol. 15, no. 5, pp. 14–21, October 2012.

[158] V. T. Ltd., "Tomi™ technology implementations," Venray Technology Ltd. press release, 2014. [Online]. Available: http://www.venraytechnology.com

[159] R. Lyon and C. Mead, "An analog electronic cochlea," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 36, no. 7, pp. 1119–1134, 1988.

[160] ——, "A CMOS VLSI cochlea," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 4, 1988, pp. 2172–2175.

[161] B. Lyonnet, C. Ioana, and M. Amin, "Human gait classification using micro-Doppler time-frequency signal representations," in *Proceedings of the IEEE Radar Conference*, November 2010, pp. 915–919.

[162] G. Ma, S.-B. Park, A. Ioffe, S. Müller-Schneiders, and A. Kummert, "A real-time detection algorithm for vision-based pedestrian protection." *International Journal of Information Acquisition*, vol. 5, no. 1, pp. 11 –30, March 2008.

[163] J. B. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, vol. 1. University of California Press, 1967, pp. 281–297.

[164] A. Manchula and S. Arumugam, "Robust facial data recognition using multi-modal fusion features in multi-variant face acquisition," *International Journal of Computer Applications*, vol. 64, no. 11, pp. 8–11, February 2013, published by Foundation of Computer Science, New York, USA.

[165] R. H. Mannell, "The brainstem auditory nuclei and centrifugal pathways," SPH307 Lectures Note - Auditory Physiology and Psychoacoustics, Macquerie University, Sidney, Australia, 2002. [Online]. Available: http://www.zainea.com/The_Brainstem_Auditory_Nuclei.htm

[166] J. Marpaung, Y. Zhang, and L. Johnson, "An implementation of an ultrasonic device for the visually impaired," in *Proceedings of the IEEE Region 5 Conference*, April 2006, pp. 287–290.

[167] M. Marwick and A. Andreou, "Retinomorphic system design in three dimensional SOI-CMOS," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, 2006, pp. 1655–1658.

[168] F. Masson, D. Puschini, P. Julian, P. Crocce, L. Arlenghi, P. Mandolesi, and A. Andreou, "Hybrid sensor network and fusion algorithm for sound source localization," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 3, May 2005, pp. 2763–2766.

[169] H. McGurk and J. MacDonald, "Hearing lips and seeing voices," *Nature*, vol. 264, pp. 746–748, December 1976.

[170] N. Mesgarani, S. V. David, J. B. Fritz, and S. A. Shamma, "Phoneme representation and classification in primary auditory cortex," *The Journal of the Acoustical Society of America*, vol. 123, no. 2, pp. 899–909, 2008.

[171] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon, "Top 500 supercomputers," Top500, November 2013. [Online]. Available: http://www.top500.org/lists/2013/11/

[172] Z. Miao, W. Ji, Y. Xu, and J. Yang, "A novel ultrasonic sensing based human face recognition," in *Proceedings of the IEEE Ultrasonics Symposium (IUS)*, November 2008, pp. 1873–1876.

[173] D. Mills, A. Thyagarajan, and B. Huffman, "Internet timekeeping around the globe," in *Proceedings of the Precision Time and Time Interval (PTTI) Applications and Planning Meeting*, December 1997, pp. 365–371.

[174] N. Minar, "A survey of the ntp network," MIT, December 1999. [Online]. Available: http://www.media.mit.edu/~nelson/research/ntp-survey99/html/

[175] J. Misra and I. Saha, "Artificial neural networks in hardware: A survey of two decades of progress," *Neurocomputing*, vol. 74, no. 1-3, pp. 239–255, December 2010.

[176] K. Moller, F. Toth, L. Wang, J. Moller, K. Arras, M. Bach, S. Schumann, and J. Guttmann, "Enhanced perception for visually impaired people," in *Proceedings of the 3rd International Conference on Bioinformatics and Biomedical Engineering (ICBBE)*, June 2009, pp. 1–4.

[177] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, April 1965.

[178] C. F. Moss and A. Surlykke, "Probing the natural scene by echolocation in bats," *Frontiers in Behavioral Neuroscience*, vol. 4, no. 33, 2010.

[179] C. Mugliette, I. Grech, O. Casha, E. Gatt, and J. Micallef, "FPGA active digital cochlea model," in *Proceedings of the 18th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2011, pp. 699–702.

[180] R. Murphy and S. N. Laboratories, "More chip cores can mean slower supercomputing, sandia simulation shows," Sandia Labs News Releases, January 2009. [Online]. Available: https://share.sandia.gov/news/resources/news_releases/more-chip-cores-can-mean-slower-supercomputing-sandia-simulation-shows/

[181] C. E. O'Connell-Rodwell, "Keeping an 'ear' to the ground: Seismic communication in elephants," *Physiology*, vol. 22, no. 4, pp. 287–294, August 2007.

[182] R. of Cyprus, "Radio frequency plan of the Republic of Cyprus," Ministry of Communications and Works, Departement of Electronic Communications, February 2013.

[183] P. Orponen, "Computational complexity of neural networks: a survey," *Nordic Journal of Computing*, vol. 1, no. 1, pp. 94–110, March 1994.

[184] M. Otero, "Application of a continuous wave radar for human gait recognition," in *Proceedings of Society of Photo-Optical Instrumentation Engineers (SPIE)*, vol. 5809, May 2005, pp. 538–548.

[185] K. Pastra and Y. Aloimonos, "The minimalist grammar of action," *Philosophical Transactions of the Royal Society B: Biological Sciences*, vol. 367, no. 1585, pp. 103–117, January 2012.

[186] D. Patterson, K. Asanovíc, and K. Keutzer, "Computer architecture is back - the berkeley view on the parallel computing landscape," ee380, Lecture, Stanford University, Stanford, California, January 2007. [Online]. Available: http://www.stanford.edu/class/ee380/Abstracts/070131-BerkeleyView1.7.pdf

[187] M. Peden, E. Krug, D. Mohan, A. Hyder, and R. Norton, *The world report on road traffic injury prevention*, M. Peden, R. Scurfield, D. Sleet, D. Mohan, A. A. Hyder, E. Jarawan, and C. Mathers, Eds. World Health Organization, 2004.

[188] A. J. Pegna, A.-S. Caldara-Schnetzer, S. H. Perrig, F. Lazeyras, A. Khateb, E. Mayer, T. Landis, and M. Seeck, "Is the right amygdala involved in visuospatial memory? evidence from MRI volumetric measures," *European Neurology*, vol. 47, no. 3, pp. 148–155, March 2002.

[189] A. J. Pegna, A. Khateb, F. Lazeyras, and M. L. Seghier, "Discriminating emotional faces without primary visual cortices involves the right amygdala," *Nature Neuroscience*, vol. 8, no. 1, pp. 24–25, January 2005.

[190] J. M. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital integrated circuits - A design perspective*, 2nd ed. Prentice Hall, 2004.

[191] S. S. Ram, Y. Li, A. Lin, and H. Ling, "Doppler-based detection and tracking of humans in indoor environments," *Journal of the Franklin Institute*, vol. 345, no. 6, pp. 679–699, April 2008.

[192] T. Ren, W. He, and E. Porsov, "Localization of the cochlear amplifier in living sensitive ears," *PLoS ONE*, vol. 6, no. 5, May 2011.

[193] D. A. Reynolds and R. C. Rose, "Robust text-independent speaker identification using gaussian mixture speaker models," *IEEE Transactions on Speech and Audio Processing*, vol. 3, no. 1, pp. 72–83, 1995.

[194] D. Robert, R. N. Miles, and R. R. Hoy, "Tympanal mechanics in the parasitoid fly ormia ochracea: intertympanal coupling during mechanical vibration," *Journal of Comparative Physiology A*, vol. 183, no. 4, pp. 443–452, 1998.

[195] D. Robert, R. Miles, and R. Hoy, "Tympanal hearing in the sarcophagid parasitoid fly emblemasoma sp.: the biomechanics of directional hearing," *Journal of Experimental Biology*, vol. 202, no. 14, pp. 1865–1876, 1999.

[196] D. Rotman, "Molecular computing," *MIT Technological Review Magazine*, pp. 1–6, May 2000. [Online]. Available: http://www.technologyreview.com/featuredstory/400728/molecular-computing/

[197] M. Rucci, G. Edelman, and J. Wray, "Adaptation of orienting behavior: from the barn owl to a robotic system," *IEEE Transactions on Robotics and Automation*, vol. 15, no. 1, pp. 96–110, February 1999.

[198] M. Rucci, G. Tononi, and G. M. Edelman, "Registration of neural maps through value-dependent learning: Modeling the alignment of auditory and visual maps in the barn owl's optic tectum," *Journal of Neuroscience*, vol. 17, pp. 334–352, 1997.

[199] M. Ryoo and J. Aggarwal, "Recognition of composite human activities through context-free grammar based representation," in *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2, October 2006, pp. 1709–1718.

[200] A. M. Sabatini and V. Colla, "A method for sonar based recognition of walking people," *Robotics and Autonomous Systems*, vol. 25, no. 1-2, pp. 117–126, 1998.

[201] E. Salinas, "Noisy neurons can certainly compute," *Nature Neuroscience : News and Views*, vol. 9, no. 11, pp. 1349–1350, November 2006.

[202] F. Savorani, G. Tomasi, and S. B. Engelsen, "icoshift: A versatile tool for the rapid alignment of 1D NMR spectra." *Journal of Magnetic Resonance*, vol. 202, no. 2, pp. 190–202, 2010.

[203] H. Sawada, R. Mukai, S. Araki, and S. Malcino, "Multiple source localization using independent component analysis," in *Proceedings of the IEEE Antennas and Propagation Society International Symposium*, vol. 4B, 2005, pp. 81–84.

[204] H. Sawada, R. Mukai, and S. Makino, "Direction of arrival estimation for multiple source signals using independent component analysis," in *Proceedings of the 7th International Symposium on Signal Processing and Its Applications*, vol. 2, 2003, pp. 411–414.

[205] S. Schraml, A. Belbachir, N. Milosevic, and P. Schö andn, "Dynamic stereo vision system for real-time tracking," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)*, june 2010, pp. 1409–1412.

[206] C. Shah, M. Bouzit, M. Youssef, and L. Vasquez, "Evaluation of RU-Netra - tactile feedback navigation system for the visually impaired," in *Proceedings of the International Workshop on Virtual Rehabilitation*, July 2006, pp. 72–77.

[207] S. Sheik, M. Coath, G. Indiveri, S. L. Denham, T. Wennekers, and E. Chicca, "Emergent auditory feature tuning in a real-time neuromorphic VLSI system," *Frontiers in Neuroscience*, vol. 6, no. 17, February 2012.

[208] L. Shestopalova, T. M. Böhm, A. Bendixen, A. G. Andreou, J. Georgiou, G. Garreau, B. Hajdu, S. L. Denham, and I. Winkler, "Do audio-visual motion cues promote segregation of auditory streams?" *Frontiers in Neuroscience, Auditory Cognitive Neuroscience*, 2014.

[209] B. G. Shinn-Cunningham, "Object-based auditory and visual attention," *Trends in Cognitive Sciences*, vol. 12, no. 5, pp. 182–186, April 2008.

[210] H. Z. Shouval, M. F. Bear, and L. N. Cooper, "A unified model of NMDA receptor-dependent bidirectional synaptic plasticity," *Proceedings of the National Academy of Sciences*, vol. 99, no. 16, pp. 10 831–10 836, June 2002.

[211] M. M. Shulaker, G. Hills, N. Patil, H. Wei, H.-Y. Chen, H.-S. P. Wong, and S. Mitra, "Carbon nanotube computer," *Nature*, vol. 501, pp. 526–530, September 2013.

[212] R. A. Silver, "Neuronal arithmetic," *Nature Reviews Neuroscience*, vol. 11, pp. 474–489, July 2010.

[213] D. Simon, "On the power of quantum computation," in *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, 1994, pp. 116–123.

[214] A. Širović, J. A. Hildebrand, and S. M. Wiggins, "Blue and fin whale call source levels and propagation range in the southern ocean," *Journal of the Acoustical Society of America*, vol. 122, no. 2, pp. 1208–1215, May 2007.

[215] T. Söderström and P. Stoica, *System Identification*. Prentice Hall International, 1989.

[216] M. Stanacevic and G. Cauwenberghs, "Gradient flow broadband beamforming and source separation," in *Proceedings of the 3rd International Conference on Independent Component Analysis and Signal Separation (ICA)*, December 2001, pp. 49–52.

[217] M. Stanacevic, G. Cauwenberghs, and G. Zweig, "Gradient flow adaptive beamforming and signal separation in a miniature microphone array," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 4, May 2002, pp. 4016–4019.

[218] W. Stürzl, R. Kempter, and J. L. van Hemmen, "Theory of arachnid prey localization," *Physical Review Letters*, vol. 84, no. 24, pp. 5668–5671, June 2000.

[219] K. Sudo, J. Yamato, A. Tomono, and K.-i. Ishii, "Gender recognition method based on silhouette, footstep, and foot pressure measurements for counting customers," *Electronics and Communications in Japan (Part II: Electronics)*, vol. 85, no. 8, pp. 54–64, 2002.

[220] N. Suga, "Principles of auditory information-processing derived from neuroethology," *Journal of Experimental Biologie*, no. 146, pp. 277–86, 1989.

[221] S. Sukittanon, A. C. Surendran, J. C. Platt, and C. J. C. Burges, "Convolutional networks for speech detection," in *Proceedings of 8th International Conference on Spoken Language Processing (Interspeech'04, ICSLP)*, 2004, pp. 433–445.

[222] H. Sutter, "The free lunch is over : A fundamental turn toward concurrency in software," August 2009. [Online]. Available: http://www.gotw.ca/publications/concurrency-ddj.htm

[223] T. Teixeira, G. Dublon, and A. Savvides, "A survey of human-sensing: Methods for detecting presence, count, location, track, and identity," *Association for Computing Machinery (ACM) Computing Surveys*, vol. 5, pp. 1–35, 2010.

[224] L. Thaler, S. R. Arnott, and M. A. Goodale, "Neural correlates of natural human echolocation in early and late blind echolocation experts," *PLoS ONE*, vol. 6, no. 5, May 2011.

[225] T. Theocharides, "Embedded and real-time systems," ece653, Lecture, University of Cyprus, Nicosia, Cyprus, January 2010.

[226] G. Tomasi, F. Savorani, and S. B. Engelsen, "icoshift: An effective tool for the alignment of chromatographic data," *Journal of Chromatography A*, vol. 1218, no. 43, pp. 7832–7840, August 2011.

[227] E. Toolbox, "Speed of sound in some common solids," Engineering Toolbox, August 2013. [Online]. Available: http://www.engineeringtoolbox.com/sound-speed-solids-d_713.html

[228] P. W. J. Van Hengel and T. Andringa, "Verbal aggression detection in complex social environments," in *Proceedings of the IEEE Conference on Advanced Video and Signal Based Surveillance (AVSS)*, 2007, pp. 15–20.

[229] A. Van Schaik and E. Fragniere, "Pseudo-voltage domain implementation of a 2-dimensional silicon cochlea," in *Proceedings of the IEEE International Symposium onCircuits and Systems (ISCAS)*, vol. 3, no. 2, 2001, pp. 185–188.

[230] A. Van Schaik and S.-C. Liu, "AER EAR: a matched silicon cochlea pair with address event representation interface," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 5, 2005, pp. 4213–4216.

[231] A. van Schaik and S. Shamma, "A neuromorphic sound localizer for a smart MEMS system," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 4, 2003, pp. 864–867.

[232] A. Van Schaik, E. Fragnière, E. Vittoz *et al.*, "Improved silicon cochlea using compatible lateral bipolar transistors," *Advances in Neural Information Processing Systems (NIPS)*, vol. 8, pp. 671–677, May 1996.

[233] A. Van Schaik and S. Shamma, "A neuromorphic sound localizer for a smart MEMS system," *Analog Integrated Circuits Signal Processing*, vol. 39, no. 3, pp. 267–273, June 2004.

[234] C. Vu, "Made in IBM labs: Researchers demonstrate initial steps toward commercial fabrication of carbon nanotubes as a successor to silicon," IBM press release, October 2012. [Online]. Available: http://www-03.ibm.com/press/us/en/pressrelease/39250.wss

[235] J. Wagner, E. Andre, F. Lingenfelser, and J. Kim, "Exploring fusion methods for multimodal emotion recognition with missing data," *IEEE Transactions on Affective Computing*, vol. 2, no. 4, pp. 206–218, 2011.

[236] A. Wallander, R. Russell, and K. Hyyppa, "A robot scorpion using ground vibrations for navigation," in *Proceedings of the Australian Conference on Robotics and Automation*, vol. 2, August 2000, pp. 75–79.

[237] C. Wallraven, M. Schultze, B. Mohler, A. Vatakis, and K. Pastra, "The POETICON enacted scenario corpus a tool for human and computational experiments on action understanding," in *IEEE International Conference on Automatic Face Gesture Recognition and Workshops*, March 2011, pp. 484–491.

[238] D. Wang, "Visual scene segmentation," in *The Handbook of Brain Theory and Neural Networks*, 2nd ed. MIT Press, January 2003, pp. 1215–1219.

[239] ——, "Computational scene analysis," in *Challenges for Computational Intelligence*, ser. Studies in Computational Intelligence, W. Duch and J. Mandziuk, Eds. Springer, 2007, vol. 63, pp. 163–191.

[240] D. Wang and G. J. Brown, *Computational Auditory Scene Analysis: Principles, Algorithms, and Applications*. Wiley-IEEE Press, 2006.

[241] W. A. Watkins, P. Tyack, K. E. Moore, and J. E. Bird, "The 20hz signals of finback whales (Balaenoptera Physalus)," *Journal of the Acoustical Society of America*, vol. 82, no. 6, pp. 1901–1912, September 1987.

[242] A. Wegener, "Multicore, the memory wall, and numerical compression," in *IEEE Computer Society of Silicon Valley*, April 2012.

[243] N. T. N. Wi, C. K. Loo, and L. Chockalingam, "Biologically inspired face recognition: Toward pose-invariance," *International Journal of Neural Systems*, vol. 22, no. 06, 2012.

[244] Wikipedia, "Auditory system," March 2014. [Online]. Available: http://en.wikipedia.org/wiki/Auditory_system

[245] ——, "List of animal sounds," March 2014. [Online]. Available: http://en.wikipedia.org/wiki/Listofanimalsounds

[246] D. E. Wilson, *The Smithsonian Book of North American Mammals*, smithsonian institution press ed., D. E. Wilson and S. Ruff, Eds. Smithsonian Books, October 1999.

[247] T.-F. Wu, C.-J. Lin, and R. C. Weng, "Probability estimates for multi-class classification by pairwise coupling," *Journal of Machine Learning Research*, vol. 5, pp. 975–1005, December 2004.

[248] Y. Xu, X. Cao, and T. Li, "Extended kalman filter based pedestrian localization for collision avoidance," in *International Conference on Mechatronics and Automation (ICMA)*, aug. 2009, pp. 4366–4370.

[249] J. Yang, Y. Hu, and G. Li, "Target recognition and tracking based on data fusion and data mining," in *Proceedings of the International Society for Optical Engineering (SPIE)*, vol. 4556, October 2001, pp. 7–14.

[250] T. Yardibi, P. Cuddihy, S. Genc, C. Bufi, M. Skubic, M. Rantz, L. Liu, and C. Phillips, "Gait characterization via pulse-Doppler radar," in *IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, March 2011, pp. 662–667.

[251] S.-Y. Yeh, C.-I. Wu, K.-H. Chang, H.-H. Chu, and J. Y.-J. Hsu, "GETA sandals: A footstep location tracking system," *Association for Computing Machinery (ACM)/Springer Journal of Personal and Ubiquitous Computing (PUC): special issue on location and context awareness*, vol. 11, no. 6, pp. 451–463, August 2007.

[252] M. Yguel, O. Aycard, D. Raulo, and C. Laugier, "Grid based fusion of off-board cameras," in *Proceedings of the IEEE Intelligent Vehicles Symposium*, june 2006, pp. 276–281.

[253] L. Yim, "Microsoft admits they lost the console war with xbox one," Press release, May 2013. [Online]. Available: http://semiaccurate.com/2013/05/22/microsoft-subtly-admits-losing-with-xbox-one/

[254] Z. Zhang and A. Andreou, "Close range bearing estimation and tracking of slow moving vehicles using the microphone arrays in the Hopkins acoustic surveillance unit," in *Proceedings of the Argentine School of Micro-Nanoelectronics, Technology and Applications (EAMTA)*, September 2008, pp. 140–143.

[255] ——, "Human identification experiments using acoustic micro-Doppler signatures," in *Proceedings of the Argentine School of Micro-Nanoelectronics, Technology and Applications (EAMTA)*, September 2008, pp. 81–86.

[256] Z. Zhang, P. Pouliquen, A. Waxman, and A. Andreou, "Acoustic micro-Doppler gait signatures of humans and animals," in *41st Annual Conference on Information Sciences and Systems (CISS)*, March 2007, pp. 627–630.

[257] Z. Zhang, P. O. Pouliquen, A. Waxman, and A. Andreou, "Acoustic micro-Doppler radar for human gait imaging," *Journal of the Acoustical Society of America Express Letters*, vol. 121, no. 3, pp. 110–113, March 2007.

Appendix A

VHDL and UCF Scripts

In this appendix are given the VHDL and UCF sources used to program the FM SYNC and DACQ units.

