1 Einleitung und Motivation

Der vorliegenden Diplomarbeit liegt als Grundgedanke ein Gesamtsystem zugrunde, mit dem von einer Videokamera gelieferte Bilder in Echtzeit mit Hilfe eines neuronalen Netzwerkes klassifiziert werden sollen. Als konkrete Klassifikationsaufgabe wurde dabei die Streckenerkennung von Eisenbahnschienen vorgesehen, d. h. aus einem fahrenden Schienenfahrzeug heraus soll sowohl die Richtung des Streckenverlaufs (links, rechts, geradeaus) als auch der Zustand (frei, besetzt) der sichtbaren Strecke erkannt werden. Neben dieser Klassifikationsaufgabe sind jedoch auch viele andere Einsatzmöglichkeiten, wie z. B. die Erkennung von Gesichtern oder Verkehrszeichen denkbar.

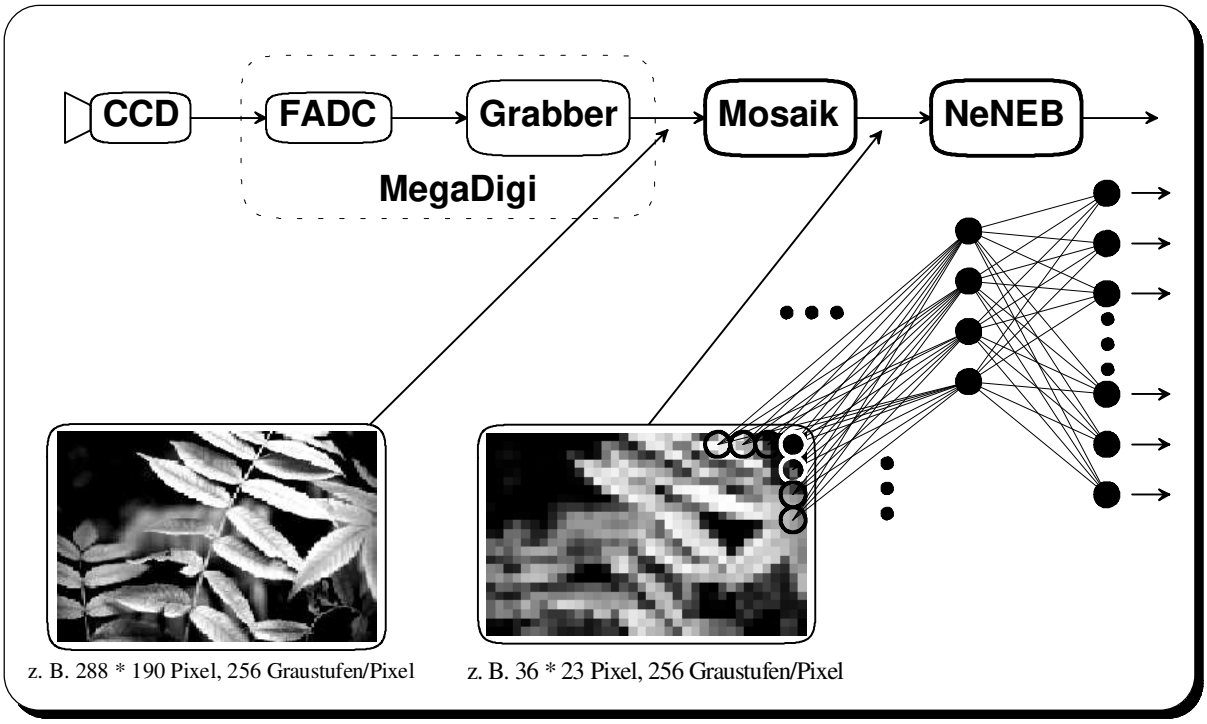


Bild 1.1: Gesamtsystem zur Echtzeit-Klassifikation von Bilddaten

Das in Bild 1.1 dargestellte Gesamtsystem besteht aus vier Teilsystemen, von denen im Rahmen dieser Diplomarbeit die beiden letzten Systeme zur Bildvorverarbeitung und Bildklassifikation entwickelt wurden:

- **Kamera**

  Hierfür wird eine handelsübliche Miniatur-*CCD*[1]-Schwarz/Weiß-Kamera verwendet, die Graustufenbilder liefert.

[1] *CCD - engl. Charge-Coupled-Device* ≈ ladungsgekoppeltes Bauelement/Schaltung

- **System zur Digitalisierung der Bilddaten (*MegaDigi*)**

  Die von der Kamera gelieferten Graustufenbilder werden mit Hilfe eines *FADC*[1] digitalisiert. Aus dem digitalen Bilddatenstrom werden dann von einem *Grabber*[2] zusammenhängende Bilder extrahiert und bildweise gespeichert bzw. an die nachfolgenden Komponenten weitergeleitet. Der *Grabber* wurde parallel im Rahmen einer Studienarbeit [Hah95] entwickelt.

- **Bildvorverarbeitung (Auflösungsreduktion, *Mosaik*)**

  Hier werden die vom Digitalisiersystem (MegaDigi) gelieferten seriellen Bilddaten durch blockweise Zusammenfassung von Pixeln reduziert, wodurch sich eine Auflösungsreduktion der von der Kamera aufgenommenen Bilder ergibt. Der Mosaik-Chip wurde im Rahmen der vorliegenden Diplomarbeit entwickelt und ist ausführlich in Kapitel 4 beschrieben.

- **Bildklassifikation (*NeNEB*)**

  In diesem Teil des Gesamtsystems erfolgt eine Klassifikation der von der Kamera aufgenommenen Bilder nach vorgegebenen Kriterien. Die Klassifikation erfolgt dabei mit Hilfe eines künstlichen neuronalen Backpropagation-Netzwerkes. Dessen Hardwarerealisierung als eigenständigen *VLSI*[3]-Chip (*NeNEB - Neuronales Netzwerk zur Echtzeitklassifikation von Bildern*) bildet den Schwerpunkt der vorliegenden Diplomarbeit. Dieser Chip soll aus dem ankommenden seriellen Bilddatenstrom die einzelnen Bilder in Echtzeit klassifizieren.

  Jedes Bildpixel (bis maximal 65536 Pixel) stellt einen Eingang des neuronalen Netzwerkes dar und ist mit jedem Neuron der ersten Schicht (vgl. Bild 1.1) verbunden. Für jede dieser Verbindungen (maximal 524288) benötigt das neuronale Netzwerk eine Kopplungsstärke, mit der der Grauwert eines Pixels gewichtet, d. h. multipliziert, wird. Anschließend werden alle gewichteten Pixelwerte für die Erregung eines Neurons aufsummiert, wodurch bei 50 Bildern pro Sekunde mit 65536 Pixeln ca. 26,2 Millionen Multiplikationen und Additionen pro Sekunde erforderlich sind. Aufgrund dieses hohen Rechenaufwandes in Echtzeit wurde nur die Ausführungsphase des neuronalen Netzwerkes (siehe Kapitel 2) implementiert, da der Lernalgorithmus den Rechenaufwand in etwa verdreifacht hätte. Das Netzwerk ist frei programmierbar, so daß es für verschiedene Aufgaben trainiert werden kann, wobei das Training auf einer Workstation anhand ausgewählter charakteristischer Bilder erfolgt.

Eine weitere Motivation für die vorliegende Diplomarbeit wurde durch das Projekt *ALVINN* (siehe Kapitel 3) gegeben. Hierbei handelt es sich um ein Fahrzeug, daß selbständig einem beliebigen Straßenverlauf folgt. Die Steuerung erfolgt dabei über ein Backpropagation-Netzwerk, daß als Eingabe ausschließlich Bildinformationen von einer Kamera erhält. Die Berechnung des neuronalen Netzwerkes erfolgt hierbei mit Hilfe dreier Sun Sparc-Workstations. Das in Bild 1.1 dargestellte Gesamtsystem hingegen stellt eine kompakte Hardwarerealisierung eines Bildklassifikationssystems dar, das mit wenigen Komponenten auskommt. Denkbar ist dabei auch eine Zusammenfassung von Grabber, Bildvorverarbeitung und Bildklassifikation in einem einzigen Chip, der dann direkt in eine Kamera implementiert werden kann.

---

[1] *FADC - engl. Flash-Analog-Digital-Converter* ≈ paralleler Analog/Digital-Konverter

[2] *engl. Grabber* ≈ Greifer, Sammler

[3] *VLSI* - engl. *Very Large Scale Integration* ≈ sehr hohe Integrationsdichte

2 Neuronale Netzwerke

Künstliche neuronale Netzwerke gewinnen in einigen Bereichen der Informatik immer mehr an Bedeutung, seit im Jahre 1943 von den Amerikanern Warren McCulloch und Walter Pitts das erste Modell eines künstlichen neuronalen Netzwerkes entworfen wurde. Seither ist eine Vielzahl Netzmodelle und damit verbundener Lernalgorithmen entworfen worden, die teilweise problemspezifisch sind. Die Architektur künstlicher neuronaler Netzwerke ist dabei eng an das biologische Vorbild, das Gehirn, angelehnt. So haben künstliche neuronale Netzwerke auch zum Verständnis des Gehirns beigetragen.

2.1 Biologisches Vorbild

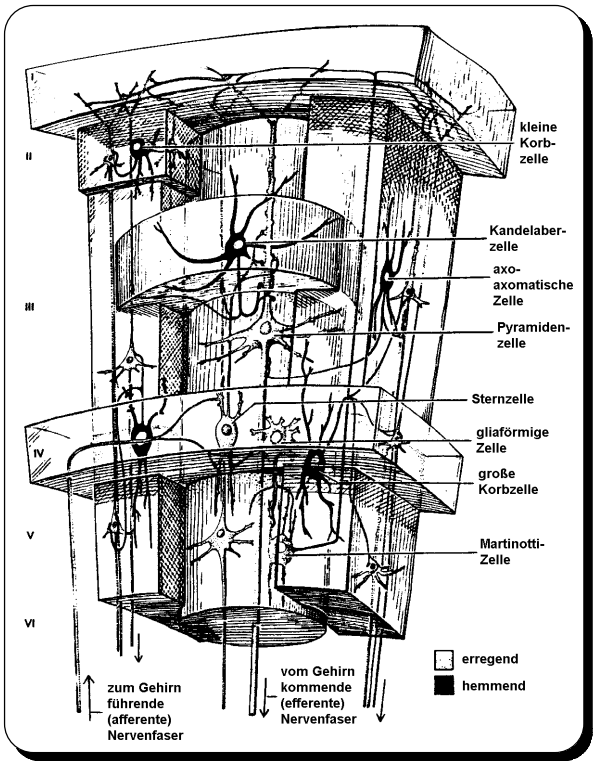


Bild 2.1: Schnitt durch den menschlichen Kortex [Roj93]

Das menschliche Gehirn besteht aus rund 10^{10} bis 10^{11} Nervenzellen, den sogenannten Neuronen. Jede Nervenzelle ist dabei mit etwa 10^4 anderen Nervenzellen vernetzt, wodurch sich 10^{14} bis 10^{15} mögliche Verbindungen ergeben. Innerhalb der Großhirnrinde, dem Kortex, lassen sich verschiedene Neuronentypen ausmachen (Bild 2.1). Obwohl die Form der einzelnen Neuronen

verschieden ist, weist der Kortex eine geschichtete hierarchische Struktur auf. Beim menschlichen Kortex werden dabei sechs Schichten unterschieden, wobei jede Schicht für bestimmte Aufgaben zuständig ist.

2.1.1 Aufbau der Nervenzellen

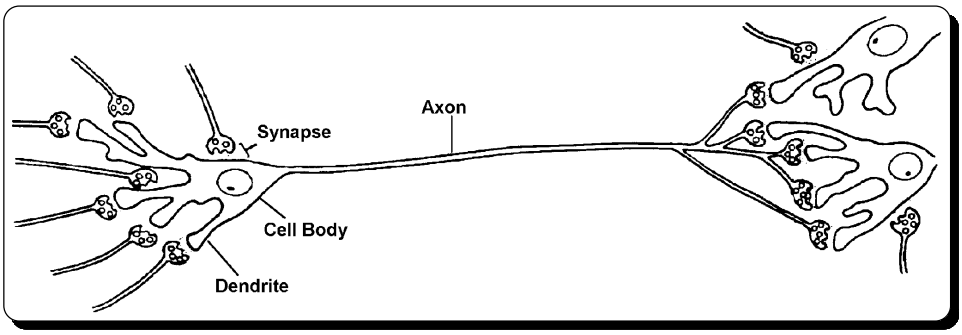


Bild 2.2: Aufbau eines Neurons und Verbindung zu anderen Neuronen [Rum86]

Neuronen sind die elementaren, informationsverarbeitenden Funktionselemente eines neuronalen Netzwerkes. Jedes Neuron besteht im wesentlichen aus vier Elementen (Bild 2.2):

- **Dendriten**

  Die Dendriten sind die fein verzweigten Ausläufer des Neurons. Sie stellen die Eingänge des Neurons dar. Typisch sind etwa 10^4 bis 10^5 Dendriten. An den Dendriten wird die Information von anderen Neuronen über die Synapsen entgegengenommen.

- **Zellkörper (Cell Body)**

  Der Zellkörper ist das eigentliche Berechnungselement des Neurons. Hier werden die Eingangssignale von den Dendriten aufsummiert. Überschreitet die Summe eine bestimmte Schwelle (*Reizschwelle*), so liefert das Neuron einen kurzzeitigen Ausgangsimpuls, das sogenannte *Aktionspotential*.

- **Axon**

  Das Axon ist die Ausgangsleitung des Neurons. Jedes Neuron besitzt maximal ein Axon; es gibt auch Neuronen ohne Axon, deren einzige Aufgabe in der Verknüpfung anderer Neuronen besteht. Über das Axon werden die Information eines Neurons an andere Neuronen weitergeleitet. Hierzu verzweigt sich das Axon in viele kleine Endstücke, an deren Enden die Synapsen liegen.

- **Synapsen**

  Die Synapsen sind die stark verzweigten Endstücke der Axone. Sie stehen über den *synaptischen Spalt* mit den Dendriten in Verbindung. Hierbei wird zwischen erregenden (*exzitatorischen*) und hemmenden (*inhibitatorischen*) Synapsen unterschieden. Eine weitere Aufgabe der Synapsen ist die Gewichtung der von dem Axon gelieferten Information.

Diese vier Elemente bilden die minimal für ein Neuron notwendige Struktur, mit denen künstliche neuronale Netzwerke modelliert werden können, wenngleich biologische Nervenzellen um

einiges komplexer sind. An dieser Stelle sei vor allem an die Energieversorgung der Zellen erinnert, die jedoch für künstliche neuronale Netzwerke ohne Bedeutung ist.

2.1.2 Funktionsweise der Nervenzellen

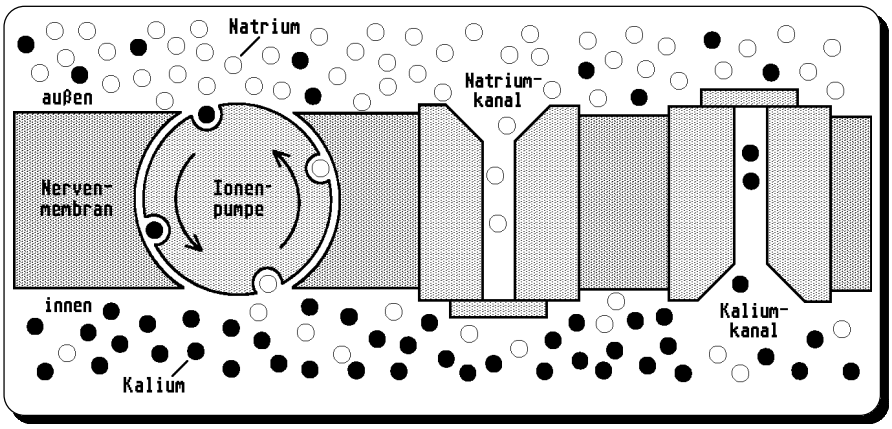


Bild 2.3: Zellmembran mit Ionenpumpe und Ionenkanälen [Gun80]

Das Innere einer Nervenzelle, wie auch ihre Umgebung, enthält hauptsächlich positive Natriumionen (Na^+), positive Kaliumionen (Ka^+) und negative Chlorionen (Cl^-). Durch sogenannte Ionenkanäle in der Zellmembran (Bild 2.3) können dabei Kaliumionen aus der Zelle und Natriumionen in die Zelle diffundieren. Diese beiden getrennten Kanäle sind im Ruhezustand der Zelle geschlossen. Für den Potentialausgleich innerhalb der Nervenzelle sorgt eine Ionenpumpe, die Natriumionen in und Kaliumionen aus der Zelle transportiert. Sie sorgt dafür, daß das Zellinnere stets ein Ruhepotential von -70 mV besitzt.

