Universität Hamburg

Fachbereich Informatik

AB „Technische Grundlagen der Informatik“

Studienarbeit

Entwurf und Aufbau eines Systems zur Mustererkennung auf Basis von binär gekoppelten Hopfield-Gardner neuronalen Netzwerken

---

von Sönke Frantz

e-mail: 1frantz@informatik.uni-hamburg.de

Rutschbahn 34
20146 Hamburg
040/41 73 68

Hamburg, im Juli 1995

Ich widme diese Arbeit meinem Großvater Carl Werner.

Danksagung

An dieser Stelle möchte ich mich ganz herzlich für die Unterstützung, die vielen Anregungen und die zeitweise benötigte Aufmunterung während des manchmal recht mühsamen Systemaufbaus sowie der Erprobung bedanken, die ich während der Erstellung dieser Arbeit erhalten habe.

Mein Dank geht hierbei insbesondere an den Betreuer dieser Arbeit, Herrn Norman Hendrich, ohne dessen Mitarbeit und Anregungen der erfolgreiche Abschluß dieser Arbeit unmöglich erschien.
Außerdem danke ich Herrn Lars Larsson, Herrn André Klindworth, Herrn Bernd Schütz, Herrn Manfred Grove, Herrn Prof. Klaus Lagemann sowie allen anderen Mitarbeitern des Arbeitsbereiches TECH.
Ferner danke ich der AG IMA des Fachbereichs Informatik für die Bereitstellung sowie die Hilfe bei der Videokameraaufnahme der fertiggestellten Platine.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

In diesem Kapitel wird zunächst der Einsatz von künstlichen neuronalen Netzen motiviert. Anschließend folgen eine Darstellung der für diese Arbeit relevanten Hopfield-Gardner neuronalen Netzwerke sowie die Vorstellung einer aktuellen Hardwareimplementierung dieser Netzart. Schließlich wird der thematische Rahmen dieser Arbeit vorgestellt, d.h. die Einbettung des zur Verfügung gestellten Chips in ein System sowie die Anbindung dieses Systems an einen PC.

1.1 Motivation

In den letzten Jahren wurden die Voraussetzungen für ein alternatives Berechenbarkeitsmodell auf Basis von KNN geschaffen.

Topologische Charakteristika künstlicher neuronaler Netze sind die massive Parallelität sowie der hohe Konnektionismus der i.a. einfachen, oft hierarchisch angeordneten Berechnungselemente. Anwendung finden KNN u.a. in Bereichen wie Computervision, Roboterkinematik, Mustererkennung oder Datenkompression, in denen konventionelle algorithmische Ansätze versagen bzw. zu aufwendig sind sowie bei der Modellierung biologischer Systeme (Vgl. [Roj93]).

Als wichtige Typen seien hier mehrschichtige Perzeptronen-Netzwerke, Hopfield-Gardner Netzwerke, Boltzmann-Maschinen und ART Netzwerke genannt.

Bei der Implementierung von KNN unterscheidet man zwischen Softwaresimulationen auf konventionellen Von-Neumann-Architekturen, dem Einsatz von Multiprozessorsystemen sowie Lösungen mit Spezialhardware. Während Softwaresimulationen geeignet sind zur Untersuchung von Netzeigenschaften sowie zur Überprüfung von Algorithmen bei geringer Netzgröße, erfordern Praxisanwendungen innovative Ansätze, um den Anforderungen in bezug auf die Rechenzeit wegen der Komplexität der Berechnungen gerecht zu werden. Zugleich sind aber auch kostengünstige wie effiziente Lösungen gefordert. Für eine umfassende Darstellung von Hardwareimplementierungen künstlicher neuronaler Netze sei auf [GP94] verwiesen.

1.2 Hopfield-Gardner Netzwerke

1.2.1 Allgemeines

Der in dieser Arbeit betrachtete Netztypus basiert auf dem 1982 entwickelten Modell des amerikanischen Physikers John Hopfield und geht ferner auf Arbeiten von E. Gardner aus dem Jahre 1987 zurück.

Hopfield-Gardner Netzwerke fallen in die Klasse der sog. *Autoassoziativspeicher*. Hierbei werden die im Netz zu speichernden Muster mit sich selbst assoziiert, d.h. für ein zu erkennendes Muster ξ^μ sollte bei der Eingabe von ξ^μ die Ausgabe ξ^μ berechnet werden. Es werden dabei auch in der Umgebung von ξ^μ gelegene Muster, d.h. verrauschte Eingaben auf ξ^μ abgebildet.

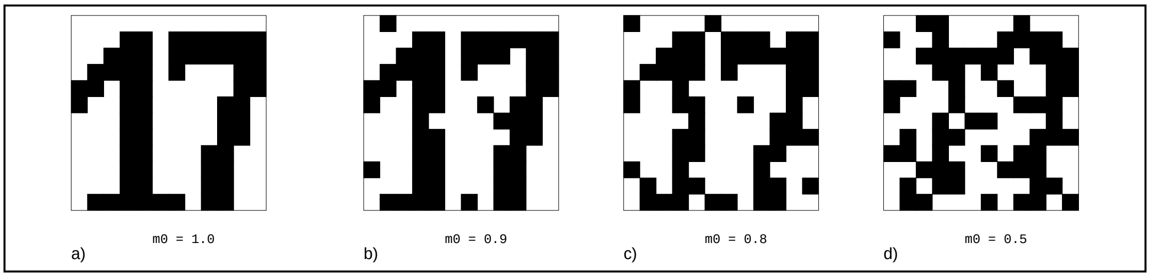

Abbildung 1.1: Zu speicherndes Muster und verrauschte Netzwerkeingaben [Hen95]

In Abbildung 1.1 sind links ein zu speicherndes Muster, rechts davon gestörte Eingaben dieses Musters abgebildet. Die m_0 repräsentieren dabei den sog. *Überlapp* zum Original. Für $m_0 = 0.8$ würde man ein Muster erhalten, das dem Original zu 80% entspricht.

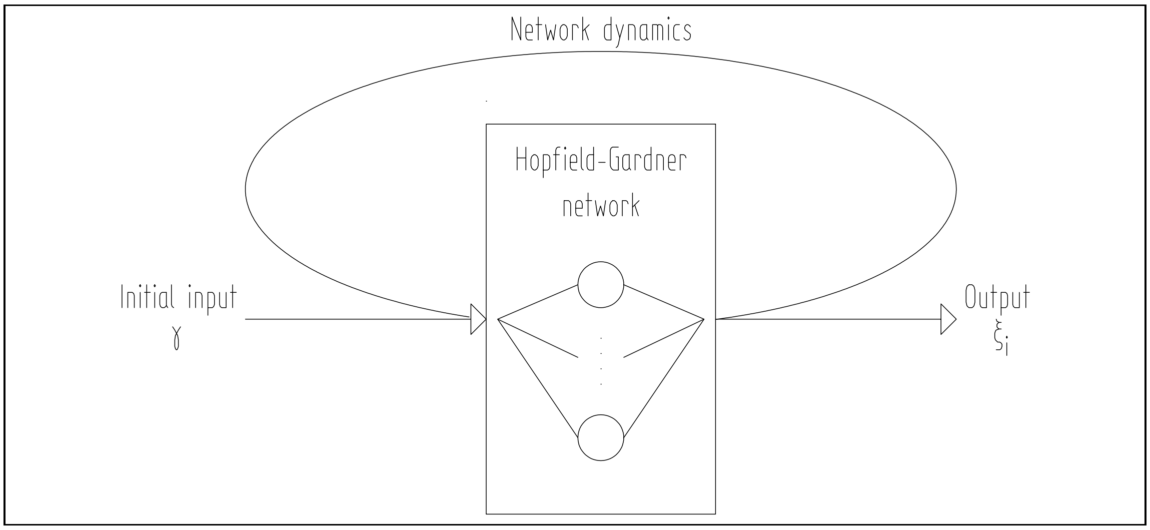

Abbildung 1.2: Netzwerkdynamik

Die Netzeingabe γ eines solchen verrauschten Musters würde bei entsprechend eingestellten

Netzparametern und wiederholter Eingabe der anschließend berechneten Ausgabe zu einer stabilen Ausgabe, dem gespeicherten Muster ξ_i, führen. Abbildung 1.2 illustriert die im folgenden auch als *Dynamik des Netzwerkes* bezeichnete Berechnungsvorschrift.

1.2.2 Formaler Ansatz von Hopfield

Das Hopfield-Gardner Modell besteht aus einer Anzahl N von Berechnungselementen, sog. *Neuronen* mit Zuständen $S_i(t) = \pm 1, 1 \leq i \leq N$, die untereinander total vernetzt sind, d.h. jedes Neuron besitzt eine Verbindung zu jedem anderen Neuron des Netzes. Selbstkopplungen, d.h. Verbindungen eines Neurons zu sich selbst existieren bei diesem Modell nicht. Die Verbindungen sind symmetrisch. Eingaben in ein Neuron über die Verbindungen werden durch die sog. *synaptischen Gewichte* J_{ij} skaliert.
Es wird angenommen, daß die einzelnen Neuronen ihren zuletzt berechneten Zustand beibehalten, bis eine erneute Auswertung ihrer Erregungen erfolgt [Roj93].

Das Netz verarbeitet bipolare Eingaben $\gamma_i = \pm 1,\ 1 \leq i \leq N$ und es berechnet sich für die *parallele Dynamik* der Zustand des Netzes zum Zeitpunkt $t+1$ bei initialem Zustand $S_i(0) = \gamma_i,\ 1 \leq i \leq N$ durch

$$S_i(t+1) = \operatorname{sgn}\left(h_i := \sum_{j=1, j\neq i}^{N} J_{ij} S_j(t) \right), \quad 1 \leq i \leq N$$

wobei J_{ij} die Kopplung des Neurons j zum Neuron i repräsentiert.
Der neue Zustand eines Neurons ergibt sich damit durch das Vorzeichen der Summe aus den mit J_{ij} gewichteten Eingaben in dieses Neuron, dem sog. *lokalen Feld* h_i.

Als *Fixpunkte* der Dynamik eines solchen Netzes werden nun Zustände bezeichnet, für die gilt

$$\forall i = 1, \ldots, N : S_i(t+1) = S_i(t).$$

Hopfield konnte zeigen, daß dieses System als autoassoziativer Speicher für binäre Muster $\xi_i^\mu = \pm 1, 1 \leq i \leq N,\ 1 \leq \mu \leq P$ wirkt.
Die synaptischen Gewichte werden dabei gemäß der Hebb-Lernregel, bei der zwei Neuronen, deren Zustände korreliert sind, eine stärkere Verbindung eingehen als Neuronen, deren Aktivitäten nicht korreliert sind, durch

$$J_{ij} = \frac{1}{N} \sum_{\mu=1}^{P} \xi_i^\mu \xi_j^\mu$$

festgelegt (Vgl. [Hen94],[Roj93]).

Die zu speichernden Muster ξ_i^μ entsprechen dabei offenbar gerade den Fixpunkten des Netzes und heißen in der Literatur auch wegen ihrer Eigenschaft, die in der Umgebung des jeweiligen Musters ξ^μ gelegenen Eingaben anzuziehen, *Attraktoren*.

Nachteilig bei dieser Bestimmung der Kopplungen sind auftretende Störterme, die je nach Korrelation der zu speichernden Muster Auswirkungen auf die Wiedererkennungseigenschaft des Netzes haben. In [Roj93] wird dies als *Crosstalk* bezeichnet. Alternativ zur

Hebb-Lernregel wurden daher Algorithmen entwickelt zur Verbesserung der Speicherkapazität sowie der sog. *Recall*-Eigenschaft, d.h. die Konvergenz des Netzzustandes unter wiederholter Anwendung der Dynamik bei einer einem der gespeicherten Muster ähnlichen Eingabe.

Idee für solche Algorithmen ist eine wiederholte Anwendung der Hebb-Lernregel zur Adjustierung der synaptischen Gewichte, um gewisse Mindeststabilitäten κ für die zu lernenden Muster zu erreichen, d.h. für die lokalen Felder des Musters ξ_i^μ, $1 \leq i \leq N$ wird gefordert, daß

$$\xi_i^\mu h_i^\mu - \kappa \geq 0, \quad 1 \leq i \leq N$$

(Vgl. [Hen95]).

1.2.3 Binär gekoppelte Hopfield-Gardner Netzwerke

In [Hen95],[Hen93] wurden nun die Eigenschaften von binär gekoppelten Hopfield-Gardner Netzwerken, d.h. Netzwerken, für die $J_{ij} = \pm 1$ gilt, untersucht sowie Lern-Algorithmen vorgestellt. Dabei wurden insbesondere Lernregeln erarbeitet, die sich für die angestrebte digitale Hardwareimplementierung besonders anbieten.

Allgemein ergeben sich selbstverständlich aufgrund der Einschränkungen auf binäre Kopplungen gewisse Berechnungsvereinfachungen in bezug auf die Arithmetik, etwa bei der Dynamik oder aber bei den zuvor angesprochenen Lernregeln.

1.3 Hardwareimplementierung binär gekoppelter Hopfield-Gardner Netzwerke

1.3.1 `NN8-Chip`

In [Hen94] wurde die Prototypenimplementierung eines binär-gekoppelten Hopfield-Gardner Netzwerkes mit 8 Neuronen in Form des `NN8-Chips` vorgestellt. Realisiert wurde der Chip in digitaler VLSI-Technologie als Standardzellenentwurf.

Der `NN8-Chip` besteht im wesentlichen aus den 8 Berechnungselementen mit jeweils zwei Energieregistern zur Verwendung für die iterative Lernregel, einer seriellen ALU sowie einem De-/Inkrementierer für die Anwendung der Lernregeln und der Netzwerkdynamik. Daneben sind Flip-Flops für jedes Neuron zur Speicherung der Neuronenzustände enthalten.
Global existiert innerhalb eines im Chip enthaltenen Controllers ein Register zur Speicherung der Sollstabilität. Jener hat ferner die Aufgabe, die zur RAM-Ansteuerung benötigten Steuer- und Adressignale sowie die intern zur Datenpfadschaltung benötigten Signale zu generieren.

Der im `NN8-Chip` enthaltene Befehlssatz umfaßt Befehle für die Anwendung der Dynamik sowie zum teilweisen Auslesen der im Chip enthaltenen Register. Daneben werden die Hebb-Lernregel sowie die in [Hen95] vorgeschlagene iterative Lernregel unterstützt.

An dieser Stelle sollte erwähnt werden, daß der dem `NN8-Chip` zugrunde liegende Entwurf generisch ist, d.h. prinzipiell ist auch ein `NN-Chip` mit mehr Neuronen als den bei dieser Implementierung gewählten 8 Neuronen denkbar. In der nachfolgenden Abbildung ist daher auch allgemein die Architektur eines `NN-Chips` illustriert.

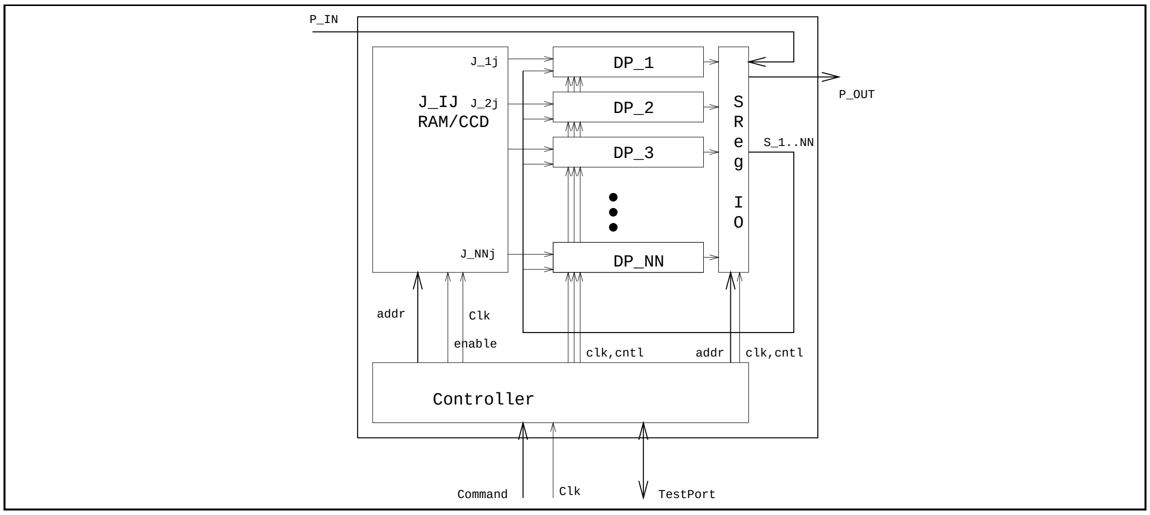

Abbildung 1.3: Hauptblöcke [Hen94]

Vernachlässigt wurde gegenüber dem Hopfield-Gardner Modell bei dieser Implementierung die Restriktion in bezug auf Selbstkopplungen — der Einfluß dieser Kopplungen bei der Berechnung des lokalen Feldes ist bei größeren Netzwerken sehr gering.

1.3.2 Netzwerkrealisierung mit mehreren `NN8-Chips`

Ein Netzwerk läßt sich nun aufbauen durch Parallelschaltung mehrerer `NN8-Chips`, die von einem gemeinsamen Host aus angesteuert werden und jeweils über einen eigenen Speicher für die synaptischen Gewichte verfügen — der Speicher zur Ablage von Mustern ist global.

Die Abbildung eines *virtuellen* Netzwerkes der Größe N auf die tatsächliche Hardwarestruktur der Kapazität $N' = 8 \cdot n$ erfolgt demnach partiell, in dem jeweils nur ein Ausschnitt der Größe $N_p \leq N' \leq N$ betrachtet wird (n entspricht dabei der Anzahl der eingesetzten `NN8-Chips`). Gegenüber sequentiellen Algorithmen zur Berechnung der Dynamik bzw. zur Anpassung der synaptischen Gewichte ergibt sich dabei maximal ein *Speedup* von N'.
Dabei weicht die Berechnungsfolge zur Bestimmung der Neuronenzustände bei der Netzwerkdynamik gegenüber dem Modell von Hopfield dahingehend ab, das die Neuronenzustände des betrachteten Ausschnitts *parallel*, die daraus resultierende Folge von Netzwerkausschnitten jedoch *sequentiell* berechnet wird.

