Dissertation

submitted to the
Combined Faculties for the Natural Sciences and for Mathematics
of the
Ruperto-Carola University of Heidelberg, Germany
for the degree of
Doctor of Natural Sciences

Presented by
Dipl.-Phys. Simon Friedmann
born in Heidelberg, Germany

Date of oral examination: July 25, 2013

A New Approach to Learning in Neuromorphic Hardware

Referees: Prof. Dr. Karlheinz Meier
Prof. Dr. Ulrich Brüning

Ein Neuer Ansatz für das Lernen in Neuromorpher Hardware

Diese Doktorarbeit stellt einen neuartigen, besonders flexiblen Zugang zum Lernen in durch das Gehirn inspirierten Rechensystemen dar. Ein klassischer Digitalprozessor wurde mit lokaler, analoger Verarbeitung kombiniert, um Flexibilität und Effizienz zu erreichen. Insbesondere erlaubt dies die Umsetzung der modulierten spike-timing dependent plasticity Lernregel. Dieser Ansatz wurde in ein abstraktes, hybrides Hardwaremodell formalisiert. Mit diesem Modell wurde Belohnugslernen anhand eines Fallbeispiels simuliert, um die Auswirkungen der Hardwareeinschränkungen abzuschätzen. Um die Machbarkeit der vorgeschlagenen Architektur zu ergründen wurde ein synthetisierbarer Plastizitätsprozessor entworfen und mittels des allgemeinen CoreMark Benchmarks getestet (Bestes Ergebnis: 1.89 pro MHz). Der Prozessor wurde auch als Teil eines 65 nm Prototypenchips produziert, auf dem er eine Fläche von 0.14 mm^2 belegt und eine maximale Taktfrequenz von 769 MHz erreicht. Zunächst wurde eine nicht-programmierbare Plastizitätsimplementierung entwickelt, die jetzt Teil des sich in Betrieb befindenden BrainScaleS wafer-scale Systems ist. Später wurde dieser Entwurf um einen Plastizitätsprozessor erweitert, um die vorgeschlagene hybride Architektur zu verwirklichen. Simulationen zeigen eine Geschwindigkeitsverbesserung von 42 % gegenüber der nicht-programmierbaren Variante. Aus der Vorbereitung für die Produktion ergibt sich ein Flächenbedarf von 6.2 % der Gesamtfläche.

A New Approach to Learning in Neuromorphic Hardware

This thesis presents a novel, highly flexible approach to plasticity and learning in brain-inspired computing systems. A classical digital processor was combined with local analog processing to achieve flexibility and efficiency. In particular, this allows for the implementation of modulated spike-timing dependent plasticity. The approach was formalized into an abstract hybrid hardware model. This model was used to simulate a reward-based learning task to estimate the effect of hardware constraints. To investigate the feasibility of the proposed architecture, a synthesizeable plasticity processor was designed and tested using the CoreMark general purpose benchmark (best score: 1.89 per MHz). The processor was also produced as part of a 65 nm prototype chip, requiring 0.14 mm^2 of die-area, and reaching a maximum clock frequency of 769 MHz. In a preparatory step a non-programmable plasticity implementation was developed, that is now part of the operational BrainScaleS wafer-scale system. This design was later extended with the plasticity processor to implement the proposed hybrid architecture. Simulations show a speed improvement of 42 % over the non-programmable variant. By preparation for production, the area requirement for the digital part is estimated to be 6.2 % of total area.

Contents

List of Figures 12

List of Tables 15

1. Introduction 17
- 1.1. The BrainScaleS wafer-scale system 19
- 1.2. Models of plasticity 21
  - 1.2.1. Spike-timing dependent plasticity 21
  - 1.2.2. Phenomenological models from biology 23
  - 1.2.3. Reward-modulated STDP 24
- 1.3. Design goals for neuromorphic hardware 25

2. Theory 27
- 2.1. Requirements for hardware 27
  - 2.1.1. Two-factor STDP 27
  - 2.1.2. Reward-modulated STDP 28
  - 2.1.3. Phenomenological models 28
- 2.2. Abstract hybrid hardware model 29
  - 2.2.1. Combining analog and digital computing 29
    - 2.2.1.1. What type of digital part is needed? 30
  - 2.2.2. The abstract hybrid hardware model 30
    - 2.2.2.1. Using the abstract model for plasticity 31
    - 2.2.2.2. Acceleration in time 32
    - 2.2.2.3. Discretized weights 32
    - 2.2.2.4. Local analog processing in the synapse 33
    - 2.2.2.5. Global digital processing by the processor 35
    - 2.2.2.6. Drift of analog storage 35
    - 2.2.2.7. Mismatch 37
    - 2.2.2.8. Dynamic analog noise 37
    - 2.2.2.9. Processing speed 38
    - 2.2.2.10. Communication latency 38
- 2.3. Reward modulated STDP 38
  - 2.3.1. Baseline performance 41
  - 2.3.2. Mapping reward modulated STDP to the AHM 44

2.3.3. Discretized weights . . . 44
2.3.4. Threshold readout . . . 47
2.3.5. Robustness to variations in the drift model . . . 50
2.3.6. Mismatch on the evaluation function . . . 52
2.3.7. Delayed reward . . . 52
2.3.8. Guidelines for hardware implementation . . . 56

3. Hardware design 59
3.1. Plasticity processor technology . . . 59
3.1.1. Design principles . . . 59
3.1.2. Instruction set architecture . . . 60
3.1.3. Microarchitecture . . . 61
3.1.4. Instruction fetching and control transfers . . . 66
3.1.5. Instruction cache . . . 69
3.1.6. Instruction scheduling . . . 72
3.1.6.1. Result shift register . . . 72
3.1.6.2. Write back channels . . . 73
3.1.6.3. Lookup cache and variable-latency operations . . . 76
3.1.6.4. Write-through optimization . . . 77
3.1.6.5. Pre-decoding instructions . . . 77
3.1.6.6. Multi cycle operations . . . 80
3.1.6.7. Scheduling instructions to functional units . . . 82
3.1.7. Execution in the back end . . . 87
3.1.8. Functional units . . . 88
3.1.8.1. Input/output over the plasticity processor bus . . . 88
3.1.8.2. Multiplier and divider . . . 89
3.1.9. Interrupts & Exceptions . . . 90
3.1.9.1. Interrupt processing . . . 91
3.1.9.2. Saving the return address . . . 91
3.1.9.3. Asynchronous interrupts . . . 91
3.1.9.4. Precise interrupt problem . . . 92
3.1.9.5. Critical and machine check interrupts . . . 92
3.1.10. Timer facility . . . 92
3.1.11. General purpose input/output registers . . . 93
3.2. On-chip bus technology . . . 93
3.2.1. Motivation and design goals . . . 93
3.2.2. Interface specification . . . 95
3.2.3. Basic bus fabric building blocks . . . 96
3.2.3.1. Bus arbiter . . . 97
3.2.3.2. Bus delay . . . 97
3.2.3.3. Bus splitter . . . 97

3.2.3.4. Example bus configuration 97
3.2.4. Additional bus building blocks 99
3.2.4.1. Register target 99
3.2.4.2. Serializer/Deserializer 99
3.2.4.3. RAM interface adapter 99
3.2.4.4. HICANN system bus adapter 100
3.2.5. Methodology: using code generation for bus specification . . . 100
3.3. STDP logic in the BrainScaleS wafer-scale system 101
3.3.1. HICANN synapse array interface 102
3.3.1.1. Structural description 102
3.3.1.2. Analog evaluation 104
3.3.1.3. Control sequences on synapses 105
3.3.2. Non-programmable STDP implementation 109
3.3.2.1. Functional overview 109
3.3.2.2. Design of the bus interconnect 110
3.3.2.3. Structure of the design 112
3.3.2.4. Operation of the access state machine 113
3.3.2.5. Automatic weight update controller 115
3.3.3. Synapse array interface adapter for programmable STDP 117
3.3.3.1. Detailed description 119
3.4. SYNAPSE special function unit 126
3.4.1. Special purpose registers 127
3.4.1.1. Vector registers 127
3.4.1.2. Look-up table registers 127
3.4.1.3. Pattern register 128
3.4.1.4. Select state register 128
3.4.2. Special purpose instructions 128
3.4.2.1. Apply mapping from look-up table to vector elements 129
3.4.2.2. Compare elements with immediate 129
3.4.2.3. Select elements from two vectors 130
3.4.2.4. Perform operation sequences 130
3.4.2.5. Swap vector register file 131
3.4.2.6. Register move instructions 132
3.4.3. Code example 134
3.4.4. Design considerations for the instruction set extension 134
3.4.5. Implementation 137
3.5. Native Vector Extension 137

4. Functional verification and software support 139
4.1. Directed verification: program level testing 140
4.1.1. Generating the expected result memory image 140

4.1.2. The CoreMark Benchmark for directed testing 142
4.1.3. Test results 142
4.2. Constrained random verification: instruction level testing 144
4.2.1. Verification framework 145
4.3. Constrained random verification: instruction sequence testing 148
4.3.1. Automatic program generation 151
4.4. Verification of the plasticity processor bus 151
4.5. Writing software for the plasticity processor 152
4.6. From source code to program execution 154

5. Hardware systems and their evaluation 155
5.1. FPGA prototype 155
5.1.1. Benchmarking with CoreMark 155
5.1.2. Influence of compiler optimization options 158
5.1.3. Influence of issue to retire latency 160
5.1.4. Influence of in-time issuing and write-through 161
5.1.5. Influence of branch prediction 161
5.1.6. Influence of variable latency load/store 162
5.1.7. Influence of instruction cache 163
5.1.8. Maximizing performance 164
5.1.9. Comparison to other processors 165
5.2. Prototype ASIC in 65 nm technology 169
5.2.1. Design overview 169
5.2.1.1. Global clock gate 170
5.2.1.2. General purpose input/output pins 171
5.2.1.3. Timer facility and interrupt controller 171
5.2.1.4. Program suspension with hardware breakpoints . . . 172
5.2.1.5. Processor options 172
5.2.2. Implementation and area requirements 173
5.2.3. Experimental results 174
5.2.3.1. Frequency and supply voltage operating range 175
5.2.3.2. Power consumption 178
5.2.3.3. Effect of clock gating 180
5.2.3.4. Power consumption by individual instructions 180
5.3. BrainScaleS wafer-scale system with non-programmable STDP 182
5.3.1. Simulation results: updating performance 184
5.3.2. Verification of automatic weight update logic in hardware . . . 185
5.3.3. Test of evaluation comparator 186
5.3.4. Event transmission crosstalk 190

5.4. In preparation: BrainScaleS wafer-scale system with plasticity processor 190
5.4.1. Design overview 191
5.4.1.1. Bus structure 192
5.4.1.2. Rate counter 193
5.4.2. Simulation results: weight updating performance 193
5.4.2.1. Benchmark programs 193
5.4.2.2. Results 197
5.4.3. Area requirements 201
5.4.4. Status of the implementation 203

6. Discussion and outlook 205
6.1. Discussion of main results 206
6.1.1. Abstract hybrid hardware model and results for reward-modulated STDP 206
6.1.2. Plasticity processor 207
6.1.3. Instruction set extension for operations on neuromorphic synapses 208
6.1.4. Non-programmable STDP implementation 209
6.1.5. 65 nm prototype 209
6.1.6. Plasticity processor in the BrainScaleS wafer-scale system . . . 211
6.1.7. From guidelines to implementation 214
6.2. Directions for future hardware 215
6.2.1. Maximizing performance in the HEPP design 215
6.2.2. Maximizing flexibility in future 65 nm systems 217
6.3. Conclusion 220

A. Tabular description of used neural network models 223

B. Supported Power ISA subset 225

C. Supplemental design description 230
C.1. External interfaces of the plasticity processor 230
C.1.1. RAM interface 230
C.1.2. Plasticity processor bus interface 230
C.2. Scheduling state machine graph 231
C.3. OCP configuration options of the plasticity processor bus 233
C.4. Synapse array interface signals 233

Glossary 235

References 239

List of Figures

1.1. The BrainScaleS wafer-scale system 20

2.1. Synapse schematic 31
2.2. Overview of the Abstract hybrid Hardware Model 32
2.3. Local analog processing 34
2.4. Simplified schematic for the drift model 36
2.5. Network model for reward modulated STDP 39
2.6. Simulation results of unconstrained baseline model 42
2.7. Comparison for generality of results 43
2.8. Sweeping the learning rate 43
2.9. Performance with discretized weights 46
2.10. Effect of weight discretization 47
2.11. Finding the optimal threshold and update constant parameters 49
2.12. Performance with threshold readout 49
2.13. Variation of analog drift 51
2.14. Threshold readout with mismatch 53
2.15. Reward timing effects on performance 54
2.16. Performance with delay and analog noise 56

3.1. Locating the processor in the Abstract Hybrid hardware Model (AHM) 60
3.2. Plasticity processor pipeline block diagram 63
3.3. Instruction streamer block diagram 67
3.4. Block structure of the instruction cache 70
3.5. Result shift register 73
3.6. Scheduling write back with RSR 74
3.7. Scheduling write back with lookup cache 75
3.8. Multi cycle instruction decoding 81
3.9. Scheduling instructions to functional units 83
3.10. Simplified scheduling state machine graph 85
3.11. Back-end overview 86
3.12. Bus in the abstract hybrid hardware model 94
3.13. Bus request 95
3.14. Plasticity processor bus building blocks 96
3.15. Example bus fabric 98

3.16. Locating the synapse interface in the AHM 102
3.17. Block diagram of the synapse array 103
3.18. Synapse array read timing 106
3.19. Synapse array write timing 107
3.20. Synapse array STDP evaluation 108
3.21. Synapse array accumulation reset 108
3.22. Non-programmable STDP in the AHM 109
3.23. Structure of the non-programmable synapse array adapter 111
3.24. State diagram of the STDP access state machine 114
3.25. Data path for automatic weight update computation 117
3.26. Location of the synapse array adapter for programmable STDP in the AHM. 118
3.27. Synapse array interface adapter for programmable STDP 120
3.28. State diagram of the sequencer state machine 121
3.29. State diagram of the bus FSM 125
3.30. SYNAPSE functional unit overview 136

4.1. Instruction level testing work-flow 144
4.2. Class diagram for instruction level testing 145
4.3. Class diagram for instruction level testing (2) 147
4.4. Class diagram for instruction level testing (3) 148
4.5. Class diagram for automatic program generation 149
4.6. Automatic program generation work-flow 150
4.7. Bus topology for bus verification 151
4.8. Programming workflow 154

5.1. FPGA evaluation platform 156
5.2. Hardware design for CoreMark evaluation 157
5.3. Influence of compiler options on performance 159
5.4. Influence of instruction latencies 160
5.5. Influence of branch prediction on CoreMark performance 162
5.6. Influence of the instruction cache size on performance 164
5.7. Maximizing performance on FPGA platform 165
5.8. Performance comparison to other processors 166
5.9. Photo of 65 nm prototype system 168
5.10. Block diagram of 65 nm prototype ASIC 169
5.11. Global clock gate schematic 170
5.12. The 65 nm design 173
5.13. Area breakdown of the 65 nm chip 174
5.14. Validated frequency and supply voltage domain 176
5.15. Low-voltage operation 177

5.16. Power measurement for 65 nm prototype 178
5.17. Transient power consumption during CoreMark execution 179
5.18. Power consumption by individual instructions 181
5.19. Photo of the BrainScaleS wafer-scale system 183
5.20. Speed of non-programmable STDP controller 184
5.21. Results from evaluation measurement with null read pattern 188
5.22. Fixed-pattern variation of comparator switch-points 189
5.23. Block diagram of BrainScaleS wafer-scale system with embedded plasticity processor 191
5.24. Bus structure of HICANN with Embedded Plasticity Processor 192
5.25. Two-factor weight update 194
5.26. HEPP weight updating performance 198
5.27. HEPP probabilistic weight updating performance 199
5.28. Area breakdown of HEPP design 202
5.29. Floorplan after placement of standard cells 204

6.1. Maximizing performance in the HEPP design 216
6.2. Maximizing flexibility in future process technologies 218

List of Tables

2.1. Model options for R-STDP 45
2.2. Simulation parameters with threshold readout 50

3.1. List of write back channels 75
3.2. Functional unit interface definition 87
3.3. Non-programmable adapter commands 112
3.4. Contents of words in the sequencer store 123
3.5. Defined opcodes 124
3.6. Address space layout 125

4.1. List of test programs used for program level testing. 141

5.1. Parameter space for CoreMark analysis 157
5.2. Configurations used for CoreMark analysis 158
5.3. Fit-parameters for power dependencies 179

B.1. Implemented registers 225
B.2. Implemented instructions 226
B.3. Interrupt vector table 229

C.1. Control word signals 232
C.2. OCP options of the PPB 233
C.3. Synapse array interface signals 233

1. Introduction

The human brain is a computing device with remarkable properties: with a power consumption on the order of 25 W (Kandel et al., 2000) and a volume of approximately 1.3 liters (Scahill et al., 2003), it can process complex sensory inputs, control the movement of our body, and perform abstract reasoning. Neither is it understood how these functional capabilities are achieved, nor are we currently capable of building machines with comparable abilities.

An outstanding property of neural tissue is its massive parallelism of comparatively slow units. In the neocortex of humans, for example, there are 20 billion neurons (Pakkenberg and Gundersen, 1997), each performing local, time-continuous, analog processing on the millisecond timescale. These neurons are interconnected at their contact points by synapses, over which signals are transmitted from one to the other. Interconnection can exhibit a very large fan-in: in neocortex of humans, neurons have thousands of synapses on average (Pakkenberg et al., 2003). The input received via these synapses affects the time-course of the neuron's membrane potential. If the membrane potential exceeds a threshold voltage at the cell body, it evokes a sharp voltage spike, called action potential, which is transmitted to other neurons along a fibre, called axon. The transmission along the axon in form of spikes represents digital communication, while the local processing performed by the neuron is analog.

In contrast to this, classical digital computers of the von-Neumann architecture (von Neumann, 1945) follow a very different principle: These machines are subdivided into specialized units for control, memory, and arithmetic and other operations. The whole machine follows a single program stored in memory, operating on data represented with binary states. Computation is performed symbolically on the foundation of boolean logic, and sequentially following the globally stored program.

The idea of *neuromorphic hardware* is to build computational machines more akin to the nervous system (Mead, 1989; Douglas et al., 1995). A key concept here is that of a physical model: The dynamical behavior of the biological equivalent is reproduced by the physical properties of the model. In classical neuromorphic hardware, those are the fundamental properties of transistors using Complementary Metal Oxide Semiconductor (CMOS) technology. In such system there is a one-to-one correspondence of neurons and synapses between the physical model and the biological counterpart. Every neuron is implemented by a distinct transistor circuit that follows differential equations modeled after the biological behavior. Just as in neural tissue, the computation performed by the neuromorphic neuron is analog, using a voltage range to represent its state.

The motivation for neuromorphic hardware is two-fold: On the one hand is the goal to develop technology that makes capabilities of the brain accessible in artificial systems. Most prominently, the ability to do complex processing with very little energy as compared to classical digital computers drives this endeavour. This is of interest, because performance of classical computers today is limited by their power consumption (Borkar and Chien, 2011; Fuller and Millett, 2011). A further desired ability is, that neural circuits are robust to imperfections of the underlying devices. For example, transmission via synapses exhibits considerable trial-to-trial variability due to the probabilistic release of neurotransmitters (Otmakhov et al., 1993; Katz, 1969). Also, the substrate can adapt to local damage by reorganizing the network structure, for example after a stroke (Weiller et al., 1992). Perhaps the most interesting feature of biological neural networks is their ability to learn. In contrast to classical computers, which are programmed to perform a certain task, neural networks learn their function from experience. In operant conditioning experiments, an animal is trained simply by providing reward and punishment depending on its actions (Staddon and Niv, 2008).

A second goal of neuromorphic hardware is to aid in the research of brain function. Here, it can offer a computationally efficient platform for large-scale network experiments (Brüderle et al., 2010). The conventionally used approach is to simulate networks on classical digital computers. However, compared to a physical model this incurs a large overhead, since the neuronal state has to be represented in an abstract symbolic form, and the dynamical behavior is numerically simulated with complex arithmetic units. Therefore, classical simulations of large scale neural networks require supercomputers (Markram, 2006; Ananthanarayanan et al., 2009) that come with high power consumption (Alam et al., 2008; CompuGreen, 2012). This limits the size of networks that can be studied, and the time frame over which behavior can be observed.

The ability to learn is based in the *plasticity* of neural tissue. Every aspect of the network, from neurons over synapses to the structure of the network, changes over time depending on the activity in the network. Including powerful plasticity mechanisms into neuromorphic hardware devices has the prospect of making learning computers possible that can be trained much like animals using reward and punishment. Devices with limited plasticity mechanisms have been demonstrated in the past, for example to counter the effect of process variations in Very Large Scale Integration (VLSI) circuits (Cameron et al., 2005; Bill et al., 2010), or for learning of pattern classification (Hafliger, 2007; Mitra et al., 2009; Sheik et al., 2012). Most neuromorphic systems built to date focus on plasticity of synapses, especially implementing Spike-Timing Dependent Plasticity (STDP) (Vogelstein et al., 2003; Indiveri et al., 2006; Schemmel et al., 2007; Ramakrishnan et al., 2011; Seo et al., 2011). However, biology paints a rather diverse picture of plasticity as will be shown in Section 1.2. This makes it unlikely, that a single mechanism or rule is sufficient to match the learning capabilities of the brain. On the other hand, the functional consequences of all the different effects found in biological experiments are to a large extent not clear. A large-scale experimental platform that is under full control by the operator can help to study these consequences. Neuromorphic hardware with flexible plasticity can be such a platform.

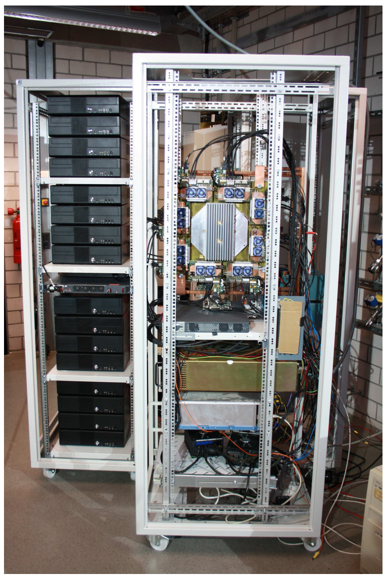
The work presented in this thesis follows a new approach, diverging from the classical idea of neuromorphic hardware with analog computing, to provide a highly flexible plasticity implementation. Instead of realization as a pure physical model, I present a hybrid system that is in part physical model and in part a classical processor of the von-Neumann architecture. This allows for advantages from both worlds: The physical model enables energy efficient, time-continuous, and massively parallel processing with small area requirements. The classical processor on the other hand adds flexibility in the implementable learning rules. I will show the usefulness of this approach using a specific reward-based learning rule as example (Chapter 2), describe in detail the hardware design necessary for such a system (Chapter 3), and present results obtained in simulations and with hardware prototypes (Chapter 5) to judge the implementability of this architecture (Chapter 6). The rest of this chapter gives background information on the preexisting hardware framework, and introduces models of plasticity.

1.1. The BrainScaleS wafer-scale system

The work presented in this thesis builds upon the hardware system designed and built within the FACETS and BrainScaleS projects (FACETS, 2010; BrainScaleS, 2012). The so called BrainScaleS wafer-scale system employs complete 20 cm wafers to realize high interconnection densities. The wafers are produced in a conventional 180 nm CMOS process and then interconnected in a post-processing step that adds an additional metal layer on top of the wafer. This way, connections across reticle boundaries, which can normally not be realized directly, are possible, so that the whole wafer can be used as one common substrate. On one such wafer, there are 40 million synapses and up to 180 thousand neurons. One neuron circuit can receive input from 224 synapses. To realize cortical scales, up to 64 neuron circuits can be combined to form a single neuron with 14 336 synapses, while simultaneously reducing the amount of totally available neurons. This inspired the name for the 5×10 mm High Input Count Analog Neural Network (HICANN) chip-unit, that is repeated on the wafer to build the substrate. These HICANN units can also be built as individual chips. Figure 1.1A shows the system with one wafer-module and the cluster of control computers. The wafer-module holds support logic for power and communication. The control cluster configures the hardware, sends stimulus, and receives the resulting activity. It is also intended to simulate interactive environments for the neural system on the wafer. The post-processed wafer is shown in Figure 1.1B.

Action potentials are transported from neurons to synapses by an asynchronous

A System



B Wafer-scale integration

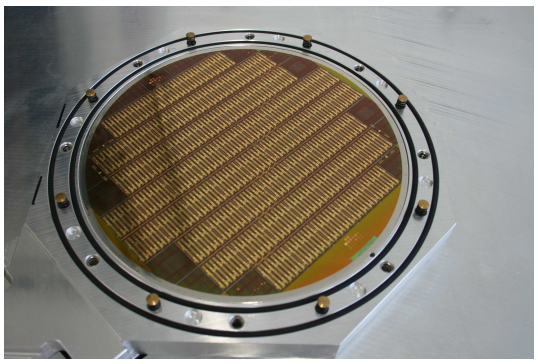


Figure 1.1.: (A) The BrainScaleS wafer-scale system in the lab. The left rack holds the cluster of control computers operating the neuromorphic system in the rack to the right. Here, the current variant with one wafer module is shown. Photo by S. Schrader. (B) Post-processed wafer. Photo by A. Grübl.

event network. This network is connection switched, with programmable switch matrices distributed over the whole wafer. An additional off-wafer network is available to interconnect multiple modules, to provide stimulation from external sources, and to observe the activity of the network (Scholze et al., 2011).

For further reference see Schemmel et al. (2008, 2010); Millner et al. (2010); Millner (2012). More detailed introductions are also given by Brüderle (2009) and Millner (2012).

1.2. Models of plasticity

Plasticity refers to the continuous activity dependent change of properties in a neural network. This change affects neurons, synapses, and the structure of the network as a whole. Cudmore and Desai (2008) provide a review on plasticity of neuron parameters. This so called intrinsic plasticity affects the electrical properties of the neuron. Further, the structure of the neural network itself changes over time in a process referred to as structural plasticity. This involves the creation of new neurons, growth processes of dendrites and axons, and the formation and elimination of synapses (Lamprecht and LeDoux, 2004; Leuner and Gould, 2010). The change of the strength of connections between neurons, referred to as synaptic plasticity, is widely believed to be at the foundation of learning and memory in neural systems (Martin et al., 2000). Although the concepts developed in this thesis address all forms of plasticity, synaptic plasticity and especially the biologically realistic spike-timing dependent plasticity rule (see next section) is the most important application for the hardware learning system, due to its functional importance for learning.

1.2.1. Spike-timing dependent plasticity

Spike-Timing Dependent Plasticity (STDP) is a synaptic learning rule. The change of synaptic weight depends on the relative timing of action potentials, generated by the neurons that the synapse interconnects (Gerstner et al., 1996; Markram et al., 1997; Bi and Poo, 1998). Reviews are given by Morrison et al. (2008) and Caporale and Dan (2008), a historical overview is presented by Markram et al. (2011).

Neurons communicate by the exchange of action potentials or spikes: sharp increases of membrane potential caused by the opening of ion-channels on the cell (Holz and Fisher, 1999). At a synapse, action potentials of the *presynaptic* neuron trigger the release of chemicals to communicate the event to the *postsynaptic* neuron. The impact the synapse has on the membrane voltage of the postsynaptic neuron is called its efficacy or weight. The STDP rule states, that the change in weight Δ depends on the time difference $\Delta t_{ij} = X_i - Y_j$ between a presynaptic action potential at time X_i and a postsynaptic one at time Y_j according to the STDP learning function $s\left(\Delta t_{ij}\right)$:

$$\Delta = s\left(\Delta t_{ij}\right) \tag{1.1}$$

In its canonical form the learning function is given as

$$s\left(\Delta t_{ij}\right) = \begin{cases} f_{+}\left(w\right)\exp\left(\frac{-\left|\Delta t_{ij}\right|}{\tau_{+}}\right) & \text{if } \Delta t_{ij} > 0 \\ -f_{-}\left(w\right)\exp\left(\frac{-\left|\Delta t_{ij}\right|}{\tau_{-}}\right) & \text{if } \Delta t_{ij} \leq 0 \end{cases} \tag{1.2}$$

with the synapse weight w, time constants $\tau_{\pm}$ and weight dependencies $f_{\pm}(w)$. Whether a synapse is potentiated or depressed depends on the temporal order of pre- and postsynaptic firing. Pre-before-post leads to potentiation, post-before-pre to depression. If s does not depend on the weight ($f_{\pm}(w) = 1$), the range of allowed weights must be limited to $w \in [0, w_{\max}]$ to prevent unrealistic situations, where the weight would grow without limits. Such rules are referred to as additive, since an increment determined only by timing is added to the weight. A limitation arises naturally, if the weight dependence is multiplicative:

$$f_{+}(w) = \gamma\left(w_{\max} - w\right) \tag{1.3}$$
$$f_{-}(w) = \lambda\gamma w \tag{1.4}$$

with proportionality factor γ and asymmetry λ. The asymmetry parameter λ controls the difference in magnitude of the two branches of the learning function s. A generalization is the variant by Gütig et al. (2003):

$$f_{+}(w) = \gamma\left(w_{\max} - w\right)^{\mu} \tag{1.5}$$
$$f_{-}(w) = \lambda\gamma w^{\mu}. \tag{1.6}$$

By selecting μ the rule can be configured to be additive ($\mu = 0$) or multiplicative ($\mu = 1$).

The form of the weight dependence has a relevant effect on the evolution of weights in a neural network. For example, under stimulation with uncorrelated action potentials that are sent with random firing intervals drawn from a Poisson distribution, additive rules lead to a bimodal weight distribution in equilibrium, where weights are either at the maximum or the minimum of the allowed range (Morrison et al., 2008). This holds true for rules with weak weight dependence ($\mu \ll 1$). For multiplicative rules and most intermediate values of μ, the equilibrium distribution is unimodal. According to observations in biology, unimodal distributions seem to be more realistic (Morrison et al., 2008).

To fully define the STDP rule, it must be stated which spike pairs (i, j) elicit weight updates. This is determined by the spike pairing rule. Besides all-to-all pairing that considers each possible combination of i and j, several nearest neighbor variants are in use (Morrison et al., 2008). Hardware systems described by Schemmel et al. (2006, 2010) use a reduced symmetric nearest neighbor rule.

By itself STDP is an unsupervised learning rule. It detects temporal coincidences in activity between pre- and postsynaptic neurons. An important feature is the sharp separation in time for pre-before-post ($\Delta t_{ij} > 0$) and post-before-pre ($\Delta t_{ij} < 0$) spike pairs.

STDP provides a number of functional capabilities (see for example Sjöström and Gerstner (2010) for a list). Examples would be latency reduction for repetitive inputs (Song et al., 2000), the development of receptive fields in cortex (Song and Abbott, 2001), and tuning for sound source localization (Gerstner et al., 1996).

1.2.2. Phenomenological models from biology

The form introduced in the previous section is commonly used in simulation studies. However, biological experiments draw a more complex picture of the rule. Other factors than pre- and postsynaptic timing can influence the amount of weight change. Measurements by Sjöström et al. (2004) show that depression can be induced by changing the postsynaptic membrane potential without any postsynaptic action potentials. Magnitude and timing of weight change are identical for both forms of induction. This would indicate, that membrane potential is the more fundamental factor than spike timing.

Measurements by Markram et al. (1997) and Sjöström et al. (2001) also show that firing rates in addition to timing influence STDP. If spike pairs are presented at high frequencies, synapses are strengthened independent of Δt_{ij}. So there is no depressing branch in the learning function s above a certain rate threshold.

In addition to frequency, STDP depends on the short term history of spikes (Froemke et al., 2010a). When using for example triplets instead of spikes, where a post-pre pair is followed by a post spike with 10 ms time intervals, the synapse is strengthened. If however pre-post-pre triplets are applied, no change is observed (Wang et al., 2005). Equation 1.2 predicts no change in both cases. As shown by Froemke et al. (2010a) short-term modulation and frequency dependence can be included into Equation 1.1 via additional suppression terms that depend on the time since last spike for the pre- and postsynaptic neuron.

As reviewed by Froemke et al. (2010b) STDP depends on the specific location of the synapse in the dendritic tree. The learning function s gradually changes with distance to the soma of the neuron. Synapses close to the soma exhibit classical STDP. With increasing distance, potentiation is suppressed until distal synapses only exhibit depression, even for pre-post spike pairs. This can be reverted again to potentiation by depolarizing the dendrite or by dendritic spikes.

STDP is found for synapses in various species and different brain regions. However, the general shape of the learning function s can be very different. For example, the temporal width $\tau_{\pm}$ of the interaction window can vary (Caporale and Dan, 2008). The learning function can be inverted in part or over the whole time range (Morrison et al., 2008). There can be negative offsets if single pre or postsynaptic spikes lead to depression. A collection of different shapes is given by Abbott and Nelson (2000) and Caporale and Dan (2008).

It is known, that the dynamics and strength of synaptic transmission are modulated by various chemicals, so called neuromodulators (Seol et al., 2007; Pawlak and Kerr, 2008; Zhang et al., 2009; Pawlak et al., 2010). For example, the neuromodulator dopamine can change depression for post-before-pre pairs into potentiation (Pawlak et al., 2010). Neuromodulators can act as a "third-factor" in addition to pre- and postsynaptic firing times in Equation 1.2. They provide a control mechanism of the unsupervised "two-factor" STDP rule affecting many synapses in a certain area. Here, especially dopamine is of interest, because of its connection to reward (Schultz et al., 1997): The dopamine signal seems to signal an error between predicted and actually received reward. This leads to the field of reward-based learning using STDP with neuromodulation as third-factor. This is discussed in the next section.

In contrast to the plain STDP rule introduced in Section 1.2.1, the functional consequences of the effects described in this section are to a large part not yet clear. Therefore, it is difficult to say what can be gained by including all or some of them in a hardware model. On the other hand, hardware with biologically realistic plasticity mechanisms offers the opportunity to help in that research by providing a platform for large-scale network experiments.

1.2.3. Reward-modulated STDP

In the classical machine learning approach of reinforcement learning (Sutton and Barto, 1998) an agent learns to perform a given task by selecting actions that maximize the total reward it receives from the environment. This idea is inspired by operant conditioning experiments (Rescorla and Wagner, 1972; Rescorla, 2008) where an animal is trained using only reward or punishment as feedback. The reward is a single scalar number that determines how well the agent performed. This stands in contrast to supervised learning, in which a teacher supplies the correct actions to take in a learning phase. In recent research, a connection between reinforcement learning and modulated STDP is beginning to emerge (Izhikevich, 2007b; Farries and Fairhall, 2007; Florian, 2007; Legenstein et al., 2008; Frémaux et al., 2010; Potjans et al., 2011). A central requirement is the solution of the so-called temporal credit assignment problem: There is a temporal delay between action and the reward it will cause. The neural network somehow has to know, which synapses contributed to the reward and should be modified. A solution is to maintain a per-synapse eligibility trace $e(t)$, that keeps a memory of recent spike activity.

$$\dot{e}(t) \quad = \quad \begin{cases} s(\Delta t_{ij}) & \text{if } t = t_{ij} \\ -\frac{1}{\tau_e} e(t) & \text{otherwise} \end{cases} \tag{1.7}$$

with the time derivative $\dot{e}(t)$, the decay time constant τ_e, and the time of the spike pair $t_{ij} = \max \left\{ X_i, Y_j \right\}$. It is important to note, that synaptic weights are not immediately changed by the occurrence of spiking activity as in the case of the plain two-factor rule. Instead, only the local eligibility trace is affected. The environment gives the reward R in reaction to the activity of the agent. However, using R directly to modulate weight change would introduce an unsupervised bias, as Frémaux et al. (2010) have shown. Instead, the success signal $S = R - \langle R \rangle$ given as difference between expected reward $\langle R \rangle$ and current reward R is used:

$$\Delta = \eta e(t) S \tag{1.8}$$

with the learning rate η. S could be implemented by dopamine signals in the brain, which exhibit such a differential behavior (Schultz et al., 1997). Equation 1.8 is called R-STDP rule (Izhikevich, 2007b; Florian, 2007) and replaces Equation 1.2 for the computation of weight changes. This rule is used in a detailed simulation study described in Chapter 2 to analyze the effect of hardware constraints on its performance in a simple spike train learning task.

1.3. Design goals for neuromorphic hardware

This thesis focuses on neuromorphic hardware systems that follow three main design goals. These goals are listed and motivated in the following text:

Large scale Neurons in the neocortex of the human brain count in the tens of billions ($21.5 \times 10^9 \pm 38\%$, Pakkenberg and Gundersen, 1997). The entire brain is even five times larger. To understand how the brain or individual substructures work, models of comparable size have to be formulated and analyzed. The classical approach is to perform simulations on super-computers employing general purpose microprocessors. However, cortical or brain scales are not yet within reach. The simulation reported in Ananthanarayanan et al. (2009) reaches 10^9 neurons using a BlueGene/P (Alam et al., 2008) with 147 456 cores. The Blue Brain Project (Markram, 2006) targets a much more detailed simulation of about 10k neurons on a BlueGene/L. By building hardware systems specialized for the emulation of neural networks, it is possible to increase efficiency and therefore reach larger scales. It is the goal of neuromorphic hardware systems designed for scalability to enable the study of networks at relevant sizes and do so more efficiently than general purpose supercomputers.

Acceleration in time Animals are able to alter their behavior based on past experiences by learning. This is often a tedious process requiring repeated interaction with an environment, for example when learning how to walk or to play a musical instrument. To study these processes that take days, weeks or even years in an artificial system requires an acceleration compared to the biological equivalent. The problem is aggravated as computer experiments often require a lot of repetitions, where in each iteration the experimenter evaluates the results and modifies the model. Reducing this turn-around time enables quick exploration by the experimenter and therefore increases the efficiency of research. With this in mind, neuromorphic hardware should provide accelerated emulation of networks.

For some cases, the acceleration can be seen as a hindrance: robotics applications for example operate necessarily in a not accelerated environment. If control models taken from biology are to be employed, environment and controller operate at different time scales. Therefore, the ideal hardware allows a wide range of acceleration factors ranging down to real time operation.

Flexibility To allow exploration of network models a system must allow to change its configuration. In general, the more flexible, the better, but of course this reduces the ability to specialize the design. So in a way, this goal is in opposition to the first two goals. This gives rise to trade-offs between flexibility and performance in terms of achievable size and acceleration. Supercomputers are on the extreme of flexibility. In principal, they can simulate arbitrary models, only limited by available memory and perhaps suffering long simulation times. Neuromorphic hardware is often rather on the other extreme, for example with neuronal dynamics fixed in full-custom circuits. One aspect of this thesis is the question of how much flexibility is possible regarding plasticity, while still achieving the first two goals.

2. Theory

In this chapter I approach the topic of neuromorphic hardware design from a theoretical perspective. Starting out from the plasticity models described in Section 1.2 a list of high level requirements is derived and an abstract model of a hardware implementation developed. This model is then studied in simulations to develop guidelines for the hardware implementation.

2.1. Requirements for hardware

The models listed in Section 1.2 require an increasing amount of capabilities.

2.1.1. Two-factor STDP

Two-factor STDP is a conceptually simple algorithm. It represents a process local to the synapse, depending only on pre- and postsynaptic activity. What makes it difficult to implement in hardware is, that this process has to be available in parallel for every synapse and be active for every spike. In the neocortex in humans there are between 10^4 and 10^5 synapses per neuron (Peter et al., 1979). So the algorithmic complexity at the individual synapse has to be kept at a minimum.

The basic two-factor STDP rule requires the ability to measure time differences between pairs of pre- and postsynaptic spikes at the individual synapse. This implies some form of memory, because information about past presynaptic spikes must be used for every postsynaptic one and vice versa. In principal, this memory is only necessary for every neuron or other source of events. However, especially in large-scale systems, distribution to the necessary synapses may be problematic, as the source may be physically far away and have a high fan-out to many synapses.

In the typical formulation, change of weight depends exponentially on the time difference. Therefore, a method for exponential weighting of the measured timing is required. This can be combined with the measurement itself, for example using local traces as described in Morrison et al. (2008). Of course, an implementation of STDP must provide modifiable synaptic weights. So in summary, this rule has the following requirements:

1. Processing per synapse and per spike.
2. Measurement of spike timings.

3. Exponential weighting of time differences.

4. Modifiable synaptic weights.

2.1.2. Reward-modulated STDP

Reward-modulated STDP is a three-factor extension of two-factor STDP and has therefore the same requirements. Additionally, it needs an environment providing rewards, with which the system can interact. This environment could for example be simulated on an attached computer - or cluster of computers - or it could be a robotic system. The concept of environment used here is the same as in reinforcement learning (Sutton and Barto, 1998).

The core idea of the rule is to modulate STDP with a reward derived global signal. So the hardware needs to provide a means to calculate this signal from the reward, distribute it to the STDP mechanism and multiply the weight change with it.

An important aspect of reward-based learning rules is, that the reward can arrive with some delay to the actions that have caused it. This is known as the distal reward or temporal credit assignment problem (Izhikevich, 2007a). In the model I have described in Section 1.2 an eligibility trace is used to solve this problem. The trace contains information about the recent activity at the synapse. Therefore, a per-synapse eligibility trace is necessary for this rule. In summary, there are these requirements:

1. All requirements of two-factor STDP.

2. An interactive, reward providing environment.

3. Computation and distribution of and multiplicative modulation with a global signal.

4. A per-synapse eligibility trace.

2.1.3. Phenomenological models

Section 1.2.2 lists a number of observations from biology regarding synaptic plasticity. Additional factors, e.g. membrane potential of the postsynaptic neuron or the mean firing rate, can affect the weight change. For an implementation, this necessitates access to these factors by the weight updating mechanism. For example, STDP depends on the short-term history of the pre- and postsynaptic activity. This requires additional state variables for each synapse with their own temporal dynamic. Observations also show, that the shape of the STDP learning function can vary drastically between cell types and brain regions. To model this, a concept of location and type for the synapse is needed, i.e. the weight updating process has to know where the synapse is located and of what type it is. Such a location should also reflect the position in the dendritic tree, to cover this dependency. Different learning functions have to be reflected in the ability to use configurable weighting of the timing measurement. In summary, these observations lead to the following requirements:

1. All requirements of two-factor STDP.
2. Access to additional state variables (e.g. membrane potential, firing rate, short-term plasticity) that have their own dynamics.
3. Information about location of synapses in regions and on the dendritic tree.
4. Information about cell- and synapse-type.
5. Ability to configure the STDP learning function.

2.2. Abstract hybrid hardware model

This section formulates an abstract model of a hardware plasticity implementation. It represents a generalization of the concrete designs presented in Chapter 3. Using an abstract model simplifies a general analysis of the hardware design space.

2.2.1. Combining analog and digital computing

Classical neuromorphic hardware systems use analog circuits to build a physical model of synapses, neurons and their interconnections (Mead, 1990; Douglas et al., 1995). The term physical model means, that circuit dynamics as described by differential equations match those of the biological model. This is a very efficient way of implementing a system that behaves like a neural one. The neural network is *emulated*. In contrast, a digital computer encodes differential equations in a software program and numerically solves them. The state of the system is represented in abstract digital values stored in memory. The neural network is *simulated*.

While the analog approach is efficient, it is also inflexible, because once the dynamics are built into the circuit they can not be changed. Configuration options and modular design improve flexibility, but the dynamics stay restricted to a family of implementable models. For digital systems, flexibility is only limited by available memory and computational power. In practice this often means, that a more detailed model takes longer to simulate and the network consists of fewer neurons and synapses. An example for this are the two simulation studies mentioned in Section 1.3. The more detailed simulation could support fewer neurons.

While discussing the requirements (Section 2.1), it was already apparent, that plasticity rules show considerable diversity. A fully analog solution would have to include all possible features with individual configurable enable switches. Depending on the rule, parts of the circuit that are not needed would be disabled. In a digital implementation, the gained flexibility would be paid for by reduced efficiency. So why not combine both approaches into a hybrid system to get the best from both worlds? An analog part models common features, like the measurement of spike timings, in a time continuous and local fashion. Most of the higher features, like modulation or access to other state variables, are realized in software running on a processor. The processor, as a rather complex and therefore large component, would likely be shared by many synapses. In this case, software would iterate over the synapses, communicate with the local analog circuit and modify the weight according to its program and the local analog state. Such a hardware system is the focus of this thesis. The AHM formulated later in this section is based on this concept.

2.2.1.1. What type of digital part is needed?

The straight forward choice for the digital part of such a hybrid system is to use a conventional micro-processor: a device that executes a stream of instructions belonging to a program stored in memory to manipulate internal registers and variables also stored in memory. An Input/Output (I/O) unit is used to interface the analog part of the system. However, it is also conceivable to use a different design concept here, for example a command sequencer or a configurable finite state machine. I will come back to the idea of a command sequencer later, but mainly I will exclude alternative implementation styles for the digital part in this thesis. This is done for three reasons: 1) The concept of a processor is well understood with a large literature on implementation and optimization. 2) A processor provides great flexibility, while a hand-crafted system might be difficult to adapt in unforeseen use cases. 3) A processor can support high-level programming languages allowing everyone proficient in that language to use it.

2.2.2. The abstract hybrid hardware model

With these considerations about the basic architecture in mind, a model can now be formulated. The goal of this model is to capture fundamental aspects of a hardware system that follows the hybrid approach detailed in the previous section.

Synapses are organized in an array of N rows with M columns. A row shares the same presynaptic input. All synapses in one column are connected to the same neuron. An individual synapse is shown in Figure 2.1. It has to fulfill two purposes: controlling the strength of the effect a presynaptic pulse has on the postsynaptic neuron and implementing the local component of STDP. The former is controlled by the weight w, which modulates the incoming *pre* pulse. The modulated inputs from all synapses in one column are summed on a vertical line feeding to the postsynaptic neuron. The shape of the postsynaptic potential (PSP) does not necessarily have to be generated in the individual synapse, but can be created in a neuron input stage. The local part of

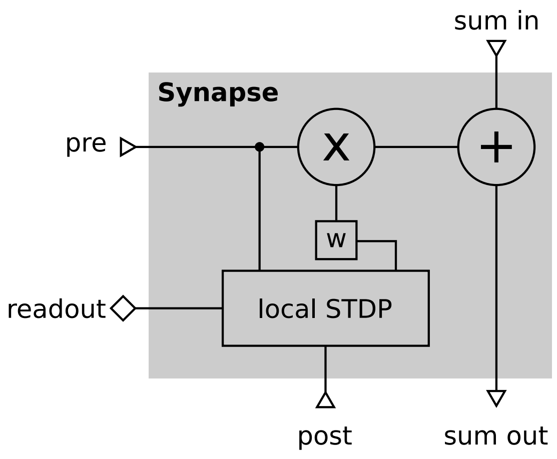


Figure 2.1.: A synapse in the abstract hybrid hardware model. It consists of a weight w that modulates presynaptic events. The output of all synapses belonging to one neuron is integrated, before it is presented to the neuron. The synapse also contains the local accumulation part of STDP.

the STDP implementation needs to see the *pre* and *post* signals and the weight. The *post* line is a feedback line from the neuron that is pulsed whenever the neuron fires.

Figure 2.2 gives an overview of the AHM. The synapse array is controlled by a synapse interface component that can selectively access the readout lines of individual synapses. Via an adapter module this interface is connected to the plasticity processor executing the plasticity program. The program and other data is stored in main memory, which the processor accesses via a bus. This bus also gives access to other components of the system that are indicated here as peripherals $1 \ldots K$. The bus can also be accessed from the outside to configure the system initially or to provide additional information during runtime, for example a reward signal. To allow for external access to synaptic weights, the adapter is also connected to the bus.

2.2.2.1. Using the abstract model for plasticity

To realize for example two-factor STDP in the AHM, the STDP component local to the synapse performs the timing measurement and exponential weighting. The result is stored locally until the processor - via adapter and synapse interface - reads it together with the weight. The program then calculates the new weight, writes it back to the synapse and indicates to the local STDP circuit, that the stored information was used to change the weight. The local circuit could for example sum the exponentially weighted time difference between pre- and postsynaptic spike pairs. When writing the new weight back, the sum would be set back to zero.

For more complex rules requiring additional state information, for example the

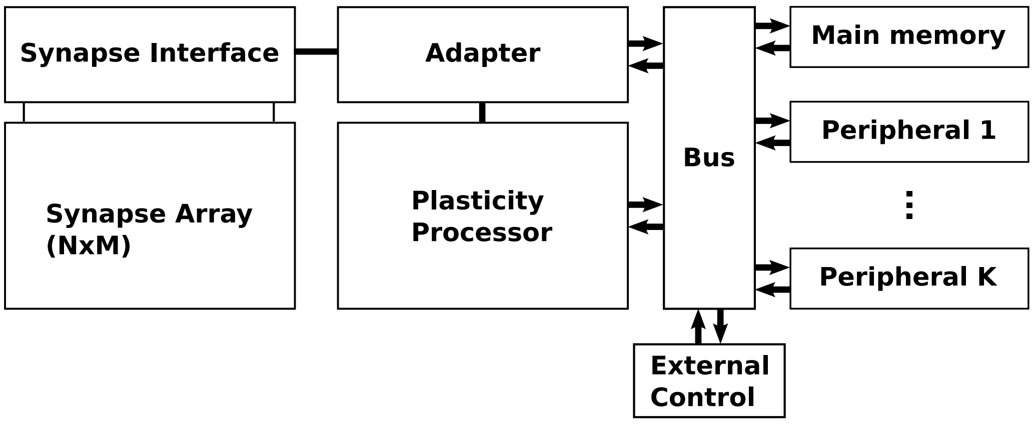


Figure 2.2.: Schematic overview of the abstract hybrid hardware model. It represents a high level view of the proposed plasticity implementation for neuromorphic hardware. See text for a description of the components.

membrane potential of the postsynaptic neuron, software would perform a bus access to an Analog to Digital Converter (ADC) peripheral when calculating the new weight. In reward based learning, the environment would send any rewards through the external interface and write a message in main memory. During the weight update, software would retrieve this message to modulate the amount of change.

2.2.2.2. Acceleration in time

The hardware system and therefore also the plasticity implementation should operate in an accelerated time domain (Section 1.3). This means, that the dynamics of the emulated network are equivalent to that of the mathematical model, but with a compressed time scale. If the model uses time ϑ, the hardware uses time t with the relation

$$\vartheta = \alpha t. \tag{2.1}$$

Where α is the acceleration factor.

2.2.2.3. Discretized weights

In general, weights can be stored as continuous analog values, discretized in a digital representation or as binary on/off switches. The AHM uses digitally represented weights of r-bit resolution, which are limited to the interval $[w_{\text{min}}, w_{\text{max}}]$. This defines the discretization step size

$$\delta_r = \frac{w_{\text{max}} - w_{\text{min}}}{2^r - 1}. \tag{2.2}$$

Thus, a weight w can assume one of the values $w_{\min}, w_{\min} + \delta, \ldots, w_{\min} + (2^r - 1)\,\delta_r = w_{\max}$. The special case of $r = 1$ corresponds to binary weights that either assume $w_{\min}$ or $w_{\max}$. In the limit of large r, the weight approaches a continuous representation.

Modifying discretized weights If a weight change of Δ is required by the learning rule, this has to be rounded to Δ_r, so that the resulting weight $w' = w + \Delta_r$ is representable. This leaves some freedom in choosing the rounding rule. The straightforward approach is to round to the nearest representable value. This has the consequence, that small weight changes are discarded, if they satisfy

$$|\Delta| \leq \frac{\delta_r}{2}. \tag{2.3}$$

For learning it is more important, that the overall effect on all weights follows the statistics of the learning rule. This opens the door for probabilistic rounding: The discretized change Δ_r is chosen with a probability proportional to the distance to the actual update Δ. If Δ is within the interval $[w_{\min} + (k-1)\delta_r, w_{\min} + k\delta_r)$, then $\Delta_r = w_{\min} + k\delta_r$ is chosen with probability p and $\Delta_r = w_{\min} + (k-1)\delta_r$ with probability $1 - p$. Given Δ, the average discretized change is

$$\begin{aligned} \langle \Delta_r \rangle &= (w_{\min} + k\delta_r)\,p + (w_{\min} + (k-1)\delta_r)\,(1-p) && (2.4)\\ &= w_{\min} + \delta_r(k-1) + \delta_r p. && (2.5) \end{aligned}$$

To get $\langle \Delta_r \rangle = \Delta$, one can choose

$$p = \frac{(\Delta - w_{\min}) - (k-1)\,\delta_r}{\delta_r}. \tag{2.6}$$

This way, averaged over all synapses and updates the discretized version matches the continuous one.

2.2.2.4. Local analog processing in the synapse

The component of the STDP circuit local to the synapse has to perform three tasks: measuring the timing of spike pairs, weighting the result according to the STDP learning function and accumulating results, until the plasticity program processes the synapse.

Measuring spike timing There are a number of possible pairing schemes to select pairs from the pre- and postsynaptic spike trains X_i and Y_j (Morrison et al., 2008). All-to-all pairing, for example, considers all presynaptic spikes occurring before a post-synaptic one, and vice versa for presynaptic events. The pairing scheme determines the set of spike pairs P evaluated at a synapse. Each pair $(i, j) \in P$ has a time difference

$$\Delta t_{ij} = X_i - Y_j. \tag{2.7}$$

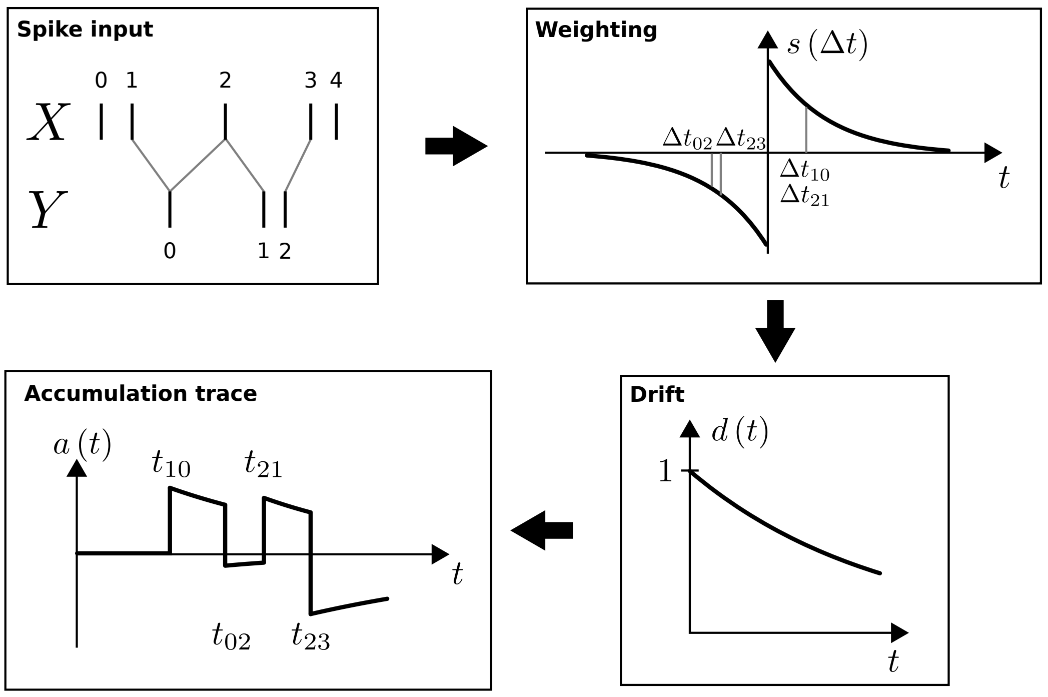


Figure 2.3.: Local analog processing by the accumulator part of the synapse. It measures the time differences of nearest neighbor spike pairs, and weights them according to the learning function $s(\Delta t)$. The result is subjected to drift according to the drift function $d(t)$ to produce the local accumulation trace $a(t)$.

It is positive for pre event X_i occurring before post event Y_j and negative otherwise. The occurrence of a pair is known at time $t_{ij} = \max\left\{X_i, Y_j\right\}$.

Local accumulation The time differences are weighted according to the STDP learning function $s\left(\Delta t_{ij}\right)$. The result is added to the local state variable $a(t)$ at the time of occurrence of the pair t_{ij}. In the time between pairs, $a(t)$ changes according to the drift function $d(t)$. At time t_r a reset mechanism initializes the trace to 0. This gives the following equation:

$$\dot{a}(t) = \begin{cases} s\left(\Delta t_{ij}, a\right) & \text{for } t \in \left\{t_{ij} | (i,j) \in P \,\text{with}\, t_{ij} > t_r\right\} \\ d(t,a) & \text{else} \end{cases} \tag{2.8}$$

$$a\left(t_r\right) = 0 \tag{2.9}$$

In Figure 2.3 local analog processing is illustrated by an example. Here, the symmetric nearest neighbor pairing scheme (Morrison et al., 2008) is used, with exponential weighting for $s\left(\Delta t_{ij}, a\right)$ and exponential decay for $d(t,a)$. Those are the settings used in Section 2.3.

Analog interface The synapse interface, triggered by the plasticity processor, evaluates the local trace a at time t_e by means of a readout and evaluation process. The evaluation function E converts the analog value a into a digital representation b:

$$b = E\left(a\left(t_e\right)\right) \tag{2.10}$$

An example for a possible implementations of the evaluation process would be an ADC discretizing the analog value to a digitally represented number. A special case used in Section 2.3 is comparison to a threshold Θ

$$\left(b_0, b_1\right) = E_\Theta\left(a(t_e)\right) = \begin{cases} (1,0) & \text{for } a(t_e) > \Theta \\ (0,1) & \text{for } a(t_e) < -\Theta \\ (0,0) & \text{else.} \end{cases} \tag{2.11}$$

2.2.2.5. Global digital processing by the processor

The plasticity processor controls the readout of synapses, calculates new weights and writes them back. This is modeled with an update function F that determines the weight change Δ

$$\Delta = F\left(b, w, P\right) \tag{2.12}$$

depending on the result b of the analog evaluation of the synapse accumulation trace, the synaptic weight w, and global parameters P. The global parameters P can represent arbitrary information, for example network state or neuromodulator concentrations. The complexity of F is only limited by available resources, i.e. the amount of main memory and processing speed.

2.2.2.6. Drift of analog storage

The drift function $d(t, a)$ models a characteristic aspect of analog memory in deep sub-micron process technologies. Due to leakage currents of various origins (Roy et al., 2003), the value stored in analog memory changes over time. The AHM assumes, that on the occurrence of a spike pair the trace a is increased or decreased precisely with the value given by $s(\Delta t_{ij}, a)$. In the time between those updates, the trace drifts according to the drift function d. The precise shape of d depends on the used circuit. Behavior is not limited to decay, but for example drift to the upper supply voltage is also possible.

This is not only a parasitic effect. For example, the eligibility trace of reward-modulated learning rules typically is exponentially decaying in theoretical models (Florian, 2007; Izhikevich, 2007b; Frémaux et al., 2010). The eligibility trace can be identified with the local accumulation variable $a(t)$. To precisely implement such models, it might be necessary to tune the circuit to exhibit a specific drift function $d(t, a)$.

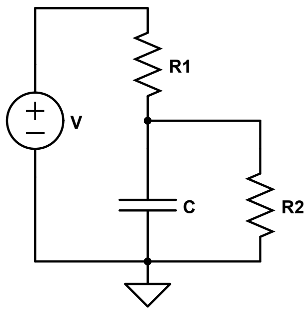


Figure 2.4.: Drift modeled with ohmic resistance to ground and to the supply voltage.

Ohmic drift If charge leaks from the capacitor through an ohmic resistance, $d(t,a)$ is proportional to the voltage difference to the leakage potential A:

$$d\left(t,a\right) = \lambda\left(A - a(t)\right) \tag{2.13}$$

Here, $\lambda = \frac{1}{\tau_e}$ is the inverse time constant. A circuit representing this model is shown in Figure 2.4. In the limit of infinite R_1, i.e. perfect isolation, the leakage potential $A = 0$ and $a(t)$ always decay towards 0. Respectively, in the limit of infinite R_2, $a(t)$ always grows towards the supply voltage.

Dual capacitors For VLSI hardware systems it is impractical to use negative voltages. However, $a(t)$ can assume negative values, if a post-before-pre pair is encountered. A possible solution is to use two capacitors and split up the accumulation into positive and negative components $a(t) = a_+(t) - a_-(t)$. To capture this in the AHM, a, s and d are extended to two-dimensional vectors $\boldsymbol{a}$, $\boldsymbol{s}$, $\boldsymbol{d}$.

$$\boldsymbol{a}(t) = \begin{pmatrix} a_+(t) \\ a_-(t) \end{pmatrix} \tag{2.14}$$

$$\boldsymbol{d}(t,\boldsymbol{a}) = \begin{pmatrix} \lambda_+\left(A_+ - a_+(t)\right) \\ \lambda_-\left(A_- - a_-(t)\right) \end{pmatrix} \tag{2.15}$$

The learning function is split into pre-before-post and post-before-pre part using the Heaviside function H:

$$\boldsymbol{s}(\Delta t_{ij},\boldsymbol{a}) = \begin{pmatrix} s(\Delta t_{ij},a)H(\Delta t_{ij}) \\ -s(\Delta t_{ij},a)H(-\Delta t_{ij}) \end{pmatrix} \tag{2.16}$$

An evaluation function E^v operating on $\boldsymbol{a}$ can be constructed from E by using the difference of the accumulating capacitors:

$$E^v(\boldsymbol{a(t)}) = E(a_+(t) - a_-(t)) \tag{2.17}$$

2.2.2.7. Mismatch

A second intrinsic aspect of analog hardware is device mismatch introduced by the manufacturing process (Kinget, 2007). Imperfections in e.g. transistor geometry cause variations in circuits that should ideally be identical. This effect, also referred to as fixed-pattern noise, leads to deviations between individual synapses in the AHM. Simply put, all synapses behave differently, but behavior of the individual one stays constant over time. As a simplification, the AHM integrates mismatch at two points: the drift and the evaluation function. For the n-th synapse, the drift function $d_n(t)$ and the evaluation function $E_n(t)$ represent the analog behavior of this particular synapse[1].

For the analysis in Section 2.3, mismatch is modeled using the following definitions: For the threshold comparison readout $E_{\Theta,n}(a)$, mismatch is modeled by drawing the threshold Θ_n from a Gaussian distribution

$$\Theta_n \in \mathcal{N}\left(\Theta, \sigma_\Theta^2\right), \tag{2.18}$$

with mean Θ and variance σ_Θ^2.

To integrate mismatch into the vectorized drift function $\boldsymbol{d}_n(t, a)$, parameters $\lambda_{n,+}^{-1} = \tau_{n,+}^{(e)}$ and $\lambda_{n,-}^{-1} = \tau_{n,-}^{(e)}$ are individually drawn from a Gaussian distribution

$$\tau_{n,\pm}^{(e)} \in \mathcal{N}\left(\tau_\pm^{(e)}, \sigma_{\tau_\pm^{(e)}}\right), \tag{2.19}$$

with means $\tau_+^{(e)}$ and $\tau_-^{(e)}$, and variances $\sigma_{\tau_+^{(e)}}$ and $\sigma_{\tau_-^{(e)}}$. To simplify analysis and keep the parameter space small, only the maximum and minimum of possible values are used for the leakage potential in the drift function:

$$A_{n,\pm} = \begin{cases} 0 & \text{for } \lambda_{n,\pm} \geq 0 \\ a_{\max} & \text{else,} \end{cases} \tag{2.20}$$

with $a_{\max}$ describing the maximum value $a_\pm$ can assume. This means, that a capacitor implementing the accumulation trace $a(t)$ can either drift towards ground or supply voltage.

2.2.2.8. Dynamic analog noise

While fixed-pattern noise stays constant over time, there is also dynamic noise, or trial-to-trial variation, on analog circuits. For example, repeatedly evaluating the same synapse using the evaluation function E will produce varying results. Noise is caused by a number of sources, for example shot noise or thermal noise (Gray et al., 2001). Additionally there are man-made sources, for example crosstalk between interconnect

[1]As long as only single synapses are considered, the index is left out for clarity.

lines or power supply noise (Shepard and Narayanan, 1997). Noise can be integrated into the AHM by introducing an additional term to the accumulation trace $a(t)$:

$$a^{\delta}(t) = a(t) + \delta_a(t) \tag{2.21}$$

$\delta_a(t)$ is distributed according to a Gaussian distribution with mean 0 and variance $\sigma_{\delta a}$.

2.2.2.9. Processing speed

The central aspect of the AHM is, that analog time-continuous circuits are combined with a digital, clocked processor. The processor accesses synapses for weight updates sequentially through the synapse interface (see Figure 2.2). While the processor is busy updating one synapse, all synapses continue to evolve according to their dynamics (Equations 2.8-2.9). So for the whole array updates do not occur simultaneously, but are stretched out over a duration of time. The n-th synapse in the array is updated at time

$$t_n = t_0 + \frac{n}{\nu_S}, \tag{2.22}$$

with the processor updating frequency ν_S measured in synapse updates per second. The updating speed ν_S is limited by the read and write access times of the array and the computational performance of the processor.

2.2.2.10. Communication latency

Whenever plasticity experiments are performed in a closed-loop setup, where additional information for the weight update is provided externally depending on the activity of the emulated network, the communication latency plays a role. Typically, external information represents a reward signal that is determined by the output of the network. Just like processing speed, as discussed in Section 2.2.2.9, communication latency adds an additional delay D_R compared to a theoretical model. Because weight updating can only commence after the reward is available, Equation 2.22 is extended by a delay term:

$$t_n = t_0' + \frac{n}{\nu} + D_R \tag{2.23}$$

Here t_0' is the hypothetical time of evaluation of the first synapse, if there was no delay.

2.3. Reward modulated STDP

The previous section formulated an abstracted hardware model, the AHM. This model is a generalization of the hardware designs presented in Chapter 3. By studying the model in simulations, the viability of the concept can be verified and guidelines for the

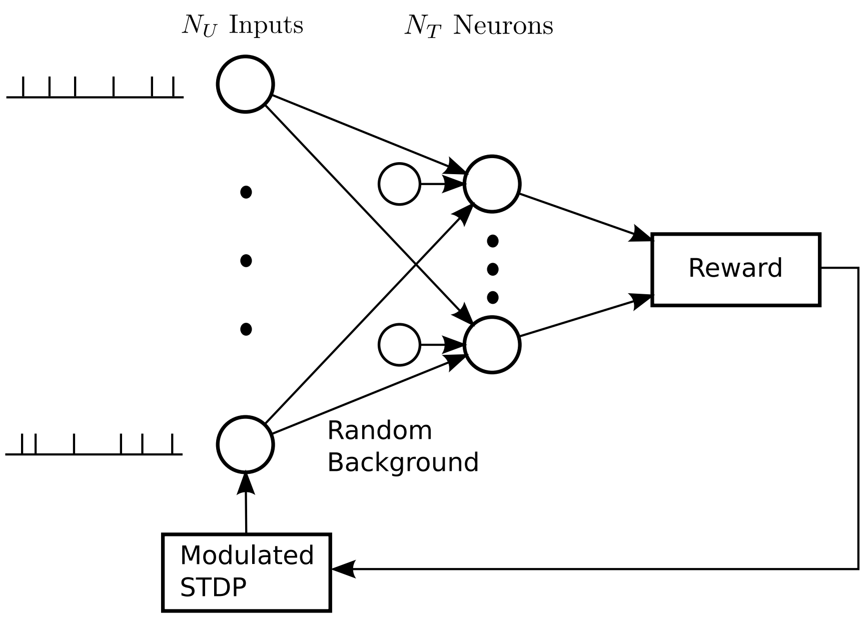


Figure 2.5.: Overview of the network used for reward modulated STDP. The network consists of two layers with feed-forward connections. Each of the $N_T = 5$ neurons sees all $N_U = 250$ inputs, and additional random background stimulation. The output firing pattern of the neurons is rewarded, and the reward modulates the weight change by STDP.

implementation developed. This section presents results from simulation of a reward modulated STDP learning rule.

The reward modulated STDP synaptic learning rule is described in Section 1.2.3. The learning task is based on the work by Frémaux et al. (2010). Findings presented here are in part submitted for publication (Friedmann et al., 2013). Figure 2.5 visualizes the benchmark network model used in the simulations. A single layer T of $N_T = 5$ neurons receives input from $N_U = 250$ stimulus sources and $N_B = 250N_T$ random Poisson sources providing background activity. All neurons see the same stimulus population U, but each receives random stimulation from disjoint, equally sized subsets of the background population B. The network is simulated in trials of $t_{\text{trial}} = 1\,\text{s}$ duration. At the end of each trial, the environment generates a reward R in response to the activity. This reward is then used with the eligibility trace local to the synapse to compute the change of synaptic weights w_{ij} between populations U and T according to Equation 1.8 ($i = 0 \dots N_U$, $j = 0 \dots N_T$). A tabular description of the simulated network is available in Appendix A.

Stimulation The input to the network consists of a defined pattern of $k = 0 \dots N_{\text{stim}}$ firing times S_{ik} of the individual input sources $i = 0 \dots N_U$. Times are drawn from a uniform distribution

$$S_{ik} \in \mathcal{U}\left(0, t_{\text{trial}}\right) \tag{2.24}$$

on the interval $[0, t_{\text{trial}}]$. For the simulations shown here, $N_{\text{stim}} = 6$. One specific stimulation pattern, generated in the described way, is denoted as S^*_{ik}. This pattern is used for all simulations if not noted otherwise to ensure comparability of results without the variability introduced by different patterns. The background sources emit spikes according to a Poisson process with firing rate ν_B.

Learning task The task the network has to learn defines how the reward is generated. Here a spike train learning task is used, where the output neurons have to reproduce a target pattern X_{target} of output spike times. The pattern is generated by simulating the network once with a set of reference weights $w_{ij} = W_{ij}$ and recording the output pattern. This way, it is guaranteed, that the network can produce the target pattern. Reference weights are drawn randomly from the uniform distribution $\mathcal{U}\left(w_{\min}, w_{\max}\right)$. A specific set of weights W^*_{ij} is used for comparisons that is generated differently using

$$W^*_{ij} = \begin{cases} \hat{W} \sin\left(\frac{i\pi}{N_U}\right) & \text{if } 0 \leq i \leq \frac{N_U}{2} \\ 0 & \text{if } \frac{N_U}{2} < i < N_U, \end{cases} \tag{2.25}$$

with amplitude $\hat{W}$ (value given in Appendix A).

