Fachbereich Informatik der Universität Hamburg

Vogt-Kölln-Str. 30 · D–22527 Hamburg · Germany

University of Hamburg — Computer Science Department

Mitteilung Nr. 238 · Memo No. 238

NN8_CHIP: Hardwarerealisierung eines Hopfield-Gardner neuronalen Netzes

Norman Hendrich, Andreas Mäder

Arbeitsbereich TECH

FBI–HH–M–238/94

Juli 1994

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

Der vorliegende Bericht ist die technische Dokumentation zu `NN8_CHIP`, der ersten Implementation der in [Hendrich 94] vorgeschlagenen voll skalierbaren Architektur für Hopfield-Gardner Netzwerke mit binären Kopplungen.

Obwohl die Architektur eigentlich in Hinblick auf Wafer-Scale Integration ausgelegt ist, macht auch der Entwurf eines einfachen Testchips mit nur acht Neuronen Sinn: `NN8_CHIP` dient als Prototyp, der mit geringem Aufwand erlaubt, die Korrektheit der Architektur und der Algorithmen zu testen. Ein Minimalsystem mit einem oder wenigen `NN8_CHIP`'s und SRAMs erlaubt zudem den Test der implementierten Lernregel für große Netzwerke, die auf normalen Workstations wegen enormer Rechenzeiten kaum zugänglich sind.

Das Design wurde im Rahmen des Entwurfsprojektes `31.331` im WS 93 begonnen und bis zum SS 94 fertiggestellt. Dabei konnten auch zum ersten Mal die, über EUROCHIP zur Verfügung gestellten, state-of-the-art Programme *Synopsys VSS*, *Synopsys Design Compiler* und *Cadence OPUS* eingesetzt werden.

- In einer kurzen Übersicht wird zunächst der Aufbau eines Hopfield-Gardner Netzwerks skizziert. Dann werden die Struktur von `NN8_CHIP` und einige Optimierungen beschrieben.
- Ein kurzer Abschnitt faßt die Erfahrungen mit den beim Entwurf von `NN8_CHIP` benutzten Werkzeugen zusammen. Dies erscheint sinnvoll, da für `NN8_CHIP` erstmals der Entwurfsablauf `VHDL` → *Synopsys VSS* → *Synopsys Design Compiler* → *Cadence OPUS* eingesetzt wurde.
- Darauf folgt die vollständige Beschreibung des in `NN8_CHIP` implementierten Befehlssatzes. Dazu gehört auch die Beschreibung des Chip-Timings.
- Der Bericht schließt mit den technischen Dokumentationen zum Pinout und den für die Postlayout Simulationen benutzten Simulationsstimuli.

2 Architektur von NN8_CHIP

An dieser Stelle sollen die Grundlagen von Hopfield-Gardner Netzwerken nur grob skizziert werden, sofern sie für `NN8_CHIP` von Bedeutung sind. Für eine vollständige Beschreibung von Hopfield-Gardner Netzwerken und der besonderen Eigenschaften binär gekoppelter Netzwerke sei auf die Literatur [Hendrich 94] verwiesen.

2.1 Grundlagen

Ein Hopfield-Gardner neuronales Netzwerk besteht aus einer Anzahl von N miteinander gekoppelten, sehr einfachen Prozessoren, den Neuronen. Jedes Neuron i hat den Wert $S_i = \pm 1$, und berechnet im einfachsten Fall der parallelen Dynamik seinen Ausgangswert $S_i(t)$ gemäß

$$S_i(t+1) = N^{-1/2} \sum_{j \neq i} J_{ij} S_j(t). \tag{1}$$

Hopfield konnte [Hopfield 82] zeigen, daß ein derartiges System als assoziativer Speicher für Binärmuster $\xi_i^\mu = \pm 1$ $(i = 1 \ldots N,\ \mu = 1 \ldots P)$ dient, wenn die Kopplungen entsprechend der Hebb-Lernregel eingestellt werden,

$$J_{ij} = N^{-1} \sum_{\mu=0}^{P-1} \xi_i^\mu \xi_j^\mu. \tag{2}$$

In den hier betrachteten binär gekoppelten Netzwerken werden die Werte der J_{ij} auf $J_{ij} = \pm 1$ eingeschränkt (geclippte Hebb-Lernregel).

Die ebenfalls in `NN8_CHIP` implementierte iterative Lernregel ist etwas aufwendiger und in [Hendrich 94] ausführlich beschrieben.

2.2 Eine skalierbare, minimale Architektur

Im Gegensatz zu anderen neuronalen Netzwerken, die teilweise eine beträchtliche numerische Präzision für die Werte der Kopplungen und Neuronen benötigen, funktioniert das Hopfield-Gardner Netzwerk auch mit binären Kopplungen — und bietet sich daher für eine digitale Implementation besonders an.

Die hier vorgeschlagene Architektur für ein Hopfield-Gardner Netzwerk ist durch die Beschränkung auf binäre Kopplungen in Bezug auf den Hardwareaufwand (jedenfalls gemessen an digitalen Realisierungen) minimal.

Da keine Kommunikationsflaschenhälse existieren, kann die Architektur in weiten Grenzen skaliert werden. Im Prinzip sind — außer in Bezug auf die Realisierungskosten — Netzwerke mit mehreren 10000 Neuronen und jeweils entsprechend vielen Synapsen pro Neuron kein Problem. Die Architektur ist daher gerade auch für Wafer-Scale Integration interessant, da zudem die Hopfield-Gardner Netzwerke hochgradig fehlertolerant sind [Hendrich 91].

`NN8_CHIP` dient in diesem Zusammenhang als Prototyp, der mit geringem Aufwand erlaubt, die Korrektheit der Architektur und die Brauchbarkeit der implementierten Algorithmen zu testen.

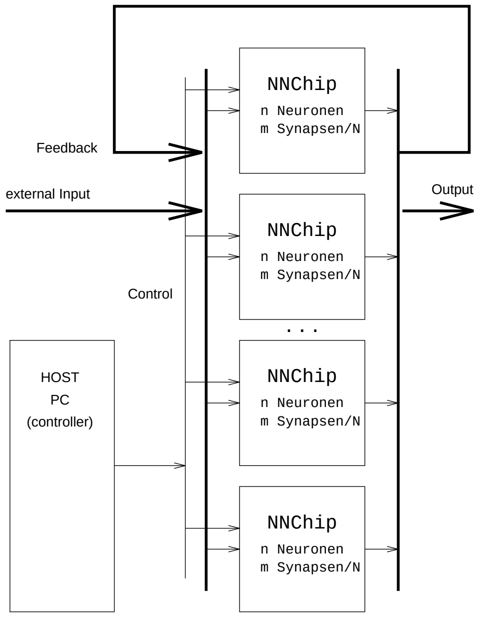

Abbildung 1: Architektur des Netzwerks

Wie in Abbildung 1 gezeigt, besteht das vorgeschlagene Hopfield-Gardner Netzwerk aus einer Anzahl von `NN8_CHIP`'s, die von einem gemeinsamen Host aus gesteuert werden. Ein- und Ausgabebusse des Netzwerks skalieren mit der Anzahl der `NN8_CHIP`'s im Netzwerk. Zu erkennende Muster werden über den `external Input` Bus geladen und vom Netzwerk vervollständigt und über `Output` ausgegeben.