Im einfachsten Fall umfaßt das physikalische Netzwerk genau die innerhalb eines `NN8-Chips` enthaltenen Neuronen, wobei sich dann eine sequentielle Netzwerkdynamik als Berechnungsfolge von $N/8$ Schritten darstellt. Selbst diese Minimalkonfiguration erscheint bei einem zu erwartenden Speedup von 8 gegenüber sequentiell implementierten Algorithmen sinnvoll.

In Abbildung 1.4 ist dagegen die Architektur eines Hopfield-Gardner Netzwerkes, bestehend aus einer Anzahl von `NN-Chips` skizziert.

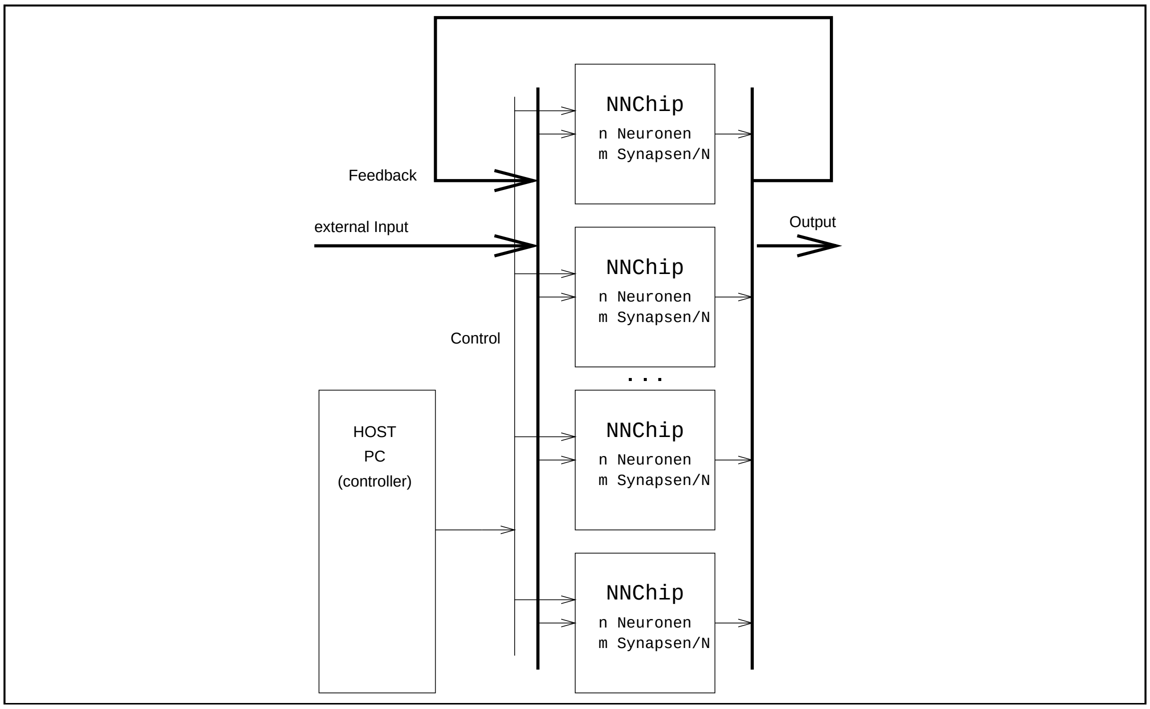

Abbildung 1.4: Architektur des Netzwerks [Hen94]

1.4 Thema der Arbeit

Grundlage für diese Arbeit ist der im vorigen Abschnitt vorgestellte, am Fachbereich Informatik der Universität Hamburg entworfene und über das EUROCHIP-Projekt gefertigte `NN8-Chip` zur Realisierung von binär gekoppelten Hopfield-Gardner Netzwerken [Hen94].

Notwendig zur Nutzung eines solchen, im vorigen Abschnitt skizzierten, Netzwerkes war die Schaffung eines Systems, in das der/die `NN8-Chips` zu integrieren war(en). Dieses System sollte separaten Speicher zur Ablage von Mustern sowie synaptischen Gewichten beinhalten. Es war an einen PC anzubinden.
Bestandteil dieser Arbeit war zum einen die geeignete hardwaremäßige Anbindung des Netzwerkes an den PC. Es wurde insbesondere darauf Wert gelegt, eine grundlegende Testmöglichkeit des gefertigten Prototypen zu schaffen. Zudem sollte, falls möglich, der volle Befehlsumfang des `NN8-Chips` unterstützt werden. Schließlich war die Entwicklung einer Softwarebibliothek zur Nutzung der Netzwerkfunktionen sowie darauf aufbauend die Implementierung von Applikationen zum Test und Einsatz des Systems notwendig. Nicht zuletzt war natürlich eine hohe Systemperformanz erwünscht.

1.4.1 Realisierung

Das System wurde auf einer peripheren PC-ISA Bus Karte, dem „Prototype Adapter" von IBM, aufgebaut.

Die Anbindung des Systems an den PC-Daten und -adressbus erfolgte der Einfachheit halber über programmierbare Portcontroller. Zur Ansteuerung des `NN8-Chips` sowie des benötigten Speichers zur Ablage der Netzparameter wurden zwei programmierbare Logikbausteine auf Basis von FPGAs, im folgenden auch als `NN8-Controller` und `Slave` bezeichnet, eingesetzt. Die Verwendung dieser ICs wurde vom Autoren aufgrund überaus positiver Erfahrungen mit dem eingesetzten Entwurfssystem MAX+PlusII der Firma ALTERA innerhalb anderer durchgeführter Projekte am AB TECH befürwortet. Notwendig war eine derartige Steuereinheit innerhalb des Systems zur Minimierung der PC-Netzwerk Kommunikation und damit der Sicherung hoher Systemperformanz.

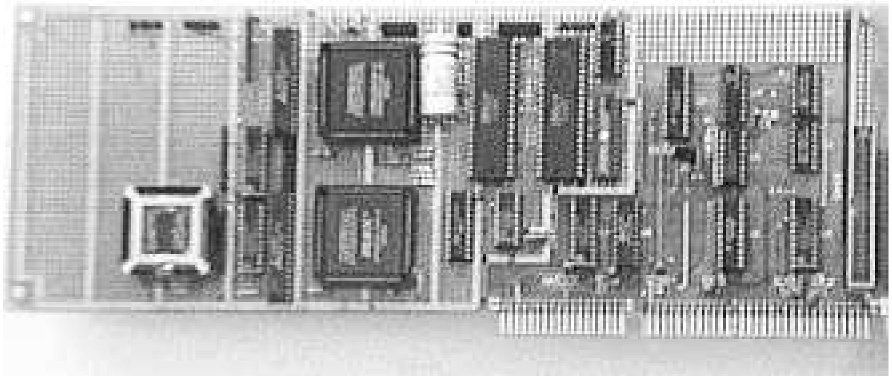

Abbildung 1.5: IBM Prototype-Adapter (Komponentenseite)

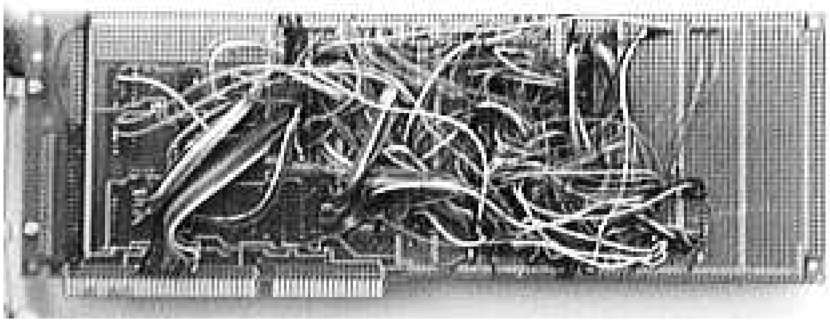

Abbildung 1.6: IBM Prototype-Adapter (Pinseite)

Die obigen Aufnahmen zeigen den bestückten IBM Prototype Adapter. Auf der rechten Seite befindet sich die Standardbestückung mit ICs. Dort werden die über den PC-ISA Bus laufenden Adress- und Steuersignale ausgewertet. Im linken Teil befinden sich die beiden

FPGAs (oben ist der `slave` zu sehen, direkt darunter liegt der `NN8-Controller`). Links von diesen beiden ICs befinden sich der Muster- und Synapsenspeicher in Form der beiden länglichen ICs sowie links außen in einem Spezialträger der `NN8-Chip`.

Als Besonderheit hierbei ist die Flexibilität des Systems aufgrund der Verwendung eines SRAM-basierten FPGAs für den `NN8-Controller` zu nennen, das entsprechend der jeweiligen Einsatzphase des Netzes, also Lernen bzw. Dynamik, konfiguriert werden kann. Den Designs wurden dabei Identifikationen zugewiesen, die von der jeweiligen Applikation abgefragt werden können. Somit kann ggf. automatisch eine Rekonfigurierung des Chips veranlaßt werden. Damit war es möglich, einen kostengünstigen Systemaufbau zu realisieren und gleichzeitig den gegebenen räumlichen Einschränkungen zu entsprechen. Nachteilig ist bei den eingesetzten Logikbausteinen, die direkt am AB TECH zur Verfügung standen, jedoch die Logik-Ressourcenbeschränkung. Diese führte u.a. zu Restriktionen in bezug auf die Netzgröße.

Es ist zunächst nur der Einsatz eines `NN8-Chips` vorgesehen — von einer Umgebung mit mehreren `NN8-Chips` und den damit verbundenen Speichern zur Aufnahme der synaptischen Kopplungen mußte aus den genannten räumlichen Einschränkungen sowie der für diese Anwendung nicht hinreichenden Chipressourcen innerhalb des Logikbausteins für den `NN8-Controller` abgesehen werden.

1.4.2 Bewertung der Systems

Der Test und Einsatz eines entworfenen und gefertigten Chips war Bestandteil dieser Arbeit und fordert daher eine Auseinandersetzung mit den Eigenschaften des Chips sowie eine Leistungsmessung des geschaffenen Systems im Vergleich mit der Implementierung auf konventionellen Rechnerarchitekturen. Dies soll nun Gegenstand dieses abschließenden Abschnittes sein, um dann die geleisteten Arbeiten in den beiden darauffolgenden Kapiteln vorzustellen.

Im Vergleich zu einer reinen Software-Implementierung auf SUN Workstations ergaben sich recht deutliche Geschwindigkeitsvorteile der Hardwareimplementierung, die in der folgenden Tabelle dokumentiert sind. Es wurden ein Netz der Größe von 768 Neuronen zugrundegelegt und dort 100 Schritte der parallel/sequentiellen Dynamik angwandt, wobei bei der Hardwareimplementierung durchschnittlich 0.026s benötigt wurden. Gemäß Chiptiming werden 222816 Taktzyklen für einen Schritt der Dynamik benötigt, bei 16.5 MHz entspricht dies etwa 0.0135s. Gemessen wurden bei einer Genauigkeit von $\frac{1}{100}$s jedoch 0.026s, wobei allerdings der Overhead für Funktionsaufrufe zur Zeitmessung sowie IO-Abfragen inbegriffen ist.

Maschine	Taktfrequenz	ØRechenzeit	Speedup	Speedup(ideal)
SUN SPARC2/40	40 MHz	0.393s	15.12	29.11
SUN microSPARC/70	70 MHz	0.167s	6.42	12.37
SUN SuperSPARC 10/51	50 MHz	0.148s	5.69	10.96
Intel 486/DX2-66	66 MHz	0.254s	9.77	18.81
`NN8-Chip`	16.5 MHz	0.026s (0.0135s)	1.0	1.0

Die benötigte Rechenzeit für Netze wächst dabei für die Dynamik quadratisch mit der Netzwerkgröße, d.h.

$$T(N) = \frac{3}{8}N^2 + \frac{17}{8}N.$$

Im Vergleich zu einer reinen Softwareimplementierung ist jedoch mit einer Vergößerung des Speedups bei steigender Netzwerkgröße aufgrund erhöhter Cache-Misses und damit verbundenen „langsamen“ Speicherzugriffen bei den eingesetzten Workstations zu rechnen.

Bei der iterativen Lernregel konnte bei der gegenwärtigen Systemkonstellation keine Performanzsteigerung gegenüber der Softwaresimulation festgestellt werden. Dies hängt zum einen mit der hohen Busbelastung zusammen - sämtliche `NN8-Chip` Befehle müssen beim Lernen via I/O-Befehl über den PC ISA-Bus an den `NN8-Controller` weitergereicht werden, während beispielsweise bei der Dynamik Sequenzen von Befehlen im `NN8-Controller` generiert werden. Andererseits wirkt sich hier der z.T. relativ hohe Bedarf an Taktzyklen für die Ausführung einzelner `NN8-Chip` Befehle aus, in den sequentiell Register beschrieben bzw. ausgelesen werden. Diese Ausführungszeiten haben einen Anteil an der gesamten Rechenzeit, der kubisch mit der Netzwerkgröße wächst. Der Einsatz mehrerer `NN8-Chips` läßt jedoch eine Kompensierung der eben angesprochenen Nachteile erwarten.

Kritisiert werden muß bei der Prototypenimplementierung des `NN8-Chips` die fehlende Möglichkeit, interne Register wie z.B. die Register zur Speicherung der Energien E_+ und E_-, auszulesen. Diese wäre vorteilhaft zur Bestimmung eines Abbruchkriteriums bei der Anwendung der iterativen Lernregel.
Auch wäre die Ausblendung des `Jij`-Datenbuses wünschenswert, um zusätzliche Bustreiber zu vermeiden.

Kapitel 2

Hardwarebeschreibung

2.1 Einleitung

Dieses Kapitel gibt einen Überblick über die geleisteten Hardwarearbeiten. Der Entwurfsprozeß zur Schaltungsbeschreibung wird skizziert. Danach folgen Beschreibungen der entstandenen Designs sowie detaillierte Befehlsübersichten mit dazugehörigem Timing. Abgerundet wird jeder Abschnitt zur Designbeschreibung durch Angabe hardwaremäßig implementierter Algorithmen (parallele/sequentielle Dynamik) bzw. Algorithmen, die via IO-Befehl abzuarbeitende Sequenzen von Befehlen beinhalten (Hebb-Lernregel sowie iterative Lernregel).

Als Systemkomponenten wurden zum einen Standard-ICs wie z.B. Bustreiber, Chips mit elementaren Logikgattern oder aber auch programmierbare Portcontroller zur Anbindung des `NN8-Controllers` an den PC-Datenbus verwendet. Zum anderen wurden integrierte Schaltungen auf Basis von sog. CPLDs (Complex Programmable Logic Devices) der Firma ALTERA wie etwa der `NN8-Controller`, der konzeptionell zum Controller gehörende `Slave` sowie ein Chip zur Auswertung der vom PC bereitgestellten Steuer- und Adressignale entworfen. Das SRAM-basierte CPLD wird In-Circuit konfiguriert, die anderen beiden CPLDs wurden durch geeignetes Gerät unmittelbar nach Fertigstellung der Schaltungsbeschreibungen programmiert.

2.2 Entwurfsvorgehen

2.2.1 Allgemeines

Die Schaltungen für die einzusetzenden CPLDs wurden dabei mittels der Hardwarebeschreibungssprache AHDL (ALTERA Hardware Description Language) spezifiziert. Die Logiksynthese und anschließende Partitionierung, d.h. die Übersetzung der erarbeiteten Verhaltens- und Strukturbeschreibungen in optimierte Strukturen erfolgt automatisch. Ebenso wird die nachfolgende Abbildung auf die jeweilige Zellstruktur der eingesetzten Bausteinfamilie sowie das sog. Fitting, in dem der gewonnene Zellverbund in eine räumliche Anordnung innerhalb des gewählten Bausteines gebracht wird, durch das MAX+plusII-Entwurfssystem von ALTERA vorgenommen.

Die anschließende unerlässliche Überprüfung der Funktionalität sowie des zeitlichen Verhaltens der entworfenen Schaltungen erfolgte mit dem VHDL Simulator der Firma Synopsys. Hierzu wurde außerdem ein am AB TECH entwickelter Compiler zur Übersetzung der auf Basis von AHDL erstellten Designs sowie der vom MAX+plusII-Compiler synthetisierten Schaltungsbeschreibungen eingesetzt, um die für die Simulationsumgebung benötigten VHDL-Modelle zu erzeugen [Kli94].

Insgesamt müssen die mit der Entwurfsumgebung gemachten Erfahrungen als sehr positiv bewertet werden. Die Möglichkeit des hierarchischen Entwurfs durch den Einsatz der Hardwarebeschreibungssprache führte zu relativ übersichtlichen Schaltungsbeschreibungen. Insbesondere die hierbei zur Verfügung gestellten Konstrukte zur Steuerwerksbeschreibung erleichterten die Entwurfsarbeit.

2.2.2 Partitionierung des Systems

Zunächst wurde eine Bedarfsanalyse durchgeführt, d.h. es wurde die Funktionalität des Systems spezifiziert. Hierbei wurde insbesondere Wert auf eine allgemeine Testmöglichkeit des `NN8-Chips` gelegt, zum anderen sollten aber auch die Arbeitsphasen Lernen und Dynamik hinreichend gut unterstützt werden, um den Datenverkehr Host-`NN8-Controller` auf ein Minimum zu reduzieren und damit eine verhältnismäßig hohe Performanz zu erreichen.