The reward R is calculated using the metric $D^{\text{spike}}[q]$ by Victor and Purpura (1996). $D^{\text{spike}}[q]$ measures the distance between spike trains by determining the cost of trans-forming one into the other by adding, moving and deleting spikes. Adding and deleting have unit cost, moving by Δt costs $q\Delta t$. So if a spike has to be moved farther than $2/q$ it is less costly to remove it and add it back at the correct time. These simulations use $1/q = 20\,\text{ms}$. The reward is calculated with a normalized version of the metric individually for every neuron j:

$$R_j = 1 - \frac{D^{\text{spike}}[q]\left(X_{\text{out,j}}, X_{\text{target}}\right)}{N_{\text{out,j}} + N_{\text{target}}}, \tag{2.26}$$

Here, $X_{\text{out},j}$ is the output spike train of neuron j with $N_{\text{out},j}$ spikes and N_{target} is the number of spikes in the target spike train. The reward R communicated to the network is the average of R_j over all neurons. Because $D^{\text{spike}}[q] \leq N_{\text{out,j}} + N_{\text{target}}$, the reward R is restricted to the interval $[0, 1]$.

The approach taken in the analysis here, is to simulate the network with and without hardware constraints and compare the learning performance. Hardware constraints are derived from the AHM and include discretized weights, effect of a readout mechanism, analog drift, mismatch and delayed reward.

2.3.1. Baseline performance

Before hardware constraints are included, this section presents the unconstrained model. The network is simulated for a total of 10 000 trials. This is repeated N_{run} times with different random seeds, so that the background stimulation is different each time. All runs use the reference weights W^*_{ij} and the input pattern S^*_{ik}. Performance is measured using the *initial performance level* R_{before} and the *final performance level* R_{after}, which represent the average of the received reward over the first 100 and the last 1 000 trials, respectively. During the first 100 trials no weight updates occur to stabilize $\langle R \rangle$ and accurately measure R_{before} for the initial weight configuration.

Figure 2.6A shows a raster plot for selected trials during learning. One can see, that while the neurons fired randomly initially, they quickly learn to produce spikes close to the target time points. Figure 2.6B shows reward over time for this simulation. Initially reward increases quickly, until after about 2 000 trials the final level is nearly reached. The trace of the success signal shows, that S is overall distributed symmetrically around 0. According to Frémaux et al. (2010) this is a requirement to eliminate an unsupervised bias, that would prevent or hinder learning. Over all $N_{\text{run}} = 20$ runs, the baseline model improves performance on the task from initially $R^{\text{base}}_{\text{before}} = 0.12 \pm 0.04$ to a final performance of $R^{\text{base}}_{\text{after}} = 0.54 \pm 0.05$.

In Figure 2.6B one can see transient drops in reward. On close inspection, these drops only last for ten or fewer trials, where the performance sinks monotonically to R_{before} and then grows monotonically to the previous level. This is also the case

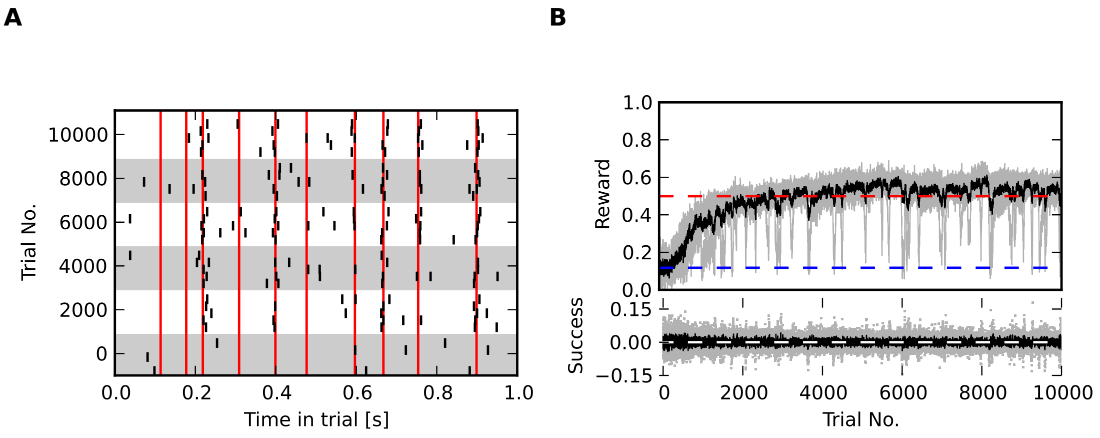


Figure 2.6.: Simulation of the unconstrained baseline model of reward modulated STDP. (A) Raster plot showing the output of all five neurons at selected trials during learning. The red vertical bars indicate the target firing times. Figure taken from Friedmann et al. (2013). (B) Evolution of reward during learning. The blue and red line mark the initial and final performance levels R_{before} and R_{after}, respectively. The lower plot shows the success signal S, which is the difference between reward R and the running average $\langle R \rangle$. The black trace shows only every 50-th point, while all points are plotted in grey.

for smaller learning rates of $\eta = 10$ or $\eta = 5$ (data not shown), but to a lesser extent. The precise reason for this effect is not understood, but since it does not affect final performance, no further investigations are performed.

This result is obtained in the special case of using W^*_{ij} and S^*_{ik}. Because the focus of this study is the impact of hardware constraints on performance, it makes sense to reduce the variability introduced by other sources, such as varying weights and stimulation patterns. It is especially important in light of long simulation times on the order of 10 h, since a high variability necessitates many repetitions of the same simulation to gather enough statistics. However, the reduction to a special case may limit the generality of the obtained results. To verify that this is not the case, simulations with random reference weights and stimulation patterns are carried out. Figure 2.7 shows results: If reference weights $W_{ij} \in \mathcal{U}\left(w_{\min}, w_{\max}\right)$ are used for $N_{\text{run}} = 20$ simulation runs, the final performance is $R^w_{\text{after}} = 0.59 \pm 0.08$. For random stimulation patterns $S_{ik} \in \mathcal{U}\left(0, t_{\text{trial}}\right)$, and using W^*_{ij}, the final performance reaches $R^s_{\text{after}} = 0.53 \pm 0.08$ averaged over 20 runs. If both reference weights and stimulation patterns are chosen randomly, final performance is $R^{sw}_{\text{after}} = 0.54 \pm 0.09$, again averaged over 20 runs.

The learning rate in Equation 1.8 allows for a trade-off between speed of convergence and achievable final reward level. For small η, only small changes are made after every

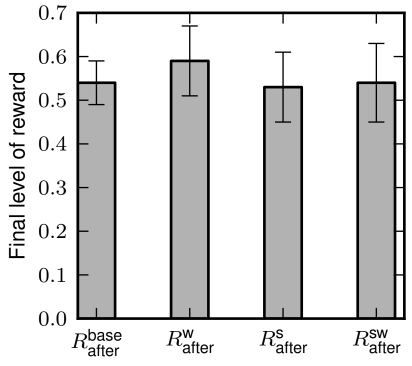


Figure 2.7.: Final level of reward of the baseline simulation $R^{\text{base}}_{\text{after}}$, compared to simulations with random weights ($R^{\text{w}}_{\text{after}}$), random stimulation pattern ($R^{\text{s}}_{\text{after}}$), and both random ($R^{\text{sw}}_{\text{after}}$). The data show, that the selected configuration of weights and stimulation are comparable to the larger class of randomly selected weights and stimulation patterns.

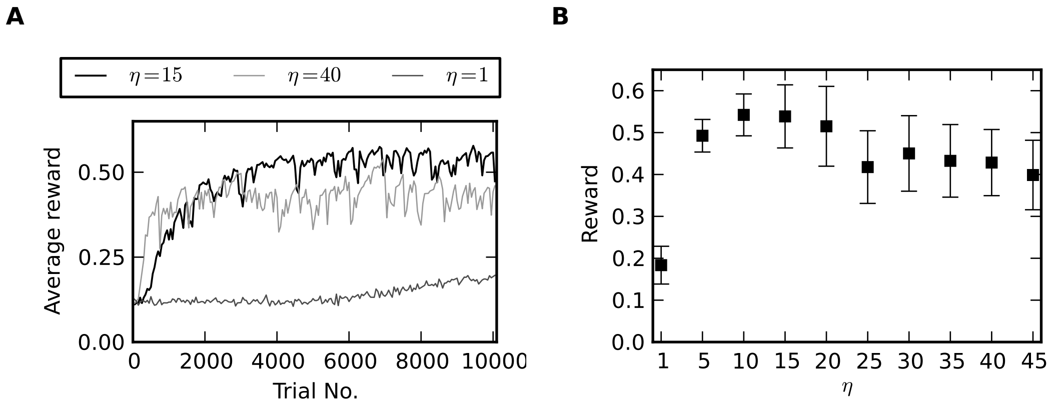


Figure 2.8.: (A) Traces of the running average of the reward $\langle R \rangle$ for different learning rates η. Only every 50-th point of the running average is shown. (B) Final performance levels for different learning rates η.

trial and the performance increases slowly. For large η, the increase is faster, but also fluctuations caused by the random background are amplified, limiting how close the network can come to optimal performance. This behavior is evident in Figures 2.8A and 2.8B. A maximum of performance is reached for $5 < \eta < 15$. The simulations reported here use $\eta = 15$ for fast convergence with good performance.

2.3.2. Mapping reward modulated STDP to the AHM

To include hardware constraints into the simulation the AHM is employed. A central aspect of the R-STDP rule is, that weight updates do not occur immediately, when the spike pair is encountered, but are deferred until reward arrives. The eligibility trace serves as memory of recent activity that, modulated by the success signal S, determines the weight update. The eligibility trace can be directly mapped to the local accumulation trace $a(t)$ in the synapse (see Section 2.2.2.4). In this case, the temporal behavior of $a(t)$, as defined by the drift function $d(t, a)$, has to reproduce the exponential decay of the eligibility trace. Here, a time constant of 500 ms is used. The reward is signalled to the processor in form of the success signal S written to main memory through the external control interface. Whenever S is updated in memory, i.e. at the end of each trial, the plasticity program starts a weight update of all synapses. Initially, only the effect of discretized weights is considered. Therefore, the evaluation function E returns the precise value of $a(t)$, processor speed ν is infinite, and communication has no latency ($D_R = 0$). The plasticity program computes the new weight according to

$$\Delta = F_{\text{theory}}\left(a\left(t_{\text{trial}}\right), S\right) = Sa\left(t_{\text{trial}}\right) \tag{2.27}$$

Table 2.1 defines all options in the framework of the AHM. Later sections add increasingly more hardware effects.

2.3.3. Discretized weights

For the analysis of the effect of discretized weights on learning performance, I tested different resolutions r and two rounding schemes described in Section 2.2.2.3: deterministic and probabilistic rounding. Figure 2.9 shows the results for $N_{\text{run}} = 20$. Already relatively low resolution of 8 or 6 bit is enough to reach performance equal to the baseline simulation with continuous weights (Figures 2.9B and 2.9C). Only the 4 bit data in Figure 2.9D show reduced performance, when using deterministic rounding. In this case, switching to probabilistic rounding (green trace in Figure 2.9D) increases performance nearly to the baseline level.

So why is it possible to improve performance for 4 bit by switching the rounding scheme? Can it also be improved for even smaller resolutions? To answer these questions one has to consider why the performance of lower resolutions is reduced.

Model option	**Implementation**	**Section**
Pairing scheme:	symmetric nearest neighbor after Morrison et al. (2008)	all
Pair weighting:	$s\left(\Delta t_{ij}, a\right) = A_{\pm} e^{-\Delta t_{ij}/\tau_{\pm}}$	all
Drift:	$d(t,a) = -\lambda a(t)$ with $\lambda = \frac{1}{500\,\mathrm{ms}}$	2.3.1-2.3.4, 2.3.6-2.3.7
	$\boldsymbol{d}_n(t,a)$ according to Eq. 2.15 and Section 2.2.2.7	2.3.5
Evaluation:	$E(a(t_e)) = a(t_e)$	2.3.1-2.3.3
	$E_{\Theta}\left(a(t_e)\right)$	2.3.4 - 2.3.5, 2.3.7
	$E_{\Theta_n}\left(a(t_e)\right)$	2.3.6
Evaluation time:	$t_e = t_{\mathrm{trial}}$	2.3.1 - 2.3.6
	$t_e = t_n$	2.3.7
Weight update:	$\Delta = SE(a(t_e))$	2.3.1-2.3.3
	$\Delta = SA(b_0 - b_a)$	2.3.4 - 2.3.7
Discretization:	$\Delta_r = \mathrm{Round}(\Delta)$ (with deterministic or probabilistic rounding) $w' = \begin{cases} w + \Delta_r & \text{for } w_{\min} \leq w + \Delta_r \leq w_{\max} \\ w_{\max} & \text{for } w + \Delta_r > w_{\max} \\ w_{\min} & \text{for } w + \Delta_r < w_{\min} \end{cases}$	2.3.3,2.3.4,2.3.6

Table 2.1.: Definitions of model options in the framework of the AHM for the simulations of reward modulated STDP.

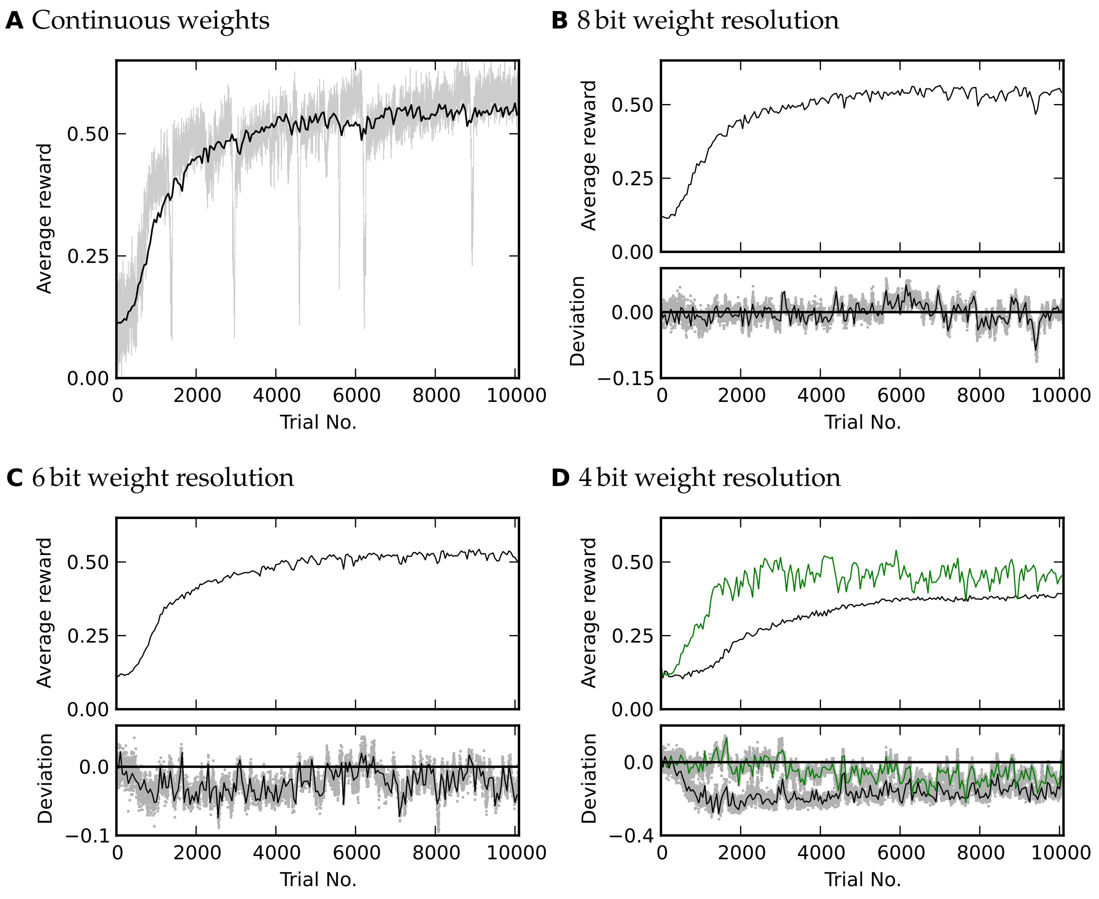
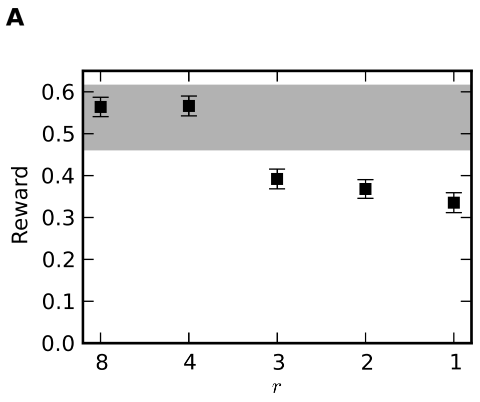


Figure 2.9.: Reward traces for different weight resolutions. Plotted is the running average of the reward $\langle R \rangle$ and, for (B)-(D), the deviation to the trace in (A). The black traces show every 50-th point of the running average. In (A) the light trace shows every point for a single arbitrarily chosen run. The shaded area in the lower plot of (B)-(D) also shows all points of the deviation trace. (D) The green trace shows performance for probabilistic weight updates. These figures were taken from Friedmann et al. (2013).



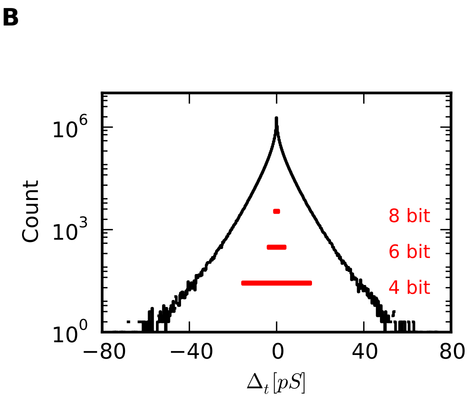


Figure 2.10.: (A) The baseline simulation learns a set of continuous weights, which are then discretized to lower resolutions. The simulation continues for 1000 trials with the discretized weights to measure the performance. The shaded area indicates one standard deviation around the baseline performance level $R^{\text{base}}_{\text{after}}$. (B) Distribution of weight updates Δ before rounding. The red bars mark $\pm\delta_r/2$ for different resolutions. All updates within this range are lost by deterministic rounding.

Two effects play a role here: 1) with lower resolution sinks the ability to represent a well performing set of weights to which the network can converge. The discretization error limits the accuracy. 2) After each trial, synaptic weights can only be changed in quantities of the discretization step δ_r. If the weight update before rounding satisfies Equation 2.3, it is discarded completely by deterministic rounding. Figure 2.10A shows performance of a "good" weight set learned by the baseline model discretized to different resolutions. Resolutions of 8 and 4 bit reach good performance, but beginning at $r = 3$ bit performance decreases. So down to 4 bit the discretization error is small enough to not limit the ability to represent a well performing set of weights. The distribution of weight updates before rounding Δ in Figure 2.10B visualizes how many updates are lost due to rounding as described by Equation 2.3 for different resolutions. One can see, that for 4 bit most of the updates are lost by rounding. So to answer the previous questions: The resolution of 4 bit weights is improved by probabilistic rounding, because this way the population average of weight changes is conserved. For smaller resolutions, performance will be reduced even with probabilistic updates, because the ability to represent a "good" set of weights is reduced by the discretization error as shown in Figure 2.10A.

2.3.4. Threshold readout

The approach of combining analog and digital processing necessitates an interface between both worlds. This interface is given by the evaluation function $E(a)$ in the

AHM. A very simple implementation of the evaluation function is comparing a with a threshold Θ as is done by E_Θ defined in Equation 2.10. The simulations reported in this section tested whether this simple mechanism is enough and how performance is affected by this constraint. The model options used in the previous section are modified in two places: the evaluation function is now $E(a(t_e)) = E_\Theta(a(t_e))$, which produces the two comparison bits b_0, b_1 depending on whether $a(t_e)$ exceeded Θ or $-\Theta$. The weight update then uses these bits to compute

$$\Delta = F_{\text{thresh}}\left(b_0, b_1, S\right) = SA\left(b_0 - b_1\right), \tag{2.28}$$

with the update constant A. The update constant A is necessary, because now there is no more direct access to the local accumulation trace a. So there are two free parameters Θ and A that have to be configured.

To this end, I employed a heuristic method following the idea, that the average update $\langle\Delta\rangle$ by Equation 2.28 should be the same as without the readout (Equation 2.27). The network is simulated in a precursor run over 100 trials and the distribution of $|a(t_{\text{trial}})|$ for all synapses and trials is recorded. The mean of this distribution is used as threshold $\Theta^* = \langle|a\left(t_{\text{trial}}\right)|\rangle$. This way, the average final accumulation value is close to the threshold. This avoids two extremes: a high threshold would only seldom be crossed for large $a\left(t_{\text{trial}}\right)$, while a low threshold would be exceeded very often. In the former case, few large updates would have to be made, and in the latter many small ones. Large updates have a stronger effect on performance and lead to a less smooth learning curve, which is considered harmful for the learning rule. Small updates are difficult to represent with low resolution weights. The selected threshold Θ^* represents a compromise between the extremes. The threshold readout causes an update of size $\pm A$ through the readout bits b_0 and b_1 in $N_p(\Theta)$ out of all N trials. By setting

$$A^* = \frac{N}{N_p(\Theta)}\langle|a\left(t_{\text{trial}}\right)|\rangle \tag{2.29}$$

the average absolute update with threshold is equal to the average update that would have been performed in the precursor run. The term $N/N_p(\Theta)$ accounts for the fact, that if the threshold is not exceeded, the weight is not changed. Its value can be estimated from the distribution of a recorded in the precursor run. Numerical values for Θ^* and A^* determined in the presented simulation are given in Table 2.2.

Figure 2.11 shows performance for different combinations of the parameters Θ and A. This simulation uses continuous weights. The heuristically determined values Θ^* and A^* reach the best performance in the tested parameter space. If the threshold is increased, the update constant must also be increased to limit the reduction in performance. This is intuitively clear from Equation 2.29: increasing Θ leads to smaller $N_p(\Theta)$ and therefore higher A. With constant Θ increasing A has the same effect as changing the learning rate η.

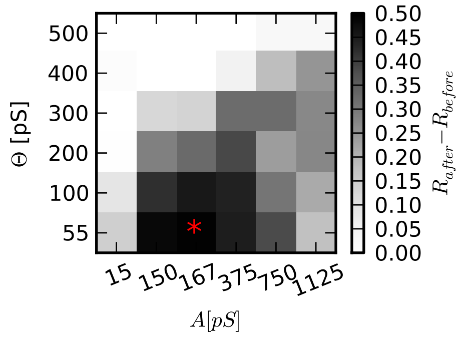


Figure 2.11.: Improvement in reward $R_{\text{after}} - R_{\text{before}}$ for different combinations of readout threshold Θ and update constant A. The red star marks the parameters Θ^* and A^* as determined heuristically.

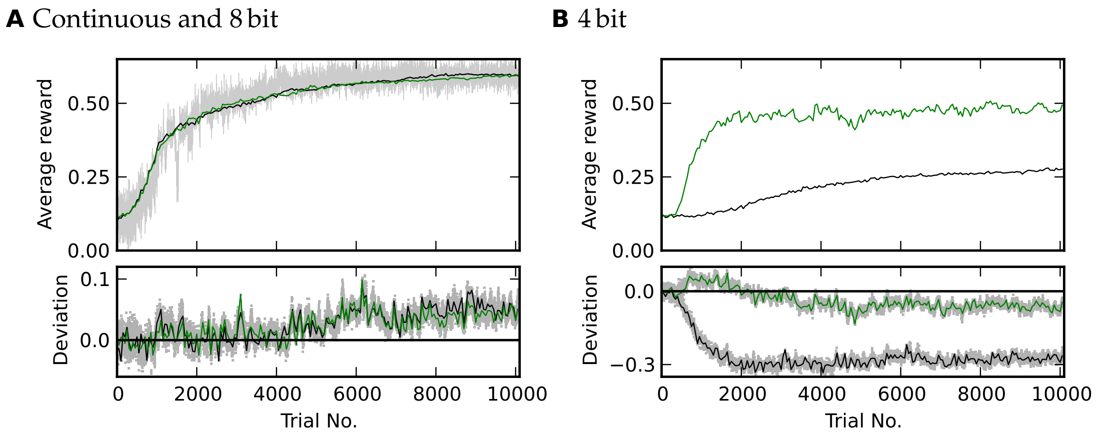


Figure 2.12.: Reward traces with threshold readout. The lower plots show the deviation to the baseline simulation shown in Figure 2.9A. (A) Continuous weights in black and 8 bit weights in green. (B) 4 bit resolution with deterministic (black) and probabilistic rounding (green). These figures were taken from Friedmann et al. (2013).

Parameter		Value
Readout threshold	Θ^*	54.77 pS
Update constant	A^*	167.2 pS
Measured quantities		**Value**
Average eligibility value	$\langle a \rangle$	(5 ± 2) pS
Average absolute eligibility value	$\langle \lvert a \rvert \rangle$	(142 ± 2) pS
Maximum eligibility value	$\max \{a(t)\}$	(1.01 ± 0.06) nS
Maximum success signal	$S_{\max}$	0.121 ± 0.008

Table 2.2.: Parameters and measurements for the simulation shown in Figure 2.12.

Figure 2.12 shows learning performance, when the threshold readout is used with the parameters Θ^* and A^*. Continuous and 8 bit weights reach equal performance (Figure 2.12A). Both rise slightly above the baseline level and show generally a less noisy trace. The threshold readout suppresses extreme weight changes, that would have a strong effect on performance. This way, variability is reduced and better performance can be achieved. For 4 bit resolution, as expected, deterministic rounding leads to poor performance, while probabilistic rounding comes close to baseline (Figure 2.12B). Table 2.2 lists a number of characteristics measured in the simulations shown here. In conclusion, the simple threshold readout E_Θ is enough to achieve performance as good as the unconstrained model.

2.3.5. Robustness to variations in the drift model

In the previous sections the drift model conveniently reproduced the by theory desired behavior of exponential decay with a time constant $\tau_e = 0.5\,\mathrm{s}$. However, this requires a controlled decay mechanism in every synapse, which might be too expensive for a practical implementation. Without such a mechanism, drift is caused by leakage currents, which are subject to device mismatch as detailed in Section 2.2.2.7. The simulations in this section tested how sensitive the learning rule is to this disturbance and whether a controlled decay mechanism is necessary. Because this type of noise is static over time and creates a fixed pattern over the synapses, the hope is, that a learning rule can adapt itself to these variations without further calibration. As a realistic model, the dual capacitor vectorized drift model is employed (Equations 2.15-2.17) with ohmic drift (Equations 2.15). Mismatch is included according to Equations 2.19-2.20, with mean time constants $\tau_+^{(e)} = \tau_-^{(e)} = \tau_e$ and variances $\sigma_{\tau_+^{(e)}} = \sigma_{\tau_-^{(e)}} = \sigma_{\tau_e}$. Also, continuous weights are used here.

Figure 2.13A shows how this drift affects the final distribution of the accumulation trace $a(t_{\mathrm{trial}})$. The lower right histogram shows the previously simulated case of decay

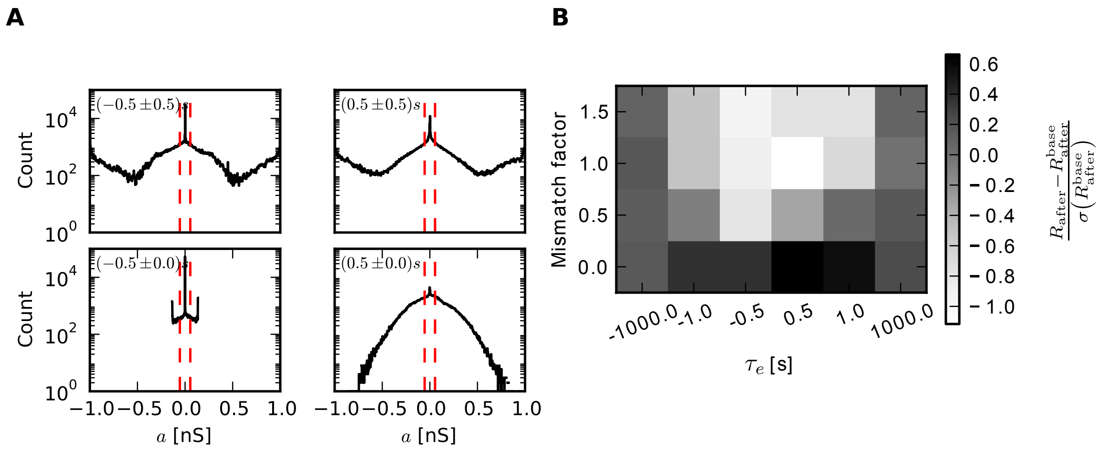


Figure 2.13.: (A) Distribution of final accumulation trace values $a(t_{\mathrm{trial}})$ for mean time constants $\tau_e = \pm 0.5\,\mathrm{s}$ and variance $\sigma_{\tau_e} = 0$ and $0.5\,\mathrm{s}$. The vertical red bars indicate the threshold $\pm\Theta$. (B) Relative performance compared to baseline $R_{\mathrm{after}} - R^{\mathrm{base}}_{\mathrm{after}}$ in units of the standard deviation of the baseline performance coded as grey value.

with a time constant of 500 ms and no mismatch. To the left, a negative time constant without mismatch causes a narrow distribution. From the beginning of each trial both capacitors a_+ and a_- drift towards a_{max}. Since the effective value seen by the evaluation function is their difference $a_+ - a_-$, the final state with both capacitors at maximum represents $a = 0$. Since both capacitors can only be charged, i.e. a_+ and a_- increased, the maximum width of the distribution is limited. The largest possible value for $a\,(t_{\mathrm{trial}})$ results for initial conditions $a_+(0) = a_{\mathrm{max}}$ and $a_-(0) = 0$. Then,

$$\begin{aligned}
a\,(t_{\mathrm{trial}}) &= a_+\,(t_{\mathrm{trial}}) - a_-\,(t_{\mathrm{trial}}) && (2.30)\\
&= a_{\mathrm{max}} - a_{\mathrm{max}}\left(1 - \exp\left(\frac{t_{\mathrm{trial}}}{\tau_-}\right)\right) && (2.31)\\
&= a_{\mathrm{max}} \exp\left(\frac{t_{\mathrm{trial}}}{\tau_-}\right). && (2.32)
\end{aligned}$$

Similarly, the smallest possible value is achieved for $a_+(0) = 0$ and $a_-(0) = a_{\mathrm{max}}$, so that the width is given by approximately $\pm 0.135 \cdot a_{\mathrm{max}}$.

When mismatch is included (upper histograms in Figure 2.13A), time constants can be negative as well as positive, even within one synapse. Therefore, there are three possible final states towards which $a(t)$ can drift: Towards 0 if both capacitors decay or grow. Towards a_{max} if a_+ grows and a_- decays and $-a_{\mathrm{max}}$ if the opposite is the case. This leads to three maxima at the respective values in the histogram. Figure 2.13B shows how these variations in the drifting behavior affect performance of the learning rule as compared to the baseline simulation. Foremost to note is, that all combinations of time constant and variance reach performance within one standard deviation of the baseline performance $R_{\text{after}}^{\text{base}}$. As observed in Figure 2.12A performance for $\tau_e = 0.5\,\text{s}$ and no mismatch is slightly better than baseline. It achieves the best performance in the tested parameter set. Very long time constants $\tau_e = \pm 1000\,\text{s}$ compared to the trial duration $t_{\text{trial}} = 1\,\text{s}$ do not cause significant changes to $a(t)$ between spike pairs. Without or with small mismatch, this setting simulates the case without drift, where good performance is also reached. The only regime, where minor degradation is observable is for small time constants with high mismatch. Here the probability is high, that within one synapse time constants have opposite sign, causing drift to $\pm a_{\text{max}}$. In this case, the synapse might exceed the threshold during evaluation, even if there were no spike pairs at all. For even smaller time constants than 0.5 s performance is expected to reduce further. In this case activity at the beginning of the trial has a reduced effect on the weight update. The eligibility trace has lost memory of the activity at the beginning of the trial, when reward is given. It becomes therefore increasingly difficult to correct the output activity at the beginning of the trial to match the target spike train.

So in conclusion, the hope that the learning rule would adapt to fixed-pattern variations has proven true. Performance is only weakly sensitive to rather drastic deviations from the theoretical decaying behavior. In the tested task, a controlled decay mechanism is not required.

2.3.6. Mismatch on the evaluation function

As described in Section 2.2.2.7, the evaluation unit is also subject to device mismatch. This is modeled by varying the threshold Θ for each synapse. Simulations now use again a drift model with time constant $\tau_e = 0.5\,\text{s}$ and continuous weights. Figure 2.14 shows performance in dependence of the variance of the threshold Θ. The maximal variance is of the same size as the threshold itself. The results indicate, that the learning rule is insensitive to variation on the readout threshold. This is probably the case, because the mean threshold and therefore the mean weight update is not altered.

2.3.7. Delayed reward

Sections 2.2.2.9 and 2.2.2.10 described how limited performance of the processor and communication latency can delay the point of time at which the reward is available for the weight update. During this additional delay the accumulation trace $a(t)$ continues to evolve according to the drift function $d(t)$, causing a deviation from the theoretical learning model. To investigate the impact on the performance, the network is simulated using a non-varying threshold Θ and the analog drift model with time constant $\tau_e = 0.5\,\text{s}$. Figures 2.15A and 2.15B show performance with increasing communication latency D_R and finite processor update frequency ν. Especially in

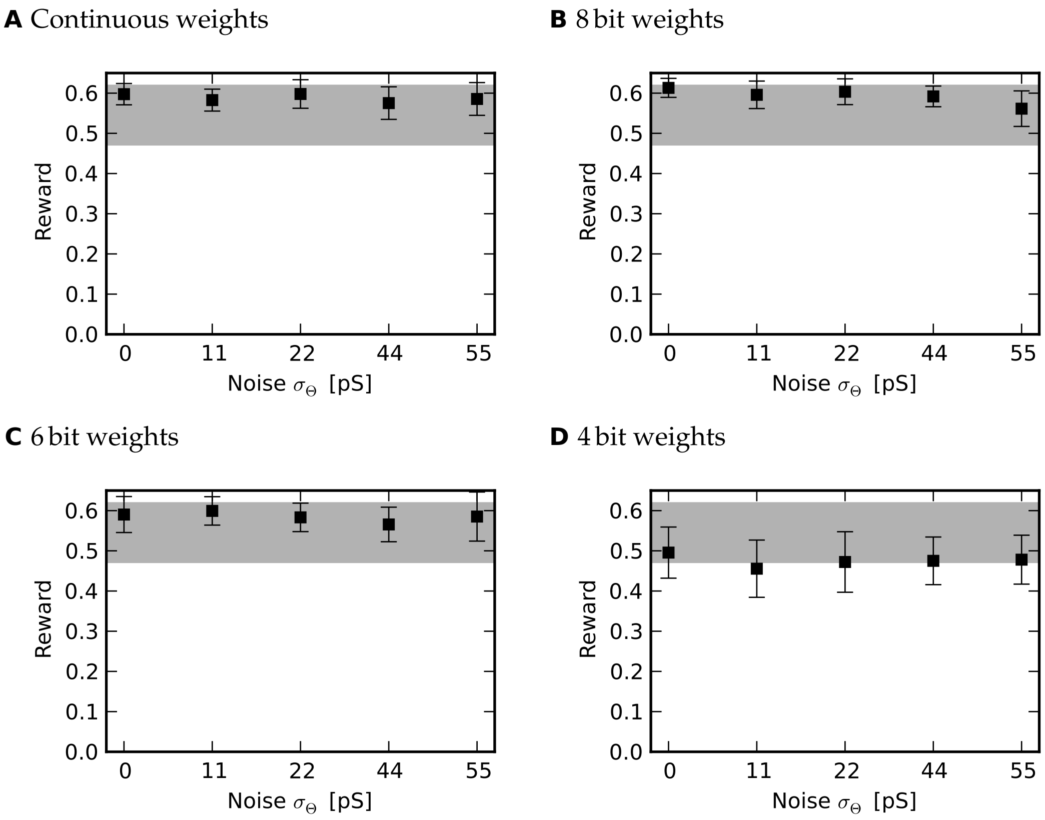


Figure 2.14.: Final performance R_{after} with mismatch on the threshold evaluation function E_Θ in dependence of the variance σ_Θ. The shaded area represents one standard deviation around $R^{\text{base}}_{\text{after}}$. (D) Updates with 4 bit weights were performed with probabilistic rounding.

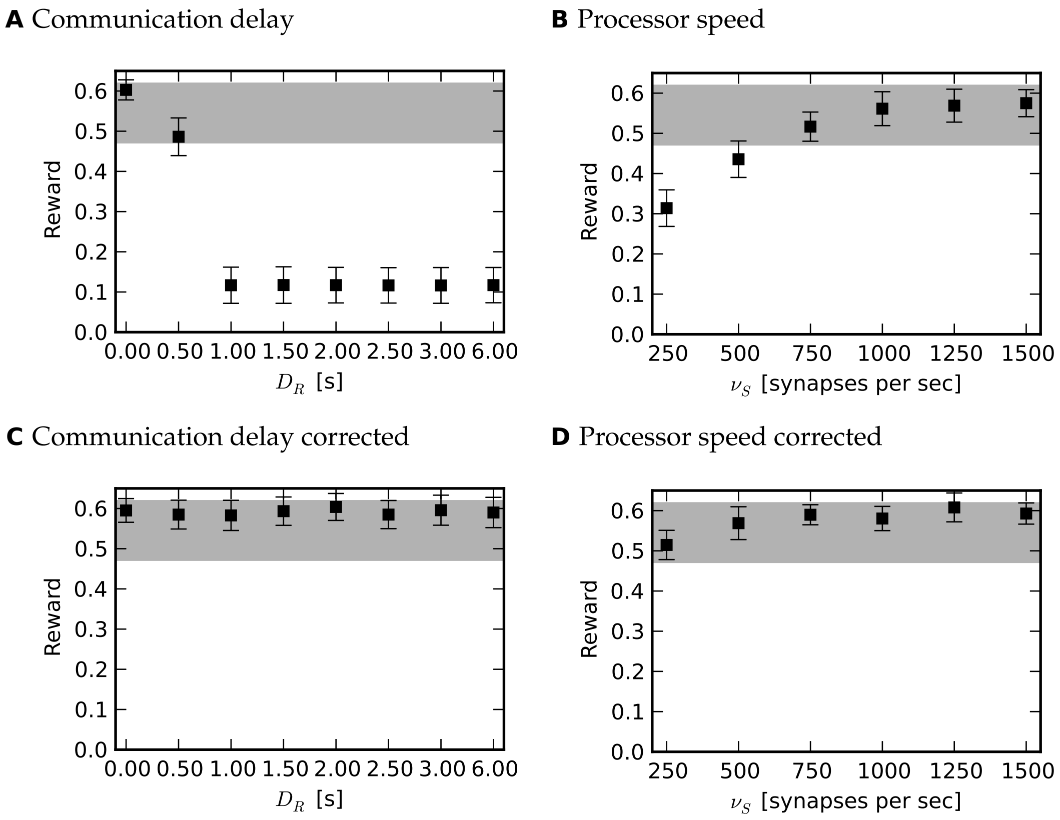


Figure 2.15.: Effect of delayed reward due to limited processor speed and communication latency with and without correction. The correction compensates the additional drift of $a(t)$ after the trial has ended and until the reward arrives. It requires knowledge of the delay and the update speed.

the face of increasing D_R learning fails for $D_R \geq 1\,\mathrm{s}$ (Figure 2.15A). In comparison, a limited updating frequency causes a more gradual failure (Figure 2.15B). For both cases Equation 2.23 determines the time of evaluation t_n for the n-th synapse. The communication latency is measured from the end of the trial at time t_{trial} until the reward is available at the processor. Communication latency and processor speed turn the accumulation trace from $a(t_{\text{trial}})$ into $a\left(t_n\right)$. In the used drift model of exponential decay this is expressed with $a_n\left(t_n\right) = \beta_n a_n\left(t_{\text{trial}}\right)$ using the factor

$$\beta_n = \exp\left(-\frac{D_R + \frac{n}{\nu_S}}{\tau_e}\right). \tag{2.33}$$

Large enough β_n will reduce the accumulation trace below the evaluation threshold Θ and thereby prevent an update that would have occurred without delay. This happens for all synapses if

$$\begin{aligned} a_{\max}\beta_n &< \Theta & (2.34) \\ \tau_e \ln\left(\frac{a_{\max}}{\Theta}\right) &< D_R + \frac{n}{\nu_S} & (2.35) \end{aligned}$$

with the maximal accumulation value $a_{\max}$. With infinite processor speed this is satisfied for $D_R > 1.497\,\mathrm{s}$ and all synapses in the presented simulations (using $a_{\max} = \max\{a(t)\}$ from Table 2.2). This is in agreement with observations in Figure 2.15A. If the delay is neglected ($D_R = 0$) and Equation 2.35 considered for $n = \frac{N_U N_T}{2}$, i.e. for half the synapses no update is possible, then $\nu_S < 417.5\,\mathrm{s}^{-1}$ limits the updating frequency. In Figure 2.15B degradation is already evident for higher frequencies. If $\nu_S < 750\,\mathrm{s}^{-1}$ is assumed as limit, $n = 475 \approx \frac{1}{3}N_U N_T$. So loss of up to about one third of all synapses due to delay can be tolerated.

If the communication latency and updating frequency are known, one can correct for delay by lowering the threshold by a factor of β_n with $n = \frac{N_U N_T}{2}$. By leaving the update constant A unchanged, weights effectively change as if the delay were not present. The result is shown in Figures 2.15C and 2.15D. If only communication latency is considered, no degradation in performance is observable over the tested range of delays (Figure 2.15C). If only updating frequency is tested, performance for $\nu_S = 250\,\mathrm{s}^{-1}$ is slightly degraded.

So far, the discussion ignored an important aspect of analog hardware: noise. The correction necessitates smaller and smaller threshold values for increasing delay. But if the threshold is reduced too far, noise can cause an exceedance even if the mean accumulation trace is zero. For Gaussian distributed noise around the accumulation trace $a(t)$ with variance σ_a, the signal-to-noise ratio $z = a/\sigma_a$ is described by

$$z(t) = \frac{a\left(t_{\text{trial}}\right)}{\sigma_a} \exp\left(-\frac{t - t_{\text{trial}}}{\tau_e}\right). \tag{2.36}$$

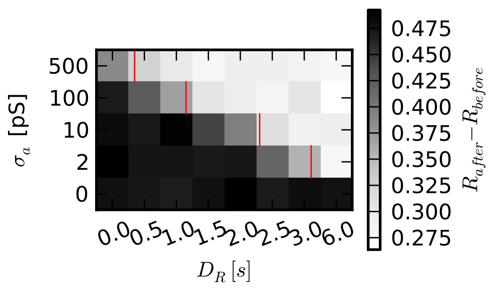


Figure 2.16.: Improvement in reward $R_{\text{after}} - R_{\text{before}}$ for different combinations of reward delay D_R and readout noise σ_a on the accumulation trace $a(t)$. The red bars mark the prediction for maximally tolerable delay based on the signal-to-noise ratio of $a(t)$ (Equation 2.37 with $z^* = 1$).

If a signal-to-noise ratio of z^* is required for learning, delay satisfying

$$D_R + \frac{n}{\nu_S} > \tau_e \ln\left(\frac{a_{\max}}{z^* \sigma_a}\right) \tag{2.37}$$

will fail to learn. This relation is tested in Figure 2.16 for noise with a variance of up to $\sigma_a = \frac{a_{\max}}{2}$ and assuming $z^* = 1$. The observations are well in agreement with the predicted limits. In comparison to Figures 2.15C and 2.15D even moderate noise levels of $0.01 a_{\max}$ limit delay to 2 s. So while the correction of the threshold Θ is necessary to allow for learning in the presence of delays, analog noise poses a hard constraint on communication latency and processor speed.

2.3.8. Guidelines for hardware implementation

Section 2.3 tested a reward modulated STDP rule in a simple spike train learning task under hardware constraints. The AHM was used as a generalized model of a neuromorphic plasticity implementation to derive the constraints. This subsection condenses the results into a set of guidelines for hardware implementation.

A first important result comes from the analysis of the impact of weight resolution on learning performance in Section 2.3.3. Already a small resolution of only 6 bit is enough to reach a performance comparable to continuous weights. For lower resolutions, as tested for 4 bit, probabilistic rounding can increase performance nearly to the baseline level. However, further reduction of the resolution (Figure 2.10A) will at some point degrade performance independent of rounding, because the discretization error does not allow to represent a well performing final set of weights. At what point this happens depends in general on the network topology, stimulation statistics, and the learning task. For implementation this means, that small weight resolutions below 8 bit are viable. A mechanism for probabilistic rounding is a good option to boost performance of low resolution weights.

A second result is, that for the interface between processor and synapses a simple threshold comparison is sufficient (Figure 2.12). Using a high resolution ADC instead would not gain any improvement. This greatly reduces the cost for the readout and evaluation circuitry in hardware.

The analysis of sensitivity to variations in the drifting behavior of the analog accumulation trace in Section 2.3.5 has shown, that a controlled mechanism for decay is not required for this task. Of course, the time constant has to be large enough compared to the trial duration. To accommodate other tasks with possibly longer trials, the time constant should be at least on the order of seconds if it can not be made configurable. Long time constants did not negatively affect performance (Figure 2.13B), but short ones do. An open question remains if non-episodic learning tasks, i.e. tasks that are not naturally separated into finite length episodes or trials, are more sensitive to the time constant. For example a continuously learning agent with an eligibility trace that does not decay at all might reinforce actions unrelated or even counter-productive to the reward. So to stay on the safe side here with a hardware system, an at least minimally configurable drifting behavior is desirable.

Simulations with fixed-pattern noise on the evaluation threshold Θ showed no impact on performance up to high variances (Figure 2.14). As long as the mean threshold does not deviate too far from the desired value Θ^*, the learning rule can adapt to the variability. This showcases an intriguing feature of neuromorphic hardware with plasticity: the ability to self-calibrate to hardware variations.

Communication latency and processor updating frequency need to be carefully optimized depending on two variables: the acceleration factor α and the time constant of the eligibility trace τ_e. In a real-time system with $\alpha = 1$, timing is of no concern, since communication latencies on the order of seconds are much larger than typical technical latencies on the order of micro- or milliseconds. Also a processor clocked in the MHz-range can compute at least thousands of weight updates per second for typical STDP functions. But in highly accelerated systems with $10^3 \leq \alpha \leq 10^5$ this is not the case any longer. Now it is primarily important how long information is retained in the eligibility trace as determined by its time constant. For long time constants compared to the delay, timing is again of no concern, at least for episodic tasks. For a continuously learning agent it may be problematic if rewards arrive with a huge delay. But on the other hand this is an intrinsic aspect of reinforcement learning also known as the distal reward or temporal credit assignment problem (Sutton and Barto, 1998; Izhikevich, 2007b): Reward arises a certain time after the rewarding action and the purpose of the eligibility trace is to allow the association of earlier actions with the reward. One insight from the analysis in Section 2.3.7 is that analog noise on the accumulation trace in the end limits how much delay is tolerable. In general, any real system will have a smallest reliably representable value for the accumulation trace $a(t)$ regardless of implementation. For delay large enough to cause $a(t)$ to drop to this level, learning will fail. Equation 2.37 can be used by hardware designers to relate communication latency, processor speed, time constant, and noise and find working combinations.

3. Hardware design

This chapter presents the hardware designed to implement a neuromorphic plasticity system following the concept of the AHM. Developed technologies are described and design considerations discussed. In Chapter 5 these technologies are combined for concrete hardware systems, which are evaluated in experiments. Throughout this chapter Figure 2.2 serves as a map to locate the described technologies in the overall context. The background for the developed designs is formed by the neuromorphic hardware systems developed in the EU-projects FACETS and BrainScaleS (FACETS, 2010; BrainScaleS, 2012). These systems aim for the design goals outlined in Section 1.3 using wafer-scale integration (Mead, 1990; Schemmel et al., 2008, 2010) as central method: The complete wafer, on which several individual, but identical Application Specific Integrated Circuits (ASICs) are produced, is left intact and the single ASICs are interconnected in a post-processing step. This way, large-scale systems can be build with high interconnection density. Components presented in this chapter are always designed with wafer-scale integration in mind.

3.1. Plasticity processor technology

The Plasticity Processor (PP) is an integral component of the overall strategy of combining analog and software based computing. Its location in the model is visualized in Figure 3.1. It is tasked by the AHM with coordinating the weight update process and computing new weights. So it has to access the analog synapse array through the array interface to select synapses for evaluation, read their weight, compute the update, and write the new weight back. To maintain maximal flexibility, the PP is designed as a general purpose microcontroller implementing a subset of the PowerISA specification (PowerISA, 2010).

3.1.1. Design principles

First I want to outline the philosophy followed in designing this processor and the design goals aimed for. For VLSI hardware design, it is always important to minimize cost in chip area. In the present case this is aggravated by the fact, that the design is intended to be integrated into the standard cell part of an existing system with a tight area budget. Especially compared with current technologies, the used 180 nm process is not very area efficient. This forces a minimalistic approach to the design.

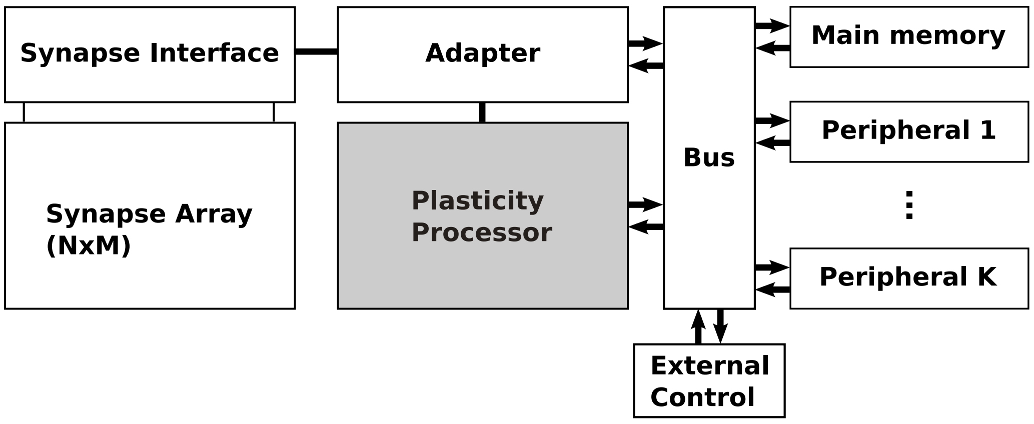


Figure 3.1.: The picture locates the plasticity processor in the framework of the AHM.

However, in the long run it is well possible, that a more modern technology, e.g. 65 nm, is used, which would increase area efficiency of standard cell digital logic. For this case it is desirable that the existing design can profit from the improvement without having to be redesigned. This can be achieved by configurable and modular Register Transfer Level (RTL) code allowing to scale the PP by simply changing a configuration option. A third goal is to keep the complexity of the design as low as reasonably possible. This is generally a good idea facilitating development and keeping the design adaptable to new demands. For a general purpose processor it is also quite a challenge to sufficiently verify correct functioning of the design in the light of arbitrary programs. Low complexity helps in reducing the amount of states that need to be tested. More or less in contrast to the low area and low complexity goals should the PP be suitable for fast processing of synaptic weights. Chapter 2 showed, that a low updating frequency by the processor negatively impacts performance. The higher the frequency, the more weights can be processed by one processor, increasing overall efficiency.

3.1.2. Instruction set architecture

The operation of a processor is controlled by a sequence of instructions in main memory known as program. What instructions are available, what arguments they take, and how they are coded is defined in the Instruction Set Architecture (ISA). In essence, the ISA forms the interface between the worlds of software and hardware. A program written in a high level programming language is translated into sequences of instructions by the compiler following the ISA. The hardware then executes the instructions following the so called sequential execution model. In this model, every instruction is executed completely in the order it appears in memory before the next is started. To increase performance real implementations typically deviate from this idealized model and overlap the execution of instructions. This is possible as long as the result does not deviate from the sequential execution model.

Power ISA In particular, very different hardware implementations can execute the same software and use the same tools, if they implement the same ISA. To not be forced to develop or adapt a compiler for the PP, I decided to use the ISA defined by PowerISA (2010). It is a widely used Reduced Instruction Set Computer (RISC) specification, for which for example the GNU Compiler Collection (GCC) can compile code (Stallman, 2012). The specification is organized in categories, of which many are optional and provide specialized functionality. It specifically differentiates server and embedded processors and supports 64 bit and 32 bit designs. Therefore, the specification can be used similar to a construction kit to build an ISA appropriate for the plasticity processor.

Implemented subset In terms of categories, the design presented in this section supports *Base, Embedded, External Control* and *Wait* in 32 bit mode, although some liberties were taken to better suite the constraints of the plasticity application. Instructions for cache handling, load-reserve/store-conditional, and synchronization (`sync`, `isync`) are not supported. A full list of implemented instructions, registers, and exceptions is given in Appendix B. Unimplemented instructions are ignored instead of raising an exception. As a further deviation, the interrupt vector addresses are custom as described in Section 3.1.9. There it is also discussed, under what conditions interrupts are precise, i.e. whether taking an interrupt can cause a deviation from the sequential execution model. The selection of implemented instructions allows for the execution of code generated by a compiler. The deviations are only relevant for system software, that therefore can not be reused from other platforms.

3.1.3. Microarchitecture

While the Instruction Set Architecture (ISA) defines coding and semantics of instructions that can be used on a processor, the mircorarchitecture describes the detailed organization and how these instructions are executed. The task of a micro-processor is to execute instructions from a program in a way that is indistinguishable to the sequential execution model for the user. However, to improve performance, it is essential to leverage parallelism in hardware. The most common way of doing this is pipelining: the execution of an instruction is divided into a series of smaller steps that each take a single clock cycle to complete. As soon as one instruction completes the first step, the second one starts, so that effectively multiple instructions execute in parallel each at a different stage of completion. This is only possible as long as there are no interdependencies between subsequent instructions. If, for example, the second instruction uses a register as input that is written by the first one, it has to be stalled until the result has been written to the register. Such a so called hazard therefore causes "bubbles" in the pipeline that reduce performance. Section 3.1.6 describes how hazards are detected and solved in the presented design.

To reduce the number of wasted cycles, out-of-order designs can dynamically change the order in which instructions are started. In case of a stall, the waiting instruction can "step aside" to allow later ones to execute, if they have no outstanding dependencies. The process of beginning the execution of an instruction after it has been fetched from memory and analyzed for hazards is called *issuing the instruction*. Completion of an instruction by writing its result - if any - back to registers is referred to as *retiring the instruction*. Out-of-order designs have to track instructions between issue and retirement to ensure equivalency to the sequential execution model. If this is not the case, reordering will introduce additional hazards. For example, the result from a preponed instruction might be overwritten by a postponed one if both write to the same register destination, which would represent a write-after-write hazard. Out-of-order designs can be classified depending on whether only issue or retirement happen out-of-order or both.

A further performance improvement tries to maximize the utilization of functional units of the processor by issuing multiple instructions simultaneously. In such a case the implementation is called super-scalar. Super-scalar out-of-order processors can have many instructions in various stages of completion executing simultaneously. The gain in performance is then only limited by Instruction Level Parallelism (ILP), i.e. the degree to which interdependencies allow parallel execution. Analyzing the program for these interdependencies and keeping track of in-flight instructions adds cost in terms of area and power. It also increases complexity and creates higher bandwidth demands for memories and register files, since, for example, several instructions need to be fetched from memory in one clock cycle to sustain super-scalar execution.

The PP attempts a trade-off between performance on one side and power/area costs and complexity on the other. It is therefore a scalar in-order issue, out-of-order retire design. This means, that one instruction per cycle can be issued in program order, but, if latencies are different, they can retire out-of-order. This form of retirement takes advantage of situations where one operation, e.g. access to memory, takes long to complete. The program can be continued up until the result from the slow operation is required. This choice is also in line with the results reported by Gonzalez and Horowitz (1996) that show pipelining to improve the energy-delay product, while performance gains by super-scalarity are offset by the increased power demand. Figure 3.2 illustrates the pipeline structure. The design is separated into front- and back end. The front end fetches instructions, detects hazards and schedules instruction issue and retirement. The back end performs the function associated with an instruction. The first stage in the pipeline is the address generation for instruction memory. It contains the Program Counter (PC) that points to the location of the current instruction and is typically incremented in every cycle. Branch instructions and interrupts can initiate a control

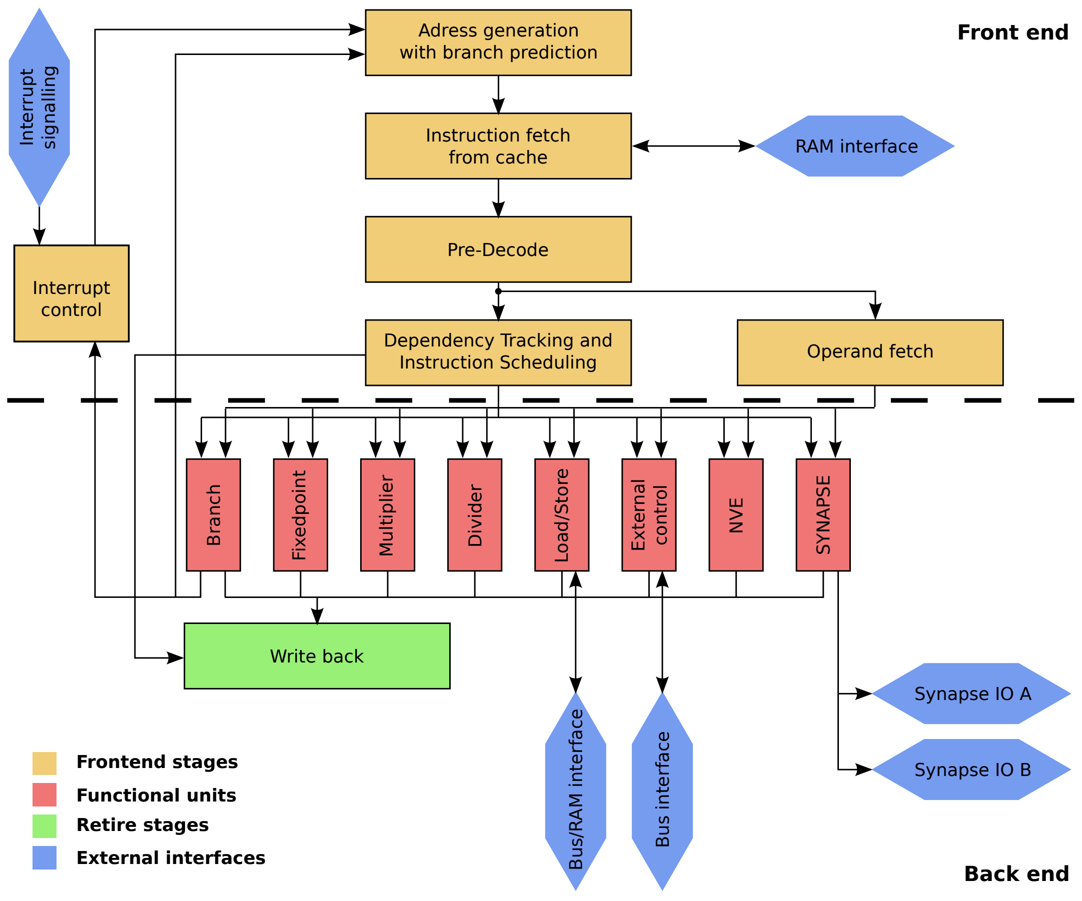


Figure 3.2.: High level view of the processor pipeline showing the separation into front and back end. See text for a description of the elements.

transfer that loads a new address into the PC from which to continue execution. This stage optionally also includes a mechanism to predict branches and speculatively do a control transfer before the branch is decided in the back end. The next stage fetches one instruction per cycle from storage. Then, it is pre-decoded to determine information necessary for hazard detection and scheduling, e.g. to which functional unit the instruction belongs and which registers it reads and writes. In the next cycle, dependency tracking will decide whether stalling is required to solve a hazard. Simultaneously, operands from the register file are fetched. Only if no dependency on an outstanding write exists is a correct result available after the fetch. The dependency tracking stage keeps information about all in-flight instructions, which are also used to schedule writing back of results to registers. A key concept that simplifies this task is that execution in the functional units takes a known, fixed number of clock cycles, after which the result is available. This allows for allocating a write back slot upon issue. Issuing an instruction is then a fire-and-forget operation in a sense, that the instruction is allowed to complete without any further delays. On the other hand, this imposes a strong constraint for the implementation of operations. For example, an I/O operation might take a not a priori known time to execute, depending on the latency of the peripheral address by the access. To relax this restriction, an alternative delayed write back method is used by such operations: The functional unit has to wait for a free write slot and then indicate completion to the dependency tracking logic. As an optimization, a write-back slot is reserved based on an expected latency. If an instruction using delayed write back returns a result in time for this reserved write back slot it can immediately finish. Otherwise it has to wait for a free slot.

The back end is organized in a number of parallel functional units, that take two or more cycles to execute an operation. Those are:

1. *Branch:* Decides the outcome and target address of branch instructions. Some interrupt related instructions are also handled here. The output of the branch unit controls the address generation unit to load a new program counter.

2. *Fixed-point:* This includes most arithmetic and logical instructions excluding multiply and divide. The rotate and shift instructions are executed here, as well as special purpose register manipulation, e.g. condition register logical and move operations.

3. *Multiplier:* A 32 bit pipelined multiplier for signed and unsigned fixed-point multiplication.

4. *Divider:* A 32 bit sequential divider for signed and unsigned fixed-point division. Because division is typically an infrequent operation, the divider is built in a way that only one division can be executed at a time with typically several ten cycles latency.

5. *Load/Store:* As is typical for RISC designs, all operations work on internal registers and only dedicated load and store instructions access main memory. They are implemented in this functional unit. Two alternatives are available: a simple version assumes direct connection to an Static Random Access Memory (SRAM) memory that provides results without delay. A more complex variant translates load and store operations into requests on a bus allowing for arbitrary delay of the response. The used bus is the Plasticity Processor Bus (PPB) described later in Section 3.2. This way also memory mapped I/O operations can be performed using the load/store unit.

6. *External control:* This unit implements the external control category of the Power ISA that provides additional input and output operations similar to load and store. It implements a simplified version of the load/store unit and provides an additional PPB interface for I/O.

7. *NVE:* A special function unit to perform look-up table based vectorized 4 bit arithmetic in the 32 bit General Purpose Register (GPR). This is an optimization for the plasticity application.

8. *SYNAPSE:* A more powerful replacement for the NEVER unit using dedicated 128 bit vector registers. The unit also has two specialized I/O interfaces for efficient reading and writing of synapses.

All functional units share standardized input and output interfaces. The operand bus supplies up to three 32 bit arguments from operand fetch. For example, some store instructions (e.g. `stwx`) need all three operands simultaneously: one for the data to be written to memory, and two as memory address and offset to select the destination in memory. The result bus reads one or two 32 bit results that are written back to registers in the write-back module. Only one word is written to general purpose registers per cycle, but for example a branch instruction can simultaneously change the `CTR` and `LNK` register.

External Interfaces The processor core block as shown in Figure 3.2 has four types of interfaces to the external world. The RAM interface is a simple synchronous interface allowing for read and write operations inside an address space. The signals are specified in Table C.1.1 in Appendix C. Writes can be masked byte-wise. The delay signal indicates that requested read data is not yet available. The PP acts as master on one RAM interface to instruction storage and, if the simple load/store variant is selected, a second one to data storage.

The interface of the PPB is based on the Open Core Protocol (OCP) specification (OCP, 2009). Only a small subset is implemented and compatibility with other OCP clients might be limited. The bus itself is described in Section 3.2. In short, the master initiates requests using a handshake method. After the request is accepted the response can arrive after an arbitrary number of cycles. They only have to follow a first-in-first-out principle, i.e. responses have to arrive at the master in exactly the same order as the requests were sent out. This bus is used by the more complex load/store unit and the external control unit, if present.

The interrupt signaling interface provides a number of signals to indicate exceptions to the processor core. Among them are for example timer events or wakeup requests, if the processor is in a power saving sleep state.

The synapse I/O interface is described in Section 3.4.

3.1.4. Instruction fetching and control transfers

The first stages in the pipeline (Figure 3.2) deal with address generation and fetching from storage to generate an instruction stream. Most of the time, it is sufficient to increment the PC in every clock cycle. However, branches and interrupts can initiate control transfers, so that after the branch or the interrupt triggering instruction, execution is resumed not at the next, but an arbitrary other address. In a pipelined processor this requires to discard partially executed instructions that were started directly after the branch, but before its outcome was decided. This introduces a branch penalty measured in the number of clock cycles it takes for a control transfer to complete. To minimize this penalty, a standard method is to predict if a branch is taken and where it will jump.

Figure 3.3 illustrates the instruction streamer module of the PP and how it deals with control transfers and branch prediction. The address of the next instruction to fetch from storage is stored in the program counter (PC) register. If jump, hold, delay and predicted taken are low, its value incremented by one is fed back to the register input through the three multiplexers.

Control transfer If downstream logic initiates a control transfer, the jump signal is asserted and the target address is presented on the jump vector signal. Multiplexer M0 now selects the new address as input to the PC. To give feedback to downstream logic, when the control transfer has completed, the jump signal is delayed by a series of flip-flops to assert the complete signal together with the instruction fetched from the target address. Between the assertion of jump and complete, downstream logic ignores data from the streamer. The time from initiating a control transfer until the first instruction from the target is available to downstream logic is at least two cycles.

Storage delay and pipeline stalls If instruction storage can not provide the requested data on the next clock cycle it asserts the delay signal and returns a no-operation instruction. This toggles M1 to keep the current value in the PC, so that a fetch can be re-attempted in the next cycle. A possible reason for delay is a miss on an

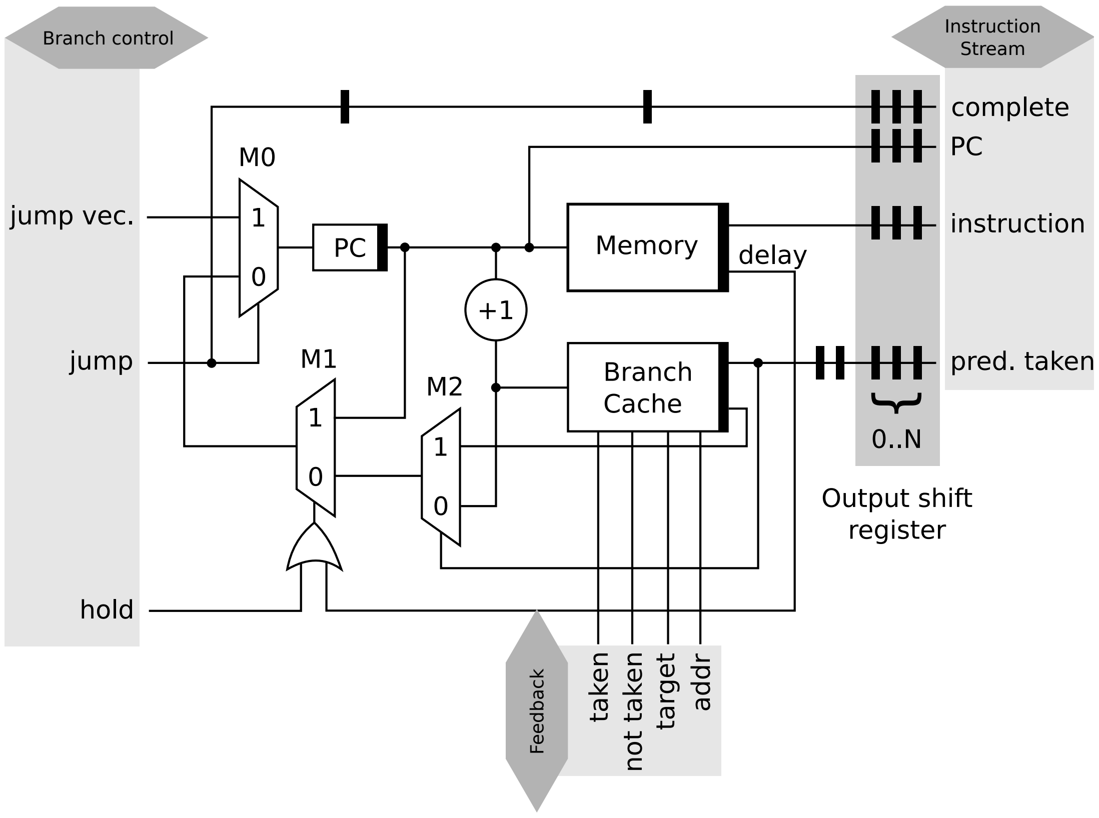


Figure 3.3.: Block diagram of the instruction streamer module. It generates addresses for fetching instructions from memory or cache while using the branch cache module to predict branches based on the instruction address. Filled, black rectangles represent flip-flops inserted for delay purposes. The output shift register can be configured with a variable number of stages. Blocks with a thick right side represent synchronous modules, where the outputs to the right are updated one clock cycle after the inputs to the left are set.

instruction cache. If a hazard is encountered, the pipeline is stalled for a number of cycles. This is indicated to the instruction streamer by asserting the hold signal and thereby keeping PC at its current value.

Branch prediction The branch prediction predicts whether the next address holds an instruction causing a control transfer and the target of the transfer. As the name "branch cache" indicates, it operates similar to a cache, maintaining a table with addresses and associated outcome and target. It is trained by the feedback input from the downstream branch functional unit that will indicate addresses holding branches, whether they are taken or not and where they jump. The cache is fully associative and uses 2 bit saturating counters (Smith, 1998). When a branch is encountered that is not already stored in the cache, its address is recorded as tag for the lookup, the target is recorded as associated data, and the counter is initialized to 2. When the same address is about to be fetched again, the cache lookup will hit and the Most Significant Bit (MSB) of the stored counter value is used to predict outcome. If the branch is predicted taken, M2 will override the incremented address as new value for the program counter, so that fetching resumes with the stored target address one cycle after the branch instruction itself. If the prediction was wrong, the downstream branch unit will detect this by the predicted taken signal of the instruction stream interface. It then signals appropriate feedback to decrement the counter for incorrectly predicted taken branches and vice versa for incorrectly predicted not taken. In parallel it causes a control transfer using the branch control interface to the correct address after the branch. Because instructions are issued in-order and because branches are decided with a minimum latency of one cycle, no incorrectly fetched instruction can retire before the true branch outcome is known. This avoids costly hardware to revert the effect of incorrectly fetched instructions after a mis-prediction.

As a variant, branch prediction can also be used with a direct-mapped cache. This allows for higher clock speeds, but sacrifices some performance (see Analysis in Section 5.1.1).

Output shift register The instruction streamer features an output shift register of configurable length. This is intended to be used in situations, where instruction storage is physically situated far away from the processor core. If this is not the case, it is configured to length 0 to minimize the branch penalty.