2.3 Hauptblöcke

Jeder `NN8_CHIP` enthält mehrere Neuronen (`DP_1, ..., DP_nn`), einen gemeinsamen Controller (`controller`) für diese Neuronen und einige Multiplexer (`SRegIO`), die für die Auswahl der benötigten globalen Ein/Ausgabe-Signale sorgen, siehe Abbildung 2.

Während das RAM für die Speicherung der Synapsen J_{ij} konzeptionell ebenfalls zum

NN8_CHIP gehört, ist es im hier vorgestellten Prototypen extern realisiert. Statt des on-Chip RAMS ist die Verwendung von externen Standard-SRAMs vorgesehen. NN8_CHIP implementiert die nötige Interfacelogik und im lokalen Controller das entsprechende Timing zur Ansteuerung der SRAMs.

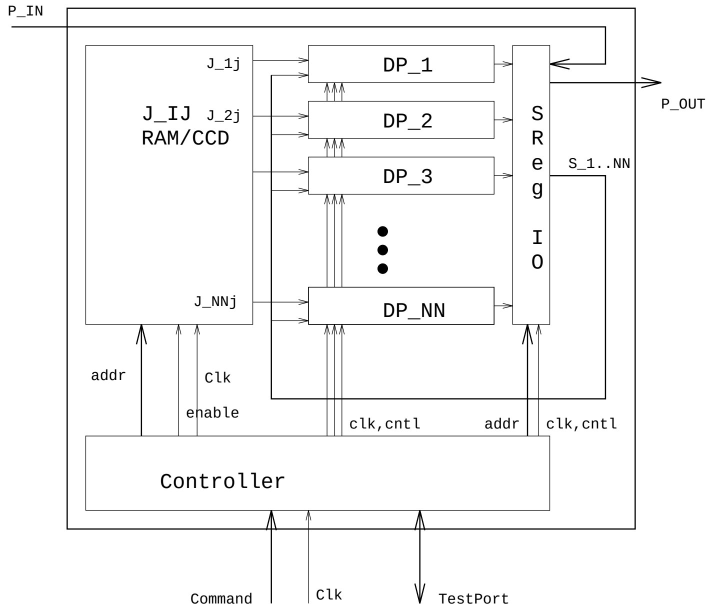

Abbildung 2: Hauptblöcke

2.4 Optimierungen

Abbildung 3 zeigt den Aufbau eines Neurons. Für die Dynamik und die Hebb-Lernregel dient ein (in NN8_CHIP auf 16 Bit) ausgelegtes Register mit Incrementer/Decrementer zur Speicherung und Summation des lokalen Feldes ($H_i = \sum_{j=0}^{N} J_{ij} S_j(t)$) sowie ein 1 Bit Register zur Speicherung des aktuellen Neuronenzustands S_i. Der Wert von S_i wird aus dem (invertierten) Vorzeichen von H_i gebildet. Die Bitbreite von 16 Bit für das H_i Register erlaubt die Verwendung von NN8_CHIP auch für sehr große Netzwerke, die im Prototypen nur durch die Beschränkung auf 16 Bit Adressen für die J_{ij} (d.h. 65536 Synapsen/Neuron) begrenzt wird.

Zur Realisierung der iterativen Lernregel werden zusätzlich zwei Schieberegister zur Speicherung von E_p und E_m (den durch den Lernalgorithmus zu minimierenden Kostenfunktionen) und eine serielle ALU benötigt.

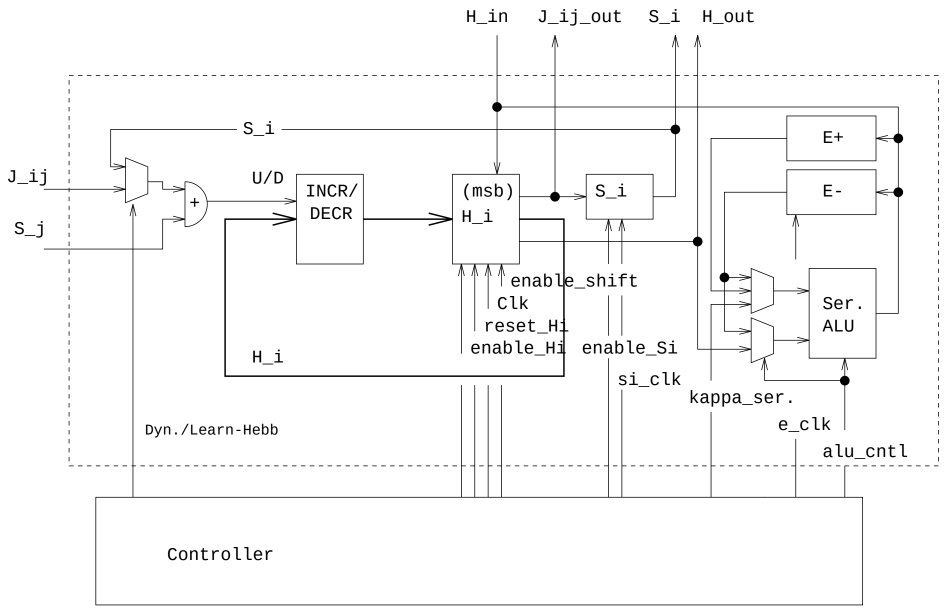

Abbildung 3: Datenpfad (Neuron)

Die Multiplikationen $\xi_i^\mu \cdot \xi_j^\mu$ (d.h. $S_i \cdot S_j$) und $J_{ij} \cdot S_j$ arbeiten auf 1 Bit Signalen und können daher in der Hardwarerealisierung direkt auf XOR Gatter abgebildet werden.

Die zur Steuerung der einzelnen Neuronen (Datenpfade) benötigten Signale sind global für alle Neuronen und werden zentral vom lokalen Controller erzeugt.