Eine Erregung der Zelle entsteht durch positive Ionen, die in die Zelle eindringen. Diese Depolarisation der Zelle bewirkt ein Ansteigen des Potentials in der Zelle auf etwa +40 mV. Die dadurch entstehende Depolarisationswelle, das sogenannte Aktionspotential (Bild 2.4), stellt das Ausgabesignal einer Nervenzelle dar, das Neuron feuert. Bevor das Ruhepotential wiederhergestellt ist, gibt es jedoch ein *overshooting*, wobei das Potential kurzzeitig auf unter -80 mV sinkt. Dieses Aktionspotential wird auf elektrochemischem Wege über das Axon an andere Nervenzellen weitergeleitet.

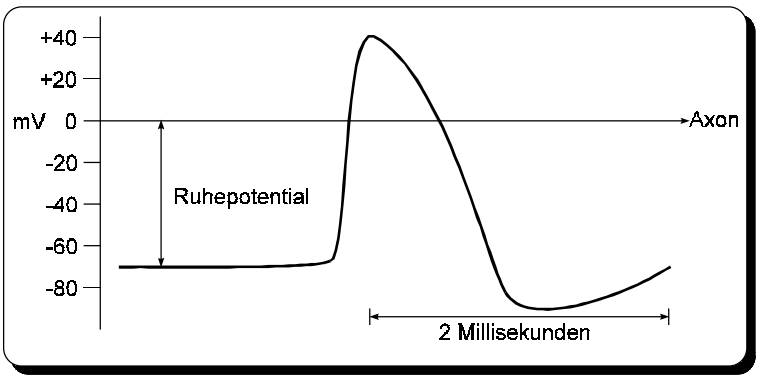


Bild 2.4: Form des Aktionspotentials entlang des Axons

Die Form des Aktionspotentials, wie auch seine Amplitude, ist bei allen Neuronen gleich. Nervenzellen verwenden zur Charakterisierung ihrer Aktivität eine Frequenzmodulation[1] anstatt einer Amplitudenmodulation[2]. Diese Art der Übertragung ist notwendig, da bei längeren Leitungswegen durchaus Amplitudenverluste auftreten können, die im Falle einer Amplitudenmodulation zu einer Verzerrung der Information führen würden. Eine stark erregte Zelle gibt deshalb innerhalb einer Zeiteinheit sehr viele Impulse ab, während eine schwach erregte Zelle innerhalb der gleichen Zeiteinheit sehr wenige Impulse abgibt. Die Impulsfrequenzen von Nervenzellen liegen im Bereich 0,1..1 kHz, wobei die minimale Zeit zwischen zwei Impulsen durch die sogenannte *Refraktärzeit* des Neurons bestimmt wird. Dies ist die Zeit, die die Zelle benötigt, um nach einer Depolarisation wieder ihr Ruhepotential von -70 mV herzustellen.

Zur Übergabe eines Impulses vom Axon zum Dendriten der Zielzelle dienen die Synapsen. Diese erscheinen als Verdickung am Ende eines Axonteilstrangs. Zwischen der Synapse und dem Dendriten der Zielzelle befindet sich der synaptische Spalt, der normalerweise geöffnet ist. Die Übertragung des elektrischen Nervensignals erfolgt auf chemischem Wege über sogenannte *Neurotransmitter*. Diese Transmitter sind in sogenannten *synaptischen Bläschen (Vesikel)*, die sich in der Synapse befinden, enthalten. Ein an den Synapsen ankommender elektrischer Impuls sorgt dafür, daß die Vesikel mit der Zellmembran der Zielzelle verschmelzen und dabei ihre Transmitterstoffe an diese abgeben. Die Transmitterstoffe wirken dabei auf die Ionenkanäle in der Zellmembran. Hierdurch werden diese geöffnet, und es können entweder positive (Erregung) oder negative (Hemmung) Ionen in die Zelle eindringen. Dabei sorgen die Synapsen auch für eine Gewichtung der Nervensignale, indem einige Synapsen mehr und andere weniger Transmitterstoffe freisetzen. Hierdurch wird die Erregung der Zielzelle bzw. die Öffnungswahrscheinlichkeit ihrer Ionenkanäle gesteuert.

2.2 Künstliche neuronale Netze

Basierend auf dem Vorbild biologischer neuronaler Netzwerke sollen künstliche neuronale Netzwerke ein vergleichbares Verhalten aufweisen. Wesentlich dabei ist, daß das Netzwerk eine Art *Black-Box* darstellt, die auf der einen Seite eine Eingabe erhält und auf der anderen Seite mit einer Ausgabe reagiert. Der genaue Aufbau des Netzwerkes, d. h. seine Struktur, ist zunächst unerheblich. So gesehen führt ein neuronales Netzwerk die Berechnung einer Abbildung $A: R^n \rightarrow R^m$ durch (Bild 2.5). Die Art, wie die Abbildung A berechnet wird, hängt jedoch wesentlich von der internen Struktur, der Topologie, des neuronalen Netzwerkes ab.

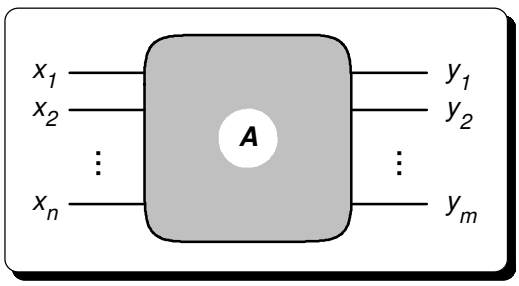


Bild 2.5: Neuronales Netzwerk als *Black-Box* [Roj93]

[1] *Frequenzmodulation* - Informationsgehalt liegt in der Häufigkeit, d. h. der Frequenz des Auftretens, eines Signals
[2] *Amplitudenmodulation* - Informationsgehalt liegt in der Höhe der Amplitude eines Signals

2.2.1 Netztopologien

Für die Netztopologie ist es von entscheidender Bedeutung, wie die Abbildung $A:R^n \rightarrow R^m$ berechnet wird. Eine von der Eingabe unabhängige Abbildung erhält man bei nicht rückgekoppelten Netzen (*feed forward*). Hierbei wird die Art der Abbildung nur durch die Vernetzung des Netzwerkes und seiner synaptischen Kopplungsstärken festgelegt. Enthält das Netzwerk dagegen Zyklen, d. h. das Ergebnis eines Neurons wird ganz oder teilweise wieder als Eingabe zu diesem zurückgeführt, so spricht man von einem rückgekoppelten Netzwerk (*feed back*). Bei diesen ist die Ausgabe sowohl von der Struktur des Netzwerkes mit seinen synaptischen Kopplungsstärken abhängig als auch von der Eingabe, d. h. es erfolgt eine *rekursive* Berechnung.

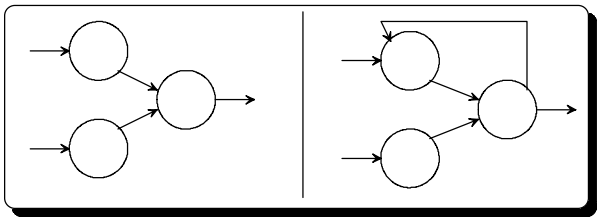


Bild 2.6: nicht rückgekoppelte (links) und rückgekoppelte (rechts) Netze

Ein weiteres Klassifikationsmerkmal neuronaler Netzwerke besteht in der Art der Vernetzung. Auch ohne Rückkopplungen muß ein Netzwerk nicht immer, wie in Bild 2.6 dargestellt, nur Verbindungen von einer zur nächsten Ebene aufweisen. Es können durchaus Neuronen der ersten Ebene mit Neuronen der vierten Ebene verbunden sein. Ebenso können Rückkopplungen sich über mehrere Ebenen erstrecken. Von einer *Schichtenarchitektur* spricht man dagegen, wenn, wie in Bild 2.7 zu sehen, Neuronen einer niedrigeren Ebene nur mit Neuronen der nächsthöheren Ebene verbunden sind. Eine Schichtenarchitektur enthält neben der *Eingabeschicht* und der *Ausgabeschicht* noch *n verdeckte* oder *verborgene Schichten (hidden layer)*. Hierbei sei angemerkt, daß nur die verborgenen Schichten Neuronen enthalten, wohingegen die Ein- und die Ausgabeschicht nur als Ein-/Ausgabeeinheiten fungieren [Roj93].

Im Gegensatz zu dem in Bild 2.7 gezeigten *vollverknüpften* Netzwerk, d. h. jedes Neuron einer Schicht ist mit jedem Neuron der nächsten Schicht verbunden, gibt es auch noch *teilverknüpfte* Netzwerke, d. h. es sind nicht alle Neuronen von benachbarten Schichten miteinander verknüpft.

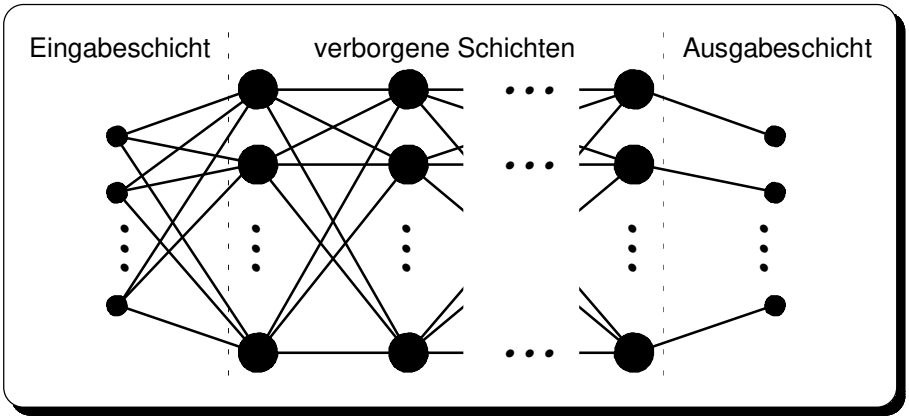


Bild 2.7: Neuronales Netzwerk in Schichtenarchitektur

2.2.2 Das Modell von McCulloch und Pitts

Eines der einfachsten Modelle für künstliche neuronale Netzwerke wurde im Jahre 1943 von den amerikanischen Forschern Warren McCulloch und Walter Pitts vorgeschlagen, die Neuronen in einem McCulloch-Pitts-Netz heißen daher auch McCulloch-Pitts-Zellen. In diesen Netzen werden ausschließlich binäre Signale verwendet. Sie zählen zu den nicht gewichteten Netzen, d. h. jede Eingabeleitung hat denselben Stellenwert in der Gesamterregung. Jedoch wird zwischen zwei verschiedenen Arten von Eingabeleitungen unterschieden:

- *erregende Eingabeleitungen* x_i
- *hemmende Eingabeleitungen* z_i

Bei den erregenden Eingabeleitungen verhält sich eine McCulloch-Pitts-Zelle wie ein normales Neuron, d. h. die Werte aller n Eingabeleitungen werden aufsummiert und mit einem Schwellwert θ verglichen. Überschreitet die Summe diesen Schwellwert, so gibt das Neuron den Wert 1 aus, andernfalls eine 0.

Die hemmenden Eingabeleitungen hingegen wirken unabhängig von den erregenden Eingabeleitungen. Sofern die McCulloch-Pitts-Zelle hemmende Eingabeleitungen besitzt ($m \geq 1$) und mindestens eine von diesen eine 1 führt, gibt die Zelle eine 0 aus, unabhängig vom Zustand der erregenden Eingabeleitungen. Daher wird auch von *absoluter Hemmung* gesprochen.

Formal läßt sich das Verhalten einer McCulloch-Pitts-Zelle wie folgt beschreiben:

$$y = \begin{cases} 0 \quad \text{wenn} \quad \sum_{i=1}^{n} x_i < \theta \ \vee \left(\sum_{j=1}^{m} z_j = 1 \ \wedge \ m \geq 1 \right) \\ 1 \quad \text{wenn} \quad \sum_{i=1}^{n} x_i \geq \theta \ \wedge \left(\sum_{j=1}^{m} z_j = 0 \ \wedge \ m \geq 1 \right) \end{cases} \tag{2.1}$$

Nach Minsky wird eine McCulloch-Pitts-Zelle als ein Kreis mit weißem Eingabeteil und schwarzem Ausgabeteil dargestellt. Auf diese Weise benötigen die Ein-/Ausgabeleitungen keine Pfeile, da die Richtung der Kanten durch das Symbol eindeutig festgelegt ist. Hemmende Eingabeleitungen werden dabei durch einen Kreis am Ende der Kante symbolisiert.

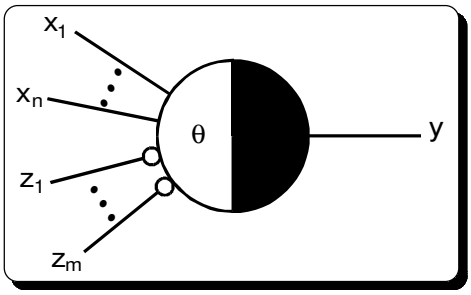


Bild 2.8: Symbol einer McCulloch-Pitts-Zelle

McCulloch-Pitts-Zellen sind funktional eng mit Logikzellen, die Boolesche Funktionen berechnen, verwandt. Dabei können die drei logischen Grundfunktionen *AND*[1], *OR*[2] und *NOT*[3] jeweils mit einer McCulloch-Pitts-Zelle berechnet werden. Da diese drei Grundfunktionen eine

[1] *AND* - Ausgabe 1, wenn alle Eingabeleitungen 1 sind
[2] *OR* - Ausgabe 1, wenn mindestens eine Eingabeleitung 1 ist
[3] *NOT* - Ausgabe stellt das Inverse der Eingabe dar, d. h. 1 wird zu 0 und umgekehrt

funktional vollständige Basismenge [Lag87] der Booleschen Algebra darstellen, kann jede beliebige logische Funktion mit einem zweischichtigen McCulloch-Pitts-Netz berechnet werden. Es kann gezeigt werden, daß eine komplexe Funktion, wie das *Exklusiv-Oder* (*XOR*[1]) nicht mit einer einzigen McCulloch-Pitts-Zelle zu berechnen ist. Dies führt auf das Problem der *linearen Trennbarkeit*, das in Kapitel 2.2.6 zu finden ist.

2.2.3 Perceptron Modell

Aus McCulloch-Pitts-Zellen bestehende neuronale Netzwerke zählen zu den ungewichteten Netzen, da hier jede Eingabeleitung den gleichen Anteil an der Gesamterregung des Neurons hat. Um die einfachen McCulloch-Pitts-Zellen zu erweitern, kann man zunächst einmal die absolut hemmenden Eingabeleitungen in *relativ hemmende* Eingabeleitungen verwandeln. Dies bedeutet, daß eine hemmende Eingabeleitung nicht sofort zur Hemmung des Neurons führt, sondern nur die Erregung um eins reduziert, d. h. bei der Aufsummierung der Werte der Eingabeleitungen erscheinen hemmende Eingabeleitungen mit negativem Vorzeichen.

Eine zweite Erweiterung besteht in der Umwandlung der ungewichteten Eingabeleitungen in gewichtete, wobei sowohl die erregenden als auch die hemmenden Eingabeleitungen gewichtet werden. In einem letzten Schritt kann dann noch die Trennung zwischen hemmenden und erregenden Eingabeleitungen aufgehoben werden. Statt dessen werden positive und negative Gewichte ω verwendet, wobei ein positives Gewicht einer Erregung und ein negatives Gewicht einer Hemmung entspricht. Mit diesen Erweiterungen wird aus der einfachen McCulloch-Pitts-Zelle ein *einfaches Perceptron*.

Dieses läßt sich formal folgendermaßen beschreiben:

$$y = \begin{cases} 0 & \text{wenn} \quad \sum_{i=1}^{n} \omega_i \cdot x_i < \theta \\ 1 & \text{wenn} \quad \sum_{i=1}^{n} \omega_i \cdot x_i \geq \theta \end{cases} \tag{2.2}$$

Obwohl ein einzelnes Perceptron hinsichtlich der berechenbaren Funktionen mächtiger als eine einzelne McCulloch-Pitts-Zelle ist, so sind Netzwerke, die nur aus Perceptronen bestehen, funktional gleichmächtig zu Netzwerken, die nur aus McCulloch-Pitts-Zellen bestehen. Dies bedeutet, daß ein gewichtetes Netzwerk in eine äquivalentes ungewichtetes Netzwerk und umgekehrt überführt werden kann [Roj93].

