This study chronicles the development of an Embedded System for Prosthetic Application (ESPA), used to collect, process, and decode neural and other bioinformatic data at high-data rates, and subsequently generate control signals for prosthetic use. Key features of the system include being battery operated, portable, wearable, and having compute power to process up to 1000 channels of broadband 30ksps neural data in addition to multiple standard interfaces for communication with prosthetic devices, clinician monitors, and other peripherals. ESPA also includes the ability for researchers and clinicians to monitor and update the processing model in real-time. The platform chosen for ESPA is the Xilinx Zynq-7000 series System on Chip (SoC), boasting a dual-core Advanced RISC Machine (ARM) microprocessor and a large Field Programmable Gate Array (FPGA) fabric space. This power combination allows for ease of interface and data routing software development alongside custom, dedicated, parallelizable neural data-processing algorithms implemented in FPGA fabric. QNX, a medical grade operating system, runs on the ARM to manage data I/O and hardware interfaces, while a reconfigurable data processing model runs in FPGA fabric. This processing model, the focus of this study, is designed using the System Generator architecture and consists of three primary functions. "Feature Extraction" conditions neural signals to extract meaningful nuggets of data, "Decoding" uses algorithms to turn these features into prosthetic control signals, and the "Parameter Engine" provides a conduit for external control of the processing model by automated programs and clinicians. ESPA has been demonstrated using two 96-channel neural recording devices, and has demonstrated Local Field Potential (LFP) and Action Potential (AP) Thresholding feature extraction capabilities, along with a Linear Discriminant Analysis (LDA) neural state decoder. The system has demonstrated streaming raw neural as well as processed data to a back-end server both over wired Ethernet and wirelessly via 802.11n. In conclusion, unique form-factor and computational power will enable ESPA to become a platform that will further both research and therapeutic benefits to institutions and individuals conducting neural-prosthetic control experiments and therapies.
Toward a Mobile Computing Platform for Neural Signal Processing

by

Jacob Komar
B.S., University of Hartford, 2009

A dissertation submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in the Department of Electrical Engineering at Brown University

Providence, Rhode Island
May 2016
This dissertation by Jacob Komar is accepted in its present form by the Department of Electrical Engineering as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date ____________

Dr. Arto V. Nurmikko, Co-Director

Date ____________

Dr. John Simeral, Co-Director

Recommended to the Graduate Council

Date ____________

Dr. Jerry Daniels, Reader

Approved by the Graduate Council

Date ____________

Dr. Peter M. Weber
Dean of the Graduate School
Jacob Komar
CV

Education

2009 – Present  Doctor of Philosophy
               Electrical Engineering
               Brown University
               Providence, RI

2009 – 2011   Master of Science
               Electrical Engineering
               Brown University
               Providence, RI

2005 – 2009   Bachelor of Science
               Computer Engineering
               University of Hartford
               Hartford, CT
Posters

Annual Society for Neuroscience Conference Chicago, IL – 2015
San Diego, CA – 2013

Annual IEEE Engineering in EMBS Conference on Neural Engineering (NER) San Diego, CA – 2013

EMBS BRAIN Grand Challenges Conference Washington, DC – 2014

Annual IEEE Engineering in Biomedical Circuits and Systems Conference Atlanta, GA – 2015

Relevant Publications


Research Interest

My research interests are geared toward the development of marketable brain-machine interface systems, both for immediate therapeutic use and for eventual full-market commercial success. I am interested in using SoC platforms to aggregate whole-body biological data (including neural data) to both learn more about the human body, as well to assist in augmenting our digital experience.

Work Experience

Sept 2010 – May 2014
Teaching Assistant
Brown University

Served as a lab teaching assistant for a Bioinstrumentation course. Assisted with LabVIEW code generation and debugging, explaining basic electronics concepts, took preparedness quizzes, and provided general assistance to students.

November 2010 – Present
Volunteer, Field Technical Advisor
FIRST Robotics

Volunteer at several events per year, lending my time and technical skills to help facilitate the operation of FIRST FRC robotics competitions all over the world, impacting thousands of high school students and promoting STEM education globally.

August 2001 – June 2010
Founder, CEO
Computers for Communities

Founded and operated a 501(c)3 non-profit organization aimed at bridging the digital divide in America. Over the existence of the program, we refurbished and donated thousands of computers and provided IT Training skills to hundreds of underserved students.
Awards

Academic
University of Hartford President’s Scholar Award (4 years)
Phi Theta Kappa

Community Service
BR!CK Award (DO SOMETHING)
Christopher Reeve Scholarship Award (Heart of America Foundation)
Magic Maker of the Year (3rd Street Community Foundation)
National Caring Award
Prudential Spirit of Community Award
Points of Light Award
Kohl’s Kids Who Care Winner
AngelSoft “Angels in Action” Winner
WFSB’s Every Day Hero

Professional Training
Laboratory Safety Training Hazardous Waste Safety Training VA Privacy and Information Security Awareness and Rules of Behavior Privacy and HIPAA Training

Electronic Design
- PCB Level: Miniaturized wireless charging board for implantable devices
- System Level: Multiple Arduino based interactive projects
  - Multiple ZigBee based small, battery powered wireless projects
- Instrumentation: Behavioral porcine feeding tray
  - Robotic prosthetic hand testing rig

Technical Experience
802.11 Analysis & Optimization
Milling/CNC/Machining
Microwire Bonding
3D Printing
Microsoldering
Printed Circuit Board Assembly

Computer Skills

Intermediate Knowledge
- JAVASCRIPT,
- Solidworks, Python
- ADOBE AFTER EFFECTS

Advanced Knowledge
- ADOBE PHOTOSHOP, HTML, LTEX,
- Vivado, SIMULINK,
- VHDL, Verilog, C++, Java,
- MATLAB, LabVIEW
- ADOBE PHOTOSHOP, ADOBE ILLUSTRATOR,
- Server Administration & Virtualization
Acknowledgments

First and foremost, I would like to thank my family. My parents, Andy and Alicia Komar have always been there for me, always trying to help me figure out how to accomplish my goals while allowing me to grow and make mistakes on my own. Their support, their sense of adventure, and their often sage advice has not only allowed me to get where I am at this tender age of 23, but has truly led me to a never-ending sense of fulfillment in my life. My sister Ana has always been there for me when I’ve needed her, even if that meant through a Skype call on the other side of the globe (as we so often are.) Our relationship has matured over the years, and we both know that we can always count on each other, which is something that I value greatly.

I can’t thank Dr. Arto Nurmikko enough. As my PI, mentor, believer, financier, and all-too-often life coach, Professor Nurmikko truly served to inspire me to continue down the path of knowledge, teaching me all the while to take advantage of life around me, and never ceasing to test my boundaries of thinking through a level of inquiry and thought about the unknown that I’ve never encountered before. As much as I dreaded waking up for (or more often, staying up for) our weekly lab meetings, his cheery anecdotes about his incredible life always made it worth it. Not only is Dr. Nurmikko one of the greatest intellectuals I’ve ever met, he truly is someone who has lived and continues to live his life to the fullest, a quality that I truly admire.

Dr. John Simeral was a peripheral player in my experience at Brown for the first few years—once the ESPA project really got off the ground, he became as important of a mentor, motivator, and technical anchor as Dr. Nurmikko. As a project organizer, his insight and agility about the resources available to us has always made our project move forward as quickly as Chris and I could keep up with it. As a mentor, John has always been an incredible resource to try to figure out how to handle a difficult situation, and I really appreciate that. Thank you John.
The relationship between Dr. Jerry Daniels, my third committee member, and myself began when I TA’d his class my second year at Brown. Over the years, as I continued to TA for him, our bond grew, and through our shared passion for robotics, we’ve maintained a relationship outside of the classroom, including through his FTC teams. JD has been someone who has always been there when I need something, whether it’s an IC or a thought experiment about how a mechanism might work.

Chris Heelan has been my partner in crime for the last five years. Working together on ESPA almost since his arrival, being able to draw a line in the sand and say STOP in order to write this dissertation has been a difficult task for both of us. I really appreciate his help and support over the years for making this project a reality, and it honestly couldn’t have happened without his help. To him, I wish him great luck in finishing out his portion of the project and look forward to congratulating him on his own dissertation in the future.

Many current and past members of the Nurmikko Lab have been a blast to work with over the years, including Dr. Dave Borton, Dr. Andrew Blaeser, Dr. Sunmee Park, Dr. Kwangdong Roh, Dr. Jing Wang, Dr. Cuong Dang, Dr. Joonhee Lee, Dr. Ilker Ozden, Dr. Farah Laiwalla, Dr. Yao Lu, Marc Powell, Nicole Dusang, and Dmitrijs Celinski just to name a few. I also had the pleasure of working with many folks over at 2 Stimson, including Dr. Beata Jarosiewicz, collaborator Dr. Carlos Vargas-Irwin, Dan Bacher, Anish Sarma, Dave Rosler, and many others.

Though Juri Minxha was only around for my first two years of study at Brown, he truly helped prepare me for what was to come. His insight into technical fields both in the electrical and neural realm helped jump-start my education, and the late nights spent preparing for meetings and making Visio diagrams assisted my integration into the lab.

To all of my FIRST robotics friends, both in Manchester, NH, as well as all over the world, I can’t thank you enough. To Matt and Kate Pilotte, my adoptive family north of Boston, I can’t thank you enough for your support over all these years. To Alex Herreid, Liz Smith, Derek Foster, Alisha Wallenstein (and Rob and JD!), Steve Krawic, Steve Sargent, Blair Hundertmark, Shahar Givol, Frank Merrick, Marc Polansky, Tyler Holtzman, Raanan de Haas, Dafna Koby, Collin and Jenny Fultz, Nick Skripsky, Kevin O’Connor, Ryan Foley, Jamee Luce, Jerry Budd, Jack
and Kathy Kentfield, Jonathan Bryant, Danny Diaz, Greg McKaskle, Tim "TBD" Bennington-Davis, Gary Hilbert, Bob Caruso, Al Skierkiewicz, Andy Baker, Mark Koors, Libby Kamen and countless others, you have served as true inspiration to me over the last seven years, and I always appreciate your support.

To my close friends who have been there to support me, even through my crazy antics, Ana Carolina Campos Medina, David Salsone, Lucas Serra, Peter Mans, Rebecca Cohen, Spany Reddi, Tyler Connelly, Jen Ky, Muryum Khalid, Steve Horbachuk, Sofya Zeylikman, and Dr. Martin Leftoff, thank you!

To Tim Madore, my lifelong best friend, who has always been there when I’ve needed him – I’m truly appreciative.

And finally, to Naubahar Agha, my (certainly better) other half in Providence. From going to lunch on weekdays to spending holidays playing video games together; from going on trips all over the world to getting in a car accident less than a mile from home together; from spending a continuous 24-hours in a monkey room to proofreading this thesis; you’ve always had my back, and you’ve always been my best friend at Brown. I don’t know what the future holds in store for either of us, but I do know that we share a life-long bond. Thank you for everything you’ve done and continue to do in my life.
Dedication

This work is dedicated to my parents, Andy and Alicia Komar, whom inspired my love of figuring things out – while always doing the right thing.
Preface

“What is real? How do you define ‘real’? If you’re talking about what you can feel, what you can smell, what you can taste and see, then ‘real’ is simply electrical signals interpreted by your brain.”

– Morpheus, The Matrix, 1999

We live in a world that, for some time, has been governed by Moore’s Law. Even if the shape of the curve doesn’t hold steady in the near-term future, the ideology behind the law continues to push the capability of our machines to new lengths, all the time. Personally, I’m a sucker for history surrounding both the first and second Industrial Revolution, because it serves as a reminder not only about what humans can accomplish in such a short span of time, but more importantly that those actions and developments can have broad, incredibly sweeping impacts on everyone living in our society, and in fact, on society as a whole.

Beginning with the harnessing of mechanical power, followed by electrical power, we saw developments throughout the 19th, 20th and 21st century that have drastically improved quality of life, allowed populations to explode, and has created a social structure that has allowed research to flourish – sometimes at the behest of something else, and sometimes just for the sake of research. Two incredible facts have always stood out to me:

• From the time of the first powered flight in 1903 by the Wright brothers in their glider with a home-cast engine and a frame made of cotton and wood, only 8 years elapsed before the first Airmail service went into effect in a commercial airplane in 1911.

• From the time that the very first piece of “space junk” (satellite) launched by humans into space (the Sputnik 1 in 1957), only 12 years elapsed before a human
was sent to the moon, landed, walked around a bit, and safely came home in 1969.

When toiling away in a lab, working on the next generation of devices or products or research in a particular field, it can sometimes be frustrating; but remembering not only breakthrough accomplishments such as these, but the short amount of time in which it can take a breakthrough to transform into something that changes society forever can really be a motivating solace.

For the past few decades, incredible work has been done on making our computing devices faster, smaller, more power efficient, and easier to use. Even more recently, effort has been put into creating “better” or more “natural” interface devices for our computing systems – where mice and keyboards dominated computers of the past, touch screens, 3D glasses, virtual reality goggles and interaction wands are no longer a vision by Computer Science experts, they are here and now in our living rooms.

However, these sorts of interfaces have been somewhat of a “natural progression” from the tech of old. The holy grail of human-machine interface has always been a direct connection into the brain – fusing them together, creating true, “natural” bionics. Movies like Tron, The Terminator and The Matrix have inspired a generation, with the public yearning for the fantasy of this sort of media to become reality. I know that I was personally inspired by such films – it seems so easy, right? Our brains are just an extraordinary complex electrical circuit... Plug some wires in, figure out the code, talk to the brain!

If only it were that easy...

Serious efforts at Brown University have been made to help further research to make this sort of dream a reality, both on engineering and clinical fronts. In particular, the Nurmikko lab first developed the Brain Implantable Chip (or BIC), a small, soft, flexible implantable device which, through the use of a laser, optically transmitted broadband neural data from a sensor embedded in the cortex of the brain through the skin. Future devices, such as the EBNC and SNBC, used radio power to transmit the same data from similar sensors. On the clinical side, the BrainGate group has, and is conducting implantable neurosensing device clinical trials in tetraplegic humans,
with the aim of improving quality of life through prosthetic control of computers and robots using neural signals.

The focus of this dissertation is on a research device, known as the ESPA, which is an essential link between these two realms. ESPA aims to help connect existing and new neural implants with prosthesis, using new, compact, battery-operated wearable technology to do all of the data processing necessary to make this type of system a reality.

“Never send a human to do a machine’s job.”

