Experimentation Platforms for Neuromorphic Event-Based Multi-Chip Systems

Daniel Bernhard Fasnacht

Diss. ETH No. 23071
Experimentation Platforms for Neuromorphic Event-Based Multi-Chip Systems

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by

DANIEL BERNHARD FASNACHT
Dipl. Informatik-Ing. ETH
Master of Science ETH in Computer Science
born on the 16th of April 1980
citizen of Muntelier FR

accepted on the recommendation of

Prof. Dr. Tobias Delbruck (ETH Zurich), examiner
Prof. Dr. Giacomo Indiveri (University of Zurich), co-examiner
Prof. Dr. Richard Hahnloser (ETH Zurich), co-examiner
Prof. Dr. Rodney J. Douglas (ETH Zurich), co-examiner

2016
The front-cover artwork is based on a T-shirt design originally created by Mathis Richter for the “Telluride Neuromorphic Cognition Engineering Workshop 2012” and was adapted and reused with permission. The back-cover artwork is a section of a closeup photograph of one of the devices build as part of this thesis.

© 2016 by Daniel Bernhard Fasnacht, unless explicitly stated otherwise.

Use of material copyrighted by the main author is permitted under the license Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0), available in full at: https://creativecommons.org/licenses/by-nc-nd/4.0/legalcode

Partial citation of the material in this thesis, with adequate referencing of the original source according to established customs in the scientific community is certainly permitted as well.

DOI: https://dx.doi.org/10.3929/ethz-a-010785412
Abstract

“Neuromorphic Systems” are artificial information processing systems which mimic the architecture of biological neural systems. “Neuromorphic VLSI Systems” are such systems typically featuring analog VLSI chips as their basis. These chips can implement neurons and synapses, or they can be of the sensory kind, implementing artificial retinas or cochleas. Most of these systems use “Address-Event Representation” (AER), a way to send artificial spikes (action-potentials), in order to communicate between multiple chips on one board, or between multiple boards that contain such chips.

This thesis addresses the problems of AER interfacing and AER processing. Traditional AER has come to its limits concerning board-to-board interfacing. Novel solutions must be found. AER processing devices are very important too, because AER streams must usually be routed to the right destination, and be filtered or be translated completely (be “mapped”) to create complex connectivity patterns in neuromorphic systems.

To provide a solution for these issues, we present the AEXv4 AER interfacing platform and MMv2 AER mapper, which communicate using a novel serial AER interface. In combination we present a software framework developed for these systems. Together they provide a high-performance, modular AER interfacing and processing platform enabling many neuromorphic systems successfully built at INI and other research labs.

The thesis also investigates how to embed AER systems to make them suitable for mobile/robotic applications. The result is that the iCub robot can be equipped with a completely neuromorphic visual pathway, built as part of the eMorph research project, partially reusing the AEXv4 system design.

The last problem addressed in this thesis is, based on all lessons learnt when solving the aforementioned problems, to create an AER interfacing and processing system, which is much more extensible than the systems presented so far in this thesis. It aims also at being able to scale to medium-scale heterogeneous multi-chip systems. For this the AEXL project is presented and shows very promising results.

The thesis will conclude by showing its impact on the field by presenting a number of neuromorphic experimental setups based on this work and publications resulting from them. Finally, the key design methodologies and strategies used during this thesis are presented and their influence on the work of this thesis and its success is explained.
Zusammenfassung

“Neuromorphe Systeme” sind künstlich geschaffene Informationsverarbeitungssysteme, die Architekturen von biologischen neuronalen Systemen nachahmen. “Neuromorphe VLSI Systeme” sind Systeme dieser Art, die auf analogen VLSI Chips basieren. Diese Chips können Neuronen und Synapsen in ihren Schaltungen implementieren, oder sie können auch sensorischer Art sein, z.B. als künstliche Retina oder Cochlea. Die meisten dieser Systeme verwenden die “Address-Event Representation” (AER), eine Methode um künstliche “Spikes” (Aktionspotentiale) zu versenden, um zwischen verschiedenen Chips auf einer Platine zu kommunizieren, oder auch zwischen mehreren Platinen die solche Chips enthalten.


Die letzte Aufgabe, welche in dieser Dissertation angegangen wird, ist es, basierend auf den bis hierhin vorgestellten Systemen und den bei der Entwicklung dieser gewonnenen Erkenntnisse, ein AER-Experimentierplattform zu entwickeln, die einfacher erweiterbar ist als die bisher
vorgestellten und zudem die Skalierfähigkeit bieten soll, um mittlere bis
grosse neuromorphe Systeme zu bauen, die viele Chips vereinen. Dabei
sollen auch heterogene multi-Chip Systeme unterstützt werden. Dazu
wurde das AEXL Projekt geschaffen, mit dessen von vielen Forschern
sehr vielversprechende Resultate produziert werden.

Zum Abschluss dieser Arbeit werden wir deren Einfluss auf das
Forschungsgebiet anhand von diversen Beispielen von Experimenten,
die auf dieser Arbeit basieren, sowie den daraus resultierenden Pub-
likationen, aufzeigen. Zuletzt werden wir die entscheidenden Entwick-
lungsmethoden und -strategien vorstellen, die im Rahmen dieser Arbeit
angewandt wurden und wesentlich zu deren Erfolg beigetragen haben.
Acknowledgments

My first thanks go to my supervisor and examiners, Prof. Tobias Delbruck, Prof. Giacomo Indiveri, Prof. Richard Hahnloser and Prof. Rodney J. Douglas.

Tobias Delbruck introduced me to this field of research already during my undergraduate studies in computer science, where he supervised the semester project\(^1\) in my minor subject *Neuroinformatics* and invited me to participate in the *Telluride Neuromorphic Engineering Workshop* on numerous accounts, besides being a great poker teacher.

Probably not by chance it was in Telluride, where Rodney J. Douglas and Giacomo Indiveri motivated me to write my master’s thesis\(^2\) at the *Institute of Neuroinformatics* (INI), which they then also supervised together with Adrian Whatley.

For my doctorate at the Department of Physics of ETH Zurich I was then able to join Giacomo Indiveri’s *Neuromorphic Cognitive Systems* (NCS) group at INI. Giacomo Indiveri was always a great mentor during my studies in his field of research. My greatest thanks also go to all the members of the NCS group, current and former, too numerous to mention them all.\(^3\)

They all were key to my studies by using and extending the systems developed as part of this thesis and by providing feedback from their experiments which was key to the success of this work. Special thanks go to former members Emre Neftci, now Assistant Professor at UC Irvine, Sadique Sheik, now Postdoc at UC San Diego and Fabio Stefanini, now Postdoc at Columbia University, who were very influential to the design decisions taken in the early stages of the work on this thesis.

These thanks must be expanded to all my colleagues at INI including staff, with special mention of my most favourite and extremely helpful secretaries, Simone Schumacher and Kathrin Aguilar, as well as my friend Claudio Luck.

My thanks also go to my foreign collaborators on the *eMorph* project, especially to Michael Hofstätter at AIT Vienna and Dr. Chiara Bartolozzi at IIT Genoa.

\(^1\) *Dichromatic spectral measurement circuit in vanilla CMOS*: [Fasnacht 07b]  
\(^2\) *A stand-alone AER communication system*: [Fasnacht 07a]  
\(^3\) Please refer to: http://ncs.ethz.ch/people
I also want to thank Prof. Hubert Käslin and Dr. Norbert Felber of the Integrated Systems Laboratory of ETHZ, both of whom were not only great teachers, but also great consultants for all my electrical engineering questions during my entire study time at ETH.

Dr. Daniel Kiper from INI also deserves my thanks for always being very helpful with my questions regarding the biological issues, especially with respect to the vision system.

My thanks also go to my friends Dr. Cyril Flaig and Dr. Alain Lehmann, from whom I could take many aspects of the \LaTeX{} template I adapted for this thesis and also to Christine Baumann, for proof-reading key parts of this work.

Finally, my greatest thanks go toward my friends and especially my family, Therese, Beat and Christian Fasnacht who not only enabled me to follow my undergraduate and doctoral studies at ETH Zurich, but also supported me in any way they could during that experience.

This work was supported under the EU FP7 framework: ICT-STREP project eMorph, (ICT-231467-eMorph), on Embodied Intelligence.
# Table of Contents

List of Figures  
List of Tables  
List of Source Code Listings  

Conventions, Abbreviations and Acronyms  

## 0.1 Conventions  
### 0.1.1 Unit Prefixes  
### 0.1.2 VHDL Signal Naming Convention  

## 0.2 Abbreviations, Acronyms and Terms  

## 1 Introduction  

### 1.1 Neuromorphic Systems  
#### 1.1.1 Neurons and Synapses in Analog VLSI Circuits  

### 1.2 Address-Event Representation  
#### 1.2.1 The Neurobiological Approach  
#### 1.2.2 The AER Approach  
#### 1.2.3 Timestamped AEs, Monitoring and Sequencing  

### 1.3 Problems Addressed in this Thesis  
#### 1.3.1 AER Interfacing  
#### 1.3.2 AER Processing  
#### 1.3.3 Embedded, Mobile and Robotic Application  
#### 1.3.4 Scalable Heterogeneous Multi-Chip AER Systems  

### 1.4 Thesis Goals  

### 1.5 Key Design Principles  
#### 1.5.1 Maximize Flexibility Where Possible  
#### 1.5.2 Use Off-The-Shelf Components Where Feasible  

### 1.6 Thesis Outline  

### 1.7 Primary Contributions  
#### 1.7.1 AEXv4 Generic AER Interfacing Platform  
#### 1.7.2 3 GHz Serial AER Board-to-Board Interface  
#### 1.7.3 Generic VHDL framework for AER Processing  
#### 1.7.4 MMv2 AER Mapper System  
#### 1.7.5 Embedded AER System for the iCub Robot  
#### 1.7.6 A Scalable Multi-Chip AER Platform
# Table of Contents

## 2 Related Work – Strengths and Weaknesses, AER Interfacing

2.1 Overview .............................................. 45
    2.1.1 Typical Requirements of a Neuromorphic Chip .... 46

2.2 Early Neuromorphic Platforms – SCX Project .......... 50
    2.2.1 Silicon Cortex Architecture .................... 50
    2.2.2 AER mapping in SCX .......................... 52
    2.2.3 SCX Scalability ................................ 52
    2.2.4 Conclusion .................................. 52

2.3 Very Influential Related Work .......................... 53
    2.3.1 PCI-AER ........................................ 53
    2.3.2 CAVIAR project – USBAERmini2 board .......... 55
    2.3.3 AMDA ........................................ 57
    2.3.4 Other Approaches ................................ 62

2.4 Inspiration and Lessons from Related Work ............ 63
    2.4.1 PCI-AER – Strengths and Weaknesses ............ 64
    2.4.2 USBAERmini2 – Strengths and Weaknesses ....... 65
    2.4.3 AMDA – Strengths and Weaknesses ............... 67

2.5 Parallel AER Interfaces .................................. 68
    2.5.1 Interface Connectors and Cables ................ 70
    2.5.2 Interface Protocols ................................ 70
    2.5.3 Conclusion on Parallel AER ..................... 72

2.6 Parallel versus Serial AER Interfaces .................. 73
    2.6.1 Problems with the Parallel AER Interfaces ..... 73
    2.6.2 Trend towards Serial Differential Signaling .... 74
    2.6.3 Serial AER Conclusion .......................... 75

2.7 Conclusion ............................................ 76

## 3 Design Requirements and Prototypes of the AEX Platform

3.1 Design Requirements for the AEX System ................ 77
    3.1.1 Functionality and Extensibility .................. 78
    3.1.2 Interfacing Capabilities ........................ 78
    3.1.3 Mechanical and Cost Constraints ................ 79
    3.1.4 Design Block Diagram .......................... 80

3.2 AEX Prototypes Version 1 & 2 .......................... 82
    3.2.1 AEXv1 ........................................ 82
    3.2.2 AEXv2 ........................................ 83

3.3 AEXv3 – First Production Version ....................... 85
    3.3.1 FPGA Clocking Optimization ...................... 85
    3.3.2 SerDes Termination Optimization ................. 85
    3.3.3 First Production AEX Version .................... 86

3.4 Conclusion ............................................. 88
# Table of Contents

4 AEXv4 – Hardware Implementation Details 89

4.1 AEXv4 Platform Overview ........................................ 90
  4.1.1 Changes compared to AEXv3 .................................. 90
  4.1.2 Board-Level Block Diagram ................................. 91
  4.1.3 AEXv4 PCB and Design for Assembly ....................... 93
  4.1.4 Components and Interfaces ................................. 94
  4.1.5 AEXv4 and AMDA Platform Joined ......................... 96
  4.1.6 Interfaces Overview ....................................... 97

4.2 Parallel AER Interfaces Implementation Details ............. 99
  4.2.1 Parallel AER Interface FPGA Implementation ........... 99

4.3 3 GHz Serial AER Interface Implementation Details .......... 103
  4.3.1 LVDS ..................................................... 103
  4.3.2 Texas Instruments TLK2501 & TLK3101 ...................... 105
  4.3.3 Cables & Wiring: Serial ATA .............................. 108
  4.3.4 Flow Control .......................................... 108
  4.3.5 Serial AER Pinout ...................................... 110
  4.3.6 Impedance Matched PCB Layout ............................ 111
  4.3.7 FPGA: Serial AER Interface VHDL ......................... 113

4.4 USB 2.0 Interface Implementation Details ................... 117
  4.4.1 USB Interface .......................................... 117
  4.4.2 FPGA: FX2 USB Interface VHDL .......................... 120
  4.4.3 FPGA: Monitor & Sequencer .............................. 121

4.5 Performance Measurements and Conclusion .................. 124
  4.5.1 Bandwidth and Latency of the Serial AER ............... 126
  4.5.2 Bandwidth and Latency of the Parallel AER ............ 128
  4.5.3 FPGA – FX2 USB Chip Interface Measurements ........... 130
  4.5.4 AEXv4 Interface Spec Sheet ............................ 131
  4.5.5 Conclusion ............................................ 132

5 Generic FPGA Codebase for AER Interfacing & Processing 133

5.1 Core AER VHDL Entities ..................................... 133
  5.1.1 RR Interface Specification ............................... 134
  5.1.2 Splitter & Filter ..................................... 134
  5.1.3 Merger ................................................. 136
  5.1.4 LongPath – spanning across the FPGA ................... 137

5.2 Generic AER Routing Fabric of the AEX .................... 139

5.3 Configuring the Fabric ....................................... 140

5.4 Conclusion .................................................... 146

6 USB Interface Software Implementation 147

6.1 USB Interface Chip Firmware ................................. 148
  6.1.1 USB Device Configuration – lsusb ....................... 148
Table of Contents

6.1.2 FX2 Firmware Main Loop Tasks 150
6.1.3 FX2 FIFO Flags Setup 151
6.2 Linux Kernel Driver 153
6.2.1 Kernel Driver Architecture 153
6.3 PC User-Space Code 157
6.4 Monitor Control – Code Walk-Through 157
6.5 Testing and Benchmark Measurements 159
6.6 Conclusion 161

7 Creating Complex and Reconfigurable AER Connectivity 163
7.1 Overview & Related Work 165
7.1.1 Mapping Operations & Mapper Types 165
7.1.2 Related Work – Replacing PCI-AER 167
7.2 Mapper System Prototype I – ColEx 167
7.3 Mapper System Prototype II – ExCol 170
7.4 MMv2 Mapper System 172
7.4.1 MMv2 PCB 173
7.4.2 Mapper Components 173
7.5 Implementation Details 177
7.5.1 PC Hardware 177
7.5.2 Mapping Table Setup & Management 177
7.5.3 Mapper PCI Board 179
7.5.4 Custom PCI Implementation 180
7.6 Probabilistic Mapping Mode 182
7.7 Performance 183
7.7.1 Performance Measurements 184
7.8 Conclusions 187
7.8.1 Limitations 187
7.8.2 Summary 187
7.8.3 Typical Performance in Experiments 188

8 AEXS – Towards Embedded and Robotic Applications 189
8.1 Embedding the AEX 190
8.1.1 FPGA 190
8.1.2 BGA board design & soldering 191
8.2 AEXS Interfaces 194
8.2.1 Serial AER 194
8.2.2 Parallel AER 195
8.2.3 USB Interface 196
8.3 Results & Conclusion 196
8.3.1 Results 196
8.3.2 Conclusion 197
# 9 eMorph: a Neuromorphic Visual Pathway for the iCub Robot

9.1 iCub Robot

9.2 eMorph – Overview and Hardware Architecture
- 9.2.1 The Eyes: DVS128
- 9.2.2 Optic Nerve, Chiasm and Tract: AEXS
- 9.2.3 LGN: General Address-Event Processor
- 9.2.4 Optic Radiations: AEXS FPGA & USB
- 9.2.5 V1 and Higher Visual Processing Areas
- 9.2.6 Overall Architecture Overview

9.3 Architecture Details
- 9.3.1 AEXS to iHead

9.4 GAEP
- 9.5 iHead FPGA
- 9.5.1 Special FPGA Configuration for Debugging

9.6 FPGA – GAEP Interface

9.7 Results & Conclusion
- 9.7.1 GAEPiface Tuning and Performance
- 9.7.2 FX2 USB Interface Tuning
- 9.7.3 iCub with iHead Neuromorphic Visual Pathway
- 9.7.4 Publications & Conclusion

# 10 Towards a Modular & Scalable Neuromorphic Platform

10.1 Goals of the AEXL Project
- 10.1.1 Code-Reuse & Backwards Compatibility
- 10.1.2 Improved Extensibility by Increased Modularity
- 10.1.3 Improved Scalability

10.2 Choosing the Foundation of the AEXL Project
- 10.2.1 FPGA of choice: Xilinx Spartan 6
- 10.2.2 Raggedstone 2 FPGA baseboard
- 10.2.3 Raggedstone 2 Key Components & Interfaces
- 10.2.4 Raggedstone 2: Expansion Ports

10.3 Initial AEXL Extension Boards
- 10.3.1 AEXL/FX2
- 10.3.2 AEXL/PAER
- 10.3.3 AEXL/SAER
- 10.3.4 AEXL/IFMEM
- 10.3.5 AEXL/DVS128

10.4 First Expansion Board Fabrication Panel

10.5 Second Expansion Board Fabrication Panel

10.6 Example AEXL Setups with Extension Boards

10.7 Scaling to an AEXL cluster
- 10.7.1 Eight Raggedstone 2 AEXL Cluster Example
Table of Contents

10.8 Conclusion .......................... 236
  10.8.1 Status .................................. 236
  10.8.2 Limitations .......................... 237
  10.8.3 AEXL became a Community Project ........................ 237

11 Impact & Conclusion ......................... 239
  11.1 Impact ............................................. 239
    11.1.1 Single AEXv4 and AMDA .................. 240
    11.1.2 Single AEXv4 with Directly Attached Chip .... 242
    11.1.3 Multi–AEXv4 and MMv2 Mapper Setups ........ 243
    11.1.4 eMorph iHead Systems ...................... 246
    11.1.5 AEXS & USB Design Reuse .................... 248
    11.1.6 AEXL .............................................. 249
    11.1.7 Publications Enabled by this Thesis ........ 252
  11.2 Future Work ........................................... 253
    11.2.1 MMv2–AEXv4–AMDA ....................... 253
    11.2.2 AEXL Project .................................... 253
  11.3 Conclusion – Design Methodologies & Strategies ........ 255
    11.3.1 Version Control Systems ...................... 255
    11.3.2 VHDL Verification & Testing for FPGAs .... 256
    11.3.3 Incremental HDL Design, Continuous Integration 258
    11.3.4 Scripted/Automated Building and Testing .... 259
    11.3.5 Release Often ................................... 259
    11.3.6 Release Management, Archive Everything .... 260
    11.3.7 Extreme Programming Concepts in HW Design . 260

Bibliography ........................................ 263
List of Figures

1.1 Optic nerve compared to a telephone trunk cable . . . . 29
1.2 AER – Address-Event Representation . . . . . . . . . . 31

2.1 Generic AER board from CAVIAR . . . . . . . . . . . . . . 47
2.2 DVS128 PARALLEL AER board . . . . . . . . . . . . . . 49
2.3 SCX-1 board . . . . . . . . . . . . . . . . . . . . . . . . . 51
2.4 PCI-AER board . . . . . . . . . . . . . . . . . . . . . . . . 53
2.5 PCI-AER block diagram . . . . . . . . . . . . . . . . . . . 54
2.6 USBAERmini2 board, top and bottom view . . . . . . . . 56
2.7 AMDA board, without a chip daughter-board . . . . . . . 59
2.8 AMDA board providing bias voltages neuromorphic chips 61
2.9 64-Channel Binaural Silicon Cochlea . . . . . . . . . . . . 62
2.10 CAVIAR type 40-pin parallel AER connector . . . . . . . 69
2.11 P2P-Type AER hand-shake . . . . . . . . . . . . . . . . . 71
2.12 SCX-Type AER hand-shake . . . . . . . . . . . . . . . . . 71

3.1 AEX block diagram according to the design requirements 81
3.2 AEX version 1 prototype . . . . . . . . . . . . . . . . . . . 83
3.3 AEX version 2 prototype . . . . . . . . . . . . . . . . . . . 84
3.4 AEX version 3, first production-grade AEX . . . . . . . . . 87

4.1 AEXv4 block diagram with FPGA implementation details 92
4.2 Assembled AEXv4 PCB, top view . . . . . . . . . . . . . . 93
4.3 Assembled AEXv4 PCB, bottom view . . . . . . . . . . . . 94
4.4 AEXv4 and AMDA board combined . . . . . . . . . . . . . 96
4.5 AEXv4 and AMDA board combined, overlay . . . . . . . . 97
4.6 AEX FPGA block diagram . . . . . . . . . . . . . . . . . . 98
4.7 DC-coupled LVDS . . . . . . . . . . . . . . . . . . . . . . . 104
4.8 AC-coupled LVDS . . . . . . . . . . . . . . . . . . . . . . . 105
4.9 TI SerDes block diagram and data-path . . . . . . . . . . 107
4.10 Serial ATA cable and connectors . . . . . . . . . . . . . 109
4.11 Serial AER interface PCB Layout . . . . . . . . . . . . . 111
4.12 Cypress FX2 – USB interface chip block diagram . . . . . 118
4.13 Cypress FX2 – independence of data-path & CPU . . . . 118
4.14 HDL entity: Fx2MonSeqRR . . . . . . . . . . . . . . . . . 121
4.15 Latency measurements setup . . . . . . . . . . . . . . . . 124
4.16 FPGA routing fabric setup for latency measurements . . . 125
4.17 Serial AER interface, single event . . . . . . . . . . . . . 127
List of Figures

4.18 Serial AER interface, latency measurement ............... 127
4.19 Parallel AER interface, single event .......................... 129
4.20 Parallel AER interface, latency measurement ............... 130

5.1 AEX generic AER routing fabric ................................. 139

6.1 aerfx2.c: zero-allocation data flow architecture ............. 154
6.2 aerfx2.c: multiple AEX device support ...................... 155
6.3 aerfx2.c: code execution context ............................... 156
6.4 aerfx2.c: locking domains ..................................... 156
6.5 aerfx2.c: loopback benchmark test mode ..................... 159

7.1 ColEx mapper prototype ......................................... 168
7.2 Enterpoint Raggedstone I PCI FPGA board ................... 170
7.3 ExCol mapper prototype assembly, front- & backside ......... 171
7.4 3D rendering of the MMv2 PCB ................................ 173
7.5 MMv2 mapper system ............................................ 174
7.6 HF-section of the MMv2 PCB .................................... 175
7.7 Mapper hardware block diagram ................................ 176
7.8 FPGA block diagram ............................................. 179
7.9 PCI Configuration-Read transaction ............................ 182
7.10 Probability mapping word format .............................. 182
7.11 Input event to first output event latency scope measurement 184
7.12 Bandwidth scope measurement ................................ 186

8.1 AEXS (PCB version 1 & version 2) ............................. 190
8.2 Traditional and BGA packages in comparison .................. 191
8.3 AEXSv2 rendered layout detail .................................. 192
8.4 Zoom to the BGA footprint, view from above ................ 193
8.5 Zoom to the BGA footprint, view from within PCB ........... 193
8.6 AEXS USB and one parallel AER interface connected ....... 195

9.1 eMorph: neuromorphic visual pathway for the iCub .......... 201
9.2 Anatomical optic wiring diagram ............................... 201
9.3 Neuromorphic optical pathway, feed-forward diagram ....... 203
9.4 eMorph – detailed block diagram ............................... 204
9.5 iHead PCB – front view .......................................... 205
9.6 iHead PCB – back view .......................................... 206
9.7 iHead PCB – front view with description overlay .......... 206
9.8 iHead: FPGA internal block diagram ........................... 208
9.9 iHead: FPGA DVS/GAEP debugging config .................. 210
9.10 iHead: test setup at AIT Vienna ............................... 213
### List of Figures

9.11 iHead: FPGA – GAEP interface tuning  .................................. 214
9.12 iHead: FPGA – FX2 USB Chip Interface Tuning  ........ 215
9.13 iHead PCB completely assembled with incl. DVS eyes  .... 216
9.14 iCub on iKart with all eMorph expansions added  .......... 217

10.1 Raggedstone 2 board  ............................................................. 223
10.2 Raggedstone 2: key components & interfaces  ................. 224
10.3 Raggedstone 2: expansion ports  ......................................... 225
10.4 AEXL/FX2 expansion board  ................................................. 226
10.5 AEXL/PAER expansion board  .............................................. 228
10.6 AEXL/SAER expansion board  .............................................. 228
10.7 AEXL/IFMEM expansion board  ............................................ 229
10.8 DVS128 on AEXL board combination  ................................. 230
10.9 AEXL expansion boards fab-panel v1  ................................. 231
10.10 AEXL expansion boards fab-panel v2  ............................... 232
10.11 AEXL with expansion boards mounted  ............................... 233
10.12 AEXL with multiple expansion boards  .............................. 233
10.13 Single AEXL cluster module (AEXL/CM)  ........................... 235
10.14 Four AEXL/CM boards cascaded  ....................................... 236

11.1 AEXv4 and AMDA board combined  ..................................... 240
11.2 PC, AEXv4 and IFMEM on his own PCB  ............................... 241
11.3 MMv2 and 4x AEXv4+AMDA in an experiment  ...................... 244
11.4 Photograph of a complex experimental setup  ....................... 244
11.5 Diagram of the setup from [Neftci 13]  ............................... 245
11.6 iCub robot on “Spektrum der Wissenschaften” ..................... 247
11.7 Example of AEXS hardware design and code resuse ............. 248
11.8 AEXL with a ROLLS chip mounted  ..................................... 249
11.9 AEXL with a CXQUAD CLOWN expansion board .................... 250
11.10 AEXL based vision experimentation setup  ......................... 251
# List of Tables

<table>
<thead>
<tr>
<th>Section</th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Typical features on a neuromorphic chip</td>
<td>46</td>
</tr>
<tr>
<td>2.2</td>
<td>AMDA board key features provided to support a chip</td>
<td>58</td>
</tr>
<tr>
<td>2.3</td>
<td>Influential factors of the PCI-AER system</td>
<td>64</td>
</tr>
<tr>
<td>2.4</td>
<td>Influential factors of the USBAERmini2 system</td>
<td>66</td>
</tr>
<tr>
<td>2.5</td>
<td>Influential factors of the AMDA system</td>
<td>67</td>
</tr>
<tr>
<td>2.6</td>
<td>Parallel AER Interface Pinout “Rome-Type”</td>
<td>68</td>
</tr>
<tr>
<td>2.7</td>
<td>Parallel AER Interface Pinout “CAVIAR-Type”</td>
<td>69</td>
</tr>
<tr>
<td>3.1</td>
<td>AEX requirements: functionality and extensibility</td>
<td>78</td>
</tr>
<tr>
<td>3.2</td>
<td>AEX requirements: Parallel AER</td>
<td>78</td>
</tr>
<tr>
<td>3.3</td>
<td>AEX requirements: board-to-board AER</td>
<td>79</td>
</tr>
<tr>
<td>3.4</td>
<td>AEX requirements: interfacing to computers</td>
<td>79</td>
</tr>
<tr>
<td>3.5</td>
<td>AEX requirements: mechanical and cost</td>
<td>80</td>
</tr>
<tr>
<td>4.1</td>
<td>AEXv4 – key components / chips</td>
<td>95</td>
</tr>
<tr>
<td>4.2</td>
<td>AEXv4 – interfaces and connectors</td>
<td>95</td>
</tr>
<tr>
<td>4.3</td>
<td>Pinout of the “3 GHz Serial AER Interface”</td>
<td>110</td>
</tr>
<tr>
<td>4.4</td>
<td>AEXv4 performance specification sheet</td>
<td>131</td>
</tr>
<tr>
<td>6.1</td>
<td>aerfx2.c kernel driver loopback benchmark</td>
<td>160</td>
</tr>
<tr>
<td>7.1</td>
<td>Physical memory-map of the PC used by the MMv2</td>
<td>178</td>
</tr>
<tr>
<td>7.2</td>
<td>MMv2 PCI implementation specification sheet</td>
<td>181</td>
</tr>
<tr>
<td>8.1</td>
<td>AEXS / USBAERmini2 comparison</td>
<td>196</td>
</tr>
</tbody>
</table>
List of Tables
# List of Source Code Listings

4.1 VHDL entity declaration of `SimplePAEROutputRR` . . . . . . . . . . . . . . . . 100
4.2 VHDL entity declaration of `SimplePAERInputRRv2` . . . . . . . . . . . . . . 101
4.3 VHDL entity declaration of `TLKiface` . . . . . . . . . . . . . . . . . . . . . . 114
4.4 VHDL entity declaration of `SAER3GHzRR` . . . . . . . . . . . . . . . . . . . . 116
4.5 VHDL entity declaration of `fx2if2` . . . . . . . . . . . . . . . . . . . . . . . . 120
4.6 VHDL entity declaration of `Fx2MonSeqRR` . . . . . . . . . . . . . . . . . . . . 122
4.7 Shell script to inject single event . . . . . . . . . . . . . . . . . . . . . . . . . . 128
5.1 VHDL entity declaration of `FilterSplitter3RR` . . . . . . . . . . . . . . . . . . . 135
5.2 VHDL entity declaration of `Merger3RR` . . . . . . . . . . . . . . . . . . . . . . . 136
5.3 VHDL entity declaration of `LongPathRR` . . . . . . . . . . . . . . . . . . . . . . 138
5.4 VHDL entity declaration of `CompatibilityFabric` . . . . . . . . . . . . . . . . . . 140
5.5 VHDL entity `AEXconfig`, version `SpinTest0` . . . . . . . . . . . . . . . . . . . . 141
6.1 USB device configuration as displayed by `lsusb` . . . . . . . . . . . . . . . . . . 148
6.2 FX2 firmware: main function / main loop . . . . . . . . . . . . . . . . . . . . . . . 150
6.3 FX2 firmware: flag pin configuration . . . . . . . . . . . . . . . . . . . . . . . . . 152
6.4 FX2 firmware: EP2 programmable flag . . . . . . . . . . . . . . . . . . . . . . . . . 152
6.5 FX2 firmware: EP6 programmable flag . . . . . . . . . . . . . . . . . . . . . . . . . 152
6.6 Kernel module aerfx2.c: URB preallocation count . . . . . . . . . . . . . . . . . . . 155
6.7 xio-bin-bin usage information . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 157
6.8 xio-bin-bin monitoring-sequencing example . . . . . . . . . . . . . . . . . . . . . . 157
6.9 kernel module aerfx2.c: EP1 OUT monitor control . . . . . . . . . . . . . . . . . . . 158
6.10 FX2 firmware: EP1 OUT command processing . . . . . . . . . . . . . . . . . . . . . 158
9.1 VHDL entity declaration of `GAEFiface` . . . . . . . . . . . . . . . . . . . . . . . . 211
List of Source Code Listings
Conventions, Abbreviations and Acronyms

0.1 Conventions

In order to avoid any uncertainty whether e.g. “Kilo” means 1000 or 1024, binary prefixes as specified by the IEC (International Electrotechnical Commission) are used throughout this document.

Also all VHDL code listings use the VHDL signal naming conventions recommended by the ETH Zurich DZ (Microelectronics Design Center). These conventions are described below.

0.1.1 Unit Prefixes

IEC binary prefixes are used throughout the text, i.e. \( \text{Gi} \) denotes \( 1024^3 \) whereas \( \text{G} \) denotes \( 1000^3 \). \( \text{G} \) is called Giga, while \( \text{Gi} \) is called Gibi. Similarly 1000 Bytes are \( \text{1kB} \) while 1024 Bytes are \( \text{1KiB} \). While \( \text{kB} \) is called Kilobyte, \( \text{KiB} \) is called Kibibyte and to get “cosmological”, \( \text{Y} \) is called Yotta which equals \( 10^{24} = 1000^8 \), while \( \text{Yi} \) is called Yobi equaling \( 2^{80} = 1024^8 \).

For a complete list of IEC binary prefixes, refer to: https://en.wikipedia.org/wiki/Binary_prefix

0.1.2 VHDL Signal Naming Convention

VHDL signal naming is according to recommendations of the ETH DZ (Microelectronics Design Center, https://www.dz.ee.ethz.ch/). The following list of rules are adapted from their internal wiki pages:
List of Source Code Listings

- Start with an upper-case letter.
- Have a suffix with syntax "x[CRSDTA][NP]?Z?B?[IO]?" ("[...]") denotes a choice, "?" means optional).
- The suffix part "[CRSDTA]" indicates the class of the signal:

<table>
<thead>
<tr>
<th>Class</th>
<th>Char</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock</td>
<td>C</td>
<td>ClkxC1</td>
<td>clock</td>
</tr>
<tr>
<td>reset</td>
<td>R</td>
<td>RstxRB</td>
<td>asynchronous reset</td>
</tr>
<tr>
<td>control/status</td>
<td>S</td>
<td>FullxS</td>
<td>control and status signals</td>
</tr>
<tr>
<td>data/address</td>
<td>D</td>
<td>RamAdrxD</td>
<td>data and address signals</td>
</tr>
<tr>
<td>test</td>
<td>T</td>
<td>ScanEnxT</td>
<td>test related signals</td>
</tr>
<tr>
<td>asynchronous</td>
<td>A</td>
<td>StrobexA</td>
<td>asynchronous signals</td>
</tr>
</tbody>
</table>

- The suffix part "[NP]?") indicates next and present state for a signal (e.g., StatexSN / StateSP, AddrCntxDN / AddrCntxDP).
- The suffix part "Z?" indicates three-state signals.
- The suffix part "B?" indicates active low signals.
- The suffix part "[IO]?" indicates input and output signals of an entity (e.g., CoeffxDI, FullxSO, ExtRamxDZIO).

This reflects the DZ naming recommendations as of a few years ago. The current (2015) recommendations differ only very slightly, they use “.” instead of “x” as the separator character.

For further reference and a great resource for both ASIC and FPGA design using VHDL, please refer to: [Kaeslin 08, Kaeslin 15].
### 0.2 Abbreviations, Acronyms and Terms

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 GHz SAER</td>
<td>Serial AER Interface, see sec. 4.3</td>
</tr>
<tr>
<td>ACK</td>
<td>Acknowledge Signal</td>
</tr>
<tr>
<td>ADC</td>
<td>Analog to Digital Converter</td>
</tr>
<tr>
<td>AE</td>
<td>one Address-Event, see sec. 1.2.2</td>
</tr>
<tr>
<td>AER</td>
<td>Address-Event-Representation, see sec. 1.2.2</td>
</tr>
<tr>
<td>AEX</td>
<td>AEX from “AMDA Extension”, see sec. 4.1</td>
</tr>
<tr>
<td>AEXL</td>
<td>AEXL from “AEX-large project”, see sec. 10</td>
</tr>
<tr>
<td>AEXv4</td>
<td>AEX version four, see sec. 4.1</td>
</tr>
<tr>
<td>AGND</td>
<td>Analog Ground Power-Rail</td>
</tr>
<tr>
<td>AMDA</td>
<td>AMDA board, see sec. 2.3.3</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>AVCC</td>
<td>Analog Supply (positive) Power-Rail</td>
</tr>
<tr>
<td>Address-Event</td>
<td>see sec. 1.2.2</td>
</tr>
<tr>
<td>Address-Event-Representation</td>
<td>see sec. 1.2.2</td>
</tr>
<tr>
<td>CAVIAR</td>
<td>CAVIAR research project, see sec. 2.3.2</td>
</tr>
<tr>
<td>CMOS</td>
<td>Complementary Metal-Oxide-Semiconductor</td>
</tr>
<tr>
<td>ColEx</td>
<td>an abandoned AER mapper project, see sec. 7.2</td>
</tr>
<tr>
<td>DAC</td>
<td>Digital to Analog Converter</td>
</tr>
<tr>
<td>DGND</td>
<td>Digital Ground Power-Rail</td>
</tr>
<tr>
<td>DVCC</td>
<td>Digital Supply (positive) Power-Rail</td>
</tr>
<tr>
<td>EM</td>
<td>Electro-Magnetic</td>
</tr>
<tr>
<td>EMC</td>
<td>Electro-Magnetic Compatibility</td>
</tr>
<tr>
<td>EMI</td>
<td>Electro-Magnetic Interference</td>
</tr>
<tr>
<td>ExCol</td>
<td>an AER mapper prototype, see sec. 7.3</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate-Array</td>
</tr>
<tr>
<td>FX2</td>
<td>Cypress FX2/FX2LP USB Interface Chip CY7C68013A</td>
</tr>
<tr>
<td>GND</td>
<td>Ground Power-Rail</td>
</tr>
<tr>
<td>GPIO</td>
<td>General Purpose (digital) Input/Output</td>
</tr>
<tr>
<td><strong>HW</strong></td>
<td>Hardware</td>
</tr>
<tr>
<td><strong>I²C</strong></td>
<td>Inter-Integrated Circuit, a serial two wire interface</td>
</tr>
<tr>
<td><strong>INI</strong></td>
<td>Institute of Neuroinformatics at ETH and University of Zurich</td>
</tr>
<tr>
<td><strong>JTAG</strong></td>
<td>Joint Test Action Group, an interface for debugging and programming chips</td>
</tr>
<tr>
<td><strong>LUT</strong></td>
<td>Look-up Table</td>
</tr>
<tr>
<td><strong>LVDS</strong></td>
<td>Low-Voltage Differential Signalling</td>
</tr>
<tr>
<td><strong>MMv2</strong></td>
<td>an AER mapper system, see sec. 7.4</td>
</tr>
<tr>
<td><strong>Mapping</strong></td>
<td>transforming Address-Events, see sec. 7.1.1</td>
</tr>
<tr>
<td><strong>Monitoring</strong></td>
<td>recording AEs, see sec. 1.2.2</td>
</tr>
<tr>
<td><strong>PCB</strong></td>
<td>Printed Circuit Board</td>
</tr>
<tr>
<td><strong>PCI</strong></td>
<td>Peripheral Component Interconnect, a PC interface</td>
</tr>
<tr>
<td><strong>PCI-AER</strong></td>
<td>see sec. 2.3.1</td>
</tr>
<tr>
<td><strong>REQ</strong></td>
<td>Request Signal</td>
</tr>
<tr>
<td><strong>SAER</strong></td>
<td>Serial AER</td>
</tr>
<tr>
<td><strong>SCX</strong></td>
<td>Silicon Cortex Project, see sec. 2.2</td>
</tr>
<tr>
<td><strong>SerDes</strong></td>
<td>Serializer-Deserializer chip or part of an FPGA</td>
</tr>
<tr>
<td><strong>SW</strong></td>
<td>Software</td>
</tr>
<tr>
<td><strong>Sequencing</strong></td>
<td>playing back timestamped AEs, see sec. 1.2.2</td>
</tr>
<tr>
<td><strong>Serial AER</strong></td>
<td>see sec. 4.3</td>
</tr>
<tr>
<td><strong>Sniffing</strong></td>
<td>Monitoring an AER stream without consuming it or interfering with it.</td>
</tr>
<tr>
<td><strong>USB</strong></td>
<td>Universal Serial Bus, a PC interface</td>
</tr>
<tr>
<td><strong>USBAERmini2</strong></td>
<td>see sec. 2.3.2</td>
</tr>
<tr>
<td><strong>VCC</strong></td>
<td>Supply (positive) Power-Rail</td>
</tr>
<tr>
<td><strong>VHDL</strong></td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td><strong>VHSIC</strong></td>
<td>Very High Speed Integrated Circuit</td>
</tr>
<tr>
<td><strong>VLSI</strong></td>
<td>Very-Large-Scale Integration</td>
</tr>
<tr>
<td><strong>ZIF-socket</strong></td>
<td>Zero Insertion Force (chip) Socket, a socket that allows to mount and exchange chips without soldering involved, typically used for testing many chips.</td>
</tr>
</tbody>
</table>
Introduction

Neuromorphic Systems and asynchronous event-based communication, “Address-Event Representation” (AER), are the core concepts of this thesis.