Das hier skizzierte Modell [Roj93] des Perceptrons stellt eine starke Vereinfachung des von dem amerikanischen Psychologen *Frank Rosenblatt* 1958 aufgestellten [Ros58] und von *Minsky* und *Papert* weiterentwickelten klassischen Perceptron-Modells dar.

2.2.4 Verallgemeinertes Modell

Basierend auf dem biologischen Vorbild führt ein Neuron eines künstlichen neuronalen Netzwerkes die Berechnung zweier Funktionen durch:

- *Integrations- oder Propagierungsfunktion:* Die Integrationsfunktion bewertet die n Eingabesignale mit einem Gewicht ω und bildet aus allen Eingabesignalen eine

[1] *XOR* - Ausgabe 1, wenn Eingabeleitungen ungleiche Werte haben

Gesamterregung des Neurons. Am häufigsten besteht die Integrationsfunktion, in direkter Analogie zum biologischen Vorbild, in einer Addition der gewichteten Eingabesignale. Hinzu kommt noch ein Parameter θ, der den Schwellwert, d. h. die Reizschwelle des Neurons darstellt. Formal läßt sich somit eine Integrationsfunktion *a* definieren, die die Erregung des *i-ten* Neurons berechnet:

$$a_i = \theta_i + \sum_{j=1}^{n} \omega_{ij} \cdot x_j \quad \text{mit} \quad x_i \in [0...1] \quad \text{und} \quad a_i, \theta_i, \omega_{ij} \in \Re \ . \tag{2.3}$$

Um bei Lernalgorithmen für den Schwellwert θ nicht eine eigene Lernregel aufstellen zu müssen, wird dieser häufig in ein Gewicht ω_{i0} transformiert, wobei

$$\omega_{i0} = -\theta \tag{2.4}$$

gilt. Mit diesem Gewicht (*bias*[1]) wird ein zusätzlicher Eingang x_0 belegt, der einen konstanten Eingangswert von 1 aufweist. Hierdurch geht Gleichung 2.3 über in

$$a_i = \sum_{j=0}^{n} \omega_{ij} \cdot x_j \ . \tag{2.5}$$

- *Aktivierungs- oder Transferfunktion:* Die Aktivierungsfunktion bewertet die Erregung des Neurons und entscheidet, welche Ausgabe eine bestimmte Erregung produzieren soll. Eine einfache Aktivierungsfunktion ist die *Heavysidesche Sprungfunktion s(x)*:

$$s(x) = \begin{cases} 0 & \text{für} \quad x \le 0 \\ 1 & \text{für} \quad x > 0 \end{cases} \ . \tag{2.6}$$

Modifiziert um den Schwellwert θ wird diese von McCulloch-Pitts-Zellen und vom Perceptron verwendet (Bild 2.9)

$$s(x) = \begin{cases} 0 & \text{für} \quad x \le \theta \\ 1 & \text{für} \quad x > \theta \end{cases} \ . \tag{2.7}$$

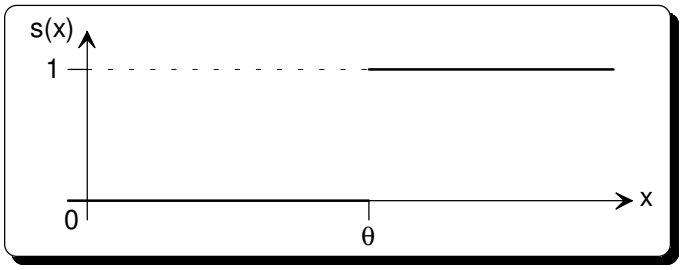


Bild 2.9: Sprungfunktion *s(x)* mit Schwellwert θ

In der Regel kommen jedoch stetige Aktivierungsfunktionen zur Anwendung, da diese dem biologischen Vorbild wesentlich näher kommen. Die Stetigkeit der Aktivierungsfunktion ist dabei von großer Wichtigkeit hinsichtlich der Konvergenz beim Trainieren eines neuronalen Netzwerkes, da hier eine Änderung der Gewichte ω in Abhängigkeit von der Ableitung der Aktivierungsfunktion erfolgt. Am häufigsten wird dabei die Sigmoidfunktion (Bild 2.10)

$$g(x) = \frac{1}{1 + e^{-x/T}} \tag{2.8}$$

[1] *engl. bias* ≈ beeinflussen

benutzt, die die Wahrscheinlichkeit des Feuerns eines Neurons widerspiegelt. Bei Ausgabewerten unterhalb 0.5 wird das Neuron als gehemmt, bei Werten über 0.5 als erregt angesehen. Der Parameter T wird hier als Temperaturkonstante[1] bezeichnet und hat typischerweise den Wert 1. Häufig findet auch die inverse Größe $c=1/T$ Anwendung. Für $T \rightarrow 0$ geht die Sigmoidfunktion in die Heavysidesche Sprungfunktion $s(x)$ über. Eine andere gebräuchliche stetige Aktivierungsfunktion ist der hyperbolische Tangens

$$g(x) = \tanh(x) . \tag{2.9}$$

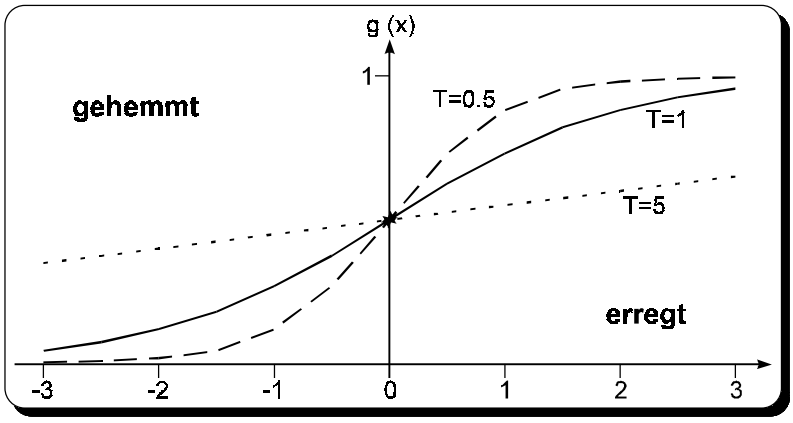


Bild 2.10: Sigmoidfunktion $g(x)$ und Zustand des Neurons

Formal gesehen stellt die Ausgabe y eines Neurons also die Berechnung einer Funktionskompostion

$$y = g(a(x)) \tag{2.10}$$

dar. Je nach verwendetem Netztyp bzw. Lernalgorithmus variiert die Form der Aktivierungsfunktion $g(a)$. Prinzipiell kann dabei jedes Neuron eine andere Aktivierungsfunktion aufweisen, zur Vereinfachung wird in künstlichen neuronalen Netzwerken jedoch meist für jedes Neuron die gleiche Aktivierungsfunktion verwendet. Für spezielle Anwendungen wird gelegentlich auch eine andere Integrationsfunktion $a(x)$ als die Addition verwendet.

2.2.5 Geometrische Veranschaulichung

Nachdem in den drei vorhergehenden Abschnitten einige Modelle für neuronale Netzwerke vorgestellt wurden, soll jetzt ihre Funktionsweise kurz grafisch veranschaulicht werden. Als Beispiel werden dafür zwei logische Grundfunktionen (Tabelle 2.1) mit je zwei Eingängen gewählt.

		AND	**OR**
x_1	x_2	y_1	y_2
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Tabelle 2.1: Funktionstabelle der logischen Funktionen AND und OR

[1] Bei biologischen Neuronen ist eine Abhängigkeit der Aktivierungsfunktion von der Temperatur zu beobachten.

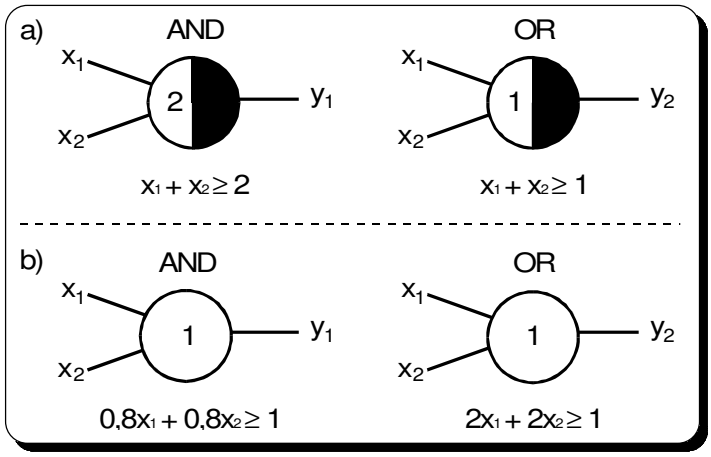


Bild 2.11: Logisches AND und OR mit McCulloch-Pitts-Zellen (a) und Perceptronen (b)

In Bild 2.11 sind die beiden Funktionen jeweils für McCulloch-Pitts-Zellen und für Perceptronen dargestellt. Der Schwellwert der Perceptronen wurde der Einfachheit halber mit 1 angenommen, es sind jedoch auch andere Werte mit entsprechender Veränderung der Gewichte möglich. Durch Einsetzen der vier möglichen Wertkombinationen für x_1 und x_2 lassen sich die Ungleichungen leicht überprüfen. Die Neuronen geben jeweils eine 1 aus, wenn die entsprechende Ungleichung erfüllt ist, andernfalls eine 0. Da hier jeweils zwei Eingangsvariablen verwendet werden, handelt es sich geometrisch gesehen um ein zweidimensionales Problem, d. h. in einem Koordinatensystem lassen sich die vier möglichen Wertkombinationen der Eingangsvariablen veranschaulichen (Bild 2.12).

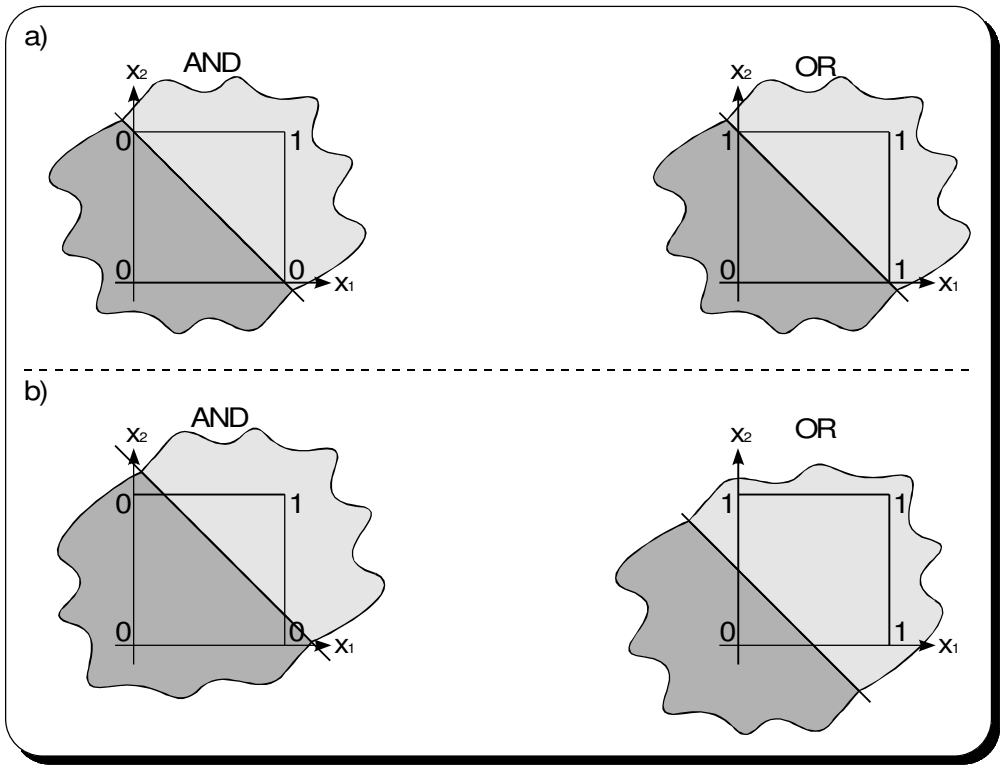


Bild 2.12: Eingaberaumaufteilung bei der AND- und bei der OR-Funktion
a) mit McCulloch-Pitts-Zellen, b) mit Perceptronen

Wie in Bild 2.12 zu sehen ist, teilen die Neuronen die Ebene (*Eingaberaum*) in zwei unterschiedliche Bereiche, d. h. die Eingabemenge wird in zwei disjunkte Mengen geteilt. Der Bereich links unterhalb der Trenngerade (dunkelgrau) enthält alle Punkte, in denen die jeweilige Funktion den Wert Null hat, während der Bereich rechts oberhalb (hellgrau), inklusive der Punkte auf der Trenngeraden, alle Punkte enthält, in denen die jeweilige Funktion den Wert Eins hat. Da eine McCulloch-Pitts-Zelle keine Gewichtung der Eingabesignale vornimmt, verläuft hier die Trenngerade im 45°-Winkel an der Grenze zu den Eins-Punkten. Bei den Perceptronen hingegen ist die Lage der Trenngeraden variabel, auch hinsichtlich des Winkels, in dem die Gerade verläuft. Sie muß nur so gewählt werden, daß alle Eins-Punkte auf der einen Seite der Geraden und alle Null-Punkte auf der anderen Seite der Geraden liegen.

Sofern ein Neuron nun mehr als zwei Eingänge hat, wird die Eingaberaumaufteilung zu einem mehrdimensionalen Problem, d. h. der Raum wird nicht mehr durch Geraden, sondern durch *Hyperebenen* geteilt. Es gibt jedoch nach wie vor zwei Teilräume, wobei der eine Teilraum ein aktives Neuron kennzeichnet, der andere Teilraum ein inaktives Neuron.

Sofern als Aktivierungsfunktion eine stetige Funktion wie die Sigmoidfunktion verwendet wird, ist die Trennung des Eingaberaumes nicht mehr scharf. Statt dessen ist der Übergang von einem Teilraum in den anderen fließend, d. h. die Ausgabewerte umfassen jetzt ein kontinuierliches Intervall zwischen 0 und 1. Hierdurch kann nicht nur zwischen einem aktiven und einem inaktiven Neuron unterschieden werden, sondern es kann auch der Aktivierungsgrad eines Neurons festgestellt werden.

2.2.6 Lineare Trennbarkeit und XOR-Problem

Im vergangenen Abschnitt wurde erläutert, wie ein Neuron mit zwei Eingängen den Eingaberaum in zwei Bereiche aufteilt. Hierfür wurden die *n* Eingänge des Neurons als ein *Eingabevektor* $\vec{x} = (x_1, x_2, ..., x_n)$ aufgefaßt. Jede Wertekombination der Eingänge beschreibt somit einen Punkt im *n*-dimensionalen Eingaberaum. Entsprechend der verwendeten Aktivierungsfunktion *g(x)* wird dieser mehr oder weniger scharf in zwei Teilräume aufgespalten. Die Hyperebene, durch die die beiden Teilräume getrennt werden, wird durch die Gleichung

$$\theta + \sum_{i=1}^{n} \omega_i \cdot x_i = 0 \tag{2.11}$$

beschrieben. Diese erhält man durch Nullsetzen der Integrationsfunktion (Gleichung 2.3).

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Tabelle 2.2: Funktionstabelle der XOR-Funktion

Ob nun ein bestimmtes Problem oder, allgemeiner, eine bestimmte Eingabemenge in zwei Teilmengen derart aufgespalten werden kann, daß die obige Trennung des Eingaberaumes in zwei Teilräume erfolgt, hängt davon ab, ob die Eingabemenge *linear teilbar* bzw. *linear trennbar* ist. An einem Beispiel, dem *XOR-Problem* (siehe Funktionstabelle in Tabelle 2.2), läßt sich die lineare Trennbarkeit verdeutlichen. Die Frage, die sich dabei stellt, ist, ob die XOR-Funktion

mit einem einzigen Perceptron zu berechnen ist oder nicht. Damit die XOR-Funktion von einem einzigen Perceptron mit Schwellwert θ berechnet werden kann, müssen folgende Ungleichungen erfüllt sein:

$$\text{für} \quad x_1 = 0, x_2 = 0 \qquad \omega_1 \cdot x_1 + \omega_2 \cdot x_2 = 0 < \theta \tag{2.12}$$

$$\text{für} \quad x_1 = 0, x_2 = 1 \qquad \omega_1 \cdot x_1 + \omega_2 \cdot x_2 = \omega_2 \geq \theta \tag{2.13}$$

$$\text{für} \quad x_1 = 1, x_2 = 0 \qquad \omega_1 \cdot x_1 + \omega_2 \cdot x_2 = \omega_1 \geq \theta \tag{2.14}$$

$$\text{für} \quad x_1 = 1, x_2 = 1 \qquad \omega_1 \cdot x_1 + \omega_2 \cdot x_2 = \omega_1 + \omega_2 < \theta \; . \tag{2.15}$$

Die vier Ungleichungen 2.12 bis 2.15 führen jedoch zu einem Widerspruch, da einerseits θ nach Gleichung 2.12 positiv ist. Andererseits sind somit nach Gleichung 2.13 und 2.14 auch ω_1 und ω_2 positiv und größer als θ. Andererseits soll die Summe aus ω_1 und ω_2 kleiner als θ sein. Aus diesem Widerspruch ergibt sich, daß ein einzelnes Perceptron die Berechnung der XOR-Funktion nicht durchführen kann. Geometrisch gesehen bedeutet dies, daß es nicht möglich ist, eine einzelne Gerade zu finden, die den Eingaberaum der XOR-Funktion linear teilt (Bild 2.13). Lösbar wird das XOR-Problem erst durch Hinzufügen einer verborgenen Schicht, wodurch das zweidimensionale Problem in ein dreidimensionales Problem übergeht.