– Agent Smith, The Matrix, 1999
Contents

List of Tables xix

List of Figures xx

1 Introduction 2
   1.1 Brain Computer Interfaces for Prosthetic Control 2
   1.2 Neural Signal Conditioning and Decoding 3
   1.3 Requirements and Design Constraints for an Embedded System for
       Prosthetic Application 5
   1.4 ESPA System Design Overview 6
   1.5 FPGA Software Development 7
       1.5.1 Interface Specifications and Protocol Definition 9
   1.6 Validation of System Operation 10
   1.7 Taking ESPA Forward Toward Experimental Integration 10

2 ESPA System Architecture 13
   2.1 ESPA As A Platform 13
   2.2 Zynq 7000 SoC 14
   2.3 ESPA Supervisor 16
   2.4 Server Side Application 16
   2.5 End Effectors 18

3 FPGA Software Development 21
   3.1 FPGA Overview 21
       3.1.1 FPGA System Architecture 21
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.2 Resource Allocation</td>
<td>23</td>
</tr>
<tr>
<td>3.1.3 Conventions</td>
<td>26</td>
</tr>
<tr>
<td>3.2 Input Glue Logic and Signal Preparation</td>
<td>28</td>
</tr>
<tr>
<td>3.3 Feature Extraction</td>
<td>31</td>
</tr>
<tr>
<td>3.3.1 Overview</td>
<td>31</td>
</tr>
<tr>
<td>3.3.2 Spike Counter</td>
<td>34</td>
</tr>
<tr>
<td>3.3.3 FIR Decimator 15k</td>
<td>36</td>
</tr>
<tr>
<td>3.3.4 Common Average Reference</td>
<td>39</td>
</tr>
<tr>
<td>3.3.5 Thresholding (Tx) Filter</td>
<td>40</td>
</tr>
<tr>
<td>3.3.6 Thresholder</td>
<td>42</td>
</tr>
<tr>
<td>3.3.7 timeResolvedSelector</td>
<td>45</td>
</tr>
<tr>
<td>3.4 Decoding</td>
<td>48</td>
</tr>
<tr>
<td>3.4.1 Neural Decoding Overview</td>
<td>48</td>
</tr>
<tr>
<td>3.4.2 Decoding Schemes</td>
<td>49</td>
</tr>
<tr>
<td>3.4.3 LDA Classifier Implementation</td>
<td>52</td>
</tr>
<tr>
<td>3.4.4 Class Means</td>
<td>52</td>
</tr>
<tr>
<td>3.4.5 Time Shifting Paradigm</td>
<td>54</td>
</tr>
<tr>
<td>3.4.6 Probability Calculations</td>
<td>56</td>
</tr>
<tr>
<td>3.4.7 Classifier</td>
<td>57</td>
</tr>
<tr>
<td>3.5 Parameter Engine</td>
<td>59</td>
</tr>
<tr>
<td>3.5.1 Overview</td>
<td>59</td>
</tr>
<tr>
<td>3.5.2 Enables</td>
<td>65</td>
</tr>
<tr>
<td>3.5.3 Data Loading</td>
<td>65</td>
</tr>
<tr>
<td>3.5.4 Stored Parameters</td>
<td>68</td>
</tr>
<tr>
<td>3.6 Data Aggregation &amp; Output Glue Logic</td>
<td>69</td>
</tr>
<tr>
<td>4 Discussion</td>
<td>73</td>
</tr>
<tr>
<td>4.1 Validation</td>
<td>73</td>
</tr>
<tr>
<td>4.1.1 SIMULINK Functionality Verification</td>
<td>73</td>
</tr>
<tr>
<td>4.1.2 Vivado Simulation</td>
<td>76</td>
</tr>
<tr>
<td>4.1.3 On-Chip Verification</td>
<td>79</td>
</tr>
<tr>
<td>4.2 Requirements for Future Components</td>
<td>80</td>
</tr>
<tr>
<td>4.2.1 Custom ESPA Hardware</td>
<td>80</td>
</tr>
</tbody>
</table>

xvii
# List of Tables

3.1 Available FPGA Resources on the Xilinx Zynq Z-7045, XC7Z045. Data extracted and reformatted from[1]. .......................................................... 24

3.2 Color key for time domains in the SysGen model. "Discrete 1", denoted in red, is the time domain supplied by the neural implant(s), supplied by the write_to_model glue piece. As this data gets decimated, slower clocks are used to run pieces of model hardware based on those decimation rates. Green denotes the input clock divided by 2, dark blue by 3, and light blue by 6. .................................................. 28

3.3 OP-Codes available for the Parameter Engine. ........................................... 61

B.1 CAR Loader State Machine. X = Valid Opcode, Y = Counter Complete. Output = Reset, Enable .............................................................. 94

B.2 30k Filter State Machine. X = Valid Opcode, Y = Counter Complete. Output = Reset, Enable, Load, cValid, rValid, rLast (decimal) .... 94

B.3 Multiply State Machine. X = Valid Opcode, Y = Counter Complete. Output = Reset, Enable .............................................................. 95

B.4 Multiply State Machine. X = FIFO Valid, Y = Valid Counter. Output = Reset, Enable .............................................................. 95

B.5 Expectation Accumulator State Machine. X = Start. Output = Next State, Enable(1b), Mux1(2b), Mux2(2b) (decimal) ......................... 96

B.6 Input Vector CAR .............................................................. 97
List of Figures

1.1 A Blackrock silicon MEA with CerePort connector. Used with permission from [3] ................. 3
1.2 Anatomy of a “typical” Action Potential – plot shows the various phases of the AP in terms of voltage as it passes a given point on the cell membrane, whose resting potential is about -70mV. After a stimulus is applied at t=1ms, depolarization occurs, raising the membrane potential above the threshold of -55mV. The membrane potential reaches a peak of around +40mV before rapidly repolarizing at t=2ms. The potential overshoots to -90mV, leading to a refractory period lasting until t=5ms, when the resting potential of -70mV is yet again reestablished. Used with permission from [7] ............... 4
1.3 Block diagram of patient utilizing ESPA system. The patient on the left (in an existing system) has a wired, broadband neural implant connected to a cart with neural data-processing hardware and computers which in turn control end-effectors. On the right is a system with ESPA in the loop; capable of communicating with a wireless neural implant, doing all data processing on-board, and wirelessly communicating with various end-effectors. .................................................. 6
1.4 An example of a System Generator IP Core GUI Dialog. FIR Compiler 7.2 ................................................................. 8
2.1 Block diagram of the Zynq-7000 SoC in context of ESPA.[1] Neural data is streamed into the FPGA where it is processed and decoded. Decoded data is transferred to the ARM processor for distribution to network or USB devices. Web interface allows administrators to view characteristics of the neural data and the processing workflow, as well as the ability to update parameters within the processing model itself.

2.2 Internal Block Diagram of Zynq SoC architecture. "Programmable Logic (PL)" space on the exterior of the diagram represents FPGA fabric, and arrows passing through the port blocks on the "Processing System (PS)" show access buses available to transfer data between the two subsystems. Generated from Vivado.

2.3 Screenshots from preliminary Python server-side software. 2.3a is the main software UI that displays a graph of the data-rate of data coming from ESPA, as well as a packet count. 2.3b is the file writing dialog, showing controls to name and split files, as well as stop and start recording.

2.4 Photograph of the radial keyboard interface (left) with the PAUSE button about to be selected, and the notebook showing the text typed in this session (right) of a BrainGate trial. Used with permission from [4].

2.5 A participant in the BrainGate trials drinking from a bottle using a DLR robotic arm. Four sequential images from the first successful trial showing participant S3 using the robotic arm to grasp the bottle, bring it towards her mouth, drink coffee from the bottle through a straw (her standard method of drinking) and place the bottle back on the table. Used with permission from [5].

3.1 Block diagram of a sample ESPA FPGA configuration. Blue blocks represent neural data sources external to ESPA. Green blocks represent data preparation and glue code. Orange blocks represent code internal to the System Generator generated IP Core module, and the yellow block represents interface to shared resources with the ARM processor.
3.2 "Detailed Implementation" tab of an FIR Compiler 7.2 GUI configurator in System Generator. The drop-down in the top section allows the user to make a choice between an Area and a Speed optimization, with a Custom option available to create a custom mixture of the two. The center section allows the user to decide where different types buffers are implemented, either in Block RAM or in distributed memory cells (or to allow the software to make the best choice.) Finally, the bottom section allows for configuration of DSP Slice Columns, a feature that can improve pipelining efficiency.

3.3 Sample Utilization data generated by the Xilinx Vivado software during Synthesis. The data in 3.3a shows a breakdown of each resource category, while the data in 3.3b is a sub-breakdown of one category.

3.4 Sample SysGen model exemplifying various conventions.

3.5 Block diagram of input logic and signal preparation on ESPA in a single 96 channel wireless recording device scheme. Serial input from the device is denoted in blue, shared memory space with ARM is denoted in yellow, System Generator hooks in orange, and the FPGA code focused on in this section is green.

3.6 An example of Manchester encoding, showing the relationship between input data, clock and the encoded signal using both conventions. The encoded bit sequence is '10100111001'. Image derived from [4].

3.7 Block diagram of the second stage of input glue logic. Port names are in black, while wire nets are in red.
3.8 Data recorded from 96-channel EBNC device. The traces on the left show 3 seconds of raw data from each channel, while the next column shows sorted waveforms, with the next column a raster plot of the timing of those spiking events, an example of an SUA feature. The block in the upper right demonstrates this spiking SUA data, and the pre-processed highpass and lowpass signals. The block in the lower right is a heat map of LFP data, a type of NUA feature. The block in the center right is a Principle Component Analysis chart, a way of reducing the number of dimensions in the SUA data presented. In this context, PCA could be considered a form of decoding. Used with permission from [5].

3.9 Overview of the Feature Extraction subsystem.

3.10 System Generator view of Spike Counter. The gateways in the lower left entitled Data Input and Chan Input come from wireless_packet_to_model. Parameter Input and Parameter WEA are fed from params_load, but sampled at the "model clock" WE Out provided by wireless_packet_to_model. thresh_valid (the enable), thresh_spikeCount and thresh_chanID are fed forward to timeResolvedSelector as a bus from Thresholder.

3.11 tLast Generator uses a relational to create the third signal to conform to the AXI4-Stream standard. tLast, along with chanId and data are grouped together into a bus.

3.12 Inside the FIR Decimator 15k block. The FIR Compiler 7.2 block generates a Xilinx IP Core module of the same name.

3.13 Configuration Dialogs for the FIR Decimator 15k.

3.14 Frequency response of 'fir1(30,.5)', the coefficients used in FIR Decimator 15k. This is suitable for subsequent decimation of the data by a factor of 2.

3.15 CAR System Generator model. Data entering through the Parameter port is fed from the Parameter Engine, while the neural data coming in through the Data Import is fed from FIR Decimator 15k.
3.16 While the most obvious difference between the frequency responses of the IIR filter in 3.16a and the FIR filter in 3.16b is the sharpness of the band edges of the magnitude response, the important distinction is in the phase response. The FIR filter has a linear phase response, whereas the IIR filter does not. This leads to the need of a “non-causal” filtering method described in [9], which requires two instances of this filter, and a buffer in between.

3.17 **Tx Filter** System Generator model. Data entering through the Coeff Loader port is fed from the Parameter Engine, while the neural data coming in through the Data Bus Import is fed from CAR.

3.18 System Generator view of **Thresholder**. The Parameters port brings in new thresholds addressable by channel, as well as a configuration to threshold rising or falling edges. 30k Filtered Data brings in the filtered data from **Tx Filter**, and Out1 sends the binned spike counts out.

3.19 Functional demonstration simulation of **Thresholder**. The blue signal on the bottom scope is the input signal, with an amplitude of 375 (cut off by top of scope). The threshold on all of the channels of this Thresholder is set to 300. The yellow signal on the bottom scope is outBin, which is signaling when the data out is valid. The top signal is the data out, or spikeCount. We count 6 blue input spikes before the valid signal, at which point the value on spikeCount is 6. This is succeeded by 5 more spikes, where we see the value on spikeCount drops to 5 during the valid signal on outBin. Finally, 6 more spikes are seen during that time period, and this is again reflected in the output.

3.20 System Generator view of **timeResolvedSelector** Part 1. This portion of the circuit creates the rolling summation and the FIFO to buffer the original values to subtract out of the summation once the window has "passed."
3.21 System Generator view of **timeResolvedSelector** Part 2. Once the rolling summations are computed, they buffered and sampled at the 0th sample (Delay5), the 12th sample (Delay6) and the 25th sample (dout of FIFO2.) The **Time Division Multiplexer** reorganizes these signals into a streaming set of 288 features, output port Out1.

3.22 Control circuit **FIFO Loader** is used to preload FIFOs in **timeResolvedSelector** with the appropriate number of values to define it's delay length.

3.23 Two participants in the BrainGate trial, S3 and T2 perform two and three dimensional reach and grasp drinking tasks. (a-f) correspond with a three dimensional task, while (g-i) correspond with a two dimensional task. (a,d,g) are a representation of average waveforms for the channels used in the histograms in the corresponding columns. (b,e,h) use histograms of spike rates centered on the given event (directional movement intention) to show the directionality preference in the neuron recorded on that channel. Similarly, (c,f,i) are histograms of spiking activity centered on a grasp event. Used with permission from [13].

3.24 Example of LDA reduction; A 2-Dimensional input feature vector with data from three distinct classes is reduced to a 1-Dimensional feature vector while preserving the differences amongst the classes by finding the "best 1D subspace." Used with permission from [16].

3.25 **Class Means** module subtracts the 288 features generated by **timeResolvedSelector** from the pre-calculated class means, which are stored in the three RAM modules in the center of the diagram, and loaded in through the **Parameters** port.

3.26 Data calculated in the **Feature Extraction** module is transferred to the **Speedy Calculator** via FIFO and RAM modules. The **System Generator** token enables the configuration of both of the subsystem **System Generator** tokens to operate under different clock domains. The visual description of the transfer mechanism in **Class 1**, **Class 2**, and **Class 3** can be seen in Figure 3.27.
3.27 Memory transfer system for time domain switching of the three sets of 
\((x - (\sum_{i=1}^{3} \frac{m_i}{t}))\) values. The internals of all three subsystems, Class 1, Class 2, and Class 3 are identical. 

3.28 Overview of Speedy Calculator decoding scheme. This subsystem operates on an independent clock from the Feature Extraction subsystem – a 250 MHz crystal onboard the ZC-706. 

3.29 Matrix Multiplier to multiply the \(W^T\) covariance matrix by each class’s \((x - (\sum_{i=1}^{3} \frac{m_i}{t}))\) term to generate class probabilities. 

3.30 Expectation for each class calculated using the m_tmpProb values generated by the Probability Calculators. 

3.31 Waveform vector of expectation of each class output of the LDA as it pertains to NHP treadmill data. Following from left to right, Class 1 and Class 2 are approximately inversely related, oscillating between Class 1 and Class 2. More difficult to see are the breaks in the line of Class 3, where some non-zero values are calculated periodically (often aligning with the sharp value spikes seen in Class 1 and Class 2.) Class 3 viably comes above zero for a notable period of time near the end of the plot, between time markers 7 and 8. The time scale for this plot is 10ms. 

3.32 params_load module takes data from shared BRAM from the ARM processor (triggered by the qnx_intsignal) and reads in 32-bit values from BRAM_PORTB_1_dout, utilizing the fast_clk. Data is then spun out to the System Generator model, one 32-bit chunk at a time, using the param_outbus, with the data_clk patched through to the param_we during this time. 

3.33 System Generator view of Parameter Writer. Signals are simply passed through the Parameter Handler shell. The Enable for each subsystem is controlled by both the input Enable, as well as determining which of the classes of Op-Codes are being using (A, B or C) by slicing and comparing that portion of the OP-Code with the respective number (depicted here in binary.)
3.34 **Enables** module of the Parameter Engine. Responsible for OP-Codes 00A0-00A3. ........................................ 65

3.35 **Data Loading** module of the Parameter Engine (part 1.) Responsible for OP-Codes 00B0 and 00B1. ........................................ 66

3.36 **Data Loading** module of the Parameter Engine (part 2.) Responsible for OP-Codes 00B4 and 00B5. ........................................ 67

3.37 **Stored Parameters** module of the Parameter Engine. Responsible for OP-Codes 00C0 and 00C1. ........................................ 69

3.38 HDL block diagram of modules to aggregate data and transfer from System Generator model to ARM. In this example, agg_data, agg chanid, class0, class1, class2, m_tmpprob1, m_tmpprob2, m_tmpprob3, and thresh_spikecount are packed in repack_model_out_245 each time. agg_valid, class_valid, m_tmpprob_valid, and thresh_valid are used to trigger we_check to produce a model_global_wesignal to force repack_model_out_245 to capture. Once the buffer in repack_model_out_245 is full (5 data frames,) the data is transferred out on data out triggered by WE out to wireless_packet_to_bram for transfer to ARM. ........................................ 70

4.1 A simulation doing in SIMULINK with data plotted in MATLAB. The test vector is defined in Appendix B.6. This waveform in this figure is the output of the CAR block on channel 1. When the CAR is enabled on channel 1 (all inputs except channel 3 are the same,) the voltage value goes to 0. When channel 3 is also turned on (with value of 0,) the result is an average of the two signals. Finally, when the CAR is turned off, the original signal is seen. ................................. 74
4.2 SIMULINK waveform scope showing proper operation of the **Threshold** module. The first and fourth waveforms are channel numbers (continuous ramping data), while the second waveform is the output of the thresholding vector. The third waveform shows the latched value for number of threshold crossings during the time period, while the fifth waveform shows the time in which those latched values are valid for transport. The first bin just catches the first waveform, hence the value of 1. The next several bins catch the three next waveforms, and the second to last bin catches four of them (due to planned input data frequency). Finally, the threshold value is changed across all channels, and the bin count goes to 0 at the end.