The first VHDL script given is used to program the FM SYNC unit. After declaration and initialization of all signals and variables, the first component instantiated is the DCM to create the necessary clocks from the main crystal frequency. There is a counter to get a 1μs period for the main processes and also a 20μs period to control the phase shift encoding the raw bits of the timestamps. Then, there is a 9.6ms counter that sets the beginning of new timestamp generation and a process that increments of the timestamp value and controls when it reach the end and has to roll-over. Then, there is the actual process controlling the phase-shift used to encode the timestamp value. Finally, several processes to control the communication between the FPGA and the computer.

```
----------------------------------------------------------------------
-- Design Name: fm_sync
-- Module Name: fm_sync
-- File Name: fm_sync.vhd
--
-- Last updated: 2012-11-23
----------------------------------------------------------------------
-- ISE Project Settings:
--
-- Family:  Spartan3A and Spartan3AN
-- Device:  XC3S50AN
-- Package: TQG144
-- Speed:   -5
----------------------------------------------------------------------
--
-- FM syncronization transmitter
--
----------------------------------------------------------------------
--
-- Carrier frequency and phase modulation information:
--
-- The transmitter uses the US radio control airplane band of 72 to 73
-- MHz, because this is the largest band available for civilian use.
-- However, the FM modulation is much greater than that used for R/C
-- models, so the carrier is chosen to be in the middle at 72.5 MHz.
--
-- To generate the carrier, a clock multiplier module (DCM) is used to
-- generate a 4x system clock (290 MHz) from an external high
-- precision oscillator.  Given the oscillator availability, a 20 MHz
-- external oscillator was selected.
--
-- The 4x system clock was chosen so that the 72.5 MHz carrier can be
-- easily produced (the output is high for two cycles high and low for
-- two clock cyles), and the phase can be easily altered by stretching
-- the period to 5 clock cycles or shortening the period to 3 clock
-- cycles.
```

```
--
-- Shortening the period to 3 clock cycles is equivalent to advancing
-- the signal by 90 degrees.  Matlab simulations indicate that this
-- causes the main spur in the DFT to shift to 72.525MHz and a second
-- spur to appear at 72.425MHz.
--
-- Lengthening the period to 5 clock cycles is equivalent to retarding
-- the signal by 90 degrees.  Matlab simulations indicate that this
-- causes the main spur in the DFT to shift to 72.475MHz and a second
-- spur to appear at 72.575MHz.
--
-- The receiver topology requires a local oscillator of a frequency
-- 10.7 MHz lower than the carrier, or about 61.8 MHz.  The closest
-- that can be achieved with available oscillators is 61.765 MHz
-- (using a 75 MHz oscillator and a 14/17 ratio clock multiplier).
-- Since this is slightly less than 61.8 MHz, the carrier should be
-- modulated by lengthening the period (or retarding the phase).
--
----------------------------------------------------------------------
--
-- Serial port information:
--
-- Commands are composed of single bytes.  The upper nibble indicates
-- the command, and the lower nibble is the data payload if needed.
-- Valid commands are:
--
-- 0000: set bits 3-0 of timestamp
-- 0001: set bits 7-4 of timestamp
-- 0010: set bits 11-8 of timestamp
-- 0011: set bits 15-12 of timestamp
-- 0100: set bits 19-16 of timestamp
-- 0101: set bits 23-20 of timestamp
-- 0110: set bits 27-24 of timestamp
-- 0111: set bits 31-28 of timestamp
-- 1000: set bits 35-32 of timestamp
-- 1001: set bits 39-36 of timestamp
-- 1010: set bits 43-40 of timestamp
-- 1011: set bits 47-44 of timestamp
-- 1100: set bits 51-48 of timestamp
-- 1101: set bits 55-52 of timestamp
-- 1110: set bits 59-56 of timestamp
-- 1111: return timestamp
--
----------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
library UNISIM;-- For Xilinx primitives
use UNISIM.VComponents.all;

entity fm_sync is
        port (
                -- FT232R serial port signals
                txd:    in  std_logic;--
                rxd:    out std_logic;--
                rts:    in  std_logic;--
                cts:    out std_logic;--
                dtr:    in  std_logic;--
                dsr:    out std_logic;--
                dcd:    out std_logic;--
                ri:     out std_logic;--

                -- FT232R auxilliary port signals
                cbus0:  in  std_logic;--
                cbus1:  in  std_logic;--
                cbus2:  in  std_logic;--
                cbus3:  in  std_logic;--

                -- FM synchronizer debug pins
                dbg0:   out std_logic;-- Upper debug pin
                dbg1:   out std_logic;-- Middle debug pin
                dbg2:   out std_logic;-- Lower debug pin

                -- Modulated carrier output
                fm_out: out std_logic;-- FM output
```

```
                -- External oscillator input
                xtal:   in  std_logic -- 20MHz oscillator
        );
end fm_sync;

architecture arch of fm_sync is
        ---------------------------------------------------------------
        -- Local signals for DCM
        ---------------------------------------------------------------
        signal   gclk:     std_logic;-- Intermediate clock signal
        signal   gclk0:    std_logic;-- DCM 0 degree output clock
        signal   gclk_fb:  std_logic;-- DCM feedback clock
        signal   clkfx:    std_logic;-- DCM synthesized output clock
        signal   clk:      std_logic;-- System clock
        ---------------------------------------------------------------
        -- Local signals for microsecond counter
        ---------------------------------------------------------------
        signal   c290_c:   std_logic_vector(8 downto 0);-- LFSR cntr
        constant c290_e:   std_logic_vector(8 downto 0):=
                b"1_1010_0000";--TC
        signal   c290_t:   std_logic_vector(3 downto 0);-- Decode
        signal   tc:       std_logic;-- TC detect
        ---------------------------------------------------------------
        -- Local signals for bit counter
        ---------------------------------------------------------------
        signal   c20_c:    std_logic_vector(4 downto 0);-- LFSR cntr
        signal   c20_pr:   std_logic;-- Primary shift
        signal   c20_se:   std_logic;-- Secondary shift
        signal   c20_cl:   std_logic;-- Debug clear
        signal   c20_tc:   std_logic;-- TC detect
        signal   c20_msb:  std_logic_vector(3 downto 0);-- MSB decode
        signal   c20_lsb:  std_logic_vector(3 downto 0);-- LSB decode
        ---------------------------------------------------------------
        -- Local signals for frame counter
        ---------------------------------------------------------------
        signal   c480_c:   std_logic_vector(8 downto 0);-- LFSR cntr
        signal   c480_t:   std_logic_vector(3 downto 0);-- TC detect
        ---------------------------------------------------------------
        -- Local signals for timestamp
        ---------------------------------------------------------------
        signal   time_s:   std_logic_vector(59 downto 0);-- 60 bits
        signal   time_c:   std_logic_vector(59 downto 1);
        signal   time_0:   std_logic_vector(59 downto 1);
        signal   time_1:   std_logic_vector(59 downto 1);
        signal   time_pr0: std_logic;-- Override en_pr normal val of 0
        signal   time_pr1: std_logic;-- Override en_pr normal val of 1
        signal   time_se0: std_logic;-- Override en_se normal val of 0
        signal   time_se1: std_logic;-- Override en_se normal val of 1
        signal   time_t:   std_logic;-- Update precursor
        signal   time_u:   std_logic;-- Timestamp update
        signal   time_i:   std_logic;-- Timestamp increment amount
        signal   wr_ack:   std_logic;-- Handshaking
        ---------------------------------------------------------------
        -- Local signals for shift control
        ---------------------------------------------------------------
        signal   data:     std_logic_vector(59 downto 0);-- 60 bits
        signal   en_pr:    std_logic;
        signal   en_se:    std_logic;
        signal   pr:       std_logic;
        signal   se:       std_logic;
        ---------------------------------------------------------------
        -- Local signals for FM modulator
        ---------------------------------------------------------------
        signal   fm:       unsigned(1 downto 0);-- 2 bits
        ---------------------------------------------------------------
        -- Local signals for serial port counters (3.020833 MegaBAUD)
        ---------------------------------------------------------------
        signal   c12_c:    std_logic_vector(3 downto 0);-- FSM cntr
        signal   c12_t:    std_logic;-- Every 12 cycles (approx 24MHz)
        signal   c96_c:    std_logic_vector(2 downto 0);-- FSM cntr
        signal   c96_t:    std_logic;-- Every 96 cycles (approx 3MHz)
        signal   c24_t:    std_logic;-- Every 24 cycles (approx 12MHz)
        ---------------------------------------------------------------
        -- Local signals for serial port input shift register
        ---------------------------------------------------------------
        signal   ser_cmd:  std_logic_vector(8 downto 0);
        signal   ser_in:   std_logic_vector(36 downto 0);
```

```
        ------------------------------------------------------------
        -- Local signals for command decoding
        ------------------------------------------------------------
        signal   ser_dec:  std_logic_vector(15 downto 0);-- 16 cmds
        ------------------------------------------------------------
        -- Local signals for command processing
        ------------------------------------------------------------
        signal   wr_req_p: std_logic;-- Handshake precursor
        signal   wr_req:   std_logic;-- Handshaking
        signal   rd_req_p: std_logic;-- Handshake precursor
        signal   rd_req:   std_logic;-- Handshaking
        signal   new_ts:   std_logic_vector(59 downto 0);-- 60 bits
        ------------------------------------------------------------
        -- Local signals for serial port output shift register
        ------------------------------------------------------------
        signal   rd_ack:   std_logic;-- Handshaking
        signal   ser_out:  std_logic_vector(79 downto 0);
        ------------------------------------------------------------
        -- Initial registered internal signal values
        ------------------------------------------------------------
        attribute INIT: string;
        attribute INIT of c290_c:   signal is b"0_0000_0000";
        attribute INIT of c290_t:   signal is b"0000";
        attribute INIT of tc:       signal is b"0";
        attribute INIT of c20_c:    signal is b"00000";
        attribute INIT of c20_pr:   signal is b"0";
        attribute INIT of c20_se:   signal is b"0";
        attribute INIT of c20_cl:   signal is b"0";
        attribute INIT of c20_tc:   signal is b"0";
        attribute INIT of c20_msb:  signal is b"0000";
        attribute INIT of c20_lsb:  signal is b"0000";
        attribute INIT of c480_c:   signal is b"0_0000_0000";
        attribute INIT of c480_t:   signal is b"0000";
        attribute INIT of time_s:   signal is b"0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0001";
        attribute INIT of time_c:   signal is b"0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_000";
        attribute INIT of time_0:   signal is b"1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_111";
        attribute INIT of time_1:   signal is b"0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_000";
        attribute INIT of time_pr0: signal is b"0";
        attribute INIT of time_pr1: signal is b"1";
        attribute INIT of time_se0: signal is b"0";
        attribute INIT of time_se1: signal is b"1";
        attribute INIT of time_t:   signal is b"0";
        attribute INIT of time_u:   signal is b"0";
        attribute INIT of time_i:   signal is b"1";
        attribute INIT of wr_ack:   signal is b"0";
        attribute INIT of data:     signal is b"0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000";
        attribute INIT of en_pr:    signal is b"0";
        attribute INIT of en_se:    signal is b"0";
        attribute INIT of pr:       signal is b"0";
        attribute INIT of se:       signal is b"0";
        attribute INIT of fm:       signal is b"00";
        attribute INIT of c12_c:    signal is b"0000";
        attribute INIT of c12_t:    signal is b"0";
        attribute INIT of c96_c:    signal is b"000";
        attribute INIT of c96_t:    signal is b"0";
        attribute INIT of c24_t:    signal is b"0";
        attribute INIT of ser_cmd:  signal is b"0_0000_0000";
        attribute INIT of ser_in:   signal is b"1111_1111_1111_1111_1111_1111_1111_1111_1111_1";
        attribute INIT of ser_dec:  signal is b"0000000000000000";
        attribute INIT of wr_req_p: signal is b"0";
        attribute INIT of wr_req:   signal is b"0";
        attribute INIT of rd_req_p: signal is b"0";
        attribute INIT of rd_req:   signal is b"0";
        attribute INIT of new_ts:   signal is b"0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0001";
        attribute INIT of rd_ack:   signal is b"0";
        attribute INIT of ser_out:  signal is b"1111111111_1111111111_1111111111_1111111111_1111111111_1111111111_1111
        ------------------------------------------------------------
        -- Initial registered external signal values
        ------------------------------------------------------------
        attribute INIT of rxd:      signal is b"1";
        attribute INIT of dbg0:     signal is b"0";
        attribute INIT of dbg1:     signal is b"0";
        attribute INIT of dbg2:     signal is b"0";
begin
        ------------------------------------------------------------
        -- DCM:
        --
```

```
-- DCM related signal path:
-- Clock input:         xtal -> IBUFG -> gclk -> DCM
-- Feedback:            DCM -> gclk0 -> BUFG -> gclk_fb
-- FM local oscillator: DCM -> clkfx -> BUFG -> clk
-------------------------------------------------------------
u_ibufg: IBUFG
        generic map (
                IOSTANDARD=>"LVCMOS33"
        )
        port map (
                O=>gclk,-- Clock buffer output
                I=>xtal-- Clock buffer input
        );
u_dcm: DCM_SP
        generic map (
                CLKDV_DIVIDE=>2.0,-- CLKDV output divider
                CLKFX_DIVIDE=>2,-- CLKFX output divider
                CLKFX_MULTIPLY=>29,-- CLKFX output multiplier
                CLKIN_DIVIDE_BY_2=>false,
                CLKIN_PERIOD=>50.0,-- Input clock period in nanoseconds
                CLKOUT_PHASE_SHIFT=>"NONE",
                CLK_FEEDBACK=>"1X",-- Use CLK0 as feedback (not CLK2X)
                DESKEW_ADJUST=>"SYSTEM_SYNCHRONOUS",
                DFS_FREQUENCY_MODE=>"LOW",
                DLL_FREQUENCY_MODE=>"LOW",
                DSS_MODE=>"NONE",
                DUTY_CYCLE_CORRECTION=>true,-- Correct CLK0 to CLK270 duty cycle
                FACTORY_JF=>X"c080",
                PHASE_SHIFT=>0,
                STARTUP_WAIT=>false
        )
        port map (
                CLK0=>gclk0,
                CLK180=>open,
                CLK270=>open,
                CLK2X=>open,
                CLK2X180=>open,
                CLK90=>open,
                CLKDV=>open,
                CLKFX=>clkfx,
                CLKFX180=>open,
                LOCKED=>open,
                PSDONE=>open,
                STATUS=>open,
                CLKFB=>gclk_fb,
                CLKIN=>gclk,
                DSSEN=>'0',
                PSCLK=>'0',
                PSEN=>'0',
                PSINCDEC=>'0',
                RST=>'0'
        );
u_fb: BUFG
        port map (
                O=>gclk_fb,
                I=>gclk0
        );
u_bufg: BUFG
        port map (
                O=>clk,
                I=>clkfx
        );

-------------------------------------------------------------
-- Divide by 290 microsecond counter
--
-- This counter produces a pulse every microsecond.  The taps
-- are at bits 8 and 4, and the sequence looks like:
--
-- cycle   0 0_0000_0000
-- cycle   1 0_0000_0001
-- cycle   2 0_0000_0011
-- cycle   3 0_0000_0111
-- ...
-- cycle 286 1_1010_0000  **
-- cycle 287 1_0100_0000  c290_t(2 downto 0)=111
-- cycle 288 0_1000_0000  c290_t(3)=1
```

```
-- cycle 289 1_0000_0001  tc=1
--
-- Because the terminal count detection is pipelined but the
-- LFSR is updated every clock cycles, the terminal count
-- needs to be detected at least 2 cycles earlier.  This has
-- been increased to 3 cycles to give the synthesizer more
-- opportunity to buffer the termincal count.
-----------------------------------------------------------------
process(clk)
begin
        if rising_edge(clk) then
                if (tc='1') then
                        c290_c<=b"0_0000_0000";
                else
                        c290_c<=c290_c(7 downto 0)&(c290_c(8)
                                xnor c290_c(4));
                end if;

                tc<=c290_t(3);
                if (c290_t(2 downto 0)="111") then
                        c290_t(3)<='1';
                else
                        c290_t(3)<='0';
                end if;

                if (c290_c(8 downto 6)=c290_e(8 downto 6))
                        then
                        c290_t(2)<='1';
                else
                        c290_t(2)<='0';
                end if;
                if (c290_c(5 downto 3)=c290_e(5 downto 3))
                        then
                        c290_t(1)<='1';
                else
                        c290_t(1)<='0';
                end if;
                if (c290_c(2 downto 0)=c290_e(2 downto 0))
                        then
                        c290_t(0)<='1';
                else
                        c290_t(0)<='0';
                end if;
        end if;
end process;

-----------------------------------------------------------------
-- Divide by 20 bit counter
--
-- This counter counts microseconds and produces several
-- pulses for controlling when the phase shifts should occur.
-- The taps are at bits 4 and 2, and the sequence looks like:
--
-- microsecond  0  00000 Primary phase shift 1
-- microsecond  1  00001 Primary phase shift 2
-- microsecond  2  00011
-- microsecond  3  00111
-- microsecond  4  01110
-- microsecond  5  11100 Debug clear 1
-- microsecond  6  11001
-- microsecond  7  10010
-- microsecond  8  00100
-- microsecond  9  01000
-- microsecond 10  10001 Secondary phase shift 1
-- microsecond 11  00010 Secondary phase shift 2
-- microsecond 12  00101
-- microsecond 13  01010
-- microsecond 14  10101
-- microsecond 15  01011 Debug clear 2
-- microsecond 16  10111
-- microsecond 17  01111
-- microsecond 18  11110
-- microsecond 19  11101 Terminal count
--
-- Bit patterns to check:
--     MSB: 0 000 Primary phase shift 1,
--                Primary phase shift 2
```

```
--                Secondary phase shift 2
--          1 010 Debug clear 2
--          2 100 Secondary phase shift 1
--          3 111 Debug clear 1
--                Terminal count
--     LSB: 0  00 Primary phase shift 1
--                Debug clear 1
--          1  01 Primary phase shift 2
--                Secondary phase shift 1
--                Terminal count
--          2  10 Secondary phase shift 2
--          3  11 Debug clear 2
--
-- Because the terminal count detection is pipelined but the
-- LFSR is not updated every clock cycle, the terminal count
-- does not need to be detected any cycles earlier.
--
-- c20_c is updated on cycle 0.
-- c20_msb and c20_lsb are updated on cycle 1.
-- c20_pr, c20_se, c20_cl and c20_tc are updated on cycle 2.
-----------------------------------------------------------------
process(clk)
begin
        if rising_edge(clk) then
                if (tc='1') then
                        if (c20_tc='1') then
                                c20_c<=b"00000";
                        else
                                c20_c<=c20_c(3 downto 0)&
                                        (c20_c(4) xnor
                                        c20_c(2));
                        end if;
                end if;

                -- Primary phase shift
                if ((c20_msb(0)='1') and
                        (c20_lsb(0)='1')) or
                        ((c20_msb(0)='1') and
                        (c20_lsb(1)='1')) then
                        c20_pr<='1';
                else
                        c20_pr<='0';
                end if;

                -- Secondary phase shift
                if ((c20_msb(2)='1') and
                        (c20_lsb(1)='1')) or
                        ((c20_msb(0)='1') and
                        (c20_lsb(2)='1')) then
                        c20_se<='1';
                else
                        c20_se<='0';
                end if;

                -- Debug clear
                if ((c20_msb(3)='1') and
                        (c20_lsb(0)='1')) or
                        ((c20_msb(1)='1') and
                        (c20_lsb(3)='1')) then
                        c20_cl<='1';
                else
                        c20_cl<='0';
                end if;

                -- Terminal count
                if (c20_msb(3)='1') and (c20_lsb(1)='1') then
                        c20_tc<='1';
                else
                        c20_tc<='0';
                end if;

                -- MSB decode
                case c20_c(4 downto 2) is
                        when b"000" =>
                                c20_msb<=b"0001";
                        when b"010" =>
                                c20_msb<=b"0010";
```

```
                        when b"100" =>
                                c20_msb<=b"0100";
                        when b"111" =>
                                c20_msb<=b"1000";
                        when others =>
                                c20_msb<=b"0000";
                end case;

                -- LSB decode
                case c20_c(1 downto 0) is
                        when b"00" =>
                                c20_lsb<=b"0001";
                        when b"01" =>
                                c20_lsb<=b"0010";
                        when b"10" =>
                                c20_lsb<=b"0100";
                        when others =>
                                c20_lsb<=b"1000";
                end case;
        end if;
end process;

-------------------------------------------------------------
-- Divide by 480 frame counter
--
-- This counter counts the data bits in a frame.  The taps are
-- at bits 8 and 4, and the sequence looks like:
--
-- data bit   0 0_0000_0000
-- data bit   1 0_0000_0001
-- data bit   2 0_0000_0011
-- data bit   3 0_0000_0111
-- ...
-- data bit 476 1_0100_0010
-- data bit 477 0_1000_0100
-- data bit 478 1_0000_1001
-- data bit 479 0_0001_0010
--
-- Because the terminal count detection is pipelined but the
-- LFSR is not updated every clock cycle, the terminal count
-- does not need to be detected any cycles earlier.
--
-- c480_c is updated on cycle 0.
-- c480_t(2 downto 0) are updated on cycle 1.
-- c480_t(3) is updated on cycle 2.
-------------------------------------------------------------
process(clk)
begin
        if rising_edge(clk) then
                if (tc='1') and (c20_tc='1') then
                        if (c480_t(3)='1') then
                                c480_c<=b"0_0000_0000";
                        else
                                c480_c<=c480_c(7 downto 0)&
                                        (c480_c(8) xnor
                                        c480_c(4));
                        end if;
                end if;

                if (c480_t(2 downto 0)="111") then
                        c480_t(3)<='1';
                else
                        c480_t(3)<='0';
                end if;

                if (c480_c(8 downto 6)=b"000") then
                        c480_t(2)<='1';
                else
                        c480_t(2)<='0';
                end if;
                if (c480_c(5 downto 3)=b"010") then
                        c480_t(1)<='1';
                else
                        c480_t(1)<='0';
                end if;
                if (c480_c(2 downto 0)=b"010") then
                        c480_t(0)<='1';
```

```
                else
                        c480_t(0)<='0';
                end if;
        end if;
end process;

-------------------------------------------------------------
-- Timestamp carry-save counter
--
-- The timestamp must behave like a normal binary counter
-- which increments by 1 every frame (9.6ms).  Therefore, it
-- cannot be implemented using a LFSR.
--
-- Hence, the timestamp counter is implemented as a carry-save
-- counter.  This means that the carry propagates one bit at a
-- time every clock cycle.  Since the timestamp is 60 bits, it
-- will take 60 clock cycles to compute the new value.
--
-- The signal that controls the incrementing of the timestamp
-- must drive each data latch of the carry save adder, which
-- is too big a load for a single signal.  The signal is
-- therefore copied/delayed once to give the synthesizer more
-- opportunity to buffer the it.
--
-- The process communicate with the serial input process via
-- the handshaking signal wr_req and wr_ack.
--
-- time_t updates on cycle 0.
-- time_u updates on cycle 1.
-- time_s(0) and time_c(0) update on cycle 2.
-- ...
-- time_s(58) and time_c(58) update on cycle 60.
-- time_s(59) updates on cycle 61.
-- time_0(1) updates on cycle 62
-- ...
-- time_0(59) updates on cycle 120
-------------------------------------------------------------
process(clk)
begin
        if rising_edge(clk) then
                -- Set flag if it's time to update/increment
                -- the timestamp
                if (tc='1') and (c20_tc='1') and
                        (c480_t(3)='1') then
                        time_t<='1';
                else
                        time_t<='0';
                end if;
                time_u<=time_t;

                -- Timestamp carry-save counter
                if (time_u='1') then
                        if (wr_req='1') then
                                -- Update timestamp with new
                                -- value
                                time_s<=new_ts;
                                time_c<=b"0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000
                        else
                                -- Increment timestamp
                                time_s<=time_s(59 downto 0) xor
                                        (time_c(59 downto 1)&time_i);
                                time_c<=time_s(58 downto 0) and
                                        (time_c(58 downto 1)&time_i);
                        end if;
                else
                        -- Propagate carry
                        time_s<=time_s(59 downto 0) xor
                                (time_c(59 downto 1)&'0');
                        time_c<=time_s(58 downto 0) and
                                (time_c(58 downto 1)&'0');
                end if;

                -- Timestamp all-zeros and all-ones detection
                time_0<=time_s(59 downto 1) or
                        (time_0(58 downto 1)&time_s(0));
                time_1<=time_s(59 downto 1) and
                        (time_1(58 downto 1)&time_s(0));
```

```
                -- Timestamp increment value
                if (time_0(59)='0') or (time_1(59)='1') then
                        time_i<='0';
                else
                        time_i<='1';
                end if;

                -- Phase shift enable override
                if (time_0(59)='0') then
                        time_pr0<='1';
                        time_pr1<='1';
                        time_se0<='0';
                        time_se1<='0';
                elsif (time_1(59)='1') then
                        time_pr0<='1';
                        time_pr1<='1';
                        time_se0<='1';
                        time_se1<='1';
                else
                        time_pr0<='0';
                        time_pr1<='1';
                        time_se0<='0';
                        time_se1<='1';
                end if;

                -- Timestamp update handshaking
                if (time_u='1') and (wr_req='1') then
                        wr_ack<='1';
                elsif (wr_req='0') then
                        wr_ack<='0';
                end if;
        end if;
end process;

-------------------------------------------------------------
-- Shift enable control
--
-- The timestamps bits are encoded using a primary and a
-- secondary phase shift in the FM carrier.  Whether or not
-- the pahse shifts occur indicate if the data is a 0, a 1 or
-- a start bit:
--
--           | start | bit 0 | bit 1 |
-- ----------+-------+-------+-------+
-- primary   |  no   |  yes  |  yes  |
-- secondary |  yes  |  no   |  yes  |
-- ----------+-------+-------+-------+
--
-- The 0 bit is the most common, because a total of 480 bits
-- are actually transmitted, of which only 60 encode the
-- timestamp and one encodes the start bit.
--
-- The timestamp is sent LSB first so that future increases in
-- the number of timestamp bits will remain compatible with
-- this implementation.
--
-- en_pr, en_se and data are updated on cycle 0.
-- pr and se are also updated on cycle 0.
-------------------------------------------------------------
process(clk)
begin
        if rising_edge(clk) then
                if (tc='1') and (c20_tc='1') then
                        if (c480_t(3)='1') then
                                en_pr<=time_pr0;
                                en_se<=time_se1;
                                data<=time_s;
                        else
                                if (data(0)='1') then
                                        en_pr<=time_pr1;
                                        en_se<=time_se1;
                                else
                                        en_pr<=time_pr1;
                                        en_se<=time_se0;
                                end if;
                                data<="0"&data(59 downto 1);
```

```
                end if;
            end if;

            if ((tc='1') and (c20_pr='1')) and
                (en_pr='1') then
                pr<='1';
            else
                pr<='0';
            end if;
            if ((tc='1') and (c20_se='1')) and
                (en_se='1') then
                se<='1';
            else
                se<='0';
            end if;
        end if;
end process;

-------------------------------------------------------------
-- FM output
--
-- The fm counter is a 2 bit counter which depends only on two
-- other inputs (pr and se).  The logic for each bit can
-- therefore be implemented with a single LUT (which in the
-- Spartan 3 have a maximum of 4 inputs).  Hence, no LFSR or
-- carry save counter is required.
--
-- fm normally counts each clock cycle, but will hold its
-- state if either pr or se is active.
--
-- The actual output to the RF amplifier is the MSB of fm.
-------------------------------------------------------------
process(clk)
begin
        if rising_edge(clk) then
            if (pr='1') or (se='1') then
                null;
            else
                fm<=fm+1;
            end if;
        end if;
end process;
fm_out<=fm(1);

-------------------------------------------------------------
-- Serial port FSM counters (3.020833 MegaBAUD)
-------------------------------------------------------------
process(clk)
begin
        if rising_edge(clk) then
            case c12_c is
                when b"0000" =>
                    c12_c<=b"0001";
                    c12_t<='1';
                when b"0001" =>
                    c12_c<=b"0011";
                    c12_t<='0';
                when b"0011" =>
                    c12_c<=b"0010";
                    c12_t<='0';
                when b"0010" =>
                    c12_c<=b"0110";
                    c12_t<='0';
                when b"0110" =>
                    c12_c<=b"0111";
                    c12_t<='0';
                when b"0111" =>
                    c12_c<=b"0101";
                    c12_t<='0';
                when b"0101" =>
                    c12_c<=b"0100";
                    c12_t<='0';
                when b"0100" =>
                    c12_c<=b"1100";
                    c12_t<='0';
                when b"1100" =>
                    c12_c<=b"1101";
```

```
                        c12_t<='0';
                when b"1101" =>
                        c12_c<=b"1001";
                        c12_t<='0';
                when b"1001" =>
                        c12_c<=b"1000";
                        c12_t<='0';
                when others =>
                        c12_c<=b"0000";
                        c12_t<='0';
        end case;
    end if;
end process;

process(clk)
begin
    if rising_edge(clk) then
        if (c12_t='1') then
                case c96_c is
                        when b"000" =>
                                c96_c<=b"001";
                                c96_t<='1';
                                c24_t<='1';
                        when b"001" =>
                                c96_c<=b"011";
                                c96_t<='0';
                                c24_t<='0';
                        when b"011" =>
                                c96_c<=b"010";
                                c96_t<='0';
                                c24_t<='1';
                        when b"010" =>
                                c96_c<=b"110";
                                c96_t<='0';
                                c24_t<='0';
                        when b"110" =>
                                c96_c<=b"111";
                                c96_t<='0';
                                c24_t<='1';
                        when b"111" =>
                                c96_c<=b"101";
                                c96_t<='0';
                                c24_t<='0';
                        when b"101" =>
                                c96_c<=b"100";
                                c96_t<='0';
                                c24_t<='1';
                        when others =>
                                c96_c<=b"000";
                                c96_t<='0';
                                c24_t<='0';
                end case;
        else
                c96_t<='0';
                c24_t<='0';
        end if;
    end if;
end process;

-------------------------------------------------------------
-- Serial port input shift register
-------------------------------------------------------------
process(clk)
begin
    if rising_edge(clk) then
        if (c24_t='1') then
                if (ser_in(36 downto 34)="000") then
                        ser_cmd(8)<='1';
                        ser_cmd(7)<=ser_in(3);
                        ser_cmd(6)<=ser_in(7);
                        ser_cmd(5)<=ser_in(11);
                        ser_cmd(4)<=ser_in(15);
                        ser_cmd(3)<=ser_in(19);
                        ser_cmd(2)<=ser_in(23);
                        ser_cmd(1)<=ser_in(27);
                        ser_cmd(0)<=ser_in(31);
                        ser_in<=b"1111_1111_1111_1111_1111_1111_1111_1111_1111"&txd;
```

```
                        else
                                ser_cmd(8)<='0';
                                ser_in<=ser_in(35 downto 0)&
                                        txd;
                        end if;
                else
                        ser_cmd(8)<='0';
                end if;
        end if;
end process;

-------------------------------------------------------------
-- Serial port command decoding
-------------------------------------------------------------
process(clk)
begin
        if rising_edge(clk) then
                case (ser_cmd(8 downto 4)) is
                        when "10000" =>
                                ser_dec<="0000000000000001";
                        when "10001" =>
                                ser_dec<="0000000000000010";
                        when "10010" =>
                                ser_dec<="0000000000000100";
                        when "10011" =>
                                ser_dec<="0000000000001000";
                        when "10100" =>
                                ser_dec<="0000000000010000";
                        when "10101" =>
                                ser_dec<="0000000000100000";
                        when "10110" =>
                                ser_dec<="0000000001000000";
                        when "10111" =>
                                ser_dec<="0000000010000000";
                        when "11000" =>
                                ser_dec<="0000000100000000";
                        when "11001" =>
                                ser_dec<="0000001000000000";
                        when "11010" =>
                                ser_dec<="0000010000000000";
                        when "11011" =>
                                ser_dec<="0000100000000000";
                        when "11100" =>
                                ser_dec<="0001000000000000";
                        when "11101" =>
                                ser_dec<="0010000000000000";
                        when "11110" =>
                                ser_dec<="0100000000000000";
                        when "11111" =>
                                ser_dec<="1000000000000000";
                        when others =>
                                ser_dec<="0000000000000000";
                end case;
        end if;
end process;

-------------------------------------------------------------
-- Serial port command processing
-------------------------------------------------------------
process(clk)
begin
        if rising_edge(clk) then
                -- CMD 0 to 14: load a nibble of new timestamp
                if (ser_dec(0)='1') then
                        wr_req_p<='1';
                        new_ts(3 downto 0)<=
                                ser_cmd(3 downto 0);
                elsif (wr_ack='1') then
                        wr_req_p<='0';
                end if;
                if (ser_dec(1)='1') then
                        new_ts(7 downto 4)<=
                                ser_cmd(3 downto 0);
                end if;
                if (ser_dec(2)='1') then
                        new_ts(11 downto 8)<=
                                ser_cmd(3 downto 0);
```

```
                end if;
                if (ser_dec(3)='1') then
                        new_ts(15 downto 12)<=
                                ser_cmd(3 downto 0);
                end if;
                if (ser_dec(4)='1') then
                        new_ts(19 downto 16)<=
                                ser_cmd(3 downto 0);
                end if;
                if (ser_dec(5)='1') then
                        new_ts(23 downto 20)<=
                                ser_cmd(3 downto 0);
                end if;
                if (ser_dec(6)='1') then
                        new_ts(27 downto 24)<=
                                ser_cmd(3 downto 0);
                end if;
                if (ser_dec(7)='1') then
                        new_ts(31 downto 28)<=
                                ser_cmd(3 downto 0);
                end if;
                if (ser_dec(8)='1') then
                        new_ts(35 downto 32)<=
                                ser_cmd(3 downto 0);
                end if;
                if (ser_dec(9)='1') then
                        new_ts(39 downto 36)<=
                                ser_cmd(3 downto 0);
                end if;
                if (ser_dec(10)='1') then
                        new_ts(43 downto 40)<=
                                ser_cmd(3 downto 0);
                end if;
                if (ser_dec(11)='1') then
                        new_ts(47 downto 44)<=
                                ser_cmd(3 downto 0);
                end if;
                if (ser_dec(12)='1') then
                        new_ts(51 downto 48)<=
                                ser_cmd(3 downto 0);
                end if;
                if (ser_dec(13)='1') then
                        new_ts(55 downto 52)<=
                                ser_cmd(3 downto 0);
                end if;
                if (ser_dec(14)='1') then
                        new_ts(59 downto 56)<=
                                ser_cmd(3 downto 0);
                end if;
                -- CMD 15: read back time stamp
                if (ser_dec(15)='1') then
                        rd_req_p<='1';
                elsif (rd_ack='1') then
                        rd_req_p<='0';
                end if;

                wr_req<=wr_req_p;
                rd_req<=rd_req_p;
        end if;
end process;

-------------------------------------------------------------
-- Serial port output shift register
-------------------------------------------------------------
process(clk)
begin
        if rising_edge(clk) then
                rxd<=ser_out(0);

                if (rd_req='1') and (time_u='1') then
                        ser_out(0)<='1';
                        ser_out(1)<='0';
                        ser_out(2)<=time_s(0);
                        ser_out(3)<=time_s(1);
                        ser_out(4)<=time_s(2);
                        ser_out(5)<=time_s(3);
                        ser_out(6)<=time_s(4);
```

```
            ser_out(7)<=time_s(5);
            ser_out(8)<=time_s(6);
            ser_out(9)<=time_s(7);
            ser_out(10)<='1';
            ser_out(11)<='0';
            ser_out(12)<=time_s(8);
            ser_out(13)<=time_s(9);
            ser_out(14)<=time_s(10);
            ser_out(15)<=time_s(11);
            ser_out(16)<=time_s(12);
            ser_out(17)<=time_s(13);
            ser_out(18)<=time_s(14);
            ser_out(19)<=time_s(15);
            ser_out(20)<='1';
            ser_out(21)<='0';
            ser_out(22)<=time_s(16);
            ser_out(23)<=time_s(17);
            ser_out(24)<=time_s(18);
            ser_out(25)<=time_s(19);
            ser_out(26)<=time_s(20);
            ser_out(27)<=time_s(21);
            ser_out(28)<=time_s(22);
            ser_out(29)<=time_s(23);
            ser_out(30)<='1';
            ser_out(31)<='0';
            ser_out(32)<=time_s(24);
            ser_out(33)<=time_s(25);
            ser_out(34)<=time_s(26);
            ser_out(35)<=time_s(27);
            ser_out(36)<=time_s(28);
            ser_out(37)<=time_s(29);
            ser_out(38)<=time_s(30);
            ser_out(39)<=time_s(31);
            ser_out(40)<='1';
            ser_out(41)<='0';
            ser_out(42)<=time_s(32);
            ser_out(43)<=time_s(33);
            ser_out(44)<=time_s(34);
            ser_out(45)<=time_s(35);
            ser_out(46)<=time_s(36);
            ser_out(47)<=time_s(37);
            ser_out(48)<=time_s(38);
            ser_out(49)<=time_s(39);
            ser_out(50)<='1';
            ser_out(51)<='0';
            ser_out(52)<=time_s(40);
            ser_out(53)<=time_s(41);
            ser_out(54)<=time_s(42);
            ser_out(55)<=time_s(43);
            ser_out(56)<=time_s(44);
            ser_out(57)<=time_s(45);
            ser_out(58)<=time_s(46);
            ser_out(59)<=time_s(47);
            ser_out(60)<='1';
            ser_out(61)<='0';
            ser_out(62)<=time_s(48);
            ser_out(63)<=time_s(49);
            ser_out(64)<=time_s(50);
            ser_out(65)<=time_s(51);
            ser_out(66)<=time_s(52);
            ser_out(67)<=time_s(53);
            ser_out(68)<=time_s(54);
            ser_out(69)<=time_s(55);
            ser_out(70)<='1';
            ser_out(71)<='0';
            ser_out(72)<=time_s(56);
            ser_out(73)<=time_s(57);
            ser_out(74)<=time_s(58);
            ser_out(75)<=time_s(59);
            ser_out(76)<='0';
            ser_out(77)<='0';
            ser_out(78)<='0';
            ser_out(79)<='0';
            rd_ack<='1';
    elsif (c96_t='1') then
            ser_out<='1'&ser_out(79 downto 1);
    end if;
```

```
                        if (rd_req='0') then
                                rd_ack<='0';
                        end if;
                end if;
        end process;

        ----------------------------------------------------------------
        -- Serial port signal loopback
        ----------------------------------------------------------------
        cts<=rts;
        dsr<=dtr;
        dcd<=dtr;
        ri<='1';

        ----------------------------------------------------------------
        -- Debug
        ----------------------------------------------------------------
        process(clk)
        begin
                if rising_edge(clk) then
                        -- DBG0 triggers on the start of a frame
                        if (time_u='1') then
                                dbg0<='1';
                        elsif (tc='1') and (c20_cl='1') then
                                dbg0<='0';
                        end if;

                        -- DBG1 triggers on any phase shift
                        if (pr='1') or (se='1') then
                                dbg1<='1';
                        elsif (tc='1') and (c20_cl='1') then
                                dbg1<='0';
                        end if;

                        -- DBG2 displays the timestamp
                        dbg2<=data(0);
                end if;
        end process;
end arch;
```