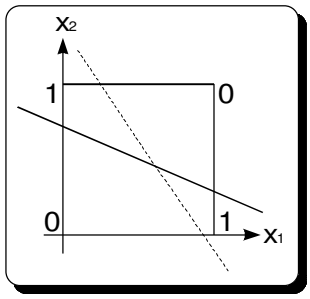


Bild 2.13: Eingaberaum der XOR-Funktion

2.2.7 Lernen in neuronalen Netzen

In Bild 2.5 wurde ein künstliches neuronales Netzwerk als eine Black-Box betrachtet, die eine Abbildung $A: R^n \rightarrow R^m$ berechnet. Das Auffinden dieser Abbildung A wird als *Lernen* oder *Trainieren* des neuronalen Netzwerkes bezeichnet.

Lernalgorithmen für neuronale Netzwerke lassen sich in zwei wichtige Klassen unterteilen:

- *überwachtes Lernen*
- *unüberwachtes Lernen*

Von überwachtem Lernen wird gesprochen, wenn beim Lernen stets das Ergebnis, d. h. die korrekte Ausgabe des Netzwerkes, zu einer bestimmten Menge von Eingaben bekannt ist. Die Gewichte im Netzwerk werden nun, ausgehend von zufälligen Startwerten, solange verändert, bis das Netzwerk mit einer korrekten bzw. weitgehend korrekten Ausgabe auf die Eingabemenge antwortet. Häufig wird dabei keine absolute Identität von Vorgabe und Ausgabe erwartet, sondern nur das Unterschreiten einer festgelegten Abweichung, d. h. es wird ein minimaler Fehler definiert, bei dessen Erreichen oder Unterschreiten der Lernvorgang beendet wird. Daher unterscheidet man auch zwischen *korrigierendem Lernen* und *Verstärkungslernen*. Von korrigierendem Lernen spricht man dabei, wenn die Fehlergröße zwischen Ausgabe und Vorgabe für die Korrektur der Gewichte derart verwendet wird, daß der Fehler in einem Schritt

behoben wird, d. h. die Fehlergröße ist ein Maß für die Veränderung der Gewichte. Hingegen wird beim Verstärkungslernen nur die Korrektheit jedes Ein-/Ausgabepaares überprüft. Das Ergebnis dieser *binären Entscheidung* (*wahr* oder *falsch*) wird für die Korrektur der Gewichte herangezogen, d. h. es gibt kein Maß für die erforderliche Änderung der Gewichte.

Unüberwachtes Lernen liegt hingegen vor, wenn die Ausgabemenge zu einer bestimmten Eingabemenge unbekannt ist. Das Netzwerk ist in einem solchen Fall *selbstorganisierend*, d. h. das Netzwerk muß selbst herausfinden, welches die optimalen Gewichte für eine bestimmte Eingabe sind. Dieser Vorgang der Selbstorganisation findet auch im Gehirn von Lebewesen statt. Bei der Analyse des visuellen Kortex von Katzen wurde herausgefunden, daß Katzen, die bei der Geburt auf einem Auge blind waren, eine andere Verschaltung des visuellen Kortex aufwiesen als Katzen, die auf beiden Augen sehen konnten [Roj93].

2.2.8 Ausführungsphase neuronaler Netzwerke

Das Ziel beim Einsatz neuronaler Netzwerke ist nicht nur die Ermittlung einer Ausgabe für eine bekannte, d. h. zuvor gelernte, Menge von Eingaben (*Eingabemuster*). Vielmehr soll ein trainiertes Netzwerk sowohl auf bekannte Eingabemuster als auch auf unbekannte Eingabemuster, die einem bekannten Muster ähneln, mit einer dazu passenden Ausgabe antworten. Dieser Vorgang wird *Generalisierung* genannt, wobei unter der Ähnlichkeit zweier Muster verstanden wird, daß beide Muster sich in nur wenigen Komponenten geringfügig unterscheiden, d. h. nur wenige Eingangsneuronen bzw. Ausgangsneuronen haben unterschiedliche Werte. Hierin liegt die Stärke und der Vorteil neuronaler Netzwerke gegenüber algorithmischen Lösungen, bei denen der Lösungs*weg* und die Ein- bzw. Ausgabemenge stets fest vorgegeben ist, wohingegen bei neuronalen Netzwerken das Lösungs*ziel* anhand verschiedener Beispiele (*Lernmuster*) vorgegeben wird.

2.3 Backpropagation-Algorithmus

Lange Zeit gab es keinen allgemeingültigen Algorithmus zum Trainieren mehrschichtiger nicht rückgekoppelter neuronaler Netzwerke. Dies änderte sich, als Rummelhart und McCelland 1985 den *Backpropagation (BPG)-Algorithmus* vorstellten [Rum85]. Der Backpropagation-Algorithmus ist ein überwachtes Lernverfahren mit Verstärkung (vgl. Kapitel 2.2.7). Hierbei werden die synaptischen Gewichte in Abhängigkeit von einer Fehlergröße verändert.

2.3.1 Netzmodell

Der Backpropagation-Algorithmus wird zum Trainieren nicht rückgekoppelter Netzwerke in Schichtenarchitektur (vgl. Kap. 2.2.1) verwendet (Bild 2.14). Daher sind jeweils nur Neuronen in benachbarten Schichten verbunden, wobei der Signalfluß nur in einer Richtung von der Eingabeschicht über die verdeckte(n) Schicht(en) zur Ausgabeschicht erfolgt. Das Netzwerk ist voll verbunden, d. h. jedes Neuron der einen Schicht ist mit jedem Neuron der nächsten Schicht verbunden.

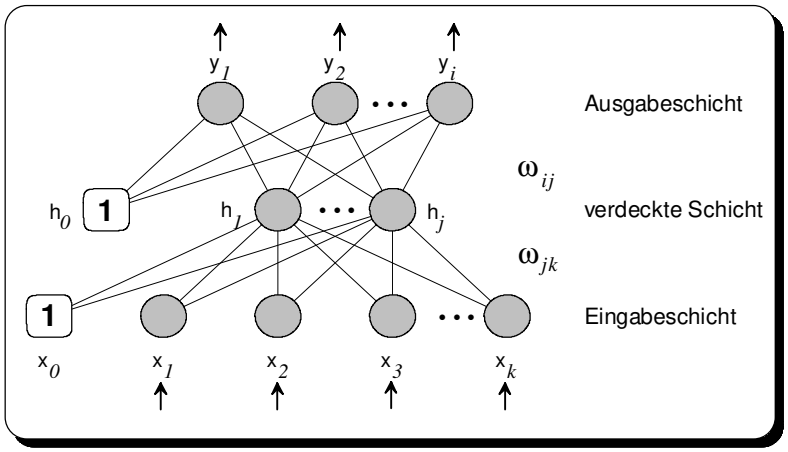


Bild 2.14: BPG-Netz mit einer verdeckten Schicht

Die *k* Eingänge des Netzwerkes können ebenso wie die Ausgaben der *j* verdeckten Neuronen und der *i* Ausgabeneuronen als Spaltenvektor aufgefaßt werden:

$$\vec{x} = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_k \end{pmatrix} \quad \text{und} \quad \vec{h} = \begin{pmatrix} h_0 \\ h_1 \\ \vdots \\ h_j \end{pmatrix} \quad \text{und} \quad \vec{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_i \end{pmatrix}. \tag{2.16}$$

Hierbei ist bereits die Erweiterung der Eingabe- und der verdeckten Schicht um einen konstanten Eingabewert $x_0=1$ und $h_0=1$ (*bias*) berücksichtigt (vgl. Kapitel 2.2.4). Dieser vereinfacht die Darstellung des Algorithmus, da der Schwellwert θ der Neuronen nicht separat berücksichtigt werden muß. Somit berechnet ein Backpropagation-Netzwerk eine Abbildung *A*

$$\vec{y} = A(\vec{x}).$$

Jedes Neuron berechnet dabei seine Erregung gemäß Gleichung (2.5) sowie seine Ausgabe nach der Sigmoidfunktion (vgl. Gleichung (2.8))

$$g(a) = \frac{1}{1+e^{-ca}} \ . \tag{2.17}$$

Dabei gilt für die verdeckten Neuronen

$$h_j = g(a_j) \quad \text{mit} \quad a_j = \sum_k \omega_{jk} \cdot x_k \tag{2.18}$$

und für die Ausgabeneuronen

$$y_i = g(a_i) \quad \text{mit} \quad a_i = \sum_j \omega_{ij} \cdot h_j \ . \tag{2.19}$$

Da im Netzwerk jeweils alle Neuronen in zwei benachbarten Schichten vollständig verbunden sind, ergibt sich für die Gewichte zwischen zwei Schichten eine zweidimensionale Matrix Ω

$$\Omega_{jk} = \begin{pmatrix} \omega_{10} & \cdots & \omega_{1k} \\ \vdots & \ddots & \vdots \\ \omega_{j0} & \cdots & \omega_{jk} \end{pmatrix} \quad \text{und} \quad \Omega_{ij} = \begin{pmatrix} \omega_{10} & \cdots & \omega_{1j} \\ \vdots & \ddots & \vdots \\ \omega_{i0} & \cdots & \omega_{ij} \end{pmatrix}. \tag{2.20}$$

Somit lassen sich die Berechnungen der einzelnen Schichten als Matrixmultiplikation darstellen:

$$\vec{h} = g(\Omega_{jk} \cdot \vec{x}) \quad \text{und} \quad \vec{y} = g(\Omega_{ij} \cdot \vec{h}). \tag{2.21}$$

Aus den beiden Gleichungen (2.21) wird auch ersichtlich, warum als Aktivierungsfunktion *g(a)* eine *nichtlineare* Funktion verwendet wird. Wäre diese nämlich *linear*, d. h. nur eine Gerade, so könnte die Ausgabe des Netzwerkes $\vec{y}$ ausschließlich durch Matrixmultiplikationen und Multiplikation mit einer Konstanten ermittelt werden. Dies würde aber bedeuten, daß die einzelnen synaptischen Gewichte ω analytisch bestimmbar sind und somit kein aufwendiger Trainingsalgorithmus nötig wäre. Weiterhin würde dies bedeuten, daß das Netzwerk nur *orthogonale* Eingangsvektoren unterschieden kann, was eine erhebliche Einschränkung der Menge der möglichen Eingangsvektoren darstellen würde. Aufgrund der *Assoziativeigenschaft* der Matrixmultiplikation wären dann auch die verdeckten Neuronenschichten ohne Bedeutung, da die Abbildung $\vec{y} = A(\vec{x})$ dann durch eine einzige Matrixmultiplikation darstellbar wäre.

2.3.2 Lernalgorithmus

Die Abbildung *A*, die von einem BPG-Netzwerk berechnet wird, stellt aufgrund der Nichtlinearität der Aktivierungsfunktion *g(a)* ein nichtlineares Gleichungssystem dar. Der BPG-Lernalgorithmus kann daher als ein numerisches Verfahren zur Lösung dieses Gleichungssystems angesehen werden. Für ein bestimmtes Lernproblem mit *m* Lernmustern sucht der BPG-Algorithmus das Minimum einer Fehlerfunktion *E*, indem der negativen Gradientenrichtung ∇E gefolgt wird. Hierzu muß die Fehlerfunktion an allen Stellen partiell nach den synaptischen Gewichten ω differenzierbar sein. Die Fehlerfunktion *E* ist definiert durch

$$E = \frac{1}{2} \cdot \sum_{i=1}^{m} (\vec{t}_i - \vec{y}_i)^2 \,. \tag{2.22}$$

Hierbei stellt der Vektor $\vec{t}_i$ die erwünschte Ausgabe des Netzwerkes dar, der Vektor $\vec{y}_i$ die tatsächliche Netzwerkausgabe.

Vor Beginn eines Lernzyklus werden zunächst alle synaptischen Gewichte ω im Netzwerk mit zufälligen Startwerten initialisiert. Ein Lernzyklus beginnt dann mit einem zufällig aus den *m* Lernmustern $\left(\vec{x}_i; \vec{t}_i\right)$ ausgewählten Muster. Für dieses wird die Netzwerk-Ausgabe $\vec{y}$ gemäß (2.21) berechnet. Dieser Vorgang wird als *feed forward* bezeichnet. Anschließend folgt der erste *Backpropagation*-Schritt, indem zunächst die Korrekturen der synaptischen Gewichte ω_{ij} der Ausgabeneuronen in Abhängigkeit von der partiellen Ableitung der Fehlerfunktion *E* berechnet werden:

$$\Delta\omega_{ij} = -\eta \cdot \frac{\partial E}{\partial \omega_{ij}} = -\eta \cdot \delta_i \cdot h_j \tag{2.23}$$

mit

$$\delta_i = (y_i - t_i) \cdot g'(a_i) \,. \tag{2.24}$$

Anschließend folgt der zweite Backpropagation-Schritt, indem die Korrekturen der Gewichte der verdeckten Schichten berechnet werden:

$$\Delta\omega_{jk} = -\eta \cdot \frac{\partial E}{\partial \omega_{jk}} = -\eta \cdot \delta_j \cdot x_k \tag{2.25}$$

mit

$$\delta_j = \sum_i \omega_{ij} \cdot \delta_i \cdot g'(a_j) \,. \tag{2.26}$$

Der Parameter η wird hierbei als *Lernrate* bezeichnet, ein typischer Wert hierfür ist $\eta = 0{,}01$. Mit g' wird die erste Ableitung der Aktivierungsfunktion bezeichnet. Für die Sigmoidfunktion (2.17) ergibt sich hierfür

$$g'(a) = c \cdot g(a) \cdot g(-a). \tag{2.27}$$

Dies erweist sich bei der soft- bzw. hardwaremäßigen Realisierung des BPG-Algorithmus von Vorteil, da die Berechnung der Ableitung von *g(a)* auf die Funktion selbst sowie die gespiegelte Funktion *g(-a)* zurückgeführt wird.

Nachdem alle Korrekturen für ein Lernmuster berechnet worden sind, werden die synaptischen Gewichte entsprechend korrigiert. Anschließend werden die Korrekturen für das nächste Lernmuster vorgenommen und so weiter, bis für alle *m* Lernmuster Korrekturen durchgeführt sind. Dieser Vorgang wird so oft wiederholt, bis der Fehler *E* unter einer festgelegten Schranke liegt bzw. eine vorher festgelegte maximale Anzahl von Lernzyklen erreicht ist. Bei dieser Vorgehensweise wird vom *online-Training* gesprochen. Werden hingegen erst die Korrekturen für alle *m* Lernmuster berechnet und erst dann die Gewichte entsprechend korrigiert, so spricht man vom *Batch-Verfahren*.