Die Ressourcen der hierbei in Frage kommenden Logikbausteine genügten dabei keineswegs der Komplexität des Systems, so daß mehrere Chips eingesetzt werden mußten und dabei aufgrund der räumlichen Restriktionen zusätzlich zur Laufzeit mit verschiedenen Controller-Versionen gearbeitet wird. Im Rahmen dieser Arbeit wurden dazu drei Designs erstellt: Das Design `basic` unterstützt fast alle elementaren `NN8-Chip` Befehle sowie die Hebb-Lernregel; die Designs `iterate` und `dynamic` erlauben eine effiziente Unterstützung der iterativen Lernregel sowie der Netzwerkdynamik. Wie bereits eingangs erwähnt, besteht der `NN8-Controller` aus zwei CPLD's, wobei dem `Slave`-Chip die Aufgabe der Generierung der benötigten Steuer- und Adressignale zur Ansteuerung des in das System integrierten Speichers zukommt. Da diese Signale grundsätzlich benötigt werden und die Befehle zur Speicheransteuerung unabhängig von der jeweiligen Arbeitsphase des Netzwerkes in vollem Umfang zur Anwendung kommen, wurde ein EEPLD (Electric Erasable Programmable Logic Device) der MAX 7000-Familie von ALTERA verwendet, das nach Festlegung des Befehlssatzes zur Speicheransteuerung und anschließendem Schaltungsentwurf programmiert wurde. Die Kommunikation zwischen Hauptbaustein, einem CPLD der FLEX 8000-Familie von ALTERA und dem `Slave` erfolgt unidirektional, die Steuerwerke zur Ablaufsteuerung sind in beiden Chips nahezu identisch.

Für den Entwurf des Hauptteils des `NN8-Controllers` ergaben sich erhebliche Probleme aufgrund der vorgenommenen Pinzuweisungen: Die Anpassung der synthetisierten Zellstruktur an die physikalische Struktur der ICs war bei einigen Entwürfen mit dem eingesetzten Werkzeug nicht möglich.

Daneben mußte ständig die Komplexität eingebundener sog. Subdesigns beobachtet wer-

den. Dies führte zu gewissen Einschränkungen beim Entwurf, da bei zu großen Entitäten innerhalb des Designs die anschließende Synthese des MAX+plusII-Compilers Schaltungsbeschreibungen lieferte, die aufgrund zu großer Verzögerungszeiten den Laufzeitanforderungen während des Betriebes nicht genügten.
Ferner mußte auf eine Hardwarerealisierung des Algorithmus für die Anwendung der Hebb-Lernregel verzichtet werden; der Entwurf zur Realisierung der sequentiellen Netzwerkdynamik implizierte Einschränkungen im Hinblick auf die Netzwerkgröße. Außerdem beschränkt sich das Design `iterate` aufgrund der Komplexität der iterativen Lernregel lediglich auf eine Unterstützung der dafür benötigten Untermenge des `NN8`-Befehlssatzes.

Berücksichtigt man jedoch, daß größere Einschränkungen bzw. ein geringer Grad der Hardwarerealisierung der benötigten Algorithmen lediglich die Lernphase des Netzwerkes betreffen, so ist der gefundene Kompromiß auch im Hinblick auf den Platzbedarf sowie Materialkosten durchaus akzeptabel.

Schließlich wurden zur Anbindung des `NN8-Controllers` an den Host Portcontroller in Form von Standard-ICs verwendet; zur Ansteuerung jener Bausteine war jedoch eine differenziertere Auswertung der übergebenen Steuer- und Adressignale vom PC notwendig, die ebenfalls durch ein CPLD vergleichsweise geringer Komplexität, aber mit für diese Aufgabe relativ großen Pinressourcen, aus der CLASSIC-Familie von ALTERA realisiert wurde — der Einsatz von Standard-ICs war aus Platzgründen nicht möglich.

2.2.3 Zusammenfassung der Einschränkungen

Abschließend seien hier die wichtigsten Systembeschränkungen, die sich zum einen aus den Schaltungsbeschreibungen selber, aber auch aus den synthetisierten Strukturbeschreibungen für die verwendeten Logikbausteine, ergeben.

1. Die Netzwerkgröße ist auf 1024 Neuronen beschränkt.

2. Grundsätzlich erwies sich die Zellstruktur der Logikbausteinfamilie FLEX 8000 in bezug auf diese Anwendung, die relativ komplexe Steuerwerke und Registertypen erforderlich machte, als kritisch im Timing — kleine Lookup-Tabellen zur Logikgenerierung führen zu großem Ressourcenbedarf und damit zu relativ langen Signalpfaden. Die maximale Arbeitsfrequenz des Systems liegt gemäß Timinganalysen des MAX+plusII-Compilers bei ca. 18 MHz. Eingesetzt wird das System derzeit bei einer Frequenz von 16.5 MHz in einem PC mit Intel 80486 Prozessor unter dem Betriebssystem MS-DOS 6.2.

3. Das System unterstützt bisher nur einen `NN8-Chip`, die Integration von mehreren `NN8-Chips` ist aus räumlichen Gründen nicht möglich. Zudem schließt die derzeitige Auslastung der eingesetzten Logikbausteine von z.T. über 95% die in diesem Fall notwendige Überarbeitung der Schaltungsbeschreibungen aus.

Nachfolgend ist die Systemarchitektur, bestehend aus den Komponenten Host, NN8-Controller und Slave, NN8-Chip sowie Speicher illustriert.

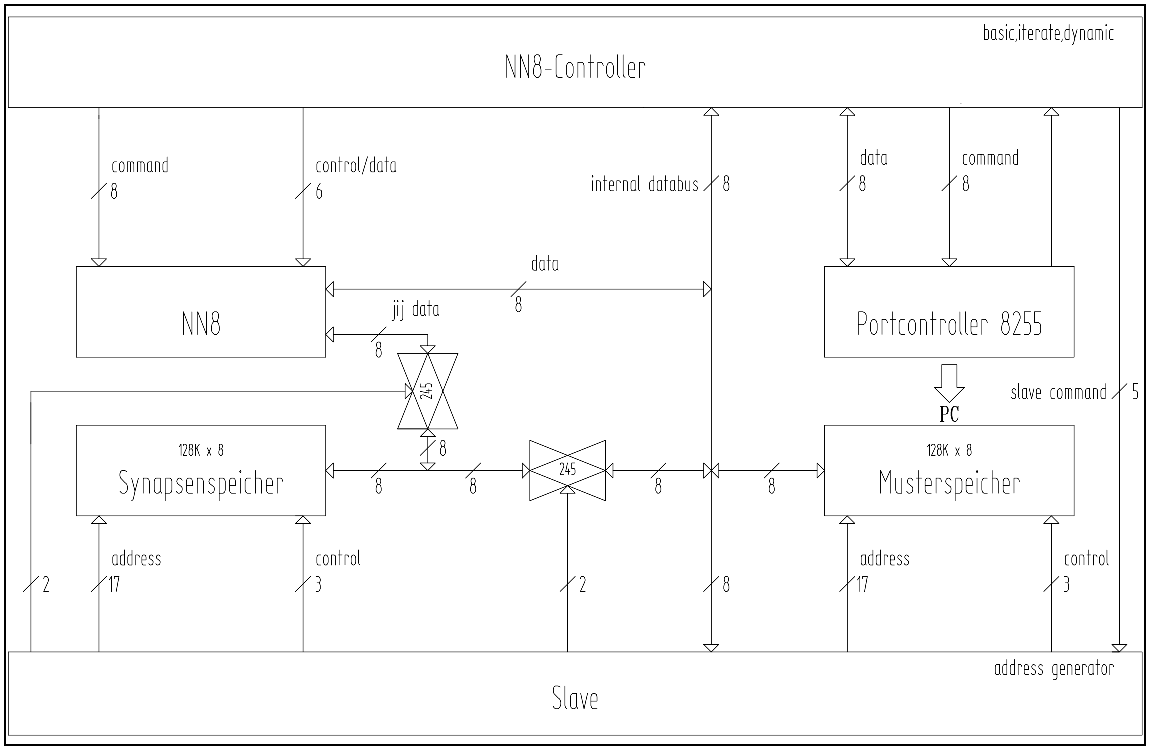

Abbildung 2.1: Daten- und Steuersignalpfade Portcontroller 8255 — NN8-Controller — NN8-Chip — RAM

2.3 Beschreibung des NN8-Controllers

2.3.1 Ansteuerung

Während eines I/O-Befehls vom PC werden neben der angelegten Adresse die IORead- bzw. IOWrite-Signale PC-seitig aktiviert. Die für die Ansteuerung der Portcontroller 8255 und damit des NN8-Controllers verwendeten Adressignale (relativ zur Adresse des Prototype Adapters) zur Generierung des jeweiligen ChipEnable-Signals sowie des gemeinsamen Reset-Signals sind a3 und a4, die Selektion des jeweiligen Ports wird durch die Adressignale a2 und a1 vorgenommen. Die Datenein- und Ausgänge der Portcontroller sind an den PC-Datenbus angeschlossen, wobei die Bits 7 bis 0 an den Portcontroller 8255 I und die Bits 15 bis 8 an den Portcontroller 8255 II gehen.

Das Decodierungsschema, implementiert im Baustein `Adr_Decode` für die Auswahl der Portcontroller 8255 ist nun gegeben durch

a4	a3	nChipEnable I	nChipEnable II	Reset
0	0	1	1	1
0	1	0	1	0
1	0	1	0	0
1	1	0	0	0

Die Auswahl des jeweiligen Ports erfolgt gemäß Hersteller-Spezifikation durch

a2	a1	Modus
0	0	Port A
0	1	Port B
1	0	Port C
1	1	Program

Der `NN8-Controller` Kommando-Port (bezeichnet mit `porta_low`) ist an Port A des Portcontrollers 8255 I angeschlossen, der externe Datenbus (bezeichnet mit `bidir_porta_high`) ist mit Port A des Portcontrollers 8255 II verbunden. Zusätzlich wurden noch für Rückmeldungen die `busy`-Leitung des `NN8-Controllers` an Port B, Bit 0 und die Ready-Leitung des `NN8-Chips` an Port B, Bit 1 angeschlossen.

2.3.2 Befehlsformat und -abarbeitung

Die Befehlsbreite des `NN8-Controllers` umfaßt 8 bzw. 16 Bit, wobei die unteren 8 Bit eines Datenwortes als Operationscode interpretiert werden. Zusätzlich werden bei einigen Befehlen die Bits 15 bis 8 ausgewertet, etwa bei Speicherzugriffen, zur Indizierung im `NN8-Controller` gespeicherter Datenwörter oder zur Befehlsweitergabe an den `NN8-Chip`.

Die Operationscodebreite beträgt 7 Bit, das 8. Bit dient zur Kennzeichnung eines neuen Controller-Befehls. Es muß also darauf geachtet werden, mit jedem neu angelegten Befehl das 8. Bit zu toggeln. Diese Festlegung wurde aufgrund der Tatsache, daß Schreibzyklen des PC's länger als einen Takt dauern, Controller-Befehle jedoch für die Ausführung z.T. nur einen Takt benötigen, vorgenommen. Man hätte diese Maßnahme durch erhöhten Hardwareaufwand vermeiden können, dies war jedoch aus Platzgünden nicht möglich. Die in Kapitel 3 vorgestellte Softwarebibliothek trägt dem eben beschriebenen Umstand jedoch Rechnung.

Zu beachten ist außerdem, daß einmal an den `NN8-Controller` gesandte Befehle vollständig abgearbeitet werden — zwischenzeitlich angelegte Befehle werden ignoriert. Bei den hier vorgestellten Befehlen ist jedoch die Bearbeitungsdauer eines Befehls grundsätzlich kleiner als die Dauer eines I/O-Befehls, so daß i.a. jeder abgesandte Befehl abgearbeitet wird. Im Zweifel sollte, wie auch in den folgenden Abschnitten empfohlen, das Statussignal `busy` des `NN8-Controllers` abgefragt werden. Dieses wird z.B. in der eingesetzten Umgebung notwendig im Zusammenhang mit der Anwendung der iterativen Lernregel und der Netzwerkdynamik.

2.3.3 Verwendetes Speichermodell

Dieser Abschnitt beschreibt die Festlegung der Byte-Interpretation sowie die Abbildung der Neuronenzustände und der synaptischen Gewichte auf den in das System integrierten Speicher.

In Abstimmung mit dem gewählten Dateiformat zur Speicherung der Netzparameter werden das Bit 7 des Datenregisters `data_reg` des `NN8-Controllers` als LSB und Bit 0 als MSB interpretiert.

Die Neuronenzustände sind entsprechend der Wortbreite der verwendeten Speicherbausteine blockweise zu jeweils 8 Bit abzulegen. Die synaptischen Gewichte werden in ähnlicher Weise gespeichert, wobei die Gewichte eines Neuronenblockes zu einem anderen Neuron in einem Wort zusammengefaßt sind und grundsätzlich die Gewichte zwischen einem Block und dem Netz zusammenhängend im Speicher liegen müssen.
Sowohl für Muster als auch Gewichte gilt analog, daß Bit 7 als LSB und Bit 0 als MSB interpretiert werden.

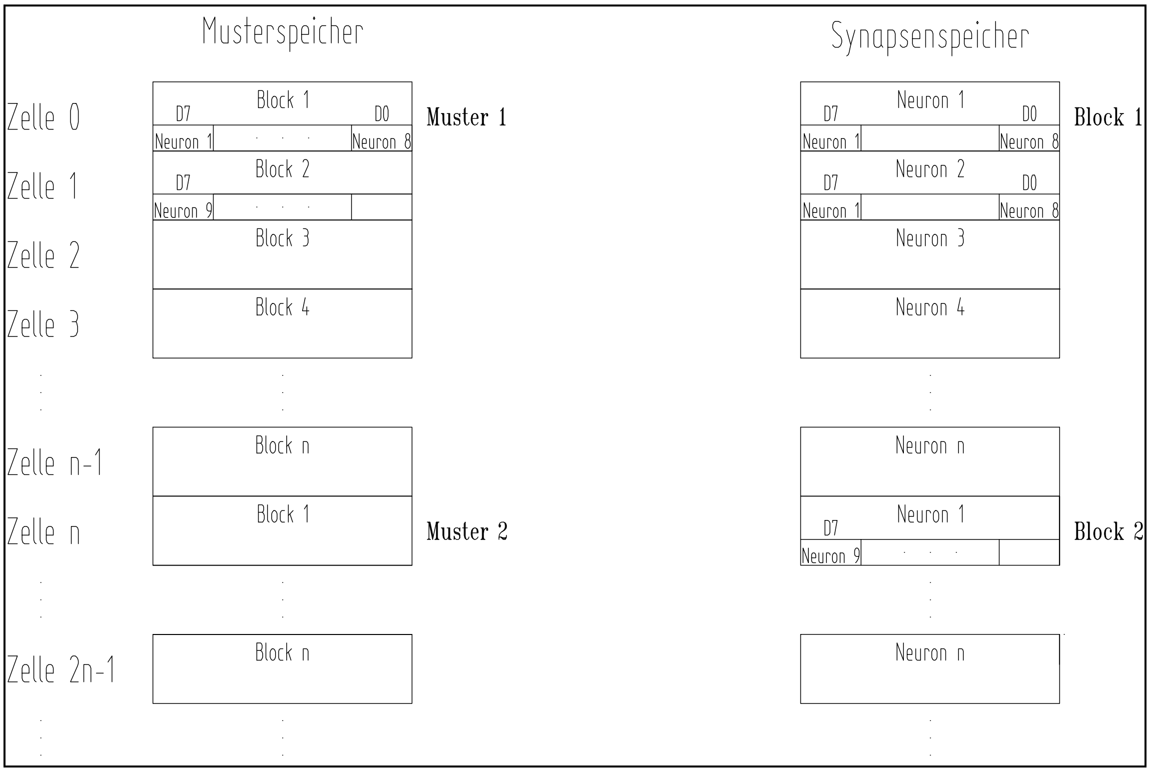

Abbildung 2.2: Muster- und Synapsenspeicherorganisation

2.3.4 Allgemeine `NN8-Controller` Befehle

Allen `NN8-Controller` Designs gemein ist das Steuer- und Operationswerk zur Adressierung des Kartenspeichers. Dazu gehören zwei parallel/sequentiell ladbare 17-Bit Zähler (`pattern_ram_adr_reg` bzw. `synapse_ram_adr_reg`) zur Bereitstellung der Adressignale des Muster- und Synapsenspeichers sowie ein 13-Bit Register, im folgenden mit `basis_reg` bezeichnet, zur Adressszwischenspeicherung, das jedoch z.Zt. lediglich im Design `dynamic` verwendet wird und nicht von außen ansprechbar ist.

Grundsätzlich wurde bei allen Designs die Möglichkeit der hierarchischen Schaltungsbeschreibung genutzt. Dies erlaubte eine relativ einfache Einbindung des in diesem Abschnitt beschriebenen `NN8-Controller` Kerns zur Speicheradressierung bei den nachfolgend beschriebenen Entwürfen.

