Faculty of Physics and Astronomy
University of Heidelberg

Diploma thesis
in Physics

submitted by
Simon Friedmann
born in Heidelberg

October 2009

Extending a Hardware Neural Network Beyond Chip Boundaries

This diploma thesis has been carried out by Simon Friedmann at the
Kirchhoff Institute for Physics
University of Heidelberg
under the supervision of
Prof. Dr. Karlheinz Meier

Abstract

Extending a Hardware Neural Network Beyond Chip Boundaries

This thesis presents the design, the implementation and the experimental testing of an FPGA-based networking system for a neuromorphic VLSI hardware device. An isochronous gigabit transport network is used for the exchange of digital spike events among multiple network modules. The high speedup factor of the utilized chip compared to biological timescales imposes challenging latency and throughput requirements on the digital logic. Event processing inside the chip necessitates sorting of spike events by time before transmission to the chip. These demands are met with a low-latency adaptive readout-rate sorting module embedded in the network logic. The design process was facilitated by simulations using the SystemC extension of the C++ programming language in combination with the VHDL hardware description language. Support for the implemented network was integrated into an existing software framework to allow experimental tests of functionality and performance. Delays of 142 ms and event rates of 340 Hz in biological dimensions were demonstrated in experiments. For the next revision of the used artificial neural network ASIC delays of 9.1 ms and rates of 5.3 kHz are expected. When each chip has neuronal connections to every other, the resources of the FPGA device allow for a maximum network size of 64 chips.

Ausweitung eines neuronalen Netzwerks in Hardware über Chip-Grenzen hinaus

Die vorliegende Arbeit präsentiert den Entwurf, die Verwirklichung und die experimentelle Überprüfung eines FPGA-basierten Systems zur Vernetzung eines neuromorphen Hardwaresystems in hochintegrierter Schaltungstechnik. Für den Austausch von Aktionspotentialen zwischen mehreren Netzwerkmodulen wird ein isochrones Gigabit-Transportnetzwerk verwendet. Der hohe Beschleunigungsfaktor des verwendeten Chips im Vergleich zu biologischen Zeitskalen stellt hohe Latenz- und Durchsatzanforderungen an die digitale Logik. Die Verarbeitung von Aktionspotentialen innerhalb des Chips erfordert deren zeitliche Sortierung vor der Übertragung zum Chip. Diesen Anforderungen wird mit einem Sortiermodul mit niedriger Latenz und adaptiver Ausleserate begegnet, das in der Vernetzungslogik eingebettet ist. Der Entwurfsvorgang wurde durch Simulationen unterstützt, die die SystemC-Erweiterung der Programmiersprache C++ in Kombination mit der Hardwarebeschreibungssprache VHDL verwenden. Das implementierte Netzwerk wurde in existierende Software integriert, um experimentelle Funktionalitäts- und Leistungstests durchzuführen. In Experimenten wurden Laufzeiten von 142 ms und Ereignisraten von 340 Hz in biologischen Einheiten nachgewiesen. Für die nächste Revision des verwendeten künstlichen neuronalen Netzwerk ASICs werden Laufzeiten von 9.1 ms und Raten von 5.3 kHz erwartet. Die Ressourcen des FPGAs erlauben eine maximale Netzwerkgröße von 64 Chips, wenn jeder Chip neuronale Verbindungen zu jedem anderen unterhält.

Contents

Chapter 1

Introduction

The last 60 years saw the rise of computer technology to a central part in our society. With the advent of computing machines came the idea, that intelligence, which previously was only attributed to humans or animals, is independent of brain tissue. In the late 1950s, in a phase of nearly limitless expectations, thinking machines were envisioned, which would equal humans in intellectual tasks.

> "It is not my aim to surprise or shock you – if indeed that were possible in an age of nuclear fission and prospective interplanetary travel. But the simplest way I can summarize the situation is to say that there are now in the world machines that think, that learn, and that create. Moreover, their ability to do these things is going to increase rapidly until in a visible future – the range of problems they can handle will be coextensive with the range to which the human mind has been applied."
>
> — *Herbert A. Simon, 1958* [SN58]

Although more than 50 years later computers are impressive and powerful devices, they are still not considered to be thinking. No machine has passed Turing's "imitation game", for which it must appear indistinguishable from a human to a human [Tur50] [loe09]. The classical paradigm of sequentially computing with discrete states – the Turing Model – is therefore perhaps not the ideal basis for such machines. Today, the field of artificial intelligence has left behind this early hype and settled to much less ambitious topics like pattern recognition and autonomous control, where it is quite successful.

In other fields of information technology, important topics concern the further increase of computational power, which is becoming more and more challenging. The doubling of transistors per processor every 18 months, usually referred to as Moore's Law [Moo65], is physically limited by minimal possible sizes of semiconductor structures. Using faster clock speeds leads to higher energy consumption. Modern processors have already surpassed the average hot plate when it comes to power densities [hHcF05].

Therefore, since a few years ago, the main processor manufacturers have switched to increase performance by integrating multiple processing units onto a single die. But it has proven to be difficult to write software which fully utilizes this parallelism. Already in the early days of computing it was found, that for the sequential processing paradigm the speedup gained through parallelization is limited by the amount of sequential code in the program [Amd67]. In practice most algorithms require a certain fraction of operations which can't be executed simultaneously. If this fraction is one percent, the algorithm can't be executed more than a hundred times faster than on a single unit, no matter how many processors are used.

In summary, as of today, computers have failed to become intelligent, making them faster is increasingly difficult and they are not ideally suited for parallelization. Perhaps one can get around

these problems by taking a fundamentally different approach to computing? The thinking devices that we know of are the higher developed brains, e.g. mammalian and especially human. Its outstanding properties are its ability to adapt, or learn, its low power consumption of only a few watt [LS03] [Mea90] and its inherent parallelism. Thereby it seems to be an ideal candidate to search for new computing paradigms.

There are even more incentives for research on the brain or neural networks in general. Another reason is curiosity. It has been man's desire ever since to understand the world around him and what lives in it. Researching the one tool which empowered him to do so, seems an obvious consequence. A better understanding of the principles of the brain will probably also lead to a better understanding of diseases related to the brain and might help to develop medicaments and therapies.

One approach in neuroscientific research is trying to use methods from electrical engineering to build integrated circuits behaving like biological neural systems [MM88][Mea90]. This allows for the construction of neuromorphic structures, which are significantly faster than biology [SBMO07][SFM08], making experiments possible that are not feasible with computer simulations. Super-realtime experiments enable statistics intensive analyses like parameter sweeps or long-term experiments over weeks and months of simulated time [Brü09]. Also from the technological point of view this approach is interesting. Microchips implementing these structures can have properties like low power consumption, inherent parallelism and fault tolerance [Mea90]. They can be used for large-scale networks on small space [SFM08].

Besides implementing realistic models of neurons and synapses, it is of importance to provide well-suited means for their interconnection. Few neurons on individual chips have only limited computational capabilities. It therefore makes sense to combine multiple chips, each implementing a small network itself, to a larger system.

This thesis presents the final piece for such an inter-chip network within the framework of an existing neuromorphic experimentation platform.

Overview of this Thesis

The goal of this work is to develop, implement and test a network technology to enable multi-chip experiments with an existing artificial neural network chip and an existing isochronous transport network.

Since realism is an overall goal for this system, some biological background is given in the next section. Thereafter, the FACETS research project is introduced, which is a multidisciplinary effort with a focus on finding new computing paradigms inspired by the brain. The technical environment for the system is outlined in Chapter 2. The presentation of new work starts with Chapter 3, which describes the working principles of the event network and its components. The following Chapter 4 discusses methodological aspects of the simulations used in the development process. Chapter 5 gives the details of the implemented design and supporting software and Chapter 6 presents experimental results. The scalability of the system is evaluated in Chapter 7. Thereafter, the final chapter discusses the results and gives perspectives for future improvements and developments of the system.

1.1 Neural Networks

Like the single chip itself, the inter-chip network should be designed to be usable for biologically relevant experiments. Therefore, a basic understanding of the matter is required. This section presents some aspects of biological neural networks, which are of importance for this work.

Neurons and synapses *Neurons* are considered as central components of information processing in biological neural networks, such as the human brain [LMBA06] [BBJ+05]. In a simplified picture, information flows in one direction along the neuron. Signals are received by the dendritic tree, "processed" by the cell body, the soma, and forwarded to other neurons by the axon. The axon is connected to dendrites of other neurons by *synapses*. There, signals are transported to the postsynaptic neuron by means of chemical neurotransmitters. However, along the neuron itself, signals travel as electrical potentials, that propagate along the cell's membrane. The brain contains large numbers of neurons, which are strongly interconnected. The cat visual cortex for example contains approximately 50000 neurons in 1 mm^3, which are connected to 6000 other cells [BC85].

Early models of neurons by McCulloch and Pitts [MP43] assumed, that neurons are in either an active or inactive state, depending on whether their summed inputs are above a certain threshold. More recent models reflect the spiking mechanism of neurons [Maa97]: The soma generates a stereotypical voltage curve, called action potential or *spike*, on its membrane, which travel down the axon and evokes a change in postsynaptic potential (PSP). When the potential on the membrane of the postsynaptic neuron exceeds a threshold, it produces an action potential itself.

Properties of Neural Networks

In these kinds of models, information is coded in the *timing* of spike events [Maa97]. This complements the coding in firing-rates with a wide field of possible time-based codes. Which codes are actually used in biology is not definitely known and subject of ongoing research [Sof95] [SKdRvSB98] [dZ00]. For an artificial neural network that uses continuous-time spiking neurons, biological timing constraints must be respected, when transporting spike events between chips.

Action potential propagation delays Delays in biological neural networks are introduced for example by the physical length of the axon. The velocity at which action potentials travel along it is determined by several characteristics of the cell and ranges between 0.1 m/s and 100 m/s [Deb04]. According to [SSR02], horizontal cortico-cortical connections[1] have delays of the order of 0.2 m/s, giving a delay of 50 ms for neurons 1 cm apart. Measurements of axons in the visual cortex of cats found length up to 12.55 cm [BDM05]. Synaptic delay is within 0.4 to 4.0 ms (mean at 0.75 ms) according to [KM65]. The typical rise time of an excitatory PSP of 3 to 10 ms may also delay propagation [SSR02].

These considerations show, that propagation delays (i.e. latency) in biological neural networks cannot be fixed to a single value. Instead they are on the scale of milliseconds to hundreds of milliseconds.

Firing rates In awake and attentive animals a strong synaptic bombardment of neurons in the neocortex region of the brain is observed [DRP03]. This results in so-called high-conductance states, in which the input resistance is reduced and the membrane voltage is less negative than in non high-conductance states. In this state, neurons typically fire irregularly at rates between 5 to 40 Hz, according to [STG01].

1.2 The FACETS Project

FACETS stands for Fast Analog Computing with Emergent Transient States. It is an interdisciplinary research collaboration of sixteen European groups, funded by the European Commission as part of the Information Society Technologies (IST) program [ea05]. The goal of the project is to find new computing paradigms inspired by brain activity, that are different from the well established Turing model [Tur37]. Within the FACETS project the realization of such newfound paradigms is to be prepared by laying down a theoretical and experimental foundation. More

[1] Connections within the neocortex of the human brain.

information on the goals, ongoing activities and published results can be found on the project website [FAC09].

The interdisciplinary approach combines scientists from several fields. Biologists gather data in single cell and network experiments on neural tissue in-vivo and in-vitro. Modelers use this data for computer simulations and to build a theoretical understanding. Based upon this work, neuromorphic hardware devices are designed following two branches. One tries to mimic biology as close as possible in small networks [ZBS+06] with the possibility of hybrid living-artificial neural networks [BRG+07]. The other exploits the intrinsic properties of silicon, that lead to timescales faster than biology, when decreasing element size. A small size of artificial neurons and synapses allows for a high number of elements per chip and thus enables large-scale networks.

For this thesis the *Spikey* chip, following the second branch, plays a major role and is introduced in Chapter 2, along with its system environment. The chip and its environment form the FACETS stage-1 system. Single chips are mounted on support modules, which among other things provide network connectivity between the modules. In contrast, in the FACETS stage-2 system [SFM08], which is currently under development, the neural network chips are not separated into individual dies, but are directly interconnected on the production wafer. Thereby, a much higher connection density can be achieved than in stage-1. Using these connections, events can be exchanged by means of an asynchronous digital network protocol that was specifically designed for this purpose. For event communication to other wafers or to a host computer a second digital network is provided by external chips. The stage-2 system also features an improved neuron implementation, which allows a flexible configuration of neuron count and the number of presynaptic connections per neuron. Neurons with up to 14366 incoming connections are possible in the system [Mil09].

Chapter 2

Environment of the Experiment

This chapter gives an overview of the existing hardware and software components. It introduces the FACETS stage-1 system, which is a hardware platform that can be used for the exploration of artificial neural networks. These are the *Spikey* analog neural network ASIC[1] and the isochronous multi-class gigabit network architecture (MCGN).

2.1 The FACETS Stage-1 System

The Stage-1 system is developed in context of the FACETS project as a multi-chip neural network platform. A schematic overview can be seen in Figure 2.1, technical details are listed in Table 2.1. The *Spikey* analog neural network chip plays a central role in the system. One of the chips at a time is situated on a *Nathan network module*, where it is connected to an FPGA[2]. Up to sixteen of these network modules can be plugged into a single *Backplane* to form a larger network. The *Backplane* is connected to a controlling PC for configuration access.

2.1.1 Nathan Network Module

The *Nathan* network module serves two purposes: It supplies infrastructure for the operation of an artificial neural network ASIC (ANNA) and it enables several of these to communicate among each other within a large scale network. To accomplish this in a flexible way, the network module features a Xilinx Virtex-2 Pro FPGA [Xil07], directly connected to the neural network chip. This FPGA has eight serial multi gigabit tranceivers (MGT), four of which are connected to the backplane adapter. The four remaining can be used for additional interconnections between the network modules, e.g. to interconnect multiple backplanes. The work described in this thesis utilizes the transport network established by these multi-gigabit tranceivers.

Other parts on the board are a DDR[3]-SDRAM[4] module of typically 256 MByte, two 512 KByte SRAM[5] chips, a digital to analog converter to generate reference voltages and currents for the ANNA and a temperature sensor for monitoring.

An extensive description can be found in [Grü03].

FPGA device details An FPGA is a programmable device used for the implementation of digital logic circuits. In contrast to a processor, which executes sequences of instructions, the basic elements of an FPGA are configured to form a circuit described on the gate level. In the given device the basic logic elements are organized in *configurable logic blocks (CLB)*. They consist of four *slices*, which contain two function generators each. Each function generator is programmed

[1] Application Specific Integrated Circuit
[2] Field Programmable Gate Array
[3] Double Data Rate
[4] Synchronous Dynamic Random Access Memory
[5] Static Random Access Memory

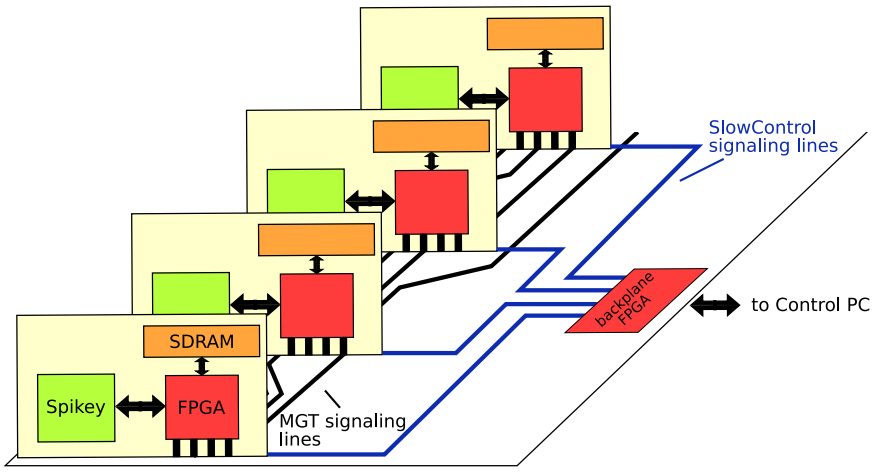

Figure 2.1: Schematic view of the FACETS stage-1 system with one backplane. Only four nathan network modules and not all signaling lines are shown for clarity.

to implement a boolean function with 4 bit wide input. Its output can either be asynchronously passed out of the slice or via a Flip-Flop storage element. Function generators are implemented as SRAM lookup tables (LUT4) that can also be used as random access memory or shift registers.

Using the Flip-Flop, circuits synchronous to a clock signal can be build. The FPGA has signaling lines dedicated to clock networks to allow for a low skew distribution of the clock signal. These networks are driven by *global clock buffers (BUFG)*. *Digital clock managers (DCM)* provide various features for the conditioning of clock signals, such as frequency modification and phase shifting.

The CLB elements are arranged in a regular structure with slices forming columns and rows. Inserted into the structure are columns with specialized blocks. These blocks contain the MGT logic interfaces and dedicated memory resources called *BlockRAM*. The latter are memory cells with a capacity of 18 kBit and two configurable access ports. The *PowerPC* block available in the FPGA is not used for this thesis. A detailed description of the device is given in [Xil07].

2.1.2 Backplane

The *Backplane* provides power, connectivity and clock signals for the network modules. It is designed to fit into a standard 19" rack, where it takes a height of three rack units[6]. It contains two jumper-selectable alternative clock sources with frequencies of 100 or 156.25 MHz, which can be used as clock inputs to the *Nathan* FPGAs. Each of the network module slots is individually connected to a central FPGA on the *Backplane*. These connections are used to load bitstream configuration files onto the *Nathan* FPGAs and for configuration and control access during experiments. This is done by the *SlowControl*, which is a moderate speed and bandwidth network, connecting the programmable logic with a standard, off-the-shelf PC.

The signal lines of the MGT links are routed between the module slots in a fixed topology shown in Figure 2.2. Each node represents one *Nathan* board and is connected to four nearest neighbors via one MGT link per neighbor. Since the network modules are not point-to-point interconnected with each other, nodes must be able to route traffic through which is not directed at them. Since the *Backplane* has sixteen slots, the maximum distance between two nodes is four hops.

A detailed description of a previous version can be found in [Grü03]. For the work presented here a revised board has been used. The *Backplane* description above refers to this newer version.

[6] 1 rack unit = 44.45 mm

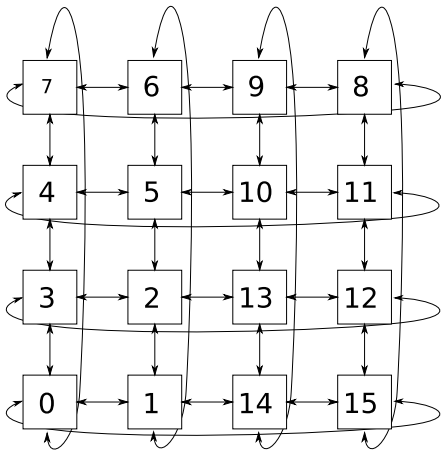

Figure 2.2: The multi-gigabit tranceiver links of the *Nathan* network modules are interconnected on the *Backplane* in a toroidal topology. Every node has four nearest neighbors and the maximum distance is four hops. The numbers shown are used to address the modules.

2.1.3 SlowControl Access

Setup and control of neural experiments is done using the *SlowControl* network. This includes among other things the reading and writing to the SDRAM on *Nathan*, the configuration of the ANNA and programming of routing information for inter-chip communication. The controlling PC can communicate with client logic modules implemented on the FPGA by reading and writing 32 bit data words to 32 bit wide addresses. Sixteen such modules can be individually addressed in every device. For example on *Nathan* there may be a module for RAM access and one to communicate with the ANNA, among others.

The controlling PC uses the custom-made *Darkwing* PCI[7] card as access device to the hardware. It is connected to the *Backplane* by a SCSI[8] cable. On the original *Backplane*, the network modules are physically connected in a ring topology, which is reflected by a TokenRing (IEEE 802.5) like architecture of the *SlowControl*. With the new revision the modules are point-to-point connected to the *Backplane* FPGA as a central hub. However, this thesis uses a FPGA programmed for software compatibility, that emulates the previous topology.

There is ongoing work to replace the *Darkwing* card with a Gigabit Ethernet link to the *Backplane*. This is to improve data rates and to communicate with the *Nathan* modules in parallel. More on this and a more detailed description of the *SlowControl* can be found in [Dre08].

2.2 The Multi-Class Gigabit Network Architecture

The multi-class gigabit network (MCGN) architecture was designed in the *Electronic Vision(s)* group with the purpose to interconnect the *Nathan* network modules. It is a multi-class network that combines transport of priority traffic with quality of service (QoS) requirements and the exchange of packet based best-effort traffic. A detailed description can be found in [Phi08a] a outline is given in [PSM09]. The following paragraphs define some important concepts.

Quality of service QoS is a general computer networking concept and is described for example in [PD03]. A network with QoS can give guarantees for some or all of its performance characteristics, e.g. constant throughput, maximum error rates or maximum latency. Typical examples are multimedia streaming applications, where a constant data rate is required or telephony, where

[7] Peripheral Component Interconnect
[8] Small Computer System Interface

***Nathan* FPGA: Xilinx Virtex-II Pro (XC2VP7-6 FF 672)**	
Slices	4928
Dedicated RAM blocks (18 KByte)	44
Multi-gigabit tranceivers	8
max. MGT data rate	3.125 GBit/s
DDR-SDRAM	
Size	256 or 512 MByte (max. 2 GByte)
Data bus width	64 bit
Max. clock frequency	133 MHz
SlowControl	
Clock frequency	40 to 100 MHz
Logic modules per FPGA	16
Address width	32 bit
Data width	32 bit

Table 2.1: Device types and performance characteristics of the FACETS stage-1 system.

latency and jitter[9] must be bounded. In packet switched networks, this can be implemented by allowing routers to prioritize individual packets based on their traffic class. One example is the Differentiated Services standard for the Internet Protocol [NBBB98] [BBC+98]. Another method is to preallocate network resources for a connection, as the Integrated Services architecture does [BCS94].

Priority traffic Quality of service demands for artificial neural networks are directly derived from biological properties of their ideal. The relevant characteristics are described in 1.1. The main QoS requirement is fixed delay with low jitter, since timing of spike events is assumed to be relevant for information processing. The delay must also be in a biologically realistic time domain with respect to the ANNA[10]. MCGN provides this QoS by using resource reservation and time division multiplexing (TDM) for priority data.

Best-effort traffic Resources not reserved for priority data can be used for packet based traffic without QoS requirements. Such traffic can comprise the distribution of stimulus data for the neural network among the SDRAM modules of several *Nathan* boards.

The work presented in this thesis only deals with the handling of priority traffic. Therefore the further discussion focuses on this traffic class. The interested reader may refer to [Phi08a] for more information on the integration of both traffic classes.

Besides guaranteeing QoS it is necessary to allow a huge number of inter-neuron connections. MCGN provides scalability by focusing on inter-chip connections, over which several neuronal links are established. The network is implemented within the FPGAs on *Nathan* and uses the MGT links. The next section gives a short overview of the functional operation of MCGN. Thereafter the interface to user logic and a service for system-wide synchronization are described.

2.2.1 Principles of Operation

Time division multiplexing Several chip-to-chip connections can share a single MGT link. This is done by dividing the time into equally sized slots. During one slot a fixed number of bits is transmitted. A fixed number of slots and a trailing gap form a frame, which is periodically repeated. Each slot contains data of a single connection only. For example to transport four connections over a single link, a frame consisting of four slots can be used, each being assigned to

[9] Jitter is the variation in end-to-end delay over a network connection
[10] The *Spikey* chip works on an accelerated timescale

a different connection. During the gap no user data is transmitted, instead lower network layers may insert data of their own.

Resource reservation The slot to connection assignment is done before the experiment i.e. resources are preallocated for the whole network and do not change during the run. The assignment is done frame-wise, so the allocation pattern repeats periodically with every frame.

In case only one slot of a larger frame is allocated for a single connection, data has to be buffered by the user logic until the time division reserved for this connection is reached. The waiting time ranges between zero slots and the frame length. This waiting time is the only source of jitter for priority traffic on the part of the network.

Synchronization For the operation of MCGN a global synchronization of all network nodes is required. This is necessary to ensure that data arriving over multiple links at one node belongs to the same slot within each frame. This way, a connection routed through a node can simply be forwarded by connecting the input to the output during the correct time-slot.

To detect synchronization or the loss of it, a special bit-sequence - the sync character - is inserted in each inter-frame gap. The network is synchronized, when for every node, all incoming sync characters arrive at the correct local time. To achieve this condition, the controlling PC adjusts the local time counter in every node and additionally programs small delay buffers at the node's outputs during the network setup phase. The network stays synchronized, because all network modules are clocked from the same source on the *Backplane*.

Besides network operation, the synchronization provides a global time base with the resolution of a single clock period for the whole system.

2.2.2 Interconnecting User Logic and Network

The Switch

MCGN implements a switch within every network node. This switch has two types of ports: MGT-ports interface the MGT links to the neighboring nodes and user-ports interface user logic inside the FPGA. Besides directing traffic from a user-port to one or more MGT-ports, the switch acts as a router within the network, forwarding traffic from MGT-port to MGT-port. Since resource reservation determines which outgoing port of the switch reads which incoming port for every time-slot, the scheduling of the switch is done by a look-up of a static routing table. A diagram of the switch can be seen in Figure 2.3.

In the given implementation the switch has a data path width of 16 bit and two cycle slots, which are 32 bit wide.

Interface to User Logic

The switch presents timing information and handshake signals for every data direction of the user-ports to the user logic.

Handshake To exchange data the respective destination (for example the switch logic) asserts the `accept` signal to indicate that it is ready to receive data. In the next clock cycle the source (for example the user logic) writes to `data` and asserts `valid`. Multiple cycles have to be used, in case the slot size is larger than 16 bit. Because the switch features no internal buffers, it can not hold back data, when the user logic is not ready. `accept` is therefore not evaluated by the switch.

Timing The current position within the time-frame is presented to the user in multiple ways: The `enable` signal is low during the inter-frame gap and high during the data period. The current slot number is denoted by `slot` (7 bit). On the last clock cycle of the data period `last` is asserted and during transmission of the synchronization character `sync` is set. For multi-cycle

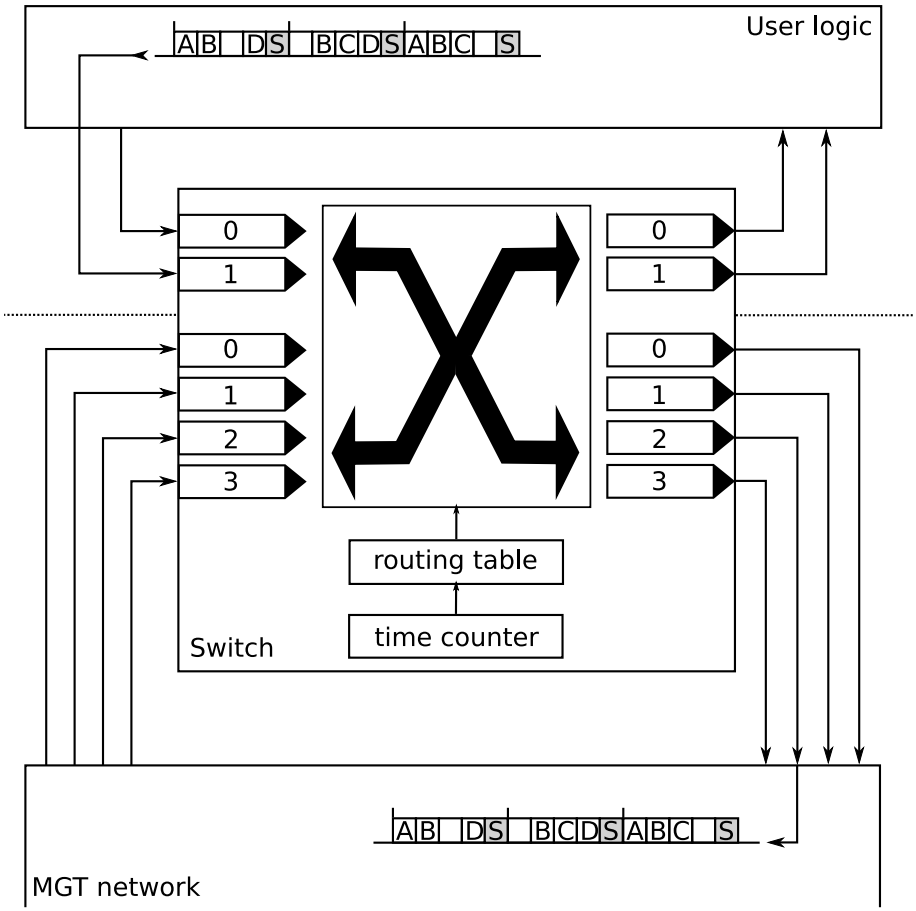

Figure 2.3: The diagram shows the MCGN switch as a link between user logic and network. It is drawn here in an example configuration with two user- and four MGT-ports. A data stream with a frame length of four passes from user-port 1 through the switch to MGT-port 2.