2.3.3 Konvergenz

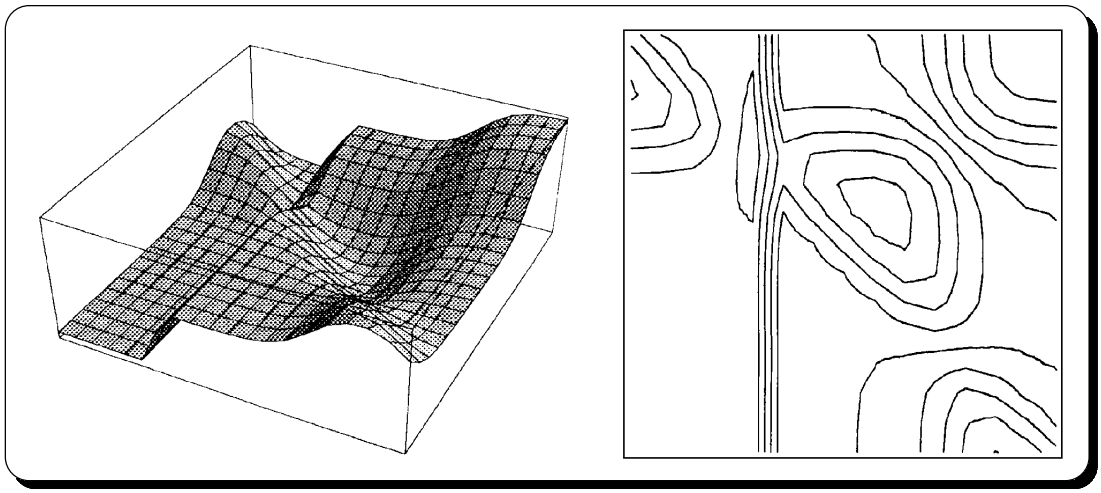


Bild 2.15: Fehlerfunktion (links) für ein Lernproblem mit Höhenlinien (rechts) [Roj93]

Das Ziel des Backpropagation-Algorithmus ist das Auffinden des Minimums der Fehlerfunktion *E*. Bei einfachen Problemstellungen, wie z. B. den logischen Funktionen AND und OR, besitzt die Fehlerfunktion nur ein einziges *globales Minimum*, das die optimale Lösung des Problems darstellt. Wird die Struktur des Netzwerkes jedoch umfangreicher, z. B. mit mehreren verdeckten Neuronenschichten, so entstehen häufig *lokale Minima*, die zwar *eine mögliche Lösung* des Problems darstellen, jedoch nicht die *optimale Lösung*. Da der Backpropagation-Algorithmus ein Gradientenabstiegsverfahren darstellt, hat der Algorithmus keine Möglichkeit, ein einmal erreichtes lokales Minimum zu überwinden. Da ein Minimum ein Tal der Fehlerfunktion darstellt, müßte der Algorithmus kurzzeitig der Fehlerfunktion in aufsteigender Richtung folgen, um das Tal verlassen zu können (Bild 2.15).

2.3.4 Varianten des BPG-Algorithmus

Vom Backpropagation-Algorithmus sind zahlreiche Varianten bekannt, von denen einige im folgenden kurz erläutert werden. Dabei werden für die Beschreibung alle Gewichte im Netzwerk zu einem Gewichtsvektor $\vec{\omega}$ zusammengefaßt. Ein neuer Gewichtsvektor $\vec{\omega}_t$ berechnet sich dann aus dem vorherigen Gewichtsvektor $\vec{\omega}_{t-1}$ und dem Gradienten der Fehlerfunktion E multipliziert mit der Lernrate η

$$\vec{\omega}_t = \vec{\omega}_{t-1} - \eta \cdot \vec{\nabla} E_t = \vec{\omega}_{t-1} + \Delta\vec{\omega}_t \; . \tag{2.28}$$

- **Backpropagation mit Impuls**

  Besitzt die Fehlerfunktion nur ein Minimum in einem schmalen Tal, so führt das Gradientenabstiegsverfahren zu unerwünschten Oszillationen, d. h. das Tal wird immer knapp von einer Seite auf die andere übersprungen bzw. es erfolgen abwechselnd Korrekturen mit unterschiedlichen Vorzeichen (Bild 2.16). Zur Vermeidung solcher Oszillationen wird daher ein *Impulsterm* eingefügt, der diese Oszillationen dämpfen soll:

  $$\Delta\vec{\omega}_t = -\eta \cdot \vec{\nabla} E_t + \alpha \cdot \Delta\vec{\omega}_{t-1} \; . \tag{2.29}$$

  Der Impulsterm sorgt dafür, daß bei jeder Korrektur der Gewichte ein Teil der vorhergehenden Korrektur berücksichtigt wird. Dies vermindert die Oszillationen, da aufgrund der unterschiedlichen Vorzeichen der aktuellen und der vorhergehenden Korrektur die Korrekturwerte rasch kleiner werden, womit das schmale Tal des Minimums schneller gefunden wird. Ein weiterer Vorteil des Impulstermes ergibt sich in sehr flachen Gebieten der Fehlerfunktion. Hierbei bildet sich bei nahezu konstantem Gradienten eine *geometrische Reihe* von Korrekturtermen, wodurch die Schrittlänge ständig vergrößert wird. Auf diese Weise können solche flachen Regionen schnell überwunden werden. Das in Kapitel 2.2.6 beschriebene XOR-Problem konvergiert bei Hinzunahme des Impulstermes in ca. 100 Lernzyklen, während ohne Impulsterm ca. 600-1000 Lernzyklen benötigt werden.

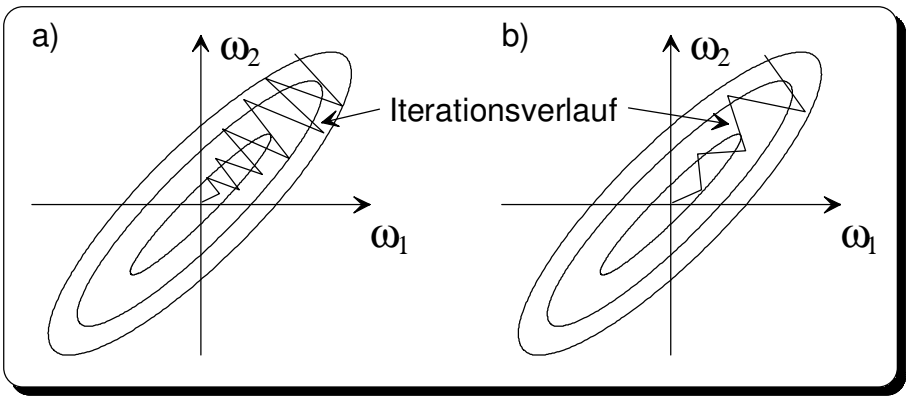


Bild 2.16: Backpropagation ohne (a) und mit Impulsterm (b) [Roj93]

- **Backpropagation mit variabler Schrittlänge**

  Bei diesem Verfahren wird versucht, die Konvergenzgeschwindigkeit durch Verändern der Schrittlänge, d. h. durch Anpassen der Lernkonstante η an die Fehlerfläche, zu verbessern. Dabei wird der Gradient der Fehlerfunktion E nicht jedesmal neu berechnet, sondern nur wenn die Fehlerfunktion wieder steigende Werte annimmt. In diesem Fall wird die letze Korrektur aufgehoben und der Gradient neu berechnet. Zur Anpassung der Schrittlänge wird gezählt, wie oft eine Korrektur in dieselbe Richtung des Gradienten der Fehlerfunktion E vorgenommen wurde. Wurden dabei wenige Korrekturen vorgenommen, so wird die Schrittlänge verkleinert, d. h. der Wert der Lernkonstante η wird verringert, wohingegen bei vielen Korrekturen die Schrittlänge durch Erhöhen von η vergrößert wird. Dieses Verfahren wurde 1988 von Hush und Salas als *gradient reuse* Algorithmus vorgestellt.

- **Quickprop**

  Zur Beschleunigung des Lernvorgangs lassen sich Informationen über die Krümmung der Fehlerfläche verwenden, wobei die zweite partielle Ableitung der Fehlerfunktion E, erforderlich ist. Die Berechnung der zweiten Ableitung erfordert dabei jedoch größeren Aufwand, weshalb Scott Fahlmann 1989 den *Quickprop*-Algoritmus vorgeschlagen hat, der die zweite partielle Ableitung lediglich approximiert. Die Fehlerfunktion wird dafür für jedes Gewicht in Richtung des Gradienten quadratisch approximiert.

Das Ziel aller beschriebenen Varianten des Backpropagation-Algorithmus ist die Verbesserung der Konvergenzgeschwindigkeit, d. h. die Beschleunigung des Lernvorgangs. Die Bestimmung der optimalen Werte der Parameter, wie z. B. der Lernrate η oder der Konstante α beim Impulsterm, muß dabei meist in Abhängigkeit vom Lernproblem empirisch bestimmt werden. Allgemeingültige Werte dieser Parameter existieren nicht.

3 ALVINN

Neben vielen anderen Anwendungen eignen sich neuronale Netzwerke, insbesondere Backpropagation-Netzwerke auch für Aufgaben aus dem Bereich der Bilderkennung. Unter diesem Aspekt wurde an der Carnegie Mellon University von Dean A. Pomerleau *ALVINN* (*Autonomous Land Vehicle In a Neural Network*, [Pom93]) entwickelt, das im folgenden kurz beschrieben wird.

3.1 Überblick

Bei der Entwicklung von ALVINN setzte man sich zum Ziel, Fahrzeuge mit Hilfe von neuronalen Netzwerken visomotorisch zu steuern. Als Netzwerktyp wurde für ALVINN ein Backpropagation-Netzwerk gewählt, da der Backpropagation-Algorithmus, im Gegensatz zu vielen anderen Lernalgorithmen, ein überwachtes Lernen durch Beispiele ermöglicht. Dies entspricht der Situation in einem Fahrzeug, wo das Lenkverhalten eines menschlichen Fahrzeuglenkers vom Netzwerk "beobachtet" werden kann. Vom Backpropagation-Algorithmus existieren zahlreiche Abwandlungen, wobei für ALVINN Quickprop ausgewählt wurde, da dieser schneller konvergiert als der normale nicht modifizierte Backpropagation-Algorithmus und somit einen schnelleren Lernerfolg ermöglicht (vgl. [Roj93]).

Bei der Wahl der Netzwerkarchitektur wurde bei ALVINN mit verschiedenen Strukturen experimentiert. Das Spektrum umfaßte dabei 0 bis 2 verdeckte Schichten sowie 0 bis 70 verdeckte Neuronen. Letztendlich entschied man sich jedoch für ein relativ einfaches BPG-Netzwerk mit einer verborgenen Schicht (Bild 3.1). Die Experimente hatten ergeben, daß keine der untersuchten Architekturen nennenswert bessere Ergebnisse lieferte als die gewählte.

Im Rahmen der Testreihen von ALVINN wurde vorwiegend die Steuerung eines Kleinlasters (*Navlab*) auf verschiedenen Straßentypen untersucht. Die Straßentypen reichten dabei vom einfachen Waldweg bis zum mehrspurigen Highway. Bei den Versuchen wurde das Netzwerk häufig "on-the-fly" trainiert, d. h. es wurde ein menschlicher Fahrer etwa fünf Minuten beobachtet, aus dessen Fahrverhalten sich dann die Trainingsdatenmenge ergab. Beim stationären Training des Netzwerkes wurde zur Gewinnung einer möglichst großen Trainingsdatenmenge ein Verfahren namens *structured noise* verwendet. Hierbei werden aus vorhandenen Bildern algorithmisch verfremdete ähnliche Bilder erzeugt, um die Menge aufzunehmender Bilder zu reduzieren. Aus einem Bild mit einer geraden Fahrbahn in Bildmitte wird so ein Satz von 10 oder 20 Bildern, wobei die Fahrbahn auf jedem neuen Bild ein wenig weiter rechts oder links der Bildmitte liegt.

Das Navlab ist zur Berechnung des Netzwerkes mit insgesamt drei Sun SparcStations bestückt. Als Eingabesensoren werden vorne und hinten am Fahrzeug jeweils eine Farbkamera

verwendet. Die Steuerung der Lenkung wird von einem 68020-Prozessor übernommen, der auch die Positionsbestimmung des Fahrzeuges relativ zur Startposition durchführt.

Mit dem bei ALVINN verwendeten Netzwerk wurde im Rahmen der Forschungen auch ein von der NASA[1] entwickelter Roboter gesteuert, der sich, ohne Vorhandensein der Erdanziehungskraft, außerhalb einer Raumstation bewegen kann.

3.2 Netzwerk-Architektur

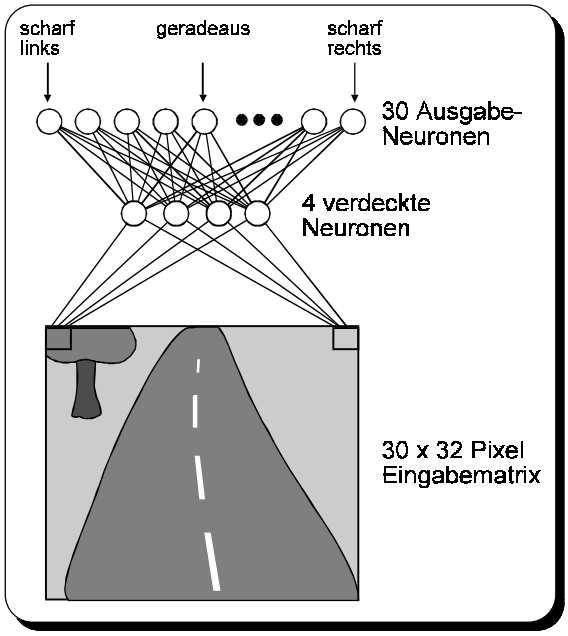


Bild 3.1: ALVINN Netzwerk-Architektur [Pom93]

Das bei ALVINN verwendete Backpropagation-Netzwerk besteht aus drei Schichten:

- **Eingabeschicht**

  Als Eingabeschicht dient eine zweidimensionale Matrix mit 30 x 32 Elementen. Diese Matrix kann von verschiedenen Eingabesensoren, z. B. von einer Videokamera, stammen, wobei stets eine Bildvorverarbeitung inklusive Auflösungsreduktion erfolgt.

- **Verdeckte Schicht**

  In der verdeckten Schicht werden 4 Neuronen verwendet.

- **Ausgabeschicht**

  Als Ausgabe liefert das Netzwerk einen Vektor mit 30 Elementen (= Ausgabeneuronen), der die Richtung angibt, in die das Fahrzeug gesteuert werden muß, um auf der Fahrbahn zu bleiben.

Das Netzwerk selbst ist voll verbunden, d. h. jede Eingabestelle ist mit jedem verdeckten Neuron und jedes verdecktes Neuron ist mit jedem Ausgangsneuron verbunden.

[1] *NASA* - amerikanische Raumfahrtbehörde

3.3 Bildvorverarbeitung bei ALVINN

ALVINN ermöglicht die Verwendung unterschiedlicher Bildsensoren (Videokamera, Infrarotkamera, Laserentferungsmesser), da ein Bild für das Netzwerk nur eine beliebige zweidimensionale Matrix darstellt. In der Regel werden jedoch Videobilder, die von einer Schwarz/ Weiß- oder Farbkamera geliefert werden, verwendet. Allen Aufnahmeverfahren ist jedoch gemeinsam, daß die dabei gewonnenen Bilder zunächst einmal durch Auflösungsreduktion auf die Größe der Eingabematrix (30 x 32) des Netzwerkes gebracht werden müssen. Anschliessend ist eine Normierung der Bildpunkthelligkeiten erforderlich, damit alle Bilder gleiches Helligkeitsniveau haben.

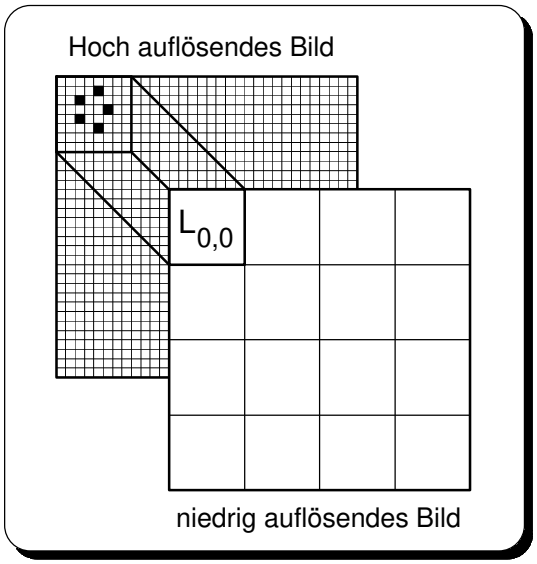


Bild 3.2: Reduktion der Auflösung bei ALVINN [Pom93]

Bei der bei ALVINN verwendeten Ausgangsbildgröße von 480 x 512 Pixeln müssen jeweils Quadrate der Größe 16 x 16 = 256 Pixel zu einem Pixel des reduzierten Bildes ($L_{0,0}$, siehe Bild 3.2) zusammengefaßt werden. Hierfür werden bei ALVINN etwa 3 % der 256 Pixel zufällig ausgewählt und anschließend arithmetisch gemittelt. Der daraus resultierende Wert ist dann ein Maß für den mittleren Helligkeitswert des Ausgangsbildes in diesem Bereich.

Im Anschluß an die Auflösungsreduktion werden die Bilder einem Histogrammausgleich unterzogen. Hierbei werden jeweils die dunkelsten und hellsten 5 % der Pixel auf den minimalen (-1) bzw. maximalen (+1) Wert gesetzt. Die restlichen 90 % der Pixel werden dann proportional zu ihrer Helligkeit über den gesamten Wertebereich normiert.