We will thus start with a quick introduction to neuromorphic systems and then explain AER and why it is used in such systems, even though AER is quite unlike how biological neural systems communicate.

We will then present the problems addressed in this thesis, what the goals of the thesis are and what key design principles were guiding us throughout the entire thesis.

Finally the introductory chapter will conclude by giving an overview of the chapters the thesis consists of, and by summarizing the key contributions to the field of research that resulted from this thesis.

1.1 Neuromorphic Systems

Neuromorphic information processing and computation is investigated by many research labs worldwide by building neuromorphic systems, i.e. systems that somehow mimic the computational principles and other concepts learnt from research on biological neural systems.

A wide variety of approaches are used to build such systems, from software simulations to hardware implementations.

Software simulations of neuromorphic systems range from simulators for small-scale neural networks targeting regular personal computers to large-scale simulations on supercomputers at various levels of abstraction of models of neural systems. Recently such simulations have also been
implemented on graphics processing units, extending neuromorphic computation into the area of general-purpose GPU research.

FPGA based or accelerated simulations of spiking neural networks are an approach on the middle ground between pure software and fully customized hardware.

Lastly a number of research groups also work on systems involving neuromorphic silicon chips, VLSI implementations of neurons or entire neural networks. These chips can use very different design principles. There are chips using the analog electrical properties of VLSI circuits to directly model the biological characteristics of neurons and synapses. Other chips use pure digital computation to implement neurons and synapses. Chips can also be synchronously clocked or they can employ asynchronous circuits. Chips using a mixture of these principles are also possible, such as chips consisting of synchronous and asynchronous parts and/or using both analog and digital circuits.

1.1.1 Neurons and Synapses in Analog VLSI Circuits

In 1989, Carver Mead published two books describing how to implement neural systems in analog VLSI circuits [Mead 89b, Mead 89a], after he and his collaborators made the observation that VLSI transistors operating in sub-threshold regime have some similarities in their characteristics to characteristics observed in biological neural systems.

Inspiring many other researchers, he began using these newly discovered principles to implement neural-like systems in VLSI circuits, “Neuromorphic VLSI Systems” were born.

For a summary of the history of the field of neuromorphic engineering by people who were there from the very beginning, please refer to [Liu 15, sec. 1.1, p.3f].

1.2 Address-Event Representation

As we will see, Address-Event Representation is how neurons and synapses are usually connected to each other in neuromorphic VLSI systems. AER can be used to communicate on-chip, to interface between multiple chips on the same board, and even for communication between chips on multiple boards more complex experimental setups. Parallel AER interfaces are the most common approach in traditional neuromorphic multi-chip systems.
1.2 Address-Event Representation

Figure 1.1: Optic nerve compared to a telephone trunk cable, with wires having been fanned-out into a tree-like configuration.

We will also see that AER communicates quite unlike nature does, given that we want to do things as neuromorphic as possible. This seems odd at first, but after the following sections, it will become obvious that given the technology available to us currently, AER communication is simply a necessity without alternative.

1.2.1 The Neurobiological Approach

Figure 1.1a shows an anatomical depiction of the human optic pathway (looking at a brain preparation from below) showing (top of figure to bottom) one eye, the optic nerves which merge at the optic chiasm and split into the optic tracts, which terminating in the two Lateral

---

1 © Image plate in public domain.
2 © Túrelio (via Wikimedia-Commons), 2007, License: Creative Commons CC-BY-SA-3.0-de. Exhibit at Fernmeldemuseum Aachen.
1 Introduction

Geniculate Nuclei (LGN), parts of the Thalamus (from [Gray 18]). For details about how the optic nerve, chiasm and tract are wired up, refer to figure 9.2.

According to various sources, each optic nerve and tract usually contains about or a bit over a million axons each [Kandel 00, Rea 14]. “The diameter of the optic nerve increases from about 1.6 mm within the eye to 3.5 mm in the orbit to 4.5 mm within the cranial space.” (Quoted from: [Wikipedia 15]).

The diameter varies because of various degrees of myelination. Myelin is the isolation material of the axons, the more of it there is around an axon, the faster the axon can propagate a spike (action-potential). The velocities spikes propagate at vary from 0.45 m/s to 1.0 m/s within the eye (where there is no space for myelin) and between 1.3 m/s and 20 m/s, depending on the diameter of the optic nerve (and thus degree of myelination) and can be approximated by $V = 3.2 \cdot D$, where: $V$: Velocity in m/s, $D$: Diameter in mm. The numbers are taken from a study on rhesus monkeys: [Ogden 66], where the optic nerve leaves the eye, at a diameter of 1.6 mm, we can calculate that an axon takes about 2 µm of diametrical area.

Now let’s compare this to a urban trunk cable of a telephone company, as they are still widely used even today. Figure 1.1b shows a cross-section of such a cable.

According to measurements performed personally, such a cable containing 2’400 fibers (a fiber in a telephone cable is assumed to consist of a pair of wires) has a diameter of about 10 cm. We can thus calculate that a single telephone wire occupies diametrical area of 1.6 mm$^2$ which equals $1.6 \cdot 10^6 \mu$m$^2$, which is about 800’000 times more area per wire/axon compared to the optic nerve.

While the copper cable is at a massive disadvantage when we compare how many wires per area there are, when comparing velocities instead, we get quite the opposite result. As mentioned, when leaving the eye, the axonal velocity is roughly 1 m/s. Let’s take the electron shock-front velocity in copper for comparison (which is the physical upper bound for velocities for information transfer in copper). This velocity is about $\frac{1}{2} \cdot c = 1.5 \cdot 10^8$ m/s, or 150 million times faster than the axonal velocity we mentioned before.

We can conclude, that nature is better at biochemical information transfer systems using massive amount of “wires” (axons), while electrical engineers can build electrical (also optical) information transfer...
1.2 Address-Event Representation

systems using much fewer “wires” than nature, due to the higher bandwidth available per “wire”.

1.2.2 The AER Approach

Why we have to use something like the “Address-Event Representation”, or AER, is quite understandable after what we just saw.

We have to use substantially fewer copper wires, than biology has axons available, anything else is technically impossible.

The AER approach is to number all the neurons from which the axons emerge, in the case discussed, the eyes, these are the “Retinal Ganglia Cells” (RGC), which each emit an axon into the optic nerve.

When such a numbered neuron fires a spike, we don’t have to send that spike down a wire dedicated to that neuron, we can simply send the number of that neuron, encoded in some way, over a few copper wires. In traditional parallel AER we simply take the binary encoding of that number, and send it down a parallel set of wires, called a bus. The number of wires required with that encoding is:

Figure 1.2: AER – Address-Event Representation, adapted from [Deiss 98]
1 Introduction

\[ \#\text{wires} = \lceil \log_2(\#\text{neurons}) \rceil \]

Figure 1.2 illustrates an AER connection between the numbered neurons in the source chip, and similarly numbered synapses in the destination chip.

This can be understood as a form of time-division multiplexing, because one address bus is used by multiple neurons at different times (made possible by the much higher speed of electrical signals).

For further details on AER communication, concepts and discussion of issues arising in AER, refer to [Liu 15, Chapter 2 – “Communication”].

1.2.3 Timestamped Address-Events – AER Monitoring and Sequencing

In AER spikes are typically transmitted as fast as possible once they occur, and the time of the spike is not explicitly encoded, hence the saying that “time represents itself”.

If it is required to record and store AER streams, one obviously also needs to store the spike timing. For this purpose “AER Monitors” are used. They attach a timestamp to each address-event as soon as it arrives at the monitor.

For the reverse process, “AER Sequencers” are used. The receive address-events with timestamp or inter-spike-interval values attached, and play back non-timestamped address-events according to that timing information.

Thus, we can have two AER domains, one where “time represents itself” and one where we have explicit timing information attached to each address-event. Monitors and sequencers form the gateway between these two domains. For an extensive discussion of AER monitoring and sequencing, and the issues associated with it, refer to [Liu 15, sec. 13.1.1f., p. 307ff.].

1.3 Problems Addressed in this Thesis

Now that we have explained the Address-Event Representation, let’s introduce the key problems addressed in this thesis.
1.3 Problems Addressed in this Thesis

1.3.1 AER Interfacing

One of the key issues addressed here, is how to interface AER systems on various levels, between neuromorphic chips, between multiple boards containing neuromorphic chips, between a board with a neuromorphic chip and a computer or between an entire neuromorphic multi-board multi-chip system and computers.

AER interfacing solutions available to researchers at INI before the work on this thesis were often not fitting the needs of researchers wanting to perform experiments with neuromorphic VLSI chips, and thus new AER interfacing systems and concepts were required.

1.3.2 AER Processing

AER processing is also an important challenge in neuromorphic systems. One needs to route and filter address-events (AEs) and to perform complex operations such as translating an AE into another or even many other AEs, which is called “AER Mapping”.

Obviously the level of complexity of AER processing required in a neuromorphic system varies substantially from system to system, but in order to get a complex connectivity pattern between an AER source (neurons) and AER destination (synapses) complex mapping operations are required.

1.3.3 Embedded, Mobile and Robotic Application of AER

Often AER systems are quite bulky and hardly suitable for embedded, mobile or robotic application. Hence, we also investigate how we can approach that issue in order to provide AER interfacing and processing e.g. for a robotic application.

1.3.4 Scalable Heterogeneous Multi-Chip AER Systems

Finally scalability of an information processing system is a big question in current research as it is, not only restricted to neuromorphic information processing systems.

In our case, the question is how to build AER interfacing and processing systems in order to achieve scalability of these components and thus scalability of an entire neuromorphic system.
1 Introduction

1.4 Thesis Goals

Based on the problems we just discussed, the goal of this thesis is to build experimentation platforms solving AER interfacing and processing issues of neuromorphic computation systems as built at the Institute of Neuroinformatics at ETH Zurich and the University of Zurich (INI).

A wide range of neuromorphic chips have been developed at INI, e.g. sensory chips such as silicon retinas and cochleas as well as numerous neuromorphic chips implementing neuron and synapse circuits, most of them using AER for communication.

Based on all this research a next generation neuromorphic platform for such sensors and chips will be developed and investigated. Custom neuromorphic sensors and processing chips will be integrated with FPGAs to allow for easy reconfigurable routing and digital processing in the system. Common parallel AER (Address-Event Representation) communication will be used and novel approaches of AER communication employing low-voltage differential signaling (LVDS) to build high-speed serial AER interfaces will be developed.

The various components that form the parts of such a system should be reusable in a modular way. This will allow for customization of a neuromorphic computation system to suit a specific problem, and the modularity and versatility of the platform presented will enable it to be used not only with the current generation of systems, but also with future generations that will be developed at INI and other research institutions.

We will investigate how to enable applications of these systems in mobile robotic settings where space and weight constraints are of concern. Here the modularity and versatility should maximize the reuse of platform components and minimize the additional engineering effort and risk when customizing the system to the new constraints of a highly embedded application.

Finally, the platforms will be evolved towards a highly scalable system, allowing for the integration of dozens of neuromorphic chips, still keeping the modularity and versatility of the platform. This will form the basis of a system capable to be evolved to form clusters of neuromorphic VLSI chips, not only as a homogeneous multi-chip neuromorphic information processing platform, but also allow for heterogeneous multi-chip cluster systems which integrate a variety of neuromorphic sensory and processing chips in future experimental setups.
1.5 Key Design Principles

Two major design principles were key to the choice of architecture in all the systems that were built as part of this thesis.

1.5.1 Maximize Flexibility Where Possible

Neuromorphic chips implementing neurons and synapses typically use mixed-signal VLSI circuits. This makes it necessary to build such chips as a full-custom ASIC (application specific integrated circuit). The chips available at the start of this thesis typically consisted of:

- Input AER Circuitry
- Synapses
- Neurons
- Output AER Circuitry

This means that this part is carved in stone (etched in silicon, to be more precise), and cannot be modified after being sent for fabrication. The development cycles are very long too. It usually takes months for a chip to come back from fabrication.

In order to have more flexibility in the overall neuromorphic system, maximizing flexibility of the other parts of the system was always a key principle. This can be achieved by using highly-flexible components such as FPGAs and CPUs between the neuromorphic ASICS, to perform the AER routing, filtering, mapping and interfacing tasks required in a complete neuromorphic system.

Why Maximize Flexibility?

First of all, you minimize your overall design risk by implementing as many parts of your system as possible in a flexible substrate rather than in an immutable ASIC. You are also reducing the size of your ASIC, which usually improves your ASIC testing performance or coverage.

If your routing algorithm which doesn’t behave as expected is part of the ASIC, you need to create a new revision of that ASIC after a usually cumbersome search for what might cause the unexpected behaviour.

However if that routing algorithm is implemented in an FPGA, the problem can be found with much more ease, because it is possible to observe internal signals in an FPGA quite easily, unlike in an ASIC.
1 Introduction

Fixing it by rebuilding and reprogramming the FPGA is usually a matter of minutes, after the bug has been found.

Due to these drastically shorter development cycles, the turn-around time and flexibility of your research, e.g. on various routing methods, is drastically improved.

1.5.2 Use Off-The-Shelf Components Where Feasible

We also integrate off-the-shelf boards as parts in our systems wherever applicable, e.g. in case of the MMv2 AER Mapper in chapter 7 an FPGA board and even parts of a PC were used.

The systems presented where we could not do that had specific mechanical constraints, the AEXv4 (chapter 4) had to connect directly to the already existing AMDA board, the iHead board had to fit into the iCub robot (chapter 9).

Why Use Off-The-Shelf Components?

Again, you reduce your design risk by trusting you buy something that works as specified, but you also increase your turn-around time by not developing that part the system yourself.
1.6 Thesis Outline

In this section, we provide an overview of the outline of the thesis by giving a short summary of the contents and main focus of each thesis chapter.

**Chapter 2,** “Related Work – Strengths and Weaknesses, AER Interfacing”, shows what it typically takes to get from a neuromorphic VLSI chip to a functional neuromorphic experimentation platform using that chip or multiple chips. We show approaches to provide a solution to that problem, focussing strongly on those systems pre-dating this thesis as they had the most influence on the work presented here. Then we further analyze the systems considered very influential for this thesis and conclude on what their strengths and weaknesses are, either in their design, or through observation of researchers using those systems.

Finally, in order to give you a better understanding of how these systems typically communicate, we offer a detailed overview of parallel AER interfacing. Then we explain, why parallel AER interfaces are not suitable for medium to long distance communication, and why high-speed serial AER interfaces using LVDS technology can solve these problems.

**Chapter 3,** “Design Requirements and Prototypes of the “AEX” AER Interfacing Platform”, starts compiling a list of design requirements we established for a new AER interfacing and processing platform, mostly based on what we learnt from the strength and weaknesses of the influential systems predating this thesis, which we presented in the previous chapter.

We then introduce the first two prototype versions of a system called AEX, which were built and described already in [Fasnacht 07a]. Then we present the third prototype, AEXv3, which was the first version of the AEX platform to meet the design requirements we have previously established. This was also the first version to be used by other researchers to perform experiments with various neuromorphic VLSI chips.

**Chapter 4,** “AEXv4 – Hardware Implementation Details”, details the fourth and final version of the AEX platform, the AEXv4. It is the result of a complete overhaul of the previously mentioned AEX prototype designs, with various optimizations and a much more compact design, reducing the PCB size to almost half of the last prototype.
1 Introduction

We then describe the hardware implementation details of the AEXv4: the Parallel AER interface implementations, the novel “3 GHz Serial AER interface”, and the high-speed USB 2.0 interface. The FPGA implementation of these interfaces is also explained, by describing the key VHDL entities developed to enable the central FPGA on the AEX to communicate via these three types of AER interfaces.

Finally the chapter is concluded with benchmark measurements to demonstrate the interface performance, and a specification sheet of the AEXv4 interfacing capabilities.

Chapter 5, “Generic FPGA Codebase for AER Interfacing and Processing”, describes the VHDL entities that form the “generic AER routing fabric” which is used to route, filter and process address-events at the core of the FPGA of AEX platform. The chapter follows a bottom-up approach, starting with a description of the interface common to all VHDL entities presented, then the basic building blocks at to bottom of the fabric.

Eventually the highest level is described in detail. It abstracts the generic AER routing fabric at such a high level that even a researcher with zero VHDL knowledge can configure and customize the AEX systems in his experimental setup without help from a FPGA/VHDL expert.

Chapter 6, “USB Interface Software Implementation”, describes all the software aspects of the AEX not described so far, namely the high-speed USB interface. The chapter follows the data-path from the AEX to the PC, thus starting with the firmware that was written for the FX2 USB interface chip present on the AEX.

Then the Linux kernel driver is introduced including many of the architectural details which give the kernel driver its extremely high performance. The user-space reference implementation communicating with the kernel driver is then presented and an example is given how these three software components, firmware, kernel driver and user-space code, operate with each other.

Eventually, we demonstrate the measurement results acquired to show the extremely high performance of the kernel driver, showing that it performs more than 100 times than the underlying USB protocol limits it to.

Chapter 7, “Creating Complex and Reconfigurable AER Connectivity”, starts by explaining that in order to achieve any non-trivial con-
nectivity in an AER system, “AER mapper systems” are required to translate address-events to form complex, sometimes even non-static connectivity between neuromorphic neurons and synapses. We explain how and why an earlier architectural approach was abandoned, and what new approach was devised to fit our needs. Then the first AER mapper prototype using our new architecture is outlined, which led then to the design of the “MMv2 AER mapper” system.

The second half of the chapter describes the implementation details of both the MMv2 hardware, as well as the architecture of the FPGA-implemented mapper system components. Then an alternate operation mode of the mapper, which allows for probabilistic mappings, is explained.

Finally, the chapter concludes by establishing the key performance parameters of the mapper, the latency and bandwidth, under a number of different mapping table fan-out conditions. The chapter concludes that the bandwidth is only limited by the PCI bus, not the mapper, and that for all multi-chip experiments performed with that mapper system, the MMv2 mapper was never considered the limiting factor.

Chapter 8, “AEXS – Towards Embedded and Robotic Applications”, introduces the AEXS system, an extremely compact AER interfacing platform based on the AEX platform and most of the material presented in the previous chapters.

This system was built specifically for embedded, mobile and robotic applications, where space is most frequently constrained. The AEXS served as a direct prototype and proof-of-concept for the work performed for the eMorph project, which is detailed in the next chapter.

The chapter ends by comparing key characteristics of the AEXS to the USBAERmini2 system shown earlier as influential related work, which has very similar functionality and concludes that the AEXS clearly exceeds the capabilities of the USBAERmini2 in many ways.

Chapter 9, “eMorph – a Neuromorphic Visual Pathway for the iCub Robot”, introduces the iCub robot and the goal of the eMorph project to build a neuromorphic visual pathway for the iCub. In a neuroanatomical analogy, all the parts involved in that pathway are explained.

Then the architectural details are explained in detail, and it is shown how the AEXS from the previous chapter was combined with the GAEP neuromorphic processor and merged into the key hardware component of the eMorph project, the iHead board. The FPGA internal architecture
is then explained in detail, with a focus on how many components previously developed for the AEX platform were reused successfully, due to their generic nature.

Finally, we characterize some of the performance parameters of the iHead system using measurements and optimizations performed together with the GAEP developers at AT& in Vienna. The chapter concludes by giving an overview of the final eMorph hardware extension to the iCub robot.

Chapter 10, “Towards a Modular & Scalable Neuromorphic Experimentation Platform”, After the work presented in all preceding chapters, an AER interfacing and processing architecture was devised. It was based on FPGA technology which newly became available, while learning from all the work presented so far, including from the experience of other researchers using it, and even aiming at superseding and replacing that work in most use-cases eventually. This endeavour was named AEXL project.

Instead of going through multiple iterations of building an FPGA board from scratch, and encouraged by the good experience with the off-the-shelf FPGA board (Raggedstone 1) used in the MMv2 mapper, it was decided to build a novel AER interfacing and processing platform based on the successor of the mapper FPGA board, the “Raggedstone 2” featuring a Spartan 6 FPGA.

We explain why this FPGA board was selected as the basis of the AEXL project and describe its key features, interfacing and expansion capabilities. We will then present a variety of expansion PCBs built for the AEXL in order to attain the same features we had with the AEX system in an initial step.

Unlike the AEX project, the AEXL quickly became a collaborative project with many PhD students at INI involved. We conclude the chapter by describing what the current status of the AEXL is, and what prospects there are for future expansion to make the AEXL the basis of a medium to large-scale heterogeneous multi-chip system.

Chapter 11, “Impact & Conclusion”, starts by demonstrating the impact of this thesis by analyzing how many publications were enabled by using the work of this thesis. We also indicate how the work of this thesis has dispersed on the planet and how the AEXL became a community project.

Then we illustrate this impact by presenting a number of key experiments and publications: single-AEX experiments, multi-chip AEX
1.7 Primary Contributions

To conclude this introduction, we summarize the primary contributions of this thesis as concise as possible.

1.7.1 AEXv4 Generic AER Interfacing Platform

The AEXv4 AER interfacing system provided an ideal platform for many experiments researchers wanted to perform. For example to interface a single neuromorphic chip on an AMDA board to a computer, the PCI-AER board was used by many researchers.

For such use-cases the PCI-AER became obsolete almost immediately and was superseded in many cases already by prototypes of the AEX system.

1.7.2 3 GHz Serial AER Board-to-Board Interface

As this thesis explains, the state-of-the-art parallel AER interfaces are not very well suited for board-to-board AER communications and frequently cause problems in experimental setups. Also they can only transmit up to a few million 16-bit address-events per second, depending on cable length and other factors.
With the 3 GHz serial AER board-to-board interface, all these problems could be solved, and a much more robust interfacing solution for multi-chip multi-board setups could be provided, which even surpassed the speeds traditionally on parallel AER board-to-board interfaces drastically. With a peak address-event rate of 156.25 MHz (for 16-bit address-events), it surpasses the parallel AER performance by one to two orders of magnitude, which e.g. allowed us the integration of high-fanout AER mappers, which produce address-events at rates much too high for parallel AER interfaces.

While this interface was developed and evolved as part of the AEX platform, it is also present on the MMv2 AER mapper system, and even the AEXL platform contains a compatibility module for that interface.

1.7.3 Generic VHDL framework for AER Processing in FPGAs

The generic VHDL framework developed for the AEXv4 platform, the interfacing & monitoring sequencing parts as well as the routing and processing parts (generic AER routing fabric) were a great success, when measured on the scale of how much of these VHDL entities and groups of entities could be reused in how many of the systems presented in this thesis.

The parallel AER, routing fabric parts and all of the USB interface components for example, were/are not only used on the AEX platform, but also the AEXS, eMorph-iHead and even on the current AEXL system.

1.7.4 MMv2 AER Mapper System

The MMv2 mapper system in combination with the serial AER interfaces and the AEX/AMDA platform turned out to be quite a powerful system for medium-scale multi-chip neuromorphic experimental setups.

Before the MMv2 mapper, the most-commonly used mapper system was still the PCI-AER, because the AEX alone could not perform complex mappings. The AEX/AMDA system was now complemented with a mapper system compatible with the AEX via the same 3 GHz Serial AER interface.

Given the much higher performance of the combined MMv2–AEX–AMDA platform as compared to PCI-AER & AMDA based systems,
the PCI-AER systems were quickly abandoned almost completely by most researchers for the newly developed approach.

1.7.5 Embedded AER Processing System for the iCub Robot

The development of the AEX and then the AEXS into a key component of the eMorph-iHead system was also great success because many aspects of the hardware had to be changed and adapted only in minor or very predictable ways, which allowed for a smooth incremental hardware development and testing process. It is also a collaboration success story, because even though three consortium partners from three different countries collaborated when building the iHead system, they achieved that the iHead board built worked as expected already in its first revision.

The fact that the AEXv4 platform, hardware and software-wise could be morphed into something as unforeseen as an important part of the neuromorphic visual pathway of the iCub, confirmed that many of the design decisions and strategies taken along the way were right.

1.7.6 The AEXL Project, a Scalable Multi-Chip AER Platform

The most recent work presented in this thesis is the AEXL project. Centered around an FPGA board with many expansion header connectors, numerous expansion boards have already been designed for this project.

At the end of this thesis, the AEXL systems already superseded the AEXv4 and MMv2 AER interfacing and mapper systems. It is now the most widely used AER platform at INI. However, most of the VHDL software for the AEXv4 FPGA and all of the components for the USB interface (VHDL, firmware, kernel-driver, user-space programs) could again be reused with little to no modification in the context of the AEXL project.
1 Introduction
Related Work – Strengths and Weaknesses, AER Interfacing

Neuromorphic VLSI chips need to be interfaced to computers and in order to create multi-chip experimental setups they need to be connected to each other. In this chapter, we establish what the requirements for a generic interfacing platform for neuromorphic chips are.

We give an overview of an early approach to building a generic platform for neuromorphic experimentation systems. Then we present the related work that was most influential for this thesis and which was previously used to approach the problems we worked on and inspired the work described in this chapter. We then analyze this related work, explaining which strengths and weaknesses it has.

We also introduce the traditional parallel AER interfaces used by most of the related work presented, and then explain why there is the urgent need for a novel serial AER interface, due to limitations inherent to parallel signalling interfaces in general, not only parallel AER interfaces.

2.1 Overview

Numerous research groups have built a wide range of neuromorphic chips in recent years, many of them using AER communication interfaces. These chips range from sensory systems such as retinas and cochleas, to chips modelling properties of neurons and synapses, to spike-based learning chips and other neuromorphic processing chips. A review of some of these chips can be found in [Indiveri 07].
With the availability of a multitude of chips that can be used as building blocks for a bigger system, multi-chip neuromorphic systems have become more and more common [Serrano-Gotarredona 06].

Because of this trend towards multi-chip experimental setups the design and construction of an AER communication and processing infrastructure for building multi-chip setups was badly needed.

### 2.1.1 Typical Requirements of a Neuromorphic Chip

Event-based neuromorphic VLSI chips typically require a number of different types of interfaces / connectors (pins) on a chip. The most common types are listed in Table 2.1.

<table>
<thead>
<tr>
<th>Interfaces / Connectors / Signals Typically Present on a Neuromorphic Chip</th>
</tr>
</thead>
<tbody>
<tr>
<td>• AER input and/or output interfaces,</td>
</tr>
<tr>
<td>• configuration inputs, either:</td>
</tr>
<tr>
<td>– analog bias voltage inputs or</td>
</tr>
<tr>
<td>– control inputs for an on-chip digital bias generator,</td>
</tr>
<tr>
<td>• various analog and digital power supply and ground pins,</td>
</tr>
<tr>
<td>• sometimes other analog or digital output pins to observe internal states of the chip for debugging or measurements.</td>
</tr>
</tbody>
</table>

Table 2.1: Interfaces / Connectors / Signals Typically Present on a Neuromorphic Chip.

To build a neuromorphic system integrating such chips, numerous supporting circuits and interfacing capabilities have to be provided.

**AER Interfacing** The AER input and output interfaces need to be connected according to the desired setup, either to other chips or AER processing systems on the same or other circuit boards, or to a computer, to monitor address-events from the chip or sequence address-events to the chip.

In sections 2.3.1 and 2.3.2 we present two systems that were developed and widely used for this purpose.
2.1 Overview

Figure 2.1: Generic circuit board for AER chips from the CAVIAR with dozens of potentiometers (blue) to manually control the bias voltages of the chip. (adapted from [Häfliger et al. 04a])

**Analog Bias Voltage Inputs**  Analog bias voltage inputs need to be provided with individually controllable bias voltages. Traditionally, this was achieved by having a potentiometer for each of the bias values on the circuit board hosting the chip (Figure 2.1, adapted from [Häfliger et al. 04a]).

Alternatively, bias voltages can also be generated using DACs. The bias voltages generated by these DACs can then be controlled electronically, e.g. via a microcontroller, which receives the bias values from a computer. This provides many advantages: a certain set of bias values can be stored and restored quite easily, or the computer can even be used to sweep bias voltages in search of an optimal value. The system presented in section 2.3.3 shows a system that was developed for just this purpose.

**On-chip Digital Bias Voltage Generator**  Digital bias voltage generators are typically controlled using a simple microcontroller. Bias values can then either be programmed into the microcontroller firmware or
are sent to the microcontroller from a computer.

Such on chip digital bias voltage generator circuits are presented for example in: [Delbruck 06, Delbruck 10a]. There is also an extensive overview over digital bias voltage generator circuits presented in [Liu 15, Chapter 11 – Bias Generator Circuits].

Figure 2.2 shows an example of a neuromorphic vision sensor system which incorporates such digital bias voltage generator circuits, and provides the sensor output via a parallel AER interface.

**Analog Outputs**  Analog outputs are usually just connected to a header connector for slow signals or special coaxial connectors for high frequency outputs. They can then be observed during operation using an oscilloscope.

**Power Supply**  Obviously, the chip needs to be powered according to specification. Depending on how many different voltages and supply rails (e.g. separate rails for analog and digital sections on the chip) there are, the complexity of the power supply circuitry varies depending on the chips requirements.
2.1 Overview

Figure 2.2: DVS128 PARALLEL AER board by Angel Jimenez and T. Delbruck. The board contains a tmpdiff128 neuromorphic dynamic vision sensor (DVS) that incorporates digital bias voltage generator circuitry. The PCB with the sensor chip is mounted on a tripod on the bottom side. No optics have yet been mounted in front of the chip in this test system. The digital biases are set via a microcontroller mounted on the rear (bottom) side of the PCB, which can be interfaced to from a PC via the USB connector visible on the left PCB edge. The sensor output is accessible via the two AER connectors mounted on the rear (bottom) side (their pins can be seen on the right). Both the 20-pin “Rome-type” AER connectors (see table 2.6) and the 40-pin “CAVIAR” AER connectors (see table 2.7) are available.
2.2 Early Neuromorphic Platforms – SCX Project

While the idea to build neuromorphic systems based on neuromorphic VLSI chips has been around, for quite some time researchers were focussing on single chip experiments and how to properly design neuromorphic circuits for VLSI chips.

Nowadays, researchers in the field of neuromorphic systems nowadays often also focus on building more generic neuromorphic substrates, generic infrastructure that can be used and reused with multiple generations of chips and even large-scale multi-chip systems.

One early example aiming at providing a generic infrastructure for neuromorphic chips while being capable to scale to a large-scale multi-chip neuromorphic system was the “Silicon Cortex” (SCX) project.

2.2.1 Silicon Cortex Architecture

Quoted from Adrian Whatley, a co-designer of the SCX system, [Whatley 15b]:

“The framework was devised to solve several fundamental problems encountered in building systems of analog chips that use the address-event protocol:

- Coordinating the activity of multiple sender/receiver chips on the same bus
- Providing a method of building a distributed network of local buses sufficient to build an indefinitely large system
- Providing a software-programmable facility for translating address-events that enables the user to configure arbitrary connections between neurons
- Providing almost limitless digital interfacing opportunity via VME bus
- Providing life-support for custom analog chips by maintaining volatile analog parameters or programming analog non-volatile storage capacitors (floating gates)

The SCX-1 framework is designed to be a flexible prototyping system, accommodating chips that may be designed to be incorporated into dedicated hardware systems that are not based on programmable interconnections.”
The key component of the SCX project was the SCX-1 board which is shown in figure 2.3.

The Silicon Cortex (SCX) project combines custom neuromorphic chips containing analog silicon neurons special chips managing the Local Address-Event Bus (LAEB) and Digital Signal Processors (DSPs). Address-event buses are used to transmit spikes between the various custom chips and DSPs involved.

On the SCX-1 board shown in figure 2.3, you can see two sockets for the local neuromorphic chips communicating via the LAEB in the bottom-right corner. There is also an expansion connector in the bottom-left corner, on which a daughter-board can be connected, which can carry up to another four chips which can communicate with the LAEB. At the top of the board, there are two connectors which allow the board to be connected to a VME-bus back-plane, which allows one to build systems consisting of many such boards, carrying up to six neuromorphic chips each.
2.2.2 AER mapping in SCX

In the SCX project [Deiss 98, Deiss 99] Digital Signal Processors (DSPs) were used to interconnect neuromorphic chips and do AE (Address-Event) mapping between them in order to allow the creation of complex connectivity patterns. The mapping table was stored in up to 64 KiB of RAM. A simple single-indirection table format (look-up table) was used, (more on AER mappers in chapter 7).

2.2.3 SCX Scalability

Given that many such boards could be combined, large-scale SCX systems could be imagined.

Quoted from [Liu 15]:

“The SCX framework was designed to be a flexible prototyping, providing re-programmable connectivity among the order of 10^4 computational nodes spread across multiple chips on a single board, or more across multiple boards.”

2.2.4 Conclusion

There is also an overview on the SCX project in [Liu 15, sec. 13.2.1, p. 316f.]. This overview also explains what limitations of the SCX project were driving researchers to build their own AER systems instead of building onto what the SCX architecture provides by concluding with:

Quoted from [Liu 15]:

“Because of the bulkiness of the SCX solution and the restrictions on the directly supported chip-pinouts, designers of multi-chip systems resorted to simpler independent boards for their needs.”
2.3 Very Influential Related Work

In the following section, we will present the related work that most strongly influenced the work described in the next few chapters of this thesis.

Figure 2.4: PCI-AER board (adapted from [Whatley 15a])

2.3.1 PCI-AER

Figure 2.4 shows the PCI-AER board designed by Vitorio Dante of the ISS Rome [Dante 05]. It was used in many neuromorphic experiments, one example being [Chicca 07]. It consists of a custom made PCI card with a specialized PCI interface chip, two FPGAs and multiple RAM and FIFO chips.

This PCI card connects to a breakout board via a ribbon cable of about one meter in length. The breakout board then has multiple AER interface connectors which can be connected to boards carrying neuromorphic chips.

The PCI-AER board supports two 16 bit AER interfaces. Up to four chips can be connected to each interface, given they use the SCX type AER interface, which allows multiple chips to share the same AER data lines (while each target has individual REQ and ACK lines, to arbitrate
Figure 2.5: PCI-AER block diagram
(adapted from [Whatley 15a])
bus access). The pinout of these parallel AER interfaces is shown in table 2.6.

The PCI-AER system can “monitor” incoming streams and then send the timestamped address-events via PCI to a program running on the computer. Vice-versa it can also “sequence” timestamped data provided by the computer via the PCI bus to any or all of the output channels.

As can be seen in the block-diagram of the PCI-AER system (Figure 2.5, adapted from [Whatley 15a]), there are two FPGAs implementing different functionalities of the system. While the first FPGA in the PCI-AER board mostly handles monitoring and sequencing, the second FPGA implements AER mapping functionality. A mapping table in a double-indirection format can be sent from the computer to the PCI-AER board, where it is stored in an SRAM which is 2 MWords in size. When an incoming address-event is to be mapped, the second FPGA looks up the resulting address-events in the SRAM and sends them out on an AER output.

Further details on the PCI-AER system can also be found in section 13.2.2 of [Liu 15, sec. 13.2.2, p. 317f.]. Technical specification and documentation as well as kernel driver and library software are available at [Whatley 15a].

### 2.3.2 CAVIAR project – USBAERmini2 board

Figure 2.6 (adapted from [Berner 06]) shows the USBAERmini2 build as part of the CAVIAR project and presented in [Berner 07] and [Berner 06].

This generic AER interfacing board features one parallel AER input for monitoring address-events from a chip and one parallel AER output for sequencing address-events from a computer to a chip. There is also a third parallel AER connector which is an AER output that produces the same address-events as present on the parallel AER input. It is used to “sniff” address-events.

In the USBAERmini2 board depicted in figure 2.6, all three AER interfaces are assembled with both the 20-pin “Rome-type” AER connectors (see table 2.6) and the 40-pin CAVIAR AER connectors (see table 2.7).

The USB interface to connect to a computer for monitoring and sequencing is implemented using a Cypress FX2 USB 2.0 high-speed
Figure 2.6: USBAERmini2 board, top and bottom view.
interface chip. Handling the AER communication and communication with the USB chip is done with a Xilinx CPLD chip. These two chips are mounted on the bottom side of the PCB.

2.3.3 AMDA

The traditional way of providing bias voltages via potentiometers, as for example in the board in figure 2.1, resulted in a very tedious experimentation process, with researchers spending hours tweaking their pot settings. Researchers have been looking for better approaches to improve the situation by generating bias voltages using high-precision DACs that can be controlled from a microcontroller or even a computer.

An early example following exactly this approach is the so called DUCK board ([Zahnd et al. 15]). It can simultaneously generate 24 bias voltages and has some additional features.

The most comprehensive system built at INI to resolve the dozens of potentiometers headache is the AMDA board, shown in figure 2.7. While incorporating many of the design principles of the aforementioned DUCK board, one of the key changes was that it was designed as a chip-carrier board.

In order to use a neuromorphic chip with the AMDA board, a simple daughter-board PCB has to be fabricated in order to plug the neuromorphic chip directly onto the AMDA board. This allows for free choice of package form factor and pinout for the neuromorphic chips to be used with the AMDA board, unlike e.g. the generic board previously shown in figure 2.1, where chip package and pinout are predetermined by the board supporting the chip.

The key features the AMDA board provides to a neuromorphic chip mounted on top of it (via a daughter-board) are summarized in Table 2.2.

The parallel AER input connectors of the AMDA board are compatible with those of the PCI-AER system. They also use the “Rome-Type” AER interface connector described in table 2.6.

