RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

Lukas Pilz

Towards Precision Measurements of the BrainScaleS On-Wafer Communication Infrastructure

Internship report

KIRCHHOFF-INSTITUT FÜR PHYSIK

Abstract

The Electronic Visions group is developing the neuromorphic BrainScaleS wafer-scale system, which uses an on-wafer short range communications system named Layer 1 (L1). The aim of this internship was to build a software test, which is able to verify and quantify L1 functionality. This test uses the vertical repeaters to receive artificially generated spikes sent via the L1 bus and thus checks for errors in the on-wafer communication. The quantification is achieved by using the highly accurate Playback Memory module which is being developed by a collaborating group at TU Dresden and part of the Kintex-FPGA setup used for I/O purposes. The main challenge was the implementation of new communication protocols in the existing software stack and the debugging of the Field-Programmable Gate Array (FPGA) setup because this technology hadn't been used in this way before.

Zusammenfassung

Die Electronic Visions Gruppe entwickelt das neuromorphe wafer-scale BrainScales, welches einen wafer-internen Kommunikationsbus für kleine Reichweiten namens L1 hat. Das Ziel dieses Projektpraktikums war die Entwicklung eines Softwaretest, welcher in der Lage ist die L1-Funktionalität zu verifizieren und quantifizieren. Dieser Test benutzt die Vertikal-Repeater um künstlich generierteSpikes, welche über den L1 Bus gesendet werden zu empfangen und überprüft diese dann auf Fehler. Die Quantifizierung wird über die Benutzung des sogenannten Playback-Memory-Moduls erreicht, welches von einer Forschungsgruppe der TU Dresden als Teil des für I/O Zwecke benutzten Kintex-FPGA-setups entwickelt wird und eine sehr hohe Genauigkeit garantiert. Die größte Herausforderung war die Implementierung neuer Kommunikationsprotokolle in den existierenden Softwarestack und das Debuggen des FPGA-Setups, da diese Technologie noch nicht in dieser Art und Weise benutzt wurde.

Contents

1 Introduction

The building blocks of the BrainScaleS wafer system are the 384 High-Input Count Analog Neuronal Network Chips (HICANNs). Among other circuits, they consist of the synapse block, the respective output buffers to buffer postsynaptic spikes and the merger tree to merge all of them onto the L1 inter-HICANN communication bus. The signals coming from the merger tree get injected onto the L1 by the so called sending repeaters, which are placed horizontally on one data line and simultaneously drive the signals coming from other HICANNs. "The digital controller of the HICANN uses a clock frequency [...] generated by an internal PLL (Phase locked loop) from the external reference clock" [*Schemmel et al.*, 2010, chapter 2.2.1]. This Phase-Locked Loop (PLL) frequency can be set to values between 100 MHz and 250 MHz, which is suspected to influence the signal quality by making bitflips more likely.

The repeaters in general are used to restore signal amplitude and timing precision of signals. They include a Delay-Locked Loop (DLL), to which the signal is being resampled before being passed on along the bus (cf. *HBP SP9 partners* [2014]). This DLL however first has to "learn" the length of a packet by locking on spikes with neuron number 0, which takes some time. It is possible to operate the repeaters in debugging mode, where they store the first three signals (addresses and times) in an internal Static Random Access Memory (SRAM), which can be read out.

Rather then using 'real' (on-chip-neuron generated) spikes, one can either generate artificial spikes on the chip by using the Background Event Generators (BEGs) or send a predefined spike-train from the computer down to the wafer using the Field-Programmable Gate Arrays (FPGAs) Playback Memory (PbMem). The BEGs can be configured to send spikes with a certain inter spike interval, but the start of the firing process can't be timed exactly. The spike's timing however can be exactly defined as they are held in PbMem with a timestamp, which tells the FPGA when to release it.