The second script given is the UCF file for FM SYNC unit.
It is the physical mapping between the pins of the FPGA and the signals used in the previous VHDL script.

```
#########################################################
#                                                       #
# XC3S50ATQ144 Spartan3AN with 3 blocks of 18kbits RAM  #
#                                                       #
#########################################################

# Last updated: 2012-11-23

# Timing constraints for 20MHz oscillator
NET xtal PERIOD=50ns;
# Timing constraints for 290MHz system clock
NET clk PERIOD=3ns;# Max is 3.448

# Pin assignements
#NET TMS      LOC=P1;    #TMS,JTAG,VCCAUX
#NET TDI      LOC=P2;    #TDI,JTAG,VCCAUX
#NET NC       LOC=P3;    #IO_L02P_3,IO,BANK3
#NET NC       LOC=P4;    #IO_L01P_3,IO,BANK3
#NET NC       LOC=P5;    #IO_L02N_3,IO,BANK3
#NET NC       LOC=P6;    #IO_L01N_3,IO,BANK3
#NET NC       LOC=P7;    #IO_L03P_3,IO,BANK3
#NET NC       LOC=P8;    #IO_L03N_3,IO,BANK3
#NET GND      LOC=P9;    #GND,GND,GND
#NET NC       LOC=P10;   #IO_L04P_3,IO,BANK3
#NET NC       LOC=P11;   #IO_L04N_3/VREF_3,IO,BANK3
#NET NC       LOC=P12;   #IO_L05P_3/LHCLK0,LHCLK,BANK3
#NET NC       LOC=P13;   #IO_L05N_3/LHCLK1,LHCLK,BANK3
#NET 3V3      LOC=P14;   #VCCO_3,VCCO,BANK3
#NET NC       LOC=P15;   #IO_L06P_3/LHCLK2,LHCLK,BANK3
#NET NC       LOC=P16;   #IO_L06N_3/IRDY2/LHCLK3,LHCLK,BANK3
#NET GND      LOC=P17;   #GND,GND,GND
```

```
#NET NC       LOC=P18;   #IO_L07P_3/LHCLK4,LHCLK,BANK3
NET dbg2      LOC=P19;   #IO_L08P_3/TRDY2/LHCLK6,LHCLK,BANK3
#NET NC       LOC=P20;   #IO_L07N_3/LHCLK5,LHCLK,BANK3
#NET 3V3      LOC=P21;   #IO_L08N_3/LHCLK7,LHCLK,BANK3
#NET 1V2      LOC=P22;   #VCCINT,VCCINT,VCCINT
#NET 3V3      LOC=P23;   #VCCO_3,VCCO,BANK3
NET dbg1      LOC=P24;   #IO_L09P_3,IO,BANK3
#NET NC       LOC=P25;   #IO_L09N_3,IO,BANK3
#NET GND      LOC=P26;   #GND,GND,GND
#NET NC       LOC=P27;   #IO_L10P_3,IO,BANK3
#NET NC       LOC=P28;   #IO_L11P_3,IO,BANK3
NET dbg0      LOC=P29;   #IO_L10N_3,IO,BANK3
#NET NC       LOC=P30;   #IO_L11N_3,IO,BANK3
#NET NC       LOC=P31;   #IO_L12P_3,IO,BANK3
#NET NC       LOC=P32;   #IO_L12N_3,IO,BANK3
#NET NC       LOC=P33;   #IP_L13P_3,INPUT,BANK3
#NET GND      LOC=P34;   #GND,GND,GND
#NET 3V3      LOC=P35;   #IP_L13N_3/VREF_3,INPUT,BANK3
#NET 3V3      LOC=P36;   #VCCAUX,VCCAUX,VCCAUX
#NET 3V3      LOC=P37;   #IO_L01P_2/M1,DUAL,BANK2
#NET 3V3      LOC=P38;   #IO_L01N_2/M0,DUAL,BANK2
#NET GND      LOC=P39;   #IO_L02P_2/M2,DUAL,BANK2
#NET 3V3      LOC=P40;   #VCCO_2,VCCO,BANK2
#NET NC       LOC=P41;   #IO_L02N_2/CSO_B,DUAL,BANK2
#NET NC       LOC=P42;   #IO_L03P_2/RDWR_B,DUAL,BANK2
#NET 3V3      LOC=P43;   #IO_L04P_2/VS2,DUAL,BANK2
#NET 3V3      LOC=P44;   #IO_L03N_2/VS1,DUAL,BANK2
#NET 3V3      LOC=P45;   #IO_L04N_2/VS0,DUAL,BANK2
#NET NC       LOC=P46;   #IO_L05P_2,IO,BANK2
#NET NC       LOC=P47;   #IO_L06P_2,IO,BANK2
#NET NC       LOC=P48;   #IO_L05N_2/D7,DUAL,BANK2
#NET NC       LOC=P49;   #IO_L06N_2/D6,DUAL,BANK2
#NET NC       LOC=P50;   #IO_L07P_2/D5,DUAL,BANK2
#NET 3V3      LOC=P51;   #IO_L07N_2/D4,DUAL,BANK2
#NET 1V2      LOC=P52;   #VCCINT,VCCINT,VCCINT
#NET 3V3      LOC=P53;   #IP_2/VREF_2,INPUT,BANK2
#NET NC       LOC=P54;   #IO_L08P_2/GCLK14,GCLK,BANK2
#NET NC       LOC=P55;   #IO_L08N_2/GCLK15,GCLK,BANK2
#NET GND      LOC=P56;   #GND,GND,GND
#NET NC       LOC=P57;   #IO_L09P_2/GCLK0,GCLK,BANK2
#NET NC       LOC=P58;   #IO_L10P_2/GCLK2,GCLK,BANK2
#NET NC       LOC=P59;   #IO_L09N_2/GCLK1,GCLK,BANK2
#NET NC       LOC=P60;   #IO_L10N_2/GCLK3,GCLK,BANK2
#NET 3V3      LOC=P61;   #VCCO_2,VCCO,BANK2
#NET NC       LOC=P62;   #IO_2/MOSI/CSI_B,DUAL,BANK2
#NET NC       LOC=P63;   #IO_L11P_2/AWAKE,PWRMGMT,BANK2
#NET NC       LOC=P64;   #IO_L11N_2/DOUT,DUAL,BANK2
#NET GND      LOC=P65;   #GND,GND,GND
#NET 3V3      LOC=P66;   #VCCAUX,VCCAUX,VCCAUX
#NET NC       LOC=P67;   #IO_L12P_2/INIT_B,DUAL,BANK2
#NET NC       LOC=P68;   #IO_L12N_2/D3,DUAL,BANK2
#NET NC       LOC=P69;   #IO_L13P_2/D2,DUAL,BANK2
#NET NC       LOC=P70;   #IO_L14P_2/D1,DUAL,BANK2
#NET NC       LOC=P71;   #IO_L13N_2/D0/DIN/MISO,DUAL,BANK2
#NET NC       LOC=P72;   #IO_L14N_2/CCLK,DUAL,BANK2
#NET NC       LOC=P73;   #DONE,CONFIG,VCCAUX
#NET GND      LOC=P74;   #SUSPEND,PWRMGMT,VCCAUX
#NET NC       LOC=P75;   #IO_L02P_1/LDC1,DUAL,BANK1
#NET NC       LOC=P76;   #IO_L01P_1/HDC,DUAL,BANK1
#NET NC       LOC=P77;   #IO_L02N_1/LDC0,DUAL,BANK1
#NET NC       LOC=P78;   #IO_L01N_1/LDC2,DUAL,BANK1
#NET NC       LOC=P79;   #IO_1,IO,BANK1
#NET 3V3      LOC=P80;   #IP_1/VREF_1,INPUT,BANK1
#NET GND      LOC=P81;   #GND,GND,GND
#NET NC       LOC=P82;   #IO_L03P_1,IO,BANK1
#NET NC       LOC=P83;   #IO_L04P_1/RHCLK0,RHCLK,BANK1
#NET NC       LOC=P84;   #IO_L03N_1,IO,BANK1
#NET NC       LOC=P85;   #IO_L04N_1/RHCLK1,RHCLK,BANK1
#NET 3V3      LOC=P86;   #VCCO_1,VCCO,BANK1
NET fm_out    LOC=P87;   #IO_L05P_1/RHCLK2,RHCLK,BANK1
#NET NC       LOC=P88;   #IO_L05N_1/TRDY1/RHCLK3,RHCLK,BANK1
#NET GND      LOC=P89;   #GND,GND,GND
#NET NC       LOC=P90;   #IO_L06P_1/RHCLK4,RHCLK,BANK1
#NET NC       LOC=P91;   #IO_L07P_1/IRDY1/RHCLK6,RHCLK,BANK1
#NET NC       LOC=P92;   #IO_L06N_1/RHCLK5,RHCLK,BANK1
#NET 3V3      LOC=P93;   #IO_L07N_1/RHCLK7,RHCLK,BANK1
#NET 1V2      LOC=P94;   #VCCINT,VCCINT,VCCINT
```

```
#NET 3V3      LOC=P95;   #VCCO_1,VCCO,BANK1
#NET NC       LOC=P96;   #IO_L08P_1,IO,BANK1
#NET 3V3      LOC=P97;   #IP_1/VREF_1,INPUT,BANK1
#NET NC       LOC=P98;   #IO_L08N_1,IO,BANK1
#NET NC       LOC=P99;   #IO_L09P_1,IO,BANK1
#NET GND      LOC=P100;  #GND,GND,GND
#NET NC       LOC=P101;  #IO_L09N_1,IO,BANK1
#NET NC       LOC=P102;  #IO_L10P_1,IO,BANK1
#NET NC       LOC=P103;  #IO_L11P_1,IO,BANK1
#NET NC       LOC=P104;  #IO_L10N_1,IO,BANK1
#NET NC       LOC=P105;  #IO_L11N_1,IO,BANK1
#NET GND      LOC=P106;  #GND,GND,GND
#NET TDO      LOC=P107;  #TDO,JTAG,VCCAUX
#NET 3V3      LOC=P108;  #VCCAUX,VCCAUX,VCCAUX
#NET TCK      LOC=P109;  #TCK,JTAG,VCCAUX
#NET NC       LOC=P110;  #IO_L01P_0,IO,BANK0
#NET NC       LOC=P111;  #IO_L01N_0,IO,BANK0
#NET NC       LOC=P112;  #IO_L02P_0/VREF_0,IO,BANK0
#NET NC       LOC=P113;  #IO_L02N_0,IO,BANK0
#NET NC       LOC=P114;  #IO_L04P_0,IO,BANK0
#NET NC       LOC=P115;  #IO_L03P_0,IO,BANK0
#NET NC       LOC=P116;  #IO_L04N_0,IO,BANK0
#NET NC       LOC=P117;  #IO_L03N_0,IO,BANK0
#NET GND      LOC=P118;  #GND,GND,GND
#NET 3V3      LOC=P119;  #VCCO_0,VCCO,BANK0
#NET NC       LOC=P120;  #IO_L05P_0,IO,BANK0
#NET NC       LOC=P121;  #IO_L05N_0,IO,BANK0
#NET 1V2      LOC=P122;  #VCCINT,VCCINT,VCCINT
#NET 3V3      LOC=P123;  #IP_0/VREF_0,INPUT,BANK0
NET xtal      LOC=P124;  #IO_L06P_0/GCLK4,GCLK,BANK0
NET cbus3     LOC=P125;  #IO_L07P_0/GCLK6,GCLK,BANK0
NET cbus2     LOC=P126;  #IO_L06N_0/GCLK5,GCLK,BANK0
NET cts       LOC=P127;  #IO_L07N_0/GCLK7,GCLK,BANK0
#NET GND      LOC=P128;  #GND,GND,GND
NET dcd       LOC=P129;  #IO_L08P_0/GCLK8,GCLK,BANK0
NET dsr       LOC=P130;  #IO_L09P_0/GCLK10,GCLK,BANK0
NET ri        LOC=P131;  #IO_L08N_0/GCLK9,GCLK,BANK0
NET rxd       LOC=P132;  #IO_L09N_0/GCLK11,GCLK,BANK0
#NET 3V3      LOC=P133;  #VCCAUX,VCCAUX,VCCAUX
#NET NC       LOC=P134;  #IO_L10P_0,IO,BANK0
NET rts       LOC=P135;  #IO_L10N_0,IO,BANK0
#NET 3V3      LOC=P136;  #VCCO_0,VCCO,BANK0
#NET GND      LOC=P137;  #GND,GND,GND
NET dtr       LOC=P138;  #IO_L11P_0,IO,BANK0
NET txd       LOC=P139;  #IO_L11N_0,IO,BANK0
#NET 3V3      LOC=P140;  #IP_0,INPUT,BANK0
NET cbus1     LOC=P141;  #IO_L12P_0/VREF_0,IO,BANK0
NET cbus0     LOC=P142;  #IO_0,IO,BANK0
#NET GND      LOC=P143;  #IO_L12N_0/PUDC_B,DUAL,BANK0
#NET 3V3      LOC=P144;  #PROG_B,CONFIG,VCCAUX
```

The third VHDL script given is used to program the DACQ units.
After declaration and initialization of all signals and variables, the first component instantiated is the DCM to create the necessary clocks from the main crystal frequency. Then, there are the instantiations of the RAM block where the data will be temporally stored, before being sent to the computer. Next, follows the FM receiver process, which is used to detect the raw bit of the timestamp (5μs period) and adjust the clock period if local clock is too fast or too slow. Then, there is the process to decode start of reception of new timestamp and storing new data bits. Then, the processes that implement the synchronization is instantiated with the update of local clock every 2μs of 1 clock cycle (11ns). First the counters and then the comparison between the 2 sync references (one sent from FM SYNC unit and one generated from the local oscillator). Next, follows the process to decode the RSSI signal. Then, follows the control of the DACs, which is instantiated only if ultrasonic emission is needed (3 instantiations for 3 different ultrasonic frequencies). Then, follows the control of the ADCs, which controls the sampling of the signal coming from the connected sensors, here again, we have different instantiations for different type of sensors. The data sampled is sent to the FPGA or in case of daughter board used to