3 Entwurfsvorgehen

`NN8_CHIP` ist der erste an der Universität Hamburg entstandene Chipentwurf, der durchgängig in VHDL spezifiziert und mit Testumgebungen simuliert wurde. Deshalb erscheint es sinnvoll, hier auch kurz auf den Entwurfsablauf einzugehen.

3.1 Spezifikation in VHDL

`NN8_CHIP` wurde von Anfang an vollständig in VHDL beschrieben. Da Teile des Datenpfades, wie im vorigen Kapitel erläutert, bereits weitgehend entworfen und optimiert waren, wurden diese in einer Mischung aus Verhaltens- und Strukturbeschreibung spezifiziert. Für den Controller und die Neuron-Ein/Ausgabelogik wurden dagegen zunächst nur Verhaltensmodelle erstellt.

Die Möglichkeiten von VHDL wurden dabei in hohem Maße ausgenutzt. Alle Verhaltensmodelle und die meisten Strukturmodelle sind über `generic()` Parameter in weiten Grenzen parametrisierbar. Für alle Modelle wurden Testumgebungen erstellt, in denen zuerst die Verhaltens- und nach der Logiksynthese mit dem *Synopsys Design Compiler* auch die Strukturmodelle ausgetestet wurden.

3.2 Systemsimulation

Nach dem Entwurf und Test der einzelnen Blöcke von `NN8_CHIP`, (Datenpfade, Controller, Ein/Ausgabe) wurden diese zu dem, ebenfalls über `generic`'s parametrisierbaren, Gesamtmodell zusammengefaßt.

Für die Simulation des Gesamtsystems ist die — bei herkömmlichen Simulatoren unumgängliche — Benutzung der 'Waveforms', also der Darstellung der Ausgangssignale gegen die Simulationszeit, sehr unübersichtlich. Zum Beispiel verfügt ein vergleichbarer Chip mit 64 Neuronen über drei 64-Bit Ein- und Ausgabebusse, die in der Simulation der Lernregeln mit zufälligen Mustern belegt werden. Hier ist die Auswertung der normalen Simulationsergebnisse fast aussichtslos.

Durch die vom *Synopsys VSS* gebotene Möglichkeit, den VHDL Code durch eigene Programme in der internen C-Shell ähnlichen Programmiersprache zu ergänzen und insbesondere formatierte Textausgaben zu produzieren, konnte die Simulation weitaus effizienter gestaltet werden. Abbildung 4 zeigt die laufende Simulation, ein unübersichtliches Waveform Fenster und die durch die eigenen Zusatzprogramme erzeugten Darstellungen des Eingabemusters (im Beispiel ein verrauschter Buchstabe 'V') und des vom Netz produzierten Ausgabemusters (im Beispiel das korrekte 'V').

Ein weiterer Vorteil des *Synopsys VSS* Simulators ist die Möglichkeit der interaktiven Simulation. Die Simulation kann jederzeit unterbrochen werden, um die Werte von Signalen oder

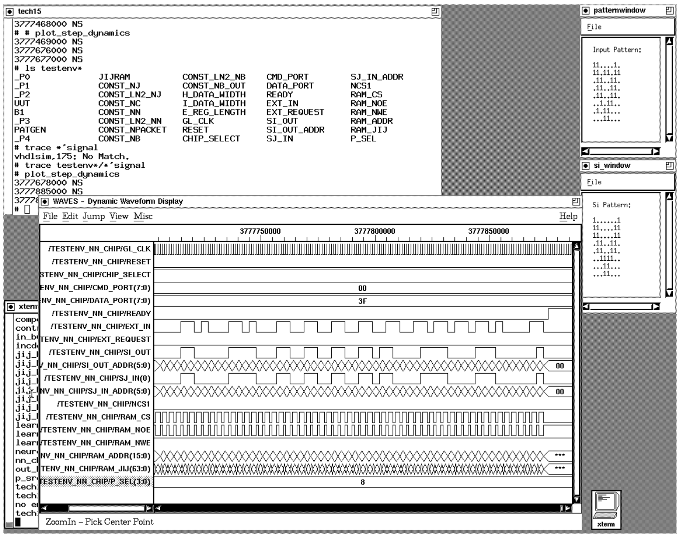

Abbildung 4: Systemsimulationsumgebung (Muster 'V' erkennen)

Variablen auszulesen oder interaktiv zu setzen. Dies ist insbesondere bei der Fehlersuche in noch nicht vollständig korrekten Modellen ein gewaltiger Vorteil gegenüber den üblichen kompilierenden Simulatoren. Der Nachteil der geringeren Simulationsgeschwindigkeit muß dafür in Kauf genommen werden. Die neue Version 3.1 des *Synopsys VSS* Simulators bietet deshalb die Auswahl zwischen interaktivem (langsamen) und kompiliertem (schnellen) Modus.

3.3 Synthesegerechte Beschreibung

Zur Logiksynthese aus den VHDL Verhaltensmodellen wurde der *Synopsys Design Compiler* eingesetzt. Obwohl die Erfahrungen mit der Logiksynthese selbst als sehr positiv einzustufen sind, traten beim Entwurf zunächst Probleme mit der Synthese der VHDL Verhaltensmodelle auf.

Zum einen kann die Logiksynthese im *Synopsys Design Compiler* nur für einen Subset des vollen VHDL Sprachumfangs, wie er im Simulator implementiert ist, eingesetzt werden. Dies betraf aber nur wenige Modelle, da die Architekturbeschreibung zum größten Teil schon auf RT Ebene vorlag.

Zum anderen waren, auf Grund unserer mangelnden Erfahrungen mit den Synthesewerkzeugen, die VHDL Verhaltensmodelle für die Synthese nicht optimal geeignet. Dies führte teilweise zu relativ ineffizienten Gatterrealisierungen. Dem konnte aber durch Umschreiben der Modelle in 'synthesegerechtes' VHDL abgeholfen werden. Da die Probleme immer an denselben Stellen auftraten (zum Beispiel legt der *Synopsys Design Compiler* für Signale, die nur in `IF` Anweisungen in einem `process` zugewiesen werden, automatisch Latches an), konnten die Beschreibungen sehr schnell optimiert und synthesegerecht gestaltet werden.

Nach diesen Modifikationen lieferte die Logiksynthese in allen Fällen sehr gute Gatterrealisierungen und erstaunlich gut lesbare Schematics. Auch die automatische Überprüfung der Fanout-Regeln bei der Logiksynthese (bei automatisch oder von Hand ausgewähltem Wire-load Modell) erscheint sehr wirksam.