Apart from spikes the Playback Memory can also hold HICANN configuration packets, which are used to insure precise timing of configuration of HICANN components. The structure in which the spikes are held in PbMem is called a "Pulse Group" and consists of the FPGA timestamp (when the FPGA is to release this group), the number of spikes therein and for each spike the target address and HICANN timestamp. HICANN configuration packets are held in a similar group structure with a FPGA timestamp, a variable which holds the number of configuration packets in this group and the configuration packets themselves. These packets get assembled in the lower levels of the software stack and then loaded in the Playback Memory, where the program they compound can be started at will.

The software stack consists of the high-level Python for the Hybrid Multiscale Facility (PyHMF) layer (a custom PyNN (PyNN) implementation), where neurons and weights can be defined and networks can be constructed. This is being used by the mapping layer called marocco to generate whole HICANN configurations, which are being passed down to the Stateful Hardware Abstraction Layer (StHAL). This container uses Hardware Abstraction Layer Backend (HALbe) coordinates and datastructures to call HALbe-Backend functions, which then configure the hardware using the hicann-system layer. There these commands are getting converted to bit-configurations and network packets. As my test is a low-level hardware test it is located in HALbe and changes were made to HALbe and hicann-system along this work (for further information cf. *Müller* [2014]).

The L1 test I implemented has the aim to verify and quantify the L1 functionality. It sets up a connection from a Sending Repeater to a Vertical Repeater and then tests it by sending spikes along this route and reading the Vertical Repeater's SRAM to see whether or not they reached it. This is done along all possible connections and with different PLL frequencies and can be used to study the influence of the PLL frequency on signal quality. Further quantifications to be investigated with this test would for example include the locking behaviour of the repeater's DLLs. As the locking process is highly time-critical, exactly timed HICANN configuration commands are required to investigate this.

2 Changes made to the Software Stack

2.1 Callstack for asynchronous usage

Until now the HICANN configuration was purely asynchronous (without exact timing insured by PbMem) and only the spikes were buffered in the Playback memory. In the future it will be possible to use the PbMem also for time-critical configuration (e.g. in L1 tests) and eventually for the majority of experiments. In this section I firstly present the current workings of the software stack (cf. Figure 1) and then the changes done made to insure PbMem integration (cf. Figure 2).

Because this software stack grew historically from only the hicann-system layer, there is no exact layering and structure between HALbe and hicann-system as of now. In the future it is planned to move all hardware abstraction to the HALbe layer and use hicann-system only for communication protocols and bit formatting of commands. For now there are instances of abstracted objects in HALbe as well as in hicann-system. The usual commandflow of the configuration of a repeater block works as follows.

In HALbe-test a repeater block coordinate and an instance are initialized and the instance is modified (configurations are set etc.), then these objects get passed on to the `write_repeater_-block` subroutine in `HICANNBackend.cpp`. In `HICANNBackend.cpp` an instance of the hicann-system `repeater_control` class gets initialized and the values which were set get extracted from the `RepeaterBlock` instance. These get then packaged into multiple write commands which are sent by using the `write_data` method of `repeater_control`.

In this method conversions are being made and the `write_cmd` method of the base class `ctrlmod` is called. There the command is reformatted and passed on to the `s2ctrl` subobjects' `issueCommand` method, which immediately forwards it to the respective derived `s2comm` member. Normally `s2c_jtagphys_2fpga_arq` gets this command and sends it down asynchronously via `host_al_-controller`'s `addHICANNConfigBulk` method.