4.3 An excerpt of the generated HDL Schematic view of ESPA. Shown in the figure is part of the **CAR** module, with the **Accumulator** and **Register1** exploded to show basic components like gates, adders and registers.

4.4 Vivado Behavioral Simulation output of the full System Generator model plugged into the Xilinx Zynq hybrid-subsystem. Depicted here are the outputs of the classifiers, as they shift to an unknown ”neutral” state on their first successful computation (no non-zero features had been computed at this point.)
4.5 Vivado Simulator output with test points at various points in the model. (a) shows the sample data vector being fed into the model, which consists of both a high and low frequency sine component. (b) shows the result of the first 15k filter, which should decimate the signal but not alter the content in any major way (the fill time of the filter can be observed.) (c) shows the output of the CAR module, which for the purpose of this demonstration, was disabled (the signal being fed to each channel is identical, so enabling the CAR would result in an output of 0 on all channels.) The delay of this module (even though it’s off) can be observed by comparing the rising edge to that of (b). Finally, (d) shows the output of the tx filter, which successfully removes the slow sine component, but leaves the fast sine component intact. The delay of this filter loading can be observed as well.

4.6 Concept drawing of Custom ESPA Hardware.

4.7 Photograph of a sample case filled with electronics envisioned for the Concept A back-end server. A sealed plastic case with latches houses the needed server electronics, power supply, storage drives and wireless router.

4.8 Photograph of a sample case filled with electronics envisioned for the Concept B back-end server. A mobile road case houses a full blade server, rack-mounted terminal, storage bays, managed networking hardware and power management. Wireless router sits inside the case with external antennas. Used with permission from Gator Cases, Inc.

4.9 Mobile Medical Monitoring System (M^3) concept diagram.

A.1 Power bands calculated in LFP Feature Extractor

A.2 Sample data processed by the LFP Feature Extractor

A.3 Memory Subsystem found in Speedy Calculator

A.4 Memory Controller found in Memory Subsystem
Listings

3.1 Code example describing functionality of the Classifier . . . . . . . . 57
4.1 EML example generates ramp data across 96 channels (0 indexed) . . 74
Introduction

1.1 Brain Computer Interfaces for Prosthetic Control

Brain Computer Interfaces (BCI) have been a topic of study for the better part of a century, starting with Hans Berger’s discovery and development of electroencephalography (EEG) in 1924.[1] Beginning with his silver wires under the scalp, to silver foils on the surface of the head, to the modern silicon-based Micro Electrode Arrays (MEAs) and gelled EEG caps of the present, there is a long history of recording and understanding information recorded from the brain. When Jacques Vidal originally coined the term in his 1973 project called ”The Brain Computer Interface Project” [2], he set out to ”attempt to evaluate the feasibility and practicality of utilizing the brain signals in a man-computer dialogue while at the same time developing a novel tool for the study of the neurophysiological phenomena that govern the production and the control of observable neuroelectric events.” He correctly speculated that ”long-range implications” of BCI ”can only be speculated upon at present.”

Much work has been done since Vidal’s dream of ”the ultimate goal ... a genuine prosthetic extension of the brain” set sail. In addition to numerous animal BCI research projects, such as retinal BCI work done at UC Berkeley[4] or motor control BCI work done at Duke University[5], state of the art human BCI research is conducted at Brown and Stanford Universities, whom conduct the BrainGate2\(^1\) clinical trials.[6]

\(^1\)Caution: Investigational Device. Limited by Federal Law to Investigational Use.
In these trials, participants who have "limited or no ability to use both hands due to cervical spinal cord injury, brainstem stroke, muscular dystrophy, or Amyotrophic Lateral Sclerosis (ALS)" are fitted with one or more Micro Electrode Arrays (MEAs) (Figure 1.1). Signals are collected from the electrodes, and using a series of signal conditioning and decoding processes, create "decoded" computer signals for the control of various prosthesis, including 2D control of a cursor and 3D control of a robotic limb.

1.2 Neural Signal Conditioning and Decoding

Neural signals are sensed using a variety of instruments, such as Micro Electrode Arrays (MEAs), electrocorticography (ECoG) arrays, electroencephalography (EEG) caps, or even through lesser-known techniques such as optogenetic sensors. These signals are amplified and digitized into the digital realm, and either wirelessly or by wire are transmitted to a computer for processing.

Before translation or "Decoding" of the neural signals can be performed, these signals must first be conditioned using Digital Signal Processing processes, in order to extract "features." These features range in type, depending on the type and fidelity
of the input signal. For instance, signals captured from an EEG cap or an ECoG array will not contain Action Potential or "spiking" data, but may contain Local Field Potential (LFP)-like data. In contrast, signals recorded from an MEA can be processed to produce both LFP as well as "spiking" features. Some of these features are as simple as band-pass filters followed by a power calculation (also known as band-power), while others are more complex, consisting of several filters, as well as noise reduction techniques, adaptive thresholding, and non-stationarity reduction. Some systems also include "spike sorting" algorithms, which try to differentiate spiking between neurons on a single recording electrode; thereby producing multiple spiking "features" per recording channel.

![Figure 1.2: Anatomy of a “typical” Action Potential – plot shows the various phases of the AP in terms of voltage as it passes a given point on the cell membrane, whose resting potential is about -70mV. After a stimulus is applied at t=1ms, depolarization occurs, raising the membrane potential above the threshold of -55mV. The membrane potential reaches a peak of around +40mV before rapidly repolarizing at t=2ms. The potential overshoots to -90mV, leading to a refractory period lasting until t=5ms, when the resting potential of -70mV is yet again reestablished. Used with permission from [7]](image)

After neural "features" are extracted from the digitized data, they then must be decoded into meaningful signals that can be used to inform or control a prosthetic device. Many types of decoders exist, but the general concept is simple: use pattern recognition and statistical methods in your extracted feature data against a pre-calculated and sometimes adaptive set of coefficients that are "learned" and calculated in a closed-loop environment. In other words, statistically identify patterns when the
intention of the neural signal is known (instructing the subject to imagine or actually perform a specific action,) so that these patterns can later be identified when the intention is not known.

1.3 Requirements and Design Constraints for an Embedded System for Prosthetic Application

System requirements for neural data collection and processing setups vary wildly, however, they all contain the same basic components. On the front-end is an amplifier and digitizer circuit. This digital data is then delivered either by wire, fiber optic cable, or wirelessly to a receiving circuit which then passes the data to a Digital Signal Processor (DSP). At this point, a combination of DSP and computer hardware generates the neural features and passes them through the decoder. The decoded outputs are then prepped for transmission to the appropriate prosthetic device. Examples can be found in the literature of low data-rate FPGA systems, such as Lin’s EEG BCI[8] and Shyu’s SSVEP BCI[9], partial solutions like Karkare’s 16-Channel Spike-Sorting ASIC[10], and ARM based solutions, again for low data-rate, like Puggle[11] (500ks total).

This body of work describes the development and implementation of a data processing system, on an embedded platform, intended to control a prosthetic using neural signals on a device that is mobile, wearable, and battery-operated. This system is known as the Embedded System for Prosthetic Application (ESPA). Design requirements include inputs for hundreds of channels of neural data, computational horsepower to extract features from each of those channels, computational horsepower to decode those features, a system to manage and store data, a means of connecting the system to control a prosthetic, and subsystems to monitor and manage the device from both a technician’s as well as a clinician’s standpoint. Additionally, this system must be small enough that it can be worn comfortably by a patient, and must be battery operated with a lifetime that matches the runtime of other component’s in a patient’s system (nominally 8 hours.) A back-end server for long-term data storage, tabulation, and management processes is also required.
Figure 1.3: Block diagram of patient utilizing ESPA system. The patient on the left (in an existing system) has a wired, broadband neural implant connected to a cart with neural data-processing hardware and computers which in turn control end-effectors. On the right is a system with ESPA in the loop; capable of communicating with a wireless neural implant, doing all data processing on-board, and wirelessly communicating with various end-effectors.

In addition to the hardware system described above, a "software" system is required to carry out the necessary computations required for feature extraction, decoding, system and subsystem management, prosthetic control, data I/O and management, network interfaces and system control. This software system also need to be able to be understood and maintained (to some degree) by non-Computer Engineers.

1.4 ESPA System Design Overview

Given the design requirements, a system was designed utilizing the Xilinx Zynq SoC at it’s core. The Zynq offers a unique ARM processor / FPGA solution on a single die. ESPA utilizes the ARM processor to manage I/O, running a medical-grade Real-Time Operating System (RTOS) called QNX. QNX manages device drivers and data throughput for storage of raw and processed neural data to long-term memory, as well as for transfer over a wired or wireless network. Additionally, QNX can establish and manage connections to various prosthetics over various interfaces. In addition to some custom hardware I/O, the FPGA fabric is primarily used for signal conditioning and decoding of the neural signals. The majority of the body of work that this thesis
encompasses focuses on this FPGA system design.

The Zynq is integrated into a larger system[12] that includes a power management circuit, physical USB and Ethernet I/O connectors, an 802.11n wifi card, and local flash memory storage. Also attached to the FPGA is a wireless receiver circuit, used to capture, digitize and format the neural data transmitted using a Brown 100-channel 20ksps neural transmitter, known as an EBNC. In the final ESPA solution, all of these components are to be integrated into a "motherboard" and a wireless receiver "daughter card", however, for the work described here, discrete components, including a development board, were used.

1.5 FPGA Software Development

The centerpiece of this thesis is the development of a programming capability for applying the wearable/portable, wireless neural signal processor/decoder for brain-computer interfaces, but to enable its dissemination across broader application spaces. Signal conditioning, feature extraction and decoding in ESPA are all implemented in FPGA fabric. While the most straight-forward way of accomplishing this from an initial standpoint would likely have been to use a mixture of IP Core blocks with Hardware Description Language (HDL) code, one of the design requirements is that the code is relatively easy and straightforward to understand and maintain. This led to a hybrid HDL implementation structure, utilizing flexible HDL pieces, Mathworks/Xilinx System Generator via SIMULINK, Mathworks HDL Coder as well as Xilinx IP Core modules. The major advantage is having core computation components running in SIMULINK; this environment is more conducive to understanding, utilizing a graphical programming language with GUI parameterization.
One major advantage of using FPGA fabric over a general purpose CPU is the ability to parallelize processes. Filters across multiple channels can be running simultaneously, never threatening timing issues, as every piece in the pipeline is executing in lock-step. Additionally, some tasks can be performed on dedicated DSP slices, for which, on the Z-7045, a theoretical 1334 GMACs/sec worth of compute is available alone.[13]

A key development that allows for neural computation to be done in FPGA fabric was the development of the Parameter Engine. The Parameter Engine is, at its core, a CPU implemented in the FPGA that allows for data to be passed from the external world, through the ARM processor, into the FPGA model, and then appropriately
modify the behavior of the model. Examples of these parameters are updating filter coefficients, enabling or disabling subsystems, enabling or disabling processing on particular channels, and setting threshold voltage levels. The Parameter Engine is designed in a modular way, to allow future expansion of use cases.

The design and modularity of the Parameter Engine is critical to Embedded System for Prosthetic Application (ESPA), as any real-time control software needs external inputs to be able to tune and make modifications on-the-fly. This unique modular system design is a key to supporting the dynamic nature of the research performed with the system; as processing models are updated and iterated, new parameterized features will be added, and need to be controlled by the Parameter Engine.

1.5.1 Interface Specifications and Protocol Definition

Major functional subsystems have their interfaces defined, such that it is easy to replace a subsystem block with a different SIMULINK model, or even a different type of code. The utilization and implementation of Black Box objects allows for inserting direct Verilog code, or even generating HDL from Embedded MATLAB Language (EML), with direct interface (including simulation capability) in the MathWorks software.

The interfaces between subsystems, while relatively rigid (many requiring redesign if new subsystems are required), are well defined and allow drop-in replacement of modules. Additionally, as the SIMULINK model is compiled to a Xilinx Intellectual Property (IP) Core Catalog block, connecting additional Xilinx IP Core pieces is a relatively straightforward process. Care has been taken to make the Parameter Engine flexible enough to easily accommodate new computation pieces.

External to the SoC, protocols have been defined to move data in and out of ESPA. Based on the configuration selected on the ARM processor, data can be saved directly to local flash memory, or can be streamed out over a wired or wireless UDP network via Ethernet or 802.11n.
1.6 Validation of System Operation

As various pieces of ESPA have been developed, they have been tested and used in various ways. Early renditions of ESPA were used in various neural recording projects, including a freely moving swine demonstration\cite{14}, as well as a 24-hour sleep study in a Non-Human Primate\cite{15}. Additionally, various computational components of ESPA have been tested in simulation as well as in hardware on the benchtop, using both simulated and actual recorded neural data. Some of these simulations are included in this document.

1.7 Taking ESPA Forward Toward Experimental Integration

The primary goal of this work is to make a key set of contributions towards a complete neural data processing platform that can be used by researchers, and eventually patients, to study and create prosthetic systems for individuals suffering from disability. Some additional steps are required to achieve this goal, including the development of a back-end server system, an integrated software suite for interfacing with ESPA remotely (via tablet or computer), as well as targeted neural processing algorithms, particularly the data decoder. ESPA is presented as a powerful research platform, targeting pathways for human therapeutic use.
References


2

ESPA System Architecture

2.1 ESPA As A Platform

As neural recording and stimulation devices have evolved over the decades, particularly in the research setting, there has been a need for the integration of other sensors and actuators into the control systems that operate these devices. While there has been much effort to develop wearable wireless signal processing, analysis and communication devices for diagnostic and therapeutic use, the integration of these realms when it comes to neuroengineering has been particularly challenging, and often ad-hoc, as the exceptionally large amounts of real-time data recorded from the nervous system can be difficult for a computer system to handle, let alone integrate with other signals.

This work aims to create a wearable, battery operated platform capable of receiving data from one or more neural recording devices, receive data from biometric and environmental sensors, and process and decode data to enable real-time neural control by disabled persons of assistive devices, ranging from telecommunication to advanced robotics. Of course, designing a platform with such a wide range of criteria is a challenge, and is broken down into a set of constraints.

- Command and control of the system by a supervisory system
- Common interfaces (such as USB and Ethernet)
- Flexible data processing environment
- Large amount of parallel computation power for neural data processing and decoding
- Wireless interfaces for neural devices
- Wireless interfaces for client and monitoring devices (802.11n, Bluetooth)
- Back-end service capabilities for monitoring and adjustment of the system in real-time

### 2.2 Zynq 7000 SoC

The Xilinx Zynq 7000-series System on Chip (SoC) was chosen as the base hardware platform on upon which to build ESPA. The Zynq is an SoC that contains both a dual-core ARM processor as well as an abundant amount of FPGA fabric together on die. In combination with a shared memory space, there is a plethora of I/O available on the SoC. Running an OS on the ARM processor makes building a command and control supervisory system much easier, as applications can be written in standard programming languages and interfaced with system resources, such as networked resources (either Ethernet or 802.11), or USB devices (such as storage devices or even a USB robotic prosthetic). The FPGA resources allow for the design of complex, parallelized computation tasks, as well as straight-forward implementation of custom interfaces to hardware devices.
Figure 2.1: Block diagram of the Zynq-7000 SoC in context of ESPA.[1] Neural data is streamed into the FPGA where it is processed and decoded. Decoded data is transferred to the ARM processor for distribution to network or USB devices. Web interface allows administrators to view characteristics of the neural data and the processing workflow, as well as the ability to update parameters within the processing model itself.

Digital data can be collected from custom interfaces through FPGA pins, or through standard interfaces via drivers running on the OS on the ARM. A high-speed AXI bus operating at up to 1.2GB/s is available between the ARM and the FPGA fabric. DDR3 memory is accessible by the ARM on an 800MB/s bus.