Figure 2.7 shows an photograph of the AMDA board with the mentioned components labeled. The components labelled are:
### AMDA Board Key Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bias Voltages</strong></td>
<td>96 DAC channels to provide bias voltages to the chip, 16 bit resolution, DAC: Linear Technology LTC2600</td>
</tr>
<tr>
<td><strong>ADCs</strong></td>
<td>8 ADC channels to sample analog voltages produced by the chip, 12 bit resolution, ADC: Maxim MAX1227</td>
</tr>
<tr>
<td><strong>AER Output</strong></td>
<td>One 16-bit AER output interface (from the chip)</td>
</tr>
<tr>
<td><strong>AER Input</strong></td>
<td>One 16-bit AER input interface (to the chip)</td>
</tr>
<tr>
<td><strong>Power</strong></td>
<td>Power supply for chip and interface (separate analog and digital supply rails)</td>
</tr>
<tr>
<td><strong>Microcontroller</strong></td>
<td>Atmel Mega128 microcontroller to control the board (DACs, ADCs)</td>
</tr>
<tr>
<td><strong>Control Interface</strong></td>
<td>FTDI USB-serial converter chip to interface the microcontroller to a computer to set / get DAC (bias) &amp; ADC values, etc.</td>
</tr>
<tr>
<td><strong>Optocouplers</strong></td>
<td>The digital and analog sections of the AMDA board are separated electrically by using optocouplers between the microcontroller section and the DAC/ADC section. This improves the stability of the bias voltages provided to the neuromorphic chip.</td>
</tr>
<tr>
<td><strong>Scanner Interface</strong></td>
<td>The Scanner Interface consists of eight GPIO signals provided to the neuromorphic chip, which can be controlled set/read from the Atmel microcontroller. Four of them are digital inputs to the chip, four of them are digital outputs from the chip. Depending on the chip, this interface can be used in various ways, e.g. to select (scan) which analog signals the chip should present to the eight ADC channels.</td>
</tr>
</tbody>
</table>

Table 2.2: AMDA board key features provided to support the neuromorphic chip mounted via a daughter-board.
Figure 2.7: AMDA board, without a chip daughter-board mounted, bottom half: label overlay (see text).
Sections of the AMDA board PCB

The AMDA board PCB consists of a number of distinct sections, some even separated electrically via optocouplers to prevent electrical noise present in one (e.g. a digital) section from affecting another section (e.g. analog circuitry).

These sections are:

- **A**: Analog Section
- **C**: Microcontroller Section
- **D**: Digital Section including AER
- **PD**: Digital Power Section
- **PA**: Analog Power Section

**Analog Section**: The analog section contains all the following components, with their labels in figure 2.7 all starting with the character “A”:

- **A1**: Mezzanine connector for the chip daughter-board carrying analog signals and power (2x)
- **A2**: DACs, 4x group of three LTC2600
- **A3**: ADC, MAX1227

**Microcontroller Section**: The control section contains all the following components, with their labels in figure 2.7 all starting with the character “C”:

- **C1**: Atmel Microcontroller
- **C2**: FT232 USB-Serial interface chip and USB connector
- **C3**: Optocouplers (5x)

**Digital Section**: The digital and AER input-output section contains all the following components, with their labels in figure 2.7 all starting with the character “D”:

- **D1**: Mezzanine connector for the chip daughter-board digital signals and power
- **D2**: AER output connector (Rome-Type)
- **D3**: AER input connector (Rome-Type)
- **D4**: Voltage level shifters for 5V AER compatibility (2x2)
- **D5**: CPLD for optional SCX/P2P translation
Figure 2.8: The AMDA board providing bias voltages generated by DACs to two different neuromorphic chips, each mounted on top of a chip-specific daughter-board.
• **D6**: Shift registers for the scanner interface

Figure 2.8 shows the AMDA board in with a neuromorphic chip mounted on top (using a chip daughter-board). In the upper picture, the chip-cavity, silicon die, and bond-wires are all visible because in this chip the usual ceramic cover used in the die packaging process was replaced by a plain glass cover. The lower image shows an AMDA board with another neuromorphic chip and a setup requiring many scope probes attached to it. The oscilloscope probes can be seen on the right side of the photograph connected to various signals provided on header pins on the daughter-board.

### 2.3.4 Other Approaches

Before the systems we just presented were available, classical electrical-engineering approaches were often the only way to develop and experiment with neuromorphic chips. Digital Oscilloscopes, Logic-Analyzers and general purpose digital data acquisition systems often had to be used.

![Image](image-url)

Figure 2.9: 64-Channel Binaural Silicon Cochlea with integrated AER monitor and USB interface. (adapted from [Liu 10]).
As explained in [Chicca 07] the approaches using general purpose instruments for processing AER usually suffer drawbacks regarding asynchronous communication or on-line analysis of the acquired data. This is what motivated researchers to design special purpose hardware for building and debugging multi-chip AER systems, such as the ones we just presented.

Alternatively, some research groups building multi-chip AER systems don’t use generic AER infrastructure, but build special purpose PCBs on a per project basis or in an other way using much less generic approaches as described for example in [Choi 05] and [Merolla 07].

One example from INI taking this approach is presented in [Liu 10] and [Liu 15, sec. 4.3, p83ff.]: A binaural silicon cochlea chip is mounted together with microphones, pre-amps and the monitoring and USB interface (Figure 2.9). The chips and USB connector left of the label USB interface in this figure are identical to the one in the USBAERmini2 board presented in 2.3.2. The Silicon Cochlea Board can be mounted on a tripod and only requires a laptop connected via USB, resulting in a very compact setup for experimentation with and demonstration of the silicon cochlea chip.

The hardware presented in this thesis, however, is generic AER infrastructure, which is designed to be as modular and extensible as possible in order to maximize the infrastructure reuse to make research and experimentation with neuromorphic chips as efficient as possible.

2.4 Inspiration and Lessons from Related Work

One of the first goals of this thesis was to create a generic AER interfacing system, which integrates all the interfacing capabilities of the systems presented in the previous section, while improving on a number of their deficiencies.

For this reason, we first analyze the strengths and weaknesses of the related work presented, to then derive our design requirements for the AEX system introduced in this chapter.
2 Related Work – Strengths and Weaknesses, AER Interfacing

Positive features that influenced this work:

⊕ monitoring and sequencing capabilities
⊕ multiple parallel AER interfaces
⊕ easy control of the system from PC
⊕ open-source Linux Kernel driver

Drawbacks we decided to address comprise:

⊖ PCI interface is a complicating factor for some experiments
⊖ the use of the ribbon cable between main and breakout board and the integration of the system into a computer presumably facilitate EM issues in connected neuromorphic hardware
⊖ mapping capabilities of the system is quite limited
⊖ processing of AER streams wider than 16 bit is not possible
⊖ very hard to customize or debug due to its high complexity

Table 2.3: Influential factors of the PCI-AER system

2.4.1 PCI-AER – Strengths and Weaknesses

The PCI-AER board has proven to be a very powerful system invaluable to many of our experiments with neuromorphic chips. It was clear that its monitoring and sequencing functionality were a must-have for any future AER interfacing system built as part of this thesis.

But there are a number of drawbacks of the PCI-AER systems that we wanted to address in this thesis: The PCI form factor of the system requires a full personal computer to be used with every setup involving the PCI-AER board. In times where laptops are omnipresent, this has proven to be hindering in many cases. One obvious example is a simple demonstration of a neuromorphic chip at a conference or other out-of-lab situations.

Also the long ribbon cable between the PCI board and the break-out board has proven to sometimes cause or exacerbate electrical instabilities or problems with experimental setups. Since the PCI-AER system provides little isolation of EM-disturbances produced by the computer to be prevented from affecting the AER systems attached, instabilities could occur.
2.4 Inspiration and Lessons from Related Work

Also the mapping functionality has some deficiencies for certain experiments. The mapping table size is quite limited. Furthermore, changing or updating the mapping table is non-trivial and usually requires rewriting the complete table (or big parts of it).

Mapping aspects are not relevant for the thesis work presented in this chapter, which is only concerned with interfacing. However, it is addressed in chapter 7. Table 2.3 summarizes features and drawbacks of the AMDA system that influenced this work.

2.4.2 USBAERmini2 – Strengths and Weaknesses

The USBAERmini2 board from the CAVIAR project did already address one of the drawbacks of the PCI-AER system we just mentioned. It is using USB to interface to the computer, not PCI. However it is employing a proprietary USB driver, which has to be licensed by each user and which is only compatible with Windows operating systems. This hinders ease of use and exchange between research groups.

The board provides a very convenient monitoring and sequencing system, but it does not allow for any operations on the address-event streams. This cannot be easily circumvented by changing the programming of the CPLD which interfaces between the parallel AER connectors and the USB interface chip. While the Xilinx CPLD on the USBAERmini2 board is a very compact and cheap solution for the task at hand, its resources are used up almost completely by the default functionality of the USBAERmini2. This is also partially the case, because unlike FPGAs, CPLDs are not very well suited for implementing register transfer level logic designs.

The USBAERmini2 also still requires ribbon cables to attach AER devices. While the connector, cables and pinout used (CAVIAR type AER interface) are an improvement over the ones used by the PCI-AER system (Rome-type AER interface), they are still a sub-optimal solution for board-to-board AER interfaces, especially as systems grow and many boards are connected together. Table 2.4 summarizes features and drawbacks of the USBAERmini2 system that influenced this work.
Positive features that influenced this work:

⊕ monitoring and sequencing capabilities
⊕ USB interface improves usability and mobility
⊕ easy control of the system from PC
⊕ compact form factor

Drawbacks we decided to address comprise:

⊖ PCI interface is sometimes limiting
⊖ ribbon cables are used to connect to other AER boards
⊖ no mapping or even simple processing capabilities
⊖ hard to extend since almost no CPLD resources are unused
⊖ processing of AER streams wider than 16 bit is not possible
⊖ proprietary driver, limited mostly to Windows environments

Table 2.4: Influential factors of the USBAERmini2 system
2.4 Inspiration and Lessons from Related Work

2.4.3 AMDA – Strengths and Weaknesses

As described before, the AMDA system provides for most needs we have to support a neuromorphic chip. Since the AMDA board is available, most of the chips developed at INI were tested and used in experiments mounted on top of an AMDA board.

The main reason for this is the ease of providing many bias voltages to the chip, and the ease to control these during debugging or experiments via software from a computer.

Only since the aforementioned on-chip digital bias voltage generators have matured and gained widespread use, the main advantage of using the AMDA boards has lost significance. Table 2.5 summarizes features and drawbacks of the AMDA system that influenced this work.

<table>
<thead>
<tr>
<th>Things the AMDA boards already takes care of:</th>
</tr>
</thead>
<tbody>
<tr>
<td>☑️ provides bias voltages to the chip</td>
</tr>
<tr>
<td>☑️ USB interface to easily control bias values</td>
</tr>
<tr>
<td>☑️ provides analog and digital power rails to the chip</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>What the AMDA board doesn’t do for us:</th>
</tr>
</thead>
<tbody>
<tr>
<td>☐️ does not interfere with AER I/O in any significant way</td>
</tr>
<tr>
<td>☐️ no AER sequencing / monitoring / processing</td>
</tr>
<tr>
<td>☐️ typically requires ribbon cables to connect to other AER devices</td>
</tr>
</tbody>
</table>

Table 2.5: Influential factors of the AMDA system
2.5 Parallel AER Interfaces

The two most commonly used connectors for parallel AER Interfaces are the:

- “Rome-Type” AER connector, [Dante 05], shown in table 2.6, and the
- “CAVIAR-Type” AER connector, [Häfliger et al. 04b], shown in table 2.7.

All systems presented in section 2.3 implement a parallel AER interface some way. The generic AER circuit board in figure 2.1 as well as the USBAERmini2 board use CAVIAR-Type AER connectors to interface to other boards. The PCI-AER board and the AMDA board use Rome-Type AER connectors. Frequently the two systems have been used together, connected via their parallel AER interfaces. Even the Silicon Cochlea board shown in figure 2.9 uses a parallel AER interface, though only on-board, between the neuromorphic chip AEREAR2 and the PCB section labelled USB interface.

<table>
<thead>
<tr>
<th>Parallel AER Interface Pinout “Rome-Type”</th>
</tr>
</thead>
<tbody>
<tr>
<td>AE0</td>
</tr>
<tr>
<td>AE2</td>
</tr>
<tr>
<td>AE4</td>
</tr>
<tr>
<td>AE6</td>
</tr>
<tr>
<td>AE8</td>
</tr>
<tr>
<td>AE10</td>
</tr>
<tr>
<td>AE12</td>
</tr>
<tr>
<td>AE14</td>
</tr>
<tr>
<td>GND</td>
</tr>
<tr>
<td>REQ</td>
</tr>
</tbody>
</table>

(male header connector, seen from top)

Table 2.6: Parallel AER Interface Pinout “Rome-Type”
2.5 Parallel AER Interfaces

<table>
<thead>
<tr>
<th>Parallel AER Interface Pinout “CAVIAR-Type”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
</tr>
<tr>
<td>AE7</td>
</tr>
<tr>
<td>AE6</td>
</tr>
<tr>
<td>AE5</td>
</tr>
<tr>
<td>AE4</td>
</tr>
<tr>
<td>AE3</td>
</tr>
<tr>
<td>AE2</td>
</tr>
<tr>
<td>AE1</td>
</tr>
<tr>
<td>AE0</td>
</tr>
<tr>
<td>GND</td>
</tr>
<tr>
<td>REQ</td>
</tr>
<tr>
<td>Reserved</td>
</tr>
<tr>
<td>Reserved</td>
</tr>
<tr>
<td>Reserved</td>
</tr>
<tr>
<td>ACK</td>
</tr>
<tr>
<td>Reserved</td>
</tr>
<tr>
<td>Reserved</td>
</tr>
<tr>
<td>Reserved</td>
</tr>
<tr>
<td>Reserved</td>
</tr>
<tr>
<td>Reserved</td>
</tr>
</tbody>
</table>

(male header connector, seen from top)

Table 2.7: Parallel AER Interface Pinout “CAVIAR-Type”

Figure 2.10: 40-pin connector with 80 pin ribbon cable used in the CAVIAR-type parallel AER interface ([Fasnacht 07a]).
2.5.1 Interface Connectors and Cables

Both use two-row 2.54 mm pitch standard header connectors on the board side, with or without a connector latching mechanism. The “Rome” connector has 20 pins, while the CAVIAR connector has 40 pins.

Both connectors are specified for a 16 bit AER interface. The reason the CAVIAR connector uses a connector with twice the pins, thus making it twice the size, lies in the very special cables that are used in the CAVIAR interfaces. Those are taken from regular PC hardware supply: ATA hard drive interface cable compatible with the ATA speed specification UDMA66 and up (shown in figure 2.10).

These ribbon cables have the required 40 pin connector, but they have 80 wire half-pitch cables between them, where every other wire is connected to ground. This interleaved ground wire scheme reduces two problems commonly found with parallel AER interfaces:

- near-end cross-talk between the adjacent data lines
- ground-bounce issues, due to the much higher ground wire count and higher number of ground pins on the connector. The ground connection between the two boards connected is significantly better, when compared to “Rome-Type” connections.

2.5.2 Interface Protocols

There are a variety of protocols used in parallel AER interfaces. They vary in the type of hand-shake used, whether signals are interpreted as high- or low-active, but also in what voltage levels are used for which signals.

P2P and SCX hand-shake types

There are two commonly used AER signalling standards used in parallel AER interfaces

- **P2P** (Point-To-Point) parallel AER protocol, and the
- **SCX** (Silicon-Cortex) parallel AER protocol.

In both protocols, the AER source generates the REQ (request) and data (address) signals, the AER sink generates the ACK (acknowledge) signal. The request and acknowledge signals are used to perform a asynchronous four-way-handshake.
2.5 Parallel AER Interfaces

In the P2P mode, shown in figure 2.11 (adapted from [Häfliger et al. 04b]), the sender has to signal the valid address to be transferred to the receiver already when it asserts REQ, and keep the address lines valid, until the receiver asserts ACK. The handshake is completed by the sender deasserting REQ and then the receiver deasserting ACK.

In SCX protocol mode (Fig. 2.12, adapted from [Häfliger et al. 04b]), however, the AER source signals its intent to transmit an address by asserting REQ. However it must drive the address-bus with the valid address only after the AER sink asserts ACK. The source then deasserts REQ, but keeps the address-bus driven to the valid address until the AER sink deasserts ACK.

This allows that multiple AER sources share the same address-bus (while still having separate REQ/ACK signals to the AER sink). The

![Diagram of P2P-Type AER hand-shake](image)

**Figure 2.11**: **P2P-Type** (point-to-point) AER hand-shake, low-active REQ & ACK (adapted from [Häfliger et al. 04b]).

![Diagram of SCX-Type AER hand-shake](image)

**Figure 2.12**: **SCX-Type** AER hand-shake, low-active REQ & ACK, allows for multiple senders sharing the data-bus (adapted from [Häfliger et al. 04b]).
AER sink can then arbitrate the bus access between the sources by signalling ACK to only one source at a time.

In both protocols, described here in figures 2.11 and 2.12, REQ and ACK signals are defined to be low-active. However there also are AER devices, which use high-active REQ and ACK signals, most frequently in conjunction with the P2P protocol type.

Signalling Voltage Levels

In addition, voltage levels have to be defined for the address and handshake signals. While older devices often used 5 V, most current parallel AER interfaces use 3.3 V signalling.

With newer chips frequently using even lower I/O voltages such as 2.5 V and 1.8 V, parallel AER interface implementations are likely to follow that trend too.

2.5.3 Conclusion on Parallel AER

Due to these many different options in how a parallel AER interface can be implemented, interoperability between various devices can often be very challenging, and frequently requires additional glue-logic, such as inverters and level-shifters, sometimes even CPLDs or FPGAs.
2.6 Parallel versus Serial AER Interfaces

Parallel AER interfaces typically consist of e.g. 16 data lines that carry the address plus two handshake lines to transfer that address from source to destination.

This type of interface has been the de-facto standard for most AER systems. They have been used on-chip, on PCBs, e.g. between chips, but also to interface between PCBs (between multiple boards), usually connecting these PCBs with ribbon cables and header connectors.

While such parallel AER interfaces are very simple and unproblematic on-chip or on-board, they have proven to be increasingly problematic when used to connect between multiple boards.

2.6.1 Problems with the Parallel AER Interfaces

With the speeds that AER chips and systems have reached, the parallel AER approach in board to board communication has become a limiting factor in the system as it can cause unreliable behavior.

With the frequencies on parallel AER in the order of tens of Megahertz, the wavelength of those frequencies has shrunken to about the order of magnitude of our experimental setups, or only a little above.

One rule of thumb in electrical engineering says that if the signal wavelength is not at least one, if not two orders of magnitude above the spatial size of the system, the RF properties of the signals have to be taken into account. Wires cannot be interpreted any more as perfect conductors with the same potential at every point, but have to be treated as transmission lines.

Because these problems are usually not taken into account in parallel AER links using ribbon cables, the following problems can be observed:

**RF Sensitivity**  Because ribbon cables are usually neither shielded nor are the signals properly terminated, RF interference and sensitivity can be observed. They can disturb other components nearby, for example sensitive analog signals one wants to measure with an oscilloscope. But they can also be susceptible to being disturbed by RF sources themselves.
Cross-Talk Because the signal wires within a parallel AER link are usually also not shielded from each other (except maybe a little bit by interleaved ground wires). Thus they can also disturb other signals part of the same parallel AER link (near-end cross-talk) or signals running in the opposed direction (far-end cross-talk).

Ground-Bounce Usually parallel AER links use quite strong drivers on the sender end, and very often, they do not have proper resistive termination at the receiver end. This causes ground-bounce problems, meaning that the ground level of the two boards communicating with each other sometimes differ in the order of magnitude of the signal levels.

This can cause problems interpreting the signal levels on the receiver end correctly, as signal voltage levels only make sense with respect to a common ground potential that now sometimes significantly differ between the sender and the receiver end due to ground-bounce.

2.6.2 Trend towards Serial Differential Signaling

Single-Ended Signaling to Differential Signaling These problems have also played a major role in industrial and consumer electronics in general. The solution engineers came up with was to use even faster but differential links, and to carefully control the line impedance at every point between the sender and receiver.

In such a differential signaling scheme there is always a pair of wires that carry the opposite sense signal. The absolute value of the voltages on the signal wire do not have any meaning, only the voltage difference between the two corresponding wires has.

These so-called differential pairs are then usually shielded, thus not having the problems of RF sensitivity or cross-talking to other signal wires. Also RF emissions are drastically reduced because of the controlled impedance from signal source all the way to the properly terminated receiving end.

By using differential signaling, the ground-bounce problem is also eliminated. A differential driver always pushes as much charge into one wire as it pulls from the other. Thus the net charge flow is always zero.

Parallel to Serial Because the data rates that can be achieved using differential signaling are orders of magnitude higher than with traditional
2.6 Parallel versus Serial AER Interfaces

single-ended signaling, less (but better) wires are nowadays used to achieve the same or better bandwidth than with the lots of parallel wires in traditional bus links.

As an example, let’s look at hard-drive interfaces: Traditional (now mostly extinct) IDE / parallel ATA achieved up to 800 Mbit/s using 16 single-ended data signals, but only in one direction at a time (half-duplex).

Serial ATA has 2 differential pairs, thus four signal wires, one pair to send, one pair to receive. Each pair can transmit up to 6 Gbit/s, thus in both directions together 12 Gbit/s, more than an order of magnitude faster than its predecessor.

Examples from Consumer Electronics These advantages have led to a clear trend of parallel interfaces being replaced by serial interfaces in many areas. A few examples:

- Parallel printer port → USB
- PCI & AGP → PCI-express
- IDE (Parallel ATA) harddisk interface → SATA (Serial ATA)
- SCSI → SAS (Serial attached SCSI)

2.6.3 Serial AER Conclusion

We thus decided to implement our board to board AER communication links using serial differential signaling to eliminate the need of parallel AER interfaces between boards.

A somewhat similar approach has also been advocated in [Berge 07], however using very high-end Xilinx Virtex series FPGAs.

At that time, these very high-priced FPGAs were the only ones to have integrated dedicated Serializer-Deserializer hardware, which is required to implement a reasonable high-speed serial AER interface.

However this high price would have been incompatible with our requirement that our AER interfacing system should be implementable with simple and affordable components. Using Virtex-class FPGAs (as in [Berge 07]) as a basis of our serial AER approach was clearly not an option, because the cost per board would just be prohibitive, once you think about mass producing a dozen or two of these AER interfacing systems.
We thus had to find another solution, and as you will see in detail in section 4.3, we found one. At a fraction of the cost but with higher performance than the one presented in [Berge 07].

2.7 Conclusion

In this chapter, we introduced the key related work that had the most influence on the AER interfacing solution we are going to present next. In section 2.4 we pointed out which strengths and weaknesses we identified in the related work shown above.

We also discussed the traditional parallel AER interfaces, identified its limitations, and explained why and how we came to the conclusion that we need to develop a novel high-speed serial AER interface.

In the next chapter, we will use all this information to conclude what the design requirements for the AER interfacing platforms built for this thesis are.
Design Requirements and Prototypes of the AEX AER Interfacing Platform

Based on the analysis of related work in the previous chapter, we conclude which design requirements our generic interfacing platform for neuromorphic chips needs to fulfill.

We then present the initial prototypes of this system we call AEX, AMDA EXtension Board, a generic AER interfacing platform, optimized to be operated in combination with the AMDA board presented in 2.3.3.

The hardware and software implementation details of the final version of this system, the AEXv4 will be presented in the two following chapters.

3.1 Design Requirements for the AEX System

Based on all the related work and other material presented so far, we will now formulate our design requirements for the AEX neuromorphic interfacing and processing platform.

These requirements can be divided into three categories: functionality and extensibility, interfacing capabilities and, finally, mechanical and cost constraints we are required to meet in order to create a system feasible for widespread use in our research.
3 Design Requirements and Prototypes of the AEX Platform

3.1.1 Functionality and Extensibility

In table 3.1 we list our requirements in terms of which functionality we must be capable to implement in the FPGA to provide the facilities necessary for our AER processing needs.

- Generous FPGA resources for future expansion of functionality
- Monitoring and sequencing at very high time resolution
- 32 bit address-events used throughout system, with the only exception of the AMDA compatible 16 bit parallel AER connection
- Easily configurable FPGA code to allow users to configure routing, filtering and tagging of address event streams
- FPGA codebase that is easily expandable for more complex operations and processing

Table 3.1: AEX requirements: functionality and extensibility

3.1.2 Interfacing Capabilities

The key requirements of the AEX systems lie within its interfacing capabilities.

Parallel AER Interfacing

Obviously, we need to interface to neuromorphic chips, which is most commonly done through parallel AER interfaces. Table 3.2 describes the requirements we have for this type of interface.

- One 16 bit AER input, optionally extensible to >16 bit
- One 16 bit AER output, optionally extensible to >16 bit
- Ability to be connected directly to the AMDA board without the need for any cables
- Other AER devices can be connected ribbon cables
- Configurable AER protocol type: P2P or SCX
- Configurable REQ and ACK active values (high- or low-active)
- Configurable delays for incoming and outgoing AER signals
- Spare pins for the parallel AER interface, to be used to connect AER devices with more than 16 bit data width

Table 3.2: AEX requirements: Parallel AER
3.1 Design Requirements for the AEX System

Solution for Robust High-Speed Board-to-Board AER Interfacing

We also want to build multi-chip systems involving multiple AEX systems, hence we need to interface between AEX systems. This same high-speed AER interface also has to be capable to be implemented on other, more advanced AER processing systems, such as AER mappers as described in chapter 7. The requirements for that interface are summarized in table 3.3.

- Very high speed AER interface between boards
- Range of $\geq 1$ m
- More robust signalling than ribbon cables: no ground-bound, cross-talk, EM-interference, etc.
- Compact and easily available connectors and cables

Table 3.3: AEX requirements: board-to-board AER

Interfacing to Computers

Finally, in order to observe and record the results of our experiments with neuromorphic chips, we want to interface the AEX systems to computers. The same interface also has to be capable to send data from a computer to an AEX system, e.g. in order to provide a certain predefined or generated stimulus to a neuromorphic chip. The requirements for our AEX to computer interface are summarized in table 3.4.

- USB interface for ease of use
- Highest possible performance on high-speed USB 2.0 should be achieved
- Custom open-source driver for the Linux Kernel to maximize performance

Table 3.4: AEX requirements: interfacing to computers

3.1.3 Mechanical and Cost Constraints

Last but not least, there are mechanical and also cost constraints that affect our design. First and foremost, the system should be as compact and portable as possible. This enables us to create simple
3 Design Requirements and Prototypes of the AEX Platform

demonstrations of a neuromorphic chip involving only a laptop and a compact AEX plus neuromorphic chip.

Also, ease of assembly is an issue. Being able to assemble and also modify the system in-house shortens developments cycles and avoids dependencies on third-parties. This is especially important during prototyping phase. But optimizing ease of assembly also affects the cost of out-source series production later on.

Cost in terms of total component cost a.k.a. BOM (bill of material) cost is also an issue to strongly consider. Demonstrations like the aforementioned Serial AER implementation presented in [Berge 07] depend on very expensive FPGAs, as we described in section 2.6.3.

While this might not be an issue if only a couple devices are required, if dozens of these devices are required by numerous researchers in multiple labs, having a low BOM-cost is a big pro.

Table 3.5 summarizes all those final constraints just mentioned:

- Compact PCB design
- PCB design optimized for easy in-house assembly
- Relatively low component and board cost
- Optimized for use together with the AMDA board

Table 3.5: AEX requirements: mechanical and cost

3.1.4 Design Block Diagram

Figure 3.1 shows the block diagram according to the design goals we have formulated in this section. It shows the key components (green) and connectors (blue) and how they need to be interconnected.
3.1 Design Requirements for the AEX System

Figure 3.1: AEX block diagram according to the design requirements
3 Design Requirements and Prototypes of the AEX Platform

3.2 AEX Prototypes Version 1 & 2

During the work on [Fasnacht 07a], prototypes both for the AER interface hardware as well as software (FPGA codebase and USB drivers) were developed.

3.2.1 AEXv1

The first two prototypes of the AEX system were built during the work on [Fasnacht 07a]. Figure 3.2 shows the first AEX prototype.

PCB Design

The AEXv1 PCB design was implemented by an external contractor. Numerous deficiencies were identified quickly upon assembly and testing of the first prototype. Just one example is that differential impedance calculations for the LVDS traces for the high-speed Serial AER interface were off so much, that it was impossible to get the Serial AER interface to work with the PCB revision AEXv1.

Choice of FPGA and SerDes

The work on these prototypes has demonstrated that both the initial choice of FPGA, a Xilinx Spartan 3 and of the SerDes, used in the first prototypes were not ideal.

The FPGA, a Xilinx Spartan 3, has proven to sometimes be sensitive to power supply ramp-up timing and order. This could cause undefined behaviour after power-up which was a serious problem. Because Xilinx only solved these issues in the successor series Spartan 3E, it was decided, that we needed to switch to this newer FPGA series.

The SerDes used in the prototype, the TLK2501 from Texas Instruments, requires external termination resistors, which was making PCB design challenging. Its successor, the TLK3101 [TIN 08b] became widely available about the time when the work on [Fasnacht 07a] had to be concluded. This newer version was faster than its predecessor, up to 3.125 GHz instead of 2.50 GHz. More importantly though, the termination resistors were now integrated in the chip, which allowed for a
3.2 AEX Prototypes Version 1 & 2

Figure 3.2: AEX version 1 prototype ([Fasnacht 07a])

much more robust signalling. For these reasons it was decided to switch to the newer TLK3101 SerDes. This allowed for increased link quality, higher maximum bandwidth and simpler PCB design. It was decided that the next AEX version should support both the TLK2501 and the upcoming TLK3101.

3.2.2 AEXv2

Still as part of the work on [Fasnacht 07a], a second revision (Figure 3.3) of the PCB was designed. Besides many corrections and improvements to the schematic, PCB layout of the AEXv2 was re-done from scratch, this time by the author himself, rather than by an external contractor as with the AEXv1.

Due to these unplanned circumstances, the work in [Fasnacht 07a] resulted in a very extensive proof-of-concept, but not yet in a system that could be used on an every day basis by other researchers. For further details on the numerous hardware improvements made between the AEXv1 and AEXv2, please refer to [Fasnacht 07a].
3 Design Requirements and Prototypes of the AEX Platform

Based on this prototype and the initial experience gained with the novel Serial AER interface design, [Fasnacht 08] was published describing the AEX platform, with a focus on the novel Serial AER interface.

Figure 3.3: AEX version 2 prototype ([Fasnacht 08])
3.3 AEXv3 – First Production Version

The AEXv3 (figure 3.4) implements only very small but very crucial changes over the AEXv2. The requirements to implement those and produce a new batch of PCBs was identified only when implementing the final FPGA codebase for the AEX and after identifying robustness issues in the Serial AER interface when using the older TLK2501 SerDes.

3.3.1 FPGA Clocking Optimization

Some changes to the clock distribution on the PCB were required since the clock signal distribution and use of a clock signal within an FPGA proved to be more restricted than originally assumed.

Core Clock Global Clock Inputs

It was found that the clock from the external oscillator could be used for only one clock domain (DCM, digital clock manager) if it was present on only one global clock input (GCLK) pin on the FPGA.

Since multiple sections of the FPGA should be able to be clocked by different clocks generated from different DCMs, the external clock needed to be present on more than one GCLK pin.

In the AEXv3 this was fixed by providing that same clock from the external oscillator on two more GCLK pins.

FX2 Interface Clock

To allow for the interface clock between the FPGA and the FX2 USB chip to be produced by either the FPGA or the FX2 chip, this signal also had to be present on a global clock input (GCLK) pin of the FPGA.

This allowed us to shift that clock in the FPGA using a DCM to meet the (very tight) timing constraints of the FX2 chip a lot better.

3.3.2 SerDes Termination Optimization

With the TLK2501 SerDes on the AEXv2 it was found that the Serial AER link did not work reliably. The layout of the termination resistors
required for the TLK2501 (unlike with the TLK3101) was identified as the source of these problems.

In the initial PCB layout, the PCB design of the receiving side of the serial AER interface was as follows ("⇒" signifies a differential LVDS trace):

\[ \text{SATA connector} \Rightarrow \text{termination resistor} \Rightarrow \text{TLK2501 LVDS inputs} \]

This was resolved by moving the termination resistors on the bottom side of the PCB, right behind the SerDes. This allowed for a much better signal to be present at the SerDes LVDS input pins and solved that issue.

\[ \text{SATA connector} \Rightarrow \text{TLK2501 LVDS inputs} \Rightarrow \text{vias to bottom side of the PCB underneath the TLK2501} \Rightarrow \text{termination resistor} \]

(Now placed on the bottom side of the PCB, instead of the top)

### 3.3.3 First Production AEX Version

With these changes implemented, the AEX version 3 became the first fully functional hardware revision of the AEX platform, with the AEXv3 hardware fulfilling all design requirements as previously specified in section 3.1.

Only one change was manually implemented when assembling the PCB: The reset circuits of the FPGA and the FX2 USB interface chip were coupled in order to have only one common reset button: “S2”.

For this, one short piece of coil wire was ran between the two circuits, as can be seen in figure 3.4 on the top side of the PCB. Instead one of the push-buttons could be left unassembled: “S1”.

This change allowed for consistent and predictable reset timings of all circuits on the whole board by eliminating two independent reset-domains.
Figure 3.4: AEX version 3, the first production-grade AEX version, PCB front and back side.
3 Design Requirements and Prototypes of the AEX Platform

3.4 Conclusion

In this section we established the complete design requirements for the AEX interfacing platform. Then we described the first two prototypes of the AEX system, AEXv1 & AEXv2, as well as the first completely operational release, the AEXv3. In the next two chapters we will go into the details of the final AEX system developed as part of this thesis, the AEXv4 (AEX version 4):

Chapter 4 will present the details of the AEXv4 hardware, including all its hardware interfaces and VHDL modules implementing the FPGA parts of those interfaces.

Chapter 5 will present the generic FPGA framework for AER interfacing, filtering and processing, written for the AEXv4 and compatible systems introduced in later chapters.

Chapter 6 will present the software details of the USB interface, i.e. the FX2 firmware, the “aerfx2” kernel drivers and user-space software developed to connect the AEX platform to PCs.
This chapter focuses on the details of the AEXv4 interfacing and processing platform for neuromorphic chips. The related work, motivation and the design requirements as well as a description of the various prototype stages of the AEX system were described in the previous chapter.

First we will give a general overview of the AEXv4, its architecture and the principal components involved.

Then we’ll focus on the interfaces of that system: the parallel AER interface to attach neuromorphic chips, the novel serial AER interface that was introduced with this system in order to connect multiple AEXv4 systems into a multi-chip experimental setup and to attach more advanced AER processing systems such as mappers, finally we’ll describe the USB interface of the AEXv4, used to connect to a PC in order to monitor/sequence AER streams.

Along with the interface hardware, we’ll also give an overview of the VHDL entities, which form the part of the interface implemented in the FPGA, and present these interfaces to the AER routing and processing parts within the FPGA.

Finally, the chapter is concluded with some benchmark measurements to demonstrate the interface performance, and a spec sheet of the AEXv4 interfacing capabilities.

Software aspects in relation to the AEX system presented here, i.e. the generic FPGA codebase for AER routing and processing written for the AEX system will be presented in chapter 5.
The USB interface firmware, kernel driver and user-space code to
connect the AEX system to a computer or laptop will be presented in
chapter 6.

4.1 AEXv4 Platform Overview

To give a comprehensive overview of the AEXv4 system, we start by
discussing the changes of the AEXv4 design compared to its predecessor
AEXv3, we’ll describe the system in a top down way, starting with the
top level block diagram and the PCB, then explaining the components
and interfaces.

4.1.1 Changes compared to AEXv3

AEX Hardware  The most prominent change between the AEXv3 and
the AEXv4 is certainly the drastic reduction in size. The AEXv3
measures 135 mm * 105 mm. The AEXv4 on the other hand only
measures 107 mm * 75 mm.

This could be achieved by the combination a number of factors:
The complete power section was redesigned from scratch, using more
compact components. Some connectors were replaced by smaller (yet
fully compatible) versions, e.g. the USB and the JTAG connector.

The level shifters present in all previous versions of the AEX were
removed. This after determining, that the FPGA I/O of the “Spartan
3E” used were robust enough, to handle 5 V AER input if necessary.
Also most newer devices were using 3.3 V AER instead of 5 V anyways.

Last but not least, the PCB placement and layout of the AEXv4 was
done from scratch with a focus on getting the PCB as small as possible.

Where possible, it was attempted to keep the AEXv4 “electrically
equivalent” to the AEXv3, in order to make sure that people who
already used customized FPGA configurations for their AEXv3 could
migrate to the AEXv4 with little to no modifications.

In order to make the assembly of the AEXv4 easier, many components
present on the bottom side of previous AEX PCB revisions, were now
placed on the top side.
AEX Software  Even bigger were the changes on the AEX Software front. Here almost everything was reimplemented from scratch eventually, based on the lessons learned from previous versions.

The FPGA codebase, interface code and AER routing/processing code, was reimplemented completely.

Also the USB interface parts were updated. The kernel driver was updated to work with the latest versions of the Linux Kernel, and a new set of user-space reference tools to test the system was implemented.

All these software aspects of the AEX platform will be discussed in the next chapter.

4.1.2 Board-Level Block Diagram

Figure 4.1 shows the detailed block diagram of the AEXv4 system. Interface connectors, parallel AER, serial AER, USB and others are drawn in blue (rectangular boxes). Chips are drawn in green (rounded boxes).

In the center of everything is the FPGA, this part of the diagram will be shown again later in figure 4.6 in more detail when the FPGA internals of the interface implementations are discussed.
Figure 4.1: AEXv4 block diagram with FPGA implementation details
4.1.3 AEXv4 PCB and Design for Assembly

Figure 4.2 shows the top side of the AEXv4 PCB, figure 4.3 shows the bottom side.

It can be clearly seen, that in an effort to design for assembly, the component count on the bottom side was kept to an absolute minimum, unlike in previous versions of the AEX PCB.

This obviously makes it easier to manually assemble the AEXv4, since almost all components can be soldered on without turning the board upside down.

However for industrial assembly, this becomes much more important. Here soldering is usually done in a reflow-soldering process. The pick-and-place process has to be performed twice, once for each side. Also, in order to keep bottom side components in place during soldering, the bottom side components would have to be glued on in a separate processing step, to prevent them from falling off during reflow-soldering.

With having only that few components on the bottom side, a much cheaper solution for small series production is possible:
The top side components are picked-and-placed automatically and are the reflow-soldered.

The few bottom side components are then hand-soldered after the automated assembly of the top side.

For this to work though, it is crucial, that the contact and pad finishing of the PCB is chemical gold plating, not the more common and much cheaper chemical tin plating.

Tin plated pads and contacts would completely oxidize during the reflow-soldering step, which would make it impossible to hand-solder the bottom side components in the final assembly step.

