Paktikumsbericht

Maximilian Denne

18.04.2013 - 27.06.2013

Vermessung von Floating Gates auf Schnelligkeit und Reproduzierbarkeit

Inhaltsverzeichnis

1 Motivation

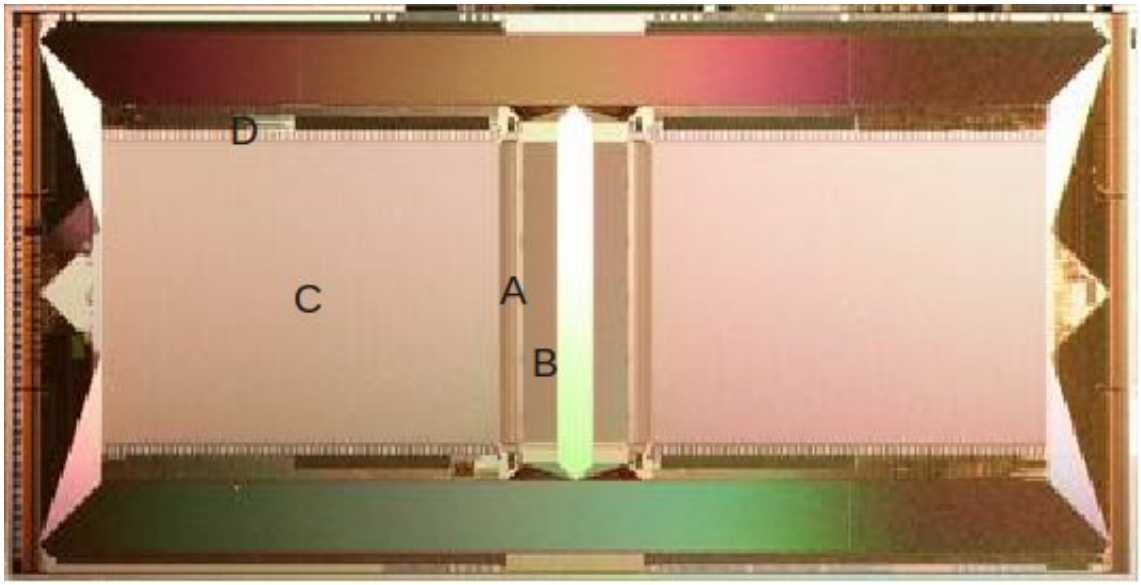
Ich meinem auf 8 Wochen angesetzten Projektpraktikum in der Forschungsgruppe F9 Electronic Visions, sollten meine Aufgaben zu Beginn die Einarbeitung in die Programmierung mit Python sein, um dann mich an die eigentliche Aufgabe, die Optimierung der Programmierparameter der Floating Gates unter Berücksichtigung gleichbleibender Standardabweichung, zu widmen. Im Laufe des Praktikums untersuchte ich dann insbesondere die Programmierschnelligkeit bei sich minimal änderten Genauigkeit. Nach diesen Messungen und Auswertungen dieser, sollte eine weitere Aufgabe sein, die Reproduzierbarkeit von festen Floating Gate Werten für einen kompletten Block zu untersuchen. Diese zwei Hauptaspekte sollte den Inhalt meines Praktikums bilden. Im Folgenden zeige ich zur Wiederholung und allgemeinem Verständnis die Grundlagen der Floating Gates auf, deren Architektur, Schaltbilder und Funktionsweise. Darauf folgend der von mir erarbeitet und programmierte Teil, mit anschließender Resultatspräsentation.

2 Grundlagen

Zur kurzen Einführung zeige ich in diesem Abschnitt die Grundstruktur des HICANN-Chips sowie die Architektur des Floating-Gate-Blocks, deren Grundschaltung, Funktionsweise bzw. Programmierweise und die Programmierparameter.

2.1 Übersicht über den HICANN-Chip

In Abb. 1 ist ein Hicann-Chip zu sehen, worauf die wichtigsten Areale markiert sind. Der Chips weist eine Grösse von $0.5cm^2$ auf. Im Bereich (A) ist der Neuronenblock gezeigt und in (B) die dazugehörigen X Floating-Gates. (C) und (D) zeigen die Synapse beziehungsweise die Synapsendriver welche die Synapsen steuern. Ingesamt befinden sich 256 Neuronen auf einer Chiphälfte sowie 129 x 24 Floating-Gate-Zellen. Hinzu kommen 256 x 224 Synapsen sowie 2 x 56 Synapsendriver. Die andere Hälfte des Chips ist in der selben Weise aufgebaut. Diese Daten stammen alle aus der Dissertation von Sebastian Millner (Quelle [2]).



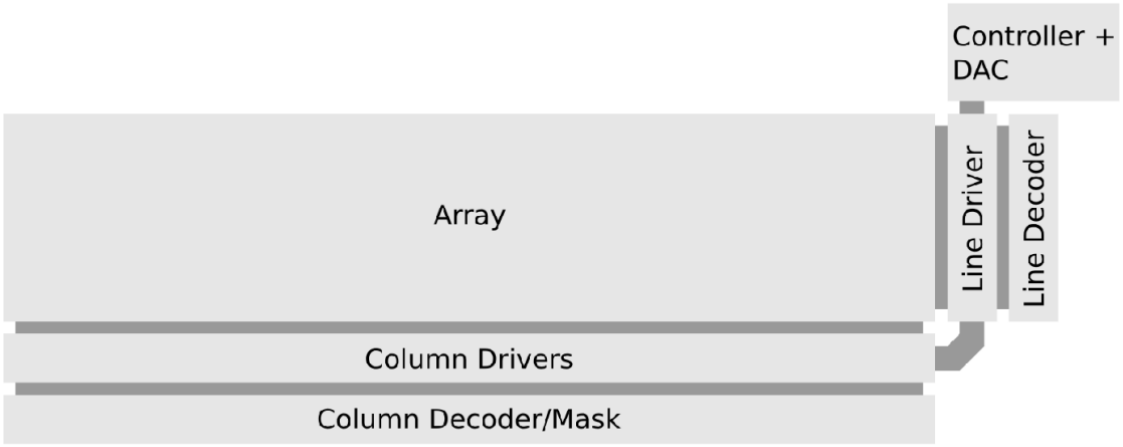
Quelle: Electronic Visions [1]

Abbildung 1: Übersicht eines HICANN-Chips

Insgesamt befinden sich 384 von diesen Hicann-Chips auf dem Wafer. Somit ergibt dies knapp 200 000 Neuronen und mehr als 40 Millionen Synapsen. Im Vergleich dazu liegt die Zahl der Neuronen im menschlichen Gehirn etwa 100 Milliarden und die der Synapsen sogar bei etwa 100 Billionen. (Quelle: www.wikipedia.de)

2.2 Floating-Gate-Block

Der Floating Gate Block besteht aus 24 Zeilen, da es eben genau 24 Neuron-Parameter (12 Spannung, 12 Strom) gibt. Hinzu kommen 128 Spalten individuell für die Neuronen eingeteilt und einer Spalte die für alle Neuronen besteht. Diese Anordnung ist in Abb. 2 dargestellt.