3.4 Layout

Für die Layoutgenerierung aus der mit *Synopsys Design Compiler* erstellten Netzliste wurde *Cadence OPUS* eingesetzt. Das entstandene Layout in Abbildung 5 zeugt von der Effizienz der *Cadence OPUS*-Layoutalgorithmen. Das zur Fertigung eingereichte Layout ist padbestimmt, weil für das Bonden bei der Fertigung durch EUROCHIP ein Mindestabstand der Padflecken einzuhalten ist. Die Chipgröße beträgt $3.62 \cdot 3.62\text{mm}^2$.

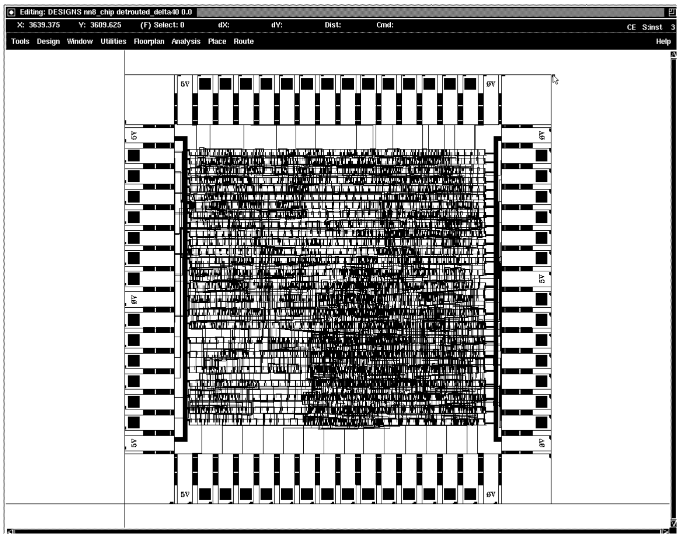

Abbildung 5: Chiplayout

Leider unterstützt der ES2 Cadence Design Kit noch nicht VHDL, sondern nur den Verilog-XL Simulator, so daß für die Postlayoutsimulationen doch die VHDL Umgebung verlassen werden mußte. Zwei exemplarische Simulationsergebnisse sind im Anhang C zur Dokumentation des Verhaltens von `NN8_CHIP` gezeigt.

4 Befehlssatz und Beschreibung

Dieser Abschnitt beschreibt vollständig den in NN8_CHIP implementierten Befehlssatz. Zu jedem Befehl gibt es eine textuelle Kurzbeschreibung sowie die binäre Codierung (cmd_port(7:0)) und die Timinganforderungen. Danach folgen in Abschnitt 4.3 drei Beispiele, wie die Algorithmen des neuronalen Netzes mit Hilfe der einzelnen Befehle programmiert werden.

4.1 Allgemeines

NN8_CHIP erwartet einen Einphasentakt am Eingang gl_clk. Alle internen Register des internen Controllers werden mit der steigenden Flanke von gl_clk getaktet, ebenso einige Register der Datenpfade (Neuronen). Um Hazards zu vermeiden, werden die abgeleiteten Takte für die Neuronen im Controller über Flipflops gepuffert, die mit der fallenden Flanke von gl_clk aktiviert werden.

Der über einen High-Pegel an reset ausgelöste Reset des Controllers ist synchron. NN8_CHIP benötigt wenigstens eine steigende und eine fallende Flanke von gl_clk während reset='1', um sicher einen Reset auszuführen. Dieser synchrone Reset von NN8_CHIP sollte nicht mit dem Befehl reset (s.u.) verwechselt werden, der zusätzlich die internen I- und J- Register im Controller zurücksetzt.

Die Selektion über den Eingang chip_select (aktiv '1') wird nur ausgewertet, wenn NN8_CHIP sich im Zustand idle befindet. Das heißt, NN8_CHIP führt einen aktiven Befehl noch zu Ende aus, bevor chip_select beachtet wird.

4.2 Befehlssatz

4.2.1 Reset

Codierung: $^{7}\boxed{\text{1000 0000}}^{0}$

Beschreibung: Synchroner Reset des Controllers von NN8_CHIP auf der steigenden Flanke von gl_clk (1 Takt).

4.2.2 Idle

Codierung: $^{7}\boxed{\text{0000 0000}}^{0}$

Beschreibung: NOP Befehl (Wartezustand).

NN8_CHIP setzt ready auf '1' und geht in den Zustand idle_state. Dort wartet NN8_CHIP, bis ein anderer Befehl auf cmd_port gelegt wird(1 Takt).

4.2.3 Load Pattern

Codierung: $^{7}\boxed{\texttt{0000 0010}}^{0}$

Beschreibung: Laden eines externen Musters vom Eingang ext_in.

NN8_CHIP setzt ext_request auf '1' (1 Takt) und beginnt eine Schleife, um const_NN Neuronenzustände vom Eingang ext_in einzulesen (jeder Durchlauf dauert 3 Takte). Danach geht NN8_CHIP wieder in den Zustand idle.

```
gl_clk      __11__11__11__11__11__11__11__11__11__11__11__11__11__11__1
cmd_port    xxxxx02020202.............................................
                  ^
                  |lp 1   1   1   2   2   2   3   3   3   4   4   4  5
ext_request ______11111111111111111111111111111111111111 ....
ext_in      xxxxxxxxx11111111111122222222222233333333333344444444444455
```

4.2.4 Step Dynamics

Codierung: $^{7}\boxed{\texttt{0000 0100}}^{0}$

Beschreibung: Ein voller Schritt der Netzwerk Dynamik: Hi-Register löschen, lokales Feld aufsummieren, Ausgangswert Si setzen.

NN8_CHIP löscht die H_i-Register aller on-Chip Neuronen (1 Takt) und beginnt dann mit einer Schleife zur Summation der lokalen Felder. Bei jedem Schleifendurchlauf wird das RAM angesprochen, um die Werte der Synapsen ($J_{(i=0..NN-1),j}$) zu lesen. Außerdem wird der aktuelle Wert von S_j am Eingang Sj_in erwartet (ein Takt Vorbereitung, jeder Durchlauf dauert 3 Takte, insgesamt NJ (Anzahl der Synapsen pro Neuron) Durchläufe). Schließlich setzt NN8_CHIP den neuen Ausgabewert aller Neuronen (1 Takt) und geht in den Zustand idle.

```
Timing:

gl_clk    ____1111____1111____1111____1111____1111____1111
state    idle sum_loc genaddr gencs   sumHi   genaddr gencs.. idle

RAMaddr  xxxxxxxxxxxxx000000000000000000000000111111111111111
RAM_cs   xxxxxxxxxxxxx________1111111111111111________11111111
RAM_nOE  xxxxx1111111111111111111111111111111111111111111111

SREGaddr xxxxxxxxxxxxx0000000000000000000000001111111111111111
SREG_cs  xxxxxxxxxxxxx11111111________________11111111___________

Hi_clk_int    _________________________11111111_______________
Hi_clk   _________________________________11111111____________
```

4.2.5 Set Si

Codierung: ^{7} $\boxed{\texttt{0000 0101}}$ ^{0}

Beschreibung: Berechnung der Ausgangsfunktion, setzt die Werte aller Neuronen.