The arrangement of the multiplexers M0-2 determines the priority of branch prediction, pipeline stall, and control transfer in setting the next instruction address. An explicit control transfer always has the highest priority, so that it can take effect during a pipeline stall. Prediction has the lowest priority. If it is overridden by a stall, it will re-predict in the next cycle.

In the current implementation, only one cycle is available for branch prediction. This possibly limits the complexity of the implementable prediction algorithm, especially the cache size. However, a prediction latency of one cycle is required to jump to the predicted target directly after the branch in the instruction stream. Allowing a higher latency makes it necessary to discard fetched instructions if a branch is predicted taken independent of the correctness of the prediction. To work around this limitation, one would have to either speculatively predict more than one instruction into the future or increase the fetch bandwidth together with a buffering queue after the memory. Under this constraint, it is not possible to use the fetched instruction for the prediction, for example to include a static prediction for un-cached branches.

The instruction streamer represents a universal streaming unit that is not specific to a particular ISA. It can also be used for other applications apart from micro-processors, where a stream of data is generated from a possibly cached memory. In the context of neuromorphic hardware, such an application is for example the playback of stimulation events for a neuromorphic device, which is typically implemented within a controlling Field Programmable Gate Array (FPGA). In this case instructions would be events with a time stamp that indicates at which time they should be sent to the neuromorphic substrate. Logic downstream from the instruction streamer would compare this time stamp to the current system time and throttle the incoming flow similar to pipeline stalls. The concept of branches can be used to implement loops in the stimulation pattern, for example for repetitive background stimulation.

3.1.5. Instruction cache

The PP has two distinct interfaces to instruction and data storage. This is necessary to remove the von-Neumann bottleneck (Backus, 1978): with a single interface the bandwidth of reading and writing data is reduced by accesses required for instruction retrieval. However, if distinct SRAM blocks are used on-chip, it must be decided beforehand how much storage is reserved for program and data. Space that is not used by one can not be transfered to the other, so that memory is wasted. One alternative is a dual-port memory, so that both accesses can be made in parallel. This is paid for with a less area efficient memory, since SRAM cells require a second port. While this is still a viable option, for example for FPGA implementations, where integrated SRAM blocks are dual-ported anyway, a more elegant solution is the use of an instruction cache. Due to loops in the program, a sufficiently large cache reduces the number of accesses to its backing store allowing more bandwidth for data accesses. This cache can be relatively simple, because the processor only reads from it.

Figure 3.4 shows a high-level view of the read-only instruction cache. It is a direct mapped implementation exposing a RAM interface to the processor on the front and a PPB interface on the back towards storage. Data is stored in three memory structures: Two register files maintain information about valid entries per cache line and their tag.

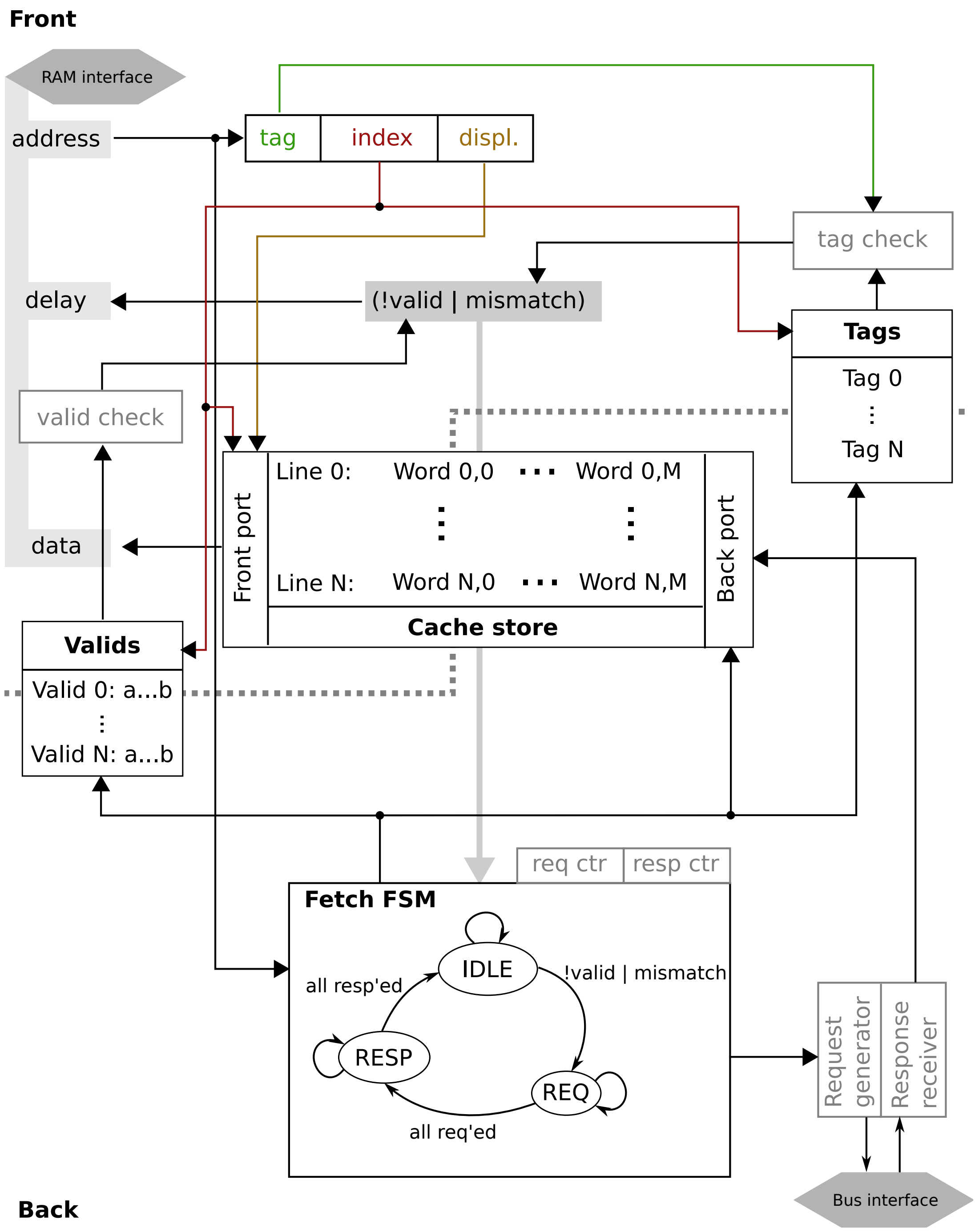


Figure 3.4.: Schematic overview of the read-only cache used as instruction cache. The front side faces the processor through the RAM interface and the back side accesses main memory through a bus interface. The cache store is a dual-port memory that is external to the control logic. It is connected by two additional RAM interfaces not explicitly shown in the figure. See main text for a description of cache operation.

The cached data are kept in a dual-ported store with one read and one write port. The cache store is situated external to the module and accessed by two RAM interfaces (not shown in the figure). This allows efficient implementation using a generated SRAM macro. In FPGA based implementations, SRAM hard-macro blocks are used, while the wafer-scale system with plasticity processor uses latch based memory (see Chapter 5 for descriptions of the systems). The front side performs lookup in the cache for requests via the RAM interface and in particular detects misses, i.e. requests to data not available in the store. The back side is triggered by misses and fills the store from main memory.

Front Data in the cache store is organized in N lines of M words each. For each line two valid pointers a, b are kept and a line valid bit. The pointers indicate the first and the last valid entry in the line masked by the per-line bit. Upon a read request, address is split into tag, index, and displacement. The index selects the cache line and is used to address valid-, tag-, and cache store. The displacement selects the word within the line and is used as offset into the cache store and to test validity. The result from the tag store is compared with the tag taken from the address to ensure, that the requested data is stored at the selected cache line. If the requested word is marked as valid and the tags match, the request is a hit and output from the cache store can be forwarded to the requesting processor. Otherwise the request misses, which is indicated to the processor by asserting the delay signal. This signal is evaluated by the instruction streamer module shown in Figure 3.3.

Back A miss will also trigger the back side to retrieve the missing data from main storage. The fetch Finite State Machine (FSM) invalidates the selected line while transitioning into the REQ state. It then refills the line by generating requests on the PPB interface. The first request is generated for the selected word within the cache line. As soon as the corresponding response is received, the new tag is stored for the line. The valid table is updated by setting the line valid and a and b to the retrieved word. This allows the front side to hit for this address in the next cycle, without waiting for the whole line to be filled. This is the reason for keeping pointers a, b additionally to the line valid bit. If the bus to main storage allows for fetching one word per cycle, the processor does not need to be delayed any further, even if it requests a new address every cycle. Thus, the cost of cache misses is kept small. The fetch FSM continues generating requests, until the request counter reaches N, i.e. all words in the line are requested. It then transitions into the RESP state to receive the remaining responses until the response counter indicates, that the complete line has been filled. Responses are also processed while in the REQ state.

While filling of a line is in progress, a new miss might occur in a different line. In this case, the front side sees the miss and waits. The back side continues filling the current line until the FSM enters the IDLE state. Then, it starts a new refill sequence for the second miss.

The read-only instruction cache represents a universal design that operates without knowledge of instruction coding or access patterns. The fetch FSM could be extended to pre-fetch cache lines trying to reduce the number of misses. This is supported by the Power ISA with the instruction cache block touch (icbt) instruction, that references addresses expected to be needed soon. However, inserting these instructions into the program increases its size and adds potentially unnecessary memory accesses for pre-fetch. This can increase power consumption (Zhuang and Pande, 2007). The use of pointers to mark valid entries in a line allows for early continuation after a miss, reducing miss latency. Because the cache is direct mapped, standard SRAM macros can be used to implement the main data store.

3.1.6. Instruction scheduling

Because of pipelining, several instructions are ongoing simultaneously at various stages of completion. Interdependencies between subsequent instructions can cause data hazards. There are three types of data hazards classified by the order and type of accesses (Hennessy and Patterson, 2007): for read-after-write hazards, a subsequent instruction reads a result from an earlier instruction. Reading the result has to be delayed until the write is complete. A write-after-read hazard exists, if a subsequent operation writes a register, before the earlier one has read it. In write-after-write hazards, two instructions write to the same location in the incorrect order. Write-after-read hazards are excluded in the presented implementation, because of in-order issuing and the requirement for functional units to store operands internally during execution. Therefore, it is guaranteed, that all operands are read, before the next operation can write a result. The other two types of hazards have to be detected and solved by stalling the pipeline until the hazards no longer exists.

3.1.6.1. Result shift register

The RSR, shown in Figure 3.5, is a component to track the execution of in-flight instructions, in particular when and where they write a result (Smith and Pleszkun, 1985). Its operation is quite simple: depending on the latency L of an operation, the destination register number is inserted into a shift register at stage L. The register is shifted in every cycle, so that after L cycles - in synchrony with the completion of the operation - the write falls out of the register. To detect read-after-write hazards, upon a read access the source register is compared in parallel to the stored destinations in the shift register. As long as there is a match, a hazard exists. To accommodate instructions reading up to three registers, as many test ports with comparators are

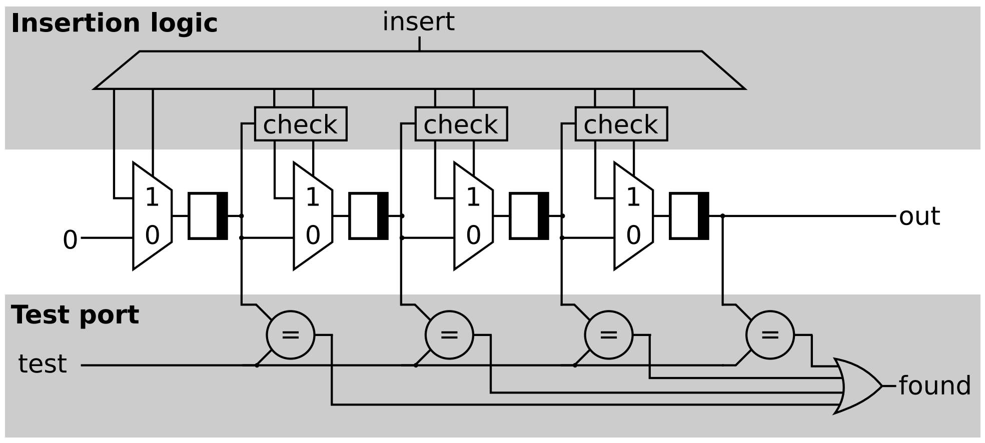


Figure 3.5.: Simplified schematic overview of a RSR with four stages and one test port. The concept is based on the design presented by Smith and Pleszkun (1985). The RSR controls the write back of instruction results, and detects data hazards between operations.

required. A comparison of the write destination to the stored writes in higher stages allows for detection of write-after-write hazards (not shown in Figure 3.5). Additionally a collision check prevents inserting writes into stages already holding a write.

Because the RSR tracks writes, it is straightforward to use it to schedule write back to registers. To this end, stages additionally hold the functional unit of an operation to act as source for the write. If the last entry of the shift register is marked valid, the destination register is written from the functional unit coded in the source field. This is illustrated in Figure 3.6. Write destination and read sources are identified by the pre-decode unit (Figure 3.2) and fed to the RSR. Matches on the read test ports and the write-after-write test port are combined with a logical nor to get the ready signal. If asserted the current instruction is ready for issue and written to the RSR. After L cycles a write configures the write back multiplexers to select the functional unit executing the instruction and the destination register. The empty signal is asserted if no write is held at any stage of the shift register. It is used for example to wait for all in-flight instructions to finish, before transitioning into a sleep state.

3.1.6.2. Write back channels

The Power ISA defines a number of different registers besides the 32×32 bit general purpose registers. For example, there are condition registers used as output of comparison operations and input to branches. A number of instructions write to more than one register, for example recording add with overflow (addo.). Table B in Appendix B gives a full list of registers. For each subset of registers that can be written simultaneously

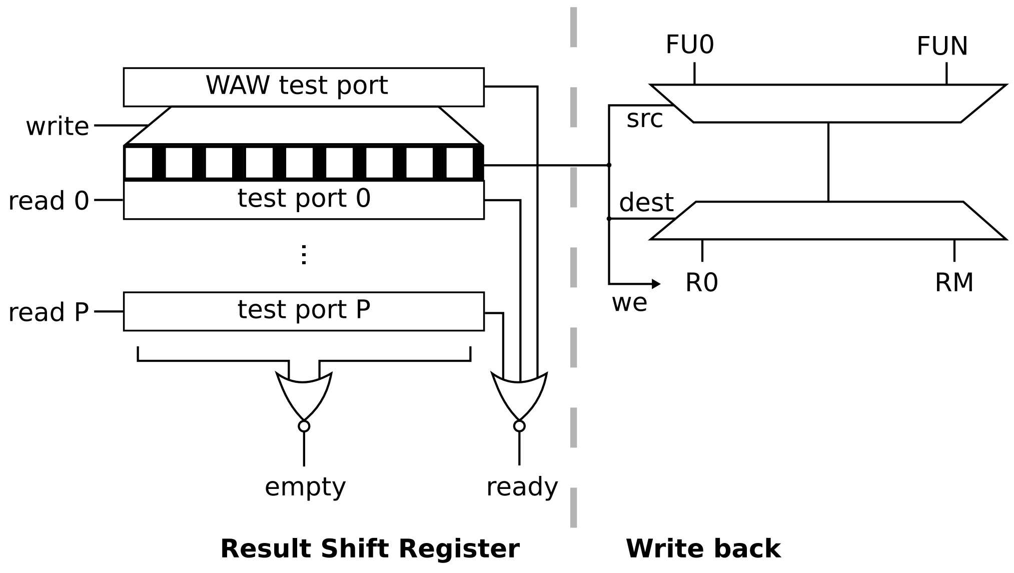


Figure 3.6.: Scheduling write back to registers and detecting data hazards using the RSR: The right side shows the data path from the output of functional units FU0. . . FUN to registers R0. . . RM. The left side shows the RSR that controls the data path on the right. Test ports $0 \ldots P$ detect read-after-write hazards for new instructions, the WAW test port detects write-after-write hazards. If neither hazard is detected the instruction is ready for issue. The RSR can also detect, whether the pipeline is currently empty.

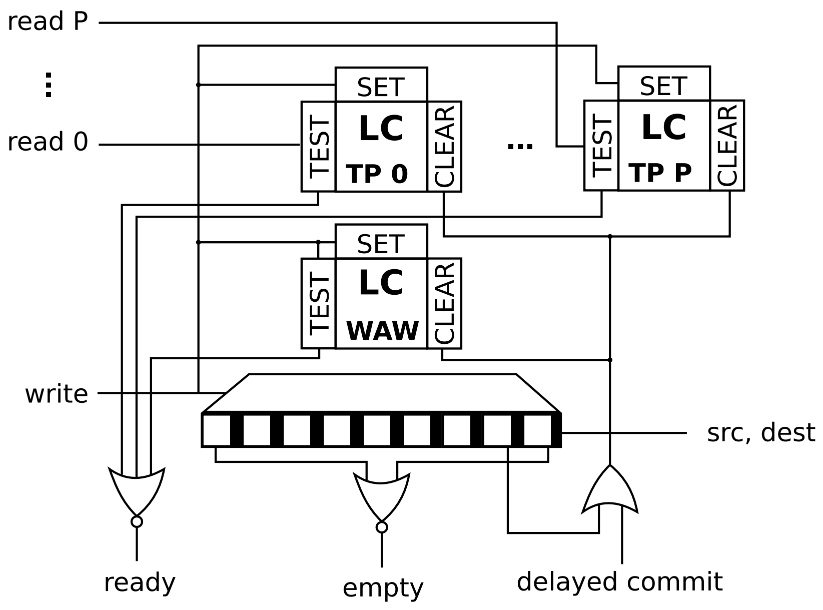


Figure 3.7.: Write back to registers using the RSR with lookup cache. Test ports are replaced with the lookup cache (LC) structure explained in the text. This is an area and speed optimization for long shift registers. The lookup cache is also required, if instructions can commit results delayed to their scheduled write back slot.

Write back channel	**Stages**	**Ports**	**DC**	**Destinations**
General purpose registers	8 or 16	3	yes	32
Special purpose registers	4	2	no	1024
Condition register field 0..7	8 or 16	1	no	1
Counter register	4	1	no	1
Link register	4	1	no	1
Fixed-point exception register	8 or 16	1	no	4
Machine state register	4	1	no	1
Branch dummy	4	-	no	-
Memory dummy	4	-	no	-

Table 3.1.: List of write back channels. The number of RSR stages is defined by the maximal latency of instructions writing through the respective channel. Only the general purpose register file requires the ability to perform delayed commits (DC) of results, because writes to other registers always have deterministic timings. The branch and memory dummy channels are not used to detect data hazards, but to synchronize control operations. For example, before entering a sleep state, the front end has to ensure, that all instructions currently in-flight have retired. The empty signal from the RSR is used for this.

with the others, there is an individual write back channel. Each write back channel has its own RSR to schedule writes and detect hazards. A list of the channels is given in Table 3.1. Depending on the maximum latency of instructions writing over a particular channel, the respective RSR can be shortened to save area. The shown maximal length of eight is used by the multiplier. The latency of the divider is even greater (28 cycles), but here a trick is used to keep the shift register short: division is treated as a multi cycle instruction (see Section 3.1.6.6) with L cycles. This blocks issuing of other instructions while the divide is ongoing, yet as divides occur relatively seldom in typical code, this is a good compromise. Apart from length, write channels employ different numbers of read ports and destination registers. The special-purpose register file according to specification can address up to 1024 registers. However, not all of those registers are defined and only few need to be implemented by a minimal design. The PP has only 29 special-purpose registers. To save area a reduced representation can be used in the RSR, but this was not done for the presented implementation. The "Delayed commits" column indicates whether the channel requires the ability to retire instructions with a variable latency. This necessitates the use of a lookup cache in the RSR, which is detailed in the next section.

3.1.6.3. Lookup cache and variable-latency operations

The approach outlined in Figure 3.6 is straightforward and simple, but has a fundamental flaw: it can not deal with operations, for that latency is variable. Such operations are for example loads via the PPB, which return after an unspecified amount of cycles, depending on which address is being used. The lookup cache represents an alternative replacing the test ports to detect hazards. The shift register works as before for fixed-latency operations to schedule result write back. Variable-latency instructions use the delayed commit mechanism that employs a handshake with the write back module to take opportunity of free write slots in the RSR. When a slot is available, the write is signaled to the RSR with lookup cache. The lookup cache simply stores one bit per destination to mark, whether an in-flight operation writes to it. It is set by the write port of the RSR and cleared by either the output from the RSR or the delayed commit. So in a sense it caches the lookup into the shift register in a separate memory. This concept is shown in Figure 3.7. One lookup cache is used for each test port and another one for write-after-write detection. Variable-latency instructions can use the shift register to reserve a write slot for an expected default latency. If they take longer, they have to wait for a free slot. Such a slot will eventually occur, when the pipeline stalls for a dependent operation that reads or writes the result register. [1]

The lookup cache is necessary for variable-latency instructions in the presented framework. Apart from that, it can serve as an area and timing optimization compared

[1] If this is not the case and there are never any free write slots, the instruction will never retire. However, such code is considered irrelevant.

to the comparator stages of the RSR. The test ports of the shift-register constitute in essence a content addressable memory. With the lookup cache this is turned into a classical memory saving a comparison unit per memory item. Which approach to use depends on the number of destination registers that need to be tracked and the length of the shift register. For long shift registers and few destinations, the lookup cache is smaller, since it removes comparators and allows for a smaller NOR-reduction for the ready signal. With many destinations, the sparse representation in the shift-register allows for better scaling in the number of flip-flops and the size of comparators.

3.1.6.4. Write-through optimization

So far, scheduling assumed, that a dependent instruction can only read an operand one cycle after it was written. However, one cycle can be saved if the register file allows write-through: if the same register is read and written in the same cycle, the read will return the newly written data. This creates a timing path starting from the source flip-flops of the write to the destination flip-flops of the read. Without write-through this path is broken by the register file. On the other hand, it reduces the penalty of data hazards by one cycle. Therefore, the option to use write-through allows for a trade-off between maximum clock frequency and computational efficiency.

For hazard detection this means, that dependent instructions can be issued one cycle earlier than without write-through. To implement this in the RSR, it is enough to exclude the last stage of the shift register from checks by the test ports. When using a lookup cache, the reservation for the in-flight instruction is cleared one cycle earlier by using the second last stage of the RSR, instead of the last.

3.1.6.5. Pre-decoding instructions

Writing and checking the RSR is done in the fourth stage of the processor pipeline (Figure 3.2). The preceding stage in the pipeline performs pre-decoding of instructions. Its task is to determine information necessary for dependency tracking and scheduling. For example, the source and destination registers, latency, and used functional unit. The functional units perform further decoding later in the pipeline to determine the remaining control signals, e.g. which operation to perform in the fixed-point data path. For the front end, the pre-decoded information is sufficient. All signals of the control word are listed in Appendix C on page 231.

The basic operation of the pre-decoder is to map the 32 bit instruction word onto the 123 bit control word. Each signal in the control word can be computed independently from the others in combinatorial logic. In an early design variant, this type of decoding operation lead to a large amount of repetitive RTL code. Therefore, I decided to use a more methodical approach in later iterations using macro generated code. The RTL code of the design is written using the SystemVerilog hardware description language

(SystemVerilog, 2004). Instead of using the built-in pre-processor macros, I selected the m4 language (Kernighan and Ritchie, 1977) for code generation for its more advanced capabilities. The features of m4 are mostly relevant later in the context of specifying bus structure (Section 3.2.5).

For pre-decode, comparison macros using the `unique case` SystemVerilog construct are defined, and lists of opcodes by category compiled. For every pre-decode signal, a decode function compares with the instruction and sets the signal in the resulting control word appropriately.

Listing 3.1: Decoding the read_ra signal using macros.

```
 1 ...
 2
 3 DEF_FUN_CMP(pd_read_ra, logic, 1'b0)
 4   STORE_IMMEDIATE_OPS, STORE_INDEXED_OPS, LOAD_NOUPDATE_OPS,
 5   OP(Op_lmw), OP(Op_stmw),
 6   OP(Op_addi), OP(Op_addis):
 7     rv = (fst.inst.x.ra != 0);  // only read when RA != 0
 8
 9   ADDSUB_REG_OPS, OPXO(Xop_neg),
10   OP(Op_addic), OP(Op_addic_rec), OP(Op_subfic),
11   MUL_OPS,
12   LOGICAL_OPS, COMPARE_OPS, TRAP_OPS, ROTATE_OPS,
13   OPX(Xop_mtspr), OPX(Xop_mtocrf),
14   DEV_CTRL_OPS:
15     rv = 1'b1;
16
17   DIV_OPS:
18     rv = 1'b1;
19 END_FUN_CMP
20
21 ...
22
23 PD_MAP(read_ra)
```

An example for such a function using macros is given by Listing 3.1 for the `read_ra` signal. This signal is set if the general purpose register given in the bit-field `RA` of the instruction word is read from the register file by this operation. Invoking the `DEF_FUN_CMP` macro, name, type, and default value of the control signal `read_ra` are declared. Lines 4 to 18 list all instruction names that do read the general purpose register given in `RA`. Line 7 implements the RA | 0 referencing mode (PowerISA, 2010): the instructions listed on the previous lines always read zero if operand RA references register 0. The `PD_MAP` macro in line 23 enables decoding by the defined function for the `read_ra` signal. Listing 3.2 shows the emitted code after preprocessing by m4.

The shown methodology simplifies specification of decoding logic and makes the

Listing 3.2: SystemVerilog code after pre-processing of Listing 3.1.

```
...

function automatic logic pd_read_ra(input Fetch_state fst);
  logic rv;

  unique casez({fst.inst.x.opcd, fst.inst.x.xo})
    {Op_stw, 10'bz}, {Op_stwu, 10'bz}, {Op_sth, 10'bz}, {Op_sthu, 10'bz
        },
    {Op_stb, 10'bz}, {Op_stbu, 10'bz}, {Op_stmw, 10'bz}, {Op_alu_xo,
        Xop_stwx}, {Op_alu_xo, Xop_stwux},
    ...
    {Op_addi, 10'bz}, {Op_addis, 10'bz}:
      rv = (fst.inst.x.ra != 0);  // only read when RA != 0

    {Op_alu_xo, 1'bz, Xop_add}, {Op_alu_xo, 1'bz, Xop_addc}, {Op_alu_xo,
         1'bz, Xop_adde},
    {Op_alu_xo, 1'bz, Xop_addme}, {Op_alu_xo, 1'bz, Xop_addze},
    ...
    {Op_alu_xo, Xop_eciwx}, {Op_alu_xo, Xop_ecowx}:
      rv = 1'b1;

    {Op_alu_xo, 1'bz, Xop_divw}, {Op_alu_xo, 1'bz, Xop_divwu}:
      rv = 1'b1;

    default:
      rv = 1'b0;
  endcase

  return rv;
endfunction

...

always_comb predec.read_ra = pd_read_ra(fst);
```

code more readable. Especially, the use of instruction lists has proven useful in increasing maintainability of the source. A new instruction can typically be integrated into pre-decode by listing it in all decoding functions, for which it deviates from the default value. The macros defined here are also used by decode logic in the functional units and for control of operand fetching. On the other hand, m4 is only used for text substitution, which could also be done using built-in `` `define `` macros. Therefore, for this task alone it would be better to avoid an additional tool. However, for other tasks (see Section 3.2.5), the built-in macro language is not sufficient, and thus has to be used anyway.

The pre-decode pipeline stage performs most of the decoding work and is the main dependency on the ISA. The rest of the front end is completely agnostic of instruction coding. Using macros, this coding can be efficiently specified in the RTL source.

3.1.6.6. Multi cycle operations

A few instructions in the Power ISA proof difficult to implement: load multiple (lmw) and the load with update variants (lbzu, lhzu, . . .). These instructions write to more than one general purpose register, while the write back channel provides only one write port. In the case of store multiple (stmw) the same problem arises for the memory. This is solved by performing multiple writes sequentially for a single instruction. To implement this in a generic way, a universal framework for multi cycle operations is provided.

Figure 3.8 shows how this framework operates. The pre-decode module defines decoding logic for every control signal that changes during the multi cycle operation. It consists of decoding logic for every cycle in the sequence and a multiplexer to select the appropriate one. The multiplexer is controlled by a counter incrementing with every cycle. Pre-decode detects the beginning of a multi cycle operation by the instruction's operation code. It then asserts the `is_multicycle` signal to start the cycle counter. This also holds instruction fetching until all cycles have completed. The total number of cycles is determined by the `multicycle` signal of the control word. All cycles of the operation are tested for hazards, which are indicated in the usual way by deasserting the `ready` signal and thereby stalling the counter. When the counter compares equal to `multicycle` it is reset and `hold` to the instruction streamer removed.

The load and store multiple instructions write or read general purpose registers from a specified index up to the last one. Therefore, the number of multi cycles is variable and needs to be determined by the pre-decode unit. Parameterized decoding functions compute, which registers to read and write using a static mapping from counter values to register index. Since the counter starts at zero and only the last register is always referenced, CTR = 0 is mapped to index 31 and indices decrease from there. So if a lmw instruction in the program references register 1, execution

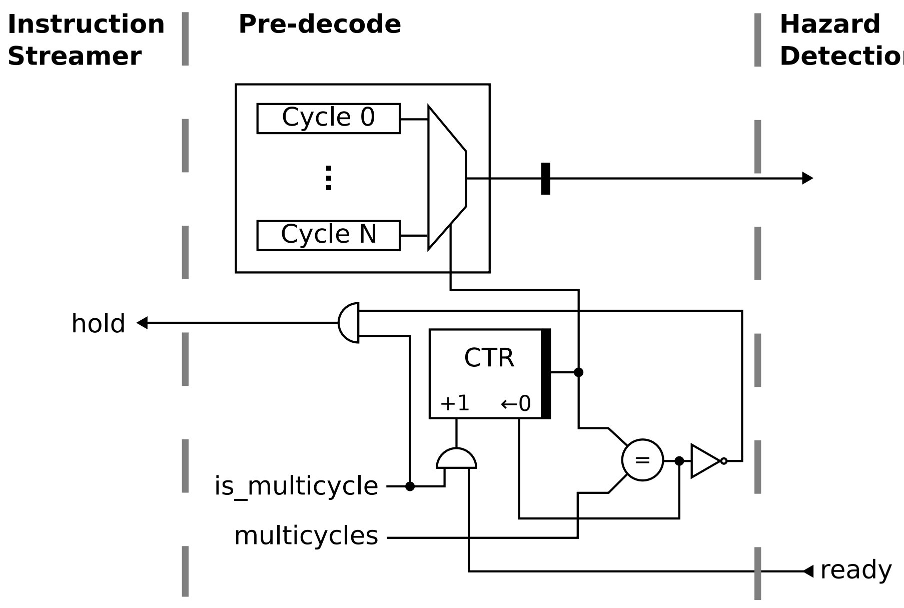


Figure 3.8.: Simplified schematic of the control logic for multi cycle instructions. Normal instruction fetch is stalled (`hold` signal), and a predetermined sequence of control signals is send to downstream logic. For the remaining processor, a multi cycle instruction is equivalent to a short sequence of operations.

takes at least 31 cycles. If a data hazard exists for any register in the accesses range, the pipeline is stalled, when the parameterized decoding function selects this particular register. The load with update instructions write the retrieved result from memory and its effective address to two general purpose registers in two cycles. The divide instruction is implemented as multi cycle operation to reduce the required length of the RSR. Pre-decode generates the appropriate number of no-operations after a divide to delay execution of subsequent instructions until the divide has completed. Since division is used only infrequently in typical programs (Weicker, 1984; Patterson and Hennessy, 1996), it is viable to completely block execution.

This framework for multi cycle execution is similar to the concept of micro-operations (Wilkes, 1969). It is however employed only minimally for just the control signals of just the instructions requiring it.

3.1.6.7. Scheduling instructions to functional units

The last stage of the front end has to send instructions to functional units and control operand fetching. To do this, it has to consider detected hazards and whether functional units are ready to accept a new instruction. This task is fulfilled by the scheduling block shown in Figure 3.9A. It distributes the fetched and pre-decoded instruction to one of the functional units if the `ready` signal from hazard detection is asserted. The information passed to the functional unit is comprised of the full instruction, the control word, the current and next instruction address, and a valid bit. Functional units either are always ready to accept instructions (FU A in the figure) or use an Functional Unit Manager (FUM) to compute the ready signal. An operation may for example take multiple clock cycles to execute. A counter in the FUM removes the ready signal until the execution has finished. Other units can be used without constraints during this time. Therefore, in the terminology of Patterson and Hennessy (1996), managers detect structural hazards, i.e. hazards resulting from unavailable execution resources.

Delayed write back The FUM also coordinates delayed write back of variable-latency instructions. This is illustrated in Figure 3.9B: The functional unit requests a write back slot from its associated manager, when it has completed execution. Control of the write back channel for the destination registers is chained through all managers supporting delayed write back. If on the in port no write is signalled, the manager inserts a write for the requesting unit on the out port. The order of the chain therefore defines the priority, with which write back slots are granted. Fixed-latency writes from the RSR have highest priority and are always granted. When acknowledging the write to the functional unit, the manager also informs the RSR in the hazard detection via the `delayed_commit` port shown in Figure 3.7. Currently, only the general purpose and fixed-point exception register write back channels are chained through managers in this manner. FUMs control load/store using the PPB, external control, and the divider.

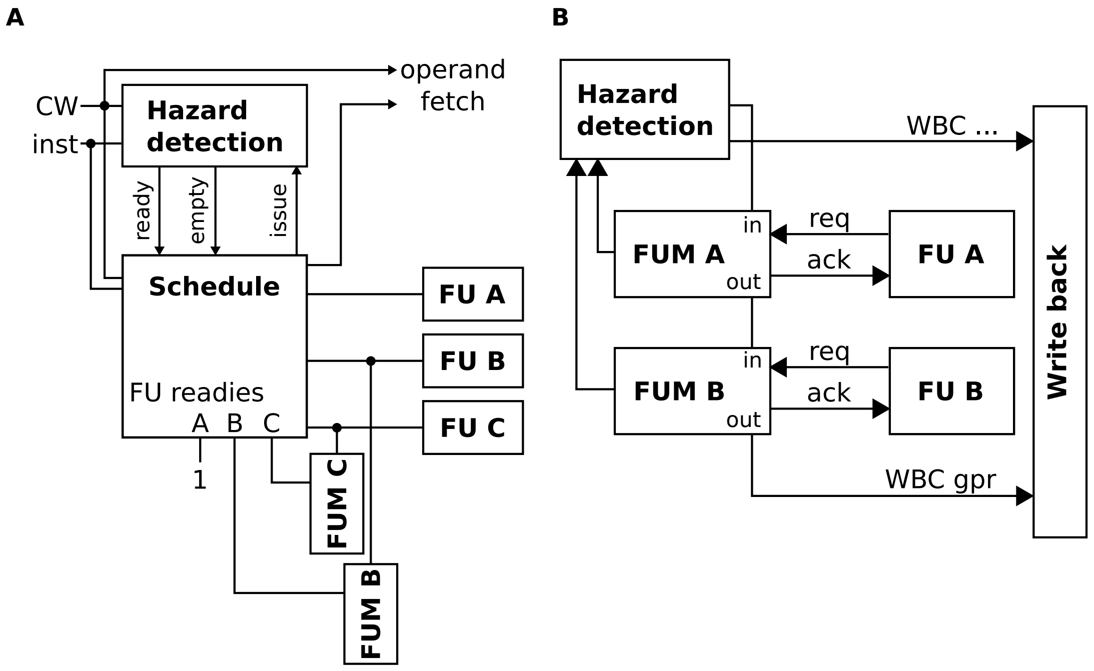


Figure 3.9.: Distributing instructions to functional units. (A) The schedule block considers the ready signals from instruction tracking and functional units to issue instructions to the associated functional unit, when they are free of hazards and the unit is ready to execute. The Functional Unit Manager (FUM) determines, if a functional unit is ready to accept new instructions. Most units do not need an FUM, since they accept instructions in every cycle. (B) Control of delayed write back using FUMs. The write back channel control signals from the RSR are chained through FUM blocks. When using delayed write back, functional units request a write back slot, which is granted by the manager, if the upstream channel does not indicate a write. In this way, the arrangement of the chain defines write back priority, giving highest priority to fixed-latency instructions, then load/store, external control and divider.

Only load/store and external control have a delayed write back capability. A manager is only needed for the divider, if the multi cycle trick described in Section 3.1.6.6 is not used. This is for example the case in FPGA implementations, where hard-macros for division are available that have an acceptable latency.

Controlling state machine The schedule block in Figure 3.9A is implemented as a Finite State Machine (FSM). While executing sequences of instructions without branches, it simply uses the ready signals from hazard detection and FUMs to distribute the incoming instruction stream to the correct functional unit. Beyond that, the FSM has to deal with program start after reset, control transfers, context synchronization (explained below), interrupts, and transition into a sleep state. The controlling FSM is the central control unit of program execution. The rest of the front end acts as data path for it, processing the instruction stream. Functional unit managers perform specialized control operations for individual functional units. They keep track of execution state and coordinate delayed write back with other functional units.

Figure 3.10 shows a simplified version of the state diagram of the controlling FSM. Here, some transitions for unusual situations are left out. The full diagram is shown in Appendix C on page 230.

After reset, the FSM is in the `RESET` state. The instruction streamer starts to fill the pipeline by fetching instructions from storage. The stream marks the first retrieved instruction with a valid signal. When it passes pre-decode, the controlling state machine transitions into the `FETCHING` state. This is the state of normal program execution, where operations are send to the appropriate functional unit, if no hazards exists.

The branch functional unit causes control transfers and signals them to the instruction streamer and the controlling FSM. As described in Section 3.1.4, the streamer will deliver invalid operations for a number of cycles until the transfer is complete. Downstream logic detects completion by monitoring the delayed `jump` signal (Figure 3.3). The controlling FSM transitions to the `JUMPING` state, when a branch is signalled. In this state, it ignores incoming instructions. It remains in this state, until it detects completion, upon which it transitions back to the `FETCHING` state.

Instruction execution is affected by certain register bits and other processor state information, which are called the context of the instruction. Context-altering instructions are not required to take effect in the program order defined by the sequential execution model. To ensure, that an instruction issued after a context-altering instruction is executed in the new context, context synchronization has to be used. The controlling state machine implements context synchronization by transitioning to the `SYNCING` state, before executing the synchronizing instruction. This causes the pipeline to stall, until all RSRs of the instruction tracking module are empty, i.e. all previous operations have retired. Then, normal operation resumes in the `FETCHING` state.

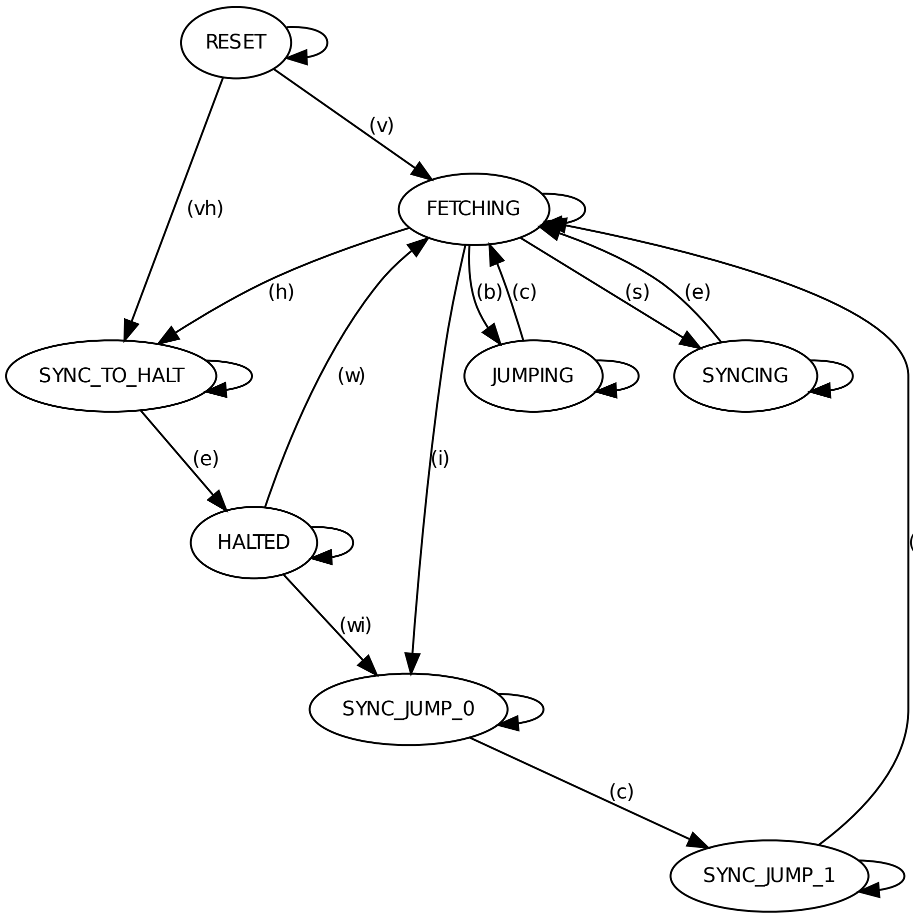


Figure 3.10.: Simplified state machine diagram for the scheduling FSM. Transitions are labeled with the asserted input signals, that trigger the transition: valid instruction from instruction streamer (v), halt (h), context synchronization (s), control transfer complete (c), branch (b), interrupt (i), all RSRs empty (e), and wakeup from sleep (w). After reset, the FSM is in the `RESET` state. Some transitions for special cases are omitted. The full graph is shown in Appendix C.1.2.

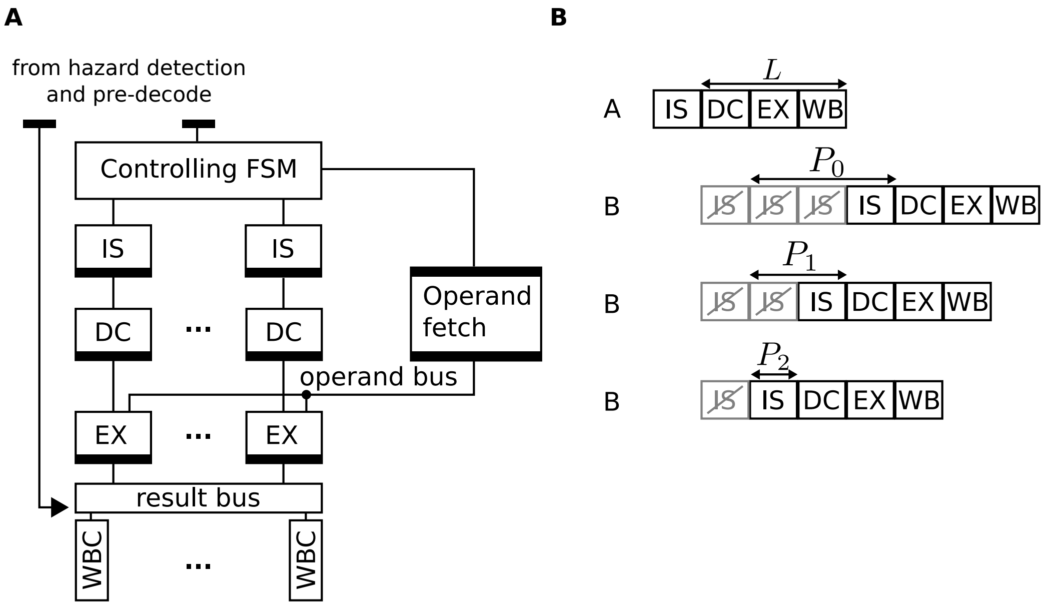


Figure 3.11.: (A) Pipeline stages of functional units in the back end. Functional units operate in parallel. Execution starts after issue (IS), and takes one cycle for decode and operand fetch (DC). After that, operands are available for the execution stage (EX). (B) Penalties P_0, P_1, and P_2 by data hazards depending on issue-to-retire latency L for three scenarios: issue after retire, issue in time, and issue in time with write-through. Instruction B reads a result from instruction A. Each box represents one clock cycle, in which execution passes through one of the pipeline stages issue (IS), decode and operand fetch (DC), execute (EX), and write back (WB). To indicate, that B waits for issue due to the hazard, the issue box is greyed out.

Interrupts are also context synchronizing. In this case, the controlling FSM combines synchronization with a control transfer. After the interrupt is signaled, the `SYNC_-JUMP_0` state waits for completion of the control transfer and `SYNC_JUMP_1` for the empty signal. After that, normal fetching resumes in the new context.

The wait instruction puts the processor in a sleep state, in which execution is disabled until an external source wakes it up again. To the FSM this is indicated by the halt signal, causing a transition to `SYNC_TO_HALT` to wait for the empty signal. While sleeping, the FSM remains in the `HALTED` state until wake-up is triggered by an interrupt. Typically, clock gating will disable the processor clock after the `HALTED` state is reached and only re-enable it for leaving the state.

A Operand bus

Name	Width	Description
`a`	32	Operand A
`b`	32	Operand B
`c`	32	Operand C
`cin`	1	Carry-in
`so`	1	Summary overflow
`cr`	4	Condition register field

B Result bus

Name	Width	Description
`res_a`	32	Result A
`res_b`	32	Result B
`cout`	1	Carry-out
`ov`	1	Overflow
`crf`	4	Condition register field
`msr`	32	Machine state register
`valid`	1	Valid bit

Table 3.2.: Interfaces for functional units. (A) Data input to functional units through the operand bus. (B) Data output from functional units through the result bus.

3.1.7. Execution in the back end

At the end of the front end of the pipeline, instructions are issued to their associated functional units. The controlling FSM writes instruction, its address, and the – possibly predicted – next address into an issue slot and marks the data as valid. Simultaneously, it starts to fetch operands required for execution. Figure 3.11A visualizes execution in the back end. Operand fetching involves readout of register storage and multiplexing results to a number of ports of the operand bus. This is performed within one clock cycle after issue in parallel to decoding of the instruction in the functional unit. In the next cycle, operands are available on the operand bus and execution begins. Depending on the operation, this can take a number of cycles. At the end, a result is presented on the result bus, which the write back channels read as controlled by their RSRs or through delayed write back. Therefore, the minimum latency of instructions from issue to retire is three cycles.

Back-end interface Functional units are connected to two standardized interfaces: the operand and result buses. Their signals are listed in Table 3.2. The operand bus provides three 32 bit operands and a number of status signals (Table 3.2A). The `cin` signal is used for extended arithmetic operations with more than 32 bit data types. The `so` and `cr` signals are used for operations on the fixed-point exception and condition registers. The result bus (Table 3.2B) carries two 32 bit words, because some instructions provide two result words. For example, a branch can decrement the counter register and record the return address in the link register, providing new values for both registers on `res_a` and `res_b`. The valid bit is relevant for delayed write back. If a fixed-latency write slot is allocated for a variable-latency instruction, deasserting valid prevents, that incorrect data is written to registers. The remaining bits of the interface are used to record status information.

Penalty by data hazards The issue to retire latency L determines the penalty by data hazards. Figure 3.11B shows pipelined execution of two dependent instructions A and B. The penalty is $P_0 = L$, if B is issued only after A has committed its result to the register file. This is the case, if all stages of the RSR including the last one controlling the write back channel are checked by test ports (Figure 3.5). This wastes one cycle, because operand fetching could start directly after write back of A. By removing the last stage of the RSR from hazard checking, subsequent instructions can be started in time, so that they enter the operand fetch state, when the result is available. This reduces the penalty by one $P_1 = L - 1$. If the register file supports write-through (Section 3.1.6.4), write back and operand fetch can occur in the same cycle. By removing the last two stages of the RSR from test ports, instructions issue in time for this scenario. The penalty is then further reduced to $P_2 = L - 2$.

To some degree, the latency of instructions can be configured. Multiply and divide can have arbitrary latency limited only by the used implementation (see Section 3.1.8.2). Fixed-point instructions can have a minimum latency of $L = 2$, fixed-latency load/store of $L = 3$. A latency $L = 2$ is achieved by omitting the output registers after the execute stage. This forms a timing path starting from the register file, through the execution data path, and back to the register file, possibly reducing the maximum frequency of the design. In this case, there is no penalty for data hazards, if write-through is used. A complete list of latencies is given in Appendix B.

3.1.8. Functional units

For the PP, functional units are relatively simple to design, because challenging arithmetic operations, e.g. floating-point math, are left out. Also, because of the back end architecture, functional units can be designed independent of each other. With the fire-and-forget approach of execution for fixed-latency instructions, the unit simply represents a pipeline to compute its function.

3.1.8.1. Input/output over the plasticity processor bus

Load/store and external control perform data I/O over two PPB ports. They both employ a common bus access module to generate requests on the bus. The main difference is, that external control supports only two instructions to read or write 32 bit words, while load/store can perform byte and halfword accesses.

The PPB is described in more detail in Section 3.2. The processor acts as master, initiating read and write requests to addresses. The request is complete, when the bus has accepted it. Some time after request initiation, the bus generates a response to the request. To complete the response, the master reads the provided data and raises an accept signal. The request and response phases are completely decoupled to allow pipelining. The master can initiate a new request immediately after a first one was accepted, independent of whether the response was already received. The master assigns responses to requests by their ordering. The bus guarantees, that responses arrive in the same order as requests are accepted. To track completion and detect errors, write requests also generate a response.

The bus access module The bus access module has to perform three tasks: generating requests and responses on the bus, stalling the pipeline if requests can not be handed out fast enough, and retirement of responses using the delayed write back method. To do this, two First-In First-Out (FIFO) queues are being used for requests and responses. A new instruction inserts target address, the type of the request, and – for writes – write data into the request queue. In parallel it inserts the destination register into the response queue. A request generator reads the request queue and if it is not empty, it starts an I/O operation on the bus. Upon acceptance by the bus, the entry is removed from the queue and, if available, the next entry processed. Independent of this, responses from the bus initiate delayed write back using the information provided in the response queue. When the delayed write is acknowledged by the front end, the entry is removed from the queue. In the meantime no further responses are accepted. There are two pipeline stages after issue and before an access is stored in the queues. Therefore, the unit is marked as not ready by the manager in the front end, if only two entries remain free in the queues. This way, it is guaranteed, that all entries that are issued are inserted into the request and response queues.

Beyond the basic operation, the module provides facilities to perform byte and halfword accesses potentially with sign extension. It also detects requests, that are incorrectly aligned, i.e. word accesses not aligned to word boundaries and halfword acceses not aligned to halfword boundaries. The Power ISA allows embedded systems to not implement such accesses and instead raise an alignment exception. If such an exceptions exists for the current instruction, is checked after the target address is calculated after the operand fetch phase. If the access is misaligned, the access is marked as such in the request and response queues. The operation is still carried out normally, but no request is generated on the bus and no data written back to registers. Instead of write back of a result, the completion of the instruction is signalled to dependency tracking in the front end using the same mechanism as if a result was written.

3.1.8.2. Multiplier and divider

To perform the arithmetic operations I relied on either synthesizable components from the DesignWare data path building blocks library (DesignWare, 2013) or vendor supplied generated IP cores for FPGA hard-macros. These blocks are preceded by control logic that decodes the instruction and configures the macros to perform the appropriate mathematical operation. After them, status flags for the condition register, overflow, and division by zero are generated. The divider operates sequentially, i.e. after a division is started, it is unavailable until computation completes and the result is returned. This is done, because integer division is assumed to be an infrequent operation and a sequential implementation saves pipelining flip-flops and therefore area. Multiplication on the other hand, is a much more often used instruction, for example in address computation in two-dimensional arrays. Therefore, a pipelined multiplier is used that can start a new operation in every clock cycle. The specific DesignWare components used are `DW_div_seq` and `DW_mult_pipe`.

3.1.9. Interrupts & Exceptions

External events and certain instructions can cause exceptions to occur. Depending on the current context defined by particular bits in status and control registers this causes an interrupt to be taken. An interrupt suspends current program execution and transfers control to a predetermined instruction address, the interrupt vector. The interrupt subsystem of the PP is rather rudimentary. There are four reasons to include it at all for the plasticity application: 1) Interrupts can be used to insert break- and watchpoints into a program for debugging purposes. 2) Timer exceptions allow time based wake-up from sleep states. 3) Receiving messages from the controlling computer system without polling. 4) To detect alignment errors of load/store instructions. The following subset of interrupts is available in the implementation:

1. Alignment: Triggered for mis-aligned load/store accesses.

2. Decrementer: Invoked after the decrement register of the timer facility has reached zero.

3. Doorbell: Logic external to the processor core can signal a doorbell exception, for example to indicate that a message is available in memory.

4. External input: Triggered by external logic to indicate an arbitrary asynchronous event in the system.

5. Fixed-interval: Invoked in fixed time intervals by the timer facility.

6. Program: Handles exceptions directly caused by the program. Only used for trap instructions.

3.1.9.1. Interrupt processing

The interrupt control unit shown in Figure 3.2 coordinates the execution of interrupts. If either internally or externally an exception exists, it is signaled to the control unit. It checks context bits and interrupt priorities, if multiple exceptions are signaled simultaneously, to determine if an interrupt needs to be taken. Apart from alignment and program interrupt, all others are masked by the external enable (`EE`) bit of the machine state register. For decrementer and fixed-interval interrupts there are additional enable bits in the timer control register (`FIE` and `DIE`). The current program counter and machine state register are saved in two 32 bit save and restore registers. The external enable bit is cleared to block further interrupts from external sources. A synchronizing interrupt control transfer is started by asserting appropriate signals to the instruction streamer (Section 3.1.4) and controlling FSM (Section 3.1.6.7). Execution is resumed at a predetermined address, the interrupt vector. The interrupt vector table used by the PP deviates from the specification and is listen in Appendix B on page 228.

When the service routine at the interrupt vector has finished, a return from interrupt instruction is used. It transfers control to the saved program counter and restores the saved machine state register. This will also re-enable the `EE` bit, if it was enabled before the interrupt.

3.1.9.2. Saving the return address

The saved return address depends on the interrupt type. The alignment and program interrupt are synchronous, i.e. the interrupt is caused directly by the execution of an instruction. The restore address saved in register `SRR0` indicates the causing instruction, that is the mis-aligned load or store, or the trap instruction. The other interrupts are asynchronous to program execution. To reduce interrupt latency, the front end will immediately start a control transfer to the interrupt vector. The saved return address is that of the next operation to be issued by the controlling state machine. There are two special cases: if the processor is in a sleep state after a wait instruction, interrupts resume at the wait instruction, sending the core back to sleep. If the service routine decides to resume normal operation, it sets the restore register to a different address before issuing the return from interrupt instruction. If the interrupt is signaled after issue of a branch instruction and before its decision by the branch functional unit, current execution is speculative. In this case, the return address is always set to the branch instruction.

3.1.9.3. Asynchronous interrupts

External exceptions are signalled using a handshake with the interrupt control unit. The source has to keep its request after detecting an exception until the processor acknowledges, that it has taken the associated interrupt. This is necessary for two reasons: first of all, exceptions may occur simultaneously or while external interrupts are disabled (`EE` = 0). The handshake ensures, that interrupts can be taken sequentially in such situations. The second reason is that during context synchronization and wake-up from sleep, especially when the clock is disabled during sleep, some time passes, before the interrupt control unit can initiate the control transfer.

3.1.9.4. Precise interrupt problem

A synchronous interrupt is precise, if all instructions prior to the one causing the exception have completed execution and all following ones have not. This is an important property to be able to resume program execution after an exception at the instruction following the one causing the exception. In a pipelined processor that allows out-of-order retirement, as is the case for the PP, this is potentially difficult to guarantee. If an exception is only detected after a subsequent instruction has retired out of order, the requirements for preciseness are violated. Smith and Pleszkun (1985) describe a number of methods to assure preciseness of interrupts, when a result shift register is used. They propose for example a reorder buffer, that allows to revert results committed out of order.

In the presented design such mechanisms are not needed. Only the alignment and program interrupts are specified as synchronous and precise. For them, exceptions are tested with minimum latency after the operand fetch phase. A subsequent instruction can retire in the next cycle at the earliest. The controlling state machine disables write back globally from this next cycle on, during the synchronizing control transfer (states `SYNC_JUMP_0` and `SYNC_JUMP_1` in Figure 3.10). Therefore, the pipeline runs empty while issue of new instructions is blocked until the transfer to the interrupt vector is complete.

3.1.9.5. Critical and machine check interrupts

Although not available as interrupts, save/restore registers, enable bits in the machine state register, and appropriate return from interrupt instructions (rfmci, rfci) are implemented and can be used. This forms the basis to implement prioritized interrupts in a future system. For example, critical doorbell could be used as high-priority signalling channel that can be reacted upon, even if another interrupt is currently processed. So far, I did not see a practical use for prioritization in the context of the plasticity application.

3.1.10. Timer facility

The timer facility provides a 64 bit time base register that is continuously incremented synchronous to the core clock. The facility is not subjected to clock gating, when the processor is in a sleep state. Therefore, the time base register can be used to measure the duration of a sleep phase. The unit offers two interrupt sources: the fixed-interval timer and the decrementer. The watchdog timer is not implemented. Both sources can be used to wake the PP from sleep states.

The fixed-interval timer sets the fixed-interval timer interrupt status bit (`FIS`) in the timer status register (`TSR`), when a particular bit of the time base transitions from 0 to 1. If enabled, this will also invoke the associated interrupt. The bit can be selected in the timer control register (`TCR`) from position 8, 12, 16 or 20. For a 500 MHz clock, this allows for periods from 1.024 μs to 4 ms.

The decrementer register counts down and sets the decrementer interrupt status bit (`DIS`) in the `TSR`, if decrementing occurs for the value 1. Depending on the configuration, it then assumes value 0 and stops decrementing or it reloads a value from the decrementer auto reload register (`DECAR`). If enabled this triggers an decrementer interrupt to be taken.

3.1.11. General purpose input/output registers

Mainly for debugging purposes the PP has three 32 bit special purpose registers referred to as General purpose Input/Output (GIO) registers. The processor core makes them available to surrounding support logic for simple I/O tasks. A typical example would be the control of pins on a chip. Register GIN is read-only and returns the value presented by support logic. Conversely, the GOUT register is write-only and presents its value to support logic. The GOE register is intended as bitwise output enable register. Support logic can for example use this value to selectively activate tri-state drivers on off-chip pins. It can be read and written. Software accesses these registers using the mtspr and mfspr instructions.

3.2. On-chip bus technology

Section 3.1 describes the inner workings of the PP core. In Section 3.1.3 I already described two interfaces used on the boundary of the core: the RAM interface and the PPB interface. The protocol and the fabric used for implementation of the PPB are the subject of this section. In the framework of the AHM, the task of the PPB is to interconnect the plasticity processor with peripherals and main memory, and to provide access to the synapse array interface adapter for the external control computer system (Figure 3.12).

3.2.1. Motivation and design goals

The PPB is intended as on-chip bus that connects the PP to peripheral devices using a generic interface. On the processor side, it is controlled by the variable-latency load/s-

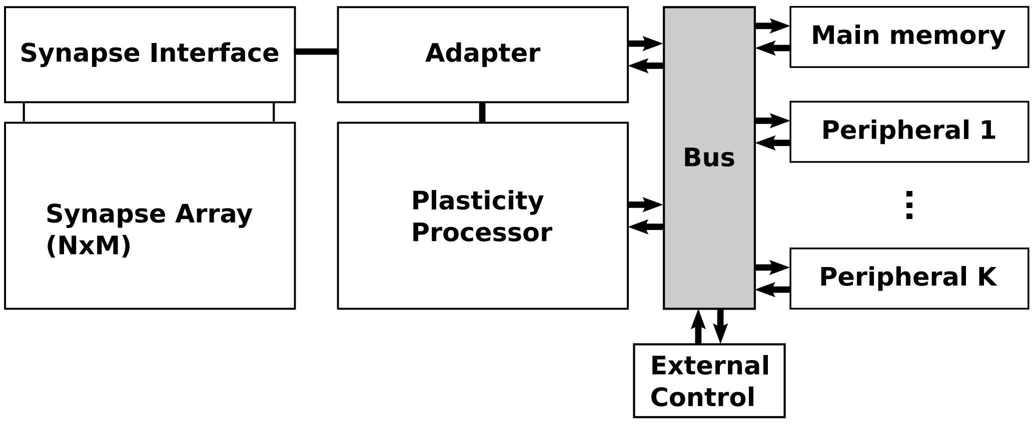


Figure 3.12.: Location of the Plasticity Processor Bus (PPB) in the framework of the AHM.

tore or the external control functional units. Therefore, it allows for memory mapped I/O with peripherals. Originally, the bus was considered for access to the synapse array in the context of the plasticity application. However, benchmarks showed (see Section 5.4), that better performance can be achieved with a specialized instruction set extension using a dedicated I/O interface (described in Section 3.4). It remains the task of the bus to connect to other modules of the neuromorphic device, such as analog parameter storage for neurons, or configuration of event network routing switches. This is important to enable plasticity rules, that not only affect synaptic weights, but for example change the wiring of the neuronal network (structural plasticity). In future systems it is also envisioned to pass messages through the PPB for inter-processor communication. Apart from that, the PPB is used for instruction fetching behind the cache (Section 3.1.5) and data exchange between general purpose registers and main memory (Section 3.1.8.1).

These considerations define the design goals for the bus: The fabric has to offer access to multiple peripheral slaves, of which one is typically main memory. It needs to bridge physically long distances by inserting delay flip-flops. This is mostly motivated by the floorplan of the HICANN chip (Section 5.4), which leads to connections of multiple millimeter between master and slave. In such situations, pipelined operation should allow for good throughput, even though latency is high. Pipelined operation means, that multiple accesses can be initiated before the first one is completed. As a last goal, it must be possible to initiate bus transfers from multiple masters in parallel. For one thing, this is to arbitrate access between the external controlling computer system and the on-chip processor. For another thing, to arbitrate between instruction fetching (by the cache) and load/store accesses.

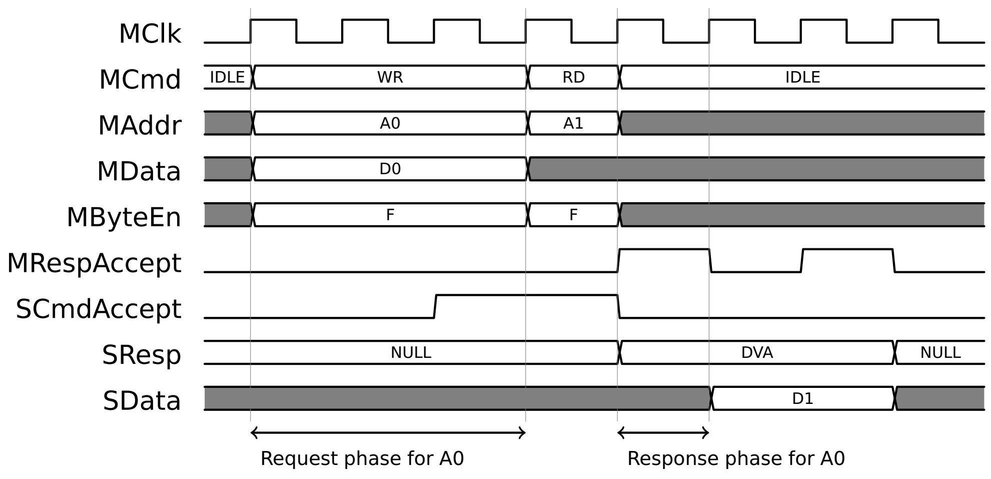


Figure 3.13.: Timing diagram of two transfers on the PPB: A write operation to address A0 and a read operation on address A1.

3.2.2. Interface specification

The PPB fabric connects a number of bus masters initiating transfers with multiple bus slaves. The fabric directs transfers to the correct slave based on the address of the transfer. The interface follows the OCP specification (OCP, 2009). Used configuration options are listed in Appendix C on page 233. It supports read and write operations in a 32 bit address space with 32 bit data words in big-endian byte order. The request and response phase each use a handshake to control the rate of requests and responses. Writes also have a response phase, i.e. they need to be acknowledged from the slave. Data in transfers can be enabled byte-wise.

Figure 3.13 shows a write and a read transfer by an example. All signals starting with "M" are driven by the master and all signals driven by the slave start with an "S". The request phase of the write transfer is initiated by the master by presenting the command `WR` on MCmd. The slave accepts the command, when it is ready by asserting SCmdAccept. This ends the request phase and the master can initiate the next request. The slave initiates the response phase by presenting `DVA` on SResp. The response phase can start in the same clock cycle as the request phase at the earliest for a 0-latency transfer. This is the case for slaves generating the response combinatorially from the request. The response phase can start an arbitrary number of cycles after the request phase starts or ends. There can be multiple completed request phases, before the first response phase is started. The master completes the response phase by asserting the MRespAccept signal. As can be seen in the diagram, the first write causes a response with SResp = `DVA`. The contents of SData in this response is ignored. Refer to the OCP specification (OCP, 2009) for further details.

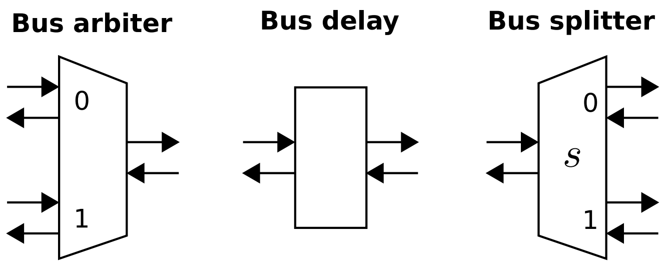


Figure 3.14.: Bus fabric for the PPB is implemented by combining three basic building blocks. Splitter and arbiter forward requests and responses combinatorially. The delay block inserts a delay pipelining stage into the datapath. This allows to shorten logic paths to improve timing.

3.2.3. Basic bus fabric building blocks

From the perspective of the bus fabric there are two independent data channels. The request channel transports requests from the master to the slave and the response channel transports data in the opposite direction. The SCmdAccept and MRespAccept signals serve as handshake signals to exert back pressure on the sending side, i.e., they control how fast the other side can send requests or responses. Fabric needs to assure correct ordering of requests and responses for each master and slave. Every master must see responses in the same order as requests were sent out. Every slave must see requests originating from the same master in the same order as they were initiated. The slave has to respond in the order it receives its requests. The relative ordering of requests between multiple masters is undefined.

These considerations guide the design of three basic building blocks for bus fabric shown in Figure 3.14. They all use the PPB interface to connect to each other.

1. Bus arbiter: Arbitrates between two masters with priority given to master 0.

2. Bus delay: Inserts one cycle delay in the request and response channel, to allow for pipelining of requests and responses.

3. Bus splitter: Diverts requests to slave 1, if bit *s* in the address is set, and slave 0 otherwise.

This allows to build arbitrary bus structures connecting any number of masters with any number of slaves. Some structures are shown in Chapter 5 for systems using the PPB.

3.2.3.1. Bus arbiter

Since transfers are not tagged with the identity of the master they were initiated by, the arbiter records which master initiated a request in a FIFO queue. The response is then forwarded to the correct master depending on the output of the queue, removing the last item upon completion of a response phase. During a request phase the arbiter is locked, meaning that the request phase has to finish, before the other master can initiate a request. When both masters request simultaneously, the one with higher priority gets the lock. If the lower priority master typically has long request phases, this will reduce access bandwidth for the high priority one. The MReset_n signal to the slave is either driven only by master 0 or is deasserted if one or the other master deasserts MReset_n. The length of the FIFO queue determines the number of simultaneously open transfers. If the queue is full SCmdAccept is held low to both masters until a response phase completes and the FIFO can track the next transfer.

3.2.3.2. Bus delay

The purpose of the delay element is to break long combinatorial logic paths between master and slave that limit the maximum clock frequency. Request and response channels are treated completely independent: For each direction, there is a two entry deep FIFO queue. While up to one entry is in the queue, incoming transmissions are accepted. If the slave downstream of the delay module signals, that it can not accept further requests, the delay module will signal this to the upstream master in the next cycle. Therefore, it needs to be able to buffer one request to compensate the incurred latency. The same is true for the response data path.

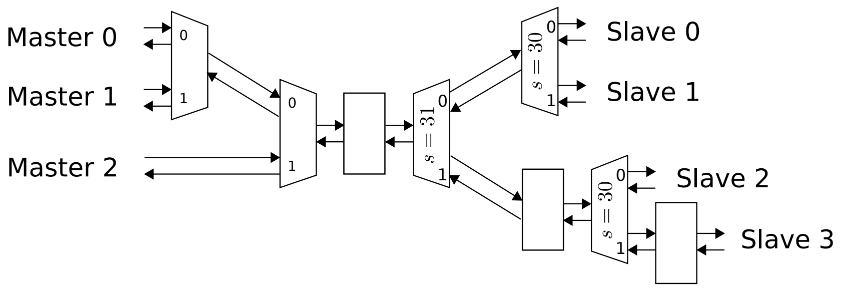
3.2.3.3. Bus splitter

The splitter is in a way a reversed version of the arbiter. Requests are forwarded to one of two outputs depending on the bit at position *s* in the address. The origin of the request is recorded in a FIFO queue. Responses are forwarded to the master indicated by the last element from the queue. The depth of the queue limits the number of open transfers to both slaves of the splitter. Signals are passed without delay flip-flops from one side to the other, thereby forming timing paths between master and slave.

3.2.3.4. Example bus configuration

Figure 3.15A shows an example bus network connecting three masters with four slaves. Figure 3.15B shows the timing diagram for a read transfer from master 0 to slave 3. The arbiter computes the SCmdAccept signal combinatorially from MCmd, SCmdAccept of the slave and the full flag of the tracking queue. It is therefore only asserted while the command is presented. The request passes through the three delay stages and is

A



B

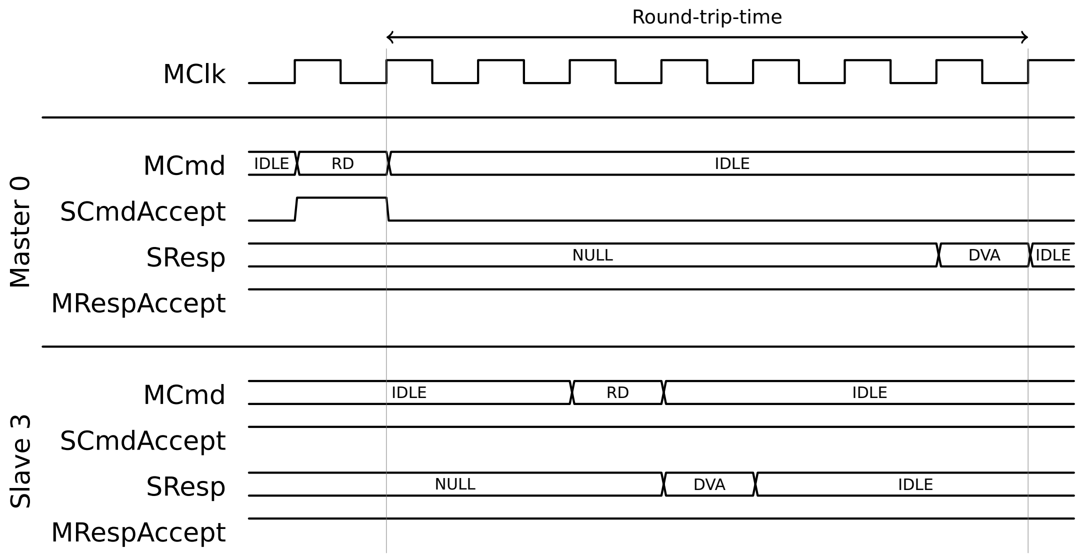


Figure 3.15.: (A) An example use of the building blocks shown in Figure 3.14. (B) Timing diagram for transfers in the example bus fabric. Only the handshake signals are shown.

then presented to the interface of the slave. The configuration of the splitters specifies the address space of slave 3 to 0xC0000000…0xFFFFFFFF[2]. The slave presents the response in the next cycle. It appears back at the interface of the master after three cycles.