the Opal Kelly as well. Then, we have a process that controls the pre-processing of the data (filtering). The parameters of the filter depends on the sensors used. Next, follows the control of storing the data into the temporary storing location (RAM blocks). Finally, several processes to control the communication between the FPGA and the computer.

```
----------------------------------------------------------------------
-- Design Name: dacq
-- Module Name: dacq
-- File Name: dacq.vhd
--
-- Last updated: 2012-11-23
----------------------------------------------------------------------
-- ISE Project Settings:
--
-- Family:  Spartan3A and Spartan3AN
-- Device:  XC3S50AN
-- Package: TQG144
-- Speed:   -4
----------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
library UNISIM;-- For Xilinx primitives
use UNISIM.VComponents.all;

entity dacq is
        port(
                -- serial interface
                ft_txd:   in  std_logic;
                ft_rxd:   out std_logic;
                ft_rts:   in  std_logic;
                ft_cts:   out std_logic;
                ft_dtr:   in  std_logic;
                ft_dsr:   out std_logic;
                ft_dcd:   out std_logic;
                ft_ri:    out std_logic;
                ft_cbus0: in  std_logic;
                ft_cbus1: in  std_logic;
                ft_cbus2: in  std_logic;
                ft_cbus3: in  std_logic;

                -- DAC interface
                ndac_ncs: out std_logic;
                ndac_clk: out std_logic;
                ndac_din: out std_logic;
                ndac_ncl: out std_logic;

                pdac_ncs: out std_logic;
                pdac_clk: out std_logic;
                pdac_din: out std_logic;
                pdac_ncl: out std_logic;

                -- ADC interface
                pga0_ncs: out std_logic;
                pga0_dio: out std_logic;
                pga0_clk: out std_logic;
                adc0_sdi: out std_logic;
                adc0_sck: out std_logic;
                adc0_cnv: out std_logic;
                adc0_sdo: in  std_logic;

                pga1_ncs: out std_logic;
                pga1_dio: out std_logic;
                pga1_clk: out std_logic;
                adc1_sdi: out std_logic;
                adc1_sck: out std_logic;
                adc1_cnv: out std_logic;
                adc1_sdo: in  std_logic;

                pga2_ncs: out std_logic;
                pga2_dio: out std_logic;
                pga2_clk: out std_logic;
```

```
                adc2_sdi: out std_logic;
                adc2_sck: out std_logic;
                adc2_cnv: out std_logic;
                adc2_sdo: in  std_logic;

                pga3_ncs: out std_logic;
                pga3_dio: out std_logic;
                pga3_clk: out std_logic;
                adc3_sdi: out std_logic;
                adc3_sck: out std_logic;
                adc3_cnv: out std_logic;
                adc3_sdo: in  std_logic;

                -- FM receiver
                fm_ncs:   out std_logic;
                fm_clk:   out std_logic;
                fm_cmp:   in  std_logic;
                fm_dat:   in  std_logic;
                fm_lo:    out std_logic;-- (14/17)*75MHz = 61.765MHz

                -- Opal Kelly interface
                ok_fm:    out std_logic;
                ok_adc:   out std_logic_vector(15 downto 0);
                ok_state: out std_logic_vector(1 downto 0);
                ok_dac:   in  std_logic_vector(15 downto 0);
                ok_ctrl:  in  std_logic_vector(8 downto 0);

                -- Clock input
                xtal:     in  std_logic-- 75MHz oscillator
        );
end dacq;

architecture arch of dacq is
        ------------------------------------------------------------------
        -- Module selection
        ------------------------------------------------------------------
        subtype md is unsigned(31 downto 0);
        subtype gn is unsigned(1 downto 0);
        -- Standard module types:
        constant ASU1: md:=b"0100_0001_0101_0011_0101_0101_0011_0001";
        constant ASU2: md:=b"0100_0001_0101_0011_0101_0101_0011_0010";
        constant GEO1: md:=b"0100_0111_0100_0101_0100_1111_0011_0001";
        constant GEO2: md:=b"0100_0111_0100_0101_0100_1111_0011_0010";
        constant GEO3: md:=b"0100_0111_0100_0101_0100_1111_0011_0011";
        constant ICP1: md:=b"0100_1001_0100_0011_0101_0000_0011_0001";
        constant US25: md:=b"0101_0101_0101_0011_0011_0010_0011_0101";
        constant US33: md:=b"0101_0101_0101_0011_0011_0011_0011_0011";
        constant US40: md:=b"0101_0101_0101_0011_0011_0100_0011_0000";
        constant VBM1: md:=b"0101_0110_0100_0010_0100_1101_0011_0001";
        -- Records raw FM bits:
        constant DBUG: md:=b"0100_0100_0100_0010_0101_0101_0100_0111";
        -- Mono version of VBM1:
        constant MONO: md:=b"0100_1101_0100_1111_0100_1110_0100_1111";
        -- Non transmitting version of USxx:
        constant MON1: md:=b"0100_1101_0100_1111_0100_1110_0011_0001";
        constant MON2: md:=b"0100_1101_0100_1111_0100_1110_0011_0010";
        constant MON3: md:=b"0100_1101_0100_1111_0100_1110_0011_0011";
        -- Module generation specifiers:
        constant dacq_gen1: gn:=b"01";-- First gen
        constant dacq_gen2: gn:=b"10";-- Second gen with Opal Kelly
        -- Place module selection here:
        constant module_id: md:=ASU1;
        constant dacq_gen: gn:=dacq_gen1;-- Must use dacq_gen1.ucf
--        constant dacq_gen: gn:=dacq_gen2;-- Must use dacq_gen2.ucf
        ------------------------------------------------------------------
        -- Helper function to compute next buffer (increment mod 3)
        ------------------------------------------------------------------
        function nxt_buf(current: in std_logic_vector(1 downto 0))
                return std_logic_vector is
        begin
                case (current) is
                        when b"00" => return b"01";
                        when b"01" => return b"10";
                        when b"10" => return b"00";
                        when others => return b"11";
                end case;
        end function nxt_buf;
```

```
-------------------------------------------------------------
-- Helper function to compute prev buffer (decrement mod 3)
-------------------------------------------------------------
function prv_buf(current: std_logic_vector(1 downto 0))
        return std_logic_vector is
begin
        case (current) is
                when b"00" => return b"10";
                when b"01" => return b"00";
                when b"10" => return b"01";
                when others => return b"11";
        end case;
end function prv_buf;
-------------------------------------------------------------
-- Helper function to return number of ADC channels
-------------------------------------------------------------
function get_num_adc(module_id: unsigned(31 downto 0))
        return unsigned is
begin
        case (module_id) is
                when ASU1 => return b"100";
                when ASU2 => return b"100";
                when DBUG => return b"000";
                when GEO1 => return b"011";
                when GEO2 => return b"011";
                when GEO3 => return b"011";
                when ICP1 => return b"010";
                when MONO => return b"001";
                when MON1 => return b"001";
                when MON2 => return b"001";
                when MON3 => return b"001";
                when US25 => return b"001";
                when US33 => return b"001";
                when US40 => return b"001";
                when VBM1 => return b"010";
                when others => return b"000";
        end case;
end function get_num_adc;
-------------------------------------------------------------
-- Helper function to return number of DAC channels
-------------------------------------------------------------
function get_num_dac(module_id: unsigned(31 downto 0))
        return unsigned is
begin
        case (module_id) is
                when ASU1 => return b"00";
                when ASU2 => return b"00";
                when DBUG => return b"00";
                when GEO1 => return b"00";
                when GEO2 => return b"00";
                when GEO3 => return b"00";
                when ICP1 => return b"00";
                when MONO => return b"00";
                when MON1 => return b"00";
                when MON2 => return b"00";
                when MON3 => return b"00";
                when US25 => return b"10";
                when US33 => return b"10";
                when US40 => return b"10";
                when VBM1 => return b"00";
                when others => return b"00";
        end case;
end function get_num_dac;
-------------------------------------------------------------
-- Helper function to return filter coefficient
-------------------------------------------------------------
function get_coef(module_id: unsigned(31 downto 0))
        return unsigned is
begin
        case (module_id) is
                when ASU1 =>
                        -- 0.145364, for 12.500kHz
                        return b"0_0010_0101_0011_0110_1";
                when ASU2 =>
                        -- 0.145364, for 12.500kHz
                        return b"0_0010_0101_0011_0110_1";
                when DBUG =>
```

```
                        -- 1.00000, for no filter
                        return b"1_0000_0000_0000_0000_0";
                when GEO1 =>
                        -- 0.188961, for 16.667kHz
                        return b"0_0011_0000_0110_0000_0";
                when GEO2 =>
                        -- 0.188961, for 16.667kHz
                        return b"0_0011_0000_0110_0000_0";
                when GEO3 =>
                        -- 0.188961, for 16.667kHz
                        return b"0_0011_0000_0110_0000_0";
                when ICP1 =>
                        -- 0.269597, for 25.000kHz
                        return b"0_0100_0101_0000_0100_1";
                when MONO =>
                        -- 0.466512, for 50.000kHz
                        return b"0_0111_0111_0110_1101_1";
                when MON1 =>
                        -- 0.466512, for 50.000kHz
                        return b"0_0111_0111_0110_1101_1";
                when MON2 =>
                        -- 0.466512, for 50.000kHz
                        return b"0_0111_0111_0110_1101_1";
                when MON3 =>
                        -- 0.466512, for 50.000kHz
                        return b"0_0111_0111_0110_1101_1";
                when US25 =>
                        -- 0.466512, for 50.000kHz
                        return b"0_0111_0111_0110_1101_1";
                when US33 =>
                        -- 0.466512, for 50.000kHz
                        return b"0_0111_0111_0110_1101_1";
                when US40 =>
                        -- 0.466512, for 50.000kHz
                        return b"0_0111_0111_0110_1101_1";
                when VBM1 =>
                        -- 0.269597, for 25.000kHz
                        return b"0_0100_0101_0000_0100_1";
                when others =>
                        -- 1.00000, for no filter
                        return b"1_0000_0000_0000_0000_0";
        end case;
end function get_coef;
-------------------------------------------------------------
-- Constants
-------------------------------------------------------------
constant num_adc:    unsigned(1 downto 0):=
        get_num_adc(module_id);
constant num_dac:    unsigned(1 downto 0):=
        get_num_dac(module_id);
constant coef:       unsigned(17 downto 0):=
        get_coef(module_id);
-------------------------------------------------------------
-- Local signals for DCM
-------------------------------------------------------------
signal   gclk:       std_logic;
signal   gclk0:      std_logic;
signal   clk:        std_logic;
signal   clkfx:      std_logic;
signal   clki0:      std_logic;
signal   clkifb:     std_logic;
signal   clkifx:     std_logic;
signal   clki:       std_logic;
-------------------------------------------------------------
-- Local signals for FM receiver
-------------------------------------------------------------
signal   fm_smp:     std_logic;
signal   cnt_5us:    unsigned(8 downto 0);
signal   cnt_ones:   unsigned(8 downto 0);
signal   fm_sr:      std_logic_vector(11 downto 0);
signal   fm_bits:    unsigned(15 downto 0);
-------------------------------------------------------------
-- Local signals for pattern detector
-------------------------------------------------------------
signal   fm_sync:    std_logic_vector(0 downto 0);
signal   fm_cnt:     unsigned(7 downto 0);
signal   timestamp:  unsigned(59 downto 0);
```

```
-------------------------------------------------------------
-- Local signals for sequencer
-------------------------------------------------------------
signal   seq_cnt:    unsigned(36 downto 0);
signal   seq_acc:    unsigned(19 downto 0);
signal   seq_sync:   std_logic_vector(0 downto 0);
-------------------------------------------------------------
-- Local signals for phase detector
-------------------------------------------------------------
signal   phs_cur:    unsigned(1 downto 0);
signal   phs_prv:    unsigned(1 downto 0);
signal   phs_vld:    unsigned(5 downto 0);
signal   phs_cnt:    unsigned(19 downto 0);
signal   phs_lst:    unsigned(19 downto 0);
signal   phs_inc:    unsigned(18 downto 0);
-------------------------------------------------------------
-- Local signals for RSSI ADC
-------------------------------------------------------------
signal   fm_dat_d:   std_logic;
signal   rssi:       unsigned(7 downto 0);
-------------------------------------------------------------
-- Local signals for DAC table lookup
-------------------------------------------------------------
signal   dac_cnt:    unsigned(5 downto 0);
signal   dac_phs:    unsigned(4 downto 0);
signal   dac_dat:    unsigned(15 downto 0);
signal   dac_ndat:   unsigned(15 downto 0);
signal   dac_pdat:   unsigned(15 downto 0);
-------------------------------------------------------------
-- Local signals for DAC control
-------------------------------------------------------------
signal   dac_ncs:    std_logic;
signal   dac_clk:    std_logic;
signal   dac_ndin:   std_logic;
signal   dac_pdin:   std_logic;
-------------------------------------------------------------
-- Local signals for PGA control
-------------------------------------------------------------
signal   pga_ncs:    std_logic;
signal   pga_clk:    std_logic;
signal   pga_dio:    std_logic;
signal   pga_act:    std_logic;
signal   pga_lst:    unsigned(7 downto 0);
-------------------------------------------------------------
-- Local signals for ADC control
-------------------------------------------------------------
signal   adc_sck:    std_logic;
signal   adc_cnv:    std_logic;
signal   adc0_sdo_d: std_logic;
signal   adc1_sdo_d: std_logic;
signal   adc2_sdo_d: std_logic;
signal   adc3_sdo_d: std_logic;
signal   adc0_in:    unsigned(15 downto 0);
signal   adc1_in:    unsigned(15 downto 0);
signal   adc2_in:    unsigned(15 downto 0);
signal   adc3_in:    unsigned(15 downto 0);
-------------------------------------------------------------
-- Local signals for ADC filtering
-------------------------------------------------------------
signal   prod:       std_logic_vector(35 downto 0);-- No reg
signal   mult:       unsigned(17 downto 0);-- No reg
signal   round:      unsigned(16 downto 0);-- No reg
signal   filt0:      unsigned(15 downto 0);
signal   filt1:      unsigned(15 downto 0);
signal   filt2:      unsigned(15 downto 0);
signal   filt3:      unsigned(15 downto 0);
-------------------------------------------------------------
-- Local signals for RAM data storage
-------------------------------------------------------------
signal   ram_cnt:    unsigned(2 downto 0);
signal   ram_addr:   unsigned(9 downto 0);
signal   ram_data:   unsigned(15 downto 0);
signal   ram_wea:    std_logic_vector(1 downto 0);
signal   ram_nnb:    std_logic;
signal   ram_min:    unsigned(15 downto 0);
signal   ram_max:    unsigned(15 downto 0);
signal   ram_wen:    std_logic_vector(2 downto 0);-- No reg
```

```
------------------------------------------------------------
-- Local signals for serial transmit
------------------------------------------------------------
signal   dtr_d:      std_logic;
signal   rts_d:      std_logic;
signal   ser_rst:    std_logic;
signal   ser_cyc:    unsigned(4 downto 0);
signal   ser_bit:    unsigned(3 downto 0);
signal   ser_addr:   unsigned(10 downto 0);
signal   ser_out:    std_logic_vector(7 downto 0);
signal   ser_nnb:    std_logic_vector(0 downto 0);
------------------------------------------------------------
-- Local signals for buffer controller
------------------------------------------------------------
signal   ctrl_gen:   std_logic_vector(1 downto 0);
signal   ctrl_ser:   std_logic_vector(1 downto 0);
signal   ctrl_spr:   std_logic;
signal   ctrl_data:  std_logic_vector(23 downto 0);-- No reg
signal   ctrl_out:   std_logic_vector(7 downto 0);-- No reg
------------------------------------------------------------
-- Local signals for serial receive
------------------------------------------------------------
signal   ser_cnt:    unsigned(8 downto 0);
signal   ser_in:     unsigned(6 downto 0);
signal   pga:        unsigned(15 downto 0);
signal   txd_d:      std_logic;
------------------------------------------------------------
-- Initial registered internal signal values
------------------------------------------------------------
attribute INIT: string;
attribute INIT of fm_smp:     signal is b"0";
attribute INIT of cnt_5us:    signal is b"0_0011_1110";
attribute INIT of cnt_ones:   signal is b"0_0001_1111";
attribute INIT of fm_sr:      signal is b"1000_1000_1000";
attribute INIT of fm_bits:    signal is b"1111_1111_1111_1111";
attribute INIT of fm_sync:    signal is b"0";
attribute INIT of fm_cnt:     signal is b"000000_00";
attribute INIT of timestamp:  signal is b"0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000";
attribute INIT of seq_cnt:    signal is b"0000_0000_0000_0000_00_0000_0000_000_0_000_0000";
attribute INIT of seq_acc:    signal is b"00_0000_0000_0000_0000_00";
attribute INIT of seq_sync:   signal is b"0";
attribute INIT of phs_cur:    signal is b"01";
attribute INIT of phs_prv:    signal is b"01";
attribute INIT of phs_vld:    signal is b"00_0000";
attribute INIT of phs_cnt:    signal is b"00_0000_0000_0000_0000_00";
attribute INIT of phs_lst:    signal is b"00_0000_0000_0000_0000_00";
attribute INIT of phs_inc:    signal is b"1_0000_0000_0000_0000_00";
attribute INIT of fm_dat_d:   signal is b"0";
attribute INIT of rssi:       signal is b"0000_0001";
attribute INIT of dac_cnt:    signal is b"000000";
attribute INIT of dac_phs:    signal is b"00000";
attribute INIT of dac_dat:    signal is b"1000_0000_0000_0000";
attribute INIT of dac_ndat:   signal is b"1000_0000_0000_0000";
attribute INIT of dac_pdat:   signal is b"1000_0000_0000_0000";
attribute INIT of dac_ncs:    signal is b"1";
attribute INIT of dac_clk:    signal is b"0";
attribute INIT of dac_ndin:   signal is b"0";
attribute INIT of dac_pdin:   signal is b"0";
attribute INIT of pga_ncs:    signal is b"1";
attribute INIT of pga_clk:    signal is b"0";
attribute INIT of pga_dio:    signal is b"0";
attribute INIT of pga_act:    signal is b"0";
attribute INIT of pga_lst:    signal is b"0000_0000";
attribute INIT of adc_sck:    signal is b"0";
attribute INIT of adc_cnv:    signal is b"1";
attribute INIT of adc0_sdo_d: signal is b"1";
attribute INIT of adc1_sdo_d: signal is b"1";
attribute INIT of adc2_sdo_d: signal is b"1";
attribute INIT of adc3_sdo_d: signal is b"1";
attribute INIT of adc0_in:    signal is b"1111_1111_1111_1111";
attribute INIT of adc1_in:    signal is b"1111_1111_1111_1111";
attribute INIT of adc2_in:    signal is b"1111_1111_1111_1111";
attribute INIT of adc3_in:    signal is b"1111_1111_1111_1111";
attribute INIT of filt0:      signal is b"1000_0000_0000_0000";
attribute INIT of filt1:      signal is b"1000_0000_0000_0000";
attribute INIT of filt2:      signal is b"1000_0000_0000_0000";
attribute INIT of filt3:      signal is b"1000_0000_0000_0000";
```

```
	attribute INIT of ram_cnt:    signal is b"000";
	attribute INIT of ram_addr:   signal is b"00_0000_0000";
	attribute INIT of ram_data:   signal is b"1111_1111_1111_1111";
	attribute INIT of ram_wea:    signal is b"00";
	attribute INIT of ram_nnb:    signal is b"0";
	attribute INIT of ram_min:    signal is b"1111_1111_1111_1111";
	attribute INIT of ram_max:    signal is b"0000_0000_0000_0000";
	attribute INIT of dtr_d:      signal is b"1";
	attribute INIT of rts_d:      signal is b"1";
	attribute INIT of ser_rst:    signal is b"1";
	attribute INIT of ser_cyc:    signal is b"11101";
	attribute INIT of ser_bit:    signal is b"1001";
	attribute INIT of ser_addr:   signal is b"000_0000_0000";
	attribute INIT of ser_out:    signal is b"11111111";
	attribute INIT of ser_nnb:    signal is b"0";
	attribute INIT of ctrl_gen:   signal is b"00";
	attribute INIT of ctrl_ser:   signal is b"11";
	attribute INIT of ctrl_spr:   signal is b"0";
	attribute INIT of ser_cnt:    signal is b"1001_01110";
	attribute INIT of ser_in:     signal is b"000_0000";
	attribute INIT of pga:        signal is b"0010_1010_0000_0001";
	attribute INIT of txd_d:      signal is b"1";
	------------------------------------------------------------
	-- Initial registered external signal values
	------------------------------------------------------------
	attribute INIT of fm_clk:     signal is b"0";
	attribute INIT of fm_ncs:     signal is b"0";
	attribute INIT of ft_rxd:     signal is b"1";
	attribute INIT of ok_fm:      signal is b"0";
	attribute INIT of ok_adc:     signal is b"1111_1111_1111_1111";
	attribute INIT of ok_state:   signal is b"11";
begin
	------------------------------------------------------------
	-- DCM:
	-- Clock input:         xtal -> IBUFG -> gclk -> DCM
	-- Feedback:            DCM -> gclk0 -> BUFG -> clk
	-- FM local oscillator: DCM -> clkfx -> BUFG -> fm_lo
	--
	-- Feedback:            DCM -> clki0 -> BUFG -> clkifb
	-- Internal clock:      DCM -> clkifx -> BUFG -> clki
	------------------------------------------------------------
	u_ibufg: IBUFG
		generic map (
			IOSTANDARD=>"LVCMOS33"
		)
		port map (
			O=>gclk,
			I=>xtal
		);
	u_dcm: DCM_SP
		generic map (
			CLKDV_DIVIDE=>2.0,
			CLKFX_DIVIDE=>17,
			CLKFX_MULTIPLY=>14,
			CLKIN_DIVIDE_BY_2=>false,
			CLKIN_PERIOD=>13.0,
			CLKOUT_PHASE_SHIFT=>"NONE",
			CLK_FEEDBACK=>"1X",
			DESKEW_ADJUST=>"SYSTEM_SYNCHRONOUS",
			DFS_FREQUENCY_MODE=>"LOW",
			DLL_FREQUENCY_MODE=>"LOW",
			DSS_MODE=>"NONE",
			DUTY_CYCLE_CORRECTION=>true,
			FACTORY_JF=>X"c080",
			PHASE_SHIFT=>0,
			STARTUP_WAIT=>false
		)
		port map (
			CLK0=>gclk0,
			CLK180=>open,
			CLK270=>open,
			CLK2X=>open,
			CLK2X180=>open,
			CLK90=>open,
			CLKDV=>open,
			CLKFX=>clkfx,
			CLKFX180=>open,
```

```
                LOCKED=>open,
                PSDONE=>open,
                STATUS=>open,
                CLKFB=>clk,
                CLKIN=>gclk,
                DSSEN=>'0',
                PSCLK=>'0',
                PSEN=>'0',
                PSINCDEC=>'0',
                RST=>'0'
        );
u_fb: BUFG
        port map (
                O=>clk,
                I=>gclk0
        );
u_bufg: BUFG
        port map (
                O=>fm_lo,
                I=>clkfx
        );

u_dcmi: DCM_SP
        generic map (
                CLKDV_DIVIDE=>2.0,
                CLKFX_DIVIDE=>5,
                CLKFX_MULTIPLY=>6,
                CLKIN_DIVIDE_BY_2=>false,
                CLKIN_PERIOD=>13.0,
                CLKOUT_PHASE_SHIFT=>"NONE",
                CLK_FEEDBACK=>"1X",
                DESKEW_ADJUST=>"SYSTEM_SYNCHRONOUS",
                DFS_FREQUENCY_MODE=>"LOW",
                DLL_FREQUENCY_MODE=>"LOW",
                DSS_MODE=>"NONE",
                DUTY_CYCLE_CORRECTION=>true,
                FACTORY_JF=>X"c080",
                PHASE_SHIFT=>0,
                STARTUP_WAIT=>false
        )
        port map (
                CLK0=>clki0,
                CLK180=>open,
                CLK270=>open,
                CLK2X=>open,
                CLK2X180=>open,
                CLK90=>open,
                CLKDV=>open,
                CLKFX=>clkifx,
                CLKFX180=>open,
                LOCKED=>open,
                PSDONE=>open,
                STATUS=>open,
                CLKFB=>clkifb,
                CLKIN=>gclk,
                DSSEN=>'0',
                PSCLK=>'0',
                PSEN=>'0',
                PSINCDEC=>'0',
                RST=>'0'
        );
u_fbi: BUFG
        port map (
                O=>clkifb,
                I=>clki0
        );
u_bufgi: BUFG
        port map (
                O=>clki,
                I=>clkifx
        );

-----------------------------------------------------------------
-- RAM block instantiation
-----------------------------------------------------------------
blk_0: RAMB16BWE_S18_S9
        generic map (
```

```
                INIT_A=>X"00000",
                INIT_B=>X"000",
                SIM_COLLISION_CHECK=>"ALL",
                SRVAL_A=>X"00000",
                SRVAL_B=>X"000",
                WRITE_MODE_A=>"WRITE_FIRST",
                WRITE_MODE_B=>"WRITE_FIRST"
        )
        port map (
                DOA=>open,
                DOB=>ctrl_data(7 downto 0),
                DOPA=>open,
                DOPB=>open,
                ADDRA=>std_logic_vector(ram_addr),
                ADDRB=>std_logic_vector(ser_addr),
                CLKA=>clki,
                CLKB=>clki,
                DIA=>std_logic_vector(ram_data),
                DIB=>b"00000000",
                DIPA=>b"00",
                DIPB=>b"0",
                ENA=>ram_wen(0),
                ENB=>'1',
                SSRA=>'0',
                SSRB=>'0',
                WEA=>ram_wea,
                WEB=>'0'
        );
blk_1: RAMB16BWE_S18_S9
        generic map (
                INIT_A=>X"00000",
                INIT_B=>X"000",
                SIM_COLLISION_CHECK=>"ALL",
                SRVAL_A=>X"00000",
                SRVAL_B=>X"000",
                WRITE_MODE_A=>"WRITE_FIRST",
                WRITE_MODE_B=>"WRITE_FIRST"
        )
        port map (
                DOA=>open,
                DOB=>ctrl_data(15 downto 8),
                DOPA=>open,
                DOPB=>open,
                ADDRA=>std_logic_vector(ram_addr),
                ADDRB=>std_logic_vector(ser_addr),
                CLKA=>clki,
                CLKB=>clki,
                DIA=>std_logic_vector(ram_data),
                DIB=>b"00000000",
                DIPA=>b"00",
                DIPB=>b"0",
                ENA=>ram_wen(1),
                ENB=>'1',
                SSRA=>'0',
                SSRB=>'0',
                WEA=>ram_wea,
                WEB=>'0'
        );
blk_2: RAMB16BWE_S18_S9
        generic map (
                INIT_A=>X"00000",
                INIT_B=>X"000",
                SIM_COLLISION_CHECK=>"ALL",
                SRVAL_A=>X"00000",
                SRVAL_B=>X"000",
                WRITE_MODE_A=>"WRITE_FIRST",
                WRITE_MODE_B=>"WRITE_FIRST"
        )
        port map (
                DOA=>open,
                DOB=>ctrl_data(23 downto 16),
                DOPA=>open,
                DOPB=>open,
                ADDRA=>std_logic_vector(ram_addr),
                ADDRB=>std_logic_vector(ser_addr),
                CLKA=>clki,
                CLKB=>clki,
```

```
                DIA=>std_logic_vector(ram_data),
                DIB=>b"00000000",
                DIPA=>b"00",
                DIPB=>b"0",
                ENA=>ram_wen(2),
                ENB=>'1',
                SSRA=>'0',
                SSRB=>'0',
                WEA=>ram_wea,
                WEB=>'0'
        );

-----------------------------------------------------------------
-- 18x18 multiplier block instantiation
-----------------------------------------------------------------
mult_0: MULT18X18SIO
        generic map (
                AREG=>0,
                BREG=>0,
                B_INPUT=>"DIRECT",
                PREG=>1
        )
        port map (
                BCOUT=>open,
                P=>prod,
                A=>std_logic_vector(mult),
                B=>std_logic_vector(coef),
                BCIN=>b"00_0000_0000_0000_0000",
                CEA=>'1',
                CEB=>'1',
                CEP=>'1',
                CLK=>clki,
                RSTA=>'0',
                RSTB=>'0',
                RSTP=>'0'
        );

-----------------------------------------------------------------
-- FM receiver
--
-- The signal from the FM receiver section is sampled into
-- fm_smp.
--
-- cnt_5us nominally counts from 62 (b"0_0011_1110") to 511
-- (450 clock cycles).  The count is started at 61
-- (b"0_0011_1101") if the interval needs to be lengthened, or
-- at 63 (b"0_0011_1111") if it needs to be shortened.
--
-- cnt_ones counts the number of times the comparator output
-- is high.  If the number exceeds 225, the bit for that
-- interval is considered to be a one, otherwise it is
-- considered to be a zero.  To make the detection easier, the
-- counter is started from 31 (b"0_0001_1111"), so that the
-- MSB is set when the number of ones reaches 226.
--
-- The pulse width from the comparator is expected to be
-- shorter than 5us, so if the interval ends with the
-- comparator output high, it is assumed that the intervals
-- are straddling bit boundaries, and that the interval phase
-- needs to be adjusted.  The direction of the adjustment is
-- based on whether the interval had a majority of ones
-- (cnt_ones(8)='1'), in which case the interval is stretched,
-- or a majority of zeros (cnt_ones(8)='0'), in which case the
-- interval is shrunk.
--
-- Ideal:
-- :     1     :     0     :     1     :     0     :
-- : _________ :           : _________ :           :
-- :/         \:___________:/         \:___________:
--
-- Interval ending late, shrink needed:
-- :     1     :     0     :     1     :     0     :
-- :_______    :          _:_______    :          _:
-- :       \___:_________/ :       \___:_________/ :
--
-- Interval ending early, stretch needed:
-- :     1     :     0     :     1     :     0     :
```

```
-- :    _______:_          :    _______:_          :
-- :___/       : \_________:___/       : \_________:
--
-- The raw bits (cnt_ones(8)) are stored in shift register
-- fm_sr.
-------------------------------------------------------------
process(clki)
begin
        if (rising_edge(clki)) then
                -- Sample comparator output
                fm_smp<=fm_cmp;

                -- Interval counter
                if (cnt_5us=b"1_1111_1111") then
                        if (fm_smp='1') and
                                (cnt_ones(8)='1') then
                                -- Lengthen interval
                                cnt_5us<=b"0_0011_1101";
                        elsif (fm_smp='1') and
                                (cnt_ones(8)='0') then
                                -- Shorten interval
                                cnt_5us<=b"0_0011_1111";
                        else
                                -- Maintain interval
                                cnt_5us<=b"0_0011_1110";
                        end if;
                else
                        cnt_5us<=cnt_5us+1;
                end if;

                -- Ones counter
                if (cnt_5us=b"1_1111_1111") then
                        if (fm_smp='1') then
                                -- Start at 32
                                cnt_ones<=b"0_0010_0000";
                        else
                                -- Start at 31
                                cnt_ones<=b"0_0001_1111";
                        end if;
                else
                        if (fm_smp='1') then
                                cnt_ones<=cnt_ones+1;
                        end if;
                end if;

                -- Raw bits shift register
                if (cnt_5us=b"1_1111_1111") then
                        if (cnt_ones(8)='1') then
                                fm_sr<=fm_sr(10 downto 0)&'1';
                                ok_fm<='1';
                        else
                                fm_sr<=fm_sr(10 downto 0)&'0';
                                ok_fm<='0';
                        end if;
                end if;

                -- Sample the fm comparator output every 2/3us
                -- (1.5MHz)
                if (num_adc=b"000") then
                        if (seq_cnt(7 downto 0)=b"0_010_1001") or
                                (seq_cnt(7 downto 0)=b"0_110_0101") or
                                (seq_cnt(7 downto 0)=b"1_100_0111") then
                                fm_bits<=fm_bits(14 downto 0)&
                                        fm_smp;
                        end if;
                else
                        fm_bits<=b"1111_1111_1111_1111";
                end if;
        end if;
end process;

-------------------------------------------------------------
-- Pattern detector
--
-- A one data bit is encoded as 1010.  A zero data bit is
-- encoded as 1000.  A frame sync data bit is encoded as 0010.
-- The frame sync data bit is always preceeded by at least two
```

```
-- zero data bits, so the pattern to look for is 1000 1000
-- 0010.
-------------------------------------------------------------
process(clki)
begin
        if (rising_edge(clki)) then
                -- Detect frame sync
                if (cnt_5us=b"1_1111_1111") and
                        (fm_sr=b"1000_1000_0010") then
                        fm_sync<=b"1";
                else
                        fm_sync<=b"0";
                end if;

                -- Raw bit counter
                if (cnt_5us=b"1_1111_1111") then
                        if (fm_sr=b"1000_1000_0010") then
                                fm_cnt(1 downto 0)<=b"00";
                                fm_cnt(7 downto 2)<=b"000000";
                        else
                                fm_cnt(1 downto 0)<=
                                        fm_cnt(1 downto 0)+1;
                                if (fm_cnt(1 downto 0)=b"11") and
                                        (fm_cnt(7 downto 2)/=b"111100") then
                                        fm_cnt(7 downto 2)<=
                                                fm_cnt(7 downto 2)+1;
                                end if;
                        end if;
                end if;

                -- Data bit shift register
                if (cnt_5us=b"1_1111_1111") and
                        (fm_cnt(1 downto 0)=b"11") and
                        (fm_cnt(7 downto 2)/=b"111100") then
                        if (fm_sr(3 downto 0)=b"1010") then
                                timestamp<='1'&
                                        timestamp(59 downto 1);
                        else
                                timestamp<='0'&
                                        timestamp(59 downto 1);
                        end if;
                end if;
        end if;
end process;

-------------------------------------------------------------
-- Sequencer counter
--
-- The sequence counter is divided into four parts:
--
-- seq_cnt(6 downto 0):   Starts at 38 and counts to 127 when
--                        seq_cnt(7) is 1, starts at 37, 38 or
--                        39 when seq_cnt(7) is 0.
-- seq_cnt(7):            Starts at 0 and counts to 1.
-- seq_cnt(10 downto 8):  Starts at 0 and counts to 4.
-- seq_cnt(20 downto 11): Starts at 0 and counts to 959.
-- seq_cnt(36 downto 21): Starts at 1 and counts to 65535
--
-- Note that seq(10 downto 7) effectively starts at 0 and
-- counts to 9. But because seq_cnt(7) is used to control the
-- range of the lower seven bits, it is grouped seperately.
--
-- The lower 8 bits may start at one of three possible values,
-- according to whether the next 2 microsecond interval needs
-- to be stretched, maintained, or shrunk.  But the sequence
-- will always cover the following pattern:
--
--  0_37 0_010_0101 (optional -- stretch)
--  0_38 0_010_0110 (optional -- maintain)
--  0_39 0_010_0111
-- ...
-- 0_127 0_111_1111
--  1_38 1_010_0110
-- ...
-- 1_127 1_111_1111
--
-- This permits the values from x_39 to x_127 to be reliably
```

```
-- used by the DACs (which run a 1MHz), and the values 0_39 to
-- 1_127 to be reliably used by the ADCs (which run at
-- 500kHz).  The values 0_37 and 0_38 should be designed to
-- occur during the stretchable portion of the ADC sequence.
--
-- The cycle length is dependent on the value of phs_inc,
-- which is a fixed-point number, with a single integer bit.
-- It is added to seq_acc 4800 times before being updated (on
-- seq_sync).
--
-- Examples:
--
-- phs_inc=0.0000: 4800*181.0=868800 cycles -> 9653.33us
-- phs_inc=0.5000: 4800*180.5=866400 cycles -> 9626.67us
-- phs_inc=1.0000: 4800*180.0=864000 cycles -> 9600.00us
-- phs_inc=1.5000: 4800*179.5=861600 cycles -> 9573.33us
-- phs_inc=2.0000: 4800*179.0=859200 cycles -> 9546.67us
-----------------------------------------------------------------
process(clki)
begin
        if (rising_edge(clki)) then
                -- Bits 7 downto 0
                if (seq_cnt(7 downto 0)=b"1_111_1111") then
                        case (seq_acc(19 downto 18)) is
                                when b"00" =>
                                        -- Start at 37
                                        seq_cnt(7 downto 0)<=
                                                b"0_010_0101";
                                when b"01" =>
                                        -- Start at 38
                                        seq_cnt(7 downto 0)<=
                                                b"0_010_0110";
                                when others =>
                                        -- Start at 39
                                        seq_cnt(7 downto 0)<=
                                                b"0_010_0111";
                        end case;
                        seq_acc<=(b"00"&seq_acc(17 downto 0))+
                                (b"0"&phs_inc);
                elsif (seq_cnt(7 downto 0)=b"0_111_1111") then
                        -- Start at 38
                        seq_cnt(7 downto 0)<=
                                b"1_010_0110";
                else
                        -- Increment
                        seq_cnt(6 downto 0)<=
                                seq_cnt(6 downto 0)+1;
                end if;

                -- Bits 10 downto 8
                if (seq_cnt(7 downto 0)=b"1_111_1111") then
                        if (seq_cnt(10 downto 8)=b"100") then
                                seq_cnt(10 downto 8)<=b"000";
                        else
                                seq_cnt(10 downto 8)<=
                                        seq_cnt(10 downto 8)+1;
                        end if;
                end if;

                -- Bits 20 downto 11
                if (seq_cnt(7 downto 0)=b"1_111_1111") and
                        (seq_cnt(10 downto 8)=b"100") then
                        if (seq_cnt(20 downto 11)=b"11_1011_1111") then
                                seq_cnt(20 downto 11)<=
                                        b"00_0000_0000";
                        else
                                seq_cnt(20 downto 11)<=
                                        seq_cnt(20 downto 11)+1;
                        end if;
                end if;

                -- Bits 36 downto 21
                if (seq_cnt(7 downto 0)=b"1_111_1111") and
                        (seq_cnt(10 downto 8)=b"100") and
                        (seq_cnt(20 downto 11)=b"11_1011_1111") then
                        if (seq_cnt(36 downto 21)=b"1111_1111_1111_1111") then
                                seq_cnt(36 downto 21)<=
```

```
                                        b"0000_0000_0000_0001";
                        else
                                seq_cnt(36 downto 21)<=
                                        seq_cnt(36 downto 21)+1;
                        end if;
                end if;

                -- Sequencer sync
                if (seq_cnt(7 downto 0)=b"1_111_1111") and
                        (seq_cnt(10 downto 8)=b"100") and
                        (seq_cnt(20 downto 11)=b"11_1011_1111") then
                        seq_sync<=b"1";
                else
                        seq_sync<=b"0";
                end if;
        end if;
end process;

-------------------------------------------------------------
-- Phase detector
--
-- The phase detector measures the number of clock cycles
-- between seq_sync and fm_sync, and sets the increment
-- accordingly.
--
-- phs_cnt is reset whenever either seq_sync or fm_sync is
-- seen.  phs_lst records the last value of phs_cnt.
--
-- phs_inc is a fixed-point number with a one bit integer
-- portion.  It is set to 1-phase_error, where phase_error is
-- the phs_lst (when phs_lst<=phs_cnt) or -phs_cnt (when
-- phs_lst>phs_cnt).  In the special case where the two syncs
-- occur simultaneously, phase_error is zero by definition, so
-- phs_inc is set to 1.
--
-- This algorithm works fine if the sync pulses alternates,
-- which is what happens when we are locked in to the
-- transmitter.  But if the transmitter is not on, we get a
-- lot of spurious fm_sync.
--
-- To handle this, we keep track in phs_cur of the number of
-- fm_sync seen in each seq_sync interval.  This number should
-- be nominally 1, but can occasionally be 0 or 2.  It should
-- never be 3, nor should there be consecutive intervals of
-- non-one values.  If an error is detected, phs_vld is
-- incremented.  If a normal interval count occurs, phs_vld is
-- decremented.  The phase increment is only set to a
-- non-unity value if phs_vld is zero.
-------------------------------------------------------------
process(clki)
begin
        if (rising_edge(clki)) then
                -- Count number of fm_sync pulses between
                -- seq_sync
                case (seq_sync&fm_sync) is
                        when b"01"  =>
                                if (phs_cur/=b"11") then
                                        phs_cur<=phs_cur+1;
                                end if;
                        when b"10"  =>
                                phs_cur<=b"00";
                        when b"11"  =>
                                phs_cur<=b"01";
                        when others =>
                                null;
                end case;

                -- Error checking
                if (seq_sync=b"1") then
                        if ((phs_cur=b"01") and
                                (phs_prv/=b"11")) or
                                ((phs_prv=b"01") and
                                (phs_cur/=b"11")) then
                                if (phs_vld/=b"00_0000") then
                                        phs_vld<=phs_vld-1;
                                end if;
                        else
```

```
                                if (phs_vld/=b"11_1111") then
                                        phs_vld<=phs_vld+1;
                                end if;
                        end if;
                        phs_prv<=phs_cur;
                end if;

                -- Interval count
                if (seq_sync=b"1") or (fm_sync=b"1") then
                        phs_lst<=phs_cnt;
                        phs_cnt<=b"00_0000_0000_0000_0000_00";
                else
                        phs_cnt<=phs_cnt+1;
                end if;

                -- Increment adjust
                if (seq_sync=b"1") then
                        if (fm_sync/=b"1") and
                                (phs_vld=b"00_0000") then
                                if (phs_lst>phs_cnt) then
                                        if (phs_cnt>b"00_1111_1111_1111_1111_11") then
                                                phs_inc<=
                                                        b"1_1111_1111_1111_1111_11";
                                        else
                                                phs_inc<=
                                                        b"1_0000_0000_0000_0000_00"+phs_cnt(18 downto 0);
                                        end if;
                                else
                                        if (phs_lst>b"00_1111_1111_1111_1111_11") then
                                                phs_inc<=
                                                        b"0_0000_0000_0000_0000_01";
                                        else
                                                phs_inc<=
                                                        b"1_0000_0000_0000_0000_00"-phs_lst(18 downto 0);
                                        end if;
                                end if;
                        else
                                phs_inc<=
                                        b"1_0000_0000_0000_0000_00";
                        end if;
                end if;
        end if;
end process;

-------------------------------------------------------------
-- RSSI ADC control
--
-- The code keeps fm_ncs low so that the ADC will output zeros
-- indefinetely.  Otherwise, when fm_ncs goes high, the output
-- becomes high-Z, and the line will float.
--
-- The maximum clock rate is 14.4Mhz, but there is no
-- convenient sequencer bits to achieve this.  The best we can
-- do is 9 or 10 MHz.  Since speed is not required, the chosen
-- clock rate is 1 MHz to keep the code simple.
--
-- Because seq_cnt(10 downto 7) only goes from 0 to 9, but we
-- need 12 cycles, the conversion must be initiated at the end
-- of the previous frame.  Fortunately, the null bits make
-- this easy to do.
--
-- 11_1011_1111__100__0__101_0010: fm_clk<=0, fm_ncs<=1
-- 11_1011_1111__100__0__111_1111: fm_clk<=1
--
-- 11_1011_1111__100__1__101_0010: fm_clk<=0, fm_ncs<=0
-- 11_1011_1111__100__1__111_1111: fm_clk<=1, fm_dat is null
--
-- 00_0000_0000__000__0__101_0010: fm_clk<=0
-- 00_0000_0000__000__0__111_1111: fm_clk<=1, fm_dat is null
--
-- 00_0000_0000__000__1__101_0010: fm_clk<=0
-- 00_0000_0000__000__1__111_1111: fm_clk<=1, fm_dat is null
--
-- 00_0000_0000__001__0__101_0010: fm_clk<=0
-- 00_0000_0000__001__0__111_1111: fm_clk<=1, rssi(7)<=fm_dat
--
-- 00_0000_0000__001__1__101_0010: fm_clk<=0
```

```
-- 00_0000_0000__001__1__111_1111: fm_clk<=1, rssi(6)<=fm_dat
--
-- 00_0000_0000__010__0__101_0010: fm_clk<=0
-- 00_0000_0000__010__0__111_1111: fm_clk<=1, rssi(5)<=fm_dat
--
-- 00_0000_0000__010__1__101_0010: fm_clk<=0
-- 00_0000_0000__010__1__111_1111: fm_clk<=1, rssi(4)<=fm_dat
--
-- 00_0000_0000__011__0__101_0010: fm_clk<=0
-- 00_0000_0000__011__0__111_1111: fm_clk<=1, rssi(3)<=fm_dat
--
-- 00_0000_0000__011__1__101_0010: fm_clk<=0
-- 00_0000_0000__011__1__111_1111: fm_clk<=1, rssi(2)<=fm_dat
--
-- 00_0000_0000__100__0__101_0010: fm_clk<=0
-- 00_0000_0000__100__0__111_1111: fm_clk<=1, rssi(1)<=fm_dat
--
-- 00_0000_0000__100__1__101_0010: fm_clk<=0
-- 00_0000_0000__100__1__111_1111: fm_clk<=1, rssi(0)<=fm_dat
-------------------------------------------------------------
process(clki)
begin
        if (rising_edge(clki)) then-- Maximum is 14.4MHz
                -- Update input latch
                fm_dat_d<=fm_dat;

                -- RSSI ADC clock
                if (seq_cnt(6 downto 0)=b"101_0010") then
                        fm_clk<='0';
                elsif (seq_cnt(6 downto 0)=b"111_1111") then
                        fm_clk<='1';
                end if;

                -- RSSI ADC chip select
                if (seq_cnt(6 downto 0)=b"101_0010") and
                        (seq_cnt(10 downto 8)=b"100") and
                        (seq_cnt(20 downto 11)=b"11_1011_1111") then
                        if (seq_cnt(7)='0') then
                                fm_ncs<='1';
                        else
                                fm_ncs<='0';
                        end if;
                end if;

                -- RSSI ADC data
                if (seq_cnt(6 downto 0)=b"111_1111") and
                        (seq_cnt(20 downto 11)=b"00_0000_0000") then
                        case (seq_cnt(10 downto 7)) is
                                when b"0010" =>
                                        rssi(7)<=fm_dat_d;
                                when b"0011" =>
                                        rssi(6)<=fm_dat_d;
                                when b"0100" =>
                                        rssi(5)<=fm_dat_d;
                                when b"0101" =>
                                        rssi(4)<=fm_dat_d;
                                when b"0110" =>
                                        rssi(3)<=fm_dat_d;
                                when b"0111" =>
                                        rssi(2)<=fm_dat_d;
                                when b"1000" =>
                                        rssi(1)<=fm_dat_d;
                                when b"1001" =>
                                        if (rssi(7 downto 1)=b"0000_000") then
                                                rssi(0)<=
                                                        '1';
                                        else
                                                rssi(0)<=
                                                        fm_dat_d;
                                        end if;
                                when others  =>
                                        null;
                        end case;
                end if;
        end if;
end process;
```

```
DAC_LOOKUP_40_GEN: if (module_id=US40) generate
        ----------------------------------------------------
        -- DAC table lookup
        --
        -- 40.000kHz differential sine output requires 25
        -- sample points, of which only 13 are unique.
        --
        -- dac_cnt  dac_phs
        -- 00_0000  0_0000
        -- 00_0001  0_0001
        -- 00_0010  0_0010
        -- 00_0011  0_0011
        -- 00_0100  0_0100
        -- 00_0101  0_0101
        -- 00_0110  0_0110
        -- 00_0111  0_0111
        -- 00_1000  0_1000
        -- 00_1001  0_1001
        -- 00_1010  0_1010
        -- 00_1011  0_1011
        -- 00_1100  0_1100
        -- 00_1101  1_1100
        -- 00_1110  1_1011
        -- 00_1111  1_1010
        -- 01_0000  1_1001
        -- 01_0001  1_1000
        -- 01_0010  1_0111
        -- 01_0011  1_0110
        -- 01_0100  1_0101
        -- 01_0101  1_0100
        -- 01_0110  1_0011
        -- 01_0111  1_0010
        -- 01_1000  1_0001
        ----------------------------------------------------
        process(clki)
        begin
                if (rising_edge(clki)) then
                        -- Sequence counter
                        if (seq_cnt(6 downto 0)=b"111_1100") then
                                if (dac_cnt(4 downto 0)=b"11000") or
                                        (seq_cnt(20 downto 7)=b"11_1011_1111_100_1") then
                                        dac_cnt(4 downto 0)<=b"00000";
                                else
                                        dac_cnt(4 downto 0)<=dac_cnt(4 downto 0)+1;
                                end if;
                        end if;

                        -- Table reduction phase adjustement
                        if (seq_cnt(6 downto 0)=b"111_1101") then
                                if (dac_cnt(4 downto 0)<b"01101") then
                                        dac_phs(3 downto 0)<=dac_cnt(3 downto 0);
                                        dac_phs(4)<='0';
                                else
                                        dac_phs(3 downto 0)<=b"1001"-dac_cnt(3 downto 0);
                                        dac_phs(4)<='1';
                                end if;
                        end if;

                        -- Sine lookup
                        if (seq_cnt(6 downto 0)=b"111_1110") then
                                case (dac_phs(3 downto 0)) is
                                        when b"0000" =>
                                                dac_dat<=b"1000_0000_0000_0000";
                                        when b"0001" =>
                                                dac_dat<=b"1001_1111_1101_0101";
                                        when b"0010" =>
                                                dac_dat<=b"1011_1101_1010_1010";
                                        when b"0011" =>
                                                dac_dat<=b"1101_0111_1001_1111";
                                        when b"0100" =>
                                                dac_dat<=b"1110_1100_0001_0010";
                                        when b"0101" =>
                                                dac_dat<=b"1111_1001_1011_1011";
                                        when b"0110" =>
                                                dac_dat<=b"1111_1111_1011_1110";
                                        when b"0111" =>
                                                dac_dat<=b"1111_1101_1011_1011";
```

```
                                        when b"1000" =>
                                                dac_dat<=b"1111_0011_1101_0000";
                                        when b"1001" =>
                                                dac_dat<=b"1110_0010_1001_1111";
                                        when b"1010" =>
                                                dac_dat<=b"1100_1011_0011_1100";
                                        when b"1011" =>
                                                dac_dat<=b"1010_1111_0001_1110";
                                        when others  =>
                                                dac_dat<=b"1001_0000_0000_1011";
                                end case;
                        end if;

                        -- Output assignment
                        if (seq_cnt(6 downto 0)=b"111_1111") then
                                if (dac_phs(4)='0') then
                                        dac_ndat<=b"0000_0000_0000_0000"-dac_dat;
                                        dac_pdat<=dac_dat;
                                else
                                        dac_ndat<=dac_dat;
                                        dac_pdat<=b"0000_0000_0000_0000"-dac_dat;
                                end if;
                        end if;
                end if;
        end process;
end generate;

DAC_LOOKUP_33_GEN: if (module_id=US33) generate
        ------------------------------------------------------
        -- DAC table lookup
        --
        -- 33.333kHz differential sine output requires 30
        -- sample points, of which only 8 are unique.
        --
        -- dac_cnt dac_phs
        -- 00_0000  0_000
        -- 00_0001  0_001
        -- 00_0010  0_010
        -- 00_0011  0_011
        -- 00_0100  0_100
        -- 00_0101  0_101
        -- 00_0110  0_110
        -- 00_0111  0_111
        -- 00_1000  0_111
        -- 00_1001  0_110
        -- 00_1010  0_101
        -- 00_1011  0_100
        -- 00_1100  0_011
        -- 00_1101  0_010
        -- 00_1110  0_001
        -- 01_0000  1_000
        -- 01_0001  1_001
        -- 01_0010  1_010
        -- 01_0011  1_011
        -- 01_0100  1_100
        -- 01_0101  1_101
        -- 01_0110  1_110
        -- 01_0111  1_111
        -- 01_1000  1_111
        -- 01_1001  1_110
        -- 01_1010  1_101
        -- 01_1011  1_100
        -- 01_1100  1_011
        -- 01_1101  1_010
        -- 01_1110  1_001
        ------------------------------------------------------
        process(clki)
        begin
                if (rising_edge(clki)) then
                        -- Sequence counter
                        if (seq_cnt(6 downto 0)=b"111_1100") then
                                if (seq_cnt(20 downto 7)=b"11_1011_1111_100_1") then
                                        dac_cnt(4 downto 0)<=b"00000";
                                elsif (dac_cnt(3 downto 0)=b"1110") then
                                        dac_cnt(3 downto 0)<=b"0000";
                                        dac_cnt(4 downto 4)<=dac_cnt(4 downto 4)+1;
                                else
```

```
                                        dac_cnt(3 downto 0)<=dac_cnt(3 downto 0)+1;
                                end if;
                        end if;

                        -- Table reduction phase adjustement
                        if (seq_cnt(6 downto 0)=b"111_1101") then
                                dac_phs(3)<=dac_cnt(4);
                                if (dac_cnt(3 downto 0)<b"1000") then
                                        dac_phs(2 downto 0)<=dac_cnt(2 downto 0);
                                else
                                        dac_phs(2 downto 0)<=b"111"-dac_cnt(2 downto 0);
                                end if;
                        end if;

                        -- Sine lookup
                        if (seq_cnt(6 downto 0)=b"111_1110") then
                                case (dac_phs(2 downto 0)) is
                                        when b"000" =>
                                                dac_dat<=b"1000_0000_0000_0000";
                                        when b"001" =>
                                                dac_dat<=b"1001_1010_1001_1101";
                                        when b"010" =>
                                                dac_dat<=b"1011_0100_0001_0000";
                                        when b"011" =>
                                                dac_dat<=b"1100_1011_0011_1100";
                                        when b"100" =>
                                                dac_dat<=b"1101_1111_0001_1111";
                                        when b"101" =>
                                                dac_dat<=b"1110_1110_1101_1001";
                                        when b"110" =>
                                                dac_dat<=b"1111_1001_1011_1011";
                                        when others =>
                                                dac_dat<=b"1111_1111_0100_1011";
                                end case;
                        end if;

                        -- Output assignment
                        if (seq_cnt(6 downto 0)=b"111_1111") then
                                if (dac_phs(3)='0') then
                                        dac_ndat<=b"0000_0000_0000_0000"-dac_dat;
                                        dac_pdat<=dac_dat;
                                else
                                        dac_ndat<=dac_dat;
                                        dac_pdat<=b"0000_0000_0000_0000"-dac_dat;
                                end if;
                        end if;
                end if;
        end process;
end generate;

DAC_LOOKUP_25_GEN: if (module_id=US25) generate
        ------------------------------------------------------
        -- DAC table lookup
        --
        -- 25.000kHz differential sine output requires 40
        -- sample points, of which only 11 are unique.
        --
        -- dac_cnt dac_phs
        -- 00_0000  0_0000
        -- 00_0001  0_0001
        -- 00_0010  0_0010
        -- 00_0011  0_0011
        -- 00_0100  0_0100
        -- 00_0101  0_0101
        -- 00_0110  0_0110
        -- 00_0111  0_0111
        -- 00_1000  0_1000
        -- 00_1001  0_1001
        -- 01_0000  0_1010
        -- 01_0001  0_1001
        -- 01_0010  0_1000
        -- 01_0011  0_0111
        -- 01_0100  0_0110
        -- 01_0101  0_0101
        -- 01_0110  0_0100
        -- 01_0111  0_0011
        -- 01_1000  0_0010
```

```
-- 01_1001  0_0001
-- 10_0000  1_0000
-- 10_0001  1_0001
-- 10_0010  1_0010
-- 10_0011  1_0011
-- 10_0100  1_0100
-- 10_0101  1_0101
-- 10_0110  1_0110
-- 10_0111  1_0111
-- 10_1000  1_1000
-- 10_1001  1_1001
-- 11_0000  1_1010
-- 11_0001  1_1001
-- 11_0010  1_1000
-- 11_0011  1_0111
-- 11_0100  1_0110
-- 11_0101  1_0101
-- 11_0110  1_0100
-- 11_0111  1_0011
-- 11_1000  1_0010
-- 11_1001  1_0001
------------------------------------------------------
process(clki)
begin
        if (rising_edge(clki)) then
                -- Sequence counter
                if (seq_cnt(6 downto 0)=b"111_1100") then
                        if (seq_cnt(20 downto 7)=b"11_1011_1111_100_1") then
                                dac_cnt(5 downto 0)<=b"000000";
                        elsif (dac_cnt(3 downto 0)=b"1001") then
                                dac_cnt(3 downto 0)<=b"0000";
                                dac_cnt(5 downto 4)<=dac_cnt(5 downto 4)+1;
                        else
                                dac_cnt(3 downto 0)<=dac_cnt(3 downto 0)+1;
                        end if;
                end if;

                -- Table reduction phase adjustement
                if (seq_cnt(6 downto 0)=b"111_1101") then
                        dac_phs(4)<=dac_cnt(5);
                        if (dac_cnt(4)='0') then
                                dac_phs(3 downto 0)<=dac_cnt(3 downto 0);
                        else
                                dac_phs(3 downto 0)<=b"1010"-dac_cnt(3 downto 0);
                        end if;
                end if;

                -- Sine lookup
                if (seq_cnt(6 downto 0)=b"111_1110") then
                        case (dac_phs(3 downto 0)) is
                                when b"0000" =>
                                        dac_dat<=b"1000_0000_0000_0000";
                                when b"0001" =>
                                        dac_dat<=b"1001_0100_0000_0110";
                                when b"0010" =>
                                        dac_dat<=b"1010_0111_1000_1110";
                                when b"0011" =>
                                        dac_dat<=b"1011_1010_0001_1100";
                                when b"0100" =>
                                        dac_dat<=b"1100_1011_0011_1100";
                                when b"0101" =>
                                        dac_dat<=b"1101_1010_1000_0010";
                                when b"0110" =>
                                        dac_dat<=b"1110_0111_1000_1101";
                                when b"0111" =>
                                        dac_dat<=b"1111_0010_0000_1100";
                                when b"1000" =>
                                        dac_dat<=b"1111_1001_1011_1011";
                                when b"1001" =>
                                        dac_dat<=b"1111_1110_0110_1100";
                                when others  =>
                                        dac_dat<=b"1111_1111_1111_1111";
                        end case;
                end if;

                -- Output assignment
                if (seq_cnt(6 downto 0)=b"111_1111") then
```

```
                                if (dac_phs(4)='0') then
                                        dac_ndat<=b"0000_0000_0000_0000"-dac_dat;
                                        dac_pdat<=dac_dat;
                                else
                                        dac_ndat<=dac_dat;
                                        dac_pdat<=b"0000_0000_0000_0000"-dac_dat;
                                end if;
                        end if;
                end if;
        end process;
end generate;

DAC_LOOKUP_NO_DAC: if (num_dac=b"00") generate
        dac_ndat<=b"1000_0000_0000_0000";
        dac_pdat<=b"1000_0000_0000_0000";
end generate;

-------------------------------------------------------------
-- DAC control
-------------------------------------------------------------
ndac_ncs<=dac_ncs when (num_dac/=b"00") else '1';
ndac_clk<=dac_clk when (num_dac/=b"00") else '0';
ndac_din<=dac_ndin when (num_dac/=b"00") else '0';
ndac_ncl<='1';
pdac_ncs<=dac_ncs when (num_dac/=b"00") else '1';
pdac_clk<=dac_clk when (num_dac/=b"00") else '0';
pdac_din<=dac_pdin when (num_dac/=b"00") else '0';
pdac_ncl<='1';
process(clki)
begin
        if (rising_edge(clki)) then
                case (seq_cnt(6 downto 0)) is
                        when b"101_1111" =>
                                dac_ncs<='0';
                                dac_ndin<=dac_ndat(15);
                                dac_pdin<=dac_pdat(15);
                        when b"110_0000" =>
                                dac_clk<='1';
                        when b"110_0001" =>
                                dac_clk<='0';
                                dac_ndin<=dac_ndat(14);
                                dac_pdin<=dac_pdat(14);
                        when b"110_0010" =>
                                dac_clk<='1';
                        when b"110_0011" =>
                                dac_clk<='0';
                                dac_ndin<=dac_ndat(13);
                                dac_pdin<=dac_pdat(13);
                        when b"110_0100" =>
                                dac_clk<='1';
                        when b"110_0101" =>
                                dac_clk<='0';
                                dac_ndin<=dac_ndat(12);
                                dac_pdin<=dac_pdat(12);
                        when b"110_0110" =>
                                dac_clk<='1';
                        when b"110_0111" =>
                                dac_clk<='0';
                                dac_ndin<=dac_ndat(11);
                                dac_pdin<=dac_pdat(11);
                        when b"110_1000" =>
                                dac_clk<='1';
                        when b"110_1001" =>
                                dac_clk<='0';
                                dac_ndin<=dac_ndat(10);
                                dac_pdin<=dac_pdat(10);
                        when b"110_1010" =>
                                dac_clk<='1';
                        when b"110_1011" =>
                                dac_clk<='0';
                                dac_ndin<=dac_ndat(9);
                                dac_pdin<=dac_pdat(9);
                        when b"110_1100" =>
                                dac_clk<='1';
                        when b"110_1101" =>
                                dac_clk<='0';
                                dac_ndin<=dac_ndat(8);
```

```
                dac_pdin<=dac_pdat(8);
            when b"110_1110" =>
                dac_clk<='1';
            when b"110_1111" =>
                dac_clk<='0';
                dac_ndin<=dac_ndat(7);
                dac_pdin<=dac_pdat(7);
            when b"111_0000" =>
                dac_clk<='1';
            when b"111_0001" =>
                dac_clk<='0';
                dac_ndin<=dac_ndat(6);
                dac_pdin<=dac_pdat(6);
            when b"111_0010" =>
                dac_clk<='1';
            when b"111_0011" =>
                dac_clk<='0';
                dac_ndin<=dac_ndat(5);
                dac_pdin<=dac_pdat(5);
            when b"111_0100" =>
                dac_clk<='1';
            when b"111_0101" =>
                dac_clk<='0';
                dac_ndin<=dac_ndat(4);
                dac_pdin<=dac_pdat(4);
            when b"111_0110" =>
                dac_clk<='1';
            when b"111_0111" =>
                dac_clk<='0';
                dac_ndin<=dac_ndat(3);
                dac_pdin<=dac_pdat(3);
            when b"111_1000" =>
                dac_clk<='1';
            when b"111_1001" =>
                dac_clk<='0';
                dac_ndin<=dac_ndat(2);
                dac_pdin<=dac_pdat(2);
            when b"111_1010" =>
                dac_clk<='1';
            when b"111_1011" =>
                dac_clk<='0';
                dac_ndin<=dac_ndat(1);
                dac_pdin<=dac_pdat(1);
            when b"111_1100" =>
                dac_clk<='1';
            when b"111_1101" =>
                dac_clk<='0';
                dac_ndin<=dac_ndat(0);
                dac_pdin<=dac_pdat(0);
            when b"111_1110" =>
                dac_clk<='1';
            when b"111_1111" =>
                dac_ncs<='1';
                dac_clk<='0';
            when others =>
                null;
        end case;
    end if;
end process;

-------------------------------------------------------------
-- PGA control
-------------------------------------------------------------
pga0_ncs<=pga_ncs when (num_adc>b"000") else '1';
pga0_dio<=pga_dio when (num_adc>b"000") else '0';
pga0_clk<=pga_clk when (num_adc>b"000") else '0';
pga1_ncs<=pga_ncs when (num_adc>b"001") else '1';
pga1_dio<=pga_dio when (num_adc>b"001") else '0';
pga1_clk<=pga_clk when (num_adc>b"001") else '0';
pga2_ncs<=pga_ncs when (num_adc>b"010") else '1';
pga2_dio<=pga_dio when (num_adc>b"010") else '0';
pga2_clk<=pga_clk when (num_adc>b"010") else '0';
pga3_ncs<=pga_ncs when (num_adc>b"011") else '1';
pga3_dio<=pga_dio when (num_adc>b"011") else '0';
pga3_clk<=pga_clk when (num_adc>b"011") else '0';
process(clki)
begin
```

```
if (rising_edge(clki)) then
        if (seq_cnt(7 downto 0)=b"0_010_1010") and
                (pga_lst/=pga(7 downto 0)) then
                pga_act<='1';
        elsif (seq_cnt(7 downto 0)=b"1_111_1111") and
                (pga_lst=pga(7 downto 0)) then
                pga_act<='0';
        end if;

        if (pga_act='1') then
                case (seq_cnt(7 downto 0)) is
                        when b"0_010_1111" =>
                                pga_ncs<='0';
                                pga_dio<=pga(15);
                        when b"0_011_0100" =>
                                pga_clk<='1';
                        when b"0_011_1001" =>
                                pga_clk<='0';
                                pga_dio<=pga(14);
                        when b"0_011_1110" =>
                                pga_clk<='1';
                        when b"0_100_0011" =>
                                pga_clk<='0';
                                pga_dio<=pga(13);
                        when b"0_100_1000" =>
                                pga_clk<='1';
                        when b"0_100_1101" =>
                                pga_clk<='0';
                                pga_dio<=pga(12);
                        when b"0_101_0010" =>
                                pga_clk<='1';
                        when b"0_101_0111" =>
                                pga_clk<='0';
                                pga_dio<=pga(11);
                        when b"0_101_1100" =>
                                pga_clk<='1';
                        when b"0_110_0001" =>
                                pga_clk<='0';
                                pga_dio<=pga(10);
                        when b"0_110_0110" =>
                                pga_clk<='1';
                        when b"0_110_1011" =>
                                pga_clk<='0';
                                pga_dio<=pga(9);
                        when b"0_111_0000" =>
                                pga_clk<='1';
                        when b"0_111_0101" =>
                                pga_clk<='0';
                                pga_dio<=pga(8);
                        when b"0_111_1010" =>
                                pga_clk<='1';
                        when b"0_111_1111" =>
                                pga_clk<='0';
                                pga_dio<=pga(7);
                        when b"1_010_1010" =>
                                pga_clk<='1';
                        when b"1_010_1111" =>
                                pga_clk<='0';
                                pga_dio<=pga(6);
                        when b"1_011_0100" =>
                                pga_clk<='1';
                        when b"1_011_1001" =>
                                pga_clk<='0';
                                pga_dio<=pga(5);
                        when b"1_011_1110" =>
                                pga_clk<='1';
                        when b"1_100_0011" =>
                                pga_clk<='0';
                                pga_dio<=pga(4);
                        when b"1_100_1000" =>
                                pga_clk<='1';
                        when b"1_100_1101" =>
                                pga_clk<='0';
                                pga_dio<=pga(3);
                        when b"1_101_0010" =>
                                pga_clk<='1';
                        when b"1_101_0111" =>
```

```
                                        pga_clk<='0';
                                        pga_dio<=pga(2);
                                    when b"1_101_1100" =>
                                        pga_clk<='1';
                                    when b"1_110_0001" =>
                                        pga_clk<='0';
                                        pga_dio<=pga(1);
                                    when b"1_110_0110" =>
                                        pga_clk<='1';
                                    when b"1_110_1011" =>
                                        pga_clk<='0';
                                        pga_dio<=pga(0);
                                    when b"1_111_0000" =>
                                        pga_clk<='1';
                                    when b"1_111_0101" =>
                                        pga_clk<='0';
                                    when b"1_111_1010" =>
                                        pga_ncs<='1';
                                        pga_lst<=pga(7 downto 0);
                                    when others =>
                                        null;
                            end case;
                    end if;
            end if;
    end process;

    ------------------------------------------------------------
    -- ADC control
    --
    -- ADC values of 0000_0000_0000_0000 are mapped to
    -- 0000_0000_0000_0001 to prevent all-zeros in the data
    -- stream.  The all-zeros is reserved for encoding the frame
    -- sync.
    --
    -- ADC control signals on the following seq_cnt(7 downto 0):
    --
    -- 1_101_1110 (1_94): adc_sck high
    -- 1_101_1111 (1_95): adc_sck low
    -- 1_110_0000 (1_96): adc_sck high, adc bit 15
    -- 1_110_0001 (1_97): adc_sck low
    -- 1_110_0010 (1_98): adc_sck high, adc bit 14
    -- 1_110_0011 (1_99): adc_sck low
    -- 1_110_0100 (1_100): adc_sck high, adc bit 13
    -- 1_110_0101 (1_101): adc_sck low
    -- 1_110_0110 (1_102): adc_sck high, adc bit 12
    -- 1_110_0111 (1_103): adc_sck low
    -- 1_110_1000 (1_104): adc_sck high, adc bit 11
    -- 1_110_1001 (1_105): adc_sck low
    -- 1_110_1010 (1_106): adc_sck high, adc bit 10
    -- 1_110_1011 (1_107): adc_sck low
    -- 1_110_1100 (1_108): adc_sck high, adc bit 9
    -- 1_110_1101 (1_109): adc_sck low
    -- 1_110_1110 (1_110): adc_sck high, adc bit 8
    -- 1_110_1111 (1_111): adc_sck low
    -- 1_111_0000 (1_112): adc_sck high, adc bit 7
    -- 1_111_0001 (1_113): adc_sck low
    -- 1_111_0010 (1_114): adc_sck high, adc bit 6
    -- 1_111_0011 (1_115): adc_sck low
    -- 1_111_0100 (1_116): adc_sck high, adc bit 5
    -- 1_111_0101 (1_117): adc_sck low
    -- 1_111_0110 (1_118): adc_sck high, adc bit 4
    -- 1_111_0111 (1_119): adc_sck low
    -- 1_111_1000 (1_120): adc_sck high, adc bit 3
    -- 1_111_1001 (1_121): adc_sck low
    -- 1_111_1010 (1_122): adc_sck high, adc bit 2
    -- 1_111_1011 (1_123): adc_sck low
    -- 1_111_1100 (1_124): adc_sck high, adc bit 1
    -- 1_111_1101 (1_125): adc_sck low, adc_cnv low
    -- 1_111_1110 (1_126): adc_sck high, adc bit 0
    -- 1_111_1111 (1_127): adc_cnv high
    ------------------------------------------------------------
    adc0_sdi<='0';
    adc0_sck<=adc_sck when (num_adc>b"000") else '1';
    adc0_cnv<=adc_cnv when (num_adc>b"000") else '1';
    adc1_sdi<='0';
    adc1_sck<=adc_sck when (num_adc>b"001") else '1';
    adc1_cnv<=adc_cnv when (num_adc>b"001") else '1';
```

```
adc2_sdi<='0';
adc2_sck<=adc_sck when (num_adc>b"010") else '1';
adc2_cnv<=adc_cnv when (num_adc>b"010") else '1';
adc3_sdi<='0';
adc3_sck<=adc_sck when (num_adc>b"011") else '1';
adc3_cnv<=adc_cnv when (num_adc>b"011") else '1';
process(clki)
begin
        if (rising_edge(clki)) then
                -- Update input latches
                if (num_adc>b"000") then
                        adc0_sdo_d<=adc0_sdo;
                else
                        adc0_sdo_d<='1';
                end if;
                if (num_adc>b"001") then
                        adc1_sdo_d<=adc1_sdo;
                else
                        adc1_sdo_d<='1';
                end if;
                if (num_adc>b"010") then
                        adc2_sdo_d<=adc2_sdo;
                else
                        adc2_sdo_d<='1';
                end if;
                if (num_adc>b"011") then
                        adc3_sdo_d<=adc3_sdo;
                else
                        adc3_sdo_d<='1';
                end if;

                case (seq_cnt(7 downto 0)) is
                        when b"1_101_1110" =>
                                adc_sck<='1';
                        when b"1_101_1111" =>
                                adc_sck<='0';
                        when b"1_110_0000" =>
                                adc_sck<='1';
                                adc0_in(15)<=adc0_sdo_d;
                                adc1_in(15)<=adc1_sdo_d;
                                adc2_in(15)<=adc2_sdo_d;
                                adc3_in(15)<=adc3_sdo_d;
                        when b"1_110_0001" =>
                                adc_sck<='0';
                        when b"1_110_0010" =>
                                adc_sck<='1';
                                adc0_in(14)<=adc0_sdo_d;
                                adc1_in(14)<=adc1_sdo_d;
                                adc2_in(14)<=adc2_sdo_d;
                                adc3_in(14)<=adc3_sdo_d;
                        when b"1_110_0011" =>
                                adc_sck<='0';
                        when b"1_110_0100" =>
                                adc_sck<='1';
                                adc0_in(13)<=adc0_sdo_d;
                                adc1_in(13)<=adc1_sdo_d;
                                adc2_in(13)<=adc2_sdo_d;
                                adc3_in(13)<=adc3_sdo_d;
                        when b"1_110_0101" =>
                                adc_sck<='0';
                        when b"1_110_0110" =>
                                adc_sck<='1';
                                adc0_in(12)<=adc0_sdo_d;
                                adc1_in(12)<=adc1_sdo_d;
                                adc2_in(12)<=adc2_sdo_d;
                                adc3_in(12)<=adc3_sdo_d;
                        when b"1_110_0111" =>
                                adc_sck<='0';
                        when b"1_110_1000" =>
                                adc_sck<='1';
                                adc0_in(11)<=adc0_sdo_d;
                                adc1_in(11)<=adc1_sdo_d;
                                adc2_in(11)<=adc2_sdo_d;
                                adc3_in(11)<=adc3_sdo_d;
                        when b"1_110_1001" =>
                                adc_sck<='0';
                        when b"1_110_1010" =>
```

```
            adc_sck<='1';
            adc0_in(10)<=adc0_sdo_d;
            adc1_in(10)<=adc1_sdo_d;
            adc2_in(10)<=adc2_sdo_d;
            adc3_in(10)<=adc3_sdo_d;
        when b"1_110_1011" =>
            adc_sck<='0';
        when b"1_110_1100" =>
            adc_sck<='1';
            adc0_in(9)<=adc0_sdo_d;
            adc1_in(9)<=adc1_sdo_d;
            adc2_in(9)<=adc2_sdo_d;
            adc3_in(9)<=adc3_sdo_d;
        when b"1_110_1101" =>
            adc_sck<='0';
        when b"1_110_1110" =>
            adc_sck<='1';
            adc0_in(8)<=adc0_sdo_d;
            adc1_in(8)<=adc1_sdo_d;
            adc2_in(8)<=adc2_sdo_d;
            adc3_in(8)<=adc3_sdo_d;
        when b"1_110_1111" =>
            adc_sck<='0';
        when b"1_111_0000" =>
            adc_sck<='1';
            adc0_in(7)<=adc0_sdo_d;
            adc1_in(7)<=adc1_sdo_d;
            adc2_in(7)<=adc2_sdo_d;
            adc3_in(7)<=adc3_sdo_d;
        when b"1_111_0001" =>
            adc_sck<='0';
        when b"1_111_0010" =>
            adc_sck<='1';
            adc0_in(6)<=adc0_sdo_d;
            adc1_in(6)<=adc1_sdo_d;
            adc2_in(6)<=adc2_sdo_d;
            adc3_in(6)<=adc3_sdo_d;
        when b"1_111_0011" =>
            adc_sck<='0';
        when b"1_111_0100" =>
            adc_sck<='1';
            adc0_in(5)<=adc0_sdo_d;
            adc1_in(5)<=adc1_sdo_d;
            adc2_in(5)<=adc2_sdo_d;
            adc3_in(5)<=adc3_sdo_d;
        when b"1_111_0101" =>
            adc_sck<='0';
        when b"1_111_0110" =>
            adc_sck<='1';
            adc0_in(4)<=adc0_sdo_d;
            adc1_in(4)<=adc1_sdo_d;
            adc2_in(4)<=adc2_sdo_d;
            adc3_in(4)<=adc3_sdo_d;
        when b"1_111_0111" =>
            adc_sck<='0';
        when b"1_111_1000" =>
            adc_sck<='1';
            adc0_in(3)<=adc0_sdo_d;
            adc1_in(3)<=adc1_sdo_d;
            adc2_in(3)<=adc2_sdo_d;
            adc3_in(3)<=adc3_sdo_d;
        when b"1_111_1001" =>
            adc_sck<='0';
        when b"1_111_1010" =>
            adc_sck<='1';
            adc0_in(2)<=adc0_sdo_d;
            adc1_in(2)<=adc1_sdo_d;
            adc2_in(2)<=adc2_sdo_d;
            adc3_in(2)<=adc3_sdo_d;
        when b"1_111_1011" =>
            adc_sck<='0';
        when b"1_111_1100" =>
            adc_sck<='1';
            adc0_in(1)<=adc0_sdo_d;
            adc1_in(1)<=adc1_sdo_d;
            adc2_in(1)<=adc2_sdo_d;
            adc3_in(1)<=adc3_sdo_d;
```

```
                        when b"1_111_1101" =>
                                adc_sck<='0';
                                adc_cnv<='0';
                        when b"1_111_1110" =>
                                if (adc0_in(15 downto 1)/=b"0000_0000_0000_000") then
                                        adc0_in(0)<=
                                                adc0_sdo_d;
                                else
                                        adc0_in(0)<=
                                                '1';
                                end if;
                                if (adc1_in(15 downto 1)/=b"0000_0000_0000_000") then
                                        adc1_in(0)<=
                                                adc1_sdo_d;
                                else
                                        adc1_in(0)<=
                                                '1';
                                end if;
                                if (adc2_in(15 downto 1)/=b"0000_0000_0000_000") then
                                        adc2_in(0)<=
                                                adc2_sdo_d;
                                else
                                        adc2_in(0)<=
                                                '1';
                                end if;
                                if (adc3_in(15 downto 1)/=b"0000_0000_0000_000") then
                                        adc3_in(0)<=
                                                adc3_sdo_d;
                                else
                                        adc3_in(0)<=
                                                '1';
                                end if;
                        when b"1_111_1111" =>
                                adc_cnv<='1';
                        when others =>
                                null;
                end case;
        end if;
end process;

-------------------------------------------------------------
-- ADC output to Opal Kelly
--
-- The raw ADC output is sent to the Opal Kelly via dedicated
-- pins. Since the four ADCs are sampled at 500ksps, the
-- output to the Opal Kelly must be updated at 2Msps.  The
-- state pins indicate which ADC value is being outputted
-- (gray code):
--
-- State 00: ADC 0
-- State 01: ADC 1
-- State 11: ADC 2
-- State 10: ADC 3
--
-- The ADC data registers are being updated from seq_cnt
-- 1_110_0000 (1_96) to 1_111_1110 (1_126), so the registers
-- must be read at other times, spaced 45 clock cycles apart.
-- The chosen times are:
--
-- 0_010_0111 (0_39)
-- 0_101_0100 (0_84)
-- 1_010_0111 (1_39)
-- 1_101_0100 (1_84)
-------------------------------------------------------------
process(clki)
begin
        if (rising_edge(clki)) then
                if (dacq_gen=dacq_gen2) then
                        case (seq_cnt(7 downto 0)) is
                                when b"0_010_0111" =>
                                        ok_adc<=
                                                std_logic_vector(adc0_in);
                                        ok_state<=b"00";
                                when b"0_101_0100" =>
                                        ok_adc<=
                                                std_logic_vector(adc1_in);
                                        ok_state<=b"01";
```

```
                                when b"1_010_0111" =>
                                        ok_adc<=
                                                std_logic_vector(adc2_in);
                                        ok_state<=b"11";
                                when b"1_101_0100" =>
                                        ok_adc<=
                                                std_logic_vector(adc3_in);
                                        ok_state<=b"10";
                                when others =>
                                        null;
                        end case;
                else
                        ok_adc<=b"1111_1111_1111_1111";
                        ok_state<=b"11";
                end if;
        end if;
end process;

-------------------------------------------------------------
-- ADC filtering
--
-- The filter output is:
--
-- y[n+1] = alpha * y[n] + (1-alpha) * x[n]
--
-- or equivalently:
--
-- y[n+1] = y[n] + (1-alpha) * (x[n]-y[n])
--
-- where alpha = exp(-2*pi*Fc/Fs).
--
-- Fs=500kHz, and Fc=12.5kHz, 16.667kHz, 25kHz or 50kHz.
--
-- The difference (x[n]-y[n]) goes from -65534 to +65534 (17
-- bits).  (1-alpha) is encoded as a fixed point number, with
-- one sign bit, no integer bit and 17 fractional bits.
--
-- The product (1-alpha) * (x[n]-y[n]) therefore also has 17
-- fractional bits.  Before truncating, a 1 followed by 16
-- zeros must be added to achieve rounding of the
-- result. Alternately, we can truncate the lower 16 bits, add
-- 1 (effectively 0.5), and truncate one more bit.
--
-- The computation is pipelined over the five clock cycles of
-- the two microsecond intervals.
--
-- The ADC data is available on seq_cnt(7 downto 0) = 1_127,
-- but we wait one extra cycle to start with seq_cnt(7) = 0.
-- Data is processed on the following seq_cnt(7 downto 0):
--
-- 0_010_0111 (0_39): Channel 0 data enters multiplier
-- 0_010_1000 (0_40): Channel 1 data enters multiplier
-- 0_010_1001 (0_41): Channel 0 data exits multiplier
--                    Channel 2 data enters multiplier
-- 0_010_1010 (0_42): Channel 1 data exits multiplier
--                    Channel 3 data enters multiplier
-- 0_010_1011 (0_43): Channel 2 data exits multiplier
-- 0_010_1100 (0_44): Channel 3 data exits multiplier
-------------------------------------------------------------
round<=unsigned(prod(32 downto 16))+1;
process(clki)
begin
        if (rising_edge(clki)) then
                case (seq_cnt(7 downto 0)) is
                        -- Filter with 2 clock cycle
                        -- propagation
                        when b"0_010_0111" =>
                                mult<=(b"00"&adc0_in)-(b"00"&filt0);
                        when b"0_010_1000" =>
                                mult<=(b"00"&adc1_in)-(b"00"&filt1);
                        when b"0_010_1001" =>
                                filt0<=filt0+round(16 downto 1);
                                mult<=(b"00"&adc2_in)-(b"00"&filt2);
                        when b"0_010_1010" =>
                                filt1<=filt1+round(16 downto 1);
                                mult<=(b"00"&adc3_in)-(b"00"&filt3);
                        when b"0_010_1011" =>
```

```
                                filt2<=filt2+round(16 downto 1);
                        when b"0_010_1100" =>
                                filt3<=filt3+round(16 downto 1);
                        when others =>
                                null;
                end case;
        end if;
end process;

-------------------------------------------------------------
-- RAM storage
--
-- Depending on the number of channels, we do one of the
-- following:
--
-- Store 1 sample every 10 microseconds
-- Store 2 samples every 20 microseconds
-- Store 3 samples every 30 microseconds
-- Store 4 samples every 40 microseconds
--
-- The storage sequence starts on seq_cnt(10 downto 0) =
-- 0_0_39, because values 0_0_0 to 0_0_36 are not used, and
-- 0_0_37 and 0_0_38 are not always used.
--
-- Data is stored in RAM on the following seq_cnt(10 downto
-- 0):
--
-- 000_0_010_0111 (0_0_39): Need new buffer
-- 000_0_010_1000 (0_0_40): Address  0: Sync
--                          (0000_0000_1111_1111)
-- 000_0_010_1001 (0_0_41): Address  1: Sync
--                          (0000_0000_0000_0000)
-- 000_0_010_1010 (0_0_42): Address  2: Module ID
-- 000_0_010_1011 (0_0_43): Address  3: Module ID
-- 000_0_010_1100 (0_0_44): Address  4: Frame index
-- 000_0_010_1101 (0_0_45): Address  5: Timestamp
-- 000_0_010_1110 (0_0_46): Address  6: Timestamp
-- 000_0_010_1111 (0_0_47): Address  7: Timestamp
-- 000_0_011_0000 (0_0_48): Address  8: Timestamp
-- 000_0_011_0001 (0_0_49): Address  9: PGA/RSSI
-- 000_0_011_0010 (0_0_50): Address seq_cnt(20 downto 11)+12:
--                          Channel 0 data
-- 000_0_011_0011 (0_0_51): Address seq_cnt(20 downto 11)+13:
--                          Channel 1 data
-- 000_0_011_0100 (0_0_52): Address seq_cnt(20 downto 11)+14:
--                          Channel 2 data
-- 000_0_011_0101 (0_0_53): Address seq_cnt(20 downto 11)+15:
--                          Channel 3 data
-- 000_0_011_0110 (0_0_54): Address 10: Minimum channel value
-- 000_0_011_0111 (0_0_55): Address 11: Maximum channel value
-- 000_0_011_1000 (0_0_56): RAM write ends
-------------------------------------------------------------
process(clki)
begin
        if (rising_edge(clki)) then
                -- Sequence counter
                if (seq_cnt(10 downto 0)=b"100_1_111_1111") then
                        if (num_adc=b"000") or
                                (ram_cnt=(num_adc-1)) or
                                (seq_cnt(20 downto 11)=b"11_1011_1111") then
                                ram_cnt<=b"000";
                        else
                                ram_cnt<=ram_cnt+1;
                        end if;
                end if;

                -- Store data
                if (ram_cnt=b"000") then
                        case (seq_cnt(10 downto 0)) is
                                when b"000_0_010_0111" =>
                                        -- Need new buffer
                                        if (seq_cnt(20 downto 11)=b"00_0000_0000") then
                                                ram_nnb<='1';
                                                ram_min<=b"1111_1111_1111_1111";
                                                ram_max<=b"0000_0000_0000_0000";
                                        end if;
                                when b"000_0_010_1000" =>
```

```
        -- Sync
        ram_nnb<='0';
        ram_wea<=b"11";
        ram_addr<=b"00_0000_0000";
        ram_data<=b"0000_0000_1111_1111";
when b"000_0_010_1001" =>
        -- Sync
        ram_addr<=b"00_0000_0001";
        ram_data<=b"0000_0000_0000_0000";
when b"000_0_010_1010" =>
        -- Module ID characters 2 and 1
        ram_addr<=b"00_0000_0010";
        ram_data<=module_id(23 downto 16)&module_id(31 downto 24);
when b"000_0_010_1011" =>
        -- Module ID characters 4 and 3
        ram_addr<=b"00_0000_0011";
        ram_data<=module_id(7 downto 0)&module_id(15 downto 8);
when b"000_0_010_1100" =>
        -- Frame index
        ram_addr<=b"00_0000_0100";
        ram_data<=seq_cnt(36 downto 21);
when b"000_0_010_1101" =>
        -- Timestamp
        ram_addr<=b"00_0000_0101";
        ram_data<=b"1"&timestamp(14 downto 0);
when b"000_0_010_1110" =>
        -- Timestamp
        ram_addr<=b"00_0000_0110";
        ram_data<=b"1"&timestamp(29 downto 15);
when b"000_0_010_1111" =>
        -- Timestamp
        ram_addr<=b"00_0000_0111";
        ram_data<=b"1"&timestamp(44 downto 30);
when b"000_0_011_0000" =>
        -- Timestamp
        ram_addr<=b"00_0000_1000";
        ram_data<=b"1"&timestamp(59 downto 45);
when b"000_0_011_0001" =>
        -- PGA/RSSI
        ram_addr<=b"00_0000_1001";
        ram_data<=pga_lst&rssi;
when b"000_0_011_0010" =>
        -- Channel 0 data
        if (num_adc>b"000") then
                ram_addr<=seq_cnt(20 downto 11)+12;
                ram_data<=filt0;
                if (ram_min>filt0) then
                        ram_min<=filt0;
                end if;
                if (ram_max<filt0) then
                        ram_max<=filt0;
                end if;
        else
                ram_addr<=seq_cnt(20 downto 11)+12;
                ram_data<=fm_bits;
                if (ram_min>fm_bits) then
                        ram_min<=fm_bits;
                end if;
                if (ram_max<fm_bits) then
                        ram_max<=fm_bits;
                end if;
        end if;
when b"000_0_011_0011" =>
        -- Channel 1 data
        if (num_adc>b"001") then
                ram_addr<=seq_cnt(20 downto 11)+13;
                ram_data<=filt1;
                if (ram_min>filt1) then
                        ram_min<=filt1;
                end if;
                if (ram_max<filt1) then
                        ram_max<=filt1;
                end if;
        end if;
when b"000_0_011_0100" =>
        -- Channel 2 data
        if (num_adc>b"010") then
```

```
                                        ram_addr<=seq_cnt(20 downto 11)+14;
                                        ram_data<=filt2;
                                        if (ram_min>filt2) then
                                                ram_min<=filt2;
                                        end if;
                                        if (ram_max<filt2) then
                                                ram_max<=filt2;
                                        end if;
                                end if;
                        when b"000_0_011_0101" =>
                                -- Channel 3 data
                                if (num_adc>b"011") then
                                        ram_addr<=seq_cnt(20 downto 11)+15;
                                        ram_data<=filt3;
                                        if (ram_min>filt3) then
                                                ram_min<=filt3;
                                        end if;
                                        if (ram_max<filt3) then
                                                ram_max<=filt3;
                                        end if;
                                end if;
                        when b"000_0_011_0110" =>
                                -- Channel minimum value
                                ram_addr<=b"00_0000_1010";
                                ram_data<=ram_min;
                        when b"000_0_011_0111" =>
                                -- Channel maximum value
                                ram_addr<=b"00_0000_1011";
                                ram_data<=ram_max;
                        when b"000_0_011_1000" =>
                                ram_wea<=b"00";
                        when others =>
                                null;
                end case;
        end if;
end if;
end process;

-------------------------------------------------------------
-- Transmit serial data
-------------------------------------------------------------
process(clki)
begin
        if (rising_edge(clki)) then
                ---------------------------------------------
                -- Sample input signals
                ---------------------------------------------
                dtr_d<=ft_dtr;
                rts_d<=ft_rts;

                ---------------------------------------------
                -- Serial reset is asserted when DTR goes
                -- high, and deasserted when both DTR and RTS
                -- are low.
                ---------------------------------------------
                if (dtr_d='0') and (rts_d='0') then
                        ser_rst<='0';
                elsif (dtr_d='1') then
                        ser_rst<='1';
                end if;

                ---------------------------------------------
                -- ser_cyc counts from 0 to 29.
                -- ser_bit counts from 0 to 9.
                -- ser_cyc and ser_bit hold at 29 and 9,
                -- respectively.
                ---------------------------------------------
                if (ser_cyc/=b"11101") then
                        ser_cyc<=ser_cyc+1;
                elsif (ser_bit/=b"1001") then
                        ser_cyc<=b"00000";
                        ser_bit<=ser_bit+1;
                elsif (ser_rst='0') and
                        (ctrl_ser/=b"11") and
                        (rts_d='0') and
                        (dtr_d='0') then
                        ser_cyc<=b"00000";
```

```
                ser_bit<=b"0000";
            end if;

            ------------------------------------------
            -- ser_addr counts from 0 to 1943,
            -- incrementing every time ser_cyc and ser_bit
            -- reach 27 and 9 (respectively).
            --
            -- It takes one clock cycle for the memory
            -- output to update, and one more clock cycle
            -- for ser_out to update.  So ser_addr is
            -- updated 2 clock cycles before ser_cyc and
            -- ser_bit roll over.
            ------------------------------------------
            if (ser_cyc=b"11101") and
                (ser_bit=b"1001") and
                (ser_rst='1') then
                ser_addr<=b"000_0000_0000";
            elsif (ser_cyc=b"11011") and
                (ser_bit=b"1001") then
                if (ser_addr=b"111_1001_0111") then
                    ser_addr<=b"000_0000_0000";
                else
                    ser_addr<=ser_addr+1;
                end if;
            end if;

            ------------------------------------------
            -- ser_out gets the data from the memory one
            -- clock at the beginning of the serial cycle.
            ------------------------------------------
            if (ser_cyc=b"00000") and
                (ser_bit=b"0000") then
                ser_out<=ctrl_out;
            end if;

            ------------------------------------------
            -- ser_nnb activates for one clock cycle and
            -- one clock cycle before the first read from
            -- the memory buffer.  ser_nnb also activates
            -- on the falling edge of ser_rst.
            ------------------------------------------
            if ((dtr_d='0') and
                (rts_d='0') and
                (ser_rst='1')) or
                ((ser_cyc=b"11011") and
                (ser_bit=b"1001") and
                (ser_addr=b"111_1001_0111")) then
                ser_nnb<="1";
            else
                ser_nnb<="0";
            end if;

            ------------------------------------------
            -- ft_rxd is the RS-232 serial output
            ------------------------------------------
            case (ser_bit) is
                when b"0000" => ft_rxd<='0';
                when b"0001" => ft_rxd<=ser_out(0);
                when b"0010" => ft_rxd<=ser_out(1);
                when b"0011" => ft_rxd<=ser_out(2);
                when b"0100" => ft_rxd<=ser_out(3);
                when b"0101" => ft_rxd<=ser_out(4);
                when b"0110" => ft_rxd<=ser_out(5);
                when b"0111" => ft_rxd<=ser_out(6);
                when b"1000" => ft_rxd<=ser_out(7);
                when others  => ft_rxd<='1';
            end case;
        end if;
    end process;

    ------------------------------------------------------------
    -- Buffer control
    --
    -- ctrl_ser encodes the buffer being used by the serial port
    -- process:
    --
```

```
--     00: buf 0
--     01: buf 1
--     10: buf 2
--     11: starved
--
-- ctrl_gen encodes the buffer being used by the data
-- generator process:
--
--     00: buf 0
--     01: buf 1
--     10: buf 2
--
-- ctrl_spr encodes whether or not the spare buffer is full:
--
--     0: spare is empty
--     1: spare is full
--
-- If the data generator needs a new buffer:
--     give it the next buffer (mod 3) unless it is being used
--         by the serial port,
--     if the serial port is starving, give it the buffer just
--         filled, otherwise set the full flag.
--
-- If the serial port needs a new buffer:
--     if the spare buffer is emtpy, set the starved state
--         otherwise give it the buffer not being used by the
--         data generator.
-----------------------------------------------------------------
ctrl_out<=ctrl_data(7 downto 0) when (ctrl_ser=b"00") else
        ctrl_data(15 downto 8) when (ctrl_ser=b"01") else
        ctrl_data(23 downto 16) when (ctrl_ser=b"10") else
        b"0000_0000";
with ctrl_gen select
        ram_wen<="001" when b"00",
                 "010" when b"01",
                 "100" when b"10",
                 "000" when others;

process(clki)
begin
        if (rising_edge(clki)) then
                case (ser_nnb&ram_nnb) is
                        when b"01" => -- Data generator needs a new buffer
                                if (ctrl_gen/=b"11") then
                                        if (ctrl_ser/=nxt_buf(ctrl_gen)) then
                                                ctrl_gen<=nxt_buf(ctrl_gen);
                                        else
                                                ctrl_gen<=prv_buf(ctrl_gen);
                                        end if;
                                        if (ctrl_ser=b"11") then
                                                ctrl_ser<=ctrl_gen;
                                                ctrl_spr<='0';
                                        else
                                                ctrl_spr<='1';
                                        end if;
                                else -- Something went wrong so reset to sensible values
                                        if (ctrl_ser/=b"00") then
                                                ctrl_gen<=b"00";
                                        else
                                                ctrl_gen<=b"01";
                                        end if;
                                        ctrl_spr<='0';
                                end if;
                        when b"10" => -- Serial port needs a new buffer
                                if (ctrl_ser/=b"11") then
                                        if (ctrl_spr='0') then
                                                ctrl_ser<=b"11";
                                        else
                                                if (ctrl_gen/=nxt_buf(ctrl_ser)) then
                                                        ctrl_ser<=nxt_buf(ctrl_ser);
                                                else
                                                        ctrl_ser<=prv_buf(ctrl_ser);
                                                end if;
                                        end if;
                                end if;
                                ctrl_spr<='0';
                        when b"11" => -- Data generator and serial port need new buffers
```

```
                        if (ctrl_gen/=b"11") then
                                if (ctrl_ser/=b"11") then
                                        if (ctrl_spr='0') then
                                                ctrl_gen<=ctrl_ser;
                                                ctrl_ser<=ctrl_gen;
                                        else
                                                ctrl_gen<=ctrl_ser;
                                                if (ctrl_gen/=nxt_buf(ctrl_ser)) then
                                                        ctrl_ser<=nxt_buf(ctrl_ser);
                                                else
                                                        ctrl_ser<=prv_buf(ctrl_ser);
                                                end if;
                                        end if;
                                else
                                        ctrl_gen<=nxt_buf(ctrl_gen);
                                        ctrl_ser<=ctrl_gen;
                                        ctrl_spr<='0';
                                end if;
                        else -- Something went wrong so reset to sensible values
                                ctrl_gen<=b"00";
                                ctrl_ser<=b"11";
                                ctrl_spr<='0';
                        end if;
                when others =>
                        null;
        end case;
    end if;
end process;

-------------------------------------------------------------
-- Receive serial data
-------------------------------------------------------------
ft_cts<='0';
ft_dsr<='0';
ft_dcd<='0';
ft_ri<='1';
process(clki)
begin
    if (rising_edge(clki)) then
        -- Update counter
        if (ser_cnt/=b"1001_01110") then
            if (ser_cnt(4 downto 0)=b"11101") then
                ser_cnt(4 downto 0)<=
                    b"00000";
                ser_cnt(8 downto 5)<=
                    ser_cnt(8 downto 5)+1;
            else
                ser_cnt(4 downto 0)<=
                    ser_cnt(4 downto 0)+1;
            end if;
        elsif (txd_d='0') then
            ser_cnt<=b"0000_00000";
        end if;

        -- Collect serial data
        if (ser_cnt(4 downto 0)=b"01110") then
            case (ser_cnt(8 downto 5)) is
                when b"0001" =>
                    ser_in(0)<=txd_d;
                when b"0010" =>
                    ser_in(1)<=txd_d;
                when b"0011" =>
                    ser_in(2)<=txd_d;
                when b"0100" =>
                    ser_in(3)<=txd_d;
                when b"0101" =>
                    ser_in(4)<=txd_d;
                when b"0110" =>
                    ser_in(5)<=txd_d;
                when b"0111" =>
                    ser_in(6)<=txd_d;
                when b"1000" =>
                    pga<=b"0010_1010"&
                        txd_d&ser_in;
                when others =>
                    null;
            end case;
```

```
                    end if;

                    -- Update latch
                    txd_d<=ft_txd;
                end if;
        end process;
end arch;
```