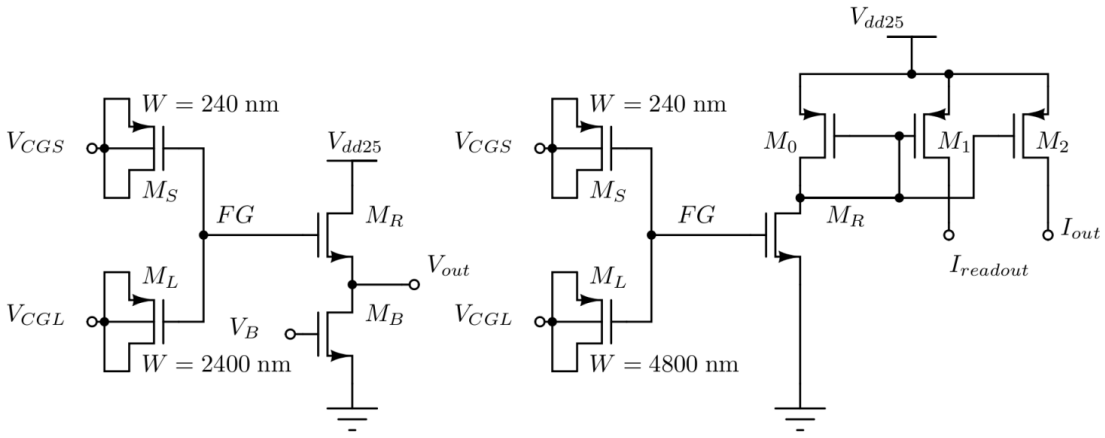
Quelle: Sebastian Millner [2]

Abbildung 2: Architektur des Floating-Gate-Blocks

Die Zeilen und Spalten werden über die Driver gesteuert und programmiert. Beide sind mit dem Controller und einem DAC (digital-to-analog converter) verbunden. Die bei dem Spalten-Decoder erwähnte Maske dient zur Programmierung und regelt an welche Spalte nun die Programmierspannung angelegt werden soll und an welche nicht.

2.3 Schaltungsbilder

Hier in Abb. 3 sind zum Einem die Spannungszellen Abb. 3(a) und zum Anderen die Stromzellen Abb. (b), wie sie auf dem Hicann implementiert sind zusehen. Bei dem Spannungsschaltung ist zu erkennen, dass die Auslesespannung nur über einen Auslesetranisitor M_R kommt. Dieser bildet mit dem Transistor M_B einen Sourcefolger und steuern die Spannung für die Neuronen, sowie V_{out}. Der Auslsesestrom bei den Stromzellen erfolgt über einen Stromspiegel. Bei dieser Schaltung ist M_R direkt mit Masse verbunden, und der erzeugte Strom wird über M_0 und M_2 zu den Neuronen gespiegelt. M_1 dient hierbei wieder als Auslesetransistor für den Strom I_{out}.



(a) Voltage cell

(b) Current cell

Quelle: Sebastian Millner [2]

Abbildung 3: Schaltbilder der Floating-Gate Zellen

V_{CGL} und V_{CGS} stellen die jeweiligen Reihen- bzw. Spaltenspannungen. Hierüber werden die Floating-Gates programmiert, indem eine Spannungsdifferenz angelegt wird. Mit Spannungspulsen dieser Differenz schreibt man die gewollte Spannung auf die Floating-Gates. Für diesen Vorgang gibt es bestimmte Parameter.

2.4 Programmierparameter

Es gibt insgesamt 8 Parameter, mit denen die Genauigkeit und Geschwindigkeit des Schreibvorgangs geändert werden können.

Maxcycle: Die Floating-Gates werden im Grunde erstmal auf "gut Glück" programmiert und erst nach dem Schreiben wird der Wert verglichen (mittels Komparator). Es wird nun so solang geschrieben bis der Wert mit dem Sollwert übereinstimmt, sollte er dies nach Maxcycle-Durchgängen nicht tun, wird abbgebrochen und zur nächsten Zelle weitergegangen.

fg_bias: Steuerstrom

fg_biasn: Dieser Parameter kontrolliert die Spannung am V_B-Transistor

Pulselength: Dieser Parameter bestimmt die Zeitskala, die für grundlegende Operationen (wie z.B. Schreibpulse oder Lesezyklen) verwenden, durch ändern der Controller-Uhr.

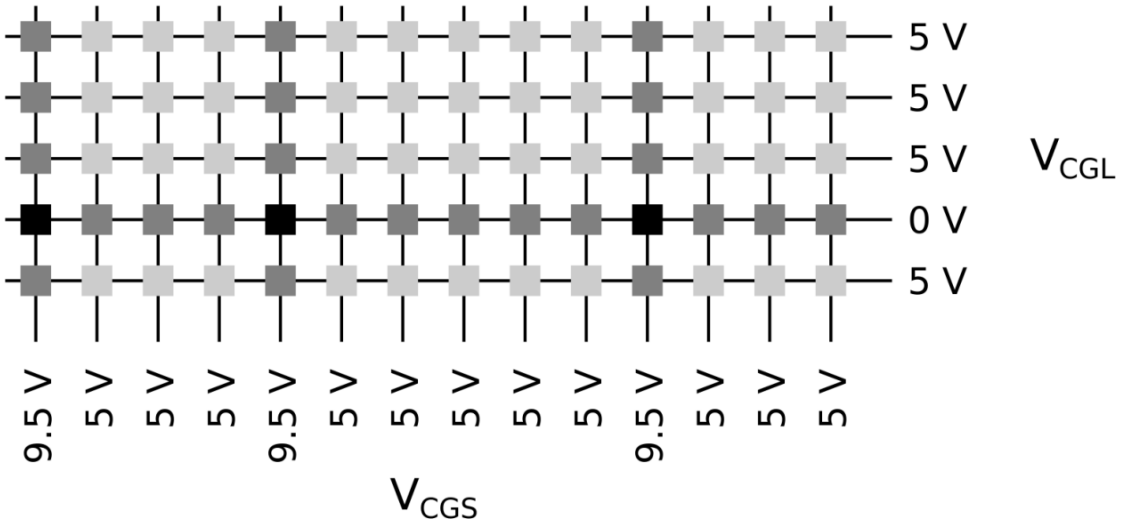
Readtime: Wartezeit bist die Leitung vom Flaoting-Gate bis zum Komparator aufgeladen ist, somit misst der Komparator nicht zu früh und mittelt falsche Werte.