4.1.4 Components and Interfaces

Tables 4.1 and 4.2 list the components and interface connectors used in the AEXv4.
4.1 AEXv4 Platform Overview

<table>
<thead>
<tr>
<th>AEX (v4) – Principal Components / Chips</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FPGA</strong></td>
</tr>
<tr>
<td><strong>USB Interface Chip</strong></td>
</tr>
<tr>
<td><strong>Serial AER Interface Chip</strong></td>
</tr>
<tr>
<td><strong>Other Active Components</strong></td>
</tr>
</tbody>
</table>

Table 4.1: AEXv4 – key components / chips

<table>
<thead>
<tr>
<th>AEX (v4) – Interfaces / Connectors</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parallel AER In</strong></td>
</tr>
<tr>
<td><strong>Parallel AER Out</strong></td>
</tr>
<tr>
<td><strong>Serial AER I/O</strong></td>
</tr>
<tr>
<td><strong>USB</strong></td>
</tr>
<tr>
<td><strong>Extra I/O</strong></td>
</tr>
<tr>
<td><strong>JTAG</strong></td>
</tr>
<tr>
<td><strong>Power</strong></td>
</tr>
</tbody>
</table>

Table 4.2: AEXv4 – interfaces and connectors
4.1.5 AEXv4 and AMDA Platform Joined

Figure 4.4: AEXv4 and AMDA board combined

Figure 4.4 shows the AEXv4 connected directly to an AMDA board. This is the typical configuration in which the AEX board is used to build an experimental setup with a neuromorphic chip. A TMS1 chip from the eMorph project is mounted on the AMDA board in this picture.

Figure 4.5 shows the AEXv4 with an overlay identifying the key components and connectors.
4.1.6 Interfaces Overview

In the next sections, we describe the three interfaces, that the AEX system implements. On one hand there are the Parallel AER input and output interfaces and the novel high-speed Serial AER input and output interface which transceive address-events “just-in-time”, i.e. as fast as possible and without any timestamps, on the other hand, the USB interface, which transmits timestamped address-events between a computer and the monitor and sequencer units implemented in the FPGA on the AEX.

- The Parallel AER interface is described in section 4.2,
- the 3 GHz Serial AER interface is described in section 4.3,
- the high-speed USB 2.0 interface is described in section 4.4.

We have previously seen the top level block diagram of the AEXv4 in figure 4.1. The FPGA internal block diagram showing the FPGA implementation of all the interfaces is enlarged in figure 4.6.

In the FPGA block diagram, the interface implementation entities are drawn in orange, FIFO buffers in striped blue. The internal entities between the USB interface and the core, the monitor, sequencer and timestamp counter are drawn in green.

The generic AER routing fabric at the core of figure 4.6 is left as a black box since we are going to focus on the hardware and interfaces in
this chapter.

This black box however will be opened up in the next chapter, in figure 5.1.

Figure 4.6: AEX FPGA block diagram (FPGA top-level, showing interface implementation details).
4.2 Parallel AER Interfaces Implementation Details

The parallel AER input and output interfaces are used to connect directly to neuromorphic chips. As we have shown in figure 4.4 this is typically done by attaching the AEX board parallel AER connectors directly to an AMDA board.

Alternatively, the AEX parallel AER interfaces can also be connected to other hardware via ribbon cables.

The pinout of the interface is the “Rome-type” parallel AER connector, which has been described in table 2.6.

The interface uses 3.3V signalling, but the FPGA inputs are robust enough, to sustain any current limited 5V AER signal, such as those present on the AMDA board.

The FPGA can be configured for both active-high and active-low handshake signals, and it supports both common handshake schemes use in AER interfaces:

- P2P-type, as presented in figure 2.11
- SCX-type, as presented in figure 2.12

In addition to the type of handshake scheme used, the parallel AER interface implementation supports many ways to tweak the interface timing, e.g. the option to delay handshake signals by an arbitrary number of FPGA clock cycles.

4.2.1 Parallel AER Interface FPGA Implementation

The top level entities implementing the parallel AER interfaces are: SimplePAEROutputRR for the parallel AER output and SimplePAERInputRRv2 for the parallel AER input.

We will now present the VHDL entity declarations of these entities and explain how they are being connected to and used.

SimplePAEROutputRR

Listing 4.1 shows the entity declaration of SimplePAEROutputRR. This entity corresponds to the block “PAER Tx Interface” including the adjacent FIFO block in figure 4.6.
As virtually all VHDL entities we are going to present, SimplePAEROutputRR needs a clock and reset signal: ClkxCI and RstxRBI.

VHDL signal naming conventions being used are explained in 0.1.2.

The parallel AER output connector pins of the AEX are directly connected to the entity signals AerAckxAI and AerReqxSO (AER handshake) and AerDataxDO AER data lines.

The VHDL generics of this entity all have sensible default values and usually don’t need to be changed.

To select whether to use high- or low-active handshake signals, AerReqActiveLevelxDI and AerAckActiveLevelxDI need to be wired up to the required active level. These are implemented as VHDL signals, not VHDL generics, in order to be able to change this, without rebuilding the FPGA code. This could even be done e.g. via a jumper or switch connected to a free GPIO pin on the AEX.

The signals InpDataxDI, InpSrcRdyxSI and InpDstRdyxSO are used to connect this interface in the generic AER routing fabric.

Listing 4.1: VHDL entity declaration of SimplePAEROutputRR

```vhdl
entity SimplePAEROutputRR is
  generic (  
    paer_width : positive := 16;  
    internal_width : positive := 32;  
    ack_stable_cycles : natural := 2;  
    req_delay_cycles : natural := 4;  
    output_fifo_depth : positive := 1  
  );
  port (  
    -- clk rst  
    ClkxCI : in std_ulogic;  
    RstxRBI : in std_ulogic;  

    -- parallel AER  
    AerAckxAI : in std_ulogic;  
    AerReqxSO : out std_ulogic;  
    AerDataxDO : out std_ulogic_vector(paer_width-1 downto 0);  

    -- configuration  
    AerReqActiveLevelxDI : in std_ulogic;  
    AerAckActiveLevelxDI : in std_ulogic;  

    -- output  
    InpDataxDI : in std_ulogic_vector(internal_width-1 downto 0);  
    InpSrcRdyxSI : in std_ulogic;  
    InpDstRdyxSO : out std_ulogic  
  );
end SimplePAEROutputRR;
```
SimplePAERInputRRv2

Listing 4.2 shows the entity declaration of SimplePAERInputRRv2. This entity corresponds to the block “PAER Rx Interface” including the adjacent FIFO block in figure 4.6.

The generics in this entity are explained in the comments in the code listing. The most important one is data_on_req_release, which needs to be changed to true if the interface should operate in SCX mode.

The parallel AER input connector pins of the AEX are directly connected to the entity signals AerReqxAI and AerAckxSO (AER handshake) and AerDataxADI AER data lines.

The signals OutDataxDO, OutSrcRdyxSO and OutDstRdyxSI are used to connect this interface in the generic AER routing fabric.

AerHighBitsxDI defines the values of the higher-order bits. The parallel AER input of the AEX has 16 data lines, but in the FPGA, address-events are represented as 32 bit values. This signal defines the values of the higher-order 16 bits which need to be attached to an incoming 16 bit address-event.

All other signals are equivalent to the ones in the previously presented entity.

Listing 4.2: VHDL entity declaration of SimplePAERInputRRv2

```vhdl
-- req_stable_cycles:
-- this is the number of cycles the level on req has to be stable
-- in order for a value change to be detected (and not interpreted
-- as a possible glitch)
--
-- data_on_req_release:
-- if false, data is sampled as soon as a valid assertion of req
-- is detected.
-- if true, data is only sampled as soon as req is detected to be
deasserted again, i.e. in SCX protocol

entity SimplePAERInputRRv2 is
    generic (
        paer_width : positive := 16;
        internal_width : positive := 32;
        req_stable_cycles : positive := 2;
        data_on_req_release : boolean := false;
        input_fifo_depth : positive := 1
    );
    port (
        ClkxCI : in std_ulogic;
        RstxRBI : in std_ulogic;
    );
```
-- parallel AER
AerReqxAI : in std_ulogic;
AerAckxSO : out std_ulogic;
AerDataxADI : in std_logic_vector(paer_width-1 downto 0);

-- configuration
AerHighBitsxDI : in std_logic_vector(internal_width-1-paer_width downto 0);
AerReqActiveLevelxDI : in std_logic;
AerAckActiveLevelxDI : in std_logic;

-- output
OutDataxDO : out std_logic_vector(internal_width-1 downto 0);
OutSrcRdyxSO : out std_logic;
OutDstRdyxSI : in std_logic;
end SimplePAERInputRRv2;
4.3 3 GHz Serial AER Interface Implementation Details

As we have already discussed in section 2.6, there is the need to have robust high-speed board-to-board AER interfaces. In this section we describe the details of the solution we chose to implement to solve this problem, a LVDS based serial AER interface, capable of running at speeds of about 3 GHz.

4.3.1 LVDS

Traditional signalling in digital logic designs is very simple: One wire is run from a source (driver) to a destination (buffer, register), both of which have a common ground (GND) potential. A high voltage, traditionally 5 V, signals a value of “1”, a low voltage of 0 V signals a value of “0”.

In LVDS, things are more complicated. Two signalling wires are required between the source and the destination. The source drives two wires such that they always have the opposite values of each other. The destination then compares the voltages of these two wires. Let’s call these two wires P and N, and the voltages on them \( V_P \) and \( V_N \).

Then, the receiver, by comparing the two voltages, interprets the condition \( V_P > V_N \) as a value of “1”, and correspondingly \( V_P < V_N \) as a value of “0”.

Another important factor in LVDS signalling, is that the impedance of the driver, the connecting transmission line, and the termination impedance at the receiving buffer are all matched.

The driver drives each of the two corresponding signal lines with an impedance of \( Z_0 = 50 \Omega \), the two corresponding lines have a differential impedance of \( Z_{\text{diff}} = 100 \Omega \) and finally the termination resistor, or resistor network, has a termination impedance of again \( Z_{\text{diff}} = 100 \Omega \).

Important is, that all parts involved have exactly matched impedance: driver, PCB traces, connectors, cables, termination. If this is achieved, we have a perfect transmission line, which avoids any reflections of the signal. Only then is it possible, to transmit signals in the Gigahertz range.

As an example, let’s assume we have a differential signalling cable of one meter length, and send bit over it at a rate of 2.5 GHz.
On a differential signalling cable made of copper, the signal travels at a speed of about 0.5c. We can calculate the “length” of a bit to be:

\[
L_{\text{bit}} = \frac{0.5 \times c}{f} = \frac{0.5 \times 3 \times 10^8 \text{ m/s}}{2.5 \times 1 \text{ s}} = 0.06 \text{ m}
\]

This means that we have almost 17 bits traveling on our one meter cable at the same time. Any significant reflections caused by impedance mismatch would blend bits into each other and prevent successful communication.

To learn more about the details and various implementations of LVDS-type signalling, the “LVDS Owner’s Manual” by Texas Instruments (formerly by National Semiconductors) is an excellent resource: [TIN 08a].

**AC Coupling versus DC Coupling**

If the differential driver and the differential receiver are directly connected, one speaks of a DC coupled link (Figure 4.7, adapted from [TIN 07]).

If decoupling capacitors are put into the data lines, one speaks of an AC coupled differential link (Figure 4.8, adapted from [TIN 07]). These capacitors avoid DC currents from flowing over the data cables.

AC coupled links have the advantage that they remove the requirement of a common ground reference on both the transmit and receive side altogether. Although we would not have any ground bounce problems due to the differential signaling anyways, this can be a big advantage in bigger setups. If we have big setups with lots of boards, we may also have loops in them that have an area in the order of square meters.
4.3 3 GHz Serial AER Interface Implementation Details

These loops would act as an antenna for example for line frequency noise at 50Hz.

Because there are only capacitively coupled links between our boards when we use AC coupling, we eliminate this problem too. Thus, no conducting loops are possible in our setups.

The only disadvantage of AC coupling is that one must somehow ensure that the same amount of “1” and “0” bits are transmitted. Typically, this is done using a code such as 8b/10b-coding (8 bit / 10 bit coding), which produces a 10 bit word with the same amount of “1” and “0” bits in it for every possible 8 bit word.

This encoding is typically already implemented in a serializer part of an LVDS chip, and the decoding in the deserializer part, as explained in the next.

4.3.2 Texas Instruments TLK2501 & TLK3101

Serializer–Deserializer

To implement the Serial AER interface we needed a special SerDes chip, a serializer–deserializer. This was the solution to our problem described earlier in section 2.6.3, that the only FPGAs including dedicated serializer-deserializer hardware, were just way too expensive, to be a candidate for our serial AER interfacing solution. Choosing an affordable dedicated high-performance SerDes chip as the TLK2501 or TLK3101 in combination with a very affordable Xilinx Spartan 3/3E series FPGA, we could achieve higher performance than [Berge 07] at a fraction of the cost.
The serializer part gets its data over a parallel bus from the FPGA, in our application at a speed of 125 MHz at a bus width of 16 bit. It then encodes each 16 bit work using 8b/10b-coding (twice) to encode it into a 20 bit word, and outputs these 20 bit over a serial differential link at 20 times the bus clock rate, thus at a rate of 2.5 GHz.

The deserializer does the exact opposite, receiving the 2.5 GHz signal, recovering its clock, checking for errors while decoding the 8b/10b-coding and then outputs the data over another 16 bit bus running at 125 MHz to the FPGA.

Figure 4.9, adapted from [TIN 08b], shows the block diagram of the TLK3101, the SerDes chip we have chosen for this application. The chip includes both a serializer and a deserializer. The outgoing data-path involving the serializer is marked with a red arrow. It transmits AER data to another board over a cable. The incoming data path is marked with a blue arrow. Here the deserializer receives an AER stream from another board and decodes it for further processing in the local AEX FPGA.

As can be seen in the block diagram (Fig. 4.9, the transmit data path (serializer, green arrow) and the receive data path (deserializer, blue arrow) are almost completely independent (separated by the dashed line). The only thing they share is that the transmit clock (GTX_CLK) is used as a reference clock for the receiver clock recovery circuitry, and test structures and test data paths, which are disabled during regular operation.

The TLK2501 and its successor the TLK3101 are identical for the most part, with the exception of two differences relevant for our purposes:

- The TLK2501 can sustain bit-rates up to a maximum frequency of 2.5 GHz. The TLK3101 however can operate at bit-rates up to 3.125 GHz (but can also be clocked to 2.5 GHz and thus “talk” to a TLK2501.
- The TLK2501 requires external termination resistors. As explained in the previous chapter, this is very challenging to get right when doing the PCB layout. The TLK3101 however has built-in termination resistors, which makes PCB layout a lot simpler and the serial link more robust.

While we have modified AEX boards and successfully tested Serial AER at bit-rates of 3.125 GHz, basically all the AEX boards used in research setups are using 2.5 GHz Serial AER, in order to be compatible, no matter whether equipped with a TLK2501 or the newer TLK3101.
Figure 4.9: TI SerDes block diagram and data-path
(adapted from [TIN 08b])
4.3.3 Cables & Wiring: Serial ATA

We chose to use Serial ATA cables (Figure 4.10) to transmit and receive data to and from other boards. These cables are a perfect match for use with the SerDes chips, and they are also very cheaply available in every computer store.

The cables are also quite well suited for building experimental setups, as they are a lot less bulky, compared to the ribbon cables used for example in multi-chip setups of the CAVIAR project.

Normal Serial ATA cables are available at length up to 1 m, eSATA cables up to 2 m. SATA uses bit-rates up to 6 GHz, which exceeds our requirements by about a factor of two.

4.3.4 Flow Control

In Parallel AER, stopping to handshake with the sender allows the receiver to throttle the sender in case he is not ready to process an event, and thus to avoid that events are dropped.

We also needed something similar in our serial AER implementation. We decided to use the second differential pair available in the SATA cables for that purpose.

The first wire connects from the transmitting SerDes to the receiving SerDes, the second pair connects the receiving FPGA to the transmitting FPGA, thus forming a back-channel. Using this channel the receiving FPGA can tell the sender that it is not ready to process any further data. The sender can then stop, and resume as soon as the receiving FPGA signals that it can process data again.

We chose to use AC coupling in this back channel too. If not, using AC coupling in the data links would have been useless when we create a DC link on the flow-control back-channel.

This also means that our flow control signal needs to be DC free, in order not to build up charge in the coupling capacitors. We defined the flow control signal to be a rectangular oscillation. The meaning lies in the frequency. If the receiver sends a clock of half the 125 MHz, i.e. 62.5 MHz. This tells the sender that it can send data.

If the receiver wants to signal “Stop” to the sender, it sends a frequency of an eighth of the 125 MHz clock.
Figure 4.10: Serial ATA cable and connectors. In the lower half, the cable is cut open. Left to right: four solid ground wires, red isolation, one differential pair with shielding foil intact, red isolation, the other differential pair with wires separated and shielding foil partially removed. (adapted from [Fasnacht 07a])
These two frequencies can be very easily distinguished using a simple state machine in the sender FPGA, even though the edges of the signal are not synchronous to any clock in the sender FPGA.

The sender FPGA can count for how long the flow control signal keeps its value. If this count is one to three 125 MHz cycles, it means ready to send, if it is more, the sender has to stop.

With the sender FPGA sampling the flow control signal at 125 MHz, the 62.5 MHz we use can also be derived from the Nyquist–Shannon Sampling Theorem. It is the so-called Nyquist frequency equal to half the sampling frequency, the maximum frequency that can be recovered by the sender FPGA.

### 4.3.5 Serial AER Pinout

The pinout we decided to use for our Serial AER implementation on the SATA connector is documented in table 4.3.

<table>
<thead>
<tr>
<th>3 GHz Serial AER Pinout</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GND</td>
</tr>
<tr>
<td>2</td>
<td>SAER+</td>
</tr>
<tr>
<td>3</td>
<td>SAER+</td>
</tr>
<tr>
<td>4</td>
<td>GND</td>
</tr>
<tr>
<td>5</td>
<td>FLOW+</td>
</tr>
<tr>
<td>6</td>
<td>FLOW-</td>
</tr>
<tr>
<td>7</td>
<td>GND</td>
</tr>
</tbody>
</table>

Table 4.3: Pinout of the “3 GHz Serial AER Interface”, pin numbers according to SATA specification.
4.3.6 Impedance Matched PCB Layout

Figure 4.11 shows the top layer of a PCB section implementing the serial AER interface discussed. Pads with drill holes and vias connecting to the bottom side or an inner layer of the PCB are drawn in gray. Traces and pads where SMD components are soldered onto are drawn in yellow. The high-frequency serial AER traces with matched length and impedance are high-lighted in red.

Center / top is where the SerDes chip is located, centrally over the big gray square, which serves as a heat-sink pad for the SerDes chip. Located at the bottom of the figure are the two SATA connectors. The Serial AER input on the left, the output on the right.

The traces of the flow control lines also present on these two connectors are invisible, since they connect to the SATA connectors on the bottom side of the PCB (not shown here). The through-hole pins on the SATA connector used for the flow control lines are highlighted in purple.

The red traces are the differential signalling traces. The trace endpoints on the top side are highlighted in blue. In the lower part of the figure, these are the through-hole pins (blue) on the SATA connector.
carrying the Serial AER LVDS signals, in the upper half, the traces end
at vias (blue) which connect to the bottom side of the PCB.

Where the red differential traces are interrupted near the SATA
connectors, is where the AC coupling capacitors (green) are soldered on
to connect them.

It can be seen that the distance between the corresponding traces
varies quite a bit, and that they are much wider apart, than they are
wide themselves. This is because the traces are dimensioned to couple
mostly to the underlying solid copper plane with an impedance of
$Z_0 = 50 \Omega$ each, and not directly between each other.

One can also note that the traces have no sharp turns. At such high
frequencies, especially right-angle turns might cause reflections.

Also it is notable, that the two red traces nearer to the middle of
the figure, seem to take “detours”, while the outer two do not. This is
to ensure, each pair of traces have exactly the same length. Mismatch
in diff-trace length would cause a considerable smear in the resulting
differential signal edges seen by the receiver.

After the traces contact the pads where the SerDes chip is soldered
onto, they connect to four vias (blue). These connect to the bottom
side of the PCB (not shown), where in case of the TLK2501 SerDes,
the termination resistors have to be assembled, while for the TLK3101,
due to the termination resistors present within the chip, no components
must be assembled on the bottom side.
4.3.7 FPGA Implementation: Serial AER Interface VHDL

Finally, let’s have a look at the FPGA parts of the implementation of the serial AER interface. We’ll do this in a bottom-up order.

First we’ll look at the VHDL entity talking directly to the TLK2501 or TLK3101 SerDes, which is called **TLKiface**. This unit also implements the flow-control protocol described above in 4.3.4. When we refer back to figure 4.1, the entity **TLKiface** corresponds to the two orange blocks labelled “SerDes Rx Interface” and “SerDes Tx Interface”.

The second VHDL entity we’re going to describe here is **SAER3GhzRR**. This entity contains within the mentioned **TLKiface**, as well as the adjacent FIFOs (the two blue blocks in figure 4.1, next to the SerDes blocks), and presents the serial AER interface to the “generic AER routing fabric” block.

**TLKiface**

Listing 4.3 shows the entity declaration of **TLKiface**. At the very beginning, there is a comment section, where the delay of the flow control back-channel was calculated, based on the delays in the logic that generates and analyzes the flow-control signal, the delay inherent to the TLK SerDes and the delay of the cable.

From this we know how much space we must have left in the receive-FIFO when the receiving end starts signalling “Stop” over the flow-control back-channel.

Let’s discuss the most important signals of the entity. There are two clock domains, and thus two clocks. **TxClkxCIC** is the system master clock, which is also fed into the **GTC_CLK** pin of the SerDes. All signals of the serializer half of the SerDes are synchronous to this clock. **RxClkxCIC** is the clock recovered by the deserializer from the incoming serial data stream, and comes from the SerDes pin **RX_CLK**. All deserializer part signals are synchronous to this clock.

The 16 bit data bus **PinsTxDxD0** and the two signals **PinTxEnxS0** and **PinTxErxS0** are used to send data from the FPGA to the serializer part.

The 16 bit data bus **PinsRxDxDI** and the two signals **PinRxDvxSI** and **PinRxErxSI** are used to receive data from the deserializer part into the FPGA.

The flow-control back-channel signal is generated on **RxFlowStopxAO** and received on **TxFlowStopxAI**.
All signals starting with TxFifo are connecting to the FIFO containing AER data to be sent out via the SerDes, signals starting with RxFifo feed received data into the FIFO for inbound data from the SerDes.

The FIFO for inbound data is configured to assert RxFifoProgFullxSI early enough, according to the back-channel delay calculations presented at the top of the listing.

The remaining signals are either for test purposes or to signal link activity to the user (via LED).

Listing 4.3: VHDL entity declaration of TLKiface

```vhdl
library ieee;
use ieee.numeric_std.all;
use ieee.std_logic_1164.all;

entity TLKiface is
  port (  
    -- tx and rx clock, reset  
    TxClkxCI : in std_ulogic;  
    RxClkxCI : in std_ulogic;  
    RstxRBI : in std_ulogic;  
    -- status  
    StatusTxActivexSO : out std_ulogic;  -- txclk dom  
    StatusRxActivexSO : out std_ulogic;  -- rxclk dom  
  )
  -- tlk control pins - txclk domain  
  PinPrbsEnxSO : out std_ulogic;  
  PinLoopEnxSO : out std_ulogic;  
  PinTxEnxSO : out std_ulogic;  
  PinTxErrxSO : out std_ulogic;  
  -- tlk control pins - rxclk domain  
  PinRxDvxSI : in std_ulogic;
end TLKiface;
```

**4.3 3 GHz Serial AER Interface Implementation Details**

```vhdl
PinRxErxSI : in std_ulogic;
--
-- tlk data pins
PinsTxDDO : out std_ulogic_vector(15 downto 0);
  -- txclk domain
PinsRxDDI : in std_ulogic_vector(15 downto 0);
  -- rxclk domain
--
-- serial AER handshake
RxFlowStopxAO : out std_ulogic;
TxFlowStopxAI : in std_ulogic;
--
-- rx FIFO - rxclk domain
RxFifoDataxDI : in std_ulogic_vector(31 downto 0);
RxFifoWriteSO : out std_ulogic;
RxFifoProgFullSI : in std_ulogic;
--
-- tx FIFO - txclk domain
-- has to be FWFT style!
TxFifoDataxDI : in std_ulogic_vector(31 downto 0);
TxFifoReadSO : out std_ulogic;
TxFifoEmptySI : in std_ulogic;
--
TestEnableTxFloodSI : in std_ulogic;
--
DebugSO : out std_ulogic_vector(7 downto 0);
);
--
-- entity port attributes for IOP registers:
------------------------------------------------------------------
attribute iob : string;
--
attribute iob of PinPrbsEnxSO : signal is "FORCE";
attribute iob of PinLoopEnxSO : signal is "FORCE";
attribute iob of PinTxEnxSO : signal is "FORCE";
attribute iob of PinTxErxSO : signal is "FORCE";
attribute iob of PinRxDxSI : signal is "FORCE";
attribute iob of PinsTxDxDO : signal is "FORCE";
attribute iob of PinsRxDxDI : signal is "FORCE";
end TLKiface;
------------------------------------------------------------------
```

**Forced IOB-Register Placement**

The `attribute iob of ... : signal is "FORCE";` lines towards the end of listing 4.3 instruct the “map & place” part of the FPGA build process to ensure, that the input/output registers of the signals tagged that way, to be placed into the IO pad registers of the FPGA, or to abort the “map & place” process, if this is impossible to achieve for some reason. This ensures that the most timing critical signals are always registered right at the pad, and not some slice register somewhere within the FPGA.
4 AEXv4 – Hardware Implementation Details

**SAER3GhzRR**

Listing 4.4 shows the entity declaration of **SAER3GhzRR**. This entity merely encapsulates the just presented entity **TLKiface** as well as the two FIFOs for data inbound and outbound of the serial AER interface.

It presents the serial AER interface to the generic AER routing fabric via the following signals: **SaerTxDataxDI**, **SerTxSrcRdyxSI** and **SaerTxDstRdyxSO** are used for the outbound data, from the routing fabric to the serial interface. **SaerRxDataxDO**, **SerRxSrcRdyxSO** and **SaerRxDstRdyxSI** are used for the inbound data, from the serial interface to the routing fabric.

**Listing 4.4: VHDL entity declaration of **SAER3GhzRR****

```vhdl
entity SAER3GhzRR is
  port (
    SaerTxClkPinxCI : in std_ulogic;
    SaerRxClkPinxCI : in std_ulogic;
    --
    SaerTxClkxCO : out std_ulogic;
    SaerRxClkxCO : out std_ulogic;
    --
    SaerRxFifoReadClkxCI : in std_ulogic;
    SaerTxFifoWriteClkxCI : in std_ulogic;
    --
    SaerResetxRBI : in std_ulogic;
    --
    TlkPinLckRefxSBO : out std_ulogic;
    TlkPinPrbsEnxSO : out std_ulogic;
    TlkPinLoopEnxSO : out std_ulogic;
    TlkPinEnablexSO : out std_ulogic;
    TlkPinTxErxSO : out std_ulogic;
    TlkPinRxDvxSI : in std_ulogic_vector(15 downto 0);
    TlkPinsTxdxDO : out std_ulogic_vector(15 downto 0);
    --
    SaerRxStopxAO : out std_ulogic;
    SaerTxStopxAI : in std_ulogic;
    --
    SaerRxDataxDO : out std_ulogic_vector(31 downto 0);
    SaerRxSrcRdyxSO : out std_ulogic;
    SaerRxDstRdyxSI : in std_ulogic;
    --
    SaerTxDataxDI : in std_ulogic_vector(31 downto 0);
    SaerTxSrcRdyxSI : in std_ulogic;
    SaerTxDstRdyxSO : out std_ulogic;
    --
    StatusTxActivexSO : out std_ulogic;
    StatusRxActivexSO : out std_ulogic;
    --
    TestEnableTxFloodxSI : in std_ulogic;
    TestEnableSaerLoopbackxSI : in std_ulogic;
    SaerDebugxDO : out std_ulogic_vector(7 downto 0)
  );
end SAER3GhzRR;
```
4.4 USB 2.0 Interface Implementation Details

4.4.1 USB Interface

The USB interface is used to connect the AEX to a computer and send timestamped data back and forth between the Monitor & Sequencer in the FPGA and the Computer.

Cypress FX2 – CY7C68013A

The chip chosen for implementing the USB interface is the well-known Cypress FX2, more exact the second revision FX2LP, part number CY7C68013A in its 56-pin SSOP variant. The names FX2 and FX2LP are used interchangeably, because there is no functional difference between the two versions of the chip (except for their power consumption).

As seen in its block diagram in figure 4.12, the chip is quite complex and consists of:

- 8051-type microprocessor core
- USB2.0 high speed transceiver and hardware implemented USB endpoint logic, the SIE (Serial Interface Engine)
- 4 KiB of FIFO RAM to buffer data between the USB SIE and the 16 bit peripheral interface
- GPIF (general purpose interface) controller (unused in our design)

One reason to use this device was also that there is quite a lot of a know-how available, because this device was e.g. also used in the USBAER-mini2 board, Tobi Delbruecks recent retina designs [Lichtsteiner 06], and a project the author was previously involved in at the ETHZ Computer Graphics Lab [Weyrich 07].

The most important reason though, was the the FX2 was the only device available on the market, that was sufficiently customizable for our purposes, while achieving the maximum speed possible over a high-speed USB 2.0 interface.

Our measurements using the FX2 could show that it can transmit or receive data at speeds ranging between 40 MBytes/s and 41 MBytes/s. The actual value measures depended not on variations inherent to the FX2 interface chip, but they depended on the motherboard used to perform the speed test with. This clearly indicated that it was the USB
Figure 4.12: Cypress FX2 – USB interface chip block diagram (adapted from [CY 15])

Figure 4.13: Cypress FX2 – simplified block diagram, independence of data-path & CPU are high-lighted.
host controller implementation of the motherboard chip-set that caused these variations.

The closest competitor to the Cypress FX2 was the FTDI FT2232H high-speed USB UART chips. In our tests we could only achieve bandwidths of less than about 25 MBytes/s using this chip. Colleagues from the CAVIAR project confirmed this observation, and we thus settled to use the Cypress FX2 for our USB interface implementation.

One of the most important features of this chip is that the 8051 core is not involved in the data stream, unless explicitly configured to be. This means that the firmware running in the 8051 has nothing to do except to set up the configuration registers and thus configure the chips peripheral interfaces, FIFOs and USB SIE.

Then the data path only involves the USB SIE (USB Serial Interface Engine), the FIFOs, and the peripheral interface. This is shown in figure 4.13 where the the CPU and its RAM are highlighted in green and blue, while all the components involved in the data-path are highlighted orange. Data flowing from USB to/from the FPGA never crosses the dashed line separating the data-path sections from the rest of the chip. (Figures 4.12 and 4.13 are adapted from [CY 15]).

\textbf{FX2 $\rightarrow$ FPGA interface}

The interface connecting the FX2 with the FPGA is configured in the "Slave FIFO" mode in the FX2. Its width is configured to 8 bit and synchronous clocking. It has to be operated in half-duplex mode, which makes the interface implementation in the FPGA much more complex.

FPGA thus is the bus-master controls the read-enable and write-enable lines, and can supply an interface clock up to 48 MHz fast. This gives a maximum bandwidth of 48 MBytes/s between the FPGA and the FX2. This is more than we ever could transfer between a FX2 and a computer which is a little more than 40 MBytes/s.

\textbf{FX2 Firmware}

As mentioned before, the 8051 is not involved in the data transfer between USB and the FPGA. It only configures the FX2. This reduces the firmware to an init routine that sets up registers, plus a main loop doing nothing.
All the firmware was written from scratch without the need for any frameworks available from Cypress and others. It was compiled using the open-source “Small Device C Compiler”, SDCC, from [SDCC 07].

Details on the FX2 firmware will be presented in 6.1.

4.4.2 FPGA: FX2 USB Interface VHDL

Listing 4.5 shows the VHDL entity declaration of the fx2if2 entity, which interfaces between the FX2 chip and the two FIFOs which buffer inbound and outbound data of the USB interface. fx2if2 is equivalent to the block labelled “FX2 Interface” in figure 4.1.

Signal naming is straightforward: Signals starting with Fx2Pin connect directly to the correspondingly named signal of the FX2 chip.

Signals starting with DstFifo connect to the FIFO that buffers data received from the FX2, signals starting with SrcFifo connect to the FIFO buffering data to be sent out to the FX2.

ClkxCI is clocking the entire interface. This 48 MHz clock is originally generated by the FX2 on its IFCLK pin. In the FPGA it is processed by a DCM (Digital Clock Manager) and phase shifted, to enable the FPGA to match the timing constraints of the FX2 interface.

Listing 4.5: VHDL entity declaration of fx2if2

```vhdl
entity fx2if2 is
  port ( 
    ClkxCI : in std_ulogic;
    RstxRBI : in std_ulogic;
    --
    Fx2PinFLAGDxSI : in std_ulogic;
    -- source FIFO (to USB)
    -- FIFO has to be ‘First Word Fallthrough’ Style!
    SrcFifoDataxDI : in std_logic_vector(7 downto 0);
    SrcFifoReadxSO : out std_ulogic;
    SrcFifoEmptyxSI : in std_ulogic;
    SrcFifoAlmostEmptyxSI : in std_ulogic;
    -- destination FIFO (from USB)
    DstFifoDataxDO : out std_logic_vector(7 downto 0);
    DstFifoWritexSO : out std_ulogic;
    DstFifoFullxSI : in std_ulogic;
    DstFifoAlmostFullxSI : in std_ulogic;
    DstFifoAlmostFull2xSI : in std_ulogic;
    -- FX2 interface pins
    Fx2PinFIFOADDR1xSO : out std_ulogic;
    Fx2PinSL0ExSBG : out std_ulogic;
    Fx2PinSL1DxBBO : out std_ulogic;
    Fx2PinSLWRxSBG : out std_ulogic;
    Fx2PinPKTENDxSBG : out std_ulogic;
  );
end fx2if2;
```
The attribute `iob` of ... : signal is "FORCE"; lines serve the same purpose as previously explained in 4.3.7.

### 4.4.3 FPGA: Monitor & Sequencer

![Diagram of FX2 Interface](image)
Figure 4.14 shows a section of figure 4.6, the section which is encapsulated by the high-level VHDL entity Fx2MonSeqRR, whose entity declaration is shown in listing 4.6.

The VHDL entity Fx2MonSeqRR thus comprises of:

- FX2 interface HDL: fx2if2
- large FIFOs for incoming and outgoing data
- Timestamp Counter
- Sequencer to inject AER data to the routing fabric
- Monitor to record AER data from the routing fabric

In the listing 4.6, we can identify three generics, all of whom are set to “false” for normal operation.

Setting TestEnableSequencerNoWait to true causes the sequencer to spew out AEs as fast as possible, while usually it would sequence them according to their attached inter-spike-interval timestamp.

Setting TestEnableSequencerToMonitorLoopback causes the sequencer to be looped back directly to the monitor. Setting this to true can be used to test the entire USB interface, from PC to FPGA and back.

Signals prefixed with Fx2Pin come from the underlying fx2if2 entity, and are connected to the corresponding pins of the FX2 USB interface chip.

Signals in the section “Input to Monitor” and “Output from Sequencer” are the connection to the generic AER routing fabric. Their “RR” handshake modality will be explained in the next chapter.

Listing 4.6: VHDL entity declaration of Fx2MonSeqRR

```
-- Fx2MonSeqRR

library ieee;
use ieee.std_logic_1164.all;

-- Xilinx primitives:
library UNISIM;
use UNISIM.VComponents.all;

-- entity

entity Fx2MonSeqRR is
generic (  
  TestEnableSequencerNoWait : boolean;
  TestEnableSequencerToMonitorLoopback : boolean;
  EnableMonitorControlsSequencerToo : boolean
);
```
4.4 USB 2.0 Interface Implementation Details

```vhdl
port (  
  -- clock and reset stuff  
  ResetxRBI      : in std_ulogic;  
  --  
  CoreClkxCI     : in std_ulogic;  
  RealIFCLKxCI   : in std_ulogic;  
  ShiftedIFCLKxCO  : out std_ulogic;  
  --  
  -- FX2 pins  
  Fx2PinFIFOADDR1xSO : out std_ulogic;  
  Fx2PinSLOExSB0   : out std_ulogic;  
  Fx2PinSLRDxSB0   : out std_ulogic;  
  Fx2PinSLWRxSB0   : out std_ulogic;  
  Fx2PinPKTENDxSB0 : out std_ulogic;  
  --  
  Fx2PinFLAGAxSI  : in std_ulogic;  
  Fx2PinFLAGBxSI  : in std_ulogic;  
  Fx2PinFLAGCxSI  : in std_ulogic;  
  Fx2PinFLAGDxSI  : in std_ulogic;  
  -- pulldown in UCF  
  Fx2PinPAINTxSI0 : in std_ulogic;  
  -- pulldown in UCF  
  Fx2PinsPBxDIO   : inout std_ulogic_vector(7 downto 0);  
  --  
  -- Input to Monitor  
  MonInAddrxDI   : in std_ulogic_vector(31 downto 0);  
  MonInSrcRdyxSI : in std_ulogic;  
  MonInDestRdyxSO : out std_ulogic;  
  --  
  -- Output from Sequencer  
  SeqOutAddrxDO  : out std_ulogic_vector(31 downto 0);  
  SeqOutSrcRdyxSO : out std_ulogic;  
  SeqOutDestRdyxSI : in std_ulogic;  
);  
end Fx2MonSeqRR;
```
4.5 Performance Measurements and Conclusion

In order to get scope measurements of the latency of both the serial and parallel AER interfaces, we used the following setup:

An AEXv4 board clocked at the typical 125 MHz was made to loop every address-event present in the system through both the parallel and serial AER interface. A ribbon cable was used to loop back the parallel AER output to the parallel AER input, similar a 0.5 m SATA cable was used to loop the serial AER output back to the serial AER input. Figure 4.15 shows the setup. (The parallel AER and serial AER loop-back cables are curled up only to fit on the photograph.)

The FPGA routing fabric, which will be explained in detail in the next chapter in figure 5.1 was put in a special configuration, which is illustrated in 4.16.

The dashed routing paths are deactivated for the measurement, only the solid arrow routing paths are active. Similar, Merger and Filter blocks faded out take no part in the process.

The routing is configured as follows: An address-event coming in via
4.5 Performance Measurements and Conclusion

Figure 4.16: FPGA routing fabric setup for latency measurements, (solid data-paths enabled, dashed ones disabled)
parallel AER is routed to the serial AER output. An address-event coming in via the serial AER input is routed to the parallel AER output. In combination with the aforementioned loop-back cables installed, we have a circle. An address-event entering that structure will circulate forever.

In addition to that, the USB interface, via the Sequencer block is connected to the serial AER output. This allows us to inject address-events into that circle.

After the board comes out of reset, no address-events are present in the circle. Only after we inject something via USB, we have address-events circulating.

The oscilloscope used to perform the following measurements was a Tektronix TDS2024C (200 MHz, 2 GS/s).

### 4.5.1 Bandwidth and Latency of the Serial AER Interface

In the figures 4.17 and 4.18 we measure two signals of the TLK3101 SerDes chip: TX\(_{EN}\) (scope channel 1) and RX\(_{DV}\).