Alle Si-Register werden gemäß dem Wert in den Hi-Registern gesetzt ($Si =$'1', wenn $Hi > 0$), (1 Takt).

4.2.6 Sum Local Field

Codierung: ^{7} $\boxed{\texttt{0000 0110}}$ ^{0}

Beschreibung: Summation der lokalen Felder in allen Neuronen.

`NN8_CHIP` durchläuft eine Schleife zur Summation der lokalen Felder. Bei jedem Schleifendurchlauf wird das RAM angesprochen, um die Werte der Synapsen ($J_{(i=0..NN-1),j}$) zu lesen. Außerdem wird der aktuelle Wert von S_j am Eingang `Sj_in` erwartet (ein Takt Vorbereitung, jeder Durchlauf dauert 3 Takte, insgesamt `NJ` (Anzahl der Synapsen pro Neuron) Durchläufe).

Zum Timing siehe die Beschreibung von `Step Dynamics`.

4.2.7 Step Hi

Codierung: $^{7}\boxed{\texttt{0000 0111}}^{0}$

Beschreibung: Ein Incr/Decr Schritt für alle Hi-Register.

NN8_CHIP erzeugt einen Takt für die internen Hi-Register, so daß ein Incr/Decr Schritt ausgeführt wird (1 Takt). Dieser Befehl ist nur sinnvoll zum Chiptest, oder wenn gleichzeitig Werte für Sj_in und parallel alle Jij von außen gesetzt werden können. (NN8_CHIP spricht bei diesem Befehl jedenfalls nicht das externe RAM an).

4.2.8 Clear Hi

Codierung: $^{7}\boxed{\texttt{0000 1000}}^{0}$

Beschreibung: Alle Hi-Register in NN8_CHIP werden auf Null gesetzt (1 Takt).

4.2.9 Clear J register

Codierung: $^{7}\boxed{\texttt{0000 1001}}^{0}$

Beschreibung: Das J Register im Controller wird gelöscht
(1 Takt).

4.2.10 Load J register low byte

Codierung: $^{7}\boxed{\texttt{0000 1010}}^{0}$

Beschreibung: Laden der Bits 7..0 des J-Registers vom Datenbus data_port(7..0)
(1 Takt).

4.2.11 Load J register high byte

Codierung: $^{7}\boxed{\texttt{0000 1011}}^{0}$

Beschreibung: Laden der Bits 15..8 des J-Registers vom Datenbus data_port(7..0)
(1 Takt).

4.2.12 Clear Kappa

Codierung: ^{7} `0000 1100` ^{0}

Beschreibung: Kappa-Register löschen.

Das interne Schieberegister für die gewünschte Sollstabilität wird gelöscht — durch Einschieben von entsprechend vielen Nullen (32 Takte).

4.2.13 Load Kappa (1 bit)

Codierung: ^{7} `0000 1101` ^{0}

Beschreibung: Ein Bit in das Kappa-Register einschieben.

Im ersten Takt nach `load kappa` liest `NN8_CHIP` ein Bit vom Datenbus `data_port(0)` in das Kappa-Register hinein (1 Takt) und wartet anschließend 7 Takte, um die Synchronisation zu erleichtern.

Um einen gegebenen Wert in das Kappa-Register einzulesen, ist die Befehlsfolge `clear kappa`, 16 * `load kappa` erforderlich. Das LSB muss dabei zuerst, das MSB zuletzt angelegt werden.

4.2.14 Clear E_p

Codierung: ^{7} `0000 1110` ^{0}

Beschreibung: Löscht die E_p Register an allen Neuronen ($1 + 2 * 16 = 33$ Takte).

Seiteneffekt: Um die E_p Register takten zu können, muß zunächst Theta aktiviert (und daher modifiziert) werden.

4.2.15 Clear E_m

Codierung: ^{7} `0000 1111` ^{0}

Beschreibung: Löscht die E_m Register an allen Neuronen ($1 + 2 * 16 = 33$ Takte).

Seiteneffekt: Um die E_m Register takten zu können, muß zunächst Theta aktiviert (und daher modifiziert) werden.

4.2.16 Step Hebb

Codierung: ^{7} `0001 0000` ^{0}

Beschreibung: Ein Schritt der Hebb Lernregel (4 Takte).

Vorher muß das zu lernende Muster in die S_i geladen werden (mit `load pattern`). Dann wird für alle Neuronen $H_i := H_i + S_i * S_j$ aufsummiert, wobei der j-Index aus dem j-Register entnommen wird.

4.2.17 Set Jij

Codierung: ^{7} `0001 0001` ^{0}

Beschreibung: Werte für alle Synapsen $J_{(0..NN-1)}, j$ in das externe RAM schreiben. Der jeweilige Wert wird aus den S_i entnommen (3 Takte).

4.2.18 Calc Ep sub Em

Codierung: ^{7} `0001 0011` ^{0}

Beschreibung: Berechne $E_{p,i} := E_{p,i} - E_{m,i}$ an allen Neuronen $(1 + 2 * 16 = 33$ Takte).

Evtl. vorher mit `set theta` die E Register aktivieren.

4.2.19 Calc Kappa sub Hi

Codierung: ^{7} `0001 0100` ^{0}

Beschreibung: Berechne $H_i := \kappa - H_i$ an allen Neuronen $(1 + 2 * 16 = 33$ Takte).

4.2.20 Calc Ep plus Hi

Codierung: ^{7} `0001 0101` ^{0}

Beschreibung: Berechne $E_{p,i} := H_i + E_{p,i}$ an allen Neuronen $(1 + 2 * 16 = 33$ Takte).

4.2.21 Calc Em plus Hi

Codierung: ^{7} `0001 0110` ^{0}

Beschreibung: Berechne $E_{m,i} := H_i + E_{m,i}$ an allen Neuronen $(1 + 2 * 16 = 33$ Takte).

4.2.22 Calc Kappa plus Hi

Codierung: $^{7}\boxed{\texttt{0001 0111}}^{0}$

Beschreibung: Berechne $H_i := \kappa + H_i$ an allen Neuronen $(1 + 2 * 16 = 33$ Takte).