Voltagewritetime bzw. Currentwritetime: Länge der Schreibpulse in Taktzyklen, wobei es für jede Zellenart einen Parameter gibt.

Acceleratorstep: Der Tunnelstrom wird schwächer, wenn der Wert eine Zelle sich erhöht. Dieser Umstand macht es schwierig hohe Werte zuerreichen. Deshalb wurde ein Parameter implementiert, welcher nach einer bestimmten Anzahl an Schreibzyklen, in dem der Sollwert nicht erreicht wurde, die Schreibpulse verdoppelt.

2.5 Programmiervorgang

Dieser Abschnitt soll verdeutlichen wie das Programmieren der Floating-Gates abläuft. Zum Veranschaulichen ist in Abb. 4 das FG - Array zu sehen.



Quelle: Sebastian Millner [2]

Abbildung 4: Floating-Gate Array. Die Quadrate repräsentieren die Zellen.

In dieser Grafik ist V_{CGS} mit 9.5 V angegeben, allerdings habe ich bei meinen Messungen 11 V verwendet. Will man nun eine bestimmte Zeile mit Werten schreiben, setzt man V_{CGL} auf 0 V für diese Zeile. Weiterhin wird für jede Spalte die eben erwähnten 9.5 V bzw 11 V angelegt. Alle anderen Zeilen werden auf 5 V gesetzt. Bei den Spalten, die nach dem ersten Schreibzyklus ihren Sollwert erreicht haben, wird dann nicht mehr 11 V, sondern auch die 5 V angelegt. Dies regelt die in 2.2 erwähnte Maske. So weiterführend werden die Zellen nach "oben" programmiert, bis jede Zelle ihren Sollwert erreicht hat.

3 Programmierter Code

Nach diesem Grundlagenüberblick komme ich nun zu dem Teil was ich selbst programmiert habe. Nach dem Erstellen der Gui mit PyQt, die im Folgenden genauer erklärt wird, musste ich mir ein Programm zum Schreiben und Auslesen der Floating-Gates zusammenbauen. Hierfür suchte ich den nötigen Teilcode aus schon vorhandenen Testskripten und Codes zusammen.

3.1 Die GUI

Abbildung 5: Die Gui

In Abb. 5 ist die von mir erstellte Gui zu sehen. Um später einfacher meine Messungen durchzuführen, bei denen die einzelnen Parameter meist in kleinen Schritten geändert werden mussten, sollte die Gui beim Durchsweepen helfen. Deshalb implementierte ich für jeden Parameter einen Slider und ein dazugehöriges Display. Die Range der Slider entsprach natürlich der Bitranges der Parameter. Eine Klasse "Speichern" im Code hielt die eingestellten Werte der Slider, um diese später an das Schreib und Ausleseprogramm zu übergeben zu können. Als Grundeinstellung der Slider wurden die default-Werte genommen. Der Button "Messung" startet die Messung, indem beim Drücken des Buttons, das Schreibprogramm gestartet wird. Um die gerade verwendeten Parametereinstellungen festzuhalten, printed der Button "Speichern" die Werte in das Terminal.

3.2 Schreib- und Leseprogramm

Aus den schon vorhandenen Codes, wie "test HICANNBackendHWTests.py", "test AnalogToADCHW-Test.py", ..., suchten wir dann die passenden Zeilen heraus. Ein Pseudocode ist unterhalb abbgebildet, der helfen soll die vorgehensweise zu verstehen und nachzuvollziehen.

```
def   test_HICANNInit:
        # aus  test_HICANNBackendHWTests.py

        # configure ADC
        # get calibration data of ADC
        # set Analog

        # setzen der Floating-Gate-Parameter über FG.Config!

for i in range(1,101):
        for n in range(0,128):
                # Random-Werte erzeugen von 0 - 1023
                # Neuron-Parameter auswählen und Digitalwert setzen

        # Werte schreiben mit set_fg_values (Zeitmessung implementiert)

        for m in range(0,128):
        # aus test_AnalogToADCHWTest.py

        # auslesen der Zellen mit get_fg_cell & get_trace

plot_voltage(Digitalwerte, Analogwerte)
# Gespeicherte Werte mittels Plotprogramm veranschaulicht!
```

Die Funktion "test_HICANNInit" konfiguriert den ADC, setzt die Kalibration sowie den HICANN. Darauf werden die FG-Parameter gesetzt. Zuerst wurden diese immer per Hand geändert, später dann verbunden mit der Gui und somit über die Slider verändert. Die darauf folgenden Schleifen werden in Abschnitt 4 "Messmethoden" genauer erklärt. Die ist das Grundgerüst zum Schreiben und Auslesen, nun wird je nach Messung den dafür benötigte Implementierung hinzugefügt oder abgeändert. Nach dem erstellen dieser beider Code und einigen Probemessung wurde, nun die Gui und das gerade eben erwähnte Programm verbunden und die Messungen konnten nun so durchgeführt werden, wie man sich das zu Beginn vorstellte. Die einfache Handhabung mit der Gui erleichterte die doch recht häufigen Messungen mit verschiendenen Parametereinstellungen.

4 Messmethoden

In diesem Abschnitt erläutere ich meine Vorgehensweise bei den einzelnen Messungen, wie ich die benötigte Codes implementierte und die gesammelten Daten verarbeitet habe. Da es zwei Hauptaspekte bei meiner Arbeit gab, teile ich diesen Abschnitt in zwei Teile.