TX\(_{EN}\) is high when the FPGA transmits a 16 bit word to the SerDes to be sent out over the serial AER link, RX\(_{DV}\) is high when the SerDes transmits a 16 bit word received over the serial AER link back to the FPGA.

Figure 4.17 shows a snapshot at the start of the experiment. Initially there is no activity, then suddenly an even (injected via USB) starts circulating. While at this resolution we cannot really see how long it takes for an event to cross the Serial AER link, we can see that in order to complete the whole loop (Serial AER loop-back → FPGA Routing → Parallel AER loop-back → FPGA Routing and back to the serial AER), it takes the address-event about two time-divs, which is about 500 ns.

Figure 4.18 shows a close up look at the same signals. We measure the time from the rising edge on TX\(_{EN}\) to the rising edge of RX\(_{DV}\) using the scope cursors and get roughly 50 ns. To get the total serial AER link latency, we have to add one clock cycle of 8 ns to that value (for the second 16 bit word to arrive), and come to a total link latency of 58 ns over a link distance of 0.5 m.

The bandwidth of the serial AER interface is more easily calculated than measured. The SerDes requires two clock cycles to transmit an
4.5 Performance Measurements and Conclusion

Figure 4.17: Serial AER interface, single event coming in and starting to circulate. (Ch1: TX_EN, Ch2: RX_DV)

Figure 4.18: Serial AER interface, latency measurement. (Ch1: TX_EN, Ch2: RX_DV)
address-event, and is capable of sending them back-to-back. From this we can conclude that with the SerDes being clocked at 125 MHz, we get an address-event bandwidth of **62.5 MHz**.

Listing 4.7: Shell script `aex_inject_single.sh` to inject a single event for these measurements.

```bash
#!/bin/sh
cd ../../../aerfx2/kmod/xio/
./stim/xgen.py 0 1 -1 | 
./x86_64/xio-bin-dx MONSEQ 1000000 /dev/aerfx2_0
exit
```

Listing 4.7 shows the simple commands from the xio codebase used to inject a single address-event for this measurement (see next chapter for details on `xgen.py` and `xio`).

### 4.5.2 Bandwidth and Latency of the Parallel AER Interface

In the figures 4.19 and 4.20 we measure the **REQ** (scope channel 1) and **ACK** (scope channel 2) signals of the parallel AER interface. Parallel AER Output and Input interface were configured to use the P2P hand-shake protocol (see figure 2.11) with high-active request and acknowledge signals.

The first figure 4.19, as in the serial AER measurements before, shows the start of the experiment, with a single address-event suddenly being injected via the USB interface, then looping through the whole system.

Again we can see that that single address event takes roughly 500 ns to cycle through the whole loop (2 time-divs of 250 ns) in the first plot.

In figure 4.20 we can see the details of a single parallel AER hand-shake. The interfaces are clocked at the AEX core clock of 62.5 MHz, which makes a clock cycle 16 ns long. We can see that from **REQ** being asserted, it takes 5 cycles until **ACK** is asserted by the receiving end. 6 cycles later, **REQ** is de-asserted, and again 5 cycles later, **ACK** is de-asserted.

We can conclude, that in the measured interface setup, the receiving end takes 5 cycles to react, the sending end takes 6 cycles. This gives us a total of 22 cycles to complete a full four phase hand-shake, a total of 352 ns. This gives us a parallel AER interface address-event rate of 2.84 MHz.
4.5 Performance Measurements and Conclusion

The parallel AER interface measured in this setup has in no way been optimized for higher AER bandwidth, because most chips used with the AEX system have even slower parallel AER interfaces.

The parallel AER interfaces could easily be run at the system clock of 125 MHz instead of the core clock of 62.5 MHz, DDR registers could be used to synchronize the incoming asynchronous hand-shake signals.

But then we would have to introduce wait-states, since this would be too fast for many chips and result in unstable parallel AER communication.

Another observation one can make in these two measurements, is one of the mentioned problems parallel AER interfaces using ribbon cables have: cross-talk.

Both scope probes have been attached at the sending end, the parallel AER output of the AEX board. This is where REQ is driven, while ACK is driven on the other end of the ribbon cable. This is why we have a quite nice REQ signal, but a very messy ACK signal. The ACK signal is affected by every edge of the REQ signal, which is called far-end cross-talk. In addition, ACK overshoots with its own edges, which is because we have reflections, because we are using unterminated single ended signalling. Both of these issues are eliminated with the serial AER interface we introduced before.
4 AEXv4 – Hardware Implementation Details

Figure 4.20: Parallel AER interface, latency measurement.
(Ch1: REQ, Ch2: ACK)

4.5.3 FPGA – FX2 USB Chip Interface Measurements

Many measurements have also been performed on the FPGA – FX2 USB chip interface, in order to verify that the timing constraints imposed on the FPGA by the FX2 interface specifications are met.

We will provide one example of such a measurement in the chapter on the iHead board built for the eMorph project, which uses the exact same USB interface as we use in the AEXv4 platform.

An eye-diagram measurement of the interface clock and the SLWR time-critical interface control signal made to verify these timing constraints will be presented in section 9.7.2.
4.5 Performance Measurements and Conclusion

4.5.4 AEXv4 Interface Spec Sheet

Table 4.4 summarizes the key performance characteristics of the AEX platform, which we have established in this chapter:

<table>
<thead>
<tr>
<th>AEXv4 Performance Spec Sheet</th>
<th>typical</th>
<th>maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Clock</td>
<td>125.0 MHz</td>
<td>156.25 MHz</td>
</tr>
<tr>
<td>Core Clock</td>
<td>62.5 MHz</td>
<td>78.125 MHz</td>
</tr>
<tr>
<td>Mon/Seq Timestamp Resolution (8 core clock cycles)</td>
<td>128.0 ns</td>
<td>102.4 ns</td>
</tr>
<tr>
<td>Serial AER Bit-Rate</td>
<td>2.5 GHz</td>
<td>3.125 GHz</td>
</tr>
<tr>
<td>Serial AER 32 bit AE Bandwidth</td>
<td>62.5 MHz</td>
<td>78.125 MHz</td>
</tr>
<tr>
<td>Serial AER Link Latency (0.5 m)</td>
<td>58 ns (measured)</td>
<td>47 ns (estimated)</td>
</tr>
<tr>
<td>Parallel AER Bandwidth</td>
<td>chip dependent</td>
<td></td>
</tr>
<tr>
<td>USB Bandwidth</td>
<td>40 MBytes/s</td>
<td>41 MBytes/s</td>
</tr>
<tr>
<td>USB AER Event-Rate</td>
<td>5.0 MHz</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.4: AEXv4 performance specification sheet

An overview of the AEX board and and the 3 GHz Serial AER interface is also published in: [Liu 15, sec. 13.2.5, p. 324f.].
4.5.5 Conclusion

While in the previous chapters we established the motivation, need for and the design requirements for the AEX interfacing platform for neuromorphic chips, in this chapter, we gave a detailed overview of these design requirements where implemented in hardware in the AEXv4 system.

The details of the hardware and FPGA implementations of the three interfaces of the AEXv4 system were explained. As part of that, we also discussed the details of our novel serial AER interface, which overcomes the problems observed in board-to-board parallel AER interfacing.

We then concluded with a number of measurements performed on the system, and established the key performance characteristics of the AEXv4 platform hardware.

While the focus of this chapter was primarily the hardware of the AEX platform, next chapters will focus on the various software aspects of the system: We will present the generic FPGA framework for AER interfacing and processing written for the AEXv4 and all the firmware, kernel drivers and user-space software developed for the USB interface present on the AEX platform.
In the previous chapter, we have presented the hardware implementation details of the AEXv4 interfacing system. The FPGA-implemented parts of the parallel AER, serial AER and USB monitoring and sequencing interfaces have also been presented.

In this chapter we will now describe the FPGA codebase, which is used to tie all these parts together. These are the parts used to route, filter and process the AER traffic send and received by all the interfaces.

The parts will be introduced in a bottom-up order. We will first explain according to which convention the various parts involved are communicating with each other.

Then the core entities will be described: the Splitter, Filter, Merger and LongPath entities.

Then we will show how these entities can be combined to for the generic AER routing fabric implemented in the AEX FPGA, and present the simple most level of abstraction, the CompatibilityFabric which allows people with almost zero knowledge of VHDL or FPGAs to configure the AEX routing for their needs.

### 5.1 Core AER VHDL Entities

After establishing the interfacing convention used for interacting between the entities described here, we will explain how the basic entities for AER routing, filtering and processing are used.
5.1.1 RR Interface Specification

In digital synchronous interfaces, we usually have a source and a destination (sink), where one acts as the master, and the other as the slave. Either participant can be the interface master and we thus have two types of interfaces: One where the source is the master, one where the destination is the master.

Because there can be only one master in a such as master-slave type interface, these two types of interfaces are incompatible, and there is a significant effort, when connecting the two types, usually involving control logic and registers, sometimes even requiring FIFOs.

For our interfaces, we chose a different approach, which unlike the just described master-slave type interfaces is completely symmetric.

Both the source and destination present a “Ready” signal, which we call:

- \( \text{SrcRdy} \): Source Ready
- \( \text{DstRdy} \): Destination Ready

When the destination is ready to receive an AER word, it asserts \( \text{DstRdy} \).

When the source is ready to transmit an AER word, it presents that AER word on the AER data lines and asserts \( \text{SrcRdy} \).

If at the next rising clock edge, both source and destination have signalled “Ready”, the AER word is considered to be transmitted from the source to the destination.

This means that both the source and the destination entity have to “calculate” whether a transmission happened, which sounds like we are doing things twice and waste FPGA resources, however one of the key functionalities of the FPGA synthesis tools, is exactly to identify such duplicate identical logic and to remove such redundancy in order to preserve resources.

5.1.2 Splitter & Filter

The Splitter & Filter entity shown in listing 5.1 has AER input interface, and three AER output interfaces.

The entity generics are pretty self-explanatory: one can set the width of the AER interfaces, which defaults to 32 bits, and can configure the entity to provide FIFO buffers at the input and/or output ports.
5.1 Core AER VHDL Entities

The AER input interface has signals called:

- **InpDataxDI**: AER Data 32 bit Input
- **InpSrcRdyxSI**: Source Ready Input
- **InpDstRdyxSO**: Destination Ready Output

There are three AER output interfaces with signals called:

- **Out[ABC]DataxDO**: AER Data 32 bit Output
- **Out[ABC]SrcRdyxSO**: Source Ready Output
- **Out[ABC]DstRdyxSI**: Destination Ready Input

Please refer to the VHDL signal naming convention provided in 0.1.2. In accordance with that convention, the last letter of the signal names, "I" or "O" indicate whether a port is an input or an output port.

Note that the AER input interface and the AER output interfaces have exact opposite port directions for all signals. This means any two such ports in the various entities we are going to present here can be wired up directly, without any further logic required.

Listing 5.1: VHDL entity declaration of FilterSplitter3RR

```vhdl
entity FilterSplitter3RR is
  generic (
    width       : positive := 32;
    pre_fifo_depth : positive := 4;
    post_fifos_depth : positive := 4
  );
  port (
    ClkxCI       : in std_ulogic;
    RstxRBI      : in std_ulogic;
    --
    InpDataxDI   : in std_ulogic_vector(width-1 downto 0);
    InpSrcRdyxSI : in std_ulogic;
    InpDstRdyxSO : out std_ulogic;
    --
    FilterValuexDO : out std_ulogic_vector(width-1 downto 0);
    FilterResultToAxSI : in std_ulogic;
    FilterResultToBxSI : in std_ulogic;
    FilterResultToCxSI : in std_ulogic;
    --
    OutADataxDO : out std_ulogic_vector(width-1 downto 0);
    OutASrcRdyxSO : out std_ulogic;
    OutADstRdyxSI : in std_ulogic;
    --
    OutBDataxDO : out std_ulogic_vector(width-1 downto 0);
    OutBSrcRdyxSO : out std_ulogic;
    OutBDstRdyxSI : in std_ulogic;
    --
    OutCDataxDO : out std_ulogic_vector(width-1 downto 0);
    OutCSrcRdyxSO : out std_ulogic;
    OutCDstRdyxSI : in std_ulogic
  );
end FilterSplitter3RR;
```
In order to be capable to filter AER data and decide which address-event goes where, the following signals are present:

- **FilterValuexDO**: The AE value we want to operate on
- **FilterResultToAxSI**: Send that AE to output interface A?
- **FilterResultToBxSI**: Send that AE to output interface B?
- **FilterResultToCxSI**: Send that AE to output interface C?

When we use the **FilterSplitter3RR**, we have to also code a function which based on **FilterValuexDO**, produces the three values which determines on which output an AE is forwarded or dropped: **FilterResultTo[ABC]xSI**.

The function can be of any kind, e.g. a integer range check, a binary operation deciding on some bit of **FilterValuexDO**, etc.

Also it can send an AE to any number of the three outputs, even to none at all to drop an AE. The only limitation to that function is that it has to be purely combinatorial and fast enough to be calculated within one clock cycle.

Similar to **FilterSplitter3RR**, there is an entity called **FilterSplitter2RR**, which only has two output interfaces. Of course both of these entities can be cascaded into a binary/ternary tree structure, if more than three outputs are required.

### 5.1.3 Merger

Listing 5.2 shows opposite of the splitter presented in the last section, the AER stream merger **Merger3RR**.

Because there is no filtering capability here, the **Merger3RR** is a quite simple entity. It has three AER input interfaces and merges the incoming AER data into one stream produced at the single AER output interface.

The entity generics have the exact same meaning as in the previous entity.

The most interesting functionality of that entity is “under the hood”: When more than one, maybe even all three AER inputs provide data simultaneously, there is no preference for a one input over another. In that case, the merger reads AER data from the inputs in a round-robin fashion.

#### Listing 5.2: VHDL entity declaration of **Merger3RR**
Again there also exists an entity called `Merger2RR` with only has two input interfaces. More complex merger structures again can be constructed by combining multiple of those mergers in a binary/ternary tree structure.

### 5.1.4 LongPath – spanning across the FPGA

Sometimes it turns out to be non-trivial to bring data from one side of and FPGA to another. That is if all the data and flow control the signals cannot be routed in such a way, to reach from source to destination in well under one clock cycle.

Sometimes the problem is also difficult to detect. Let’s say we have a parallel AER interface on one side directly connected to a serial AER interface on the opposite side of the FPGA. It might very well happen, that the synthesis software is unable to achieve to required timing constraints, and reports the problem to be in the serial AER interface (rather than the path between the two interfaces). This is because “place & route” tried to stretch out the serial AER and parallel AER interfaces across the FPGA in order to match the timing constraints.

The solution to prevent these types of issues is to feed data that has to cross greater spatial distances on the FPGA into a cascade of
5 Generic FPGA Codebase for AER Interfacing & Processing

registers, which then need to be buffered by a FIFO, who’s minimum depth depends on the depth of the register cascade.

This whole structure of register cascade and FIFO registers can be distributed across the spatial distance to be crossed on the FPGA by the “place & route” algorithm.

Of course this construct introduces latency, according to the register cascade depth chosen, but sometimes this is required for the “place & route” algorithm to achieve the timing constraints required.

LongPathRR show in listing 5.3 encapsulates exactly this register cascade plus FIFO structure with a “RR-type” interface on both ends.

The calculations on the resulting FIFO depth depending on the generic parameters chosen are documented in the comments at the beginning of the listing.

Listing 5.3: VHDL entity declaration of LongPathRR

```vhdl
-- LongPathRR -- Register-Chain & FIFO to get across spatial distances
library ieee;
use ieee.std_logic_1164.all;

-- reg_depth, extra_fifo_depth
-- a reg_depth of x requires a fifo depth of 2*x + 2
-- the extra_fifo_depth is added to that, the actual FIFO depth
-- we get is thus: 2 * reg_depth + 2 + extra_fifo_depth

entity LongPathRR is
    generic (
        width : positive;
        reg_depth : positive := 2;
        extra_fifo_depth : natural := 0
    );
    port (
        ClockxCI : in std_ulogic;
        ResetxRBI : in std_ulogic;
        --
        InpDataxDI : in std_logic_vector(width-1 downto 0);
        InpSrcRdyxSI : in std_logic;
        InpDstRdyxSO : out std_logic;
        --
        OutDataxDO : out std_logic_vector(width-1 downto 0);
        OutSrcRdyxSO : out std_logic;
        OutDsrRdyxSI : in std_logic
    );
end entity LongPathRR;
```

138
5.2 Generic AER Routing Fabric of the AEX

Figure 5.1 shows the structure of the generic AER routing fabric, which was already presented as a black box at the sender of the block diagram of the entire FPGA back in figure 4.6.

On the left edge, the fabric is connected to the monitor and sequencer (4.4.3) which connect to the USB interface. On the top side the fabric connects to the parallel AER input and output interfaces presented in 4.2, on the bottom side to the serial AER interface presented in 4.3.

The key components are three AER three-way splitters (5.1.2) and three AER three-way mergers (5.1.3).
Listing 5.4 show the VHDL entity declaration of that fabric with all the ports to the interfaces mentioned. The entity is named a bit confusingly CompatibilityFabric for historical reasons.

Listing 5.4: VHDL entity declaration of CompatibilityFabric

```
library ieee;
use ieee.std_logic_1164.all;

entity CompatibilityFabric is
  port (CoreClkxCI : in std_ulogic;
        ResetxRBI : in std_ulogic;
        --
        SequencerDataxDI : in std_logic_vector(31 downto 0);
        SequencerSrcRdyxSI : in std_ulogic;
        SequencerDstRdyxSO : out std_ulogic;
        --
        MonitorDataxDO : out std_logic_vector(31 downto 0);
        MonitorSrcRdyxSO : out std_ulogic;
        MonitorDstRdyxSI : in std_ulogic;
        --
        PIAerDataxDI : in std_logic_vector(31 downto 0);
        PIAerSrcRdyxSI : in std_logic;
        PIAerDstRdyxSO : out std_ulogic;
        --
        POAerDataxDI : out std_logic_vector(31 downto 0);
        POAerSrcRdyxSI : out std_logic;
        POAerDstRdyxSO : in std_logic;
        --
        SIAerDataxDI : in std_logic_vector(31 downto 0);
        SIAerSrcRdyxSI : in std_logic;
        SIAerDstRdyxSO : out std_logic;
        --
        SOAerDataxDI : out std_logic_vector(31 downto 0);
        SOAerSrcRdyxSO : out std_logic;
        SOAerDstRdyxSI : in std_logic);
end CompatibilityFabric;
```

5.3 Configuring the Fabric

The while researchers could write and customize their own fabric to customize an AEX system to their needs, they usually resort to using the generic CompatibilityFabric.

However even this fabric must be configured to suit the task at hand, but this can be done by customizing a simple configuration file for the
5.3 Configuring the Fabric

CompatibilityFabric, a so-called AEXconfig entity.

Listing 5.5 shows the SpinTest0 version of such an AEXconfig. This is also the AEXconfig which has been used to perform then benchmark measurements of the AEXv4 in 4.5.

As mentioned in the comments of the entity declaration and implementation, only the constant declarations between line 80 and 150 should be modified. The rest of the code is constant to port assignments boiler-plate code which must be identical for all AEXconfig versions.

Listing 5.5: VHDL entity AEXconfig, version SpinTest0

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity AEXconfig is
port (  
signal confPathEnablePPxS : out std_logic;
signal confPathEnablePSxS : out std_logic;
signal confPathEnablePUxS : out std_logic;

signal confPathEnableSPxS : out std_logic;
signal confPathEnableSSxS : out std_logic;
signal confPathEnableSUxS : out std_logic;

signal confPathEnableUPxS : out std_logic;
signal confPathEnableUSxS : out std_logic;
signal confPathEnableUUxS : out std_logic;

-- PARALLEL to *

signal confPPFilterRangeMinxD : out std_logic_vector(31 downto 0);
signal confPPFilterRangeMaxxD : out std_logic_vector(31 downto 0);
signal confPPOutMaskAndxD : out std_logic_vector(31 downto 0);
signal confPPOutMaskOrxD : out std_logic_vector(31 downto 0);

-- SERIAL to *

signal confSPFilterRangeMinxD : out std_logic_vector(31 downto 0);
signal confSPFilterRangeMaxxD : out std_logic_vector(31 downto 0);
signal confSPOutMaskAndxD : out std_logic_vector(31 downto 0);
signal confSPOutMaskOrxD : out std_logic_vector(31 downto 0);
end entity;
```
5 Generic FPGA Codebase for AER Interfacing & Processing

```vhdl
signal confSUFilterRangeMinxD : out std_ulogic_vector(31 downto 0);
signal confSUFilterRangeMaxxD : out std_ulogic_vector(31 downto 0);
signal confSUOutMaskAndxD : out std_ulogic_vector(31 downto 0);
signal confSUOutMaskOrxD : out std_ulogic_vector(31 downto 0);

-- USB to *
signal confUSFilterRangeMinxD : out std_ulogic_vector(31 downto 0);
signal confUSFilterRangeMaxxD : out std_ulogic_vector(31 downto 0);
signal confUSOutMaskAndxD : out std_ulogic_vector(31 downto 0);
signal confUSOutMaskOrxD : out std_ulogic_vector(31 downto 0);

signal confUPFilterRangeMinxD : out std_ulogic_vector(31 downto 0);
signal confUPFilterRangeMaxxD : out std_ulogic_vector(31 downto 0);
signal confUPOutMaskAndxD : out std_ulogic_vector(31 downto 0);
signal confUPOutMaskOrxD : out std_ulogic_vector(31 downto 0);

signal confUUFilterRangeMinxD : out std_ulogic_vector(31 downto 0);
signal confUUFilterRangeMaxxD : out std_ulogic_vector(31 downto 0);
signal confUUOutMaskAndxD : out std_ulogic_vector(31 downto 0);
signal confUUOutMaskOrxD : out std_ulogic_vector(31 downto 0);
end AEXconfig;
```

-- NO CHANGES BEFORE THIS POINT

-- architecture YOUR_CONFIGURATION_NAME of AEXconfig is:
architecture SpinTest0 of AEXconfig is

-- enabled paths:
constant PathEnablePPxS : std_ulogic := '0';
constant PathEnablePSxS : std_ulogic := '1';
constant PathEnablePUxS : std_ulogic := '0';
constant PathEnableSPxS : std_ulogic := '1';
constant PathEnableSSxS : std_ulogic := '0';
constant PathEnableSUxS : std_ulogic := '0';
constant PathEnableUPxS : std_ulogic := '0';
constant PathEnableUSxS : std_ulogic := '1';
constant PathEnableUUxS : std_ulogic := '0';

-- filters and masks:

-- PARALLEL to *
constant PPFilterRangeMinxD: std_ulogic_vector(31 downto 0) := x"00000000";
constant PPFilterRangeMaxxD: std_ulogic_vector(31 downto 0) := x"FFFFFFFF";
constant PPOutMaskAndxD : std_ulogic_vector(31 downto 0) := x"FFFFFFFF";
constant PPOutMaskOrxD : std_ulogic_vector(31 downto 0) := x"00000000";
constant PSFilterRangeMinxD: std_ulogic_vector(31 downto 0) := x"00000000";
constant PSFilterRangeMaxxD: std_ulogic_vector(31 downto 0) := x"FFFFFFFF";
constant PSOutMaskAndxD : std_ulogic_vector(31 downto 0) := x"FFFFFFFF";
constant PSOutMaskOrxD : std_ulogic_vector(31 downto 0) := x"00000000";
constant PUFilterRangeMinxD: std_ulogic_vector(31 downto 0) := x"00000000";
constant PUFilterRangeMaxxD: std_ulogic_vector(31 downto 0) := x"FFFFFFFF";
5.3 Configuring the Fabric

constant PUOutMaskAndxD : std_ulogic_vector(31 downto 0) := x"FFFFFFFF";
-- mark path PU by setting a bit 28:
constant PUOutMaskOrxD : std_ulogic_vector(31 downto 0) := x"10000000";

-- SERIAL to *
constant SPFilterRangeMinxD: std_ulogic_vector(31 downto 0) := x"00000000";
constant SPFilterRangeMaxxD: std_ulogic_vector(31 downto 0) := x"FFFFFFFF";
constant SPOutMaskAndxD : std_ulogic_vector(31 downto 0) := x"FFFFFFFF";
constant SPOutMaskOrxD : std_ulogic_vector(31 downto 0) := x"00000000";

-- USB to *
constant USFilterRangeMinxD: std_ulogic_vector(31 downto 0) := x"00000000";
constant USFilterRangeMaxxD: std_ulogic_vector(31 downto 0) := x"FFFFFFFF";
constant USOutMaskAndxD : std_ulogic_vector(31 downto 0) := x"FFFFFFFF";
constant USOutMaskOrxD : std_ulogic_vector(31 downto 0) := x"00000000";

-- NO CHANGES PAST THIS POINT

constant UPFilterRangeMinxD: std_ulogic_vector(31 downto 0) := x"00000000";
constant UPFilterRangeMaxxD: std_ulogic_vector(31 downto 0) := x"FFFFFFFF";
constant UPOutMaskAndxD : std_ulogic_vector(31 downto 0) := x"FFFFFFFF";
constant UPOutMaskOrxD : std_ulogic_vector(31 downto 0) := x"00000000";
constant UUFilterRangeMinxD: std_ulogic_vector(31 downto 0) := x"000000DD";
constant UUFilterRangeMaxxD: std_ulogic_vector(31 downto 0) := x"FFFFFFFF";
constant UUOutMaskAndxD : std_ulogic_vector(31 downto 0) := x"FFFFFFFF";
-- mark path UU by setting a bit 30:
constant UUOutMaskOrxD : std_ulogic_vector(31 downto 0) := x"40000000";

begin

confPathEnablePPxS <= PathEnablePPxS;
confPathEnablePSxS <= PathEnablePSxS;
confPathEnablePUxS <= PathEnablePUxS;
--
confPathEnableSPxS <= PathEnableSPxS;
confPathEnableSSxS <= PathEnableSSxS;
confPathEnableSUxS <= PathEnableSUxS;
--
confPathEnableUPxS <= PathEnableUPxS;
confPathEnableUSxS <= PathEnableUSxS;
confPathEnableUUXS <= PathEnableUUXS;
--
confPPFilterRangeMinxD <= PPFILTERRANGE_MINxD;
confPPFilterRangeMaxxD <= PPFILTERRANGE_MAXxD;
confPPOutMaskAndxD <= PPOUTMASK_ANDxD;
confPPOutMaskOrxD <= PPOUTMASK_ORxD;
--
confPSFilterRangeMinxD <= PSFILTERRANGE_MINxD;
confPSFilterRangeMaxxD <= PSFILTERRANGE_MAXxD;
confPSOutMaskAndxD <= PSSOUTMASK_ANDxD;
confPSOutMaskOrxD <= PSSOUTMASK_ORxD;

end

143
There are three interfaces as discussed, which are denoted by the letters:

- **“U”**: USB (Monitor & Sequencer)
- **“P”**: Parallel AER Interfaces
- **“S”**: Serial AER Interfaces

Since there are three bi-directional interfaces, and all can be connected to all, there are a total of nine paths between the interfaces. “PU” for example denotes the path from the parallel AER input to the USB interface (the monitor), “SS” denotes the path from the serial AER input to the serial AER output, “US” denotes the path from the USB interface (the sequencer) to the serial AER output, etc.
5.3 Configuring the Fabric

In the “enabled paths” section, line 85 to 95, each of these paths can be either enabled, or disabled completely, e.g. setting PathEnableSUxS to “0” disables the path from the serial AER input to the USB interface (monitor). For a path disabled here, all settings regarding that path in the section below have no effect, because a disabling a path here has complete preference.

In the section “filters and masks”, address-events on each path can be filtered and modified.

For each path XY, there is a XYFilterRangeMinxD and XYFilterRangeMaxxD. Address-events whose numeric value is within that range (inclusive) pass the filter, those outside are discarded from that path.

In addition address-events can be modified using two binary masks: first XYOutMaskAndxD is and-ed bitwise to the AE, then XYOutMaskOrxD is or-ed bitwise to the AE.

Arbitrary bits thus can be set or cleared in an address-event. This is frequently used to tag a certain address-event in some way, or to add a binary offset to an address-event stream, to make it distinguishable from a stream from another source using the same address range per default.

As mentioned SpinTest0 is the AEXconfig used for the AEXv4 measurements in section 4.5. If you compare the PathEnableXYxS settings in the listing 5.5, you can see that they match exactly what was shown as the fabric configuration back in figure 4.16.
5.4 Conclusion

In this chapter we have presented a number of VHDL entities which form the basis of our generic AER processing codebase, such as AER mergers and splitters and the “RR-type” interface used between all these VHDL entities has also been described.

We have then shown how these components are used in the case of the AEX platform in the case of the CompatibilityFabric and the user-specified AEXconfig entities, which form the “generic AER routing fabric” used in the AEX platform, and we have explained, how even non-VHDL fluent users can configure their AEX platforms using a simple set of parameters and switches. A complex multi-chip setup using a set of four “cooperating” AEXconfig instances will be demonstrated in 11.1.3.

Many of the VHDL entities for AER routing, processing, filtering, etc. we have characterized in this chapter and the VHDL interfacing entities introduced in the previous chapter will be reused, many of them unmodified, in systems such as the AEXS, iHead and AEXL which will be the topic later in this thesis.

The next chapter will explain all the non-HDL/FPGA related software aspects of the USB interface implementation used.
Back in section 4.4 the hardware aspects of the USB interface of the AEX platform were characterized. In this chapter, we will complete the picture of the USB interface by presenting all the software parts involved in this interface.

These software aspects of the USB interface consists of these following parts, here ordered from the AEX FPGA to the program running on the PC:

- FPGA interface to FX2
- FX2 USB chip firmware
- “aerfx2” Linux kernel driver
- PC user-space code (e.g. “xio”)

The FPGA entity `fx2if2` that interfaces to the FX2 has already been explained in section 4.4.2.

The Cypress FX2 USB interface chip firmware, the Linux kernel driver and the xio reference user-space code implementation will be discussed in the next sections.

After that we will give an example on how the USB communication works to control the monitor function of the target device in section 6.4. Eventually we will benchmark the driver and measure the performance that can be achieved by its zero-allocation architecture in section 6.5.

We will conclude the chapter with an analysis of the performance of the components presented in this chapter, and an outlook in the light of possible future USB 3 super-speed based AER interfaces.
6.1 USB Interface Chip Firmware

The FX2 firmware code mostly consists of an initialization routine, which configures all aspects of the FX2 chip in the ways we need them to be configured for our USB interface application.

Going through this code would be too tedious and meaningless, without having a very in-depth understanding of the inner workings of the FX2, which are documented by Cypress on a hundreds of pages long “Technical Reference Manual”. We will thus take a different approach, and only present the most relevant and interesting parts of the code:

In 6.1.1 we will have a look at the configured and running USB device connected to a Linux PC, using the `lsusb` utility.

In 6.1.2 we will present the main function and main loop of the firmware, and explain what the firmware does, after all the initial configuration and setup code was executed.

Eventually we will have a look at a very relevant part of the configuration and setup code in 6.1.3. This code configures a number of flag signals, which are presented to the FPGA, for it to know about the state of the FIFO fill levels within the FX2.

6.1.1 USB Device Configuration – `lsusb`

Listing 6.1 shows the output of the Linux command `lsusb -v` for an AEX board connected to that computer. Sections of the output irrelevant for our purposes are cut out and replaced with “…”.

We can see that there are three endpoint configurations shown, which we use to interact with the AEX board. According to the USB Specification, the term “OUT” always means directed from the host (computer) to the device (e.g. our AEX), and “IN” signifies the reverse.

A USB device can have multiple endpoints, “EP” in short, which are the equivalent of ports in network lingo. An IN EP is thus a port the host can read data from, and OUT EP is a port the host can send data to.

Listing 6.1: USB device configuration as displayed by `lsusb`

```
1 # lsusb -d 04b4:8613 -v
2 Bus 002 Device 040: ID 04b4:8613 Cypress Semiconductor Corp. \ 
3 CY7C68013 EZ-USB FX2 USB 2.0 Development Kit
4 Device Descriptor:
5  bLength 18
```

148
6.1 USB Interface Chip Firmware

| bLength | 9 |
| bDescriptorType | 2 |
| wTotalLength | 171 |
| bNumInterfaces | 1 |
| bConfigurationValue | 1 |
| iConfiguration | 0 |
| bmAttributes | 0x80 |
| MaxPower | 100mA |

Interface Descriptor:

| bLength | 7 |
| bDescriptorType | 5 |
| bEndpointAddress | 0x01 EP 1 OUT |
| bmAttributes | 2 |
| Transfer Type | Bulk |
| Synch Type | None |
| Usage Type | Data |
| wMaxPacketSize | 0x0200 1x 512 bytes |
| bInterval | 0 |

Endpoint Descriptor:

| bLength | 7 |
| bDescriptorType | 5 |
| bEndpointAddress | 0x02 EP 2 OUT |
| bmAttributes | 2 |
| Transfer Type | Bulk |
| Synch Type | None |
| Usage Type | Data |
| wMaxPacketSize | 0x0200 1x 512 bytes |
| bInterval | 0 |

Endpoint Descriptor:

| bLength | 7 |
| bDescriptorType | 5 |
| bEndpointAddress | 0x86 EP 6 IN |
| bmAttributes | 2 |
| Transfer Type | Bulk |
| Synch Type | None |
| Usage Type | Data |
| wMaxPacketSize | 0x0200 1x 512 bytes |
| bInterval | 0 |
6 USB Interface Software Implementation

We use EP 2 OUT is used to send AER data to the AEX in order to be sequenced out into our setup. Vice versa, EP 6 IN is used to read data that we monitored on the AEX.

EP 1 OUT is used not for data transfer, but for control. The primary purpose is to tell the AEX to enable or disable the monitor. This will be explained in more detail in section 6.4.

6.1.2 FX2 Firmware Main Loop Tasks

Listing 6.2 shows the main function, also containing the main loop, which is executed continuously as long as the FX2 is running.

The first thing the main function does, is calling the init() function we programmed in the firmware. This is where most of the work is done. All components of the FX2 are configured and set up in the way we need it to operate in the AEX. Besides many other things, this includes configuring the interface to the FPGA in the “Slave FIFO” mode our AER data from EP2 and EP6, and setting up the flag signals, which indicate to the FPGA, what the current status of the FIFOs is. This flag setup will be described in more detail in section 6.1.3.

Listing 6.2: FX2 firmware: main function / main loop

```c
void main(void)
{
    init();

    while (1) {
        enforcehighspeed();

        // signal FxReady to fpga (0x80);
        IOA |= PORTA_FXREADY;

        while (1) {
            // did we get a command on ep1out?
            if (EP1OUT_HASDATA) {
                process_ep1out_command();
                ep1out_rearm();
            }

            // did a packet arrive over USB?
            if (!EP2EF) {
                OUTPKTEND = 0x02; // commit packet
                SYNCDELAY;
            }

            // try to detect a disconnect
            if (!IS_HIGHSPEED) {
                break;
            }

        }

        // NOT signal FxReady to fpga
    }
}
```

150
6.1 USB Interface Chip Firmware

Once *init()* is executed, the main loop is entered. *enforcehighspeed()* is then called to make sure the FX2 USB interface operates in “high-speed USB” mode. Operation in slower USB 2.0 interface speeds is not supported, and the function only completes, when a successful high-speed (480 MHz) USB connection is established.

Once this is the case, we signal to the FPGA that the FX2 is ready to operate by asserting “FxReady”, a signal which is presented on bit 7 of GPIO port A of the FX2.

Then the inner loop is entered. The inner loop has three very simple tasks: It checks and processes any control commands which might have arrived on EP1, It checks whether any OUT AER data has arrived on EP2, and commits that data to the FIFO.

At last it monitors whether we still have a stable high-speed connection. If so, it loops back to checking EP1 for new data.

Only if the USB link goes down or drops to a slower speed for some reason, the inner loop is left. The “FxReady” signal is deasserted to tell the FPGA that there is a problem with the USB connection, and finally *enforcehighspeed()* is called again, which waits until the USB connection is up at the required speed again.

6.1.3 FX2 FIFO Flags Setup

The AER traffic, incoming and outgoing, is buffered in the FX2 internal FIFOs. The status (fill level) of these FIFOs is represented on multiple pins of the FX2 chip, connected to the FPGA, for the FPGA interface *fx2if2* to interpret and act upon, e.g. by not writing data to a FIFO which is already full and unable to store any further data, or in order for the FPGA to know, when the FX2 has received AER data for the FPGA to process, by providing a flag, which signals, whether the FIFO from which the FPGA reads data from the PC is empty or not.

We will now describe these flags and how they are configured by the FX2 firmware programmed for our USB interface.

**Flag Pin Configuration:** Listing 6.3 shows which flags are configured by the firmware to be presented on which pins of the FX2 chip. There are four flag pins on the FX2, labelled from A through D.
A flag pin can either be configured to signal a default FIFO status, such as full or empty, or to signal whether the FIFO fill level exceeds a certain threshold (programmable flag).

**Listing 6.3: FX2 firmware: flag pin configuration**

```c
// flag config
PINFLAGSAB = 0x64; // flag B: EP6PF, flag A: EP2PF
SYNCDELAY;
PINFLAGSCD = 0x08; // flag D: unused/default, flag C: EP2EF
SYNCDELAY;
```

As can be seen in listing 6.3,

- **FLAGA** signals **EP2PF**, the “endpoint 2 programmable flag”,
- **FLAGB** signals **EP6PF**, the “endpoint 6 programmable flag”,
- **FLAGC** signals **EP2EF**, the “endpoint 2 empty flag”,
- **FLAGD** is left unused.

**EP2 (OUT) Programmable Flag:** Listing 6.4 shows how the **EP2PF** is programmed: The flag is asserted high whenever there are more than 4 bytes in the EP2 OUT FIFO.

**Listing 6.4: FX2 firmware: EP2 programmable flag**

```c
// ep2: flag high if >= 4 in fifo
EP2FIFOPFH = 0x80; // DECIS=1, PKTSTAT=0, PKTS=0, PFC=0
SYNCDELAY;
EP2FIFOPFL = 0x04; // PFC += 4
SYNCDELAY;
```

**EP6 (IN) Programmable Flag:** Listing 6.5 shows how the **EP6PF** is programmed: The flag is asserted high, whenever there are less than 32 bytes free in the EP6 IN FIFO.

**Listing 6.5: FX2 firmware: EP6 programmable flag**