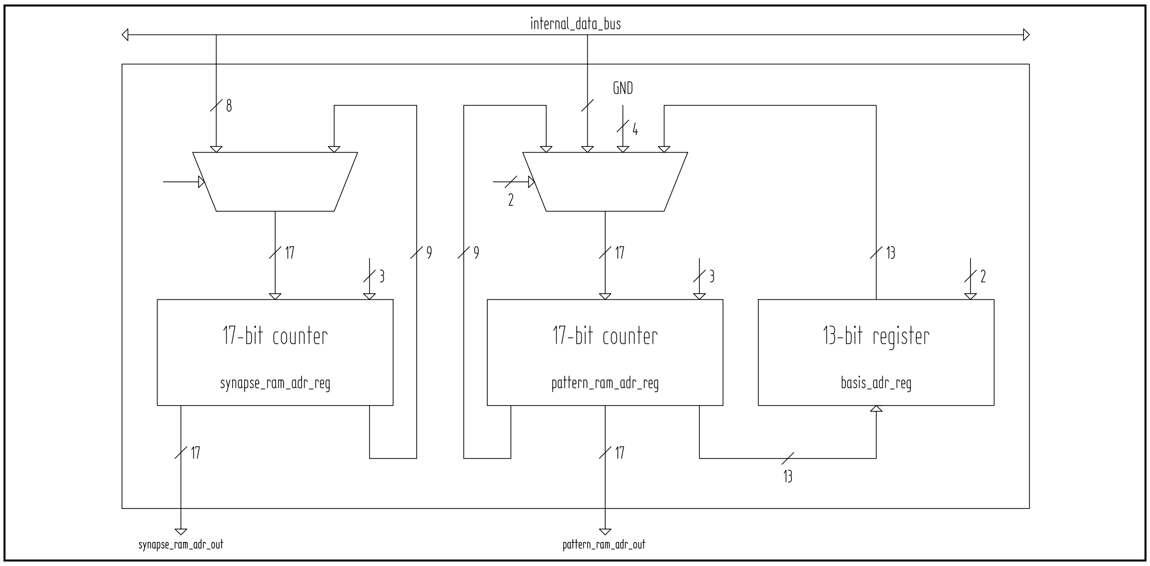

Abbildung 2.3: RT-Beschreibung von `slave`

Außerdem ist ein 8-Bit Schieberegister (im folgenden ist dieses mit `data_reg` bezeichnet) zur Pufferung von Daten im Kommunikationsring PC-`NN8-Controller`-`NN8-Chip`-Speicher verfügbar. Je nach Design besteht die Möglichkeit bi- bzw. unidirektional zu shiften.

Lese- und Schreibzyklen auf den Muster- und Synapsenspeicher dauern grundsätzlich 2 Takte, um Timingprobleme, etwa durch zwischengeschaltete Bustreiber und damit verbundene Verzögerungszeiten zu vermeiden. Zudem dauern Speicherzugriffe seitens des `NN8-Chips` immer zwei Takte und fordern z.B. bei der Dynamik oder dem Lernen einen zum `NN8-Controller` synchronen Betrieb.

Es wurde beim Entwurf Wert auf eine Realisierung der Speicherzugriffe mit minimalem Overhead gelegt, um insbesondere bei langsamen Systemen den Busverkehr über den PC-ISA Bus auf das Nötigste einzuschränken. Aus dieser Motivation heraus wurden Speicherzugriffe mit Postinkrement des Adresszählers sowie ein optimiertes Auslesen des Speichers ermoglicht.

Die im folgenden beschriebenen `NN8-Controller` Befehle sind in allen drei Designs identisch.

- `CONTR_get_controller_id`

  Kodierung ^{6}111 1111^{0}

  Eine design-spezifische Kennung wird in `data_reg` abgespeichert, um die jeweilige Konfiguration identifizieren zu können. Dies ermöglicht vor dem Aufruf einer komplexen Funktion die Überprüfung des `NN8-Controllers` und ggf. die anschließende Neukonfigurierung (1 Takt).

- `CONTR_reset_pattern_adr_reg`

  Kodierung ^{6}000 0001^{0}

  Rücksetzen von `pattern_ram_adr_reg` auf Null (2 Takte).

- `CONTR_write2pattern_adr_reg` *address part*

  Kodierung ^{6}000 0010^{0}

  Laden von `pattern_ram_adr_reg`. Hierzu erfolgt ein Shift des alten Zählerstandes zur Speicherung des übergebenen Adressteils. Zum Laden des Zählers sind demnach drei Aufrufe dieses Befehls notwendig, beginnend mit Bit 16 (im Datenwort Bit 0), anschließend die Bits 15 bis 8 und schließlich die Bits 7 bis 0 (2 Takte).

- `CONTR_increm_pattern_adr_reg`

  Kodierung ^{6}000 0011^{0}

  Inkrementieren von `pattern_ram_adr_reg` (2 Takte).

- `CONTR_write_pattern` *data*

  Kodierung ^{6}000 0100^{0}

  Speichern des übergebenen Datums in der gerade adressierten Zelle des Musterspeichers (3 Takte).

- `CONTR_increm_write_pattern` *data*

  Kodierung ^{6}000 0101^{0}

  Speichern des übergebenen Datums in der gerade adressierten Zelle des Musterspeichers, anschließend wird `pattern_ram_adr_reg` inkrementiert (4 Takte).

- `CONTR_read_pattern`

  Kodierung ^{6}000 0110^{0}

  Auslesen der gerade adressierten Zelle des Musterspeichers, das Muster wird anschließend in `data_reg` gespeichert (3 Takte).

- `CONTR_increm_read_pattern`

  Kodierung ^{6}000 0111^{0}

  Auslesen des Musterspeichers, das Muster wird anschließend in `data_reg` gespeichert. Zusätzlich wird `pattern_ram_adr_reg` inkrementiert (3 Takte).

- `CONTR_init_fast_read_of_pattern_ram`

  Kodierung ‘001 1100’

  Das vollständige Auslesen des Musterspeichers wird vorbereitet. Dazu wird der Speicherinhalt der gerade adressierten Zelle in `data_reg` geladen; anschließend wird `pattern_ram_adr_reg` inkrementiert. Jeder I/O-Lesezyklus impliziert ein erneutes Auslesen und darauf folgendes Inkrementieren von `pattern_ram_adr_reg` (5 Takte).

- `CONTR_exit_fast_read_of_pattern_ram`

  Kodierung ‘001 1101’

  Die vorige Operation, d.h. `CONTR_fast_read_of_pattern_ram`, wird abgebrochen. Die Durchführung dieses Befehls ist unbedingt notwendig, um ein anschließendes korrektes Arbeiten zu gewährleisten (1 Takt).

- `CONTR_reset_synapse_adr_reg`

  Kodierung ‘000 1000’

  Rücksetzen von `synapse_ram_adr_reg` auf Null (2 Takte).

- `CONTR_write2synapse_adr_reg` *address part*

  Kodierung ‘000 1001’

  Laden von `synapse_ram_adr_reg`, analog zu `CONTR_write2pattern_adr_reg` (2 Takte).

- `CONTR_increm_synapse_adr_reg`

  Kodierung ‘000 1010’

  Inkrementieren von `synapse_ram_adr_reg` (2 Takte).

- `CONTR_write_synapse_weight` *data*

  Kodierung ‘000 1011’

  Speichern des vom PC übergebenen Datums in der gerade adressierten Zelle des Synapsenspeichers (4 Takte).

- `CONTR_increm_write_synapse_weight` *data*

  Kodierung ‘000 1100’

  Speichern des übergebenen Datums in der gerade adressierten Zelle des Synapsenspeichers, anschließend wird `synapse_ram_adr_reg` inkrementiert (5 Takte).

- `CONTR_read_synapse_weight`

  Kodierung ‘000 1101’

  Auslesen der gerade adressierten Zelle des Synapsenspeichers, das Gewicht wird anschließend in `data_reg` gespeichert (4 Takte).

- `CONTR_increm_read_synapse_weight`

  Kodierung ^{6}000 1110^{0}

  Auslesen der gerade adressierten Zelle des Synapsenspeichers, das Gewicht wird anschließend in `data_reg` gespeichert. Zusätzlich wird `synapse_ram_adr_reg` inkrementiert (5 Takte).

- `CONTR_init_fast_read_of_synapse_ram`

  Kodierung ^{6}001 0100^{0}

  Das vollständige Auslesen des Synapsenspeichers wird vorbereitet. Dazu wird der Speicherinhalt der gerade adressierten Zelle in `data_reg` geladen; anschließend wird der Adresszähler `synapse_ram_adr_reg` inkrementiert. Jeder I/O-Lesezyklus vom PC impliziert ein erneutes Auslesen und darauf folgendes Inkrementieren von `synapse_ram_adr_reg` (6 Takte).

- `CONTR_exit_fast_read_of_synapse_ram`

  Kodierung ^{6}001 0101^{0}

  Die vorige Operation, d.h. `CONTR_exit_fast_read_of_synapse_ram`, wird abgebrochen. Die Durchführung dieses Befehls ist unbedingt notwendig, um ein anschließendes korrektes Arbeiten zu gewährleisten (1 Takt).

- `CONTR_write2data_reg` *data*

  Kodierung ^{6}100 1100^{0}

  `data_reg` wird mit dem vom PC übergebenen Wert geladen (1 Takt).

Das Auslesen des Datenregisters, in dem sämtliche Werte zwischengespeichert werden, in den PC erfolgt über einen einfachen Lesezugriff auf Port A des angeschlossenen Portcontrollers. Dabei ist zu beachten, das der Datenbus des `NN8-Controller` an die Datenleitungen 15 bis 8 angeschlossen ist und daher das gelesene Wort entsprechend interpretiert wird.

Bei Lesezyklen wird vorausgesetzt, daß der Portcontroller vor dem eigentlichen Lesen entsprechend programmiert wurde – der `NN8-Controller` greift in jedem Fall bei aktivem `IORead`-Signal und adressiertem Port A schreibend auf den Bus zu.

2.3.5 Spezielle `NN8-Controller` Befehle

2.3.5.1 Design `basic`

Das Design `basic` unterstützt neben der Hebb-Lernregel die wichtigsten `NN8`-Befehle. Es ist insbesondere zum Test des `NN8-Chips` geeignet, d.h. um die Funktion der einzelnen Befehle sowie die Timingvorgaben zu überprüfen. Diesem Design wurde die Identifikation ^{7}11110000^{0} zugewiesen.

Ergänzt wurde das Grunddesign aus Abschnitt 2.3.4 neben dem zusätzlichen Steuerwerk um einen 4-Bit Zähler zum Auslesen der Neuronenzustände.

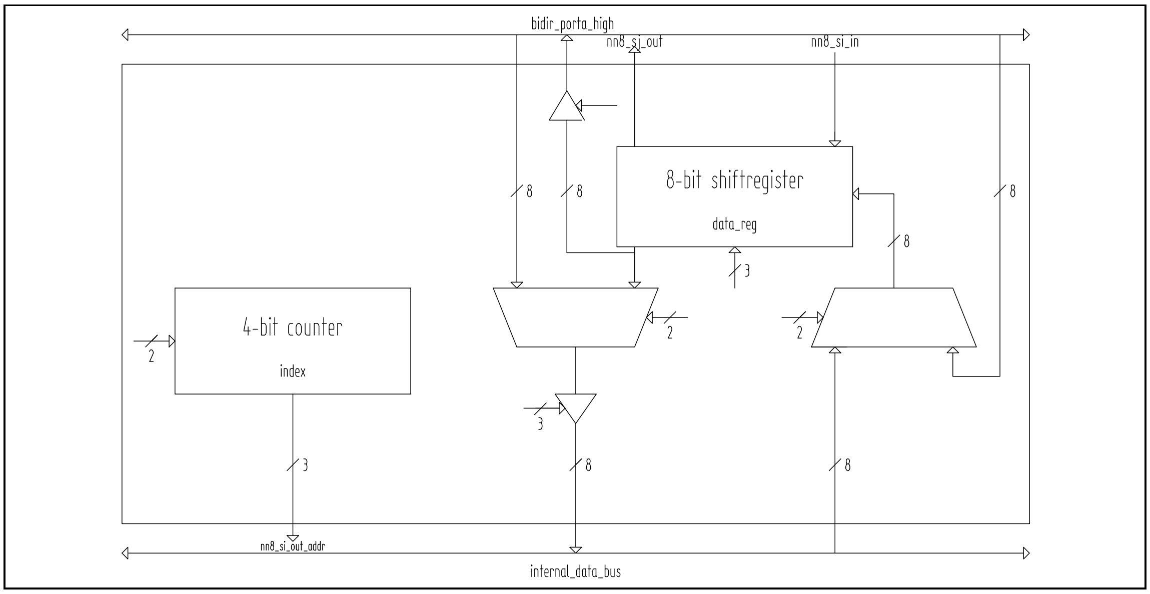

Abbildung 2.4: RT-Beschreibung von `basic`

Neben einer kurzen Beschreibung der zusätzlich zur Verfügung stehenden Befehle ist eine algorithmische Beschreibung für die Hebb-Lernregel angegeben.

- `CONTR_execute_nn8_command` *command*

  Kodierung ^{6}100 1110^{0}

  Das übergebene Datum wird an den Command-Port des `NN8-Chip` für insgesamt einen Takt angelegt - das Datum wird vom `NN8-Controller` nicht ausgewertet und ist daher lediglich zu Testzwecken geeignet (1 Takt).

- `CNN8_load_nj_register_low` *lower part*

  Kodierung ^{6}001 1010^{0}

  Laden der Bits 7..0 des `NN8-Chip` NJ-Registers (2 Takte).

- `CNN8_load_nj_register_high` *higher part*

  Kodierung '001 1011'

  Laden der Bits 15..8 des `NN8-Chip` NJ-Registers (2 Takte).

- `CNN8_set_si`

  Kodierung '100 0011'

  `NN8`-Befehl '`Set Si`' zur Berechnung der Werte aller Neuronen (1 Takt).

- `CNN8_load_pattern`

  Kodierung '001 1001'

  `NN8`-Befehl '`Load Pattern`' zum Laden eines externen Musters. Das Muster wird dem Register `data_reg` des Controllers entnommen, diesem Befehl sollte also ein entsprechender Befehl zum Laden des Registers vorangehen (26 Takte).

- `CNN8_clear_hi`

  Kodierung '100 0100'

  `NN8`-Befehl '`Clear Hi`' zum Rücksetzen des Hi-Registers des `NN8-Chips` (1 Takt).

- `CNN8_step_hebb` *neuron*

  Kodierung '100 1011'

  `NN8`-Befehl '`Step Hebb`', in dem ein Schritt der Hebb-Lernregel ausgeführt wird (vgl. dazu algorithmische Beschreibung *Hebbian*). Dieser Befehl benötigt als Argument *neuron* den Index des jeweiligen Bits, das über einen dem Datenregister `data_reg` vorgeschalteten Multiplexer in Zelle 0 geladen wird und damit am Ausgang `sj_out` angelegt wird (5 Takte).

- `CNN8_get_si`

  Kodierung '100 0111'

  Die Neuronenwerte werden aus dem `NN8-Chip` ausgelesen und im Register `data_reg` abgelegt (9 Takte).

- `CONTR_write_data_reg2synapse_ram`

  Kodierung '100 1001'

  Der Inhalt des Datenregisters `data_reg` wird in die gerade adressierte Zelle des Synapsenspeichers geschrieben, anschließend wird `synapse_ram_adr_reg` inkrementiert (5 Takte).

Der Algorithmus für die Hebb-Lernregel läßt sich nun, aufbauend auf den in den letzten Abschnitten vorgestellten Befehlen, angeben, wobei für eine vollständige Beschreibung auf Kapitel 1 sowie die Literatur [Hen95],[Hen94] sowie [Roj93] verwiesen sei.

Mit n sei die Anzahl der Neuronenblöcke (*stripes*) bezeichnet (jeder Block besteht aus 8 Neuronen), p ist die Anzahl zu trainierender Muster (*pattern sets*). Über die äußeren beiden Schleifen werden nacheinander die Synapsen ausgewählt, die innere Schleife ist zuständig für die Einstellung der synaptischen Gewichte gemäß der Hebb-Lernregel.

```
Algorithmus Hebbian

  CONTR_reset_synapse_adr_reg
  FOR stripe=0 TO n − 1 BEGIN
    FOR synapse=0 TO 8 · n − 1 BEGIN
      CNN8_clear_hi
      FOR set=0 TO p − 1 BEGIN
        /* Provide the specified value at neurons in the choosen pattern stripe */
        CONTR_write2pattern_adr_reg n · set + stripe
        CONTR_read_pattern
        CNN8_load_pattern
        /* Read the value of the considered neuron in the choosen pattern */
        CONTR_write2pattern_adr_reg n · set + synapse/8
        CONTR_read_pattern
        /* Set the new synapse value according to the hebb learning rule */
        CNN8_step_hebb synapse mod 8
      END
      CNN8_set_si
      CNN8_get_si
      CONTR_write_data_reg2synapse_ram
    END
  END
```

2.3.5.2 Design `iterate`

Das Design `iterate` mit der Identifikation '00111100' unterstützt die Teilmenge des `NN8`-Befehlssatzes zur Implementierung der iterativen Lernregel.
Das Grunddesign wurde dazu neben einem weiteren Steuerwerk um ein 8-Bit Register zur Zwischenspeicherung eines Musters während der Durchführung des `NN8`-Befehls '`Step E Learning`' sowie einen 10-Bit De-/Inkrementierer zur Schleifensteuerung erweitert. Demnach werden ebenfalls Netzwerke mit maximal 1024 Neuronen unterstützt.

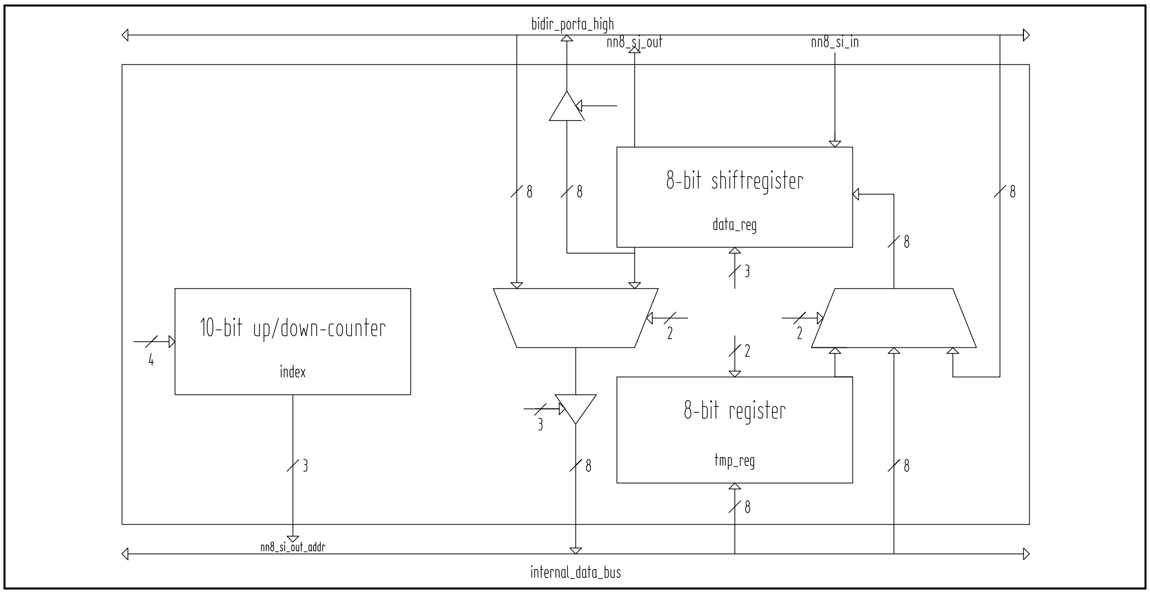

Abbildung 2.5: RT-Beschreibung von `iterate`

Am Ende dieses Abschnittes ist nun ebenfalls, aufbauend auf den nachfolgend vorgestellten Befehlen, eine algorithmische Beschreibung der iterativen Lernregel angegeben.