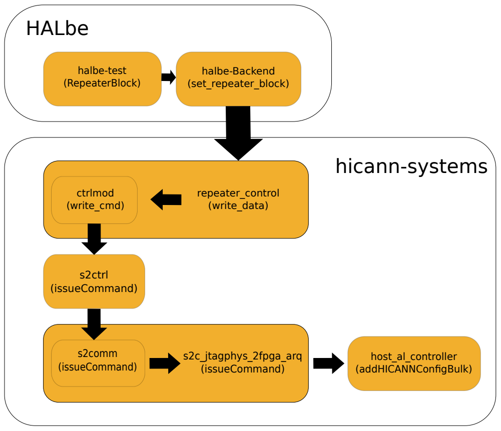
HALbe
halbe-test (RepeaterBlock)
halbe-Backend (set_repeater_block)
hicann-systems
ctrlmod (write_cmd)
repeater_control (write_data)
s2ctrl (issueCommand)
s2comm (issueCommand)
s2c_jtagphys_2fpga_arq (issueCommand)
host_al_controller (addHICANNConfigBulk)

Figure 1: Image of existing callstack for asynchronous usage

2.2 Callstack after Pbmem integration

Changes on several layers to implement the usage of the PbMem for configuration data were done. The major difference between the currently implemented Playback Memory spikes and HICANN configuration data is, that there is a delay after each command due to a non-zero execution time on hardware. The execution of this command may not be interrupted by another command, because is possible that the commands do not arrive correctly at the specified components. To track this implicit delay a second time variable (`next_playback_reltime`) had to be implemented in `host_al_controller`.

Because of future plans to extend the capabilities of HALbe and reduce hicann-system to a mere communications layer all changes presented here are preliminary. On the most low-level layer (in `host_al_controller`) the existing `addPlaybackConfig` method was fixed and improved, which now takes the delay, time (in cycles), payload and HICANN number and first checks if the minimal distance to the last packet is met. Then it has to send overflow packets if the time (due to an overflow in the 14 bit FPGA time counter) is greater then `systime_ctr/2` to signal the FPGA, that the timing is correct. Afterwards it has to override the `last_playback_reltime` variable with the `fpga_time` of the current package and increments the `next_playback_reltime` by the given delay time.

The `host_al_controller` achieves this by using a modified incrementation method, which doesn't only increment the time but also checks if the delay is bigger then the systime counter. In this case an overflow packet has to be sent to the FPGA to catch the overflow. In general overflow packets tell the FPGA to wait a specific number of systime counter overflows before it is allowed to release the next packet. Other implemented methods are `get_playback_reltime`, which can return either the `next_playback_reltime` or `last_playback_reltime` and a reset method, which defaults the two `reltime`s to 100 and 0 respectively to avoid possible glitches in the PbMem.

In the layers above `host_al_controller` methods were implemented for compatibility reasons allowing to call this function with respect to the aforementioned layering. To give the user the option to use the Playback Memory a private `bool` variable (`config_via_pbmem`) was introduced in the implementation of the `ctrlmod` class, which is the base class of all hardware abstracted objects. This variable can be changed by calling the `togglePbmemConfig` method which is inherited by e.g. the repeater and neuron control classes. On the same layer the `write_cmd` and `read_cmd` methods were changed to check whether or not the `config_via_pbmem` parameter is set and to use the `issueCommandPbmem` method of the communications classes if this were the case, which in turn eventually calls the `addPlaybackConfig` method of `host_al_controller`. This effectively provides a way to send the commands of every control module via the Playback Memory and to turn this feature on and off at will.

The aforementioned delay due to busy hardware components or bottlenecks in the communication was implemented in the `repeater_control` and `neuron_control` classes as a private variable and is being passed down from there by the `write_cmd` and `read_cmd` methods. In the future this delay has to be investigated further and possibly split up in multiple delays depending on the commands and repeaters in question for optimal access speeds.

The big difference in experiment design between using the Playback Memory and sending down the command asynchronously is the time at which answers to read commands are being received by the host. For example, if some values are to be read out of a register until now the asynchronous method sends down the respective command, then waits for its execution and blocks everything else until it has received an answer. When using the Playback Memory all these commands have to be stored in its program and then executed one after another at experiment start. Thus issuing commands and evaluating return data is decoupled. Here the chip (and communications channels) has to guarantee the order of the answers as the packets do not have any identification which would enable the software to sort them by itself.