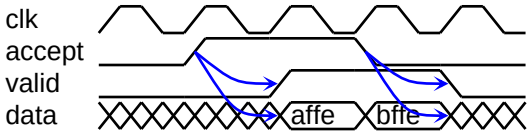

Figure 2.4: Timing diagram of the handshake at the user-port, with transmission of a single 2 cycle slot.

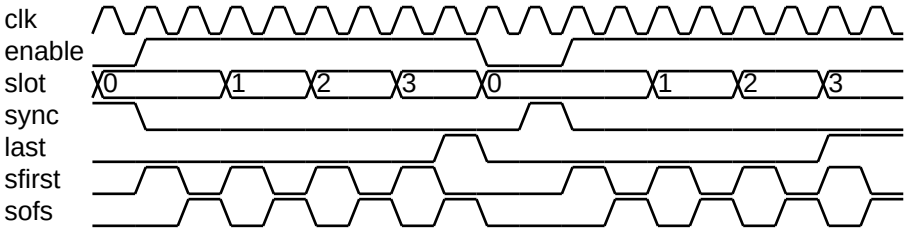

Figure 2.5: Diagram of the timing information supplied by the switch for a frame length of four and 32 bit wide slots.

slots `sfirst` indicates the first cycle within the slot and `sofs` the offset within the slot. In the given implementation `sofs` is only 1 bit wide and slots are 2 cycles or 32 bit long respectively.

The implementation allows to adjust how far in advance the timing information of the transmit side is given. This is useful for pipelined applications, e.g. to read from a RAM address depending on the slot number.

2.2.3 Global Synchronous Signal

Distribution of the Signal

In the synchronization process mentioned above, MCGN is synchronized with the precision of a single clock cycle. During the process, the transmission delays of all links become known. In the synchronized system it is possible to provide a global synchronization service to the user logic, using the obtained delays. The service allows a single network module to raise a global synchronous signal (GSS), that will be asserted on all modules at the same clock cycle.

To accomplish this the raising module initializes a counter to a sufficient large number and distributes this value, decremented by the respective transmission delays, to its neighboring nodes. These upon reception initialize their counters themselves and distribute the signal to their neighbors excluding the originally sending one. The counters on all nodes are also decremented during each clock cycle. This way the value decrements in time and space as it propagates through the network. All counters reach zero in the same clock cycle, leading to the simultaneous assertion of the GSS event on all nodes.

Interface to User Logic

The MCGN architecture allows for multiple GSS IDs to signal different global events. For each of these a set of interface signals is defined:

Raising Before GSS events can be raised or received `enable` must be asserted. This is important, as in an unsynchronized network spurious events may be detected. After that, `raise` may be high for a single clock cycle to start distribution of the GSS.

Receiving When `enable` is high, the occurrence of a GSS is indicated by `event`. The signal will stay high until `ack` is set.

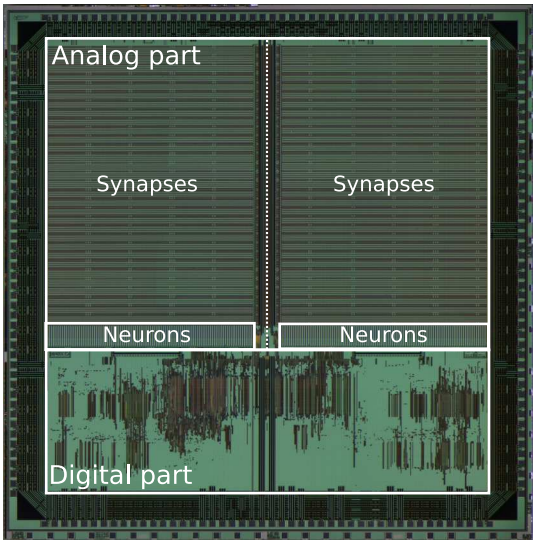

Figure 2.6: Microphotograph of the *Spikey* chip. The upper regular part contains synapses, synapse drivers and neurons. The synapse drivers are situated in between the two synapse blocks. The digital part on the lower half of the picture can be identified by its irregular structure. Adapted from [Grü07] with permission.

2.3 The Spikey Analog Neural Network Chip

Neurons and synapses in the stage-1 system are implemented as analog circuits on the *Spikey* ASIC. It features 384 neurons with 256 synapses each, totaling in 98304 synapses. The chip is fabricated in a standard 180 nm CMOS[11] process and has a die size of $5 \times 5\,mm$. It operates at continuous time with an acceleration factor of about 10^5 compared to biological real-time. The implemented neuron is modeled on the leaky integrate and fire model [GK02] and spike timing dependent plasticity (STDP) mechanisms are available in every synapse. Short term plasticity features allows to configure individual synapse drivers as facilitating or depressing, increasing or decreasing synaptic weights depending on firing rates. A microphotograph is shown in Figure 2.6.

Spikey was developed in the *Electronic Vision(s)* group and is available in a third revision, with a fourth version currently in production. A detailed description of the chip is given in [Grü07]. The implemented neuron and synapse models can be calibrated to match biological characteristics, enabling the chip to be used as a neuroscientific tool [Brü09]. It is accessible from a simulation device independent modeling language and neuroscientific experiments have been performed on it [Kap08] [Bil08].

This section will give a very basic overview and highlight the aspects relevant to the following chapters. First of all, structure and function of *Spikey* are outlined, then the controller implemented on the *Nathan* FPGA is described and finally, the focus will be on support software and experiment workflow.

2.3.1 Structural Overview

Digital and analog part *Spikey* is a mixed-signal chip. It contains an analog part implementing neurons and synapses, which takes up the larger part of the area. The digital part has a supporting role. It provides configuration access to parameters of the analog part, contains a reference time counter and communicates with the outside world.

The two parts differ in a fundamental way: The analog computation is done in true continuous time. The implemented circuit mimics the neuron model's characteristics for membrane voltages

[11] Complementary Metal Oxide Semiconductor

and currents. The digital part uses quantized time, which is determined by a clock signal. A conversion between the two time domains is done by the time to digital (TDC) and digital to time converters (DTC). They convert action potentials of the artificial neurons into digital events and vice versa.

Network blocks As can be seen in Figure 2.6, the analog part is split into two identical network blocks. Each of them contains 192 neurons with 256 synapses. The latter are arranged in a matrix, connecting synapse drivers on the rows with neurons on the columns.

Frequencies Multiple clocks at different frequencies are used on the chip. The external clock input of the chip is driven by the *Nathan* FPGA and reconstructed by a phase locked loop (PLL). So *Spikey* and *Nathan* are synchronous to each other. On *Spikey* there is a fast and a slow clock, with the latter having half the frequency of the primer. The fast clock is used to increment the reference time counter, which is used to digitize events. The FPGA's main clock has the same frequency as the slower clock. The chip is designed to work at frequencies of up to 400 MHz for the fast clock.

Neuronal events Inside the analog part, action potentials are represented as voltage spikes within a continuous time domain. For transportation off chip, these events are digitized into a discrete time domain by assigning them an 8 bit *time stamp* and a 4 bit *time bin*. The time stamp is derived from a reference time counter. The time bin is calculated by subdividing one fast clock cycle into sixteen subdivisions. A 9 bit address field completes the neuronal event and identifies the synapse driver for incoming events and the neuron number for outgoing ones.

Event buffers Events going into and out of the network block are stored in FIFO[12] buffers inside the digital part. A DTC is associated with each input buffer, which is in turn connected to a block of 64 synapse drivers. Every block has two buffer - DTC pairs, leading to sixteen input buffers and DTCs in total. Two pairs are used per block, because event time is referenced to the fast clock, while the event input buffers are inside the slow clock domain. One buffer per DTC is therefore associated with the rising and one with the falling edge of the slow clock. Thus, for a given time stamp at most eight events can be simultaneously injected into the synapse arrays. Because of the split in two network blocks, each neuron can therefore receive four events at maximum within one cycle.

Outgoing events are digitized by the TDCs and stored in six output FIFO buffers with the fast clock. One buffer is connected to 64 neurons and can receive an event from one of them in every cycle.

Interface to the outside world All communication is done over two 8 bit links, which operate at frequencies of up to 400 MHz with double data rate. Their physical characteristics are based on the HyperTransport specification [Hyp06]. They give a maximum rate of 1.6 GBytes/s. To the FPGA logic this appears as one 64 bit word within one slow clock cycle.

2.3.2 Functional Description

Synchronization Since the clock signals on *Spikey* are derived from the FPGA clock, time counters on both are synchronous by design. But because the physical distribution of the signal imposes a propagation delay, there is an uncertainty of at maximum one clock period between the two clock edges. This can lead to one-off mismatches between FPGA and *Spikey* time counters, requiring a synchronization mechanism. This mechanism is implemented within the *Spikey* controller logic on the FPGA.

A correct synchronization between FPGA and *Spikey* is important, because the time counter on *Nathan* is used to augment the 8 bit time stamps of events coming from the chip. For multi-chip

[12] First In First Out

Packet slot	Neuron numbers
0	0 - 63
	256 - 319
1	64 - 127
	320 - 383
2	128 - 191
	384 - 447

Table 2.2: The table shows the fixed assignment between neuron addresses and packet slots.

experiments this mechanism works in conjunction with MCGN's network wide synchronization to realize a common time on all neural network chips and FPGAs.

Event Generation and Digitization Digital events are transmitted to the chip over the HyperTransport link and demultiplexed to an event input buffer. When the buffer is not empty, the first element is removed from it and its time stamp is compared to the current system time. On match, the event is forwarded to a DTC and the addressed synapse driver to generate a spike.

When the neuron circuit generates an action potential, the logic of TDC and event output buffer produces a digital event with the current system time and neuron address and stores it in the output buffer. If multiple events fire within the same clock cycle, a priority encoder delays events with higher neuron number prior to digitization in the TDC. They are then sampled within the next clock cycle.

Event Loopback Mode As a debugging and verification feature the chip contains an event loopback module, that - when enabled - replaces the functionality of the analog part. Events coming from the input buffers are pipelined through this module and then fed into the output buffer, as if they were coming from the network blocks. In the process their time stamp is increased by 10.

2.3.3 Communicating with Programmable Logic

The *Nathan* FPGA is connected to *Spikey* by means of the HyperTransport link. This interface provides transmission and reception of one 64 bit data packet per FPGA clock cycle. It is used for all communication with *Spikey*, which includes configuration, stimulation and monitoring of the implemented neural network.

Event packing A single digital event is 21 bit wide. Three of these can be packed into one 64 bit data packet using three separate packet slots. As additional bits are required for protocol data and valid bits, the highest nibble[13] of the time stamps is stored only once inside the packet. This of course reduces the number of possibilities to combine events and has to be taken into account by the sending logic.

For events coming from *Spikey* the datapath layout restricts, in which packet slots events can appear. Each slot is associated with two output buffers, which hold events of neurons connected to their TDC only. The mapping of neuron addresses to packet slots is listed in Table 2.2.

Interface logic The existing controller module in the FPGA logic (cf. Figure 2.7) consists of two parts: `spikey_control` implements the low-level access to *Spikey*. `spikey_sei` is a data source and sink for `spikey_control` reading from and writing to the *PlaybackMemory* within the local SDRAM.

In the *PlaybackMemory* a sequence of commands can be stored, which is consecutively sent to the chip. Special delay commands can be used to control the timing. Stimulus events are

[13] Two 4 bit nibbles form an 8 bit byte

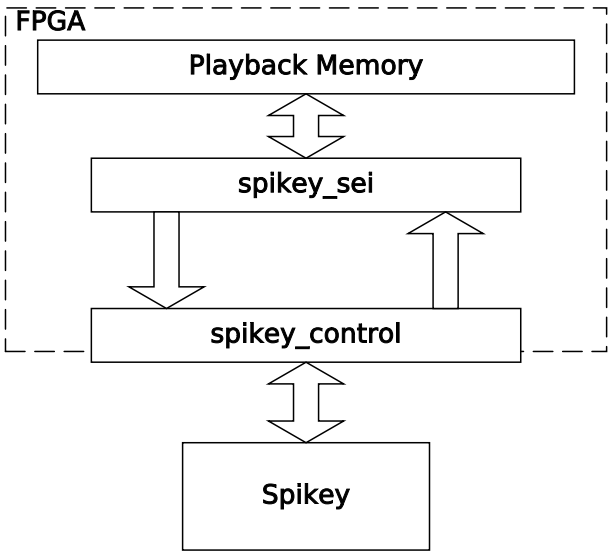

Figure 2.7: Schematic overview of the interface logic for *Spikey* within the *Nathan* FPGA. `spikey_control` is the low-level interface to the chip. `spikey_sei` transfers events and control data from the *PlaybackMemory* to *Spikey* and in the opposite direction.

coded as event commands, preceding the actual event data and providing a delay time to wait afterwards, and event packets. The latter hold three events each and use the same encoding as the link interface.

2.3.4 Software and Experiment Workflow

A single chip experiment on the stage-1 system is performed by going through the following steps:

- Transmission of stimulus event data and configuration commands to the *PlaybackMemory*
- Synchronization between *Nathan* FPGA and *Spikey*
- Playback of the *PlaybackMemory*
- Transmission of result event data to the control PC

Stimulus events and configuration commands are assembled into a sequence of instructions, that are written to the *PlaybackMemory* and executed similarly to a program in a CPU[14]. This access uses the *SlowControl* network after assembling is done in software on the attached PC. This software is either the `createtb` testbench program or the low level part of a PyNN script (discussed below).

When all data has been transmitted and stored in the SDRAM, `spikey_sei` starts to read the just transmitted program back into FIFOs implemented in BlockRAM resources of the FPGA. A predefined time elapses, before the experiment is started, to give the FIFOs time to fill.

Then `spikey_sei` begins the processing of the command sequence, which must contain a special command to synchronize time counters. As soon as the FIFOs run empty, the logic ends the experiment and disables event processing. When having sustained high event rates programmed into the *PlaybackMemory*, this might happen before the end of the sequence is reached, because the SDRAM can not provide the high data rates of the *Spikey* interface continuously. Therefor a large amount of BlockRAM cells is allocated for the read and write FIFO buffers.

After the experiment has ended, the PC reads result data out of the SDRAM via the *SlowControl*.

[14] Central Processing Unit

The testbench software framework The `createtb` software was developed to verify the *Spikey* ASIC and its FPGA controller in simulation and experiments with real hardware. It provides the low level framework to interface the chip and perform automated tests. On invocation it is given one or more *test modes* to execute. These test or validate certain aspects of the chip. A set of different communication modes is provided of which two are of interest: In the *file* mode read and write operations, that would be performed over the *SlowControl* are written to a file, which can be parsed by the simulation testbench. Non-interactive testmodes - that is testmodes that do not require the results of their read operations during runtime - can thus be used in a completely simulated environment. The *msgqueue* communication mode uses the real *SlowControl* to talk to the actual hardware. It is used to perform experiments with *Spikey*.

Software for neuroscientific experiments For neuroscientific experiments, i.e. experiments which use a description of the artificial neural network in biological terms, another software system is used. Using the same hardware access code as `createtb`, the stage-1 system is made available as backend for the PyNN modeling framework. PyNN allows a description of neural networks using the Python programming language. The description is independent of the actually used simulator. In case of the hardware system as backend, the biological network parameters are transformed to the configuration parameters of the system. The transformation uses individual calibration files to account for variations between chips.

Chapter 3

Concepts for Multi Chip Operation

The previous chapter summarizes the state of the FACETS stage-1 system, as it is used for single-chip operation. A revised version of the *Spikey* ASIC can be used for experiments and a low latency isochronous interconnect is available as basis for multi-chip operation.

This chapter outlines concepts, which link the *Spikey* controller and MCGN to support the formation of large scale artificial neural networks across chip boundaries. The coming section starts with an analysis of the existing system and derives constraints for the design. After these abstract considerations, the chapter continues with an explanation of the operation principles and an introduction of the basic logic blocks.

3.1 Initial Considerations

A system overview is presented in Figure 3.1. The event network is formed by logic on top of the *Spikey* controller and the MCGN implementation. It uses the priority traffic class of MCGN to communicate with other network modules. To this end it routes events inside the FPGA: Coming from *Spikey*, they are directed to a user port and time slot in such a way, that they are delivered to the intended destination chip. The *source event* has to be translated into a *target event* by mapping neuron addresses onto synapse driver addresses and adding a delay to the time stamp.

3.1.1 Spikey Interface Constraints

Section 2.3.2 outlines how the *Spikey* analog part converts digital into analog events. After transmission to the chip, the digital part processes events in order of arrival following the first in first out principle. When their time stamp matches the system time, a buffer forwards events to the DTC for conversion. It is therefore necessary, that the FPGA ensures well ordered event streams to the chip.

The problem To illustrate this problem by an example, consider the processing steps on *Spikey* (see Section 2.3 or [Grü07]). For a sequence of events within the event input buffers on the chip, the first event arriving is immediately taken out of the FIFO and stored in a register stage. A logic process compares its time stamp to the current system time in every clock cycle. When the timers have counted up to match the event time stamp, the next event – if available – is taken from the FIFO. Because time stamps are 8 bit wide a match will occur at the latest, after the timer has been incremented by 256.

Assuming two sequential events for the same point of time and thus with identical time stamps that follow each other in the FIFO, the DTC will convert the first one as expected. When the second one passes to the register stage, the time counter has already been incremented, blocking

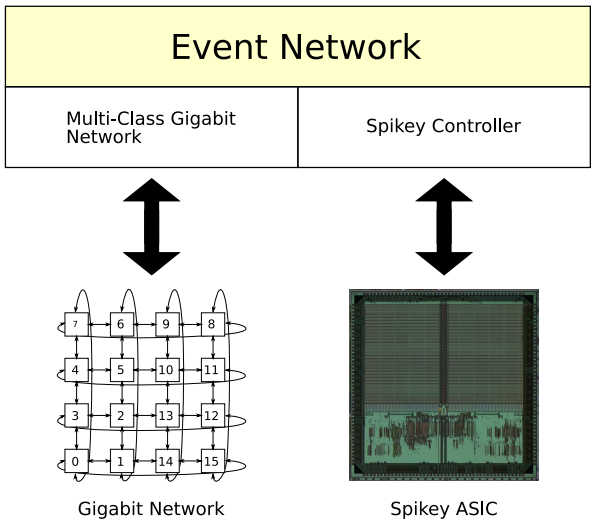

Figure 3.1: Overview of the system. The event network logic is on top of the *Spikey* controller and the MCGN implementation. Thereby, it links the *Spikey* chip to the gigabit network.

the FIFO until the counter matches again on the next wraparound. Of course, during this time other events have to wait in the buffer and so also future event processing is disturbed.

This problem arises not only for events with identical time stamps, but also for those, where the second event has an earlier time stamp than the first one. To prevent it, it must be assured by external facilities, that events going to the same event input buffer are in strict monotonic order before they are transmitted to *Spikey*. In other words, event streams to certain subranges of synapse driver addresses must be sorted on the FPGA.

Incidence

In a multi chip network there are two types of sources for events: Other network chips and the *PlaybackMemory*. A strict monotonic ordering of streams within the *PlaybackMemory*, going to a single input buffer, is assured by the existing software. As described in Section 2.3.2 TDCs on *Spikey* digitize events in a strict monotonic order, too. It seems necessary, to name some situations, where the order is violated and blocking of the buffers can occur.

Simultaneous source events For a setup, in which two source neurons in different blocks are connected to one destination synapse driver, with a certain probability both sources will spike within the same clock period. Without further prevention those events will reach the destination chip with identical time stamps, resulting in a blocked buffer.

Buffer race conditions For the same setup consider the following situation at the output event buffers of the source chip: When activity in both neuron blocks differs, the numbers of events in those FIFOs may be diverging. But because the propagation time through a FIFO is directly proportional to the number of contained elements, this can lead to events generated at a later point in time overtaking events, passing through the other buffer. Again, when streams passing through these two buffers share a common destination, the arriving stream may not be in correct order.

Network jitter Extending this setup to a network of three chips, with two chips containing one source neuron each and the third being the destination, imposes an additional uncertainty on the transportation time. Events have to wait before transmission, until a valid data slot is available,

which leads to delays, that can be considered random. Between connections routed via different numbers of intermediate hops, the propagation delays also differ by fixed offsets. So transmission times from both sources are subject to variations and events with earlier time stamps may overtake later ones. When streams are merged together at the destination, the temporal order is lost.

3.1.2 Spikey and Network Data Rates

This section discusses the data rates of *Spikey* and the gigabit network and analyzes, what consequences result for the FPGA logic between them.

It was stated in Section 2.2.2, that the network in the given implementation has a 16 bit wide datapath. That means in every FPGA clock cycle 16 bit of data can be passed to the switch or received from it. It was also mentioned, that events processed by *Spikey* are 21 bit wide. 12 bit for time stamp and time bin and 9 bit for the address. So at least two clock cycles are required to pass a single event. The interface to the *Spikey* chip on the other hand can deliver up to three events in a single FPGA clock cycle. But events on the chip are digitized and generated twice that fast. So in one clock cycle the system time counter, which is the reference for the time stamp, increases by two.

Example For an FPGA frequency of 100 MHz the system time counter is referenced to a 200 MHz clock. At maximum one TDC can digitize 200×10^6 events per second and insert them into its event output buffer. From this FIFO 100×10^6 events can be read per second and send off chip. The FPGA in turn can send 50×10^6 events per second over a single MGT link.

These numbers are only peak event rates. The event packing algorithm used on the *Spikey* interface introduces an additional overhead, as the implemented logic inserts empty packets, whenever the common high nibble is increased. Overhead is also present within the MCGN: At the end of every frame is a gap, where no data is transmitted.

Consequences for the design These throughput estimates show, that the total interface rate of *Spikey* is much higher than the one of the network. Six bundled MGTs would be required, to provide the equivalent bandwidth. The design is should support multiple MGT links in parallel, but limitations on logic size imposed by the FPGA need to be kept in mind. To support the bandwidth of six MGTs, the MCGN switch would have to be implemented with six user ports, totaling in twelve ports. For each one a buffer structure is necessary to store events, until a valid slot is available.

The fixed topology of the *Backplane* prevents an arbitrary allocation of the MGT bandwidth between random nodes. Four of the eight MGTs are hard wired. Combining them to interconnect two particular nodes may require taking detours over other nodes. This may or may not be acceptable, depending on the latency requirements. The remaining four MGTs can be freely interconnected by cables.

Despite of the high *Spikey* interface rate, spike trains can have even higher rates, since the system time is incremented twice as fast.

Concluding from these observations, the FPGA logic should be designed to allow maximum rates for a short time (bursts). Sustained rates should fully utilize the available bandwidth. This applies to both the *Spikey* interface and the gigabit network in transmit and receive direction.

3.1.3 Digital Event Timing

From the experimenters point of view, it is desirable to have selectable delays on artificial neuronal connections to model for example different axonal lengths (cf. Section 1.1). This means, that a spike generated at one neuron, should be presented to the postsynaptic neuron after a configurable time. With *Spikey* this is not possible for connections inside a single chip, since there is no way to delay the spike deliberately. Therefore all those connections have the same delay.

Now with event processing outside of *Spikey*, this limitation can be overcome. The transportation over the network introduces a natural latency, which is subject to jitter and may also

vary depending on the path, an event takes. The design should make it possible to program an artificial delay for every neuronal connection with the network latency as lower limit. Jitter should be compensated for to provide deterministic timing. The dynamic range of this programmability should be large enough to cover the range of short and long paths between network nodes. At least connections with up to four hops should be possible simultaneously, to allow fully interconnected networks on the *Backplane*.

Since the implemented neuron model is time based, the timing of the network should satisfy biological requirements. At a speed-up factor of about 10^5, an axonal delay of 20 ms in the biological time domain requires chip-to-chip transmission latencies of 200 ns. For a FPGA clock frequency of 156.25 MHz the time for a single hop transmission of the MCGN alone is given as 160..180 ns [Phi08a]. Hence, the timing requirements are on the limit of what the hardware can provide. Therefore minimizing latency must be a primary objective of the design.

3.2 Design Principles

This section outlines the principles of multi-chip operation. It discusses general aspects, while the section following describes core components, which realize these principles.

3.2.1 Sorting of Event Streams

To satisfy the requirements stated in Section 3.1.1, event streams going to *Spikey* have to be sorted by their time stamps. Classical sorting algorithms (e.g. BubbleSort, QuickSort [SS02]) often assume, that the data to be sorted is stored in a randomly accessible memory. Then they define a sequence of comparison and memory movement operations to iteratively transform the data into a sorted state. These algorithms require the complete data to be available before they start, thereby introducing additional delay. Using such an algorithm to sort the event stream, all events within a certain range of time stamps would have to be stored in order of arrival. Then, the algorithm would iteratively sort the events and afterwards transmit them to the chip. During sorting, no further events for this range of time stamps can be accepted. Thus the time required for sorting, determines the minimal achievable delay.

This time can be reduced by shortening the time stamp range wherein sorting takes place. But only events within this range can be accepted and stored by the sorter. A large size is necessary due to uncertainties in the transmission latency (cf. Section 3.1.1). It is also desirable, since it provides a higher dynamic range of programmable delays to the experimenter.

Insertion Sort The idea of Insertion Sort [SS02] is very simple: Each input element is directly inserted at the correct position within the output array. The position can be determined by sequentially comparing the new element to the already placed elements. It is then inserted before the first element which compares greater. To reduce comparison operations, a binary search [SS02] can be used to find the proper insertion point.

The algorithm exhibits two desirable features: The output data stream is always well ordered and the input data is processed in one pass, therefore allowing a streamed operation.

The main disadvantage is, that for every insertion all following elements of the output array have to be moved down one place. This leads to a comparatively high runtime complexity.

Modified Insertion Sort Exploiting the special characteristics of the problem at hand, this disadvantage can be overcome. Events are not required to be stored in a sequence without gaps. Instead they can be stored in a hash table with the time stamp as key. To get a sorted stream, the readout side only has to read from the table at the index corresponding to the current time. An example of the algorithm in progress is given in Figure 3.2.