4.2.23 Set Theta

Codierung: $^{7}\boxed{\texttt{0001 1000}}^{0}$

Beschreibung: Alle Θ Register an allen Neuronen setzen (1 Takt).

Dies ist vor allen Operationen mit den E-Registern notwendig, um die durch `eval theta` evtl. gesetzten Taktausblendungen wieder zu aktivieren.

4.2.24 Eval Theta

Codierung: $^{7}\boxed{\texttt{0001 1001}}^{0}$

Beschreibung: An allen Neuronen die Θ Register setzen, wenn $\kappa > H_i$ (1 Takt).

4.2.25 Sum Invert Sj

Codierung: $^{7}\boxed{\texttt{0001 1100}}^{0}$

Beschreibung: Vorbereitung zur Berechnung von $E-$ nach der Berechnung von $E+$ während der iterativen Lernregel (6 Takte):

1. Invertieren von S_{jj} (jj ist der Index der gerade zu lernenden Synapse) 2. Zweimal Aufsummieren von $J_{i,jj} * S_{jj}$ auf H_i, dies entspricht gerade dem Effekt der invertieren Kopplung. Nach `sum invert Sj` muß noch zum Aufsummierren von κ und H_i ein `calc kappa plus Hi` durchgeführt werden.

4.2.26 Step E learning

Codierung: $^{7}\boxed{\texttt{0001 1101}}^{0}$

Beschreibung: Summation von $H_i := H_i + J_{i,j} * S_j * S_i$ für die iterative Lernregel.

`step_e_learning` entspricht ansonsten (auch in Bezug auf das Timing) `sum_local_field` mit gesetztem `iter_learn` Bit.

4.2.27 Conditional Load Jij

Codierung: ^{7} `0001 1110` ^{0}

Beschreibung: Bedingtes Laden von $J_{i,j}$ für die iterative Lernregel (3 Takte).

Wenn $E_m < E_p$ soll die aktuelle Synapse ($J_{i,jj}$) invertiert werden. Der Wert von $E_p - E_m$ steht nach `calc Ep sum Em` im E_p Register zur Verfügung. Abhängig vom Vorzeichen von E_p wird $J_{i,jj}$ direkt oder invertiert nach S_i geladen und von dort mittels `set Jij` wieder ins RAM geschrieben.

4.2.28 Load NJ register low

Codierung: ^{7} `0010 0000` ^{0}

Beschreibung: Laden der Bits `(7..0)` des NJ (Anzahl Synapsen/Neuron) Registers vom Datenbus `data_port(7..0)` (1 Takt).

4.2.29 Load NJ register high

Codierung: ^{7} `0010 0001` ^{0}

Beschreibung: Laden der Bits `(15..8)` des NJ (Anzahl Synapsen/Neuron) Registers vom Datenbus `data_port(7..0)` (1 Takt).

4.2.30 Read J register low

Codierung: ^{7} `0010 0010` ^{0}

Beschreibung: Ausgabe des aktuellen Wert des J-Registers Bits `(7..0)` auf den Datenbus `data_port(7..0)` (1 Takt).

4.2.31 Read J register high

Codierung: ^{7} `0010 0011` ^{0}

Beschreibung: Ausgabe des aktuellen Wert des J-Registers Bits `(15..8)` auf den Datenbus `data_port(7..0)` (1 Takt).

4.3 Beispiele für Befehlssequenzen

Die folgenden Unterkapitel enthalten exemplarische Befehlssequenzen in Pseudocode für die Netzwerkdynamik und die von `NN8_CHIP` unterstützten Lernregeln.

4.3.1 Dynamik

In der Arbeitsphase des Hopfield-Gardner Netzes werden die zu erkennenden Muster zunächst geladen, und dann wird die Dynamik des Netzes (ein oder mehrere Schritte) gestartet:

```
load_pattern            -- load an external pattern
step_dynamics           -- one step of the dynamics
                        -- break off here, or :
step_dynamics           -- additional steps of the dynamics
...
step_dynamics           -- until convergence or cycle reached
```

4.3.2 Hebb Lernregel

Die Hebb-Lernregel wird parallel an allen Neuronen des `NN8_CHIP` ausgeführt, so daß nur eine äußere Schleife über alle Synapsenindizes j benötigt wird. In der inneren Schleife müssen alle zu lernenden Muster einmal geladen werden:

```
for all synapses ( j = 0; j < NJ; j++) begin
   -- load j into j register
   load_j_reg_low( j.low )
   load_j_reg_high( j.high )
   clear_Hi
   for all patterns( p = 0; p < maxP; p++) begin
      load_pattern( p )
      step_hebb              -- sum up Hi
   end
   set_si                    -- copy the not(MSB(Hi)) to Si
   set_jij                   -- set the new (Hebb) value for Jij
end
```

4.3.3 Iterative Lernregel

Der Pseudocode für die iterative Lernregel ist schon etwas aufwendiger. Eine äußere Schleife steuert lediglich die Anzahl der Lerniterationen. In der mittleren Schleife wird der Index der gerade zu lernenen (einzustellenden) Synapse gesetzt — im Beispiel nach einem einfachen round-robin Verfahren durch die for-Schleife. Dort werden die Werte von E_p und E_m zunächst auf Null gesetzt.

In der inneren Schleife werden nacheinander alle Muster geladen und die Energien E_p (mit dem Wert der gerade zu lernenden Synapse) und E_m (mit dem invertierten Wert der aktiven Synapse) berechnet. Nur die Anteile der lokalen Felder, die kleiner als κ sind, gehen dabei in die Summation ein. Nach einem Durchlauf durch die innere Schleife wird dann geprüft, ob E_p größer als E_m ist, und entsprechend die zu lernende Synapse eingestellt.

```
for ( iter = 0; iter < max_iter; iter++) begin
   for all synapses ( j = 0; j < NJ; j++) begin
      load_j_reg_low( j.low )       -- load j index into the net
      load_j_reg_high( j.high )
      set_theta                     -- enable, then clear E+/- registers
      clear_e_p
      clear_e_m
      for all patterns( p = 0; p < maxP; p++) begin
         load_pattern( p )
         -- calculate E+
         clear_Hi
         step_e_learning
         calc_kappa_sub_Hi
         eval_theta
         calc_e_p_plus_Hi
         -- now calculate E- (inverted S_i,j)
         clear_Hi
         step_e_learning
         sum_invert_sj
         calc_kappa_sub_Hi
         eval_theta
         calc_e_m_plus_Hi
      end
      -- now test (E+ > E-) and set synapse accordingly
      set_theta
      calc_e_p_sub_e_m
      cond_load_jij
      set_jij
   end
end
```

A Applikationsschaltung

Die folgenden Schaltbilder skizzieren eine minimale Applikationsschaltung mit nur einem NN8_CHIP, einem Standard SRAM (etwa 64K*8 Bit) und einem Hostsystem (zum Beispiel PC). Dies ist bereits ein vollständiges System, mit dem nicht nur die Funktion von NN8_CHIP verifiziert werden kann. So ist zum Beispiel die Simulation der iterativen Lernregel auch für Netzwerke möglich, die aus Zeitgründen kaum auf einer Workstation bzw. einem PC simuliert werden könnten.