- `CNN8_load_nj_register_low` *lower part*
- `CNN8_load_nj_register_high` *higher part*
- `CNN8_load_pattern`
- `CNN8_get_si`

- `CNN8_execute_command` *command*

  Kodierung '100 1110'

  Das vom PC übergebene Datum wird an den Command-Port des `NN8-Chips` für insgesamt einen Takt angelegt. Gleichzeitig wird der Tristate-Treiber des `NN8-Controllers` aktiviert, um den Inhalt von `data_reg` auf den Dateneingang des `NN8-Chips` zu legen. Diese Erweiterung war notwendig, um z.B den `NN8`-Befehl '`Load Kappa`' kostengünstig ausführen zu können (2 Takte).

- `CNN8_step_e_learning`

  Kodierung '100 1111'

  Die lokalen Felder eines Neuronenblockes werden aufsummiert (`NN8`-Befehl '`Step E Learning`'), jedoch unter Berücksichtigung des aktuellen Neuronenzustandes. Zur Vorbereitung müssen die Adressregister des `NN8-Controllers` entsprechend geladen werden, ebenfalls muß die Netzwerkgröße über das NJ-Register festgelgt werden. Letzteres ist unbedingt notwendig, da zur Schleifensteuerung das für diese Anwendung benötigte Register fortlaufend dekrementiert wird und daher initial vor jedem Befehlsaufruf neu geladen werden muß (4+NJ*3 Takte, NJ ist dabei die Anzahl der Synapsen pro Neuron).

- `CNN8_sum_invert_sj` *neuron*

  Kodierung '101 0000'

  Über den Befehl `NN8`-Befehl `Sum Invert Sj` wird die Berechnung der Energie E_- vorbereitet. Dazu muß zunächst `synapse_ram_adr_reg` mit der Adresse der gerade zu lernenden Synapse geladen werden. Ferner muß der Wert des jeweiligen Neurons an den Eingang `S_j` des `NN8-Chips` angelegt werden. Dies geschieht durch Auslesen des Blockes, in dem sich das betreffende Neuron befindet, über das Argument *neuron* wird das Neuron indiziert (7 Takte).

- `CNN8_conditional_load_jij`

  Kodierung '101 0001'

  In Abhängigkeit der berechneten Energien E_+ und E_- wird über den `NN8`-Befehl '`Conditional Load Jij`' die nichtinvertierte bzw. invertierte Kopplung in den S_i-Registern des `NN8-Chips` abgelegt. Der `NN8-Controller` generiert hierfür die benötigten Adress- und Steuersignale – `synapse_ram_adr_reg` muß daher die Adresse der ausgewählten Kopplung enthalten (4 Takte).

Mit n sei die Anzahl der Neuronenblöcke (*stripes*) bezeichnet (jeder Block besteht aus 8 Neuronen), p ist die Anzahl zu trainierender Muster (*pattern sets*). Wieder wird hier auf die Literatur [Hen94],[Hen95] verwiesen.

Über die äußeren beiden Schleifen wird die jeweils zu lernende Synapse ausgewählt. In der inneren Schleife werden anschließend die lokalen Feld für sämtliche zu speichernden Muster berechnet, um daraus die Energien zu berechnen.

```
Algorithmus Iterate

 WHILE stability criteria are not fulfilled BEGIN
   FOR stripe=0 TO n − 1 BEGIN
     FOR synapse=0 To 8 · n − 1 BEGIN
       CNN8_execute_command Set_Theta
       CNN8_execute_command Clear_e_p
       CNN8_execute_command Clear_e_m
       FOR set=0 TO p − 1 BEGIN
         CNN8_execute_command Clear_hi
         CONTR_write2pattern_adr_reg set · n
         CONTR_write2synapse_adr_reg 8 · stripe · n
         CNN8_step_e_learning
         CNN8_execute_command Calc Kappa Sub Hi
         CNN8_execute_command Eval Theta
         CNN8_execute_command Calc Ep Plus Hi
         CONTR_write2pattern_adr_reg set · n + synapse/8
         CONTR_write2synapse_adr_reg 8 · stripe · n + synapse
         CNN8_sum_invert_sj synapse mod 8
         CNN8_execute_command Eval Theta
         CNN8_execute_command Calc Em Plus Hi
       END
       CNN8_execute_command Set Theta
       CNN8_execute_command Calc Ep Sub Em
       CNN8_cond_load_jij
       CNN8_get_si
       CNN8_write_data_reg2synapse_ram
     END
   END
 END
```

2.3.5.3 Design `dynamic`

Das Design `dynamic` hat die Identifikation '00001111' und unterstützt die Durchführung der sequentiellen Netzwerk-Dynamik – aus Platzgründen mußte jedoch die Netzgröße auf maximal 1024 Neuronen beschränkt werden.
Das Grunddesign wurde dazu neben einem weiteren Steuerwerk um ein 8-Bit Register zur Zwischenspeicherung eines Musters während der Durchführung des `NN8`-Befehls '`Step Dynamics`', ein 10-Bit Register zur zur Speicherung der Netzgröße sowie einen 10-Bit und 7-Bit Zähler zur Ablaufsteuerung des am Ende dieses Abschnittes angegebenen Algorithmus erweitert. Ferner wurden Befehle geschaffen, die das gleichzeitige Adressieren des Muster- und Synapsenspeichers ermöglichen.

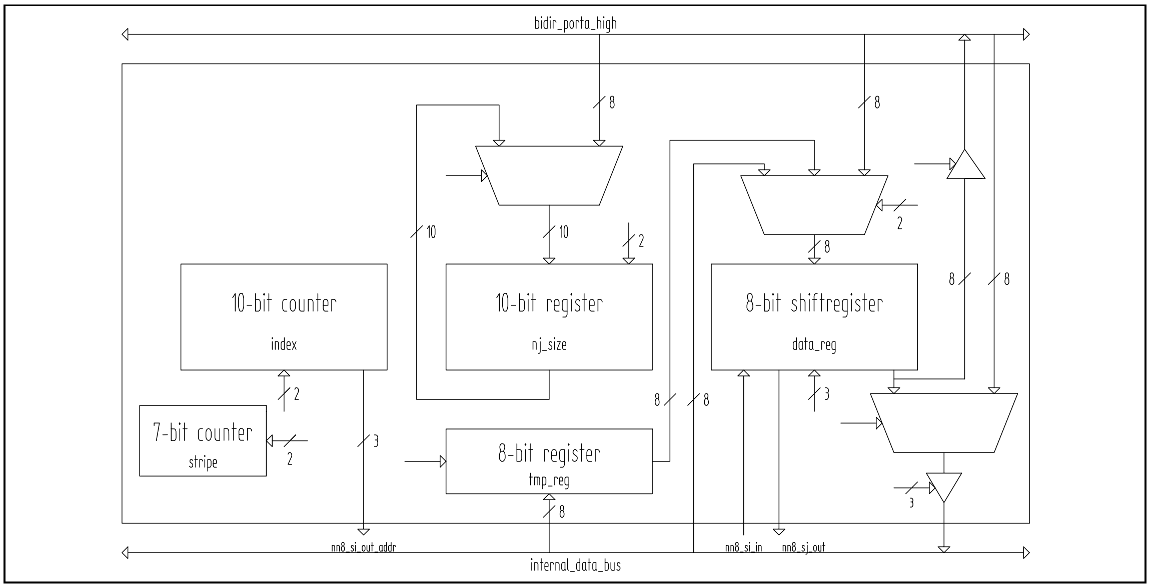

Abbildung 2.6: RT-Beschreibung von `dynamic`

- `CNN8_load_nj_register_low` *lower part*
- `CNN8_load_nj_register_high` *higher part*
- `CNN8_step_dynamics`

  Kodierung '100 0001'

  Durchführung eines Schrittes der sequentiellen Netzwerkdynamik. Voraussetzung ist hierfür ein entsprechendes Laden des NJ-Registers mit dem Wert NJ-1. Zu Beginn der Ausführung dieses Befehls werden die Zähler `pattern_ram_adr_reg` sowie `synapse_ram_adr_reg` und das Register `basis_reg` zurückgesetzt. Demzufolge muß das zu bearbeitende Muster ab Adresse 0 des Musterspeichers abgelegt sein ($1+NJ/8 \cdot (17+3 \cdot NJ)$ Takte, NJ ist dabei die Anzahl der Synapsen pro Neuron).

Nachfolgend ist der mit dem Controller-Befehl `CNN8_step_dynamics` realisierte Algorithmus zur Durchführung der sequentiellen Netzwerkdynamik angegeben. Die Abbruchbedingungen der beiden ineinander geschachtelten Schleifen werden direkt aus dem Inhalt des NJ-Registers abgeleitet, das die Anzahl der Synapsen pro Neuron enthält. Die Anzahl der Neuronenblöcke ergibt sich dabei aus der Division durch 8, da in jeder Speicherzelle die Werte von jeweils 8 Neuronen abgelegt sind.

Während der Durchführung werden jeweils alle 24 Takte (ein Summationsschritt dauert 3 Takte) parallel Lesezugriffe auf den Speicher für die Muster sowie den Speicher für synaptische Gewichte ausgeführt. Das gelesene Wort aus dem Musterspeicher wird dabei zunächst im Register `tmp_reg` zwischengespeichert, um zu Beginn des nächsten Zyklus in `data_reg` kopiert zu werden. In allen anderen Zyklen erfolgt lediglich ein Shifting von `data_reg`, um an den Ausgang `nn8_sj_out` des `NN8-Controllers` den Wert des gerade adressierten Neurons anzulegen.
Beachtet werden sollte die Festlegung, daß die Neuronenzustände im Musterspeicher beginnend bei Adresse 0 abgelegt sind.

```
Algorithmus Dynamic

  /* Instruct Slave to clear all on-chip registers, special command */
  reset_all
  FOR stripe=0 TO n − 1 BEGIN
    CONTR_reset_pattern_adr_reg
    CONTR_increm_read_pattern
    /* Initiate the NN8 cycle for the dynamic */
    CNN8_step_dynamics
    FOR synapse=0 TO 8 · n − 1 BEGIN
      IF synapse mod 8==7 THEN
        /* Initiate in parallel new pattern-read cycle */
        CONCURRENT
          CONTR_increm_read_pattern
          NN8_synapse_ram_read_cycle
      ELSE
        /* Simple shift of the register data_reg */
        NN8_synapse_ram_read_cycle
    END
    CNN8_get_si
    /* Instruct Slave to choose the right stripe, special command */
    copy_basis2pattern_adr_reg
    CONTR_increm_write_pattern
    /* Instruct Slave to save next stripe address, special command */
    copy_pattern2basis_adr_reg
  END
```

2.3.6 Anbindung des NN8-Chips

Die vom NN8-Chip bereitgestellten Steuer- und Adressignale für den Zugriff auf den Speicher für synaptische Gewichte konnten aus Platzgründen nicht benutzt werden: Hierfür wären Multiplexer sowohl für die Steuer- als auch die Adressignale notwendig geworden, da unabhängig von den Speicherzugriffen des NN8-Chips vom Host aus ebenfalls eine Schreib- und Lesemöglichkeit bestehen muß und diese durch den NN8-Controller zur Verfügung gestellt wird.

Das chip_select-Signal des NN8-Chips ist zur Einsparung von Pinressourcen ständig aktiv, dafür wird lediglich im Bedarfsfall an den NN8-Kommandoport für jeweils einen Takt ein vom NOP-Befehl abweichendes Wort angelegt.

Der NN8-Datenbus wurde direkt an den Bus zwischen NN8-Controller, Slave und den Speicher für Muster angeschlossen, lediglich der Port für synaptische Gewichte Jij wurde von jenem durch zwei Bustreiber getrennt, d.h. durch eine Stufe vom Speicher für synaptische Gewichte und eine weitere Stufe vom sog. internal_data_bus.

Die Signale ext_in sowie sj_in des NN8-Chips werden beide von einer ausgezeichneten Zelle des NN8-Controller Datenregisters data_reg abgeleitet und konnten so auf einen Ausgang des CPLDs beschränkt werden.

Schließlich wurden die Signale si_out_adr zum Auslesen der Neuronenzustände vom jeweiligen Inhalt des in allen Entwürfen implementierten Zählers abgeleitet.

Kapitel 3

Softwareanbindung des Systems

3.1 Einleitung

In diesem Kapitel soll nun die in C++ entwickelte Softwarebibliothek vorgestellt werden, die es gestattet, mit verhältnismäßig geringem Aufwand die Möglichkeiten des im vorigen Kapitel vorgestellten Systems in eigenen Applikationen zu nutzen.

Aufbauend auf dieser Bibliothek wurden verschiedene Programme entwickelt, die es zum einen gestatten, die Funktionalität des `NN8-Chips` zu überprüfen sowie Systemkomponenten zu testen, zum anderen aber auch eine Applikation, die es schließlich ermöglicht, den `NN8-Chip` praktisch einzusetzen.

Bei der Entwicklung wurde, soweit möglich, auf Portabilität geachtet — hardwareabhängige Routinen wurden in der nachfolgend beschriebenen Klasse `IOCard` separiert.
Die Quelltexte wurden mit dem C++-Compiler, Version 3.1, von Borland übersetzt.

Zunächst wird in Abschnitt 3.2 eine Beschreibung der zugrundeliegende Datenstrukturen gegeben; anschließend werden die in jenen Klassen implementierten Methoden vorgestellt und schließlich folgt in Abschnitt 3.4 ein Überblick über die gegenwärtig zur Verfügung stehenden Applikationen.

3.2 Datenstrukturen

3.2.1 `class IOCard`

- `iocard.h`
- `iocard.cc`

Auf dieser Low-Level-Ebene sind kartenspezifische Informationen wie z.B. die Basisadresse `io_basis_adr` und relative Adressen bzgl. der Basisadresse zur Adressierung der beiden eingesetzten Portcontroller verfügbar.

Die Kontrollwörter `control_word_PC8255_I` und `control_word_PC8255_II` wurden ausschließlich zu Debugzwecken gespeichert, da ein Auslesen der Konfiguration bei der gewählten Bausteinreihe nicht möglich ist - eine Auswertung dieser Wörter vor der Programmierung der Controller bzw. vor Lese-/Schreibzugriffen erfolgt jedoch z.Zt. nicht.

3.2.2 class FLEXControl

- `flexcntr.h`
- `flexcntr.cc`

Die Klasse `FLEXControl` ermöglicht den Zugriff auf Statussignale des `NN8-Controller` Bausteins. Die Konstruktion eines Objektes ist auch ohne Konfigurierung des Controllers möglich.

3.2.3 class ControllerLink

- `cntrlink.h`
- `cntrlink.cc`

Diese Klasse stellt die Schnittstelle zum `NN8-Controller` dar.
In Abschnitt 2.3.2 wurde bereits das Befehlsformat vorgestellt. `command_toggle` enthält den Wert des Bits 7 beim nächsten auszuführenden Befehl, `initial_toggle` gewährleistet, daß vor dem ersten auszuführenden Befehl `command_toggle` und das im Portcontroller gespeicherte Bit synchronisiert werden über ein entsprechendes Statement im Konstruktor `ControllerLink()`, indem zweimal ein `NOP`-Befehl an den `NN8-Controller` gesandt wird.
Diese Klasse enthält fast ausschließlich atomare Methoden, auf denen sich dann komplexe Funktionen mit Speicheradressierung aufbauen lassen.

3.2.4 class HGNet

- `hgnet.h`
- `hgnet.cc`

In dieser Klasse erfolgt schließlich die softwaretechnische Anbindung des `NN8-Chips`.
PC-seitig werden lediglich Muster verwaltet, die synaptischen Gewichte liegen ausschließlich im Kartenspeicher. Zusätzlich wird bei Bedarf Speicher alloziert zur Mustermanipulation (`net_statep`) - dieser Speicherbereich wird nachfolgend auch als interner Puffer bezeichnet.
Bei den Netzwerkparametern `row_size` und `column_size` entspricht `column_size` der tatsächlichen Netzbreite durch 8. Dies ist jedoch eine interne Festlegung, die die Übergabe des Parameters `column_size` bei einem Funktionsaufruf nicht betrifft.

In gewissen Methoden der Klasse `HGNet` erfolgt die Auswertung von Environmentvariablen, über die individuelle Pfade und Namen der Konfigurationsdateien für den `NN8-Controller` festgelegt werden können — dies sind die Variablen `HG_BASIC_PATH`, `HG_ITERATE_PATH` sowie `HG_DYNAMIC_PATH`. Falls die Environmentvariablen undefiniert sind, wird nach den Dateien '`.\basic.ttf`', '`.\iterate.ttf`' bzw. '`.\dynamic.ttf`' gesucht.