The need for comparison and movement operations is eliminated altogether. It is therefore possible to insert events within a single clock cycle, so that the throughput of the data path is not reduced. As a positive side effect this algorithm also removes events with identical time stamps

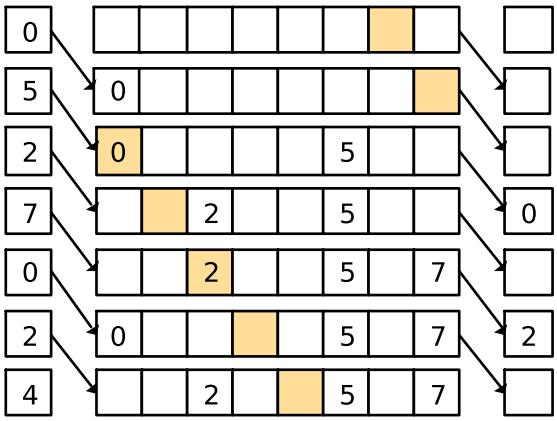

Figure 3.2: Illustration of the modified Insertion Sort algorithm: Every row shows the situation within the current clock cycle with time advancing downwards. To the left are time stamps of incoming events. To the right are outgoing events. The middle part represents the hash table with eight entries. The highlighted cell is the one being currently read. Its content will be output in the next clock cycle.

from the stream, which is stated as a requirement in Section 3.1.1. To make optimal use of this feature one would implement one hash table for every block of synapse drivers. Otherwise also non-colliding events with identical time stamps have to be dropped. How many hash tables can be implemented is primarily a question of available logic and memory resources.

Look ahead extension The algorithm introduced so far has a severe drawback. It was stated in Section 3.1.2, that the system time counter is incremented with the doubled FPGA frequency. This counter is the reference for the event time stamps stored in the table. Because there must be a table entry for every time stamp, the algorithm would make it necessary to read with twice the FPGA frequency. There are two arguments opposing this: First, it would make the implementation more difficult. The planned frequency for the FPGA clock is 156.25 MHz, so the table would require a readout frequency of 312.5 MHz. For example the maximum clock frequency of the BlockRAM storage elements specified in the datasheet [Xil07] is 355 MHz for register to register transfers. At this frequency only very short logic paths are possible, making the place and route process more difficult. Even more complications arise, because the existing *Spikey* controller and the network logic were not designed for such high frequencies and would have to remain at 156.25 MHz. This would introduce additional clock boundary crossings, which require special care and resources. Furthermore, the limited clock generation and distribution elements of the FPGA, which are used to full extend in existing designs, would be burdened even further. The other argument against is, that the link to *Spikey* also operates at the FPGA frequency. Even if every entry of the table could be read, not all events stored therein could be transferred.

A better solution is to not only consider one entry of the table per clock cycle, but to also look ahead at future entries. This way a decision can be made which entry is to be read next. For example when only few entries contain valid elements, one can skip empty ones and “fast forward” to the next valid event. When there is a dense sequence of events, one can use the time gained during the fast forward to read more of them than would be possible otherwise.

Of course the range of the fast forward must be limited. A minimum offset to the system time counter must be guaranteed, to account for the transmission time to the destination synapse driver on the chip. The maximum offset is a parameter that can be optimized. Latency is limited by the time difference of the read position and the system time. The further the readout process is allowed to advance into the future, the greater the minimum latency will be.

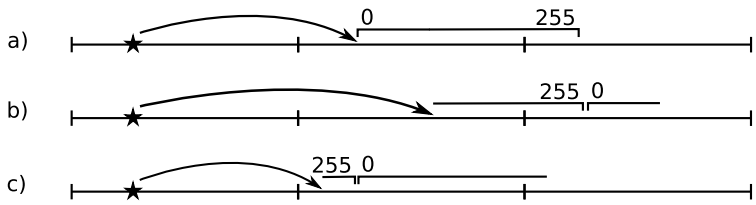

Figure 3.3: The diagram illustrates the effect of jitter amplification. The three subpictures show the transportation process of a single event for slightly different latencies. Time increases from left to right. Ticks mark overflows of an 8 bit time counter. The event is digitized at the time indicated by the asterisk. The arrow illustrates the transportation time to the destination *Nathan*. The bracket above the timeline, marked with "0" and "255", indicates, when the event will be delivered by the DTC depending on the programmed delay.

3.2.2 Providing Deterministic Neuronal Delays

An 8 bit time stamp and a 4 bit time bin represent the timing information of digital events coming from *Spikey*. The number stored within the time stamp is taken from the system time counter, clocked with the fast clock on the chip. The time bin is an extension to increase precision of the spike generation by the DTC. Processing of events on *Spikey* is - outside of the analog part - always done with the precision of the time stamp. The event input and output buffers treat the time bin information as payload data and pass it through unmodified. Therefore, the same is done outside of the chip. According to [Phi08a], the minimal transmission delay for the transport network is 25 FPGA clock cycles or 50 time stamps. This does not include additional waiting delays inside buffers on *Spikey* and higher level FPGA logic. To provide the required dynamic range for the programmable delays, while limiting the datapath width, it is decided not to make the delays programmable with the precision of the time bins. Instead the time bin is treated as payload data, that is only forwarded unmodified. In biological terms the precision of the delays would therefore be 320 μs at a speed-up factor of 10^5.

Delay mechanism To artificially delay an event, the time stamp must be incremented to the destination time. The DTC will then pass the event to the analog part at the intended instant. Due to the sorted-by-time stamp requirement of *Spikey* (cf. Section 3.1.1), it is desirable to transmit events at the latest moment possible. Otherwise events arriving later, but intended for an earlier target time, would have to be dropped. Therefore an intermediate storage memory is required.

This memory is provided by the sorting table described above. The only thing to do is increment the time stamp, before the event is inserted into the table.

Jitter amplification Figure 3.3 illustrates a problematic effect when using 8 bit time stamps: A small variation in transmission time t_d can lead to the event being delayed for the length of a full counter period P, where $P = 256$ for 8 bit time stamps. When t_d is greater than P (illustration b), for some programmed delay values D the resulting target time stamp will be delivered one period too late. One example value in the diagram is $D = 0$, for which in b) the event arrives 256 cycles later than in a). So instead of being compensated, a small variation is amplified in this case. When t_d is close to P this can happen at random due to network jitter.

Subfigure c) shows, that this is not only a problem for lower values of D, but can also happen to the upper end of the range. For a programmed delay of $D = 255$, events in a) and b) arrive in period three, but in c) are delivered in period two.

The root of the problem is, that because of the timing relations in the system, there may be more than one wraparound of an 8 bit time counter within the lifetime of an event. This leads to ambiguous situations, as the receiving side has no definite information about when the event was digitized. Therefore it can not reliably decide when it should be delivered. Consequently, events would not be delivered deterministically.

Time stamp extension To solve this problem, the easiest way is to extend the time stamp by additional bits. That way P can be made much larger than t_d and it is assured, that only one wraparound of the time counter occurs during the lifetime of an event.

To allow for long range connections, it was decided to extend the time stamp by 4 bit. That way extended events are 25 bits wide and still easily fit into a single two cycle network slot of 32 bit. It has to be determined by experiments (cf. Section 6.2.1), whether this extension is large enough.

3.2.3 Routing Algorithm

It was discussed in Section 2.2, that routing between network nodes is configured by programming the static routing table of the MCGN switch. But additional routing is required inside the FPGA to direct events to a designated user port and time slot of the MCGN. Also source events have to be mapped to target events before transmission to *Spikey*.

Logical connections Between two neurons logical connections are established. A connection has a fixed axonal delay, by which timestamps are incremented as outlined in the previous subsection. Multiple logical connections, that interconnect neurons on the same pair of chips are bundled together. A number of time slots, which are routed through the gigabit network to the destination node, are associated with it. Each bundle is associated with a set of time slots by which they are transported through the gigabit network.

Connection tags To distinguish connections within one bundle, every connection is tagged with a `subnr`. The neuron address of the source event is not used for this purpose directly. The receiving side would have to store an entry for every neuron address of every remote *Nathan* it is connected to. By introduction of the `subnr` it is possible to reduce the amount of required entries, because information needs only to be stored for remote neurons, which are actually connected.

3.3 Building Blocks

In the following section, the basic logic components, that form the design, are described individually. All of them are implemented in every *Nathan* FPGA. The top components are separated by direction of the data path. `Sender` reads events coming from *Spikey* and passes them to the network. `Receiver` writes events coming from the network to *Spikey*. Event streams are sorted at the destination within the `Sorter` block, which is described separately.

3.3.1 Sender

Figure 3.4 shows an overview of the sending logic. On the left side is the *Spikey* controller interface described in Section 2.3.3. This controller has been altered to provide events coming in from *Spikey* to the outside logic. It presents three events from a single packet in parallel. `Source_gen` contains a lookup table (LUT), that stores the initial routing information. It transforms events according to these information and forwards them to the `Send_switch`, which forwards event streams to the requested user port. There, the `Transmit_buffer` holds events until a time slot for them is available. It interfaces the MCGN switch logic and is responsible of inserting events into the correct network slots.

Source_gen

`Source_gen`, which stands for source event generator, is the first event processing stage. Its LUT stores whether events should be transported at all and, if so, where they should be send to. By default all events, that occur on *Spikey* are also transmitted to the FPGA, to be stored within the *PlaybackMemory*. Since not all of them may be connected to neurons on other chips, it is

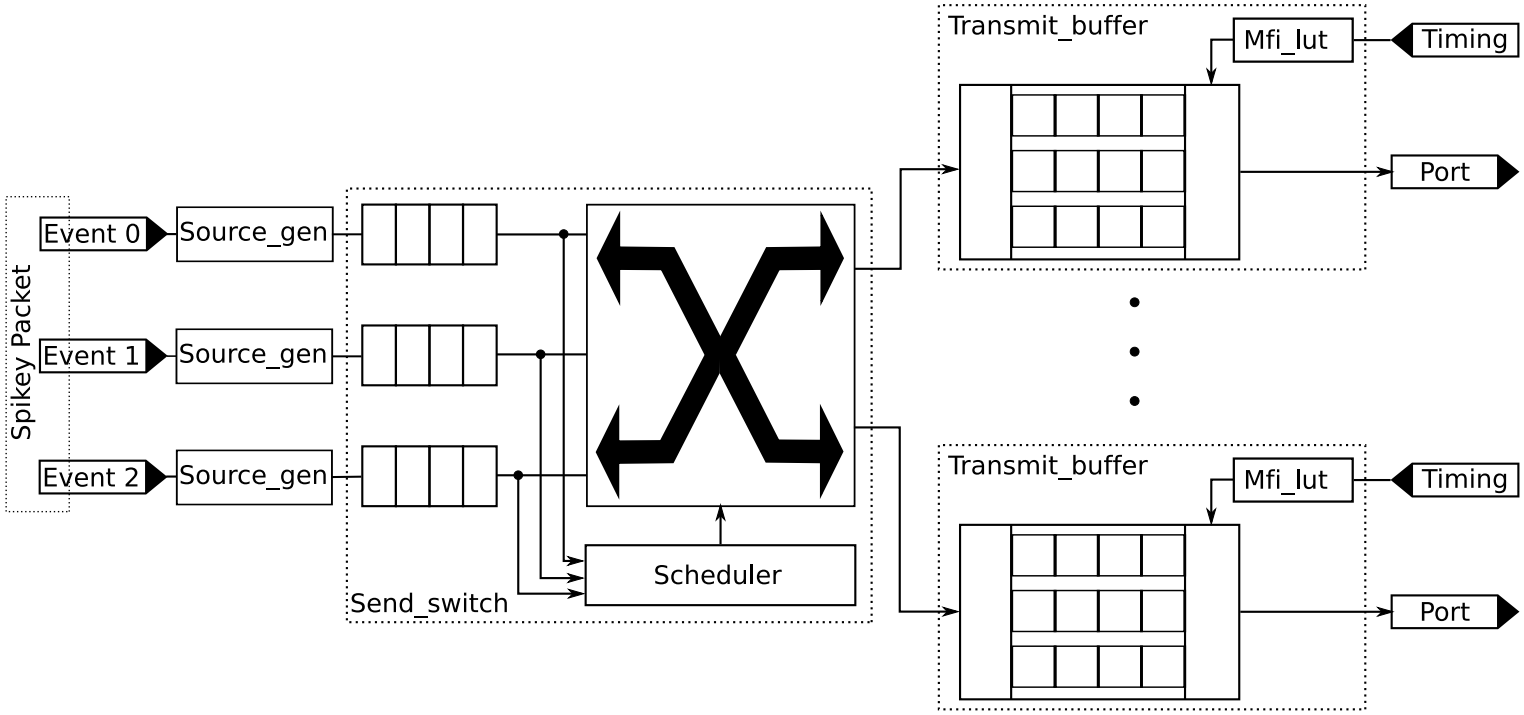

Figure 3.4: Data flows from left to right through the sending logic. The `Send_switch` connects the *Spikey* interface to the network interface. The dots indicate, that a flexible number of user ports can be used by instantiating multiple `Transmit_buffer`. Refer to the main text for a discussion of the components.

necessary to drop unwanted ones. This is done by setting the drop-bit in the respective lookup table entry.

If the event is to be transmitted, three pieces of routing information are provided:

- The user port number
- A number identifying the connection bundle for that user port.
- A `subnr` within the bundle.

To reduce the index size of the LUT a special feature of the *Spikey* interface is used. According to Table 2.2 neurons are linked to fixed slots within an event packet. So only a subset of all neuron addresses will be presented on a single slot. Because of this, it is sufficient to only use 7 bit as index into the lookup table. The tables of all three `Source_gen` instances must of course contain different entries.

Time stamp extension `Source_gen` is also responsible for calculating the extended time stamp. Events from *Spikey* have 8 bit wide time stamps, which are to be supplemented with further bits from the FPGA system time counter. But because time passes between event generation on *Spikey* and reception on *Nathan*, the additional bits can not just be copied from the counter. Instead, two cases are distinguished:

- When the 8 bit time stamp is lesser than the lower 8 bit from the FPGA time counter upon reception, it is assumed, that the event was digitized within the same period. The extended bits are copied from the time counter.
- When the time stamp is greater or equal to the lower byte, the nibble from the time counter is taken decremented by one.

Send_switch

The `Send_switch` forwards event streams from the three input slots to one or more `Transmit_buffer` blocks. It is required, because events from any packet slot may need to be transported via any user port.

Alternative without Send_switch Considering a design without `Send_switch` points out the reasons for its use: In this alternative the three input slots would be statically connected to three user ports. Then events from one neuron block could only reach a single user port. If neurons within this block are connected to neurons on multiple different remote chips, a time slot at the associated user port must be reserved for each destination node. If this is the case for all blocks, over-reservation of network resources would be the consequence.

A bundle transporting events from all three input slots at low rates would still need at least three time slots in total, even if one slot would suffice for the rate. It would also impact latency, as for an identical number of totally reserved slots, fewer are available for a single `Transmit_buffer`. So on average events would have to wait longer.

Besides that, this alternative would reduce flexibility in the number of user ports, because it requires exactly three ports.

Switching The switch is an input buffered switch, with a crossbar interconnecting input and output ports. Incoming data is buffered, before it enters the crossbar and a scheduler configures the crossbar depending on the outputs requested by the input events. The details of the implemented switch are not important for the principle of operation and can be tuned to optimize performance. However the input buffer can not be omitted to allow situations, where events from two slots request the same output port.

The scheduler selects the input to output mapping depending on the user port number requested by the event, which is provided by `Source_gen`. The data passed through contains the extended event time stamp and bin, the `subnr` and the connection bundle number.

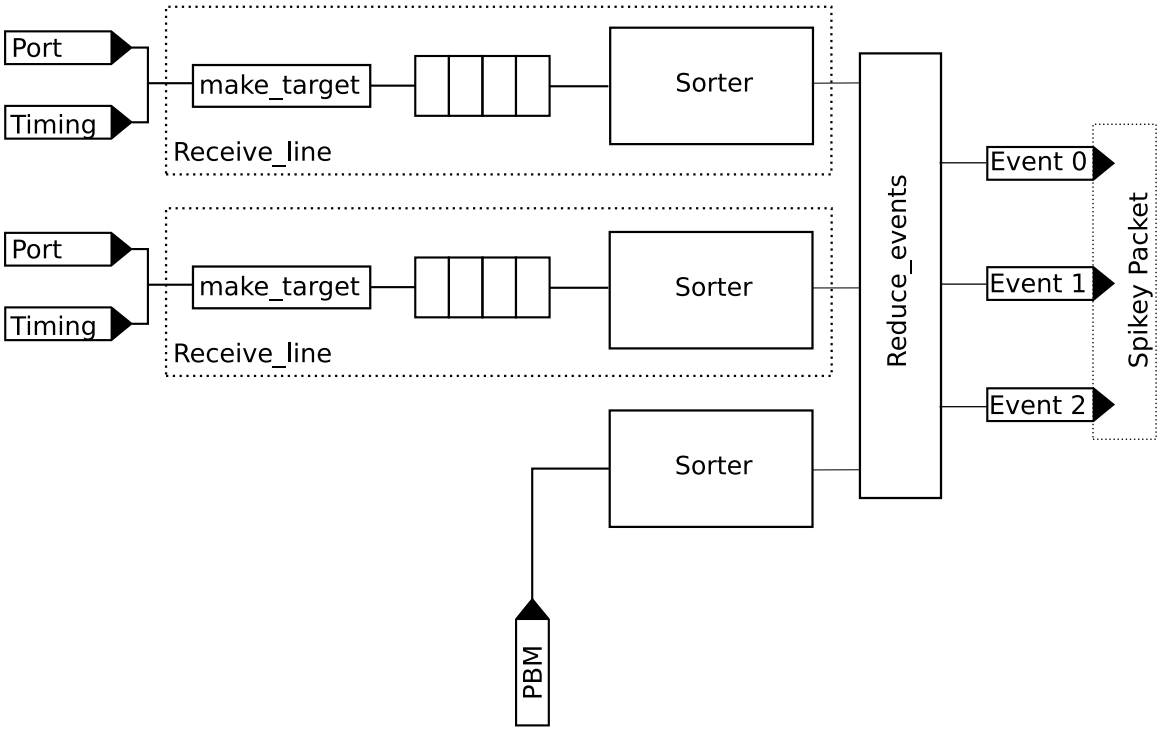

Figure 3.5: In this illustration the receiving logic is shown. Data coming from the network flows from left to right. Events from the *PlaybackMemory* (labeled with "PBM") are mixed with two network streams, using a `Sorter` block as an adapter. The *Spikey* interface controller is on the right side. Other combinations of `Receive_line`s and *PlaybackMemory* stream adapters are possible. Refer to the main text for a discussion of the components.

Transmit_buffer

For every port a `Transmit_buffer` is instantiated. Its purpose is to store events until a valid slot for their connection is available. It contains the logic to transmit the event to the user port. Timing details of the interface can be found in Figures 2.4 and 2.5.

MultiFIFOs A central component is the `MultiFIFO` block, which is reused from code for [Phi08a]. It organizes multiple logical FIFOs within one memory element. It is suitable for situations, where multiple queues are needed, but only one of them is read or written at a time. This is applicable here, as `Send_switch` forwards one event at a time and on the reading side, one event at a time is forwarded to the MCGN switch. Each of the queues inside a `MultiFIFO` stores events for one connection bundle.

Mfi_lut To determine when to read which queue, the `Mfi_lut` maps the slot number provided by the MCGN switch onto the index of the corresponding queue. This index number is referred to as multi FIFO index (`mfi`). It identifies the connection bundle on the local user port.

3.3.2 Receiver

Figure 3.5 gives an overview of the datapath of `Receiver`. It shows a special configuration, where two user ports and one event stream from the *PlaybackMemory* are used. It is also possible to use two or three streams from the *PlaybackMemory* by sacrificing one `Receive_line` for each.

Events coming from the network are mapped to target events within `make_target`. They are afterwards stored in the `Recv_queue` FIFO memory and then fed into the `Sorter` module. Events

from the *PlaybackMemory* are send as is and therefore no `make_target` module is required. The output FIFOs of the *PlaybackMemory* also make any other queues unnecessary.

Because the individual streams are processed in parallel, checks for synapse driver collisions (cf. Section 3.1.1) need to be done in the `Reduce_events` module. All operations, that require information of all streams, have to be done or supported by `Reduce_events`. It produces an event packet that is then transferred to the *Spikey* controller. Its structure, illustrated in Figure 2.7, needs to be modified to allow external input of events and to provide streams from the *PlaybackMemory* to the outside. All events to *Spikey* have to pass through `Receiver`.

Discussion of the `Sorter` module is - due to its complexity - separated into another subsection. The rest of the components is discussed here.

make_target

As outlined in the description of the routing algorithm (cf. Section 3.2.3), events have to be mapped onto target events, before they are passed to *Spikey*. This is done by the `make_target` process. It increments the timing information of the source event by the programmed delay and fills in the target synapse driver address from a lookup table. Index into the lookup table is the `subnr` of the arriving event and the number of the connection bundle. The latter is derived from the network time slot, in which the event arrives. The slot number is used as index into the `Slot_lut` lookup table, which provides the bundle number.

At this point it should be noted, that there is no unique number identifying connection bundles. Instead, on the receiving side a local virtual connection (`lvc`) number is used, that is unique for each connection bundle arriving at a specific user port of the receiving FPGA. On the sending side the `mfi` is used in a similar fashion (see above). So `mfi` and `lvc` are not identical numbers. This is done to allow for a reduction of the amount of routing information, that needs to be stored per port. By using only local identifiers, the lookup tables can be made smaller.

For each arriving event the `Recv_lut` stores an 8 bit delay value and the address of the targeted synapse driver. The `subnr` in the address field is replaced by the synapse driver address. For calculation of the target time stamp, the delay value is the combination of a common high nibble and the individual 8 bit connection delay.

Recv_queue

Before events are stored in `Sorter`, they may be delayed inside `Recv_queue`. It was described in Section 3.2.1, that sorting is done by storing events in a hash table with time stamps as keys. Therefore the time range of the `Sorter` is limited by the size of the table and events with time stamps too far in the future can not be accepted. Instead they are delayed in the `Recv_queue`. At which precise time events are forwarded to the `Sorter` is determined by an accept condition, which is discussed later.

Reduce_events

The `Sorter` modules work in parallel without direct interaction between each other. The `Reduce_events` module provides services to coordinate these units. Its purpose is to reduce multiple event streams into a single packet stream, that can be send to *Spikey*. It has to take measures to synchronize the readout processes in `Sorter` and to prevent collisions, when multiple events for the same time stamp and synapse driver are valid in one packet. As it has to closely interact with the sorting mechanism, a description of how these features are achieved is given in the next section.

3.3.3 Sorter

The purpose of the `Sorter` module is to implement the sorting algorithm described in Section 3.2.1. It is part of the receiver logic and can take input from `Recv_queue` or directly from the *Playback-*

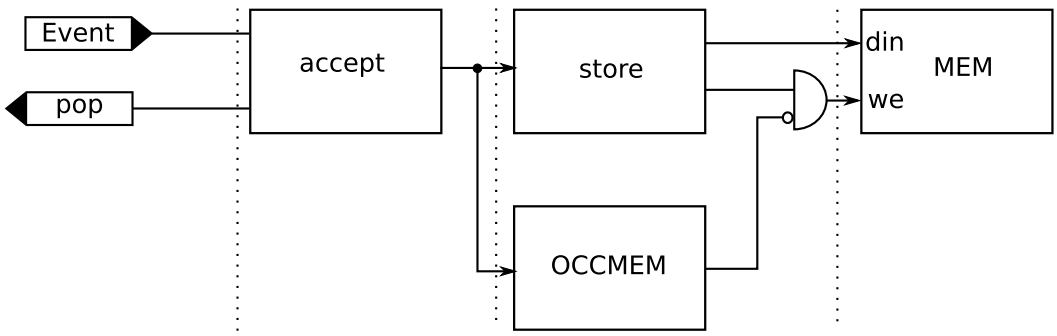

Figure 3.6: This figure gives an overview over a part of the datapath within `Sorter`. It shows the path of incoming events until they reach the memory. The dotted lines separate the pipeline stages. MEM and OCCMEM are RAM modules, whereas accept and store are registered combinational logic blocks. Reading of events is illustrated in Figure 3.7.

Memory. However `Sorter` is aware of the kind of input source and uses handshake signaling only with `Recv_queue`. This handshake is required to delay events with time stamps too far in the future. The *PlaybackMemory* readout logic delays events itself, making a handshake unnecessary.

Overview To implement the sorting algorithm, a table is required, where events can be stored with their time stamp as key. In digital logic, such a table is provided by random access memory (RAM) elements. Time stamps serve as the storage address within the RAM. To keep the required size small, only the lower 8 bit of the extended time stamp are used. So the RAM must provide storage for 256 events. The stored data consists of 9 bit synapse driver address and 4 bit time bin, resulting in a RAM size of 3328 bit.

Because events are continuously fed into the `Sorter` and read from it, a dual port RAM is needed. This means, that it must be possible to perform two access operations, either read or write, simultaneously in every clock cycle. If only one port was available, the input and output side would have to alternatively access the memory, effectively reducing the throughput by a factor of two.

Entries in the memory have to be marked as valid or invalid. If one stored an additional valid bit for this purpose in the same memory as the event data, two accesses on the RAM would be necessary to store a single event. At first a read operation would have to check whether the address is not already occupied, and, if so, a write operation would store the event. This would also reduce the effective throughput by a factor of two.

While the need for two accesses can not be eliminated, if one does not want to overwrite stored events, full throughput can be maintained by storing the valid bit on a separate memory. That way the acceptance of events can be pipelined and the two memory operations are carried out in parallel.

It was also stated, that the algorithm uses a look ahead mechanism to check multiple addresses for valid events, within one clock cycle. This can be accomplished by making one port of the memory, which is containing valid bits, wider then the other. That way multiple valid bits can be read out within one clock cycle.

Writing of events

To support the description of the storage process, Figure 3.6 shows the relevant part of the datapath. MEM is the RAM for event data and OCCMEM the one for valid bits. Storage is pipelined into three stages:

Acceptance of events The accept logic reads the event presented on the input and decides whether it should be considered for storage. The decision is made, based on its time stamp and

the current time. Due to the size of the memory, it can be stored only when its time stamp is less than 256 time stamps in the future. When this is true, `pop` is asserted to remove the event from the preceding FIFO stage. The accept condition is derived in the following:

The extended 12 bit time stamp T_c is compared to two comparison values T_{high} and T_{low} of the same size. They are calculated from the current address, at which the readout process is reading. Two comparison values are needed, because the algorithm reads a window of w valid bits at once. Events, that would need to be stored within this window, but arrive after it was read from memory, must be dropped, because a decision of what events to read has already been made at this point of time.

Let R_c be the current read position extended to 12 bit, then $T_{low} := R_c + 2w$ and $T_{high} := R_c + 256$. The accept condition is then given by

$$(T_c < T_{high}) \wedge (T_c > T_{low}) \qquad \text{for } R_{ext} \neq 15 \tag{3.1}$$

$$T_c < T_{high} \qquad \text{for } (R_{ext} = 15) \wedge (T_{ext} = 0) \tag{3.2}$$

$$T_c > T_{low} \qquad \text{for } (R_{ext} = 15) \wedge (T_{ext} = 15) \tag{3.3}$$

R_{ext} and T_{ext} are the highest nibble of R_c and T_c respectively.

Three separate equations are necessary to account for wraparound effects of the 12 bit time counter. Condition (3.1) is the normal case: Neither T_{low} nor T_{high} will overflow, when $R_{ext} = 15$, and it can be directly checked if the time stamp is within the valid range. Condition (3.2) is used, when the time stamp lies within the next period of the 12 bit counter, but the read position has not wrapped around yet. The event may be less than 256 time stamps in the future, but $T_c > T_{low}$ would never be true. This condition can be dropped, since the case differentiation implies, that the target point of time is greater than the current read position. Similarly for Condition (3.3), $T_c < T_{high}$ would not be fullfillable, because T_{high} has already wrapped around. Here, too, the condition can be dropped, because the case differentiation implies, that the target time stamp is at maximum 256 time stamps in the future.