3.2.4. Additional bus building blocks

The basic building blocks from Section 3.2.3 are sufficient to build arbitrary bus networks. For optimization and convenience there are additional generic components.

3.2.4.1. Register target

The register target bus component provides read and write access via the PPB for a configurable number of registers. A write mask allows for selection of writable registers. A user input can be used to override the register contents returned by bus read operations. The module selects a register based on the presented address using a base mask M, a base address A, and an offset mask O. A transfer referencing address a accesses register $n = a \,\&\, O$, if $a \,\&\, M = A$[3].

3.2.4.2. Serializer/Deserializer

The Serializer/Deserializer (SerDes) bus component performs transfers serially over a reduced number of lines. It was motivated by the need to connect bus endpoints through a routing bottleneck on the HICANN chip. The block operates independently for request and response channel. The signals belonging to request or response are combined into a parallel word. This word is registered and transferred sequentially over a configurable number N_{serdes} of data lines to a deserializer on the receiving side. It reconstructs the parallel word and presents it to the receiving interface. Until the parallel word has been completely transferred to the other side, the accept signal is kept low. Changing the width N_{serdes} of the serial line offers a trade-off between bandwidth and required routing resources. Serializer and deserializer communicate by three signals: MReset_n is propagated to the slave, a strobe signal, and the serial data line. Both blocks share the same clock.

3.2.4.3. RAM interface adapter

This adapter is designed to access slaves exposing a RAM interface (Section 3.1.3) from the PPB. The delay signal on the RAM interface is not supported and must remain low. Incoming requests are processed by a state machine that immediately

[2]Numbers beginning with "0x" are given in hexadecimal representation
[3]& denotes bit-wise and

acknowledges the request by initiating a response phase. Address, data, and byte enables are forwarded asynchronously and enable and write enable are generated from the MCmd field. If the response is not immediately accepted (MRespAccept = 0), the request and the returned data are registered. New requests are not accepted (SCmdAccept = 0) until the response phase of the previous transfer is completed.

3.2.4.4. HICANN system bus adapter

The HICANN ASIC also uses an OCP based bus to interconnect system components. However, it is configured with incompatible options, making the adapter module necessary. For example, it does not have the MRespAccept signal and writes do not cause a response. The employed bus fabric terminates request and response phase together by asserting SCmdAccept and, for reads, SResp = `DVA`. This blocks subsequent transfers and prevents pipelined operation. Therefore, the adapter needs to decouple both buses, so that requests can be pipelined in the PPB.

The adapter from PPB to HICANN bus uses FIFO queues to track requests and responses. Incoming requests are pushed to the request queue and presented to the HICANN bus from its tail. The returned data is pushed to the response queue. From its tail, responses are initiate to the PPB. The adapter for the opposite direction is simpler: a state machine goes through the request and response phase on the PPB and returns the result to the HICANN bus, when the transfer is complete.

3.2.5. Methodology: using code generation for bus specification

Specifying a bus network in the RTL description results in lengthy and repetitive code. For every node and connection a SystemVerilog module or interface is instantiated. The structure of the described network is not directly obvious from the source code. Similar to the approach taken in Section 3.1.6.5, code generation using the M4 language can be used to allow for a more concise description. Listing 3.3 shows a description of the network illustrated in Figure 3.15. The network description is surrounded by the `bus_begin` and `bus_end` macros. The former sets a prefix "testbus" for the network and specifies the used clock. Connectivity is determined using a stack: every macro takes its inputs from the top of the stack and pushes its output onto it. The `master` macro serves as entry point to the network that does not remove an input. Its argument is pushed onto the stack. When the first `arb` macro is encountered in line 5, the stack holds two inputs. They are removed by `arb` and replaced with its arbitrated output. The `delay` macro consumes and produces one entry on the stack and `split` takes one, pushing two. With `slave` an output is added to the network removing one entry from the stack. It uses its argument to define a macro that can be used by user code to refer to the output SystemVerilog interface. The `slave(1)` macro in line 10 for example defines the macro `testbus_slave_1` that expands to the name of the interface for

Listing 3.3: Example M4 code to describe the bus network shown in Figure 3.15

```
include(bus.m4)
...
bus_begin(testbus, clk)
  master(bus_master_0)
  master(bus_master_1) arb
    master(bus_master_2) arb delay
      split(31)
        split(30)
          slave(0)
          slave(1)
        delay split(30)
            slave(2)
            delay slave(3)
bus_end()
```

this slave.

The stack is implemented with the `pushdef` and `popdef` operators of M4 that treat macro definitions as a stack. Names for intermediate nodes and interfaces are generated automatically by use of the bus prefix and a counter variable set by `bus_begin`. Interfaces are automatically instantiated as needed and connected to the specified clock.

In contrast to the use of M4 for the pre-decode logic (Section 3.1.6.5), Listing 3.3 shows much more elaborate code generation. The result is a short and easily readable description of the bus network that translates to SystemVerilog code. The use of stack processing goes beyond what is possible with the built-in SystenVerilog `` `define`` macros.

3.3. STDP logic in the BrainScaleS wafer-scale system

The previous sections in this chapter described generic hardware designs for a general purpose micro-processor and an associated on-chip bus. This section introduces specialized logic for the plasticity application. First the interface to the synapse array in the HICANN system is defined in Section 3.3.1. Section 3.3.2 describes a non-programmable STDP implementation using this interface. This implementation was designed as a precursor design for a processor based approach and is used in the current generation of the wafer-scale neuromorphic computing system. Section 3.3.3 introduces the adapter between processor and synapse array interface.

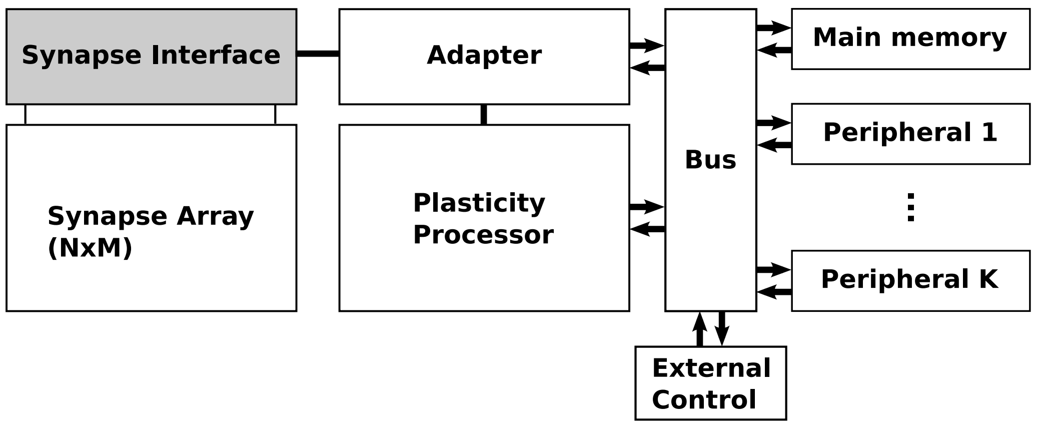


Figure 3.16.: The synapse interface in the framework of the AHM.

3.3.1. HICANN synapse array interface

The location of the synapse array interface within the AHM is highlighted in Figure 3.16. It provides access to synaptic weights and control of the evaluation process (Section 2.2.2.4). In HICANN the synapse also stores an individual address to determine if it should forward a presynaptic event. These addresses are referred to as "decoder addresses". Also, reading and writing configuration data for the synapse line drivers that inject presynaptic events from the on-wafer event network into the synapse array is intertwined with synapse weight readout. Therefore, the interface has to perform read/write accesses to synapse weights and decoder addresses, read/write on synapse line driver configuration SRAM, and control of the evaluation process including accumulator reset. The interface is asynchronous and so all changes on signal inputs take immediate effect. This has to be kept in mind when designing the logic to drive the interface. The design of the synapses, synapse line drivers, and the array was not part of this thesis.

3.3.1.1. Structural description

Figure 3.17 shows the organization of the synapse array. The shown signal names belong to the synapse array interface and are also listed with their polarity and width in Appendix C on page 233.

The synapse driver SRAM consists of three regions, that can be selected using the syn_endrvb, syn_enctrlb, and syn_engmaxb enable signals. The bitlines are exposed as syn_d and syn_db. Wordlines are driven from an address decoder shared with the synapse array. The address is presented to syn_a and in inverted form to syn_ab. Two rows share the same address and the 2 bit wide syn_en signal is used to disambiguate them. The bits enable the synapse line driver on the left or right side of the array and

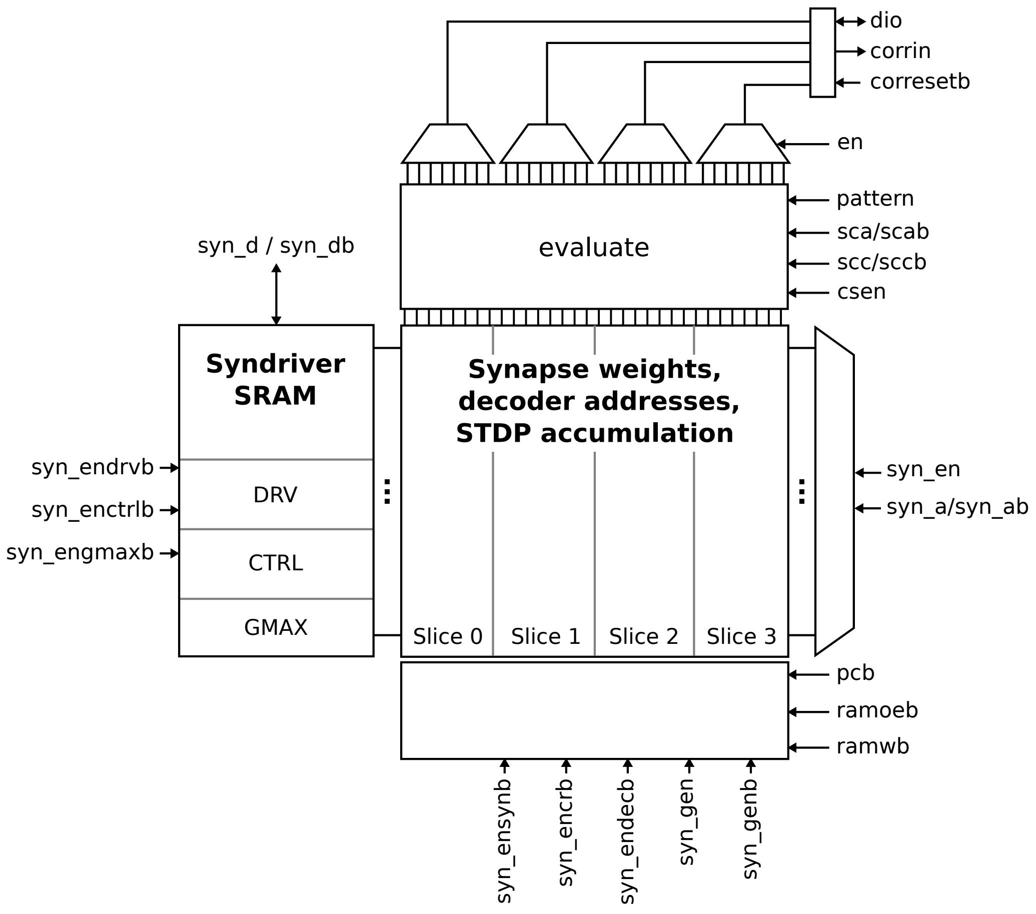


Figure 3.17.: A block diagram showing the organization of the synapse array and its interface signals.

thereby determine which synapse row is addressed. Each side has eight bitlines that are concatenated in syn_d and syn_db.

The main array of synapses is separated into four slices of 64 columns each. The whole array consists of 224 rows with 256 columns. Every synapse has a 4 bit weight, 4 bit decoder address, and two analog accumulation capacitors for STDP correlation measurement. Enables syn_ensynb, syn_endecb, and syn_encrb activate weight, decoder, or correlation readout. Synapse and decoder data is read out via four bitlines per column. All 256 bit for one slice are multiplexed to a 32 bit bus. These four buses are concatenated to the 128 bit dio bus. The multiplexer is configured by the 32 bit en signal that contains eight one-hot coded bits for each slice (multiplexing 64×4 bit to 8×4 bit for each slice). The data direction of the dio bus is controlled by the ramoeb and ramwb signals. For ramwb = 0, dio is treated as input and its value is driven to the synapse SRAM for the selected columns. For ramoeb = 0, dio is an output and driven with the value from the selected bitlines. If both ramwb and ramoeb are high, dio is in a high-impedance state. The pcb signal triggers pre-charge of the bitlines.

3.3.1.2. Analog evaluation

For the evaluation of the local accumulation capacitors, the evaluate block is used. The syn_encrb signal triggers readout of the capacitors to a temporary storage location in the evaluation unit. The sca/scab pair of control signals for one and scc/sccb for the other capacitor control whether both, one, or none of them are stored. The 4 bit pattern control the evaluation operation. With csen a digital bit is generated from the analog evaluation result and presented on the corrin bus, which is multiplexed as for the synapse bitlines. A low-active reset bus corresetb is split to the selected columns to reset both accumulation capacitors.

The evaluation block in Figure 3.17 implements a generic evaluation function E^H (see Section 2.2.2.4 for the definition of evaluation function). It is configured by an evaluation pattern p with bits $p = (e_{aa}, e_{ac}, e_{ca}, e_{cc})$ that control the evaluation operation. Two analog parameter voltages V_{th},V_{tl} are provided from the global parameter storage. The accumulation trace $a(t)$ in terms of the AHM is represented using the dual capacitor value with a positive "causal" capacitor with value $a_+(t)$ and a negative "acausal" one with value $a_-(t)$. The combined trace is then given by $a(t) = a_+(t) - a_-(t)$. The temporal storage locations V_c, V_a in the evaluation block are set depending on whether scc or sca is asserted:

$$V_c \leftarrow a_+(t) \quad \text{if scc} = 1 \wedge \text{sccb} = 0 \tag{3.1}$$

$$V_a \leftarrow a_-(t) \quad \text{if sca} = 1 \wedge \text{scab} = 0 \tag{3.2}$$

If neither scc nor sca is set, V_c and V_a remain unchanged. The evaluation function is described by

$$E^H\left(V_{tl}, V_{th}, p, V_c, V_a\right) = \begin{cases} 1 & \text{if } \frac{V_{tl}+e_{ac}V_c+e_{ca}V_a}{1+e_{ac}+e_{ca}} > \frac{V_{th}+e_{cc}V_c+e_{aa}V_a}{1+e_{cc}+e_{aa}} \\ 0 & \text{else.} \end{cases} \tag{3.3}$$

The *null read* pattern $p_{\text{null}} = (0, 0, 0, 0)$ sets all evaluation bits to zero and therefore performs the following readout operation

$$E^H\left(V_{tl}, V_{th}, p_{\text{null}}\right) = \begin{cases} 1 & \text{if } V_{tl} > V_{th} \\ 0 & \text{else} \end{cases} \tag{3.4}$$

This is a useful test-pattern, since it allows to characterize the evaluation circuit by configuring different values for V_{th} and V_{tl}. It also allows for deterministic generation of evaluation result bits to test weight update logic. The *absolute threshold* patterns

$$p^+_{\text{abs}} = (e_{aa} = 0, e_{ac} = 1, e_{ca} = 0, e_{cc} = 0) \tag{3.5}$$
$$p^-_{\text{abs}} = (e_{aa} = 0, e_{ac} = 0, e_{ca} = 1, e_{cc} = 0) \tag{3.6}$$

compare to a threshold $\Theta = 2V_{th} - V_{tl}$:

$$E^H\left(\Theta, p^+_{\text{abs}}, V_c\right) = \begin{cases} 1 & \text{if } V_c > \Theta \\ 0 & \text{else} \end{cases} \tag{3.7}$$

$$E^H\left(\Theta, p^-_{\text{abs}}, V_a\right) = \begin{cases} 1 & \text{if } V_a > \Theta \\ 0 & \text{else} \end{cases} \tag{3.8}$$

The *relative threshold* patterns

$$p^+_{\text{rel}} = (e_{aa} = 1, e_{ac} = 1, e_{ca} = 0, e_{cc} = 0) \tag{3.9}$$
$$p^-_{\text{rel}} = (e_{aa} = 0, e_{ac} = 0, e_{ca} = 1, e_{cc} = 1) \tag{3.10}$$

compare the difference of the temporal storage capacitors to Θ:

$$E^H\left(\Theta, p^+_{\text{rel}}, V_c, V_a\right) = \begin{cases} 1 & \text{if } V_c - V_a > \Theta \\ 0 & \text{else} \end{cases} \tag{3.11}$$

$$E^H\left(\Theta, p^-_{\text{rel}}, V_c, V_a\right) = \begin{cases} 1 & \text{if } V_a - V_c > \Theta \\ 0 & \text{else} \end{cases} \tag{3.12}$$

3.3.1.3. Control sequences on synapses

To perform accesses on synapses, interface signals have to be set appropriately in the correct order. Since the interface is asynchronous and weights are stored in a physically large SRAM array, special care has to be taken with the timing of signals. Two control units generating the sequences presented in this section are explained in Sections 3.3.2 and 3.3.3.

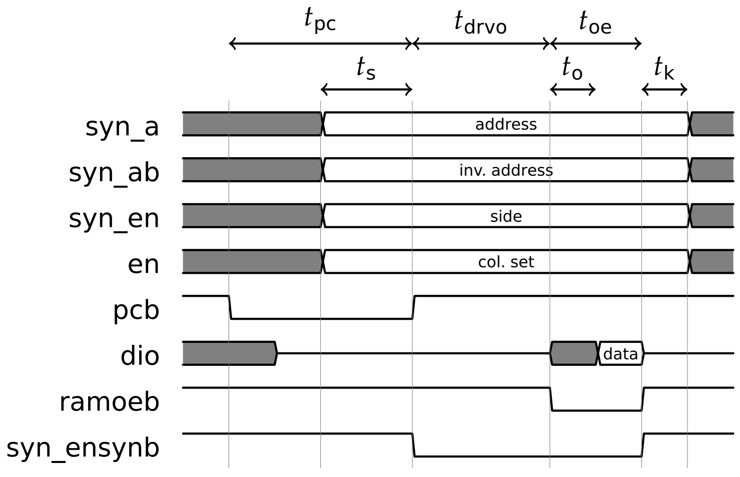


Figure 3.18.: Timing diagram of a synapse array read operation. Signals not shown are at their inactive level.

Reading weight or decoder address Figure 3.18 shows a timing diagram for a read access to a synapse weight. The timing for reading decoder bits is identical, but instead of syn_ensynb the syn_endecb signal is used. Every constraint that applies to syn_ensynb mentioned below also applies to syn_endecb. Synapses are addressed in one 128 bit *column set*. The column set is selected by presenting an address on the syn_a and syn_ab address bus, side selection on syn_en, and an enable pattern on en. A typical enable pattern would be en $= 0x80808080$ to select the first 32 bit from every slice. Each byte of en configures one multiplexer and only one bit per byte may be set.

When syn_ensynb is active, the wordline for synapse weights of the currently selected row is activated. Therefore, the address decoder must have settled before syn_ensynb is active and the address lines must stay stable while it is active. This safety margin between syn_a/syn_ab and syn_ensynb is described by the settling time t_s and the keep time t_k. The SRAM bitlines are pre-charged for a time t_{pc} by activating pcb. Afterwards, the wordline is activated for a time t_{drvo} to drive the stored bit value onto the bitlines. Activating ramoeb for time t_{oe} enables the dio output driver to present the result to user logic. The output is stable after time t_{o}.

Writing weight or decoder address Figure 3.19 shows timing of a write access to a synapse weight. The same timing is used for decoder address writing by exchanging syn_ensynb with syn_endecb. Again, the address decoder settling and keep times t_{s} and t_{k} have to be satisfied. Additionally, after presenting the write data on dio and selecting the demultiplexer configuration with en, the user must wait for a settling time t_{ds} until the demultiplexer has settled. Then, the bitlines are driven for a time t_{drvi}

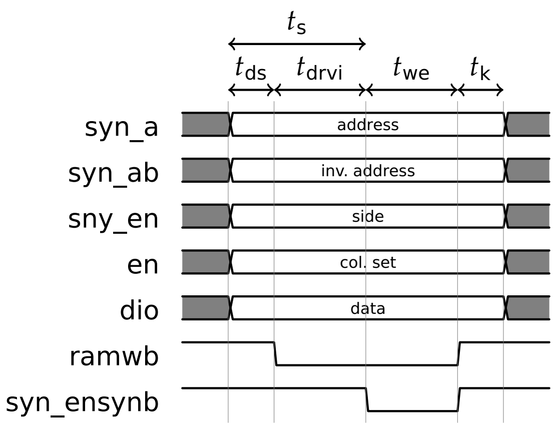


Figure 3.19.: Timing diagram of a synapse array write operation. Signals not shown are at their inactive level.

by activating ramwb. When they have reached a stable state the wordline is activated for the selected row by activating syn_ensynb for duration t_{we}.

Accumulator readout and evaluation The timing diagram for a readout and evaluation sequence of the local accumulators in the synapse is shown in Figure 3.20. Settling and keep times $t_{\mathrm{s}}, t_{\mathrm{k}}$ have to be satisfied between syn_a/syn_ab and syn_-encrb. While syn_encrb is active, the accumulation circuit drives the readout lines for time t_{cro}. The temporal storage capacitors V_{c} and V_{a} are set by activating scc and sca, respectively[4] for time t_{sc}. Asserting the 4 bit evaluation pattern p triggers the evaluation operation as described in Section 3.3.1.2. The pattern may not overlap with activation of sca and scc, which is accounted for by the waiting time t_{scw}. The pattern is kept for time t_{e}. When the analog comparison is finished, activation of csen for a time t_{csen} generates a digital bit from the result and drives it to the corrin port as selected by the multiplexer configuration en. The output is stable after time t_{co}.

Accumulator reset The accumulator reset clears the local capacitors to zero. The corresetb bus controls which of the synapses are reset in a column set. The reset at the accumulation circuit is enabled by the weight word line. Therefore, syn_ensynb must be activated for a reset. This implicates, that accumulation can only be reset during a synapse weight read or write sequence. Figure 3.21 shows timing for reset during a write access. This is the most likely case for the STDP application: After reading weights and evaluating accumulation, new weights are written and the accumulators reset. Two additional timing parameters are relevant: the settling time of the corresetb

[4]Not shown in the figure: sccb and scab are the inverted versions of scc and sca

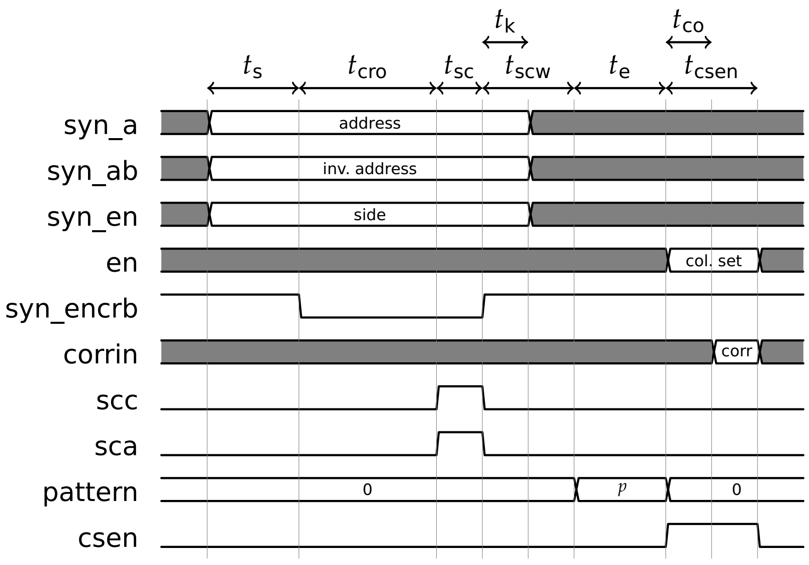


Figure 3.20.: Timing diagram of readout and evaluation of the local accumulation capacitors. Signals not shown are at their inactive level.

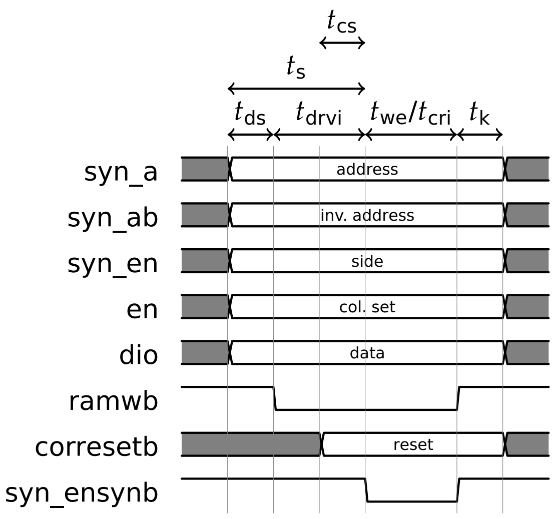


Figure 3.21.: Timing diagram of reset to zero of the local accumulation capacitors. Signals not shown are at their inactive level.

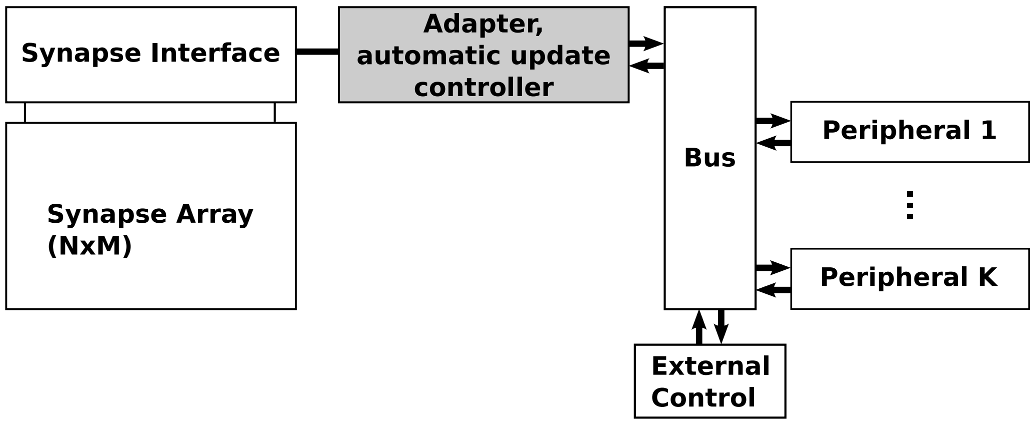


Figure 3.22.: The non-programmable STDP implementation with automatic update controller represents a reduced version of the AHM. There is no processor, no main memory, and the automatic update controller is part of the synapse array interface adapter.

multiplexer t_{cs} and the time needed for reset by the cell t_{cri}. In a combined write/reset cycle, syn_ensynb needs to be pulled down for the larger of the times t_{we} and t_{cri}.

3.3.2. Non-programmable STDP implementation

This section describes the digital part of a non-programmable STDP implementation for the HICANN wafer-scale system. It translates between external requests and the synapse array interface described in Section 3.3.1. Its main function is to perform automatic weight updates to implement STDP in ongoing network operation. The plasticity system is outlined in Figure 3.22 in the context of the AHM. The main focus of this section is the highlighted adapter with the automatic weight update controller. Note, that the bus used here is the HICANN system bus and not the one described in Section 3.2.

3.3.2.1. Functional overview

The user operates the neuromorphic system from a control computer system by performing read and write accesses via the external control interface. These accesses are distributed to the addressed peripheral in one HICANN ASIC by the system bus. Two of these peripherals are the adapters for the two synapse arrays. The user can perform a number of basic control operations: He can read and write synapse weights and decoder addresses, as well as synapse line driver configuration SRAM. He can perform evaluation and reset of accumulation capacitors. And finally, he can configure the automated update process and start and stop it.

The automatic weight update process follows a simple algorithm: For each row it iterates over all column sets reading out synaptic weights. Then, the accumulators are evaluated with two pre-configured patterns p_a and p_b. The resulting correlation bits $b_{a/b} = E^H\left(p_{a/b}\right)$ are used to select a 16×4 bit look-up table $L\left(b_a, b_b, w\right)$. This look-up table holds the new weight w' for every old weight w, so that

$$w' \quad = \quad L\left(b_a, b_b, w\right). \qquad (3.13)$$

For $b_a = b_b = 0$ the weight is kept unchanged, i.e. $L(0, 0, w) = w$. In the design, there is no look-up table for this case, to save the associated flip-flops. To my knowledge, there is no use-case requiring it. The new weight is then written back to the synapses. If enabled by reset configuration, the accumulation is reset to zero for those synapses that had b_a or b_b set. This iteration is repeated for a range of synapse rows either indefinitely or in singleshot mode just once.

3.3.2.2. Design of the bus interconnect

Weights and decoders make up 112 kiB, synapse line driver SRAM 672 byte of configuration data. This is the largest block of configuration data in the system and makes a high-throughput connection desirable. However, it is situated far away from the external control interface on the die, making delay flip-flops in the data path necessary to meet timing requirements. Since the PPB described in Section 3.2 was not yet developed at the time the adapter was designed, the paths to and from it were build as shift-registers. This means that request data from the master travels over a fixed number of cycles to the slave. The slave generates the response in the next cycle and the result appears at the master after a total round-trip-time that is two times the shift-register length. In contrast to the hitherto used system bus interconnect, this allows to send a new request in every cycle instead of having to wait for the complete round-trip-time until SCmdAccept is asserted. However, without buffering and an ability to exert back-pressure, this allows only for single cycle latency operation by the adapter. The notification, that a request takes longer to fulfill would only arrive at the sender after a certain latency. Requests sent during that time would be dropped. An improved solution for a future upgrade is to use the delay component described in Section 3.2.3.2.

To accommodate the constraint of single cycle latency operation, all requests are made to registers in the adapter. An operation taking a longer time is then performed asynchronously and saves results back to local registers. A second transfer is then needed to fetch the result. This transfer can either be timed with the knowledge of the execution time, or otherwise polling needs to determine the end of the operation.

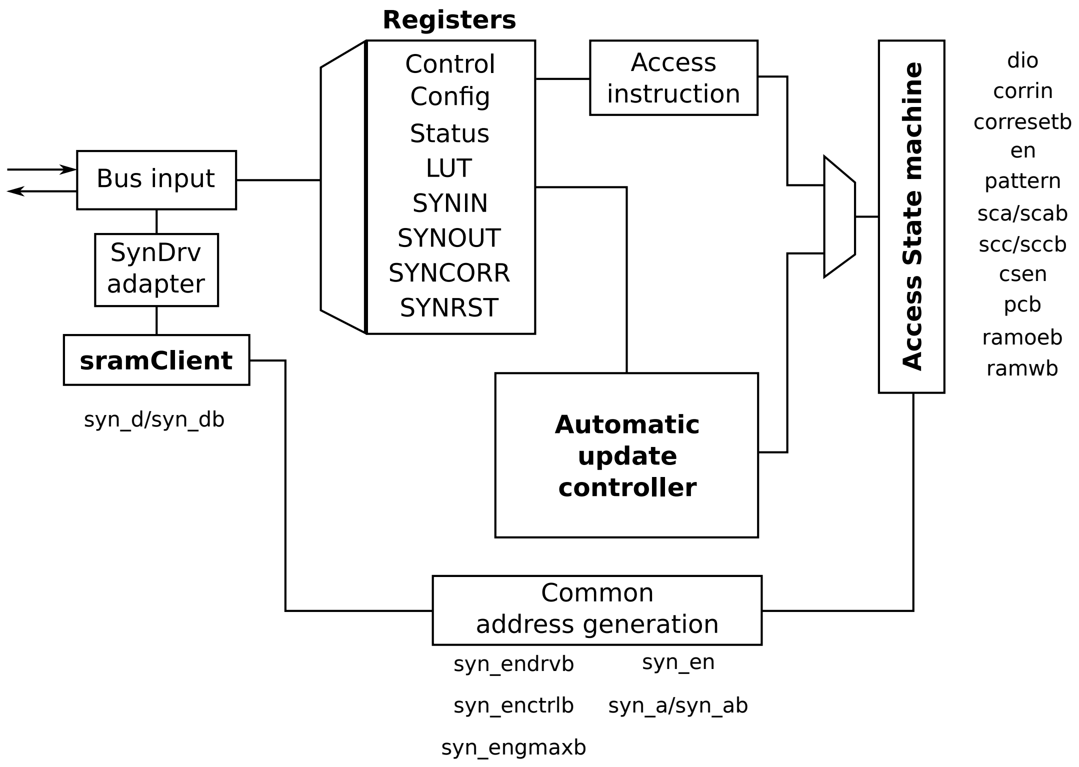


Figure 3.23.: The toplevel structure of the non-programmable synapse array adapter. The figure shows all synapse array interface signals (Figure 3.17) and from which part of the design they are driven.

Name	Descriptions
`idle`	Do nothing
`start_read`	Open row for weight read accesses
`read`	Read weights from the selected location
`write`	Write weights to selected location
`rst_corr`	Reset local accumulation for selected location
`start_rdec`	Open row for decoder address read access
`rdec`	Read decoder addresses from selected location
`wdec`	Write decoder addresses to selected location
`close_row`	Close a row previously opened
`auto`	Enable the automatic update controller

Table 3.3.: Command descriptions of the non-programmable synapse array adapter.

3.3.2.3. Structure of the design

Figure 3.23 shows the internal organization of the non-programmable synapse array adapter. Read and write requests from the HICANN system bus either access local registers or are passed to an adapter for synapse driver access. The latter serves as proxy for the SRAM controller module sramClient developed by Schemmel (2011). The sramClient module together with the common address generation performs accesses on the synapse driver SRAM memory. The address generation sets the row address and enable signals as determined by the address of the bus request for synapse driver accesses. Otherwise, the access state machine controls addressing. The contents of the control register determines if and what type of access operation is performed by the access state machine. The config register allows for setting timing parameters for the synapse array interface access. The status register indicates whether an operation is currently ongoing. The LUT register hold the look-up tables $L(b_a, b_b, w)$ (Section 3.3.1.2). The remaining registers are used for input and output of data to and from the array via the access state machine.

The contents of the control register is condensed into an access instruction consisting of an command code, a target row, and a target column set. The instruction is presented as valid to the state machine, if a certain bit in the control register is written to one. Commands are listed in Table 3.3. For the `auto` command, control of the access state machine is handed over to the auto update controller. It executes the described weight updating algorithm by issuing access instructions to the state machine and processing the data registers. A full description of the user interface in terms of address space layout and bit-positions within the registers is given by Schemmel et al. (2012).

3.3.2.4. Operation of the access state machine

The commands listed in Table 3.3, except for `idle` and `auto`, translate directly into the control sequences described in Section 3.3.1.3. The state machine has to assure, that all signals are applied in the correct order and respecting the timing constraints. Figure 3.24 shows the simplified state diagram of the access state machine. States and transitions associated with decoder address access are not shown, since they are identical to the weight read and write states. After reset, the FSM is in the IDLE state keeping all control signals to the array interface in their inactive state. A valid access instruction will initiate a control sequence by transitioning into the corresponding state. To control timing, the state machine employs a counter that is used by some of the states.

To read data from the SRAM array bitlines have to first be pre-charged and then cells have to drive their internal state onto them. This requires the time $t_{\mathrm{pc}} + t_{\mathrm{drvo}}$. This happens for all columns in parallel, but only one eighth of them can be de-multiplexed to the data I/O lines. Therefore, a row is kept "open" after pre-charge and enable phase and the eight column sets can be read out successively requiring only the output time t_{o}.

For the STDP application, weights and correlation are typically read together. Therefore, there is a common readout sequence and only a bit in the control register (encr) controls, whether accumulators are evaluated in a read sequence or not.

Reading weights and decoder addresses The required timing of control signals is shown in Figure 3.18. If the access instruction holds the command `read` or `start_-read` while the FSM is in the IDLE state, the row is being opened by transitioning into the READ_WEIGHT_PRE state. In this state, the bitlines are pre-charged (pcb = 0) for c_{predel} clock cycles. When the counter matches the configured number of cycles, a transition to state READ_WEIGHT_EN deactivates pre-charge and activates the enable signal syn_ensynb. The state is held for a configurable number of clock cycles (parameter c_{endel}) determining time t_{drvo}. After that, the FSM assumes state ROW_SELECTED_WEIGHT. In this state the row is held "open" by keeping the enable signals active and thereby the word line enabled.

Starting from this state, reads of all column sets in the opened row can be performed directly without going through the lengthy pre-charge and enable sequence. A `read` command will cause a transition to READ_WEIGHT_SAMPLE. This state is held for a number of cycles configured by parameter c_{oedel}. It enables the output drivers to the data I/O bus and registers the result in the SYNOUT register. Afterwards it transitions back to the ROW_SELECTED_WEIGHT state.

Reading of decoder addresses is symmetric: A second set of states is used that activate syn_endecb instead of syn_ensynb. Results are registered to the SYNCORR register.

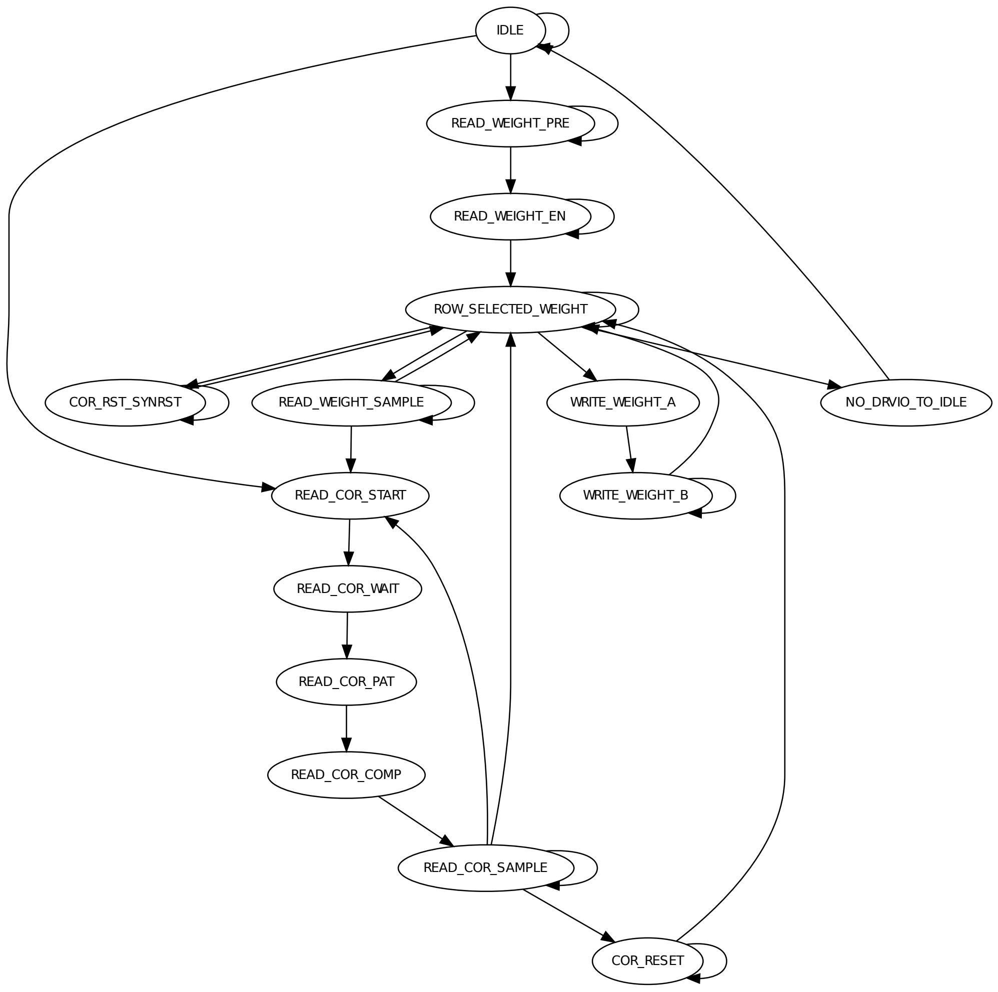


Figure 3.24.: State diagram of the STDP access state machine performing control sequences on the synapse array interface. Transition conditions and outputs are described in the main text. States for reading and writing of decoder addresses are not shown for clarity. They mirror behavior for weight access (states with "WEIGHT" in the name).

Reading and evaluating STDP accumulation To evaluate the STDP accumulation capacitors of the synapses a standard read sequence is started with the encr bit in the control register set. This will cause a transition from READ_WEIGHT_SAMPLE to READ_COR_START instead of back to ROW_SELECTED_WEIGHT. The readout and evaluation is then a sequence with fixed timing: READ_COR_START activated the scc and sca signals to load the temporal storage locations V_c and V_a in the evaluation unit. A wait cycle ensures, that t_{scw} is positive and loading is not overlapped with the application of the evaluation pattern. READ_COR_PAT presents an evaluation pattern that is stored in the configuration register. READ_COR_COMP activates csen and finally READ_COR_SAMPLE registers the result in the SYNCORR register while holding csen active. The READ_COR_SAMPLE remains active for $c_{\text{oedel}} + 1$ clock cycles. The read performs always two evaluations of the accumulators for the two patterns stored in the configuration register. A counter is used, that is incremented after the pattern is applied. This counter is used to select a pattern in the configuration register and to select one half of the SYNCORR register as write destination. After the first evaluation the FSM loops from READ_COR_SAMPLE back to READ_COR_START for the second evaluation. If the without_reset bit is not set in the control register, the COR_RESET state follows after the second evaluation sequence. It resets accumulators for which one of the patterns returned one as indicated in the SYNCORR register. Finally, the state machine returns to ROW_SELECTED_WEIGHT.

Writing weights and decoder addresses Writing involves two states: In the first one, the adapter begins to drive the dio port with the data to write. In the second state, the bitline write drivers are enabled for c_{wrdel} cycles.

Manual reset of accumulators For the normal STDP algorithm, reset occurs automatically after the evaluation operation depending on the evaluated bits b_a, b_b. To selectively perform a reset of synapses indicated in the SYNRST register, the `rst_corr` command is used. The single state COR_RST_SYNRST is used to apply a reset pattern on corresetb for $c_{\text{rst}} = 2$ cycles.

Closing the row If a command other than `read`, `write`, or `rst_corr` is received while in state ROW_SELECTED_WEIGHT, the row is "closed" by transitioning over NO_DRVIO_TO_IDLE back to the IDLE state. The intermediate state is used to ensure the time t_{k} is positive. It deactivates the syn_ensynb and, if active, syn_encrb enable signals.

3.3.2.5. Automatic weight update controller

The automatic weight update controller performs the algorithm outlined in Section 3.3.2.1. It is started by writing an `auto` command to the control register. The controller then uses the access state machine to read and write weights and evaluate and reset accumulators.

Micro-programmed control The automatic update controller is micro-programmed: A read-only memory holds words of control signals that send instructions to the access FSM. Conditional branches are supported for looping. Listing 3.4 shows an excerpt from the SystemVerilog code describing the seven line long micro-program. Every line corresponds to one control word in the read-only memory. The columns are: Branch condition, branch target, continue condition, instruction for the access state machine, operation for the row address pointer, and two bits to trigger weight update computation (`2'b10`) and increment the column set pointer (`2'b01`). The continue condition is evaluated after the current operation, i.e. `cont_not_busy` means continue with the next control word after the access instruction given in this word is complete.

Listing 3.4: Micro-program of the automatic update controller.

```
1 { jump_none,  5'h0,cont_not_busy, DSC_START_READ,row_hold,2'b00 }
2 { jump_none,  5'h0,cont_not_busy, DSC_READ,      row_hold,2'b00 }
3 { jump_none,  5'h0,cont_done,     DSC_IDLE,      row_hold,2'b10 }
4 { jump_none,  5'h0,cont_not_busy, DSC_WRITE,     row_hold,2'b00 }
5 { jump_row_ip,5'h1,cont_immediate,DSC_IDLE,      row_hold,2'b01 }
6 { jump_none,  5'h0,cont_not_busy, DSC_CLOSE_ROW, row_hold,2'b00 }
7 { jump,       5'h0,cont_immediate,DSC_IDLE,      row_inc, 2'b00 }
```

In the first line, a `start_read` instruction is sent to the access FSM to open the row currently addressed by the row pointer. The `cont_not_busy` code causes the sequencer to proceed to the next word, when the busy signal goes low again, i.e. the row is open. Line 2 then fetches weights using a `read` command. After that, weight update computation is performed by setting the first bit in the last field. The `cont_-done` code causes continuation after all weights have been updated and the results stored in the SYNIN register. Line 4 writes the new weights back to synapse array. The next operation does two things: it increments the column set pointer by rotating the pattern presented on the en signal of the synapse array interface by one position to the right. The conditional branch `jump_row_ip` branches to line 2 while the column set pointer does not wrap around to the first position. Therefore, lines 2 to 5 represent a loop iterating over all column sets in one row. When a row is completely processed, it is closed by line 6 and finally line 7 increments the row pointer before looping back to the start of the program.

Execution of the program is stopped in two cases: if the controller is configured for single shot mode (continuous = 0 in the control register) and there is a `row_inc` on the last row of the enabled range. Or, if a different command than `auto` is written to

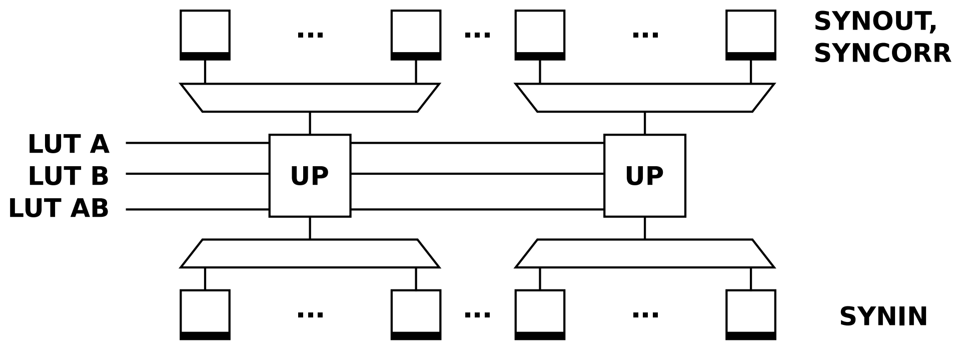


Figure 3.25.: The data path used by the automatic weight update controller. Weights are computed in parallel by look-up table based updating units (UP). In the produced BrainScaleS wafer-scale system there are eight synapses per update unit.

the control register. The enabled range of rows is configured in the control register.

Weight update datapath Figure 3.25 shows how new weights are computed by the automatic update controller. Result data from the `read` command is multiplexed to an updater that selects the correct table to use and performs the look-up of the new weight. The number of update units per synapses is compile-time configurable. At least one has to be implemented per synapse array slice. This gives a range of one to eight cycles for weight computation. In the produced single-chip and wafer-scale systems this parameter is set for eight cycle updates. Update is started by a control word of the micro-program that has the update bit set. When all synapses have been updated the micro-program sequencer is signalled. The control word can wait for this event by using the `cont_done` condition for continuation.

3.3.3. Synapse array interface adapter for programmable STDP

For the programmable plasticity implementation an improved synapse array interface adapter was developed. Figure 3.26 highlights the location of the adapter in the AHM. With plasticity processor in the system, the adapter has to perform accesses to synapses from external control, as well as the internal processor. Additionally, as I will show in Chapter 5, a specialized adapter allows for better performance of the weight update algorithm running on the PP. There are two central ideas to improve performance: first, using a wider 128 bit I/O datapath, so that the processor can calculate weight updates on a larger number of synapses in parallel. Second, a programmable sequencer instead of the access FSM described in Section 3.3.2.4 to allow for fine tuning of the control sequences presented to the synapse array interface.

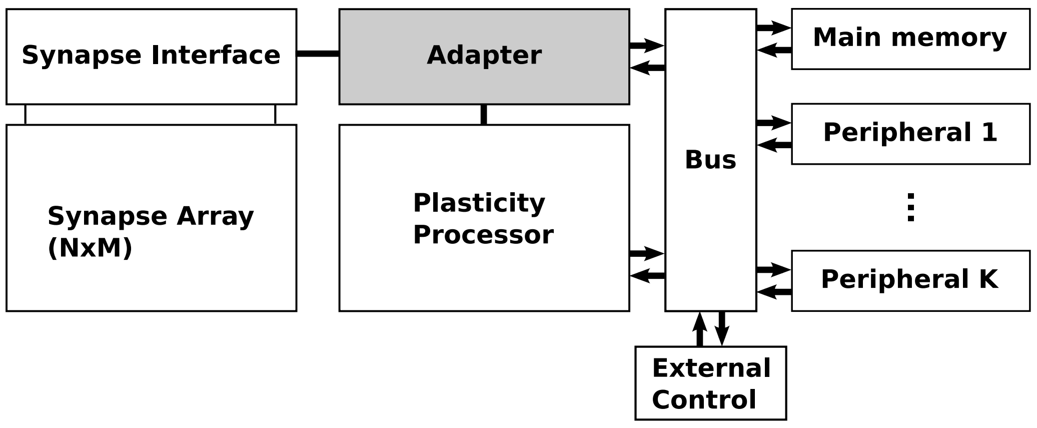


Figure 3.26.: Location of the synapse array adapter for programmable STDP in the AHM.

Wide I/O bus The synapse array interface has a 128 bit wide, bi-directional data port dio (Figure 3.17). This is the maximum amount of data that can be read and written at once. Therefore, it makes sense to use an equally sized bus to the processor, if the processor features a special-function unit to perform weight updates in parallel for this amount of data. In Section 3.4 such a special-function unit is introduced. This I/O bus is used to transfer weight and decoder addresses, as well as evaluation readout data. While the dio port provides 128 bit of weights and decoder addresses in one read access, there are only 32 bit of evaluation results per access. Typically, an STDP program will perform multiple evaluations with different patterns and use the combined information to compute the new weight. To offload bit-manipulation operations from software, results from up to four evaluations are combined into one 128 bit vector in an interspersed manner. This way, each half-byte in the vector holds the results from four evaluations with different programmable patterns of one synapse.

Programmable control sequencer The access FSM is a straightforward solution for performing control sequences on the synapse array interface. However, the resulting state diagram (Figure 3.24) is relatively complex, although the state machine only goes through fixed sequences with controlled timing in reaction to a presented command. An easier way to achieve this is using a sequencer similar to the approach taken for the automatic update controller (Section 3.3.2.5): Control sequences are stored in a memory block. A command triggers the execution of the control sequence stored at an associated address. Making this memory writable by the PP allows for micro-optimization of access sequences. For example a plasticity program may want to scan the array using a specific evaluation pattern without considering weights or performing resets. To do this, it can write an appropriate sequence to the sequencer memory effectively creating a new specialized command. Furthermore, in an experimental system, such as the BrainScaleS wafer-scale system, it is a benefit to have detailed control over internal processes. This increases the chances to be able to fix problems arising after tape-out without having to change the design.

Data loop-back mode To facilitate testing of the digital design, the new adapter features a loop-back mode for data transfers to the synapse array. With the loop-back mode enabled, the output registers (synapse array to controller) sample directly from the input register (controller to synapse array), instead of the data port. This allows for test programs that execute with defined results without depending on the analog part of the design.

Arbitration between external and internal accesses The new adapter provides two interfaces to access synapses: one is the specialized interface with the 128 bit data buses used by the PP. The other implements the PPB interface and is connected to the internal system bus. The latter interface can also be used by the processor. Therefore, non-timing critical operations, like changing the sequencer memory or enabling loop-back mode, are performed through the PPB interface. Timing critical accesses from the processor use the specialized bus, which does not allow for the slow control operations. The external controller always uses the PPB interface.

3.3.3.1. Detailed description

Figure 3.27 shows the control sequencer and its datapath. The sequencer (Figure 3.27A) contains two memories: The opcode table maps up to 16 operation codes to address offsets and the I/O sequencer store holds control sequences with a total number of 32 entries. If a new command is presented, its opcode is used as index into the opcode table and the returned data loaded to the sequence counter. This counter indexes the sequencer store that in turn provides a 24 bit word that contains control signals for the synapse array interface and the internal datapath of the adapter. Each word in the store represents one step in the access sequence. Signals for the synapse array interface are presented through a latch. The latch is there to ensure, that control signals stay stable in-between control operations.

Control operations A control operation consists of a command code (opcode), a row number, and a column set address. The row number is used to set the syn_a/syn_-ab, and syn_en signals of the synapse array interface to address a single row in the array. The column set address is translated into the one-hot coded en port of the interface (see Section 3.3.1) to select which columns are referenced by the operation. The semantics of the operation and whether data is written or read by it, is defined by the contents of the sequencer store. Table 3.5 lists a number of command code names that are

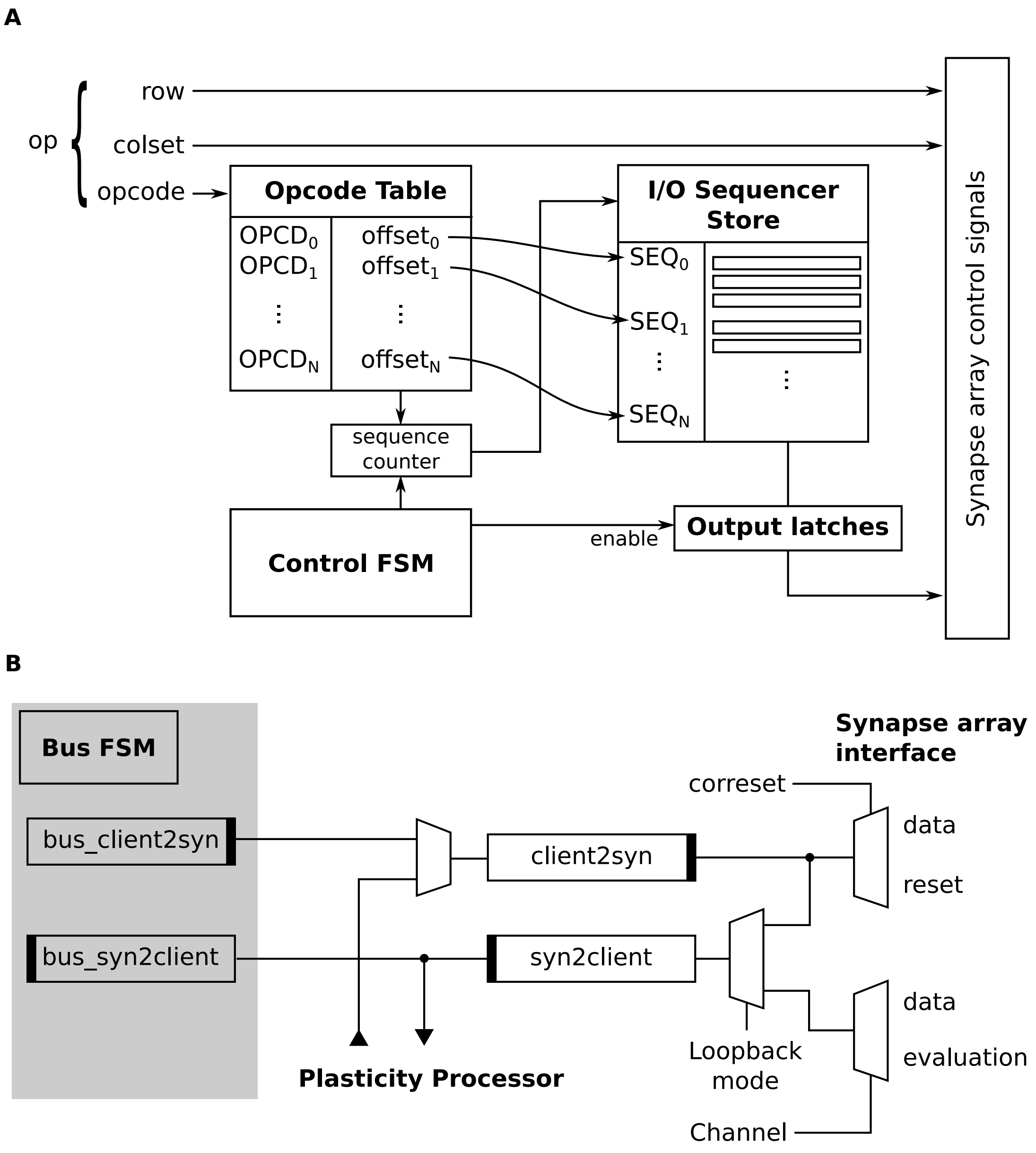


Figure 3.27.: (A) Control sequencer for the synapse array interface adapter of the programmable STDP implementation. (B) Datapath for the synapse array interface adapter. Registers are marked as boxes with a black bar to the right or left side. The black bar marks the output side of the register.

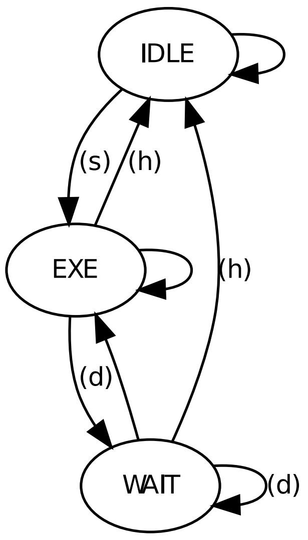


Figure 3.28.: State diagram of the sequencer state machine. It is controlled by three inputs: Start (s) triggers the execution of a sequence. The halt signal from the sequencer store (h) marks the end of the sequence. The delay value from the sequencer store is counted down to zero for each entry in the sequence. While this counter is positive the state machine sees a delay signal (d). Edges are labeled with the active control signals that trigger the associated transition. Inactive signals are omitted. Unlabeled transitions are taken, if none of the other transitions from this state are taken.

associated with the lower 14 positions in the opcode table. After reset, the contents of opcode table and sequencer store define meaningful operations for these codes similar to the commands defined in Table 3.3. The `user` and `ext` commands point to address 0 in the sequencer store and are intended to be programmed by the user. The `idle` command also has no sequence associated with it and is meant as a no-operation code. Commands `open_weights` and `open_dec` activate weight or decoder address wordlines of the addressed row. In the open row, `read` and `write` perform input and output of weight and decoder address data, depending on which open command was used previously. With `close` the wordline is deactivated again. Commands prefixed with `corr_` orchestrate the evaluation process: accumulator values are loaded to the temporal storage locations in the evaluation unit, evaluated with a given pattern, and the result sampled to the output register. After that, accumulators are reset back to zero depending on a reset pattern given on the data port.

Sequencer FSM Sequencing is controlled by the sequencer FSM, of which the state diagram is shown in Figure 3.28. If the start of a sequence is signalled either from the processor or via the PPB interface, the state machine transitions into an executing state (EXE). In this state the counter is incremented in every cycle to present a new set of signals to the synapse array interface through the latch. Every word in the sequence contains a 6 bit delay value that is loaded to a decrementing counter, when the word is fetched from the sequencer store. If the value is greater than zero, the state machine assumes a waiting state (WAIT) until the counter reaches zero. This allows for timing control of every step in an access sequence. The end of a sequence is marked by a bit in the last sequence word. When it is encountered and the associated delay of the word has elapsed, the state machine transitions back to the IDLE state. The output latch in Figure 3.27A is enabled by the state machine. Only in the executing and waiting states is the latch transparent in the positive clock phase. Therefore, control signals maintain the last value given until the next sequence is started. Thereby, for example the wordline for a synapse row can be kept active between accesses as it was done in the non-programmable adapter using the `start_read` and `close_row` commands.

Datapath Figure 3.27B shows the datapath of the adapter. Its task is to exchange data between the synapse array interface, the plasticity processor, and the bus interface. Central are the 128 bit `syn2client` and `client2syn` registers. The `syn2client` register reads from three potential sources: the synapse array dio or corrin port, or the `client2syn` register. For the latter, loop-back mode has to be enabled by a preceding bus access. Whether the dio or corrin port is read is decided by a bit in the current sequence word. The output of the `syn2client` register is read by the processor and, for bus read requests, by the 128 bit `bus_syn2client` register. In the opposite direction, the `client2syn` register forwards data either from the `bus_client2syn` register, for bus writes, or from the processor interface. The `correset` bit in the sequence word controls if its contents is presented to the data or the accumulation reset port of the synapse array interface.

Coding of words in the sequencer store The coding of words in the sequencer store is shown in Table 3.4. Bits 23. . . 14 control the execution of the sequence and configure the internal datapath. The `halt` bit marks the end of a sequence in the sequencer store. The `delay` field holds the value to be loaded to the decrementing counter and determines how long the current word of the sequence is active. The bits `data_valid` and `data_channel` control reading of data from the synapse array to an internal register. The two channels are: one for synapse weights and decoder addresses (`data_channel` = 0), the other for evaluation results (`data_channel` = 1). Both are stored in the same 128 bit register. The `change_en` bit is an additional enable on the output latches of the synapse array enable ports (syn_encrb, syn_ensynb, and syn_endecb). Only if it is set do the outputs change to the values specified in the associated fields of the sequence word. Using this bit, common control sequences

Name	**Position**	**Description**
	Sequencer control signals	
`halt`	23	Marks last entry of the current sequence
`delay`	22...17	Delay the next word from the sequence by the given number of cycles
`data_valid`	16	Sample incoming data from the synapse array to the syn2client register
`data_channel`	15	Select the data channel to which to sample
`change_en`	14	Change syn_encrb, syn_ensynb, and syn_-endecb only when this bit is set
	Synapse array interface control signals	
`encrb`	13	Value for syn_encrb
`ensynb`	12	Value for syn_ensynb
`endecb`	11	Value for syn_endecb
`syn_en`	10	Activate one of the syn_en bit chosen by the row address
`csen`	9	Value for csen
`ramoeb`	8	Value for ramoeb
`scc`	7	Value for scc
`sca`	6	Value for sca
`ramwb`	5	Value for ramwb
`pcb`	4	Value for pcb
`drv_io`	3	Enable driver on the dio port
`correset`	2	User data enables accumulation reset
`eval`	1	Activate evaluation pattern
`pat_rst`	0	Reset the pattern counter to zero

Table 3.4.: Coding of words in the sequencer store. The upper bits control the execution of the sequence and configure the internal datapath. The lower bits are either directly forwarded to the synapse array interface (see Figure 3.17) or directly control inputs to the synapse array interface. For example `syn_en` is converted to the 2 bit syn_en signal enabling the left or right side of synapse drivers depending on the currently selected row address.

`idle`	`close`	`user_0`
`open_weights`	`corr_load`	`user_1`
`open_dec`	`corr_eval`	`ext_0`
`read`	`corr_sample`	`ext_1`
`write`	`corr_reset`	

Table 3.5.: List of pre-defined opcode names. These names are labels for addresses in the opcode table that hold offsets into the sequencer store. Opcode table and sequencer store can be freely programmed to define new operations.

for weight and decoder address access can be defined. An initial sequence opens a row for weight or decoder address access just like the `start_read` command of the adapter of the non-programmable implementation. The read and write sequence do not change enables and therefore automatically read the correct data.

The lower bits 13. . . 0 are mostly directly presented to the synapse array interface. There are some exceptions: The syn_en port of the interface is 2 bit wide enabling the left or right side of the synapse array. To allow for a coding independent of the target row, the 2 bit signal is computed automatically depending on the provided row address if the 1 bit `syn_en` field in the sequencer store is set. The `drv_io` bit enables the driver from the adapter to the array interface on the dio port. `correset` controls, whether the internal output register is forwarded to the data input/output port (dio) or the accumulator reset (corresetb). The 128 bit content of the internal register is reduced to the 32 bit corresetb signal by performing a logical or on each half-byte and inverting the result. Multiple 4 bit evaluation patterns are provided by the processor from its internal pattern register (see Section 3.4). A counter selects the currently active pattern. By setting the `eval` bit in a sequence, the currently selected pattern is asserted to the array interface and the counter incremented. With the `pat_rst` bit, the counter can be reset back to zero.

Bus access state machine Figure 3.29 shows the state diagram of the bus FSM. The state machine handles requests via the PPB to synapse driver SRAM, the opcode table, sequencer store, and internal registers, for example `bus_syn2client` and `bus_client2syn`. It also triggers the control sequencer to perform access operations. Table 3.6 gives the address map for the bus interface. Depending on the type of request – read or write – and the address, the state machine performs different actions. Edges in the state diagram in Figure 3.29 are marked with "w" if the transition is taken on write requests and "r" for read requests. If the address falls in the synapse driver SRAM range, edges with label "s" are taken. Accesses to the op register follow the edge labeled with "o". All other requests reference internal registers and cause transitions not labels with "s" or "o". In this case, data can be read or written within one cycle.

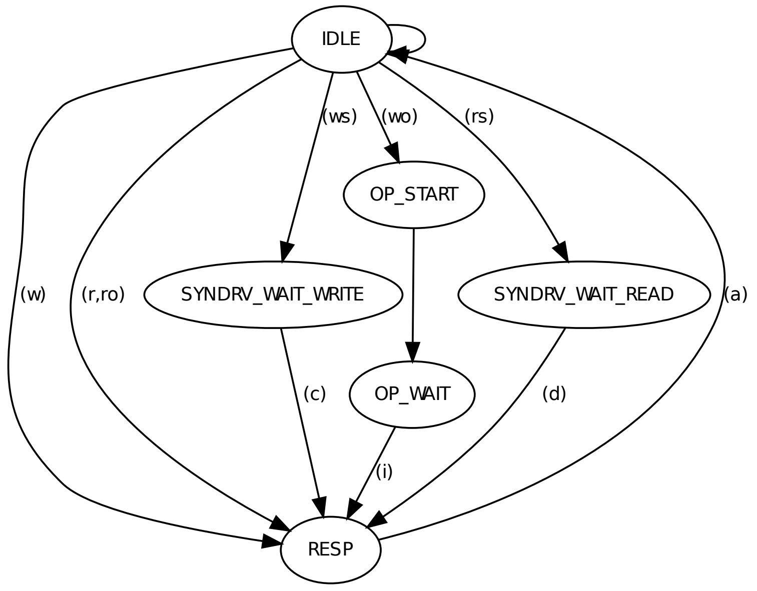


Figure 3.29.: State diagram of the bus FSM. Edges are labeled with the active control signals that trigger the associated transition. Inactive signals are omitted. Unlabeled transitions are taken, if none of the other transitions from this state are taken. The control signals are defined in the text.

Name	Lowest address	Highest address
Opcode table	`0x0000`	`0x00ff`
Sequencer store	`0x0100`	`0x01ff`
`bus_client2syn`	`0x0200`	`0x0203`
`bus_syn2client`	`0x0204`	`0x0207`
op register	`0x020a`	`0x020a`
gen register	`0x020b`	`0x020b`
dllresetb register	`0x020c`	`0x020c`
Loop-back enable register	`0x020d`	`0x020d`
Synapse driver SRAM	`0x1000`	`0x12ff`

Table 3.6.: Address space layout for requests on the PPB interface. Registers are addressed in big-endian byte-order.

The state machine assumes the RESP state to wait for the MRespAccept signal (label “a”) from the bus master (see Section 3.2). For a write to synapse driver SRAM, the state machine goes into the SYNDRV_WAIT_WRITE state. The write request is forwarded to the sramCtrl module by Schemmel (2011) also used in the non-programmable implementation. If it accepts the request (label “c”), the write is complete and FSM waits in the RESP state to complete the response phase. Similarly, read requests use the intermediate SYNDRV_WAIT_READ state to wait for the returned read data (label “d”), before transitioning to RESP. Operations are handled with two intermediate states, where the first triggers the control sequencer to initiate a sequence using opcode, row and column set address from the op register. The second state waits for the completion of the sequence before returning the response via the RESP state.

This implementation is notably different from the way the non-programmable implementation handles external requests (see Section 3.3.2.2). Requests here do not have to be completed within one cycle. Especially lengthy control sequences communicate to the requester, when they have completed by returning a response. This simplifies the timing of requests by the master. Instead of having to check for a busy state as before, requests can be made back to back. The handshake of the PPB ensures the correct timing. Also, for synapse driver SRAM reads only one request is required, that will directly return the correct data. Thus, by the use of the PPB full pipelining of requests is possible, while eliminating the need for polling.

3.4. SYNAPSE special function unit

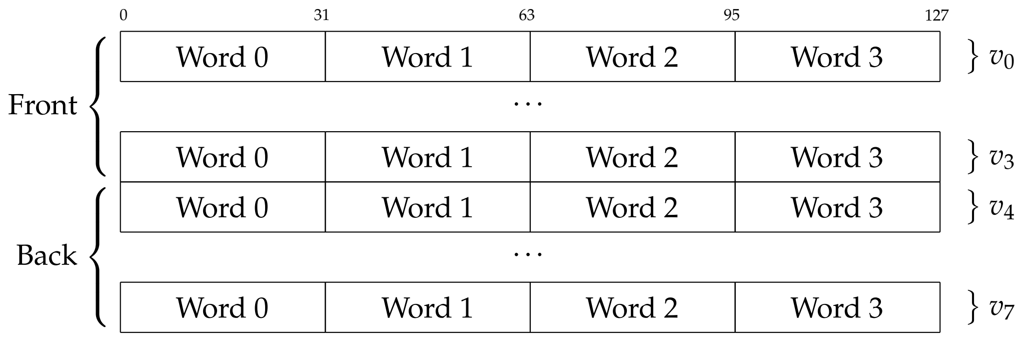
On one HICANN ASIC there are two synapse arrays with a total of 114 688 synapses. It operates with an acceleration factor α between 10^3 and 10^5 (see Equation 2.1). To achieve a weight update rate in the order of magnitude of one second in biological time for every synapse, $1.15 \cdot 10^8$ to $1.15 \cdot 10^{10}$ new weights need to be processed per real-time second. Since gigahertz clock frequencies are unrealistic for a standard-cell processor in 180 nm process technology, processing needs to be parallelized as much as possible. On the algorithmic level, the problem mainly exhibits data-level parallelism: many synapses are processed using one or a few algorithms. A well established concept for data-parallel processing in the taxonomy of Flynn (1972) is Sinlge Instruction Multiple Data (SIMD) processing. One instruction operates on a one-dimensional array, or vector, of data. This section describes the SYNAPSE special-function unit for the acceleration of weight processing using SIMD processing. It implements an extension to the instruction set using 128 bit vectors that each hold 32×4 bit synaptic weights. The width of 128 bit was chosen, because the data port dio of the synapse array interface is of this size. Therefore, 128 bit can be read and written in one access. The extension also includes asynchronous I/O operations for these vectors using the specialized I/O adapter presented in Section 3.3.3. Making I/O asynchronous means that data transfers do not block the execution of the program. This allows for overlap between weight computation and data transfer, increasing performance.