```c
// announce fifo is full (less than 32B available):
// ep6: flag high if >= 1p + (512-32)B in fifo
// 512 - 32 = 480 = 0x01E0 = 256 + 224
EP6FIFOPFH = 0x89; // DECIS=1, PKTSTAT=0, PKTS=1, PFC=256
SYNCDELAY;
EP6FIFOPFL = 0xe0; // PFC += 224
SYNCDELAY;
```

**EP6PF** is used to detect early, when the FPGA has to stop writing data to the FX2, since the respective FIFO is running full.

**EP2PF** is used by the FPGA to detect that it can burst-read data from the FX2. When **EP2PF** is deasserted, this means that only few bytes
are left to be read from the FX2. The FPGA then starts reading those bytes more slowly, until the FX2 also asserts EP2EF, which indicates, that even the last byte available to be read from the FX2 has been consumed by the FPGA.

6.2 Linux Kernel Driver

In [Fasnacht 07a], two Linux Kernel drivers were designed. The first one called “aerio” supported in-kernel merging and splitting of multiple AER streams when multiple AEX devices were connected.

This concept was abandoned, because of the extra complexity in-kernel merging and splitting added to the driver code.

In consequence a simpler driver architecture was proposed, called aerfx2, and an initial proof-of-concept of that aerfx2 was developed.

The driver was expanded to support devices other, than the AEX, and functionality such as the monitor control via EP1 mentioned above were added. Also the compatibility with various Linux kernel versions was expanded. The driver can be compiled with anything from ancient 2.6 Linux kernel versions to latest kernel version 4.2.

6.2.1 Kernel Driver Architecture

One key concept was retained throughout all the versions of the aerfx2” driver developed: The “Zero-Allocation Data Flow Architecture”, which is presented in figure 6.1. (Figures 6.1 and following are adapted from [Fasnacht 07a].)

Allocating memory in the kernel comes with some difficulty. There is no way, that the kernel can guarantee, that an allocation succeeds within a time necessary for our driver to operate properly.

Depending on how the allocation is done, it can either fail or the allocation can block maybe even indefinitely.

For this reason, the aerfx2 driver pre-allocates all the memory required for the so-called USB request blocks (URBs) already when a new device (e.g. the AEX) is connected to the PC.

Listing 6.6 shows the lines of the driver, which control how many URBs are pre-allocated for both IN- and OUT-traffic.
Figure 6.1: aerfx2.c: zero-allocation data flow architecture
In figure 6.1, the blue rectangles illustrate our pre-allocated URBs. If the driver receives AER data from user-space, (AER data is illustrated by the green rectangles), it takes and empty, but already allocated URB, and stores the AER data in that memory. It then forwards that URB to the USB host controller hardware via `usb_submit_urb()` for transmission via USB to the FX2.

As soon as the USB host controller has transmitted the data in that URB, the driver receives that URB back. However it does not delete that unused URB by freeing that chunk of memory, but stores it in a list of empty URBs, which will be recycled to transmit further data.

The same strategy we just described for the OUT direction also applies for the IN directed AER traffic.

Figure 6.2 illustrates the situation with three AEX devices connected. The driver operates completely independent for each of the devices. For each AEX connected, the driver creates a separate character device for user-space programs to communicate with the kernel driver. The character device for the first connected AEX is called `/dev/aerfx2_0`, the character device for the second AEX `/dev/aerfx2_1` et cetera.

Figure 6.2: aerfx2.c: multiple AEX device support

Figure 6.3 displays which parts of the aerfx2 driver operate in which execution context. The only specialty to mention here is SoftIRQ.
context. If the driver receives IN data from the monitor, but there is not program, which wants to read that AER data from the character device, a kernel tasklet is instantiated, which will discard the AER data and recycle the URB in SoftIRQ context.

Figure 6.3: aerfx2.c: code execution context

Figure 6.4 illustrates that the driver operates with two different locking contexts, corresponding to two kernel spinlocks, “ilock” for IN traffic and “olock” for OUT traffic. Using these locks ensures, that under no circumstances, an URB could get corrupted or lost, e.g. because two user-space processes access the driver concurrently.

Figure 6.4: aerfx2.c: locking domains
6.3 PC User-Space Code

**xio** is the user-space code reference implementation.

When compiled, multiple binaries are produced, each supporting different AER input and output formats. When called without any arguments it prints out its usage information, including which input and output AER format it supports (listing 6.7).

Listing 6.7: xio-bin-bin usage information

```plaintext
# ./xio-bin-bin
xio takes as mandatory arguments:
1. mode: { MONSEQ | MONONLY | SEQONLY }
2. the number of microseconds to monitor after sequencing
3. the device file, e.g. /dev/aerfx2_0
it reads from stdin and writes to stdout
input format: INPUT_BIN_U32U32LE
output format: OUTPUT_BIN_U32U32LE
```

Listing 6.8 shows an example invocation of xio. In this example, xio is invoked to operate both in monitor and sequencer mode (MONSEQ) on the first device connected to the PC, `/dev/aerfx2.0`. The sequencer data is read from the file `sequencedata.bin` via stdin in binary format, the monitor data is stored into the file `monitordata.bin` via stdout. The monitor is configured to run for five more seconds, after all sequencer data has been consumed.

Listing 6.8: xio-bin-bin monitoring-sequencing example

```plaintext
# ./xio-bin-bin MONSEQ 5000 /dev/aerfx2_0 <sequencedata.bin >monitordata.bin
```

6.4 Code Walk-Through – EP1 OUT Transfers for Monitor Control

As an example to illustrate, we will have a closer look at how all these parts play together, in the case of what happens when a user-space program such as xio wants to monitor data.

When a user space program wants to monitor data, it signals that when it calls `open(...)` to the driver by performing the open call with access mode `O_RDONLY` if it only wants to monitor, or `O_RDWR` if it wants to monitor and sequence.
If the access mode is either of these two flags, the driver immediately calls the function `aex_monitor(...)` presented in listing 6.9 with the argument `enable` set to one.

Listing 6.9: kernel module aerfx2.c: EP1 OUT monitor control

```c
void aex_monitor(struct usb_aerfx2 *dev, int enable) {
    int ret, rlen;
    const int urblen = 64;
    unsigned char urbdata[urblen];

    // AEX only:
    if ( ((dev->boardtype != BOARDTYPE_AEX) ||
          (dev->boardtype != BOARDTYPE_AEX_NEW)) ) return;

    if (enable) {
        dev_info(&dev->udev->dev, "aex_monitor:␣enable\n");
    } else {
        dev_info(&dev->udev->dev, "aex_monitor:␣disable\n");
    }

    urbdata[0] = EP1CMD_MONITOR;
    if (enable) {
        urbdata[1] = 0x01;
    } else {
        urbdata[1] = 0x00;
    }

    ret = usb_bulk_msg(dev->udev, usb_sndbulkpipe(dev->udev, 1),
                       urbdata, urblen, &rlen, 1000);

    if (ret) {
        dev_dbg(&dev->udev->dev, "%s,␣usb_bulk_msg:␣%d\n", __FUNCTION__, ret);
    }
}
```

This function then sends a control command out on EP1 with the function `usb_bulk_msg(...). The first byte of that message has the value `EP1CMD_MONITOR`, the second byte has a value of one to signal the AEX to enable monitoring.

The main loop of the FX2 firmware we presented in listing 6.2 then detects that incoming control command, and calls `process_ep1out_command()` which is shown in listing 6.10. This function then asserts bit 0 of FX2 GPIO port A to signal the FPGA to enable the monitor entity.

Listing 6.10: FX2 firmware: EP1 OUT command processing

```c
void process_ep1out_command(void) {
    switch (EP1OUTBUF[0]) {
    case EP1CMD_MONITOR:
        if (EP1OUTBUF[1]) {
            IOA |= PORTA_MONEN; // 0x80 | 0x01 = 0x81
        } else { // 0x80 & 0xFE = 0x80
            IOA &= ~PORTA_MONEN;
        }
    }
}
```
6.5 Testing and Benchmark Measurements

In order to test and benchmark the driver, the first approach was to configure the FPGA on an AEX board, to simply loop-back all the data it receives via USB, or to simply discard all data it receives as fast as possible or to generated data and send it to the PC via USB as fast as possible.

This was sufficient to perform the USB bandwidth measurements we presented back in table 4.4. However, this approach was insufficient to measure the performance of the aerfx2 kernel driver.

The initial approach was to measure CPU usage of the kernel driver while under the maximum load, that the hardware could sustain. The problem was simply that the CPU load caused by this load was too small to serve as a reliable measurement or benchmark for the driver.

We thus had to resort to a benchmark, which was independent of the USB hardware. The aerfx2 driver was thus modified, to loop back AER data right in the driver itself.

While this loop-back driver mode was initially introduced, to test

![Diagram of aerfx2.c: loopback benchmark test mode](image-url)
and verify that the driver functions correctly under high load, we then also used that mode for benchmark measurements of the aerfx2 driver.

Figure 6.5 shows how the driver moves around AER data and unused URBs in loopback mode. The dashed arrows are the data-paths disabled in loopback mode. They are replaced with two new data-paths (the red arrows). One loops back URBs containing AER data, the other handles the recycling of empty URBs.

The data-paths are chosen in a way that all components of the driver, which are USB-hardware independent, are still active and involved in this benchmark mode.

The results of these measurements are presented in table 6.1.

<table>
<thead>
<tr>
<th>PC / CPU Type</th>
<th>Loopback Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laptop: Lenovo X1 Carbon 2015 / Intel i7-5500U</td>
<td>5.0 GBytes/s</td>
</tr>
<tr>
<td>Aged Gaming PC / AMD Phenom II X6 1100T 3.3 GHz</td>
<td>5.7 GBytes/s</td>
</tr>
</tbody>
</table>

Table 6.1: aerfx2.c kernel driver loopback benchmark

These measurements show, that due to the zero-allocation architecture of the kernel driver, we are capable of moving Gigabytes of data per second through the driver, while our USB 2.0 high-speed hardware maxes out at 40 Mbytes/s.

Even on a ultra portable laptop, the driver can sustain a bandwidth which is more than two decimal orders of magnitude higher, than the USB hardware.
6.6 Conclusion

The USB interface software components described in this chapter: the FX2 firmware, the aerfx2 kernel driver, and the user-space code reference implementation are very successfully used with the AEX platform.

But not only. Besides the for the AEX platform, the same code could be used with very little modification with all other systems with AER over USB interfacing capability presented in this thesis:

- AEX (all versions)
- AEXS
- iCub/iHead
- AEXL/FX2

Even some systems which were developed by other people completely rely on that codebase for the USB interfacing capabilities. We will illustrate this by showing you examples in chapter 11.

Last but not least, in section 6.5 we could also demonstrate that the performance of the software components involved in the USB interfacing could sustain orders magnitude higher bandwidth, than USB 2.0 high-speed hardware provides.

USB 3 super-speed devices support bandwidths of circa 5 Gbit/s full-duplex, which amounts to about 1.25 Gbytes/s total. The numbers we have shown in the benchmarks indicate that the aerfx2.c could even easily keep up with USB 3 super-speed AER interface hardware, if such hardware will be developed in future.
6 USB Interface Software Implementation
Creating Complex and Reconfigurable AER Connectivity

Any non-trivial AER network connectivity requires some sort of “AER mapper system”.

To create more complex network topologies between neuromorphic chips, with arbitrary connectivity patterns, with fan-in and/or fanout greater than one, it is necessary to implement a way to map events from source addresses (typically the address of a firing neuron) to one or more desired destination addresses (typically the address of a synapse accumulating spikes). A variety of mapping approaches are possible, from algorithmic mappings to memory-based look-up tables.

Such mappings are usually realized with custom (digital) hardware devices, called “AER mappers”. While single neuromorphic chips typically offer little flexibility in the type of networks or architectures that they can implement (due to the mostly analog nature of the circuits used in these chips), neuromorphic Address-Event systems that use mappers to manage the communication of signals among its computing elements offer the advantage of reconfigurability: by changing the mapping algorithm, or the content of the mapper’s look-up tables, arbitrary network topologies can be implemented, learned, or evolved.

In theory multi-chip AER systems could have reconfigurable on-chip connectivity or even on-chip mapping functionality, without requiring the use of an explicit off-chip mapper. But reconfigurable mappings require usually significant amounts of memory. Hence the mapping functionality implemented on-chip is usually quite limited in practice, since on-chip memory is a quite costly resource due to its significant amounts of additional chip-area required.
To overcome this problem and extend the limited mapping capabilities of the neuromorphic chips in the system, it is necessary to use dedicated mapping devices. Such AER mappers commonly use dedicated RAM chips to store the mapping information and FPGAs to handle the I/O interfaces and to control the mapping process.

**Previous Publication:**

A lot of the material in this chapter is based on the publication [Fasnacht 11a], “A PCI based high-fanout AER mapper with 2 GiB RAM look-up table, 0.8 µs latency and 66 MHz output event-rate”, by Giacomo Indiveri and the author of this thesis.

This publication describes the “MMv2 AER Mapper System”, which has enabled a wide range of multi-chip experiments. The abstract states:

“Neuromorphic systems have been increasing in size and complexity in recent years, thanks also the adoption of the Address-Event Representation (AER) as a standard for transmitting signals among chips, and building multi-chip event-based systems. AER mapper devices that route Address-Events from multiple sources to different multiple destinations are crucial components of these systems, as they allow users to flexibly compose multi-chip setups, re-configure them, and program different architecture or network topologies.

In this paper we present a PCI based high-fanout AER mapper with 2 GiB RAM look-up table, 0.8 µs latency and 66 MHz output event-rate. Integrated with a PC it forms a very flexible and affordable AER experimental platform which is suitable for prototyping and research projects. Indeed, multiple instances of this system are already being used to perform various types of AER experiments. In addition, the system’s ability to implement probabilistic address-event mappings further extends the range of experiments that can be performed using this platform.

We describe the hardware system implementation details, compare our approach to previously proposed ones, and present experimental results which demonstrate how the system provides optimal performance for experiments with high average fanout and how for low fanout mappings the
limiting factor is given by the 0.8 µs latency induced by the PC on each random memory access.”

Because the MMv2 AER mapper system has enabled numerous multi-chip AER experiments at our institute and even at other research facilities, it also led to quite a few of publication by various researchers. Meanwhile there are probably tens of them. We’ve lost track at some point. Some exemplary publications will be presented in chapter 11.

7.1 Overview & Related Work

There are a number of ways one can imagine to map address-events. For each of them, the various hardware approaches, e.g. CPU/RAM architectures or FPGAs, are quite differently suited to implement a given approach. [Liu 15, p. 313, sec. 13.1.3] gives an extensive overview of mapping methodologies and devices.

7.1.1 Mapping Operations & Mapper Types

(adapted from [Fasnacht 07a])

A mapping operation takes an AER input stream an produces a different output stream by changing the addresses of the input events in some way. All the mapping operations shown here have in common that they do not operate on timestamps, but only on addresses. They either work on streams without timestamps, or if they work on streams on monitored data with timestamps attached, they never change the timestamp.

LUT Mapper – One-to-One

A LUT (Look-Up Table) mapper consists of a memory block used as a LUT. A given input address is used as a pointer in the memory block, and the value it points to is used as output address. This gives us a very versatile operation, but uses a sometimes significant amount of memory, e.g. in an FPGA where integrated memory is a very scarce resource in most cases. For LUT operating on n-bit addresses, n \cdot 2^n bits of memory are used to store the LUT. For an example of 16 bit addresses, the LUT uses 1 Mibit of memory.
Creating Complex and Reconfigurable AER Connectivity

Such a simple LUT mapper can also be looked at as a form of single-indirection data structure, where the input address is used as a pointer to the output address.

**Algorithmic Mapper – One-to-One**

An arithmetic mapper does not take a LUT, but a certain function / algorithm to translate the input addresses to the output addresses.

One very simple but very frequently used example is to add a certain offset to a given address, i.e. this can be used to make two AER streams non-overlapping before merging them. If we have two streams using the addresses \([0, X - 1]\), we could apply the operation \(+X\) to the one of the streams before merging the two together. The result would be combined a stream in the address range \([0, 2 \cdot X - 1]\).

Such mappings are usually very simple to implement in FPGAs, and in all the FPGA based systems we have presented so far, AEX, AEXS and eMorph/iHead, exactly this was very frequently done.

**One-to-Many Mappers**

One to many mapping generates many events out of one event. Thus it can for example be used to create one layer of a feed-forward neuronal network.

This function is implemented on the PCI-AER board, presented in 2.3.1, and on the FPGA based boards with additional RAM to store the mapping table. Often in such mappers the external RAM available to the FPGA is quite scarce and there usually is a limitation the number of events that can be mapped to, for example one input event can map to a maximum of 16 output events.

These one to many mappers are usually implemented using a double-indirection data structure, where the incoming address is used as a first pointer indicating the location of a second pointer pointing to a memory location that contains a list of addresses that are then used as the output.

**Complex Operations**

One can also think of complex operations, which could perform:
7.2 Mapper System Prototype I – ColEx

- statistical analysis of AER streams, reaction based for example on the mean firing rate of a group of neurons, or on correlation measures across AEs.
- other algorithmic operations of arbitrary complexity, e.g. learning evolving mapping schemes.
- probabilistic mappings, which e.g. map an event or discard it with a certain configurable probability. The MMv2 mapper we present in this chapter actually has this capability.

7.1.2 Related Work – Replacing the PCI-AER Mapper

The PCI-AER mapper system used for many experiments at our institute, which was presented in section 2.3.1 was certainly very influential for the work we present in this chapter.

Not only did we analyze the strengths and weaknesses of the PCI-AER mapper very thoroughly in section 2.4.1, it was always the goal of the work presented here to surpass and “retire” the PCI-AER mapper system by providing our researchers with something much higher performance, flexibility, stability (by not using parallel AER interfaces) and added features such as probabilistic mapping operations.

7.2 Mapper System Prototype I – ColEx

During the master thesis [Fasnacht 07a], the hardware for the prototype of a mapper system called ColEx was built. This system consisted of a “Colibri” System-On-Chip (SoC) module connected to a custom PCB with an FPGA and 3 GHz Serial AER interfaces (section 4.3) to allow communication with AEXv4 boards (section 4.1). The name “ColEx” stands for Colibri Extension board.

Figure 7.1 shows the ColEx hardware at the end of the master thesis. Software wise, the mapper was unfinished. The Colibri processor module and the custom developed ColEx PCB

Decision to Abandon the ColEx Mapper Architecture

During the master thesis, only the ColEx PCB could be finished. It could be verified that the FPGA and Serial AER interfaces were working correctly, however due to the limited time for the master thesis, the
Creating Complex and Reconfigurable AER Connectivity

Figure 7.1: ColEx mapper prototype from [Fasnacht 07a]. In red-dashed box: Colibri PXA Processor Module mounted on its huge evaluation board, in green-dotted box: ColEx (Colibri Extension) PCB built during the work on [Fasnacht 07a]. The ColEx contains a Spartan 3E FPGA in the center and two 3 GHz serial AER I/O interfaces (section 4.3).
work stopped there. Communication between the Colibri processor module and the FPGA on the ColEx PCB was never established.

It was considered to take up that work and finish the system as part of this thesis. However upon taking a closer look at the architecture, especially the Colibri processing module chosen, more and more doubt came up that following that approach was the right decision to make, first of all for some technical limitations in the Colibri module and the PXA270 processor it carries:

**Discouragement from the Hardware Front:**

- After the Colibri interfacing capabilities were studies more extensively in order to estimate the work required to establish a fast communication path between the Colibri and the ColEx FPGA, there were doubts on whether the I/O performance and processing power of the PXA270 processor would be sufficient for a mapping system satisfying our needs.
- Also, the 64 MiB SDRAM available on the Colibri SoC (system-on-chip) was considered on low end of what would be desirable for a future mapping platform.

But there were also some very discouraging developments in the supply chain with respect to both the PXA270 from Intel and Toradex, the swiss company building the Colibri modules.

More discouraging developments were observed on the software front for the PXA270 processor series, mainly with respect to the capability to run and effort required to install Linux on the Colibri module.

**Discouragement from the Supply-Chain and Software Front:**

- The XScale PXA series SoC architecture the Colibri module is based on was abandoned by Intel and sold to Marvell. This not only made the future of that architecture uncertain, but also made it difficult to find appropriate and current documentation for that architecture, because Marvell (at least at that time) was much worse at openly providing documentation to open-source software developers, than Intel has been.
- Maybe as a consequence, the Linux community lost quite a bit of the interest in that architecture, and support for embedded Linux distributions compatible with it, got limited to commercial subscription based suppliers.
Creating Complex and Reconfigurable AER Connectivity

- Even Toradex, the company selling the Colibri modules, ceased to provide their own Linux support for their modules and referred customers to third party providers, where one now would have had to buy Linux distributions working on the Colibri platform.

The combination of all these factors made it clear that the project to build a AER mapper based on the Colibri module and XScale PXA processor series had to be abandoned.

However the ColEx PCB which was built using a lot of resources, mostly time but also money, could be reused in the second mapper prototype, even though this mapper would have a completely different architecture.

This second prototype had the codename “ExCol” standing for Ex-Colibri Mapper System and will be presented in the next section:

7.3 Mapper System Prototype II – ExCol

![Image of Enterpoint Raggedstone I PCI FPGA board](https://example.com/figure7_2.png)

**Figure 7.2: Enterpoint Raggedstone I PCI FPGA board, © Enterpoint Ltd, reprinted with permission.**

**ColEx to ExCol:** It was determined, that instead of using the Colibri SoC module as a PC platform, we would attempt to replace that module with a regular PC architecture, providing gigabytes of cheap RAM and reasonably fast I/O interfaces.

Most of the work that went into the ColEx project until then, was the design and testing of the custom PCB comprising of a Xilinx Spartan 3E FPGA and two 3 GHz Serial AER interfaces.
Figure 7.3: ExCol mapper prototype assembly, front- & backside
To interface that board to a PC, we chose to go through the PCI bus, and to use a off the shelf FPGA board with a PCI interface, the Raggedstone I board from the british company Enterpoint (Fig. 7.2).

The ColEx PCB and the Raggedstone PCB were joined together using a standard prototyping PCB and wired up manually using coil wire. This new ExCol “contraption” is show in figure 7.3.

### 7.4 MMv2 Mapper System

The final mapper system was named “MMv2”. It was almost completely identical to the ExCol prototype, functionally even equivalent, but for the production AER mapper system some simplifications could be made.

The key simplification was that there were two FPGAs in the ExCol system: one on the reused ColEx PCB, one on the Raggedstone board. The only reason for having two quite powerful FPGAs in the prototype, was the reuse of the ColEx system together with the Raggedstone board left no other choice.

Because of that it was clear that in the final mapper system, all FPGA implemented functionality would be implemented in the FPGA present on the Raggedstone board, in order to remove the second FPGA.

Another decision was that only one daughter-board for the Raggedstone would be built, but in a way, that two of them could be mounted onto the Raggedstone (which has connectors for two daughter-boards).

Because of that, only one serial AER interface was to be present per daughter-board. Two would also have been very difficult to implement, due to the limited amount of pins available on the daughter-board connectors on the Raggedstone.

Eventually practice showed even that one 3 GHz Serial AER I/O interface would be sufficient. For the experiments researchers wanted to performed, because they would daisy-chain multiple AEX boards via serial AER links with the mapper closing the loop. Due to the very high speed of the serial AER interfaces compared to the typical event rates of chips used in multi-chip experiments, this ring topology never even imposed a performance limitation on an experiment.
7.4 MMv2 Mapper System

Figure 7.4: 3D rendering of the MMv2 PCB, a daughter-board with 3 GHz serial AER interfaces, mounted directly onto the Raggedstone FPGA board expansion connectors.

7.4.1 MMv2 PCB

After the ExCol prototype proved to be a promising approach, the MMv2 PCB (Fig. 7.4) was designed. Since the functionality embedded in the two FPGAs in the ExCol system could be implemented in one FPGA, a quite compact mapper system could be built.

Figure 7.5 shows the whole MMv2 system, comprised of the Raggedstone PCI FPGA board (blue), the MMv2 daughter-board (green) and the PC motherboard below. (The MMv2 PCB is rotated 90deg compared to the 3D rendering in Fig. 7.4)

7.4.2 Mapper Components

As displayed in figure 7.7 the mapper hardware thus consists of:

- a PC motherboard with 4 GiB of DRAM for storing the mapping table,
- a PCI card with ab FPGA receives the input data, performs the look-ups in the PC RAM, and sends out the results over the output AER interface,
- 3 GHz Serial AER interfaces for AER input and output,
The LUT is stored and managed in the PC motherboard with 4 GiB of RAM installed. All other components of a regular PC are kept to the bare minimum or even removed. A USB memory stick is used as a hard-drive replacement containing a minimal Linux setup that manages the download of mapping tables over the network.

The 3 GHz Serial AER interfaces (section: 4.3) are implemented using a custom daughter-board mounted onto the Raggedstone FPGA board. Figure 7.6 show the section of the MMv2 PCB where this interface is implemented. The two pairs of differential signaling coupled traces can be identified, as well as the island in the ground and supply plane, to which the 3 GHz signals couple, isolated from the rest of the supply- and ground-plane parts.

As mentioned, in the ExCol prototype, two FPGAs were present, one implementing the 3 GHz Serial AER interfaces, one implementing the PCI interface and the mapper functionality. In the MMv2, both these parts were merged and implemented in the FPGA present on the Raggedstone board.

That single FPGA contains logic to handle the Serial AER input and output, the mapping process, and the PCI interface enumeration and I/O in order to access the mapping table in the PC main memory.
Figure 7.6: HF-Section of the MMv2 PCB, showing the 3 GHz SAER LVDS traces. One can also identify the analog-VDD and analog-GND plane islands, separated by a gray gap from the rest of these planes.
Creating Complex and Reconfigurable AER Connectivity

Figure 7.7: Mapper hardware block diagram

CPU

Motherboard

PCI ↕

Southbridge ↕

Main Memory ↕

Main Memory, 4GiB DRAM

(reserved)

Mapping Table (2GiB)

Operating System

Spartan-3 FPGA

- PCI Enumeration
- PCI Target
- PCI Initiator (PCI Bus Master)

Serial AER I/O

->

SAER Tx

<-

3.125GHz

SAER Rx
7.5 Implementation Details

In this section we will go through the implementation details of the MMv2 mapper system. The PC hardware used and how it is used to program mapping tables will be presented first, then the FPGA implementation details, which next to the serial AER I/O and mapping logic includes a PCI interface implementation written completely from scratch and tailored to the task at hand.

7.5.1 PC Hardware

The PC hardware consists of a motherboard with the Intel P35/ICH9R chip-set, 4 GiB DDR2 RAM and a regular CPU.\(^1\)

Unlike in the most recent Intel & AMD CPUs, in this architecture the memory controller is not in the CPU, but in the north-bridge. If a PCI device needs to access the main memory the data-path is PCI ⇔ south-bridge ⇔ north-bridge ⇔ DRAM. This means that the CPU is not involved in the actual mapping operation. It is just used to run a minimal Linux environment to provide access to the system over the network. Graphics card, monitor or keyboard are not part of a typical setup as all user-interaction is done through the network.

7.5.2 Mapping Table Setup & Management

To set up the mapping table and store it in the PC memory, two things must be done: First we need to tell the Linux Kernel, that it must not use the memory space we intend to use. Not doing that leads to an obvious disaster. Second, we need to program the mapping table, i.e. write it to the memory we reserved in the format the FPGA expects to be present there.

Physical Memory Map

In order to reserve most of the main memory for the mapping table, the operating system is told to use only 1 GiB of RAM. This is achieved by providing the \texttt{mem=1G} statement\(^2\) as a Linux boot parameter via the GRUB\(^3\) boot-loader. The mapping table is stored in the next 2 GiB of

\(\footnote{\text{Motherboard: Asus P5K WS (Intel P35/ICH9R), RAM: Mushkin 2x2 GiB, DDR2 800 MHz (PC2-6400), CPU: Intel Core 2 Duo E6600 2.4 GHz}}\)
7 Creating Complex and Reconfigurable AER Connectivity

RAM. The top-most gigabyte is left untouched. BIOS and PCI devices reserve address regions just below 4 GB which must not be overwritten by the mapping table.

The resulting physical memory map is shown in table 7.1.

<table>
<thead>
<tr>
<th>Memory Range:</th>
<th>Used by:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 GiB – 1 GiB</td>
<td>Operating System</td>
</tr>
<tr>
<td>1 GiB – 3 GiB</td>
<td>Mapping Table</td>
</tr>
<tr>
<td>3 GiB – 4 GiB</td>
<td>Reserved for BIOS &amp; PCI</td>
</tr>
</tbody>
</table>

Table 7.1: Physical memory-map of the PC used by the MMv2

Setting Mapping Tables

The user can transfer a mapping table, or a script to generate a mapping table via the network. The memory region reserved for the mapping table can then be written to through an `mmap()` on `/dev/mem`.

Numerous researchers using the MMv2 mapper system have written their own software to produce, load and even partially update the mapping table. Some even being used to change the connectivity produced by the mapper online, e.g. to adapt mapping tables by some means of learning algorithm.

Mapping Table Format

A simple mapping table format is used. The input AE are 16 bits wide. This means that there are 64 Ki (2^{16}) different input AEs.

The 2 GB of mapping table space are split uniformly into 64 Ki pieces giving 32 KiB of RAM per possible input value. With 2 Bytes per AE this allows us to store up to 16 Ki of output AEs for a given input AE. A reserved sentinel address value (0xFFFF) is used to mark the end of a sequence of output AEs.

This means that one input AE can generate up to 16 Ki−1 output AEs, i.e. the maximum fanout of the mapper is ~16 Ki.

One advantage of this very simple mapping table format is that it requires only one contiguous access to the mapping table RAM in order

\[2\text{kernel-parameters.txt}, \text{Linux Kernel Source, http://kernel.org/}\]
\[3\text{GRUB boot-loader: http://www.gnu.org/software/grub/}\]
to get all the values for one input AE. This is beneficial, as random access to main memory is much more costly than sequential access.

Also the calculation to find the right memory address for a given input AE is simple: The numeric value of the input AE is multiplied with 32 KiB, and the mapping table offset (1 GiB) is added. For the FPGA this is an almost free operation as we multiply with a power of two, which is simply a bit shift. The resulting bit-shift and addition can be done in nanoseconds resulting in a single cycle operation in the FPGA.

The mapping table format is of course not very memory efficient. If a certain source address does not map to anything, the respective memory cannot be used for storing additional mapping information for another source address.

### 7.5.3 Mapper PCI Board

The basis for the custom mapper hardware is a modified *Raggedstone 1* PCI FPGA development board containing a Xilinx Spartan-3 FPGA (XC3S1500-4FGG456C). This FPGA connects to both the PCI bus and a custom daughter-board that implements the Serial AER interface (depicted in Fig. 7.5).

**FPGA Internal Architecture**

As illustrated in the FPGA block-diagram (Fig. 7.8) the FPGA contains logic to handle the Serial AER input and output, the mapping process,
and the PCI interface enumeration and I/O in order to access the mapping table in the PC main memory.

### 7.5.4 Custom PCI Implementation

The mapper uses a custom PCI implementation that was developed from scratch without using any IP-cores. This was motivated to get as much control over the PCI communication as possible and the ability to analyze and optimize any aspect of it.

**Implementing the PCI Spec from scratch:** Other PCI VHDL implementations and IP-cores were considered, but various readily available PCI implementations were analyzed and found to be not appropriate because they were either incomplete or did implement burst-read/-write transactions in a way that was not fast enough for our needs. PCI implementations without published HDL were ruled out because of the limited low-level customizability.

**Violating the PCI Spec for fun and profit:** In our PCI VHDL implementation we chose to ignore “Arbitration Latency” and the “Master Latency Timer” (PCI Specification 3.0, Figure 3-4, [PCI-SIG 04]). This means that other PCI bus-masters can be kept from getting access to the PC bus longer than usual. In the case of the PCI mapper this is a desired behavior though, as ongoing mapping activity should not be suspended to grant bus access to other PCI devices. Even PCI access requested by the CPU is not able to interrupt an ongoing mapping transfer, since in our case, priority clearly lies with reliable mapping timings, and not process / kernel execution latency, because the CPU / operating system is not involved in the mapping process.

Our custom PCI implementation has undergone practical testing in combination with a multitude of different motherboards and has shown to be working reliably. The key characteristics of the implementation are summarized in table 7.2.