This difference in design was implemented by isolating the methods, which send the read command and receive the answer from the combined `read_data` method in the respective control classes. This also required the same isolation being done in some of HALbes functions like `get_repeater_block` or `get_repeater`, which were split into `read_repeater[_block]` and `receive_repeater[_block]`.

To implement the possibility of using the Playback Memory in HALbe several functions in `HICANNBackend.cpp` and `HICANNBackendHelper.cpp` were changed to incorporate arguments like the `bool` variable whether or not to use the Playback Memory and the release time for the first packet. As the `read_repeater_block` and `set_repeater_block` functions for example consist of many commands being sent, there has to be an automatic incrementation of the release time, which is passed on to the send methods. However to not break existing use cases and provide a smooth integration into the existing software stack these variables are defaulted to asynchronous use.

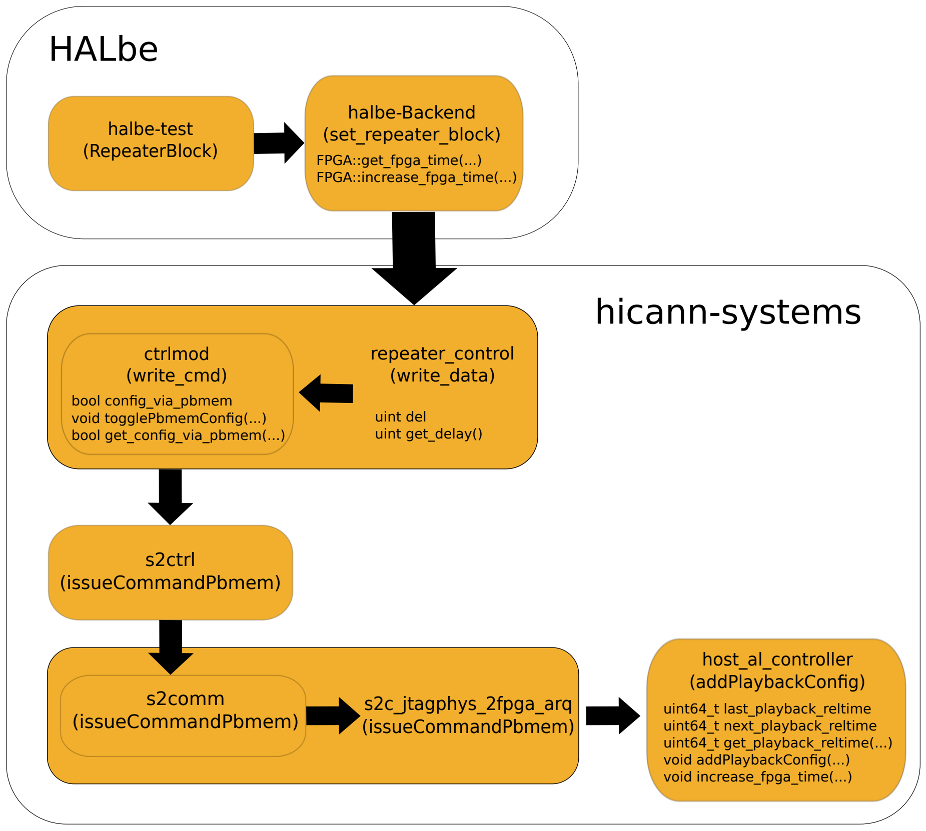

Figure 2: Image of callstack after integration of PbMem. The indexed text shows implemented functions and variables.

3 L1 Vertical Repeater Transmission Test

3.1 Test implementation

As mentioned in section 1 this test is used to verify and quantify the L1 functionality. As such there is the test function itself, in which the address of the spikes as well as the delay with which to release them and other parameters are defined. This function loops over different PLL frequencies and within each frequency over all possible connections to vertical repeaters. These connections are then given to the actual test function, which returns an object containing the connection parameters (Sending Repeater and Vertical Repeater), the received addresses and respective times. This information is then stored in a results vector, which is afterwards used to generate the statistics.