The fourth script given is the UCF file for DACQ units.
It is the physical mapping between the pins of the FPGA and the signals used in the previous VHDL script.

```
##########################################################
#                                                        #
# XC3S50ATQ144 Spartan3AN with 3 blocks of 18kbits RAM   #
#                                                        #
##########################################################

# Last updated: 2012-11-23

# Timing constraints for 75MHz oscillator
NET xtal CLOCK_DEDICATED_ROUTE=FALSE;
NET xtal PERIOD=12.5ns;#13.3ns maximum

# Timing constraints for internal 90MHz clock
NET clki PERIOD=10.5ns;#11.1ns maximum

# Pin assignements
#NET TMS          LOC=P1;    #TMS
#NET TDI          LOC=P2;    #TDI
NET fm_cmp        LOC=P3;    #IO_3,XBUS-6
NET fm_dat        LOC=P4;    #IO_4,XBUS-8
NET pdac_ncs      LOC=P5;    #IO_5,XBUS-10
NET ok_adc<0>     LOC=P6;    #IO_6,XBUS-1
NET ok_adc<1>     LOC=P7;    #IO_7,XBUS-3
NET ok_adc<2>     LOC=P8;    #IO_8,XBUS-5
#NET GND          LOC=P9;    #GND
NET ok_adc<3>     LOC=P10;   #IO_10,XBUS-7
NET ok_adc<4>     LOC=P11;   #IO_11,XBUS-9
NET ok_adc<5>     LOC=P12;   #IO_12,XBUS-11
NET ok_adc<6>     LOC=P13;   #IO_13,XBUS-12,XBUS-13
#NET 3V3          LOC=P14;   #3V3
NET ok_adc<7>     LOC=P15;   #IO_15,XBUS-14,XBUS-15
NET ok_adc<8>     LOC=P16;   #IO_16,XBUS-16,XBUS-17
#NET GND          LOC=P17;   #GND
NET ok_adc<9>     LOC=P18;   #IO_18,XBUS-18,XBUS-19
NET pdac_clk      LOC=P19;   #IO_19,XBUS-20,XBUS-21
NET pdac_din      LOC=P20;   #IO_20,XBUS-22,XBUS-23
NET pdac_ncl      LOC=P21;   #IO_21,XBUS-24,XBUS-25
#NET 1V2          LOC=P22;   #1V2
#NET 3V3          LOC=P23;   #3V3
NET ndac_ncs      LOC=P24;   #IO_24,XBUS-26,XBUS-27
NET ndac_clk      LOC=P25;   #IO_25,XBUS-28,XBUS-29
#NET GND          LOC=P26;   #GND
NET ndac_din      LOC=P27;   #IO_27,XBUS-30,XBUS-31
NET ndac_ncl      LOC=P28;   #IO_28,XBUS-32,XBUS-33
NET pga3_ncs      LOC=P29;   #IO_29,XBUS-34,XBUS-35
NET pga3_dio      LOC=P30;   #IO_30,XBUS-36,XBUS-37
NET ok_adc<10>    LOC=P31;   #IO_31,XBUS-38,XBUS-39
NET ok_adc<11>    LOC=P32;   #IO_32,XBUS-40,XBUS-41
NET ok_ctrl<1>    LOC=P33;   #IP_33,XBUS-42,XBUS-43
#NET GND          LOC=P34;   #GND
NET ok_ctrl<2>    LOC=P35;   #IP_35,XBUS-45
#NET 3V3          LOC=P36;   #3V3
#NET 3V3          LOC=P37;   #M1
#NET 3V3          LOC=P38;   #M0
#NET GND          LOC=P39;   #M2
#NET 3V3          LOC=P40;   #3V3
NET ok_adc<12>    LOC=P41;   #IO_41,XBUS-47
NET ok_adc<13>    LOC=P42;   #IO_42,XBUS-49
#NET 3V3          LOC=P43;   #VS2
#NET 3V3          LOC=P44;   #VS1
#NET 3V3          LOC=P45;   #VS0
NET ok_adc<14>    LOC=P46;   #IO_46,XBUS-51
NET ok_adc<15>    LOC=P47;   #IO_47,XBUS-53
```

```
NET ok_state<0>  LOC=P48;   #IO_48,XBUS-55
NET ok_state<1>  LOC=P49;   #IO_49,XBUS-57
NET pga3_clk     LOC=P50;   #IO_50,XBUS-44
NET adc3_sdi     LOC=P51;   #IO_51,XBUS-46
#NET 1V2         LOC=P52;   #1V2
NET ok_ctrl<3>   LOC=P53;   #IP_53,XBUS-48
NET adc3_sck     LOC=P54;   #IO_GCLK_54,XBUS-50
NET adc3_cnv     LOC=P55;   #IO_GCLK_55,XBUS-52
#NET GND         LOC=P56;   #GND
NET adc3_sdo     LOC=P57;   #IO_GCLK_57,XBUS-54
NET pga2_ncs     LOC=P58;   #IO_GCLK_58,XBUS-56
NET pga2_dio     LOC=P59;   #IO_GCLK_59,YBUS-57
NET xtal         LOC=P60;   #IO_GCLK_60,XTAL
#NET 3V3         LOC=P61;   #3V3
NET pga2_clk     LOC=P62;   #IO_62,YBUS-55
NET adc2_sdi     LOC=P63;   #IO_63,YBUS-53
NET adc2_sck     LOC=P64;   #IO_64,YBUS-51
#NET GND         LOC=P65;   #GND
#NET 3V3         LOC=P66;   #3V3
#NET NC          LOC=P67;   #INIT_B
NET ok_ctrl<8>   LOC=P68;   #IO_68,YBUS-49
NET adc2_cnv     LOC=P69;   #IO_69,YBUS-47
NET adc2_sdo     LOC=P70;   #IO_70,YBUS-45
NET ok_ctrl<7>   LOC=P71;   #IO_71,YBUS-56
NET ok_ctrl<6>   LOC=P72;   #IO_72,YBUS-54
#NET NC          LOC=P73;   #DONE
#NET GND         LOC=P74;   #SUSPEND
NET ok_fm        LOC=P75;   #IO_75,YBUS-52
NET ok_dac<15>   LOC=P76;   #IO_76,YBUS-50
NET ok_dac<14>   LOC=P77;   #IO_77,YBUS-48
NET ok_dac<13>   LOC=P78;   #IO_78,YBUS-46
NET ok_dac<12>   LOC=P79;   #IO_79,YBUS-44
NET ok_ctrl<5>   LOC=P80;   #IP_80,YBUS-42,YBUS-43
#NET GND         LOC=P81;   #GND
NET ok_dac<11>   LOC=P82;   #IO_82,YBUS-40,YBUS-41
NET ok_dac<10>   LOC=P83;   #IO_83,YBUS-38,YBUS-39
NET pga1_ncs     LOC=P84;   #IO_84,YBUS-36,YBUS-37
NET pga1_dio     LOC=P85;   #IO_85,YBUS-34,YBUS-35
#NET 3V3         LOC=P86;   #3V3
NET pga1_clk     LOC=P87;   #IO_87,YBUS-32,YBUS-33
NET adc1_sdi     LOC=P88;   #IO_88,YBUS-30,YBUS-31
#NET GND         LOC=P89;   #GND
NET adc1_sck     LOC=P90;   #IO_90,YBUS-28,YBUS-29
NET adc1_cnv     LOC=P91;   #IO_91,YBUS-26,YBUS-27
NET adc1_sdo     LOC=P92;   #IO_92,YBUS-24,YBUS-25
NET pga0_ncs     LOC=P93;   #IO_93,YBUS-22,YBUS-23
#NET 1V2         LOC=P94;   #1V2
#NET 3V3         LOC=P95;   #3V3
NET pga0_dio     LOC=P96;   #IO_96,YBUS-20,YBUS-21
NET ok_ctrl<4>   LOC=P97;   #IP_97,YBUS-18,YBUS-19
NET ok_dac<9>    LOC=P98;   #IO_98,YBUS-16,YBUS-17
NET ok_dac<8>    LOC=P99;   #IO_99,YBUS-14,YBUS-15
#NET GND         LOC=P100;  #GND
NET ok_dac<7>    LOC=P101;  #IO_101,YBUS-12,YBUS-13
NET ok_dac<6>    LOC=P102;  #IO_102,YBUS-10
NET ok_dac<5>    LOC=P103;  #IO_103,YBUS-8
NET ok_dac<4>    LOC=P104;  #IO_104,YBUS-6
NET ok_dac<3>    LOC=P105;  #IO_105,YBUS-4
#NET GND         LOC=P106;  #GND
#NET TDO         LOC=P107;  #TDO
#NET 3V3         LOC=P108;  #3V3
#NET TCK         LOC=P109;  #TCK
NET ok_dac<2>    LOC=P110;  #IO_110,YBUS-2
NET ok_dac<0>    LOC=P111;  #IO_111,YBUS-0
NET pga0_clk     LOC=P112;  #IO_112,YBUS-11
NET adc0_sdi     LOC=P113;  #IO_113,YBUS-9
NET adc0_sck     LOC=P114;  #IO_114,YBUS-7
NET adc0_cnv     LOC=P115;  #IO_115,YBUS-5
NET adc0_sdo     LOC=P116;  #IO_116,YBUS-3
NET ok_dac<1>    LOC=P117;  #IO_117,YBUS-1
#NET GND         LOC=P118;  #GND
#NET 3V3         LOC=P119;  #3V3
#NET NC          LOC=P120;  #IO_120
#NET NC          LOC=P121;  #IO_121
#NET 1V2         LOC=P122;  #1V2
NET ft_cbus3     LOC=P123;  #IP_123,CBUS3
NET ft_cbus2     LOC=P124;  #IO_GCLK_124,CBUS2
```

```
NET ft_cts       LOC=P125;  #IO_GCLK_125,
NET ft_dcd       LOC=P126;  #IO_GCLK_126,
NET ft_dsr       LOC=P127;  #IO_GCLK_127,
#NET GND         LOC=P128;  #GND
NET ft_ri        LOC=P129;  #IO_GCLK_129,
NET ft_rts       LOC=P130;  #IO_GCLK_130,
NET ft_dtr       LOC=P131;  #IO_GCLK_131,
NET ft_txd       LOC=P132;  #IO_GCLK_132,TXD
#NET 3V3         LOC=P133;  #3V3
NET ft_rxd       LOC=P134;  #IO_134,RXD
NET ft_cbus1     LOC=P135;  #IO_135,CBUS1
#NET 3V3         LOC=P136;  #3V3
#NET GND         LOC=P137;  #GND
NET ft_cbus0     LOC=P138;  #IO_138,CBUS0
NET fm_lo        LOC=P139;  #IO_139
NET ok_ctrl<0>   LOC=P140;  #IP_140,XBUS-0
NET fm_ncs       LOC=P141;  #IO_141,XBUS-2
NET fm_clk       LOC=P142;  #IO_142,XBUS-4
#NET GND         LOC=P143;  #PUDC_B
#NET 3V3         LOC=P144;  #PROG_B
```