4.1 Teil 1: Programmierschnelligkeit

Im ersten Teil meiner Messungen war meine Aufgabe den Satz an FG-Programmierparameter so zu optmieren, dass die Programmierschnelligkeit minimiert wird. Dabei sollte natürlich die Standardabweichung nicht aus den Augen gelassen werden und sollte sich nicht wesentlich ändern. Hierfür habe ich hier nochmals den Code aufgeführt der mit die benötigten Daten lieferte.

```
for i range(1,101):
        for n in range(0,128):
                # Random-Werte erzeugen von 0 - 1023
                # Neuron-Parameter auswählen und Digitalwert setzen
        # Werte schreiben mit set_fg_values( Zeitmessung implementiert)
        for m in range(0,128):
                # aus test_AnalogToADCHWTest.py
                # auslesen der Zellen mit get_fg_cell & get_trace
```

Es wurden Randomwerte von 0 - 1023 erzeugt, wobei für jede FG-Zelle ein neuer erzeugt wurde. Dann wählte man den Neuronparameter aus, wobei ich hierbei nur je einen Spannungs/- und Stromparameter nahm. Somit wurde in jede Zelle immer wieder ein andere Wert gesetzt und dann mit "set_fg_values" geschrieben. Gleich im Anschluss wurde die Ziele wieder über "get_fg_cell" ausgelesen und die Werte in einer Liste gespeichert. Dieser Vorgang wurde dann 100 mal wiederholt um eine gute Statistik zu bekommen. Somit bekamm man für jeden Digitalwert mehrere Analogwerte, die dann gemittelt wurden und die Standardabweichung berechnet wurde. Um dies zu veranschaulichen, erstellte ich Plots auf denen die Digitalwerte gegen die Analog aufgetragen wurden. Um immer einen Blick für die Standardabweichung zu haben, trug ich in einem Zusatzplot die Standardabweichung logaritmisch gegen die Digitalwerte auf. In Abb. 7 sieht man beispielhaft ein solchen Plot.

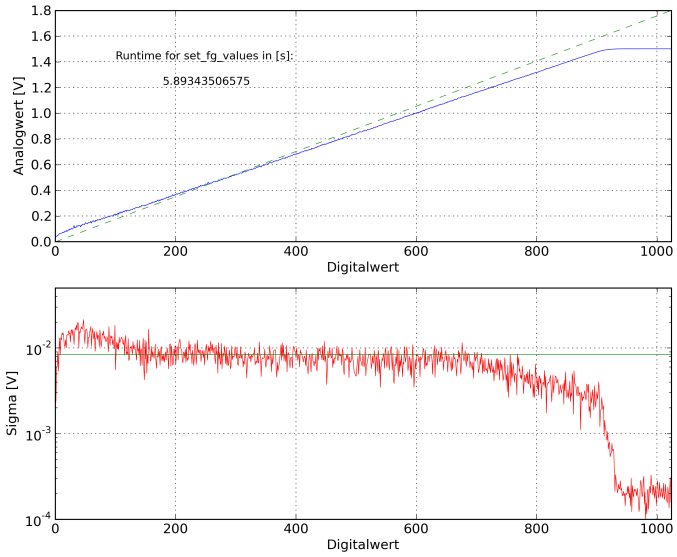


Abbildung 6: Plot zur Genauigkeit der Floating-Gates

Die gestrichelte grüne Gerade ist die Optimalgerade, also so sollte die Kurve im besten Fall ausehen. Hierfür habe ich angenommen, dass wenn man die Digitalwerte durch ihren Maximalwert 1023 und dann mit der maximal zu erwartenden Spannung der Floating Gates von 1.8 V multipliziert, die ideal Werte erhält. Da dies ein linearer Zusammenhang ist, erscheint im Plot die Ideallinie als Gerade. Auf was genau ich geachtet habe bei meine Messungen, wird in den Resultaten erläutert.

4.2 Teil 2: Reproduzierbarkeit

Im zweiten Teil ging es darum, eine Übersicht über einen kompletten Floating-Gate Block zubekommen. Dabei ging es genauer gesagt, um die Reproduzierbarkeit von festen FG-Werten. Auf der nächsten Seite wird dargestellt wie ich vorgegangen bin. Hierbei gibt es 2 wesentliche Unterschiede. Und zwar habe ich einmal das Programmieren der FG-Werte in die grosse Schleife implementiert, sodass bei jedem Durchlauf neu programmiert wurde. Die Messungen wurde für den festen FG-Digitalwert von 511 durchgeführt. Um zuzeigen, dass der Fehler grösstenteils durch das wiederholende Programmieren kommt, führte ich die Messung nochmals mit einmaligen Programmieren durch. Hierbei wurde die Programmierzeile vor die Schleife implementiert. Ansonsten ist die Auslese bei beiden Vorgehen die selbe. Mit 2 Schleifen, sowohl über die Zeilen (von 0 - 23) als auch über die Spalten (von 0 - 128) wird der FG-Block ausgelesen. Zum Speichern der aufgenommenen Daten habe ich noch eine Datenstruktur geschaffen mit der ich mir beliebige Daten herausziehen kann. Diese wird mit “pickle“ gespeichert und kann somit jederzeit wieder verwendet werden. Die macht es möglich die Messung nur einmal durchzuführen und dann trotzdem alle notwendigen Information zu behalten.

```
# aus HMFUtil.cpp

(# Setzen eines festen FG-Werts mit fg.block.fgarray für
   gesamten Block)

for i in range(1,51):
        (# Setzen eines festen FG - Wertes mit fgblock.fgarray für gesamten Block)

        # Werte schreiben mit set_fg_values

        for n in range(0,24):

                for m in range(0,128):
                # aus test_AnalogToADCHWTest.py

                # auslesen der Zellen mit get_fg_cell & get_trace

                try:
                    m[a][n].extend([value])
                    except IndexError:
                    try:
                        m[a].append([])
                        m[a][n].extend([value])
                    except IndexError:
                        m.append([])
                        m[a].append([])
                        m[a][n].extend([value])
```

Hiermit ist der Theorie/- und Programmierteil soweit abgeschlossen und somit komme ich nun zu den von mir ermittelten Ergebnissen.