3.2.5 Fehlerbehandlung

- `error_h.h`
- `error_h.cc`

Sämtliche nachfolgend vorgestellten Funktionen überprüfen die Systemumgebung sowie die übergebenen Parameter und rufen im Fehlerfall die Funktion `error_handler(char *,int,char *)` zusammen mit dem Namen der Funktion, in dem der Fehler aufgetreten ist, einem Fehlercode sowie optional einer differenzierten Fehlerbeschreibung auf. Dort wird eine Fehlermeldung generiert, die von der jeweiligen Applikation im Bedarfsfall durch Aufruf der Funktion `eval_error()` auf dem Standardfehlerkanal ausgegeben wird.

Der Fehlercode ist in der global verfügbaren Variablen `global_error` abgelegt und wird durch Aufruf von `eval_error()` auf Null gesetzt. Diese Variable ist unbedingt nach einem Funktionsaufruf auszuwerten, falls der Rückgabewert nicht den Fehlercode repräsentiert. Die Fehlernummern sind grundsätzlich kleiner als Null.

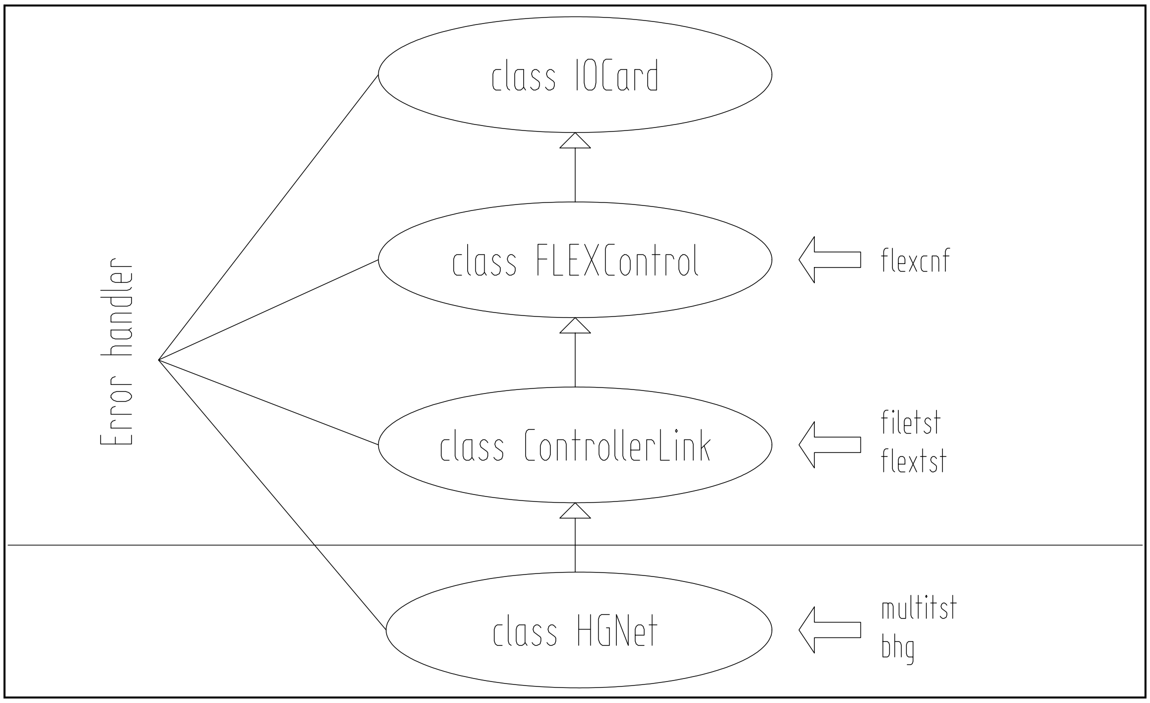

Abbildung 3.1: Softwarekomponenten und -beziehungen

Die obige Abbildung illustriert die Beziehungen zwischen den vorgestellten Softwaremodulen.

3.3 Bibliotheksbeschreibung

3.3.1 `class IOCard`

- `void iopc_reset()`

  Die Portcontroller I und II werden zurückgesetzt.

- `int iopc_program(unsigned int port_controller,unsigned int control_word)`

  Der/die adressierte(n) Portcontroller wird/werden gem. `control_word` programmiert. Für die Programmierung beider Portcontroller wird das Kontrollwort für den Portcontroller I in den unteren 8 Bit erwartet, jenes für den Portcontroller II in den oberen 8 Bit. Bei der Programmierung nur eines Portcontrollers werden die unteren 8 Bit ausgewertet.
  Es werden vor Durchführung des I/O-Befehls die Argumente überprüft — dementsprechend ist auch das Resultat zu bewerten.

  Rückgabe

0	,	falls der I/O-Befehl abgearbeitet wurde,
<0		sonst.

- `int io_write(unsigned int port_controller,unsigned int port,unsigned int data)`

  Das Datum `data` wird an den jeweiligen Port der/des adressierten Portcontroller(s) angelegt. Es werden vor Durchführung des I/O-Befehls die Argumente überprüft — dementsprechend ist auch das Resultat zu bewerten.

  Rückgabe

0	,	falls der I/O-Befehl abgearbeitet werden konnte,
<0		sonst.

- `unsigned int io_read(unsigned int port_controller,unsigned int port)`

  Der jeweilige Port der/des adressierten Portcontroller(s) wird ausgelesen. Es werden vor Durchführung des I/O-Befehls die Argumente überpruft — dementsprechend ist auch das Resultat zu bewerten.

  Rückgabe

  Rückgabewert der Funktion `inport()`, falls der I/O-Befehl abgearbeitet wurde. Ansonsten wird eine Null zurückgegeben.

- `unsigned int iopc_status(unsigned int port_controller=PC8255_both)`

  Das im Objekt gespeicherte Kontrollwort der/des Portcontroller(s) wird zurückgegeben.

3.3.2 class FLEXControl

- `int flex_device_config(const char *configuration_filep,unsigned int block_size=0x8000)`

  Der `NN8-Controller` wird gem. `configuration_filep` konfiguriert. Über den optionalen Parameter `block_size` wird die Anzahl der Bytes festgelegt, die jeweils blockweise aus der Datei gelesen werden. Insbesondere bei relativ langsamen Systemen empfiehlt es sich, `block_size` auf die Größe der Datei zu setzen.

  Rückgabe

0	,	falls das Device konfiguriert werden konnte,
<0		sonst.

- `unsigned int flex_device_status()`

  Der Status des Devices, d.h. logische Signalpegel von `nStatus` sowie `ConfDone`, wird in Form eines Bitvektors zurückgegeben.
  Die Auswertung kann dann erfolgen durch eine geeignete Verknüpfung mit den in der Klasse definierten Konstanten `FLEX_nStatus` und `FLEX_ConfDone`.

3.3.3 class ControllerLink

3.3.3.1 Methoden zur `NN8-Controller` Ansteuerung

- `int flex_config(const char *config_filenamep)`

  Diese Methode ruft die Funktion `FLEXControl.flex_device_config(...)` auf und sendet anschließend zur Synchronisation zwei `CONTR_Nop`-Befehle an den `NN8-Controller`. Die Methode `FLEXControl.flex_device_config` dient nur zur allgemeinen Konfigurierung eines FLEX 8000-Logikbausteins und mußte daher entsprechend erweitert werden.

  Rückgabe

0	,	falls das Device konfiguriert werden konnte,
<0		sonst.

- `void write2controller(unsigned int cmd)`

  Der 8-bit breite Befehl `cmd` wird an den `NN8-Controller` weitergegeben (Vgl. dazu Abschnitt 2.3.2).

- `void write2controller(unsigned int cmd,unsigned int data)`

  Der 16-bit breite Befehl `cmd` wird an den `NN8-Controller` weitergegeben (Vgl. dazu Abschnitt 2.3.2).

- `void prepare4writing()`

  Der Portcontroller II wird für PC-seitige Schreibzugriffe auf Port A (Befehlsweitergabe an den `NN8-Controller`) programmiert.

- `void prepare4reading()`

  Der Portcontroller II wird für PC-seitige Lesezugriffe auf Port A (Auslesen des Registers `data_reg` des `NN8-Controllers`) programmiert.

- `unsigned int get_controller_id()`

  Zunächst wird der Befehl `CONTR_get_controller_id` an den `NN8-Controller` abgesandt. Anschließend wird das Register `data_reg` ausgelesen.

  Rückgabe

  Konfigurationsidentifikation des `NN8-Controllers`, falls dieser konfiguriert wurde.

- `unsigned int controller_is_busy()`

  Der Status des `NN8-Controllers` wird zurückgegeben. Diese Funktion wird u.a. in der Methode `dynamic()` benutzt und sollte grundsätzlich im Zweifel vor dem nächsten, den Controller adressierenden Befehl aufgerufen werden.

  Rückgabe

  1 , falls z.Zt. ein Befehl im `NN8-Controller` abgearbeitet wird,
  0 sonst.

- `unsigned int read_data_reg()`

  Der Inhalt des Registers `data_reg` wird ausgelesen, falls der `NN8-Controller` konfiguriert wurde. Der Portcontroller II wird entsprechend programmiert durch Aufrufe der Methoden `prepare4reading()` und `prepare4writing()`.

- `unsigned int raw_read_data_reg()`

  Der Inhalt des Registers `data_reg` wird ausgelesen, falls der `NN8-Controller` konfiguriert wurde. Der Portcontroller wird *nicht* programmiert. Diese Funktion sollte beim Auslesen des Kartenspeichers verwendet werden, um unnötige I/O-Befehle zu unterdrücken.

- `void write2data_reg(unsigned int data)`

  `data` wird im Register `data_reg` gespeichert, falls der `NN8-Controller` konfiguriert wurde.

3.3.3.2 Methoden zur Adressierung des Musterspeichers

Bei den nachfolgenden Methoden wird auf die Beschreibung im Hardwareteil in Abschnitt 2.3.4 verwiesen. Den Lesezugriffen geht jeweils ein entsprechender `NN8-Controller` Befehl voraus.

- `void clear_pattern_address()`

- `void write_pattern_address(unsigned long address)`

  Der Parameter `address` wird, separiert in entsprechende Adressteile, mit dem Befehl `CONTR_write2pattern_adr_reg` in das `Slave`-Adressregister geschrieben.

- `void increm_pattern_address()`
- `void write_pattern(unsigned int pattern)`
- `void write_pattern_with_address_incr(unsigned int pattern)`
- `unsigned int read_pattern()`
- `unsigned int read_pattern_with_address_incr()`
- `void enter_pattern_ram_fast_reading_mode()`

  Der Befehl `CONTR_init_fast_read_of_pattern_ram` wird an den `NN8-Controller` abgesandt, anschließend wird der Portcontroller II für PC-seitige Lesezugriffe auf Port A programmiert.
- `void leave_pattern_ram_fast_reading_mode()`

  Der Portcontroller II wird für PC-seitige Schreibzugriffe auf Port A programmiert. Anschließend wird der Befehl `exit_fast_read_of_pattern_ram` an den `NN8-Controller` gesandt.
- `void write_data_reg2pattern_ram()`

3.3.3.3 Methoden zur Adressierung des Synapsenspeichers

Bei den nachfolgenden Methoden wird auf die Beschreibung im Hardwareteil in Abschnitt 2.3.4 verwiesen. Den Lesezugriffen geht jeweils ein entsprechender `NN8-Controller` Befehl voraus.

- `void clear_synapse_address()`
- `void write_synapse_address(unsigned long address)`

  Der Parameter `address` wird, separiert in entsprechende Adressteile, mit dem Befehl `CONTR_write2pattern_adr_reg` in das `Slave`-Adressregister geschrieben.
- `void increm_synapse_address()`
- `void write_synapse_weight(unsigned int synapse_weight)`
- `void write_synapse_weight_with_address_incr(unsigned int synapse_weight)`
- `unsigned int read_synapse_weight()`
- `unsigned int read_synapse_weight_with_address_incr()`
- `void enter_synapse_ram_fast_reading_mode()`

  Der Befehl `CONTR_init_fast_read_of_synapse_ram` wird an den `NN8-Controller` abgesandt, anschließend wird der Portcontroller II für PC-seitige Lesezugriffe auf Port A programmiert.

- `void leave_synapse_ram_fast_reading_mode()`

  Der Portcontroller II wird für PC-seitige Schreibzugriffe auf Port A programmiert. Anschließend wird der Befehl `exit_fast_read_of_synapse_ram` an den `NN-Controller` gesandt.

- `void write_data_reg2synapse_ram()`

3.3.4 `class HGNet`

An dieser Stelle muß darauf hingewiesen werden, daß nur in komplexeren Funktionen die Designidentifikation ausgelesen wird. Der Anwender hat daher vor dem Aufruf entsprechender Routinen dieses zu tun und ggf. das Device selbst zu konfigurieren.

3.3.4.1 Methoden zur Netzwerkmanipulation

- `unsigned int nn8_is_busy()`

  Der Status des `NN8-Chips` wird zurückgegeben. Diese Funktion wird u.a. in Methoden zur Ausführung von `NN8`-Befehlen, die wesentlich länger als ein I/O-Zyklus dauern und damit eine Synchronisation mit dem PC bedingen, benutzt.

  Rückgabe

1	,	falls z.Zt. ein Befehl im `NN8-Chip` abgearbeitet wird,
0		sonst.

- `int identify_configuration(unsigned int cnf_should_be)`

  Die Konfigurationsidentifikation des `NN8-Controllers` wird ausgelesen und mit dem Argument `cnf_should_be`, das einer der drei in der Klasse `HGNet` definierten Konstanten `CONTR_basic_design`, `CONTR_iterate_design` bzw. `CONTR_dynamic_design` entsprechen sollte, verglichen.

  Rückgabe

1	,	falls der Vergleich wahr ist,
0		sonst.

- `int resize_net(unsigned int row_size,unsigned int column_size)`

  Die Netzwerkgröße wird den übergebenen Parametern entsprechend gesetzt. Allozierte Speicherbereiche werden bei einer in bezug auf die vorige Einstellung abweichenden Gesamtgröße des Netzwerkes freigegeben.
  Im Fehlerfall, d.h. bei unzulässigen Argumenten wird die aktuelle Netzwerkkonfiguration *nicht* geändert.

  Rückgabe

0	,	falls die Netzwerkgröße geändert wurde,
<0		sonst.

3.3.4.2 Methoden zur Konfigurierung der synaptischen Gewichte

- `int read_synapse_weights_from_file(char *filenamep)`

  Die synaptischen Gewichte werden aus der Datei `filenamep` eingelesen. Dabei wird, falls die Netzwerkparameter von der aktuellen Netzwerkgröße abweichen, `resize_net(...)` aufgerufen.

  Die Datei besteht aus einem Header, in dem die Netzwerkgröße festgelegt ist durch die Zeile 'M $< row\_size > < column\_size > < total\_size >$' und in dem ferner Kommentarzeilen, beginnend mit '#' sowie Leerzeilen, d.h. Zeilen, die lediglich aus Leerzeichen, Tabulatoren sowie LF und CR bestehen, enthalten sein dürfen.

  Anschließend folgt der Datenteil, in dem, jeweils beginnend mit einer Statuszeile, die jedoch z.Zt. nicht ausgewertet wird, die synaptischen Gewichte zwischen den Neuronen eines Blockes und den Neuronen des gesamten Netzwerkes, enthalten sind. Die Daten müssen als Hexadezimalziffern vorliegen, wobei die synaptischen Gewichte des jeweiligen Blocks zu einem anderen Neuron in zwei aufeinanderfolgenden Hexadezimalziffern, also einem Byte, vorliegen müssen.
  LF sowie CR dürfen innerhalb eines die synaptischen Gewichte enthaltenden Blockes beliebig eingestreut werden, empfohlen wird jedoch eine den Parametern *row_size* und *column_size* entsprechende Form.

  Es werden Dateiformate sowohl im UNIX-Format (LF am Zeilenende) als auch im DOS-Format (LF+CR am Zeilenende) verarbeitet.

  Beispiel

  ```
  # Networksize equals 2x8 (2 stripes)
  M 2 8 16

  S 0 16
  1a
  3c
  S 1 16
  56
  f8
  ```

  Rückgabe

0	,	falls die synaptischen Gewichte eingelesen wurden,
<0		sonst.

- `int write_synapse_weights2file(char *filenamep)`

  Die synaptischen Gewichte werden aus dem Kartenspeicher ausgelesen und in der Datei `filenamep` in dem zuvor beschriebenen Format gespeichert.

  Rückgabe

0	,	falls die synaptischen Gewichte geschrieben wurden,
<0		sonst.

- `int init_synapse_weights(int value)`

  Die synaptischen Gewichte werden in Abhängigkeit von `value` mit 1(`value`=1), -1(`value`=-1) bzw. zufällig (`value`=0) initialisiert.

  Rückgabe

0	,	falls die synaptischen Gewichte initialisiert wurden,
<0		sonst.

3.3.4.3 Methoden zur Musterkonfigurierung

- `int read_pattern_from_file(char *filenamep)`

  Die in der Datei `filenamep` gespeicherten Muster werden eingelesen und sowohl im PC als auch im Kartenspeicher abgelegt. Dabei wird, falls die Netzwerkparameter von der aktuellen Netzwerkgröße abweichen, `resize_net(...)` aufgerufen.

  Die Datei besteht aus einem Header, in dem die Netzwerkgröße sowie die Anzahl der Muster durch die Zeile 'P $< number\_of\_pattern >$ $< row\_size >$ $< column\_size >$ $< total\_size >$' festgelegt sind und in dem ferner Kommentarzeilen, beginnend mit '#' sowie Leerzeilen, d.h. Zeilen, die lediglich aus Leerzeichen, Tabulatoren sowie LF und CR bestehen, enthalten sein dürfen.

  Anschließend folgt der Datenteil, in dem, für jedes Muster beginnend mit einer Statuszeile, die jedoch z.Zt. nicht ausgewertet wird, die Neuronenwerte enthalten sind. Die Daten müssen wieder als Hexadezimalziffern vorliegen.
  LF sowie CR dürfen innerhalb eines Musters beliebig eingestreut werden, empfohlen wird jedoch eine den Parametern *row_size* und *column_size* entsprechende Form.