The fifth VHDL script given is used to tune the FM receiver of the the DACQ units.
After declaration and initialization of all signals and variables, the first component instantiated is the DCM to create the necessary clocks from the main crystal frequency. Next, follows the FM receiver process, which is used to detect the raw bit of the timestamp (5μs period) and adjust the clock period if local clock is too fast or too slow. Then, there is a process called statistics gathering that used to evaluate how often the 5μs period is out of sync (too fast or too slow). Then, a process sends those statistics to the computer.

```
----------------------------------------------------------------------
-- Design Name: fm_tune
-- Module Name: fm_tune (top level for implementation)
-- File name: fm_tune.vhd
----------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
library UNISIM;-- For Xilinx primitives
use UNISIM.VComponents.all;

entity fm_tune is
    port(
        -- serial interface
        txd:      in  std_logic;
        rxd:      out std_logic;
        rts:      in  std_logic;
        cts:      out std_logic;
        dtr:      in  std_logic;
        dsr:      out std_logic;
        dcd:      out std_logic;
        ri:       out std_logic;
        cbus0:    in  std_logic;
        cbus1:    in  std_logic;
        cbus2:    in  std_logic;
        cbus3:    in  std_logic;
--        cbus4:    in  std_logic;-- rcvr.ucf only

        -- DAC interface
        ndac_ncs: out std_logic;-- dacq.ucf only
        ndac_clk: out std_logic;-- dacq.ucf only
        ndac_din: out std_logic;-- dacq.ucf only
        ndac_ncl: out std_logic;-- dacq.ucf only

        pdac_ncs: out std_logic;-- dacq.ucf only
        pdac_clk: out std_logic;-- dacq.ucf only
        pdac_din: out std_logic;-- dacq.ucf only
        pdac_ncl: out std_logic;-- dacq.ucf only

        -- ADC interface
```

```
        pga0_ncs: out std_logic;-- dacq.ucf only
        pga0_dio: out std_logic;-- dacq.ucf only
        pga0_clk: out std_logic;-- dacq.ucf only
        adc0_sdi: out std_logic;-- dacq.ucf only
        adc0_sck: out std_logic;-- dacq.ucf only
        adc0_cnv: out std_logic;-- dacq.ucf only
        adc0_sdo: in  std_logic;-- dacq.ucf only

        pga1_ncs: out std_logic;-- dacq.ucf only
        pga1_dio: out std_logic;-- dacq.ucf only
        pga1_clk: out std_logic;-- dacq.ucf only
        adc1_sdi: out std_logic;-- dacq.ucf only
        adc1_sck: out std_logic;-- dacq.ucf only
        adc1_cnv: out std_logic;-- dacq.ucf only
        adc1_sdo: in  std_logic;-- dacq.ucf only

        pga2_ncs: out std_logic;-- dacq.ucf only
        pga2_dio: out std_logic;-- dacq.ucf only
        pga2_clk: out std_logic;-- dacq.ucf only
        adc2_sdi: out std_logic;-- dacq.ucf only
        adc2_sck: out std_logic;-- dacq.ucf only
        adc2_cnv: out std_logic;-- dacq.ucf only
        adc2_sdo: in  std_logic;-- dacq.ucf only

        pga3_ncs: out std_logic;-- dacq.ucf only
        pga3_dio: out std_logic;-- dacq.ucf only
        pga3_clk: out std_logic;-- dacq.ucf only
        adc3_sdi: out std_logic;-- dacq.ucf only
        adc3_sck: out std_logic;-- dacq.ucf only
        adc3_cnv: out std_logic;-- dacq.ucf only
        adc3_sdo: in  std_logic;-- dacq.ucf only

        -- FM receiver
        fm_ncs:   out std_logic;
        fm_clk:   out std_logic;
        fm_cmp:   in  std_logic;
        fm_dat:   in  std_logic;
        fm_lo:    out std_logic;

        -- Digital I/O
--        rng1:     out std_logic;-- rcvr.ucf only
--        rng2:     out std_logic;-- rcvr.ucf only

        -- Clock input
--        xtlen:    out std_logic;-- rcvr.ucf only
        xtal:     in  std_logic
    );
end fm_tune;

architecture arch of fm_tune is
    ---------------------------------------------------------------
    -- Local signals for DCM
    ---------------------------------------------------------------
    signal gclk:   std_logic;
    signal gclk0:  std_logic;
    signal clk:    std_logic;
    signal clkfx:  std_logic;
    signal clki0:  std_logic;
    signal clkifb: std_logic;
    signal clkifx: std_logic;
    signal clki:   std_logic;
    ---------------------------------------------------------------
    -- Local signals for FM receiver
    ---------------------------------------------------------------
    signal fm_smp: std_logic;
    signal cnt_5us: unsigned(8 downto 0);
    signal cnt_ones: unsigned(8 downto 0);
    ---------------------------------------------------------------
    -- Local signals for statistics gathering
    ---------------------------------------------------------------
    signal ones_cnt: unsigned(7 downto 0);
    signal ones_avg: unsigned(8 downto 0);
    signal ones_tmp: unsigned(16 downto 0);
    signal zero_cnt: unsigned(7 downto 0);
    signal zero_avg: unsigned(8 downto 0);
    signal zero_tmp: unsigned(16 downto 0);
    signal perc_cnt: unsigned(6 downto 0);
```

```
    signal perc_tmp: unsigned(6 downto 0);
    signal perc_avg: unsigned(6 downto 0);
    ----------------------------------------------------------------
    -- Serial port state machine
    ----------------------------------------------------------------
    signal ser_state: unsigned(9 downto 0);
    alias ser_cnt: unsigned(4 downto 0) is ser_state(4 downto 0);
    alias ser_bit: unsigned(4 downto 0) is ser_state(9 downto 5);
    signal ser_out: unsigned(15 downto 0);
    signal txd_d: std_logic;
    ----------------------------------------------------------------
    -- Initial registered signal values
    ----------------------------------------------------------------
    attribute INIT: string;
    attribute INIT of fm_smp: signal is b"0";
    attribute INIT of cnt_5us: signal is b"0_0011_1110";
    attribute INIT of cnt_ones: signal is b"0_0000_0000";
    attribute INIT of ones_cnt: signal is b"0000_0000";
    attribute INIT of ones_avg: signal is b"1_1110_0000";
    attribute INIT of ones_tmp: signal is b"1_1110_0000_0000_0000";
    attribute INIT of zero_cnt: signal is b"0000_0000";
    attribute INIT of zero_avg: signal is b"0_0001_1111";
    attribute INIT of zero_tmp: signal is b"0_0001_1111_0000_0000";
    attribute INIT of ser_state: signal is b"10100_00000";
    attribute INIT of ser_out: signal is b"0000_0000_0000_0000";
    attribute INIT of txd_d: signal is b"1";
begin
--    xtlen<='1';-- rcvr.ucf only
    ----------------------------------------------------------------
    -- DCM:
    -- Clock input:         xtal -> IBUFG -> gclk -> DCM
    -- Feedback:            DCM -> gclk0 -> BUFG -> clk
    -- FM local oscillator: DCM -> clkfx -> BUFG -> fm_lo
    --
    -- Feedback:            DCM -> clki0 -> BUFG -> clkifb
    -- Internal clock:      DCM -> clkifx -> BUFG -> clki
    ----------------------------------------------------------------
    u_ibufg: IBUFG
        generic map (
            IOSTANDARD=>"LVCMOS33"
        )
        port map (
            O=>gclk,
            I=>xtal
        );
    u_dcm: DCM_SP
        generic map (
            CLKDV_DIVIDE=>2.0,
            CLKFX_DIVIDE=>17,
            CLKFX_MULTIPLY=>14,
            CLKIN_DIVIDE_BY_2=>false,
            CLKIN_PERIOD=>13.0,
            CLKOUT_PHASE_SHIFT=>"NONE",
            CLK_FEEDBACK=>"1X",
            DESKEW_ADJUST=>"SYSTEM_SYNCHRONOUS",
            DFS_FREQUENCY_MODE=>"LOW",
            DLL_FREQUENCY_MODE=>"LOW",
            DSS_MODE=>"NONE",
            DUTY_CYCLE_CORRECTION=>true,
            FACTORY_JF=>X"c080",
            PHASE_SHIFT=>0,
            STARTUP_WAIT=>false
        )
        port map (
            CLK0=>gclk0,
            CLK180=>open,
            CLK270=>open,
            CLK2X=>open,
            CLK2X180=>open,
            CLK90=>open,
            CLKDV=>open,
            CLKFX=>clkfx,
            CLKFX180=>open,
            LOCKED=>open,
            PSDONE=>open,
            STATUS=>open,
            CLKFB=>clk,
```

```
        CLKIN=>gclk,
        DSSEN=>'0',
        PSCLK=>'0',
        PSEN=>'0',
        PSINCDEC=>'0',
        RST=>'0'
    );
u_fb: BUFG
    port map (
        O=>clk,
        I=>gclk0
    );
u_bufg: BUFG
    port map (
        O=>fm_lo,
        I=>clkfx
    );

u_dcmi: DCM_SP
    generic map (
        CLKDV_DIVIDE=>2.0,
        CLKFX_DIVIDE=>5,
        CLKFX_MULTIPLY=>6,
        CLKIN_DIVIDE_BY_2=>false,
        CLKIN_PERIOD=>13.0,
        CLKOUT_PHASE_SHIFT=>"NONE",
        CLK_FEEDBACK=>"1X",
        DESKEW_ADJUST=>"SYSTEM_SYNCHRONOUS",
        DFS_FREQUENCY_MODE=>"LOW",
        DLL_FREQUENCY_MODE=>"LOW",
        DSS_MODE=>"NONE",
        DUTY_CYCLE_CORRECTION=>true,
        FACTORY_JF=>X"c080",
        PHASE_SHIFT=>0,
        STARTUP_WAIT=>false
    )
    port map (
        CLK0=>clki0,
        CLK180=>open,
        CLK270=>open,
        CLK2X=>open,
        CLK2X180=>open,
        CLK90=>open,
        CLKDV=>open,
        CLKFX=>clkifx,
        CLKFX180=>open,
        LOCKED=>open,
        PSDONE=>open,
        STATUS=>open,
        CLKFB=>clkifb,
        CLKIN=>gclk,
        DSSEN=>'0',
        PSCLK=>'0',
        PSEN=>'0',
        PSINCDEC=>'0',
        RST=>'0'
    );
u_fbi: BUFG
    port map (
        O=>clkifb,
        I=>clki0
    );
u_bufgi: BUFG
    port map (
        O=>clki,
        I=>clkifx
    );

----------------------------------------------------------------------
-- FM receiver
----------------------------------------------------------------------
process(clki)
begin
    if (rising_edge(clki)) then
        -- Sample comparator output
        fm_smp<=fm_cmp;
```

```
        -- Interval counter
        if (cnt_5us=b"1_1111_1111") then
            if (fm_smp='1') and (cnt_ones(8)='1') then
                cnt_5us<=b"0_0011_1101";-- Lengthen interval
            elsif (fm_smp='1') and (cnt_ones(8)='0') then
                cnt_5us<=b"0_0011_1111";-- Shorten interval
            else
                cnt_5us<=b"0_0011_1110";
            end if;
        else
            cnt_5us<=cnt_5us+1;
        end if;

        -- Ones counter
        if (cnt_5us=b"1_1111_1111") then
            if (fm_smp='1') then
                cnt_ones<=b"0_0010_0000";-- Start at 32
            else
                cnt_ones<=b"0_0001_1111";-- Start at 31
            end if;
        else
            if (fm_smp='1') then
                cnt_ones<=cnt_ones+1;
            end if;
        end if;
    end if;
end process;

-----------------------------------------------------------------
-- Statistics gathering
-----------------------------------------------------------------
process(clki)
begin
    if (rising_edge(clki)) then
        if (cnt_5us=b"1_1111_1111") then
            if (cnt_ones(8)='1') then
                if (ones_cnt=b"1111_1111") then
                    ones_avg<=ones_tmp(16 downto 8);
                    ones_tmp<=(b"0000_0000"&cnt_ones);
                else
                    ones_tmp<=ones_tmp+(b"0000_0000"&cnt_ones);
                end if;
                ones_cnt<=ones_cnt+1;

                if (perc_cnt=b"110_0011") then
                    perc_avg<=perc_tmp;
                    perc_tmp<=b"000_0001";
                    perc_cnt<=b"000_0000";
                else
                    perc_tmp<=perc_tmp+1;
                    perc_cnt<=perc_cnt+1;
                end if;
            else
                if (zero_cnt=b"1111_1111") then
                    zero_avg<=zero_tmp(16 downto 8);
                    zero_tmp<=(b"0000_0000"&cnt_ones);
                else
                    zero_tmp<=zero_tmp+(b"0000_0000"&cnt_ones);
                end if;
                zero_cnt<=zero_cnt+1;

                if (perc_cnt=b"110_0011") then
                    perc_avg<=perc_tmp;
                    perc_tmp<=b"000_0000";
                    perc_cnt<=b"000_0000";
                else
                    perc_cnt<=perc_cnt+1;
                end if;
            end if;
        end if;
    end if;
end process;

-----------------------------------------------------------------
-- Serial port state machine
--
-- Detect that a byte is being sent, and transmit statistics.
```

```
--
-- State uses two nested counters:
--     ser_cnt counts 30 clock cyles (90MHz/3MegaBAUD)
--     ser_bit counts 10 bit periods
----------------------------------------------------------------------
-- Serial port loopback
cts<=rts;
dsr<=dtr;
dcd<=dtr;
ri<='1';
process(clki)
begin
    if (rising_edge(clki)) then
        case ser_state is
            when b"00000_00000" =>
                rxd<='0';
            when b"00001_00000" =>
                rxd<=ser_out(0);
            when b"00010_00000" =>
                rxd<=ser_out(1);
            when b"00011_00000" =>
                rxd<=ser_out(2);
            when b"00100_00000" =>
                rxd<=ser_out(3);
            when b"00101_00000" =>
                rxd<=ser_out(4);
            when b"00110_00000" =>
                rxd<=ser_out(5);
            when b"00111_00000" =>
                rxd<=ser_out(6);
            when b"01000_00000" =>
                rxd<=ser_out(7);
            when b"01001_00000" =>
                rxd<='1';
            when b"01010_00000" =>
                rxd<='0';
            when b"01011_00000" =>
                rxd<=ser_out(8);
            when b"01100_00000" =>
                rxd<=ser_out(9);
            when b"01101_00000" =>
                rxd<=ser_out(10);
            when b"01110_00000" =>
                rxd<=ser_out(11);
            when b"01111_00000" =>
                rxd<=ser_out(12);
            when b"10000_00000" =>
                rxd<=ser_out(13);
            when b"10001_00000" =>
                rxd<=ser_out(14);
            when b"10010_00000" =>
                rxd<=ser_out(15);
            when b"10011_00000" =>
                rxd<='1';
            when b"10100_00000" =>
                if (txd_d='0') then
                    ser_state<=b"00000_00000";
                    ser_out(8 downto 0)<=ones_avg-zero_avg;
                    ser_out(15 downto 9)<=perc_avg;
                end if;
            when others =>
                null;
        end case;

        -- Increment state counter
        if (ser_bit=b"10100") then
            null;
        elsif (ser_cnt=b"11101") then
            ser_cnt<=b"00000";
            ser_bit<=ser_bit+1;
        else
            ser_cnt<=ser_cnt+1;
        end if;

        -- Update input latch
        txd_d<=txd;
    end if;
```

```
    end process;

    ----------------------------------------------------------------
    -- DAC control signals
    ----------------------------------------------------------------
    ndac_ncs<='0';-- dacq.ucf only
    ndac_clk<='0';-- dacq.ucf only
    ndac_din<='0';-- dacq.ucf only
    ndac_ncl<='0';-- dacq.ucf only

    pdac_ncs<='0';-- dacq.ucf only
    pdac_clk<='0';-- dacq.ucf only
    pdac_din<='0';-- dacq.ucf only
    pdac_ncl<='0';-- dacq.ucf only

    ----------------------------------------------------------------
    -- ADC control signals
    ----------------------------------------------------------------
    pga0_ncs<='0';-- dacq.ucf only
    pga0_dio<='0';-- dacq.ucf only
    pga0_clk<='0';-- dacq.ucf only
    adc0_sdi<='0';-- dacq.ucf only
    adc0_sck<='0';-- dacq.ucf only
    adc0_cnv<='0';-- dacq.ucf only

    pga1_ncs<='0';-- dacq.ucf only
    pga1_dio<='0';-- dacq.ucf only
    pga1_clk<='0';-- dacq.ucf only
    adc1_sdi<='0';-- dacq.ucf only
    adc1_sck<='0';-- dacq.ucf only
    adc1_cnv<='0';-- dacq.ucf only

    pga2_ncs<='0';-- dacq.ucf only
    pga2_dio<='0';-- dacq.ucf only
    pga2_clk<='0';-- dacq.ucf only
    adc2_sdi<='0';-- dacq.ucf only
    adc2_sck<='0';-- dacq.ucf only
    adc2_cnv<='0';-- dacq.ucf only

    pga3_ncs<='0';-- dacq.ucf only
    pga3_dio<='0';-- dacq.ucf only
    pga3_clk<='0';-- dacq.ucf only
    adc3_sdi<='0';-- dacq.ucf only
    adc3_sck<='0';-- dacq.ucf only
    adc3_cnv<='0';-- dacq.ucf only

    ----------------------------------------------------------------
    -- FM receiver
    ----------------------------------------------------------------
    fm_ncs<='0';
    fm_clk<='0';

    ----------------------------------------------------------------
    -- Digital I/O
    ----------------------------------------------------------------
--    rng1<='0';-- rcvr.ucf only
--    rng2<='0';-- rcvr.ucf only
end arch;
```