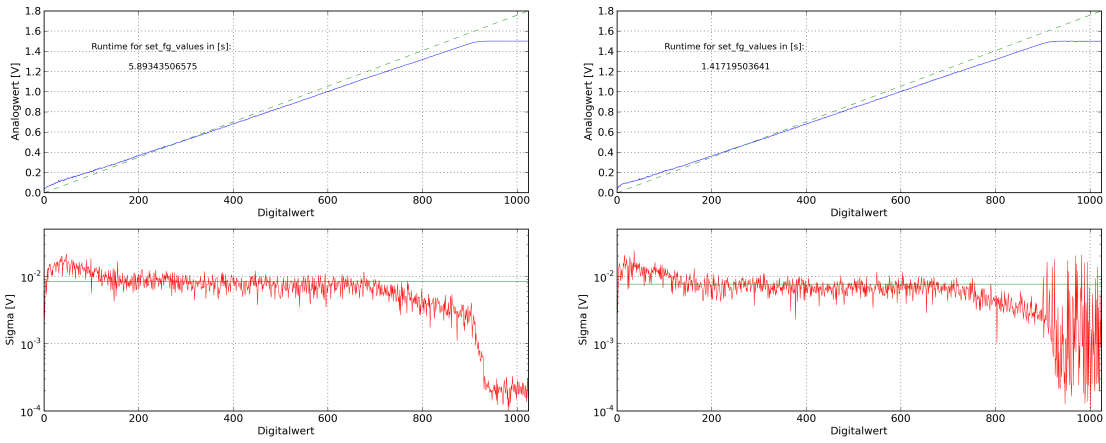
5 Resultate

In diesem Kapitel präsentiere ich nun meine Resultate. Auch hier habe dies wieder in zwei Abschnitte geteilt. Da ich zu Anfang meiner Arbeit die Spannungszeile E_syni zum Messen genommen habe, habe ich mit dieser auch den Hauptteil meiner Messungen gemacht und anhand deren Plots die Verbesserungen der Parametern gemacht. Trotzdem habe ich in Teil B noch zum Vergleich eine Stromzeile vermessen und hierbei auch mit den Parametern expermentiert.

5.1 Optimierung der FG-Parameter

5.1.1 Spannungszeile

Wie schon in Abschnitt 4 erwähnt, plotte ich die Digitalwerte gegen die gemessenen Analogwerte. Um dabei eine Maß für die Genauigkeit zuerhalten, habe ich die optimale Gerade und den Standardabweichungsplot zur Hilfe genommen. Somit kann ich diese immer zum Vergleich heranziehen. Nun habe ich einerseits beim Durchsweepen der Parameter darauf geachtet, dass die Standardabweichung, die ich direkt unterhalb auch geplottet habe, nicht sonderlich grösser wird, als auch ein Auge auf die Programmierschnelligkeit geworfen. Für den grössten Teil meiner Messung habe ich eine Spannungszeile (E_syni) vermessen. Aber im Folgenden werde ich auch in Teil B noch die Plots einer Stromzeile zeigen. In Abb. 7 a) sieht man den Plot bei den voreingestellten default-Werten mit logarithmisch aufgetragener Standardabweichung. Hingegen man in Abb. 7 b) den Graphen mit den von mir optimierten Parametern “Maxcycle“ und “readtime“. Ich habe mich im Laufe meiner Messungen auf diese zwei Parameter konzentriert, da ich hier die grössten Veränderungen feststellen konnte.



(a) Default Values: Maxcycle = 255 , Readtime = 63

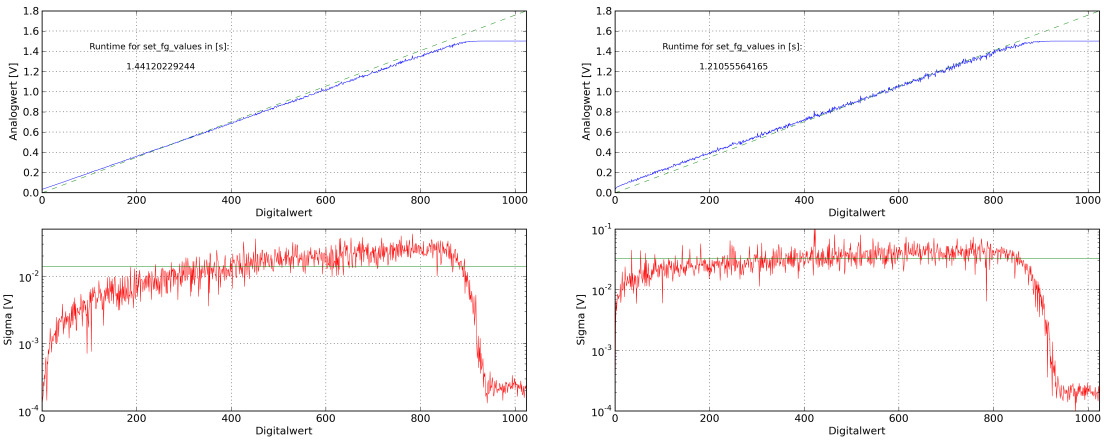
(b) Optmierte Werte: Maxcycle = 90 , Readtime = 15

Abbildung 7: Default Werte und die optimierten Werte im Vergleich

Wie man erkennen kann, sieht man dass die Standardabweichung sich nicht signifikant ändert, soll heißen, der Mittelwert der Standardabweichung schwankt leicht um etwa 9 mV. Die Programmierzeit, die im Plot dargestellt ist, um mehr 4 Sekunden gesunken ist. Da ich bei dem Parameter “Maxcycle“ in 10er Schritten mich dem Endwert näherte, kam ich zu der Erkenntnis das 90 der in meinen Augen beste Wert ist. Genauso ging ich bei “Readtime“ vor, wo ich in 5er Schritten vorging und somit bei 15 landete. Dies ist der Grund wie ich zu relativ “glatten“ Zahlenwerten kam.

5.1.2 Stromzeile

Zur Vollständigkeit und zum Vergleich lege ich hier die Graphiken eines Stromparameters (I_pl) dar. Hierbei habe ich einmal eine Messung mit den von mir optmierten Werte für Maxcycle und Readtime und den minimal Wert (1) für den Parameter “currentwritetime“ aufgenommen. Dies ist auch der default Wert.Und zum Anderen eine Messung mit dem Maximalwert (63) für eben erwähnten Paramter. Die daraus folgenden Plots sind in Abb. 8 zu sehen.



(a) Maxcycle=90 , Readtime=15 , currentwritetime=1 (b) Maxcycle=90 , Readtime=15 , currentwritetime=63

Abbildung 8: Optimierte Parameter mit minimal und maximalem Wert für currentwritetime

Wie man erkennt, ist die Standardabweichung bei dem Stromzeilen höher. Natürlich kann man dies jetzt nicht pauschal für alle Stromparameter sagen, da ich nur eine vermessen habe, aber jedoch greife ich diesen Punkt im nächsten Abschnitt nochmals auf. Eine mögliche Erklärung für die höhere Standardabweichung könnte jedoch sein, dass die Spannung einer Stromzelle über einen Tranistor in den Strom umgewandelt wird. Dieser wird dann über einen Messwiderstand als Spannung ausgelesen. Hierbei führt nun ein kleiner Fehler bei Stromwerte zu einem grösseren Fehler bei der ausgelesenen Spannung. Wie in der Abbildung dargestellt, verläuft die Spannung (U_out) linear. Nun ist der Drainstrom der current-Zellen proportional zum Quadrat der Floating-Gate-Spannung, somit wirken sich Fehler der Messspannung größer aus! Hinzu kommt noch ein Faktor 2.5 vom Messstrom, welcher proportional über eben erwähnten Faktor zusammenhängt. Daraufhin schwanken die Stromzellen mehr als die Spannungszellen.