3.4.1. Special purpose registers

The SYNAPSE instruction set extension adds a number of new application specific registers.

3.4.1.1. Vector registers

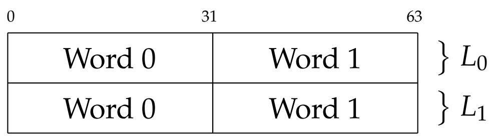
There are eight 128 bit wide vector registers v_0 to v_7:



They are separated into a back and a front side as viewed by software. Only the front side can be accessed by the program, while the back side is used for I/O. The `synswp` instruction allows to swap front and back side. After the instruction the previous content of registers v_0 to v_3 is available in v_4 to v_7 and vice versa. Elements $v_{i,0} \dots v_{i,31}$ for the i-th vector are positioned from most significant to least significant bit.

3.4.1.2. Look-up table registers

There are two look-up tables for 4 bit arithmetics L_0 and L_1:



The replacement value for weight 0 ($L_i(w = 0)$; see Equation 3.13) is stored at position $0 \dots 3$, that for weight 15 ($L_i(w = 15)$) at $60 \dots 63$.

3.4.1.3. Pattern register

Up to four evaluation patterns can be stored in the 16 bit pattern register:

0	1	2	3	
$\overline{e_{aa}}$	$\overline{e_{ac}}$	$\overline{e_{ca}}$	$\overline{e_{cc}}$	$\} \; p_0$
4	5	6	7	
$\overline{e_{aa}}$	$\overline{e_{ac}}$	$\overline{e_{ca}}$	$\overline{e_{cc}}$	$\} \; p_1$
8	9	10	11	
$\overline{e_{aa}}$	$\overline{e_{ac}}$	$\overline{e_{ca}}$	$\overline{e_{cc}}$	$\} \; p_2$
12	13	14	15	
$\overline{e_{aa}}$	$\overline{e_{ac}}$	$\overline{e_{ca}}$	$\overline{e_{cc}}$	$\} \; p_3$

For technical reasons, patterns are stored inverted – as indicated by the bar over the symbol – with respect to the notation of Section 3.3.1.2.

3.4.1.4. Select state register

An internal select state is maintained that is set by a vector compare operation (syncmpi, see below) and used by a element-wise vector select instruction (syns, see below):

0	31
Select state	

The select state bits $s_0 \ldots s_{31}$ correspond each to one element in a vector register. They are positioned from most- to least-significant bit in the select state register.

3.4.2. Special purpose instructions

Instructions defined by the SYNAPSE extension use the special purpose registers to compute weights and perform I/O with the synapse array. Instructions are coded using the X- and D-form specified by PowerISA (2010). Both have a 6 bit primary opcode and are 32 bit long. The X-form allows for a 10 bit extended opcode XO and three 5 bit register locations RT, RA, and RB. The latter is used by arithmetic instructions that save comparison results – greater than, less than or equal to zero – to a condition register if the bit is set. The D-form is used for instructions with 16 bit immediate data D, i.e. data coded directly into the instruction. It allows for two 5 bit register locations RT and RA.

In the following instruction coding is specified in tabular form. The first row specifies the instruction form by showing the positions of the associated fields. The second row shows how these fields are interpreted by the instruction. Ignored bits are indicated by double slashes "//". Symbols y, a, and b stand for 128 bit vector register indices. The symbol l is a look-up table index and p an evaluation pattern index. Indices for general purpose registers are denoted using o, z, and t. The contents of a general purpose register is referred to by g_t for register t. The data returned from the synapse array interface adapter is written as 4 bit elements $d_0 \dots d_{31}$.

3.4.2.1. Apply mapping from look-up table to vector elements

synm y, a, l

OPCD	RT		RA		RB		XO	
4	//	y	//	a	//	l	648	//

Elements in register v_a are replaced using the look-up table L_l and the output is written to register v_y.

$$v_{y,i} = L_l\left(v_{a,i}\right) \quad \text{for all} \quad i \in [0, 31] \tag{3.14}$$

This instruction is the main facility to compute 4 bit weights.

3.4.2.2. Compare elements with immediate

syncmpi a, $(m << 4)\&c$ (opcode 6)
syncmpi. a, $(m << 4)\&c$ (opcode 9)

OPCD	RT	RA		D		
6,9	//	//	a	//	m	c

Mask elements in register v_a with bit-mask m and compare to pattern c. Write the result to the select state register.

$$s_i = \begin{cases} 1 & \text{if } v_{a,i}\&m = c \\ 0 & \text{else} \end{cases} \quad \text{for all} \quad i \in [0, 31] \tag{3.15}$$

If the variant recording to the condition register is used (syncmpi.), the first condition register field (CR0) is written. If no select state bit s_i was set by the instruction, the equal to zero bit (EQ) is set in CR0 and greater and lesser than zero (GT,LT) are unset. If at least one of the s_i is set, EQ and LT are unset and GT is set.

This instruction is used to process evaluation result data. Each evaluation by the synapse array interface adapter contributes one of up to four bits to each element of a vector register. The syncmpi and syncmpi. instructions can be used to compare this result to the given pattern c, while using a mask m. The main purpose of the mask is to select, which bits were set by the evaluation sequence. For example, if only two evaluations are performed, $m = 0011$ (binary) compares only the set bits and treats the others as don't care. The select state register is used by the select instruction syns to combine two vector registers. The recording variant syncmpi. is meant to be followed by a branch instruction using the condition register field. This way, computing code can be bypassed if no bits are set.

3.4.2.3. Select elements from two vectors

syns y, a, b

OPCD	RT		RA		RB		XO	
4	//	y	//	a	//	b	649	//

Select elements from vector v_a and v_b depending on the select state register contents and write the result to v_y.

$$v_{y,i} = \begin{cases} v_{a,i} & \text{if } s_i = 0 \\ v_{b,i} & \text{else} \end{cases} \quad \text{for all} \quad i \in [0, 31] \tag{3.16}$$

This instruction together with syncmpi and syncmpi. is used to account for the evaluation result in the weight computation. Depending on the result from evaluation, the update has to be computed differently. For example, the evaluation might test, whether a synapse encounters more pre-before-post than post-before-pre spike pairs. Depending on which of them is seen more, the weight should be increased or decreased. However, synm uses only one look-up table to compute the new weight for 32 synapses simultaneously. The compare and select mechanism allows for an elegant solution: for all weights, both variants are computed. Then syncmpi and syncmpi. identify first synapses that should be increased, then those that should be decreased. With syns the final result of the computation can be combined from the two speculatively computed intermediate results.

3.4.2.4. Perform operation sequences

synops o, z, t

OPCD	RT	RA	RB	XO	
4	o	z	t	656	//

Send the 4 bit operation codes given in g_o in order from most- to least-significant bit to the synapse array interface adapter, where they are used as control sequence operation codes. Use the address given by $g_z + g_t$ for the control sequences. The 128 bit result from the adapter $d_0 \ldots d_{31}$, if any, is written to v_4 if the control sequencer selects the data channel. Otherwise, for evaluation result data, every n-th bit of each element in vector v_5 is set to the corresponding bit in the returned data vector. Here n

is the current value of the pattern counter maintained by the control sequencer (see Section 3.3.3).

$$m = (1 << n)$$
$$\begin{cases} v_{4,i} = d_i & \text{if data channel used} \\ v_{5,i} = v_{5,i} | \, (m \& d_i) & \text{else} \end{cases} \quad \text{for all } i \in [0, 31] \tag{3.17}$$

The general purpose register g_o can hold up to eight 4 bit codes for the control sequencer of the adapter. A code of value zero terminates an operation sequence with less than eight operations. For each code, the control sequencer presents the stored sequence of control signals to the synapse array interface. The `data_valid` and `data_channel` bits in the sequence (Table 3.4) determine if data is written to the vector registers.

The SYNAPSE functional units stays ready, while the operation sequence is executed. If a second synops instruction is encountered, before the previous one has finished, the functional unit is marked not ready until the first one has completed. Typically, a user program will use the synswp instruction to wait for the completion of an operation sequence and to make the result available to the program. While the sequence is being executed, other instructions of the functional unit are available to compute weight updates. This allows for an efficient overlap of computation and communication.

3.4.2.5. Swap vector register file

synswp

OPCD	RT	RA	RB	XO	
4	//	//	//	657	//

Wait for the completion of all previous synops instructions and exchange the contents of front and back vector registers.

$$\begin{aligned} &\text{for all } i \in [0, 3]: \\ u_i &= v_i \\ v_i &= v_{i+4} \\ v_{i+4} &= u_i \end{aligned} \tag{3.18}$$

This instruction waits for all ongoing I/O operations to complete. By swapping the register file it makes I/O results available to the weight computation commands operating on the front registers $v_0 \ldots v_3$. Separating registers into front and back, where only front registers are usable by software, is a simplification for data hazard detection. Because synswp waits for the completion of I/O operations before swapping registers, it is not possible for software to reference a vector register, before it contains a valid result.

3.4.2.6. Register move instructions

synmtl l, z, t

OPCD	RT		RA	RB	XO	
4	//	l	z	t	650	//

Write the contents of general purpose register z to word 0 of L_l and that of general purpose register t to word 1.

synmtvr y, z, w

OPCD	RT		RA	RB		XO	
4	//	y	z	//	w	651	//

Write the contents of general purpose register z to the w-th word of vector register y.

synmfvr o, a, w

OPCD	RT	RA		RB		XO	
4	o	//	a	//	w	652	//

Write the w-th word of vector register a to general purpose register o.

synmvvr y, a

OPCD	RT		RA		RB	XO	
4	//	y	//	a	//	655	//

Write the contents of vector register a to vector register y.

synmtp z

OPCD	RT	RA	RB	XO	
4	//	z	//	653	//

Write the upper half-word of general purpose register z to the pattern register.

synmfp o

OPCD	RT	RA	RB	XO	
4	o	//	//	654	//

Write the pattern register to the upper half-word of general purpose register o.

Listing 3.5: Example code using the SYNAPSE instruction set extension for writing and reading weights.

```
# first load general purpose registers
li r5, 0
li r6, 0
lis r7, 0x1450
lwz r8, 0(0)
lwz r9, 4(0)
lwz r10, 8(0)
lwz r11, 12(0)

# load vector register
synmtvr v0, r8, 0
synmtvr v0, r9, 1
synmtvr v0, r10, 2
synmtvr v0, r11, 3

# perform operation sequence to write weights
synops r7, r5, r6

# load and perform read sequence
lis r7, 0x1350
synops r7, r5, r6
synswp

# read vector register to general purpose registers
synmfvr v0, r12, 0
synmfvr v0, r13, 1
synmfvr v0, r14, 2
synmfvr v0, r15, 3
```

3.4.3. Code example

This section goes through an assembler code example to explain the usage of the presented instruction set extension for writing and reading weights. Listing 3.5 shows the example program. First in lines 2 to 8 a number of general purpose registers are loaded. General purpose registers r5 and r6 are used as row address and offset for the synops instruction. They are initialized to zero using the load immediate instruction li. The lis instruction loads an immediate value to the upper half-word of the destination register. The value loaded to register r7 is an operation sequence consisting of three opcodes from Table 3.5. In order of execution they are: `open_weights` (1), `write` (4), and `close` (5). After that, in lines 5 to 8 four general purpose registers are loaded from memory addresses 0 to 15 using the load word and zero (lwz) instruction. The vector register v_0 is initialized with the just loaded values using the synmtvr instruction on lines 11 to 14. The write is started on line 17 with synops using the previously configured operation sequence and address registers. On line 20, the read sequence is loaded, replacing the code for `write` (4) with the one for `read` (3). The synops instructions waits for the previous one to finish, before starting the read. With synswp on line 22 vector registers are swapped, so that the result of the read can be transferred to general purpose registers on lines 25 to 28.

3.4.4. Design considerations for the instruction set extension

Operation codes The Power ISA uses the X-form of instructions to extend the primary 6 bit opcode with a 10 bit extended code. This gives room for 2^{16} potential instructions, but leaves relatively few bits in the instruction word to parameterize the operation. Especially, there is no room to give an immediate operand. For this purpose there is the D-form, which allows for 16 bit immediate values. However, there can be a maximum of 2^{6} D-form instructions, since there is no extended opcode. In PowerISA (2010) there are six unassigned primary opcodes available, of which 6 and 9 are two. These opcodes are used for the syncmp and syncmpi. instructions to avoid collisions with other operations. Executing a program containing SYNAPSE instructions on a generic Power ISA compliant processor would therefore cause an invalid instruction exception. The X-form instructions use primary opcode 4, which is assigned to instructions with extended opcodes in the vector, legacy multiply-accumulate, and signal processing categories (PowerISA, 2010). The chosen extended opcodes 648 to 657 are unassigned for primary opcode 4. The opcode spaces are visualized in (PowerISA, 2010, Appendix F).

I/O operation sequences The goal of the synops instruction is to provide a simple way to initiate asynchronous I/O operations. If I/O is performed asynchronously, computation and communication can be overlapped. A sequence of control operations is specified in one go. This is in addition to the option of programming custom control sequences to the synapse array interface adapter (Section 3.3.3). Sequences programmed to the adapter represent operations with a fundamental function, e.g. reading a column set. The `synops` instruction then goes through a complete sequence of these fundamental operations parallel to normal program execution. Starting a complete sequence per `synops` reduces code size and additional requirements for synchronization with I/O operations.

Partitioned register file Having the vector register file partitioned into front and back parts has two advantages. First of all, it simplifies data hazard detection. The software program can not access registers before they are written with the result from an operation sequence. Therefore, in flight writes do not need to be tracked for the back side of the register file. This saves tracking resources in the processor front end (see Section 3.1.6). The second advantage is, that it allows the use of memory blocks with fewer read and write ports. Front and back side each get one dedicated block with one read and one write port. With `synswp` the role of the two blocks is switched around. If instead one would use only one block, four ports were required, since instructions and the synapse array interface adapter can simultaneously read and write one register.

Vector select SIMD processing is only effective if the same code is executed for all data elements. However, STDP requires different treatment of individual synapses depending on the result of the analog evaluation. Doing this with branches would necessitate iteration over the vector elements and therefore, one would lose the benefit of parallel vector processing. With the selection mechanism by `syns` the necessity for branches can be removed by transferring the evaluation of condition codes from the control path to the data path. An alternative would be conditional execution by `synm`, where only elements are modified for which the select state bit is set.

Vector registers local to functional unit In contrast to other registers of the processor, e.g. general purpose registers, the vector registers are only accessible from the SYNAPSE special function unit. Especially, load and store operations can not use the vector registers, but instead data has to be transferred via general purpose registers. This is an acceptable performance penalty for the STDP application, where I/O is primarily performed with the synapse array. On the other hand, it allows for a local implementation of the register file. This way, reading and writing registers does not utilize the operand and result bus (see Section 3.1.7). Especially, a 128 bit bus is not needed outside of the functional unit.

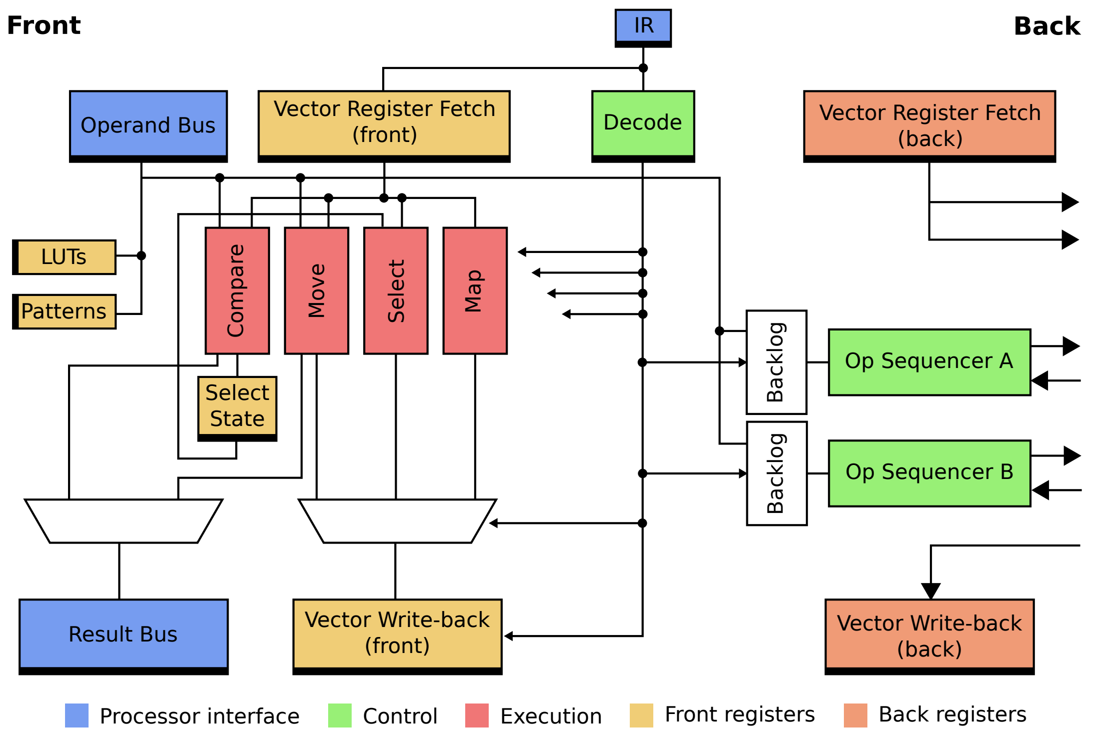


Figure 3.30.: Overview of the SYNAPSE functional unit implementation. Thick edges on a box indicate a registered output. A detailed description is given in the text.

3.4.5. Implementation

Figure 3.30 shows the internal organization of the SYNAPSE special-function unit. The unit is connected to the processor using the standard back end interface described in Section 3.1.7. The instruction is presented on the IR register and one cycle later fetched operands are available via the operand bus. As stated above, the vector registers are internal to the functional unit. Two vector operands VA and VB are fetched in parallel to the decode phase. The four execution units Compare, Move, Select, and Map compute results in a single cycle. The Compare unit implements the syncmpi and syncmpi. instructions. It reads operand b from the operand bus containing the instruction immediate, and vector operand VA. Move performs data exchange between vector and general purpose registers, reading VA and b from the operand bus. Select and Map execute syns and synm, respectively. Results from these units are multiplexed to the result bus and back to the vector register file depending on instruction type. The vector register file uses write-through like the general purpose register file (Section 3.1.6.4). Look-up tables and evaluation patterns are written one cycle after decode and do not use a dedicated fetch cycle for reading. Therefore, all instructions of the SYNAPSE functional unit, except for those writing to general purpose or condition registers (synmfvr, syncmpi.), have an issue to retire latency $L = 2$ and thus penalty $P_2 = 0$ for data hazards (see Section 3.1.7). This means, they can be executed back-to-back.

The back side of the functional unit is connected to the synapse array interface adapter described in Section 3.3.3. Vector register v_4 is statically forwarded to the adapter, while in the opposite direction data is recorded to v_4 and v_5. Which of the destination registers is used is determined by the `data_channel` bit in the control sequence executed in the adapter (Section 3.3.3). Data is written whenever the `data_valid` bit in the control sequence is set. The execution of the operation sequence given to the synops instruction is performed by two operation sequencers. Only one of them is active at a time depending on the address of the operation. A backlog of one operation sequence is maintained. If a new sequence is given, before the current one has completed, the new one is saved to a register and the functional unit marked as not-ready to prevent any further synops instructions to be issued. This situation is already detected in the decode phase to assure, that no instructions are lost.

3.5. Native Vector Extension

The Native Vector Extension (NVE) was a precursor variant of the SYNAPSE instruction set extension. It is only described briefly here, because it is not intended for implementation, but is used for comparisons in Chapter 5.

The concept behind the extension is to enable 4 bit vector operations in 32 bit general purpose registers. To this end, it provides compare, select, and map operations identical to those of SYNAPSE (Sections 3.4.2.1-3.4.2.3), but with reduced width. It provides no capabilities for I/O. Instead, the general purpose load/store, or the external control unit have to be used to access the synapse array. Therefore, only one fourth of the bandwidth available to SYNAPSE can be used.

Native Vector Extension (NVE) represents a minimal specialization for the problem of 4 bit weight computation. Compared to SYNAPSE, it does not require a costly dedicated register file. Its lack of I/O bandwidth and wide registers however, lead to reduced performance, as will be shown in Section 5.4.2.

4. Functional verification and software support

The so far presented designs of plasticity processor (Section 3.1) and its bus (Section 3.2) pose a difficult verification problem. The processor must behave according to specification for arbitrary programs and the design can use the PPB in arbitrary topologies. The task of functional verification is to systematically exercise the design and verify, that it performs according to its specification.

Directed tests apply a fixed input, to which the testbench holds a static expected result for comparison with the actual result. The test passes if expectation and actual result match. This concept of testing is well suited to ensure that specific problems are not present in the design. An example would be a regression test that ascertains, that an error identified once is not introduced again. Also with this method the knowledge of the designer about problematic input patterns for example can be used to identify problems more quickly. However, directed testing is not well suited for finding unsuspected errors.

Here, a better approach is to use random testing or more precisely constrained random testing. The design under test is exercised with randomly generated input patterns and the result is compared to a prediction of the result by the testbench. The generation of random input is constrained to a certain subset of valid input patterns according to the specification of the design. Typically the random distribution of the input is weighted to make corner cases more likely to occur. Constrained random testing is accompanied by coverage analysis, which provides a metric of how much of the design has been exercised by the random stimulation. Constrained random testing and functional coverage analysis are directly supported by the SystemVerilog hardware description language.

In addition to only comparing the result against the expectation, assertions and coverage statements can be inserted into the design. Assertions specify an erroneous condition in the design that may not occur. Coverage statements on the other hand test that certain conditions do occur. Conditions in SystemVerilog assertions are not limited to boolean expressions, but can instead be sequences, formulated as implications. For example, if condition A is true in cycle one and B is true in the next cycle, condition C must be true in the third cycle, or otherwise an error is reported. I used assertions and coverage statements throughout the design, but will not go into any further detail here. Instead, the following sections describe how the design of processor and bus is verified using directed and constrained random testing. A special challenge here is the automatic generation of random programs as input to the processor design and the prediction of the expected result for these programs.

4.1. Directed verification: program level testing

A straightforward way of testing a processor design is to execute a number of programs on it, for which the expected result is known in advance. Such a program test consists of three parts: the instruction memory image, the initial data memory image, and the expected final data memory image. The instruction and initial data memory images are loaded to the corresponding memory blocks of the processor in the testbench. After that the program is started. By convention test programs finish using the `wait` instruction and entering a sleep state. The testbench detects the completion of the program by monitoring the sleep state of the processor. A maximum duration for program execution is also specified, to avoid indefinite waiting times. If the program does not enter the sleep state in time, reset is asserted after the specified time. Then, the testbench reads the data memory content and compares it with the stored expected memory image. Discrepancies are reported with address, expected, and actual value on a byte-by-byte basis. This process is repeated for a number of test programs. Listing 4.1 shows the *load-add-store* test program with assembly source, code, data, and expected memory image. It is a simple program consisting only of the three instructions `load`, `add`, and `store` that each have data dependencies on the previous one. Typically, test programs are more complex than this example. Table 4.1 lists the main test programs used in the program level testing suite. Further programs were developed by Nonnenmacher (2011).

4.1.1. Generating the expected result memory image

Reference images for the programs listed in Table 4.1 are generated manually. This is only a viable approach for relatively small programs. An automatic approach was developed by Nonnenmacher (2011) under my supervision. In this approach programs are written in the C programming language. This allows compilation targeting the PP and conventional personal computers. By using an additional layer for encapsulation an expected result image can be generated through execution on the personal computer. This is accomplished through the use of a dedicated memory block in the program, which is written out to a file after the program has finished. With this approach only programs written specifically for testing can be used for reference image generation, because they have to use the dedicated memory block. To generate reference images using arbitrary C code, a software emulator of a Power ISA compliant processor could be used. However, such an approach was not taken.

Name	**Description**
load_add_store	Simple test for data hazard handling.
branch	Simple test of unconditional branch.
branch_cond	Test of conditional branching implementing a loop.
simple_interlock_test	Extended version of the load_add_store test with multiple interdependent load, add, and store instructions.
interlocks	Interlock testing in conjunction with branching.
logic	Exercising logical operations (e.g. `and`, `xor`, ...).
func_call	Testing function calls with `bl` and `blr`. Implements a 64 bit add and 32 bit multiply function.
special_reg	Test special purpose registers.
cshell	Testing the assembly shell around C-programs with a simple C-program performing arithmetic and memory operations.
load_with_update	Test of updating variants of load/store operations.
mem_multiple	Test of `stmw` and `lmw` instructions.
mul	Signed and unsigned multiplication for different inputs.
interrupt	Alignment and trap interrupt.
div	Like mul, but for division.
cr_complex	Advanced condition register manipulation, e.g. logical operations.
other_instructions	Primarily instructions for counting ones (`popcntb`) and computing parity (`prtyw`).
debugging	Case study for debugging of a recursive faculty function using a trap interrupt.
ext_ctrl	Write data to a bus endpoint behaving like a memory using external control instructions, read it back, and store the result to main memory.
ee_vsprintf	Fragment from the output function of the CoreMark benchmark: bytewise copy from one memory location to another.

Table 4.1.: List of test programs used for program level testing.

(a) Assembly source

```
lwz   3, 0(0)
lwz   4, 4(0)
add   5, 3, 4
stw   5, 8(0)
wait
```

(b) Instruction image

0 … 31

80	60	00	00
80	80	00	04
7C	A3	22	14
90	A0	00	08
7C	00	00	7C

(c) Initial data image

0 … 31

00	00	00	0F
00	00	00	02
00	00	00	00

(d) Expected data image

0 … 31

00	00	00	0F
00	00	00	02
00	00	00	11

Listing 4.1: The load-add-store test program used for program level testing. Memory images are given bytewise in hexadecimal numbers.

4.1.2. The CoreMark Benchmark for directed testing

An alternative method of deciding whether a test has passed or failed is a consistency self-check in the test program itself. This is less safe, because it relies on the design itself, which may show undefined behavior if a functional error is present. For example, if the self-check uses a comparison operation to compare the result of a computation to an expected value, but the compare instruction erroneously always reports a match, the self-check can not detect this error. However, if the design has already passed a battery of basic tests, self-checking represents an option for tests with increased complexity without the need of reference memory images.

The CoreMark benchmark by EEMBC (2012) is aimed at performance testing of embedded processors. It performs three main tests and computes a CRC32 (Peterson and Brown, 1961) checksum of the result. The checksum is then compared against a pre-stored expected value. The three tests perform matrix manipulation, linked list manipulation, and execution of a state machine. The CoreMark benchmark therefore offers the ability to a) test the design with code that was not written on purpose for testing of the PP and b) test commonly used algorithms.

4.1.3. Test results

Listing 4.2 shows an example report from the program level testing simulation. The report lists all programs together with a pass/fail indication and some performance

```
=====================================================
 SUMMARY:
 0        load_add_store : ok     CPI = 3.25 (  13/   4)
                                | BP:(--no branches--)
 1                branch : ok     CPI = 4.67 (  14/   3)
                                | BP:(T:1.00 + NT:0.00 /    1)
 2           branch_cond : ok     CPI = 2.65 (  53/  20)
                                | BP:(T:0.80 + NT:0.00 /    5)
 3  simple_interlock_test : ok     CPI = 1.95 (  41/  21)
                                | BP:(--no branches--)
 4            interlocks : ok     CPI = 3.29 ( 102/  31)
                                | BP:(T:0.91 + NT:0.00 /   11)
 5                 logic : ok     CPI = 1.52 ( 105/  69)
                                | BP:(--no branches--)
 6             func_call : ok     CPI = 2.04 ( 484/ 237)
                                | BP:(T:0.76 + NT:0.00 /   45)
 7           special_reg : ok     CPI = 2.33 (  56/  24)
                                | BP:(T:1.00 + NT:0.00 /    1)
 8              c/cshell : ok     CPI = 1.90 ( 219/ 115)
                                | BP:(T:0.93 + NT:0.00 /   14)
 9      load_with_update : ok     CPI = 1.67 ( 211/ 126)
                                | BP:(T:0.50 + NT:0.00 /    2)
10          mem_multiple : ok     CPI = 1.11 (  82/  74)
                                | BP:(--no branches--)
11                   mul : ok     CPI = 1.34 (  86/  64)
                                | BP:(--no branches--)
12             interrupt : ok     CPI = 2.72 ( 223/  82)
                                | BP:(T:0.63 + NT:0.00 /   30)
13                   div : ok     CPI = 4.77 ( 429/  90)
                                | BP:(T:0.82 + NT:0.00 /   11)
14            cr_complex : ok     CPI = 1.87 ( 247/ 132)
                                | BP:(T:0.75 + NT:0.00 /    8)
15    other_instructions : ok     CPI = 1.25 (  55/  44)
                                | BP:(--no branches--)
16             debugging : ok     CPI = 1.77 ( 165/  93)
                                | BP:(T:0.85 + NT:0.00 /   13)
17              ext_ctrl : ok     CPI = 2.84 ( 957/ 337)
                                | BP:(T:0.97 + NT:0.00 /   64)
18           ee_vsprintf : ok     CPI = 2.25 (2321/1032)
                                | BP:(T:1.00 + NT:0.00 /  258)
```

Listing 4.2: Simulator output for program level testing. Besides reporting whether a test has passed or failed, further performance metrics of the simulation are shown. The raw output from the simulator is edited for readability. See main text for details.

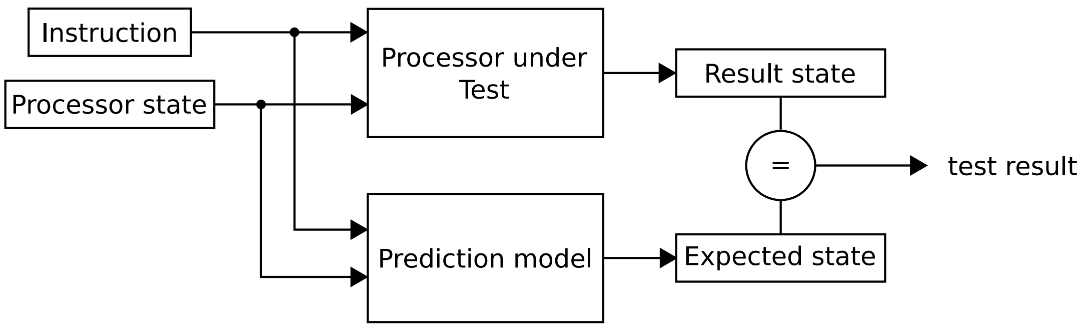


Figure 4.1.: Work-flow of the instruction level testing method. Instruction and processor state are generated randomly and send as input to the processor under test and a prediction model in the testbench. Both return results, which have to match for the test to be successful.

metrics. For each program the output shows Clocks Per Instruction (CPI) followed by total cycles and number of instructions in brackets. For the branch prediction accuracy, it shows percentages of mispredictions. Labeled with `T` are branches that where taken, but not predicted and vice versa `NT` shows the percentage of not taken branches that were predicted to be taken. The number after the slash is the total number of branches. This simulation used a branch cache with 16 entries (see Section 3.1.4).

4.2. Constrained random verification: instruction level testing

Program level testing decides correctness for a complete program as a whole. To deduce the specific error or errors from this information requires additional manual labor and can be difficult for long test programs. For example, for the test programs developed by Nonnenmacher (2011) finding the cause of a mismatch in the result and expected data images requires a detailed analysis of the assembly source. Also it is laborious to achieve good test coverage of the design. Each instruction or sequence of instructions must be written explicitly as part of a test program. To facilitate testing, achieve better coverage, and allow for quicker identification of the cause of a problem, instruction level testing is used. Using this method, the testbench randomly generates a single instruction and initial processor state and predicts the processor state after the instruction has executed. It uses the constrained random verification features of the SystemVerilog hardware description language. The work-flow is visualized in Figure 4.1.

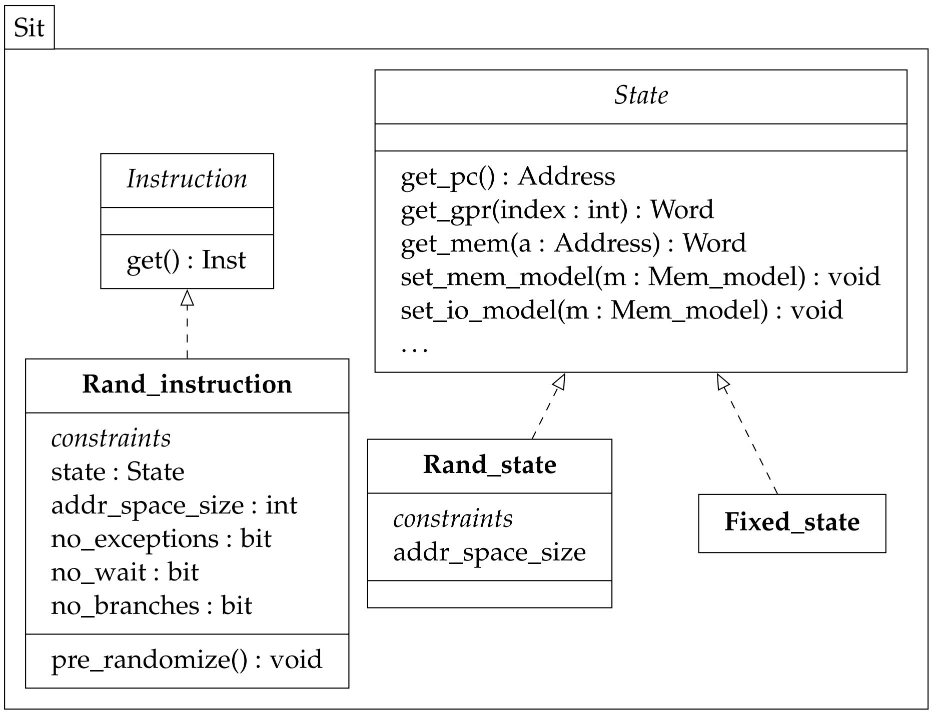


Figure 4.2.: Class diagram for instruction level testing. Here classes for handling instructions and processor state are shown.

4.2.1. Verification framework

The first problem to solve for instruction level testing is the generation of valid instructions and processor state. The state of the processor is defined by its internal registers. Additionally, the state of main memory is required to predict results for certain instructions, e.g. those of the load/store functional unit. Figure 4.2 shows a diagram of the classes used by the testbench for instruction and state. The instruction word itself is stored using the `Inst` data type. It holds the bit-exact 32 bit image of the instruction. The abstract `Instruction` class defines an interface to generate an instance of the `Inst` type. Its only implementation generates instruction words using the SystemVerilog random constraints facility. Constraints are formulated as conditions that must hold true for the generated instruction. When calling the `randomize()` member function of a class, the simulator employs a solver that tries to generate a new set of random values fulfilling all given constraints. Here, constraints are used to exclude instruction codings outside of the specification. For example, the load with update variants of load instructions may not specify the same general purpose register as target for loaded data and the address.

The constraint mechanism is also used for further limiting the generated instructions.

Load and store operations must be aligned to word boundaries or otherwise an interrupt is taken. This is defined behavior according to specification, but not useful during single instruction testing. By using the processor state, instruction generation can be constrained to aligned accesses only. For this, the `Rand_instruction` class must have knowledge of the processor state, hence the state member.

The `pre_randomize()` member serves as callback for the SystemVerilog randomizer. It is invoked after a call to `randomize()` and before actual randomization. Here it is used to dynamically enable and disable constraints depending on configuration options and whether the `state` member is specified to hold a valid state. If a valid processor state is available, memory accesses are aligned and branches are limited to the address space indicated by `addr_space_size`. Further options enable constraints to eliminate exceptions (`no_exceptions`), the occurrence of the wait instruction (`no_wait`) and of branches (`no_branches`).

Processor state The processor state is modeled with the abstract `State` class. It provides getter functions for all registers and for memory. The memory state is modelled using implementations of the abstract `Mem_model` class. Depending on the bus topology, I/O bus and load/store bus can see separate memory spaces. This is accounted for by individual `Mem_model` instances for I/O and load/store. The two implementations of the abstract `State` class either produce random register contents in the case of `Rand_inst`, or hold a static state that can be modified through additional setter functions. The latter is for example used to store the expected state generated by the prediction. The former again uses SystemVerilog constrained randomization features to generate the state. Here, constraints are simpler than for instruction generation: They assure, that reserved bits in special purpose registers are zero and limit registers that can be used as branch targets to an address range determined by `addr_-space_size`. Those registers are CTR, LNK, SRR0, CSRR0, and MCSRR0. Constraints are also used to bias the contents of general purpose registers to typical corner cases of arithmetic operations: The minimum and maximum values for unsigned representation 0 and 0xFFFFFFFF and minimum and maximum for signed two's-complement representation 0x80000000 (-2^{31}) and 0x7FFFFFFF ($2^{31}-1$).

Modeling memory Implementations of the abstract `Mem_model` class allow for get and set access to memory (Figure 4.3). The whole memory starting from a given address can be cleared to zero using the `clear()` function. The `iter_first()` and `iter_next()` members provide a simple iteration interface. This is necessary for sparse memory representations as it is implemented by the `Sparse_mem_model` class. With `iter_first()` the referenced `Address` variable is initialized to the first location in the memory and its contents is returned via the `Word`-type output variable. The return value is 0 if there is no entry in the memory. A call to `iter_next` moves

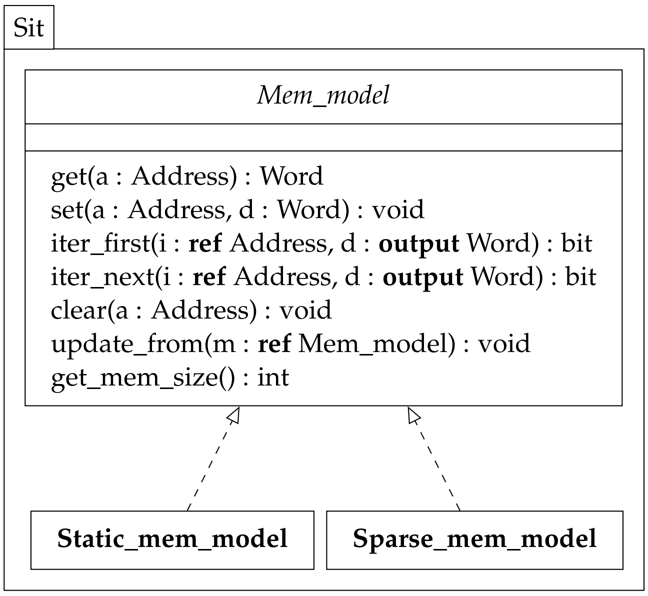

Sit
Mem_model
get(a : Address) : Word
set(a : Address, d : Word) : void
iter_first(i : ref Address, d : output Word) : bit
iter_next(i : ref Address, d : output Word) : bit
clear(a : Address) : void
update_from(m : ref Mem_model) : void
get_mem_size() : int
Static_mem_model
Sparse_mem_model

Figure 4.3.: Class diagram for instruction level testing. These classes model the state of memory.

the `Address` variable to the next location and returns that value. If there is no next value, the function returns 0. The `update_from` member uses the iteration interface of another `Mem_model` implementation to copy all defined memory locations over to itself.

The `Static_mem_model` implementation returns a single default return value for all locations. Writes have no effect. On the other hand, `Sparse_mem_model` uses SystemVerilog associative arrays to store memory values. Such a sparse representation is necessary if large memory spaces are to be used for testing. Otherwise, a 4 GiB array would be necessary for the 32 bit address space. If a location is read that was previously not written, it does not exist in the associative array and zero is returned as default value.

Executing the test The abstract `Instruction_loader` class (Figure 4.4) defines member functions to load instruction and state to the design under test, and to retrieve the current processor state. The testbench defines a derived class that knows the hierarchical names of processor internal registers and uses those to directly set and get the registers. The `Predictor` class (Figure 4.4) provides a single member function returning the expected result state for a given instruction and initial state. It is basically a software re-implementation of the processor logic.

To perform single instruction testing the following sequence of actions is taken:

1. Initialize main memory to zero.

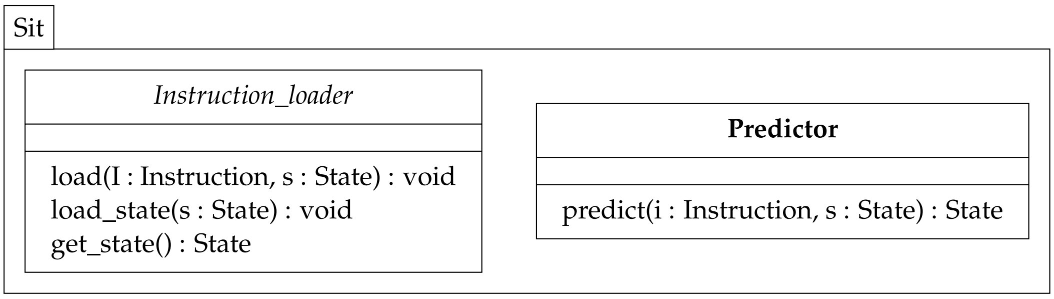

Sit
Instruction_loader
load(I : Instruction, s : State) : void
load_state(s : State) : void
get_state() : State
Predictor
predict(i : Instruction, s : State) : State

Figure 4.4.: Class diagram for instruction level testing. Here, classes for setting and retrieving the state of the design under test, and to predict the result of individual instructions are shown.

2. Generate a random initial state using the `Rand_state` class.
3. Load the initial state to the processor under test using `Instruction_loader`.
4. Generate a random instruction using `Rand_instruction`.
5. Predict the expected result for this instruction using `Predictor`.
6. Write the instruction to main memory.
7. Release reset.
8. Wait until the processor enters the sleep state or a maximum number of clock cycles has elapsed.
9. Read the result state back using `Instruction_loader`.
10. Compare expected and actual result.
11. Generate a report indicating success or failure.
12. Repeat.

This process is repeated until sufficient coverage of the design is achieved.

4.3. Constrained random verification: instruction sequence testing

Instruction level testing is a special case of the more general instruction sequence testing. Here, not only a single instruction is executed in the testbench, but a randomly generated program consisting of a fixed number of operations is used. For a pipelined

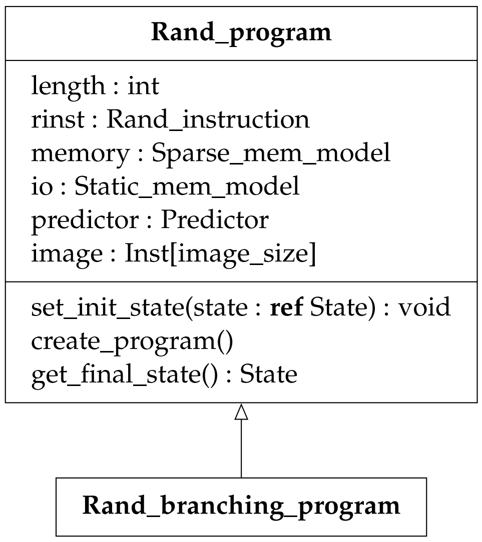


Figure 4.5.: Class diagram for automatic program generation.

processor design, such as the PP, it is not sufficient to test only the correctness of single instructions. The state of the design depends on the sequence of instructions that are currently in execution. For example, logic for resolving interdependencies between instructions can only be tested using sequences.

The basic idea is to use the prediction mechanism subsequently for all instructions in the sequence using the intermittent state as initial state for prediction in each step. This approach requires to also predict the control flow of the program, i.e. the outcome and target of branches, which was not necessary for testing of single instructions. To avoid the halting problem (Turing, 1937), loops in the generated program are forbidden. So branches may not transfer control back to an instruction that has already been executed.

Figure 4.5 shows two newly introduced classes for instruction sequence generation. The `Rand_program` base class produces programs of fixed length without branches. Its derived class `Rand_branching_program` adds the ability to include branches. The length of the program is the number of instructions that are executed. The classes use the previously introduced framework to generate random instructions, model memory and I/O space, and predict the resulting state. The `Predictor` assumes all I/O operations to always read zero and, that a load to a memory location returns either zero or a value that was previously written to the same address. Therefore, the `Sparse_mem_model` and `Static_mem_model` classes are used here.

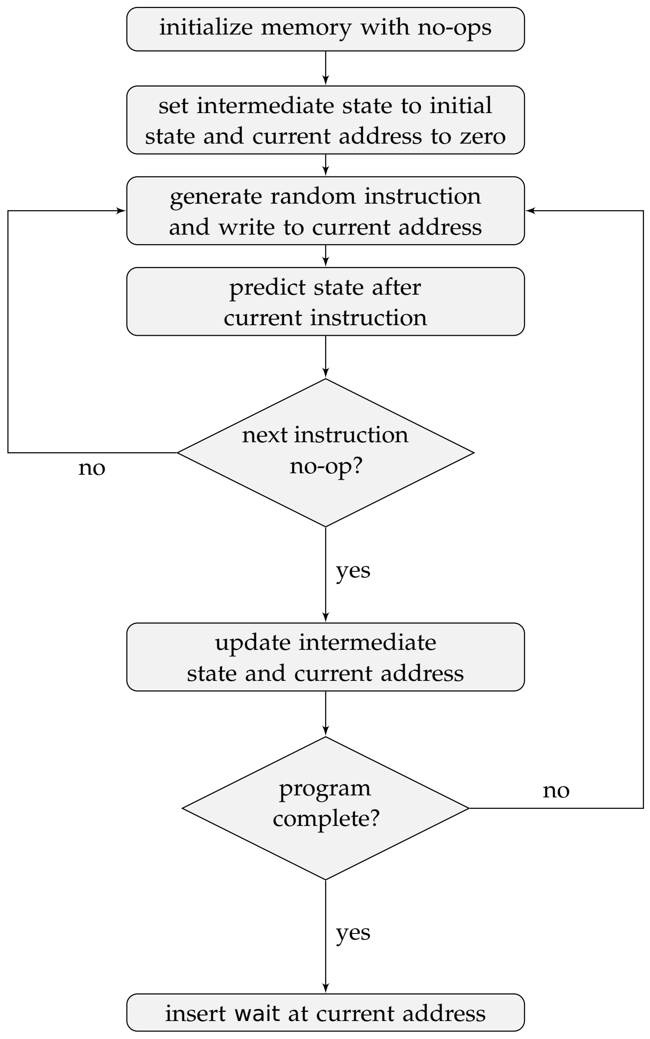

initialize memory with no-ops
set intermediate state to initial state and current address to zero
generate random instruction and write to current address
predict state after current instruction
next instruction no-op?
no
yes
update intermediate state and current address
program complete?
no
yes
insert wait at current address

Figure 4.6.: Flowchart visualizing the algorithm for automatic test program generation.

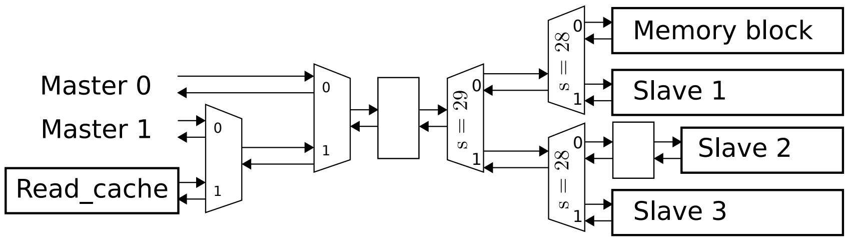


Figure 4.7.: PPB topology used for bus verification. The third master is a cache module presented in Section 3.1.5. The memory block slave internally uses the PPB to RAM interface adapter shown in Section 3.2.4.3. Other slaves are implemented as generic bus targets that can be configured for different accept and response latencies individually for read and write. They act as memory, always return either zero or one, or return the address.

4.3.1. Automatic program generation

The algorithm for automatic program generation is visualized with a flowchart in Figure 4.6. The generated program is stored in the member `image` of the generator class. Randomization constraints on `Rand_instruction` configured by the `addr_-space_size` member ensure, that a branch will not jump to a location outside of the image. With the predicted program counter the generator checks the image location of the next instruction address. If it does not hold a no-op, the current instruction is a branch starting a loop. The testbench then draws new instructions until one is found, that does not create a loop. This process is repeated until the sequence consists of a number of instructions given by the `length` member of the generator classes. Finally, a wait instruction is inserted to mark the end of the sequence for the testbench.

4.4. Verification of the plasticity processor bus

Unfortunately, it is not possible in SystemVerilog to dynamically generate module hierarchies. Therefore, one can not systematically generate random topologies for testing, without resorting to additional tools. Although, it would certainly be possible to solve this problem using M4 and the bus description mini-language introduced in Section 3.2.5, I did not implement such a system. Instead, the testbench uses a fixed topology trying to cover the most common cases. There are three masters and four slaves as is shown in Figure 4.7. It simulates randomly generated parallel accesses through the bus by multiple masters.

Read/write test The first test is a classical memory test: The testbench first writes a block of random data to an address range, then reads the result back and compares to what it had written. This is done in parallel on masters zero and one for disjoint address spaces. The used address spaces are mapped to the memory block and slave one.

Odd/even test Now the same address space within the memory block is used, where one master accesses even addresses the other odd ones. One word of data is written and immediately read back and compared from a random even or odd address. A random number of waiting cycles is introduced after the read and the write to desynchronize both clients. Intervals follow a Poisson distribution.

Cache test This is a memory test that writes a block of data to the memory block via master one and then reads it back through the cache using the RAM interface. Simultaneously the testbench performs the even half of an odd/even test via master zero.

4.5. Writing software for the plasticity processor

The interface between hardware and software is defined by the Instruction Set Architecture (ISA). This chapter lists additional conventions going beyond this specification, such as calling conventions or memory space organization. These conventions are optional and can be changed without having to modify the hardware.

I used the toolchain around the GNU Compiler Collection (GCC) (Stallman, 2012) to generate binary memory images for the processor. The targeted Application Binary Interface (ABI) is the PowerPC Embedded ABI (EABI) described by IBM (1998a).

To support the instructions of the SYNAPSE special function unit the assembler source code needs to be patched. Since SYNAPSE instructions use pre-defined instruction formats, the new operations can be easily added to the opcode table of the assembler. Since the opcode table is shared for all tools of the binutils package (Free Software Foundation, 2013), this also enables correct disassembler output.

Shell for C programs A small shell of assembly code provides a minimal environment for the actual C program. It contains the interrupt vector table and initializes the stack pointer in general purpose register 1 according to the EABI specification (IBM, 1998a). It initially points to a 64 bit stack frame at the end of main memory. Upon a function call the prologue of the function will save the LNK register to this frame. The shell branches to an externally defined symbol `start` using the linking branch instruction bl. So the entry point to the C program is a function declared `void start()`. When the program returns from this function, the shell enters and indefinite loop with a wait instruction in the body. This will put the processor into the sleep state.

If it wakes up for example due to an interrupt event, it will return to sleep after the interrupt service routine has completed.

The interrupt vector table consists of branch instructions to externally defined service routines. When linking the program, it must be ensured, that the shell code is placed at instruction memory address zero. Otherwise, the interrupt vector table will not be at the correct location.

Linking Since there is no dynamic linker all programs are linked statically. The GNU is Not Unix (GNU) linker uses so called linker scripts to further control how the executable is generated. The linker script defines the memory space and how sections for code and data are arranged in the memory space. For systems with two disjoint memory blocks for code and data the convention is to use addresses $0 \ldots 2^{30} - 1$ for instructions and $2^{30} \ldots 2^{31} - 1$ for data. If the system has a joined memory space, the whole range is used for instruction and data with code starting at address zero. The linker script is therefore specific for the target system. It also contains the exact amount of available physical memory.

The compiler normally places variables in a number of different sections that are optimized for specific usage patterns or underly different access permissions if there is a Memory Management Unit (MMU). For the PP this is not necessary and so the linker script combines them into the .text and .data sections.

For interrupt handling the shell requires external symbols for interrupt service routines to be defined. If the C source code does not define such functions, a linking error is caused. The linker script offers a way to avoid this, by defining the required symbols automatically, if they are not present in the source. The default address is the reset vector, restarting the program if an unhandled interrupt is taken.

Standard library The C standard library offers an interface to functionality provided by the operating system. For example, there are functions for file I/O and printing text to a terminal. Also, dynamic memory management is handled by library functions. I did not port a C standard library to the PP, because its usefulness would be strongly limited by the constraints of the BrainScaleS wafer-scale system. For example, the concept of files or terminal input and output have no direct equivalent in the system. Dynamic memory management could probably be of use, but the very limited amount of available memory in the range of few kilobytes makes it less attractive. A dynamic memory management system would introduce memory overhead by adding code and a data structure to track allocated memory blocks.

If in the future a C standard library is needed, the newlib library (Vinschen and Johnston, 2013) is an ideal candidate. It is targeted at embedded systems and offers a modular design that is easy to port to a new architecture. The lack of a C library also implicates that it is not possible to run programs written in C++ on the PP.

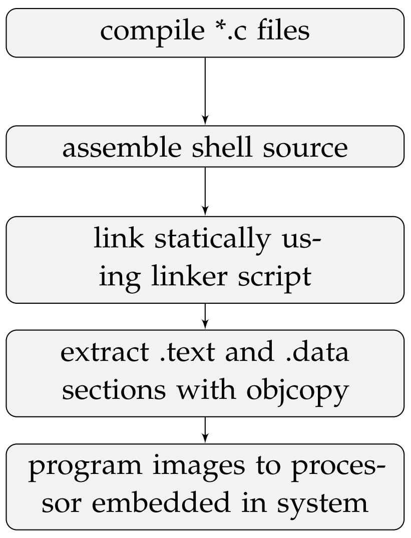

compile *.c files
assemble shell source
link statically using linker script
extract .text and .data sections with objcopy
program images to processor embedded in system

Figure 4.8.: Workflow for programming the processor starting from the source code.

4.6. From source code to program execution

Figure 4.8 illustrates the process of executing a program within a hardware system starting from the source code. The source for the program and the shell is translated into binary object files and linked to an Exectuable and Linking Format (ELF) executable (Zucker and Karhi, 1995). The binary data of the text and data sections are then extracted using the objcopy tool of binutils. This gives raw, binary data files that can be loaded to the processor in the system using a system specific control software.

5. Hardware systems and their evaluation

This chapter introduces four hardware systems using the technologies presented in Chapter 3 and evaluates them using different performance metrics. The first system is an FPGA based prototype platform mainly used for evaluation of the design of the Plasticity Processor (PP). The second system is the BrainScaleS wafer-scale system with the non-programmable STDP implementation from Section 3.3.2. The third one is a prototype ASIC using a 65 nm process technology to test the plasticity processor. The last system presented here is the BrainScaleS wafer-scale system with embedded PP. This system has not yet been manufactured and so only simulation results are presented.

5.1. FPGA prototype

During development the design of the processor was constantly evaluated using an FPGA based hardware platform. I used the ML505 evaluation board (Xil, 2008) that is equipped with the Virtex-5 110LXT FPGA (Xil, 2009b) at the slowest speed grade 1. The SystemVerilog hardware description is synthesized with Synplify by Synopsys (Syn, 2012). For low-level implementation, and the generation of a programming bitfile I used the Xilinx software tools (Xil, 2012b).

5.1.1. Benchmarking with CoreMark

The CoreMark benchmark (EEMBC, 2012) was already mentioned in the context of functional verification in Section 4. It tests four commonly used algorithms: linked list manipulation, matrix operations, state machine operation, and computation of a CRC32 checksum of results. The benchmark repeats these tests for n_{iter} iterations and measures the number of clock cycles n_{cyc}. Using the clock frequency f_{clk} of the processor this gives the performance metric

$$C_{\mathrm{perf}} = \frac{n_{\mathrm{iter}} f_{\mathrm{clk}}}{n_{\mathrm{cyc}}} \tag{5.1}$$

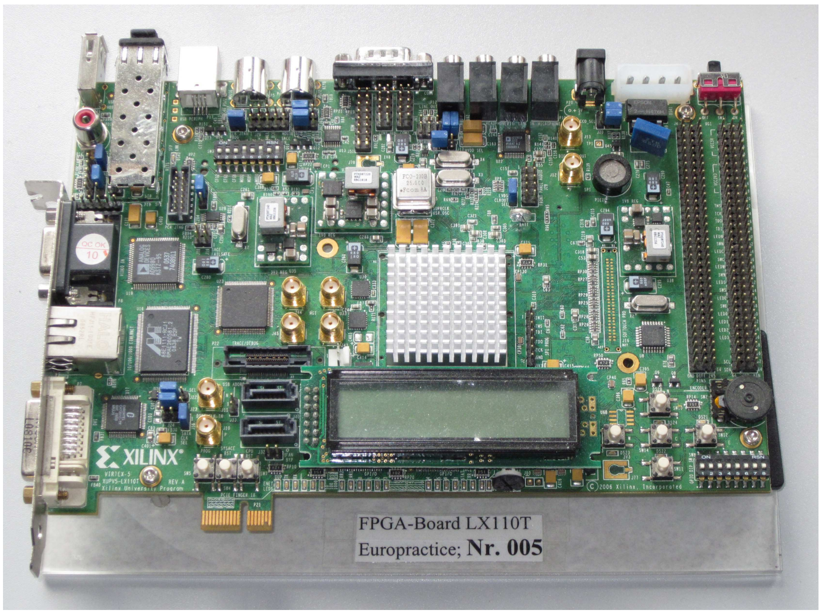


Figure 5.1.: Photo of the ML505 evaluation platform using a Virtex-5 FPGA by Xilinx (Xil, 2008).

measured in iterations per second (s^{-1}). To measure efficiency of the design, this metric is normalized with the clock frequency in MHz

$$C_{\text{eff}} = \frac{C_{\text{perf}}}{f_{\text{clk}}} \cdot 10^6. \tag{5.2}$$

C_{eff} is a dimensionless number. In contrast to performance evaluation on computers with preemptive multi-tasking, there is no random element in program execution for the tested system. Results do not show trial-to-trial variation for identical parameters.

These two metrics are now measured for different configurations of the processor design to find performance critical options. Figure 5.2 shows the structure of the system. First, a disjoint memory configuration is used with fixed-latency load/store operations (Figure 5.2A), then the variable latency variant is tested (Figure 5.2B). Table 5.1 lists all parameters that are modified during the analysis.

Porting CoreMark to the plasticity processor In order to execute the CoreMark benchmark on the PP some adaptations to the code have to be made: Methods for time measurement and text output need to be supplied. The timer facility described in Section 3.1.10 provides the 64 bit time base register clocked with the processor core clock. This is used to measure start and stop time of benchmark execution. For text output one could use the serial port of the evaluation board, for example through

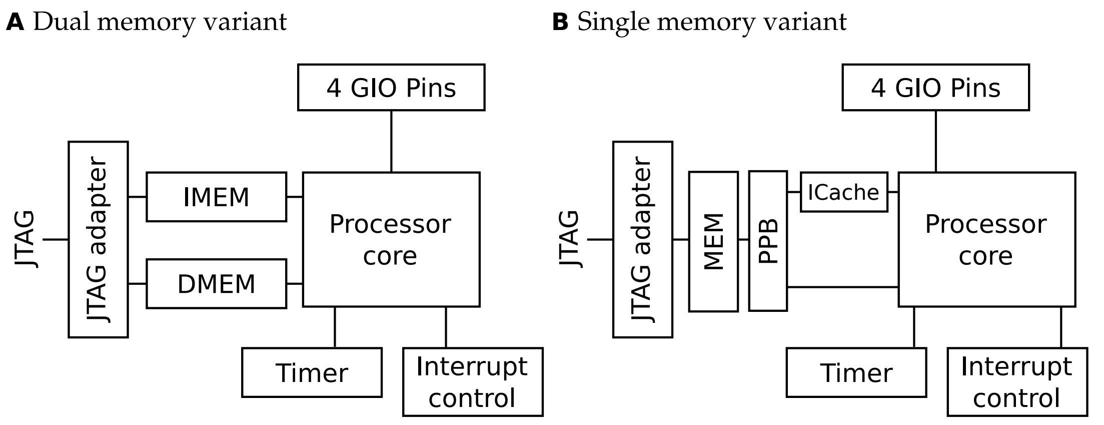


Figure 5.2.: Structure of the design used for CoreMark evaluation. (A) Distinct memory blocks for instructions and data representing a Harvard architecture. (B) One memory block with instruction cache (ICache) for code retrieval. Data accesses use the PPB, which is shared for instruction fetching and load/store operations.

Option	**Description**
f_{clk}	Processor clock frequency.
L_{LS}	Issue to retire latency of load/store instructions.
L_{MUL}	Issue to retire latency of integer multiplication.
L_{DIV}	Issue to retire latency of integer division.
L_{def}	Issue to retire latency for all other instructions except branches and wait.
s_{b}	The branch prediction table has $2^{s_{\text{b}}}$ entries.
p_{b}	Implementation of the branch cache: either direct-mapped (DM) or fully-associative (FA).
s_{i}	The instruction cache has $2^{s_{\text{i}}}$ entries (always direct-mapped).
s_{d}	The size of a cache line in the instruction cache is $2^{s_{\text{d}}}$.
p_{wt}	Whether the write-through optimization is used ($p_{\text{wt}} \in \{\text{true}, \text{false}\}$).
p_{it}	Whether instructions are issued in time. ($p_{\text{it}} \in \{\text{true}, \text{false}\}$).
s_{acc}	Number of entries in the bus access module (Section 3.1.8.1).
s_{bus}	Number of in-fligh requests in bus building blocks (Section 3.2.3).

Table 5.1.: List of parameters that are modified during the analysis using the CoreMark benchmark.

No.	f_{clk} [MHz]	L_{LS}	L_{MUL}	L_{DIV}	L_{def}	s_{b}	p_{b}	s_{i}	s_{d}	p_{wt}	p_{it}	s_{acc}	s_{bus}
1	50	4	7	29	4	8	DM	0	0	n	n	-	-
2	50	var.	var.	var.	var.	0	-	0	0	n	n	-	-
3	50	3	4	29	3	0	-	0	0	var.	var.	-	-
4	50	3	4	29	3	var.	DM	0	0	y	y	-	-
5	50	3	4	29	3	var.	FA	0	0	y	y	-	-
6	50	4	3	29	2	6	FA	10	4	y	y	16	16
7	50	6	5	13	3	4	FA	var.	var.	y	y	16	16
65	500	4	15	38	4	0	-	0	0	n	n	-	-
180	62.5	6	9	29	3	4	FA	7	4	y	y	16	4

Table 5.2.: List of configurations used for tests reported in this section. The abbreviation "var." indicates, that this parameter is varied by the test. Letters "y" and "n" stand for yes or true and no or false, respectively.

the external control I/O port. However, to keep the setup simple I instead only copy the output to an internal buffer in memory. Control software then reads the data memory back through the Joint Test Action Group (JTAG) interface after the program has finished.

Some further options control how CoreMark performs the benchmark: Initially a number of seed parameters are set to control how the test algorithms are performed. It is important, that the compiler does not know these parameters. Otherwise, it might remove the code under test and give results that do not represent processor performance. Therefore, the program retrieves these seeds initially either from command line arguments, from a system function, or from memory locations marked in a way, that prevents removal of the code. Since there is neither a command line nor an operating system, I selected the latter method. Also, the benchmark is configured to use static memory allocation, since there currently is no software support for dynamic memory management.

5.1.2. Influence of compiler optimization options

The compiler translates the source code into machine code and has a strong impact on performance. I used the GNU compiler collection (GCC, Stallman, 2012) in versions 4.5.0 and 4.6.3. The performance difference between programs produced by both versions is negligible (data not shown).

The first question was, which options passed to the compiler give the best results.

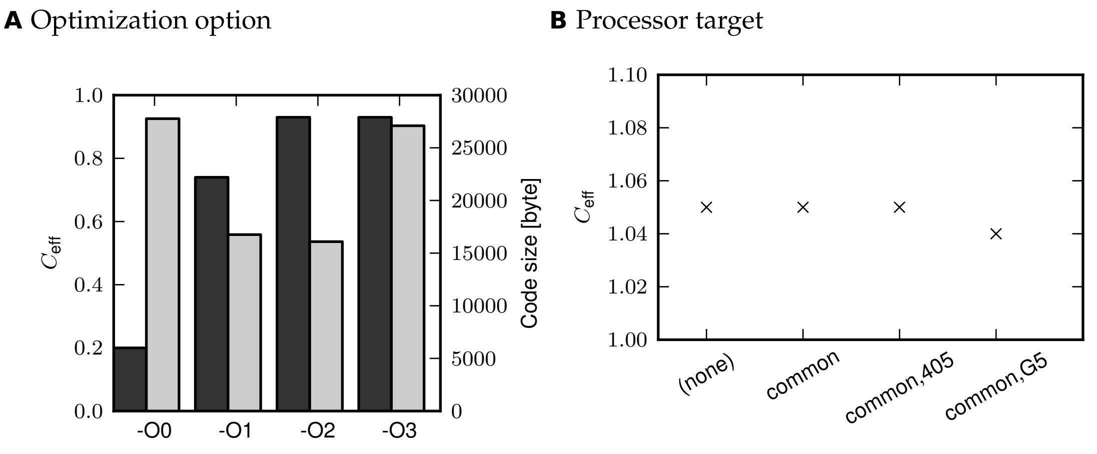


Figure 5.3.: (A) The diagram shows code size in gray and CoreMark efficiency C_{eff} in black, when using different optimization options for the compiler. The highest optimization option does not further increase performance, but results in larger code images. (B) Performance depending on the selected processor target. The plot shows results for giving neither -mcpu nor -mtune, -mcpu=common, and -mcpu=common with different -mtune. The used configuration is listed in Table 5.2 under number 1.

The used configuration of the PP is given in Table 5.2 under number 1. The command line switches "-O0" through "-O3" enable increasingly more code optimization techniques by the compiler. A list of these techniques is given by Stallman (2012). Figure 5.3A shows efficiency C_{perf} and size of the code image in dependence on the optimization level. The data show that the best performance is reached for levels two and three. However, for level three the code is much larger than for level two. In the remaining tests the benchmark is therefore compiled with option "-O2".

The GNU compiler knows instruction latencies for a large number of processors and uses them to schedule operations in the generated code. To find out whether it would be useful to add such latency information for PP, I tested performance with different latency information for other processors. The command line options used are "-mcpu=common" to select a generic processor architecture and "-mtune=<*type*>" to select latency information for a specific processor type. Figure 5.3B shows performance C_{eff} for some combinations of these options. The specifically selected processor types are the PowerPC 405 (IBM, 1998b) and Power G5 (e.g. the 970, Rohrer et al., 2004). Only for the G5 target is performance minimally reduced. This indicates only a small influence of different latency models on performance. Therefore, I do not expect larger performance gains by making the correct latency information available to the compiler.

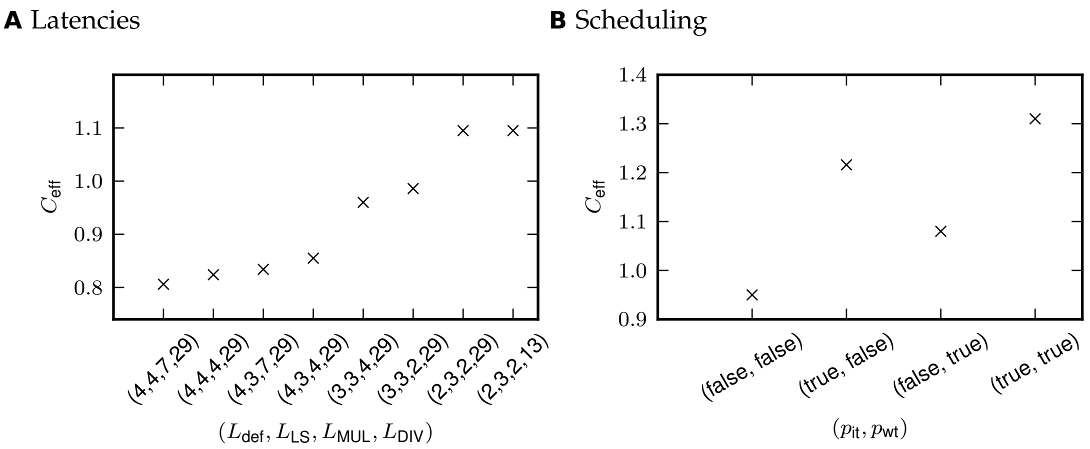


Figure 5.4.: (A) Performance results for different instruction latencies. Tuples show the combination of instruction latencies for load/store L_{LS}, integer multiplication L_{MUL}, integer division L_{DIV}, and all other instructions L_{def}. Configuration is given under number 2 in Table 5.2. (B) Performance results depending on whether instructions are issued in time $p_{\text{it}} = \text{true}$ and whether write-through is used $p_{\text{wt}} = \text{true}$. Configuration under number 3 in Table 5.2.

5.1.3. Influence of issue to retire latency

Figure 5.4A shows the impact of latency on performance in the CoreMark benchmark. The parameters used for the processor design are shown in Table 5.2 under number 2. The number of cycles n_{cyc} required to execute the program is given by the average clock cycles per instruction c_i and the instruction frequency f_i for the classes of operations default (def), load/store (LS), multiply (MUL), divide (DIV) $i \in \{\text{def}, \text{LS}, \text{MUL}, \text{DIV}\}$:

$$n_{\text{cyc}} = \sum_i c_i \cdot f_i \tag{5.3}$$

In the given architecture, $c_i = 1$ if all operands are immediately available. Otherwise, the instruction computing the operands incurs a penalty P depending on its issue-to-retire latency L, and scheduling configuration options p_{it} and p_{wt} (see Section 3.1.7). Latencies for individual instruction classes are given as L_{def}, L_{LS}, L_{MUL}, and L_{DIV} in Table 5.2. The penalty ranges between $P_0 = L$ and $P_2 = L - 2$. Runs in Figure 5.4A use $p_{\text{wt}} = \text{false}$ and $p_{\text{it}} = \text{false}$, so that $c_i \in [1, 1 + L]$. With the given pipeline structure, L can not be reduced below two: one cycle is taken by the register file for operand fetching, and one for writing the result back (see for example Figure 3.11A).