When events come from the *PlaybackMemory*, no handshake signaling is used. Then the accept mechanism is of no use, because events can not be delayed and are lost, if they are not accepted. However, the readout algorithm would lose events when they arrive between R_c and $R_c + 2w$. These situations are actively detected with another accept condition: Let R be the current 8 bit read address, then, also 8 bit wide, $T_l := R$ and $T_h := R + 2w$. T is the 8 bit event time stamp and the accept condition is

$$(T \geq T_h) \vee (T < T_l) \tag{3.4}$$

During operation of the network, this accept condition will never be false. This is the case, because the *PlaybackMemory* itself ensures a minimum time difference between the target point of time and the time of delivery to `Sorter`. Therefore such collisions will not occur.

The interaction between the accept process and `Recv_queue` can lead to prolonged blockage of `Sorter`, when merging two event streams. If variability of time stamps of the two streams is comparable to the range of the sorting process, an event in `Recv_queue` can block following ones. The blockade can last long enough, so that, when leaving the queue, the reading position has already passed their time stamp. In this case, they would be delayed until the time stamp is valid again after the next wraparound of the 12 bit system time, blocking the queue themselves. To prevent this, a drop condition is evaluated and, when true, events are removed from the queue, without storing them in `Sorter`.

The condition assumes, that events more than 2048 time stamps in the future are outdated and should be dropped. The usable range of the 12 bit time stamps is therefore limited to 2048 instead of 4096.

Storage of events When an event is accepted for storage, it is passed to the next pipeline stage. The `store` block in Figure 3.6, behaves like a register and delays the event for one clock

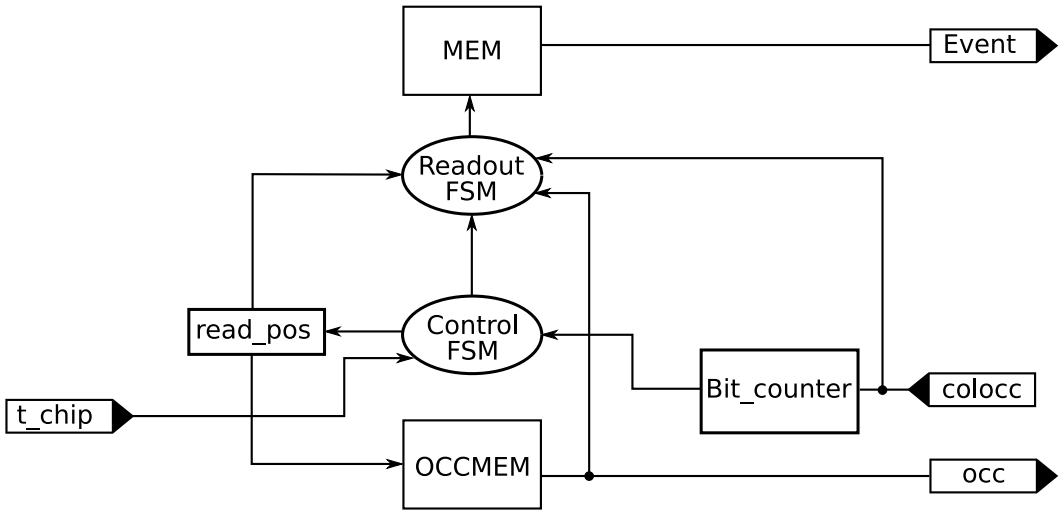

Figure 3.7: The picture shows the reading side of `Sorter`. Two finite state machines (FSM) control the readout of the `MEM` and `OCCMEM` memories. Not shown in the figure is an `enable` signal for `Sorter`, which is used to start the readout. Writing of events is illustrated in Figure 3.6.

cycle. Simultaneously, a lookup in `OCCMEM` is performed, to check whether the address destined for the event is already occupied. In the same cycle the write enable of `OCCMEM` is asserted, to mark the slot as occupied. This can be done without knowledge of the result of the read operation, because the valid bit for this address will be set afterwards, whether it was occupied before or not. The actual writing process to `MEM` is performed in the next clock cycle. The result from `OCCMEM` is combined with the valid bit from `store` and presented to the write enable of the destination RAM. Only when the slot was free before, is the write enable asserted, thereby preserving already stored events.

Reading of events

The reading logic of the `Sorter` performs several tasks: Events must be read synchronously to the system time counter. The look ahead feature presented in Section 3.2.1 requires the parallel processing of a range of valid bits. The reading process must adapt to the population of valid events within this range. Doing so the time difference between reading position and system time must stay within limits. And last, multiple parallel `Sorter`s must be synchronized to prevent race conditions in event streams to synapse drivers as detailed in Section 3.1.1. This is done in cooperation with the `Reduce_events` module of `Receiver`.

Figure 3.7 illustrates the datapath for reading from the `Sorter` memories. The readout process is controlled by two coupled finite state machines (FSM). The readout FSM selects the actual address to read from, while the control FSM steers the process on a higher level. It reads the current system time from the `t_chip` port and writes the `read_pos` register storing the current start address of the window of valid bits. The content of this window is output on the `occ` port and also used by the readout FSM to select addresses. The `Reduce_events` module performs an or-reduction of the `occ` signals from all parallel `Sorter`s. The result is fed back through the `colocc` (for collective occupation) port and the number of set bits therein is counted by the `Bit_counter` combinational logic block. The resulting event stream is given out through the `Event` port.

Control FSM The Control FSM writes to `read_pos` and determines how many events are to be read from the next window. It takes the system time and the number of set bits in the collective window as inputs. Basically, the FSM controls where and how much to read, but not which events exactly. By using `t_chip` and `colocc` as inputs, the respective FSMs in all `Sorter` instances see

the same inputs and behave identically. So output events have always the same time stamp and race conditions in event streams to synapse drivers are prevented.

It also means, that the control FSM sees the combined stream of all `Sorter` modules, which contains events from other streams along with its own. This is not the optimum solution for maximizing performance, but other concepts, that improve on this part, need to check each event against the history of events from all other `Sorter` blocks. Another possibility is to sort by addresses, before events are passed to `Sorter`, requiring a more complex logic. See Section 8.2 for more on this.

The `read_pos` register holds the start address of the reading window. For a window width $w = 8$, for example, it will successively have the values 0, 8, 16, 24, etc. The FSM exerts control over the reading rate by varying the time between updates of `read_pos`. Depending on the time difference b between `read_pos` and `t_chip` and the number m of set bits within `colocc`, the number n of events to process within one window is calculated. This calculation is performed, when a new window has been read from `OCCMEM`. It is tried to maximize n and b, while keeping $b_{min} \leq b < b_{max}$, for given limits b_{min} and b_{max}. Due to the relations between system time and FPGA clock, $n = \frac{w}{2}$ leads to no change in b[1].

When no bit is set in `colocc`, the FSM will set n to a minimal value, until $b = b_{max}$ and thereafter $n = \frac{w}{2}$. When a full window is encountered, $n = w$ allows to read all events, by decreasing b towards b_{min}. The calculation must choose n in such a way, that b does not underrun b_{min}.

Readout FSM The readout FSM iterates over the valid bits in `colocc` and calculates the address of the associated events within `MEM`. It takes `read_pos` as base address and adds the bit position within `colocc`. The event is only marked as valid on the output of `Sorter`, when its bit in `occ` is also set. The state machine is signaled to restart by the control FSM, when a new window was read from `OCCMEM`.

[1] n is required to be even

Chapter 4

Simulations for Prototyping and Testing

This chapter describes the simulations that helped prepare the implementation and the tools that were used in the process. The simulations themselves are not primarily intended to produce specific results, but serve as a development tool. Therefore, one can consider the conceptual design (Chapter 3) and the implementation itself (Chapter 5) as results of the described activities. This chapter focuses on methodological aspects.

4.1 Why Simulate?

Hardware design is usually an iterative task. Designers often have only a basic idea as a starting point and find more sophisticated solutions as their understanding of the system increases. They then can go back to an earlier stage and incorporate the new ideas, usually attaining an improved solution. In this process, the time it takes to implement a new idea and test it, defines how fast development progresses. The level of detail of the preliminary testing determines the efficiency of each iteration.

In the initial development phase of digital logic, testing is done in simulations. They play a central role during the development process, since, in particular for ASIC design, implementation in hardware is expensive. Even for FPGAs, which can be reprogrammed with new circuits at any time, simulations serve important purposes:

Simulation as prototype Especially when designing on the system level, with several devices interacting, it is beneficial to be able to concentrate on the higher levels of the design at first. Common hardware description languages allow more abstract constructs for simulation than for synthesis. For example, to generate random numbers one, can use a predefined function, whereas the description for implementation must precisely model the circuits to generate the random numbers. Therefore, in a high-level simulation of the system the basic functionality of the overall concepts can be tested.

Simulation as verification tool Simulations are important to verify the functionality of the system. This may be done at multiple stages during the development process, e.g. after the prototyping phase, before beginning with the implementation. Even for the final version, which can also be tested on an FPGA, simulations are important, because they allow more insight than experiments on the hardware. Every signal can be inspected and modified, while monitoring access to hardware devices is often limited.

Another advantage is, that, in simulation, a single part of the design can be isolated and placed in an artificial test environment. Therein, controlled test patterns can be applied to check the behavior for a set of predefined situations. This is especially useful, when a seldom occurring problem

is found. It can then be reproduced in a minimal environment and direct tests of modifications aimed at solving the issue are possible.

4.2 Tools and Methodology

For this thesis, simulations were used for prototyping as well as for verification. Moreover, during the development process, the prototype simulation was transformed into a verification simulation. Foremost, this was possible by using the SystemC[1] hardware description language and the ModelSim simulation software[2].

SystemC SystemC is an extension of the C++ programming language meant for hardware description. The focus – in contrast to VHDL or Verilog – is on system level simulation, especially, when the design comprises hardware and software components. SystemC extends C++ by means of a class library and requires no special compiler software. Therefore, a host of tools and code packages can be used for development and inside the simulation. The final simulation is a program, that, when executed, uses e.g. terminal and file output to report results. With a general purpose programming language as basis, it is possible to include sophisticated self-checking capabilities into the simulation itself.

Due to its system level approach, the language is ideally suited to simulate the event communication within the stage-1 system. Details, like the workings of the *Spikey* controller logic or the serialization of the gigabit transceivers, can be abstracted into high level methods. Thus, the complexity is reduced and the runtime is improved, allowing for faster development cycles. Complex parts, e.g. the event sorting algorithm, are rapidly added and can easily be modified, without too much concern of the actual circuits. Hence, a high-level system simulation allows the exploration of concepts. Another prospect of using SystemC is, that existing simulation code, written in pure C++, can be integrated.

ModelSim ModelSim is a simulator, which supports the VHDL, Verilog and SystemC hardware description languages. Mixed-language simulations combining blocks described in any of theses are possible. Also of interest is an interface to the C programming language, which can be used to call C functions from within VHDL or Verilog code.

ModelSim is integrated into the FPGA design flow of the *Electronic Vision(s)* group, which was used for this thesis. The integration allows simulation of the VHDL code used for synthesis in a test environment containing a complete stage-1 system. The simulator is therefore also used for verification tests on the final design.

Event based simulation Both SystemC, as well as ModelSim use an event based concept to simulate concurrent activities of the simulated units. The simulated design consists of concurrent processes that are executed on specific events. In digital logic, these events are signal transitions, on which the processes are sensitive. An example would be a register, that changes its output in reaction to an event caused by a rising clock edge. Each pending event is stored in a list together with its designated point of time, at which it is processed. The running of processes may in turn generate events, that are then added to the list.

4.3 Simulations

The approach for this thesis was to start with a SystemC prototype simulation to find suitable concepts for the realization of the event network. The prototype used existing C++ code (described in [Grü07]) to simulate *Spikey* and MCGN. The prototype was then successively translated to

[1] SystemC is standardized as IEEE Std. 1666-2005
[2] ModelSim from MentorGraphics is used in version 6.3a

VHDL for synthesis. In the following text, the specific steps of refinement of the simulations are detailed.

4.3.1 SystemC Prototype

Sorting of event streams was considered to be central from the beginning and so `Sorter` was the first component of the simulation. To not be too far away from the properties of the targeted hardware device, the process of sorting itself was implemented in detail. The difficulty lies in the timing of read and write operations on a single memory, while only one operation per port can be performed per clock cycle. To transfer this to the simulation, a global clock signal equivalent to the FPGA clock is used. However, the memory is not precisely modeled, but an array data structure is used. The designer has to check manually, that the number of memory operations does not exceed the limit.

Successively, the other logic blocks were added to the simulation. For the *Spikey* chip and its control logic, as well as for MCGN, existing C++ code blocks were available. They were developed as part of [Grü07] for the simulation of a proposed implementation of the event network. Since that simulation did not use SystemC, the code does not follow the event based approach, but has data interdependencies programmed implicitly into its procedural description. This means, that function calls are ordered in such a way, that in every clock cycle inputs are first processed, before new output values are stored. The code therefore does not make the concept of clock cycles explicit in contrast to SystemC, where each new cycle is treated as an event, that triggers the processing of input data. This difference is resolved by adapter modules with event triggered processes that make the function calls to the existing code in the correct order.

The simulation model of the *Spikey* chip supports several modes for event generation. The one used most frequently generates random events from a poisson process and allows to switch periodically to a higher rate to simulate bursting.

Event tracking To allow a fast development cycle, the simulation itself contains methods for a quick evaluation. The `Event_tracker` class assigns every event a unique tracking number upon its generation by the *Spikey* block. This number is used at several monitoring points to record the path an event takes. With this data, `Event_tracker` performs automated checks and generates a report.

For example, one check reports every event that does not arrive at a destination *Spikey* block and was not explicitly recorded as dropped. This ensures, that all possibilities of event loss in the system are known and understood. Another check measures the simulated time of arrival and detects events delivered too late or too early.

Besides the results of the checks, the report contains information about the maximum number of events in FIFOs, drop rates detailed by reason and configuration of the lookup tables. Thereby changes to event processing can quickly be evaluated, since the report points out functional errors and shows performance figures.

Large-scale simulations The SystemC simulation can interpret configuration files from the `mapper` software tool. This program was developed as part of [Phi08a] to map randomly generated artificial neural network descriptions to MCGN switch configurations. The existing C++ code blocks simulating the switches can read these files to configure their routing tables. The latest version of `mapper` supports the new event processing logic by additionally specifying the routing parameters described in Section 3.2.3 in the generated configuration files. Hence, it is possible to produce a configuration of MCGN and event network based on random neural networks and run experiments with it. This was used for large-scale simulations of a completely equipped *Backplane* with random connections between the *Nathan* network modules.

4.3.2 Mixed-Language Simulation

After the prototype simulation had evolved into a complete system for event processing, its purpose changed from serving as exploration platform for ideas to being a model for the implementation. The event network blocks, as described in Section 3.3, were rewritten in the VHDL hardware description language for implementation on the FPGA. In a mixed-language setup with ModelSim, they were used as replacements for the previous SystemC blocks and subjected to the same tests. This allowed to validate, that behavior was unchanged, yet required additions for event tracking.

The tracking number is treated as part of event data in the SystemC simulation. In the VHDL description, it is passed along with the event. FIFOs and buffers provide additional memory structures parallel to those for the event data. These additional structures are enclosed in compiler pragma statements that disable them for synthesis. Also included in this extra simulation logic are processes to detect and record dropping of events. To use the `Event_tracker` class in this simulation as well, the C interface provided by ModelSim is used. A C++ library has access to the `Event_tracker` object and exports access functions with C bindings, which ModelSim makes available in VHDL.

4.3.3 Simulating the Implementation

The existing design flow supports the simulation of the hardware description used for implementation. Subject to the simulation is a complete *Backplane* with 16 *Nathan* modules. The system is accessed via the *SlowControl*, which uses a file communication model to simulate control commands from the PC. Access operations are listed in a file and carried out in order during the simulation. Results are written to a response file for later evaluation. The command file can be generated by the `createtb` test software (cf. Section 2.3.4) and thus, the complete software hardware stack can be simulated. Only the analog part of the *Spikey* chip is not part of the simulation, because ModelSim does not support analog circuits.

4.3.4 Tests of Single Blocks

For a number of logic blocks, isolated simulations were developed, which only contain the individual component in a controlled environment. For the `Send_switch` block the test presents different stimulation patterns to the switch, to see if it forwards them correctly. For example, first a stream of one event per clock cycle with alternating requested output port is presented, thereafter one event per cycle at different input ports and last, one event per cycle to a fixed port and a second event every other cycle to another input port. Similar tests for `Sorter` allow to test specific arrival patterns of events and to systematically test relations between e.g. time of arrival and destined time stamp.

Chapter 5

Implementation

Chapter 3 outlined the basic workings of the required logic for multi chip operation of the FACETS stage-1 system. The fundamental principles and the basic structure were presented. This was done on a higher level of abstraction to focus on the general characteristics. This chapter will descend further down into concrete details of the implementation on the *Nathan* FPGA and of supporting software. For the creation of the final circuits, the design is described in the VHDL programming language. The software tools from Xilinx[1] are used for synthesis.

The text will continue, where Chapter 3 ended, and start with the implementation of the building blocks introduced there. Then, the integration of the design into the existing FPGA framework logic is described and the process of running an experiment outlined. The chapter will conclude with debugging facilities and the supporting software.

5.1 Implementation of the Building-Blocks

Section 3.3 introduced several components as building blocks of the design. This chapter will complete their description by supplementing details of their implementation. A focus will be set on aspects that are relevant for the operation and that may be useful to work building up on the design.

5.1.1 Lookup Tables

Various lookup tables are present in `Sender` and `Receiver`. These are:

- The `Send_lut` within the `Source_generator` of `Sender` stores the initial routing information.
- `Mfi_lut` maps slot numbers to multi fifo indices on the sending side.
- `Slot_lut` serves a similar purpose on the receiving side, mapping slot numbers to local virtual connection numbers.
- `Recv_lut` holds information to translate source to target events within `Receiver`.

The content of these tables determines the configuration of the network connections across chip boundaries. Its encoding is also relevant to user software, which wants to utilize the network.

Send_lut Three tables with different content exist, one for each *Spikey* packet slot. They contain 128 entries, that are 12 bit wide. The 7 bit index into the table is given by bit 8 and bits 6 to 0 of the neuron address. This exploits the fixed mapping of neurons to packet slots (cf. Table 2.2). Two blocks of neurons are wired to one packet slot and for one slot they are distinguished by bit 8 of the address. One LUT entry contains the following information:

[1] ISE WebPack version 10.1.03

Bits	*Name*	*Description*
11	`drop`	set, when event should not be processed
10..9	`port`	user port to send this event to
8..6	`mfi`	MultiFIFO queue to store this event in
5..0	`subnr`	connection tag to substitute address

Every `Send_lut` uses one BlockRAM in a dual port configuration. One port is used for lookup during experiments, the other for offline programming. The lookup port is configured 128×16 bit, which covers only 2 kbit of the 18 kbit available, leaving room for the addition of more information in the future. The programming port is configured 256×8 bit.

Mfi_lut and Slot_lut These tables both identify the connection bundle transported over a given slot, but they don't hold identical information. They contain 64 entries each due to the slot number size of 6 bit. `Mfi_lut` gives out a 3 bit MultiFIFO index and `Slot_lut` returns a 4 bit `lvc` number for the following lookup within `Recv_lut`. Both tables are implemented in single port RAM blocks using distributed logic resources.

Recv_lut The `Recv_lut` has 1024 entries of 17 bit each. The concatenation of `subnr` and `lvc` serves as index. The following information is stored:

Bits	*Name*	*Description*
16..8	`target_addr`	synapse driver address for the target event
7..0	`delay`	time stamp delay value

The table is stored in a single dual port BlockRAM with both ports configured as 1024×17 bit. Thus, 17 of 18 kbit are used, leaving room for a possible addition of one bit per entry. Because the BlockRAMs are one of the more expensive resources on the FPGA, it is very desirable in terms of scalability to fit the `Recv_lut` into only one BlockRAM module. Under this constraint it is not possible to use a larger index and hence, the total amount of bits for `subnr` and `lvc` is predetermined.

5.1.2 Implementation of the Send_switch

The `Send_switch` directs event streams to requested user ports. It is an input buffered switch and consists of three parts: Input buffer stage, crossbar and scheduler. The input buffer stage in the current implementation is a single register. The crossbar is implemented as one multiplexer per output, for which the scheduler - implemented as priority encoder - generates the selects. Each input raises a request signal to its destination port and the scheduler selects the input with highest priority for transportation within the next cycle. That way, it is possible for one input port to starve out the others, when multiple streams are to be routed to the same port.

The `Send_switch` implementation could be improved in several ways: The scheduler could be extended to support a dynamic priority scheme and deeper FIFOs with virtual output queues (VOQ) could be used. For VOQs, within one input FIFO, multiple virtual queues are managed in parallel - one for each output - to prevent head of line blocking. This would allow to implement iSLIP or other more advanced schedulers. Altogether, this would increase throughput and eliminate unfairness caused by starvation [McK99].

However, due to its simplicity, the `Send_switch` requires only few resources, while still providing a fully functional component. Since it was predictable, that the amount of available logic resources would be a severe limitation, a minimal solution was given advantage. This leaves open the possibility for improvements, should resources be available after the implementation is finished.

5.1.3 Transmit_buffer and Event Representation on the Network

The `Transmit_buffer` stores events until a valid network slot is available. It uses the `MultiFIFO` component, which stores its data inside a single BlockRAM. The memory determines the possible range for the configuration parameters number of FIFOs, their depth and width of entries. The used configuration is $8 \times 4 \times 32$ bit or 1 kbit in total of 18 kbit available. Pointers into the memory are managed by external FIFO logic. The amount of resources required for this logic, increases primarily with the number of FIFOs, since additional pointer registers are required for each. It grows to a lesser extend with the depth, as only additional bits for the pointers are needed. The RAM allows an upscaling of the parameters to a certain point.

Event representation Event data is transported in two cycles of a single network time slot. The first cycle holds time and the second address information.

Cycle	*Bits*	*Name*
0		
	15..8	`timestamp`
	7..4	`timebin`
	3..0	`timeext`
1		
	15..9	zero
	8..0	`address`

Actually only the lower six bits of `address` are occupied by `subnr`. The VHDL datapath description handles 8 bit sized `address` fields to allow an easy extension in the future. The synthesis tool may however recognize the two upper bits as constant and use that for optimization.

Transmission process The interface to the MCGN network port was discussed in Section 2.2.2, Figures 2.4 and 2.5 on page 9 show timing diagrams of an example transmission process. The `slot` signal of the timing information is used as index for the `Mfi_lut`, by which it is mapped asynchronously onto a FIFO index. When the associated FIFO is not empty, the logic checks if the switch is ready to accept new data. This is the case, when signals `accept` and `enable` are set and the first cycle of a new slot has started (indicated by `sofs`) or a first cycle has already been transmitted. Timing information is presented one cycle in advance, so the transmission process can be registered. Because of the registered operation, the `pop` signal to the FIFO is issued, while the first cycle is transmitted. The event will then be removed from the FIFO with the next clock edge, while the same FIFO is still selected. Would it be set later, the wrong queue would be popped.

5.1.4 Implementation of the Sorter

For the two memories outlined in Section 3.3.3 dual-port BlockRAMs are used. Regarding only size, it would be beneficial to implement `OCCMEM`, which stores only 256 bits, as distributed RAM. But, since two independent ports, both capable of read and write operations, are required, a BlockRAM must be used, because distributed RAM can not provide this feature. Its "left" port which stores incoming events is configured 256×1 bit. The "right" port which is used by the readout logic is configured 32×8 bit. This gives a look ahead window of eight events. For `MEM` both ports are configured 256×32 bit. Only the "left" port is writable, because event data doesn't need to be overwritten on readout. Thus, an implementation in distributed memory would be possible, if BlockRAMs needed to be saved and logic resources were available.

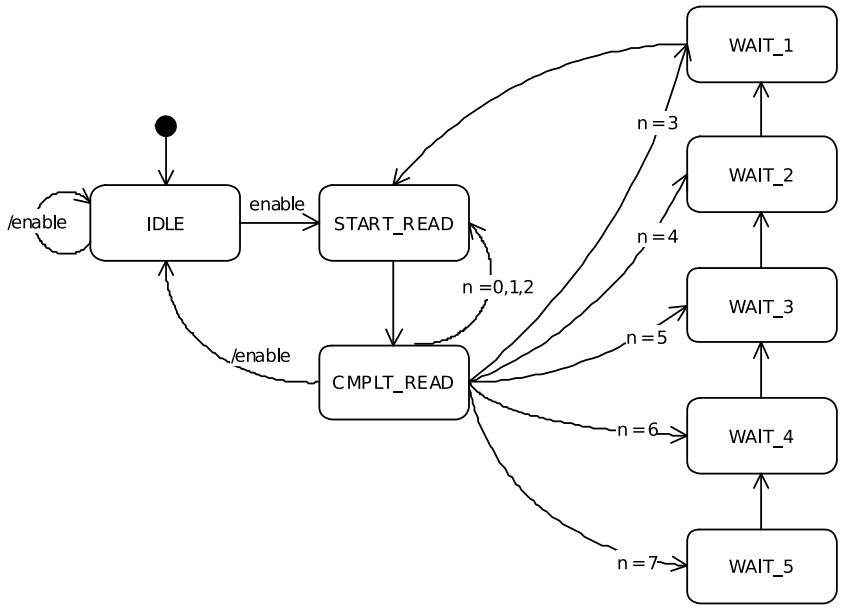

Figure 5.1: This is the state diagram for the control state machine. Transition labels show only the evaluated inputs, while all other inputs are ignored. n is the number of events, that may be processed within one window. `IDLE` is the reset state.

Bit_counter For the readout algorithm it is necessary to count the number of bits set in the `colocc` signal (cf. Section 3.3.3). This is done by the `Bit_counter` combinational logic block. A first implementation formulated its functionality straightforward as VHDL for-loop with a counting variable. But the *XST* synthesis software inferred an adder and a multiplexer for every iteration of the loop. For an eight bit window this resulted in a very long combinational path, which made place and route difficult. To get a solution, that takes up less resources and is also faster, an alternative approach was chosen: For the `Bit_counter_4` component a logic function is directly designed to count the set bits within one nibble. Two of them are instantiated for the lower and upper half of the `colocc` signal and their outputs are fed into a 3 bit adder. The 4 bit input x is mapped onto a 3 bit output y by the boolean function f, $y = f(x)$. y is a binary number between 0 and 4, that is equal to the number of ones in x. The truth table for f can easily be written down and coded as VHDL code.

Because x is 4 bit wide, f can be implemented within one lookup table (LUT4) of the FPGA logic resources for every bit in y. `Bit_counter_4` is therefore realized in three parallel LUT4, which is the minimal achievable size on this FPGA. It is also the fastest possible solution, since only one logic lookup is required. It was also verified with the netlist produced by synthesis, that only three LUT4 are inferred.

Control state machine

This FSM controls the readout process on a high level. It sets the `read_pos` register and determines how many events may be read within one window. Figure 5.1 shows the state diagram. There are three types of states: The FSM is in the `IDLE` state after reset and as long as `enable` is held low. This allows to disable event processing until *Spikey* is set up and synchronized. `START_READ` and `CMPLT_READ` are the main working states, where memory is accessed and the number n of events to process is calculated. The `WAIT` states are used to give the readout FSM enough time to process the calculated number of events.