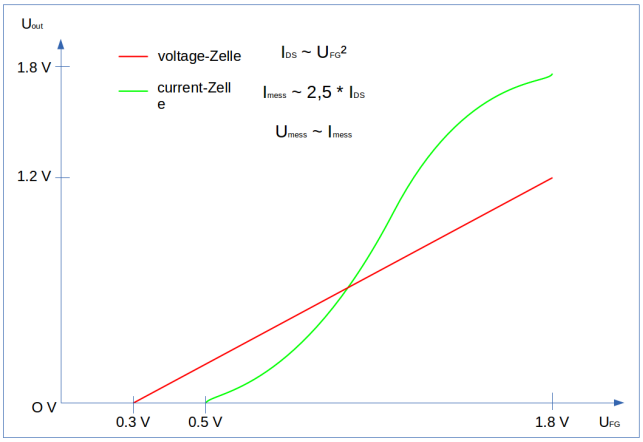


Abbildung 9: Vergleich der Auslesespannungen von Strom- und Spannungszellen

5.2 Genauigkeit eines FG-Blocks

Die in Abschnitt 4.2 erwähnten gesammelten Daten nutze ich nun um eine Übersicht über den kompletten FG-Block zu erhalten. Hierbei sind die Zeilen gegen die Spalten aufgetragen und als dritte Dimension ist als Farbplot die Standardabweichung jeder Zelle dargestellt. Auch Mittelwertplots und Histogramme einer einzelnen Zeile, sowie Histogramme einzelner Zellen war hiermit möglich.

5.2.1 Mehrfaches Programmieren

Zuerst ist hier der Farbplot der mit den Daten aus der mehrfach programmierten Messung aufgezeigt.

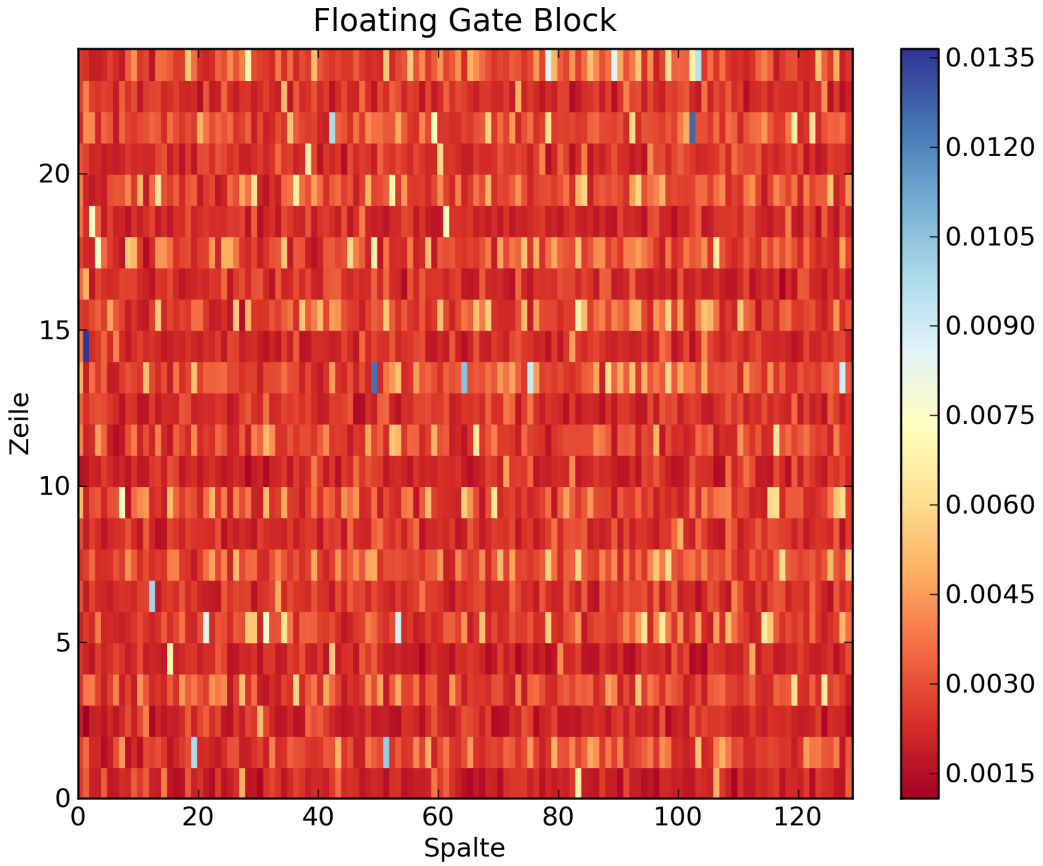


Abbildung 10: Farbplot des Floating Gate Blocks bei mehrfachen Programmieren (Farblegende in Volt)

Sofort zu erkennen sind die einzelnen Ausreisser mancher Zellen. Sowie dass die Stromzeilen doch mehr schwanken, als die Spannungszeilen. Deshalb habe ich sowohl Zeile 14 (Spannungparameter) als auch Zeile 3 (Stromparamter) genauer vermessen mittels Mittelwertsplot bzw. Histogramm.