Performance increases with decreasing latency: The largest increments are achieved by reducing L_{def} first from four to three and then to the minimum two. This is to be expected considering Equation 5.3: L_{def} affects cycles per instruction c_{def} for the largest number of instructions f_{def}. Therefore, it has the largest impact on n_{cyc} in Equation 5.3. The second largest impact on performance have load/store instructions. Comparing the first three points in Figure 5.4A, one sees a larger improvement of C_{eff} for the reduction of L_{LS} from four to three, than for the reduction of L_{MUL} from seven to four. Note, that L_{LS} is reduced only by 25 %, while L_{MUL} shrinks by over 40 %. Latency of load/store can not be optimized further in this system, because at least one cycle for a synchronous read on the memory is required. Using the PPB based load/store unit, the latency is even higher (CoreMark results shown later). The minimum latency configuration $(L_{\mathrm{def}}, L_{\mathrm{LS}}, L_{\mathrm{MUL}}) = (2, 3, 2)$ achieves an efficiency of $C_{\mathrm{eff}} = 1.095$. The latency of the divider has nearly no impact on performance, as the last two points in Figure 5.4A show.

5.1.4. Influence of in-time issuing and write-through

As discussed in Section 3.1.7, the penalty by data hazards can be reduced by optimizing the scheduling of instructions by the front-end. Instructions can be scheduled in time to read operands immediately after they are available in the register file. This is possible if the operand is the result of a fixed-latency operation. The Result Shift Register (RSR) holds precise timing information for all in-flight instructions. Therefore, the front-end can plan the point of time for issue using that information. Since there is no penalty associated with this feature, there is no reason not to use it. In contrast to that, writing through the register file requires a modification that produces a longer timing path than without the feature. If read and write to the same register occur within the same clock cycle, the written data is forwarded to the read port. This way, the mechanism works similar to a bypass network (Hennessy and Patterson, 2007, p. 147) connecting the output of the source functional unit to the input of the target unit.

Figure 5.4B shows results depending on whether these two features are used. Together, they increase performance by 38 %. Most of this is attributable to issuing in time (p_{it}), as the steps from first to second, and third to fourth point show. Write-through has an effect comparable in size to the reduction of default latency from $L_{\mathrm{def}} = 3$ to $L_{\mathrm{def}} = 2$. When both optimizations are in place, instructions with $L = 2$ do not cause any penalty for data hazards.

5.1.5. Influence of branch prediction

With write-through, in-time issuing, and most latencies at two cycles, the penalty by data hazards is nearly completely removed. The next step to take is to reduce the penalty caused by branches. Figure 5.5A shows performance for different sizes of a direct-mapped branch cache as described in Section 3.1.4. The used configuration is listed under number 4 in Table 5.2. Overall, performance can be improved by 23 % when using a branch prediction table of size $s_b = 9$. Nearly equal performance

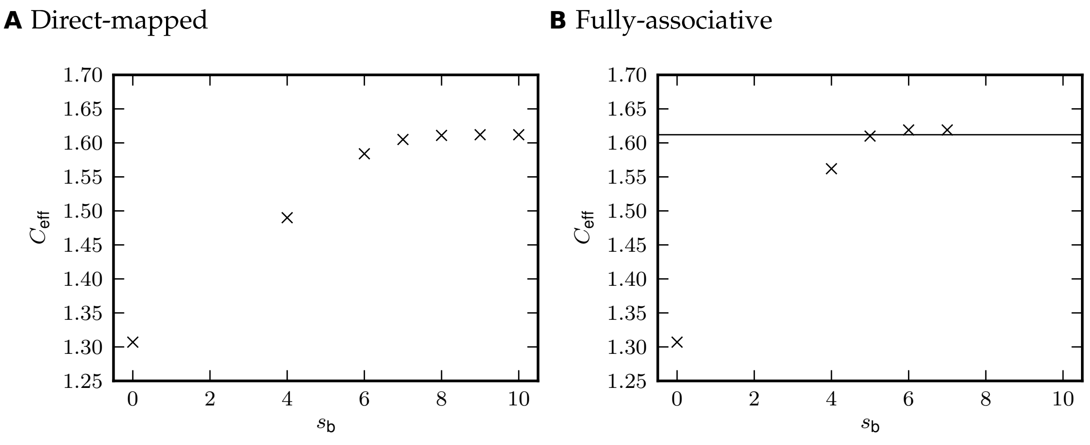


Figure 5.5.: (A) Performance for various branch prediction table sizes using a direct-mapped branch cache. Configuration number 4 in Table 5.2. (B) Performance for different table sizes when using a fully associative cache. The horizontal line marks the best performance when using a direct-mapped cache of size $s_b = 9$. Configuration number 5 in Table 5.2, which differs only in the cache implementation from number 4 used for Figure (A).

($C_\mathrm{eff} = 1.611$ instead of 1.612) is already achieved for $s_b = 8$. An economical choice would be to use $s_b = 7$, which gives $C_\mathrm{eff} = 1.605$ with half the cache size. Please note, that the branch cache has 2^{s_b} entries.

Figure 5.5B shows performance for a fully associative branch cache with otherwise identical configuration. In comparison to the direct-mapped variant, performance increases faster with growing cache size. Already for $s_b = 5$ the maximum level from Figure 5.5A is nearly reached ($C_\mathrm{eff} = 1.610$ instead of 1.612). Larger sizes even slightly surpass the direct-mapped implementation, but not to a large extent ($C_\mathrm{eff} = 1.619$ over 1.612).

These results show, that even with a rather simple branch prediction mechanism, which is completely agnostic of the instruction set, considerable performance improvement can be achieved. The fully associative variant achieves good performance improvements with a relatively small table size.

5.1.6. Influence of variable latency load/store

So far, the tested system used two distinct SRAM memories for code and data, representing a Harvard architecture (Hennessy and Patterson, 2007). This is ideal for performance, because instructions and data can be fetched in parallel without any conflicts. However, memory can not be used efficiently in all cases, since storage sizes are fixed. For example, a program may not fit into memory, although its total size is smaller than the total amount of available storage, because code or data parts are larger than their respective memory block. Since early studies for the BrainScaleS wafer-scale system showed only very small amounts in the order of 10 kiB of memory to be implementable, I decided to use a single main memory. Instruction fetch uses a read-only cache to reduce the von-Neumann bottleneck (Backus, 1978), while load/store directly accesses main memory through the PPB. In this case, collisions between accesses from the cache and the load/store unit can occur. The latency of load and store operations is therefore not known in advance. Also, the need to support arbitrary bus structures enforces load/store instructions with variable latency.

In this and the next section the performance penalty incurred by this architecture is studied. Performance is degraded by two effects: Moving code to the cache blocks data accesses, and the load/store unit with variable latency requires an additional cycle to maintain request and retire queues (see Section 3.1.8.1). Additionally, dependent instructions can not be issued in time, because the front-end does not know in advance, when an operation will finish. Write-through however is unaffected. By reserving a write-back slot at the expected completion time, instructions can retire without longer delays if they finish fast enough. Otherwise, a longer waiting time may be required until no other functional unit uses the write port of the register file. Using the configuration under number 6 in Table 5.2, performance $C_{\mathrm{perf}} = 75.712\,\mathrm{s}^{-1}$ and an efficiency $C_{\mathrm{eff}} = 1.514$ is achieved with variable latency load/store.

5.1.7. Influence of instruction cache

The direct-mapped instruction cache is described in Section 3.1.5. Two configuration options select how the cache is structured: The cache size s_i determines the total number of instruction words 2^{s_i} that can be stored in the cache. The displacement size s_d controls the size of individual cache lines, which is given by 2^{s_d}.

Figure 5.6 shows how CoreMark performance depends on these two parameters. The configuration number 7 in Table 5.2 is used for these measurements. Figure 5.6A varies s_i while keeping $s_d = 4$ fixed. Maximum performance is reached for $s_i = 9$ and is not improved by going to larger cache sizes. There is a relatively large drop from $s_i = 7$ to $s_i = 6$. In the latter case there are only four cache lines and even only two for $s_i = 5$. This means, that for every miss a large fraction of the cache is invalidated, increasing the likelihood that useful operations are pushed out.

This shows, when varying the cache line size while keeping the total size constant. Figure 5.6B shows the dependence on s_d while keeping s_i fixed at seven or ten. One can see, that a reduction from $s_d = 4$ to $s_d = 3$ gains performance, whereas the opposite is true for longer cache lines. This effect depends on total size, since for $s_i = 10$ an increase to $s_d = 5$ does not affect performance.

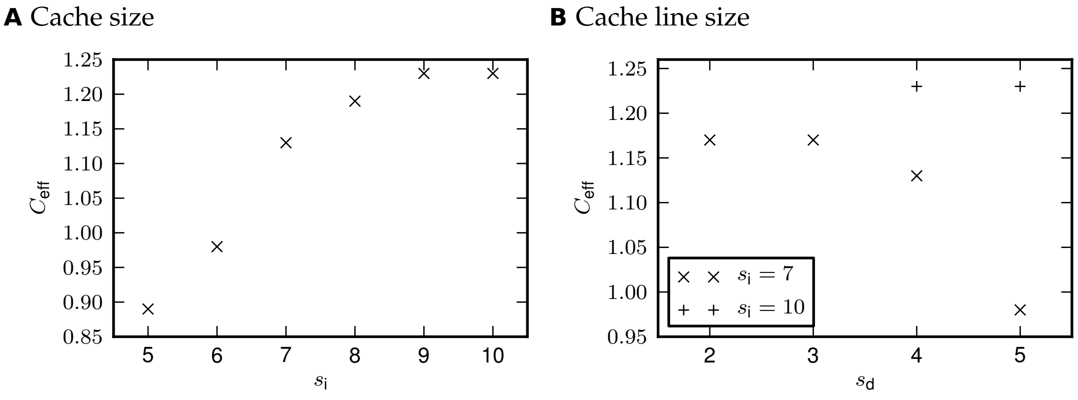


Figure 5.6.: CoreMark performance depending on the instruction cache and cache line size in a system with single main memory. (A) Performance depending on cache size given by parameter s_i. The total number of entries in the cache is 2^{s_i}. (B) Performance depending on line size for two different cache sizes. The cache line has 2^{s_d} entries.

5.1.8. Maximizing performance

To maximize performance C_{perf} in the CoreMark benchmark one can either increase the clock frequency f_{clk} or the efficiency C_{eff}. Those are often opposing goals. When increasing efficiency requires deeper logic paths, this also limits the maximum frequency. The presented processor provides many options to tune efficiency as the previous sections have shown. This section tries to find a maximum performance configuration for the Virtex-5 FPGA. Using different implementation technology, such as the TSMC 65 nm or the UMC 180 nm processes discussed below, will likely require different configurations to maximize performance.

The variant with highest efficiency $C_{\text{eff}} = 1.885$ has minimal instruction latencies $(L_{\text{def}}, L_{\text{LS}}, L_{\text{MUL}}, L_{\text{DIV}}) = (2, 3, 2, 17)$, a fully-associative branch prediction with $s_b = 6$, and uses separate memories for code and data. Figure 5.7 shows how increasing frequency f_{clk} on this design requires decreasing C_{eff}, but improves absolute performance C_{perf}.

Initially, the branch prediction limits frequency. Switching from the fully associative variant to a direct mapped branch cache shortens the critical path, as does decreasing cache size. To reach 70 MHz, additionally the multiplier latency needs to be increased by one cycle. So the 70 MHz design in Figure 5.7 uses a direct mapped branch cache with $s_b = 5$ and increased multiplier latency $L_{\text{MUL}} = 3$. This results in slightly less efficiency C_{eff}, but improved performance C_{perf}. To achieve significantly higher f_{clk}, branch prediction has to be removed completely. Also a further increased multiplier

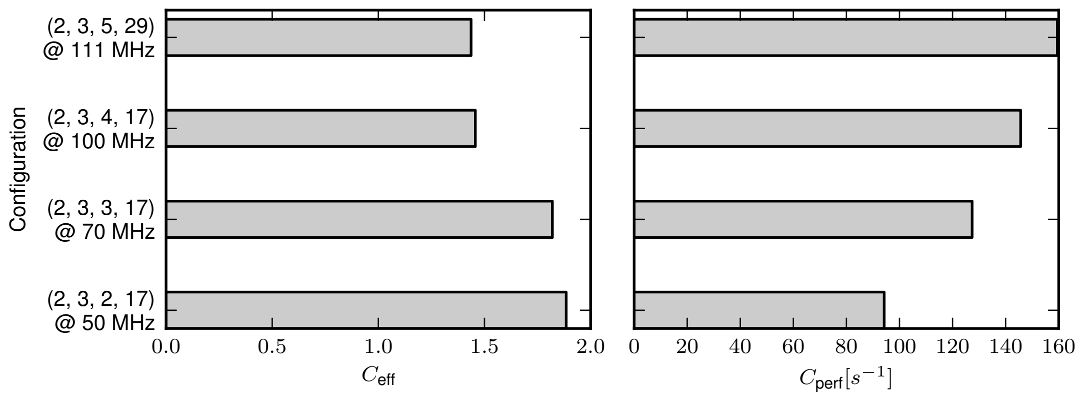


Figure 5.7.: Performance and efficiency for different configurations and clock frequencies. Latencies are given as tuples $(L_{\text{def}}, L_{\text{LS}}, L_{\text{MUL}}, L_{\text{DIV}})$. Absolute performance C_{perf} can be maximized by increasing the clock frequency. To increase the clock frequency, the efficiency C_{eff} has to be reduced by switching from fully associative to direct mapped branch cache. Over a clock frequency of 70 MHz, the branch prediction has to be omitted totally and latencies for multiply and divide are increased.

latency $L_{\text{MUL}} = 4$ is necessary. At $f_{\text{clk}} = 100\,\text{MHz}$ this gives a degraded efficiency $C_{\text{eff}} = 1.457$, but still improved performance $C_{\text{perf}} = 145.668\,\text{s}^{-1}$. With further increased multiplier and divider latencies the design reaches $f_{\text{clk}} = 111\,\text{MHz}$ and a performance of $C_{\text{perf}} = 159.509\,\text{s}^{-1}$. At this point, the critical path lies in the processor front-end, specifically in the dependency tracking and scheduling pipeline stage (Figure 3.2). To reach even higher frequencies, this stage would need to be divided into two. In this case, the front-end knows only after a latency of two cycles whether an instruction is ready for issue or not. This further complicates scheduling logic and would require most likely a major redesign of this part.

5.1.9. Comparison to other processors

By using a standardized benchmark one can compare results to other designs. Figure 5.8 shows results for a selection of other processors compared to the presented design. Data is taken from EEMBC (2013), except for the OpenRISC 1200 processor (OR1200), for which results are taken from OpenCores (2013). For the PP results for four different points in the design space are reported:

(a) Highest efficiency variant with dual memory architecture.

(b) Highest efficiency with single memory and instruction cache.

(c) Highest performance variant.

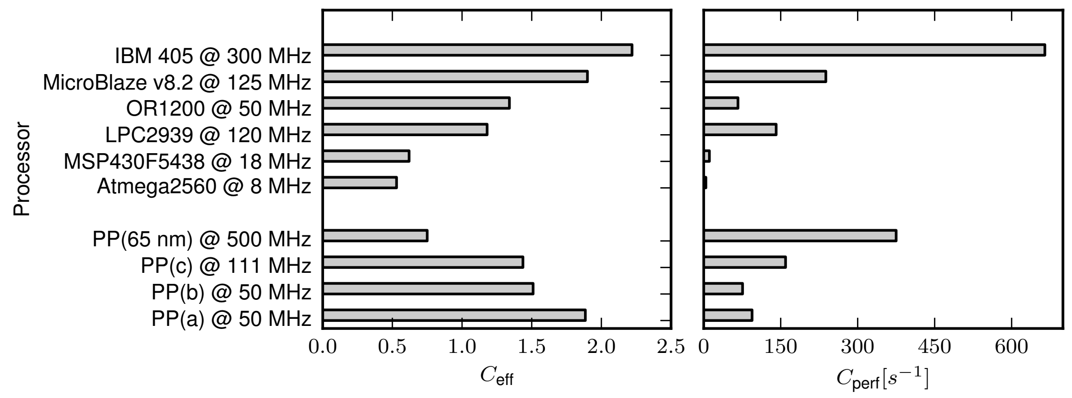


Figure 5.8.: Comparison of performance to other processors. Left: Efficiency C_{eff}. Right: Absolute performance C_{perf}.

(65nm) Performance of the 65 nm prototype variant discussed in Section 5.2.

The 65 nm prototype was produced before I optimized the design on the FPGA platform and thus not necessarily represents the maximal achievable performance in this technology. For example the design was produced without in-time issue (p_{it} = false), which boosts performance without impact on timing (Figure 5.4B). The full configuration is listed under number 65 in Table 5.2. ATmega2560 (Atmel, 2012) is an 8 bit and the MSP430 (Texas Instruments, 2010) a 16 bit micro-controller for embedded applications. They represent the lower end in the performance spectrum in both efficiency and absolute performance. The FPGA based designs are more than twice as efficient and achieve a much higher clock frequency.

The LPC2939 by NXP Semiconductors is based on the ARM968 Intellectual Property (IP) core (NXP Semiconductors, 2010). The same core is used by the SpiNNaker project to simulate large-scale neural networks with micro-processors that are interconnected with an asynchronous event network (Furber et al., 2012; Khan et al., 2008). It implements a Harvard architecture with two 32 kiB memories for code and data, as does the dual memory variant of the PP. The multiplier can only compute 32×16 bit operands and there is no hardware divide (ARM Ltd., 2007). Also, there are only sixteen general purpose registers instead of 32 in the PP. Apart from this, both ISAs are comparable. The PP design surpassed the LPC2939 in efficiency and for variant (c) also in absolute performance on the FPGA.

The OpenRISC 1200 processor is developed by an open source hardware project (OpenRISC Project, 2013). It is a 32 bit scalar processor with 32 general purpose registers and 32 kiB instruction and data caches. Multiplication takes three, division 32 clock cycles. The result shown in Figure 5.8 was also obtained on a Virtex-5 FPGA. Therefore, ISA and implementation platform are similar to the work presented here.

At the same clock speed PP achieves 41 % higher efficiency C_{eff}. A more than twice as high clock frequency can be used, while still staying more efficient.

Xilinx Inc., the manufacturer of the used FPGA, offers the configurable MicroBlaze processor IP core (Xil, 2012a). It is a 32 bit, scalar micro-processor with 32 general purpose registers. The result shown in Figure 5.8 was obtained using a variant configured with a five-stage pipeline and two 16 kiB caches for instructions and data. The full configuration of the tested processor is not reported (EEMBC, 2013), but the design supports single cycle integer multiplication and load with a latency of one. The architecture also supports delayed branch slots, i.e. one instruction following a branch is executed before jumping to the destination address. MicroBlaze achieves a slightly better efficiency $C_{\text{eff}} = 1.9$ compared to 1.885 for the PP. It also uses a higher clock frequency of 125 MHz. This might be due to the use of an FPGA with a higher speed grade.

The PowerPC 405 processor core by IBM implements the PowerISA (2006) in a scalar, five-stage pipeline architecture. Figure 5.8 shows results for a hard-macro co-processor in a Virtex-4 FPGA by Xilinx (Xil, 2009a). The used ISA is nearly identical to the one of PP. Notable exception is the presence of integer multiply-accumulate operations in the PowerPC 405. According to IBM (2005) instructions generally complete in one cycle, while multiplication takes up to three depending on the operand values. The core uses only a static branch prediction scheme, but decides branches as early as possible in the pipeline. Only if a read-after-write hazard for the condition, counter, or link register exists, does a branch take more than one cycle to execute. The PowerPC 405 achieves an efficiency $C_{\text{eff}} = 2.22$ well above the best-case for PP.

This comparison shows the PP design to achieve an efficiency in the CoreMark benchmark comparable or better to industry developed commercial processors. Comparing with MicroBlaze suggests, that there is still room for improving clock frequency of the best-efficiency variant (a). For example, the branch cache – representing the limiting component for higher frequencies – could be optimized for the FPGA architecture to support shorter timings. Better performance of the PowerPC 405 could to a large part be attributable to the multiply accumulate instructions that can accelerate the matrix multiplication task in the CoreMark program. However, also handling of branches as early as possible reduces the penalty by control transfers. This could, to some extent, be transferred to the PP. For example, by deciding branches in the predecode stage as long as there is no outstanding write to a register the transfer depends on. In conclusion, the systematic optimization using the CoreMark benchmark and comparison with results from other designs has helped in improving performance of PP.

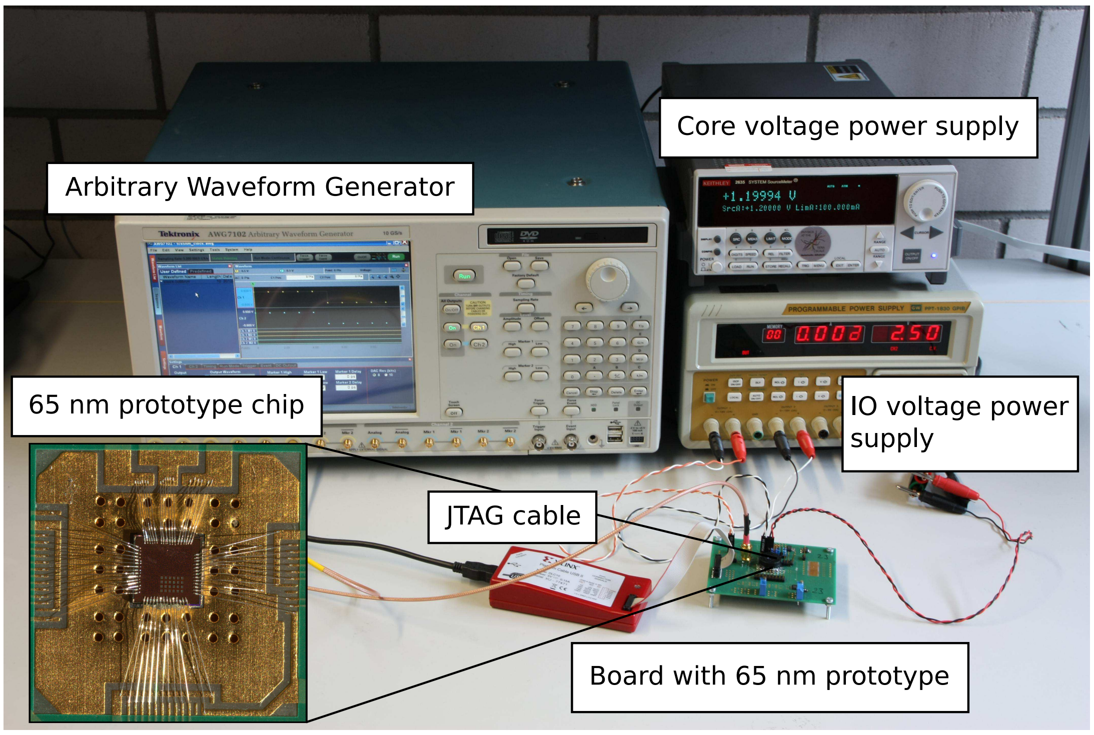

Core voltage power supply
Arbitrary Waveform Generator
65 nm prototype chip
IO voltage power supply
JTAG cable
Board with 65 nm prototype

Figure 5.9.: The photo shows the experimental setup used to evaluate the 65 nm prototype ASIC.

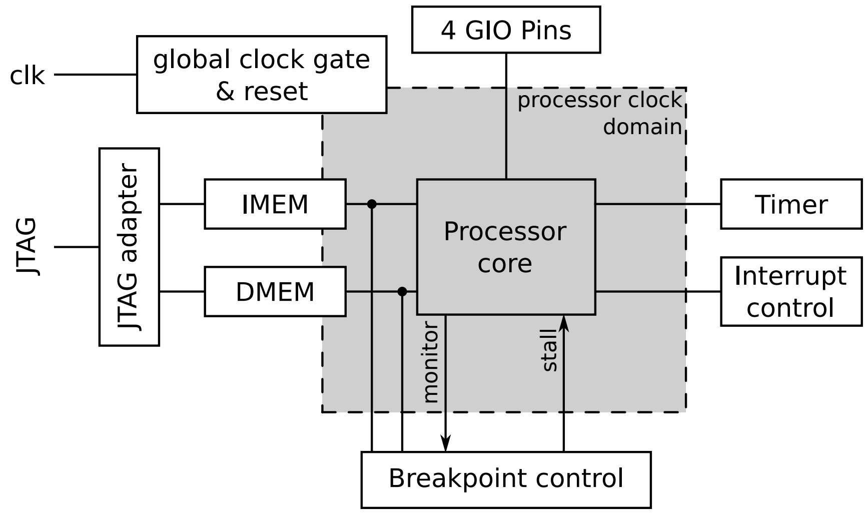


Figure 5.10.: Overview of the design used for the 65 nm prototype ASIC. The shaded area marks the clock domain of the processor under control of the global clock gate.

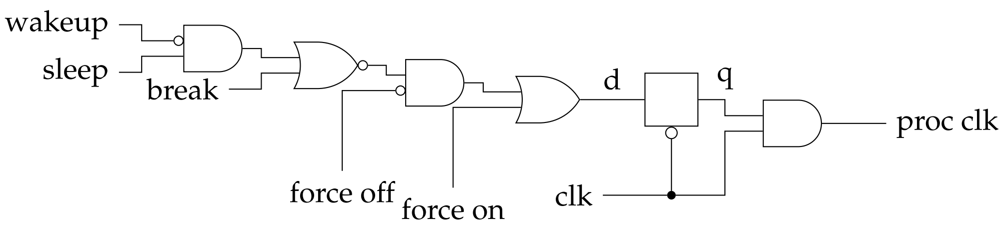
5.2. Prototype ASIC in 65 nm technology

The BrainScaleS wafer-scale system is produced using the 180 nm process by United Microelectronics Corporation (UMC), which is already more than ten years old. The prototype ASIC discussed in this section was produced using the 65 nm low-power process by Taiwan Semiconducator Manufacturing Company (TSMC). Figure 5.9 shows a photo of the system. It was intended to test basic building blocks for future neuromorphic hardware systems using that technology. On the one hand those are analog components, such as analog memories and Digital to Analog Converters (DACs), which are not part of this thesis. On the other hand the PP was integrated to get realistic estimates for reachable clock frequency, area, and power consumption.

5.2.1. Design overview

Figure 5.10 shows the high-level structure of the design. The processor has two disjoint SRAM memories for instructions and data (IMEM and DMEM) that are connected using the RAM interface without an instruction cache. Both memories are 16 kiB, dual-ported SRAM macros provided by TSMC. The second port provides access from outside the chip using a JTAG interface. The processor and the memory ports connected to it lie within the processor clock domain. This clock domain is controlled by a global clock gate and separate reset. The clock gate is primarily intended to turn off the clock when the processor enters the sleep state. Four GIO pins allow for bi-directional signalling with off-chip components (see Section 3.1.11). The pins can also be used to

A Circuit diagram



B Timing diagram

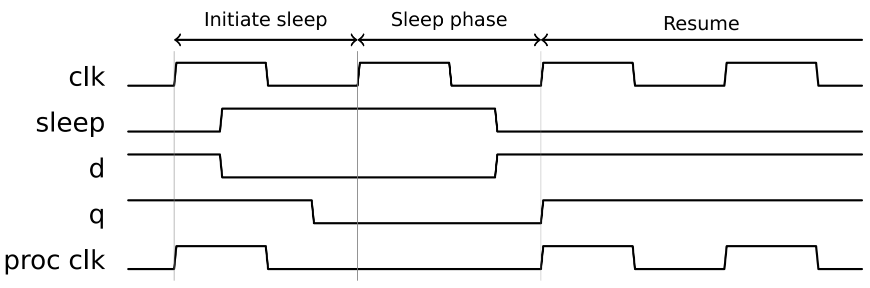


Figure 5.11.: (A) Schematic for the global clock gate. Whether the processor clock is enabled or disabled is determined by a number of input signals that are reduced to the input of a latch in a combinatorial logic tree. The latch is transparent if the clock is low. Its output gates the clock via an AND gate. (B) Timing diagram showing the operation of the clock gate. The row labeled "d" shows the input to the latch, while the one with "q" shows its output.

trigger interrupts. The timer facility provides the time base and an interrupt source as described in Section 3.1.10. It is situated outside of the processor clock domain to allow wakeup on timer interrupts. The breakpoint control unit provides additional debugging capabilities. It can stop execution if a given instruction or memory address is fetched or if a given program counter value is encountered.

5.2.1.1. Global clock gate

Figure 5.11A shows the logic used for the global clock gate. The circuit has to make sure, that the clock to the processor is free of glitches and that the time between rising and falling edge of the gated clock is not reduced compared to the input. Also, there are a number of control signals that have to be prioritized for correct operation of the gate. Otherwise one might build a circuit that can not wakeup from sleep again, for example.

Clock gating is controlled by five enable signals. The processor asserts `sleep`, when entering the sleep state after a wait instruction. The interrupt controller asserts `wakeup` to terminate the sleep state. To halt execution upon hitting a breakpoint, the breakpoint controller asserts `break` to disable the clock. To increase resilience against implementation errors, the `force off` and `force on` signals have the highest priority and override the other control signals. They are set from the outside via the JTAG interface. Especially, if `force on` is set, the processor clock will always remain enabled regardless of all other control signals.

Figure 5.11B shows control of the clock gate in operation: the processor clock is disabled for a single cycle. A latch is necessary to ensure, that glitches from the control input are not propagated to the gated clock. Pulses on the processor clock network that are shorter than half the clock period for which the processor clock domain is constrained can have a detrimental effect to correct functioning. If it causes setup or hold timing violations on flip-flops in the clock domain, this can lead to undefined behavior. The combination of low-active latch and AND gate allow for one clock period of settling time. In the high phase of the clock, the latch keeps its output stable and glitches can not propagate to the gate. In the low phase, the output of the AND gate is always low and so possible glitches are also not propagated. It is important that latch and gate see the clock in the same phase. In the prototype ASIC this was ensured by placing the standard cells for latch and gate next to each other. Behind the clock gate a balanced buffer tree distributes the clock.

5.2.1.2. General purpose input/output pins

The processor can directly control four dedicated pins on the ASIC using the four least significant bits of the GIO registers (see Section 3.1.11). Register GIN reads the logic level on the pins. If the corresponding bit is set in GOE, register GOUT is driven to the pin.

Additionally, the pins are available as interrupt sources. The interrupt configuration and control register (ICCR) configures each pin individually to be either level or edge sensitive, or masked out. Both polarities and edge types can be selected. If the interrupt controller detects a matching event on one of the pins, it signals an external input exception to the processor core, possibly waking it up from a sleep state. The controller signals the exception only if the external enable bit `EE` is set in the machine state register `MSR`.

5.2.1.3. Timer facility and interrupt controller

The timer facility implements the time counter, decrementer and fixed-interval timers as described in Section 3.1.10. It is placed outside of the processor clock domain together with the interrupt controller, so that the timer continues during sleep phases. This allows for wakeup to be triggered by timed events using interrupts. Apart from the already discussed interrupt sources, an external user can trigger a doorbell exception via the JTAG interface. If the external enable bit in the machine state register is set, the interrupt controller signals the exception to the processor core, possibly waking it up.

5.2.1.4. Program suspension with hardware breakpoints

Breakpoints stop the execution of a program, when a specified condition is met. The user can then inspect the state of registers and memory to debug the program. Normally the `trap` instruction and its variants provide this capability. For example, the "debugging" test program of the program level testing suite (see Section 4.1) writes the contents of the general purpose register file to main memory after the last recursion in a recursive faculty calculation using a trap interrupt. However, since this assumes a correctly functioning processor, I added hardware support for breakpoints. The breakpoint controller monitors the instruction and data fetch ports, and the Program Counter (PC) at the input to the pre-decode pipeline stage. When enabled, it disables the processor clock through the global clock gate if one of the following conditions is true:

- The PC matches the instruction breakpoint address.
- The instruction breakpoint address is read from instruction storage.
- The processor writes to the data breakpoint address.
- The processor reads from the data breakpoint address.

The user then can inspect the current instruction word at the input to the pre-decode stage and memory contents. She can resume execution by JTAG command that re-enables the global clock gate.

5.2.1.5. Processor options

The PP core used for the 65 nm prototype uses hardware multiplier and divider from the DesignWare IP library. They have a latencies of $L_{\text{MUL}} = 14$ and $L_{\text{DIV}} = 37$ cycles. For fixed-latency load/store functional unit has a latency of $L_{\text{LS}} = 3$. The default latency for other instructions is also $L_{\text{def}} = 3$. There is neither a branch prediction nor an instruction cache in the system. The instruction cache is not necessary if a dedicated memory stores instructions and would not improve performance. The branch prediction logic was relatively new at the time of the tape-out and not yet sufficiently validated. Since it is an essential part in the front-end of the processor the risk of introducing severe errors by including it was too high. The design was constrained for a clock frequency $f_{\text{clk}} = 500\,\text{MHz}$. All options are summarized in Table 5.2 under number 65.

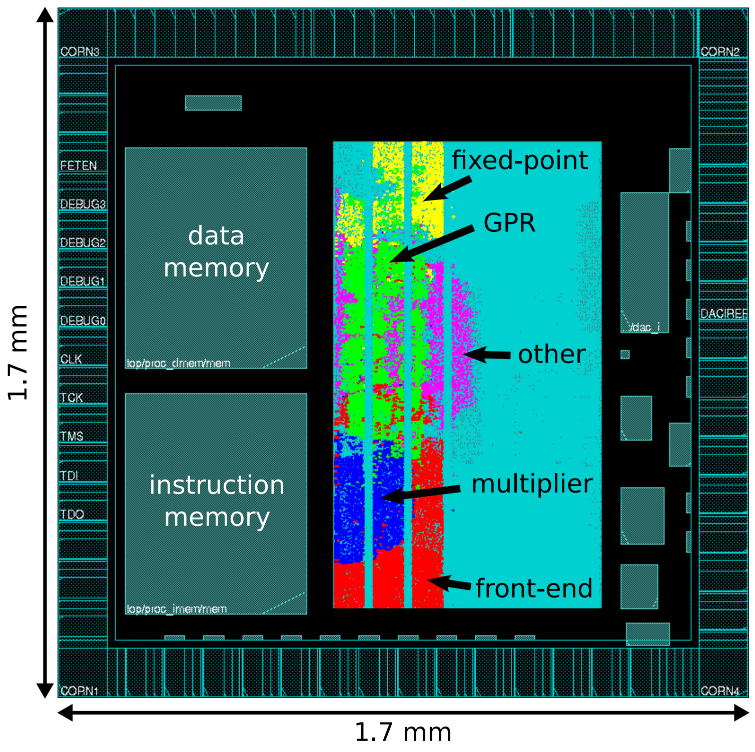


Figure 5.12.: Layout of the 65 nm prototype ASIC with plasticity processor. To the left are two SRAM memory blocks of 16 kiB size. Standard cell logic belonging to the processor is highlighted: front-end (red), multiplier (blue), general purpose registers (green), fixed-point functional unit (yellow). Other standard cell logic and blocks are unrelated to the processor and not part of this thesis.

5.2.2. Implementation and area requirements

The design was synthesized using Synopsys DesignCompiler (Synopsys, 2012) and implemented with Cadence Encounter (Cadence Design Systems, 2012) for the TSMC low-power 65 nm process technology. The implementation flow was created by Hartel et al. (2011). Figure 5.12 shows the layout of the produced chip. Only a small part of the available area is used, with the two SRAM memories and the processor core making up largest fraction of the design. An area breakdown is given in Figure 5.13. Overall, the two memory blocks contribute the largest part to total area. The plasticity processor core requires 0.140 mm^2. The largest part of that is for general purpose registers (Figure 5.13B), followed by the front-end. The front-end contains the instruction tracking logic to detect interdependencies and the control logic to organize program execution (Section 3.1.6). The multiplier is the largest functional unit. It contains a data path for $32 \times 32 = 64$ bit pipelined multiplication with 15 pipeline stages. This results

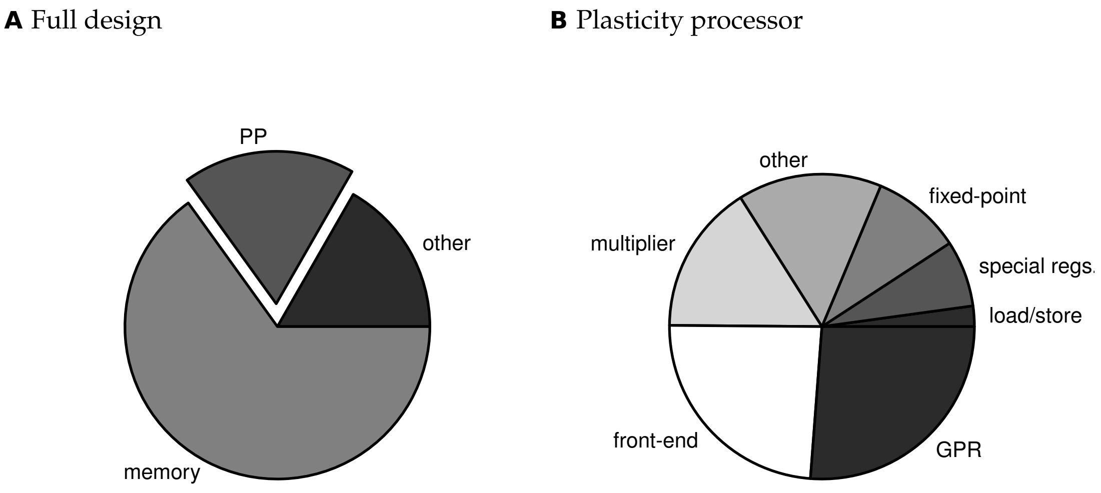


Figure 5.13.: (A) Area distribution for the full design. Most area is consumed by data and instruction SRAM blocks. (B) Breakdown of area requirements for sub-units of the plasticity processor.

in a large number of pipelining registers and arithmetic logic. The divider module is part of the "other" fraction. Its hierarchy was dissolved by synthesis and so an isolated area value is difficult to obtain. The fixed-point functional unit implements the largest number of instructions, but makes up only a relatively small fraction of total area. The "special registers" are additional architecturally defined registers, e.g. the condition or link register. The load/store unit in the dual-memory variant of the PP takes only a negligible amount of area.

These results show, that even with a physically small ASIC ($1.7 \times 1.7\,\mathrm{mm}^2$) complex processor designs can be prototyped. The inner area of the die has a size of $1.4 \times 1.4\,\mathrm{mm}^2$. Assuming a multi-processor system would share the two memory blocks and use the same processor as implemented here multiple times, ten such processors could be implemented on this chip. On the other hand, with one processor the design still offers plenty of space for neuromorphic synapses and neurons to build a fully functional 65 nm neuromorphic prototype.

5.2.3. Experimental results

Statical timing analysis reported no timing violations in the typical and fast corners for $f_{\mathrm{clk}} = 500\,\mathrm{MHz}$. In the slow corner some violations were reported. The corners are parameter sets, e.g. temperature, supply voltage, and process variations, that together affect the speed of the logic gates. In the fast corner, the highest performance can be reached, while in the slow corner speed is degraded. A first test with a produced chip (#23) showed correct execution of the CoreMark benchmark at $f_{\mathrm{clk}} = 596\,\mathrm{MHz}$. The achieved performance was $C_{\mathrm{perf}} = 449\,\frac{\mathrm{Iterations}}{\mathrm{s}}$ with an efficiency of $C_{\mathrm{eff}} = 0.75\,\frac{\mathrm{Iterations}}{\mathrm{s \cdot MHz}}$.

5.2.3.1. Frequency and supply voltage operating range

This section describes a systematic measurement of the operating range concerning supply voltage and clock frequency for three chips. To decide whether a chip is working at a given point within the voltage and frequency space, the CoreMark benchmark is used. Figure 5.9 shows the test setup. The computer remotely controls an Arbitrary Waveform Generator (AWG) to supply the clock frequency f_{clk}[1]. The core supply voltage V_c is supplied by a programmable power supply[2]. The nominal core supply voltage is $V_c = 1.2\,\mathrm{V}$. Another supply powers the I/O cells of the chip with a nominal voltage $V_{\mathrm{io}} = 2.5\,\mathrm{V}$.

The test is performed in the following way: The control computer steps through the voltage range in equally sized steps. For each voltage, the frequencies $f_{\mathrm{clk}} = 50\,\mathrm{MHz}, 100\,\mathrm{MHz}, 200\,\mathrm{MHz}, \ldots, 800\,\mathrm{MHz}$ are tested until the CoreMark test program returns invalid results. Validity of results is determined by the text output stored in the in-memory output buffer. The output contains the exact number of timer cycles and CRC32 checksums of the results, which are compared to a reference that is known to be correct. Between the first failing frequency f_0 and the last successful one f_1, a binary search locates the precise point of failure. The frequency

$$f_t = \frac{f_0 + f_1}{2} \tag{5.4}$$

is tested. If it fails, the process is repeated with $f_0' = f_0$, $f_1' = f_t$, otherwise with $f_0' = f_t$, $f_1' = f_1$. Refinement is stopped after eight repetitions and thus the point of failure is located to within an interval with width

$$\delta = \frac{f_1 - f_0}{2^8}. \tag{5.5}$$

For values over 200 MHz this results in a width of $\delta = 0.39\,\mathrm{MHz}$ in the given setup.

Figures 5.14A to 5.14C shows results for this test for three chips. Figure 5.14D shows only the highest clock frequency passing the CoreMark test for each chip. Overall chip #23 achieves the highest performance. The highest reachable clock frequency was within $769.53 \ldots 769.92\,\mathrm{MHz}$ for $V_c = 1.4\,\mathrm{V}$. In the tested range, f_{clk} depends approximately linearly on the core voltage V_c. The other chips show a less linear dependency. Chip #22 falls below the target frequency $f_{\mathrm{clk}} = 500\,\mathrm{MHz}$ at the nominal

[1]Tektronix AWG7102 Arbitrary Waveform Generator
[2]Keithley 2635 SYSTEM SourceMeter

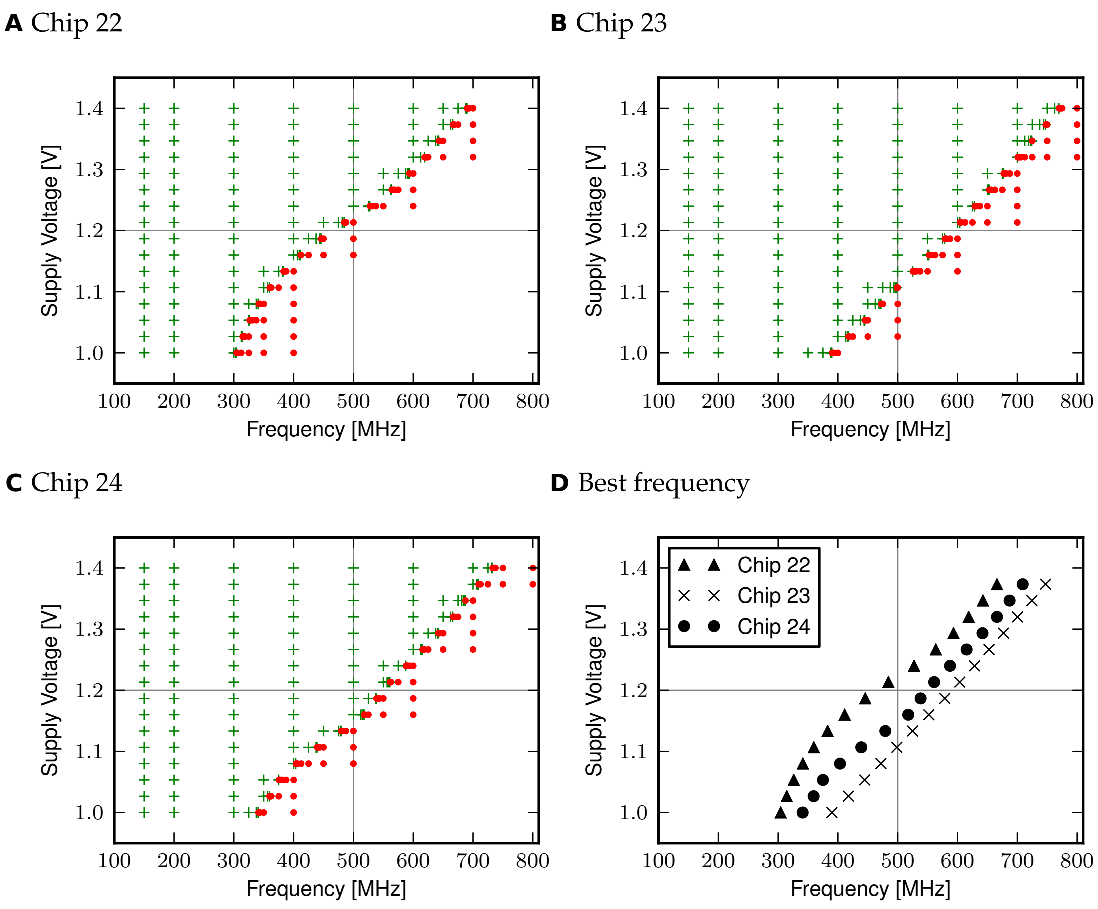


Figure 5.14.: (A)-(C) The plots show in which range of supply voltage and clock frequency three individual chips can be operated. Green plus signs show a successful execution of the CoreMark benchmark with 2000 iterations. Red dots show failure. (D) Highest frequency with successful execution depending on the supply voltage for all three chips in one plot.

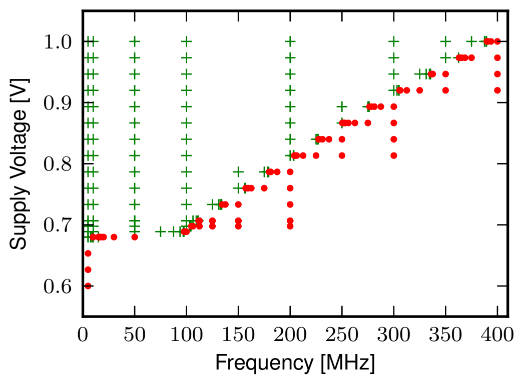


Figure 5.15.: Measurement to find the lowest reachable supply voltage. As in Figure 5.14, green plus signs indicate successful execution of the CoreMark benchmark, while red dots mark failure. Results where obtained for chip #23.

supply voltage $V_c = 1.2\,\text{V}$. At $V_c = 1.213\,\text{V}$ the best achievable frequency is within $484.38 \ldots 484.77\,\text{V}$.

The results show a large range of frequencies to be achievable by varying the supply voltage by only $\pm 200\,\text{mV}$ around the nominal value. Of course, this is paid for with an increased power consumption at higher voltages, for which Section 5.2.3.2 shows measurements. Over the range from 1.0 to 1.4 V, clock speed can be improved nearly by a factor of two. A way to capitalize on this in a future system would be Dynamic Voltage and Frequency Scaling (DVFS), which increases supply voltage to allow for higher frequencies during operation, when there is an increased workload. For typical plasticity programs in the BrainScaleS wafer-scale system with a processor that continuously iterates over the synapse array, this is a less useful feature, since the workload is constant. It is currently being investigated, whether future 65 nm systems could process every spike in software. For such a system, DVFS could be a valuable feature to save energy while still having the capacity to process bursts of events.

To find out how far V_c can be reduced, I tested voltages down to 0.6 V. In order to prevent a large difference between I/O and core voltages V_{io} and V_c, the I/O voltage is reduced to $V_{\text{io}} = 2.0\,\text{V}$ in this test. Figure 5.15 shows the result for chip #23. The same protocol is used as above, but now the initial frequency test starts at 5 MHz. The first successful execution is observed at $V_c = 0.68\,\text{V}$ with a maximum frequency within $15.0 \ldots 15.16\,\text{MHz}$. With higher voltage, the plot shows a sharp increase and then continues linearly as in Figure 5.14B. Concluding from this data there is considerable headroom to reduce supply voltage for power saving.

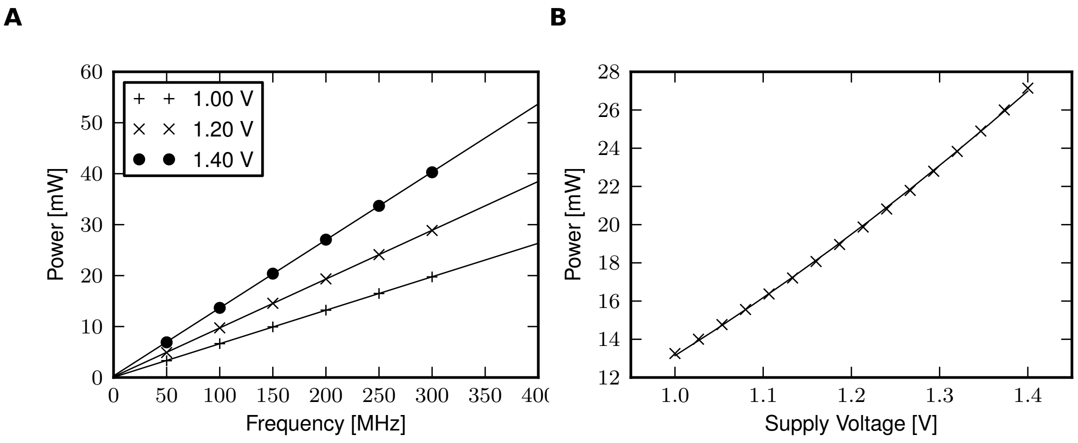


Figure 5.16.: (A) Power consumption measured during execution of the CoreMark benchmark program depending on clock frequency and measured for different supply voltages. The error on power is below 4.6 μW. Data obtained from chip #22. (B) Dependency of power on supply voltage for $f_{\text{clk}} = 200\,\text{MHz}$ on chip #23. Error on power is below 4.8 μW.

5.2.3.2. Power consumption

To get a realistic estimate of power consumption, the current I_c drawn from the power supply on V_c is measured during execution of the CoreMark benchmark. The power supply measures typically eight times within the first five seconds of execution. The number of measurements is limited by time and not by number of samples to assure data is taken before the program is finished and the processor enters the sleep state. The supply itself averages over 25 cycles of the detected power line frequency. With a cycle period of 20 ms, the total measurement takes 500 ms. The standard deviation between the typically eight measurements for one combination of f_{clk} and V_c is taken as error on the power measurement.

Power consumption P_c is caused by charging and discharging capacitances in the chip. It depends linearly on frequency and quadratically on supply voltage:

$$P_c \sim f_{\text{clk}} \cdot {V_c}^2 \qquad (5.6)$$

Figure 5.16A shows power consumption depending on clock frequency for three different supply voltages. The linear relationship between power and frequency is apparent. Fitting results are given in Table 5.3. The quadratic dependency on supply voltage is shown in Figure 5.16B.

Fit	Fitted function	Parameters	
Power-frequency dependency for $V_c = 1\,\mathrm{V}$	$P_c = m f_{\mathrm{clk}} + c$	m	$(6.57 \pm 0.01) \times 10^{-5}\,\mathrm{W/MHz}$
		c	$(5.3 \pm 2.7) \times 10^{-5}\,\mathrm{W}$
Power-frequency dependency for $V_c = 1.2\,\mathrm{V}$	$P_c = m f_{\mathrm{clk}} + c$	m	$(9.58 \pm 0.02) \times 10^{-5}\,\mathrm{W/MHz}$
		c	$(1.5 \pm 0.4) \times 10^{-4}\,\mathrm{W}$
Power-frequency dependency for $V_c = 1.4\,\mathrm{V}$	$P_c = m f_{\mathrm{clk}} + c$	m	$(1.335 \pm 0.003) \times 10^{-4}\,\mathrm{W/MHz}$
		c	$(3.0 \pm 0.6) \times 10^{-4}\,\mathrm{W}$
Power-voltage dependency for $f_{\mathrm{clk}} = 200\,\mathrm{MHz}$	$P_c = a V_c^2 + b$	a	$(1.442 \pm 0.006) \times 10^{-2}\,\mathrm{A/V}$
		b	$(-1.28 \pm 0.01) \times 10^{-3}\,\mathrm{W}$

Table 5.3.: Fit parameters for data shown in Figure 5.16.

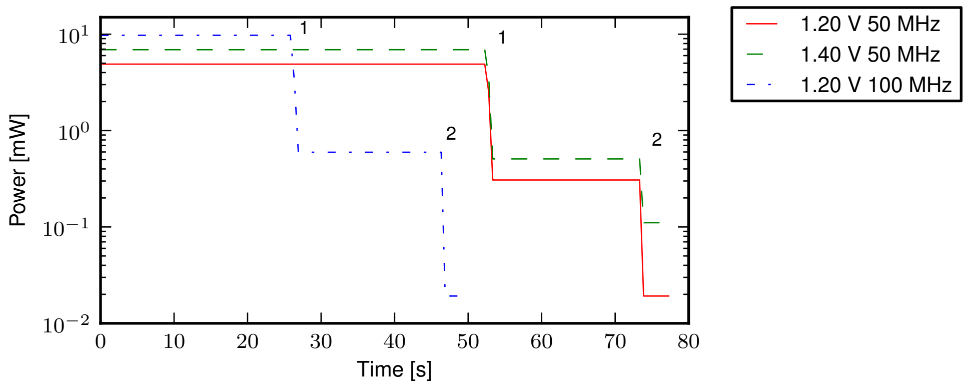


Figure 5.17.: Power consumption during execution of the benchmark. The plot shows the operation of the global clock gate: The processor clock is disabled after the program has finished causing a sharp drop in power consumption (1). A few seconds later, the clock output of the AWG is also disabled, further reducing power (2).

5.2.3.3. Effect of clock gating

Figure 5.17 illustrates the effect of the global clock gate (see Section 5.2.1.1). It shows power consumption during and after the execution of the CoreMark program for different clock frequencies and supply voltages. Initially, the benchmark is running and drawing a constant average current. Upon completion, the `wait` instruction initiates sleep and disables the processor clock. This reduces power by more than an order of magnitude. After 20 seconds the control software disables the output of the AWG generating the clock. This again reduces power consumption by an order of magnitude.

Without external clock input there is no activity in the chip and remaining power consumption is caused by leakage currents. Leakage is voltage dependent: the trace for 1.4 V ends at a much higher level than for 1.2 V. With external clock but disabled clock gate, power is consumed by digital logic outside the processor clock domain (see Figure 5.10). A large portion is probably due to the clock tree distributing the signal over the die area. There is only little logic outside the processor. The power level depends on frequency and voltage as the traces show.

The data presented in Figure 5.11 show clock gating to be an effective method for power saving. It allows drastic savings under software control within few clock cycles. Yet, there is still room for improvement: With disabled processor clock, the only activity in the chip should be related to the timer facility including the decrementer and fixed-interval interrupts. Those are primarily a 64 bit and a 32 bit counter with comparator. The design including the clock tree should be optimized to allow this part to function with minimal power consumption. This could be achieved through a minimal clock network for this part and gating the clock for the rest of the design outside of the processor. The ultimate limit to this optimization is the leakage power level.

Power gating would allow going beyond this limit by shutting off the supply voltage to unused portions of the design. This is more difficult to achieve than clock gating, because flip-flops inside the power gated area lose their state and have to be re-initialized before execution can resume. This could be done in a simple way by saving architectural registers to main memory using a software routine during power shut down. SRAM memory can not be fully de-powered without losing its contents. However, a standby mode with much reduced operating voltage can be used, in which the memory retains its state.

5.2.3.4. Power consumption by individual instructions

The experimental setup used so far can easily be extended to measure the power consumption of individual instructions. The instruction under test is continuously repeated for several seconds by the processor. During this time the current drawn from

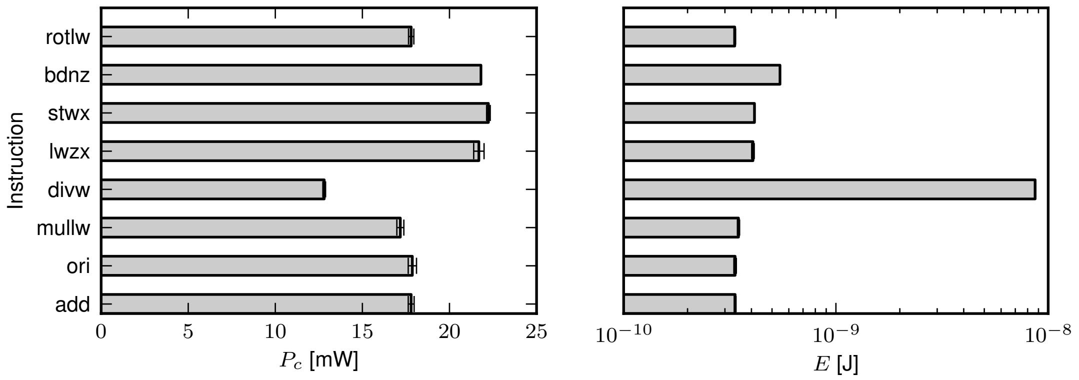


Figure 5.18.: Power consumption measured while executing individual instructions in a loop. The operation under test is repeated 64 times, before the bdnz instruction jumps back to the beginnig of the block. The right plot shows the energy per operation derived from this data. Errorbars indicate the variation from different operand values: where applicable 0xAAAAAAAA and 0 are used as operands. Data obtained from chip #23.

the power supply is measured. To measure power consumed by the instruction under test and not by the code used to repeat that instruction, special care has to be taken: the instruction is repeated in the program 64 times. This block of 64 times the same instruction forms the body of a loop implemented with the bdnz (branch decrementing while non-zero) instruction. The experiment is carried out with $f_{\text{clk}} = 200\,\text{MHz}$ and $V_c = 1.2\,\text{V}$.

Figure 5.18 shows results for a subset of instructions covering all general purpose functional units. add, or and rotlw are operations of the fixed-point unit, mullw uses the multiplier, divw the divider, lwsx and stwx are load/store operations, and bdnz is a branch. The power measurement averages over two sets of operand values for all instructions except bdnz. For or, add, rotlw, mullw, and divw the two operands are either both zero or 0x55555555 and 0xAAAAAAAA. For lwsx and stwx the value read and written to memory is either zero or 0xAAAAAAAA. In case of bdnz there is no sensible way of varying operands.

Power consumption is similar for most instructions. Load and store consume more power as is to be expected for accesses to the SRAM array. Division consumes the least power, because of the used multi cycle trick and the long latency (Section 3.1.6.6): During the division the front-end is halted until the operation is complete. Somewhat surprising is the high power demand by the branch instruction.

The data can be explained by assuming, that power consumption is primarily determined by accesses to instruction and data storage. The multi cycle trick reduces the temporal density of fetches. In contrast, without a branch prediction, control transfers introduce an overhead for fetches: the instruction streamer (Section 3.1.4) treats the branch as not taken an fetches following instructions until the branch functional unit signals the outcome. Therefore, it causes a maximal fetch density.

The plot on the right side of Figure 5.18 shows the energy required for a single operation. To compute this number, the test program measures the time t_{loop} taken for the instruction loop using the timer facility. The energy is then given by

$$E = \frac{P_c \cdot t_{\text{loop}}}{n_{\text{repeat}}} \tag{5.7}$$

with the number of single instruction repetitions n_{repeat}. The data show `add` instructions to require $(3.35 \pm 0.03) \times 10^{-10}$ J. They also show the `div` instruction to be much less efficient at $(8.68 \pm 0.05) \times 10^{-9}$ J. This is about a factor of 26 more energy for division than adding, which is comparable to the ratio of required clock cycles (measured to be 36 in this test). This indicates, that the difference is mainly caused by the different number of clock cycles per instruction for `div` and `add`. The `div` instruction blocks the execution of all other instructions until it has finished (multi cycle trick, see Section 3.1.6.6). This results in a high number of cycles-per-instruction. Assuming that most power is consumed for clocking, this explains the drastic deviation in power consumption for this operation. The assumption is supported by the observation, that pipelined multiplication is comparable in energy to addition, although it algorithmically consists of multiple additions. Thus, cycle count and not the number of involved logic gates drives energy cost. Therefore, one can conclude that pipelining improves energy cost per instruction. More generally, a processor with high computational efficiency per clock cycle reduces the number of required cycles for a given program and therefore is also more energy efficient.

The test program used in this section also demonstrates the capability to resume from the sleep state using the decrementer interrupt. Between repetitions of individual operations, software enters the sleep state using a `wait` instruction. This allows control software, which is measuring the current to detect when instructions are switched. Before entering the sleep state, the decrementer is activated to trigger an interrupt after four seconds. The interrupt service routine increments the save-and-restore register `SRR0` that holds the address of the interrupt instruction, before returning. This causes execution to resume directly behind the `wait`, after returning from the interrupt.

5.3. BrainScaleS wafer-scale system with non-programmable STDP

The currently used generation of the BrainScaleS wafer-scale system uses the non-programmable STDP implementation presented in Section 3.3.2. This section analyzes

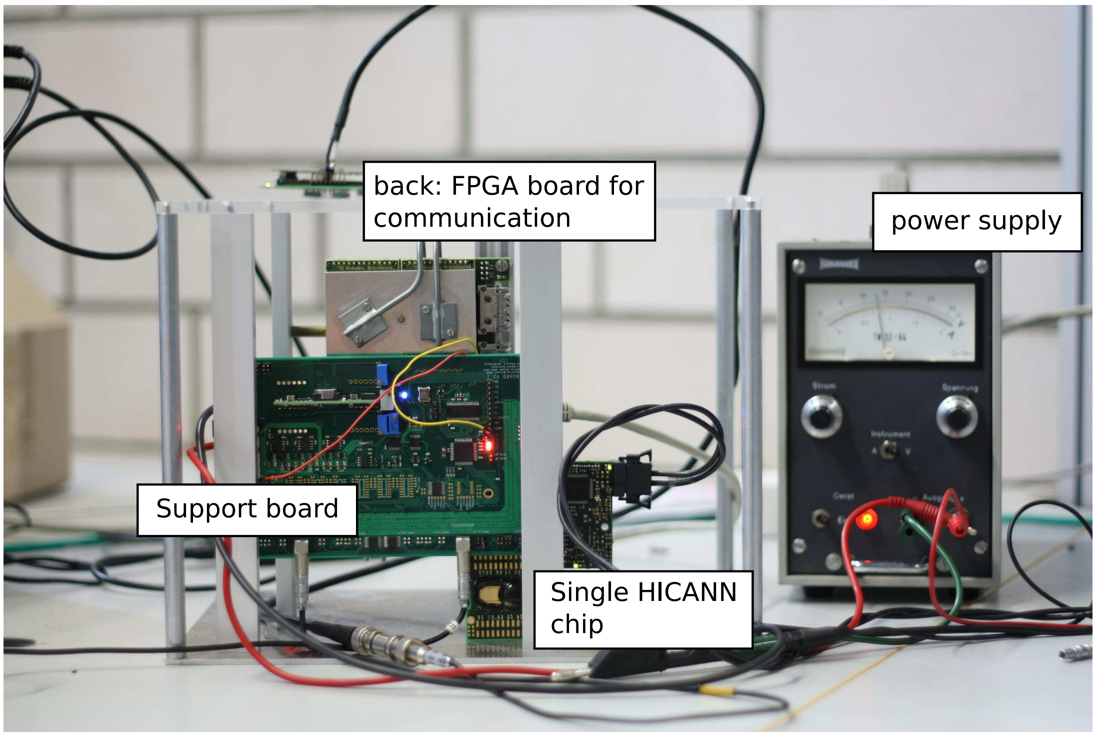


Figure 5.19.: Photo of the experimental setup with single HICANN to measure analog characteristics relevant for STDP.

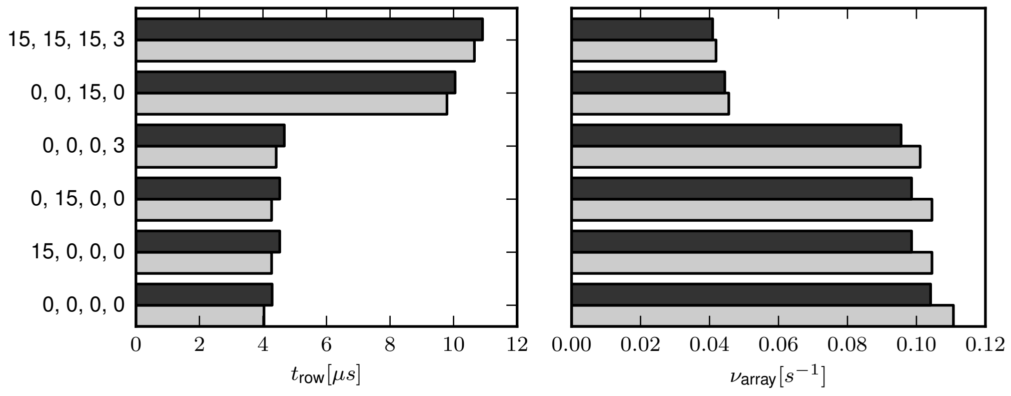


Figure 5.20.: Duration t_{row} for the update of one synapse row in simulation using the non-programmable weight update controller for different timing configurations. The timing configuration consists of the parameters c_{predel}, c_{endel}, c_{oedel}, and c_{wrdel} as they are defined in Section 3.3.2. The controller is clocked at 62.5 MHz. The light bars show performance if resetting of accumulation capacitors is omitted. The right chart shows the same data using the updating rate ν_{array} for the full array of 224×256 synapses in the biological time domain assuming a speed-up $\alpha = 10^4$.

the weight update mechanism first in simulation and then shows results from initial measurements.

5.3.1. Simulation results: updating performance

The simulations presented in this section measure the time t_{row} taken for updating one row of synapses using the non-programmable weight update controller presented in Section 3.3.2. The timing of the update is completely deterministic and independent of synapse weights or evaluation results. The duration shown in Figure 5.20 is the time the `busy` signal of the automatic update controller is high. This is the time starting from setting the `auto` command until the micro-program has finished for the last column set. The duration of weight update scales linearly with the number of synapse rows (data not shown). The four timing parameters c_{predel}, c_{endel}, c_{oedel}, and c_{wrdel} control the number of clock cycles used for pre-charge, word-line enable, driving the output, and driving the bitlines. Except for c_{wrdel} these parameters range between zero and 15. c_{wrdel} is limited to the interval $[0, 3]$. After reset all values are set to their maximum. This way, the user starts with a maximally reliable configuration, but has the option to optimize performance by choosing a faster timing. To optimize performance, the correct values to use have to be determined in experiments, so that read and write access to synapse weights and decoders is error-free. Doing this can reduce the time per synapse row at best to 39 % of the default case according to the top and bottom bars in Figure 5.20. The plots in between show the impact on performance by individual parameters. Parameters c_{predel} and c_{endel} have the same effect. This is evident from Figure 3.18: Both waiting periods occur together and so changing either of the two parameters has the same effect on performance. Because these waiting period occur only once per row, they have only a small influence on performance. Delays controlled by c_{wrdel} and c_{oedel} occur for every column set, eight times per row. Additionally, c_{oedel} is used for the output of the weights and two times for the output of evaluation results. Therefore, it has the highest impact on performance.

For a speed-up factor of $\alpha = 10^4$ one array with $224 \times 256 = 57\,344$ synapses can be updated with a maximum frequency of $\nu_{\text{array}} = 0.104\,\text{Hz}$ in biological time (best-case timing, maximum clock frequency, and use of evaluation reset). For worst-case timing this reduces to $\nu_{\text{array}} = 0.041\,\text{Hz}$.

5.3.2. Verification of automatic weight update logic in hardware

Figure 5.19 shows a photo of the experimental setup used for this test. Individual HICANN ASICs can be produced as single chips and operated in isolation. Since the STDP controller only operates locally on one HICANN, a single chip setup is sufficient for testing.

The first test verifies operation of the digital logic of the weight update controller. The *null read* pattern p_{null} defined in Section 3.3.1.2 allows for control over the analog evaluation result $E^H(V_{tl}, V_{th}, p_{\text{null}})$ by configuring analog parameters V_{th} and V_{tl} (Equation 3.4). Parameter voltages and currents are stored using analog floating gate memories (Millner, 2012; Schemmel et al., 2012). The user specifies a 10 bit digital value that defines the analog value programmed to a memory cell. This value is from hereon given with arbitrary units (arb. unit). The translation between programming value and effective voltage is not necessarily linear. There are saturation effects for high and low values (Kononov, 2011). However, a change of one of the programming value corresponds to about 1.8 mV for a supply voltage of 1.8 V. By setting $V_{th} = 0$ arb. unit and $V_{tl} = 1023$ arb. unit on the respective floating-gate cells, $E^H = 1$ when using p_{null}. With the evaluation circuit returning deterministic results, the outcome of the weight update is determined by the contents of look-up tables. In the given case both evaluation patterns are set to p_{null} so that the combined look-up table $L(1,1,w)$ defines resulting weights. The test uses three tables:

$$L_0(1,1,w) = w \tag{5.8}$$

$$L_+(1,1,w) = \begin{cases} w+1 & \text{if } w < 15 \\ 15 & \text{else} \end{cases} \tag{5.9}$$

$$L_-(1,1,w) = \begin{cases} w-1 & \text{if } w > 0 \\ 0 & \text{else} \end{cases} \tag{5.10}$$

The test initially writes all synapse weights to the same value w_0 and then performs an automatic weight update cycle. The controller is operated in single-shot mode, where it will stop after one iteration over the array. After the cycle is complete, test software reads the final weights w_1 back and compares them to the expected value $L_i\left(1, 1, w_0\right)$. If all synapse weights match the expectation the test passes. This process is repeated for all sixteen possible values of w_0 and all three look-up tables L_0, L_+, and L_-.

Verified for one chip, this test passes for all weights and all synapses. Beyond proving functionality of the digital logic of the update controller, this shows the correct use of the synapse array interface, especially correctness of signal timings. Previously, this was verified by manual inspection of waveforms. The test does not validate operation of the synaptic accumulation circuit and the charge readout. These circuits are based on those used by a previous neuromorphic system and have already been tested in that context (Schemmel et al., 2006; Pfeil et al., 2012).

The test also uncovered a problem in the design of the automatic update controller: The range of weights to process is specified using a first and last row number a and b given in the control register. The update controller should iterate from row a to b and then jump back to a. However, this is not the case: instead of returning to a the controller always resumes at row 0. This is not a severe problem. The consequence for the user is, that plastic synapses can only be configured in a block starting at row 0.