In a strict formulation, the state would also include the `read_pos` register, because it is used in the transition logic in the same way as the state. The states in Figure 5.1 would then exist multiple

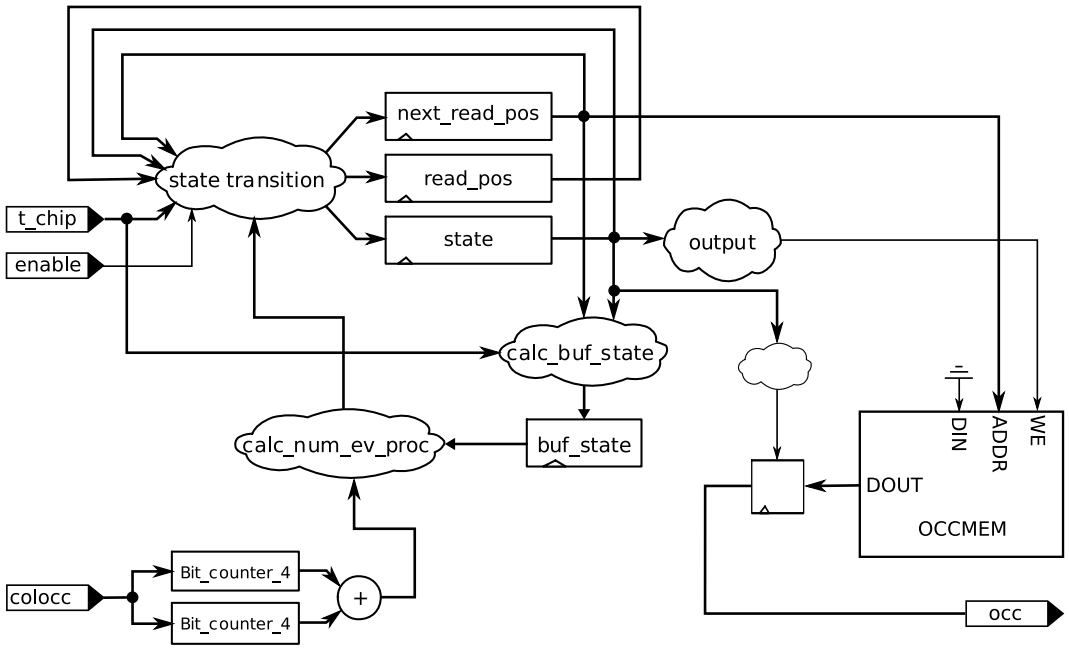

Figure 5.2: In this figure the logic elements that are implementing the control state machine and the related elements are shown. Clouds represent combinational logic and a triangle on the lower edge of a rectangle indicates a register. Buses are drawn bold.

times for every possible `read_pos` value. The FSM can also be understood as hierarchical state machine, where for each `read_pos` value Figure 5.1 shows the sub-states and their transitions. Then `START_READ` would transit to `CMPLT_READ` within another sub-state.

Figure 5.2 gives an overview of the logic elements used to implement the FSM and some supporting logic. The `next_read_pos` register holds the value `read_pos` will be updated to on the next transition from `START_READ` to `READ_CMPLT`. It is used to do calculations depending on the reading position, before `read_pos` is updated.

calc_buf_state The `calc_buf_state` process calculates the time delta b between `t_chip` and the reading position and distinguishes between several cases:

$$\texttt{buf\_state} = \begin{cases} \texttt{top} & b \geq b_{max} \\ \texttt{bottom} & b \leq b_{min} \\ \texttt{middle\_limited} & b \leq b_{min} + (w-1) \\ \texttt{middle\_free} & \text{else} \end{cases} \tag{5.1}$$

$$b = \texttt{next\_read\_pos} - \texttt{t\_chip} \tag{5.2}$$

`buf_state` is calculated only during the `START_READ` state. That way, in the `READ_CMPLT` state, information about the time delta is available in a form, that can be easily processed by the `calc_num_ev_proc` process.

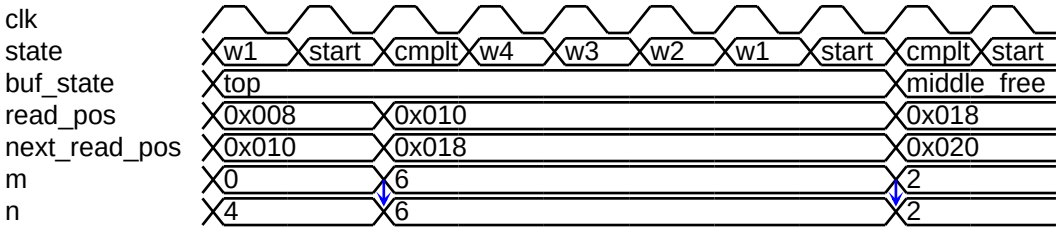

Figure 5.3: The diagram shows an excerpt of the readout process. States are abbreviated for readability: "start" stands for `START_READ`, "cmplt" for `READ_CMPLT` and "w1" etc. for `WAIT_1` etc. `read_pos` and `next_read_pos` are given in hexadecimal numbers. At the beginning the buffer is at the top and no events are set in the window. The second window holds six events and all of them are going to be processed. Consequently the buffer is below top for the next window.

calc_num_ev_proc In the next cycle, during the `READ_CMPLT` state, n is calculated by this combinational process.

$$n = \begin{cases} \max(w/2, m) & \texttt{buf\_state} = \texttt{top} \\ \min(w/2, m) & \texttt{buf\_state} = \texttt{bottom} \\ b - b_{min} & \texttt{buf\_state} = \texttt{middle\_limited} \\ m & \texttt{buf\_state} = \texttt{middle\_free} \end{cases} \tag{5.3}$$

Processing $w/2$ event slots doesn't change b. Thus in the `top` case, at least that much events are processed, to not increase the time delta any further. The opposite is the case for `bottom`. To not decrease the delta further, at maximum $w/2$ slots are processed. Between `top` and `bottom`, `middle_limited` indicates, that processing more than $b - b_{min}$ slots would underrun the lower limit. Processing exactly that number leads to being in state `bottom`, when the next window is read. It would make sense to set $n = \max(b - b_{min}, m)$ here, but the simpler solution above was chosen, to save an additional comparator and multiplexer. This does not affect the principle of operation, because in all other cases n depends on m.

OCCMEM access The upper five bits from `next_read_pos` serve as address for the `OCCMEM` memory. In the `START_READ` state the output is saved in a register. In the next cycle, the state changes to `READ_CMPLT` and the address is changed. One cycle later, the output presents the result for the new address. For $n = 0, 1, 2$, according to the state diagram, the state will now be `START_READ` again and so the output will be saved again. It is registered at the earliest point of time possible in this situation, where $n = 0, 1, 2$. Also during `START_READ`, the window is erased by asserting the write enable signal of `OCCMEM`. The data input is wired to zero, as indicated by the ground symbol in Figure 5.2.

The output of the register is fed through asynchronous logic in the `Reduce_events` module back to the `colocc` port of the `Sorter`. It then goes asynchronously through `Bit_counter`, `calc_num_ev_proc` and the state transition logic. This is a rather long logical path and having a register after the memory output greatly improves timing, as the clock to output time for a register is much smaller than for a BlockRAM.

An example of the overall resulting timing can be seen in Figure 5.3.

Synchronous start The readout process has to be started synchronous to the *Spikey* system time. The start signal is given from the outside by raising `enable`. Because the output from `OCCMEM` is registered in the `START_READ` state, a valid address must be presented to the memory already in the `IDLE` state. Therefore `read_pos` is set to $\texttt{t\_chip} + b_{max}$ during this state. `next_read_pos` is set to $\texttt{t\_chip} + b_{max} + w$.

For correct processing it is necessary to start on a system time that is divisible by w. Otherwise, the correspondence between bits in `occ` and the reading position would be broken. The readout

FSM assumes, that bit i belongs to time stamp `read_pos` $+ i$. Since only the upper five bits of `read_pos` are used as address for `OCCMEM`, an additional offset would be introduced.

This condition has to be ensured outside of `Sorter` within the logic driving `enable`. That way, scalability is improved, because the logic is needed only once for multiple `Sorter` instances.

Readout state machine

Besides the control FSM a second state machine selects the actual event from memory. Its purpose is to present an address to `MEM` and to mark the validity of events, which were read from the memory. Its state `i_valid` is a bit-index into `colocc`. After reset and for every new window (when the control FSM is in `START_READ`) it starts at `i_valid` $= 0$. Then in every clock cycle, it advances to the next set bit.

The address to `MEM` is given by `read_pos` + `i_valid`. Because the collective occupation bits are used to calculate `i_valid`, these addresses not necessarily hold valid events. For this reason, the valid bit on the outgoing event port is set from the individual occupation bits in `occ`.

5.2 Integration into the FPGA

The design described in this thesis builds on existing work. It combines and extends two designs, one containing the *Spikey* controller and the other the gigabit network. Both have particular requirements on the limited dedicated resources for clock generation and distribution, which have to be conjoined into one utilization scheme. Also some modifications needed to be done inside the *Spikey* control logic to realize event streams from and to other *Nathan* modules.

5.2.1 Clocking

As described in Chapter 2 the *Backplane* provides a common clock for all *Nathan* modules, which themselves generate the clock signal for *Spikey*. The maximum frequency for the network is 156.25 MHz [Phi08a], which is also attainable for the *Spikey* ANNA [Grü07]. Therefore, this is also the targeted maximum clock frequency for this design.

The existing designs, which were using either the network or the *Spikey* controller, make different use of the clocking resources on the FPGA. These usage schemes have to be combined into a single one that fits into the device.

Required clock signals The different parts of the design have different requirements on clocking.

- The DDR-SDRAM controller logic needs two clock signals situated on the upper edge of the FPGA, which are 180° phase shifted to each other. The frequency may not exceed 140 MHz [Sch05]. Experiments later on have shown, that also a minimum of 60 MHz must be provided.

- The *SlowControl* ring clock is fed into the FPGA on the bottom edge via an external pin and must be distributed over a dedicated clock network inside the device. The *SlowControl* client logic derives a separate slower signal from the system clock to ease place and route.

- The MCGN needs a reference clock of up to 156.25 MHz for the MGTs and its interface. It implements a special feature to minimize delay, that requires one clock buffer multiplexer (BUFGMUX) [Phi08a].

- The clock input to *Spikey* is generated on the FPGA and multiplied inside the chip. Half the frequency of the *Spikey* slow clock has to be provided on a differential IOB. For the HyperTransport link, that operates at the fast *Spikey* frequency with DDR, three clock signals with 0°, 90° and 180° phase shift are needed [Grü07].

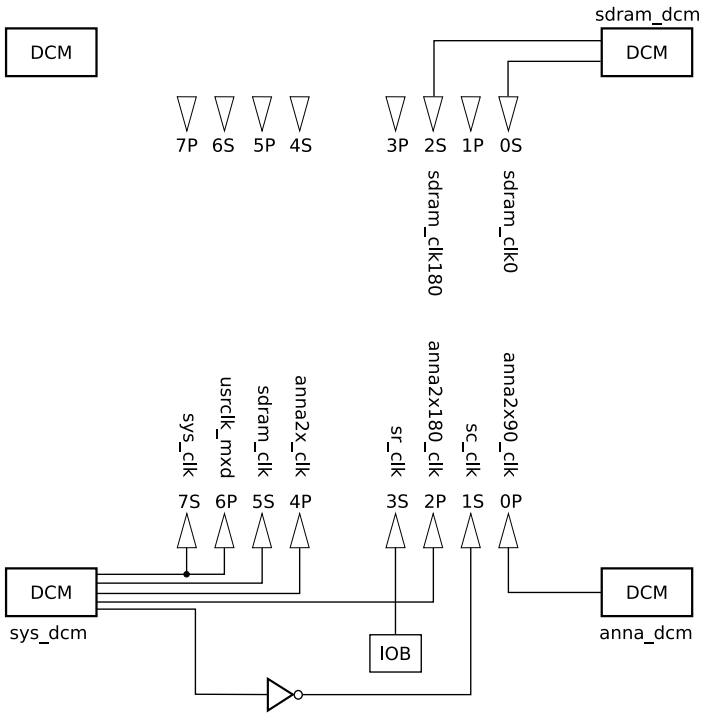

Figure 5.4: Shown is the utilization of DCM and BUFGMUX resources on the FPGA. The DCMs are located on the four corners of the chip and BUFGMUX are drawn as triangles, labeled 7P, 6S etc. The `sr_clk` is directly given on a clock buffer from an input/output block (IOB) connected to an external pin. For `sc_clk` general purpose routing lines are used, since for each DCM only four dedicated lines are available. The top left DCM is not used.

FPGA clocking resources The used FPGA device has four digital clock managers (DCM) and sixteen global clock multiplexer buffers (BUFGMUX), that drive sixteen clock domains from external or internal sources. The DCM uses a delay locked loop (DLL) to generate de-skewed clock signals with multiplied or divided frequencies and can dynamically do phase shifting. BUFGMUX are used to drive the dedicated clock networks and can be configured as simple buffers, possibly with an enable signal, or as multiplexers to dynamically switch between two clocks.

Various constraints for the usage of these components exist, because only limited routing resources dedicated to clock signals are available. The FPGA is divided into four quadrants, each having eight clock networks. Two BUFGMUX - designated as primary and secondary - share access lines to one quadrant and thus compete for access. In the multiplexer configuration, two neighboring buffers also share inputs. Using general purpose routing lines as inputs to clock buffers is possible, but must be done with great care, since they have higher and more variable delays. If more than eight clocks are used in a design, manual placement of the DCM and BUFGMUX components is usually required. Details can be found in [Xil02].

DCM and BUFGMUX usage Figure 5.4 illustrates the usage of DCM and BUFGMUX resources. The system DCM in the lower left corner reconstructs the external clock signal of the *Backplane* and provides it as the main clock `sys_clk` of the design to all quadrants. Buffer 7P may therefore not be used. `usrclk_mxd` interfaces the MGTs and is used for the min delay patch mentioned above. This buffer is used in a clock multiplexer configuration - with one input being ground - and shares its inputs with 7S. In the given configuration only `sys_clk` can be placed next to `userclk_mxd`, since only they have a common input.

Pins to the SDRAM module are situated at the top edge and thus clock buffers for the link are placed at the top, too. The `sdram_dcm` has `sdram_clk` as input and generates the phase shifted DDR clocks for the external module. Because the clock signal from the RAM chip is fed back to the DCM for de-skewing, a separate DCM is required.

The anna2x clocks are used for the *Spikey* interface DDR registers. The signal `anna2x_clk` and the 180° phase shifted version `anna2x180_clk` are provided by the system DCM on dedicated ports, that generate a doubled and inverted doubled signal respectively. Because of the unknown phase relation between FPGA and *Spikey*, it may be required to shift the sampling clock [Grü07]. This can be done by switching to the shifted `anna2x90_clk`, that is generated by `anna_dcm` from the doubled system clock. The DCM is configured as variable phase shifter, which can be programmed via *SlowControl*. The inverted clock signal `anna2x270_clk` is produced by local inversion clocking, i.e. no global clock network is used.

The *SlowControl* ring clock is provided via pin and directly routed to a clock buffer. It is only required by lower network levels and has therefore not to be routed to every quadrant. Unfortunately a buffer on the rather crowded bottom edge must be used, since the IOB is situated here. The system DCM feeds the `sc_clk` buffer over general purpose routing resources, because only four outputs of a DCM can use dedicated resources and `sc_clk` is in no particular phase relation to other clocks. It uses an output of the system DCM that provides a signal with a frequency divided by four.

Alternative configuration Early experiments have shown, that at least two of the used *Spikey* chips had difficulties at 156.25 MHz. For the test an existing FPGA configuration file with existing test software, that writes and reads data to the parameter RAM of *Spikey*, was used. It was therefore decided to implement an alternative clocking scheme, which can be selected prior to synthesis and runs at 100 MHz. At this frequency, the `sdram_clk` is operated at full system clock speed to stay above the minimum of 50 MHz.

5.2.2 Integration with the Spikey Controller

The implementation of the existing *Spikey* control logic is described in [Grü07]. An outline was given in 2.3.3. The only event source is the *PlaybackMemory*. With multi-chip network operation this has to be extended to include event streams coming over the network.

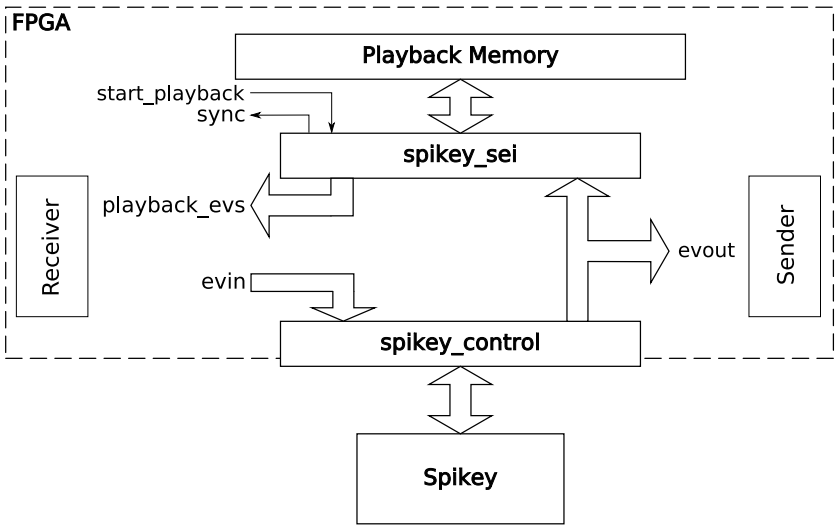

Figure 5.5: The altered spikey interface logic. The original can be seen in Figure 2.7. Event streams have been redirected to use `playback_evs`, `evin` and `evout` ports. The `start_playback` and `sync` control signals were added.

Figure 5.5 shows the modifications, that were done to the control logic. Several additional ports were introduced: The event stream from the *PlaybackMemory* is forwarded to the outside through `playback_evs`. It is fed into `Receiver`, to be merged with incoming streams (cf. Figure 3.5). The resulting stream is handed to `spikey_control` over `evin`. Events from spikey branch off to `Sender`, before they are stored in the *PlaybackMemory*.

Modified HyperTransport datapath The HyperTransport framing and deframing on the FPGA is done with twice the system clock frequency. The framing logic samples data to be transmitted with this speed and thus only half the system clock period minus setup time are left for routing. In case of a 156.25 MHz, this is less than 3.2 ns.

The design often failed static timing analysis due to one of these paths. To get around this, an additional register was placed in the datapath of framer and deframer logic, prolonging the required routing delay by a full system clock period. Since this increases link latency by one cycle per direction, the register can be disabled by VHDL generic parameter before synthesis.

Because synchronization of *Spikey* explicitly uses the link latencies for its calculations, it must be adapted to whether the register is used or not.

Additional control signals The control signals `start_playback` and `sync` allow control of *PlaybackMemory* processing from outside the controller logic. `start_playback` starts the SDRAM read request to fill the buffers. After a predetermined time `spikey_sei` starts processing of the buffer contents. Synchronization of *Spikey* is performed upon a command stored in the *PlaybackMemory* and `sync` indicates its completion.

PlaybackMemory buffers The *PlaybackMemory* is stored in the SDRAM of the *Nathan* board. The interface logic inside the FPGA uses FIFO buffers for reading and writing. They are required to cross the clock domain between the main FPGA clock and the SDRAM clock and to buffer data, while the SDRAM is in a refresh cycle[2]. The size of theses buffers was reduced from 2048 entries to 1024 to free additional BlockRAMs.

The sustainable data rate of the SDRAM is smaller than that of the *Spikey* interface. Both can transfer one 64 bit word per clock cycle, but even if operated at full system clock frequency,

[2]SDRAM is dynamic memory and requires periodic refresh cycles to keep its content.

Component	BlockRAM	LUT4	Flip-Flop
`Sender`	5	581	300
`Transmit_buffer`	1	197	83
`Send_switch`	0	132	83
Other	3	55	51
`Receiver`	10	868	596
`Sorter` (PBM)	2	165	94
`Receive_line`	4	310	235
`Sorter`	2	202	99
Other	0	83	32
Total	15	1449	896

Table 5.1: Resource consumption of individual components. The module hierarchy is indicated by indentation and *Other* refers to not explicitly named logic on the same level. The `Sorter` module labeled with (PBM) is the one attached to the *PlaybackMemory*. The given numbers assume the default configuration, where `Transmit_buffer` and `Receive_line` are instantiated twice.

the maximum data rate of the SDRAM is reduced by refresh cycles. Therefore experiments at high rates are limited by the size of available buffers.

5.2.3 Resource Requirements

This subsection gives an overview of the utilized FPGA resources in the final design. The given numbers are extracted after the synthesis and place and route steps for the default configuration. It includes *Spikey* controller, gigabit network and SDRAM controller, as well as `Sender` and `Receiver`. The MCGN switch is configured to use four of the eight available MGT links and provides two internal user ports. therefore `Sender` uses two `Transmit_buffer` and `Receiver` two `Receive_line` components. Of the debugging features only `Drop_counter` is enabled (cf. Section 5.4 for a discussion of debugging components)

This design takes up 96% (4732 of 4928) of logic slices and 88% (39 of 44) of BlockRAMs, as given by the place and route report.

Break-down of utilization numbers Table 5.1 shows individual numbers for some submodules. They were extracted from the final design with the `floorplanner` tool from Xilinx. The receiving side is the larger one and `Receive_line` the most complex component. Relevant for scalability are most of all `Transmit_buffer` and `Receive_line`, because they are instantiated once for every user port of the MCGN switch.

5.3 Running of Experiments

An experiment run with an artificial neural network distributed over multiple chips involves three steps: First the interconnecting network has to be synchronized (cf. Section 2.2.1), second network connections have to be configured and then all nodes have to be started synchronously. Synchronization and configuration of the MCGN switch are described in [Phi08a] and the remaining configuration is described in the next section. After that comes a description of the startup procedure.

5.3.1 Configuration by SlowControl

The cross-chip inter-neuron connections have to be configured by writing the lookup tables described in Section 5.1.1. This is done by write commands via the *SlowControl* network interface (cf. Section 2.1.3). To avoid having one client module per lookup table, a central programming component `Mem_prog` was designed.

Operation outline `Mem_prog` implements a *SlowControl* client module and forwards write requests to a programming bus, to which all lookup tables are connected. It acts as the single bus master and the clients only read from the bus. Bus signals are:

Name	*width*	*Description*
`sel_type`	3 bit	Selects lookup table by type
`sel_mem`	4 bit	Selects an individual memory of those selected by type
`baddr`	12 bit	Byte-address in LUT memory
`wen`	1 bit	Write enable
`data`	8 bit	Data to be written

A *SlowControl* write request contains the selection and address bits and a 32 bit data word. A finite state machine within `Mem_prog` serializes this word to the 8 bit programming bus line, while setting `baddr` to the respective associated byte addresses. The most significiant byte is written first to the highest byte address.

Client logic This generic behavior needs only minimal logic at most of the LUTs. The two small tables `Mfi_lut` and `Slot_lut` have entries smaller than 8 bit and therefore four entries can be programmed with one *SlowControl* command. `data`'s lower three respective four bits and `baddr` are connected directly to the memory port. Logic is required only to generate the write enable signal from the selected type and id. The same holds true for `Send_lut`, except that here only the lower 16 bit of the *SlowControl* command are serialized to the memory. Because the lookup port of the memory is configured 16 bit wide (cf. Section 5.1.1), one entry is programmed by one command.

Requiring additional care, is the case for `Recv_lut`, which has 1024 entries of 17 bit. It is not possible to configure the second port with a width of 8 bit and access the complete 17 kbit range [Xil07]. Instead the second port is also 17 bit wide and local logic assembles one word for storage. Data is transfered via the 8 bit bus in three clock cycles and then written to the memory in the third cycle.

The mapping of *SlowControl* address bits to programming bus signals is given as:

SlowControl module number:		5
Address-Bits	*Contents*	
31..19	zero	
18..16	`sel_type`	000 : `Send_lut`
		001 : `Mfi_lut`
		010 : `Slot_lut`
		011 : `Recv_lut`
		100 : reserved
		101 : reserved
		110 : reserved
		111 : `Ext_delay`
15..12	`sel_mem`	
11..0	`baddr`	

Bits 11..0 in the request are the base address for `baddr` on the bus. So data bits 31..24 are written to base address plus three, bits 23..16 to base address plus two and so forth.

Other configuration targets Besides lookup tables, other targets for configuration exist, that are also connected to the programming bus. In general there is at least the `Ext_delay` register, which stores the common upper nibble of delays (cf. Section 3.2.2). But also other targets for debugging purposes may exists, depending on the synthesized features.

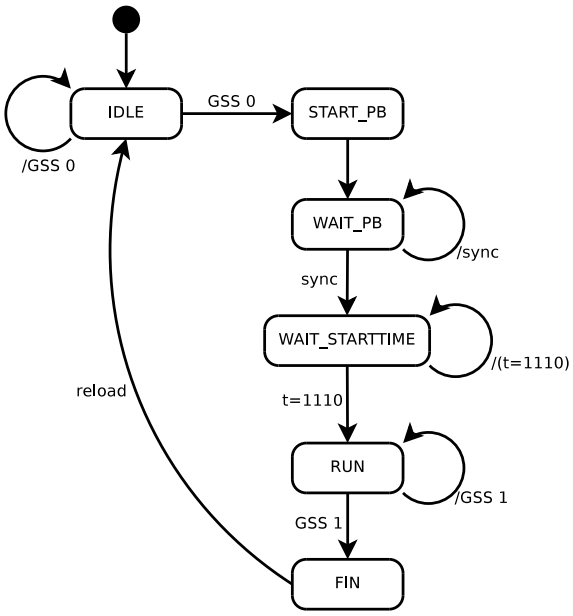

Figure 5.6: State diagram for the global control state machine. Transitions are labeled with the condition on which they are taken and "/" indicates negation. `GSS 0` and `GSS 1` are the global synchronous signals described in Section 2.2.3. `sync` is the identically named signal from the *Spikey* controller. `t` are the lower four bits of the system time counter and `reload` can be used to return to the reset state `IDLE`.

Write-only memory It should be stated explicitly, that, to the user, `Mem_prog` provides write-only memory. Read requests to the module aren't processed at all. While it would be nice to verify the contents of the LUTs, write-only allows an efficient implementation of the bus. Signals can be routed in a chain from `Mem_prog` to the attached LUTs. If the clients would also have to be able to write to the bus, one would either have to use expensive tristate buffers[3] or a star-like point to point topology.

5.3.2 Starting procedure

A synchronous start of an experiment distributed over multiple chips is coordinated by the node control state machine `NCS`, which is situated on every *Nathan* network module. It uses the GSS mechanism of the MCGN network (cf. Section 2.2.3) to start the *Spikey* controller to read the *PlaybackMemory* and enable processing of network events.

Its state diagram is shown in Figure 5.6. After reset the FSM is in the `IDLE` state. When the global synchronous signal `GSS 0` is raised, the experiment starting procedure is begun. In the `START_PB` state, the `start_playback` signal to the *Spikey* controller is set (cf. Section 5.2.2) to initiate *PlaybackMemory* readout. During the immediately following `WAIT_PB` state, `spikey_sei` signals a read request to the SDRAM client module, which will start to fill its FIFO buffer with the contents of the *PlaybackMemory*. After a given time it is assumed, that the buffer is full enough and the playback commands are read from it and processed. Among them must be a synchronization command, that will initiate the synchronization of the *Spikey* time counters. When this is done, `sync` is asserted and `NCS` enters the `WAIT_STARTTIME` state.

Section 5.1.4 stated, that `Sorter` requires to be started synchronized to the system time counter. Therefore, in `WAIT_STARTTIME`, it is waited until the system times lower nibble is 1110. In the next state the `enable` signal to `Sorter` is raised, so that in the next clock cycle it will start on

[3]The FPGA has 2464 tristate buffers compared to 9856 flip-flops

a system time divisible by sixteen. In the default configuration w is eight and so the requirement for a correct synchronous start of `Sorter` is fulfilled.