### Example PCI Configuration-Read Transaction

Figure 7.9 shows a scope-trace of a basic PCI Configuration transaction (taken with a Tektronix TDS 3054B). The signals are the PCI bus signals Clock (33.3 MHz), FRAME, IRDY (Initiator Ready), TRDY (Target Ready). Clock and signals are source-synchronous. The traces match the PCI spec waveforms as desired (according to PCI Specification 3.0, Figure 3-4, [PCI-SIG 04]).

### Mapping and Serial AER FPGA Parts

While the PCI core takes most of the VHDL and FPGA logic resources used, the FPGA also implements the Serial AER interface and the mapper control logic. Both the mapper control and Serial AER interface contain FIFOs in the data-path (in blue/dashed in Fig. 7.8).

As the mapper interfaces to neuromorphic chips through the AEX, the Serial AER interface is identical to the same interface on that board. Both the AEX and the 3 GHz Serial AER interface are explained in detail in section 4.3.

The VHDL entities implementing the interface to the TLK3101 SerDes, namely TLKiface presented in 4.3.7 could be reused without modification.

The mapper control logic is straightforward because the mapping table format is very simple. For every input AE we calculate the right spot in the mapping table to read the output AEs. We then initiate a
Figure 7.9: PCI Configuration-Read transaction. Channels (1 to 4): PCI bus clock (33.3 MHz), FRAME, IRDY, TRDY. Trace length: 200 ns

PCI burst-read from the calculated address and read until we find the sentinel value, chosen to be 0xFFFF.

### 7.6 Probabilistic Mapping Mode

The mapper can also be configured to use an alternative mapping scheme that supports probabilistic mappings. In this mode, the mapping table data is interpreted as follows: instead of the default pair of 16 bit target addresses, the 32 bits are used to store 24 bit target addresses, a 7 bit probability value and one bit to mark the last word in a fan-out sequence (see Fig. 7.10).

The $2^7$ probability values are interpreted as uniformly distributed in the range $[0.0, 1.0]$ inclusive. This means that the probability can be selected in steps of $\frac{1}{127}$.

The trivial cases of probability zero ($0000000_2$) and one ($1111111_2$)

![Figure 7.10: Probability mapping word format](image)

Figure 7.10: Probability mapping word format
are implemented directly by discarding or unconditionally forwarding that address.

For the other values we draw from a pseudo random number generator (PRNG) that provides values in the range of \([0, 126]\) inclusive. If the probability value for the current address is strictly greater than the random number, the address is forwarded and discarded otherwise.

The random number generator is based on an open source PRNG implementation (project system-rng, available from http://opencores.org/). From this PRNG 7 bit values are drawn. If the value is out of our range (i.e. we draw the value 127) the step is repeated. A small number of random values that are in our desired range are stored in a FIFO, to reduce the probability that the process stalls because the drawing step has to be repeated.

The main PRNG is constantly kept running and not only active when we draw a random number from it. As the time from the start of the FPGA to the start of the mapping process is essentially random, there is no need to seed the PRNG registers explicitly.

Throughout the rest of the text the original mapping mode is discussed and not the alternate probability mapping mode. The performance numbers given are not affected by the probability mapping mode, except that the output bandwidth is reduced by a factor of two as we can transfer only one destination address per PCI bus cycle, i.e. the output event rate in the probability mapping mode is 33.3 MHz.

The probabilistic mapping version of the MMv2 mapper is usually referred to as “MMv2p” probabilistic AER mapper system.

### 7.7 Performance

Probably the best measure of performance of an AER mapper system is how successful it is in terms of enabling experiments that produce interesting and relevant results.

Because this performance measure is quite hard to establish, we will resort to measuring two key performance characteristics: latency and bandwidth, under a number of different mapping fan-out conditions.
7 Creating Complex and Reconfigurable AER Connectivity

7.7.1 Performance Measurements

In order to perform measurements of the latency and output-bandwidth of the mapper, a number of digital signals normally only available within the FPGA were routed to a debug connector and then used to record the scope traces in this section.

Latency

Figure 7.11 shows a scope trace taken to measure the latency of the mapper. The traces are signal denoted: MappingInProgress (top), ReadInputEvent (middle), WriteOutputEvents (bottom). The mapping table is set up to map each event to itself only, thus we have a fanout of one.

The trace starts with input data becomes available. ReadInputEvent is first active for one cycle to read the incoming data. Then MappingInProgress is asserted which means that the mapper issues a memory-read on the PCI bus. Now the mapper waits for the memory to deliver the data requested. When WriteOutputEvents becomes active for one cycle, this means that the mapper received the first word from the memory, in this case being 0xFFFXXXXX where 0xXXXX is the output event, and 0xFFFF the sentinel value. With the sentinel being detected the memory-read stops, and little after, the next input event

Figure 7.11: Input event to first output event latency scope measurement, mapping fanout of 1, trace length: 2 \(\mu s\), channels (1-3): MappingInProgress, ReadInputEvent, WriteOutputEvents.
is being read and processed.

The measurement figures on the right of the traces show the high-time of MappingInProgress signal, which is about one PCI bus cycle (30 ns), and the duration from the rising-edge on MappingInProgress to the rising-edge on WriteOutputEvents, i.e., the time from the input event being available until the first mapped output event is there (ca. 0.8 µs).

**Bandwidth Limits**

Figure 7.12 shows a plot of the same signals as in Fig. 7.11, but with a mapping table fanout of 64 and of 1024. Note that the timescale is different for each plot (see caption).

The observed latency of 774 ns at a fanout of 64 is the same as in the previous latency measurement for a fanout of one. The other measurement in the figures (CH3+Width) tell us that the mapper takes 983.7 ns to map and send out 64 events and 15.27 µs for 1024 events. We can calculate that the mapper actually maps two AEs per PCI bus-cycle (30 ns). This is as expected given that our AEs are 16 bit and the PCI bus transfers 32 bits per cycle.

We can conclude that the bandwidth bottleneck is the maximum physical limit of the PCI bus.
Figure 7.12: Bandwidth scope measurement at mapping fanout of 64 (top) and 1024 (bottom). Trace length is 4.0 µs (top) and 20.0 µs (bottom).
7.8 Conclusions

We will conclude by analyzing the limitations which apply to the MMv2 mapper system, but also other mapper systems with similar architecture, then by summarizing the key performance characteristics of the MMv2 mapper system we have established in this chapter.

7.8.1 Limitations

**LUT mapping only:** Although probabilistic mappings can be efficiently implemented on the current mapper, the mapper only supports look-up table mappings. Delay or algorithmic mappings are not very well suited for this system. These types of mappings require multiple non-sequential accesses to the main memory, each of which takes 0.8 µs as elaborated.

Such mappings cannot be done at comparable efficiency on the MMv2 architecture, and were thus not considered for implementation.

**Non-Simultaneous Fanout:** One issue with high-fanout mappings common to all fan-out mapping systems of the type presented here is that the order in which AEs are transmitted is always the same.

For high-fanout cases this can cause artefacts as an AE at the end of a fanout can be transmitted considerably later than one at the beginning of the fanout, even though that these AEs are theoretically assumed to be generated at the same time, i.e. the fanout is instantaneous and not serial.

**System Scalability:** One single central mapper as presented here does not scale well from a certain size of the system on. For a system with hundreds of chips for example, a central mapper is not feasible. Such systems need to be approached by distributed and/or hierarchical mapping schemes.

7.8.2 Summary

The MMv2 AER mapper system we described in this chapter is a hybrid system comprising both FPGA and PC hardware.

A key integrating component of the system proposed is the high-performance PCI interface implementation developed exclusively for
building this mapper. The mapper can store up to 2 GiB of mapping data and is thus suitable for high-fanout mapping applications.

Together with the AEX platform hosting neuromorphic chips and interfacing between chips and the mapper we provide a highly flexible system for performing multi-chip AER experiments.

We established measurements of the key characteristics: latency and bandwidth figures, demonstrating that the input-to-output latency is 0.8 µs, and that the limiting factor is given by the time required to access the main memory from the PCI interface. We demonstrated that the mapper can support a sustained output event-rate of up to 66 MHz and that this is the maximum supported by the PCI bus.

An overview of the MMv2 AER mapper system can also be found in the book: [Liu 15, p. 325, sec. 13.2.5]

### 7.8.3 Typical Performance in Experiments

A practical multi-chip setup such as [Neftci 10b] typically consists of a number of neuromorphic chips, each of which is directly connected to one AEX board, and an AER mapper. These components are then connected using the Serial AER interface in a loop topology. Each AEX board has a certain address-space assigned. If incoming AE fall into that space, they are sent to the local chip. Otherwise they are forwarded (kept in the loop). This allows the mapper to send AEs to all chips, and allows all chips to send AEs to the mapper. The full network connectivity is controlled at one single point, the mapper described here. Several examples of multi-chip systems have been built, with this mapper, using up to five multi-neuron AER chips.

In all experiments carried out up to now, the performance of the MMv2 mapper or the AEXv4 have never been considered to be the limiting factor.

Examples of publications resulting from experiments using the AEX and MMv2 mapper system will be provided in section 11.1.3.
AEXS – Towards Embedded and Robotic Applications

This thesis project was funded in part by the EU FP7 framework ICT-STREP project called “eMorph” (ICT-231467-eMorph).

The goal of eMorph was to expand the existing humanoid robotic platform “iCub” by a neuromorphic vision system, based on two DVS128 vision sensors [Lichtsteiner 08] for stereoscopic event-driven vision.

Some early processing stages were going to be implemented by a special CPU called “GAEP” (General-purpose Address-Event Processor), a SPARC-compatible processor developed by the Austrian consortium members of the eMorph project [Hofstaetter 10].

Multiple interfacing and processing stages were to be implemented using the technology we presented in the previous chapters, based on the AEX hardware and software platform.

For an embedded / robotic use of the AEX platform, we first developed a maximally shrunk down version of the AEX platform, called the AEXS (AEX-Small). The AEXS design focus was to develop a ultra-small form factor parallel AER to USB interfacing platform. Things not required in the robotic application we focussed on were removed from the AEX, such as the 3 GHz serial AER interface, but maximum compatibility to the AEX platform was retained in order to allow a maximum reuse of hardware designs and software (FPGA and PC-interface-side).

This chapter will illustrate the details of this AEXS platform, a miniaturized version of the AEX platform, aiming at being capable of being integrated into the iCub robotic platform, a goal of the eMorph project.
In the following chapter, we will then explain how the AEXS was integrated into the key hardware-component developed as part of the eMorph project, the so-called iHead PCB, which we will show you in the next chapter.

8.1 Embedding the AEX

Figure 8.1 shows the first and second revision of the AEXS PCB.

We will now discuss the details of the changes we had to implement to develop a first version of a AEX-based system suitable for embedded/mobile/robotic applications.

8.1.1 FPGA

Due to the space constraints, we were forced to use a different FPGA as compared to the Spartan 3E Series FPGA used in the AEX.

Normally an FPGA is a completely volatile chip, which has to load its configuration on power-up from another non-volatile flash storage chip, which usually resides next to the FPGA, and is controlled and programmed via the same JTAG interface.

The “Xilinx Spartan 3AN Series” FPGAs though offer an alternative solution here. The consist of a multi-die package, e.g. a packaged chip, which contains more than one pieces of silicon with VLSI circuitry (die).
In this case, the non-volatile flash storage chip was directly integrated into the same package as the FPGA.

In addition to this change, we had to use a smaller package, and thus an entirely different soldering technology, a BGA-package (Ball-Grid-Array).

Figure 8.2 shows various package types for comparison. The leftmost chip is the Spartan 3E used in the AEX system, the BGA chip next to it is the Spartan 3AN used in the AEXS.

The Spartan 3E used in the AEX platform in its PQ208 package measures $30 \times 30 \text{ mm}^2$ while the Spartan 3AN used in the AEXS platform in its FT256 BGA package only measures a mere $17 \times 17 \text{ mm}^2$.

Even though it is smaller, the BGA package has more pins, 256 up from 208.

### 8.1.2 BGA board design & soldering

In figure 8.3 we can see a rendering of the top side of the AEXS PCB, zoomed in on the footprints of the FX2 USB interface chips with traditional surface-mount pins on each side of the chip (left side of figure), next to it the BGA footprint for the FPGA in its FT256 package.
In figure 8.4 we zoom further in to the BGA footprint and have a close view at it from the top side. The green layer is the solder-stop mask. Where it doesn’t cover the yellow top copper layer, the solder-balls of the BGA package will be soldered to. Diagonally next to most of these BGA pads, there is via structure connecting to lower layers of the PCB, either directly to the copper layer below, a power/ground plane, or further down.

This combined structure of the BGA pad and the via diagonally next to it is called a “dog-bone” layout. It is the most common and easiest to manufacture BGA contacting layout.

It is important to note, that these vias are covered with solder-stop mask, unlike any other vias on the PCB. This is important to prevent soldering mistakes with the BGA solder being sucked into the vias. This special type of vias required for BGA soldering are called “tented-vias”.

In figure 8.5 we have another look at this structure, as seen from within the PCB, with the virtual camera located between the top-layer and the first (top-most) inner plane. The z-axis in this rendering is over-scaled for illustrative purposes.

All soldering and assembly of the AEXS boards was done in-house,
8.1 Embedding the AEX

Figure 8.4: Zoom to the BGA footprint, view from above

Figure 8.5: Zoom to the BGA footprint, view from within PCB, just below top layer (z-axis over-scaled)
using a simple re-flow soldering oven. No solder paste was used to solder
the BGA packages, but a mere thin layer of flux-grease was applied
before soldering.

Since the finish of the copper layer again was gold-plated, soldering
the remainder of the components in a second step was easily possible.

8.2 AEXS Interfaces

We will now discuss the interfaces present on the AEXS. While the
USB interface to connect the AEXS to a PC was left almost completely
unchanged from how it was implemented on the AEXv4 system, there
were considerable changes to the other interfaces.

The serial AER interface section of AEX was left out due to space con-
straints and because there were no plans to integrate other components
into the iCub robot using a serial AER interface.

The parallel AER interface on the other hand was expanded in its
flexibility and capability in terms of number of AER data-lines. While
the AEX had one parallel AER input one one output, the AEXS has
two parallel AER interfaces, which each can be individually configured
to either act as a AER input or output interface.

8.2.1 Serial AER

It was quickly decided that the 3 GHz Serial AER interface present in
the AEX would not be required in the AEXS, and also would not fit
the space available in the robot head. It was thus not implemented in
the AEXS.

However there were tentative plans, to experiment with a novel low-
speed serial AER interface, where the FPGA would directly produce
the LVDS signals.

For this reasons, the AEXS platform has two 8P8C modular connec-
tors (commonly known as RJ45 connectors) on it, as can be seen in
figure 8.1.

Because it later turned out, that no other eMorph consortium member
was implementing a component using that same low-speed serial AER
interface, work on this interface was abandoned.
Because the AEXS was built in a way that allowed for this now unused section to be simply cut off, the AEXS could be modified to have an even more compact form factor.

### 8.2.2 Parallel AER

While the AEXS lacks the serial AER capabilities of the AEX platform, there are some improvements with respect to the parallel AER interfacing capabilities.

First of all, while the AEX is meant to have only one parallel AER input and one parallel AER output, in the AEXS, the two parallel AER interfaces can be either input or output interfaces, allowing also for two inputs or two output interfaces.

In addition, the number of AER data lines was increased by using up free board space and I/O pins of the FPGA. The AEXSv1 has two parallel AER input/output interfaces with up to 20 data-lines each. With the AEXSv2 we went even further: it has two parallel AER input/output interfaces with up to 26 data-lines each.

Figure 8.6 shows an AEXS board with the 8P8C connector section removed, and one parallel AER and the USB interface connected.
8.2.3 USB Interface

As mentioned, the USB interface was left unchanged and is identical to the USB interface present on the AEXv4.

8.3 Results & Conclusion

We will now summarize the characteristics of the AEXS system, and then conclude the chapter, in explaining how the AEXS came to further use.

8.3.1 Results

In order to put some key characteristics of the AEXS system presented into context, we will now compare it to a very similar system we presented as related work influential to the AEX platform back in chapter 2.

Comparison to the USBAERmini2

The AEXS system has a quite similar functionality compared to the USBAERmini2 board from the CAVIAR project we introduced in section 2.3.2 and can be seen in figure 2.6.

The AEXS is superior in many aspects though. Table 8.1 compares the USBAERmini2 and the AEXSv2 systems.

<table>
<thead>
<tr>
<th>AEXS / USBAERmini2 Comparison</th>
<th>USBAERmini2</th>
<th>AEXS (v2, no RJ45)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCB Area</td>
<td>159.2 cm²</td>
<td>19.8 cm²</td>
</tr>
<tr>
<td>Parallel AER Interface</td>
<td>1 input, 1 sniffer output, 1 output</td>
<td>2 inputs/outputs (selectable)</td>
</tr>
<tr>
<td>Parallel AER Interface Data-Width</td>
<td>2 * 16 bit</td>
<td>2 * 26 bit</td>
</tr>
<tr>
<td>Chip Cost, 2015</td>
<td>$31 (FX2 + CPLD)</td>
<td>$34 (FX2 + FPGA)</td>
</tr>
<tr>
<td>FPGA/CPLD</td>
<td>XC2C256 CPLD</td>
<td>XC3S200AN FPGA</td>
</tr>
</tbody>
</table>

Table 8.1: AEXS / USBAERmini2 comparison
It is notable to say, that due to the drastically smaller PCB are used, it is most probably cheaper to manufacture than the USBAERmini2, even though the chips are slightly more expensive, and the PCB technology required is more advanced.

Also with respect to capacity to implement register transfer level VHDL designs as used in both these systems, the FPGA used in the AEXS can hold at least 5-6 decimal orders of magnitude more complex designs when compared to the CPLD used in the USBAERmini2.

### 8.3.2 Conclusion

The AEXS was successfully developed as a miniature version of the AEX platform. Due to the careful choice of modifications made, a maximum of compatibility to the AEX platform as achieved.

In addition, the AEXS also can be considered as the first prototype of the iHead system, integral to the eMorph project, as we will see in the next chapter.

**AEX / AEXS compatibility**

Due to the almost complete backwards-compatibility to the AEX code-base used in the FPGA and for the USB interface, almost all software developed for the AEX platform could be reused without any modification or customization for the AEXS platform.

A computer attached to an AEX or AEXS system also uses the exact same driver and software stack and cannot tell the difference between the two systems, unless programmed to do so.

**Prototype for the eMorph iHead system**

Eventually, the AEXS design was used as a direct prototype to parts INI contributed to the iHead board of the eMorph project, which will be our next focus.
In the previous chapter we described the AEXS, a system derived from the AEX platform, optimized for minimum size in order to be able to be embedded into size and weight constraint systems such as a mobile robotic system.

In this chapter, we will now explain how that previous work was integrated to form an integral part of the eMorph project, the “iHead” system, forming the core of a neuromorphic visual pathway of for the iCub robotic platform.

Some of the material presented in this chapter is based on the publication [Bartolozzi 11b], by Chiara Bartolozzi, Francesco Rea, Charles Clercq of the “Italian Institute of Technology” (IIT) in Genoa, Michael Hofstätter of the “Austrian Institute of Technology” (AIT) in Vienna, Giacomo Indiveri and the author of this thesis of ETH and University of Zurich and Giorgio Metta of the “Universita degli Studi di Genova”, a joint publication by many of the eMorph project members.

9.1 iCub Robot

The iCub is a humanoid robotic platform which was developed by the EU FP7 funded “RobotCub Consortium” [Sandini 07, Metta 10]. It is built and can be bought from the IIT for about a quarter million Euros. It is used as a research platform in many European research facilities, but by also American and Asian labs.

The child-like robot is about one meter in height and has with 53 degrees of freedom of movement very extensive motor capabilities. As
of 2010, twenty iCub robots have been built and are being used in numerous research facilities.

Quoted from [Sandini 07]:

“[The iCub humanoid robotic platform is the result of] a multi-disciplinary initiative to promote collaborative research in enactive artificial cognitive systems by developing the iCub: a open systems 53 degree-of-freedom cognitive humanoid robot. At 94 cm tall, the iCub is the same size as a three year-old child. It will be able to crawl on all fours and sit up, its hands will allow dexterous manipulation, and its head and eyes are fully articulated. It has visual, vestibular, auditory, and haptic sensory capabilities. As an open system, the design and documentation of all hardware and software is licensed under the Free Software Foundation GNU licences so that the system can be freely replicated and customized. We begin this paper by outlining the enactive approach to cognition, drawing out the implications for phylogenetic configuration, the necessity for ontogenetic development, and the importance of humanoid embodiment. This is followed by a short discussion of our motivation for adopting an open-systems approach. We proceed to describe the iCub’s mechanical and electronic specifications, its software architecture, its cognitive architecture. We conclude by discussing the iCub phylogeny, i.e. the robot’s intended innate abilities, and an scenario for ontogenesis based on human neo-natal development.”

9.2 eMorph – Overview and Hardware Architecture

As you can clearly see in figure 9.1, the iCub robot wants a DVS sensor chip based neuromorphic visual pathway ;-) 

The EU FP7 framework ICT-STREP project called eMorph (ICT-231467-eMorph) aimed at building exactly that, in order to investigate, what can be achieved with a minimalistic neuromorphic visual system on the iCub robot, instead of using traditional frame based camera stereo-vision as in most humanoid robots.

Figure 9.2 shows the structure of the optic pathway in a human.

Let’s illustrate the architecture used in the neuromorphic visual pathway using an neuroanatomical analogy:
Figure 9.1: eMorph: The iCub Robot wants a neuromorphic visual pathway. Collage of the iCub robot and two DVS128 neuromorphic event-based dynamic vision sensor chips. (robot photo adapted from http://robotcub.org/)

Figure 9.2: Anatomical optic wiring diagram from retina to V1, © by Ratzinum@en.wikipedia.org, licensed under: CC-BY-2.5 & GNU-FDL
9.2.1 The Eyes: DVS128

The neuromorphic eyes of the iCub in the eMorph project are two “tmpdiff128” chips as seen in figure 9.1 (not to scale, obviously) and previously in figure 2.2 which presented the “DVS128 PARALLEL AER” board. For an overview of neuromorphic vision sensors and a description of the sensor use refer to [Lichtsteiner 06, Lichtsteiner 08, Delbruck 10b]. An extensive overview of silicon retinas can also be found in [Liu 15, Chapter 3 – Silicon Retinas].

These two neuromorphic vision sensors replace the cameras usually mounted in the movable eyes of the iCub robot. These two neuromorphic eyes are then connected via very flex-PCB cables to a PCB in the back of the head of the iCub, the so-called “iHead” PCB, which also contains most of the next components the neuromorphic visual pathways consists of.

9.2.2 Optic Nerves, Optic Chiasm and Optic Tract:
AEXS FPGA AER processing

This part of the visual pathway will be implemented in an FPGA based on the AEXS system presented in the last chapter. The two parallel AER streams from the “eyes” will be fed into the FPGA via two parallel AER inputs, then they will be tagged and merged, eventually the AER data will be forwarded via a parallel AER output interface to the next stage, the GAEP.

The FPGA will also control the bias settings of the DVS sensors.

9.2.3 LGN (Lateral Geniculate Nucleus):
GAEP (General Address-Event Processor)

The GAEP will receive the merged AER streams from the FPGA via its sensor interface (SIF).

Similar to the LGN (called simply Geniculate Nucleus in figure 9.2), it will be the first node where complex operations on the visual input can be performed. Because the GAEP contains a fully programmable SPARC CPU, there are little limits to what processing can be implemented at stage, as long as the CPU is capable of performing it fast enough.

The resulting data will then be forwarded to the FPGA (the same FPGA as before).
9.2.4 Optic Radiations: AEXS FPGA & USB Interface

The AEXS FPGA receives the output of the GAEP via a custom high-performance bi-directional interface specifically tailored to interface with the GAEP.

The FPGA then timestamps the AER data it receives in the monitor, and forwards it over the high-speed USB interface to the next stage.

9.2.5 V1 and Higher Visual Processing Areas: iKart PC

The final and most powerful processing stage of the visual pathway is a high-performance PC which is integrated in the mobile platform carrying the iCub, the so-called “iKart”.

It receives the data produced by the GAEP via USB, but can also issue various control operations via sending command back over that USB interface, e.g. to control / change the bias values of the DVS sensors, or by sending arbitrary commands and data to the processing firmware running in the GAEP.

9.2.6 Overall Architecture Overview

Figure 9.3 summarizes all the components of the neuromorphic visual pathway we have just presented in a feed-forward simplified data-flow diagram.
9.3 Architecture Details

As mentioned, the FPGA “inherited” from the AEXS system is used in two stages of the visual pathway. The previous figure 9.3 only illustrated the visual data-pathways.

Figure 9.4 show a more detailed block diagram of the entire architecture, including control data-paths and the reverse data-path over USB from the iKart PC back via the FPGA back to the GAEP. It also shows a Laptop that can be connected directly to the GAEP for programming and debugging the SPARC processor.

As mentioned before, the DVS sensors are place in the eye-bulbs of the iCub robot, the PC which can perform the highest level processing of the visual input and can control the entire robot is built into the mobile iKart base platform of the iCub.

The remainder of the components shown in the block diagram in figure 9.4 are built onto a new common PCB called iHead which is built into the scarce space in the back of the head of the iCub.

The iHead PCB incorporates:

- two parallel AER input connectors for the flex cables coming from the two DVS sensors,
- the Spartan 3AN FPGA,
9.3 Architecture Details

Figure 9.5: iHead PCB – front view

- the GAEP general address-even processor,
- SRAM and FLASH chips required by the GAEP,
- various support components and
- power, data and debug interface connectors, including the USB interface connecting to the iKart PC.

Figure 9.5 shows the iHead PCB front side, figure 9.6 its back side. This particular iHead PCB depicted is a test-system for the FPGA and USB interface parts of the system, because of that the GAEP chip is not assembled in that particular PCB (thus the huge empty space on the front-right of the PCB).

9.3.1 AEXS to iHead

Figure 9.7 shows the front side of the iHead PCB with an overlay labelling the most relevant parts.

The FPGA, USB interface and parallel AER interface where the two DVS sensors are connected encased in the red dashed rectangles are the parts of the iHead system design, which were taken directly from the AEXS which we presented in the previous chapter.
Figure 9.6: iHead PCB – back view

Figure 9.7: iHead PCB – front view with description overlay
9.4 GAEP

In their publication “An integrated 20-bit 33/5M events/s AER sensor interface with 10ns time-stamping and hardware- accelerated event pre-processing”, [Hofstaetter 10], the developers of the GAEP describe the chip as:

“[The “GAEP” is] a general purpose address-event (AER) processor based on a SPARC-compatible LEON3 core with a custom data interface for asynchronous sensor data. The main focus in the design of the sensor interface was on precisely maintaining the inherent timing information of AER sensor data while providing robust peak-rate handling, DMA functionality and a novel event-rate dependent system control mechanism. Hardware-accelerated event pre-processing includes pre-FIFO high-resolution time-stamping, address masking for ROI and event-rate dependent IRQ generation without loading the processor core. The System-on-Chip has been implemented in a 0.18µm CMOS process and achieves peak AER input event rates of 33 MAE/s and sustained event rates of 5.125 MAE/s at 10 ns time-stamp resolution. The core processes AEs at >1 MAE/s sustained rate. We discuss design considerations and implementation details and show measurement results from the fabricated chip.”

(By Michael Hofstätter, Peter Schön, Christoph Posch from AIT - Austrian Institute of Technology GmbH, Vienna). For further details on the general address-event processor and its application also refer to [Hofstatter 09, Hofstatter 11].

9.5 iHead FPGA

In figure 9.4 we have already revealed some of the internals of the FPGA. Let’s now have a look at the details of the elements of the system described which are implemented in the FPGA.

Figure 9.8 shows a detailed block diagram of the FPGA internal architecture. Interface components are highlighted in orange, FIFOs in striped blue, internal HDL entities for processing in green.

In the upper half of the diagram, you can see parts which we have previously compared to the Optic Nerve, Chiasm and Tract. The DVS
eyes are feed their AER streams into the two parallel AER interface of FPGA, then at point (A) and (B) the two streams are tagged for later processing stages to be able to distinguish input from the left and the right DVS eye. The two streams, not using non-overlapping address-ranges, are then merged and the combined stream (C) is then forwarded to the GAEP via a parallel AER output interface.

![Figure 9.8: iHead: FPGA internal block diagram](image)

In the lower half of the block diagram, the one we compared to the Optic Radiation before, we can recognize a structure, we have already seen way back in the AEX in figure 4.6, the USB interface together with two large FIFOs and the AER monitor and sequencer units.

Most HDL entities present here were generic enough already as they have been designed for and used in the AEX codebase, such that they could be re-used in the iHead FPGA with little to no modifications.

Only the “GAEP Interface” structure is completely new here. The GAEP communicates with a proprietary 32-bit half-duplex synchronous
bus interface with the FPGA. The VHDL entity \texttt{GAEPiface} was coded to implement the FPGA side of that interface and enable bi-directional communication between the FPGA monitor/sequencer and the GAEP. We will present the \texttt{GAEPiface} in 9.6.

So far we have looked at two completely independent sections in the FPGA, as it was illustrated back in figure 9.3, which only showed the feed-forward optical pathway. The entity named “Bias Control via AER” makes the only exception here. The purpose of this entity is to set and modify the bias values of the DVS sensors. This is done by a very simple protocol, which allows to control the PC connected to the iHead board, to set the bias values using a special reserved range of address-events for that purpose. The advantage of this in-band configuration is that it uses a data-path already present in the system, and thus can be implemented with little extra effort, both on the FPGA but also on the PC side.

### 9.5.1 Special FPGA Configuration for DVS/GAEP Debugging

Figure 9.9 shows the block diagram of a completely different FPGA configuration. Even though the configuration is very different in its architecture, it can be generated from the same code used to produce the previously presented FPGA configuration developed for the iHead, by changing a simple build-time switch.

The configuration was build to be able to debug the DVS sensors and the GAEP, and has completely different data-paths compared to the production FPGA configuration shown in figure 9.8. The output of the DVS sensors are not forwarded to the GAEP, but instead fed directly into the monitor. This allows for the PC attached to receive and visualize the raw data the iHead system receives from its two DVS sensors, without the GAEP interacting in any way with that data.

On the other hand, the sequencer now also is connected to the GAEP sensor interface (where the DVS data would usually end up), which allows us to send pre-recorded or created test-input to the sensor interface (SIF) of the GAEP. This can be very helpful in order to debug any issues with both the DVS sensors or the GAEP. For example it allows to supply the GAEP repeatedly with exactly the same data on the sensor interface, which allows for reproducing completely identical SIF input to the GAEP, which otherwise would not be possible.
Figure 9.9: iHead: FPGA DVS/GAEP debugging configuration: directly monitoring DVS input, sequencing to GAEP.
9.6 FPGA – GAEP Interface

Listing 9.1 shows the entity declaration of the FPGA side implementation of the FPGA – GAEP interface.

Signal vector GaepDataxDIO is the 32 bit bi-directional data-bus which the GAEP and the FPGA. GaepIOSNxSBAI, GaepReadEnxSBAI and GaepWriteEnxSBAI are produced by the GAEP who acts as the interface master, while the FPGA who acts as the interface slave produces the signals GaepReadFifoEmptyxSO, GaepReadFifoEmptyxSO and GaepReadFifoEmptyxSO to inform the GAEP about the FIFO fill-levels in the GAEPiface and newly arrived data, by triggering an interrupt.

Details on how the interface works are documented in the comment preamble of listing 9.1.

On the “other” side, where GAEPiface connects to the monitor and sequencer, it simply provides two RR-type interfaces (as defined in section 5.1.1), named with the prefixes ToGaepFifo and FromGaepFifo.

Listing 9.1: VHDL entity declaration of GAEPiface