The operating system selected for ESPA is QNX Neutrino. QNX is a UNIX-based real-time operating system, meaning that tasks can be scheduled to maintain consistency in the amount of time they take to complete (deterministically meet deadlines.)(2) QNX meets the IEC standard 62304 Software Life-Cycle for medical devices, is classified as a “medical grade operating system,” and has passed all FDA
audits. ESPA operates QNX in Symmetric Multiprocessing (SMP) mode.

2.3 ESPA Supervisor

ESPA Supervisor is a custom application running on the ARM processor. The Supervisor’s role is that of a "traffic cop," waiting to send, receive and store data as it becomes available. It can be started with different parameter sets to control its operation; for instance, whether neural data is to be saved to an SD card, or streamed out over Ethernet via UDP packets. The software automatically handles transfer from the FPGA via interrupts through the GPIO interface, which trigger bulk transfer of data from BRAM.

Figure 2.2: Internal Block Diagram of Zynq SoC architecture. "Programmable Logic (PL)" space on the exterior of the diagram represents FPGA fabric, and arrows passing through the port blocks on the "Processing System (PS)" show access buses available to transfer data between the two subsystems. Generated from Vivado.

2.4 Server Side Application

While the details of the hardware requirements of a back-end server are covered in Chapter 4.2.2 as part of a future design requirement, pieces of the server-side
application have already been built. The purpose of the server-side application is to store both raw neural data as well as processed neural data as it is a streamed out of ESPA to a backup storage solution. Additionally, this application should make a processed form of this data available to a clinician using a mobile device, as well as allow a clinician to see and modify parameters currently set in the processing model (meaning that the server-side application should be able to collect and transmit this data to ESPA.)

Presently, this server-side application is written in Python. Features include:

- Receiving UDP data from ESPA
- Storing UDP data as raw ”12-bit” formatted file
- Visualization of data throughput coming from ESPA
- Auto-splitting of file storage to prevent enormous files
- Utility to convert raw ”12-bit” formatted files to MATLAB-compatible ”16-bit”

Figure 2.3: Screenshots from preliminary Python server-side software. 2.3a is the main software UI that displays a graph of the data-rate of data coming from ESPA, as well as a packet count. 2.3b is the file writing dialog, showing controls to name and split files, as well as stop and start recording.

Some additional features that are under development or required in the near-term for operation of ESPA in a clinical setting:
• Web server for clinician interface
• Parameter passing from clinician interface to ESPA for model update
• Visualization tools for processed neural data
• Automated processes for producing new model/decoder parameters; collecting new open-loop data

2.5 End Effectors

One of the main attractions of the ESPA platform is the flexibility in interfaces for end effectors. To date, only a USB/Bluetooth interface has been developed, enabling cursor control on an Android tablet using simulated decoder outputs. In the future, this sort of interface might be used to control assistive technologies, such as a radial keyboard shown in Figure 2.4.

Figure 2.4: Photograph of the radial keyboard interface (left) with the PAUSE button about to be selected, and the notebook showing the text typed in this session (right) of a BrainGate trial. Used with permission from [4].

Future end effectors may be various robotic limbs, such as the DEKA and DLR (KUKA) arms tested and proven by the BrainGate trials [5]. These end effectors
would require a decoder with at least three dimensions of output (for X, Y and Z end-velocity control,) with a possibility of a state detector to initiate a grasp command. An example of a tetraplegic patient using such an arm with a neurally-controlled system is shown in Figure 2.5.

Figure 2.5: A participant in the BrainGate trials drinking from a bottle using a DLR robotic arm. Four sequential images from the first successful trial showing participant S3 using the robotic arm to grasp the bottle, bring it towards her mouth, drink coffee from the bottle through a straw (her standard method of drinking) and place the bottle back on the table. Used with permission from [5].)
References


3

FPGA Software Development

3.1 FPGA Overview

3.1.1 FPGA System Architecture

The FPGA in ESPA is responsible for the following roles:

- Neural data capture from serial devices
- Data preparation algorithms to produce organized, raw neural data
- Gathering new data (known as Parameters) sent by ARM to update the processing model
- Feature Extraction
- Prosthetic Decoding
- Data management for transfer to ARM

Some of these roles, particularly the data capture and transmission and glue logic on either end, are accomplished using modular Verilog blocks that are written to be application specific for the given hardware setup. Much of the rest of roles are fulfilled in a series of System Generator generated IP Core blocks that are designed and maintained in SIMULINK. Some of these blocks also contained "Black Box" objects, which can contain Verilog code, or Embedded MATLAB Language (EML) code that is converted to Verilog using the HDL Coder toolbox.
Figure 3.1: Block diagram of a sample ESPA FPGA configuration. Blue blocks represent neural data sources external to ESPA. Green blocks represent data preparation and glue code. Orange blocks represent code internal to the System Generator generated IP Core module, and the yellow block represents interface to shared resources with the ARM processor.

Figure 3.1 shows a sample configuration of the FPGA subsystem in use in ESPA. On the left in blue are two 100 channel wireless neural recording devices, asynchronously streaming serial data (with clock) into I/O pins directly attached to FPGA fabric. The green blocks represent data preparation and glue code. *sync_check* verifies the integrity of the incoming data, *write_to_model* aligns the asynchronous inputs and packages them for transport into the System Generator model, as well into the *repack_model* block, which takes this raw neural data as well as the processed data from the System Generator model and prepares it for another intermediary block, *wireless_packet_to_bram*, which has an interface with the BRAM itself. The BRAM acts as an interface (complete with interrupt and transport system) for the FPGA with the ARM processor. This interface is bi-directional: data also come from the ARM into the System Generator model through the *params_loader* block.

The System Generator Model depicted (in orange) in Figure 3.1 contains three major pieces: A feature extractor, running at the clock rate derived from the input
data, a decoder, operating at a faster crystal running on-board the SoC, and the Parameter Engine, which is asynchronously storing and updating values and switches inside the two other aforementioned blocks.

3.1.2 Resource Allocation

Efficient usage of resources is important from a few standpoints: reducing area consumed by a process allows for higher complexity processes, and reducing area consumed by a process reduces the power consumed by that process. Strategically utilizing the available resources to their maximum potential is important, though can typically be a tricky trade-off between performance (reduced area) and development time. Fortunately, this is one of the areas where developing processes using System Generator can be really beneficial.

Using Xilinx IP Core modules is a way of time-efficiently using very optimized code. Typically, IP Core modules are written in such a way that optimally maximizes the use of specialized hardware available in the FPGA fabric, such as DSP Slices and Extensible Block RAM before resorting to implementation using normal Logic Cells, Look Up Tables (LUTs) and Flip-Flops. Table 3.1 gives a summary of the available resources on the Z-7045, the Zynq chip in use on the XC706 development board, currently in use in ESPA.
<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xilinx 7 Series Programmable Logic Equivalent</td>
<td>Kintex-7 FPGA</td>
</tr>
<tr>
<td>Programmable Logic Cells (Approximate ASIC Gates)</td>
<td>350K Logic Cells (5.2M)</td>
</tr>
<tr>
<td>Look-Up Tables (LUTs)</td>
<td>218,600</td>
</tr>
<tr>
<td>Flip-Flops</td>
<td>343,800</td>
</tr>
<tr>
<td>Extensible Block RAM (#36 Kb Blocks)</td>
<td>2,180 KB (545)</td>
</tr>
<tr>
<td>Programmable DSP Slices (18x25 MACCs)</td>
<td>900</td>
</tr>
<tr>
<td>Peak DSP Performance (Symmetric FIR)</td>
<td>1,334 GMACs</td>
</tr>
<tr>
<td>PCI Express® (Root Complex or Endpoint)</td>
<td>Gen2 x8</td>
</tr>
<tr>
<td>Analog Mixed Signal (AMS / XADC)</td>
<td>2x 12 bit, MSPS ADCs with up to 17 Differential Signals</td>
</tr>
<tr>
<td>Security</td>
<td>AES and SHA 256b for Boot Code and Programmable Logic Configuration, Decryption, and Authentication</td>
</tr>
</tbody>
</table>

Table 3.1: Available FPGA Resources on the Xilinx Zynq Z-7045, XC7Z045. Data extracted and reformatted from[1].

System Generator provides the ability to implement Xilinx IP Core modules using simple GUI interfaces, and wiring them together using ”wires” and ”buses” in the SIMULINK environment. System Generator takes care of transfer of clocks, enables, reset lines, and initial conditions (an example of setting initial conditions can be seen in Figure 1.4). In addition to easing the hookup process, System Generator also provides GUI based features to customize Implementation options, including things like optimizing for Area or Speed (or a custom combination), configuring preference in memory allocation location, and configuration of DSP slice architecture. An example of this configuration window for the FIR Compiler, a block that allows for the implementation of custom FIR filters across many channels, is show in Figure 3.2.
Figure 3.2: "Detailed Implementation" tab of an FIR Compiler 7.2 GUI configurator in System Generator. The drop-down in the top section allows the user to make a choice between an Area and a Speed optimization, with a Custom option available to create a custom mixture of the two. The center section allows the user to decide where different types buffers are implemented, either in Block RAM or in distributed memory cells (or to allow the software to make the best choice.) Finally, the bottom section allows for configuration of DSP Slice Columns, a feature that can improve pipelining efficiency.

Xilinx Vivado is the Integrated Development Environment (IDE) used to configure the Zynq, as well as develop the FPGA code to be deployed in fabric. Within the environment, Verilog files, Xilinx IP Core modules, IP Core modules generated by System Generator, and even whole Block Diagrams can be integrated together to create a Generated Bit File, which gets loaded into the FPGA fabric on startup.
After the FPGA project is complete, it must first get Synthesized and Implemented from the HDL code into specific instructions used to configure the modules on the specific targeted FPGA in use (in this case, the Z-7045.) After this process is complete, Vivado generates a Utilization Report (ex. Figure 3.3), which includes a total breakdown of utilization of the different resources available in fabric (3.3a), along with a more detailed report of how those given utilization categories are broken down (3.3b.)

(a) An example of a generated Utilization graph. In this example, many signals are passed through to pins on the device for testing, so I/O utilization is fairly high.

(b) Screenshot of a snippet of the Utilization Report generated during Synthesis. This snippet shows the breakdown of the 3% LUT utilization shown in the graph in 3.3a.

Figure 3.3: Sample Utilization data generated by the Xilinx Vivado software during Synthesis. The data in 3.3a shows a breakdown of each resource category, while the data in 3.3b is a sub-breakdown of one category.

### 3.1.3 Conventions

Throughout this chapter, the following conventions will be used:

- File names are shown in **italics like this**
- Signal and bus names are show with an **underline like this**
- Modules and SysGen subsystem names are show in **bold Like This**
- SysGen time domain color coding as per Table 3.2
\begin{itemize}
  \item Other SysGen conventions described below with reference to Figure 3.4
    \begin{itemize}
    \item Yellow gateway blocks are used as pin definitions for the generated IP Core module. In this case, \textbf{Gateway In} has been configured to be a 16 bit wide port, represented as a 16 bit, unsigned signal named \texttt{in\_sig} with a binary point at the 14th bit.
    \item This signal is fed into \textbf{Down Sample}, which changed the time domain from 1 to 2 (as denoted by the $\downarrow 2$, as well as the red and green lines per Table 3.2,) taking a delay of one input clock cycle (as denoted by the $z^{-1}$.)
    \item The downsampled signal is fed into the \textbf{Convert} block, which truncates the signal and re-denotes it as a 12 bit, signed signal named \texttt{val1} with a binary point at the 6th bit.
    \item A constant is defined using the \textbf{Constant} block, explicitly as a 32 bit floating point number called \texttt{val2} with a sampling rate of 3; however, this is a constant, so defining the sampling rate is useful only in helping the compiler when using the signals against other signals.
    \item \texttt{val1} and \texttt{val2} are combined into a bus called \texttt{bus1}, which is denoted to be carrying two signals by the \{2\} notation.
    \item \texttt{bus1} is re-split into its respective components in any arbitrary order (here shown reversed from the input.) These signals feed two gateway output blocks, denoted as \textbf{Gateway Out1} and \textbf{Gateway Out2}
    \end{itemize}
\end{itemize}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3.4.png}
\caption{Sample SysGen model exemplifying various conventions.}
\end{figure}
### Table 3.2: Color key for time domains in the SysGen model.

<table>
<thead>
<tr>
<th>Color</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red</td>
<td>Discrete 1</td>
<td>1</td>
</tr>
<tr>
<td>Green</td>
<td>Discrete 2</td>
<td>2</td>
</tr>
<tr>
<td>Dark Blue</td>
<td>Discrete 3</td>
<td>3</td>
</tr>
<tr>
<td>Light Blue</td>
<td>Discrete 4</td>
<td>6</td>
</tr>
</tbody>
</table>

"Discrete 1", denoted in red, is the time domain supplied by the neural implant(s), supplied by the `write_to_model` glue piece. As this data gets decimated, slower clocks are used to run pieces of model hardware based on those decimation rates. Green denotes the input clock divided by 2, dark blue by 3, and light blue by 6.

### 3.2 Input Glue Logic and Signal Preparation

![Block diagram of input logic and signal preparation on ESPA in a single 96 channel wireless recording device scheme.](image)

Figure 3.5: Block diagram of input logic and signal preparation on ESPA in a single 96 channel wireless recording device scheme. Serial input from the device is denoted in blue, shared memory space with ARM is denoted in yellow, System Generator hooks in orange, and the FPGA code focused on in this section is green.

`design_1_wrapper.v` is the top level Verilog module for the HDL embodied in the PL of the Zynq. The signal `FCLK_CLK0` is an SoC input pin that is attached to a (programmable) crystal, operating at 250MHz. `$wireless_clk_in` is an physical input pin that is attached to the clock that accompanies the neural data as it comes off of the wireless receiver, and is operating at approximately 48 MHz[2]. While this should behave as a “clock”, it’s really more of an enable for new data into the PL.

Neural data from physical pin `$wireless_data_in` along with it’s clock on pin `$wireless_clk_in` are fed into the Manchester Decoder module in `$manchester_decoder.v`. Data is sent
from a Brown SBNC wireless recording device using Manchester encoding in order to be able to help detect errors during transmission, as well as to ease the data and clock recovery process.

Manchester encoded data is decoded by applying a simple algorithm. In order to obtain the original data, the Manchester encoded value is XOR’d with the value of the recovered clock\[3\]. An example of this can be seen in Figure 3.6. The resulting process in $\text{manchester\_decoder.v}$ produces the signal $\text{ser\_data}$, and it’s new accompanying clock, $\text{ser\_to\_sync\_check\_clk}$ (operating at exactly half the frequency of $\text{wireless\_clk\_in}$, or approximately 24MHz.)

![Manchester Encoding Diagram](image)

Figure 3.6: An example of Manchester encoding, showing the relationship between input data, clock and the encoded signal using both conventions. The encoded bit sequence is ’10100111001’. Image derived from [4]

This serial neural data, now decoded, is fed into the next module, an instantiation of $\text{wireless\_sync\_check.v}$. This module buffers an entire frame of data (meaning, one 12-bit sample of all 96 channels, accompanied by the sync word.) After it is verified that the sync word can be found at both the beginning and end of the frame, the entire (verified) frame is output to the next module, $\text{wireless\_packet\_to\_model}$, along with a clock and write enable. The input clock $\text{ser\_to\_sync\_check\_clk}$ is passed through as the output clock, $\text{sync\_check\_to\_model\_clk}$, with the output write enable $\text{sync\_to\_model\_write\_WE}$ denoting when the next frame of data on $\text{complete\_packet}$ is ready to be grabbed.
Figure 3.7: Block diagram of the second stage of input glue logic. Port names are in black, while wire nets are in red.

An instantiation of `wireless_packet_to_model.v` takes the output data frame from the previous module and prepares it to be spun into the System Generator model, 12-bit sample by sample. As soon as a new frame is available as denoted by `sync_to_model_write_WE`, every other input clock `sync_check_to_model_clk` produces a new 12-bit data value out on `to_model_data`, and the corresponding channel number on `to_model_chan`. This is also accompanied by the `to_model_WE` wire, which acts as a clock to denote when new values are being spun in. This data rate, when active, is half that of the input clock (as denoted earlier “every other input clock”), such that it toggles high on the first input clock, and low on the next, etc. This reduction in data rate is tolerated, as each output from the module is a 12-bit data, while the original clock is deriving new data at the rate of 1 bit/clock. This means that the write enable is only toggling (acting as a clock) 1/6th of the time, and is low the rest of the time. See Appendix A.1 for alternate configurations.