The end of an experiment is signaled by a second GSS signal, upon which the state machine enters the `FIN` state, in which it remains until it actively set to `IDLE` again by the `reload` signal.

SlowControl interface The GSS signals are raised from the outside by *SlowControl* write requests to any of the network modules. To prevent spurious GSS events, when the network is not yet synchronized, GSS processing must be enabled for each node beforehand. The *SlowControl* addresses to write to are:

SlowControl module number:		5
Address-Bits	*Contents*	
31	one	
30..4	zero	
3..0	`op`	0000 : raise `GSS 0`
		0001 : raise `GSS 1`
		0010 : set `reload` to `NCS`
		0100 : enable processing of both GSS signals

5.4 Debugging Features

In the early implementation phase, it is of benefit to have facilities in the design, that allow to monitor ongoing transactions. For example one wants to protocol events at multiple points in the design, to track their path. It is also advantageous to have dummy components to substitute more complex structures. These could be simple random event sources or similar. Another feature, which is also of interest in the later lifetime, is to detect and count when and where events are deliberately dropped.

For the aforementioned purposes components were designed, which can optionally be included into the design. `Event_logger` can log event streams on several channels. `Drop_counter` has multiple counters to record when and where events are discarded. As test event sources `Single_ev_src` and `Fake_spikey` can be used.

5.4.1 Logging of Event-Streams

The event logging module `Event_logger` has three channels, which can be individually selected for synthesis: The `evin` and `evout` channels are three events wide each and directly compatible to the *Spikey* interface ports. In fact the primary purpose of the component is to monitor input and output of `Sender` and `Receiver`. A third, single event channel `debug` can be enabled to record streams from various taps in the datapath. All channels can also use additional memories to store the full 32 bit wide system time along with events.

For each of them, dual-port BlockRAMs with one read-only and one write-only port serve as memory. One stores events and the lower byte of the system time, a second, optional one the time of the clock cycle in which valid is asserted. Their ports are configured as 512×32 bit and thus can store 512 events. One entry contains:

Bits	*Name*	*Description*
31..24	`timestamp`	Lower 8 bit of the time stamp
23..20	`timebin`	Event time bin
19..17	zero	
16..8	`address`	Neuron or synapse driver address
7..0	`systime`	Lower byte of the system time

A 9 bit counter points at the current write address and is incremented on every new event.

The recorded data is available to the user via a separate *SlowControl* client. Counters and memories can be read, a write command resets the counters to zero. The address bits of a *SlowControl* read request are interpreted in the following way:

SlowControl Module Number:		6
Bits	*Name*	*Description*
31..30	zero	
29	`sel_counters`	Set to one to read counters
28..27	`sel_channel`	00 : `evin`, 10 : `evout`, 01 : `debug`
26..9	zero	
8..0	`addr`	Address in memory

BlockRAM consumption Each event stream is stored in one or two RAMs, depending on whether full system time logging is enabled or not. `evin` and `evout` have three parallel streams each, and so they require three or six memories. `debug` takes one or two RAMs, leading to a total memory consumption between one and fourteen BlockRAMs.

5.4.2 Counting Discarded Events

Events can be dropped at several points in the datapath, of which some are monitored and the number of events dropped is logged. The aim of `Drop_counter` is to give an approximate view of where events are discarded, so that bottlenecks can be identified. Detecting the loss of all events is for some cases impossible or would require tremendous effort and resources. Let me give two examples to illustrate this: First, events injected into the network may be routed nowhere, since the MCGN switch was not configured to forward this particular slot. This situation can not be detected inside FPGA logic. Second, the `Sorter` may not read some valid slots in its memory due to the readout algorithm detailed in 5.1.4. Logging these drops would require a more complicated counting interface and some calculation logic or a buffer to defer incrementation.

In the default configuration there are seven counters, which are 16 bit wide. Both numbers can be configured prior to synthesis. Monitored drop points are (the given counter numbers refer to the default configuration with two user ports):

- In `Send_switch`, when two input ports simultaneously request the same output port and the FIFO of the port with lower priority is full. There is only one counter for `Send_switch` (Counter no. 0).
- The `Transmit_buffer` can overflow, when the requested queue inside the `MultiFIFO` is full. There is one counter for every `Transmit_buffer` (Counter no. 1 and 2).
- For each `Recv_queue` there is one counter, since new events are dropped, when the FIFO is full (Counter no. 4 and 6).
- `Sorter` uses one counter for the drop condition discussed in Section 3.3.3 (Counter no. 5 and 7).

For every counter exists a 1 bit increment signal, which is set, when an event is dropped. Local logic at the monitored points has to generate this signal.

Read access to the counter values is provided by a shared *SlowControl* client module. The interface specification is:

SlowControl module number:	5	
Address-Bits	*Contents*	
	Read access	
31	one	
30..20	zero	
19..16	`op`	0001 : read drop counter
2..0	`sel`	select the drop counter by number
	Write access	
31	one	
30..4	zero	
3..0	`op`	0101 : clear drop counters

The width of the `sel` field depends on the number of actually used counters. If more than eight are used, more bits will be used. On a clear request, counter will only be reset to zero, for which a bit was set in the data word. So to clear counter two only, the write request must contain a data word, where only bit two is set.

5.4.3 Event Sources for Testing

Two dummy event sources are available to substitute the *Spikey* controller in test setups without neural network ASIC. `Single_ev_src` produces exactly one event immediately after its enable signal is asserted. Address and packet slot of this event can be programmed with the `Mem_prog` infrastructure (cf. Section 5.3.1). To write the address, set `sel_type` to 101, and to 110, to write the packet slot.

`Fake_spikey` is a random event source. It uses linear feedback shift registers to generate events with random timing and addresses.

Both sources also provide a 32 bit wide system time counter, which is normally implemented by the *Spikey* controller. This allows operation without any *Spikey* related logic.

5.5 Development of Supporting Software

To ease the handling of the system, some supporting software tools were written, that use the *SlowControl* to communicate with the hardware. The `evn-prog` program can be used to interactively program the lookup tables. `evn-list` reads events recorded by `Event_logger` and presents them in readable form to the user and `evn-drop` accesses the drop counters.

Beyond that, the existing test software for the *Spikey* chip `createtb` [Grü07] was modified and extended to perform multi-chip tests with it.

All software is written in the C++ programming language.

5.5.1 Hardware Access Framework

The hardware access framework is a set of C++ classes to provide high level functionality to the programs mentioned above. Theses classes abstract *SlowControl* communication and contain methods tailored to their associated hardware modules.

SlowControl library A C library provides low-level methods to read from and write to individual *SlowControl* client modules on *Nathan*. Before they can be used, an open function must be called, which returns a handle to be used in read and write requests. A close function invalidates this handle.

Class hierarchy Figure 5.7 shows the class hierarchy of the framework. The primary base class `Slow_base` stores the handle and implements opening and closing. Derived from it the `Mem_prog` class is intended to utilize the hardware module of the same name described in Section 5.3.1.

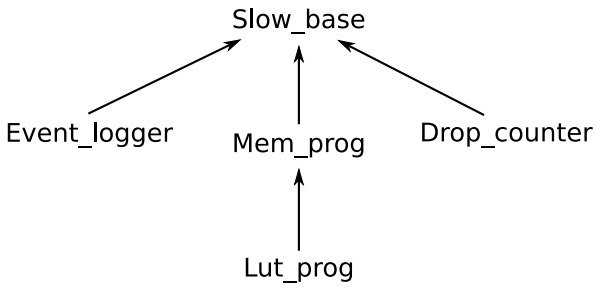

Figure 5.7: Class hierarchy of the hardware access framework. Arrows point from derived to base class.

It features a method to write data to specific programming bus targets. Its specialization, the `Lut_prog` class uses this method to provide programming functions for each programming target. So a LUT entry can directly be programmed with a call to one of these functions.

`Event_logger` is another specialization of `Slow_base` and provides functions to read counters and events from the hardware module of the same name. Similarly, `Drop_counter` implements methods to read and reset drop counters.

5.5.2 Modifications to Spikey Test Software

Experiments are performed by a modified version of the `createtb` program, which was previously used for single-chip testing. It contains the hardware abstraction layer for the *Spikey* chip and communicates by *SlowControl*. A description can be found in [Grü07].

The program contains several test modes, which perform different tests on the hardware. To evaluate the hardware design and perform multi-chip experiments, new test modes were developed. Chapter 6 describes what they do. Other modifications concern experiment workflow and the integration of the hardware access framework (cf. Section 5.5.1).

Start of playback In the original version `createtb` has a function to send a spike train to *Spikey*. It transfers event data to the SDRAM via *SlowControl* and also automatically starts the readout process on the FPGA. For multi-chip operation, these steps are separated. First, event data is distributed to one or more nodes and then, readout is started globally by raising a GSS event.

A mode variable was added to the hardware abstraction class `Spikey`, that selects whether the `sendSpikeTrain` member function should also start the experiment or only transfer data.

Packet slot utilization As described in Section 5.2.2, `Receiver` combines event streams from the *PlaybackMemory* and the network. In memory, events are coded in 64 bit words, as they are on the *Spikey* interface. That implies, that three events can be stored in parallel. In the modified design by default only one of these can be passed to the chip, since the others are reserved for network traffic. The `sendSpikeTrain` function is extended to reflect this and store only one event per 64 bit word.

The packing algorithm in use would store two events with consecutive time stamps in the same packet. When it is allowed only one event per packet, it will not transmit the second one at all. This behavior is sensible regarding the available bandwidth of the *Spikey* link. At maximum a rate of 0.5 events/time stamp can be sustained on a single slot. However, it should be possible to alter the algorithm to allow bursts at higher rates for a limited time. A readout behavior similar to the one of `Sorter` (cf. Section 3.3.3) could be implemented completely in software.

Chapter 6

Experiments and Performance Analysis

Experiments described in this chapter serve two purposes: First to test whether the whole stage-1 system behaves as expected and can be used for multi-chip neural network experiments. And second to evaluate the performance of the inter-chip network. The relevant measures for the latter are latency and throughput.

Basic test setup Most of the experiments use the same basic setup: *Spikey* is configured to use the event loopback module instead of the real analog part. That way, events that would normally be transmitted to an analog synapse driver are transferred directly to the output buffers behind the neuron circuits. They return to *Nathan* with a time stamp increased by 10 and the synapse driver number in the neuron address field.

To simulate *Spikey* activity, events are written to the *PlaybackMemory* and transferred to the chip after the experiment has started. The loopback module returns them to the FPGA, where they appear the same way as events generated by the analog part would. The `Sender` component can now transport them to another *Nathan*, where `Receiver` delivers them to its own *Spikey* chip. The loopback module there returns the events to the FPGA, where they are stored in the *PlaybackMemory* for later readout.

This setup allows to deterministically generate test activity patterns without uncertainties. This could not be guaranteed, when using the analog network blocks. To verify the result of a test mode, the spike trains read back from each module can be evaluated.

Figure 6.1 shows which *Nathan* modules are used and how they are interconnected.

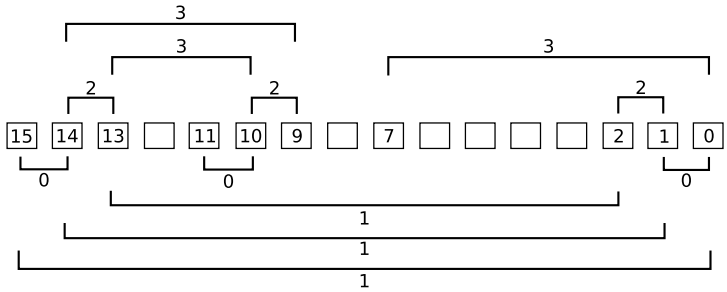

Figure 6.1: Topology of the used *Nathan* modules on the *Backplane*. Boxes indicate the available slots on the *Backplane*, where only those filled with a number are equipped with a *Nathan* board. The connections between nodes are labeled with the MGT numbers. The same number is used on both ends of a link.

System under test All tests described in this chapter use the final *Nathan* FPGA design in default configuration with four MGTs and two user ports. The 100 MHz oscillator of the *Backplane* is used as system clock. The 156.25 MHz clock is not used, because it causes high error rates on the *Spikey* HyperTransport links, as discussed in Section 5.2.1. As a consequence the FPGA uses the alternative clocking configuration with full SDRAM frequency. The used *Spikey* chips belong to revision two and three.

6.1 Validation of Correctness

The first experiments are to check if the system works as intended. This is the case, when events are transported to the correct destination in space and time. Here space means *Nathan* module and synapse driver address and the timebase is the *Spikey* system time counter. Furthermore it is checked, whether the constraints discussed in 3.1.1 are met, i.e. if events are transmitted in order of their time stamps and event buffer collisions are prevented.

6.1.1 Correctness of the Routing Mechanism

This experiment tries to deliver a single event from one *Spikey* to another to validate correctness in space. It repeatedly configures a random route, i.e. source neuron, routing parameters (`subnr`, `mfi`, etc.) and destination synapse driver. Then it generates a spike train containing the event to be routed followed by random noise. The noise events are picked from all possible neurons on the source chip, except for the test neuron. The test mode programs the lookup tables to drop all events, that come from "noise neurons". The experiment is successful, when only the chosen event is present in the recorded spike train of the destination chip.

In more detail, the stimulus spike train has the following pattern: The test event is the first one in the stimulus spike train and followed by noise. First comes a regular part, where events are placed 10 time stamps apart and all neuron numbers – except for the tested one – are swept through. After that 1000 events with random addresses – again except for the tested one – follow at random intervals between one and eleven time stamps.

That way, it is validated, that events from any source are delivered correctly, that arbitrary routing parameters can be used and that other events are not transported. The rate of the source spike train is kept low, with regular intervals of 10 time stamps between events, to prevent loss caused by bandwidth limitations in the gigabit network.

Parameters This test mode has three selectable parameters:

- The source and target *Nathan* numbers N_{src} and N_{dest}.
- The number n_i of iterations.

Performed experiments This test mode was used throughout the implementation phase to verify design changes. The final implementation succeeds reproducibly, for example with $n_i = 1 \times 10^6$, $N_{src} = 0$ and $N_{dest} = 1$. With these parameters, the test lasts nearly 67 hours, during which one million events are transported without errors. The experiment demonstrates, that all routing parameters can be used in arbitrary combinations. The deliberate rejection of individual neuron addresses is verified with billions of events.

6.1.2 Merging of Event Streams

A major task of `Receiver` is to merge incoming event streams and send them ordered by time stamp to *Spikey* (cf. Section 3.1.1). The purpose of this experiment is to demonstrate this ability. It establishes a situation that makes unordered arrival of events likely by configuring two source neurons for one target synapse driver. For both sources random spike trains are generated by poisson processes [Dow08]. After the experiment run, the test mode iterates over the events

No.	$N_{src,0}$	$N_{src,1}$	N_{dest}	R[events/time stamp]	n_i	n_e	Comment
(0)	1	7	0	0.01*	1000	30000	passed
(1)	0	2	1	0.15*	100	64	passed
(2)	0	2	1	0.30*	100	64	passed
(3)	0	2	1	0.40*	15	64	1 run failed
(4)	0	2	1	1.0*	7	64	1 run failed
(5)	0	2	1	1.0	100	64	passed
(6)	0	2	1	0.15*	4	1000	$D_{ext} = 2$, $D \in [0, 255]$
(7)	0	2	1	0.15*	100	1000	$D_{ext} = 1$, $D \in [32, 255]$

Table 6.1: Parameters and results of the performed merging experiments. Rates labeled with an asterisk (*) indicate, that events were not allowed to have consecutive time stamps. Experiments (3) and (4) were aborted after the erroneous run to inspect the cause of the error manually. The inspection - discussed in the main text - however showed failures to be not caused by unordered transmission of events.

recorded at the destination and tries to match each of them to one from the sources. The matching is done by subtracting the expected delay from the target time stamp and searching an event with this value as source time stamp. If events were not transmitted to *Spikey* in correct order, a deadlock would occur and at least one event would take 256 time stamps longer through the loopback than expected.

Besides this indirect approach the test software also searches directly for deadlocks in the target spike train. For rates above 0.05 events/timestamp, gaps of more than 128 time stamps between target events are considered as deadlocks of length 256. For rates above 0.01 events/timestamp, deadlocks of length 4096 are identified by detecting gaps of more than 2048 time stamps. The rate limits are necessary to ensure, that the respective gaps are truly deadlocks and not allowed intervals. Below these rates, only the matching described previously is used to detect erroneous events, while above both methods are used simultaneously. The matching approach is sensitive to a wide range of errors, while the gap analysis specifically finds deadlocks.

The procedure is repeated in multiple iterations, and in every single one the test mode configures a new randomly selected network configuration. These are source and destination addresses and routing parameters like `subnr`, `mfi` and `lvc`.

Parameters The test mode has six parameters:

- Two source *Nathan* numbers $N_{src,0}$ and $N_{src,1}$.
- The destination *Nathan* number N_{dest}.
- The mean firing rate R for each source spike trains.
- The number of experiment repetitions n_i.
- The number of iterations n_e to generate events in each repetition. Roughly the total number of events.

Also for some experiments the peak event rate at the source node was capped by preventing events with consecutive time stamps. The modified `createtb` program is not able to place such events into the *PlaybackMemory* (cf. Section 5.5.2). Thereby the peak rate is limited to 0.5 events/timestamp.

n_e is the number of iterations in which one or two events are randomly generated. Depending on the selected rate, the number of events is larger than n_e.

Performed Experiments

Table 6.1 shows an excerpt of used parameter sets. Capping of peak rates was in use for most of the experiments. When an error occurred during one run, abort was triggered to allow a manual examination.

Event output buffer delays Tests, especially at high rates, are sensitive to multiple failure modes apart from unordered transmission. In experiments (3) and (4) events at the end of the target spike train had no matching source events. Using `Event_logger` to analyze the event stream from `Receiver` before transmission to *Spikey*, events were found to be passed in correct order and sent as expected. The events in question had unexpected values for bits 11 to 8 of the time field. Those are the bits inferred by the FPGA logic to supplement the 8 bit time value provided by *Spikey*. The wrong upper bits in the recorded stream are probably caused by a full event output buffer on the chip (cf. Section 2.3.1). These FIFOs are 128 entries deep and get filled during a run, because spike rates were above the sustainable transfer rates of the chip interface. The propagation time through a full FIFO is equivalent to 256 time stamps. The time from analog part - or loopback module in the case at hand - to `Sender` is even larger, as further pipeline stages are passed. The FPGA logic has no means to detect from which period events originate. So a nearly full event output buffer results in events being assigned to the wrong period.

This problem was already identified in the conceptual development phase by simulation. Hence it was possible to include a solution in the fourth revision of the *Spikey* ASIC, by making the depth of the output FIFOs programmable. However, a chip of this version was not yet available for testing.

The parameter $n_e = 64$ is chosen for experiments with high rates to minimize this effect. Since the number of events may be higher than 64, as commented on the parameter n_e above, the FIFO can however still grow too full.

During experiments in which peak rates were not capped, e.g. (5), this failure mode was not observed. With $R = 1.0$ an event is generated for every time stamp, but the software is only capable of sending every second one to the *PlaybackMemory*. Therefore the total event number is far below 64 and the FIFO does not grow too full.

Low-rate experiments For example experiment (0) was performed with a low rate of 1 event every 100 time stamps on average per source. The rate is high enough to make deadlocks on *Spikey* possible, since statistically four events should interact within 256 time stamps. By using very long spike trains ($n_e = 30000$), many patterns of event arrival at the destination occur without the problems discussed in the previous paragraph.

Because of the limited SDRAM bandwidth, it is difficult to run experiments with high rates and large n_e. For rates above approximately 0.1 events/time stamp, `createtb` gives an error message to indicate, that the read buffers of the *PlaybackMemory* ran empty, before the experiment was finished. In these cases n_e must be selected to match the buffer depth, so that the complete data for one run is available in BlockRAM FIFOs.

Sorter deadlocks During early experiments with a preliminary design, very long gaps of about 4096 time stamps were observed at the destination node. `Recv_queue` buffers incoming events and presents them in order of arrival to `Sorter`. The latter accepts an event into its memory not until its target time stamp allows to store it in the sorting memory. That is the case, when it is less than 256 time stamps greater than the current reading position (cf. Section 3.3.3). Now consider two consecutive events in `Recv_queue`, which are 256 time stamps apart. When the event with the greater time stamp is ahead of the other in the queue, the reading position will be equal to the time stamp of the other event, when the first one is accepted by `Sorter`. Consequently it arrives too late to be send to *Spikey* in time. Accepting it, would result in a deadlock, since it would be delivered after the next wrap-around of the 12 bit reading position.

As a consequence the logic of `Sorter` was extended to check whether an incoming time stamp is more than 2048 time stamps in the future. In this case, it must be dropped to prevent the deadlock. This drop condition was already described in Section 3.3.3 and is included in the final design. The deadlock was not observed since then.

Influence of the extended delay Large extended delays lead to long waiting times in `Recv_queue` to compensate the difference between real link latency and demanded delay. When merging event

streams with different programmed delays, that are also subject to jitter (cf. Section 3.1.1), events with time stamps over a wide range may be present in `Recv_queue` at about the same time. Because `Sorter` only covers a range of 256 time stamps, events arriving in the wrong order and being apart more than that are lost, as a consequence of the drop condition discussed in the previous paragraph.

Large extended delays increase the probability of one event blocking a number of others inside `Recv_queue`, since they have to wait, even if the buffer is empty. This can lead to losing a large number of following entries and results in a longer period without events in the target spike train.

This was observed in experiment (6). Experiment (7), for which only the delay range was changed, does not show such gaps.

Single bit errors This test mode is also sensitive to single bit errors in events transmitted to *Spikey*. When the error affects the time stamp, unmatched events are detected. By listing source events which were not associated with any target event, a candidate can be found, that differs only in one bit from the expected event. When the address is affected, events are dropped at the source node. Yet, the corrupted events are visible in the recorded spike train. They appear as events from other, unused neuron addresses.

Single bit errors were observed during some experiments, for which accidentally an FPGA configuration with timing errors was used. After solving the timing violations for the final design, bit errors were not detected again.

6.2 Performance Measurements

The previous section described experiments, that build a basic confidence in the implemented design. This section will continue with more thorough tests, which also allow to characterize the performance of the network.

6.2.1 Latency

Key to a successful multi-chip operation is a low-latency interconnecting network. This experiment therefore serves two purposes: To measure the minimal achievable transmission time and to verify a correct delivery in time without jitter.

The test mode iterates over a wide range of programmed delays D. For each value of D it performs a number of runs. Each run generates a spike train of several hundred events with fixed intervals of 256. The time stamp of the first event is shifted by one before each run. For the next value of D, it is reseted to an initial value T_0. Because of the fixed interval of 256, in one run the events have identical 8 bit time stamps and only the time spent waiting on a valid MCGN slot is different.

As an example, the test mode starts with $D = D_0$. The spike train generated in the first run begins with event time stamps

$$T_0, \quad T_0 + 256, \quad T_0 + 512, \quad ...$$

In the next run the spike train begins with

$$T_0 + 1, \quad T_0 + 1 + 256, \quad T_0 + 1 + 512, \quad ...$$

This goes on for a predefined number of runs until D is incremented to $D = D_0 + 1$. The next run will then again start with

$$T_0, \quad T_0 + 256, \quad T_0 + 512, \quad ...$$

In summary, this iteration scheme allows to test a range of programmed delay values with all possible time stamps, while also varying timing relations to the MCGN. That makes it possible to cover a large space of possible timing combinations.

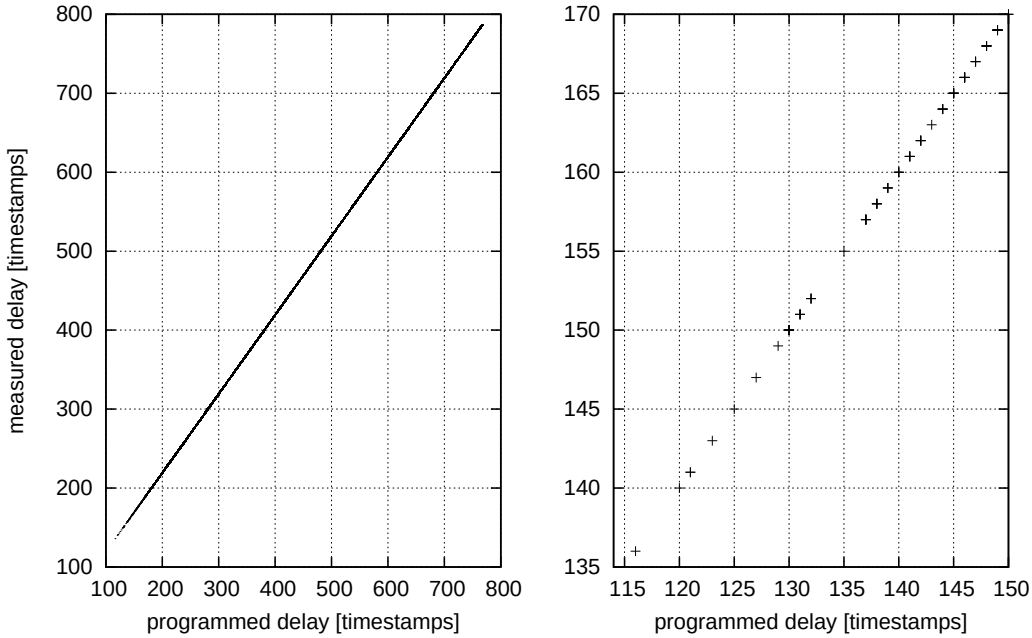

Figure 6.2: Measured delays over programmed delays. The left picture is a composition from runs over one, two, three and four hops. The right is a detail of the left, in that individual points are discernible. Each point is produced by one run. So for one programmed delay value, multiple runs with variated event timing are plotted. Points for one programmed delay are atop each other, indicating, that delivery is free of jitter. For small delay values, no events arrive, and so no points are drawn.

Resulting spike trains are analyzed as described in Section 6.1.2: For every arriving event a matching source event is searched, based on the expected delay. Besides this automatic self-check of the experiment, the measured delay of every single run is plotted against its programmed delay. This allows to graphically check for correctness. To find the minimal possible delay, a histogram is plotted, showing the number of arriving events for every measured delay value.

Source and target addresses, as well as routing parameters are again randomly chosen for every run. Experiments are executed over different physical distances by routing MCGN time slots over one or more intermediate nodes. The pass from one node to the next is called a *hop*.

Parameters The test mode has seven parameters:

- The source and target *Nathan* numbers N_{src} and N_{dest}.
- The range of programmed delay values to test as given by its lower and upper bounds D_0 and D_1.
- A step size for the delay values ΔD.
- A number of runs n_i per delay value.
- A number of events n_e per run.