```
-- GAEPiface
-- from-GAEP signaling
-- The GAEP must only start sending (up to 32 words) if
-- GaepWriteFifoFullxSO is deasserted.
-- If the GaepWriteFifoFullxSO is deasserted, the GAEP can transmit
-- up to 32 words without re-checking GaepWriteFifoFullxSO.
--
-- to-GAEP signaling
-- GaepReadFifoEmptyxSO always signals whether data is available for
-- reading by the GAEP. In addition, whenever data becomes available
-- we signal an interrupt by generating a *falling* edge on
-- GaepInterruptxSO.
-- GaepInterruptxSO thus can be simply GaepReadFifoEmptyxSO.
-- IOSN acts as a chip select. We drive the data bus whenever
-- GaepReadEnxSBAI is asserted, until it is deasserted.
--
library ieee;
use ieee.std_logic_1164.all;
```
9.7 Results & Conclusion

During a visit at AIT in Vienna, together with Michael Hofstätter, we optimized the performance of the iHead system interfaces to the maximum achievable by tuning both the GAEP configuration and the FPGA interface timing configuration.

Both the interface between the GAEP and the FPGA and the USB interface timings between the FX2 and the FPGA were optimized and verified to function at optimal performance. This was done on an elaborate test setup for the iHead PCB (figure 9.10, where the DVS sensors were replaced with special sequencing hardware to produce artificial DVS data, and a logic-analyzer and oscilloscope were used to observe the interface performance while tuning the configurations.

9.7.1 GAEPiface Tuning and Performance

After GAEP and FPGA tuning, we could operate the interface between them at the maximum speed supported by the GAEP, that is with a minimum wait-state configuration of “1”.

Figure 9.11 shows that interface in action, with the GAEP reading 32 bit words from the FPGA at maximum speed, until the signal
RdFifoEmpty is asserted by the FPGA, and the GAEP reads the final 32 bit word known to be left in the FPGA FIFO.

The horizontal grid dimension of this logic-analyzer trace is 500 ns, and we can see that the GAEP reads a 32 bit word circa every 480 ns. From this we can calculate that in this measurement the GAEP reads data from the FPGA at a rate of about 66 Mbit/s.

9.7.2 FX2 USB Interface Tuning

Given the chance we already had the test-bench ready, we also checked the timing on the interface between the FPGA and the FX2 USB interface chip.

Figure 9.12 shows a digital oscilloscope recording of the 48 MHz interface clock signal (the clear sinusoidal signal in the top of the plot) and in the bottom part, the SLWR signal, which indicates that the FPGA writes data to the FX2 IN FIFO, is plotted. The scope is set to trigger on the middle of the rising edge of the interface clock signal.

The oscilloscope is set to infinite persistence mode, which means that all traces ever plotted are kept in the image. The result is a so-called
Figure 9.11: iHead: FPGA – GAEP Interface Tuning.
Logic-analyzer channels 8 through 12 (bottom-up on the display) are: RdFifoEmpty, WrFifoFull, IRQ, IOSN, WrEn/RAMOEEn, Channels 13 to 15 are unused in this logic-analyzer recording.

eye-diagram of the SLWR signal. This means we can observe the SLWR signal in all the states has during the recording.

The medium-to-bright horizontal area in the middle of the plot is all the voltage values SLWR takes while being “high”, the equivalent horizontal area at the bottom are all the traces SLWR created while being low.

Most important though are the edges, where SLWR makes a high-to-low or low-to-high transition. The two empty areas we observed between the three SLWR transition areas are the “eyes” which give the eye-diagram its name.

There are two important observations we can make about the timing of the SLWR signal: First, the transition areas are quite narrow, clearly defined, and always happen in the range of about 1-4 ns after the clock rising edge, which is exactly what we want to match the timing specification of the FX2.

In addition we can observe, that the “eyes” are wide open both horizontally and vertically, between the high- and low-band and the transition areas. This means we have a stable and clear SLWR signal generated by FPGA, a requirement for a stable and reliable interface between the FPGA and the FX2 USB chip.
9.7 Results & Conclusion

9.7.3 iCub with iHead Neuromorphic Visual Pathway

After all components have been verified to function according to our expectations, final assembly and installation into the iCub robot could finally be started.

Figure 9.13 shows the complete and operating iHead PCB with both DVS sensors attached prior to the installation into the head of the iCub.

Figure 9.14 shows the iCub, with the new neuromorphic visual pathway fully installed and operational, the iHead PCB in the back of the head, and the DVS sensors in the eye sockets.
Figure 9.13: iHead PCB completely assembled with DVS eyes attached
Figure 9.14: iCub on iKart with all eMorph expansions added
On the top of the head, you can see that the eMorph iCub carries “winglets”. These were added, in order to carry the conventional cameras which had to be removed from the eye sockets in order to make space for the neuromorphic vision sensors. Having both types of sensors still available on the iCub was made to ease experimentation with the novel visual pathway by also still having the original camera images available for testing and comparison.

9.7.4 Publications & Conclusion

As of July 2015, the eMorph Consortium website http://emorph.eu/ list no less than 16 publications resulting from this project, many of them being collaborative works by researchers from multiple institutes. Among them are: [Bartolozzi 11b], [Bartolozzi 11b] and [Rea 13].

Adding a novel neuromorphic visual pathway to the iCub humanoid robotic platform opened up many new perspectives for research in how such a neuromorphic visual processing system can be used to expand the capabilities of the iCub robot in combination or in comparison to the traditional frame-based camera approach.

The main success with respect to this thesis was that almost all the systems and components that were presented in the previous chapters, the AEX, the AEXS, VHDL codebase for AER processing and USB interfacing hardware and software could be adapted to the task at handi. Combined with the GAEP processor and DVS sensors, they were successfully integrated into the iCub humanoid robotic platform.
After the completion on the work we described in this thesis so far, a new goal was formulated: to collect everything we learnt from building all these systems, including what we learnt about them from the experiences of researchers using it on a daily basis to perform their experiments, in order to then compile from that information the design requirements of a next generation platform for neuromorphic multi-chip experimentation systems, which would exceed the combined capabilities of all the systems presented in this thesis so far.

This new endeavour was name the “AEXL Project”, AEX-Large in contrast to the “AEXS”.

In this chapter we will first define the goals of the AEXL project, and explain how they were based on the strengths but also deficiencies identified in our previous systems, and the simple desire to improve on performance of a certain capability of the previous systems.

Then we will introduce the FPGA board which we chose to form the basis of the AEXL project, and explain why that specific board was chosen.

In a first step, the goal was to reproduce most of the functionality that we had in our previous platforms on the new AEXL system. For this purpose, a number of expansion boards were designed for the AEXL baseboard. We will describe these expansion boards. We will show you their purpose and which feature of systems previously built they now made available on the AEXL platform.
The we will also discuss a number of hypothetical applications of the AEXL platform, and show you how various expansion boards and neuromorphic chips can be combined on even a single AEXL baseboard. Finally, we will conclude by showing you with the current status of the (ongoing) AEXL project.

10.1 Goals of the AEXL Project

First we formulated and presented a number of goals we wanted to achieve in the AEXL project, which also were the motivation to even consider conceiving a quite drastically new approach, based on the lessons learnt from experience with building and using our previous platforms, but also with being open-minded to rethink every aspect of our previous architectures.

We will now try to summarize these goals as part of three distinct categories, and explain, how they came to be.

10.1.1 Code-Reuse & Backwards Compatibility

Trying to build systems with sufficient backwards compatibility to allow for a significant amount of code-reuse has been very successful design approach, as was illustrated for example by the progression from AEXv4 to AEXS to the iHead FPGA configuration, while still employing the same code in all those systems, with minimal adaptations and extensions.

We thus also wanted to use that strategy when implementing the AEXL wherever possible, even though it was clear that with the AEXL, all aspects of architecture and interfaces present on the systems were open for discussion and were to be challenged, whether they are still feasible in that next generation of AER interfacing and processing platform.

10.1.2 Improved Extensibility by Increased Modularity

The AEX platform was not very extensible. It had the interfaces that were required for the purpose it initially was meant to serve, plus a small number of GPIO signals which could be controlled directly from the FPGA, e.g. for address-event interfaces wider than the usual 16 bits, or to control a digital bias generator of a chip.
10.1 Goals of the AEXL Project

However, already due to the lack of having more IO pins available on the FPGA of the AEX platform, there was no further extensibility option, such as a connector to attach daughter-boards (extension boards) on top of the AEX.

In order to give researchers more freedom to extend the AEXL base board, a much more modular system had to be built. It was thus decided, that the AEXL should be much more modular than the AEX has been. If somebody for example would not have the need for parallel AER connectors or a USB monitor-sequencer interface, these things should now be modular components of the AEXL systems, which one could add or remove as desired.

10.1.3 Improved Scalability

As explained in chapter 7 and as will be illustrated in figure 11.3, the communication topology people used to build out of their MMv2–AEX–AMDA systems, was a loop of serial AER links, where address-events could even travel only in one direction, i.e. as a participant in such a system to reach your the neighbour “behind” you, you would have to send your AEs through the entire circle.

Nonetheless this systems was used very successful for numerous experiments resulting in numerous publications. However these experimental setups usually employed no more than a handful of neuromorphic chips. Because of that, and because of the very high speeds of the serial AER links and the MMv2 mapper usually involved, this never posed a problem for these experiments.

However, that in the long run, centralized mapping and a communication topology of a unidirectional circular graph would become prohibitive when trying to scale the system up, was always clear, even already when that architecture was in the planning stage.

It was clear that for the AEXL, a much more scalable approach was a clear goal. A system supporting a communication topology of a 2d-mesh or 2d-torus with bi-directional communication between a node and its up to four neighbours were considered adequate for our purposes, and also quite feasible to implement, even in a highly modular system, as we’ll see.
10.2 Choosing the Foundation of the AEXL Project

Choosing the foundation, the “baseboard” of the AEXL project consisted primarily on two key questions:

- Which FPGA and board-to-board AER communication system do we want to use?
- Do we want to build our own baseboard supporting a suitable FPGA, or are there FPGA-board off-the-shelf solutions available that already fulfill all our requirements?

10.2.1 FPGA of choice: Xilinx Spartan 6

Back in section 2.6.3, we have explained that at that time, the cost of an FPGA with integrated SerDes hardware circuitry was prohibitive for a to be volume produced AER interfacing platform as the AEXv4. However that was only true until Xilinx launched their Spartan 6 LXT FPGA series, which changed exactly that, as they introduced dedicated SerDes hardware in their comparably low priced Spartan class FPGA product line.

This was a game-changer, and would the Spartan 6 LXT series already have been available 5 years earlier, the AEX platform would probably never have built with a dedicated SerDes chip.

For example a Spartan 6 **XC6LX45T** is not only more powerful than the FPGAs we previously employed in our systems, it also contains four full-duplex SerDes hardware units on that same FPGA chip. These SerDes units were almost as powerful and quite similar to the TLK3101, and the two could probably even made to communicate with each other, given they both can run at 2.5 Gbit/s while using AC coupled LVDS signalling.

This change in what technology was available at what time was the first, but a key reason, to why the AEXL architecture had to be quite unlike the architecture of the AEXv4–MMv2 platform.

It was decided to follow that approach, even though that this meant that interoperating with previous systems as the AEX using the 3 GHz Serial AER interface as presented in 4.3 would become quite difficult, and require additional compatibility hardware, already for the reason that the novel serial AER links would require to employ a different pin-out for the signals on the SATA cables still being used for the serial AER links, as we’ll see later.
10.2.2 Raggedstone 2 FPGA baseboard

For the MMv2 AER mapper described in section 7.4 we chose the Raggedstone 1 FPGA board from http://enterpoint.co.uk/ as a primary component. Using that system, we always had a very good, productive and successful experience, both with the Raggedstone 1 FPGA board, but also with the company behind it.

When we found out, that that same company now also offered FPGA boards based on the Spartan 6 LXT series FPGAs, we had to have a closer look.

Eventually we decided that instead of going through multiple iterations of PCB designs to build our custom FPGA board (as we did with the AEX), we would go for the Raggedstone 2 board from Enterpoint, shown in figure 10.1, which almost perfectly matched our requirements, and in addition was even quite affordable.

With this decision we could also minimize the design risk of not knowing how many PCB iterations we had to produce, assemble and test, until we had a working system.
10 Towards a Modular & Scalable Neuromorphic Platform

10.2.3 Raggedstone 2 Key Components & Interfaces

Figure 10.2 shows the Raggedstone 2 board with key components and also the connectors we can use for board-to-board AER communication are labelled.

![Raggedstone 2 board with key components and interfaces](image)

Figure 10.2: Raggedstone 2: key components & interfaces, © Enterpoint Ltd, reprinted with permission.

Key Components

The “FPGA“ in figure 10.2 is a Xilinx Spartan 6 LXT Series FPGA, more precisely a **XC6SLX45T-2C** in a **FGG484** package. The chip right below the FPGA, labelled “DDR3”, is a DDR3 DRAM memory chip, a **MT41J64M16** from Micron Technology Inc., with a storage capacity of 1 GiBit.

The FPGA has a hardware DDR3 RAM controller, which e.g. takes care of refresh-cycles and other issues one usually has to deal manually. Obviously this is where the DDR3 chip is hooked up to. Thus we have 1 GiBit of FPGA external, by low-latency, high-bandwidth DRAM, which can for example be used to store some kind of mapping table, if the FPGA is supposed to also work as an AER mapper system.
10.2 Choosing the Foundation of the AEXL Project

Board-to-Board I/O Interfaces

As mentioned before, the FPGA we have available on the Raggedstone 2 (RS2) board has four hardware SerDes units (called “GTP Transceivers” by Xilinx), which we can use to connect from one RS2 board to another.

Three of them are wired up to the “3 SATA Connectors” (figure 10.2), the fourth is wired to the “PCIe Connector”. While that fourth GTP Transceiver wired to the PCIe board edge connector can support PCIe, it is important to know that it can also be configured to act as a plain normal SerDes, like the other three SerDes transceivers.

Using these four fully bi-directional high-speed interfaces, it is possible to build a system that supports the desired communication topologies we mentioned in section 10.1.

10.2.4 Raggedstone 2: Expansion Ports

Figure 10.3: Raggedstone 2: expansion ports, © Enterpoint Ltd, reprinted with permission.

In figure 10.3 the expansion ports of the AEXL baseboard are labelled. There are two places to mount expansion boards, one on the left side consisting of the connector rows “JL1” and “JL2”, one on the right side consisting of the connector rows “JR1” and “JR2”.

Each of these connector rows provide 34 I/O pins wired up directly to the FPGA, and 34 power-rail pins, with the “J?1” being wired to GND, and “J?2” supplying 3.3 V.
Towards a Modular & Scalable Neuromorphic Platform

This means we can mount expansion boards in two locations, each of them providing power and 68 I/Os to expansion boards.

10.3 Initial AEXL Extension Boards

After choosing the right baseboard for the AEXL project, we decided to build our initial expansion boards. These initial boards should primarily enable us to replicate most of the features of the AEXv4 platform.

But already with that first batch of expansion boards, Saber Moradi and Marc Osswald committed to build expansion boards for their own specific purposes.

This first step of expansion board development was the first time this project became a community effort. Later many researchers more would join and contribute to the AEXL project, by building more expansion boards or contributing VHDL code specific to the AEXL system.

We will now present the first and second batch of expansion boards that were built for the AEXL.

10.3.1 AEXL/FX2

Figure 10.4: AEXL/FX2 expansion board

Figure 10.4 show the AEXL/FX2 expansion board. It allows us to reuse all of the USB interface code (VHDL interfacing code, FX2
10.3 Initial AEXL Extension Boards

10.3.1 AEXL/FX2

Initial AEXL Extension Boards firmware, Linux kernel driver, user-space code using that interface) on the AEXL platform with only little modification of the VHDL code.

With having this USB interface available, it was trivial to also get the AER monitor and sequencer functionality to that new platform, as well as all the modules that formed the generic AER routing fabric.

There are two versions of the AEXL/FX2 depicted. The only difference is the placement of the micro-USB connector on the right edge of the PCB. In the first version, that connector was placed too far away from the edge of the PCB, which required us to make modifications to USB cables in order to be able to connect them. This mishap was due to misunderstanding of the geometric spec-sheet of the micro-USB connector we selected, and was corrected in the second batch of AEXL/FX2 expansion boards fabricated.

It is important to know that this board is the only expansion board presented here, which has to go in a specific location on the RS2, the top-right spot of the expansion space, and nowhere else. This is because the FX2 chip on this board provides the clock signal for the interface to the FPGA. This signal has to be fed to a special input on the FPGA (a global clock input, GCLK), which is only the case if the AEXL/FX2 is mounted in the right spot.

10.3.2 AEXL/PAER

Figure 10.5 shows parallel AER connector expansion board, version one and two of it.

This board consists of three segments. The top two implement the “Rome-Type” parallel AER connector in a fashion identical to the AEXv4. The third segment at the bottom shows a similar parallel AER connector, but supporting up to 24 instead of 16 address-event data.

The boards were fabricated in a way that they could either be used as a unity of all three segments, or they could also easily be separated from each other.

The only difference between the two versions is that in the second revision, the segment borders where the boards could be cut into three individual segments were guided by a series of regular pad holes, and not by a 1 mm milling path, which made PCB fabrication a quite a bit cheaper.

The main reason we fabricated more AEXL/PAER boards was not
because we wanted to have a new revision, but because we had run out of the AEXL/PAER boards from the initial batch.
10.3 Initial AEXL Extension Boards

10.3.3 AEXL/SAER

The tiny board in figure 10.6 is required if one would try to communi-
cate between the AEXL and the AEXv4 via a high-speed serial AER
interface.

However since as of today nobody was interested in using the AEXL
system together with the systems previously presented, as of my knowl-
edge, this board was never used. Given its tiny size and thus little
effect on the cost of the fabrication of all the expansion boards, the
decision to include this board for fabrication was certainly not affecting
the project financially in a significant manner.

10.3.4 AEXL/IFMEM

![AEXL/IFMEM expansion board](image)

Figure 10.7: AEXL/IFMEM expansion board

Saber Moradi wanted to also build an expansion board for his latest
IFMEM chip, and so we decided to build the expansion board shown in
figure 10.7.

Up to four such AEXL/IFMEM boards can be mounted on one RS2
board simultaneously. Of course one can also mount less of them in
order to free space for other expansion boards required for a certain
setup.
10 Towards a Modular & Scalable Neuromorphic Platform

10.3.5 AEXL/DVS128

Figure 10.8: DVS128 on AEXL board combination.
Left: AEXL expansion board, right: Board connecting to back of the DVS sensor board. PCBs by Marc Osswald.

As mentioned, Marc Osswald also joined the initial AEXL team and contributed two very interesting boards, depicted in figure 10.8. The left PCB, the AEXL/DVS expansion board is mounted on the AEXL platform, the right PCB replaces the monitor/sequencer/USB board in a regular DVS128 camera.

The two boards are then connected using a ribbon cable of suitable length and would feed the parallel AER stream from the DVS camera to the AEXL for further processing. It would also allow the FPGA to program the digital bias generators in the DVS sensor via that same cable.

To make sure we would not run into the usual trouble one has with parallel AER interfaces and ribbon cables, all signals were fed through series resistors before entering the ribbon cable. By selecting and adapting those resistor values, a more or less controlled impedance interface could be established which allowed us to reduce the issues mention to an extent that allowed for a stable DVS-camera to AEXL baseboard interface.

10.4 First Expansion Board Fabrication Panel

Sending many different PCBs out for fabrication is much more expensive, than assembling them to a fabrication panel, and then have a certain number of those PCB panels fabricated.
Figure 10.9: AEXL expansion boards fab-panel v1

Figure 10.9 shows the first fabrication panel we submitted. It contains (top-down, left-to-right order):

- 3x AEXL/IFMEM
- AEXL/FX2
- AEXL/PAER: (2x16, 1x24) combo board
- DVS2AER: DVS-camera side board
- AEXL/SAER
- AEXL/DVS: AEXL side DVS-camera board

10.5 Second Expansion Board Fabrication Panel

The second panel fabricated is shown in figure 10.10 contained (left-to-right)

- AEXL/PAER v2: (2x16, 1x24) combo board
• **AEXL/FX2 v2**

As mentioned reason fabricate a second panel was to fix the bug with the USB connector in the AEXL/FX2, but mainly also because we had run out of AEXL/PAER and AEXL/FX2 expansion boards.

![AEXL expansion boards fab-panel v2](image)

**Figure 10.10: AEXL expansion boards fab-panel v2**

### 10.6 Example AEXL Setups with Extension Boards

Figures 10.11 and 10.12 show two example configuration of an AEXL and a number of different daughter boards. Because the photographs were taken during initial test for mechanical fit during assembly, the boards are only partially assembled.

Figure 10.11 could be used as a setup with three cooperating IFMEM chips, which are monitored and stimulated via the USB interface. Figure 10.12 also has the USB interface for AER monitoring and sequencing, and it could for example be connected to an AMDA board via the two AEXL/PAER16 boards for an experiment with an old chip and via the AEXL/SAER board it could eventually be connect to an entire older experimental setup based on the MMv2 and AEX platforms.
10.6 Example AEXL Setups with Extension Boards

Figure 10.11: AEXL with three AEXL/IFMEM and one AEXL/FX2 expansion boards (partially assembled, photograph from initial test for mechanical fit)

Figure 10.12: AEXL with two AEXL/PAER16, a AEXL/SAER and an AEXL/FX2 expansion board. (partially assembled, photograph from initial test for mechanical fit)
10.7 Scaling to an AEXL cluster

As we mentioned already in the goals of the AEXL project in section 10.1, scalability to at least a medium-scale (possibly heterogeneous) multi-chip platform should be achievable.

However, some limitations (probably design flaws in the “Raggedstone 2”) which we will discuss in section 10.8.2 hindered progress towards that goal so far. In short, limitation prevents two “Raggedstone 2” boards from interfacing with each other with a serial AER protocol using the SATA connectors on the top side of the AEXL. The connections that could be achieved so far were flaky and often lost link synchronization and lost address-events during the time it takes to reestablish the link. We assume that this limitation can be overcome, by providing all AEXL boards involved in a setup with one common reference clock.

To achieve this, we, the author and Richard George, plan to design a fairy simple PCB, simple enough it can be drawn as a “napkin-sketch” as you can see in figure 10.13.

This AEXL Cluster Module (AEXL/CM) has to serve four purposes:

- It has to mechanically hold two RS2 PCBs (with expansion-boards mounted) in place by plugging them into PCI-express x1 sockets on the AEXL/CM,
- provide power to the RS2 boards via the PCIe sockets
- provide a common reference clock to all RS2 boards to overcome the serial AER link-loss problem discussed and
- to interconnect the two RS2 SerDes interfaces on the PCIe connector with each other.

The AEXL/CM is designed as a cascadable system, by soldering them together edge-to-edge. An example of such a setup is given in the next section.

10.7.1 Eight Raggedstone 2 AEXL Cluster Example

Figure 10.14 show four AEXL/CM in a cascaded configuration. In that configuration, all eight RS2 boards plugged in can share one single power supply, and more importantly the single reference clock required for reliable serial AER communication.

Two neighbouring RS2 baseboards on the same AEXL/CM are already connected via their PCIe connectors, and thus have bi-directional
10.7 Scaling to an AEXL cluster

serial AER connectivity. Two neighbouring RS2 baseboards will have to be wired up via a (cross-wired) SATA cable (as short as one can get) to establish a bi-directional serial AER link between them.

The first and the last boards could then be connected via a SATA cable too, to form a bi-directional loop communication topology (which is also a 1D-torus).

Because there are two free SATA ports left on each board, one could also build even bigger systems by expanding in the vertical direction (vertical with respect to figure 10.14. The two free connectors can then be used for vertical serial AER interconnects, in order to get a 2D-mesh or even 2D-torus high-speed serial AER communication topology.

Figure 10.13: Single AEXL cluster module (AEXL/CM) diagram. Blue: PCIe-x1 sockets to connect the AEXL baseboards, green: Serial AER LVDS links between A and B, red: clock signals, dashed lines: LVDS differential signals, pink: cascading connectors to neighboring boards.
10 Towards a Modular & Scalable Neuromorphic Platform

Figure 10.14: Four AEXL/CM boards cascaded to support eight Raggedstone 2 AEXL baseboards. Only the rightmost 25MHz oscillator is assembled and provides the global clock.

10.8 Conclusion

To conclude our discussion of the AEXL project, let's have a look at the status of the system at the time of writing, the current limitations and then have an outlook at what future perspective the AEXL project has and what the next steps should or could be.

10.8.1 Status

We formulated the goals of the AEXL project in section 10.1.

In terms of code-reuse, we have certainly met that goal almost all VHDL entities presented earlier in this thesis in chapters 4 and 5 could be reused, of course with the exception of the TLKiface and associated entities, since the chip this entity was specifically designed for isn’t used anymore in the AEXL project.

Because of the backwards-compatibility design of the AEXL/FX2 even the all the USB codebase presented in chapter 6 is still in use unmodified. Researchers who wrote their own software capable to interface with the AEX via USB can use that exact same code unchanged code with the AEXL platform due to the AEXL/FX2.

Due to the concept of using a baseboard with the capability to mount large expansion boards on top, and the ability to change their configuration for each setup, we certainly also reached to goal to improve on extensibility and modularity when compared to previous platforms described in this thesis.
10.8.2 Limitations

The key limitation of the AEXL project is that while serial AER links via SATA cables connected to *back to the same RS2 baseboard* work, serial AER links via SATA cables connecting *two distinct RS2 baseboards* were flaky, and thus basically unusable for the low-and-constant-latency communication needs of serial AER.

The analysis of this problem by the author and by Hesham Mostafa Elsayed, who performed a number of attempts to connect multiple distinct baseboards, independently came to the same conclusion for the reason causing these issues.

The loss-of-link and link-reestablishing observed is most probably caused by loss of clock recovery in the receiving FPGA integrated SerDes. This in turn is usually caused by having two different reference clock sources on each of the boards, which differ too much either in their clock frequency matching, or by having too much (different) clock jitter.

The AEXL/CM board we presented in 10.7 will allow researchers to verify that hypothesis, by providing a common reference clock to multiple boards, with which one should be able to alleviate this issue.

10.8.3 AEXL became a Community Project

Unlike the AEX and MMv2 projects, the AEXL project quickly became a collaborative effort with many PhD students at INI involved. At the time of writing, the following people have already contributed to the AEXL project in some way, be it by creating new expansion boards or contributing to the AEXL FPGA code-base:

- Dora Sumisławska
- Federico Corradi
- Hesham Mostafa Elsayed
- Marc Osswald
- Ning Qiao
- Richard George

In section 11.2 we will give you an outlook for future of the AEXL project including starting points for future work to bring this project to its full potential.
10 Towards a Modular & Scalable Neuromorphic Platform
Impact & Conclusion

In the final chapter of this thesis, we will first present a number of experiments performed by various researchers, in more or less the same order we have introduced these platforms throughout this thesis. This should show you the impact this thesis had in enabling a lot of research done by various researchers at multiple institutions.

We will then provide an outlook on the future potential these systems have, by analyzing which systems have potential for future expansion and by providing a few concrete examples of starting points for such projects.

Finally, we will conclude the thesis by discussing the design principles, strategies and methodologies the author of this thesis found to be most influential and valuable in building systems effectively and successful, successful also in the way that these resulting systems could then efficiently be used by other researchers for their purposes, or could even be further expanded as form of community projects.

11.1 Impact

In this section we will illustrate the impact of this thesis by providing examples of the impact of these types of systems:

- Single AEXv4 and AMDA System,
- single AEXv4 with a neuromorphic chip directly attached,
- experiments involving multiple AEX+AMDA systems and the MMv2 mapper,
- the eMorph project and finally
- an experiment based on the AEXL project.
11 Impact & Conclusion

Figure 11.1: AEXv4 and AMDA board combined

11.1.1 Single AEXv4 and AMDA

Figure 11.1 shows the single AEXv4 and AMDA platform that was explained in more detail already back in 4.1.5. This setup alone cannot do much more than be connected to a laptop or PC in order to sequence AEs to the chip and monitor what the chip sends back.

While this is not exactly a super interesting setup for experiments, it is the setup which was used for initial testing of a newly fabricated neuromorphic VLSI chip that relied on the AMDA board to provide analog bias voltages to it. However such characterization of a chip is a key requirement to acquire data required to write and publish a paper that introduces the architecture, capabilities and characteristics of a newly developed neuromorphic chip.
11.1 Impact

Figure 11.2: PC, AEXv4 and IFMEM chip his on own PCB; block diagram, adapted from [Rahimi Azghadi 15]. Labels see text.
11 Impact & Conclusion

11.1.2 Single AEXv4 with Directly Attached Chip

Figure 11.2 shows a photograph and the block diagram of a setup using the IFMEM chip directly connected to the AEXv4 through its own daughter-board IFMEM2AEX.

The IFMEM chip can be used with the AMDA board, because it uses integrated digital bias generators to produce the bias voltages, which replace the main purpose of the AMDA board.

The AEXv4 was explained in great detail in section 4.1. The remaining labels on figure 11.2 are:

- **G**: low-speed USB interface to send commands to (H)
- **H**: microcontroller used to program the on-chip digital bias generator values
- **I**: IFMEM chip
- **K**: analog signal probe cable leading from an SMA connector to an oscilloscope (not shown)

The research performed on this setup was used to write the publication [Rahimi Azghadi 15]: “Programmable Spike-Timing-Dependent Plasticity Learning Circuits in Neuromorphic VLSI Architectures”


**Abstract:**

“Hardware implementations of spiking neural networks offer promising solutions for computational tasks that require compact and low-power computing technologies. As these solutions depend on both the specific network architecture and the type of learning algorithm used, it is important to develop spiking neural network devices that offer the possibility to reconfigure their network topology and to implement different types of learning mechanisms. Here we present a neuromorphic multi-neuron VLSI device with on-chip programmable event-based hybrid analog/digital circuits; the event-based nature of the input/output signals allows the use of address-event representation infrastructures for configuring arbitrary network architectures, while the programmable synaptic efficacy circuits allow the implementation of different types of spike-based learning mechanisms. The main contributions of this article are to demonstrate
11.1 Impact

how the programmable neuromorphic system proposed can be configured to implement specific spike-based synaptic plasticity rules and to depict how it can be utilised in a cognitive task. Specifically, we explore the implementation of different spike-timing plasticity learning rules online in a hybrid system comprising a workstation and when the neuromorphic VLSI device is interfaced to it, and we demonstrate how, after training, the VLSI device can perform as a standalone component (i.e., without requiring a computer), binary classification of correlated patterns.”


11.1.3 Multi–AEXv4 and MMv2 Mapper Setups

Many experiments involved a number of AEXv4 plus AMDA boards and the MMv2 mapper. Figure 11.3 shows the block diagram of such a setup. The AEX and MMv2 systems are connected using 3 GHz Serial AER links in a loop topology.

The multi-chip system is very heterogeneous, it involves one DVS sensor and three different neuromorphic processing chips. Figure 11.4 shows a photograph of a part of the setup, the DVS being stimulated by controlled input on a flat-screen which is connected to the first AEX board in the loop, and the second AEX board connected to an AMDA board which carries the first neuromorphic processing chip. Even though there is a loop topology, due to non-overlapping address-ranges for the various AER sources and destinations and intelligent routing in the AEX boards, the mapper can connect any AER source unit (neuron, sensor) to any or numerous AER destination units (synapse).

The basis for this setup were laid by Fabio Stefanini, Emre Neftci and Sadique Sheik, who programmed the AEX board using the highest level abstraction layer of the AEX we explained in section 5.3.

This type of architecture consisting of the AEX and MMv2 mapper interconnected with high-speed serial AER links has been used for many neuromorphic experiments. It is also presented in [Liu 15, sec.13.2.5, p. 324fff.], and was the basis for experiments resulting in numerous publications.
11 Impact & Conclusion

Figure 11.3: MMv2 and 4x AEXv4+AMDA in an experiment involving a DVS retina and three different neuromorphic VLSI chips, adapted from F. Stefanini, E. Neftci & S. Sheik.

Figure 11.4: Photograph of parts of a setup as in diagram 11.3, from [Neftci 10a].
“Synthesizing Cognition in Neuromorphic Electronic Systems”

A quite recent example of a publication based on multiple AEX boards and the MMv2 mapper is [Neftci 13], titled:
“Synthesizing Cognition in Neuromorphic Electronic Systems”

Authors: Emre Neftci, Jonathan Binas, Ueli Rutishauser, Elisabetta Chicca, Giacomo Indiveria and Rodney J. Douglas

Figure 11.5 shows a diagram of the setup used to perform their experiment. Here the chips were interconnected using the AEX and MMv2 mapper systems.

Abstract:

“The quest to implement intelligent processing in electronic neuromorphic systems lacks methods for achieving reliable behavioral dynamics on substrates of inherently imprecise and noisy neurons. Here we report a solution to this problem that involves first mapping an unreliable hardware layer of spiking silicon neurons into an abstract computational layer composed of generic reliable subnetworks of model neurons and then composing the target behavioral dynamics as a ‘soft state machine’ running on these reliable subnets. In the first step, the neural networks of the abstract layer are realized on the hardware substrate by mapping the neuron circuit bias voltages to the model parameters. This mapping is obtained by an automatic method in which the electronic circuit biases are calibrated against the model
parameters by a series of population activity measurements. The abstract computational layer is formed by configuring neural networks as generic soft winner-take-all subnetworks that provide reliable processing by virtue of their active gain, signal restoration, and multistability. The necessary states and transitions of the desired high-level behavior are then easily embedded in the computational layer by introducing only sparse connections between some neurons of the various subnets. We demonstrate this synthesis method for a neuromorphic sensory agent that performs real-time context-dependent classification of motion patterns observed by a silicon retina.”


11.1.4 eMorph iHead Systems

Figure 11.6 shows the cover of “Spektrum der Wissenschaften”, the German edition of “Scientific American” from August 2015. The article explains the iCub and how researchers are planning on using it to create various levels of artificial self-awareness and artificial intelligence. The article also cites an eMorph publication: [Rea 13].

Moreover, the eMorph project resulted in 16 publications according to the eMorph project report. Many of them were even written collaboratively by authors from different project partner institutes. To list few selected publications:

- [Fasnacht 11b], “A PCI based high-fanout AER mapper with 2 GiB RAM look-up table, 0.8μs latency and 66 MHz output event-rate”, by D. B. Fasnacht and G. Indiveri, 2011
- [Bartolozzi 09], “Selective Attention in Multi-Chip Address-Event Systems”, by C. Bartolozzi and G. Indiveri, 2010
11.1 Impact


The author of this thesis was first author on one of the eMorph project publications and co-author on two others.

Figure 11.6: iCub robot on the cover of “Spektrum der Wissenschaften”. the German edition of “Scientific American”, August 2015, showing some traces of partying ;-) The article cites the eMorph publication [Rea 13].
11.1.5 AEXS & USB Design Reuse

A very recent example of successful hardware design and code reuse is shown in figure 11.7. The board partially depicted consists of a number of cxQuad neuromorphic chips, but also integrated what might be considered more or less as the AEXS hardware design, similar to how the AEXS design became part of the iHead board in eMorph.

The chip at the top-right is the Cypress FX2 (see sec. 4.4.1) USB interface chip, above it at the board edge one can see the micro-USB connector the chip is wired up to. On the right of the FX2 chip is the corresponding FPGA (the square BGA package) that interfaces between the FX2 and the rest of the board. While the AEXS used a Spartan 3AN series FPGA, this board upgraded to a Spartan 6 series FPGA, an FPGA that can be considered the “little brother” of the FPGA that is present on the Raggedstone 2 board used in the AEXL project.

Because this work has not been published yet at the time of writing, no further information about the system can be given.

Figure 11.7: Recent example of AEXS hardware design and code reuse in a system designed by Ning Qiao
11.1 Impact

Multiple researchers have contributed to the AEXL project by expanding it either by building more expansion boards for their needs, or by adding to the VHDL code-base used.

Figure 11.8 shows the AEXL with an AEXL/FX2 USB interface expansion board, and the REXv1 expansion board designed by Marc Osswald. The REXv1 uses so-called GPIO-expander chips, which typically have 16 I/O pins that can be controlled by the FPGA via only two pins on the expansion connector of the RS2 baseboard, by means of using a serial protocol such as I²C.

On top of the REXv1 expansion board, sits a daughter-board designed by Richard George carrying a ROLLS chip. The additional GPIO signals provided by the REXv1 allow for even a large chip as the ROLLS with more than 68 I/O pins to be mounted on the AEXL platform.

Figure 11.9 shows an AEXL again with the AEXL/FX2 and an expansion board called CXQUAD CLOWN board, created by Ning Qiao.
As you can see, this expansion board uses a special kind of socket to mount a cxQuad chip onto it. This ZIF-socket, for “zero insertion force socket”, allows a chip to be mounted on the board without any soldering involved.

Thus one can also very quickly and without risking any damage replace the mounted chip with another one (of the same kind). This kind of setup is used for initial testing of a batch of chips received from the fab, in order to determine which chips work and which don’t, or to measure which chip operates how “well” (fast, precise, low-power, etc.).

Figure 11.10 shows a quite complex AEXL based setup created by Marc Osswald. It consists of one AEXL with two DVS-cameras and the AEXL/PAER and AEXL/FX2 expansion boards, all mounted on a pan-tilt unit.

A second AEXL setup also uses the AEXL/FX2 and another expansion board created by Dora Sumisławska, Federico Corradi and Ning Qiao, which connects a ROLLS chip to the AEXL in order to process the visual input from the DVS cameras.

Because none of the work presented in this section is published yet, no further information can be provided as of now.
Figure 11.10: AEXL based vision experimentation setup involving two AEXL boards, one interfacing to two DVS-cameras mounted on a pan-tilt unit, another (visible bottom-right) carrying the ROLLS neuromorphic chip on an AEXL expansion board created for that chip (visible bottom-left), a setup by Marc Osswald.
11 Impact & Conclusion

11.1.7 Publications Enabled by this Thesis

It is very hard to estimate how many publications by people now or formerly at INI or our project partners have been enabled by the work presented in this thesis. One reason for this is probably best described by quoting the title of a book about marketing in the internet age [Blackshaw 08]:

“Satisfied Customers Tell Three Friends, Angry Customers Tell 3000”

In the case of the work presented in this thesis, it is quite similar: sadly less researchers than you would expect cite or even acknowledge you when they publish something based on an experiment, enabled by the work presented here.

Only when things do not work as expected can you be sure to hear back from your “customers”.

The latter happened only rarely. And the former is the reason for why we can only make guesstimates about how many publications by others were enabled by this thesis.

As mentioned before, we have hard numbers in the eMorph project report: 16 publications are listed there. I can only guess that there are at least one to two dozen more by now, and because most experiments based on the latest platform, the AEXL project are no published yet, there will be many more to come.

Another reason is that the hardware and software that was designed and presented in this thesis tends to dissipate across the globe, along with the researchers using it. As of the knowledge of the author of this thesis, the following cities are among those where our hardware “disappeared” to:

Geographical Hardware Distribution:

- Adelaide, Australia
- Bielefeld, Germany
- Genoa, Italy
- Irvine, USA
- New York, USA
- Sand Diego, USA
- Vienna, Austria
- Zurich, Switzerland
- maybe more by now
The NCS group and the INI generated roughly 80 publications during the work on this thesis, the great majority of which are based on experiments performed involving hardware presented in this document. Given the other research institutions employing that same hardware, it is very likely that the number of publications enabled in some part by this work is over one hundred.

### 11.2 Future Work

Before concluding this thesis, let’s present some starting points for future work, based on the work presented.

#### 11.2.1 MMv2–AEXv4–AMDA

We can quite safely say, that while some researchers are still using this platform, it is unlikely that it will be extended any further.

There are still publications being released based on this platform, such as [Rahimi Azghadi 15], which was released only this August 2015.

However since the AEXL project has gained much more attraction from current and newly starting researchers, it is very likely that the MMv2–AEXv4–AMDA platform will be phased out pretty soon.

#### 11.2.2 AEXL Project

As mentioned previously in section 10.8.3, the AEXL project gained quite a community of researchers extending it by building new expansion boards or by expanding the VHDL code-base for the “Raggedstone 2” baseboard, this will certainly be where it is most likely that the this project will be further continued.

For this community there are a number of starting points which could result in new capabilities or increased potential of the AEXL platform:

**AEXL Cluster Experimentation:** Since the AEXL/CM is still work-in-progress at our lab, finishing and testing the uses and capabilities of a cluster setup based on the AEXL platform will certainly be the next step.
11 Impact & Conclusion

**AEXL Cluster Distributed Routing Prototyping:** While it is quite inefficient to model the analog circuits we use to build neuromorphic VLSI synapses and neurons, for modeling AER routing circuits, the FPGA on the “Raggedstone 2” baseboard is perfectly suited for experimenting with distributed AER routing in an AEXL cluster setup.

The FPGA approach has the advantage that it allows for very rapid prototyping where results can be gained in a matter of minutes.

If routing circuitry is hardwired into a VLSI chip, one has to wait for months to receive the finished and packaged chip back from the fab, while one always has to be uncertain whether the chip then works as expected. To avoid that, the AEXL cluster could be used to investigate such distributed routing schemes, to prototype them and eventually even to provide the means for in-system testing and verification of a routing circuit that will eventually be fabricated as part of a VLSI chip. See [Kaeslin 15, sec. 5.2.3, p. 306] about “Hardware Assisted Verification [of VLSI designs].”

**AEXL Distributed Complex AER Mapper:** Another interesting project would be to use the 1 Gibit DDR3 memory to implement a highly reconfigurable ultra low-latency mapper. Unlike with the AER mapper presented in chapter 7, the latency of this architecture should be low enough to be even feasible to implement AER delay-mappings, where AEs have to be stored for a per address defined time.

It would also be very promising to investigate, how to best distribute the task of mapping AER streams in a AEXL cluster system. Research in this area could lead to very interesting and novel results.

**AEXL based AER Monitor/Sequencer:** Finally, the PCIe interfacing capabilities could be used to implement a AER monitor-sequencer with very high performance. While the USB interface used for the AEXv4 monitor-sequencer could sustain 40 MBytes/s, such PCIe based solution could sustain about 250 MBytes/s. The greatest challenge, here though, would probably be that the RS2 board plugged into a PC for monitoring and sequencing would also have to have access to the same common reference clock as the remaining RS2 baseboards in the setup. This might be achievable, by sending that clock from the AEXL/CM boards to the monitor-sequencer board via an additional coaxial or LVDS capable cable.
11.3 Conclusion – Design Methodologies & Strategies

Finally we will conclude this thesis with something that should be interesting to basically everybody performing hardware-related research. We tried to identify the design methodologies and strategies that were applied during this work, and we are going to present those methodologies and strategies which we think have been the most influential ones in aiding to reach the goals of this thesis.

While development methodologies are a huge topic in software engineering since quite some time, the influence of such methodologies has been comparably low in hardware development, be it PCBs, VHDL for FPGAs or ASICs or even analog circuits for analog VLSI chips.

Even though hardware development is quite unlike software development, we can still learn a lot from software development methodologies and apply it, maybe in a somewhat adapted form, to hardware development.

11.3.1 Version Control Systems

Let’s figure out what a “Version Control Systems” (VCS) is and when, where and for what purpose one should use it.

A short History on Version Control Systems

Version Control Systems allow you to store, retrieve, compare and manipulate multiple versions of something you store into and manage with that system.

The earliest system gaining significant popularity was CVS, the “Concurrent Versions System”, which was initially released in 1990.

Even though CVS was a usability nightmare, it took computer scientists one decade to devise something less “painful” to use than CVS. In 2000 the first release of “Subversion” was published.

While Subversion was much better in terms of usability when compared to CVS, one key architecture was still the same as in CVS: There had to be one central server, where all versions of a “project” were stored. This was identified as a clear deficiency, because almost all operations the user would perform would require interaction with the central server, which induces latency and requires for the central server to be reachable in order to work with such a system.
11 Impact & Conclusion

In order to solve this issue, the concept of a dVCS, a “distributed Version Control System”, was created. In a dVCS, there is no necessity for one central server. There can be zero, one or even many.

Most importantly, all operations such as committing a new version to a repository can be performed locally without being online.

**What goes into the dVCS?**

Short answer: Everything.

Longer answer: Everything that cannot trivially be reproduced, regenerated, compiled or built in a more or less scripted or automated fashion based on other data in a version controlled repository.

As an example: If you write a paper in latex, you certainly commit all the LaTeX files and images/graphs into the repository, but while you work on the paper, you don’t commit the PS or PDF file you produce. However there is also an exception to that exception: Once you submit (release) a copy of that PS or PDF, e.g. for consideration at a conference, you should make sure you also archive that exact copy in the repository in case there are any doubts later on what version exactly you submitted/released.

11.3.2 VHDL Verification & Testing for FPGA Applications

When describing hardware architectures in VHDL, Verification & Testing are an extremely important tool to ensure that the resulting circuits will meet the requirements specified.

During this thesis, two different approaches were used to perform verification and testing:

- Verification & Testing via Simulation
- In-System (In-Hardware) Verification & Testing

**Verification & Testing via Simulation**

In verification and testing via simulation, the entity to be tested is typically “mounted into” a “test-bench”. This test-bench then presents stimuli to the inputs of the entity being tested, and verifies that the entity produces the expected results at its outputs during simulation.
11.3 Conclusion – Design Methodologies & Strategies

For an in depth introduction to the topic please refer to [Kaeslin 15, Chapter 5: “Functional Verification”].

There are always trade-offs between simulation speed, test-coverage and other factors that are to be considered when verifying HDL by simulation.

To put it in Hubert Kaeslin’s words:

“Exhaustive verification is not practical, even for relatively modest functions. Dynamic verification, therefore, must almost always do with a partial set of test cases. The problem is to come up with a test suite of practical size and sufficient coverage.”, quoted from [Kaeslin 15, Observation 5.6, p. 316].

In-System (In-Hardware) Verification & Testing

Another option is to test HDL entities in hardware. This is a quite obvious approach when developing HDL for FPGAs, but not necessarily when developing HDL for ASICs.

However even when developing for ASICs, what is also called “hardware-assisted verification” is an option to increase verification performance. In [Kaeslin 15, sec. 5.2.3, p. 306] a system is described where a “Blue-gene” processor was tested by means of a huge FPGA cluster. This approach resulted in a speed-up of a factor of ca. 100’000 over traditional verification by simulation.

What to Choose When?

In the work presented here, the VHDL code for the AEX platform was tested using both methods.

Internal entities such as those that make up the “Generic AER Routing Fabric” were tested by simulation. Also the parallel AER interfacing entities were tested by simulation, because by design they did not interact with one specific chip, but numerous chips with different timing characteristics.

On the other hand, the VHDL entities interfacing with the FX2 USB chip and the TLK3101 SerDes chip were tested in hardware. The problem with testing such interfacing entities in simulation is that one has to develop a test-bench which behaves exactly as the chip does.
To create such a test-bench is first of all very work intensive and secondly often leads to the following scenario:

1. You learn how the chip works by reading its documentation.
2. You write a test-bench that behaves like you think the chip does.
3. You write the interfacing HDL entity according to what you learnt about the chip.
4. Simulation is a success, test-bench and interfacing entity pass all tests.
5. In hardware, the interface doesn’t work, because you got something wrong already in the first step or your test-coverage was insufficient to detect the fault.

One approach in approaching this problem is to have separate people or separate teams develop the test-bench and the interfacing entities. In research though this is in many situations not that easy to do, because only one person is tasked with developing that interface.

In the experience gained during the work on this thesis, in-hardware testing has proven to be an invaluable approach when interfaces to complicated chips such as the FX2 USB interface chip had to be tested.

### 11.3.3 Incremental HDL Design – Continuous Integration & Testing

Another invaluable approach is what we call “Incremental HDL Design”. What it means, is that development of HDL is done in small steps of isolated changes, which then are tested immediately whenever possible, and if deemed functional, are committed to the code-base to increment its functionality.

This approach comes from the experience that when you just code away on your VHDL entities for half a day, and then find out you made a mistake during that time. It is not uncommon that isolating and fixing that bug will take much longer than the half-day one spent on development only and neglecting testing.

In software development, there is a similar approach commonly known as “Continuous Integration & Testing”.

258
11.3 Conclusion – Design Methodologies & Strategies

11.3.4 Scripted/Automated Building and Testing

One major step to prevent the situation we just described is to make testing a step in your work process that can be performed frequently with ease. So testing should be scripted whenever possible to make it as simple to run those tests often.

If it is feasible and possible, the entire build process should be scripted/automated too, even when one probably has the option to perform the build process via a shiny GUI.

First of all, this allows you to better reproduce the build process, as the steps to get to the result were not some undocumented clicks, but a script, which you also committed to your dVCS repository in order to be able to reproduce that build process at a later stage with exactly identical results.

Secondly, this might enable you to even integrate testing into the build process as its last step, or at least to help you make it a natural process to first run your build scripts followed by running your test scripts.

11.3.5 Release Often

Another recommendation is that you should release your new code-base as often as possible to your fellow researchers who use it. This can sometimes be difficult, because your “customers” are already happy with their “product”, but if you don’t do it, nor convince people to use your latest version, you will harm your development process in at least two ways:

First of all, if nobody is using your latest code-base, you are going to be the only person testing it, which naturally reduces test-coverage.

Secondly, you should treat feedback from people using your hardware and code as extremely valuable input. However, the value of that input usually diminishes the older the version of a system is. The most valuable feedback is the one you get about your latest version. Only then can you be sure that things are working as they should and that you are steering your development efforts into the right direction.
11.3.6 Release Management – Archive all the Things!

When you script your building and testing stages, you can just as well script your release stage.

This usually involves writing a script that assembles all parts people need to upgrade to your latest release into one single (versioned) release archive file, which you should then upload or otherwise share with people using a previous version. Finally you should obviously announce the availability of a new release and urge people to upgrade.

Most importantly though, you should commit that release archive to your repositories. If possible it should either go into the same repository as the source and build code that produced that release. It might also go in a separate repository, but then one has to make sure by other means, that it is always clear which source-code version (e.g. git hash) was used to produce which release. If you don’t manage to keep track of that source-to-release relationship information, tracking down a bug reported to be in a certain release will quickly become a nightmare.

11.3.7 Extreme Programming Concepts in Hardware Design?

“Extreme Programming”, also known short as XP, is a set of development methodologies best known in the context of software engineering.

When one compares the strategies and methods for mixed hardware-/software development described in the paragraphs above to the methodologies advocated by software developers using XP methodologies, it quickly becomes obvious that many of them have close counterparts or even equivalents in the XP literature.

Let us introduce these XP recommended practices and then analyze what applies how to hardware development and the strategies and methods we outlined above.

Extreme Programming Practices

The “Extreme Programming Pocket Guide”, [chromatic 03], lists the recommended practices in Extreme Programming in the table of contents as follows:
11.3 Conclusion – Design Methodologies & Strategies

Coding Practices

• CP1: Code and Design Simply
• CP2: Refactor Mercilessly
• CP3: Develop Coding Standards
• CP4: Develop a Common Vocabulary

Developer Practices

• DP1: Adopt Test-Driven Development
• DP2: Practice Pair Programming
• DP3: Adopt Collective Code Ownership
• DP4: Integrate Continually

Business Practices

• BP1: Add a Customer to the Team
• BP2: Play the Planning Game
• BP3: Release Regularly
• BP4: Work at Sustainable Pace

What to apply in Hardware Design?

CP1 – “Code and Design Simply” was probably too obvious to the author to mention it. The simpler a feasible solution to a given problem is, the easier this solution can be understood completely and the less likely it is that we introduce a bug when changing it. This is true for hardware design just as well as it is for software development.

CP2 – “Refactor Mercilessly”, the opinion of hardware designers probably differ from those of the XP software community. Maybe the “refactor when necessary” approach is more common for hardware designers. And there is one more thing to consider when refactoring an HDL design: Usually the old and new implementations should be functionally equivalent. This should be exploited in verification by simulation, because the old and new implementations can simply be compared with each other.

CP3 – “Develop Coding Standards” is something commonly done in hardware design too. One example is the “VHDL Signal Naming Convention” we presented in section 0.1.2 and used throughout this thesis. Of course there can be also further conventions, e.g. about naming of entities and procedures in VHDL, or about patterns used to build finite state machines in VHDL.
11 Impact & Conclusion

There are a number of XP practices which are only really applicable when an entire team of researchers works together on the same task, which is rarely the case for Ph.D. projects. XP practices CP4, DP2, and BP2 are thus rarely applicable in our case.

**DP1** – “Adopt Test-Driven Development” is basically what we described above as “Scripted/Automated Building and Testing”, **DP4** – “Integrate Continually” is what we described above as “Incremental HDL Design – Continuous Integration & Testing”.

**DP3** – “Adopt Collective Code Ownership” is something that hopefully happens once a project such as the “AEXL project” becomes a community effort. Working together on a common VHDL code-base can not only be very efficient for everybody, it can also be very educative when new researchers join such a community.

**BP1** – “Add a Customer to the Team” and **BP3** – “Release Regularly” were both described before as “Release Often”. By having your “customers” use your latest code-base, you at least partially integrate them into your “team” by increasing the likelihood to get helpful feedback from them.

Last but not least, **BP4** – “Work at Sustainable Pace” is probably something that mostly depends on the character of a Ph.D. student, and is a recommendation that is frequently ignored.

More on XP

A lot of information about the concept of Extreme Programming can also be found on Wikipedia¹ and plenty of other websites. In any case we strongly recommend to read the “Extreme Programming Pocket Guide” from O’Reilly: [chromatic 03] and / or other XP literature, even when mostly doing hardware development only.

¹https://en.wikipedia.org/wiki/Extreme_programming


263
Bibliography


[Delbruck 10a] T. Delbruck, R. Berner, P. Lichtsteiner & C. Dualibe. 32-bit Configurable bias current generator with sub-off-current capability. In International Symposium


[Häfliger et al. 04a] P. Häfliger et al. Circuit board with potentiometers from CAVIAR project. Info: [http://heim.ifi.uio.no/~hafliger/CAVIAR/](http://heim.ifi.uio.no/~hafliger/CAVIAR/),
Bibliography


266


Bibliography

Social Cognition: From Babies to Robots.


Bibliography