The sixth script given is the UCF file used to tune the FM receiver of the the DACQ units.
It is the physical mapping between the pins of the FPGA and the signals used in the previous VHDL script.

```
#########################################################
#                                                       #
# XC3S50ATQ144 Spartan3AN with 3 blocks of 18kbits RAM  #
#                                                       #
#########################################################


# Timing constraints for 75MHz oscillator
NET xtal CLOCK_DEDICATED_ROUTE=FALSE;
NET xtal PERIOD=12.5ns;
```

```
# Timing constraints for internal 90MHz clock
NET clki PERIOD=11ns;

# Pin assignements
#NET TMS       LOC=P1;    #TMS,JTAG,VCCAUX
#NET TDI       LOC=P2;    #TDI,JTAG,VCCAUX
#NET NC        LOC=P3;    #IO_L02P_3,IO,BANK3
#NET NC        LOC=P4;    #IO_L01P_3,IO,BANK3
#NET NC        LOC=P5;    #IO_L02N_3,IO,BANK3
#NET NC        LOC=P6;    #IO_L01N_3,IO,BANK3
#NET NC        LOC=P7;    #IO_L03P_3,IO,BANK3
#NET NC        LOC=P8;    #IO_L03N_3,IO,BANK3
#NET GND       LOC=P9;    #GND,GND,GND
#NET NC        LOC=P10;   #IO_L04P_3,IO,BANK3
#NET NC        LOC=P11;   #IO_L04N_3/VREF_3,IO,BANK3
#NET NC        LOC=P12;   #IO_L05P_3/LHCLK0,LHCLK,BANK3
#NET NC        LOC=P13;   #IO_L05N_3/LHCLK1,LHCLK,BANK3
#NET 3V3       LOC=P14;   #VCCO_3,VCCO,BANK3
#NET NC        LOC=P15;   #IO_L06P_3/LHCLK2,LHCLK,BANK3
#NET NC        LOC=P16;   #IO_L06N_3/IRDY2/LHCLK3,LHCLK,BANK3
#NET GND       LOC=P17;   #GND,GND,GND
#NET NC        LOC=P18;   #IO_L07P_3/LHCLK4,LHCLK,BANK3
#NET NC        LOC=P19;   #IO_L08P_3/TRDY2/LHCLK6,LHCLK,BANK3
NET fm_lo     LOC=P20;   #IO_L07N_3/LHCLK5,LHCLK,BANK3
#NET 3V3       LOC=P21;   #IO_L08N_3/LHCLK7,LHCLK,BANK3
#NET 1V2       LOC=P22;   #VCCINT,VCCINT,VCCINT
#NET 3V3       LOC=P23;   #VCCO_3,VCCO,BANK3
NET fm_ncs    LOC=P24;   #IO_L09P_3,IO,BANK3
NET fm_clk    LOC=P25;   #IO_L09N_3,IO,BANK3
#NET GND       LOC=P26;   #GND,GND,GND
NET fm_cmp    LOC=P27;   #IO_L10P_3,IO,BANK3
NET fm_dat    LOC=P28;   #IO_L11P_3,IO,BANK3
NET ndac_ncs  LOC=P29;   #IO_L10N_3,IO,BANK3
NET ndac_clk  LOC=P30;   #IO_L11N_3,IO,BANK3
NET ndac_din  LOC=P31;   #IO_L12P_3,IO,BANK3
NET ndac_ncl  LOC=P32;   #IO_L12N_3,IO,BANK3
#NET 3V3       LOC=P33;   #IP_L13P_3,INPUT,BANK3
#NET GND       LOC=P34;   #GND,GND,GND
#NET 3V3       LOC=P35;   #IP_L13N_3/VREF_3,INPUT,BANK3
#NET 3V3       LOC=P36;   #VCCAUX,VCCAUX,VCCAUX
#NET 3V3       LOC=P37;   #IO_L01P_2/M1,DUAL,BANK2
#NET 3V3       LOC=P38;   #IO_L01N_2/M0,DUAL,BANK2
#NET GND       LOC=P39;   #IO_L02P_2/M2,DUAL,BANK2
#NET 3V3       LOC=P40;   #VCCO_2,VCCO,BANK2
NET pdac_ncs  LOC=P41;   #IO_L02N_2/CSO_B,DUAL,BANK2
NET pdac_clk  LOC=P42;   #IO_L03P_2/RDWR_B,DUAL,BANK2
#NET 3V3       LOC=P43;   #IO_L04P_2/VS2,DUAL,BANK2
#NET 3V3       LOC=P44;   #IO_L03N_2/VS1,DUAL,BANK2
#NET 3V3       LOC=P45;   #IO_L04N_2/VS0,DUAL,BANK2
NET pdac_din  LOC=P46;   #IO_L05P_2,IO,BANK2
NET pdac_ncl  LOC=P47;   #IO_L06P_2,IO,BANK2
#NET NC        LOC=P48;   #IO_L05N_2/D7,DUAL,BANK2
NET pga3_ncs  LOC=P49;   #IO_L06N_2/D6,DUAL,BANK2
NET pga3_dio  LOC=P50;   #IO_L07P_2/D5,DUAL,BANK2
#NET 3V3       LOC=P51;   #IO_L07N_2/D4,DUAL,BANK2
#NET 1V2       LOC=P52;   #VCCINT,VCCINT,VCCINT
#NET 3V3       LOC=P53;   #IP_2/VREF_2,INPUT,BANK2
#NET NC        LOC=P54;   #IO_L08P_2/GCLK14,GCLK,BANK2
NET pga3_clk  LOC=P55;   #IO_L08N_2/GCLK15,GCLK,BANK2
#NET GND       LOC=P56;   #GND,GND,GND
NET adc3_sdi  LOC=P57;   #IO_L09P_2/GCLK0,GCLK,BANK2
#NET NC        LOC=P58;   #IO_L10P_2/GCLK2,GCLK,BANK2
#NET NC        LOC=P59;   #IO_L09N_2/GCLK1,GCLK,BANK2
#NET NC        LOC=P60;   #IO_L10N_2/GCLK3,GCLK,BANK2
#NET 3V3       LOC=P61;   #VCCO_2,VCCO,BANK2
NET adc3_sck  LOC=P62;   #IO_2/MOSI/CSI_B,DUAL,BANK2
NET adc3_cnv  LOC=P63;   #IO_L11P_2/AWAKE,PWRMGMT,BANK2
#NET NC        LOC=P64;   #IO_L11N_2/DOUT,DUAL,BANK2
#NET GND       LOC=P65;   #GND,GND,GND
#NET 3V3       LOC=P66;   #VCCAUX,VCCAUX,VCCAUX
#NET NC        LOC=P67;   #IO_L12P_2/INIT_B,DUAL,BANK2
#NET NC        LOC=P68;   #IO_L12N_2/D3,DUAL,BANK2
NET adc3_sdo  LOC=P69;   #IO_L13P_2/D2,DUAL,BANK2
#NET NC        LOC=P70;   #IO_L14P_2/D1,DUAL,BANK2
NET pga2_ncs  LOC=P71;   #IO_L13N_2/D0/DIN/MISO,DUAL,BANK2
NET pga2_dio  LOC=P72;   #IO_L14N_2/CCLK,DUAL,BANK2
#NET NC        LOC=P73;   #DONE,CONFIG,VCCAUX
```

```
#NET GND      LOC=P74;   #SUSPEND,PWRMGMT,VCCAUX
NET pga2_clk LOC=P75;   #IO_L02P_1/LDC1,DUAL,BANK1
NET adc2_sdi LOC=P76;   #IO_L01P_1/HDC,DUAL,BANK1
NET adc2_sck LOC=P77;   #IO_L02N_1/LDC0,DUAL,BANK1
NET adc2_cnv LOC=P78;   #IO_L01N_1/LDC2,DUAL,BANK1
NET adc2_sdo LOC=P79;   #IO_1,IO,BANK1
#NET 3V3      LOC=P80;   #IP_1/VREF_1,INPUT,BANK1
#NET GND      LOC=P81;   #GND,GND,GND
NET pga1_ncs LOC=P82;   #IO_L03P_1,IO,BANK1
NET pga1_dio LOC=P83;   #IO_L04P_1/RHCLK0,RHCLK,BANK1
NET pga1_clk LOC=P84;   #IO_L03N_1,IO,BANK1
NET adc1_sdi LOC=P85;   #IO_L04N_1/RHCLK1,RHCLK,BANK1
#NET 3V3      LOC=P86;   #VCCO_1,VCCO,BANK1
NET adc1_sck LOC=P87;   #IO_L05P_1/RHCLK2,RHCLK,BANK1
NET adc1_cnv LOC=P88;   #IO_L05N_1/TRDY1/RHCLK3,RHCLK,BANK1
#NET GND      LOC=P89;   #GND,GND,GND
NET adc1_sdo LOC=P90;   #IO_L06P_1/RHCLK4,RHCLK,BANK1
NET pga0_ncs LOC=P91;   #IO_L07P_1/IRDY1/RHCLK6,RHCLK,BANK1
NET pga0_dio LOC=P92;   #IO_L06N_1/RHCLK5,RHCLK,BANK1
#NET 3V3      LOC=P93;   #IO_L07N_1/RHCLK7,RHCLK,BANK1
#NET 1V2      LOC=P94;   #VCCINT,VCCINT,VCCINT
#NET 3V3      LOC=P95;   #VCCO_1,VCCO,BANK1
NET pga0_clk LOC=P96;   #IO_L08P_1,IO,BANK1
#NET 3V3      LOC=P97;   #IP_1/VREF_1,INPUT,BANK1
NET adc0_sdi LOC=P98;   #IO_L08N_1,IO,BANK1
NET adc0_sck LOC=P99;   #IO_L09P_1,IO,BANK1
#NET GND      LOC=P100;  #GND,GND,GND
NET adc0_cnv LOC=P101;  #IO_L09N_1,IO,BANK1
NET adc0_sdo LOC=P102;  #IO_L10P_1,IO,BANK1
#NET NC       LOC=P103;  #IO_L11P_1,IO,BANK1
#NET NC       LOC=P104;  #IO_L10N_1,IO,BANK1
#NET NC       LOC=P105;  #IO_L11N_1,IO,BANK1
#NET GND      LOC=P106;  #GND,GND,GND
#NET TDO      LOC=P107;  #TDO,JTAG,VCCAUX
#NET 3V3      LOC=P108;  #VCCAUX,VCCAUX,VCCAUX
#NET TCK      LOC=P109;  #TCK,JTAG,VCCAUX
#NET NC       LOC=P110;  #IO_L01P_0,IO,BANK0
#NET NC       LOC=P111;  #IO_L01N_0,IO,BANK0
#NET NC       LOC=P112;  #IO_L02P_0/VREF_0,IO,BANK0
#NET NC       LOC=P113;  #IO_L02N_0,IO,BANK0
#NET NC       LOC=P114;  #IO_L04P_0,IO,BANK0
#NET NC       LOC=P115;  #IO_L03P_0,IO,BANK0
#NET NC       LOC=P116;  #IO_L04N_0,IO,BANK0
#NET NC       LOC=P117;  #IO_L03N_0,IO,BANK0
#NET GND      LOC=P118;  #GND,GND,GND
#NET 3V3      LOC=P119;  #VCCO_0,VCCO,BANK0
#NET NC       LOC=P120;  #IO_L05P_0,IO,BANK0
#NET NC       LOC=P121;  #IO_L05N_0,IO,BANK0
#NET 1V2      LOC=P122;  #VCCINT,VCCINT,VCCINT
#NET 3V3      LOC=P123;  #IP_0/VREF_0,INPUT,BANK0
NET xtal     LOC=P124;  #IO_L06P_0/GCLK4,GCLK,BANK0
NET cbus3    LOC=P125;  #IO_L07P_0/GCLK6,GCLK,BANK0
NET cbus2    LOC=P126;  #IO_L06N_0/GCLK5,GCLK,BANK0
NET cts      LOC=P127;  #IO_L07N_0/GCLK7,GCLK,BANK0
#NET GND      LOC=P128;  #GND,GND,GND
NET dcd      LOC=P129;  #IO_L08P_0/GCLK8,GCLK,BANK0
NET dsr      LOC=P130;  #IO_L09P_0/GCLK10,GCLK,BANK0
NET ri       LOC=P131;  #IO_L08N_0/GCLK9,GCLK,BANK0
NET rxd      LOC=P132;  #IO_L09N_0/GCLK11,GCLK,BANK0
#NET 3V3      LOC=P133;  #VCCAUX,VCCAUX,VCCAUX
#NET NC       LOC=P134;  #IO_L10P_0,IO,BANK0
NET rts      LOC=P135;  #IO_L10N_0,IO,BANK0
#NET 3V3      LOC=P136;  #VCCO_0,VCCO,BANK0
#NET GND      LOC=P137;  #GND,GND,GND
NET dtr      LOC=P138;  #IO_L11P_0,IO,BANK0
NET txd      LOC=P139;  #IO_L11N_0,IO,BANK0
#NET 3V3      LOC=P140;  #IP_0,INPUT,BANK0
NET cbus1    LOC=P141;  #IO_L12P_0/VREF_0,IO,BANK0
NET cbus0    LOC=P142;  #IO_0,IO,BANK0
#NET GND      LOC=P143;  #IO_L12N_0/PUDC_B,DUAL,BANK0
#NET 3V3      LOC=P144;  #PROG_B,CONFIG,VCCAUX
```

Appendix B

Matlab® Scripts

In this appendix are given the Matlab® scripts used to interact with the FM SYNC and DACQ units.

The first Matlab® script given is used to set a new initial timestamp value for FM SYNC unit.

```
function set_timestamp(timestamp)
% set_timestamp(timestamp)
%     Write timestamp to FM transmitter.  If no argument is given,
%     set_timestampt uses the current (UNIX style) time.

% Last updated: 2012-11-23

% Use the current time if no argument is given.
if nargin<1
        unix_time=etime(clock,[1970 1 1 0 0 0]);
        timestamp=round(unix_time/9.6e-3);
end

if ~isscalar(timestamp)
        error('The timestamp must be a scalar.');
end
if isa(timestamp,'char')||...
                isa(timestamp,'int8')||isa(timestamp,'uint8')||...
                isa(timestamp,'int16')||isa(timestamp,'uint16')||...
                isa(timestamp,'int32')||isa(timestamp,'uint32')
        % char range is 0 to 65535
        % int8 range is -128 to 127
        % uint8 range is 0 to 255
        % int16 range is -32768 to 32767
        % uint16 range is 0 to 65535
        % int32 range is -2147483648 to 2147483647
        % uint32 range is 0 to 4294967295
        %
        % In each case, the type can be safely promoted to int64, then
        % typecast to uint64.
        timestamp=typecast(int64(timestamp),'uint64');
elseif isa(timestamp,'int64')
        % For an int64, do just the typecast.
        timestamp=typecast(timestamp,'uint64');
elseif isa(timestamp,'single')||isa(timestamp,'double')
        % These are signed types, so we must go through int64.  But
        % check the conversion for accuracy.
        if double(int64(timestamp))~=timestamp
                fprintf('[Warning: Conversion from floating point '...
                        'to integer resulted in some loss of '...
                        'precision.');
        end
        timestamp=typecast(int64(timestamp),'uint64');
elseif ~isa(timestamp,'uint64')
        error('Don''t know how to convert timestamp to uint64.');
end

data=char(zeros(1,15));
data(1)=bitshift(timestamp,-56,4)+224;
```

```
data(2)=bitshift(timestamp,-52,4)+208;
data(3)=bitshift(timestamp,-48,4)+192;
data(4)=bitshift(timestamp,-44,4)+176;
data(5)=bitshift(timestamp,-40,4)+160;
data(6)=bitshift(timestamp,-36,4)+144;
data(7)=bitshift(timestamp,-32,4)+128;
data(8)=bitshift(timestamp,-28,4)+112;
data(9)=bitshift(timestamp,-24,4)+96;
data(10)=bitshift(timestamp,-20,4)+80;
data(11)=bitshift(timestamp,-16,4)+64;
data(12)=bitshift(timestamp,-12,4)+48;
data(13)=bitshift(timestamp,-8,4)+32;
data(14)=bitshift(timestamp,-4,4)+16;
data(15)=bitshift(timestamp,0,4)+0;

fm=sio_open('COM5');
sio_baud(fm,3000000);
sio_write(fm,data);
sio_close(fm);
```

The second Matlab® script given is used to read out the current timestamp value of FM SYNC unit.

```
function timestamp=get_timestamp
% timestamp = get_timestamp
%     Read timestamp from FM transmitter.

% Last updated: 2012-11-23

fm=sio_open('COM5');
sio_baud(fm,3000000);
sio_write(fm,char(255));
data=uint64(abs(sio_read(fm,8)));
sio_close(fm);

timestamp=...
        bitshift(data(1),0,8)+bitshift(data(2),8,16)+...
        bitshift(data(3),16,24)+bitshift(data(4),24,32)+...
        bitshift(data(5),32,40)+bitshift(data(6),40,48)+...
        bitshift(data(7),48,56)+bitshift(data(8),56,60);

unix_time=round(double(timestamp)*9.6e-3);
fprintf('%s\n',datestr(addtodate(datenum([1970 1 1 0 0 0]),...
        unix_time,'second')));
```

The third Matlab® script given is used to help tuning the FM receiver of a DACQ unit.

```
dacq=sio_open('COM3');
%dacq=sio_open('COM4');
%dacq=sio_open('COM5');
%dacq=sio_open('COM6');
sio_baud(dacq,3000000);
%sio_handshake(dacq,21);

for i=1:1000
    sio_write(dacq,char(0));
    x=double(sio_read(dacq,2));
    d=x(1)+bitshift(bitand(x(2),1),8);
    p=bitshift(x(2),-1,7);
    if p<10
        fprintf('Difference is %d, turn clockwise\n',d);
    elseif p>90
        fprintf('Difference is %d, turn counter-clockwise\n',d);
    else
        fprintf('Difference is %d, percent is %d\n',d,p);
    end
    pause(0.1);
end
sio_close(dacq);
```

Appendix C

Python® Scripts

In this appendix are given the Python® scripts used to implement the sand-scorpion spiking neural network model.

The first Python® script given is the one created by Brette *et al.* in [25, 107]. First, parameters are initialized to the desired values to take into account the characteristics of the simulated sonic wave. Then, the sonic wave is generated. Then, the 2 neural processes (one for the leg and one for the command neurons) are implemented, each one with its own differential equation. Finally, the azimuth of the sonic source is calculated.

```
'''
Adapted from
Theory of Arachnid Prey Localization
W. Sturzl, R. Kempter, and J. L. van Hemmen
PRL 2000

Poisson inputs are replaced by integrate-and-fire neurons

Romain Brette
'''
from brian import *

# Parameters
degree=2*pi/360.
duration=500*ms
R=2.5*cm # radius of scorpion
vr=50*meter/second # Rayleigh wave speed
phi=144*degree # angle of prey
A=250*Hz
deltaI=.7*ms # inhibitory delay
gamma=(22.5+45*arange(8))*degree # leg angle
delay=R/vr*(1-cos(phi-gamma))  # wave delay

# Wave (vector w)
t=arange(int(duration/defaultclock.dt)+1)*defaultclock.dt
Dtot=0.
w=0.
for f in range(150,451):
    D=exp(-(f-300)**2/(2*(50**2)))
    xi=2*pi*rand()
    w+=100*D*cos(2*pi*f*t+xi)
    Dtot+=D
w=.01*w/Dtot

# Rates from the wave
def rates(t):
    return w[array(t/defaultclock.dt,dtype=int)]

# Leg mechanical receptors
tau_legs=1*ms
sigma=.01
eqs_legs="""
```

```
dv/dt=(1+rates(t-d)waist-v)/tau_legs+sigma*(2./tau_legs)**.5*xi:1
d : second
"""
legs=NeuronGroup(8,model=eqs_legs,threshold=1,reset=0,refractory=1*ms,unit_checking=False)
legs.d=delay
spikes_legs=SpikeCounter(legs)

# Command neurons
tau=1*ms
taus=1*ms
wex=7
winh=-2
eqs_neuron='''
dv/dt=(x-v)/tau : 1
dx/dt=(y-x)/taus : 1 # alpha currents
dy/dt=-y/taus : 1
'''
neurons=NeuronGroup(8,model=eqs_neuron,threshold=1,reset=0)
synapses_ex=IdentityConnection(legs,neurons,'y',weight=wex)
synapses_inh=Connection(legs,neurons,'y',delay=deltaI)
for i in range(8):
    synapses_inh[i,(4+i-1)%8]=winh
    synapses_inh[i,(4+i)%8]=winh
    synapses_inh[i,(4+i+1)%8]=winh
spikes=SpikeCounter(neurons)

run(duration)
nspikes=spikes.count
x=sum(nspikes*exp(gamma*1j))
print "Angle (deg):",arctan(imag(x)/real(x))/degree
polar(concatenate((gamma,[gamma[0]+2*pi])),concatenate((nspikes,[nspikes[0]]))/duration)
show()
```

The second Python® script given is our modified version of the previous implementation.
First, some functions are define to handle the reading of the source file containing the data from the seismic sensors. Then, parameters are initialized to the desired values to take into account the characteristics of the experimental environment. Then, the 2 neural processes (one for the leg and one for the command neurons) are implemented, each one with its own differential equation. Finally, the azimuth of the sonic source is calculated. finally, the results are displayed.

```
'''
Adapted from
Theory of Arachnid Prey Localization
W. Sturzl, R. Kempter, and J. L. van Hemmen
PRL 2000

Poisson inputs are replaced by integrate-and-fire neurons

Romain Brette

modify: monday 13th march 2013
G.Garreau

reason: vmware crash
v6.0-clean/final version with real data, moving target, 5 legs
'''
from brian import *
import os, time, numpy
from matplotlib import pyplot

# functions definition
def ensure_file(f):
    """ ensures that file f exists """
    if not os.path.isfile(f):
        print 'The file ' + str(f) + ' does not exist.'
        print 'End.'
        exit(1)

def get_experimental_data(filename):
    """
```

```
    Read the experimental data from the binary files
    Return three vectors, one for each of the axis (x,y,z)
    """
    print

    '''get data file information'''
    try:
        st = os.stat(filename)
    except IOError:
        print "Failed to get information about", filename
    else:
        print 'Data file name: ' + filename
        #print "Data file size:", st[ST_SIZE]
        #print "Data file modified:", time.asctime(time.localtime(st[ST_MTIME]))

    z_vector = numpy.loadtxt(fname = filename, dtype = numpy.float64, delimiter='\n', unpack=True)

    z_vector = z_vector[0:-1:n2]

    ''' plot z vector'''
#    plot(z_vector,color='red') #red
#    show()
    print "length(z)", len(z_vector)
    print "min of z_vector:", z_vector.min()
    print "mean of z_vector:", z_vector.mean()
    print "max of z_vector:", z_vector.max()

    return (z_vector*gain)

# Rates from the wave
# rates is for multi sources
def rates(t):
    """
    return the output of the seismic sensors at time t
    """
    if(np.size(t)<5):
        return 0.

#    print "t", t

    global z1_vector
    global z2_vector
    global z3_vector
    global z4_vector
    global z5_vector
#delay[4]=0.1e-3*second
#delay[1]=0.1e-3*second
#delay[0]=0.2e-3*second
#delay[3]=0.0e-3*second
#delay[2]=0.2e-3*second
    global t_index

#    temp_array = array(t/defaultclock.dt,dtype=int)

    data_array=zeros(shape=(5),dtype= numpy.float64, order='C')
    data_array[0]=z1_vector[array(t[0]/defaultclock.dt,dtype=int)]
    data_array[1]=z2_vector[array(t[1]/defaultclock.dt,dtype=int)]
    data_array[2]=z3_vector[array(t[2]/defaultclock.dt,dtype=int)]
    data_array[3]=z4_vector[array(t[3]/defaultclock.dt,dtype=int)]
    data_array[4]=z5_vector[array(t[4]/defaultclock.dt,dtype=int)]

    t_index=array(t/defaultclock.dt,dtype=int)
#    print "t_index_rates", t_index

    return data_array

#filenames of the input data files
# multiple sources
# circs file is 3.3e6 samples -> 99sec of data
# tune / circs / randcircs / randradius // nofilt / filt
seismic2_filename = 'F:\SCORPION_v2\data_set\ucy4guillaume1_seismic2_circ3_norm_std.txt' #data for seismic sensor 1
seismic3_filename = 'F:\SCORPION_v2\data_set\ucy4guillaume1_seismic3_circ3_norm_std.txt' #data for seismic sensor 1
seismic4_filename = 'F:\SCORPION_v2\data_set\ucy4guillaume1_seismic4_circ3_norm_std.txt' #data for seismic sensor 1
seismic5_filename = 'F:\SCORPION_v2\data_set\ucy4guillaume1_seismic5_circ3_norm_std.txt' #data for seismic sensor 1
seismic6_filename = 'F:\SCORPION_v2\data_set\ucy4guillaume1_seismic6_circ3_norm_std.txt' #data for seismic sensor 1

#name results files 43562 for ucy2 // 52463 for parking2
```

```
fname = '6.0_Angles_5legs_ucy4guillaume1_circ3_norm_std_1000ms_88sim_thleg1_vr335ms_gain02_52463'

# Parameters
n =1 # sampling rate reduction producing txt file
n2 = 10 # sampling rate reduction in reading txt file
duration=1000*ms
R=15*cm # radius of scorpion (initial value 2.5cm)
vr=335.0*meter/second # Rayleigh wave speed (initial value 50m/s)
mdt = 2*R/vr # max delta t, max inter-leg delay
mdt2=1.0*mdt
mdt3=(1/1.0)*mdt
print "Max delay", mdt
degree=2*pi/360. # degree to rad conversion
phi=10*degree # angle of prey in rad
print "Expected angle", phi/degree
#A=250*Hz
deltaI=.7*mdt # inhibitory delay initial value, 0.7ms
gamma=(0+72*arange(5))*degree # leg angle for 6 legs
#delay=R/vr*(1-cos(phi-gamma))  # wave delay
# delay multiple sources
delay = zeros(shape=(5),dtype= numpy.float64, order='C') # delay as we have internal delay of each sensor
# tdelay 2-3-4-5-6: [0.1,0.1,0.2,0.2,0.0]e-3*second - trigo order 4-3-5-6-2
print "Delay matrix", delay
loop = 88 # nber of time to repeat the simulation so to have average performances
print "Nber of run to execute:", loop
print "total simulation time is:", loop*duration
angle_array = numpy.zeros(loop)  #numpy array of estimated angles
sampling_freq = (int(1e5/(3*n*n2))+1)*Hz # sampling frequency data, original code 10kHz
defaultclock.dt = (1/sampling_freq) # default value is 0.1ms
gain = 0.2 # normalization coefficient for data

'''
Ensure that data files exist
'''
ensure_file(seismic2_filename)
ensure_file(seismic3_filename)
ensure_file(seismic4_filename)
ensure_file(seismic5_filename)
ensure_file(seismic6_filename)

'''
 Get the experimental data from the binary files
 The data represent the output of the three seismic sensors
'''
# multiple sources
(z2_vector) = get_experimental_data(seismic2_filename)
(z5_vector) = get_experimental_data(seismic3_filename)
(z3_vector) = get_experimental_data(seismic4_filename)
(z1_vector) = get_experimental_data(seismic5_filename)
(z4_vector) = get_experimental_data(seismic6_filename)

# 1 source
#(z1_vector) = get_experimental_data(seismic1_filename)

print "Max simulation time allowed (s):", int(len(z1_vector)/sampling_freq)

# Wave (vector w)
t=arange(int(loop*duration/defaultclock.dt)+1)*defaultclock.dt
#t=arange(len(z1_vector))*defaultclock.dt

# Leg mechanical receptors
tau_legs=mdt3
# sigma=.01 # +sigma*(2./tau_legs)**.5*xi
eqs_legs="""
dv/dt=(1+rates(t-d)-v)/tau_legs:1
d : second
"""
legs=NeuronGroup(5,model=eqs_legs,threshold=1,reset=0,refractory=mdt2,unit_checking=False)
legs.d=delay
spikes_legs=SpikeCounter(legs)

# Command neurons
tau=mdt2
taus=mdt2
wex=7
winh=-2
eqs_neuron='''
```

```
dv/dt=(x-v)/tau : 1
dx/dt=(y-x)/taus : 1 # alpha currents
dy/dt=-y/taus : 1
'''
neurons=NeuronGroup(5,model=eqs_neuron,threshold=1,reset=0)
synapses_ex=IdentityConnection(legs,neurons,'y',weight=wex)
synapses_inh=Connection(legs,neurons,'y',delay=deltaI)
for i in range(5):
    synapses_inh[i,(2+i)%5]=winh
    synapses_inh[i,(2+i+1)%5]=winh
spikes=SpikeCounter(neurons)

for i in range(loop):
    print "i", i
    run(duration)
    nspikes=spikes.count
    print "Nspikes:", nspikes
    x=sum(nspikes*exp(gamma*1j))

    if real(x) > 0:
        angle = (arctan(imag(x)/real(x))/degree)%360
    else:
        angle = ((arctan(imag(x)/real(x))/degree)+180)%360

    print "Angle:", angle
    angle_array[i] = angle

    legs.reset()
    spikes_legs.reinit()
    neurons.reset()
    spikes.reinit()

print "t_end", t_index*defaultclock.dt

#print "Min", angle_array.min()
#print "Mean", angle_array.mean()
#print "Max", angle_array.max()
#print "Std", angle_array.std()

fig = pyplot.figure()
fig.canvas.set_window_title('Subject angle plot')
pyplot.suptitle('Subject angle with parameters: \n mdt= '+str(mdt)+' simulation duration = '+str(duration)+' and repeti
                color = 'black', style='normal',fontsize='12',bbox=dict(facecolor='white',ec='none',alpha=1.0,boxstyle=
plot(duration*arange(loop),angle_array,color='blue')
#show()
pyplot.savefig('Plot_'+fname+'.png',format='png')

numpy.savetxt(fname+'.txt', angle_array, fmt='%2.1f', delimiter='\n')

#polar(concatenate((gamma,[gamma[0]+2*pi])),concatenate((nspikes,[nspikes[0]]))/duration)
#show()

#numpy.savetxt('signalseismic4_310degree.txt', w, fmt='%2.6f', delimiter='\n')
#numpy.savetxt('delay_310degree.txt', delay, fmt='%2.6f', delimiter='\n')

print "END"
raw_input('')
```

Appendix D

Synchronization & Phase-Lock

Digital counting circuits

Let's consider the following two problems: 1- Given a 20MHz system clock, write VHDL code (for an FPGA) that generates a 1MHz square wave. Trivial! 2- Given a 20MHz system clock, write VHDL code (for an FPGA) that generates a 3MHz square wave. Not so trivial...