Da als Eingabe des Netzwerkes nur Graustufenbilder bzw. genauer monochrome[1] Bilder verwendet werden, müssen eventuelle Farbbilder geeignet in Graustufenbilder konvertiert werden. Bei ALVINN werden die von der Kamera gelieferten Farbbilder so in Graustufenbilder konvertiert, daß Schatten von Umgebungsobjekten weitgehend eliminiert werden. Dabei wird ausgenutzt, daß derartige Schatten typischerweise einen hohen Blauanteil aufweisen [Pom93].

[1] *griech. monochrom* ≈ einfarbig

3.4 Ausgangsrepräsentation

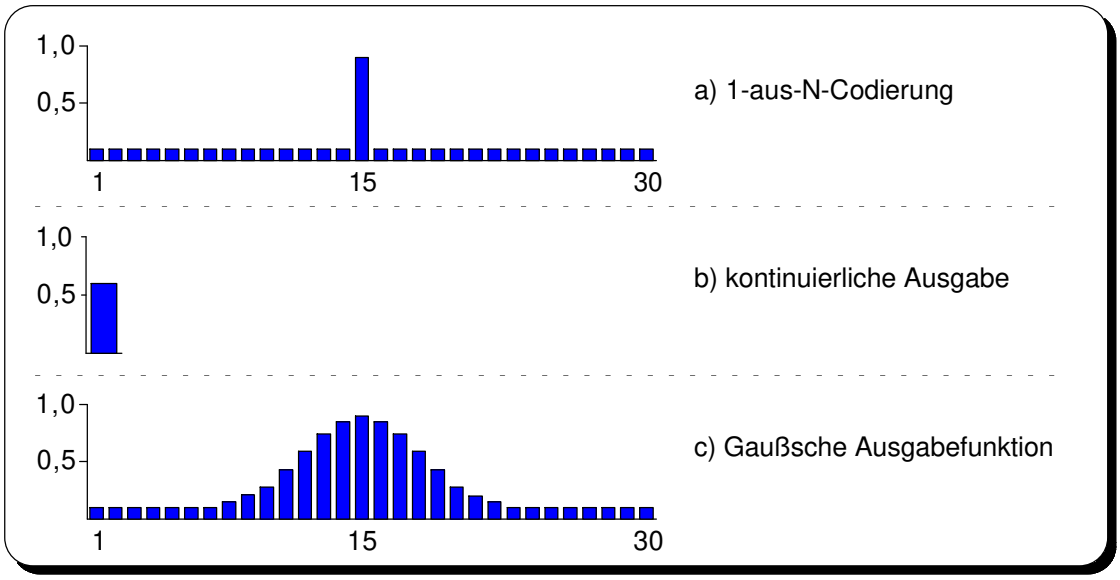


Bild 3.3: verschiedene Möglichkeiten der Ausgangsrepräsentation

Bei ALVINN wird die Ausgabe des neuronalen Netzwerkes dazu verwendet, ein Fahrzeug so zu lenken, daß es einem beliebigen Fahrbahnverlauf folgt. Hierbei gibt es verschiedene Möglichkeiten, wie die Richtungsinformation zur Steuerung der Lenkung aus der Netzwerkausgabe durch geeignete Codierung ermittelt werden kann:

- **1-aus-N-Codierung** (Bild 3.3a)

  Diese Codierung verwendet eine bestimmte Anzahl von Ausgabeneuronen, wobei jedem Neuron eine bestimmte Richtung zugeordnet ist. Dabei kann immer nur ein Neuron aktiv sein, während die anderen inaktiv sind. Der Nachteil dieser Lösung besteht darin, daß zum einen die Anzahl der möglichen Ausgabewerte auf die Anzahl der Ausgabeneuronen beschränkt ist und so keine kontinuierliche Steuerung eines Lenkmotors möglich ist. Ein weiterer Nachteil besteht in der Notwendigkeit, daß für jeden möglichen Ausgabewert, d. h. für die Aktivierung jedes Ausgabeneurons, mindestens ein Testmuster zur Verfügung stehen muß. Fehlt ein Muster, so nimmt das Netzwerk an, das betreffende Ausgabeneuron wäre immer inaktiv.

- **kontinuierliche Ausgabe eines Neurons** (Bild 3.3b)

  Hierbei wird nur ein Ausgabeneuron verwendet, wobei das Neuron eine kontinuierliche Ausgabe zwischen zwei Extremwerten, z. B. 0 und 1, liefert. Je nach verwendeter Stellenzahl liefert diese Lösung zwar mehr mögliche Ausgabewerte als die 1-aus-N-Codierung, jedoch führt dabei jede mögliche Netzwerkeingabe zu einer Reaktion. Da alle möglichen Ausgabewerte gültige Werte darstellen, kann die Plausibilität der Reaktion nicht überprüft werden, wie z. B. bei der 1-aus-N-Codierung. Hier gehören bestimmte Reaktionen, wie die gleichzeitige Aktivierung zweier Neuronen, nicht zur Menge der gültigen Ausgabewerte. Die kontinuierliche Ausgabe führt jedoch immer zu einem gültigen Ausgabewert, wenn auch zu einem, der nicht zur Eingabe paßt. Ein weiterer Nachteil dieser Codierung liegt im schlechten Konvergenzverhalten beim Lernen.

- **Gaußsche Ausgabefunktion** (Bild 3.3c)

  Hierfür wird wie bei der 1-aus-N-Codierung eine größere Anzahl von Ausgabeneuronen verwendet. Jedoch ist dabei nicht immer nur ein Neuron aktiviert, sondern die Aktivierung aller Neuronen entspricht einer Gaußschen Funktion. Durch Interpolation können dabei leichte Fehlreaktionen einzelner Ausgabeneuronen ausgeglichen werden. Als Reaktion des Netzwerkes wird dabei das Neuron, über dem das Maximum der Gaußschen Funktion liegt, gewertet und damit die Lenkrichtung bestimmt. Die einzelnen Ausgabeneuronen liegen dabei auf der Abszisse, während die Ordinate die Aktivierung der einzelnen Ausgabeneuronen darstellt. Liegt das Maximum zwischen zwei Neuronen, so ergeben sich Zwischenwerte, d. h. liegt das Maximum zwischen Neuron 15 und 16, so wäre der Ausgabewert z. B. 15,3.

ALVINN verwendet die letztgenannte Möglichkeit der Ausgangsrepräsentation mit insgesamt 30 Ausgabeneuronen, wobei das erste Neuron eine Linksbewegung, das fünfzehnte Neuron geradeaus und das dreißigste Neuron eine Rechtsbewegung anzeigt.

4 Bildvorverarbeitung

Im Rahmen der vorliegenden Diplomarbeit wurde zunächst ein FPGA (*Field Programmable Gate Array*) zur Auflösungsreduktion von Bilddaten entwickelt, das im folgenden beschrieben wird. Es ist wie das Digitalisiersystem MegaDigi [Hah95] und der NeNEB-Chip Teil eines Bildklassifikationssystems (vgl. Kapitel 1), wobei der Schwerpunkt dieser Arbeit in der Realisierung des NeNEB-Chips liegt.

4.1 Notwendigkeit und Einordnung

Der NeNEB-Chip ist ein Teil eines Gesamtsystems zur Echtzeit-Klassifikation von Bilddaten (vgl. Kapitel 1). Die Eingangsbildgröße des NeNEB-Chips beträgt maximal 256 x 256 Pixel bei 50 Bildern pro Sekunde. Videokameras liefern jedoch typischerweise eine höhere Auflösung. Daher ist eine Auflösungsreduktion der von der Kamera gelieferten Bilder nötig, wobei nur Graustufenbilder bzw. monochrome Bilder betrachtet werden (vgl. Bild 4.1). Der Begriff "Graustufe" wird im folgenden stets synonym für Helligkeitswert eines Pixels verwendet, unabhängig davon, ob es sich tatsächlich um ein Graustufenbild handelt oder um den Auszug eines Farbbildes in einer Grundfarbe oder gar um ein Infrarotbild.

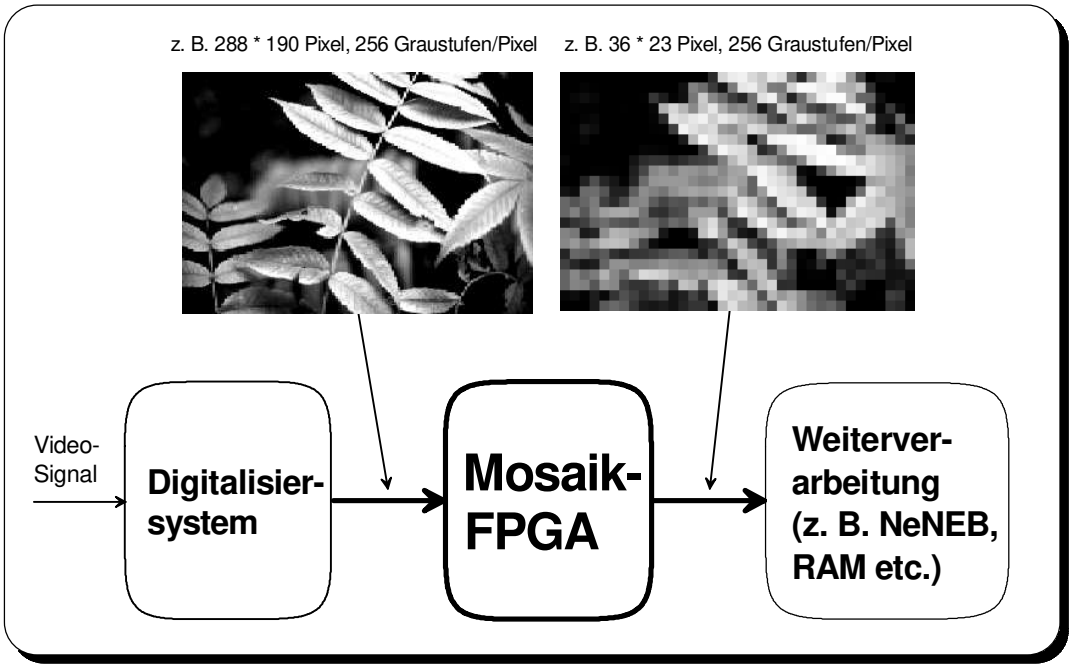


Bild 4.1: Gesamtsystem mit Mosaik-FPGA zur Auflösungsreduktion

4.2 Auswahl des Reduktionsverfahrens

Bei der Auswahl des Reduktionsverfahrens waren drei Randbedingungen maßgebend:

1. Echtzeitverarbeitung
2. Hardware-nah zu realisierender Reduktionsalgorithmus (FPGA)
3. serieller Pixeldatenstrom

Die erste Randbedingung der Echtzeitverarbeitung wird dadurch relativiert, daß die Berechnung des Reduktionsalgorithmus in einem seriellen Datenstrom von Bilddaten (Randbedingung 3) erfolgt und somit nicht ein komplettes Bild parallel berechnet werden muß. Leider erschwert dies auch die Gestaltung des Reduktionsalgorithmus.

Durch die zweite Forderung eines Hardware-nah zu realisierenden Reduktionsalgorithmus fiel die Wahl auf eine einfache Mosaikbildung, d. h. Ermittlung des mittleren Grauwertes einer quadratischen Region von Pixeln für das reduzierte Pixel (siehe Bild 4.2). Um bei diesem Verfahren optimale Ergebnisse zu erzielen, ist es eigentlich notwendig, daß das gesamte Bild für die Berechnung vorliegt. In diesem Fall werden einfach alle Pixelwerte in der quadratischen Region aufaddiert und abschließend durch die Anzahl der Pixel des Quadrates geteilt (arithmetischer Mittelwert). Bei diesem Verfahren ergeben sich auch bei ganzzahliger Rechnung nur geringe Rundungsfehler. Da die Bilddaten jedoch nicht parallel vorliegen, mußte das Verfahren etwas modifiziert werden, wie im folgenden beschrieben wird.

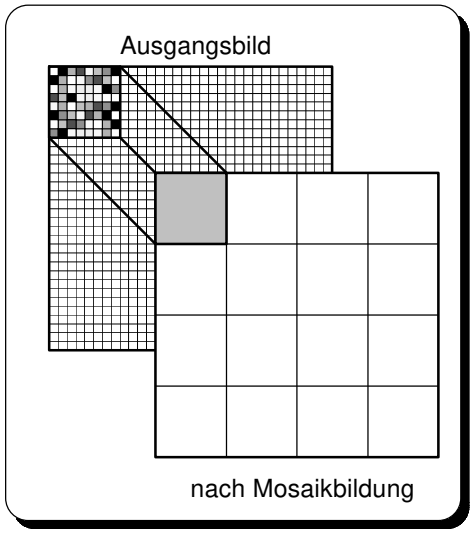


Bild 4.2: Mosaikbildung zur Auflösungsreduktion

4.3 Darstellung des implementierten Algorithmus

Der serielle Bilddatenstrom besteht aus einzelnen Bytes, die jeweils den Grauwert eines Pixels darstellen. Die Reihenfolge der Pixel ist dabei, wie bei Videobildern üblich, zeilenweise von oben nach unten sowie innerhalb der Zeilen spaltenweise von links nach rechts. Ein Quadrat von Pixeln (vgl. Bild 4.2) ist daher erst nach mehreren Zeilen voll verfügbar. Somit kann eine Aufaddierung der Pixelwerte eines zusammenzufassenden Quadrates auch nur zeilenweise erfolgen, wobei jeweils Zwischenergebnisse gespeichert werden müssen, da innerhalb einer Zeile

des Ausgangsbildes jeweils mehrere Quadrate liegen. Je nach Kantenlänge des Quadrates, d. h. je nachdem wieviel Pixelwerte zusammengefaßt werden, variiert die Größe des Zwischenspeichers, da im Falle vieler zusammenzufassender Pixelwerte entsprechend große Zwischenergebnisse bzw. Summationsergebnisse auftreten können. Bei einer Quadratgröße von 16 x 16 Pixeln beträgt z. B. die erforderliche Speicherwortbreite

$$\lceil \log_2(255 * 16 * 16) \rceil = 16 \text{ Bit } [1].$$

Jede weitere Verdoppelung der Kantenlänge des Quadrats läßt die benötigte Speicherwortbreite um 2 Bit ansteigen. Würde man jedoch in dem Zwischenspeicher nicht die Summationsergebnisse jeder Quadratzeile speichern, sondern jeweils nach einer Quadratzeile nur den entsprechenden Mittelwert, so würde sich die benötigte Speicherwortbreite auf

$$\lceil \log_2(255 * 16) \rceil = 12 \text{ Bit}$$

reduzieren. Dies erscheint auf den ersten Blick nicht gerade sinnvoll, da sich die benötigte Anzahl nur auf 75 % reduziert. Jedoch ist dabei zu beachten, daß bei größeren Quadraten der Gewinn bzw. die Einsparung ebenfalls prozentual größer wird. Die Verringerung der Speicherwortbreite hat aber auch andere Vorteile hinsichtlich der gewünschten Implementierung als FPGA sowie unter der Berücksichtigung, daß eine Echtzeitverarbeitung des Algorithmus vonnöten ist:

- Die Recheneinheiten (Addierer, Register) im FPGA benötigen ebenfalls die ermittelte Speicherwortbreite, jedoch ist die Anzahl der Logikzellen in einem FPGA (128 beim verwendeten FPGA) eng begrenzt.
- Jedes zusätzliche Bit bei den Recheneinheiten vermindert aufgrund komplexer werdender Logik die Arbeitsgeschwindigkeit.