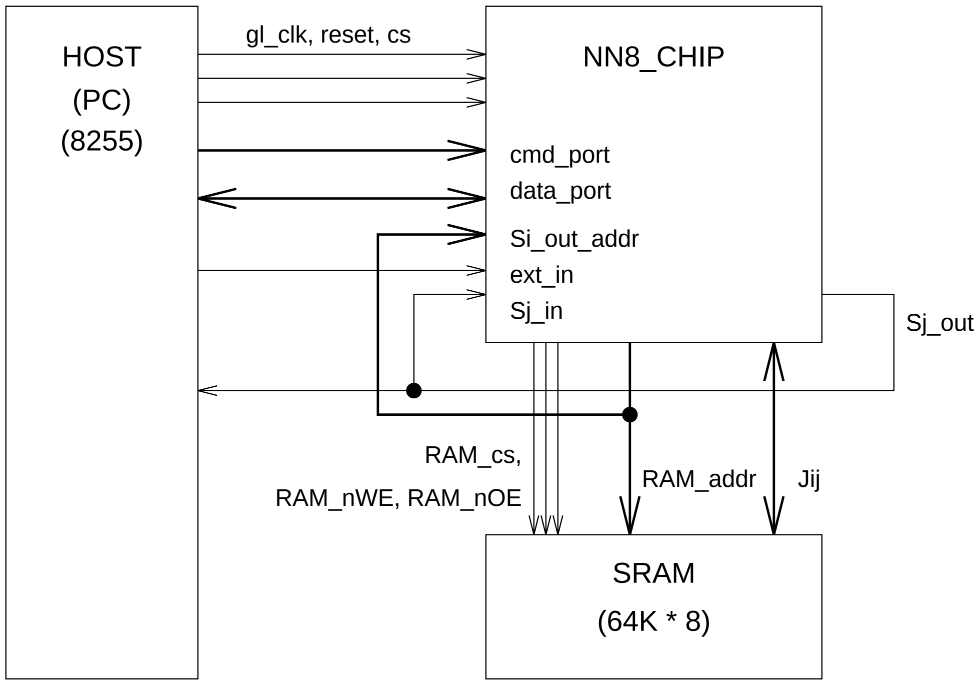

Abbildung 6: Applikationsschaltung

B Pin-Assignment, 'nn8_chip', CC68 package

```
         60         44
      61 ++++++++++++++ 43
         +            +
         +            +
         +            +
      68 +            +
       1 +            +
         +            +
         +            +
       9 +            + 27
         ++++++++++++++
         10         26

Pin Name Typ   Funktion
(upper left)
68 |U112 inout Jij<5>
67 |U111 inout Jij<4>
66 |U110 inout Jij<3>
65 |U109 inout Jij<2>
64 |U108 inout Jij<1>
63 |U107 inout Jij<0>
62 |I4 PWR(core)
61 --- ---

(top)
60 |I2 PWR
59 |U118 output RAM_addr<0>
58 |U119 output RAM_addr<1>
57 |U120 output RAM_addr<2>
56 |U121 output RAM_addr<3>
55 |U122 output RAM_addr<4>
54 |U123 output RAM_addr<5>
53 |U124 output RAM_addr<6>
52 |U125 output RAM_addr<7>
51 |U126 output RAM_addr<8>
50 |U127 output RAM_addr<9>
49 |U128 output RAM_addr<10>
48 |U129 output RAM_addr<11>
47 |U130 output RAM_addr<12>
46 |U131 output RAM_addr<13>
45 |I7 PWR
44 --- ---

(right)
```

```
43 |I10 GND (core)
42 |U137 output  Si_out
41 |U150 input chip_select
40 |U117 output  RAM_nOE
39 |U116 output  RAM_nWE
38 |U115  output  RAM_cs
37 |I_controller|U1853 inout data_port<7>
36 |I1 PWR
35 |I_controller|U1852 inout data_port<6>
34 |I_controller|U1851 inout data_port<5>
33 |I_controller|U1850 inout data_port<4>
32 |I_controller|U1849 inout data_port<3>
31 |I_controller|U1848 inout data_port<2>
30 |I_controller|U1847 inout data_port<1>
29 |I_controller|U1846 inout data_port<0>
28 |I9 GND (core)
27 --- ---

(bottom)
26 --- ---
25 |I8 GND
24 |U151 input reset
23 |U152 input gl_clk
22 |U138 input Sj_in
21 |U140 input ext_in
20 |U141 output ready
19 |U139 output ext_request
18 |U149 input cmd_port<7>
17 |U148 input cmd_port<6>
16 |U147 input cmd_port<5>
15 |U146 input cmd_port<4>
14 |U145 input cmd_port<3>
13 |U144 input cmd_port<2>
12 |U143 input cmd_port<1>
11 |U142 input cmd_port<0>
10 |I3 GND

(lower left)
9 |I5 PWR (core)
8 |U132 output RAM_addr<14>
7 |U133 output RAM_addr<15>
6 |U136 input Si_out_addr<2>
5 |U135 input Si_out_addr<1>
4 |U134 input Si_out_addr<0>
3 |U114 inout Jij<7>
2 |I6 GND
1 |U113 inout Jij<6>
```