This test uses the HALbe API; it is based on an existing asynchronous test written by Sebastian Schmitt, which uses the Python API of HALbe which was modified to be able to use the Playback Memory. The Test does a number of configuration steps before sending down the spikes and read commands. A bool variable `use_pbmem` was implemented which, as the name implies switches between Playback Memory and asynchronous mode.

First of all the `HICANN::Init` is being executed to set up the connections (high speed links etc.). Afterwards the merger tree is configured and one of the BEGs is turned on and provided with the address and the period with which to generate the spikes. The Digital Network Chip (DNC) merger configuration is adjusted, so that they merge the incoming spikes from the Playback Memory with the BEG events onto the L1 bus. Then the gigabit link to the reticle itself is configured and the directions of the DNC mergers get set. Furthermore the crossbar switch corresponding to the respective connection and the input direction of the vertical repeater are being set.

Now the repeater's full flag has to be reset and the dllresetb register has to be resetted to enable locking. After waiting for the repeaters to lock, the BEG used for locking has to be turned off (in PbMem mode) or set to the correct data address (in asynchronous mode). To avoid jamming the FPGA a `HICANN::flush` was inserted, which insures the cleanliness of the Host and HICANN ARQ (cf. *Müller* [2014]). The repeaters' start `testdata` input bit now is being set to true and thus the repeater is ready to record incoming traffic.

After sending the spikes down a read command for the repeater block is being sent. As this is a PbMem test, all timing-critical commands are actually buffered in the Playback Memorys external Double Data Rate Synchronous Dynamic Random Access Memory (DDR3-SDRAM) and a start signal has to be given. After the experiment is finished, the FPGA has to be stopped and the data, which is received from the HICANN has to be read out and then processed. After it being stored in the `RepeaterBlock` variable, the experiment is evaluated.

There are three possible outcomes: the time of the event is

- greater then zero
- equal to zero and the full flag is set
- equal to zero and the full flag is not set

In the first case the spike was successfully received with the respective address. In the second and third case the transmission wasn't successful due to a counter overflow in the repeater's SRAM register (which can just mean spikes don't come fast enough, but they are valid nonetheless) or no spike at all being received. Having this in mind a passed experiment is defined by all spikes being received (so counter overflow is not an error) and them having the right addresses (when the operation is more stable a check on the times could also be defined to see, whether they have the right distance (with some tolerance)).

The statistics generated after all the PLL frequencies with all connections have been tested include the total number of connections versus the number of successful tests (also in percent), the output of all failed connections with the type of error and what went wrong as well as three most frequent false addresses. After the PLL frequency wise output, there is a list of connections sorted by the number of fails, which aims to help the user to detect transmission errors.

A file output was implemented, which stores the name of the connections in the first line followed by the information if there was a fail on this connection per test run in a Comma seperated values file (CSV) and a directory, which is created from the current timestamp at the beginning of each test. For each PLL frequency the test results are stored in a separate file, which can be evaluated using a python script. This script is located in the results directory and takes the directory of the data, the wafer number and the FPGA number as commandline arguments.

For debugging purposes a switch RAM test (`SwitchRAMHWBackendTest`) was implemented and the `VerticalRepeaterHWTest` was changed to support Playback Memory execution. Both tests essentially generate random data, which is written in the respective configuration SRAMs, read out again via Playback Memory and then checked against the generated data.

3.2 Test results

For the test to run correctly it is imperative to make sure, that all the external voltages (particularly VOL and VOH) are set correctly or otherwise there will be a high error rate. After this configuration the results from the test will (as expected) vary from HICANN to HICANN. Exemplarily a run of a PLL frequency sensitive HICANN (cf. Figure 3), a HICANN with a great likelyhood of hardware defects (cf. Figure 4) and a HICANN without hardware defects (cf. Figure 5) will be shown.