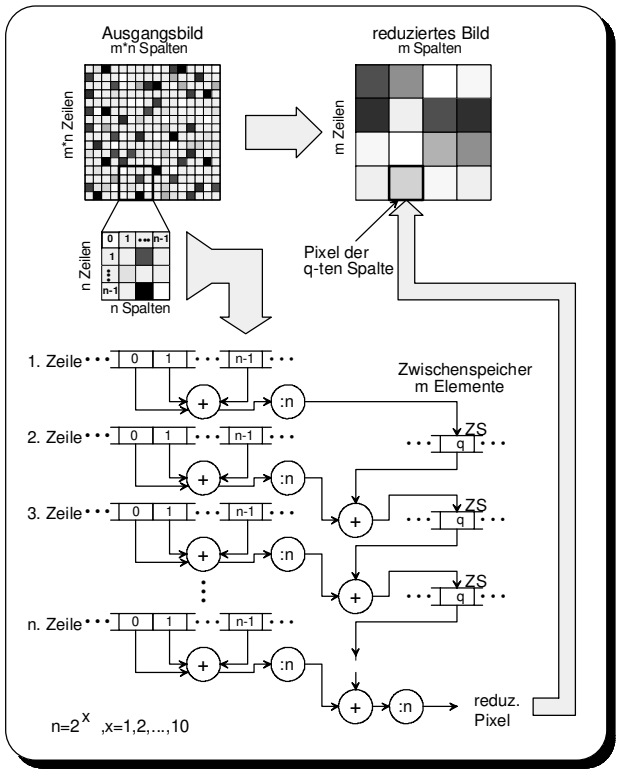


Bild 4.3: Algorithmus zur Mosaikbildung

[1] größter Grauwert eines Pixels = 255; 16 x 16 Pixel im Pixel-Quadrat

Das Bild 4.3 auf der vorhergehenden Seite stellt den implementierten Algorithmus schematisch dar. Als Beispiel wird hier ein Bild mit *m*n* Zeilen und *m*n* Spalten betrachtet, wobei *n* eine Zweierpotenz und das Produkt aus *n* und *m* eine ganze Zahl ist. Das zusammenzufassende Quadrat enthält somit n^2 Pixel. Es ist im Beispielbild das *q-te* ($0 \leq q \leq m-1$) zusammenzufassende Quadrat innerhalb einer Zeile des Ausgangsbildes und somit auch das *q-te* Pixel innerhalb einer Zeile des reduzierten Bildes. Im Zwischenspeicher, der mindestens *m* Elemente groß sein muß, wird ebenfalls der *q-te* Speicherplatz für die Zwischenspeicherung benutzt. Die Berechnung erfolgt nun, wie zuvor schon beschrieben, indem zunächst die Pixelwerte der ersten Zeile des zusammenzufassenden Quadrates aufaddiert werden. Anschließend wird das Summationsergebnis durch die Anzahl *n* der Pixel geteilt und das Divisionsergebnis, welches das Mittel der Pixelwerte darstellt, im Zwischenspeicher gesichert. Mit den weiteren in der Zeile des Ausgangsbildes zusammenzufassenden Quadraten wird ebenso verfahren, bis das Zeilenende erreicht ist. Daraufhin wird mit der zweiten Zeile der zusammenzufassenden Quadrate begonnen. Auch hier werden wieder die Pixelwerte aufaddiert und das Ergebnis durch die Anzahl der Pixel geteilt. Allerdings wird das Divisionsergebnis nun zum Inhalt des Zwischenspeichers addiert. Diese Prozedur wird für alle weiteren Zeilen eines zusammenzufassenden Quadrates wiederholt. Am Ende enthält der Zwischenspeicher die Summe aller Zeilenmittel. Diese Summe wird nun wiederum durch die Anzahl *n* der Zeilen geteilt. Das Ergebnis ist der mittlere Grauwert des zusammenzufassenden Quadrates. Formal kann der Algorithmus wie folgt beschrieben werden:

$$P(I,J) = \left[\sum_{j=0}^{n-1} \left(\left(\sum_{i=0}^{n-1} p(i,j) \right) \operatorname{div} n \right) \right] \operatorname{div} n \,^{1}. \tag{4.1}$$

Hierbei bedeutet *p(i,j)* den Wert des Pixels der *i*-ten Zeile und der *j*-ten Spalte im Originalbild, *P(I,J)* den Wert des Pixels der *I*-ten Zeile und der *J*-ten Spalte im reduzierten Bild (vgl. Bild 4.4).

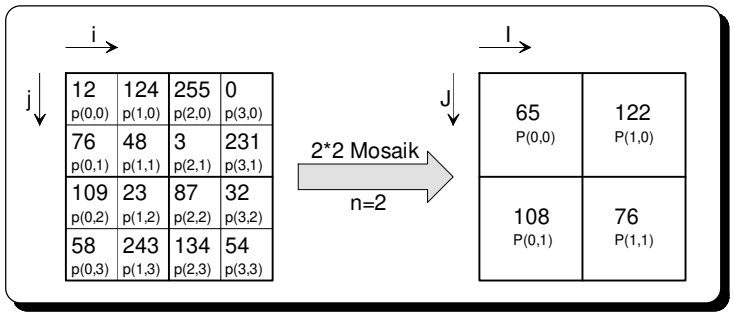


Bild 4.4: Mosaikbildung mit Zahlenbeispielen (Zahlen kennzeichnen Grauwerte des Pixels)

4.4 Mosaik-FPGA zur Auflösungsreduktion

In diesem Abschnitt wird das entwickelte Mosaik-FPGA zur Auflösungsreduktion mit seinen Eigenschaften und dem inneren Aufbau beschrieben. Zusätzlich erfolgt ein kurzer Überblick über den Aufbau des verwendeten FPGA inklusive der Vorgehensweise beim Entwurf einer Logikschaltung für das FPGA.

[1] *div* - ganzzahlige Division ohne Rest

4.4.1 Übersicht und Eigenschaften

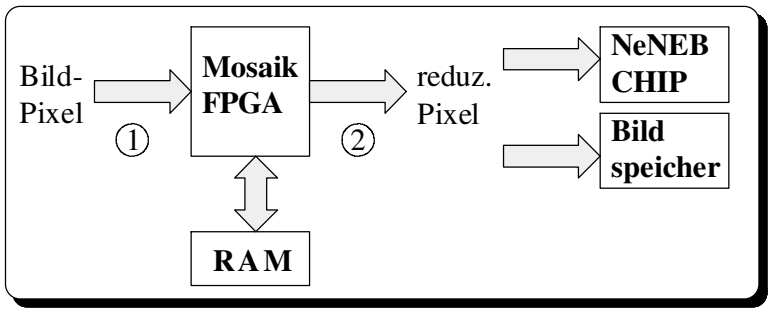


Bild 4.5: Signalflußübersicht Mosaik-FPGA

Die Signalflußübersicht auf Bild 4.5 zeigt die Funktion des Mosaik-FPGA. Auf der linken Seite ankommende Pixel eines hochauflösenden Bildes werden vom Mosaik-FPGA, unter Zuhilfenahme des im linken unteren Teil des Bildes zu sehenden Zwischenspeichers, quadratisch zusammengefaßt. Auf der rechten Seite werden die Pixel des entstehenden niedrig auflösenden Bildes ausgegeben, um dann von verschiedenen anderen Einheiten weiterverarbeitet werden zu können. Diese Weiterverarbeitungseinheiten können z. B. der NeNEB-Chip oder auch nur ein einfacher Bildspeicher sein.

Für die Bildgrößen an den Punkten À und Á gilt folgendes:

- **Punkt 1:**
  hochauflösendes Bild der Größe $i \times j$ (Spalten × Zeilen) mit 8 Bit/Pixel, Pixelrate pr
- **Punkt 2:**
  niedrigauflösendes Bild der Größe $I \times J$ mit 8 Bit/Pixel, reduzierte Pixelrate $pr'=pr/2^{2R}$, $I = i \; div \; 2^R$, $J = j \; div \; 2^R$ (R = *Anzahl der Reduzierstufen*)

Die Größe der zusammenzufassenden Pixelquadrate beträgt 2^{2R} Pixel, mit einer Kantenlänge der Quadrate von 2^R Pixeln. Der Wert 2^R entspricht dabei dem Wert n aus dem vorigen Abschnitt. R kann im Bereich zwischen 1 und 10 liegen.

Die Werte von i und j sind prinzipiell nicht limitiert. Einzig die Größe von I ist auf die Größe des Zwischenspeichers von 512 Elementen begrenzt (vgl. vorigen Abschnitt). Hieraus resultiert auch eine Begrenzung von i in Abhängigkeit von R, und zwar auf den Wert $I*2^R$. Aufgrund der zeilenorientierten Arbeitsweise des Chips ist j nicht limitiert, da eine Ausgabe stets dann erfolgen kann, wenn alle 2^R Zeilen eines Pixelquadrates bearbeitet worden sind. Der gewählte Algorithmus benötigt als Parameter nur die Anzahl der gewünschten Reduzierstufen R. Die Zeilenlänge des Ausgangsbildes wird nicht benötigt, da dem System ein Zeilenanfang-Signal zur Verfügung steht. Die Größe des Zwischenspeichers wird von der Zeilenlänge i und der Anzahl der Reduzierstufen R bestimmt.

Obwohl stets Quadrate, deren Kantenlänge Zweierpotenzen darstellen, zusammengefaßt werden, ist es nicht erforderlich, daß die Zeilenlänge i und die Zeilenanzahl j Zweierpotenzen darstellen. Für die Quadratbildung werden jedoch nur volle Quadrate herangezogen, d. h. übrigbleibende Spalten bzw. Zeilen werden nicht berücksichtigt.

4.4.2 Signalbeschreibungen

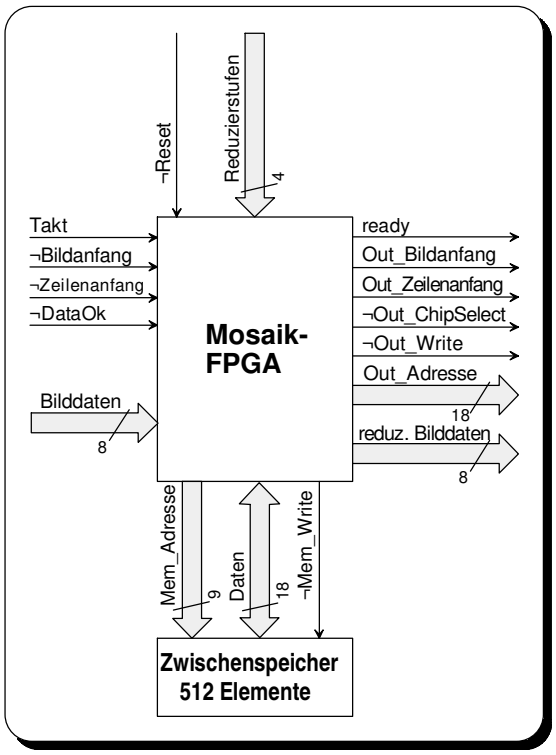


Bild 4.6: Übersicht Mosaik-FPGA

Signalname	**Rich-tung**	**Beschreibung**
¬*Reset*	in	asynchroner low-aktiver Rücksetzeingang
Reduzierstufen	in	4-Bit-Auswahlbus zur Steuerung der Auflösungsreduktion
Takt	in	Systemtakt des Digitalisiersystems
¬*Bildanfang*	in	low-aktives Signal zur Kennzeichnung des Beginns eines neuen Bildes
¬*Zeilenanfang*	in	low-aktives Signal zur Kennzeichnung des Beginns einer neuen Zeile
¬*DataOk*	in	low-aktives Signal, das ein gültiges Pixeldatum kennzeichnet
Bilddaten	in	8-Bit-Pixeldatenbus
ready	out	signalisiert Betriebsbereitschaft des Chips
Out_Bildanfang	out	analog zum Signal ¬*Bildanfang*, jedoch Beginn des Ausgabebildes
Out_Zeilenanfang	out	analog zum Signal ¬*Zeilenanfang*, jedoch Beginn einer neuen Zeile des Ausgabebildes
¬*Out_ChipSelect*	out	low-aktives Auswahlsignal zur Ansteuerung eines externen statischen Bildspeichers
¬*Out_Write*	out	low-aktives Schreibsignal für einen externen Bildspeicher
Out_Adresse	out	18-Bit-Adreßbus für einen externen Bildspeicher
r*educ. Bilddaten*	out	8-Bit-Ausgabebus für reduzierte Bilddaten
¬*Mem-Write*	out	low-aktives Schreibsignal, inaktiv bedeutet Lesezugriff
Mem_Adresse	out	9-Bit-Adreßbus
Daten	bidir	bidirektionaler 18-Bit-Datenbus
Legende: in - Eingang out - Ausgang bidir - bidirektional		

Tabelle 4.1: Signalbeschreibungen Mosaik-FPGA

Das Bild 4.6 auf der vorhergehenden Seite gibt einen Überblick über die Signale beim Mosaik-FPGA. Die Signale sind in vier Gruppen, entsprechend den vier Seiten des Symbols, wie folgt aufgeteilt:

- **obere Seite:** Auf der oberen Seite befinden sich Eingangssignale zur Initialisierung des Chips.
- **linke Seite**: Hier befinden sich die Eingangssignale, die vom Digitalisiersystem geliefert werden.
- **rechte Seite**: Dies ist die Ausgabeseite des Mosaik-FPGA mit allen erforderlichen Ausgabesignalen zur Weiterverarbeitung der reduzierten Bilddaten.
- **untere Seite**: Auf dieser Seite befinden sich die Signale zur Steuerung des Zwischenspeichers.

Eine Beschreibung der Signale sowie die Signalflußrichtungen sind in Tabelle 4.1 enthalten.

4.4.3 Timing

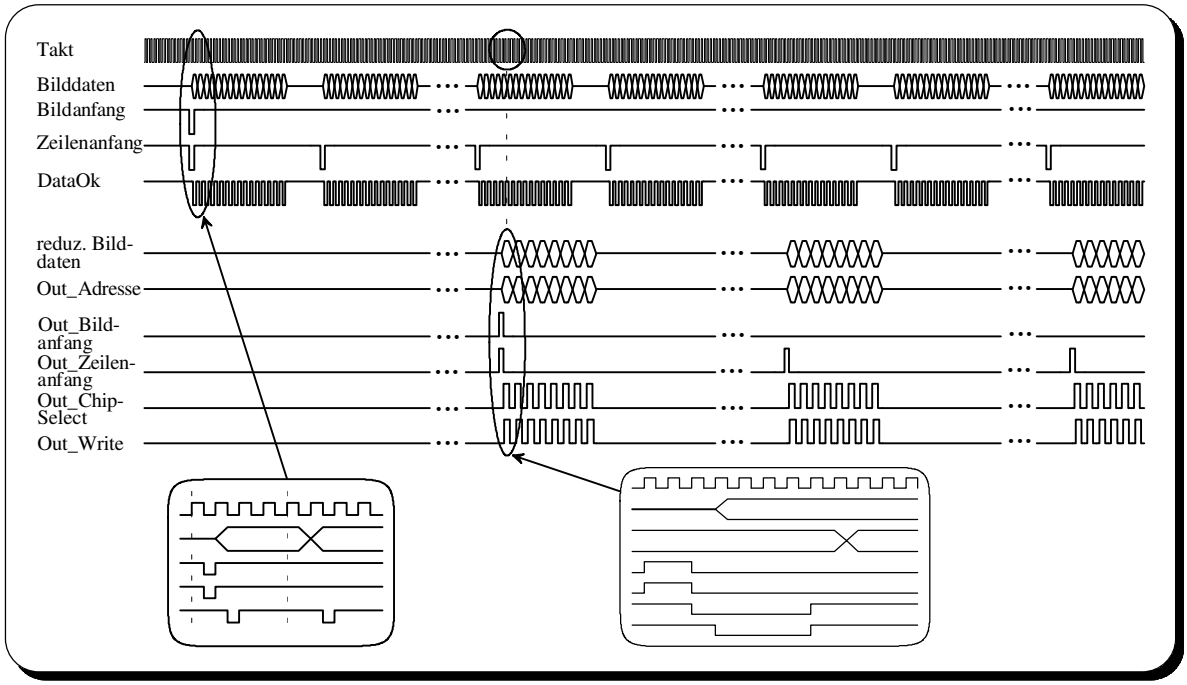


Bild 4.7: Timing Mosaik-FPGA

Das in Bild 4.1 dargestellte Gesamtsystem arbeitet komplett mit einem einheitlichen Systemtakt (typ. 32 MHz). Alle Signale in diesem System werden synchron zur steigenden oder fallenden Flanke des Systemtaktes aktiviert und deaktiviert.

Auf Bild 4.7 ist das Zusammenspiel der einzelnen Signale in Relation zum Systemtakt beim Mosaik-FPGA dargestellt. In der oberen Bildhälfte befinden sich die vom Digitalisiersystem gelieferten Signale, während im unteren Teil des Bildes die Ausgangssignale vom Mosaik-FPGA zu sehen sind.

Zu Beginn eines neuen Bildes aktiviert das Digitalisiersystem das low-aktive Signal *Bildanfang*. Im gleichen Takt oder auch wenige Takte später wird das low-aktive Signal *Zeilenanfang* aktiviert, um anzuzeigen, daß eine neue Zeile beginnt.

Die *Bilddaten* haben eine Rate von maximal 8 MHz. Gültige *Bilddaten* werden dabei durch das low-aktive Signal *DataOk* gekennzeichnet. Das Signal *DataOk* wird dabei mindestens einen Takt später aktiviert als das Signal *Zeilenanfang* (siehe Ausschnittvergrößerung links unten in Bild 4.7). Jeweils nach Ende einer Zeile werden für die Zeit des Strahlrücklaufs keine Bilddaten gesendet.