5.3.3. Test of evaluation comparator

The central component of the evaluation circuit is a comparator implementing Equation 3.3. The *null read* pattern p_{null} lends itself to characterize its analog behavior: The parameter voltages V_{tl} and V_{th} are configured using mean voltage V_m and difference variable δ:

$$V_{tl} = V_m - \delta \tag{5.11}$$

$$V_{th} = V_m + \delta \tag{5.12}$$

The evaluation circuit is expected to return bit $b = E^H\left(V_{th}, V_{tl}, p_{\text{null}}\right)$:

$$b = \begin{cases} 1 & \text{if } \delta < 0 \\ 0 & \text{if } \delta > 0 \\ 0 \text{ or } 1 & \text{otherwise} \end{cases} \tag{5.13}$$

The test iterates over all synapses and performs evaluation operations using the `read` command (Section 3.3.1.3). This is repeated N_r times for one value of δ. In each repetition, the analog parameter storage is reprogrammed. Therefore, the evaluation result b includes the programming variation of the floating-gate parameter memory (Kononov, 2011). For each synapse in row i and column j, the evaluation in repetition

$k = 0 \ldots N_r - 1$ returns the result bit b_{ijk}. The repetition average therefore is:

$$\left\langle b_{ij} \right\rangle_{\text{rep}} = \frac{1}{N_r} \sum_{k=0}^{N_r-1} b_{ijk} \tag{5.14}$$

Since there is one comparator for all synapses in one column, the column average is also of interest:

$$\left\langle b_{j} \right\rangle_{\text{col}} = \frac{1}{N_r \cdot 224} \sum_{k=0}^{N_r-1} \sum_{i=0}^{223} b_{ijk} \tag{5.15}$$

Results Figure 5.21 (B)-(C) show $\left\langle b_{ij} \right\rangle_{\text{rep}}$ for all synapses in the top array block of one HICANN. For $\delta = -20$ arb. unit according to Equation 5.13 the expected result is $\left\langle b_{ij} \right\rangle_{\text{rep}} = 1$. This is fulfilled by all synapses in the right half of the chip ($j = 128 \ldots 255$), but not on the left (Figure 5.21A). The left half exhibits a strong columnar structure with a weaker row-wise pattern. At $\delta = 20$ arb. unit all synapses on left and right half meet the expectation $\left\langle b_{ij} \right\rangle_{\text{rep}} = 0$ (Figure 5.21B). For $\delta = 0$ arb. unit the result of the comparison is undefined returning either 1 or 0. Figure 5.21C shows $\left\langle b_{ij} \right\rangle_{\text{rep}} = 0$ for the left half. The right half shows a columnar pattern: for each column the variation of $\left\langle b_{ij} \right\rangle_{\text{rep}}$ is small, but between columns it is larger.

First of all, these results for different values of δ show that left and right side are not equal. The right side is behaving as expected, while the left shows distorted results. Going to larger negative differences $\delta < -64$ arb. unit (data not shown) gives a homogeneous picture again, where $\left\langle b_{ij} \right\rangle_{\text{rep}} = 1$ for all synapses. This indicates, that for columns $j = 0 \ldots 127$ the effective difference $\tilde{\delta}$ is shifted towards negative values. Parameter voltages V_{th} and V_{tl} are stored on four floating-gate cells per array: one set of them for the left and one for the right side. So a difference in their performance could be at the root of the observed behavior. A complete malfunction for the left side can be excluded, since then changing δ would have no or only random effect. Further tests should also vary the side-wise bias parameter for the evaluation circuit (V_{br} in Schemmel et al., 2012). In this test it was set equally for both sides to a medium value of 512.

The second observation is, that the columnar structure visible for example in Figure 5.21C on the right half shows fixed-pattern variations caused by the evaluation circuit due to device mismatch. The result for the individual circuit is highly reliable: the variation within one column is small. Between columns there is variation from the manufacturing process. Figure 5.21D shows the column average $\left\langle b_{j} \right\rangle_{\text{col}}$ in dependence of the difference $V_{th} - V_{tl}$. For the right side (columns $127 \ldots 255$ and therefore lower part in this plot), each column exhibits a clear point in the neighborhood of $\delta = 0$ at which it switches from $b = 1$ to $b = 0$. This is the expected behavior. For the left side

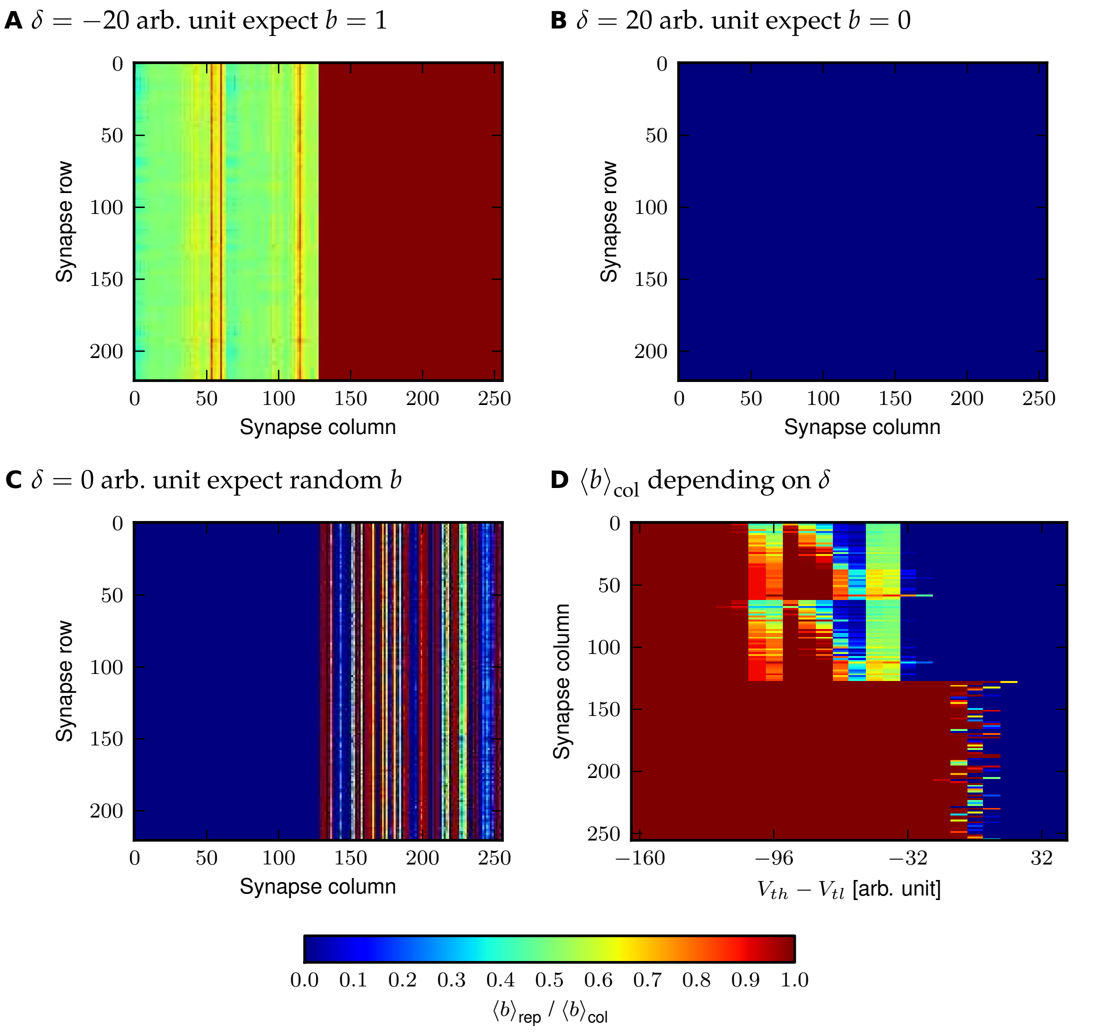


Figure 5.21.: (A)-(C) Measured repetition average $\langle b_{ij} \rangle_{\text{rep}}$ for all synapses at different values for δ. (D) Dependency of column average $\langle b \rangle_{\text{col},j}$ on voltage difference $V_{th} - V_{tl} = 2\delta$. Voltages are given in units of the 10 bit floating-gate programming value.

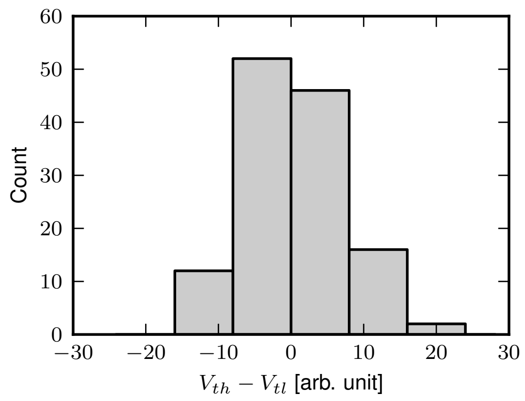


Figure 5.22.: Histogram over the voltage difference at which the comparator in the evaluation circuit switches from $b = 1$ to $b = 0$ in Figure 5.21D when increasing δ. Only columns 128 . . . 255 are shown.

(columns 0 . . . 127) now also a repeating pattern for blocks of 64 columns becomes apparent. This is just the size of one so-called array slice that is repeated four times in the schematic to form the synapse array. Therefore, this structure hints at effects introduced at this level of the design. Peculiar about the observed structure is, that the probability for getting $b = 1$ first dips with increasing δ, before returning to nearly 1 and then decreasing. This means, that the comparator is not just seeing shifted values for V_{th} and V_{tl}, but malfunctions in some other way. Also, there is a pronounced structure between -96 arb. unit $< V_{th} - V_{tl} < -64$ arb. unit showing a dependency on the column number. This hints at a geometrical cause, for example length of wires from one side to the comparator in the column.

Fixed-pattern variation The fixed-pattern variation of the evaluation comparator can not be calibrated for, because there is only one set of parameters V_{th}, V_{tl} per half of the array. Therefore, the magnitude of variation limits the precision of the evaluation process. Figure 5.22 shows the distribution of the switching point over the right side of the array. This point was determined to lie within the interval $[\delta_{n-1}, \delta_n]$ if $\langle b_j \rangle_{\text{col}} > 0.5$ for δ_{n-1} and $\langle b_j \rangle_{\text{col}} < 0.5$ for δ_n. This method works for the right half, but would be problematic for the left side as Figure 5.21D shows. On the left side, there is no clear switching point, but rather a broad range in which results are erratic. The resulting histogram has a mean of (0.5 ± 7.1). If used with an absolute comparison pattern for a threshold $\Theta = 512$ this standard deviation would be 1.4 % of the threshold, which is far below the worst-case values tested for the reward-modulated learning task in Section 2.3.6.

The data presented in this section shows the evaluation circuit and the controller using it to be functional. However, the reason for the erratic behavior of the left side is not evident from the collected data. Before characterizing more chips over ranges of the center voltage V_m and the comparator bias V_{br} it is too early to draw any final conclusions. It is important to note, that the left side is still able to correctly compare for larger differences. Therefore, the observed behavior represents a performance degradation rather than a malfunction. If a learning rule turns out to be sensitive to these effects and if no solution can be found, the left side can not be used for plastic synapses. In this case updates in this part can be prevented by setting the comparison parameters V_{th} and V_{tl} in such a way, that exceeding the threshold is impossible. For example, for the *absolute threshold* pattern $V_{th} = 1023$ and $V_{tl} = 0$ (see Section 3.3.1.2).

5.3.4. Event transmission crosstalk

It was first observed by Kononov (2013), that read and write operations on the synapse array induce crosstalk on the synaptic input of neurons. Synapses signal events to the neuron by a current on a shared line along one column. This line runs in parallel to the bitlines of the weight and decoder SRAM cells over the whole height of nearly 5 mm of one synapse array. When reading or writing, these bitlines are driven – for pre-charge or the data to write – causing a noise signal on the synaptic input through capacitive coupling. Depending on the configuration of the neuron input circuit this leads to effects on the membrane potential of varying magnitude. In first experiments by Kononov (2013) the effect was comparable to real synaptic input from event stimulation. The effect was also confirmed in simulation by Millner (2013). It poses a potentially severe problem for using STDP in HICANN, as the update algorithm periodically reads and writes weights. This might cause a global, periodic, and synchronized disturbance of all active neurons. So far, a systematic investigation of the effect has not been carried out, so that a final conclusion for the functionality of STDP can not be drawn. It might still be possible by increasing the strength of real synaptic input to perform STDP experiments with the existing system. However, observations so far suggest, that a hardware modification is required to reduce crosstalk significantly. This could be done in the next generation system that includes a plasticity processor and is discussed in the next section.

5.4. In preparation: BrainScaleS wafer-scale system with plasticity processor

This section presents results for the BrainScaleS wafer-scale system based on the HICANN with Embedded Plasticity Processor (HEPP) design. The system uses the PP with SYNAPSE special function unit and the synapse array interface adapter presented

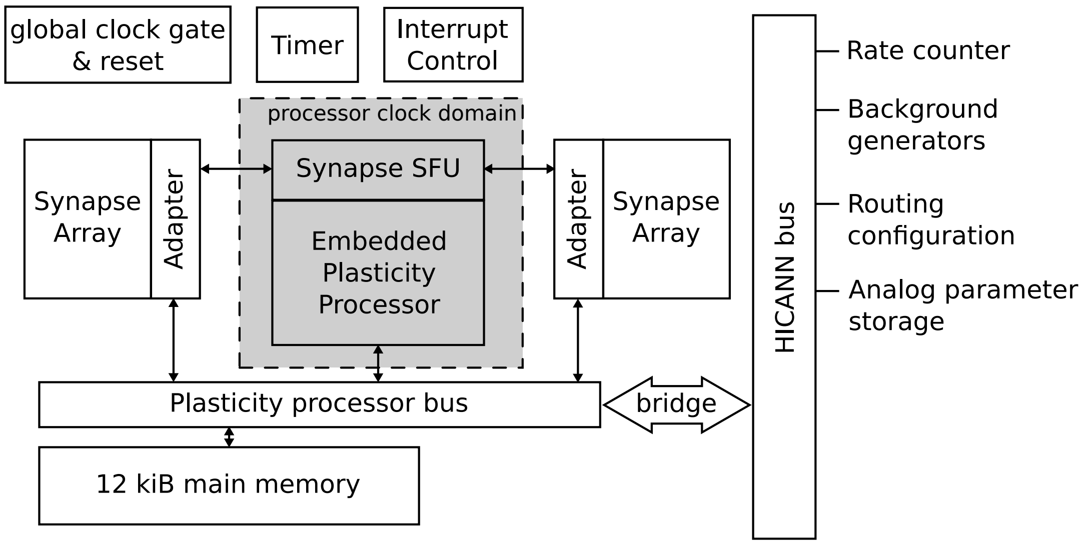


Figure 5.23.: The block diagram shows the structure of the programmable plasticity implementation in the BrainScaleS wafer-scale system.

in Section 3.3.3. It has not yet been produced due to unexpected complications caused by the special organization of metal layers used for the asynchronous event communication system. This is explained later in Section 5.4.4. Here a synthesizable design prepared for tape-out is presented. Further changes are only required outside of this design.

5.4.1. Design overview

Figure 5.23 shows an overview of the structure of the system. It uses the same clock gating mechanism as the 65 nm prototype (Section 5.2.1.1) to save power if the programmable plasticity is not needed for network emulation. A hardware breakpoint mechanism is not planned, since the processor is now considered reliable enough to use internal trap based debugging. Therefore, the `break` enable signal to the clock gate is tied low. The timer and interrupt facilities are also taken from the 65 nm prototype. The GIO port of the processor, which controls pins in the prototype, is not available. Using the bus technology described in Section 3.2 the processor has access to 12 kiB of main memory distributed over three 4 kiB SRAM blocks. Memory is separated into three blocks to satisfy constraints imposed by the arrangement of full-custom blocks and the asynchronous event network. Instruction fetching is cached using a 128 entry directed-mapped cache with cache lines of 16 entries (see Section 3.1.5). An arbitration module, as described in Section 3.2.4, arbitrates access between the processor and the HICANN system bus. The HICANN system bus is divided in two

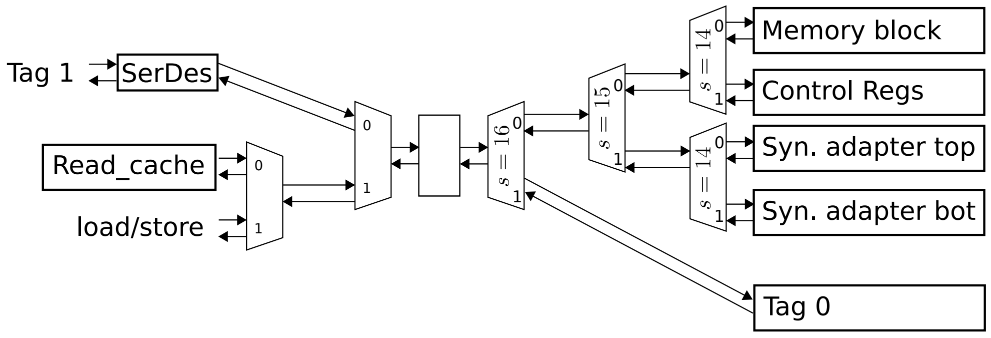


Figure 5.24.: Bus structure of the BrainScaleS wafer-scale system with embedded plasticity processor. The tag 1 HICANN system bus can act as master on the bus. Other masters are the instruction cache of the processor (`Read_cache`), and the load/store unit. The SerDes block is used to reduce the number of required signalling lines over a long distance on the die. Access to tag 0 of the HICANN system bus is provided by an adapter slave (Section 3.2.4.4). Other peripherals are main memory, control registers, and top and bottom synapse array adapter interfaces.

disjoint bus parts referred to as tag 0 and tag 1. The tag 1 part is dedicated to plasticity and can act as master on the PPB. Conversely, the PPB can act as master on the tag 0 bus that interfaces all other components in the system. The user loads a program to the processor or to read and write synapses via tag 1. The plasticity program uses the access to tag 0 for advanced learning rules, for example to reconfigure the network connectivity.

The four peripherals most relevant to plasticity rules in the system are shown in Figure 5.23: rate counters allow for limited measurement of event rates of individual neurons or populations of neurons. Background generators stimulate the asynchronous event network with random or fixed-interval events at configurable rates. The routing configuration for the asynchronous event network is stored in switch matrices distributed over the wafer. The processor can access those matrices local to its die. Analog parameters are stored on floating-gate memories (Millner, 2012) and can be modified between emulation runs. Changing them during network operation is not possible as the programming interfere with neuron dynamics.

5.4.1.1. Bus structure

Figure 5.24 shows the structure of the PPB used by HEPP in more detail. Notable is, that the tag 1 part of the HICANN system bus connects through the Serializer/Deserializer (SerDes) bus component presented in Section 3.2.4.2. Processor logic and the root of the tag 1 bus are situated far apart on the die and the bus has to pass through a constriction caused by the floorplan. Serialization reduces the necessary amount of routing resources for the cost of decreased performance. Since this path is only taken for external requests, a decrease of performance is acceptable.

Both tags of the HICANN system bus use 16 bit addresses, while the PPB uses 32 bit addresses. The first splitter after the delay register stage multiplexes based on bit 16. Therefore, the whole address range of tag 0 is visible to the processor in the range 0x10000 to 0x1FFFF. Also, the external user can reach all peripherals except for the tag 0 adapter, to which he has direct access anyway, using 16 bit addresses.

5.4.1.2. Rate counter

To support learning rules that depend on neuronal firing rates I designed a rate counting facility for the HEPP design. Eight 16 bit counters are integrated into the so-called merger-tree (Schemmel et al., 2012). The merger-tree routes events from local neurons, background event generators, and external stimulation to eight local drivers on the asynchronous event network. Each neuron has a 6 bit source address number depending on its position on the chip. Rate counters are added at the last stage of the tree, before transmission to the network. Each counter n is configured with a mask m and compare address a. If the merger-tree forwards an event for transmission, of which the address b matches a masked comparison to the address, the counter is increased:

$$n \quad \leftarrow \quad \begin{cases} n+1 & \text{if } m\&b = a \\ n & \text{else} \end{cases} \tag{5.16}$$

If currently the counter holds the highest representable number $2^{16} - 1$ while a matching event is detected, n is left unchanged and the overflow bit s_o is set.

Counter, overflow bit, compare address, and mask are accessible through the tag 0 HICANN system bus that is also used for configuration of the merger-tree. Counters are reset by writing to their associated address.

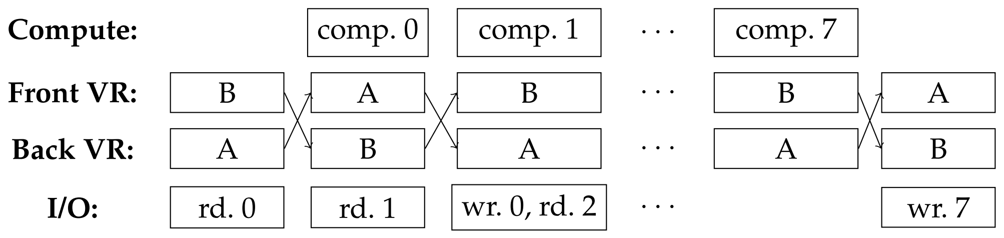
5.4.2. Simulation results: weight updating performance

To measure the weight updating performance of the plasticity processor in HEPP I measured timing information for a collection of benchmarking programs. In the simplest case the updating algorithm of the non-programmable implementation is reproduced (Section 5.3.1). Section 5.4.2.1 describes the programs and Section 5.4.2.2 shows performance results.

5.4.2.1. Benchmark programs

All benchmark programs perform updates for ten synapse rows and measure the required time in clock cycles using the timer facility. The processor is operated at 62.5 MHz and uses the SYNAPSE special-function unit for the update. Results are

A Two-factor update process for one row



B Inner loop source code

```
synswp
synops <write sequence>, <row>, <previous column set>
synm v2, v0, l0
synm v3, v0, l1
syncmpi v1, 0x1

synops <read and evaluate sequence>, <row>, <next column set>

syns v0, v0, v2
syncmpi v1, 0x2
syns v0, v0, v3
```

Figure 5.25.: Two-factor weight update using the SYNAPSE extension. (A) Computation and I/O tasks of the program are drawn as boxes over time. The Front VR an Back VR rows illustrate which half of the vector register file is currently accessible by software (front) or by I/O (back). (B) Assembler source code of the inner loop. Operands given symbolically in angle brackets "<.>" refer to general purpose registers. Instructions are defined in Section 3.4.

reported using the time for one synapse row t_{row} and the updating frequency for one array of 224 rows in biological time ν_{array} assuming a speed-up $\alpha = 10^4$. The same metrics were reported in Section 5.3.1 for the non-programmable implementation.

Two-factor weight update This program is equivalent to the fixed algorithm used by the non-programmable implementation. Weights are updated by a fixed look-up table using the SYNAPSE special function unit. Figure 5.25A illustrates the update process for one row. The program iterates over the eight column sets while overlapping computation and I/O. First, after the row has been opened, column set 0 is loaded to a vector register in the back of the register file. Then, the register file is swapped, and reading of column set 1 is started, while updates for set 0 are computed. After the next swap, the result of the computation is written back, before a read for column set 2 is started. This repeats, up to and including column set 6. For the last set, no new read operation is initiated, and new weights for it are written back after a final swap of the register file.

Figure 5.25B shows the assembly source code of the inner loop of the program, to illustrate how this scheme is implemented with the SYNAPSE special function unit. The code does not show the initial and final part, in which the first two column sets are read, and the last one written. I/O operations are initiated with `synops` instructions, and computation uses `synm`, `syncmpi`, and `syns`.

The limiting factor for updating is the time required for the I/O operations started in lines 2 and 7, and further the waiting time by `synswp`, if a previous operation has not yet completed. Therefore, `synops` has to be placed with sufficient distance to previous instances to avoid waiting cycles. The space in between is filled with computational instructions.

Two-factor dual row weight update This test program is motivated by the results by Pfeil et al. (2012). They tested a synchrony detection task using STDP and simulated the non-programmable weight updating mechanism. They identified the simultaneous reset mechanism of accumulation capacitors in the synapses as cause for degraded performance compared to a system where both accumulators are reset individually. Due to the tight layout of the synapse array it is not possible to add an additional reset line for each column without reducing synapse density (Schemmel, 2012). However, it is possible to simulate multiple reset lines. Synapse rows can be combined to increase resolution of the weights. When doing this, two synapses – one below the other in the same column – see the same input from the network and their output goes to the same neuron. To simulate multiple reset lines the program evaluates capacitor a_+ from the upper row and a_- from the lower row. So when resetting one synapse, information in the other row is retained. Only 4 bit weights are used, which are stored in the upper synapse, while the lower one is set to weight zero.

This dual row operation changes the iteration scheme: instead of processing one row in total before going to the next one, the program has to evaluate column sets from both rows to compute the weight change. Every row switch involves closing and opening operations consuming additional time. Also, due to the partitioned vector register file, evaluation data from the upper and lower rows are stored to different partitions, which are not accessible simultaneously. Therefore, the data has to be moved to the general purpose register file and loaded back again.

Three-factor dual-row weight update A simple case of a three-factor learning rule is the reward modulated R-STDP rule analyzed in Section 2.3. According to Equation 2.28, the increment or decrement of the synaptic weight is multiplied with an external factor, the success signal S. Therefore, software has to change the look-up tables when a change in the success signal is indicated from the external controller. In the learning task of Section 2.3 a non-zero success signal is given at the end of each trial and all synapses are updated once with this success signal.

The three-factor benchmark task computes the new look-up tables itself from the success signal before iterating once over the array. The external controller sends the success signal S[3], upon which the plasticity program multiplies the stored update constant A with S using 32 bit fixed-point arithmetic with 16 bit for the fractional part.

$$\Delta = S \cdot A \tag{5.17}$$

The result is rounded and reduced to a 4 bit representation:

$$\Delta_4 = \mathit{Round}(\Delta) \tag{5.18}$$

With this offset the program computes the two new look-up tables L_0 and L_1:

$$L_0(w) = \begin{cases} w + \Delta_4 & \text{for } w + \Delta_4 < 15 \\ 15 & \text{else} \end{cases} \tag{5.19}$$

$$L_1(w) = \begin{cases} w + \Delta_4 & \text{for } w - \Delta_4 > 0 \\ 0 & \text{else} \end{cases} \tag{5.20}$$

$$\tag{5.21}$$

The actual weight update is then identical to the dual row two-factor benchmark. Note, that if reward is only given at the end of a trial it is not necessary to emulate separate reset lines by using two rows, since all accumulators are reset before starting the next trial anyway.

Three-factor dual-row probabilistic weight update Section 2.3 showed probabilistic updates to give better learning performance for 4 bit weights. The plasticity program can implement this using a software Random Number Generator (RNG). For each synapse with probability p, as defined in Equation 2.6, the bit-pattern 0001 is set in a different vector register. With probability $1 - p$ the field is set to zero. Then the syncmpi instruction tests against the pattern 0001 to set the internal compare state. Now, syns selects, based on this randomly generated compare state, from the two update alternatives. In the test program used here, the two alternatives are: update according to look-up table, or no update at all. To set the patterns randomly for each synapse, the RNG generates a 32 bit number for each position and compares it against a threshold $T = p \cdot (2^{32} - 1)$. If the random number is below T, 0001 is written and 0000 otherwise. I tested three algorithms for random number generation: linear-congruential, XOR-shift, and multiply-with-carry (Press et al., 1992).

[3]This might be signalled to software by using the doorbell interrupt, possibly waking the processor from the sleep state.

Three-factor dual-row 8 bit weight update Even better results than with probabilistic updates can be achieved for higher weight resolutions in the spike train learning task (Section 2.3). In dual-row operation weight data from both rows can be combined to form an 8 bit weight. In this case, the look-up table based arithmetic of the SYNAPSE extension is no longer useful, as it is restricted to 4 bit. Therefore, weight update has now to be computed with the general purpose part of the processor. To do this, weight and correlation data is first moved from vector registers to general purpose registers. Then the new weight is computed by adding the offset Δ, and moved back to a vector register. When moving between vector and general purpose registers, the upper and lower 4 bit part of the full 8 bit weight need to be extracted and packed back again using shift and bit-wise logical operations.

Evaluation scan If the probability of updating is small, the plasticity program can be optimized. Instead of computing updates for all synapses, the program searches for those in need of update. Only if by evaluation such a synapse is found, is the update computed.

The evaluation scan benchmark program only performs a single evaluation operation per synapse. The syncmpi. instruction sets a bit in the condition register, if at least one comparison matches. This bit is evaluated by a conditional branch instruction that jumps to weight update code if necessary.

5.4.2.2. Results

Figure 5.26 shows performance results for HEPP. The used configuration for the plasticity processor is listed under number 180 in Table 5.2. As is to be expected the evaluation scan program shows the fastest updating speed. Two-factor updating takes more then twice the time for one row. Two-factor and two-factor dual-row show nearly the same performance as long as the pre-charge time t_{pre} is negligible. For 15 cycle pre-charge, programs using dual-row updating perform worse than those with single-row updates, because pre-charge is necessary for every column set and not only every row. It is important to note, that the time t_{row} refers to technical synapse rows. In dual-row mode, programs update only half as many logical synapses then single-row programs in this time. The result here means, that the whole system can be operated in single- and double-synapse mode with the same updating speed. However, measured in synapses per second, the latter mode reaches only half the throughput of the former.

The time required for setting the look-up tables in the three-factor program is negligible when considering the update rate of the whole array ν_{array} (Figure 5.26B). Therefore, it is a viable option to configure the tables using the plasticity processor instead of sending pre-computed ones from the external environment.

The comparison with the non-programmable implementation of the HICANN system (Section 5.3) shows a reduction of t_{row} to 58 % for two-factor single-row updating.

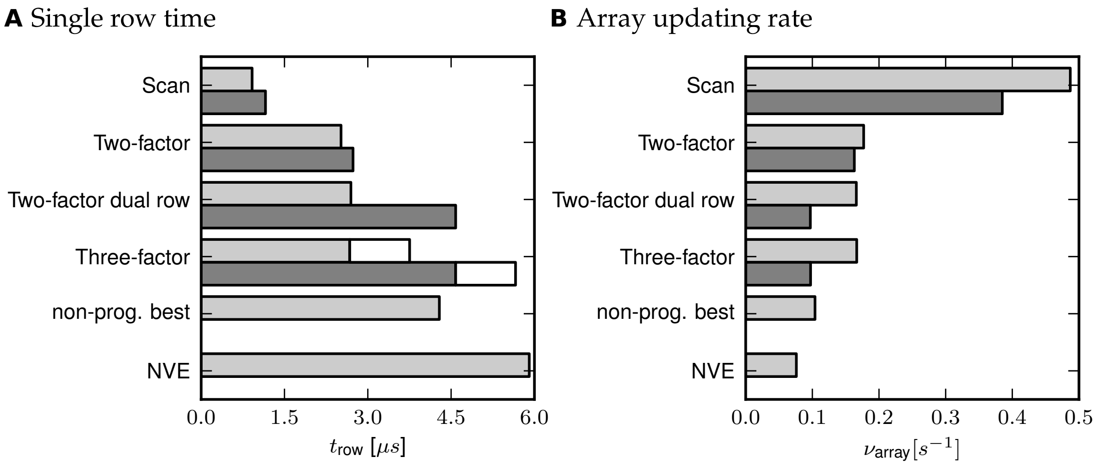


Figure 5.26.: Performance of weight updating in HEPP measured with four benchmarking programs. For comparison the best-case performance of the non-programmable implementation, of updating with the native vector extension (NVE), and without specialized instructions (non-SFU) are also shown. (A) Absolute time for one row of synapses. Best-case performance is shown in light gray. Bars in dark gray show performance when using a 15 cycle pre-charge duration. For the three factor benchmark the white part shows the time for setting new look-up tables depending on the third factor. This is only done once for one iteration over the array. (B) The resulting array updating rate ν_{array} assuming a speed-up of $\alpha = 10^4$.

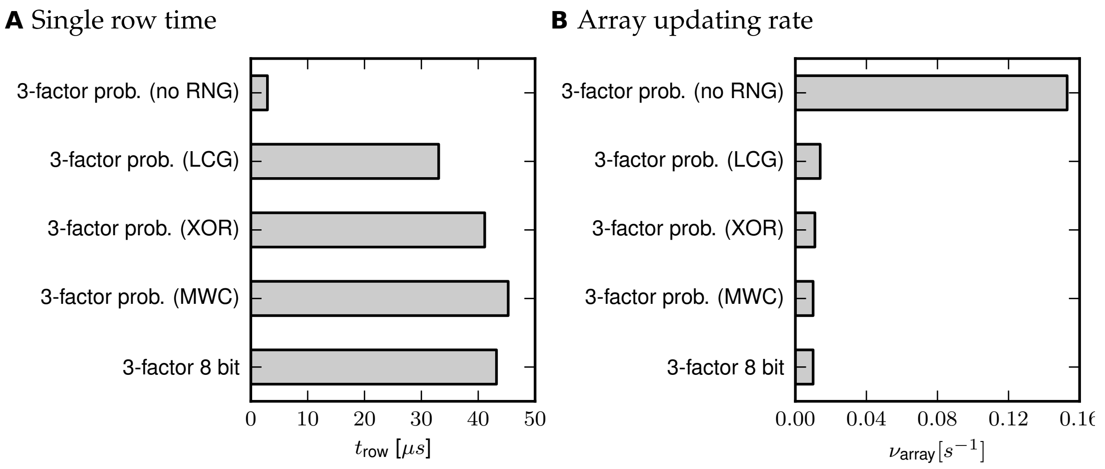


Figure 5.27.: Updating performance for three-factor probabilistic weight updating. The four tested programs use different RNGs: dummy RNG returning a fixed number, linear-congruential generator (LCG), XOR-shift (XOR), and multiply-with-carry (MWC) (Press et al., 1992). (A) Absolute time taken for one synapse row. (B) Array update rate in biological time assuming a speed-up of $\alpha = 10^4$ and 224 synapse rows per array.

This is, for one thing, due to the programmable I/O controller that allows for more aggressive timings at the synapse array interface and, for another, because of parallel weight computation enabled by the SYNAPSE special-function unit.

Figure 5.26 also shows updating speed of the NVE design variant described in Section 3.5. The NVE special-function unit is similar in concept to SYNAPSE: it provides specialized instructions for look-up table based weight processing, and comparison and select operations equivalent to syncmpi and syns. However, instead of dedicated 128 bit vector registers it uses the 32 bit general purpose registers and does not have comparable I/O capabilities as are provided by synops and synswp. In their place, it uses the load/store or external control I/O units. This design also used a specialized synapse array interface adapter that provides memory mapped I/O sequences. So to read eight synapse weights into a general purpose register a single load instruction is sufficient. The result in Figure 5.26 shows this approach to be more than two times slower compared with programs using the SYNAPSE functional unit. This underpins the importance of a wide SIMD datapath with dedicated I/O capabilities.

Probabilistic updates Figure 5.27 shows performance results for the probabilistic three-factor test program. This benchmark is an order of magnitude slower then the deterministic ones. The reason for this is the generation of random numbers in software: If the RNG is removed from the code, updating performance is comparable to deterministic algorithms. In place of the RNG a fixed number is returned. Special means were taken to prevent the compiler from exploiting this by removing code that is now unnecessary. Manual inspection of the generated assembly source showed, that only the generation of random numbers was removed from the program. The random selection instructions, as described above, are still executed. Thereby, only the random number generation itself remains as cause for the degraded performance. Despite using a multiply operation, which has a relatively long latency of eight cycles, the linear-congruential method is faster than the XOR-shift variant, which requires more instructions in total. The multiply-with-carry algorithm uses two multiplications and is the slowest of the tested variants.

In the tested benchmark program the RNG is executed for every synapse in the system. Performance could potentially be improved if a random pattern generation algorithm is used that sets bits in a 32 bit word with parameterized probability in parallel. This would reduce the number of RNG calls by a factor of eight. It might also be possible to perform a random choice with less granularity with a minimal impact on learning performance For example, if the random choice is made only once per array and then used for all synapses, over many repetitions the correct probability is approached. In case of the reward-modulated learning task studied in Section 2.3, learning takes up to 10 000 trials giving ample time for approximation. A better solution would be to add hardware random number generators to the instruction set, to quickly generate random numbers.

8 bit weight resolution Computation of 8 bit weights is comparable in performance to probabilistic updates. The two primary causes for degraded performance are that updates need to be computed sequentially, since there is no parallel 8 bit adder, and that weights need to be extracted and packed from and to the SYNAPSE vector registers. Manual inspection shows, that the latter leads to a large fraction of the code being dedicated to shift and bit-wise logical operations.

In principal, specialized instructions for packing and extraction of higher resolution weights could be added to the SYNAPSE extension. However, the separation of the vector register file into front and back prevents simultaneous access to result data from two consecutive read operations. The `synops` instruction could be extended to allow for the specification of a destination register. Thus, two read sequences could store their results in different vector registers. Then dedicated extract instructions could return a single 8 bit weight to a general purpose register, and a pack instruction would move the weight to the respective locations in two vector registers.

An even more advanced approach would be to include a parallel adder in the SYNAPSE extension together with the above mentioned capability to store `synops` results to different target registers. The adder would then add a fixed offset from a general purpose register to each full 8 bit weight and store the result to the remaining two vector registers.

Conclusion on performance Array update rates ν_{array} for two- and three-factor rules are by a factor of five below the rate of 1 Hz identified by Pfeil et al. (2012) as critical speed for the synchrony detection task. This means, that for this task only one fifth of synapses per array may be used. This limits the total amount of plastic synapses available to an emulated network. However, as long as the network does not utilize a complete wafer, synapses can be distributed to other arrays and processors to satisfy the speed constraint. All synapses connecting to the same neuron have to be implemented within the same array. In the BrainScaleS wafer-scale system neurons can have up to 14 336 synapses, which is one fourth of the array. Therefore, a network with that many plastic synapses per neuron that requires updates with 1 Hz is not implementable with the processor or the non-programmable implementation.

The only other task for which the effect of updating speed was studied is the reward-modulated learning rule presented in Section 2.3. Here, updating speed is less critical. The required performance depends on the decay time constant and the analog readout noise, which are both parameters that have not yet been measured in hardware. Preliminary experiments on an older chip-scale system with similar synapses indicate, that for differential evaluation patterns the decay time constant is in the range of seconds in real-time (Pfeil, 2012). In this case updating speed is irrelevant for trial-by-trial learning tasks.

5.4.3. Area requirements

To estimate the area required for implementation of the plasticity processor, the design was synthesized using Synopsys DesignCompiler (Synopsys, 2012) and standard cells were placed using Cadence Encounter (Cadence Design Systems, 2012). This was done as part of the existing implementation flow of the HICANN design used in previous tape-outs. Area requirements given here are obtained using the "report gate count" feature of First Encounter.

Figure 5.28 shows results for the configuration used for performance measurements in Section 5.4.2. All plasticity related digital logic, excluding the local accumulation in the synapse, make up 6.2 % of the totally used area. Nearly half of the total area is dedicated to full-custom analog neurons and synapses.

Within the plasticity related part (Figure 5.28B), most area is consumed by the PP core logic (0.895 mm^2), followed by the three main memory SRAM macros. The synapse array interface adapter is comparatively area intensive due to the wide 128 bit data path and the sequencer and opcode table memories implemented with latches. The 128-entry instruction cache also uses latch based memory instead of more area efficient SRAM. This is due to layout constraints in the design. Also, area consumed by the PPB is not negligible. The bus uses 32 bit address and data lines plus additional

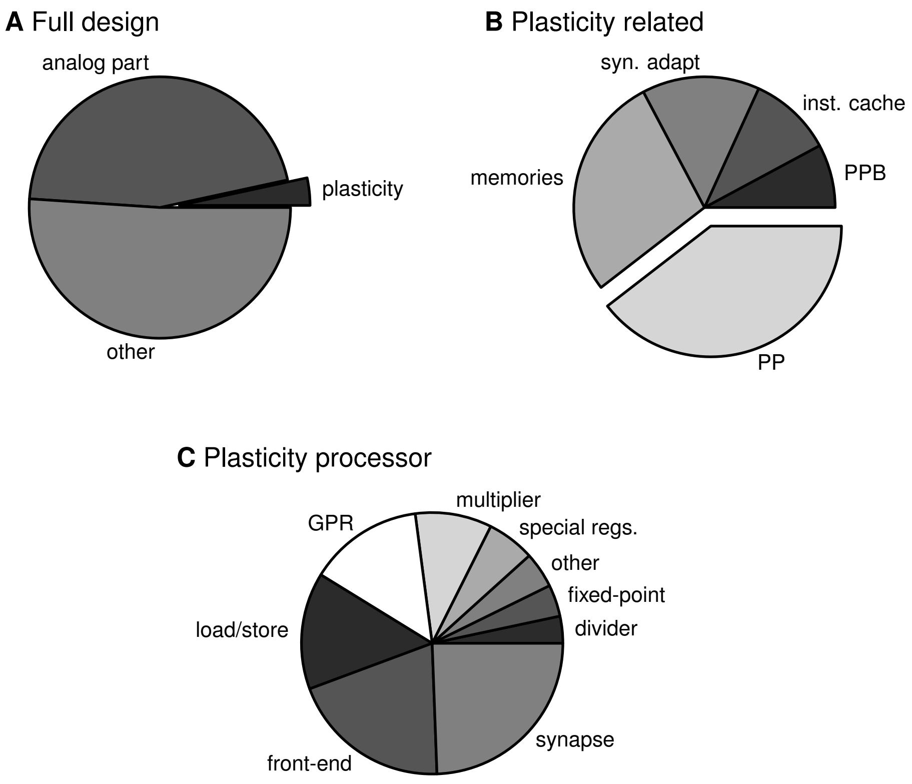


Figure 5.28.: Area requirements for the plasticity related logic in the HEPP design using the same configuration as was used for performance measurements in Section 5.4.2. (A) Breakdown of total design area. (B) Breakdown of the plasticity related part. (C) Area requirements within the plasticity processor.

control signals. For master-to-slave and slave-to-master directions separate data lines are provided. All these signals are registered for delay and Serializer/Deserializer (SerDes) bus elements (see also Figure 5.24).

For the plasticity processor core (Figure 5.28C), the largest sub-units are the SYNAPSE special-function unit, front-end, load/store unit, and the general purpose register file. The SYNAPSE special-function unit features 8×128 bit vector registers – the same number of bits as the general purpose register file – plus the parallel data path for 128 bit operations. The front-end includes the 16-entry fully-associative branch cache, tracking logic for instruction interdependencies, and decoding and control logic. The large size of the load/store unit is explained by the 16-entry deep response and request queues (see Section 3.1.8.1). To save area, this can be reduced to a minimum depth of three. For the plasticity programs in Figure 5.26, this should not have an impact on performance, since the inner loop of update computation does not require memory access. The fixed-point functional unit, executing most of the provided instructions, is comparatively small. Special registers are special purpose registers defined by the Power ISA, such as the condition register and several others. Under "other" there is for example the branch functional unit, and interrupt related logic among others. Multiplier and divider represent a relevant area investment. None of the benchmark programs in Section 5.4.2.1 use division operations. Therefore, the area investment is probably not justified and the functional unit should be removed if area needs to be reduced. Programs would then have to revert to divisions in software, if they require this operation. Multiplication is more useful, for example, it is used in the three factor benchmark and for the generation of random numbers using the linear-congruential method. In the former case, new look-up tables can be computed externally by the control computer, and in the latter, the XOR-shift method, for example, could serve as alternative. Therefore, this functional unit is also a candidate for removal if necessary.

5.4.4. Status of the implementation

As of this writing, the design is not fully implementable. The implementation was carried out up until the placement step, where standard cells from the synthesized netlist are placed on the chip. This shows, that sufficient area is available to realize all logic gates. However, the routing step that interconnects placed standard cells fails. This is due to the asynchronous event network blocking the top three metal layers (layers 4 to 6) out of six available in the process for a large part of the area, where the processor is situated. The lowest blocked layer 4 would otherwise be used for interconnecting wires in the vertical direction. As one can see in Figure 5.29, the processor core is stretched out in a way that especially requires vertical connections. Note, that Figure 5.29 is rotated 90 degrees counter clock-wise and so vertical connections are horizontal in the picture. If layer 4 is made available for interconnection, all connections can be realized, as was shown in a trial implementation run carried out by

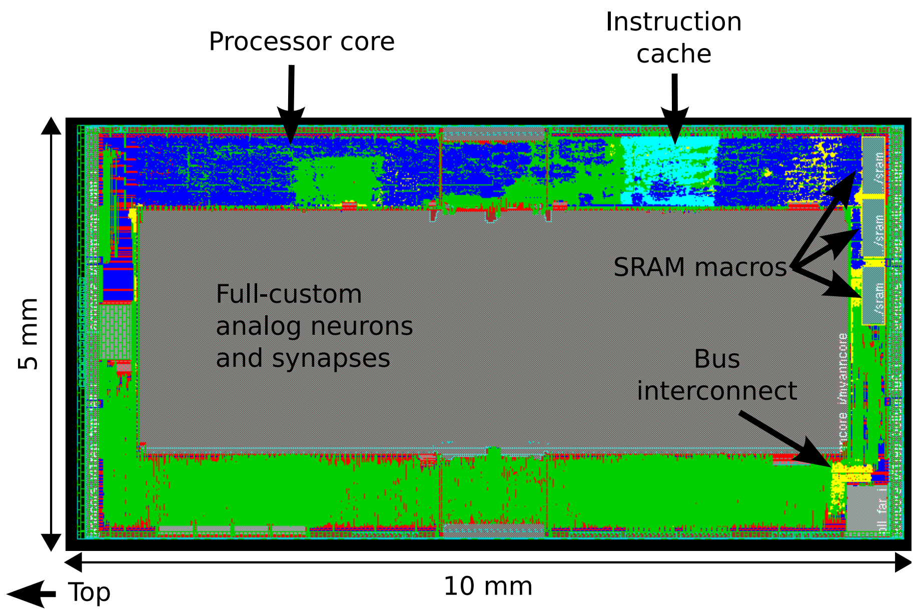


Figure 5.29.: The picture shows the HEPP design after placement of standard cells including the lowest three metal layers. Processor core logic is highlighted in blue, the instruction cache in cyan, and the bus interconnect in yellow. Three SRAM macros provide a total of 12 kiB of main memory. The design is rotate 90 degrees counter clockwise to the conventional orientation.

Grübl (2012). However, this interferes with the power distribution network and the switching matrices of the event network, making a major redesign of this part of the design necessary (Grübl, 2012).

Figure 5.29 shows the result for a different configuration of the plasticity processor than was used in Sections 5.4.2.2 and 5.4.3. Here, the multiplier, divider, and branch prediction are removed from the design. This version was also used for the preliminary routing test in which layer 4 was available for interconnection.

6. Discussion and outlook

The work reported in this thesis investigates options for building biologically inspired and realistic plasticity mechanisms for neuromorphic hardware. The starting point was the idea to use a hybrid system combining classical analog neuromorphic neurons and synapses with a general purpose micro-processor. I first formalized this hybrid concept in the theoretical Abstract Hybrid hardware Model (AHM) and then investigated it using a specific reward modulated learning task (Chapter 2). Implementing the concept in hardware requires the design of a number of components, which are described in Chapter 3. The most important component is the Plasticity Processor (PP) described in Section 3.1. The digital part of the non-programmable STDP implementation used in the current version of the BrainScaleS wafer-scale system (Section 3.3.2) was also designed as a preparatory step. Starting with the experience gained from this design, a new I/O sub-unit and the SYNAPSE instruction set extension were conceived and designed (Sections 3.3.3 and 3.4). These developed technologies were put together in four different systems and evaluated (Chapter 5). An FPGA platform was used first, to verify correct functionality of the processor core, and second, to measure performance in general purpose tasks using the industry standard CoreMark benchmark (EEMBC, 2012). Results for a large number of configuration options are reported in Section 5.1. The processor was included in a prototype chip built with a 65 nm process technology to verify the design in silicon in a low-risk setting as compared to implementation in the BrainScaleS wafer-scale system (Section 5.2). This provided valuable data on achievable clock frequency and power consumption that is relevant input for the design process of future neuromorphic systems planned for this technology. The performance of the non-programmable STDP implementation in the current BrainScaleS wafer-scale system was studied in simulation and initial measurements of the analog evaluation circuit were carried out (Section 5.3). These experiments especially showed the digital weight updating logic that was designed as part of this thesis to be functional. Further, the HICANN with Embedded Plasticity Processor (HEPP) design that is planned as next generation implementation of the BrainScaleS wafer-scale system was evaluated in simulations. Performance for a number of plasticity related benchmark tasks was estimated using simulations (Section 5.4). The preparation for tape-out showed, that further work on parts of the design unrelated to plasticity has to be carried out, before hardware can be produced.

In this chapter I want to outline and discuss the main results from this work and come to a conclusion to what is achievable in regard to plasticity using the presented technologies. The outlook extends this picture to the question of what might be achievable in the future by continuing from this work as starting point.

6.1. Discussion of main results

This section discusses the main results obtained in this thesis.

6.1.1. Abstract hybrid hardware model and results for reward-modulated STDP

The theoretical analysis in Chapter 2 formulates the AHM as a tool to explore the learning performance of a neuromorphic system in simulations. The model itself is abstract in that it is independent of a specific implementation technology or details of the system's architecture. It is hybrid in that it assumes a combination of local, time-continuous and global, time-discrete processing. Therefore, results obtained with this model apply to a large class of potential systems and not just the ones presented here. The central assumption is, that the local, time-continuous part implements an eligibility trace that is evaluated by the global part. Whether the eligibility trace uses analog or digital circuits, or some other means, e.g. memristive devices (Chua and Kang, 1976; Strukov et al., 2008; Snider, 2008), is not relevant.

The AHM was used to explore hardware design variations for a reward-modulated spike train learning task using computer simulations. Results from this study are summarized in Section 2.3.8 in the form of guidelines for hardware design. In essence the study showed the hybrid concept simulated with realistic hardware constraints to be sufficient for this particular task.

Many tasks can be formulated as spike train learning. For example, for pattern recognition the network learns to produce a special output spike train, when the input, also coded as spike train, contains a specific pattern. This is studied by Frémaux et al. (2010) using the same network architecture and learning rule as was used here. In more recent work the authors employ a more complex network structure to extend this rule to Temporal Difference (TD) learning using actor and critic neuron populations (Frémaux et al., 2013). In this framework they are able to solve among others the cart pole task, in which a pole on a movable cart has to be balanced in an upright position. New weights are calculated in the same way as in the R-STDP rule tested in Section 2.3, and only the neuronal architecture and the meaning of the modulation factor are different. It now represents the TD-error coded as population firing rate. Using rate counters (Section 5.4.1.2) this can be made accessible to the plasticity program. Therefore, this updating mechanism is also implementable in the presented hardware model. Results from Section 2.3 do not allow a prediction of learning performance.

In conclusion, the AHM represents a tool for the analysis of learning rules under hardware constraints. To be able to emulate the large class of multiplicatively modulated STDP rules, the central requirements for hardware are the presence of an eligibility trace per synapse and multiplicative modulation of the weight change.

6.1.2. Plasticity processor

The plasticity processor design represents a major technical development carried out for this thesis. It is a 32 bit scalar, in-order issue, and out-of-order retire RISC processor implementing the Power ISA (PowerISA, 2010). The challenge of verifying correctness considering the very large input space of possible programs was met with elaborate simulations and an FPGA-based, as well as a silicon prototype. For the former, programs were generated and verified automatically. Correctness was further validated with numerous runs of the CoreMark benchmark software (EEMBC, 2012) and a suite of dedicated test programs (Section 4.1 and Nonnenmacher, 2011). In conclusion, the processor can be considered usable for general purpose programs. Limitations in the current state of the design would be the lack of a Memory Management Unit (MMU) and of a cache for data supporting read and write.

The performance data gathered using the CoreMark benchmark showed the design to be competitive with commercially available processors. In particular performance is superior compared to the open source OR1200 processor (OpenRISC Project, 2013) or the ARM968 core by NXP Semiconductors (2010).

Together with the Plasticity Processor Bus (PPB) the PP can be used as building block in future neuromorphic systems. Its configurability and modular design allow for the use in different process technologies and with varying area and frequency goals. Use of different technologies was demonstrated with the 65 nm prototype and the HEPP design (Sections 5.2 and 5.4). Configurability is evident from the performance evaluation carried out on the FPGA platform (Section 5.1.1). Taking the SYNAPSE extension as example it is easy to add further specialized execution units into the design. For example, a parallel fixed-point multiply-accumulate SIMD unit could provide accelerated computing for synaptic weights with more than 4 bit. Such new instructions can be single or multi cycle operations using the framework described in Section 3.1.6.6. In both cases they can complete with fixed latency using the Result Shift Register (RSR) for scheduling of write-back, or, with variable latency, using the delayed commit mechanism (Section 3.1.6.3). Using external control instructions (Section 3.1.8.1 and PowerISA, 2010) a second port for I/O can be added to the processor beside the load/store unit. This can be used as optimization, for example, to connect some peripherals with a faster bus interconnect than load/store via external control.

When starting a new design it is always a difficult decision, whether to continue from an existing project or to start a new one, as I did for the plasticity processor. In hindsight the question remains, if it was justified to develop a new general purpose micro-processor. Especially the comparison of CoreMark performance shows, that the new design is at least not worse in performance compared to established processors. Even commercial designs are outperformed or at least matched. Further, existing high quality designs, like the ARM IP cores, are not freely available and are often too expensive for a research project. Choosing a commercial design would also create a dependency on a particular company. This might be a hindrance for future neuromorphic systems, for example, if the company does not provide the used processor for a new technology. The most important reason for developing a new design is, that it allows for the optimization of every detail towards the intended application of plasticity. The resulting design as presented in this thesis is a good match to the particular requirements. For example, the overall architecture was selected for low power- and area-cost, to allow for integration into the BrainScaleS wafer-scale system. All features typically found in processors that are not required for the intended application, e.g. distinction between supervisor and program mode, a MMU, or a data cache, were left out. So although it would probably have been possible to achieve the goal of programmable plasticity starting from an existing processor, I argue, that it was the best decision to start a new design.

6.1.3. Instruction set extension for operations on neuromorphic synapses

To exploit the high amount of data parallelism in the used STDP algorithm I developed the SYNAPSE instruction set extension with 32×4 bit vector operations and registers. With this extension it was possible to outperform the previous implementation while simultaneously increasing flexibility in the programmable HEPP system. The comparison to the NVE variant of the design in Section 5.4.2 confirms the necessity of using highly parallel weight computation and specialized wide I/O buses. NVE performs I/O through load/store or external control operations and computes on 8×4 bit vectors stored in general purpose registers. Using SYNAPSE instead, the number of input, output, and computing operations can be reduced by a factor of four, due to the vector size of 32×4 bit. The same inefficiencies would be present in a system, where a dedicated I/O controller moves large blocks of synapse data to and from main memory.

In the context of brain-inspired computing and learning, there are instruction set extensions for the acceleration of software simulations in the literature. For example, the system described by Al Maashri et al. (2013) is an FPGA accelerator card used with a standard personal computer to simulate a high-level visual processing model. Shapiro et al. (2011) use an automatically generated ISA on an FPGA platform to accelerate the simulation of spiking neural networks. Madrenas and Moreno (2009) employ simple, parallel processing elements in an SIMD array with the intention to simulate neural networks. To my knowledge, the use of a specialized instruction set extension in combination with analog neuromorphic circuits is a new approach first presented here. It combines advantages from the worlds of analog and digital processing: Local, time-continuous, and fixed analog circuits perform the massively parallel detection of spike coincidences. Global, time-discrete, and programmable digital logic computes new weights with great flexibility. Together, this allows for good energy efficiency and flexibility. The SYNAPSE special-function unit plays a key role in this concept. It represents the interface between plasticity software and the analog circuitry. Using specialized instructions allows for higher throughput of weight update computations as compared to a plain general purpose processor. Ways to accelerate probabilistic and 8 bit updates are proposed in Section 5.4.2.

6.1.4. Non-programmable STDP implementation

The concept of combining local analog processing with global digital weight updates for STDP was already implemented in a non-programmable way on a previous chip-scale system (Schemmel et al., 2004, 2006, 2007). The non-programmable STDP implementation for the BrainScaleS wafer-scale system (Section 3.3.2) provides the same high-level capabilities to a modeler using the system. It allows for the use of two-factor STDP for unsupervised learning tasks, such as were demonstrated on the chip-scale system by Pfeil et al. (2012).

By making the micro-program of the automatic weight update process (Section 3.3.2.5) modifiable, for example by storing it in latch based memory, this implementation could serve as minimalistic and functionally constrained alternative to the more elaborate processor based HEPP design. With small modifications to the existing micro-program sequencer, this would enable for example the combination of two rows to emulate separate reset lines for a_+ and a_- (see also Section 5.4.2). However, due to the lack of appropriate arithmetic units, this does not allow for increased synaptic weight resolution. Also, it would not be possible to modify the contents of look-up tables from within the system, for example to use different updating rules for different synapses. Three-factor rules could thus only be implemented by changing look-up tables from the external control computer. Further restrictions compared to the HEPP design would be no access to rate counters or other system components, and no additional state information in main memory, e.g. for neuromodulator chemical concentrations. Therefore, this implementation even with modifiable micro-program store does not reach the capabilities of a PP based design.

6.1.5. 65 nm prototype

The production of the 65 nm prototype ASIC served two functions: first, it validates correct functionality of the PP in silicon, and second, it gives power and performance data that can be used for future neuromorphic designs in this process technology. In the configuration the processor was implemented, a normalized CoreMark performance of

$C_{\text{eff}} = 0.75$ is reached. Results reported in Section 5.1.1 for the FPGA implementation show that a much better performance is reachable. This is due to optimizations that were introduced only after tape-out of the prototype. For example, the issue in time feature can be enabled without impact on clock frequency, but considerable effect on performance (see Section 5.1.4). Other optimizations missing on the produced version, like write-through on general purpose registers, or branch prediction, could have an impact on frequency. The instruction latency for multiplications L_{MUL} is relatively high compared to the FPGA variants, because here a component from the DesignWare library is used, while for the FPGA hard-macro multiplier blocks are available. Also to support higher clock frequencies, latencies of other instructions are generally larger than for the FPGA. In light of the results presented in Section 5.1.3, which show the greatest improvement is achieved by reducing the latency L_{def}, it might be worthwhile to selectively optimize this datapath for shorter latency, while trying to keep the impact on frequency small. In conclusion, the reported performance does not represent the best-case achievable for the 65 nm technology and the PP design. An improved version could now be produced.

The processor was designed for a 500 MHz clock frequency. Measurements in Section 5.2.3, show this to be reached at the nominal supply voltage of 1.2 V for two out of three tested chips. Static timing analysis reported remaining timing violations in the "slow" corner, while "typical" and "fast" were free of errors. The corners represent sets of parameters like operating voltage, temperature, and process variations, that in combination result in slow, fast, or typical performance of the logic cells. Therefore, one chip failing to reach 500 MHz at 1.2 V is not surprising. The sample size is much too small to draw any statistical significant conclusions. The measurements show the relation between maximum clock frequency and supply voltage to be approximately linear in the tested range. With the best performing chip, a maximum frequency of 769.53 MHz at 1.4 V was reached consuming (102.9 ± 0.03) mW. This corresponds to a CoreMark performance of $577.15\, s^{-1}$ and an energy efficiency of $C_{\text{perf}}/P_c = (5.61 \pm 0.02)\,(\text{mJ})^{-1}$ (CoreMark iterations per mJ). Varying supply voltage between 1 V and 1.4 V allows for nearly doubling the maximum frequency. In this range, power scales with a factor between $(65.7 \pm 0.1)\,\mu\text{W}/\text{MHz}$ and $(133.5 \pm 0.3)\,\mu\text{W}/\text{MHz}$ depending on supply voltage.

If a future system in this technology requires a processor with a given clock frequency, statically varying supply voltage allows for compensation of manufacturing variations. This is relevant for a wafer-scale system, where a block not fulfilling its specification has to be marked unusable, but can not be exchanged on the wafer. It therefore wastes precious wafer area. If power can be varied on a per-block basis, on the one hand, a higher voltage can be applied to make too slow processors usable, and on the other hand, a lower voltage for fast processors saves power. Going to dynamic voltage and frequency scaling (DVFS, Pouwelse et al., 2001; Simunic et al., 2001) is only necessary, if varying workloads have to be handled. In the HEPP design, this is typically not the case, because the plasticity program always iterates with maximum rate over all synapses. The workload is therefore constant. This would be different in a system that implements also the local accumulation part of the AHM in software. Here, the workload would scale with event rate and DVFS could be used for power saving.

For power saving, it is especially interesting how far the supply voltage can be reduced. Experiments revealed a sharp drop of maximum frequency below 0.7 V. The lowest voltage at which the processor functioned correctly was 0.68 V with a maximum frequency of 15 MHz. Below that, the processor did not function for frequencies greater or equal to 5 MHz. For higher voltages, the linear relation between voltage and frequency is maintained.

The transient power consumption during and after execution of the CoreMark program shows the global clock gate to be an effective method for power saving, reducing consumption by an order of magnitude. However, as the comparison to disabled clock input to the whole chip in Figure 5.17 shows, there is still room for reducing power for the off-state of the global clock gate. Logic and especially the clock tree outside the processor have to be carefully optimized for low-power operation.

In Section 5.2.3.4 the energy costs of individual instructions was measured. The main factor determining energy cost is the number of clock cycles required per instruction, followed by the number of memory accesses. The complexity of datapath logic nearly has no effect, as for example the comparison of multiplication and addition in Figure 5.18 shows. Therefore, to optimize for power, it is important to design an efficient front-end of the processor, while the back-end is less critical. According to Gonzalez and Horowitz (1996) the additional energy cost of superscalar architectures outweighs the performance gain when using the energy delay product as metric. Zyuban and Kogge (2000) also shows, that for many sub-components of a superscalar architecture energy per instruction scales super-linear with increasing issue width. For example, the increased number of ports on the register file required to support parallel issue of multiple instructions leads to increased energy cost per instruction. Therefore, to maximize power-efficiency, a scalar, in-order-issue architecture, as was selected for the PP, is the best match.

6.1.6. Plasticity processor in the BrainScaleS wafer-scale system

The HEPP design presented in Section 5.4 represents the culmination of the work done for this thesis. It integrates the developed technologies described in Chapter 3 – especially the plasticity processor – into the existing BrainScaleS wafer-scale system. The result is a novel architecture for neuromorphic hardware that enables strong plasticity in an accelerated and large-scale system.

The key capabilities that are added by this architecture compared to the non-programmable implementation currently used (Section 5.3) are:

- Different plasticity rules can be used simultaneously for different synapses in the system.
- Further observables are available through the connection to the HICANN system bus: rate counters measure population rates, and information about the networks structure is held in the decoder addresses of the synapse and the event network configuration. Previously, plasticity rules could only depend on the synapse-local coincidence measurement and accumulation, and on the synapse weight itself.
- Additional state, e.g. a map of neuromodulator concentration, with its own dynamics can be stored and maintained in main memory.
- Plasticity rules can interact with the environment by messages exchanged through main memory and using the doorbell interrupt.
- Plasticity rules can affect further parameters instead of only synapse weights: network connectivity through decoder addresses and event network configuration, and event rates of random background generators on the event network. Neuron parameters on floating-gate storage cells can only be modified while the network is not in use, because the programming voltages are visible to the neuron circuit disrupting normal operation. So modifying these parameters is restricted to trial-by-trial learning.
- Dual- or even multi-row operation either to emulate separate reset lines for a_+ and a_- or to increase synapse weight resolution. The latter case comes at the cost of reduced performance, because the 4 bit vector arithmetic of the SYNAPSE functional unit can not be used. Multi-row operation could also be used in conjunction with the evaluation circuit (see Section 3.3.1.2) to average over the accumulation traces of multiple synapses to increase precision.
- Probabilistic weight updates are possible, although with reduced performance (Section 5.4.2.2).

Additional to this improved functionality, the new design can perform weight updates nearly two times faster. This is due to the programmable control sequencer in the synapse array interface adapter (Section 3.3.3) and the 128 bit parallel weight computation by SYNAPSE. Both features could also be added to the non-programmable implementation.

Measurements in the current system reported in Sections 5.3.3 and 5.3.4 revealed two open issues in the analog part of the design. The erroneous behavior of the evaluation circuit for one half of the synapse array requires further investigation. So far, it is not entirely certain, that a problem exists in the hardware design at all. The effect could be explained by a malfunctioning floating-gate cell due to a manufacturing defect, which could be tested by performing the experiment on a second chip. The other issue is crosstalk between synapse weight and decoder SRAM accesses and synaptic event transmission. It represents a real hardware defect that needs to be addressed in a future version of the system with or without plasticity processor. Further systematic measurements are required to characterize how detrimental this effect is to network operation and whether a working configuration for plasticity experiments can be found.

The analysis of the reward-modulated spike train learning task (Section 2.3) came to the conclusion that learning performance is increased by either going from 4 bit weight resolution to 8 bit or by using probabilistic updates. Both is possible in the HEPP design, but associated with significantly reduced performance. This is likely not a problem for the reward-modulated learning task, assuming that the decay time constant τ_e of the eligibility trace in hardware is very long compared to the array updating time $224 \cdot t_{\text{row}}$. This assumption is motivated by comparable measurements on the precursor chip-scale system using similar synapses performed by Pfeil (2012) suggesting decay time constants on the order of ten thousands of seconds in biological time or seconds in real time. The reason for these long decay times is the used differential evaluation comparing the similarly fast decaying a_+ and a_- to each other. Ways to improve performance for probabilistic updates or 8 bit resolution are proposed in Section 5.4.2.

Comparison to other plasticity implementations I already compared the HEPP design to other plasticity implementations in Friedmann et al. (2013): Typically, spike based neuromorphic hardware systems implement variants of STDP with fixed-function circuits (Indiveri et al., 2006; Ramakrishnan et al., 2011; Seo et al., 2011). The weight is changed according to an algorithm built into the device, offering no flexibility. Additionally, these systems update weights immediately, when spikes are encountered. There is no per-synapse state variable usable as eligibility trace. Therefore, these systems can not implement the reward modulated learning rule studied in Chapter 2, because they can not associate reward with earlier activity. This is known as the distal reward problem (Izhikevich, 2007b).

The comparison also includes two processor-based plasticity implementations (Vogelstein et al., 2003; Davies et al., 2012). The system described by Vogelstein et al. (2003) uses multiple special purpose chips: one with analog neurons, one with SRAM for synaptic connections, and one with a general purpose processor. The processor implements routing of events and plasticity in software using the connection data stored in the memory chip. This is only possible, because the number of neurons is small (1024), and they operate without acceleration ($\alpha = 1$). For accelerated systems the overhead by software and memory accesses for every spike are prohibitive. Also, integrating the processor into the same die reduces power for off-die communication. Therefore, the HEPP design has better scalability as is required for a wafer-scale system. The SpiNNaker system presented by Davies et al. (2012) consists of general purpose processors interconnected by an asynchronous event network. Due to the organization of the architecture, especially the storage of synaptic weights in off-chip memory, they only have the weight value available for presynaptic events. Therefore, they can not directly update weights for pre-before-post spike pairs. They work around this by estimating postsynaptic firing times using the current membrane potential, whenever a presynaptic event arrives. This emphasizes, that it is difficult to implement STDP in fully programmable systems. The hybrid architecture of the HEPP design represents a more efficient solution.

6.1.7. From guidelines to implementation

At the end of Chapter 2, the findings of the theoretical analysis were condensed into a set of guidelines for hardware implementation. Here, I want to discuss, how the HEPP design is in accordance with these guidelines.

To reach good performance, either 4 bit weights with probabilistic rounding, or 8 bit weights with deterministic rounding should be used. The natural resolution in the hardware system is 4 bit, but by combining rows, the resolution can be increased in increments of 4 bit at the cost of reducing the number of totally available synapses. As was discussed in the previous subsection, the HEPP design does not offer accelerated computation of 8 bit weights or probabilistic rounding.

The evaluation circuit in the HEPP design supports comparison of the difference between both accumulators to a threshold. This is in accordance with the given guidelines. The temporal behavior of the eligibility traces was not yet measured in the BrainScaleS wafer-scale system, but is expected to be slowly drifting when using the differential evaluation. Synapses do not have a controlled mechanism for decay. As long as there is no theoretical finding that clearly shows such a mechanism to be required, it is not justified to modify the synapse circuit. Any modification would likely require an increased size and therefore reduced number of available synapses in the system. The measurements on the accumulation circuit presented in Section 5.3.3 show fixed-pattern variations to be small. The spike train learning task is anyway very tolerant to fixed-pattern noise.

Delay in giving the reward is introduced by communication with the external control computer and by the finite processor updating speed. Since neither the decay time constant nor the trial-to-trial variation of the evaluation circuit are known at this time, a definite requirement for delay on reward can not be made.

In conclusion the HEPP design fulfills all requirements for reward modulated STDP as outlined in Section 2.3.8. With its speed-up factor of 10 000 and large number of neurons (up to 196 608), it will represent a very attractive research platform for this type of learning when it is produced. Especially the question of scalability of the R-STDP rule to large neural networks could be explored.

6.2. Directions for future hardware

As discussed in the previous section, the results obtained in this thesis point to some limitations in the HEPP design. On the other hand, the investigation of the 65 nm prototype hints at what might be possible in future neuromorphic systems. In this section, I want to suggest two new architectures, one to maximize performance in the 180 nm HEPP design, and one to maximize flexibility using the 65 nm technology.

6.2.1. Maximizing performance in the HEPP design

The HEPP design does not perform very well for probabilistic updates and weights with more than 4 bit resolution. I already proposed to add hardware random number generators to increase performance for probabilistic updates (Section 5.4.2). However, for higher resolution weights it is necessary to replace the look-up table based update units with real arithmetic units providing multiply and add operations. To make matters worse, 8 bit weights are distributed over two rows for one synapse. Therefore, two read and two write operations are required to aggregate the weight and store the result of the update. Also, conversion between the logical representation with 8 bit weights and the raw one from the array requires additional operations.

The design proposed here addresses these problems by introducing read and write caches that convert between the representations, and by using up to eight parallel 128 bit vector processing units. Figure 6.1 shows an overview.

Vector processing units Each vector processing unit operates on 128 bit vectors of either 32×4 bit, 16×8 bit, or 8×16 bit. It implements a multiply-add operation with operands from the read cache or a constant memory. This functionality could for example be implemented with DesignWare SIMD blocks (DWF_dp_simd_mult_tc, and DWF_dp_simd_add_tc). These blocks allow to use one hardware multiplier or adder, respectively, for all three configurations of input vectors. The result from the vector unit is either written to the read or the write cache. The vector operations can be enabled element-wise depending on the evaluation result.