C Teststimuli für Chiptest

Die folgende STL-Datei enthält die für die Postlayoutsimulationen für NN8_CHIP benutzten Teststimuli.

```
; ---STL Source Program Template---
stlinit
; note that clock inputs should be of type "clk"
defpin Si_out_addr<2:0> in
defpin Sj_in      in
defpin chip_select in
defpin cmd_port<7:0> in
defpin ext_in     in
defpin gl_clk     clk
defpin reset      in
defpin RAM_addr<15:0> out
defpin RAM_cs     out
defpin RAM_nOE    out
defpin RAM_nWE    out
defpin Si_out     out
defpin ext_request out
defpin ready      out
defpin Jij<7:0>   io
defpin data_port<7:0> io

; deftiming 1e-10 1e-09 1e-08  ; define basic time units
; define any clocks or timing (defclock, defstrobe)
deftiming 1e-10 1e-08 1e-07  ; 10 MHz Takt auf gl_clk
defclock "....11111." gl_clk
; define a basic format statement consisting of all the
; input and bidirectional nodes
defformat Si_out_addr Sj_in chip_select cmd_port ext_in reset \
          Jij data_port
; if you wish to include both input and output nodes in the
; vectors, uncomment the following defformat statement
; defformat Si_out_addr Sj_in chip_select cmd_port ext_in gl_clk \
;           reset Jij data_port RAM_addr RAM_cs RAM_nOE RAM_nWE Si_out \
;           ext_request ready
deftest
; define simulation stimulus values using xv command
; use binary truth table data, in the order of the
; defpin commands entered above.
; The following is an instance of a stimulus vector
;
;
;         Si_out_addr  chip_sel    ext_in      Jij in/out
```

```
;          |     Sj_in  |   cmd_port|   reset   |         data_port in/out
;          |     |      |   |       |   |       |         |
;
;
xv         0b000 0      1   0x01    0   0         Z X    0x33  X
rptv 10    0b000 0      1   0x01    0   0         Z X    0x33  X
;                                            --reset:
rptv 10    0b000 0      1   0x01    0   1         Z X    0x33  X
;                                            --cmd_reset:
rptv  2    0b000 0      1   0x80    0   0         Z X    0x33  X
;                                            --idle:
rptv  2    0b000 0      1   0x00    0   0         Z X    0x33  X
;                                            --clear_hi:
rptv  6    0b000 0      1   0x08    0   0         Z X    0x33  X
;                                            --set_Si:
rptv  6    0b000 0      1   0x05    0   0         Z X    0x33  X
;                                            --NJ laden
;                                            --NJ=64: 64-1 laden:
;                                            --63 = 0x00 0x3f
rptv  2    0b000 0      1   0x20    0   0         Z X    0x3f  X
rptv 20    0b000 0      1   0x00    0   0         Z X    0x00  X  ;20 idle Zyklen
rptv  2    0b000 0      1   0x21    0   0         Z X    0x00  X
rptv 20    0b000 0      1   0x00    0   0         Z X    0x00  X  ;20 idle Zyklen
;                                            --und wieder auslesen,
;                                            --dazu data_port
;                                            --mit 'Z' belegen
rptv 10    0b000 0      1   0x22    0    0        Z X    Z    0xee
rptv 20    0b000 0      1   0x00    0    0        Z X    Z    0xee
rptv 10    0b000 0      1   0x23    0    0        Z X    Z    0xee
rptv 20    0b000 0      1   0x00    0    0        Z X    Z    0xee
;                                            --clear_kappa
rptv  2    0b001 0      1   0x0C    0    0        Z X    Z     X
rptv 40    0b000 0      1   0x00    0    0        Z X    0x00  X ;40 idle Zyklen
;                                            --load_kappa
;                                            --(1+7 Zyklen/Bit)
;                                            --Wert 15: 4* 0x01 laden,
rptv  2    0b000 0      1   0x0D    0    0        Z X    0x01  X ;
;                                            --1 Bit laden (2 Zyklen)
rptv  6    0b000 0      1   0x00    0    0        Z X    0x00  X ;6 idle Zyklen
rptv  2    0b000 0      1   0x0D    0    0        Z X    0x01  X
rptv  6    0b000 0      1   0x00    0    0        Z X    0x00  X ;6 idle Zyklen
rptv  2    0b000 0      1   0x0D    0    0        Z X    0x01  X
rptv  6    0b000 0      1   0x00    0    0        Z X    0x00  X ;6 idle Zyklen
rptv  2    0b000 0      1   0x0D    0    0        Z X    0x01  X
rptv  6    0b000 0      1   0x00    0    0        Z X    0x00  X ;6 idle Zyklen
;
;                                            --clear_e_p, clear_e_m
```

```
rptv  2   0b010 0      1   0x0E    0    0      Z X    Z     X
rptv 40   0b000 0      1   0x00    0    0      Z X    0x00  X
rptv  2   0b010 0      1   0x0F    0    0      Z X    Z     X
rptv 40   0b000 0      1   0x00    0    0      Z X    0x00  X
;                                           --calc_hi_sub_kappa
;                                           --(nach clear_hi und load_kappa
;                                           -- muss Hi jetzt negativ werden)
rptv  2   0b111 0      1   0x14    0    0      Z X    Z     X
rptv 40   0b111 0      1   0x00    0    0      Z X    Z     X
;
;                                           --set Si (Hi negativ -> Si pos).
rptv  2   0b000 0      1   0x05    0    0      Z X    Z     X
rptv 20   0b000 0      1   0x00    0    0      Z X    Z     X
;
;                                           --sum_local_field,alle Si,JIj pos.
;
rptv  2   0b000 0      1   0x06    1    0      0xFF X Z     X
rptv 200  0b000 0      1   0x00    1    0      0xFF X Z     X
;
;                                           --set Si   (Hi positiv -> Si pos).
rptv  2   0b000 0      1   0x05    0    0      Z X    Z     X
rptv 20   0b000 0      1   0x00    0    0      Z X    Z     X
;
;                                           --load_pattern
rptv  2   0b111 0      1   0x02    1    0      Z X    Z     X
rptv 50   0b111 0      1   0x00    1    0      Z X    Z     X
endtest
```

Zur Dokumentation des Verhaltens von NN8_CHIP folgen jetzt noch zwei Abbildungen mit den Ergebnissen der Verilogsimulationen mit der obigen Eingabedatei:

Abbildung 7: Simulationsergebnis (Ausschnitt)

Abbildung 8: Simulationsergebnis

Literatur

[CDS 92] Cadence Design Systems Inc., *Design Framework II Release 4.2.1a Reference Manual* (1992)

[ES2 93] European Silicon Structures Ltd., *ES2 Cadence Design Kit Reference Manual* (1993)

[Hendrich 91] Norman Hendrich, *Associative memory in damaged neural networks*, Journal of Physics **A** 24, 2877 (1991)

[Hendrich 94] Norman Hendrich, *A minimal neural network architecture*, Universität Hamburg, (1994), in preparation

[Hopfield 82] J. J. Hopfield, Neural networks and physical systems with emergent collective computational abilities, *Proc. Nat. Acad. Sci.* 79, 2554–2558 (1982)

[Mäder 93] Andreas Mäder, *VHDL Kurzbeschreibung*, Universität Hamburg, (1993)

[Synopsys VSS 93] Synopsys Inc., *VHDL System Simulator Release 3.0b Reference Manual* (1993)

[Synopsys DC 93] Synopsys Inc., *Design Compiler Release 3.0b Reference Manual* (1993)