For the first problem, all it needs is to make a counter that counts 10 clock cycles (e.g. 0-9) and toggle the output when it rolls over.

If the same solution is tried for the second problem, the counter would count to 3 or 4, and the frequency obtained is 3.333MHz or 2.500MHz. How to get 3MHz?

The idea is that it is possible to get"on average" 3MHz if the counter counts to 3 twice and counts to 4 once. However, it is best to evenly spread the two values, so that if one needs (for example) 3 seven times and 4 five times, it should not be all 3 $\times$ 7 first and 4 $\times$ 5 next, but rather disperse the 3 and 4 periods among each other.

Another way to do is to add $\frac{6}{20}$ to a counter every clock cycle. Also, the counter rolls over when it reaches 6. It would go as follows (the best places to toggle the 3MHz output are marked with x):

Clock Cycle	Counter	Toggle
0	0.000	x
1	0.300	
2	0.600	
3	0.900	
4	1.200	x
5	1.500	
6	1.800	
7	2.100	x
8	2.400	
9	2.700	
10	3.000	x
11	3.300	
12	3.600	
13	3.900	
14	4.200	x
15	4.500	
16	4.800	
17	5.100	x
18	5.400	
19	5.700	
0	0.000	x

That is, it toggles when the integer portion of the counter increments.

Going further, it may have been noticed that it does not need the integer part at all, i.e. as soon as the count value equals or exceeds 1, we can subtract 1:

Clock Cycle	Counter	Shift
0	0.000	
1	0.300	
2	0.600	
3	0.900	
4	1.200	–> 0.200
5	0.500	
6	0.800	
7	1.100	–> 0.100
8	0.400	
9	0.700	
10	1.000	
11	1.300	–> 0.300
12	0.600	
13	0.900	
14	1.200	–> 0.200
15	0.500	
16	0.800	
17	1.100	–> 0.100
18	0.400	
19	0.700	
20	1.000	–> 0.000

This is essentially how DACQ unit clock (seq_acc signal in VHDL code) comes to approximate the FM transmitter.

In case fractional numbers are not desired, it is just a matter to multiply everything by an integer (here 5), and subtract it when the count equals or exceeds the value selected:

Clock Cycle	Counter	Shift
0	0.000	
1	1.500	
2	3.000	
3	4.500	
4	6.000	–> 1.000
5	2.500	
6	4.000	
7	5.500	–> 0.500
8	2.000	
9	3.500	
10	5.000	
11	6.500	–> 1.500
12	3.000	
13	4.500	
14	6.000	–> 1.000
15	2.500	
16	4.000	
17	5.500	–> 0.500
18	2.000	
19	3.500	
20	5.000	–> 0.000

Since the original ratio (6/20) was based on the two integers, it is always possible to do this. With the DACQ units, the ratio may be irrational (ratio of the true oscillator frequencies), so it was decided to stick with the fixed-point approximation.

Additional explanation for VHDL script

FM receiver

The fm_cmp becomes fm_smp to get around meta-stability in the FPGA latches.

The counter cnt_5us counts from 62 (nominally) to 511 (450 clock cycles at 90MHz is 5μs), it is reinitialised with 61 or 63 if needed to be lengthened or shortened.

The counter cnt_ones is used to decode the raw bits. It counts the number of clock cycles the comparator output is high. If fm_smp is high more than half the 5μs period (cnt_ones > 225) then it is considered to be a '1', otherwise it is considered to be a '0'. It is reset to 31 or 32 according to fm_smp still at 1 or 0, at the end of the 5μs, to stay in tune with cnt_5us that could have been stretched of 1 cycle. To make the detection easier, the counter is started from 31 (b"0_0001_1111"), so that the MSB is set when the number of ones reaches 226.

The resets are every 5μs that correspond to one raw bit of info (1 data bit is 4 raw bits).

Every 5μs, fm_sr and ok_fm are updated with 1 or 0 according to the value of raw bit (cnt_ones(8) is 1 so cnt_ones > 225, which means fm_smp is 1).

Pattern detector

When fm_sr is 00SYNC, fm_sync is set to 1, i.e. every new timestamp and thus frame. This means that fm_sync is 1 for 1 clock cycle very 9.6ms roughly.

This case is also used to reinitialize fm_cnt, fm_cnt is actually 2 counters one from 0 to 3, it is for the 4 raw bits of 1 data bit, and the other one from 0 to 60 for the data bits of the timestamp.

While decoding, every 4 raw bits and before the end of the 60 cycles if the last 4 bits of fm_sr are 1010 the timestamp is shifted with a 1 at the MSB else a 0.

fm_cnt(1 to 0) is 4*5μs or 20μs period, fm_cnt(7 to 2) is 20μs $\times$60 = 1.2ms period. In the actual implementation only 60 data bits are used for the timestamp but it is possible to use up to 479 data bits if desired. Once a valid timestamp is decoded a copy of it is done, as gradually the timestamp is erased with the zeros coming (the shift register keeps operating).

Sequencer counter

The sequencer (sequence counter) controls all the peripheral functions of the FPGA, i.e. DACs, ADCs, PGAs, etc. The DACs require that it have a 1μs period, the ADCs need a 2μs period, the sampler needs a 10μs period, the frames need a 9.6ms period. The counter period needs to be adjustable so that the DACQ can phase-lock to the FM transmitter. It is adjusted using 1 clock cycle per 2μs, or 1 part in 180 (the receiver crystal accuracy is supposedly in the 1 part per 20,000 range). The other consequence is that it needs a counter that can count 89, 90 or 91 clock cycles. The hardware is a comparator and a resettable accumulator. The comparator is slow because it has to compare all the bits in parallel and then AND the bitwise results together. The reset is fast because it's all parallel. So, if it is necessary to add logic, it's best to do it to the reset path, not the comparator. That is why the counters always count up to the same value (all ones) and just reset to different values according to whether a stretch or shrink is needed.

seq_cnt(10 to 8): 0 to 4 is 10μs (5 $\times$ seq_cnt(7)). seq_cnt(20 to 11): 0 to 959 is 9.6ms (960 $\times$ seq_cnt(10 downto 8). seq_cnt(36 to 21): 1 to 65535 is about 10.5 minutes. The upper 16 bits (36 downto 21) are just the frame counter. The 0 value is skipped to avoid ambiguity in the frame data.

Finally, every 9,6ms seq_sync is high for 1 clock cycle.

Phase detector

The counter phs_cnt is incremented by 1 every clock cycle and is reset when fm_sync or seq_sync goes high. Its purpose is to measure the phase error between the FM transmitter and the FPGA's counters.

When phs_cnt is reset phs_lst contains the value of delay between seq_sync and fm_sync so our basic synchronization period is 9.6ms. Reminder: seq_sync is the local sync pulse and fm_sync is the one decoded from the wireless signal. Several cases may appear: if there is no seq_sync pulse between 2 fm_sync, it means the local crystal is too slow; if there is 1 pulse it is fine (expected behavior); and if there are 2, it means the local crystal is running too fast.

Another counter is also used: phs_cur, updated every clock cycle, counts the number of fm_sync since the last seq_sync.

In order to detect error in the sync pulses detection an error checking process is implemented. When seq_sync is 1, the values of phs_cur and phs_prv are compared

and then phs_vld is updated. phs_vld is decremented when seq_sync is 1 and phs_cur is '01' (i.e. fm_sync is 1 too) and sync_prv is not '11' (i.e. the 2 sync pulsed in the same time, with fm_sync pulse started just before) or when phs_curr is not '11' and phs_prv is '01' (both pulses high previous clock cycle and seq_sync not 0 now), else the phs_vls incremented.

The main phase locking is done with seq_acc and phs_inc. That is where the phase detection, error integration (low-pass filtering) and frequency control (via the value of phs_inc) are found.

The increment adjustment of phs_inc (1 integer bit, 18 fractional bits) is done by integrating the error signal. phs_inc is updated with the delay between the sync pulses in number of clock cycles when seq_sync is 1 and fm_sync is 0. The code limits the value of phs_inc to be between 0 and 2 (non-inclusive), nomally 1. The value is computed from checking if phs_lst>phs_cnt, i.e. we are currently closer to the last sync pulse than the previous inter pulse delay) then if the phs_cnt is higher than '111 111 111 111 11' (or 2.88ms) then phs_inc is set to 1, else to '1&phs_cnt'. phs_inc is changed only when seq_sync high so every 9.6ms, then the increment is reported at every seq_cnt(7 to 0) = '1' or 2*mu*s. Thus phs_inc is reported 4800 times and the seq_sync is usually 9.6ms but varies from 9.54667 to 9.6533ms.

The counter seq_acc attempts to track the frame sync period. It is always incremented 4800 times per frame period (i.e. every nominal 2μs). This counter is adjusted by changing the increment value instead of the starting or ending value (as it is done for the sequencer, and following the technique explain in section D) and its two most significant bits are reset to zero every 2μs also. seq_acc is not usable as a sequencer because it does not cycle through all integers. The increment amount is phs_inc, which represents a value between 0 and 2 (nominally 1) in fixed-point. It is adjusted based on the phase error. seq_cnt is slaved to seq_acc. At the end of 2μs, it can't be more than one clock cycle ahead or behind seq_acc, so adjusting the starting value between 37, 38 or 39 is enough to stay slaved.

In other words, seq_cnt goes for another 1μs and then a 1μs ± 1 clock cycle, then 1μs, then 1μs ± 1 clock cycle until seq_sync pulses again and the phs_inc is updated, meanwhile the seq_acc is incremented of the phs_inc.

Appendix E

DACQ frames data format

Additional details for the frames data format are given in this appendix.

Sync

The sync pattern is designed to identify the beginning of a frame and the least versus most significant bytes of each word. By design, a series of three zero bytes can only ever occur at the beginning of a frame, and nowhere else in the data. This characteristic allows the host computer to re-synchronize to the data stream in case of dropped bytes.

Module ID

The module ID is a four character ASCII code uniquely identifying the data acquisition module. The module IDs currently in use are as follows.

Unit type	Module ID(s)
Ultrasound	US40 (40.0kHz)
	US33 (33.3kHz)
	US25 (25.0kHz)
ASU	ASU1
	ASU2
Seismometer	GEO1
	GEO2
	GEO3
Mannequin	ICP1
Gerbil	VBM1

Note that the three ultrasonic units differ in the ultrasound operating frequency (as indicated in the above table), whereas the two ASU units and the three seismometer units are functionally identical.

Frame index

The frame index is a 16 bit integer between 1 and 65535 that increments by one every frame. It is used to detect missing or dropped frames.

Timestamp

The timestamp is a 60 bit integer that is provided by the FM transmitter and also increments by one every frame. However, because of the possibility of bit errors in the radio reception, the timestamp should not be used to detect dropped frames. Note that bit 7 in the most significant byte of each timestamp word is high to prevent confusion with the sync pattern.

PGA and RSSI

The RSSI is the relative signal strength indication from the FM receiver circuit. It is an 8 bit integer between 1 and 255. The PGA is a copy of the programmable gain amplifier setting (see Section E).

Frame minimum

The frame mimimum is a 16 bit integer between 1 and 65535 that is the smallest sample value in the current frame. It is used to quickly adjust the PGA.

Frame maximum

The frame maximum is a 16 bit integer between 1 and 65535 that is the largest sample value in the current frame. It is used to quickly adjust the PGA.

Samples

The samples are 16 bit integers between 1 and 65535. These numbers should be interpreted as signed integers, so that a sample value of 32768 corresponds to 0 Volts, and sample values of 65535 and 1 are the maximum and minimum (respectively) of the channel's dynamic range.

PGA settings

The programmable gain amplifiers can be programmed by sending a single command byte to the data acquisition unit. The format of the command byte is as follows.

G3	G2	G1	G0	CH3	CH2	CH1	CH0

The gain setting bits operate as follows.

G3	G2	G1	G0	Gain
0	0	0	0	1
0	0	0	1	2
0	0	1	0	4
0	0	1	1	8
0	1	0	0	16
0	1	0	1	32
0	1	1	0	64
0	1	1	1	128

The channel setting bits operate as follows. Note that settings other than 0001 are only used for testing and calibration. For these settings, the PGA output (before applying the gain) represent the percentage of the ADC's full scale.

CH3	CH2	CH1	CH0	Channel
0	0	0	0	100%
0	0	0	1	normal
1	1	0	0	0%
1	1	0	1	90%
1	1	1	0	10%
1	1	1	1	50%

Appendix F

Handel-C Scripts of Auditory Pathway Implementation

In this appendix are given the Handel-C scripts used for the full implementation of the cochlea and the simplified TC model.

The first Handel-C script given is the main.
The script starts with variable definitions, and the initialization of the coefficients of the cochlea and the thalamo-cortical modules that depends on the frequency span selected, the number of channels and the number of neurons selected. Then, the cochlea is implemented, i.e. the different filters stages (first and second order IIR). The cochlea output is encoding into spike activity (using an integrate and fire model), which are used as input to the neuron layer A of the TC model. Then, the different layers are implemented (namely B_1 and B_2) and finally, the Fusi& Brader learning rule.

```
#include "Libraries.hch"


void main(void){
    unsigned count_coef i_coef;
    unsigned count_NL0 i;
    unsigned count_NL1 j, ii, jj, i_from, j_to;
    unsigned count_T step_num;
    unsigned count_z k;
    unsigned 3 times, position;

    unsigned FP Aux_dt;

    signed FP dp_cp[max_CP];
    signed FP dp_cq[max_CQ];
    signed FP dp_cs[max_CS];
    signed FP dp_zs[max_ZS];
    signed FP dp_zp[max_ZP];
    signed FP dp_zq[max_ZQ];
    signed FP dp_zr[max_ZR];
    signed FP d_s, d_p, d_r, d_q;
    signed FP tmp, tmp0, tmp1;

    signed FP V_L0[N_L0], I_ext[n_ch], I;
    unsigned 1 Z_L0[N_L0];

    signed FP V_L1[N_L1];
    unsigned 1 Z_L1[N_L1], z_current;

    signed FP V_L2[N_L2];
    unsigned 1 Z_L2[N_L2];

    signed FP a1_01[N_L1], a2_01[N_L1], W_01[N_L1], g_01[N_L1];
    signed FP a1_12[N_L2], a2_12[N_L2], W_12[N_L2], g_12[N_L2];
```

```
signed FP a1_R12[N_L1][N_L2], a2_R12[N_L1][N_L2], W_R12[N_L1][N_L2], g_R12[N_L2];

unsigned count_MD D_R12[N_L1][N_L1], i_MD;
unsigned count_CHMD temp_delays;

unsigned 1 Z_buff[n_ch][MAX_DELAY];
unsigned count_MD  Z_idx, Z_idx2;


//---------------------- Cochlea coefficients initialization--------------------
//-- BM coefficients
i_coef = 0;
do{
    par{
        i_coef++;
        if(i_coef < max_CP){
            input1 ? dp_cp[i_coef[(count_CP - 1):0]];
        }
        else if ((i_coef >= max_CP) && (i_coef < (max_CP + max_CQ))){
            input1 ? dp_cq[(i_coef - max_CP)[(count_CQ - 1):0]];
        }
        else if (i_coef >= (max_CP + max_CQ)){
            input1 ? dp_cs[(i_coef - (max_CP + max_CQ))[(count_CS - 1):0]];
        }
        else{
            delay;
        }
    }
}while(i_coef < n_coef);
//-----------------------------------------------------------------------------

//---------------------------- TC initialization ------------------------------
i = 0;
do{
    //-- initialise the membrane potentials in layer0 between 0 and 1.
    //-- initialise the raster in layer0
    //V_L0[i] = multiply(i, dt);
    Z_L0[i] = 0;
    I_ext[i] = 0;
    i++;
}while(i < N_L0);

j = 0;
do{
    //-- initialise the membrane potentials in layer1  between 0 and 1.
    //-- initialise the raster in layer1
    //-- initialise synapses in layer1
    //-- initialise synaptic weights in connection layer0 - layer1
    //V_L1[j] = multiply(j, dt);
    Z_L1[j] = 0;
    g_01[j] = 0;
    g_R12[j] = 0;
    W_01[j] = (tau << 4);
    W_12[j] = (tau << 2);
    j++;
}while(j < N_L1);

//-- initialise delays
ii = 0;
do{
    jj = 0;
    do{
        i_MD = (ii < jj ? 0@(jj - ii) : 0@(ii - jj));
        temp_delays = 0@i_MD;
        temp_delays *= MAX_DELAY;
        temp_delays /= N_L1;
        i_MD = temp_delays[(count_MD - 1):0];
        D_R12[ii][jj] = i_MD;
        output1 ! D_R12[ii][jj];
        jj++;
    }while(jj < N_L2);
    ii++;
}while(ii < N_L1);

ii = 0;
do{
    jj = 0;
```

```
    do{
        a1_R12[ii][jj] = 0;
        a2_R12[ii][jj] = 0;
        W_R12[ii][jj] = (ii != jj ? (tau >> 2) : 0);
        output2 ! W_R12[ii][jj];
        jj++;
    }while(jj < N_L2);

    Z_idx = 0;
    do{
        Z_buff[ii][Z_idx] = 0;
        Z_idx++;
    }while(Z_idx < MAX_DELAY);
    ii++;
}while(ii < N_L1);

step_num = 0; Z_idx2 = 0;
do{
    times = 0;
    do{
        input2 ? d_s;

        k = 0;
        do{
            tmp0 = MULT2(dp_cs[(k[(count_CS - 1):0]) * N_CS], dp_zs[(k[(count_ZS - 1):0]) * N_ZS]);
            tmp0 += d_s;
            tmp1 = dp_zs[(k[(count_ZS - 1):0]) * N_ZS] + tmp0;
            d_s = MULT2(tmp1, dp_cs[1 + (k[(count_CS - 1):0]) * N_CS]);
            dp_zs[(k[(count_ZS - 1):0]) * N_ZS] = tmp0;

            k++;
        }while(k < max_ZS);
        d_p = d_s;

        // Process each channel:
        i = 0;
        do{
            d_r = 0;
            // First-order IIR filter section
            tmp0 = MULT2(dp_cp[(0@i)* N_CP], dp_zp[(i[(count_ZP - 1):0]) * N_ZP]) + d_p;
            tmp1 = dp_zp[(i[(count_ZP - 1):0]) * N_ZP] + tmp0;
            d_p = MULT2(tmp1, dp_cp[1 + (0@i) * N_CP]);
            dp_zp[(i[(count_ZP - 1):0]) * N_ZP] = tmp0;

            // A cascade of two 2nd-order IIR filter sections
            // (First IIR:)
            tmp0 = MULT2(dp_cq[(0@i) * N_CQ], dp_zq[(0@i) * N_ZQ]);
            tmp = tmp0 + MULT2(dp_cq[1 + (0@i) * N_CQ], dp_zq[1 + (0@i) * N_ZQ]) + d_p;
            tmp0 = MULT2(dp_cq[3 + (0@i) * N_CQ], dp_zq[1 + (0@i) * N_ZQ]);
            d_q = tmp0 + MULT2(dp_cq[2 + (0@i) * N_CQ], tmp);

            dp_zq[1 + (0@i) * N_ZQ] = dp_zq[(0@i) * N_ZQ];
            dp_zq[(0@i) * N_ZQ] = tmp;

            // (Second IIR:)
            tmp0 = MULT2(dp_cq[(0@i) * N_CQ], dp_zr[(0@i) * N_ZQ]);
            tmp = tmp0 + MULT2(dp_cq[1 + (0@i) * N_CQ], dp_zr[1 + (0@i) * N_ZQ]) + d_q;

            tmp0 = MULT2(dp_cq[3 + (0@i) * N_CQ], dp_zr[1 + (0@i) * N_ZQ]);
            d_r = tmp0 + MULT2(dp_cq[2 + (0@i) * N_CQ], tmp);

            dp_zr[1 + (0@i) * N_ZQ] = dp_zr[(0@i) * N_ZQ];
            dp_zr[(0@i) * N_ZQ] = tmp;

            //tmp0 = MULT2(d_r, scale_factor); // the output is scaled
            tmp0 = d_r >> 3; // scaled by a factor of 0.125
            tmp0 *= (tmp0 < 0 ? -1 : 1); // Full wave rectification
            output3 ! tmp0; // save the Cochlea output

            I_ext[i] += tmp0;

            i++;
        }while(i < n_ch); // -- end of the channel loop

        times++;
    }while(times < 3); //-- end of the BM loop
```

```
//-- Here is where the TC model starts
i = 0;
do{
    //--- From a cochleagram to a spiking activity
    I = I_ext[i]/3;
    I_ext[i] = 0;
    tmp = UNO - dt;
    tmp0 = MULT2(V_L0[i], tmp);
    tmp1 = MULT2(I, dt);
    V_L0[i] = tmp0 + tmp1;
    output4 ! V_L0[i];

    Z_L0[i] = (V_L0[i] < firing_threshold ? 0 : 1);
    output5 ! Z_L0[i];
    V_L0[i] = (Z_L0[i] ? 0 : V_L0[i]);

    //-----------------------------------------------//

    //- Synaptic activity (layer 0 - layer 1)

    tmp = UNO - dtaur;
    tmp0 = MULT2(tmp, a1_01[i]); // --- checar porque el indice de a1 era 1
    tmp1 = (Z_L0[i] ? MULT2(W_01[i], dtaur) : 0);
    a1_01[i] = tmp0 + tmp1;

    tmp = UNO - dtauf;
    tmp0 = MULT2(tmp, a2_01[i]);
    tmp1 = MULT2(a1_01[i], dtauf);
    a2_01[i] = tmp0 + tmp1;

    g_01[i] = a2_01[i];
    output6 ! g_01[i];

     //-----------------------------------------------//

    //- Layer 1
    tmp = UNO - dt;
    tmp0 = MULT2(tmp, V_L1[i]);
    tmp = g_01[i];
    tmp1 = MULT2(tmp, dt);
    V_L1[i] = tmp0 + tmp1;
    output7 ! V_L1[i];

    Z_L1[i] = (V_L1[i] < firing_threshold ? 0 : 1);
    output8 ! Z_L1[i];
    V_L1[i] = (Z_L1[i] ? 0 : V_L1[i]);

    tmp = UNO - dtaur;
    tmp0 = MULT2(tmp, a1_12[i]);// --- checar porque el indice era 1
    tmp1 = (Z_L1[i] ? MULT2(W_12[i], dtaur) : 0);
    a1_12[i] = tmp0 + tmp1;

    tmp = UNO - dtauf;
    tmp0 = MULT2(tmp, a2_12[i]);
    tmp1 = MULT2(a1_12[i], dtauf);
    a2_12[i] = tmp0 + tmp1;

    g_12[i] = a2_12[i];
    output9 ! g_12[i];

    Z_buff[i][Z_idx2] = Z_L1[i];

     //-----------------------------------------------//

    //- Layer 2
    tmp = UNO - dt;
    tmp0 = MULT2(tmp, V_L2[i]);
    tmp = g_01[i] - g_12[i] + g_R12[i];
    tmp1 = MULT2(tmp, dt);
    V_L2[i] = tmp0 + tmp1;
    output10 ! V_L2[i];

    Z_L2[i] = (V_L2[i] < firing_threshold ? 0 : 1);
    output11 ! Z_L2[i];
    V_L2[i] = (Z_L2[i] ? 0 : V_L2[i]);

    i++;
```

```
}while(i < N_L0); //end of the neuron loop

//----------------- Learning Rule -------------------------
i_from = 0;
do{
    j_to = 0;
    do{
        if(step_num > 0@D_R12[i_from][j_to]){

            z_current = Z_buff[i_from][Z_idx2 - D_R12[i_from][j_to]];

            if(z_current){
                if(V_L2[j_to] > THETA_V){
                    W_R12[i_from][j_to] += UP_STEP;
                }
                else{
                    W_R12[i_from][j_to] -= DOWN_STEP;
                }
            }

            tmp = UNO - dtaur;
            tmp0 = MULT2(tmp, a1_R12[i_from][j_to]);
            tmp1 = (z_current ? MULT2(W_R12[i_from][j_to], dtaur) : 0);
            a1_R12[i_from][j_to] = tmp0 + tmp1;

            tmp = UNO - dtauf;
            tmp0 = MULT2(tmp, a2_R12[i_from][j_to]);
            tmp1 = MULT2(a1_R12[i_from][j_to], dtauf);
            a2_R12[i_from][j_to] = tmp0 + tmp1;

        }

        if(W_R12[i_from][j_to] > THETA_W){
            W_R12[i_from][j_to] += MULT2(dt, DRIFT_UP_RATE);
        }
        else{
            W_R12[i_from][j_to] -= MULT2(dt, DRIFT_DOWN_RATE);
        }

        j_to++;
    }while(j_to < N_L2);
    i_from++;
}while(i_from < N_L1);


//-----------------------------------------------------


/*i_from = 0;
do{
    j_to = 0;
    do{
        if(step_num > 0@D_R12[i_from][j_to]){

            z_current = Z_buff[i_from][Z_idx2 - D_R12[i_from][j_to]];

            tmp = UNO - dtaur;
            tmp0 = MULT2(tmp, a1_R12[i_from][j_to]);
            tmp1 = (z_current ? MULT2(W_R12[i_from][j_to], dtaur) : 0);
            a1_R12[i_from][j_to] = tmp0 + tmp1;

            tmp = UNO - dtauf;
            tmp0 = MULT2(tmp, a2_R12[i_from][j_to]);
            tmp1 = MULT2(a1_R12[i_from][j_to], dtauf);
            a2_R12[i_from][j_to] = tmp0 + tmp1;
        }
        j_to++;
    }while(j_to < N_L2);
    i_from++;
}while(i_from < N_L1);*/

ii = 0;
do{
    g_R12[ii] = 0;
    jj = 0;
    do{
        g_R12[ii] += a2_R12[jj][ii];
```

```
            jj++;
        }while(jj < N_L1);

        output12 ! g_R12[ii];
        ii++;
    }while(ii < N_L1);

    //- ring buffer
    if(step_num >= (MAX_DELAY - 1)){
        j = 0;
        do{
            Z_idx = 1;
            do{
                Z_buff[j][Z_idx - 1] = Z_buff[j][Z_idx];
                Z_idx++;
            }while(Z_idx < MAX_DELAY);
            j++;
        }while(j < N_L2);
    }

    step_num++;
    Z_idx2 += (step_num < MAX_DELAY ? 1 : 0);
}while(step_num < SIM_LENGTH); // -- end of the simulation loop

ii = 0;
do{
    jj = 0;
    do{
        output13 ! W_R12[ii][jj];
        jj++;
    }while(jj < N_L2);
    ii++;
}while(ii < N_L1);


}
```

The second Handel-C script given is the library definition file.
It contains the definition and initialization values of all the variables used in the previous scripts.

```
//_____________________________________________________________________________________________

//--General parameters

set clock = external "P1";                  // - clock signal

macro expr LOG2_FRAC = 16;                  // - Number of bits in the fractional part
macro expr LOG2_INT  = 16;                  // - Number of bits in the integer part
macro expr FP        = (LOG2_INT + LOG2_FRAC);  // - Word size
macro expr LSB       = (LOG2_FRAC);                // - Less Significant Bit
macro expr MSB       = (LOG2_INT + 2 * LOG2_FRAC - 1);  // - More significant Bit
macro expr UNO       = 1 << LOG2_FRAC;

//_____________________________________________________________________________________________

//-- Cochlea parameters

// - Filters coeficients
macro expr N_CS = 2;
macro expr N_ZS = 1;
macro expr N_CP = 2;
macro expr N_ZP = 1;
macro expr N_CQ = 4;
macro expr N_ZQ = 2;

macro expr n_ch       = 30;             // - number of channels
macro expr count_nch  = 5;              // - counter for n_ch

macro expr fs       =  32000;   // - number of inputs
macro expr count_fs  =  15;     // - counter for n_in

macro expr ts       =  10240;   // - number of inputs
macro expr count_ts  =  14;     // - counter for n_in
```

```
macro expr SIM_LENGTH = (ts * 10000)/fs;
macro expr count_T = 12;

macro expr DT_RATIO = 209715;

macro expr scale_factor = 1 << LOG2_FRAC;

macro expr n_coef     =  ((N_CP + N_CQ) * n_ch + N_CS * 3);     // - number of coefficients
macro expr count_coef =  8;                                     // - counter for n_coef

macro expr n_z        = ((N_ZP + N_ZQ) * n_ch + N_ZS * 3);      // - total number of variables
macro expr count_z    = 7;                                      // - counter for variables

// - counters for coefficients
macro expr count_CS = 3;
macro expr count_ZS = 2;
macro expr count_CP = 6;
macro expr count_ZP = 5;
macro expr count_CQ = 7;
macro expr count_ZQ = 6;

// - maximum number of values per coefficients vector
macro expr max_CS = N_CS * 3;
macro expr max_ZS = N_ZS * 3;
macro expr max_CP = N_CP * n_ch;
macro expr max_ZP = N_ZP * n_ch;
macro expr max_CQ = N_CQ * n_ch;
macro expr max_ZQ = N_ZQ * n_ch;
macro expr max_ZR = N_ZQ * n_ch;

chanin input1 with {infile = "C:/Implementations/MATLAB/HC_sim_interface/BM_coefficients.dat"};
chanin input2 with {infile = "C:/Implementations/MATLAB/HC_sim_interface/Acoustic_stimulus.dat"};

//_____________________________________________________________________________________________

//-- IFN parameters

macro expr N_L0        = 30;            // - number of neurons in layer 0
macro expr count_NL0   = 5;             // - counter for n_neurons  in layer 0

macro expr N_L1        = 30;     // - number of neurons in layer 1
macro expr count_NL1   = 5;     // - counter for n_neurons  in layer 1

macro expr N_L2        = 30;     // - number of neurons in layer 1
macro expr count_NL2   = 5;     // - counter for n_neurons  in layer 1

macro expr MAX_DELAY   = 200;
macro expr count_MD    = 8;

macro expr MAX_DELAY_STEPS  = MAX_DELAY * 10;

macro expr nch_MD = n_ch * MAX_DELAY;
macro expr count_CHMD = 13;

macro expr tau = 1 << LOG2_FRAC;
macro expr firing_threshold = 1 << LOG2_FRAC;
//macro expr dt  = (tau >> 4) + (tau >> 5) + (tau >> 7);
macro expr dt  = 6554;

macro expr taur = (tau >> 1);
macro expr tauf = (tau << 1);

macro expr dtaur = (tau >> 3) + (tau >> 4);
macro expr dtauf = (tau >> 4);


//_____________________________________________________________________________________________

//-STDP Parameters

macro expr THETA_V          = 26214;   // = 0.4
macro expr UP_STEP          = 13107;   // = 0.2
macro expr DOWN_STEP        = 16384;   // = 0.25
macro expr THETA_W          = 19661;   // = 0.30
macro expr DRIFT_UP_RATE    = 3277;     // = 0.05
macro expr DRIFT_DOWN_RATE  = 66;       // = 0.001
```

```
//macro expr DRIFT_UP_RATE    = 328;     // = 0.005
//macro expr DRIFT_DOWN_RATE  = 7;       // = 0.0001

//_____________________________________________________________________________________________________

chanout output1 with {outfile = "Delays.dat"};
chanout output2 with {outfile = "Weights.dat"};
chanout output3 with {outfile = "BMoutput.dat"};
chanout output4 with {outfile = "IFNoutput.dat"};
chanout output5 with {outfile = "IFNZoutput.dat"};
chanout output6 with {outfile = "alpha.dat"};
chanout output7 with {outfile = "IFNL1output.dat"};
chanout output8 with {outfile = "IFNZL1output.dat"};
chanout output9 with {outfile = "alpha1.dat"};
chanout output10 with {outfile = "IFNL2output.dat"};
chanout output11 with {outfile = "IFNZL2output.dat"};
chanout output12 with {outfile = "alpha2.dat"};
chanout output13 with {outfile = "Weights_after_STDP.dat"};

//_____________________________________________________________________________________________________

//-- General functions

// - Multiplication function
signed MULT2(signed a, signed b){
    unsigned 1 sign;
    signed (FP + FP) Aux;
    signed FP c;

    sign = (a < 0 ? 1 : 0);
    sign ^= (b < 0 ? 1 : 0);

    a *= (a < 0 ? -1 : 1);
    b *= (b < 0 ? -1 : 1);

    Aux = (0@a) * (0@b);
    c = Aux[MSB:LSB];
    c *= (sign ? -1 : 1);

    return c;
}
macro expr multiply(x, y) = select(width(x) == 0, 0, multiply(x \\ 1, y << 1) + (x[0] == 1 ? y : 0));

//_____________________________________________________________________________________________________
```