In the given figures, the y axis enumerates the test run and the x axis identifies the tested connection. The color of each connection is dependent on the number of fails of this connections over the PLL frequencies.

Figure 3 shows a HICANN, for which there are fails for a couple of repeaters at some PLL frequencies (namely 100 MHz and 250 MHz). We can see here, that all the connections, which show errors show them only at a few PLL frequencies. However, the failing component is not everywhere the same. One can see from the fail of the 0->60 connection and that none of the other connections from Sending Repeater 0 failed, that the component which is error-prone is somewhere behind the Sending Repeater and thus either the connection (due to bitflips, crosstalk or other problems), the switch or the Repeater 60 itself.

The next block of fails is seen in the connections coming from Sending Repeater 2 and only occurs at the PLL frequency 250 MHz. Here we see a pattern which is supposedly random in nature and points to the Sending Repeater itself, being sensitive to too high PLL frequencies as there are fails at all connections.

The third block of fails is the most interesting, as this points to sensitivities of the Sending Repeater to changes on macroscopic timescales like e.g. fluctuation in the supply voltage. It could also be related to an incomplete initialization of other HICANNs on the reticle, which is currently being investigated.

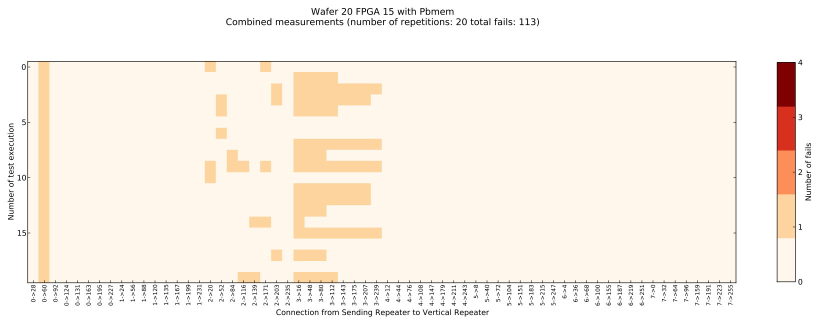

Figure 3: Visualization of the test results of a PLL frequency sensitive HICANN.

In Figure 4 one can see a broken Vertical Repeater, which fails every time (every PLL frequency and every repetition of the test) whereas all the other repeaters work fine.

And finally Figure 5 shows a HICANN, which works perfectly fine. Here all connections work on all PLL frequencies.

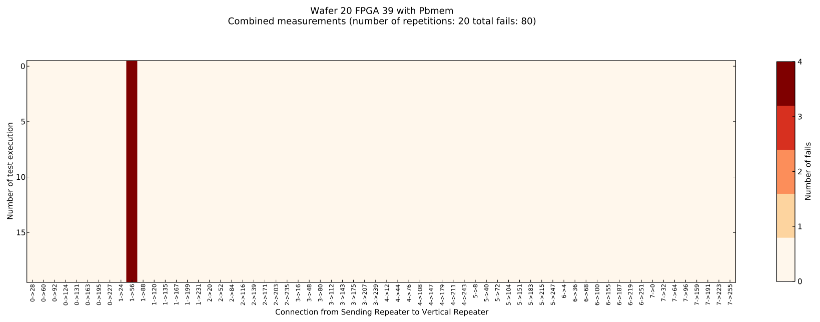

Figure 4: Visualization of the test results of HICANN with possible hardware defects.

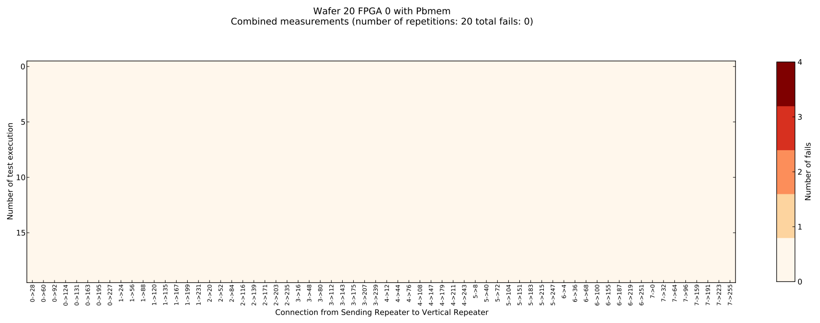

Figure 5: Visualization of a HICANN without hardware defects.