Read and write cache The read cache is divided into eight banks that each hold four 128 bit vectors and eight 32 bit correlation words. This matches the data produced in one read operation with two evaluations of the synapse accumulators with weight data written to the 128 bit vector and correlation written to the two 32 bit ones (see Section 3.3.1). Upon reading from the synapse array, the raw 4 bit weight elements are

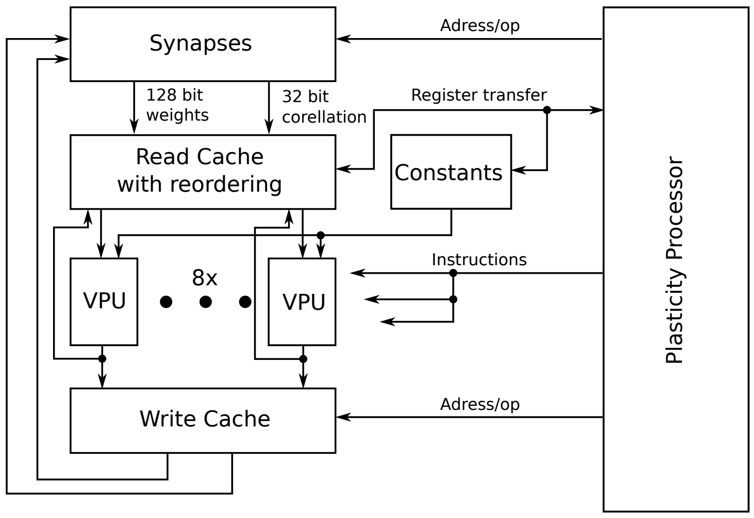


Figure 6.1.: Using parallel vector processing units and caching to remove performance bottlenecks in the HEPP design. See text for a description.

scattered to the four vectors depending on whether 4 bit, 8 bit, or 16 bit resolution is used. In the first case, the data is moved directly to the vector as it is presented on the data bus. In the last case, the first to fourth 4 bit elements are placed in the first position of each of the four vectors in the read cache. This way, the vector holds the correct logical representation with 8×16 bit weights ready for processing by the vector unit. In a similar fashion is proceeded for 8 bit weights. The write cache performs the reverse operation, reordering the logical vector to a raw representation that can be written to the array. For maximum efficiency it has to be of the same size as the read cache, so that a full logical row with 16 bit synapses can be stored. However, only one 32 bit correlation word per bank is required to control the reset of accumulators in the synapse.

Having read and write cache allows for maximal decoupling between synapse I/O and computation. The vector processing units operate only on logical synapse rows, while data is exchanged between caches and synapse array in terms of physical rows. Therefore, I/O can be performed row-wise, saving an overhead for pre-charge every time a row is opened (see also the results for the dual-row benchmark program in Section 5.4.2). The plasticity problem is separated into two that can be independently solved: 1) Efficient data transfers between synapse array and caches. 2) Computation of weight updates by the vector processing units.

A constants memory stores additional parameters for the computation, such as a multiplicative factor for three-factor plasticity rules. It is filled from the general purpose register file of the plasticity processor. Also the read cache can be read and written from general purpose registers.

Control The vector processing units and the caches are viewed by the plasticity processor as a functional unit. Dedicated instructions in the program are issued to this functional unit to, for example, read a row of synapse weights or perform a vector multiply operation. Therefore, only the datapath and cache memories need to be added to the design, and additional overhead for control is kept small.

Implementability Using the area requirements for the regular HEPP design as basis (Section 5.4.3), one can extrapolate the expected area cost for this architecture. Read and write cache have in total $2 \times 8 \times 4 \times 128\,\mathrm{bit} = 8192\,\mathrm{bit}$ vectors and $8 \times 2 \times 32\,\mathrm{bit} + 8 \times 32\,\mathrm{bit} = 768\,\mathrm{bit}$ correlation words. Assuming a constants memory of $8 \times 32\,\mathrm{bit} = 256\,\mathrm{bit}$, this is in total about 2.2 times the size of the instruction cache shown in Figure 5.28. The most area intensive component of the eight vector processing units is the multiplier. Assuming, that a 128 bit vector-multiplier is four times larger than the 32 bit scalar multiplier used by the plasticity processor, the eight vector processing units would cost 32 times that size. To estimate the total increase in area, I assume that the multiplier and SYNAPSE functional units are removed, and that the same area for a synapse array interface adapter is needed. This gives a total increase by a factor of 4.28 over the HEPP design for the digital plasticity logic. That much area is not available in the current design. However, since the digital part of the plasticity logic in the HEPP design only consumes 6.2 %, it would be possible to make this area available by reducing the size of the analog part, e.g. by reducing the number of synapse columns. This would increase the expended area for the digital part of plasticity to 26.5 %.

6.2.2. Maximizing flexibility in future 65 nm systems

The hybrid approach of combining analog computing in the synapse with globally digital weight updates allows for a good trade-off between updating speed and flexibility. However, the analog part still imposes restrictions on flexibility, for example by limiting the shape of the learning function $s(\Delta t)$ (see Chapter 1). A way to increase flexibility beyond this, is to go to a fully programmable system that processes individual spikes for each synapse. This is not a viable option for 180 nm technology with high acceleration factors of $10^3 \ldots 10^4$, which is why this hybrid approach was chosen in the first place. However, future smaller scale technologies make smaller and faster digital logic possible.

Figure 6.2 sketches such a system having the 65 nm technology in mind: The part not related to plasticity is conceptually identical to the current HEPP architecture. Synapses are arranged in a rectangular array with each column belonging to one neuron. Action potentials generated by the neuron are passed through a routing

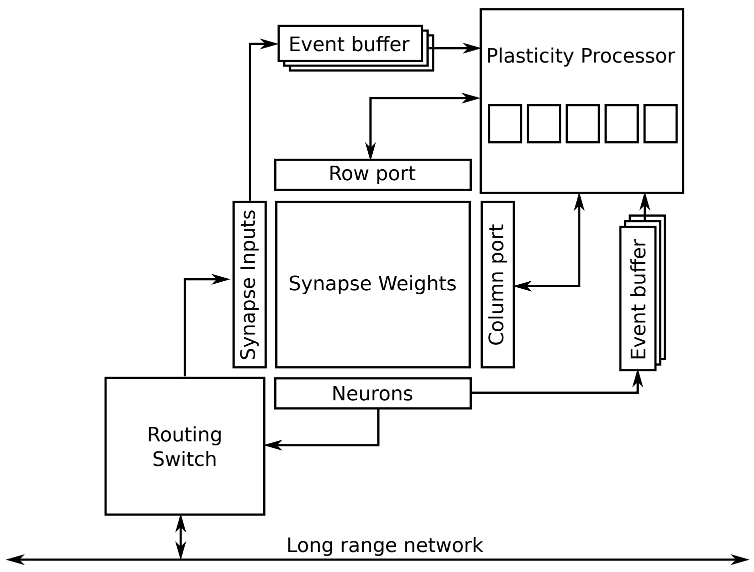


Figure 6.2.: Sketch of a neuromorphic system with fully digital plasticity implementation. The main hardware features are a transposable synapse array for efficient row and column access, buffers providing temporal and spatially sorted access to events, and a plasticity processor with parallel weight computation. See text for a detailed description.

switch to send them either to a long range network, or to the input drivers to the synapse array. Synapses do not include a local accumulation circuit, reducing the size of the array. For plasticity, pre- and postsynaptic events – from the synapse input drivers and the neurons, respectively – are send to event buffers. These event buffers are FIFO queues that present spikes in temporal order to the plasticity processor. Additionally, they are content addressable, allowing for readout of events by row- and column-address.

If reward modulated learning rules are to be implemented, it is further necessary to provide digital storage for an eligibility trace. This can either be accomplished by additional bits in the synapse array that are only interpreted by the plasticity program, or by separate memory in the plasticity processor.

Algorithm The task of the plasticity processor is to compute weight changes depending on the spike pairs extracted from the event buffers. The update algorithm for a two-factor STDP rule would follow this outline:

- Wait for new pre- or postsynaptic event in the event buffers.
- For a presynaptic event, load the row of weights to which the event was sent to internal storage of the processor. For a postsynaptic event the remaining procedure is symmetrical. Just exchange pre- and postsynaptic and row and column.
- For each synapse in the row, extract the most recent event for the postsynaptic neuron from the event buffer.
- Compute the weight change depending on the time difference.
- Write the new row of weights back to the synapse array.
- Remove presynaptic events from the event buffer that are older than a given threshold (for example three times the time constant $\tau_\pm$ of the learning function). If the capacity of the event buffer is reached, remove at least one of the oldest events.
- Repeat

Hardware optimizations To improve performance, to hardware optimizations are conceivable: Since the algorithm performs row- and column wise accesses depending on whether a pre- or postsynaptic event was encountered, the SRAM weight storage can be equipped with a row and a column port. Depending on which port is used, the full row or column is returned with one read operation. Such a transposable SRAM array is used for synapses by Seo et al. (2011). A test-chip designed by Hock et al. (2013)

for the 65 nm process was recently submitted and contains, among other components, a transposable SRAM macro.

The second optimization is to use a vectorized SIMD unit in the plasticity processor to compute the weight update. In principal, the same design could be used as was proposed in Section 6.2.1. Additional operations for access to the event buffers would need to be designed.

Back-of-the-envelope performance estimate The 65 nm prototype (Section 5.2) used a clock frequency of $f_{\text{clk}} = 500\,\text{MHz}$. So, this frequency can be assumed for the proposed design. Further, in the existing BrainScaleS wafer-scale system neurons with up to 14k synapses can be configured to realize numbers found in neocortex (Pakkenberg et al., 2003). To match this number, an array size with $N_{\text{pre}} = 128$ presynaptic inputs and $N_{\text{post}} = 128$ postsynaptic neurons could be used. Each neuron and each input contribute events with an average firing rate of ν. In neocortex, ν is on the order of 10 Hz (Plenz and Aertsen, 1996; Shafi et al., 2007). Finally, the analog neurons used in the BrainScaleS wafer-scale system can operate with a speed-up of $\alpha = 10^4$. This allows for a rough estimation of the number of clock cycles available per event in the proposed algorithm:

$$c_{\text{ev}} = \frac{f_{\text{clk}}}{\left(N_{\text{pre}} + N_{\text{post}}\right) \nu\alpha} \tag{6.1}$$

with $c_{\text{ev}} = 195\,312\alpha^{-1} = 19$. Although, this needs to be validated, it seems to be achievable to implement the proposed algorithm with this number of clock cycles, given optimized data structures in the form of the event buffers, and vectorized processing units operating on whole rows and columns. If the speed-up factor α can be reduced by an order of magnitude through slow-down of the analog dynamics of the neuron, fully-digital plasticity should therefore be possible in the 65 nm process using the proposed architecture.

6.3. Conclusion

This thesis presents a new approach to learning and plasticity in neuromorphic hardware. Instead of a pure physical model, a hybrid system is proposed that combines local, analog computation with a central processor. This approach is motivated by the diversity of plasticity mechanisms observed in biology. Flexible plasticity is a requirement for neuromorphic hardware intended as research platform for neuroscience. It enables network-level studies of the functional consequences of such elementary observations. As long as a single rule for plasticity, that leads to brain-like learning capabilities, has not been found, flexibility is also desirable for technical applications of neuromorphic hardware.

The work presented in this thesis investigated the usefulness of the hybrid approach, and the implementability in hardware. The full functional capabilities enabled by this system have yet to be explored, but the analysis of the R-STDP rule in Chapter 2 gives a first impression of what is possible with multiplicatively modulated STDP rules alone. This class of rules is used in the literature for a large number of learning tasks (Farries and Fairhall, 2007; Izhikevich, 2007a; Legenstein et al., 2008; Frémaux et al., 2010; Potjans et al., 2011; Frémaux et al., 2013). Most interestingly, it opens the gate to reinforcement learning with spiking neurons (Florian, 2007), and thereby with neuromorphic hardware.

In Section 2.1, I described a number of requirements on hardware implementations for two-factor, reward-modulated, and phenomenological models of STDP. Requirements for two-factor and reward-modulated rules are fulfilled by the programmable, as well as the non-programmable implementation. In the latter case, reward-modulation has to be performed outside of the system by pre-computing new look-up tables on the control computer. In this case, updates are limited to 4 bit resolution and deterministic rounding, limiting learning performance. Further requirements by phenomenological models are not met by the non-programmable system. With the plasticity processor, location and type of neurons and synapses can be used for plasticity, but only firing rate is accessible as additional state variable. Further observables, and the ability to modify the learning function of STDP are not available. Short-term effects, and arbitrary learning functions would be realizable with the fully digital system proposed in Section 6.2.2 using e.g. 65 nm technology.

Beyond STDP based rules, the architecture enables completely different learning paradigms. For example, an evolutionary algorithm could be used to optimize neuron parameters for a given task. Each physical neuron could implement an individual parameter set in a population, of which the parameters are mutated from trial to trial. Such a mechanism would be interesting for calibration of neurons to a specific operating state, e.g. a given firing rate, compensating device mismatch on the neuron circuits.

A second example for a new learning rule, not based on STDP is gradient decent to optimize weights for a given task. In this case the processor would systematically modify individual weights between trials. This way, the gradient of the quantity to minimize is determined. Performance is improved by following the gradient "downhill". Typically encountered slow conversion for this type of learning rule is alleviated by the high acceleration factor of the hardware system.

So far, no hardware system of the proposed type has been produced, but results from measurements and simulations make a convincing case for the implementability. The two proposed architectures in the previous section show that it is possible to further improve the HEPP design with regard to performance and flexibility. However, the former is only possible with a major investment of chip area at the cost of synapses and neurons, and the latter requires a more modern process technology.

So in conclusion, the overall result of this thesis is, that the proposed architecture can emulate a wide class of plasticity rules, and can be integrated into the BrainScaleS wafer-scale system.

A. Tabular description of used neural network models

Description of the network model used for the learning task after Nordlie et al. (2009). See Table A for numerical values for the parameter. This table was previously published in Friedmann et al. (2013).

A: Model summary		
Populations	Three: input U, random background B, target T	
Connectivity	Feed-forward	
Neuron model	Leaky-integrate-and-fire, fixed voltage threshold, fixed absolute refractory period (voltage clamp)	
Synapse model	Exponentially shaped post-synaptic conductances	
Plasticity	Three-factor STDP	
Input	Fixed-length spike-trains with uniformly distributed firing times	
B: Populations		
Name	Elements	Population size
U	Stimulus generator	N_U
B	Poisson generator	N_B
T	LIF neurons	N_T
C: Connectivity		
Source	Target	Pattern
U	T	All-to-all, initial weights w_S
B	T	Non-overlapping $250 \rightarrow 1$, weight w_B
D: Neuron and synapse model		
Name	LIF neuron	
Type	Leaky integrate-and-fire, exponential-shaped synaptic conductances	
Sub-threshold dynamics	$\begin{cases} C_m \frac{dV}{dt} = g_L\left(E_L - V\right) + g(t)\left(E_e - V\right) & \text{if } t > t^* + \tau_{\text{ref}} \\ V(t) = V_{\text{reset}} & \text{else} \end{cases}$ $g(t) = w \exp\left(-t/\tau_{\text{syn}}\right)$	
Spiking	if $V(t-) < V_{\text{th}} \wedge V(t+) \geq V_{\text{th}}$ 1. set $t^* = t$ 2. emit spike with time-stamp t^*	

Continued on next page

Continued from previous page

E: Plasticity	
Name	Three-factor STDP
Spike pairing scheme	Reduced symmetric nearest-neighbor (Morrison et al., 2008)
Weight dynamics	$\Delta = Sa(t)$ $a(t) = \sum_{\substack{i \\ t_i<t}} A_{\pm} \exp\left(\frac{\lvert\Delta t_i\rvert}{\tau_{\pm}}\right) \exp\left(-\frac{t-t_i}{\tau_e}\right)$ $w \in [w_{\min}, w_{\max}]$

F: Input		
Type	Target	Description
Stimulus generator	U	N_{stim} spikes at random firing times distributed uniformly within the trial duration.
Poisson generators	B	Independent Poisson spike-trains with rate ν_B

Numerical values for parameters. For parameter definitions see Table A and text. This table was previously published in Friedmann et al. (2013).

Parameter	Value
N_U	250
N_B	$N_T \cdot 250$
N_T	5
C_m	500 pF
g_L	10 nS
E_L	-70 mV
E_e	0 mV
τ_{ref}	10 ms
V_{reset}	-60 mV
V_{th}	-50 mV
$A_{\pm}$	$\pm$32 pS
$\tau_{\pm}$	20 ms
τ_e	0.1 . . . 1000 s
$w_{\min}$	0 nS
$w_{\max}$	0.5 nS
$\hat{W}$	0.45 nS
t_{end}	1 s

B. Supported Power ISA subset

Table B.1.: Subset of implemented registers.

Register name	**Abbreviation**	**Number of registers**
General purpose registers	GPR	32
Special purpose registers	SPR	29
Condition register	CR	1
Fixed-point exception register	XER	1
Link register	LNK	1
Counter register	CTR	1
Machine state register	MSR	1
Interrupt related:		
Save/restore register (PC)	SRR0	1
Save/restore register (MSR)	SRR1	1
Critical save/restore register (PC)	CSRR0	1
Critical save/restore register (MSR)	CSRR1	1
Machine check save/restore register (PC)	MCSRR0	1
Machine check save/restore register (MSR)	MCSRR1	1
Exception syndrome register	ESR	1
Data exception address register[1]	DEAR	1
Timer facility:		
Time base upper register	TBU	1
Time base lower register	TBL	1
Decrementer	DEC	1
Decrementer auto reload register	DECAR	1
Timer control register	TCR	1
Timer status register	TSR	1
Not in Power ISA:		
General purpose output register	GOUT	1
General purpose input register	GIN	1
General purpose output enable register	GOE	1
Interrupt configuration & control register	ICCR	1

[1]Not set by the alignement exception

Subset of implemented instructions sorted by category and instruction name. Square brackets indicate optional variants. A dot [.] denotes recording to condition register field 0 and [o] indicates recording overflow to the fixed-point exception register. For branches [l] indicates branch and link feature and [a], that absolute addresses are to be used. Instruction names and mnemonics as used by PowerISA (2010). The last column gives the configurable range of the issue-to-retire latency L (see Section 3.1.7 for the definition of L). For load/store the fixed-latency values are given. If the variable latency variant is used, an arbitrary configurable expected lantency can be used for optimization. The true latency is of course variable. The same is the case for external control instructions.

Table B.2.: Implemented instructions

Instruction	**Mnemonic**	**Category**	**FU**	L
Add	add[o][.]	Base	Fixed-point	2-4
Add Carrying	addc[o][.]	Base	Fixed-point	2-4
Add Extended	adde[o][.]	Base	Fixed-point	2-4
Add Immediate	addi	Base	Fixed-point	2-4
Add Immediate Carrying	addic[.]	Base	Fixed-point	2-4
Add Immediate Shifted	addis	Base	Fixed-point	2-4
Add to Minus One Extended	addme[o][.]	Base	Fixed-point	2-4
Add to Zero Extended	addze[o][.]	Base	Fixed-point	2-4
And	and[.]	Base	Fixed-point	2-4
And with Complement	andc[.]	Base	Fixed-point	2-4
And Immediate	andi.	Base	Fixed-point	2-4
And Immediate Shifted	andis.	Base	Fixed-point	2-4
Branch	b[l][a]	Base	Branch	1
Branch Conditional	bc[l][a]	Base	Branch	1
Branch Conditional to Count Register	bcctr[l]	Base	Branch	1
Branch Conditional to Link Register	bclr[l]	Base	Branch	1
Compare	cmp	Base	Fixed-point	2-4
Compare Immediate	cmpi	Base	Fixed-point	2-4
Compare Logical	cmpl	Base	Fixed-point	2-4
Compare Logical Immediate	cmpli	Base	Fixed-point	2-4
Count Leading Zeros Word	cntlzw	Base	Fixed-point	2-4
Condition Register And	crand	Base	Fixed-point	2-4
Condition Register And with Complement	crandc	Base	Fixed-point	2-4
Condition Register Equivalent	creqv	Base	Fixed-point	2-4
Condition Register Not And	crnand	Base	Fixed-point	2-4
Condition Register Not Or	crnor	Base	Fixed-point	2-4
Condition Register Or	cror	Base	Fixed-point	2-4
Condition Register Or with Complement	crorc	Base	Fixed-point	2-4
Condition Register eXclusive Or	crxor	Base	Fixed-point	2-4
Divide Word	divw[o][.]	Base	Divide	≥ 8
Divide Word Extended	divwe[o][.]	Base	Divide	≥ 8

Continued on next page

Continued from previous page

Instruction	**Mnemonic**	**Category**	**FU**	*L*
Divide Word Extended Unsigned	divweu[o][.]	Base	Divide	≥ 8
Divide Word Unsigned	divwu[o][.]	Base	Divide	≥ 8
Equivalent	eqv[.]	Base	Fixed-point	2-4
Extend Sign Byte	extsb[.]	Base	Fixed-point	2-4
Extend Sign Halfword	extsh[.]	Base	Fixed-point	2-4
Load Byte and Zero	lbz	Base	Load/Store	3-4
Load Byte and Zero with Update	lbzu	Base	Load/Store	3-4
Load Byte and Zero with Update Indexed	lbzux	Base	Load/Store	3-4
Load Byte and Zero Indexed	lbzx	Base	Load/Store	3-4
Load Halfword Algebraic	lha	Base	Load/Store	3-4
Load Halfword Algebraic with Update	lhau	Base	Load/Store	3-4
Load Halfword Algebraic with Update Indexed	lhaux	Base	Load/Store	3-4
Load Halfword Algebraic Indexed	lhax	Base	Load/Store	3-4
Load Halfword and Zero	lhz	Base	Load/Store	3-4
Load Halfword and Zero with Update	lhzu	Base	Load/Store	3-4
Load Halfword and Zero with Update Indexed	lhzux	Base	Load/Store	3-4
Load Halfword and Zero Indexed	lhzx	Base	Load/Store	3-4
Load Multiple Word	lmw	Base	Load/Store	3-4
Load Word and Zero	lwz	Base	Load/Store	3-4
Load Word and Zero with Update	lwzu	Base	Load/Store	3-4
Load Word and Zero with Update Indexed	lwzux	Base	Load/Store	3-4
Load Word and Zero Indexed	lwzx	Base	Load/Store	3-4
Move Condition Register Field	mcrf	Base	Fixed-point	2-4
Move From Machine State Register	mfmsr	Base	Fixed-point	2-4
Move From One Condition Register Field	mfocrf	Base	Fixed-point	2-4
Move From Special Purpose Register	mfspr	Base	Fixed-point	2-4
Move To One Condition Register Field	mtocrf	Base	Fixed-point	2-4
Move To Special Purpose Register	mtspr	Base	Fixed-point	2-4
Multiply High Word	mulhw[.]	Base	Multiply	≥ 2
Multiply High Word Unsigned	mulhwu[.]	Base	Multiply	≥ 2
Multiply Low Immediate	mulli	Base	Multiply	≥ 2
Multiply Low Word	mullw[o][.]	Base	Multiply	≥ 2
Not And	nand[.]	Base	Fixed-point	2-4
Negate	neg[o][.]	Base	Fixed-point	2-4
Not Or	nor[.]	Base	Fixed-point	2-4
Or	or[.]	Base	Fixed-point	2-4
Or with Complement	orc[.]	Base	Fixed-point	2-4
Or Immediate	ori	Base	Fixed-point	2-4
Or Immediate Shifted	oris	Base	Fixed-point	2-4
Population Count Bytes	popcntb	Base	Fixed-point	2-4
Parity Word	prtyw	Base	Fixed-point	2-4
Rotate Left Word Immediate then Mask Insert	rlwimi[.]	Base	Fixed-point	2-4

Continued on next page

Continued from previous page

Instruction	**Mnemonic**	**Category**	**FU**	*L*
Rotate Left Word Immediate then And with Mask	rlwimi[.]	Base	Fixed-point	2-4
Rotate Left Word Immediate then And with Mask	rlwinm[.]	Base	Fixed-point	2-4
Rotate Left Word Then And with Mask	rlwnm[.]	Base	Fixed-point	2-4
Shift Left Word	slw[.]	Base	Fixed-point	2-4
Shift Right Algebraic Word	sraw[.]	Base	Fixed-point	2-4
Shift Right Algebraic Word Immediate	srawi[.]	Base	Fixed-point	2-4
Shift Right Word	srw[.]	Base	Fixed-point	2-4
Store Byte	stb	Base	Load/Store	3-4
Store Byte with Update	stbu	Base	Load/Store	3-4
Store Byte with Update Indexed	stbux	Base	Load/Store	3-4
Store Byte Indexed	stbx	Base	Load/Store	3-4
Store Halfword	sth	Base	Load/Store	3-4
Store Halfword with Update	sthu	Base	Load/Store	3-4
Store Halfword with Update Indexed	sthux	Base	Load/Store	3-4
Store Halfword Indexed	sthx	Base	Load/Store	3-4
Store Multiple Word	stmw	Base	Load/Store	3-4
Store Word	stw	Base	Load/Store	3-4
Store Word with Update	stwu	Base	Load/Store	3-4
Store Word with Update Indexed	stwux	Base	Load/Store	3-4
Store Word Indexed	stwx	Base	Load/Store	3-4
Subtract From	subf[o][.]	Base	Fixed-point	2-4
Subtract From Carrying	subfc[o][.]	Base	Fixed-point	2-4
Subtract From Extended	subfe[o][.]	Base	Fixed-point	2-4
Subtract From Immediate Carrying	subfic	Base	Fixed-point	2-4
Subtract From Minus One Extended	subfme[o][.]	Base	Fixed-point	2-4
Subtract From Zero Extended	subfze[o][.]	Base	Fixed-point	2-4
Trap Word	tw	Base	Branch	1
Trap Word Immediate	twi	Base	Branch	1
eXclusive Or	xor[.]	Base	Fixed-point	2-4
eXclusive Or Immediate	xori	Base	Fixed-point	2-4
eXclusive Or Immediate Shifted	xoris	Base	Fixed-point	2-4
Move To Machine State Register	mtmsr	Embedded	Fixed-point	2-4
Return From Critical Interrupt	rfci	Embedded	Branch	1
Return From Interrupt	rfi	Embedded	Branch	1
Return From Machine Check Interrupt	rfmci	Embedded	Branch	1
External Control In Word Indexed	eciwx	Ext. Ctrl.	Ext. Ctrl.	arb.
External Control Out Word Indexed	ecowx	Ext. Ctrl.	Ext. Ctrl.	arb.
Wait	wait	Wait	-	0

Table B.3.: Addresses for interrupts.

Interrupt	**Interrupt vector address**
Machine check	0x0001
Critical input	0x0002
Data storage	0x0003
Instruction storage	0x0004
External input	0x0005
Alignment	0x0006
Program	0x0007
System call	0x0008
Doorbell	0x0009
Critical doorbell	0x000a
Fixed-interval timer	0x000b
Decrementer	0x000c

C. Supplemental design description

C.1. External interfaces of the plasticity processor

C.1.1. RAM interface

Signal name	Direction	Width	Description
en	master to slave	1	enable interface
addr	master to slave	variable	address for read and write
data_r	slave to master	variable	result data for read
data_w	master to slave	same as data_r	input data for write
we	master to slave	1	enable write operation
be	master to slave	bytes in data_r	byte enable mask for writes
delay	slave to master	1	read data not yet available

C.1.2. Plasticity processor bus interface

Processor bus interface based on the OCP specification (OCP, 2009).

Signal name	Direction	Width	Description
Clk	to master and slave	1	bus clock
MReset_n	master to slave	1	reset active low
MAddr	master to slave	variable	address for request
MCmd	master to slave	3	requested command
MData	master to slave	variable	data transfered to slave
MRespAccept	master to slave	1	handshake for response
MByteEn	master to slave	bytes in MData	byte enable mask
SCmdAccept	slave to master	1	handshake for request
SData	slave to master	variable	data transfered to master
SResp	slave to master	2	type of response

C.2. Scheduling state machine graph

State machine diagram for the scheduling FSM. Transitions are labeled with the asserted input signals, that trigger the transition: valid instruction from instruction streamer (v), halt (h), context synchronization (s), control transfer complete (c), branch (b), interrupt (i), all RSRs empty (e), and wakeup from sleep (w). After reset, the FSM is in the `RESET` state.

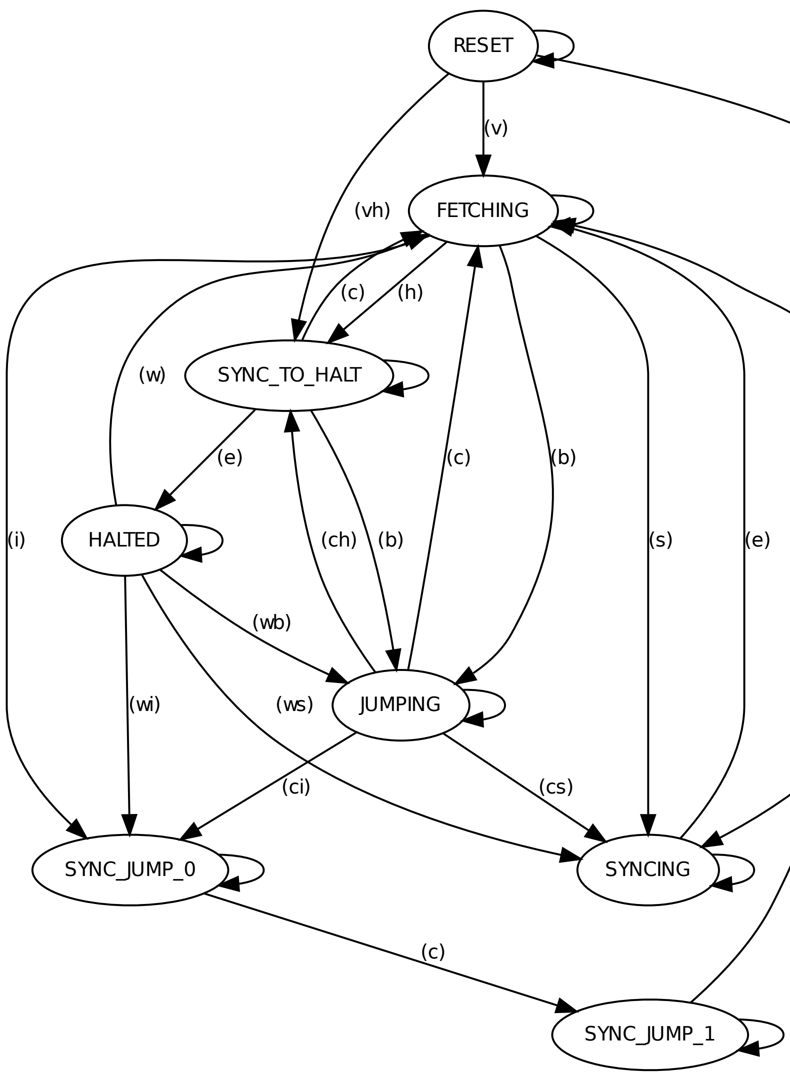


Table C.1.: The 32 bit instruction word is pre-decoded to the 123 bit control word.

Name	Width	Description
read_ra	1	Read from GPR referenced in RA
read_rb	1	Read from GPR referenced in RB
read_rt	1	Read from GPR referenced in RT
write_ra	1	Write to GPR referenced in RA
write_rt	1	Write to GPR referenced in RT
ra	5	Operand reference RA
rb	5	Operand reference RB
rt	5	Operand reference RT
b_immediate	1	Second operand is immediate
read_ctr	1	Read the counter register
write_ctr	1	Write the counter register
read_lnk	1	Read the link register
write_lnk	1	Write the link register
write_cr	8	Write for condition register fields
read_cr_0	8	Read port 0 for condition register fields
read_cr_1	8	Read port 1 for condition register fields
read_cr_2	8	Read port 2 for condition register fields
read_xer	1	Read fixed-point exception register
write_xer	1	Write fixed-point exception register
xer_dest	3	Fixed-point exception register field for write
read_spr	1	Read special purpose register SPR
read_spr2	1	Read special purpose register SPR2
spr	10	Special purpose register reference SPR
spr2	10	Special purpose register reference SPR2
write_spr	1	Write special purpose register
spr_dest	10	Special purpose register destination for write
write_mem	1	Instruction writes to memory
read_msr	1	Read machine state register
write_msr	1	Write machine state register
write_nve	1	Write to registers in NEVER functional unit
fu_set	8	Functional unit for this instruction
context_sync	1	Instruction is context synchronizing
mem_bar	1	Instruction is a memory barrier
halt	1	Halt the processor to sleep state
nd_latency	1	Instruction has variable latency
latency	3	Latency
multicycles	5	Number of multi-cycles
is_multicycle	1	Instruction is a multi-cycle instruction
is_branch	1	Instruction is a branch
is_nop	1	Instruction is a no-operation
synops	1	Is a synops instruction (SYNAPSE unit)

C.3. OCP configuration options of the plasticity processor bus

Table C.2.: OCP configuration options for the plasticity processor bus

Option name	Value
addr	1
addr_wdth	32
mdata	1
mdata_wdth	32
respaccept	1
cmdaccept	1
sdata	1
sdata_wdth	32
sresp	1
byteen	1
mreset	1
writeresp_enable	1
read_enable	1
write_enable	1
endianess	big

C.4. Synapse array interface signals

Table C.3.: List of signals of the synapse array interface.

Name	Width	Description
syn_a	8	Row address
syn_ab	8	Inverted row address
en	32	One-hot coded per-slice column selection
syn_gen	4	Global per-slice enable
syn_ensynb	1	Enable for synapse weights
syn_endecb	1	Enable for synapse decoder addresses
syn_encrb	1	Enable for synapse correlation readout
syn_endrvb	1	Enable for synapse driver configuration region DRV
syn_engmaxb	1	Enable for synapse driver configuration region GMAX
syn_enctrlb	1	Enable for synapse driver configuration region CTRL
syn_d	16	Synapse driver SRAM bitlines

Continued on next page

Continued from previous page

Name	**Width**	**Description**
syn_db	16	Synapse driver SRAM inverted bit lines
syn_en	2	Synapse driver side selection
dio	128	Bi-directional synapse data port
corrin	32	Correlation readout & evaluation output
corresetb	32	Correlation reset signals
pcb	1	Enable pre-charge on synapse bitlines
ramoeb	1	Drive dio port
ramwb	1	Drive bitlines
pattern	4	Configure readout operation
scc	1	Sense correlation “causal”
sccb	1	Inverted version of scc
sca	1	Sense correlation “acausal”
scab	1	Inverted version of sca
csen	1	Correlation sense enable to output evaluation result

Glossary

ABI Application Binary Interface. 152

ADC Analog to Digital Converter. 32, 35, 57

AHM Abstract Hybrid hardware Model. 12, 13, 30–32, 35–38, 41, 44, 45, 48, 56, 59, 60, 93, 94, 102, 104, 109, 117, 118, 205, 206, 211

ASIC Application Specific Integrated Circuit. 59, 100, 109, 126, 155, 168, 169, 171, 173, 174, 185, 209

AWG Arbitrary Waveform Generator. 175, 179, 180

CMOS Complementary Metal Oxide Semiconductor. 17, 19

CPI Clocks Per Instruction. 144

DAC Digital to Analog Converter. 169

DVFS Dynamic Voltage and Frequency Scaling. 177, 210, 211

EABI Embedded ABI. 152

ELF Exectuable and Linking Format. 154

FIFO First-In First-Out. 89, 97, 100, 219

FPGA Field Programmable Gate Array. 69, 71, 84, 90, 155, 156, 164, 166, 167, 205, 207, 208, 210

FSM Finite State Machine. 71, 72, 84–87, 91, 113, 115–118, 121, 124–126, 231

FUM Functional Unit Manager. 82–84

GCC GNU Compiler Collection. 61, 152

GIO General purpose Input/Output. 93, 169, 171, 191

GNU GNU is Not Unix. 153, 158, 159

GPR General Purpose Register. 65, 232

HEPP HICANN with Embedded Plasticity Processor. 190, 192, 193, 197, 198, 202, 204, 205, 207–209, 211, 213–217, 221

HICANN High Input Count Analog Neural Network. 19, 94, 99–102, 109, 112, 126, 183, 185, 187, 190–193, 197, 201, 212

I/O Input/Output. 30, 64–66, 88, 89, 93, 94, 113, 117–119, 126–128, 131, 134, 135, 138, 146, 149, 153, 158, 175, 177, 194, 195, 199, 205, 207, 208, 216

ILP Instruction Level Parallelism. 62

IP Intellectual Property. 166, 167, 172, 208

ISA Instruction Set Architecture. 60, 61, 65, 69, 72, 73, 80, 89, 134, 140, 152, 166, 167, 203, 207, 208, 225

JTAG Joint Test Action Group. 158, 169, 171, 172

MMU Memory Management Unit. 153, 207, 208

MSB Most Significant Bit. 68

NVE Native Vector Extension. 137, 138, 199, 208

OCP Open Core Protocol. 65, 95, 100, 230

PC Program Counter. 62, 64, 66, 68, 172

PP Plasticity Processor. 59–62, 65, 66, 69, 76, 88, 90–93, 117–119, 140, 142, 149, 153, 155, 156, 159, 165–167, 169, 172, 174, 190, 201, 205, 207, 209–211

PPB Plasticity Processor Bus. 65, 69, 71, 76, 82, 88, 93–96, 99, 100, 110, 119, 122, 124–126, 139, 151, 157, 161, 163, 192, 193, 201, 207

PSP postsynaptic potential. 30

RISC Reduced Instruction Set Computer. 61, 65, 207

RNG Random Number Generator. 196, 199, 200

RSR Result Shift Register. 72–77, 82–85, 87, 88, 161, 207, 231

RTL Register Transfer Level. 60, 77, 80, 100

SerDes Serializer/Deserializer. 99, 192, 203

SIMD Sinlge Instruction Multiple Data. 126, 135, 199, 207, 208, 215, 220

SRAM Static Random Access Memory. 65, 69, 71, 72, 102, 104–106, 109, 110, 112, 113, 124–126, 162, 169, 173, 174, 180, 181, 190, 191, 201, 204, 213, 219, 220, 233, 234

STDP Spike-Timing Dependent Plasticity. 18, 21–24, 28, 31, 39, 40, 207, 213, 214, 219, 221

TD Temporal Difference. 206

TSMC Taiwan Semiconducator Manufacturing Company. 169

UMC United Microelectronics Corporation. 169

VLSI Very Large Scale Integration. 18, 36, 59

References

Larry F Abbott and Sacha B Nelson. Synaptic plasticity: taming the beast. *Nature neuroscience*, 3:1178–1183, 2000.

Ahmed Al Maashri, Matthew Cotter, Nandhini Chandramoorthy, Michael DeBole, Chi-Li Yu, Vijaykrishnan Narayanan, and Chaitali Chakrabarti. Hardware acceleration for neuromorphic vision algorithms. *Journal of Signal Processing Systems*, 70(2):163–175, 2013. ISSN 1939-8018. doi: 10.1007/s11265-012-0699-x. URL `http://dx.doi.org/10.1007/s11265-012-0699-x`.

S. Alam, R. Barrett, M. Bast, M.R. Fahey, J. Kuehn, C. McCurdy, J. Rogers, P. Roth, R. Sankaran, J.S. Vetter, et al. Early evaluation of ibm bluegene/p. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 23. IEEE Press, 2008.

R. Ananthanarayanan, S.K. Esser, H.D. Simon, and D.S. Modha. The cat is out of the bag: cortical simulations with 10 9 neurons, 10 13 synapses. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 63. ACM, 2009.

ARM Ltd. Armv5 architecture reference manual, 2007. URL `http://infocenter.arm.com/help/index.jsp`.

Atmel. 8-bit atmel microcontroller with 64k/128k/256k bytes in-system programmable flash, October 2012. URL `http://www.atmel.com/Images/doc2549.pdf`. revision P.

John Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, August 1978. ISSN 0001-0782. doi: 10.1145/359576.359579. URL `http://doi.acm.org/10.1145/359576.359579`.

G. Q. Bi and M. M. Poo. Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type. *The Journal of neuroscience : the official journal of the Society for Neuroscience*, 18(24):10464–10472, December 1998. ISSN 0270-6474. URL `http://www.jneurosci.org/content/18/24/10464.abstract`.

Johannes Bill, Klaus Schuch, Daniel Brüderle, Johannes Schemmel, Wolfgang Maass, and Karlheinz Meier. Compensating inhomogeneities of neuromorphic VLSI devices via short-term synaptic plasticity. *Front. Comp. Neurosci.*, 4(129), 2010.

Shekhar Borkar and Andrew A Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

BrainScaleS. Research. `http://brainscales.kip.uni-heidelberg.de/public/index.html`, 2012.

Daniel Brüderle. *Neuroscientific Modeling with a Mixed-Signal VLSI Hardware System*. PhD thesis, 2009.

Daniel Brüderle, Johannes Bill, Bernhard Kaplan, Jens Kremkow, Karlheinz Meier, Eric Müller, and Johannes Schemmel. Simulator-like exploration of cortical network architectures with a mixed-signal vlsi system. In *Proceedings of the 2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2784–2787, 2010.

Inc. Cadence Design Systems. Encounter digital implementation system. www.cadence.com, 2012.

K. Cameron, V. Boonsobhak, A. Murray, and D. Renshaw. Spike timing dependent plasticity (stdp) can ameliorate process variations in neuromorphic vlsi. *Neural Networks, IEEE Transactions on*, 16(6):1626–1637, 2005. ISSN 1045-9227. doi: 10.1109/TNN.2005.852238.

Natalia Caporale and Yang Dan. Spike timing-dependent plasticity: A hebbian learning rule. *Annual review of neuroscience*, February 2008. ISSN 0147-006X. doi: http://dx.doi.org/10.1146/annurev.neuro.31.060407.125639.

Leon O Chua and Sung Mo Kang. Memristive devices and systems. *Proceedings of the IEEE*, 64 (2):209–223, 1976.

LLC. CompuGreen. Green 500 list. Website, November 2012. URL `http://www.green500.org/lists/green201211`.

R. H. Cudmore and N. S. Desai. Intrinsic plasticity. 3(2):1363, 2008.

S. Davies, F. Galluppi, A.D. Rast, and S.B. Furber. A forecast-based stdp rule suitable for neuromorphic implementation. *Neural Networks*, 32(0):3 – 14, 2012. ISSN 0893-6080. doi: 10.1016/j.neunet.2012.02.018. URL `http://www.sciencedirect.com/science/article/pii/S0893608012000470`. Selected Papers from IJCNN 2011.

DesignWare. Designware datapath building block ip, 2013. URL `http://www.synopsys.com/dw/buildingblock.php`.

R. Douglas, M. Mahowald, and C. Mead. Neuromorphic analogue VLSI. *Annu. Rev. Neurosci.*, 18:255–281, 1995.

Embedded Microprocessor Benchmark Consortium EEMBC. Coremark benchmark. website, 2012. URL `http://www.coremark.org/`.

Embedded Microprocessor Benchmark Consortium EEMBC. Coremark benchmark scores. Website, April 2013. URL `http://www.coremark.org/benchmark/index.php?pg=benchmark`.

FACETS. Fast Analog Computing with Emergent Transient States – project website. `http://www.facets-project.org`, 2010.

Michael A. Farries and Adrienne L. Fairhall. Reinforcement learning with modulated spike timingâĂŞdependent synaptic plasticity. *Journal of Neurophysiology*, 98(6):3648–3665, December 2007. doi: 10.1152/jn.00364.2007. URL `http://jn.physiology.org/content/98/6/3648.abstract`.

RÄČzvan V. Florian. Reinforcement learning through modulation of spike-timing-dependent synaptic plasticity. *Neural Computation*, 19(6):1468–1502, April 2007. ISSN 0899-7667. URL `http://dx.doi.org/10.1162/neco.2007.19.6.1468`.

Michael J Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972.

Free Software Foundation. Binutils website. website, March 2013. URL `http://www.gnu.org/software/binutils/`. Version 2.21.51.

Nicolas Frémaux, Henning Sprekeler, and Wulfram Gerstner. Functional requirements for reward-modulated spike-timing-dependent plasticity. *The Journal of Neuroscience*, 30:13326–13337, 2010.

Nicolas Frémaux, Henning Sprekeler, and Wulfram Gerstner. Reinforcement learning using a continuous time actor-critic framework with spiking neurons. *PLoS Comput Biol*, 9(4): e1003024, 04 2013. doi: 10.1371/journal.pcbi.1003024. URL `http://dx.doi.org/10.1371%2Fjournal.pcbi.1003024`.

Simon Friedmann, Nicolas Frémaux, Johannes Schemmel, Wulfram Gerstner, and Karlheinz Meier. Reward-based learning under hardware constraints - using a RISC processor in a neuromorphic system. *Frontiers in Neuromorphic Engineering*, 2013. URL `http://arxiv.org/abs/1303.6708`. submitted.

Robert C Froemke, Dominique Debanne, and Guo-Qiang Bi. Temporal modulation of spike-timing-dependent plasticity. *Frontiers in Synaptic Neuroscience*, 2(19), 2010a. ISSN 1663-3563. doi: 10.3389/fnsyn.2010.00019. URL `http://www.frontiersin.org/synaptic_neuroscience/10.3389/fnsyn.2010.00019/abstract`.

Robert C Froemke, Johannes J Letzkus, Bjorn Kampa, Giao B Hang, and Greg Stuart. Dendritic synapse location and neocortical spike-timing-dependent plasticity. *Frontiers in Synaptic Neuroscience*, 2(29), 2010b. ISSN 1663-3563. doi: 10.3389/fnsyn.2010.00029. URL `http://www.frontiersin.org/synaptic_neuroscience/10.3389/fnsyn.2010.00029/abstract`.

S.H. Fuller and L.I. Millett. Computing performance: Game over or next level? *Computer*, 44 (1):31–38, 2011. ISSN 0018-9162. doi: 10.1109/MC.2011.15.

Steve B. Furber, David R. Lester, Luis A. Plana, Jim D. Garside, Eustace Painkras, Steve Temple, and Andrew D. Brown. Overview of the SpiNNaker system architecture. *IEEE Transactions on Computers*, 99(PrePrints), 2012. ISSN 0018-9340. doi: http://doi.ieeecomputersociety.org/10.1109/TC.2012.142.

W. Gerstner, R. Kempter, J.L. Van Hemmen, H. Wagner, et al. A neuronal learning rule for sub-millisecond temporal coding. *Nature*, 383(6595):76–78, 1996.

Ricardo Gonzalez and Mark Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1284, September 1996.

Paul R. Gray, Paul J. Hurst, Stephen H. Lewis, and Robert G. Meyer. Analysis and design of analog integrated circuits, fourth edition. John Wiley & Sons, 2001. ISBN 0-471-32168-0.

Andreas Grübl. Personal communication, August 2012.

R. Gütig, R. Aharonov, S. Rotter, and Haim Sompolinsky. Learning input correlations through nonlinear temporally asymmetric hebbian plasticity. *The Journal of Neuroscience*, 23(9):3697–3714, 2003. URL `http://www.jneurosci.org/content/23/9/3697.abstract`.

P. Hafliger. Adaptive wta with an analog vlsi neuromorphic learning chip. *Neural Networks, IEEE Transactions on*, 18(2):551–572, 2007. ISSN 1045-9227. doi: 10.1109/TNN.2006.884676.

Andreas Hartel, Gvidas Sidlauskas, and Andreas Grübl. tc65nm back-end, 2011. personal communication.

John L. Hennessy and David A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann, Amsterdam, 2007.

Matthias Hock, Andreas Hartel, and Johannes Schemmel. The route65 prototype chip, April 2013. personal communication.

Ronald W. Holz and Stephen K. Fisher. *Basic Neurochemistry: Molecular, Cellular and Medical Aspects*. Lippincott-Raven, 1999. URL `http://www.ncbi.nlm.nih.gov/books/NBK27911/`.

IBM. Ppc405fx embedded processor core userâĂŹs manual, January 2005.

Microcontroller Applications IBM. Developing powerpc embedded application binary interface (eabi) compliant programs, September 1998a. Version 1.0.

Microelectronics Division IBM. The powerpc 405 core. Whitepaper, November 1998b.

G. Indiveri, E. Chicca, and R. Douglas. A VLSI array of low-power spiking neurons and bistable synapses with spike-timing dependent plasticity. *IEEE Transactions on Neural Networks*, 17(1): 211–221, Jan 2006.

E.M. Izhikevich. Solving the distal reward problem through linkage of stdp and dopamine signaling. *Cerebral Cortex*, 17(10):2443–2452, 2007a.

Eugene M. Izhikevich. Solving the distal reward problem through linkage of stdp and dopamine signaling. *Cerebral Cortex*, 17(10):2443–2452, 2007b. doi: 10.1093/cercor/bhl152. URL `http://cercor.oxfordjournals.org/content/17/10/2443.abstract`.

E. R. Kandel, J. H. Schwartz, and T. M. Jessell. *Principles of Neural Science*. McGraw-Hill, New York, 4 edition, 2000.

Bernard Katz. *The release of neural transmitter substances*, volume 10. Liverpool University Press, 1969.

Brian W. Kernighan and Dennis M. Ritchie. The m4 macro processor. Technical report, Bell Laboratories, Murray Hill, New Jersey 07974, July 1977.

MM Khan, DR Lester, Luis A Plana, A Rast, X Jin, E Painkras, and Stephen B Furber. Spinnaker: mapping neural networks onto a massively-parallel chip multiprocessor. In *Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, pages 2849–2856. IEEE, 2008.

P.R. Kinget. Device mismatch: an analog design perspective. In *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pages 1245–1248. IEEE, 2007.

Alex Kononov. Testing of an analog neuromorphic network chip. Diploma thesis (English), University of Heidelberg, HD-KIP-11-83, 2011.

Alexander Kononov. Personal communication, 2013.

Raphael Lamprecht and Joseph LeDoux. Structural plasticity and memory. *Nature Reviews Neuroscience*, 5(1):45–54, 2004.

R. Legenstein, D. Pecevski, and W. Maass. A learning theory for reward-modulated spike-timing-dependent plasticity with application to biofeedback. *PLoS Computational Biology*, 4 (10):e1000180, 2008.

Benedetta Leuner and Elizabeth Gould. Structural plasticity and hippocampal function. *Annual Review of Psychology*, 61(1):111–140, 2010. doi: 10.1146/annurev.psych.093008.100359. URL `http://www.annualreviews.org/doi/abs/10.1146/annurev.psych.093008.100359`. PMID: 19575621.

J. Madrenas and J.M. Moreno. Strategies in simd computing for complex neural bioinspired applications. In *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on*, pages 376–381, 2009. doi: 10.1109/AHS.2009.31.

H. Markram. The blue brain project. *Nature Reviews Neuroscience*, 7(2):153–160, 2006.

H. Markram, J. Lübke, and B. Sakmann. Regulation of synaptic efficacy by coincidence of postsynaptic aps. *Science*, 275:213–215, 1997.

Henry Markram, Wulfram Gerstner, and Per Jesper Sjöström. A history of spike-timing-dependent plasticity. *Frontiers in synaptic neuroscience*, 3, 2011.

Stephen Martin, P. D. Grimwood, and R. G. M. Morris. Synaptic plasticity and memory: An evaluation of the hypothesis. *Annual Review of Neuroscience*, 23:649–711, March 2000. doi: 10.1146/annurev.neuro.23.1.649.

C. A. Mead. *Analog VLSI and Neural Systems*. Addison Wesley, Reading, MA, 1989.

C. A. Mead. Neuromorphic electronic systems. *Proceedings of the IEEE*, 78:1629–1636, 1990.

Sebastian Millner. *Development of a Multi-Compartment Neuron Model Emulation*. PhD thesis, November 2012. URL `http://www.ub.uni-heidelberg.de/archiv/13979`.

Sebastian Millner. Personal communication, 2013.

Sebastian Millner, Andreas Grübl, Karlheinz Meier, Johannes Schemmel, and Marc-Olivier Schwartz. A VLSI implementation of the adaptive exponential integrate-and-fire neuron model. In J. Lafferty et al., editors, *Advances in Neural Information Processing Systems 23*, pages 1642–1650, 2010.

S. Mitra, S. Fusi, and G. Indiveri. Real-time classification of complex patterns using spike-based learning in neuromorphic vlsi. *Biomedical Circuits and Systems, IEEE Transactions on*, 3(1): 32–42, 2009. ISSN 1932-4545. doi: 10.1109/TBCAS.2008.2005781.

Abigail Morrison, Markus Diesmann, and Wulfram Gerstner. Phenomenological models of synaptic plasticity based on spike timing. *Biological Cybernetics*, 98(6):459–478, June 2008. ISSN 0340-1200. doi: 10.1007/s00422-008-0233-1.

Tobias Nonnenmacher. Verification of an embedded processor for synaptic plasticity, August 2011. Bachelor thesis.

Eilen Nordlie, Marc-Oliver Gewaltig, and Hans Ekkehard Plesser. Towards reproducible descriptions of neuronal network models. *PLoS Comput Biol*, 5(8):e1000456, 08 2009. doi: 10.1371/journal.pcbi.1000456. URL `http://dx.doi.org/10.1371%2Fjournal.pcbi.1000456`.

NXP Semiconductors. Lpc2939 arm9 microcontroller with can, lin, and usb, April 2010. URL `http://www.nxp.com/documents/data_sheet/LPC2939.pdf`. Rev. 3.

OCP. Open core protocol specification 3.0, 2009. URL `http://www.ocpip.org/home`.

OpenCores. Or1200 openrisc processor. Website, April 2013. URL `http://opencores.org/or1k/OR1200_OpenRISC_Processor`.

OpenRISC Project. Website, April 2013. URL `openrisc.net`.

Nikolai Otmakhov, Aneil M. Shirke, and Roberto Malinow. Measuring the impact of probabilistic transmission on neuronal output. *Neuron*, 10(6):1101 – 1111, 1993. ISSN 0896-6273. doi: 10.1016/0896-6273(93)90058-Y. URL `http://www.sciencedirect.com/science/article/pii/089662739390058Y`.

B. Pakkenberg and H.J. Gundersen. Neocortical Neuron Number in Humans: Effect of Age and Sex. *J Comp Neurol.*, 384(2):312–320, July 1997.

Bente Pakkenberg, Dorte Pelvig, Lisbeth Marner, Mads J Bundgaard, Hans Jørgen G Gundersen, Jens R Nyengaard, and Lisbeth Regeur. Aging and the human neocortex. *Experimental gerontology*, 38(1):95–99, 2003.

David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2 edition, 1996.

Verena Pawlak and Jason ND Kerr. Dopamine receptor activation is required for corticostriatal spike-timing-dependent plasticity. *The Journal of Neuroscience*, 28(10):2435–2446, 2008.

Verena Pawlak, Jeffery R Wickens, Alfredo Kirkwood, and Jason ND Kerr. Timing is not everything: neuromodulation opens the stdp gate. *Frontiers in synaptic neuroscience*, 2, 2010.

R. Peter et al. Synaptic density in human frontal cortexâĂŤdevelopmental changes and effects of aging. *Brain research*, 163(2):195–205, 1979.

W.W. Peterson and D.T. Brown. Cyclic codes for error detection. *Proceedings of the IRE*, 49(1): 228–235, 1961. ISSN 0096-8390. doi: 10.1109/JRPROC.1961.287814.

Thomas Pfeil. Personal communication, 2012.

Thomas Pfeil, Tobias C Potjans, Sven Schrader, Wiebke Potjans, Johannes Schemmel, Markus Diesmann, and Karlheinz Meier. Is a 4-bit synaptic weight resolution enough? - constraints on enabling spike-timing dependent plasticity in neuromorphic hardware. *Frontiers in Neuroscience*, 6(90), 2012. ISSN 1662-453X. doi: 10.3389/fnins.2012.00090.

D Plenz and Ad Aertsen. Neural dynamics in cortex-striatum co-cultures–ii. spatiotemporal characteristics of neuronal activity. *Neuroscience*, 70(4):893–924, Feb 1996.

Wiebke Potjans, Markus Diesmann, and Abigail Morrison. An imperfect dopaminergic error signal can drive temporal-difference learning. *PLoS Comput Biol*, 7(5):e1001133, 05 2011. doi: 10.1371/journal.pcbi.1001133. URL `http://dx.doi.org/10.1371%2Fjournal.pcbi.1001133`.

Johan Pouwelse, Koen Langendoen, and Henk Sips. Dynamic voltage scaling on a low-power microprocessor. In *Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 251–259. ACM, 2001.

PowerISA. PowerISA version 2.03. Technical report, power.org, September 2006. Available at `http://www.power.org/resources/reading/`.

PowerISA. PowerISA version 2.06 revision b. Technical report, power.org, July 2010. Available at `http://www.power.org/resources/reading/`.

W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T Vetterling. *Numerical recipes in C: The art of scientific computing*. Cambridge University Press, online ed. edition, 1992. URL `http://www.nr.com`.

S. Ramakrishnan, P.E. Hasler, and C. Gordon. Floating gate synapses with spike-time-dependent plasticity. *Biomedical Circuits and Systems, IEEE Transactions on*, 5(3):244–252, 2011.

R. Rescorla. Rescorla-wagner model. *Scholarpedia*, 3(3):2237, 2008. doi: 10.4249/scholarpedia.2237. URL `http://www.scholarpedia.org/article/Rescorla-Wagner_model`.

Robert A Rescorla and Allan R Wagner. A theory of pavlovian conditioning: Variations in the effectiveness of reinforcement and nonreinforcement. *Classical conditioning II: Current research and theory*, pages 64–99, 1972.

N.J. Rohrer, M. Canada, E. Cohen, M. Ringler, M. Mayfield, P. Sandon, P. Kartschoke, J. Heaslip, J. Allen, P. McCormick, T. Pfluger, J. Zimmerman, C. Lichtenau, T. Werner, G. Salem, M. Ross, D. Appenzeller, and D. Thygesen. Powerpc 970 in 130 nm and 90 nm technologies. In *Solid-State Circuits Conference, 2004. Digest of Technical Papers. ISSCC. 2004 IEEE International*, pages 68–69 Vol.1, 2004. doi: 10.1109/ISSCC.2004.1332597.

Kaushik Roy, Saibal Mukhopadhyay, and Hamid Mahmoodi-Meimand. Leakage current mechanisms and leakage reduction techniques in deep-submicrometer cmos circuits. In *Proceedings of the IEEE*, volume 91, pages 305 – 327. IEEE, February 2003.

Rachael I Scahill, Chris Frost, Rhian Jenkins, Jennifer L Whitwell, Martin N Rossor, and Nick C Fox. A longitudinal study of brain volume changes in normal aging using serial registered magnetic resonance imaging. *Archives of neurology*, 60(7):989, 2003.

J. Schemmel. personal communication, 2012.

J. Schemmel, K. Meier, and E. Muller. A new VLSI model of neural microcircuits including spike time dependent plasticity. In *Proceedings of the 2004 International Joint Conference on Neural Networks (IJCNN'04)*, pages 1711–1716. IEEE Press, 2004.

J. Schemmel, A. Grübl, K. Meier, and E. Muller. Implementing synaptic plasticity in a VLSI spiking neural network model. In *Proceedings of the 2006 International Joint Conference on Neural Networks (IJCNN)*. IEEE Press, 2006.

J. Schemmel, D. Brüderle, K. Meier, and B. Ostendorf. Modeling synaptic plasticity within networks of highly accelerated I&F neurons. In *Proceedings of the 2007 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 3367–3370. IEEE Press, 2007.

J. Schemmel, J. Fieres, and K. Meier. Wafer-scale integration of analog neural networks. In *Proceedings of the 2008 International Joint Conference on Neural Networks (IJCNN)*, 2008.

J. Schemmel, D. Brüderle, A. Grübl, M. Hock, K. Meier, and S. Millner. A wafer-scale neuromorphic hardware system for large-scale neural modeling. In *Proceedings of the 2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1947–1950, 2010.

Johannes Schemmel. Systemverilog sram controller. Personal communication, 2011.

Johannes Schemmel, Andreas Grübl, Sebastian Millner, and Simon Friedmann. Specification of the HICANN microchip. FACETS and BrainScaleS project internal documentation, 2012.

Stefan Scholze, Stefan Schiefer, Johannes Partzsch, Stephan Hartmann, Christian Georg Mayr, Sebastian Höppner, Holger Eisenreich, Stephan Henker, Bernhard Vogginger, and Rene Schüffny. VLSI implementation of a 2.8GEvent/s packet based AER interface with routing and event sorting functionality. *Frontiers in Neuromorphic Engineering*, 5(117):1–13, 2011.

Wolfram Schultz, Peter Dayan, and P Read Montague. A neural substrate of prediction and reward. *Science*, 275(5306):1593–1599, 1997.

J. Seo, B. Brezzo, Y. Liu, B.D. Parker, S.K. Esser, R.K. Montoye, B. Rajendran, J.A. Tierno, L. Chang, D.S. Modha, and D.J. Friedman. A 45nm cmos neuromorphic chip with a scalable architecture for learning in networks of spiking neurons. In *Custom Integrated Circuits Conference (CICC), 2011 IEEE*, pages 1 –4, sept. 2011. doi: 10.1109/CICC.2011.6055293.

Geun Hee Seol, Jokubas Ziburkus, ShiYong Huang, Lihua Song, In Tae Kim, Kogo Takamiya, Richard L Huganir, Hey-Kyoung Lee, and Alfredo Kirkwood. Neuromodulators control the polarity of spike-timing-dependent synaptic plasticity. *Neuron*, 55(6):919–929, 2007.

M. Shafi, Y. Zhou, J. Quintana, C. Chow, J. Fuster, and M. Bodner. Variability in neuronal activity in primate cortex during working memory tasks. *Neuroscience*, 146(3):1082 – 1108, 2007. ISSN 0306-4522. doi: 10.1016/j.neuroscience.2006.12.072. URL `http://www.sciencedirect.com/science/article/pii/S0306452206017593`.

D. Shapiro, J. Parri, J. M Desmarais, V. Groza, and M. Bolic. Asips for artificial neural networks. In *Applied Computational Intelligence and Informatics (SACI), 2011 6th IEEE International Symposium on*, pages 529–533, 2011. doi: 10.1109/SACI.2011.5873060.

Sadique Sheik, Martin Coath, Giacomo Indiveri, Susan L Denham, Thomas Wennekers, and Elisabetta Chicca. Emergent auditory feature tuning in a real-time neuromorphic vlsi system. *Frontiers in Neuroscience*, 6(17), 2012. ISSN 1662-453X. doi: 10.3389/fnins.2012.00017. URL `http://www.frontiersin.org/neuromorphic_engineering/10.3389/fnins.2012.00017/abstract`.

K.L. Shepard and V. Narayanan. Noise in deep submicron digital design. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 524–531. IEEE Computer Society, 1997.

Tajana Simunic, Luca Benini, Andrea Acquaviva, Peter Glynn, and Giovanni De Micheli. Dynamic voltage scaling and power management for portable systems. In *Proceedings of the 38th annual Design Automation Conference*, pages 524–529. ACM, 2001.

Jesper Sjöström and Wulfram Gerstner. Spike-timing dependent plasticity. *Scholarpedia*, 5 (2):1362, 2010. doi: 10.4249/scholarpedia.1362. URL `http://www.scholarpedia.org/article/Spike-timing_dependent_plasticity`.

Per Jesper Sjöström, Gina G Turrigiano, and Sacha B Nelson. Rate, timing, and cooperativity jointly determine cortical synaptic plasticity. *Neuron*, 32(6):1149 – 1164, 2001. ISSN 0896-6273. doi: 10.1016/S0896-6273(01)00542-6. URL `http://www.sciencedirect.com/science/article/pii/S0896627301005426`.

Per Jesper Sjöström, Gina G. Turrigiano, and Sacha B. Nelson. Endocannabinoid-dependent neocortical layer-5 ltd in the absence of postsynaptic spiking. *Journal of Neurophysiology*, 92(6):3338–3343, 2004. doi: 10.1152/jn.00376.2004. URL `http://jn.physiology.org/content/92/6/3338.abstract`.

James E. Smith. A study of branch prediction strategies. In *25 years of the international symposia on Computer architecture (selected papers)*, ISCA '98, pages 202–215, New York, NY, USA, 1998. ACM. ISBN 1-58113-058-9. doi: 10.1145/285930.285980. URL `http://doi.acm.org/10.1145/285930.285980`.

J.E. Smith and A.R. Pleszkun. *Implementation of precise interrupts in pipelined processors*, volume 13. IEEE Computer Society Press, 1985.

G.S. Snider. Spike-timing-dependent learning in memristive nanodevices. In *Nanoscale Architectures, 2008. NANOARCH 2008. IEEE International Symposium on*, pages 85–92, 2008. doi: 10.1109/NANOARCH.2008.4585796.

Sen Song and L. F. Abbott. Cortical development and remapping through spike timing-dependent plasticity. *Neuron*, 32(2):339–350, October 2001. URL `http://www.sciencedirect.com/science/article/B6WSS-4C5RF9F-K/2/4ea9cc32d87453c29c4d5247b9b38995`.

Sen Song, Kenneth D Miller, and Larry F Abbott. Competitive hebbian learning through spike-timing-dependent synaptic plasticity. *Nature neuroscience*, 3(9):919–926, 2000.

J. E. R. Staddon and Y. Niv. Operant conditioning. *Scholarpedia*, 3(9):2318, 2008. URL `http://www.scholarpedia.org/article/Operant_conditioning`.

Richard Stallman. *Using the GNU Compiler Collection*. Free Software Foundation, 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA, for gcc version 4.5.4 edition, 2012. URL `http://gcc.gnu.org`.

Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. The missing memristor found. *Nature*, 453(7191):80–83, 2008.

R.S. Sutton and A.G. Barto. *Reinforcement learning: An introduction*, volume 1. Cambridge Univ Press, 1998.

Inc. Synopsys. Design compiler. www.synopsys.com, 2012.

Synplify Premier Fast, Reliable FPGA Implementation and Debug. Synopsys, Inc., 700 East Middlefield Road, Mountain View, CA 94043, 2012.

SystemVerilog. *SystemVerilog 3.1a Language Reference Manual*. Accellera, 2004.

Texas Instruments. Msp430f543x, msp430f541x mixed signal microcontroller, MArch 2010. URL `http://www.ti.com/lit/gpn/msp430f5438`. Revision C.

A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1937.

J. D. Victor and K. P. Purpura. Nature and precision of temporal coding in visual cortex: a metric-space analysis. *J Neurophysiol*, 76(2):1310–1326, 1996.

Corinna Vinschen and Jeff Johnston. newlib c library. website, March 2013. URL `http://sourceware.org/newlib/`.

R. Jacob Vogelstein, Francesco Tenore, Ralf Philipp, Miriam S. Adlerstein, David H. Goldberg, and Gert Cauwenberghs. Spike timing-dependent plasticity in the address domain. In S. Thrun S. Becker and K. Obermayer, editors, *Advances in Neural Information Processing Systems 15*, pages 1147–1154, Cambridge, MA, 2003. MIT Press.

J. von Neumann. First draft of a report on the edvac. Technical report, Moore School of Electrical Engeneering Library, University of Pennsylvania, 1945. Transscript in: M. D. Godfrey: Introduction to "The first draft report on the EDVAC" by John von Neumann. IEEE Annals of the History of Computing 15(4), 27–75 (1993).

Huai-Xing Wang, Richard C Gerkin, David W Nauen, and Guo-Qiang Bi. Coactivation and timing-dependent integration of synaptic potentiation and depression. *Nature neuroscience*, 8(2):187–193, 2005.

Reinhold P Weicker. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984.

Cornelius Weiller, FranÃğois Chollet, Karl J. Friston, Richard J. S. Wise, and Richard S. J. Frackowiak. Functional reorganization of the brain in recovery from striatocapsular infarction in man. *Annals of Neurology*, 31(5):463–472, 1992. ISSN 1531-8249. doi: 10.1002/ana.410310502. URL `http://dx.doi.org/10.1002/ana.410310502`.

Maurice V. Wilkes. The growth of interest in microprogramming: A literature survey. *ACM Computing Surveys (CSUR)*, 1(3):139–145, 1969.

ML505/ML506/M ML505/ML506/ML507 Evaluation Platform User Guide. Xilinx, Inc., November 2008. v3.1.

Virtex-4 FPGA Embedded Processor Block with PowerPC 405 Processor. Xilinx, Inc., April 2009a. URL `http://www.xilinx.com/support/documentation/ip_documentation/ppc405_virtex4.pdf`. version 2.01b.

Virtex-5 FPGA User Guide. Xilinx, Inc., 2009b. URL `http://www.xilinx.com`.

MicroBlaze Processor Reference Guide. Xilinx, Inc., January 2012a. URL `http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_4/mb_ref_guide.pdf`. version 13.4.

PlanAhead User Guide. Xilinx, Inc., January 2012b. v13.4.

Ji-Chuan Zhang, Pak-Ming Lau, and Guo-Qiang Bi. Gain in sensitivity and loss in temporal contrast of stdp by dopaminergic modulation at hippocampal synapses. *Proceedings of the National Academy of Sciences*, 106(31):13028–13033, 2009.

Xiaotong Zhuang and Santosh Pande. Power-efficient prefetching for embedded processors. *ACM Trans. Embed. Comput. Syst.*, 6(1), February 2007. ISSN 1539-9087. doi: 10.1145/1210268.1210271. URL `http://doi.acm.org/10.1145/1210268.1210271`.

Steve Zucker and Kari Karhi. System v application binary interface powerpc processor supplement, September 1995. Revision A.

Victor Zyuban and Peter Kogge. Optimization of high-performance superscalar architectures for energy efficiency. In *Proceedings of the 2000 international symposium on Low power electronics and design*, pages 84–89. ACM, 2000.

Danksagungen

Zum Ende dieser Arbeit möchte ich noch all jenen danken, die mich auf dem Weg hierher unterstützt haben:

Zunächst natürlich Herrn Prof. Meier, der diese Arbeit ermöglicht hat und immer eine Unterstützung war. Herrn Prof. Brüning, dafür, dass er sich bereit erklärt hat das Zweitgutachten zu übernehmen. Herrn Prof. Klingeler und Herrn Prof. Schwarz, als weiter Mitglieder meiner Prüfungskommission.

Besonderer Dank gilt Dr. Johannes Schemmel, für den Entwurf der Synapsen und die ursprüngliche Idee eines Plastizitätsprozessors. Sein unglaubliches Fachwissen wird nur noch durch seinen Optimismus übertroffen. Matthias Hock, für spannende Chip-Submissionen, einen sorgenfreien Laboraufbau, praktische Hilfe bei unzähligen Dingen, dem rigorosen Korrekturlesen, täglichen Flurgesprächen und nicht zuletzt dafür ein guter Freund zu sein. Andreas Hartel und Dr. Andreas Grübl für die Backend Arbeit und letzterem vor allem auch fürs Korrekturlesen. Auch den restlichen Visions sei gedankt, für die freundliche Atmosphäre und ihre Hilfsbereitschaft.

Dank gilt auch Tobias Nonnenmacher, der mit seiner Bachelorarbeit einen Beitrag zur Verifikation des Prozessors geleistet hat. Dr. Nicolas Frémaux für produktive Zusammenarbeit an einem spannenden Thema und einem angenehmen Aufenthalt in Lausanne (merci beaucoup!).

Ohne die Unterstützung und Rücksichtnahme meiner Eltern, die mir stets den Rücken frei gehalten und mich motiviert haben, wäre ich sicher nicht an den Punkt gekommen diese Arbeit schreiben zu können. Danke dafür!

Ganz besonderer Dank schließlich gilt meiner Ursula, die nicht nur meine Metamorphose zum Zombie erdulden musste, sondern neben moralischer Unterstützung auch tatkräftig Korrektur gelesen hat.