In parallel with the data generator, a parameter loader is also instantiated (`params_load.v`). This takes the fast clock, `FCLK_CLK0`, as a clock input to process, but the output is actually controlled by the “clock” (`to_model_WE`) generated by the data frame processor `write_to_model` (`wireless_packet_to_model.v`). This is because the parameter engine inside the SysGen model that handles this parameter data is in the same clock domain as the input data. This means that it is the responsibility of the `params_loader` to buffer the parameter data to be spun into the model and produce the proper data output `param_to_model` and the valid line `param_we_to_model` when there are appropriate clock cycles available on `to_model_WE`. 
3.3 Feature Extraction

3.3.1 Overview

Feature Extraction

"Feature Extraction" is generally a Digital Signal Processing (DSP) problem applied to the business of neural decoding. In general, feature extraction can be broken down into three major categories:

- Pre-processing
- Single-Unit Activity (SUA)
- Multi-Unit Activity (MUA)
- Non-Unit Activity (NUA)

Pre-processing generally involves decimation and filtering of incoming neural signals down to data-rates and bands that are operable for the given type of feature extraction. SUA and MUA features require high-bandwidth data, typically with the lower frequencies filtered out. NUA analysis is generally looking at individual frequency bands, not events, at sampling rates under 1KHz.

Single-Unit Activity refers to features extracted from individual neurons. The most common SUA feature to be extracted is spiking data, in the form of either spike times or spike rates. Spike times allow for patterning exact timing on spikes on a given channel, whereas spike rates are a little more general and result in binning detected spikes into 'counts'. Both SUA and MUA features generally require electrodes in cortex, as EEG and ECoG arrays are not capable of recording information from individual neurons.
Figure 3.8: Data recorded from 96-channel EBNC device. The traces on the left show 3 seconds of raw data from each channel, while the next column shows sorted waveforms, with the next column a raster plot of the timing of those spiking events, an example of an SUA feature. The block in the upper right demonstrates this spiking SUA data, and the pre-processed highpass and lowpass signals. The block in the lower right is a heat map of LFP data, a type of NUA feature. The block in the center right is a Principle Component Analysis chart, a way of reducing the number of dimensions in the SUA data presented. In this context, PCA could be considered a form of decoding. Used with permission from [5].

Multi-Unit Activity generally refers do doing data processing on multiple channels of data containing information from groups of neurons. MUA features are generally a calculation of ”spike power”, or power in the frequency bands above those covered in NUA LFP bands, generally thought to to be contributed to more heavily by localized neurons. Researchers at Brown have had success in a particular method of calculating spike power based on [6], namely, bandpass filtering at 300Hz - 6kHz, integrating over
non-overlapping windows, and then downsampling.[7] Non-Unit Activity are features derived from data that isn’t strictly data from individual neurons. All EEG related processing methods fall under this category, but MEA recordings can also be subjected to this class of feature extraction. One type of NUA feature is Movement Related Potential (MRP), which can help provide information about intention to move. The most common type of NUA extraction, however, are Local Field Potentials, or LFP. These are power calculations done in explicit frequency bands, either using bandpass filters or an FFT. An example of an ESPA implementation of an NUA LFP feature extractor can be found in Appendix A.2.

**ESPA Feature Extraction Strategy**

The current ESPA feature path involves features derived from SUA spike rates. An additional LFP feature extractor has also been built and tested, but isn’t in use in the current decoding workflow. The details for this feature extractor can be found in Appendix A.2.

The features required for the Linear Discriminant Analysis (LDA) state decoder in use in ESPA are a series of binned spike counts, saved temporarily in a vector, and then sampled 15 bin samples apart. This requires pre-processing of the data, some noise reduction, and a spike thresholder, along with memory to store counts. This task is split into three major task groups, as seen in Figure 3.9.

![Figure 3.9: Overview of the Feature Extraction subsystem.](image)

Despite the division, the majority of the processing occurs in the **Spike Counter**
block. The other two blocks, timeResolvedSelector and Class Means are "application specific blocks", designed to take the base feature, in this case spike counts, and further process them to provide the specific features needed for the particular decoder. Additionally, the Parameter Engine lives inside the Spike Counter, as evidenced by the parameters being fed via classMeans Parameters and covMatrix Parameters.

3.3.2 Spike Counter

The Spike Counter subsystem is the heart of ESPA decoding, containing the computationally intensive pre-processing filters and noise reduction components FIR Decimator 15k, CAR, and Tx Filter, as well as the MUA spike thresholding and binning function Thresholder. Spike Counter is also the home of the Parameter Handler, though this will be covered under Section 3.5.
Figure 3.10: System Generator view of **Spike Counter**. The gateways in the lower left entitled Data Input and Chan Input come from **wireless_packet_to_model**. Parameter Input and Parameter WE are fed from **params_load**, but sampled at the "model clock" WE Out provided by **wireless_packet_to_model**. thresh_valid (the enable), thresh_spikeCount and thresh_chanID are fed forward to **timeResolvedSelector** as a bus from **Threshold**.
Referencing Figure 3.10, data is spun out of the wireless_packet_to_model block one sample per channel at a time. The signed 12 bit value is fed in via Data_Input, and is accompanied by the corresponding 7-bit (required for 96 channels) channel value on Chan_Input. The data then enters the Unit Converter. The purpose of this block is to allow a scale factor to be applied to the input signal, which at this time, is unused.

Because many Xilinx IP Core modules use the AXI4-Stream interface[8], signals are kept in data/chanid/tLast format for most of the pre-processing section. tLast is a signal that should accompany the chanid signal, going high when the last channel is being fed forward. In this case, a tLast signal needs to be generated before entering the FIR Decimator 15k block. This is accomplished by comparing the current channel with the value of the highest channel number available (in this case, 95 is the index of the 96th channel, zero included,) as shown in Figure 3.11.

![Figure 3.11: tLast Generator uses a relational to create the third signal to conform to the AXI4-Stream standard. tLast, along with chanid and data are grouped together into a bus.](image)

### 3.3.3 FIR Decimator 15k

Inside the FIR Decimator 15k, an FIR Compiler 7.2 IP Core block is used to filter the data. In addition to the tLast, chanid and data being fed into the compiler, a reset signal is generated in order to enable the block with the correct timing, as it expects the first channel of data to be available on its first valid clock. Using trial-and-error, the Last Latch was developed in conjunction with the Relational that generates that signal 2 clock cycles ahead of the proper data arriving, as is mentioned in the aresetn documentation in the FIR Compiler block. This is accomplished by wiring together
a simple OR gate with a register, causing the signal generated by the **Relational** to be latched high permanently after it goes high the first time. In hindsight, a similar enable system for aclken probably could have been used.

**Figure 3.12**: Inside the **FIR Decimator 15k** block. The **FIR Compiler 7.2** block generates a Xilinx IP Core module of the same name.

**Figure 3.13**: Configuration Dialogs for the **FIR Decimator 15k**

A simple 30-tap FIR decimate-by-2 filter is set up within the parameter dialog – coefficients of “fir1(30,.5)” were chosen, with a decimation rate of 2 selected (shown in Figure 1.4). 96 interleaved channels are configured on the second tab. Coefficients are quantized with a width of 12 bits, and the output rounding is set to truncate least
significant bits with a width of 13. TLAST is set to Vector_Framing, again through trail-by-error functionality vs. the Packet_Framing. These changes can be seen in the dialog in Figure 3.13. The delay following the tLast output was determined to be necessary through timing simulation.

Figure 3.14: Frequency response of 'fir1(30,.5)', the coefficients used in FIR Decimator 15k. This is suitable for subsequent decimation of the data by a factor of 2.

The purpose of this filter is to reduce the data rate from it’s captured rate (in this case, 24kHz, but more generally with NSP recorded data, 30kHz) down to reduce computation load and filter length on subsequent filters. For instance, if NUA LFP filters were to be used subsequently, the size of these filters can be greatly reduced by starting with a base signal at 12kHz instead of 24kHz. In the case of the SUA thresholding filter in use in this ESPA model, reducing the signal down to 12kHz makes the area and latency of the subsequent bandpass filter in Tx Filter much
more efficient. The coefficients of ”fir1(30,.5)” produce a filter that has the response shown in Figure 3.14.

### 3.3.4 Common Average Reference

After the data is decimated, it is then run through a Common Average Reference, or **CAR** module. The purpose of the CAR is to reduce noise common to all channels. This is accomplished by taking the average value of all of the participating channels and subtracting that value from those channels. This results in electrical noise affecting multiple channels to be effectively removed. The **CAR** is configurable to allow exclusion of errant channels, as well as to enable or disable (bypass) the CAR system.

![Diagram of CAR System Generator model](image)

**Figure 3.15:** **CAR** System Generator model. Data entering through the **Parameters** port is fed from the **Parameter Engine**, while the neural data coming in through the **Data In** port is fed from **FIR Decimator 15k**.

Using Figure 3.15 as a reference, the two parameter (sets) that are read from the parameter engine are the **CAR Channel Loader** and the **Enable**. The **Enable** switches a multiplexer that can bypass the math output for the original value, with the same delay path. The **CAR Channel Loader** supplies the **Channel Selector** with data to update the **RAM module** inside which holds **Booleans** to decide which channels are used in the **CAR calculation**. Simply put, all of the data points from each channel in a given frame is summed using the **Accumulator**, which is simultaneously reset as
the result is dumped into the Register. Counter and Register1 is keeping track of
how many channels are included in the calculation, to be divided out by the Divide
block. This value is then subtracted from the appropriately delayed data line path
to produce a result which is time synchronized with the channel line, and can be
swapped out for the original data using the Mux.

3.3.5 Thresholding (Tx) Filter

The processing done by the FIR Decimator 15k as well as the CAR up to this
point is considered "pre-processing." The Tx Filter is considered to be the first
feature-specific block in the model. In this case, the Tx Filter is preparing the
neural signals for a threshold crossing detector by using a bandpass filter. In some
implementations of similar models before ESPA, this filter may have been done as a
series of two IIR filters, with a buffer and reversal of data between them. This regime
is called a non-causal filtering method, as it results in a filter with zero phase. This
sort of filter was implemented due to limited computation resources; the advantage
of using an IIR filter is that it can be computed with less coefficient multiplications,
or "taps." However, the drawbacks are filter frequency responses with poor edges,
as well as a non-linear phase. The latter of these issues can pose a real problem in
detecting spikes. To combat this, the non-cause filtering method was used[9], at the
expense of latency in the workflow.

One of the major advantages of ESPA is the ability to do expansive parallelized
tasks in fabric to generate an enormous amount of compute power. In this case, the
bandpass filter for thresholding can be implemented as an FIR filter, which gives the
advantage of both sharper frequency response edges, as well as providing a linear-
phase result that doesn’t require more processing or further tricks to correct. In
Figure 3.16, the frequency response of the former IIR filter and the new FIR filter
are compared.

The thresholding filter currently in use in ESPA is a 363 tap Kaiser Window
bandpass FIR filter, with a stop frequency of 100Hz and a pass frequency of 250Hz
on the low end, and a pass frequency of 5000Hz and a stop frequency of 6000Hz on
the high end. The amplitude of the low stop band is -60dB, and the amplitude of the
high stop band is -80dB. The high stop band is lower because the transition band is
(a) Frequency response of 8-tap IIR bandpass filter. Magnitude response is in blue; phase response is in red.

(b) Frequency response of 336-tap FIR bandpass filter. Magnitude response is in blue; phase response is in red.

Figure 3.16: While the most obvious difference between the frequency responses of the IIR filter in 3.16a and the FIR filter in 3.16b is the sharpness of the band edges of the magnitude response, the important distinction is in the phase response. The FIR filter has a linear phase response, whereas the IIR filter does not. This leads to the need of a "non-causal" filtering method described in [9], which requires two instances of this filter, and a buffer in between.

much wider, making this lower noise floor cost nothing extra (the driving factor in the size of this filter is the sharpness of the low band.) This results in a filter with the frequency response shown in 3.16b.

The major difference between the implementation of **Tx Filter** and **FIR Decimator 15k** (besides the initial configuration, such as decimation options as well as filter configuration) is **Tx Filter**’s ability to reload filter coefficients while the model is operating using the Parameter Engine. The lines coming from the Parameter engine can be seen in Figure 3.17 split out of the bus on port **Coeff Loader**. Because the **FIR Compiler 7.2.1** block is preloaded with coefficients generated from the **FDATool** block, the config_tdata_data port is unused (and tied to zero.)

In order to reload the coefficients, the config_tvalid and reload_tvalid lines must be pulled high for the duration of number of cycles equal to the length of the filter, loading in one new filter coefficient each cycle into reload_tdata_data. On the last coefficient, per the AXI4-Stream standard, the reload_tlast signal must be pulled high.
Figure 3.17: **Tx Filter** System Generator model. Data entering through the **Coeff Loader** port is fed from the **Parameter Engine**, while the neural data coming in through the **Data Bus In** port is fed from **CAR**.

The filtered neural data emerges from the **FIR Compiler 7.2.1** block in a 34 bit format, which is truncated back to a 13 bit format. **FIR Compiler 7.2.1** has rounding options in it’s "Datapath Options", but in this case, truncating is done manually to allow exact selection of bits based on testing with actual signals.

The error signals are aggregated into a bus and passed out the **Error Handling Out** port. The filtered data in AXI4-Stream format is aggregated into a bus and passed out **Data Bus Out**.

### 3.3.6 Thresholder

The **Thresholder** block is the heart of the Thresholding feature extraction process. It’s role is to take the filtered data and apply a pre-determined threshold to each channel in order to detect action potentials, or "spikes". These spikes are then binned into time intervals, and output as "spike counts."
Figure 3.18: System Generator view of **Threshold**. The Parameters port brings in new thresholds addressable by channel, as well as a configuration to threshold rising or falling edges. **30k Filtered Data** brings in the filtered data from **Tx Filter**, and **Out1** sends the binned spike counts out.
The thresholding values are computed offline using a sample of data recorded after pre-processing, right before the **Thresholder**. Typically, a value which a multiple of the RMS value of the channel (typically $3 - 4 \times RMS$) is selected as the threshold level for a given channel. These values must then be loaded into the **Thresholder** by means of the Parameter Engine.

Bandpass filtered neural data enters the **Thresholder** through the **30k Filtered Data** port. The channel information is fed to the **Single Port RAM** (provided the data-loading **Mux3** isn’t selected from the **Enable** parameter from the Parameter Engine), which stores the threshold values for each channel. The data goes through the **Delay2** block to match the latency. **Mux** and **Mux1** then decide which value (the incoming data and the stored threshold) get subtracted from the other in the **AddSub** block based on the **Invert Threshold** line from the Parameter Engine. The purpose of this selector is to determine whether to trigger on the rising edge or the falling edge of the spike.

The result of the subtraction in **AddSub** is then compared against zero with **Relational**, and if it’s greater than, this event is now considered to be part of a spike. The circuitry surrounding **Dual Port RAM1** is used to essentially ”latch” this event for the given channel, to then XOR this latch using **Logical1** with the spike event, and subsequently ANDing it with itself using **Logical** in order to produce just a single value for a given spike event, even if the duration of the spike is longer than one sample (which it most likely is.) This boolean is then cast to an integer (always of 1) using **Convert**.

This spike event is then summed with the running total of spike events for that channel using **AddSub1** and stored back into **Dual Port RAM2**, which carries the running totals. **Mux2** is used to clear the running total when the bin is over. **Single Port RAM1** is used to latch the final bin count when the bin is on it’s last sample, for output to the **Out1** port. Delays are used throughout the process to keep the channel number aligned with all of the various memories.

Finally, the signal to determine when to advance bins is calculated using the logic on the top of the model. Each time the last channel of the set is cycled in, the **Accumulator** is increased by 1. The value of the **Accumulator** is compared against the **Bin Size** parameter, and when they’re equal, the boolean line is used to trigger
Mux2 to clear the counts, and Single Port RAM1 to latch the current counts for output.