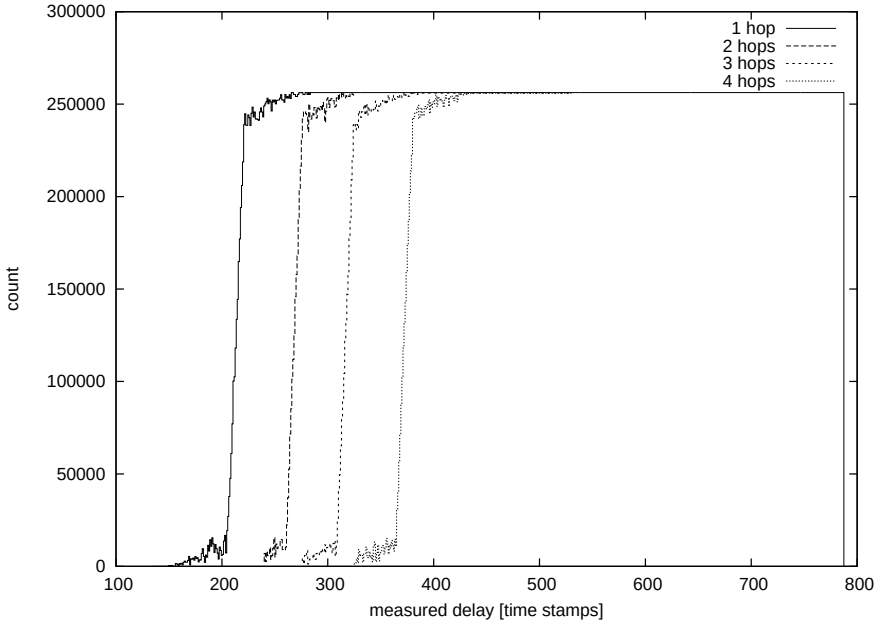

Figure 6.3: Latency measurements over multiple distances. Plotted is the histogram of measured delays for experiments using connections over a variable number of hops. For multi-hop connections, measurements covered only the transitional range, characterized by the sharp edge.

Performed experiments Figures 6.2 and 6.3 show results of performed experiments. The tested range covers delay values from $D_0 = 0$ to $D_1 = 768$ with a stepping of $\Delta D = 1$. For each delay value $n_i = 256$ runs with $n_e = 1001$, for the single-hop measurement, and $n_e = 1000$, for multi-hop measurements, were performed[1].

Figure 6.2 shows the delays measured in each run. All points for one delay value are atop each other, which shows that they arrived without jitter. They form a line with slope one, showing a linear relation between programmed and measured latency. The offset of 20 time stamps is introduced by the loopback module on *Spikey* (cf. Section 2.3.2). The event passes the loopback twice: First on the source chip, then at the destination.

Figure 6.3 shows the number of arrived events for each measured delay value in multiple experiments. For low latencies, no events arrive at all. For high latencies, a plateau is reached, where all events arrive. In between, the rate slowly increases over a range where some events arrive, then, after an abrupt increase, the rate transitions to a range where most events arrive. For a single hop, the edge is centered at time stamp 213, the transition to low delays extends for about 60 time stamps and for 70 to higher delays. For two, three and four hops, the pattern is shifted to higher delays, since the physical transmission time is longer. Overall, the minimum delay of safe transfer is 284, or $D_{ext} = 1$ and $D = 28$. Lower delays can be achieved, when some loss of events is acceptable.

Table 6.2 gives an overview of safe minimal delays for all possible distances on a single *Backplane*. The two rightmost columns show a comparison to the biological time domain for two cases: The first case represents the system under test, in which *Spikey* is clocked at 200 MHz (cf. Section 5.2.1) and has a speed-up of 10^5. The second case is for the next *Spikey* revision, which is not yet available for testing. It is designed to have a speed-up of 10^4 and should be able to operate at 312 MHz.

By design, the maximum programmable delay is 2048 time stamps, which is selected by $D_{ext} = 8$ and $D = 0$ (cf. Section 3.3.3). For a speedup of 10^5 and a 200 MHz clock, this is equivalent to 1024 ms and for 10^4 and 312 MHz to 65.6 ms.

[1] The difference in n_e happened accidentally and has no special meaning. n_e can be chosen arbitrarily.

Distance	**Time stamps**	**200 MHz,** 10^5	**312 MHz,** 10^4
1 hop	284	142.0 ms	9.1 ms
2 hops	341	170.5 ms	10.9 ms
3 hops	387	193.5 ms	12.4 ms
4 hops	439	219.5 ms	14.1 ms

Table 6.2: Minimal achievable safe latencies for multiple distances. The values in the time stamps column are taken from Figure 6.3. The two columns to the right compare them to the biological time domain for two different setups. The left one is for the actually tested system and the right one should be possible with revision four of the *Spikey* chip, which will be available shortly.

6.2.2 Transfer Rates

The purpose of the next experiment is to measure what throughput can be sustained for given event source rates. In general, the total bandwidth of the gigabit network is lower than that of the interface to *Spikey* (this was discussed in Section 3.1.2). Hence, for high rates loss of events is expected. The design contains several points, where events can be dropped for various reasons. In this experiment the number of drops is measured by comparing the recorded spike train against the stimulus train.

The test mode iterates over a range of test rates, multiplying the rate by a constant factor at every step. In one iteration, multiple runs with random poisson spike trains are performed. Each train uses randomly chosen source and target addresses and routing parameters. The event packing algorithm implemented in the `sendSpikeTrain` function of the hardware abstraction layer removes source events, if data rates are too high. Also, like for the experiment described in Section 6.1.2, the source rate is limited to 0.5 events/timestamp by preventing events with consecutive time stamps. Therefore, the program determines the actual rates at source and destination by counting the numbers of actually transmitted events. It also checks, that only events with the programmed target address arrive at the destination.

Based on the data gathered in one iteration, the mean transfer rate and standard deviation are calculated for each test rate. The final result is the achieved ratio of delivery rate to source rate.

Parameters This test mode has seven parameters:

- The source and target *Nathan* numbers N_{src} and N_{dest}.
- The test rate R_0 to begin at.
- The multiplication factor f for every iteration.
- The number of iteration steps n_i. The final rate is $R = R_0 \cdot f^{n_i - 1}$.
- The number of runs per test rate n_j.
- The number of events per spike train n_e.

Performed experiments Figure 6.4 and 6.5 show the combined results of multiple experiments. The test connection used twelve of twelve time slots of a single MGT link between $N_{src} = 0$ and $N_{dest} = 1$. The covered test rates range from 0.001 events/timestamp to 1 events/timestamp. For the lower rates $n_e = 10000$ was used (Figure 6.4). For high rates, the SDRAM bandwidth limits the length of a spike train to below 1024 (cf. Section 5.2.2). Variable lengths below this limit were used to analyze their effect on the transfer rate (Figure 6.5).

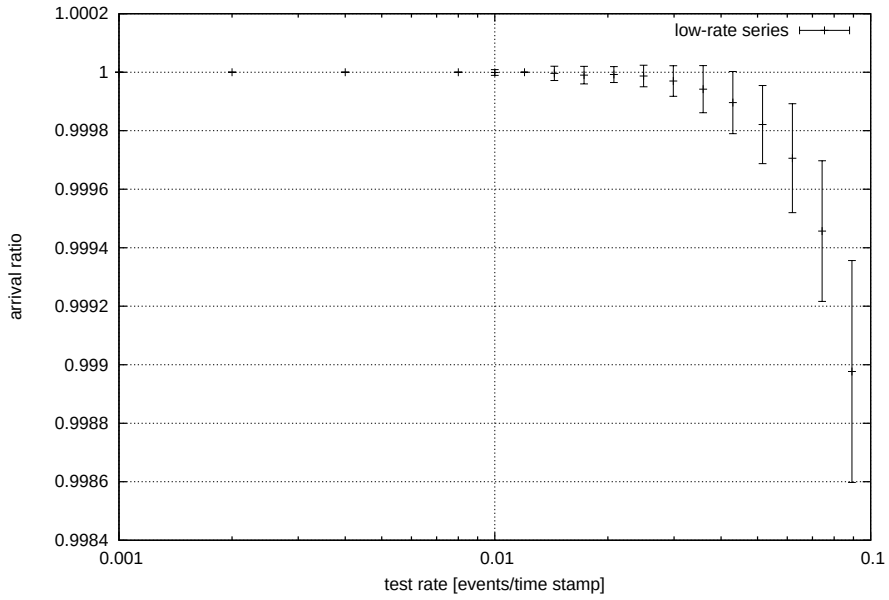

Figure 6.4: Comparison of experiments with low event rates. The ratio of delivered events to source events, is plotted against the test rate. For test rates above 0.01 events/time stamp, error bars indicate the standard deviation for 100 iterations ($n_j = 100$), below only ten iterations were used. In each iteration 10000 events were send ($n_e = 10000$).

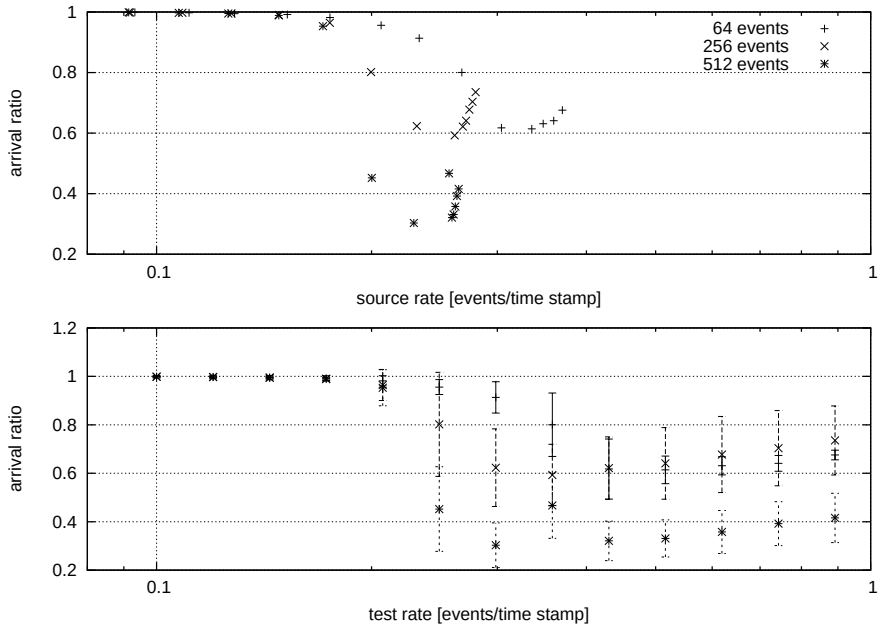

Figure 6.5: Comparison of experiments with high event rates. Both plots show the same experiments. In the upper plot the source rate is used as x-axis, i.e. the rate of events actually transmitted from the *PlaybackMemory* to *Spikey*. In the lower picture the test rate is used instead, i.e. the rate of event generation in the software. Both plots show the ratio of delivered events to source events and the standard deviation for 100 iterations ($n_j = 100$). Three experiments with different lengths of spike trains are shown. Error bars in the upper plot are not drawn for reasons of clarity.

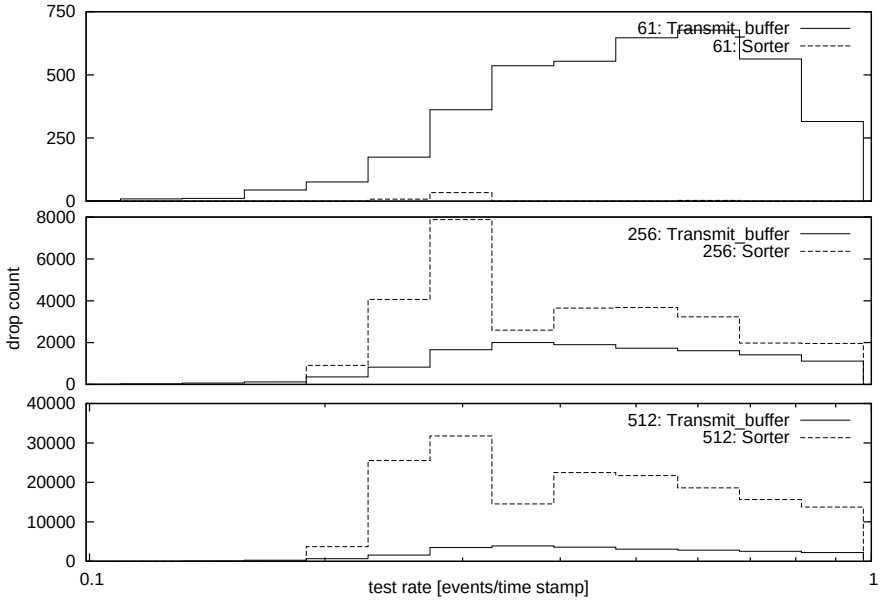

Figure 6.6: Comparison of drop rates and reasons during experiments with variable n_e of 64, 256 and 512. The plotted numbers give the totally counted drops for all runs of one test rate. All counters not shown are at zero. Note the logarithmic scale of the x-axis.

Results

According to Figure 6.5 and 6.4, the tested rates can be divided into two ranges: Up to a source rate of 0.17 events/timestamp, more than 95% of events arrive. The length of the spike train has no influence on the arrival ratio and therefore only runs with a length of 10000 events are shown, in Figure 6.4. In this range rates can be sustained over a prolonged time.

Above 0.17 events/timestamp, the arrival ratio breaks down significantly and shows a strong variance among runs with identical test rate. The length of the spike train n_e has a major influence on achievable arrival ratios and the limitation of the source rate. This range characterizes the bursting behavior of the network.

Sustainable transfer rates According to Figure 6.4, no events are dropped for rates below 0.01 events/timestamp. With one event in hundred time stamps on average, interactions between events are very unlikely. Buffers are sparsely populated and for most events the system will appear empty. For even lower rates, no other events will be transported, during the lifetime of one event. Thus, it is not necessary to test rates below 0.001 events/timestamp. Above 0.01 events/timestamp reliability degrades slowly, until it becomes manifest at 0.17 events/timestamp.

Burst performance In the range above 0.17 events/timestamp the length of the spike train n_e gains influence. The upper plot in Figure 6.5 shows, that the event packing algorithm in the `sendSpikeTrain` function limits the achievable source rate. For high test rates events are removed by the software, thus decreasing the effective spike train length. This leads to the ascending "tail" visible in all three experiments. Shorter trains are transported with less loss and hence it seems as if the arrival ratio would improve with higher rates, which is not the case. Ignoring the "tail", the rest of the points clearly show, that for short bursts a higher percentage of events arrives at the destination, than for long bursts.

Figure 6.6 shows readout values of `Drop_counter` (cf. Section 5.4.2) obtained during the experiment. It shows where events are lost for the different spike train lengths.

For $n_e = 64$ practically all loss is on the sending side inside `Transmit_buffer`. Overflow of this buffer indicates, that the event rate at the source exceeds the network bandwidth. Losing events here is inevitable, due to the given throughput relations. A larger FIFO could improve performance for bursts, but would also increase latency.

The decrease of the drop rate at high test rates is caused by the event removal of `sendSpikeTrain`. The actually transmitted spike train holds less than n_e events and therefore the absolute number of dropped events during one run is reduced, too. The effect is visible for all three burst lengths.

For higher n_e, loss caused by the input drop condition of `Sorter` (cf. Section 3.3.3) begins do dominate. In all three experiments, its associated drop rate peaks at 0.3 events/timestamp and then declines to lower rates. The `Recv_queue` FIFO has a depth of 128 entries. Events coming in at high rates fill up the buffer and thus increase the propagation time through it significantly. When transmission latency plus this propagation delay grow larger than the programmed delay, the drop condition is true and the event is dropped by the input logic of `Sorter`. During the clock cycle in which the event is being dropped, time advances by two time stamps. Hence, for dense spike trains, it is well possible, that the next event also has to be dropped. From then on, the drop rate is antagonized by the arrival rate in shrinking the FIFO to a size yielding a delay which does not cause dropping of events.

This explanation is in good agreement with the results from the performed experiments. For $n_e = 64$ the FIFO can not become more than half full. Drops at `Sorter` now only occur, when transmission jitter and small programmed delay combine negatively. Therefore the drop rate at `Sorter` is very low. For larger n_e, the FIFO can grow fuller and the effect is more probable, leading to an increased drop rate.

Recv_queue influence on throughput If the depth of `Recv_queue` is the reason for loss at `Sorter`, a smaller depth should reduce the drop rate. This would of course be accompanied by an increase in loss due to overflows of the `Recv_queue` FIFO. Figure 6.7 shows an experiment with $n_e = 512$, where `Recv_queue` was reduced to 32 entries. The number of drops at `Sorter` is reduced by about 50% compared to Figure 6.6. The arrival ratio in the lower sub picture is on average above 0.5 events/timestamp, where it was mostly below 0.4 events/timestamp in Figure 6.5. Here, one has the counterintuitive case, where a decrease in buffer size increases throughput.

Absolute rates Figure 6.8 shows a plot with absolute event rates for an experiment with $n_e = 64$. The source rate is measured after any potential removal of events by `sendSpikeTrain`. The standard deviation for each test rate is high, as would be expected for short random spike trains. At a test rate of about 0.5 events/timestamp the slope of the curve begins to decrease and the rate approaches an upper limit below 0.375 events/timestamp. This plateau is caused by the aforementioned removal of events by `sendSpikeTrain`. For lower test rates, one would ideally have a linear relation between test and source rate with a slope of one. The measured curve has a slightly smaller slope, because consecutive time stamps are prevented.

The delivery rate measured at the destination follows the source rate closely for test rates up to 0.3 events/timestamp. Above these rates, dropping of events limits the delivery rate to below 0.25 events/timestamp.

6.3 Demonstration Experiment

6.3.1 Simple Artificial Neural Network Demonstration

The purpose of this experiment is to demonstrate, that the system can be used for the experimentation with artificial neural networks on multiple chips. The idea is to configure two simple networks on two separate *Spikey* chips and transmit the output of one to the synapses of the other across the gigabit network. When the first network is stimulated, the second should exhibit activity only, when the network is configured to forward events.

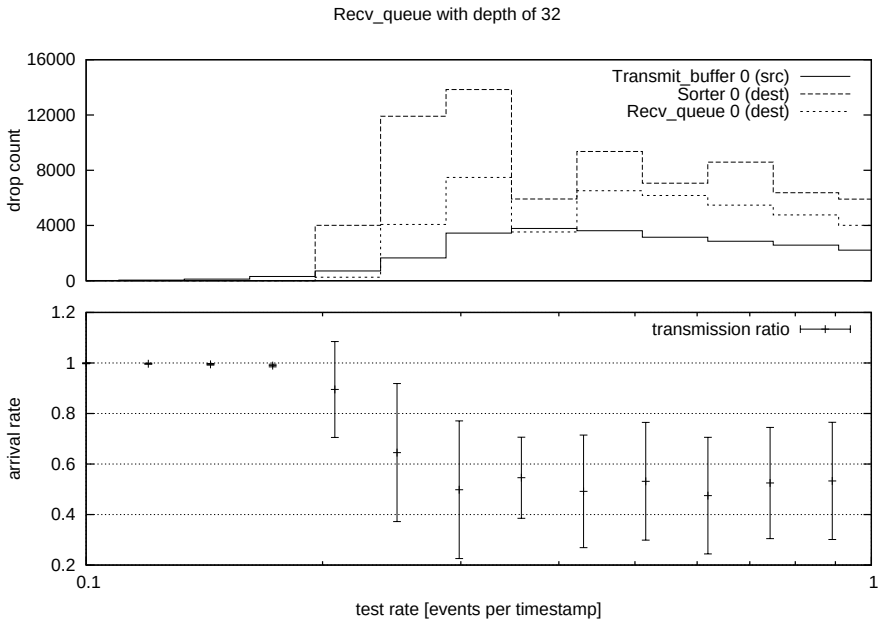

Figure 6.7: Drop rates for reduced `Recv_queue` depth of 32. Compare these plots to their counterparts in Figures 6.6 and 6.5. Loss at `Sorter` is reduced by about a factor of two, while drop at `Recv_queue` now increases. The total arrival ratio is better, than for a depth of 128.

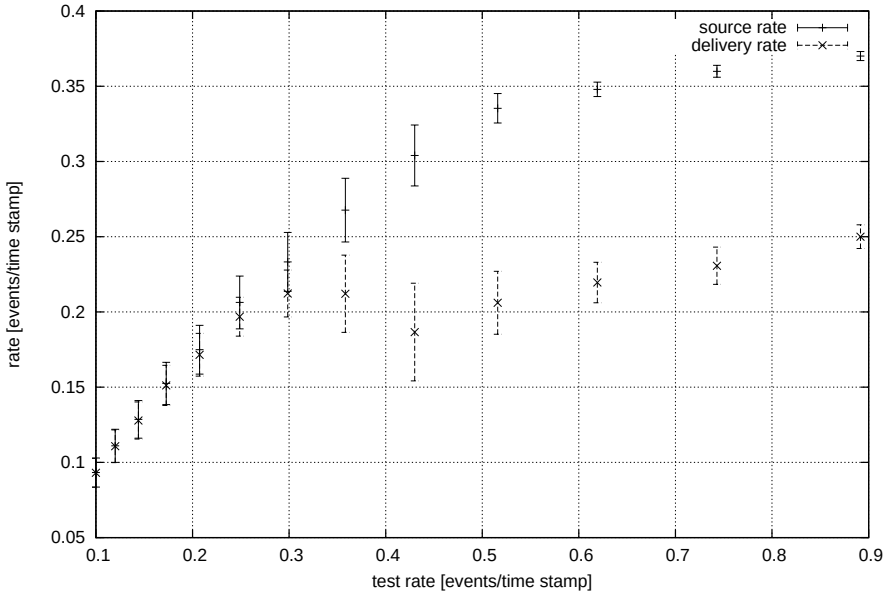

Figure 6.8: Absolute measured source and delivery rates for $n_e = 64$ and $n_j = 100$. The source rate curve is a result of the event packing algorithm implemented in the `createtb` software. The delivery rate is the number of events, that arrive at the destination.

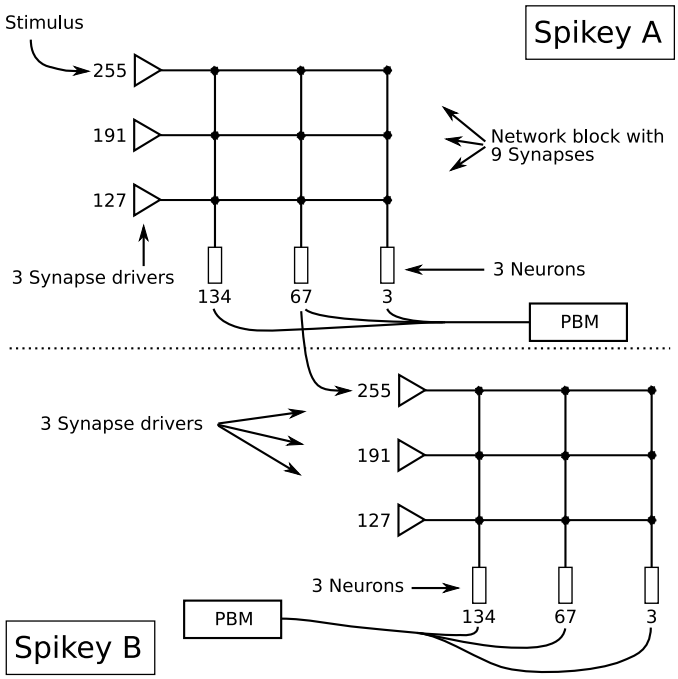

Figure 6.9: Setup of the artificial neural network demonstration experiment. The upper half shows the source *Nathan* network module on *Backplane* slot 0. Stimulus is applied to synapse driver 255 and results are stored in the *PlaybackMemory*, labeled PBM. A network connection forwards events from neuron 67 to synapse driver 255 on the destination chip. Each dot represents an active synapse in the network block.

No.	Neuron	Firing rate [events/time stamp]	
		Spikey A	**Spikey B**
(0)	3	$(4.3 \pm 0.1) \times 10^{-4}$	0.0
	67	$(4.1 \pm 0.1) \times 10^{-4}$	0.0
	134	$(3.1 \pm 0.1) \times 10^{-4}$	0.0
(1)	3	0.0	$(3.7 \pm 0.1) \times 10^{-4}$
	67	0.0	$(4.4 \pm 0.1) \times 10^{-4}$
	134	0.0	$(4.3 \pm 0.1) \times 10^{-4}$
(2)	3	$(4.1 \pm 0.1) \times 10^{-4}$	$(6.0 \pm 0.2) \times 10^{-4}$
	67	$(4.0 \pm 0.1) \times 10^{-4}$	$(7.0 \pm 0.2) \times 10^{-4}$
	134	$(3.7 \pm 0.1) \times 10^{-4}$	$(6.2 \pm 0.2) \times 10^{-4}$

Table 6.3: Artificial neural network demonstration. Experiments (0) and (1) are without network connection and only one *Spikey* is stimulated respectively. Experiment (2) is identical to (0), except for the network connection from neuron 67 of *Spikey* A to synapse driver 255 of *Spikey* B. The measured firing rates at the destination are solely produced by events transported via the inter-chip network.

To achieve this, a PyNN script was used to generate a configuration file for the *Spikey* analog part. The configuration activates in total nine synapses with maximum weight, to connect neurons 3, 67 and 134 to synapse drivers 127, 191 and 255. Figure 6.9 gives an schematic overview of the setup. Some fine tuning had to be done on the produced configuration file. Synapse driver strength was set to maximum to achieve reliable firing. Therefore it may be, that one input event evokes more than one output event, when a very strong spike is digitized multiple times.

A test mode for the `createtb` program was developed to send the configuration file to the chips, produce stimulus spike trains and gather resulting events. Stimulus events are non-random: Four bursts of seven events at intervals of two time stamps are sent to synapse driver 255 with gaps of 400 time stamps in between.

Performed experiments Table 6.3 lists results for this experiment. The first two experiments - (0) and(1) - were to validate the configuration of the analog part. There was no network connection between chips and stimulus was applied to only one of them. The results show, that neurons fire only on stimulation.

For the third experiment (2), stimulation is only applied to *Spikey* A and a connection from neuron 67 to synapse driver 255 on *Spikey* B is established. The network connection is the only difference to (0). The measured rates show that activity is evoked by spike events transmitted over the event network.

Chapter 7

Scalability of the System

This chapter summarizes scalability related characteristics of the event network. Of major impact are the dimensions of data fields in the lookup tables described in Section 5.1.1. As a reminder: There are four types of lookup tables. After an event is received from *Spikey*, `Send_lut` assigns it to a connection bundle and tag. On the remote *Nathan*, `Recv_lut` translates these into a target synapse driver address and delay. The association of bundles to MCGN time slots is performed by `Mfi_lut` and `Slot_lut`. All neuronal connections within one bundle link the same two *Nathan* modules.

The connection tag, or `subnr`, is 6 bit wide. Therefore 64 neuronal connections can be fitted into one bundle. The `Transmit_buffer` logic block maintains eight internal queues, which are associated with one bundle each. This results in an equal number of outbound bundles per user port and thus sixteen in total per *Nathan*. The inbound number of bundles is determined by the size of the local bundle identification number `lvc`, which is 4 bit wide. Thereby sixteen inbound bundles can be used per user port, totaling in 32 per *Nathan*.

Altogether, this allows for the simultaneous interchange of events between all network modules of a fully equipped *Backplane*. It is also possible to simultaneously connect all 384 neurons of one *Spikey* to synapses on another one by combining multiple bundles.

A further increase of the numbers of inbound bundles and connections per bundle is limited by the index of the `Recv_lut` lookup table. This index is composed of the connection tag `subnr` and the bundle identification `lvc`. Since 17 bit are stored per entry and the lookup table memory has a capacity of 18 kBit, at maximum a 10 bit wide address can be used for the BlockRAM storage memory [Xil07]. Therefore, `subnr` and `lvc` together may not use more than 10 bit as long as only one BlockRAM is used for the lookup table. To add one more address bit, the amount of memory elements must be doubled. According to Section 5.2.3, five BlockRAM components are still available on the FPGA. Thus, for two user ports, either the number of inbound bundles or the number of connections per bundle could be doubled.

The amount of outbound bundles can be improved by adding more queues to the `MultiFIFO` component inside the `Transmit_buffer`. One limiting element is again the underlying BlockRAM memory. For an entry size of 32 bit, the memory provides 512 entries. The product of number of FIFOs and their depth may therefore not exceed 512. However, the control logic, which maintains pointers for each virtual queue, is also enlarged, when either of the quantities are increased. Considering only the memory, 128 queues with a depth of four or 32 queues with a depth of sixteen would be possible. Such an increase would also make it necessary to enlarge the `Mfi_lut` lookup table and the `mfi` field in `Send_lut`.