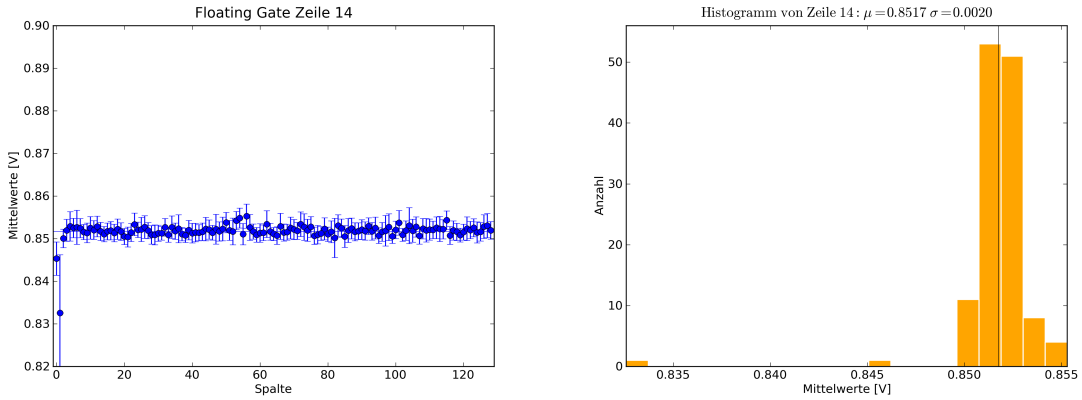


Abbildung 11: Mittelwertplot und Histogramm von Zeile 14

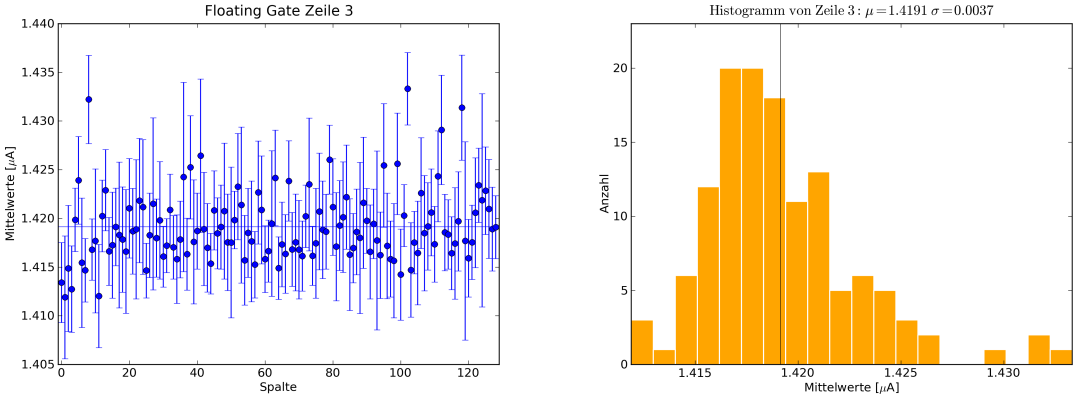


Abbildung 12: Mittelwertplot und Histogramm von Zeile 3

Auch hier ist gut zu erkennen, dass die Stromzeile grössere Schwankungen aufweisen. Auch die im Gesamtplot gesehenen Ausreisser sind auszumachen.

5.2.2 Einmaliges Programmieren

Zum Vergleich der Gesamtblock mit der Messung in dem nur einmal die Werte programmiert wurden. Leider habe ich diese Messung mit nicht so viel Durchläufen aufnehmen können und somit kann die Standardabweichung nicht zum direkten Vergleich herangezogen werden. Jedoch habe ich im Laufe voriger Messungen die Standardabweichungen bei gleicher Durchlauflänge vergleichen können und habe einen Faktor 5 ausgemacht um den der Fehler bei mehrfachen Programmieren grösser ist. Trotzdem ist in Abb. 9 der Farbplot abgebildet.

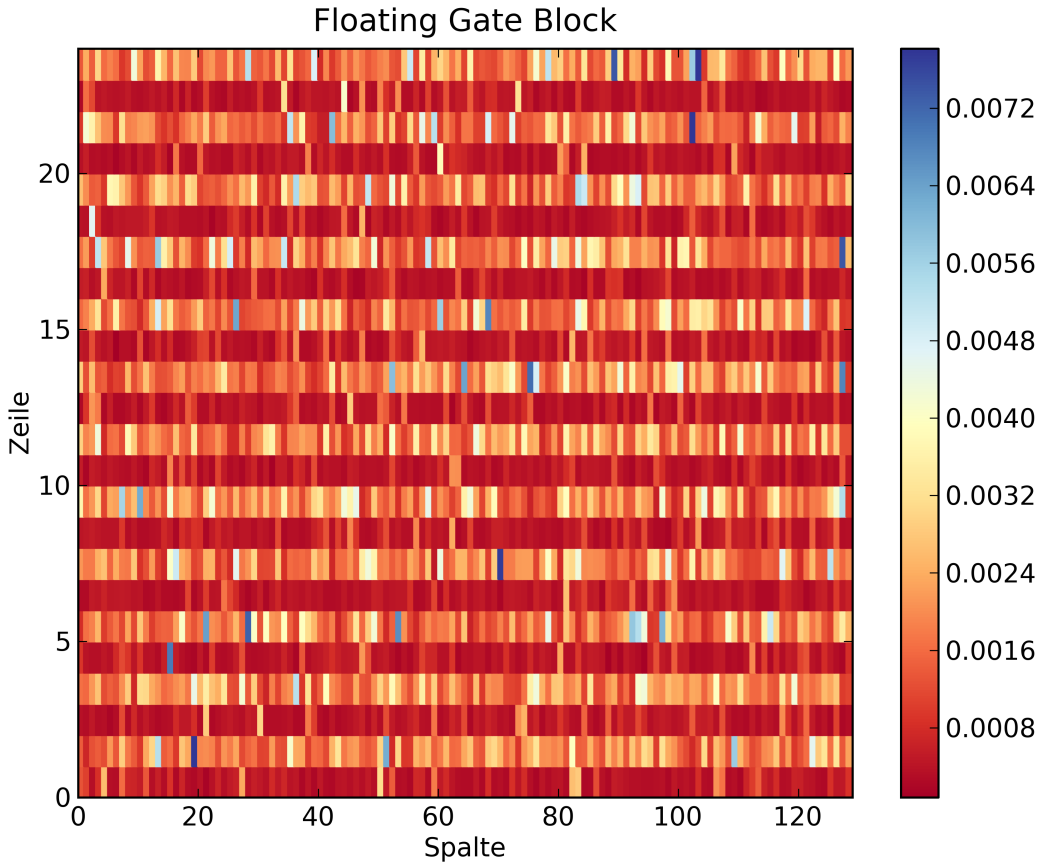


Abbildung 13: Farbplot des Floating Gate Blocks bei einmaligem Programmieren (Farblegende in Volt)

6 Fazit

Abschliessend kann man sagen, dass die Parameteroptimierung die Programmierschnelligkeit um das etwa 4-fache heruntersetze, ohne dabei die Standardabweichung zu verändern. Wobei ich mich wie schon erwähnt nur auf 2 Parameter beschränkte. Unter Umständen wäre es noch möglich bei genauerer Untersuchung das Programmieren der Floating Gates noch weiter zu optimieren, jedoch war dies im Rahmen des Prakitkum nicht intensiver möglich. Auch der 2. Teil meiner Arbeit lieferte interessante Ergebnisse, die jedoch eher informativer Gestalt waren und nicht unbedingt quantitative und veränderte Resulate hervorbrachte.
Alles in allem habe ich während meines 10 wöchigen Praktikums sehr viel neues gelernt, gerade das Einarbeiten in Python war sehr interessant und hat mich viel Neues lernen lassen. Je länger ich mich mit den Themen auseinander setzte, desto mehr und besser begriff ich das gesamte Projekt und die dahinter steckenden Thematiken und Funktionsweisen. Auch das angenehme Arbeitsklima und Atmosphäre in der Arbeitsgruppe war einer der Gründe, dass mir diese 10 Wochen viel Spaß bereitet haben.