Figure 3.19: Functional demonstration simulation of Thresholder. The blue signal on the bottom scope is the input signal, with an amplitude of 375 (cut off by top of scope). The threshold on all of the channels of this Thresholder is set to 300. The yellow signal on the bottom scope is outBin, which is signaling when the data out is valid. The top signal is the data out, or spikeCount. We count 6 blue input spikes before the valid signal, at which point the value on spikeCount is 6. This is succeeded by 5 more spikes, where we see the value on spikeCount drops to 5 during the valid signal on outBin. Finally, 6 more spikes are seen during that time period, and this is again reflected in the output.

3.3.7 timeResolvedSelector

The purpose of timeResolvedSelector is to buffer and sample the spike count neural features. The LDA decoder in use utilizes features which are delayed and sampled spike counts. Each channel of data is buffered for 15 samples and summed over a rolling buffer (Figure 3.20), and then these summations are buffered again and sampled at the 0th, 12th, and 25th newest samples (Figure 3.21). This results in a total of 96 × 3, or 288 features. The logic in Figures 3.20 and 3.21 demonstrate several FIFO and RAM cells being utilized to appropriately buffer, sum, rebuffer and sample this data. Additionally, custom FIFO controllers were created in order to pre-pad the FIFOs to the proper length to implement the correct delay. The logic for these controllers can be found in Figure 3.22.
2k was picked as we have 96 channels*15 values in the chain.

Figure 3.20: System Generator view of \texttt{timeResolvedSelector} Part 1. This portion of the circuit creates the rolling summation and the FIFO to buffer the original values to subtract out of the summation once the window has "passed."
Figure 3.21: System Generator view of timeResolvedSelector Part 2. Once the rolling summations are computed, they buffered and sampled at the 0th sample (Delay5), the 12th sample (Delay6) and the 25th sample (dout of FIFO2.) The Time Division Multiplexer reorganizes these signals into a streaming set of 288 features, output port Out1.
Figure 3.22: Control circuit FIFO Loader is used to preload FIFOs in timeResolvedSelector with the appropriate number of values to define it’s delay length.

3.4 Decoding

3.4.1 Neural Decoding Overview

"Neural Decoding" is the process of decoding data that has been ”encoded” by a network of neurons in the brain.[10] The primary goal is to use a processing algorithm on pre-processed features to reconstruct a ”message” encoded in ”neural code.” Examples of desired decoded messages may include visual stimulus, sensory information, or motor intention.

Decoding methods can be applied to any type of neural data, including NUA, SUA and MUA features. The type, number and quality of features typically dictate how well a decoder may work, using some measure of accuracy. The features used in the ESPA LDA decoder are SUA binned Action Potential rates from 96 channels of 20ksp MEA data.

Processes to decode neural data are a heavily researched topic, with the ”state of the art” changing almost constantly. One of the biggest challenges in determining which methods to use for a mobile case is finding a method that is realistically able to
be computed in real-time deterministically, meaning that the compute power required isn’t too excessive, and the latency involved to collect enough data doesn’t hamper the ability to control a prosthetic device.

3.4.2 Decoding Schemes

Many research groups have developed and tweaked decoders used in many different realms, from EEG, ECoG and MEA inputs and countless end-effectors. Two of these decoders that are of interest to the initial application of ESPA are an LDA state decoder, and a Kalman filter velocity decoder\[11\],\[12\]. The LDA state decoder has been implemented, and is described in Section 3.4.3, while the Kalman decoder (currently in use in the BrainGate trials\[13\]) has not yet been developed for ESPA.

One of the neural-prosthetic control schemes used in the BrainGate trials include a combination of a Kalman filter for velocity decoding, and an LDA state decoder to decode for a ”click” or grasp state.\[14\] This allows for 3D endpoint control of a robotic limb (or 2D control of a cursor on a screen), as well the ability to detect the intention to ”grasp” the hand or ”click” the mouse. An example of some data that shows clear preference on certain neurons for these types of actions are found in Figure 3.23. One could imagine using a large ensemble (of up to 96 or 192 channels, depending on setup) to statistically infer intention to move in a multi-dimensional space using pattern recognition. It is intended that a Kalman decoder will be implemented in ESPA in the near future.
Linear Discriminant Analysis (LDA) is another decoding method that results in classification of states. LDA was originally developed by R. A. Fisher [15] in 1936, and is a simple, robust method of reducing dimensionality of a data set and predicting which state (or class) an input data set belongs to. In the case of this ESPA model, the input data set is a vector with 300 neural features, and the predicted states are the stages of the gait of a non-human primate walking on a treadmill.
The goal of LDA is to maximize the following equation:

$$J(W) = \frac{W^T S_B W}{W^T S_W W} \quad (3.1)$$

where $S_B$ is the "between classes scatter matrix" and $S_w$ is the "within classes scatter matrix". $W$ is an eigenvector matrix that defines the subspace that the multi-dimensional data is being projected upon.

First, the matrix $W$ must be computed using training data ($x$) and the following method[17][18]

- Class mean vectors are calculated

$$m_i = \frac{1}{n_i} \sum_{x \in D_i} x_k \quad (3.2)$$

- The within-class scatter matrix $S_W$ is calculated from a sum of the scatter matrix for each class

$$S_W = \sum_{i=1}^{c} S_i \quad (3.3)$$

$$S_i = \sum_{x \in D_i} (x - m_i)(x - m_i)^T \quad (3.4)$$
• The between-class scatter matrix $S_B$ is calculated

$$S_B = \sum_{i=1}^{c} N_i (m_i - m)(m_i - m)^T$$  \hspace{1cm} (3.5)

• Differentiating equation 3.1 w.r.t. $W$ and equating to zero yields

$$\frac{2S_B W}{W^T S_W W} - \frac{W^T S_B W}{(W^T S_W W)^2} 2S_W W = 0$$  \hspace{1cm} (3.6)

• Implying that $S_B W$ is in the same direction as $S_W W$. This means that any maximizer of $J(W)$ must satisfy

$$S_W W = \lambda S_B W$$  \hspace{1cm} (3.7)

which is a standard eigenvector problem. The eigenvector matrix $W$ is solved using computational methods from $S_W^{-1} S_B$

Once this $W$ is determined using a training set of data, new data points ($x$) can be classified by projecting it into each maximally separating direction and classifying it as such

$$W^T \left( x - \left( \sum_{i=1}^{3} m_i \right) \right) > \log \frac{p(c_i)}{p(c_{tot-1})}$$  \hspace{1cm} (3.8)

This is the computation that ESPA is performing on-the-fly.

### 3.4.3 LDA Classifier Implementation

#### 3.4.4 Class Means

The first stage of the LDA Decoder is the calculation of distance vectors by subtracting the pre-computed class means for each of the three classes from the feature.

$$\left( x - \left( \sum_{i=1}^{3} \frac{m_i}{i} \right) \right)$$  \hspace{1cm} (3.9)

While conceptually this is a piece of the decoding process, from a programmatic stand-point, it is computed in the same time-domain as the rest of the feature extraction process, so it falls under the **Feature Extraction** domain.

Each class mean value is pre-computed off-board ESPA using a training set of data. Three sets of these values are generated, one for each class, resulting in $3 \times$
288 values in total. These values are loaded into the respective memory blocks, `classMeans1`, `classMeans2`, and `classMeans3` using the Parameter Engine fed over the Parameters port.

**Figure 3.25:** Class Means module subtracts the 288 features generated by timeResolvedSelector from the pre-calculated class means, which are stored in the three RAM modules in the center of the diagram, and loaded in through the Parameters port.

Before the values are subtracted, the incoming features are first converted to floating-point numbers using the Convert and Convert1 blocks in Figure 3.25. This value is then multiplied (using a the Divide block with a divisor less than 1)
by a constant to adjust for the number of samples taken into consideration against the class mean values. The subtracted values (calculated via the AddSub series of blocks) are then sent out via port Data Out for computation by the decoder.

3.4.5 Time Shifting Paradigm

Up to this point, the entire System Generator model as described is operated in the same time domain, with every block being executed on a clock that is (or a divisor of) the clock generated by the neural data transmitter. Shifting into the decoding realm, it is much more useful to move into a faster, on-board crystal-based time domain. This allows for faster serialized computation, particularly when it comes to large number of multiplications, like those required for matrix multiplication. In this case, the neural clock driven model produces new features, and then ”triggers” the transfer of data and the kickoff of the decoding processing in a separate, crystal-driven subsystem.

System Generator allows the use of specific RAM modules (namely, Dual Port RAM and FIFO) outside the scope of a System Generator token configured for a particular clock domain, as long as they fall under a System Generator token that is configured for ”Enable multiple clocks”. This allows for transfer between clock domains. In ESPA, the three subsystems Class 1, Class 2, and Class 3 facilitate the transfer of the calculated \( (x - \left( \sum_{i=1}^{3} m_i \right)) \) values from each class. These are made available in two identical matrices, \( X \) and \( X_1 \), shown in Figure 3.27, which allow matrix multiplication of inter-matrix values. Data is loaded through Port A from the Feature Extraction and is read through Port B from Speedy Calculator. The process signal is fed forward via the m_FIFO in Figure 3.26. This lets the Speedy Calculator module know that new data is available for each of the classes. Finally, pre-calculated Covariance Matrix data \( W^T \) is loaded from the Parameter Engine into m_covInvMatrix for access by the Speedy Calculator.
Figure 3.26: Data calculated in the Feature Extraction module is transferred to the Speedy Calculator via FIFO and RAM modules. The System Generator token enables the configuration of both of the subsystem System Generator tokens to operate under different clock domains. The visual description of the transfer mechanism in Class 1, Class 2, and Class 3 can be seen in Figure 3.27.

Figure 3.27: Memory transfer system for time domain switching of the three sets of \( x - (\sum_{i=1}^{3} \frac{m_i}{t}) \) values. The internals of all three subsystems, Class 1, Class 2, and Class 3 are identical.
In order to operate the shared memory, a Memory Controller was created. The details for the operation of this memory controller can be found in Appendix A.3.

3.4.6 Probability Calculations

Figure 3.28: Overview of Speedy Calculator decoding scheme. This subsystem operates on an independent clock from the Feature Extraction subsystem – a 250 MHz crystal onboard the ZC-706.

Speedy Calculator is split into two major parts, the probability calculators (Class 1, Class 2, and Class 3 in Figure 3.28), and the Classifier. Additionally, the Memory Subsystem handles the transfer of data between clock domains, and the specifics of this can be found in Appendix A.3.

The Probability Calculator is where the \( (x - \left( \sum_{i=1}^{3} \frac{m_i}{T} \right) \) values generated by Class Means are multiplied by the \( W^T \) covariance matrix term from Equation 3.8. This is done as a matrix multiplication, and is done across all three classifications at once (they share a covariance matrix, one of the core tenants of LDA.) The result of this calculation is \( m_{\text{tmpProb}} \), which is then used by the classifier to rank against the other probabilities to determine a state output in the Classifier.
Figure 3.29: Matrix Multiplier to multiply the $W^T$ covariance matrix by each class’s $(x - (\sum_{i=1}^{3} m_i / \lambda))$ term to generate class probabilities.

### 3.4.7 Classifier

The final step in the decoding model is to rank the probability values generated by the probability generators to create the class expectations.

Defining

$$m_{\text{tmpProb}} = \left( x - \left( \sum_{i=1}^{3} \frac{m_i}{\lambda} \right) \right)$$

(3.10)

For a given class $c$, this expectation would be calculated as (from Equation 3.8)

$$\text{expectation}(c) = \frac{1}{\sum_{i} e^{m_{\text{tmpProb}}[i] - m_{\text{tmpProb}}[c]}}$$

(3.11)

Which may be written in pseudo-code as

Listing 3.1: Code example describing functionality of the **Classifier**

```plaintext
for cl = 1:noOfClass
    sum = 0;
    for na = 1:noOfClass
        sum = sum + exp(m_tempProb[na] - m_tempProb[cl]);
    end
    expectation[cl] = 1 / sum;
end
```

and is implemented in System Generator as
This results in three signals indicating which state is currently being decoded from the neural data. This particular implementation of the feature extraction and LDA decoder pipeline in ESPA was designed to process data for an experiment involving a corticospinal neuroprosthetic to restore locomotion after a spinal cord injury.[19] This particular implementation of the LDA decoder is aimed at decoding intention of the various gait states of an animal’s leg during locomotion on a treadmill, in order to generate control signals for a spinal cord stimulator, allowing the animal’s own leg to act as a prosthetic. While the ESPA implementation of this decoder was never used in vivo, the algorithm was tested off-line using recorded data from an animal, and run through simulation software in conjunction with the ESPA design. Sample data can be seen in Figure 4.4.
Figure 3.31: Waveform vector of expectation of each class output of the LDA as it pertains to NHP treadmill data. Following from left to right, Class 1 and Class 2 are approximately inversely related, oscillating between Class 1 and Class 2. More difficult to see are the breaks in the line of Class 3, where some non-zero values are calculated periodically (often aligning with the sharp value spikes seen in Class 1 and Class 2.) Class 3 viably comes above zero for a notable period of time near the end of the plot, between time markers 7 and 8. The time scale for this plot is 10ms.

In the future, additional modules will be implemented in ESPA, increasing it’s utility in a variety of experiments. For example an LFP power calculation feature extractor has already been implemented, a block diagram of which and some sample outputs can be found in Appendix A.2. A Kalman Filter decoder is planned, which will enable 3D end-point velocity computation for a prosthetic limb, with possible uses in the BrainGate and other projects.

### 3.5 Parameter Engine

#### 3.5.1 Overview

One of the key components in making ESPA a viable platform for doing real-time neural data computation is the ability to adjust and tune parameters in the processing model on-the-fly, while the data processing model is running. In a typical CPU application, this is a simple task; writing new values to registers or RAM is as easy as addressing memory. However, when dedicating FPGA fabric to do similar tasks, the ability to change these values on the fly is much more complex. I developed a novel Parameter Engine as a simple way of modifying pre-determined parameters within the model using single commands written to the model asynchronously from the QNX shell.
Briefly, the design of the Parameter Engine is similar to that of a small microprocessor. It takes in instructions, called OP-Codes, followed in some cases by operands. It then updates the appropriate registers, and generates the appropriate control signals to update the other hardware components in the ESPA model. The instructions are streamed in via a piece of glue code, \texttt{params\_load.v}, which has an interrupt interface with the ARM processor. An application running in QNX takes in a file input which contains the "machine code" set of OP-Codes and operands, and feeds them to the glue code running on the FPGA (depicted in Figure 3.32.) These values are then streamed into the System Generator model, and into the \textbf{Parameter Handler} block.

Figure 3.32: \texttt{params\_load} module takes data from shared BRAM from the ARM processor (triggered by the \texttt{qnx\_int} signal) and reads in 32-bit values from BRAM\_PORTB\_1\_dout, utilizing the \texttt{fast\_clk}. Data is then spun out to the System Generator model, one 32-bit chunk at a time, using the \texttt{param\_out} bus, with the \texttt{data\_clk} patched through to the \texttt{param\_we} during this time.