  Es werden Dateiformate sowohl im UNIX-Format (LF am Zeilenende) als auch im DOS-Format (LF+CR am Zeilenende) verarbeitet.

  Beispiel

  ```
  # Networksize equals 2x8 (2 stripes), 3 patterns
  M 3 2 8 16
  P 0 16
  12
  34
  P 1 16
  56
  78
  P 2 16
  9a
  bc
  ```

  Rückgabe

0	,	falls die Muster eingelesen werden konnte,
<0		sonst.

- `int write_pattern2file(char *filenamep)`

  Die *im PC* abgelegten Muster werden in der Datei `filenamep` in dem zuvor beschriebenen Format gespeichert.

  Rückgabe

0	,	falls die Muster geschrieben werden konnte,
<0		sonst.

- `int recall_pattern_from_board_memory(unsigned char *destp,unsigned int index)`

  Das im Kartenspeicher mit `index` indizierte Muster wird in den bei Adresse `destp` beginnenden Speicherbereich kopiert.

  Rückgabe

0	,	falls das Muster kopiert werden konnte,
<0		sonst.

- `int copy_pattern2board_memory(unsigned char *srcp,unsigned int index)`

  Das im PC ab Adresse `srcp` abgelegte Muster wird in den Kartenspeicher an den mit `index` spezifizierten Bereich kopiert.

  Rückgabe

0	,	falls das Muster kopiert werden konnte,
<0		sonst.

Die folgenden Methoden rufen lediglich die beiden Funktionen für Kopiervorgänge zwischen PC und dem Kartenspeicher, `recall_pattern_from_board_memory(...)` sowie `copy_pattern2board_memory(...)`, zusammen mit einem geeigneten Pointer, der den Quell- bzw. Zielspeicherbereich wie z.B. den internen Puffer spezifiziert, auf.

- `int copy_on_board_pattern2internal_buffer(unsigned int index)`

  In Abhängigkeit der aktuellen Netzwerkgröße und des indizierten Musters wird der Kartenspeicher adressiert und der insgesamt `row_size*column_size` Bytes umfassende Speicherbereich in den internen Puffer kopiert.

  Rückgabe

0	,	falls das Muster kopiert werden konnte,
<0		sonst.

- `int copy_nn8_network_state2internal_buffer()`

  In Abhängigkeit der aktuellen Netzwerkgröße wird der Kartenspeicher adressiert und der insgesamt `row_size*column_size` Bytes umfassende Speicherbereich ab Adresse 0 in den internen Puffer kopiert.

  Rückgabe

0	,	falls das aktuelle Muster kopiert werden konnte,
<0		sonst.

- `int actualize_pattern_by_nn8_network_state(unsigned int index)`

  Das durch `index` indizierte Muster im PC wird mit dem ab Adresse 0 im Kartenspeicher vorliegenden Muster überschrieben.

  Rückgabe

0	,	falls das aktuelle Muster kopiert werden konnte,
<0		sonst.

- `int actualize_pattern_by_internal_buffer(unsigned int index)`

  Das durch `index` indizierte Muster im PC wird mit dem im internen Puffer vorliegenden Muster überschrieben.

  Rückgabe

0	,	falls der Pufferinhalt kopiert werden konnte,
<0		sonst.

- `int copy_pattern2nn8_network_state(unsigned int index)`

  Das durch `index` indizierte Muster im PC wird in den Kartenspeicher, beginnend bei Adresse 0, geschrieben.

  Rückgabe

0	,	falls das Muster kopiert werden konnte,
<0		sonst.

- `int copy_pattern2internal_buffer(unsigned int index)`

  Das durch `index` indizierte Muster im PC wird in den internen Puffer kopiert.

  Rückgabe

0	,	falls das Muster kopiert wurde,
<0		sonst.

- `int copy_internal_buffer2nn8_network_state()`

  Der Pufferinhalt wird in den Kartenspeicher, beginnend bei Adresse 0, kopiert.

  Rückgabe

0	,	falls das Muster kopiert wurde,
<0		sonst.

- `int calc_overlap(double m0)`

  Das im internen Puffer vorliegende Muster wird gem.

  $$\mu'_{ij} = \begin{cases} \mu_{ij} & \text{falls } \texttt{m0} > \text{random}() \\ -\mu_{ij} & \text{sonst} \end{cases}$$

  mit $1 \leq i \leq$ `row_size`,$1 \leq j \leq 8 \cdot$ `column_size` sowie random()$\in [0,1]$ randomisiert.

  Für `m0`=1 erhält man die Identität, `m0`<0 invertiert das Muster.

  Rückgabe

0	,	falls das Muster randomisiert wurde,
<0		sonst.

3.3.4.4 Methoden zur Anwendung der Lernregeln sowie der Dynamik

- `int hebbian()`

  Die Hebb-Lernregel wird für die gespeicherten Muster angewandt.
  Vorher wird jedoch die Konfigurationsidentifikation des `NN8-Controllers` abgefragt und ggf. ein anderes Design unter Auswertung der Environmentvariablen `HG_BASIC_PATH` geladen.

  Rückgabe

0	,	falls die Hebb-Lernregel angewandt werden konnte,
<0		sonst.

- `int iterate(int kappa,int selected_stripe)`

  Ein Schritt der iterativen Lernregel bei gegebener Sollstabilität `kappa` wird für den durch `selected_stripe` gewählten Neuronenblock auf alle synaptischen Gewichte dieses Blockes angewandt.
  Vorher wird jedoch die Konfigurationsidentifikation des `NN8-Controllers` abgefragt und ggf. unter Auswertung der Environmentvariablen `HG_ITERATE_PATH` ein anderes Design geladen.
  Falls `selected_stripe`<1 ist, wird die iterative Lernregel für jeden Neuronenblock angewandt.

  Rückgabe

0	,	falls die iterative Lernregel angewandt werden konnte,
<0		sonst.

- `int dynamic()`

  Ein Schritt der parallelen/sequentiellen Dynamik wird für das Netz durchgeführt.
  Vorher wird jedoch die Konfigurationsidentifikation des `NN8-Controllers` abgefragt und ggf. unter Auswertung der Environmentvariablen `HG_DYNAMIC_PATH` ein anderes Design geladen.

  Rückgabe

0	,	falls ein Schritt der Dynamik ausgeführt ausgeführt werden konnte,
<0		sonst.

3.3.4.5 `NN8-Chip` Funktionen

Der Vollständigkeit halber wird hier die Programmierschnittstelle zu den unterstützten `NN8-Chip` Kommandos vorgestellt — für eine entsprechende Befehlsbeschreibung wird auf den Hardwareteil sowie [Hen94] verwiesen. Die Funktionen sind als Inline-Aufrufe implementiert.

Zu beachten ist das unterschiedliche Timing der Funktionen: Ggf. muß entweder innerhalb einer Schleife das `Ready`-Signal des `NN8-Chips` abgefragt werden oder aber, überlappend mit der Befehlsausführung, die zur Verfügung stehende Rechenzeit durch geeignete Statements genutzt werden. Die Funktionen `hebbian()`, `iterate(...)` sowie `dynamic(...)` machen von dieser Möglichkeit jedoch z.Zt. noch keinen Gebrauch und überprüfen bei kritischen Routinen wiederholt die `Busy`- bzw. `Ready`-Signale. Die entsprechenden Routinen, die bei dem hier eingesetzten System kritisch sind, sind gekennzeichnet durch '†'.

- `void nn8_execute_command(unsigned int command_word)`†
- `void nn8_reset()`
- `void nn8_load_nj_register(unsigned int number_of_synapses)`
- `void nn8_load_pattern()`†
- `void nn8_clear_hi()`
- `void nn8_set_si()`
- `void nn8_get_si()`†
- `void nn8_step_hebb(unsigned char bit)`
- `void nn8_step_dynamics()`†
- `void nn8_step_e_learning()`†
- `void nn8_sum_invert_sj(unsigned char bit)`†
- `void nn8_conditional_load_jij()`
- `void nn8_clear_kappa()`†
- `void nn8_load_kappa(unsigned int stability)`
- `void nn8_clear_e_p()`†
- `void nn8_clear_e_m()`†
- `void nn8_calc_ep_sub_em()`†
- `void nn8_calc_kappa_sub_hi()`†
- `void nn8_calc_ep_plus_hi()`†
- `void nn8_calc_em_plus_hi()`†
- `void nn8_calc_kappa_plus_hi()`†
- `void nn8_set_theta()`
- `void nn8_eval_theta()`

3.3.5 Diverse Funktionen

Die im folgenden vorgestellten Funktionen wurden in dem Modul `feature` zusammengefaßt.

- `int watch_stabilities(HGNet& net,unsigned int index,double m0,unsigned int max_depth)`

  Auf das durch `index` indizierte, in `net` gespeicherte, mit `m0` verrauschte Muster werden maximal `max_depth` Schritte der Dynamik angewandt. Der Überlapp, verglichen mit dem Startzustand, wird im Falle eines stabilen Zustands ausgegeben, andernfalls die detektierte Zykluslänge.
  Falls `index`<1 ist, wird die Stabilität für *jedes* Muster berechnet.

  Rückgabe

0	,	falls die Stabilität(en) berechnet werden konnte(n),
<0		sonst.

- `long compute_energies(HGNet& net,unsigned int stripe_index,unsigned int synapse_index,unsigned int kappa)`

  Die Energien E_+ und E_- für die Kopplung `synapse_index` des Blockes `stripe_index` werden für die gespeicherten Muster berechnet.

  Zurückgegeben wird das Maximum der Summe aus den Energien E_+ bzw. E_-, im Falle eines Fehlers jedoch Null. Es sollte daher vor Auswertung des Ergebnisses die Variable `global_error` ausgewertet werden.

- `int display(HGNet& net)`

  Das Muster, das im internen Puffer von `net` gespeichert ist, wird zentriert auf dem Bildschirm ausgegeben.

  Rückgabe

0	,	falls das Muster ausgegeben werden konnte,
<0		sonst.

- `int edit_pattern(HGNet& net)`

  Die im PC gespeicherten Muster können angesehen und editiert werden.

  Rückgabe

0	,	falls Muster gespeichert sind,
<0		sonst.

- `double stopwatch()`

  Über diese Funktion wird die Zeitmessung gestartet und gleichzeitig die zeitliche Differenz zum letzten Aufruf von `stopwatch()` berechnet.

  Rückgabe

  Differenz in $\frac{1}{100}$ Sekunden zum letzten Aufruf der Funktion.

3.4 Applikationen

Im folgenden werden kurz die bereits erwähnten Konvertierungs- und Testprogramme sowie das Hauptprogramm für den praktischen Einsatz des Systems vorgestellt.

3.4.1 `bin2hex` und `hex2bin`

Die Programme `bin2hex` und `hex2bin` wurden geschrieben, um die innerhalb der Dateien für Muster und synaptische Gewichte abliegenden Daten des am AB TECH entwickelten Programms `axon++` in eine Struktur zu konvertieren, die dem zugrundeliegenden Speichermodell entspricht und daher höchste Performance bei I/O-Operationen erwarten läßt. Umgekehrt ist die Konvertierung in das von `axon++` geforderte Dateiformat möglich.

Die Programme existieren in Compilaten für UNIX-Maschinen sowie PCs, wobei insbesondere bei Daten für größere Netze die Anwendung der UNIX-Version empfohlen wird.

- `bin2hex` [-i $< filename >$ -o $< filename >$ -r $< row\_size >$ -c $< column\_size >$]

  Die im Format von `axon++` vorliegende Datei (-i) wird in das HEX-Format (-o) konvertiert. Das Programm verarbeitet sowohl Muster als auch synaptische Gewichte und benötigt zusätzlich die Zeilen- und Spaltengröße (row_size und $column\_size$) des Netzes.

- `hex2bin` [-i $< filename >$ -o $< filename >$]

  Die im HEX-Format vorliegende Datei (-i) wird in das Format von `axon++` konvertiert. Das Programm verarbeitet sowohl Muster als auch synaptische Gewichte.

3.4.2 `flexcnf`

- `flexcnf` [-f $< filename >$]

Der ALTERA FLEX-Logikbaustein wird mit den in $filename$ enthaltenen Daten konfiguriert.

3.4.3 `filetst`

- `filetst` [-f $< filename >$ -o $< filename >$ [-c $< filename >$] [-pattern|-synapse]]

Falls über '-c' eine Datei spezifiziert wurde, so wird der Logikbaustein anfangs konfiguriert. Der Muster- und Synapsenspeicher wird überprüft und/oder ausgelesen. Dazu wird, falls eine Datei als Parameter übergeben wurde (-f), der Dateiinhalt in den mit -pattern bzw. -synapse (Default=-pattern) ausgewählten Speicher geschrieben und anschließend der wieder ausgelesene Speicherinhalt mit dem Dateiinhalt verglichen. Falls eine Ausgabedatei spezifiziert wurde (-o), wird der jeweilige Speicherinhalt in diese Datei geschrieben.

3.4.4 `bhg`

Dieses Programm stellt eine Kombination der zuvor vorgestellten Funktionen dar:
So besteht die Möglichkeit Muster zu editieren, die Hebb- sowie die iterative Lernregel und die Dynamik auf ein gegebenes Netzwerk anzuwenden. Ferner besteht die Möglichkeit zur Überprüfung der Stabilitäten einzelner Muster sowie zu Performanzmessungen.

Das Programm wurde in eine Batch-Datei eingebettet, in der vor dem Aufruf die angesprochenen Environmentvariablen gesetzt werden sowie die Bildschirmauflösung eingestellt wird.

Anhang A

Verwendete Bausteine

A.1 Pin-Assignment Adr_Decode (ALTERA EP610)

Für eine detaillierte Beschreibung der Logikbausteinfamilie ALTERA CLASSIC wird auf [ALT93] sowie [Kli94] verwiesen.

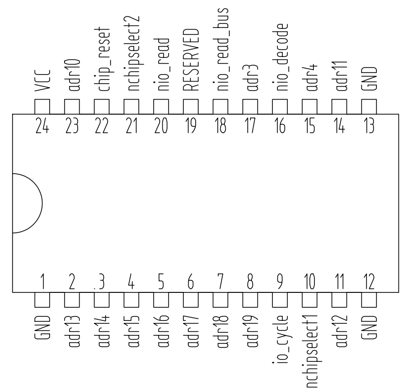

Abbildung A.1: Designspezifisches Pin-Assignment ALTERA EP610 (Adr_Decode)

A.2 Pin-Assignment NN8-Controller (ALTERA EPF8452)

Für eine detaillierte Beschreibung der Logikbausteinfamilie ALTERA FLEX 8000 wird auf [ALT93], [Kli94] sowie [Boh95] verwiesen.

Es wird das passiv-serielle Konfigurationsschema verwendet, bei dem nach einem logischen 1-0-1 Übergang des `nConfig`-Signals begleitet vom Takt `DCLK` die Konfigurationsdaten bitweise zum ALTERA FLEX Baustein über den Portcontroller übertragen werden.

Die Schemavorgabe erfolgt über die Signale `nS/P`, `MSEL0` und `MSEL1` (Vgl. dazu [ALT93, S.403 ff.]).

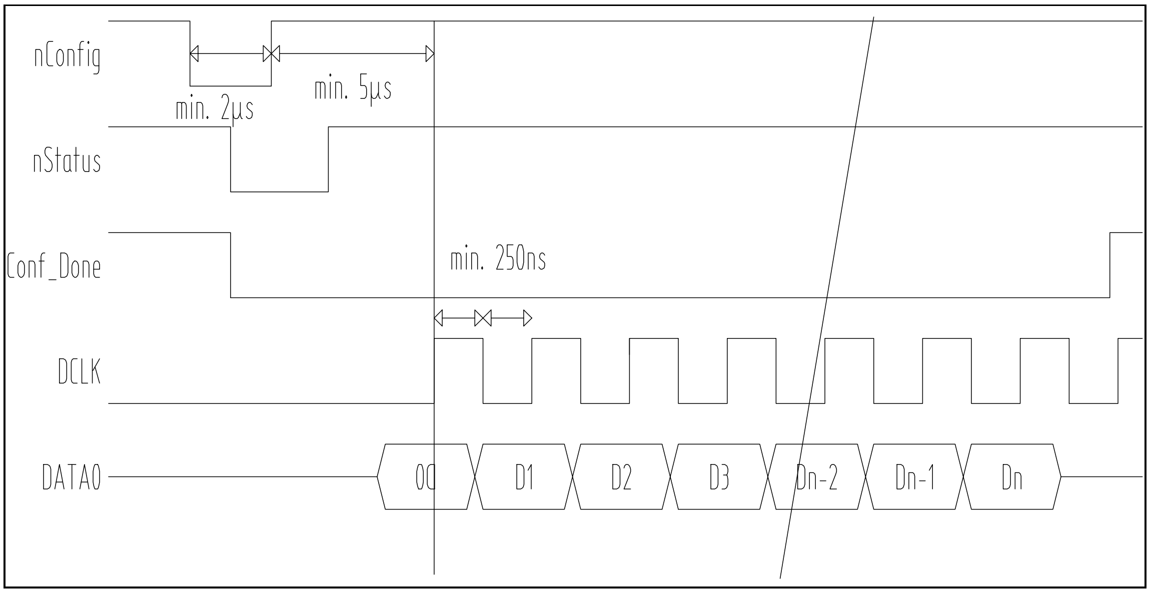

Abbildung A.2: Passiv-serielles Konfigurationsschema ALTERA FLEX 8000

Die Chipgehäuse für den `NN8-Controller` sowie den `Slave` sind identisch und liegen im 84-poligen PLCC-Format vor.

```
 -----------------------------------------------------------------_
/   11 10  9  8  7  6  5  4  3  2  1 84 83 82 81 80 79 78 77 76 75   |
| 12                                                               74 |
| 13                                                               73 |
| 14                                                               72 |
| 15                                                               71 |
| 16                                                               70 |
| 17                                                               69 |
| 18                                                               68 |
| 19                                                               67 |
| 20                                                               66 |
| 21                                                               65 |
| 22                         EPF8452LC84-2/EPM7128                 64 |
| 23                                                               63 |
| 24                                                               62 |
| 25                                                               61 |
| 26                                                               60 |
| 27                                                               59 |
| 28                                                               58 |
| 29                                                               57 |
| 30                                                               56 |
| 31                                                               55 |
| 32                                                               54 |
|_  33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53  _|
 -------------------------------------------------------------------
```

Anschlußbelegung des `NN8-Controller`-Bausteins

clk	12	internal_data_bus7	67	Conf_Done	11
por	2	internal_data_bus6	66	DCLK	10
nio_read	29	internal_data_bus5	65	DATA0	14
busy	27	internal_data_bus4	64	nStatus	32
porta_low7	43	internal_data_bus3	63	nConfig	33
porta_low6	44	internal_data_bus2	62	nSP	75
porta_low5	45	internal_data_bus1	61	MSEL1	53
porta_low4	46	internal_data_bus0	60	MSEL0	74
porta_low3	48				
porta_low2	49	nn8_ext_request	13		
porta_low1	50	nn8_si_out_addr2	72		
porta_low0	51	nn8_si_out_addr1	16		
bidir_porta_high15	34	nn8_si_out_addr0	6		
bidir_porta_high14	35	nn8_cmd7	76		
bidir_porta_high13	36	nn8_cmd6	77		
bidir_porta_high12	37	nn8_cmd5	78		
bidir_porta_high11	39	nn8_cmd4	79		
bidir_porta_high10	40	nn8_cmd3	81		
bidir_porta_high9	41	nn8_cmd2	82		
bidir_porta_high8	42	nn8_cmd1	83		
slave_cmd4	24	nn8_cmd0	84		
slave_cmd4	58	nn8_si_in	69		
slave_cmd2	57	nn8_sj_out	20		
slave_cmd1	56				
slave_cmd0	55				

- **VCC- und GND-Anschlüsse**

  `VCC : 17,38,52,59,80`

  `GND : 5,26,31,47,54,68,73`

- **Beschaltung zur passiv-seriellen Konfigurierung**

  Pull-Up Widerstände zwischen Pins `11,32,33` und `VCC`

  `VCC : 53`

  `GND : 74,75`

A.3 Pin-Assignment `Slave` (ALTERA EPM7128)

Für eine detaillierte Beschreibung der Logikbausteinfamilie ALTERA MAX 7000 wird auf [ALT93] sowie [Kli94] verwiesen.