4 Discussion & Outlook

The final implementation of Playback Memory usage in the software stack will take much more effort, because big chunks of it have to be rewritten and rethought. As outlined above, there will be a new communications mode in the future, which will take care of the bit-formatting for Playback Memory and thus the `host_al_controller` class will be made obsolete. Existing code, which implements pipelined spike sorting written by C. Mauch will be integrated and extended to provide these functionalities. Also some of the HALbe functions have to be amended, because the PbMem case requires at least the `fpga_release_time` and at best also the `use_pbmem` variable.

Over the course of designing this Playback Memory test several roadblocks in the form of bugs in the FPGA code were encountered. At first there were problems because of implicit assumptions over the sequence of configuration packets, spikes and overflow packets, which were reported and later fixed (personal communication). For bug reports the tools tshark and Wireshark were used to monitor the packages going from host to FPGA and vice versa. This took an extraordinarily long time as one had to gather data over these bugs, which were only occurring at a specific length of the spiketrain in the Playback Memory. Another bug was found using the modified `VerticalRepeaterHWTest`, which occurs when the delays between the configuration packages are too small. This issue remains unresolved and is circumvented by using very large delays between packages (cf. *UHEI* [2016]).

Although many bugs were found and fixed, the test remains unstable. Occasionally the FPGA will drop repeater block read packets and thus the exception "No HICANNConfig packet received" will be thrown in the software, because the answer to a read command is missing. For the moment also this will remain unresolved and is handled using an exception handler in the software, which catches this exception and retries this connection 10 times until eventually the exception is rethrown. Although I suspect that this has to do with timing difficulties in the FPGA and possibly the aforementioned bug, investigations are ongoing as of now.

Glossary

BEG	Background Event Generator. 8
CSV	Comma seperated values file. 9
DDR3-SDRAM	Double Data Rate Synchronous Dynamic Random Access Memory. 8
DLL	Delay-Locked Loop. 2, 3
DNC	Digital Network Chip. 8
FPGA	Field-Programmable Gate Array. I, 2, 8, 9, 12 2, 5.
HALbe	Hardware Abstraction Layer Backend. 3–6, 12
HICANN	High-Input Count Analog Neuronal Network Chip. 2–5, 8–11
HICANNARQ	HICANN ARQ protocol. 8
L1	Layer 1. I, 2–4, 8
marocco	marocco. 3
PbMem	Playback Memory. 2, 4, 5, 7, 8, 12
PLL	Phase-Locked Loop. 2, 3, 8–10
pyHALbe	Python implementation of the Hardware Abstraction Layer Backend. 8
PyHMF	Python for the Hybrid Multiscale Facility. 3
PyNN	PyNN. 3
RAM	Random Access Memory. 9
SRAM	Static Random Access Memory. 2, 3, 9
StHAL	Stateful Hardware Abstraction Layer. 3

A Bibliography

HBP SP9 partners, *Neuromorphic Platform Specification*, Human Brain Project, 2014.

Müller, E. C., Novel operation modes of accelerated neuromorphic hardware, Ph.D. thesis, Ruprecht-Karls-Universität Heidelberg, hD-KIP 14-98, 2014.

Schemmel, J., A. Grübl, and S. Millner, Specification of the HICANN microchip, FACETS project internal documentation, 2010.

UHEI, T., FPGA meeting log (https://brainscales-r.kip.uni-heidelberg.de/projects/hmf-fpga/wiki/fpgameetinglog_20160531), 2016.