The \texttt{params\_load} block takes care of synchronizing the parameter data to be loaded with the neural-data clock, effectively shifting time domains between ARM and neural. The Op-Codes are organized into groups depending on function. "Enables" and "Toggles" are in the "A" group, "Vector Loading" are in the "B" group, and "Sets" are in the "C" group. Table 3.3 shows a comprehensive list of all of the currently available Op-Codes.
<table>
<thead>
<tr>
<th>Function</th>
<th>OP-Code</th>
<th>Operand 1</th>
<th>Operand 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable CAR</td>
<td>00A0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Disable CAR</td>
<td>00A1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set Normal Threshold</td>
<td>00A2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set Inverse Threshold</td>
<td>00A3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Valid Channels</td>
<td>00B0</td>
<td>Channel</td>
<td>Enable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[0...end] LSB SET</td>
<td></td>
</tr>
<tr>
<td>Load 30k Coefficients</td>
<td>00B1</td>
<td># of coefficients</td>
<td>Coefficients[0...end]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>12 BIT FROM LSB</td>
</tr>
<tr>
<td>Load classMeans1</td>
<td>00B2</td>
<td>classMeans</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[0...287] float singles</td>
<td></td>
</tr>
<tr>
<td>Load classMeans2</td>
<td>00B3</td>
<td>classMeans</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[0...287] float singles</td>
<td></td>
</tr>
<tr>
<td>Load classMeans3</td>
<td>00B4</td>
<td>classMeans</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[0...287] float singles</td>
<td></td>
</tr>
<tr>
<td>Load covInvMatrix</td>
<td>00B5</td>
<td>row #</td>
<td>covInvMatrix</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[0...287] float singles</td>
</tr>
<tr>
<td>Set Bin Size</td>
<td>00C0</td>
<td># of samples</td>
<td>16 Bit Value (unsigned)</td>
</tr>
<tr>
<td>Set Threshold on Channel</td>
<td>00C1</td>
<td>Channel #</td>
<td>Threshold Value 12 BIT FROM LSB</td>
</tr>
</tbody>
</table>

Table 3.3: OP-Codes available for the Parameter Engine.
This is any of the 00 Ax Opcodes, intended for Enables.

This is any of the 00 Bx Opcodes, intended for loading data.

This is any of the 00 Cx Opcodes, intended for setting stored registers.

---

Figure 3.33: System Generator view of **Parameter Writer**. Signals are simply passed through the **Parameter Handler** shell. The Enable for each subsystem is controlled by both the input Enable, as well as determining which of the classes of Op-Codes are being using (A, B or C) by slicing and comparing that portion of the OP-Code with the respective number (depicted here in binary.)
The model diagram depicted in Figure 3.33 is the system overview for the Parameter Engine. Each of the three subsystems depicted (Enables, Data Loading, and Stored Parameters) are hardware tied to categorically separated commands described in Table 3.3. Each of these categories is associated with a set of OP-Codes (00Ax, 00Bx and 00Cx, respectively.) The slice logic in combination with the relations and the incoming Enable signal determine when to trigger the respective subsystem for processing.

The general methodology for processing each OP-Code is the same.

1. Determine which circuit to activate based on the OP-Code
2. Make another circuit activation decision if the OP-Code has a second part
3. Continue activation of the appropriate portions of the circuit by propagating and/or delaying the enable associated with the OP-Code for the appropriate number of clock cycles
4. Activate internal logic to store, pass, or any other function pertaining to the operation of the OP-Code
5. Propagate the corresponding command signals out to the remainder of the model, both data and enables

Many of the data outputs from various OP-Code processing logic blocks will pass through data from the command line, even when that module isn’t activated. This is because many of the blocks are designed to simply gate or create an enable signal for the corresponding model logic only during clock cycles that are pertinent; the data isn’t gated because it isn’t necessary that it’s gated. In this sense, many of these data lines out act in a quasi-bus nature.

The ability to process and update these parameter values is critical, as this is the only way for a user to control the system while it is running. When the system is first started, the model parameters built into the bit file for the FPGA are loaded, which for many large memory spaces are zeros. As an example, the vectors and matrices used in the decoding model aren’t loaded on boot, and need to be calculated from a training set of data. In the future, this training process may be part of ESPA’s functionality, but is currently done off-board by the back-end server. Once these are
calculated, they are loaded using the Parameter Engine into the model for open-loop data processing. Additionally, the threshold values for spike counting on each channel need to be calculated from raw data, and subsequently need to be loaded into the model.

While a clinician is operating the system, they may want to tweak certain values in the processing system. For example, a clinician may be looking at a spike panel, and notice that the amplitude of the spikes on a particular channel are lower than what the threshold is currently set at. The clinician would then want the ability to go into that channel from their spike panel, and readjust that threshold value such that those spikes are properly accounted for with their new amplitudes. In order to make that update in the model, the back-end server would take the data from the user application and run a script that calls the Supervisor running on the ARM, which in turn forwards the new threshold data into the appropriate memory space in the System Generator model.

There are two basic types of storage for parameters; parameters stored in memory in the Engine, and parameters stored in-place in the model, for which the Engine acts as a memory controller and loader. In the former case (typically single data values), the memory is allocated as a register directly in the Engine, and the contents of that register are pulled out as a line to the appropriate processing block in the model. An example of this are enables, held in boolean registers in the Engine, whose outputs are tied to selector lines on Muxs. Another example are single integers, such as the Bin Size for the thresholder, where the integer number of samples is stored in a register and piped out to the comparator in the Thresholder that uses this value.

The other category of parameter memory storage is in-place memory, where the Engine acts as a loader. In this case, the Engine correctly pulls the enable lines for that memory and synchronizes with the incoming data over the bus from the ARM loader. Examples of this include the classMeans and covInvMatrix, which are stored in the place where they are used. Another example is the re-loadable FIR filter Tx Filter, which keeps the coefficients in memory in the IP core block. In this case, the Engine pulls the correct lines in time in order to load the new values from the ARM interface, complying with the AXI4-Stream standard.
3.5.2 Enables

The **Enables** subsystem (Figure 3.34) is the simplest of the three classifications of parameters. OP-Codes of this designation set or unset an enable line in the processing model, and this state is held in a single register. The relational blocks on the left determine when the appropriate OP-Code is being called in order to enable the register to receive the new value (CAR Enable Register or Threshold Invert Register.)

The output of the registers is persistent, and is down-sampled (if necessary) to match the portion of the model that it’s interacting with.

![Diagram](image)

Figure 3.34: **Enables** module of the Parameter Engine. Responsible for OP-Codes 00A0-00A3.

3.5.3 Data Loading

The **Data Loading** module is responsible for handling all of the OP-Codes related to streaming in large quantities of data into registers or RAM (memory) internal to the model. This can be a complex operation, particularly because the length of the data being loaded may be variable.
There are six OP-Codes available in this class, of which, three are identical but have different targets (the classMeans loaders.) The first two, Load Valid Channels (00B0) and Load 30k Coefficients (00B1) are shown in Figure 3.35. The three Load classMeans loaders are identical, and are representing in Figure 3.36 by Load classMeans3 (00B4), as well as Load covInvMatrix (00B5).

Figure 3.35: **Data Loading** module of the Parameter Engine (part 1.) Responsible for OP-Codes 00B0 and 00B1.

The top portion of the model in Figure 3.35 is the logic that accepts parameters to enable or disable channels when performing CAR calculations. After the OP-Code is triggered, the following ninety-six values will be sampled and sent to the memory in the CAR block for storage. The OP-Code triggers the **Mealy State Machine** which operates under the state logic found in Table B.1. The data value passed forward is the LSB of the parameter bus, where a 1 includes the channel in the CAR calculation, and a 0 precludes it. After the ninety-sixth value is read, the output Enable line is pulled low.

The bottom portion of the model in Figure 3.35 is the logic that reloads the Tx
**Filter** with new coefficients. This loader is a little different, in the sense that the length of the vector being loaded in is variable, depending on the length of the FIR Compiler 7.2 configured. This means that the user must know how large the filter is currently configured to be, and needs to design a filter of the same length.

For this loader, first the OP-Code is loaded (00B1), followed by an integer representing the number of coefficients to be loaded (again, this should match the length of the filter in **Tx Filter**.) The following values to be fed in are the 12-bit coefficients (from LSB, top 4 discarded) for the filter. The **Mealy State Machine** operating under the state logic in Table B.2. The FIR Compiler 7.2 block requires not only the data, but also for the \( cValid \) and \( rValid \) lines to be pulled high during the operation. Additionally, the \( rLast \) line should be pulled high as the last coefficient is fed in. The **Mealy State Machine** takes care of the logic to drive these signals.

![Diagram](image-url)

Figure 3.36: **Data Loading** module of the Parameter Engine (part 2.) Responsible for OP-Codes 00B4 and 00B5.

The top portion of the model in Figure 3.36 is the logic specifically to load the classMeans3 vector into memory, but is identical for both classMeans1 and classMeans2. This is a "classic" loader, almost identical to the CAR loader (using the same state logic,) but the counter logic is pre-programmed for two hundred and
eighty-eight values.

The bottom portion of the model in Figure 3.36 is the logic used to load values into covInvMatrix. Identical to the classMeans loader logic, this loader has an additional piece of logic implemented which allows an offset to be fed in as an operand. In order to reload the entire matrix, the OP-Code must be called two hundred and eighty-eight times, each time accompanied by the memory offset, as well as two hundred and eighty-eight new values (the matrix is two hundred and eighty-eight square.) Individual rows can be loaded by simply calling the command once, and specifying the offset as the operand (see Table 3.3.)

### 3.5.4 Stored Parameters

Similar to the Enables classification of OP-Codes, Stored Parameters provides persistent memory that can be modified from the outside using OP-Codes, however, the Enables OP-Codes simply toggle states, while Stored Parameters OP-Codes have operands that allow specific values to be stored in the persistent memory.

The two OP-Codes currently implemented (Figure 3.37 & Table 3.3) in the Stored Parameters class are Set Bin Size (00C0) and Set Threshold on Channel (00C1). In the case of Set Bin Size, the OP-Code is provided, followed by the Operand, which is a 16-bit value representing the desired Bin Size. 1 is subtracted from this value (as it’s used in a 0 indexing regime,) and stored in Register1. This value is then available for use by the Thresholder.

In contrast, the Set Threshold on Channel command set temporarily stores both operands (the channel in question as well as the threshold to be set) in persistent memory (Register2 and Register3, respectively), followed by the generation of an enable signal (Enable) to save the data to the block RAM in the Thresholder.
Figure 3.37: Stored Parameters module of the Parameter Engine. Responsible for OP-Codes 00C0 and 00C1.

### 3.6 Data Aggregation & Output Glue Logic

Aggregating data from the model and transfer to ARM happens in four major steps:

1. Data is aggregated internally to the System Generator model for organizational purposes (optional) and set out of the model via a Gateway Out block with an accompanying write enable signal.

2. All write enables from different model outputs are sampled to see if any data has been updated by a module defined by model.out.we_check.v.

3. Once the model.out.we_check determines there is new data, the signal model_global_we signals to repack_model that new data is available. repack_model samples all of the data outputs from the model and saves them internally. (Optionally,) multiple frames of data are aggregated in repack_model.

4. After the repack_model data buffer is full, the entire frame is transferred to an instance of wireless_packet_to_bram.v for transfer to BRAM and signaling to ARM.
Figure 3.38: HDL block diagram of modules to aggregate data and transfer from System Generator model to ARM. In this example, agg_data, agg_chanid, class0, class1, class2, m_tmpprob1, m_tmpprob2, m_tmpprob3, and thresh_spikecount are packed in repack_model_out_245 each time. agg_valid, class_valid, m_tmpprob_valid, and thresh_valid are used to trigger we_check to produce a model_global_we signal to force repack_model_out_245 to capture. Once the buffer in repack_model_out_245 is full (5 data frames,) the data is transferred out on data_out triggered by WE_out to wireless_packet_to_bram for transfer to ARM.
References


4

Discussion

4.1 Validation

Validation occurs in three major stages for each iteration of the model. First, the System Generator model is simulated in SIMULINK using MATLAB test vectors. This ensures proper operation of the circuit from a functional standpoint. The IP core is then generated, and simulated both behaviorally as well as in post-synthesis in Vivado. Finally, the system is built, and the bit file is loaded onto the Zynq itself, where it is tested either with a data generator running on-chip, or with an external data source.

4.1.1 SIMULINK Functionality Verification

The first step in validating the design is ensuring correct operation of the System Generator model. One of the incredible strengths of using System Generator is the ability to simulate the model directly in the Mathworks products using SIMULINK, and the ability to pull test data directly from the MATLAB workspace. Signals can be probed at any point in the model, and either put into waveform view on a scope, or saved to a vector for sample-by-sample analysis.
Figure 4.1: A simulation doing in SIMULINK with data plotted in MATLAB. The test vector is defined in Appendix B.6. This waveform in this figure is the output of the CAR block on channel 1. When the CAR is enabled on channel 1 (all inputs except channel 3 are the same,) the voltage value goes to 0. When channel 3 is also turned on (with value of 0,) the result is an average of the two signals. Finally, when the CAR is turned off, the original signal is seen.

Another key strength of using the Mathworks software is the ability to create "testbench" files that generate data for simulation in SIMULINK, but then can also be compiled down to HDL code for validation in the subsequent two steps using the Mathworks HDL Coder product.[1] Such files are written in Embedded MATLAB Language (EML), which is similar to regular MATLAB, with constraints on functions that can be used, memory management, and I/O. A very simple testbench EML can be seen in Listing 4.1.

Listing 4.1: EML example generates ramp data across 96 channels (0 indexed)

```matlab
function [chan_output, data_output] = fir15k1test()
    persistent chan data;

    if isempty(chan)
        chan = 0; data = 0;
    end

    chan_output = chan;
```
data_output = data;

if(chan == 95)
    chan = 0;
    data = data + 1;
else
    chan = chan + 1;
end
end

In addition to using EML testbenches, built-in SIMULINK blocks and sources can be used in the System Generator model, provided that they exist outside the “Gateway” framework, and are formatted and fed in as if it were data entering ports on an HDL structure. System Generator automatically takes care of timing and sampling, even allowing the sample times on the source blocks to be modified to match the appropriate inputs. An example is a test vector generated in MATLAB in order to specifically test the functionality of the Parameter Engine’s interaction with the CAR; the definition of such a vector can be seen in Table B.6.
Figure 4.2: SIMULINK waveform scope showing proper operation of theThresholdolder module. The first and fourth waveforms are channel numbers (continuous ramping data), while the second waveform is the output of the thresholding vector. The third waveform shows the latched value for number of threshold crossings during the time period, while the forth waveform shows the time in which those latched values are valid for transport. The first bin just catches the first waveform, hence the value of 1. The next several bins catch the three next waveforms, and the second to last bin catches four of them (due to planned input data frequency). Finally, the threshold value is changed across all channels, and the bin count goes to 0 at the end.

4.1.2 Vivado Simulation

After the System Generator model has been simulated in SIMULINK, the model is then converted to an IP Core block and integrated with the rest of the ESPA SoC system in Vivado. This includes all of the glue logic to interface with the external peripherals, as well as the ARM, as outlined in 3.2 and 3.6. The system then needs to be functionally verified in Vivado to ensure that the transition from SysGen model to HDL appropriately occurred. Similar testing needs to occur after the model has been synthesized as well, to ensure that nothing critical to operation was cut in the synthesization process.
Behavioral Schematic and Simulation

Once the IP Core has been integrated into the Vivado project, a behavioral schematic model is generated. This can be manually inspected to better understand how signals were routed through the System Generator process. From the schematic view, signals can be tagged for simulation to verify outputs. An example of the schematic view is seen in Figure 4.3, and the corresponding waveforms can be seen in Figure 4.4.

Figure 4.3: An excerpt of the generated HDL Schematic view of ESPA. Shown in the figure is part of the CAR module, with the Accumulator and Register1 exploded to show basic components like gates, adders and registers.
Figure 4.4: Vivado Behavioral Simulation output of the full System Generator model plugged into the Xilinx Zynq hybrid-subsystem. Depicted here are the outputs of the classifiers, as they shift to an unknown "neutral" state on their first successful computation (no non-zero features had been computed at this point.)

Post-Synthesis Simulation

After the system has been synthesized, it must be simulated again, as occasionally vital components to the system may have been cut. For this step, it is easier to define test points as "Gateways" in the original model in order to test at various points in the workflow, as many wires are cut and renamed in the synthesization process.