Anschlußbelegung des Slave-Bausteins

clk	34	pattern_ram_nchip_enable	58	synapse_ram_nchip_enable	41
busy	36	pattern_ram_noutput_enable	61	synapse_ram_noutput_enable	37
por	35	pattern_ram_nwrite_enable	33	synapse_ram_nwrite_enable	39
cmd4	31	pattern_ram_adr_out16	9	synapse_ram_adr_out16	75
cmd3	27	pattern_ram_adr_out15	57	synapse_ram_adr_out15	73
cmd2	28	pattern_ram_adr_out14	50	synapse_ram_adr_out14	76
cmd1	29	pattern_ram_adr_out13	14	synapse_ram_adr_out13	81
cmd0	30	pattern_ram_adr_out12	15	synapse_ram_adr_out12	77
data_in7	18	pattern_ram_adr_out11	16	synapse_ram_adr_out11	74
data_in6	17	pattern_ram_adr_out10	45	synapse_ram_adr_out10	80
data_in5	20	pattern_ram_adr_out9	49	synapse_ram_adr_out9	79
data_in4	21	pattern_ram_adr_out8	5	synapse_ram_adr_out8	69
data_in3	22	pattern_ram_adr_out7	44	synapse_ram_adr_out7	67
data_in2	23	pattern_ram_adr_out6	46	synapse_ram_adr_out6	63
data_in1	24	pattern_ram_adr_out5	54	synapse_ram_adr_out5	64
data_in0	25	pattern_ram_adr_out4	55	synapse_ram_adr_out4	70
nroute_nn82synapse_ram	11	pattern_ram_adr_out3	56	synapse_ram_adr_out3	71
nroute_flex2synapse_ram	10	pattern_ram_adr_out2	60	synapse_ram_adr_out2	65
		pattern_ram_adr_out1	8	synapse_ram_adr_out1	6
		pattern_ram_adr_out0	12	synapse_ram_adr_out0	4

- VCC- und GND-Anschlüsse

  `VCC : 3,13,26,38,43,53,66,78`

  `GND : 1,2,7,19,32,42,47,59,72,82,83,84`

A.4 Pin-Assignment Portcontroller 8255

Die beiden Portcontroller werden ausschließlich im Mode 0 (basic input/ output) betrieben, d.h. einem Modus ohne Handshaking, in dem die Ports beschrieben bzw. gelesen werden können (Vgl. [SIE90, S.761 ff.]).

Die Ports A der beiden Portcontroller werden zum Datenaustausch zwischen PC und `NN8-Controller` verwendet, über Port B des Portcontrollers können die Zustände des Controllers sowie des `NN8-Chips` abgefragt werden.
Port C des Portcontrollers I (Bits 7..0) wurde mit Steuer- und Statussignalen des ALTERA FLEX 8000 Logikbausteins belegt (die Bits 3..0 sind dabei als Ausgänge zum Logikbaustein hin konfiguriert, die Bits 7..4 dienen als Eingänge für vom IC zur Verfügung gestellte Signale).

Eine Besonderheit ergab sich bei dem Signal `nConfig`, das benötigt wird, um die Konfigurierung des Devices zu initiieren (logischer 1-0-1 Übergang). Bei der Programmierung werden sämtliche Latches der Portcontroller zurückgesetzt, daher wird das Signal invertiert dem FLEX Baustein zugeführt und dementsprechend auf der Softwareseite dieser Umstand berücksichtigt.

Die beiden `Chipselect`-Anschlüsse sind nicht kurzgeschlossen. Es ist daher möglich, die Portcontroller getrennt voneinander zu adressieren.

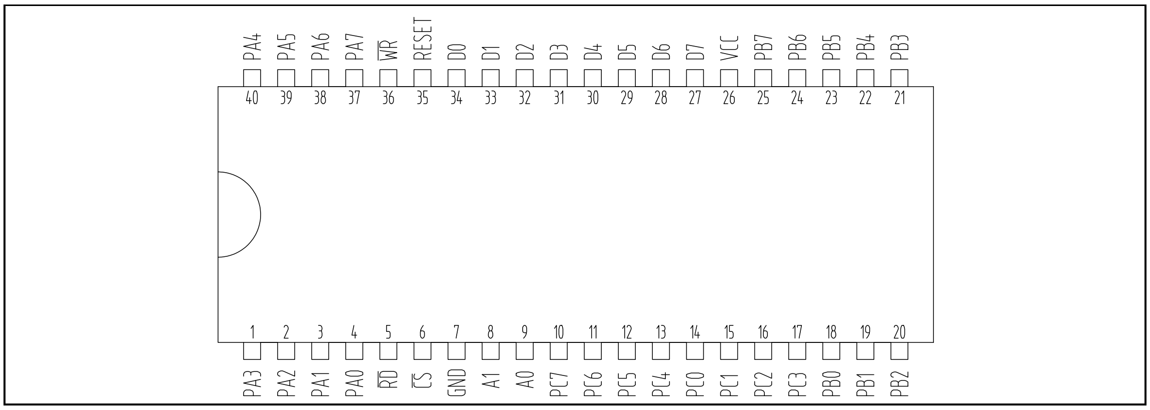

Abbildung A.3: Portcontroller OKI 82C55A-2

A.5 Pin-Assignment Cache RAM 8×128K

Zur Speicherung von Mustern und synaptischen Gewichten wurden SRAM-Bausteine verwendet, um zum einen kurze Zugriffszeiten zu ermöglichen, aber auch, um den Entwurfsprozeß zu vereinfachen.
Ausgewählt wurden Chips mit einer Kapazität von 8×128K sowohl für Muster als auch für synaptische Gewichte mit einer Zugriffszeit von 20ns (Vgl. [MOT93, S.3-45 ff.]).

`nChipselect`	`nOutputEnable`	`nWriteEnable`	*Action*
1	X	X	*Nothing*
0	0	1	*Read cycle*
0	X	0	*Write cycle*

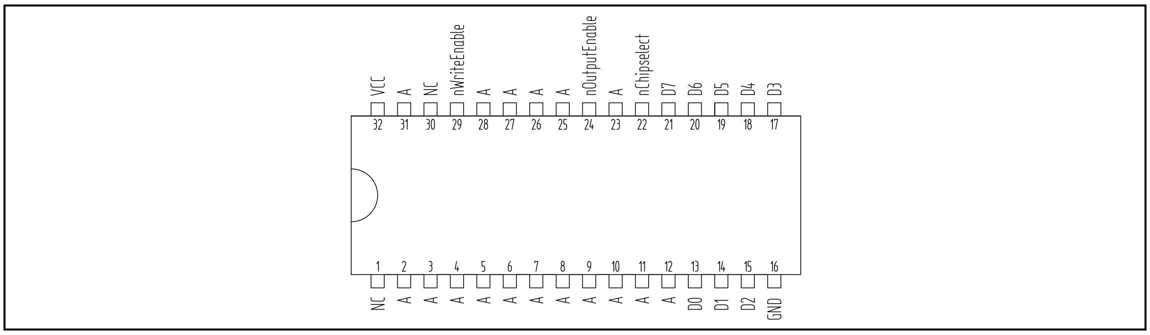

Abbildung A.4: Cache RAM ALLIANCE AS7C1024-20TPC

Anhang B

Verbindungen IBM Prototype Adapter

Dieser Abschnitt dokumentiert den Schaltplan des realisierten Systems in Form einer Verbindungsliste.

- `Prototype Adapter-Address-Decoder`

`-I/O DECODE`	$\Rightarrow$	`nio_decode`
`+I/O CYCLE`	$\Rightarrow$	`io_cycle`
`-BIOR`	$\Rightarrow$	`nio_read_bus`

  Die PC-Adressleitungen $\mathtt{A}[i]_{i=3,4,10..19}$ sind an die entsprechend bezeichneten Pins angeschlossen. Die PC-Adressleitungen `A2` und `A1` sind direkt mit den Pins `A1` und `A0` der beiden Portcontroller verbunden.

- `Address-Decoder`-Portcontroller I und II

`chip_reset`	$\Rightarrow$	RESET	I,II
`nchip_select1`	$\Rightarrow$	$\overline{\mathtt{CS}}$	I
`nchip_select2`	$\Rightarrow$	$\overline{\mathtt{CS}}$	II
`nio_read`	$\Rightarrow$	$\overline{\mathtt{RD}}$	I,II

- Portcontroller-`NN8-Controller`

  Die Anschlüsse $\mathtt{PA}[i]_{i=0..7}$ des Portcontrollers I sind mit den Pins $\mathtt{porta\_low}[i]_{i=0..7}$ des `NN8-Controllers` verbunden, die $\mathtt{PA}[i]_{i=0..7}$ des Portcontrollers II mit den Pins $\mathtt{bidir\_porta\_high}[i]_{i=8..15}$.

  Die Signale `nio_read`, `A2` und `A1` sind über ein logisches Oder mit dem Pin `nio_read` des `NN8-Controllers` verbunden.

- NN8-Controller-Slave

internal_data_bus7	⇒	data_in7	slave_cmd4	⇒	cmd4
internal_data_bus6	⇒	data_in6	slave_cmd3	⇒	cmd3
internal_data_bus5	⇒	data_in5	slave_cmd2	⇒	cmd2
internal_data_bus4	⇒	data_in4	slave_cmd1	⇒	cmd1
internal_data_bus3	⇒	data_in3	slave_cmd0	⇒	cmd0
internal_data_bus2	⇒	data_in2			
internal_data_bus1	⇒	data_in1			
internal_data_bus0	⇒	data_in0			

Die internal_data_bus$[i]_{i=0..7}$-Punkte sind durch Pull-Up Widerstände gegen VCC, die slave_cmd$[i]_{i=0..4}$-Punkte durch Pull-Down Widerstände gegen GND (NOP-Befehl bei fehlender Konfiguration) abgesichert.

- NN8-Controller-NN8-Chip

nn8_cmd7	⇒	cmd_port7	internal_data_bus4	⇒	data_port4
nn8_cmd6	⇒	cmd_port6	internal_data_bus3	⇒	data_port3
nn8_cmd5	⇒	cmd_port5	internal_data_bus2	⇒	data_port2
nn8_cmd4	⇒	cmd_port4	internal_data_bus1	⇒	data_port1
nn8_cmd3	⇒	cmd_port3	internal_data_bus0	⇒	data_port0
nn8_cmd2	⇒	cmd_port2	nn8_ext_request	⇐	ext_request
nn8_cmd1	⇒	cmd_port1	nn8_sj_out	⇒	Sj_in
nn8_cmd0	⇒	cmd_port0	nn8_si_in	⇐	Si_out
internal_data_bus7	⇒	data_port7	nn8_si_out_adr2	⇒	Si_out_addr2
internal_data_bus6	⇒	data_port6	nn8_si_out_adr1	⇒	Si_out_addr1
internal_data_bus5	⇒	data_port5	nn8_si_out_adr0	⇒	Si_out_addr0

Die nn8_cmd$[i]_{i=0..7}$-Punkte sind durch Pull-Down Widerstände gegen GND (NOP-Befehl bei fehlender Konfiguration), die anderen Pfade vom NN8-Controller zum NN8-Chip durch Pull-Up Widerstände gegen VCC abgesichert.

- Slave-Speicher für synaptische Gewichte

synapse_ram_nchip_enable	⇒	nChipselect
synapse_ram_noutput_enable	⇒	nOutputEnable
synapse_ram_nwrite_enable	⇒	nWriteEnable

Die Adressleitungen synapse_ram_adr_out$[i]_{i=0..16}$ sind an die Pins, die mit A gekennzeichnet sind, angeschlossen.

- `Slave`-Speicher für Muster

`pattern_ram_nchip_enable`	$\Rightarrow$	`nChipselect`
`pattern_ram_noutput_enable`	$\Rightarrow$	`nOutputEnable`
`pattern_ram_nwrite_enable`	$\Rightarrow$	`nWriteEnable`

  Die Adressleitungen `pattern_ram_adr_out`$[i]_{i=0..16}$ sind an die Pins, die mit `A` gekennzeichnet sind, angeschlossen.

- `Slave`-Bustreiber zwischen `Jij`$[i]_{i=0..7}$ (`NN8-Chip`) und `D`$[i]_{i=0..7}$ (Speicher für synaptische Gewichte)

`nroute_nn82synapse_ram`	$\Rightarrow$	$\overline{\texttt{G}}$
`synapse_ram_noutput_enable`	$\Rightarrow$	`DIR`

  Die `Jij`$[i]_{i=0..7}$ des `NN8-Chips` sind mit den Pins `D`$[i]_{i=0..7}$ des Bustreibers verbunden, die `Y`$[i]_{i=0..7}$ des Bustreibers mit den Pins `D`$[i]_{i=0..7}$ des Speichers für synaptische Gewichte.

- `Slave`-Bustreiber zwischen `internal_data_bus`$[i]_{i=0..7}$ (`NN8-Controller`) und `D`$[i]_{i=0..7}$ (Speicher für synaptische Gewichte)

`nroute_flex2synapse_ram`	$\Rightarrow$	$\overline{\texttt{G}}$
`synapse_ram_noutput_enable`	$\Rightarrow$	`DIR`

  Die `internal_data_bus`$[i]_{i=0..7}$ sind mit den Pins `D`$[i]_{i=0..7}$ des Bustreibers verbunden, die `Y`$[i]_{i=0..7}$ des Bustreibers mit den Pins `D`$[i]_{i=0..7}$ des Speichers für synaptische Gewichte.

  Die Punkte zwischen den Datenports der beiden Bustreiber und des Datenports des Speichers für synaptische Gewichte sind durch Pull-Up Widerstände gegen VCC abgesichert.

Literaturverzeichnis

[ALT92] *ALTERA MAX+PlusII Reference Manual.* ALTERA Corp., San Jose, CA, USA, 1992.

[ALT93] *ALTERA Data Book.* ALTERA Corp., San Jose, CA, USA, 1993.

[Boh95] Frank Bohnsack. *Entwurf eines ASICs zur Mustererkennung auf Basis von FPGAs.* Universität Hamburg, Fachbereich Informatik, 1995. Studienarbeit.

[GP94] Manfred Glesner and Werner Pöchmüller. *Neurocomputers - An overview of neural networks in VLSI.* Chapman & Hall, London, UK, 1994.

[Hen93] Norman Hendrich. *Phase-space gardening in the binary couplings memory network.* 1993. PROC. ICANN-93, Amsterdam.

[Hen94] Norman Hendrich. *NN8_Chip:Hardwarerealisierung eines Hopfield-Gardner neuronalen Netzes.* Universität Hamburg, Fachbereich Informatik, 1994. Fachbereichsmitteilung Nr. 238.

[Hen95] Norman Hendrich. *Local stability learning rules and phase space gardening on neural networks.* Universität Hamburg, Fachbereich Informatik, 1995. Fachbereichsbericht Nr. 173.

[Kli94] André Klindworth. *Einführung in AHDL - Praktikumsunterlagen „FPGA's".* Universität Hamburg, Fachbereich Informatik, 1994.

[MOT93] *MOTOROLA Fast Static RAM, BiCMOS, CMOS, and Module Data.* MOTOROLA, Phoenix, AZ, USA, 1993.

[Roj93] Raúl Rojas. *Theorie der neuronalen Netze.* Springer-Verlag, Berlin-Heidelberg, 1993.

[SIE90] *SIEMENS PC Peripherals System Components.* SIEMENS, München, 1990.

[Str93] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, USA, second edition, 1993.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++.* Addison-Wesley, Reading, MA, USA, 1994.