7 Quellen

[1] Website der Gruppe Electronic Visions:
`http://www.kip.uni-heidelberg.de/cms/vision/projects/facets/neuromorphic_hardware/waferscale_integration_system/the_hicann_chip/`

[2] Dissertation von Sebastian Millner: "Development of a Multi-Comparment Neuron Model Emulation"2012

8 Anhang

Hier im Anhang zeige ich nochmal den kompletten Code den ich zum Schreiben und Auslesen der Floating Gate Zellen benutzt habe.

```
import unittest
import time as t
import numpy as np
import matplotlib.pyplot as plt
import random
import plot_voltage
from pyhalbe import PowerBackend, Handle, Coordinate, HICANN, geometry
import pyhalbe
class HICANNBackendHWTests():
        def __init__(self, Ip, port, h, d, f):
                Enum = geometry.Enum
                highspeed = True
                arq = True
                hicann_num = 8
                self.IP = Ip
                self.PORT = port
                self.HICANN = h
                self.DNC = d
                self.FPGA = f
                ip = Coordinate.IPv4.from_string(self.IP)
                self.myPowerBackend = PowerBackend.instanceVerticalSetup()
                self.myPowerBackend.SetupReticle(highspeed, ip, self.PORT, hicann_num, arq)
                self.h = Handle.HICANN(Coordinate.HICANNGlobal(Enum(self.HICANN)))
                self.dnc = Coordinate.DNCGlobal(Enum(self.DNC))
                self.fpga = Coordinate.FPGAGlobal(Enum(self.FPGA))

        def init_adc_calibration_backend(self, adc_id):
                import pycalibtic
                '''Initialize the backend which contains ADC calibration data.
                The data will be used to convert raw ADC values to voltages.'''
                backend = pycalibtic.loadBackend(pycalibtic.loadLibrary("libcalibtic_mongo.so"))
                backend.config("host", "cetares")
                backend.config("collection", "adc")
                backend.init()
                adc_calib = pycalibtic.ADCCalibration()
                adc_calib.load(backend, adc_id)
                return adc_calib

        def test_HICANNInit(self, parameters):
                from pyhalbe import Handle, ADC, HICANN
                import pylab

                HICANN.reset(self.h)
                HICANN.init(self.h)
                HICANN.prime(self.h)

                #input_channel = ADC.INPUT_CHANNEL_7

                #configure ADC
                adc    = Handle.ADC(pyhalbe.Coordinate.ADC(23))
                chan0 = ADC.INPUT_CHANNEL_0
                chan1 = ADC.INPUT_CHANNEL_1
                trig   = ADC.TRIGGER_CHANNEL_0
                conf   = ADC.Config(200, chan0, trig)
                ADC.config(adc, conf)

                #get calibration data of adc
                adc_id = ADC.get_board_id(adc)
                adc_calib = self.init_adc_calibration_backend(adc_id)

                # set ANALOG
                ac = HICANN.Analog()
                ac.set_fg_left(Coordinate.AnalogOnHICANN(0))
                ac.set_fg_left(Coordinate.AnalogOnHICANN(1))
                HICANN.set_analog(self.h, ac)

                fgc = pyhalbe.FGControl()
                fg_conf = HICANN.FGConfig()
```

```
        fg_conf.maxcycle = parameters['maxCycle']
        fg_conf.currentwritetime = parameters['currentWriteTime']
        fg_conf.voltagewritetime = parameters['voltageWriteTime']
        fg_conf.readtime = parameters['readtime']
        fg_conf.acceleratestep = parameters['accelerateStep']
        fg_conf.pulselength = parameters['pulselength']
        fg_conf.fg_biasn = parameters['fg_biasn']
        fg_conf.fg_bias = parameters['fg_bias']
        fgc.setConfig(Coordinate.FGBlockOnHICANN(0),fg_conf)

        fgc.load_default_fg_values()

        HICANN.set_fg_values(self.h, fgc.extractBlock(Coordinate.FGBlockOnHICANN(1)))

        Digital = []
        Analog = []
        zeitwerte = []

        for i in range(1,101):
        #if 1:
                print "Run No. "+str(i)
                for n in range(0,128):
                        nrn = Coordinate.NeuronOnHICANN(geometry.X(n),geometry.Y(0))
                        v = random.randint(0,1023)
                        Digital.append(v)
                        fgc.setNeuron(nrn, HICANN.neuron_parameter.E_syni, v)

                start = t.time()
                HICANN.set_fg_values(self.h, fgc.extractBlock(Coordinate.FGBlockOnHICANN(0)))
                ende = t.time()
                zeitwerte.append(ende-start)

                for n in range(0,128):
                #if 0:
                        nrn = Coordinate.NeuronOnHICANN(geometry.X(n),geometry.Y(0))
                        HICANN.get_fg_cell(self.h, nrn, HICANN.neuron_parameter.E_syni)
                        ADC.trigger_now(adc)
                        t.sleep(0.0002)
                        trace = ADC.get_trace(adc)
                        raw_trace = ADC.get_trace(adc)
                        trace = pylab.array(raw_trace, dtype=pylab.ushort)
                        calibrated_trace = adc_calib.apply(int(chan0), trace)
                        value = np.mean(calibrated_trace)
                        Analog.append(value)

        z = np.mean(zeitwerte)
        print z, 'Sekunden'
        plot_voltage.plot(Digital, Analog, z)
        #print Analog




if __name__ == '__main__':

        import argparse
        import sys

        parser = argparse.ArgumentParser(description='')
        parser.add_argument('--ip',    action='store', required = True,
                        type=str,
                        help='ip of the setup to connect')
        parser.add_argument('--port', action='store', required = True,
                        type=int,
                        help='port of the setup to connect')
        parser.add_argument('--h', action='store', required = False,
                        type=int, default=0,
                        help='hicann to connect')
        parser.add_argument('--d', action='store', required = False,
                        type=int, default=0,
                        help='dnc to connect')
```

```
parser.add_argument('--f', action='store', required = False,
                    type=int, default=0,
                    help='fpga to connect')

args, argv = parser.parse_known_args()
argv.insert(0, sys.argv[0])

x = HICANNBackendHWTests(args.ip, args.port, args.h, args.d, args.f)
x.test_HICANNInit()
```