In a normal workflow, when signals and registers are cut that aren’t desired to be cut, synthesis directives can be inserted into the HDL code to prevent this from happening. However, because this HDL code is generated by the System Generator tool, this is not an option. Inside, the original dialogs in System Generator must be manipulated, and in some cases, alternate circuits must be defined in order to avoid these circumstances from occurring. Another trick can be to for signals as output gateways, even if the signal is not to be used. This will prevent it from being cut. In general, the outputs of the post-synthesis simulation should match the behavioral simulation.
4.1.3 On-Chip Verification

The final validation step is to load the generated bit-file onto the Zynq to verify output. Generally, an HDL test generator is used at first, verifying the output vector via the data streamed out via UDP. After this functionality is confirmed, the data input can be switched to an actual source (like a wireless neural data receiver) for validation.
4.2 Requirements for Future Components

4.2.1 Custom ESPA Hardware

While the data and design presented in this work has been developed and tested on a Xilinx Zynq reference board (ZC706)[2], ESPA is currently in the process of development to become a dedicated, battery operated, wearable hardware system. From a software design aspect, the system will be identical, but from a hardware aspect, the system will be much more usable in a practical, clinical sense. Researchers in the Nurmikko lab are working with Bay Computer Associates (BCA) to make this compact system a reality.

![Concept drawing of Custom ESPA Hardware.](image)

Figure 4.6: Concept drawing of Custom ESPA Hardware.

In addition to miniaturizing the ESPA core system (known as the "Motherboard"), a miniaturized version of the wireless receiver used for telemetry of data from Brown
EBNC and SBNC devices is being developed to "clip on" to the ESPA motherboard. This would enable a totally portable solution, from neural implant to prosthetic, without the need for wall-power or desktop computers.

4.2.2 Hardware for Back-end Server

Two preliminary hardware regimes for a back-end server have been developed. The first (Concept A) is a portable, tabletop system containing a small disk storage space, a mini-computer, and wireless hardware to communicate with ESPA. The second (Concept B) is a portable rack-mounted solution that contains a full-fledged server blade, a RAID array to store large amounts of data, smart networking switches, power management and backup solutions, and wireless hardware.

Some design elements and benefits/detractors of the first, portable design (Concept A) are:

- A latched plastic electrical box with a transparent front is used to store and connect all of the components
  - Wireless Router
  - NUC PC
  - Portable storage array
  - Display/keyboard/mouse
  - Power strip
- The NUC acts as the backend server, booting directly into the selected OS
- The display is mounted to the back of the transparent cover
- The keyboard/trackpad combo are mounted to the exterior of the box
- PRO: Compact in size, low power requirements
- PRO: Low cost
- CON: Severely reduced processing and storage capabilities
- CON: No battery backup
• CON: Requires surface for case to sit on in patient’s environment

Figure 4.7: Photograph of a sample case filled with electronics envisioned for the Concept A back-end server. A sealed plastic case with latches houses the needed server electronics, power supply, storage drives and wireless router.

Some design elements and benefits/detractors of the second, "road case" based design (Concept B) are:

• A rugged plastic/wood road case with a standard 19” rackmount inside is used to hang and connect all of the components
  – Managed network switch
  – 6-core virtualized server with 64GB+ of RAM
  – Internal storage array
  – Portable storage array
– Rackmount console
– UPS battery backup & power management
– Wireless Router

• The virtualized back-end server allows multiple operating environments to be running concurrently – maximizing design space potential

• Managed switch allows for virtualized independent networks in the case – segregating traffic to reduce network congestion

• Internal storage allows for additional data to be stored and backed up until safe transfer to the cloud is complete

• PRO: Increased back-end compute power for increased potential growth of back-end server applications

• PRO: Self-contained and on wheels; can sit in corner by itself

• PRO: Internal storage increases reliability of data chain

• PRO: Battery backup allows unit to be unplugged and moved between rooms; runtime ~1hr

• CON: Potentially larger footprint, can’t sit on a table/desk

• CON: Cost is substantially higher for full server build vs. NUC
Figure 4.8: Photograph of a sample case filled with electronics envisioned for the Concept B back-end server. A mobile road case houses a full blade server, rack-mounted terminal, storage bays, managed networking hardware and power management. Wireless router sits inside the case with external antennas. Used with permission from Gator Cases, Inc.
4.2.3 Future Work

ESPA is a platform that is targeted for neuroprosthetic use, however, this mobile processing system has the potential for applications in other fields. For example, in a retinal prosthetic system, a camera collecting millions of pixels of color data may need to run a vision algorithm to generate output signals for potentially tens of thousands of retinal stimulators. The use case of this type of system would require the patient to be mobile, and the combination of horsepower and configuration of the ESPA platform might be ideal for such a system.[3]

Another use case is be using ESPA as a platform for managing all medical sensory and environmental data for a patient, in addition to acting as a prosthetic system. In our laboratories, we have adopted a phrase; a Mobile Medical Monitoring System (M³) as short hand notation for such a broader, big data medical information computational platform. In this example, heart-rate and respiratory data could be collected and appropriately timestamped with the patient’s neural event data, allowing for real-time alerts to clinicians via a network connection, as well as possibly informing the neuroprosthetic output dynamically.
Presented in this text is a preliminary end-to-end Embedded System for Prosthetic Application (ESPA) solution. In order to make this product viable for clinical use, many upgrades and advancements will need to be made to the system, including

- Front-end UI interface for data observation
- Front-end UI interface for Parameter upload
- Back-end server hardware development
- Back-end server software development
- Improved decoding and implementation of Kalman Filter
- Improved wireless antenna and receiver design and manufacture
- New form-factor “motherboard” implementation and manufacture
As the final pieces of ESPA come together, the project has created a viable, mobile, battery-operated neural-prosthetic computer. Integrating neural, bio-sensory, environmental, and adaptive data learning into a single data-processor device that can be worn by a patient has the potential to catapult this field forward by providing better tools to researchers and clinicians working in the neural-prosthetic space. Importantly, the ESPA is upward scalable in that its compute power is capable of meeting the demands of future neural data collection systems; with up to 1000 channels of broadband data on the horizon, ESPA is ready for the challenge.
References


Appendices
Appendix A

Alternate FPGA Code Configurations for ESPA

A.1 Input Glue Logic and Signal Preparation

Other versions of `wireless_packet_to_model` have been created to accommodate different neural input configurations, including data simulators as outlined in Section 4.1.2, as well as multiple neural recording device inputs. For example, the system referenced in Figure 3.1 uses two 100 channel neural wireless recording devices with corresponding receivers, which not only need a pathway into the processing model, but need to synchronized between data samples to allow all samples from both devices (which are being received asynchronously) to enter the System Generator model in a synchronous fashion.

A.2 LFP Feature Extraction

An LFP Feature Extractor was designed for the System Generator model, with the following properties:
Figure A.1: Power bands calculated in LFP Feature Extractor

Figure A.2: Sample data processed by the LFP Feature Extractor
A.3 Memory Controller

The custom memory controller designed to manipulate the time domain shifting memory in ESPA is described with the following two figures.

Figure A.3: Memory Subsystem found in Speedy Calculator
Figure A.4: Memory Controller found in Memory Subsystem
Appendix B

Tables for FPGA Configuration

B.1 CAR Loader State Machine

<table>
<thead>
<tr>
<th>Meaning</th>
<th>Current X=0 Y=0</th>
<th>X=0 Y=1</th>
<th>X=1 Y=0</th>
<th>X=1 Y=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Update</td>
<td>0</td>
<td>0, 00</td>
<td>0, 10</td>
<td>1, 10</td>
</tr>
<tr>
<td>Counter Running</td>
<td>1</td>
<td>1, 01</td>
<td>0, 11</td>
<td>1, 01</td>
</tr>
</tbody>
</table>

Table B.1: CAR Loader State Machine. X = Valid Opcode, Y = Counter Complete. Output = Reset, Enable

B.2 30k Filter State Machine

<table>
<thead>
<tr>
<th>Meaning</th>
<th>Current X=0 Y=0</th>
<th>X=0 Y=1</th>
<th>X=1 Y=0</th>
<th>X=1 Y=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Update</td>
<td>0</td>
<td>0, 100000 (32)</td>
<td>0, 100000 (32)</td>
<td>1, 100000 (32)</td>
</tr>
<tr>
<td>Load OP1</td>
<td>1</td>
<td>2, 101000 (40)</td>
<td>2, 101000 (40)</td>
<td>2, 101000 (40)</td>
</tr>
<tr>
<td>Load all coef</td>
<td>2</td>
<td>2, 010010 (18)</td>
<td>3, 010011 (19)</td>
<td>2, 010010 (18)</td>
</tr>
<tr>
<td>Push Config</td>
<td>3</td>
<td>0, 100100 (36)</td>
<td>0, 100100 (36)</td>
<td>0, 100100 (36)</td>
</tr>
</tbody>
</table>

Table B.2: 30k Filter State Machine. X = Valid Opcode, Y = Counter Complete. Output = Reset, Enable, Load, cValid, rValid, rLast (decimal)
## B.3 Multiply State Machine

<table>
<thead>
<tr>
<th>Meaning</th>
<th>Current State</th>
<th>X=0 Y=0</th>
<th>X=0 Y=1</th>
<th>X=1 Y=0</th>
<th>X=1 Y=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Update</td>
<td>0</td>
<td>0, 0</td>
<td>0, 0</td>
<td>0, 0</td>
<td>1, 1</td>
</tr>
<tr>
<td>Valid Enable</td>
<td>1</td>
<td>0, 0</td>
<td>1, 1</td>
<td>2, 0</td>
<td>1, 1</td>
</tr>
<tr>
<td>Go Low</td>
<td>2</td>
<td>0, 0</td>
<td>0, 0</td>
<td>2, 0</td>
<td>2, 0</td>
</tr>
</tbody>
</table>

Table B.3: Multiply State Machine. X = Valid Opcode, Y = Counter Complete. Output = Reset, Enable

## B.4 Multiply State Machine 2

<table>
<thead>
<tr>
<th>Meaning</th>
<th>Current State</th>
<th>X=0 Y=0</th>
<th>X=0 Y=1</th>
<th>X=1 Y=0</th>
<th>X=1 Y=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Update</td>
<td>0</td>
<td>0, 0</td>
<td>0, 0</td>
<td>0, 0</td>
<td>1, 1</td>
</tr>
<tr>
<td>Running</td>
<td>1</td>
<td>0, 0</td>
<td>1, 1</td>
<td>0, 0</td>
<td>1, 1</td>
</tr>
</tbody>
</table>

Table B.4: Multiply State Machine. X = FIFO Valid, Y = Valid Counter. Output = Reset, Enable
### B.5 Expectation Accumulator

<table>
<thead>
<tr>
<th>Meaning</th>
<th>Current State</th>
<th>X=0</th>
<th>X=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Update</td>
<td>0</td>
<td>0, 000 (0)</td>
<td>1, 000 (0)</td>
</tr>
<tr>
<td>1-1</td>
<td>1</td>
<td>2, 100 (16)</td>
<td>2, 100 (16)</td>
</tr>
<tr>
<td>1-2</td>
<td>2</td>
<td>3, 101 (17)</td>
<td>3, 101 (17)</td>
</tr>
<tr>
<td>1-3</td>
<td>3</td>
<td>4, 102 (18)</td>
<td>4, 102 (18)</td>
</tr>
<tr>
<td>2-1</td>
<td>4</td>
<td>5, 110 (20)</td>
<td>5, 110 (20)</td>
</tr>
<tr>
<td>2-2</td>
<td>5</td>
<td>6, 111 (21)</td>
<td>6, 111 (21)</td>
</tr>
<tr>
<td>2-3</td>
<td>6</td>
<td>7, 112 (22)</td>
<td>7, 112 (22)</td>
</tr>
<tr>
<td>3-1</td>
<td>7</td>
<td>8, 120 (24)</td>
<td>8, 120 (24)</td>
</tr>
<tr>
<td>3-2</td>
<td>8</td>
<td>9, 121 (25)</td>
<td>9, 121 (25)</td>
</tr>
<tr>
<td>3-3</td>
<td>9</td>
<td>0, 122 (26)</td>
<td>0, 122 (26)</td>
</tr>
</tbody>
</table>

Table B.5: Expectation Accumulator State Machine. X = Start. Output = Next State, Enable(1b), Mux1(2b), Mux2(2b) (decimal)
## B.6 Test Input Vector CAR

<table>
<thead>
<tr>
<th>Sample</th>
<th>Parameter Bus</th>
<th>Parameter WE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:14999</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>15000</td>
<td>00A1</td>
<td>1</td>
</tr>
<tr>
<td>15001:44999</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>45000</td>
<td>00B0</td>
<td>1</td>
</tr>
<tr>
<td>45001</td>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>45002</td>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>45003</td>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>45004</td>
<td>0000</td>
<td>1</td>
</tr>
<tr>
<td>45005</td>
<td>0000</td>
<td>1</td>
</tr>
<tr>
<td>45006</td>
<td>0000</td>
<td>1</td>
</tr>
<tr>
<td>45007</td>
<td>0000</td>
<td>1</td>
</tr>
<tr>
<td>45008</td>
<td>0000</td>
<td>1</td>
</tr>
<tr>
<td>45009</td>
<td>0000</td>
<td>1</td>
</tr>
<tr>
<td>45010</td>
<td>0000</td>
<td>1</td>
</tr>
<tr>
<td>45011</td>
<td>0000</td>
<td>1</td>
</tr>
<tr>
<td>45012</td>
<td>0000</td>
<td>1</td>
</tr>
<tr>
<td>45013</td>
<td>0000</td>
<td>1</td>
</tr>
<tr>
<td>45014</td>
<td>0000</td>
<td>1</td>
</tr>
<tr>
<td>45015</td>
<td>0000</td>
<td>1</td>
</tr>
<tr>
<td>45016:74999</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>75000</td>
<td>00A0</td>
<td>1</td>
</tr>
<tr>
<td>75001:100000</td>
<td>0000</td>
<td>0</td>
</tr>
</tbody>
</table>

Table B.6: Input Vector CAR
Glossary

ALS  Amyotrophic Lateral Sclerosis. 3

AP  Action Potential. xx, 1, 4

ARM  Advanced RISC Machine. 1, 5, 6, 8, 9, 14–16, 21, 64

ASIC  Application Specific Integrated Circuit. 5

BCI  Brain Computer Interfaces. 2, 5

BRAM  Block Random Access Memory. xxvi, 16, 22, 60, 69

CAR  Common Average Reference. xviii, xxvii, xxix, 39, 74, 75, 79, 94, 97

DSP  Digital Signal Processor. xxii, 5, 8, 23–25, 31

ECoG  electrocorticography. 3, 4

EEG  electroencephalography. 2–4

EML  Embedded MATLAB Language. 9, 74, 75

ESPA  Embedded System for Prosthetic Application. xv, xvi, xviii, xx–xxii, xxviii, 1, 5, 6, 9, 10, 13, 15–18, 21–23, 28, 33, 34, 38, 40, 48–50, 52, 54, 58–60, 63, 77, 80, 81, 85–87, 90, 92

FIR  Finite Impulse Response. xvii, xxii–xxiv, 24, 25, 34, 36–42, 64, 67

FPGA  Field Programmable Gate Array. xvi, xviii, xix, xxi, xxii, 1, 5, 6, 8, 14–16, 21–26, 28, 59, 60, 63, 94

98
GPIO  General Purpose Input/Output. 16

HDL  Hardware Description Language. xxviii, 7, 9, 21, 28, 74–79

I/O  Input/Output. 6, 7, 22, 26

IIR  Infinite Impulse Response. xxiv, 40, 41

IP  Intellectual Property. xxi, xxiii, 9, 21–25, 27, 36, 37, 64, 73

LDA  Linear Discriminant Analysis. xvii, xx, xxvi, 1, 33, 45, 48–52, 56, 58, 59

LFP  Local Field Potential. xviii, xxiii, xxix, 1, 4, 32, 33, 38, 59, 90, 91

LUTs  Look Up Tables. 23

M³  Mobile Medical Monitoring System. xxix, 85, 86

MEA  Micro Electrode Array. xx, 3, 4

MUA  Multi-Unit Activity. 31, 32, 34, 48

NUA  Non-Unit Activity. xxiii, 31–33, 38, 48

NUC  Next Unit of Computer. 81

OS  Operating System. 14

RTOS  Real-Time Operating System. 6

SMP  Symmetric Multiprocessing. 16

SoC  System on Chip. xxi, 1, 6, 9, 14–16, 23, 28, 76

SUA  Single-Unit Activity. xxiii, 31–33, 38, 48

UDP  User Datagram Protocol. 9, 16, 17, 79