The signal lines of the MGT-links are hardwired on the *Backplane* (cf. Section 2.1.2). The topology can be described as a binary hypercube with edges representing links and corners representing *Nathan* modules. The dimension of the hypercube is the number of MGTs per node. To fully utilize the topology of one *Backplane*, every node must implement four MGT-links. Note, that the number of MGTs can be different from the number of user ports. The presented imple-

mentation has four MGT ports to fully cover the *Backplane* topology. Two user ports make half the theoretical bandwidth available.

The presented implementation can be scaled up to 6 MGTs by change of a parameter and resynthesizing the design. In the canonical hypercube topology this yields a six-dimensional hypercube with 64 *Nathan* modules on four *Backplane*s. In such a network, the longest path has six hops, which would, extrapolating from latency measurements in Section 6.2.1, lead to a delay of 541 time stamps. This is equivalent to a biological time of 271 ms for a *Spikey* clock of 200 MHz and speed-up of 10^5, and 17.3 ms for a clock of 312.5 MHz and speed-up of 10^4.

Of course, an arbitrary number of *Nathan* modules can be interconnected by the gigabit transport network if other topologies are used. For example, several canonically interconnected five-dimensional hypercubes could be links to form layers of a feed-forward network. In every layer, each node would connect to one node in a layer above or below using the sixth MGT.

Power consumption also has an impact on scalability. In the current setup, a single *Backplane* is powered by a commercial ATX[1] power supply.According to results from the Xilinx Power Analyzer software, the FPGA on *Nathan* consumes 4.95 W for a design with six MGTs at 100 MHz. Together with *Spikey* consuming 0.54 W [Grü09], this totals in 87.84 W for a *Backplane* with sixteen *Nathan* modules. 100 W is a safe upper limit, when also including the *Backplane* FPGA into the sum. It was already stated in Section 2.1.2, that the *Backplane* is designed to fit into a 19"-rack, where it takes 3 U[2] of height. For a common 2 m tall rack with 42 U, 12 *Backplane*s could be assembled into one rack, when 6 U are reserved for controlling PC, power supply and optional additional components. A rack would thus in total consume 1.2 kW plus 800 W for PC and additional components. Even when doubling this number to account for power supply efficiency, this is well within the range for racks used in data centers. For example IBM lists 40 kW per rack for the BlueGene/P supercomputer [IBM07].

What poses a more difficult obstacle to scaling is the demand of MCGN, that all *Nathan* modules have the same clock source. Thereby, elaborate means are required to feed one central clock signal to all *Backplane*s in the system. The *Backplane* itself provides connectors to feed in an external clock. For a single rack system, cables of about a meter in length would suffice, which is short enough for a reliable transmission of the signal. For a system with more than one rack, cable lengths of a few to several meters should be feasible. For example Serial Attached SCSI allows for a cable length of 8 m at a data rate of 3 GBit/s [sas03].

Furthermore the length of signal cables for the MGT interconnection is limited. MCGN uses programmable delay elements on every link, to balance physical latencies of the lines. According to the synchronization condition given in [Phi08b], the network is synchronized, when time slot 0 arrives in the same clock cycle on every link. Therefore, a long physical delay on one line is compensated by increasing the delay for the other lines. The largest programmable delay in the current implementation is seven cycles, which is equivalent to 70 ns at 100 MHz. Assuming the velocity of propagation at 2/3 the speed of light in vacuum, 14 m of difference in cable length could be compensated. This would shrink to 8.96 m for a clock frequency of 156.25 MHz. Cable length is also limited by the transceivers themselves. According to [mgt04], a maximum cable length of 10 m is possible at 156.25 MHz using Infiniband cables. At lower data rates 15 m long cables can be used.

In summary, a "data center scale" system consisting of four to eight racks is technically possible. It would have 768 to 1536 *Spikey* chips with 301056 to 602112 neurons and consume 16 to 32 kW of power. The limitations of throughput, per hop delay and number of connection bundles would allow only locally connected artificial neural networks. The latencies through the network would amount to hundreds of milliseconds to seconds in biological time.

[1] Advanced Technology Extended
[2] 1 rack unit = 44.45 mm

Chapter 8

Conclusion and Outlook

The goal of this thesis is the completion of an event network for a multi-chip artificial neural network hardware system. For this purpose digital event processing logic was developed to enable the exchange of digitized spike events across chip boundaries. This included devising a low-latency stream sorting logic module with an adaptive readout rate. The sorting module not only allows a temporary excess of the chip interface bandwidth, but also eliminates jitter of the transport network for neuronal connections. The design was implemented for the FACETS stage-1 system in FPGA logic. The hitherto existing workflow to setup and perform experiments was adapted to make multi-chip experiments possible. To this end, software modules and programs were developed to access the implemented logic modules. Various experiments were performed to verify the function of the event network within the stage-1 system and to measure its performance. Also, its scalability was studied to identify the limiting factors.

This chapter discusses the results gathered by these activities and gives an outlook on possible improvements and future developments.

8.1 Achievements

The most important result of this thesis is, that the exchange of spike events across chip boundaries, i.e. multi-chip operation, is now possible within the FACETS stage-1 system. The most demonstrative experiment underlining this achievement is the simple artificial neural network setup described in Section 6.3.1. It represents a prototype for more elaborate experiments using interconnected neurons on several chips. But also the remainder of experiments described in Chapter 6 made extensive use of the event network. Millions of events were transferred and verified in total. This qualitative result is complemented by quantitative performance measurements, namely those of latency, throughput and the limits of scalability.

Latency A low latency is of foremost importance for the biological relevance of the network. It imposes a lower bound on the axonal delay which can be modeled between chips. Hence, it is desirable to keep this bound as low as possible to ensure flexibility for the modeler. Flexibility also benefits from a high range of programmable delay values. The implemented event network makes programmable delays in the stage-1 system possible for the first time. By design, the maximum difference in delay among connections to a single chip is 255 time stamps and 2048 time stamps among connections to different chips. These numbers correspond to 128 ms and 1.02 s respectively in the biological time domain for a *Spikey* clock frequency of 200 MHz and a speed-up of 10^5, i.e. the tested setup. The fourth version of the *Spikey* chip is designed for a clock frequency of 312 MHz and a speed-up of 10^4, resulting in biological time differences of 8.17 ms and 65.6 ms respectively. The minimum delay for reliable communication between two chips was found to be 284 time stamps, which corresponds to 142 ms for the tested setup and 9.1 ms for *Spikey-4*.

Minimum delay connections can be used between any two directly adjacent network modules in the MGT-interconnect topology of the *Backplane*.

According to earlier considerations in Section 1.1, propagation delays of action potentials in biological neural networks range from milliseconds to hundreds of milliseconds. Since a large amount of this time is caused by the propagation delay along the axon, the delays between neurons are related to their physical distance. Therefore, low-delay connections are of great importance, when modeling circuits confined to small volumes. With *Spikey-4*, delays in the range of about 10 ms will be possible, with differences in delay in a similar range. This makes the network usable for experiments with a timing that is biologically still relevant.

The *Spikey-3* based system, with its larger latencies, can still be used for experiments with reduced biological accuracy. Also, models which use firing-rates to encode information are less affected by larger delays and should be well suited for this system.

Throughput The other important performance measure is throughput, i.e. the amount of events that can be passed through a connection per time unit. The experiments described in Section 6.2.2 show, that the range of sustainable rates, at which less than 5% of events are dropped, reaches up to 0.17 events/timestamp. This is equivalent to a biological firing rate of 340 Hz for the tested setup and 5.304 kHz for *Spikey-4*. Assuming a maximum firing rate of 40 Hz for neurons in the high-conductance state (cf. Section 1.1), this allows 8.5 and 132.6 axonal connections respectively to be transported by a single MGT link.

At rates above 0.17 events/timestamp, significant loss of events occurs. The actual fraction which arrives at the destination depends strongly on the total number of events being sent (cf. Figure 6.5).

The maximum outbound bandwidth per *Nathan* is limited by the number of user ports on the MCGN switch. The final design, that was subject of the experiments described in Chapter 6, implements two user ports. Hence, half of the total bandwidth provided by the four MGT-links of the hardwired *Backplane* links can be used in this configuration.

Scalability An analysis of the scalability of the networking system was conducted (cf. Chapter 7). It found, that the number of inter-chip connections per *Nathan* is primarily limited by the lookup table sizes in the networking logic and thus by logic and memory resources of the FPGA device. The presented implementation allows for neuronal networks with 32 neuron-to-neuron connections between any two *Nathan* modules on a fully equipped *Backplane*. With the available FPGA resources, the network can be scaled to four fully equipped *Backplane*s with the same interconnectivity. A further scaling requires fundamental changes in the FPGA design to free more resources.

For weakly interconnected networks, e.g. feed forward topologies, scaling is limited by the maximum length of cables for clock distribution and the MGT links. A “data center scale” system with up to eight racks and 1536 *Spikey* chips featuring 602112 neurons is feasible. Such a system would consume 32 kW of power in total. Due to the limitations by the network, namely large latencies and only locally interconnected topologies, and the chip inherent maximum neuronal input count of 256, such a large-scale system would only be of limited interest for neuroscientific studies.

8.2 Improving the System

The presented implementation still offers room for improvements. Fortunately, programmable logic allows for later modifications and a continuous evolution of the system. Constrained by hard limitations, like the number of logic elements the FPGA provides, some aspects of the event processing logic can be improved:

Extended sorting memory size The experiments that measured throughput (cf. Section 6.2.2) revealed, that for high rates and long experimental durations the input logic at `Sorter` is respon-

sible for more than 50% of drops. The drop condition, which was added to the input logic to avoid deadlocks in `Sorter` (cf. Section 3.3.3 and 6.1.2), will become unnecessary, if the size of the sorting memory is increased. If the size of the memory is equal to the dynamic range of time stamps, it will not be necessary to artificially retain events in `Recv_queue`, when their target time is more than 256 time stamps in the future. Hence, the deadlock inducing situations, in which one event far into the future would stall another beyond its instant of delivery, cannot occur.

Increasing the size of the sorting memory from 256 to 1024 is possible with limited additional resource requirements. No additional BlockRAM components would be required, since the one in place was not used to full capacity. An extension to the full range of 2048 would make a second BlockRAM necessary. As mentioned in Section 5.2.3, five BlockRAMs are not used in the final design. The increased sorting memory size should therefore be possible even in combination with an optional extension of `Recv_lut`, that was mentioned in the previous section.

Common extended delay In the presented implementation, the 12 bit target time stamp is calculated by adding a 12 bit delay to the source time stamp. While the lower 8 bit of the delay are individually programmable for every neuronal connection, the upper 4 bit are common to one *Nathan*. An alternative implementation would be, to store a common 12 bit delay and add the sum of this and the individual 8 bit delays to the source time. This would require eight more bits in the common delay register and an additional 12 bit adder.

The alternative implementation would allow more precise control over the usable delay range on one *Nathan*. In particular, the common 12 bit delay could be set precisely to the minimal value as measured in Section 6.2.1.

Specialized PlaybackMemory adapter The *PlaybackMemory* is integrated into the event network by using a marginally modified `Sorter` as adapter. Yet, the sorting and buffering capabilities of this complex logic block are not required. It should be possible to develop a more lightweight adapter consisting of a small FIFO and a comparator. The adapter would exhibit the same interface as `Sorter` to the `Reduce_events` module and thus deliver events synchronized with the `Sorter` blocks working in parallel. This would free a considerable amount of resources, especially the two BlockRAMs that are needed for `Sorter`.

Better support for connections with multiple destinations The current design was not optimized for 1-to-N connections, i.e. connections, which transport events from one source to more than one destination chip. To implement such a topology, one has to use the multicast mechanism of MCGN and route time slots to multiple destinations. A dedicated connection bundle would then be assigned to these time slots. A disadvantage here is the all-or-nothing nature of the approach: All connections within the bundle share the same destinations. Therefore, one bundle has to be allocated for every combination of destination *Nathan*s.

A very simple measure to improve the situation would be to drop events of undesired connections at the destination. The one unused bit of `Recv_lut` entries could be used to select events that are to be dropped. Of course, this leads to overhead, since these events occupy additional network and lookup-table resources. If on average more than half of the connections in a bundle would be dropped at a destination, another approach is more effective. An event sequencer in `Source_generator` could generate an event for every destination, based on extended routing information in `Send_lut`. They would then be handled by the network like any other event. This method has the disadvantage, that it multiplies traffic in `Sender` and on the network by the number of destinations.

Both methods combined would allow a more flexible means of implementing 1-to-N connections in the system.

Spatial sorting in Receiver Another fundamental change in the event processing logic, would be to add a spatial sorting stage to `Receiver` before the temporal sorting stage of `Sorter` blocks. A switch like `Send_switch` could direct incoming events from user ports to `Receive_line` blocks

depending on their destination address. Thus, each line could be associated with a set of synapse driver blocks (cf. Section 2.3.1). Thereby, collisions of events targeted at the same time stamp, but in separate blocks, would no longer occur. Collisions of events within the same block are illegal anyway and actively filtered out by `Reduce_events` in the current design.

The other advantage would be the elimination of the 1-to-1 relation between user ports and `Receive_line` blocks. For example four user ports could be connected to two `Receive_line`s, improving throughput. MCGN takes two clock cycles to deliver one event, but the `Receive_line` can accept one in every cycle. Two of the user ports could be delayed by a single cycle to exploit this.

Implementing this improvement would require BlockRAM and distributed memory resources for additional `Send_lut` and `Slot_lut` blocks. Further, logic resources comparable to the amount `Send_switch` were required (cf. Section 5.2.3), as well as more logic cells needed for two additional user ports on the MCGN switch. Considering the available resources, these requirements are unlikely to be satisfiable without optimizations in other parts of the design.

8.2.1 Advancements Requiring New Hardware

The improvements outlined in the previous section are all at least in principle possible using the existing hardware platform. However, it may be of interest to also consider what would be possible, when modifying some parts of the framework.

FPGA upgrade Looking at the age of the Virtex-II Pro FPGA model and the scarcity of its resources, upgrading the *Nathan* modules to a newer FPGA device comes to mind easily. While it would require a large amount of effort to redesign the layout of the printed circuit board (PCB), it would provide more sophisticated logic cells in higher quantities. A synthesis run, that was performed with the Virtex-V 110LXT model as a target to estimate improvements, listed a usage of 6% (4711 of 69120) of Flip-Flops and 7% (5407 of 69120) of logic lookup tables. This would leave plenty of room to use more user ports and MGT-links, to implement resource intensive improvements from the previous section and to use additional features of MCGN, like shared memory [Phi08b].

Real-time interaction system Complementing the established model of inserting stimulus events via the *PlaybackMemory*, the gigabit network could be used for this purpose. An adapter card could be connected to the PCI-Express bus of a controlling PC and by cable to MGT-links of *Nathan* modules. To increase the bandwidth, multiple cards, possibly even on multiple PCs of a computing cluster, could be used. The adapter would not have to be specifically designed for this purpose, but a commercial board could be used, provided it has sufficiently many MGTs.

PCI-Express features data rates of 2 GBits/s and above, which is comparable to the data rates of the gigabit network. Therefore, software could transmit and receive spike trains, while the experiment is running, allowing for the integration of the hardware system into software simulations.

Final Remarks

Concluding from the analysis of results presented in this chapter, the implemented event network is usable for neuroscientific experiments. The versatility of mappable neural networks is limited by the minimal end-to-end latencies between chips, which are large compared to biological networks in small volume. Also, the achievable connection densities do not match that of biological networks in the neocortex of the human brain (cf. Section 1.1). However, especially with *Spikey-4*, which will be available within a short time, networks that are still biologically relevant can be studied on the hardware system.

The stage-2 system following an approach of waferscale integration will finally supersede stage-1 as a large-scale hardware system. It interconnects several chips directly on the wafer by means of

signaling lines added in a postprocessing step. The asynchronous event network using these lines will allow for higher data rates and lower latencies. An improved neuron model will enable high input counts of up to 14366 incoming connections per neuron [Mil09]. But until this system is fully operational, stage-1 is the only available and tested hardware system within the FACETS project that can be used for large-scale experiments. Hence, in extending the system for multi-chip operation, the presented work has opened new experimental opportunities.

Even when stage-2 is operational stage-1 will offer possibilities for that it is better suited than the waferscale system. For example, it could be used as basis for a small scale system combining an FPGA and one or more *Spikey* chips into a single device. Such a device could possibly be build as a USB[1]-stick or in another form more accessible than the *Backplane* system. Multiple *Spikey* chips can be combined in a chain and connected to a single FPGA without requiring the use of additional pins [Grü07]. The presented event networking logic would then interconnect the chips inside the FPGA. This device could be used to give a wider audience access to neuromorphic hardware systems by providing it to other research groups.

By its low power requirements, such a device would also be applicable to embedded applications, e.g. as a controller in a robot. The translation of motor and sensor signals from spike trains to voltages and currents would be accomplished by the FPGA. An interface to the event network would enable real-time interaction between neural network and the robotic hardware system. Because of the high speedup factor of the *Spikey* chip, neural codes that use firing rates to encode information could be used for the control of fast dynamic systems. As stated in Section 8.1 rate based networks are less obstructed by the latency limitations of the event network. Therefore, the stage-1 system may – beyond being a neuroscientific experimentation platform – serve as a basis for future technological applications using neuromorphic hardware.

[1] Universal Serial Bus

List of Acronyms

ANNA	Artificial Neural Network ASIC
ASIC	Application Specific Integrated Circuit
ATX	Advanced Technology Extended
CLB	Configurable logic block of the Virtex-II Pro FPGA
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
DCM	Digital Clock Managers
DDR	Double Data Rate
DLL	Delay Locked Loop
DTC	Digital to Time Converter
FACETS	Fast Analog Computing with Emergent Transient States
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GSS	Global Synchronous Signal
IOB	Input/Output Block
IST	Information Society Technologies
LUT4	4-input lookup table of the Virtex-II Pro FPGA
LUT	Lookup Table
MCGN	Multi-Class Gigabit Network
MGT	Multi-Gigabit Tranceiver
PBM	Playback Memory
PCB	Printed Circuit Board
PCI	Peripheral Component Interconned
PC	Personal Computer
PLL	Phase Locked Loop

PSP Postsynaptic Potential

QoS Quality of Service

RAM Random Access Memory

SCSI Small Computer System Interface

SDRAM Synchronous Dynamic Random Access Memory

STDP Spike Timing Dependent Plasticity

TDC Time to Digital Converter

TDM Time Division Multiplexing

USB Universal Serial Bus

VHDL Very high speed integrated circuit Hardware Description Language

VOQ Virtual Output Queue

Bibliography

[Amd67] Gene Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, 1967.

[BBC+98] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. RFC 2475, Network Working Group, December 1998.

[BBJ+05] Theodor H. Bullock, Michael V. L. Bennett, Daniel Johnston, Robert Josephson, Eve Marder, and R. Douglas Fields. The neuron doctrine, redux. *Science*, 310:791–793, November 2005.

[BC85] C. Beaulieu and M. Colonnier. A laminar analysis of the number of round-asymmetrical and flat-symmetrical synapses on spines, dendritic trunks, and cell bodies in area 17 of the cat. *Computational Neurology*, 231(2), 1985.

[BCS94] R. Braden, D. Clark, and S. Shenker. Integrated services in the internet architecture: an overview. RFC 1633, IETF: Network Working Group, June 1994.

[BDM05] Tom Binzegger, Rodney J. Douglas, and Kevan A.C. Martin. Axons in cat visual cortex are topologically self-similar. *Cerebral Cortex*, 15(2):152–165, 2005.

[Bil08] Johannes Bill. Self-stabilizing network architectures on a neuromorphic hardware system. Diploma thesis (English), University of Heidelberg, HD-KIP-08-44, 2008.

[BRG+07] G. Bontorin, S. Renaud, A. Garenne, L. Alvado, G. Le Masson, and J. Tomas. A real-time closed-loop setup for hybrid neural networks. In *Proceedings of the 29th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBS2007)*, 2007.

[Brü09] Daniel Brüderle. *Neuroscientific Modeling with a Mixed-Signal VLSI Hardware System*. PhD thesis, Ruprecht-Karls Universität Heidelberg, 2009.

[Deb04] Dominique Debanne. Information processing in the axon. *Nat Rev Neurosci*, 5(4):304–316, 04 2004.

[Dow08] T. Downarowicz. Law of series/poisson process. *Scholarpedia*, 3(11):3922, 2008.

[Dre08] Jürgen Drexler. Entwurf und implementierung einer parallelen netzwerkschnittstelle zum betrieb künstlicher neuronaler netze. diploma thesis, Ruprecht-Karls Universität Heidelberg, 2008.

[DRP03] Alain Destexhe, Michael Rudolph, and Denis Pare. The high-conductance state of neocortical neurons in vivo. *Nature Reviews Neuroscience*, 4:739–751, 2003.

[dZ00] R. Christopher deCharms and Anthony Zador. Neural representation and the cortical code. *Annual Review of Neuroscience*, 23:613–647, March 2000.

[ea05] Karlheinz Meier et al. The facets project: Fast analog computing with emergent transient states". EU FP6-2004-IST-FETPI contract no. 15879, 2005.

[FAC09] FACETS. Fast Analog Computing with Emergent Transient States – project website. http://www.facets-project.org, 2009.

[GK02] Wulfram Gerstner and Werner Kistler. *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press, 2002.

[Grü03] Andreas Grübl. Eine fpga-basierte plattform für neuronale netze. diploma thesis, Universität Heidelberg, 2003.

[Grü07] Andreas Grübl. *VLSI Implementation of a Spiking Neural Network*. PhD thesis, Ruprecht-Karls-University, Heidelberg, 2007. Document No. HD-KIP 07-10.

[Grü09] Andreas Grübl. personal communication, 2009.

[hHcF05] Chung hsing Hsu and Wu chun Feng. A power-aware run-time system for high-performance computing. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2005.

[Hyp06] HyperTransport Technology Consortium. *HyperTransport I/O Link Specification*, revision 3.0a edition, November 2006.

[IBM07] IBM. *IBM Blue Gene/P Specification*, 2007.

[Kap08] Bernhard Kaplan. Self-organization experiments for a neuromorphic hardware device. Diploma thesis (English), University of Heidelberg, HD-KIP-08-42, 2008.

[KM65] B. Katz and R. Miledi. The measurement of synaptic delay, and the time course of acetylcholine release at the neuromuscular junction. *Proceedings of the Royal Society of London. Series B, Biological Sciences*, 161(985):483–495, February 1965.

[LMBA06] Francisco López-Muñoza, Jesús Boyab, and Cecilio Alamoa. Neuron theory, the cornerstone of neuroscience, on the centenary of the nobel prize award to santiago ramón y cajal. *Science*, 2006.

[loe09] Homepage of the loebner prize in artificial intelligence. http://www.loebner.net/Prizef/loebner-prize.html, 09 2009.

[LS03] Simon B. Laughlin and Terrence J. Sejnowski. Communication in neuronal networks. *Science*, 301(5641):1870–1874, 2003.

[Maa97] W. Maass. Networks of spiking neurons: the third generation of neural network models. *Neural Networks*, 10:1659–1671, 1997.

[McK99] Nick McKeown. The islip scheduling algorithm. *IEEE/ACM Transactions on Networking*, 7(2):188–201, 1999.

[Mea90] C. A. Mead. Neuromorphic electronic systems. *Proceedings of the IEEE*, 78:1629–1636, 1990.

[mgt04] *Infiniband Cable Characterization with RocketIO™ MGTs*, 2004.

[Mil09] Sebastian Millner. personal communication, October 2009.

[MM88] Carver A. Mead and M. A. Mahowald. A silicon model of early visual processing. *Neural Networks*, 1(1):91–97, 1988.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[MP43] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, pages 127–147, 1943.

[NBBB98] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the differentiated services field (ds field) in the ipv4 and ipv6 headers. RFC 2474, Network Working Group, December 1998.

[PD03] Larry L. Peterson and Bruce S. Davie. *Computer networks: a systems approach*. Elsevier Science Ltd., 3rd edition, 2003.

[Phi08a] Stefan Philipp. *Design and Implementation of a Multi-Class Network Architecture for Hardware Neural Networks*. PhD thesis, Ruprecht-Karls Universität Heidelberg, 2008.

[Phi08b] Stefan Philipp. *Design and Implementation of a Multi-Class Network Architecture for Hardware Neural Networks*. PhD thesis, Ruprecht-Karls Universität Heidelberg, 2008.

[PSM09] Stefan Philipp, Johannes Schemmel, and Karlheinz Meier. A qos network architecture to interconnect large-scale vlsi neural networks. In *IJCNN 2009 Conference Proceedings*, 2009.

[sas03] Serial attached scsi (sas). Specification 376-2003, ANSI/INCITS, October 2003.

[SBMO07] 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'07)*. IEEE Press, 2007.

[Sch05] T. Schmitz. *Evolution in Hardware – Eine Experimentierplattform zum parallelen Training analoger neuronaler Netzwerke*. PhD thesis, Ruprecht-Karls-University, Heidelberg, 2005.

[SFM08] 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.

[SKdRvSB98] S. P. Strong, Roland Koberle, Rob R. de Ruyter van Steveninck, and William Bialek. Entropy and information in neural spike trains. *Physical Review Letters*, 80(1):197–200, 1998.

[SN58] Herbert A. Simon and Allen Newell. Heuristic problem solving: The next advance in operations research. *Operations Research*, 6(1):1–10, January 1958.

[Sof95] William R. Softky. Simple codes versus efficient codes. *Current Opinion in Neurobiology*, (5):239–247, 1995.

[SS02] Gunter Saake and Kai-Uwe Sattler. *Algorithmen und Datenstrukturen - Eine Einführung mit Java*. dpunkt.verlag, 2002.

[SSR02] Walter Senn, Martin Schneider, and Berthold Ruf. Activity-dependent development of axonal and dendritic delays, or, why synaptic transmission should be unreliable. *Neural Computation*, 14(3):583–619, 2002.

[STG01] M Steriade, I Timofeev, and F Grenier. Natural waking and sleep states: a view from inside neocortical neurons. *J Neurophysiol*, 85(5):1969–1985, May 2001.

[Tur37] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1937.

[Tur50] Alan Turing. Computing machinery and intelligence. *Mind*, pages 433–460, 1950.

[Xil02] Xilinx, Inc., www.xilinx.com. *Virtex-II Pro Platform FPGA Handbook*, 2002.

[Xil07] Xilinx. *Virtex-II Pro and Virtex-II ProX Platform FPGAs: Complete Data Sheet*, November 2007.

[ZBS+06] Quan Zou, Yannick Bornat, Sylvain Saïghi, Jean Tomas, Sylvie Renaud, and Alain Destexhe. Analog-digital simulations of full conductance-based networks of spiking neurons with spike timing dependent plasticity. *Network: Computation in Neural Systems*, 17(3):211–233, 2006.

Acknowledgments

I want to express my gratitude to everyone who supported this work.
All the hardies and softies of the Electronic Vision(s) group and the friendly atmosphere they provided. Thank you for all the proofreading and support!

Especially I want to thank

Prof. Dr. Karlheinz Meier and Dr. Johannes Schemmel for their commitment that makes this fascinating project and my small contribution to it possible.

Prof. Dr. Udo Kebschull for providing a second opinion for this thesis.

Dr. Stefan Philipp for many discussions, ideas and support. And of course his high-quality VHDL code.

Dr. Andreas Grübl for giving the initial ideas and his help especially with *Spikey*.

Ursula Ernst for extensive proofreading and general support.

My family.

Erklärung:

Ich versichere, daß ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den 18.09.2009

.....
(Unterschrift)