Nahezu identisch zum Timing der Signale vom Digitalisiersystem ist das Timing der Ausgangssignale vom Mosaik-FPGA, jedoch mit reduzierter Pixelrate. Nach Abschluß der Berechnung eines Quadrates (vgl. Bild 4.3) erfolgt die Ausgabe der reduzierten Bilddaten (*reduz. Bilddaten*). Gültige Daten sind hierbei durch das low-aktive Signal *Out_Write* gekennzeichnet. Drei Takte vor Beginn eines neuen reduzierten Bildes wird für zwei Takte das Signal *Out_Bildanfang* aktiviert. Gleichzeitig und ebenfalls zwei Takte lang wird ein neuer Zeilenanfang durch das Signal *Out_Zeilenanfang* signalisiert.

Zur Ansteuerung eines Bildspeichers für die reduzierten Bilddaten werden noch die Signale *Out_Adresse* und *Out_ChipSelect* generiert. Die Bildspeicheradresse *Out_Adresse* wird synchron zu den reduzierten Bilddaten ausgegeben. Das low-aktive Signal *Out_ChipSelect* wird einen Takt vor Beginn des Signals *Out_Write* aktiviert und bleibt so lange wie dieses aktiv.

4.4.4 FPGA-Architektur und Entwurfsprozess

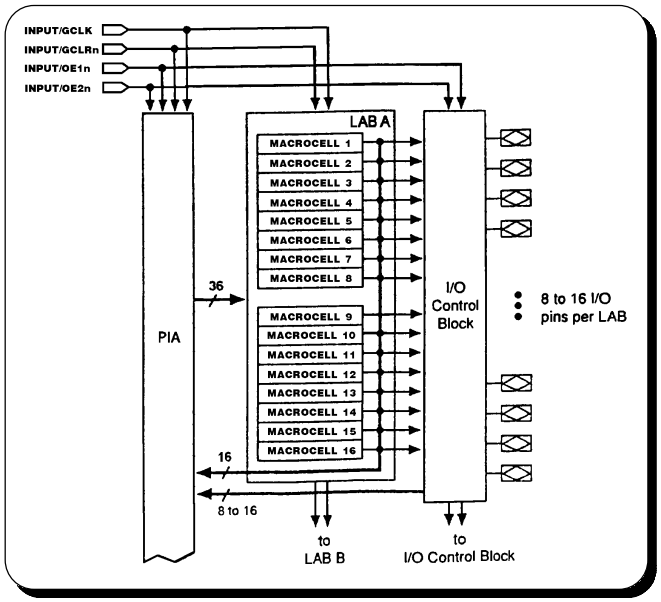


Bild 4.8: Architektur eines Altera FPGAs der MAX7000-Serie [Alt93]

Für die Realisierung des Mosaik-FPGA wurden FPGAs der MAX7000-Serie der Firma Altera verwendet. Die Bausteine dieser Serie besitzen alle den gleichen prinzipiellen Aufbau. Das wesentliche Element sind *Logic Array Blocks* (LABs), deren Struktur in Bild 4.8 dargestellt ist. In einem solchen LAB sind jeweils 16 Makrozellen, die die programmierbaren Einheiten des FPGA darstellen, zusammengefaßt. Jede dieser Makrozellen besitzt die in Bild 4.9 gezeigte

Struktur. Jedem LAB ist ein *I/O Control Block* mit 8 bis 16 Pins zugeordnet, d. h. es ist nicht jede Makrozelle mit einem Pin verbunden. Als Eingangssignale stehen jedem LAB insgesamt 36 Leitungen aus dem *Programmable Interface Array* (PIA) zur Verfügung. Zusätzlich zu vier globalen Signalen werden die Signale der Eingangspins in das PIA geführt.

Der in Bild 4.9 dargestellte Aufbau der Makrozellen zeigt, daß jede Makrozelle zwei wesentliche Blöcke besitzt. Im ersten Block können maximal dreistufige kombinatorische Logikfunktionen gebildet werden. Hierzu wird eine programmierbare UND-Matrix mit einer nachgeschalteten ODER-Stufe verwendet. Das Ausgangssignal der ODER-Stufe kann dann noch mit einem UND-Term XOR-verknüpft werden. Insgesamt stehen maximal fünf UND-Terme zur Verfügung.

Der zweite wesentliche Block ist ein D-Flip-Flop[1] mit asynchronen Setz- und Rücksetzeingängen. Da das Flip-Flop unmittelbar vor dem I/O-Pin liegt, ist es im Sinne geringer Signalverzögerungen besser, Moore-Automaten ohne Ausgangsschaltnetz zu realisieren, d. h. jeder Ausgangsvariablen wird ein Flip-Flop zugeordnet. Bei kleineren Automaten kann die Übergangsfunktion eines Flip-Flops dann vollständig innerhalb einer Makrozelle realisiert werden.

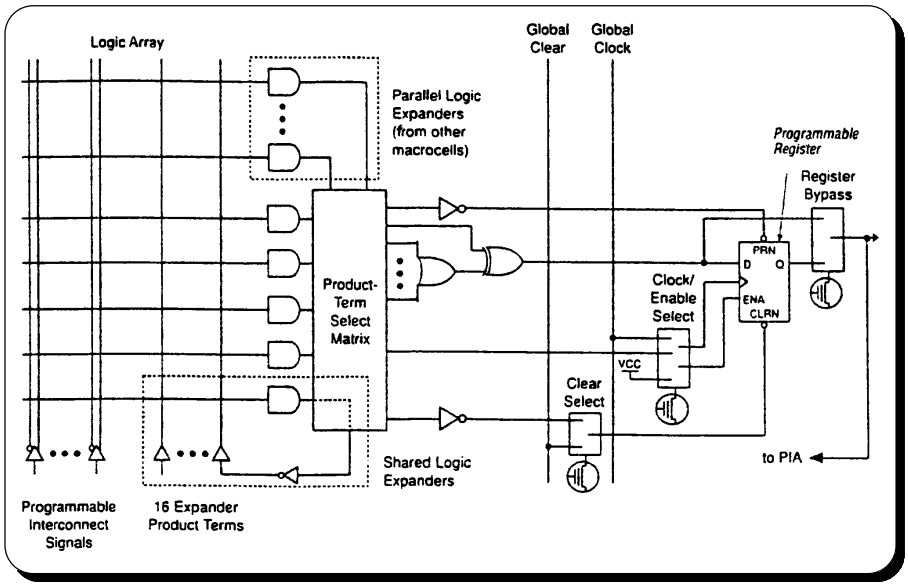


Bild 4.9: Aufbau einer Makrozelle der Altera MAX7000 FPGAs [Alt93]

Für das Mosaik-FPGA wurde der Typ EPM7128 verwendet. Diese enthalten insgesamt 128 Makrozellen, d. h. acht LABs. Die Verzögerungszeit einer Makrozelle beträgt bei diesem Typ je nach Ausführung 10, 12, 15 oder 20 ns. Da die gesamte Logik nicht in ein FPGA vom Typ EPM7128 hineinpaßte, wurden zwei FPGAs verwendet. Hierdurch konnte eine klare Trennung von Operationswerk und Steuerwerk vorgenommen werden. Versuchsweise wurde die gesamte Logik einmal auf den nächstgrößeren Typ EPM7256 abgebildet. Obwohl hier ausreichend Makrozellen (256) zur Verfügung stehen, konnte die Logik nicht partitioniert werden, da intern nicht ausreichend Verbindungsleitungen zwischen den LABs zur Verfügung standen. Daher wurde diese Lösung verworfen, zumal der Typ EPM7256 nur bis minimal 20 ns Verzögerungszeit statt der benötigten 12 ns zur Verfügung stand.

Das Operationswerk und das Steuerwerk des Mosaik-FPGA wurde in einer Hardwarebeschreibungssprache des FPGA-Chipherstellers (*AHDL - Altera Hardware Desrciption*

[1] *Flip-Flop* - digitales Kippglied mit mindestens zwei definierten Zuständen

Language) beschrieben. Diese ermöglicht eine *VHDL*[1]-ähnliche Beschreibung der im FPGA zu implementierenden Funktionen. Sie unterstützt dabei sowohl *Verhaltensbeschreibungen*[2] als auch *Strukturbeschreibungen*[3] bzw. eine Kombination von beiden, von digitalen Schaltnetzen und Schaltwerken. Zusätzlich existieren in dieser Sprache eine Reihe von Bibliotheksfunktionen, die z. B. Flip-Flops beschreiben. Weiterhin sieht AHDL eine komfortable Möglichkeit zur Beschreibung von endlichen Automaten vor. Arithmetische Funktionen wie z. B. die Addition zweier Zahlen lassen sich durch die einfache Schreibweise $a+b$ realisieren.

Die für das Mosaik-FPGA verwendete Entwicklungssoftware von Altera (MAX+plusII) vereinigt mehrere Teilsysteme:

- Design-Compiler für AHDL
- Logiksynthese und -minimierung
- Abbildung der Logikgleichungen auf das verwendete FPGA (*Mapping*)

Der Synthesealgorithmus erzeugt dabei eine der FPGA-Struktur angepaßte Schaltfunktion, die häufig keiner weiteren Nachbearbeitung bedarf. Im Falle der Addition entsteht in Abhängigkeit von der verwendeten Bitbreite meist eine gemischte Struktur aus ripple Carry[4] und Carry look-ahead[5].

4.4.5 Operationswerk

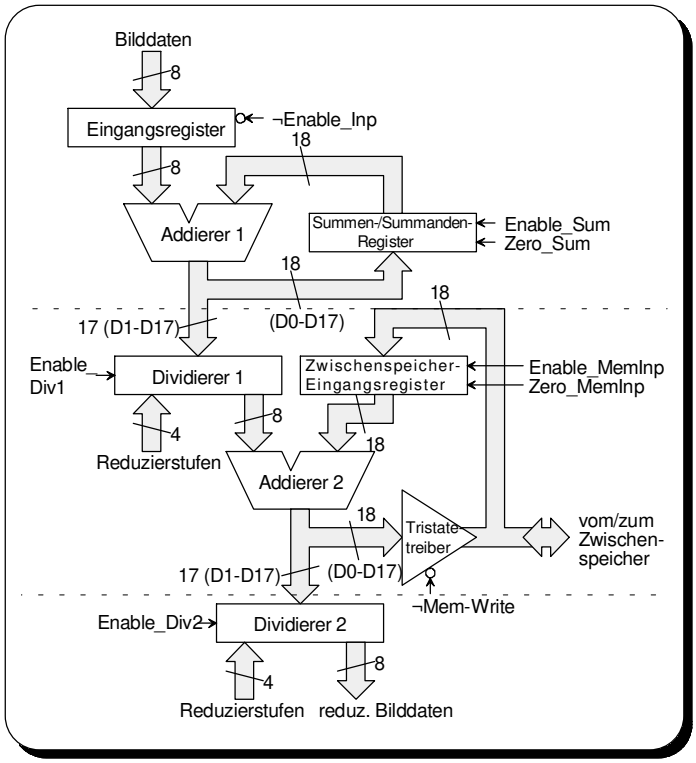


Bild 4.10: FPGA-Operationswerk

[1] *VHDL - Very High Speed Integrated Circuit Hardware Description Language* (siehe Kapitel 7)
[2] *Verhaltensbeschreibung* - Beschreibung der Funktion einer Schaltung ähnlich wie in einer höheren Programmiersprache
[3] *Strukturbeschreibung* - Beschreibung der Art, wie die Elemente einer Schaltung verbunden sind ("Gatternetzliste")
[4] *ripple Carry* - durchlaufender Übertrag, d. h. jede Stufe muß auf den Übertrag von der vorigen Stufe warten
[5] *Carry look-ahead* - vorausschauender Übertrag, d. h. eine zusätzliche Logik ermittelt alle Überträge im voraus

Das Operationswerk (Bild 4.10) besteht aus drei Funktionsblöcken:

- **Block 1 mit *Eingangsregister*, *Addierer 1* und *Summen-/Summandenregister***

  Dieser Block ist für die Aufaddierung der Pixelwerte einer Zeile eines zusammenzufassenden Quadrates zuständig.

- **Block 2 mit *Dividierer 1*, *Zwischenspeichereingangsregister*, *Addierer 2* und *Tristatetreiber***

  In diesem Block wird das Ergebnis der Addition aus Block 1 durch die Anzahl der Pixel geteilt, zum Inhalt des Zwischenspeichers addiert und das Ergebnis davon über den Tristatetreiber wieder in den Zwischenspeicher zurückgeschrieben.

- **Block 3 mit *Dividierer 2***

  Dieser letzte Funktionsblock ist für die abschließende Division des Zwischenspeicherinhalts sowie die Ausgabe der reduzierten Bilddaten zuständig.

Die einzelnen Elemente des Operationswerkes wurden ohne zusätzliche Optimierungen von der Entwicklungssoftware synthetisiert. Die besonders zeitkritische Addition in *Addierer 1* wurde dabei dreistufig realisiert, d. h. es wurden maximal drei Makrozellen hintereinandergeschaltet, wodurch sich als maximale Verzögerungszeit für die Addition die dreifache Verzögerungszeit einer Makrozelle ergibt.

Die beiden Dividierer wurden jeweils als Barrel-Schieberegister[1] (*Barrelshifter*) realisiert (siehe Bild 4.11). Aus Geschwindigkeitsgründen war dies zwar nicht zwingend erforderlich, jedoch nach Abwägung des Platzbedarfs einer Barrelshiftlogik und eines zusätzlich notwendigen Steuerautomaten für ein einfaches Schieberegister wurde dem Barrelshifter der Vorzug gegeben. Betrachtet man den Aufbau einer Makrozelle in Bild 4.9, so wird die Entscheidung deutlich. Als Register werden die Flip-Flops der Makrozellen verwendet, während der Multiplexer (*MUX 10 zu 1*) in derselben Makrozelle im Logikbereich realisiert wird.

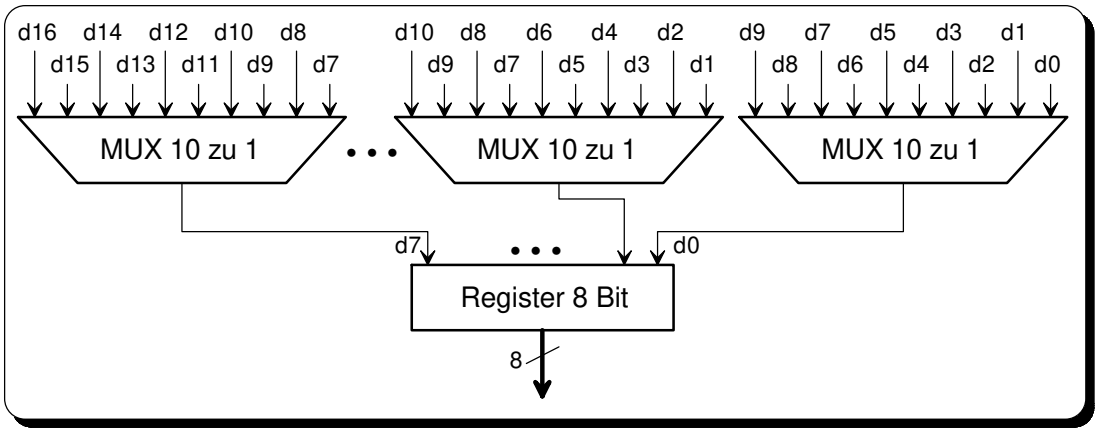


Bild 4.11: Realisierung der Dividierer als Barrelshifter

4.4.6 Steuerwerk

Analog zu den drei Funktionsblöcken des Operationswerkes ist auch das Steuerwerk (Bild 4.12) in mehrere Funktionsblöcke unterteilt. Auf der linken Seite der einzelnen Stufen

[1] *Barrel-Schieberegister* - Schiebregister, das eine Verschiebung um mehr als ein Bit in einem Takt ermöglicht

befinden sich die Signale, die vom Operationswerk geliefert werden, sowie die Eingangssignale des Mosaik-FPGA. Auf der rechten Seite sind die Steuersignale an das Operationswerk und die Ausgangssignale des Mosaik-FPGA dargestellt. In vertikaler Richtung verlaufende Signale dienen zum Aktivieren der jeweils nächsten Stufe.

Die ersten drei Stufen kontrollieren die drei Blöcke des Operationswerkes, d. h. *Stufe 1* ist für *Block 1*, *Stufe 2* für *Block 2* und *Stufe 3* für *Block 3* zuständig. Alle drei Stufen arbeiten mit dem vollen Systemtakt und werden jeweils von der vorhergehenden Stufe aktiviert. Somit ist eine voll parallele Arbeitsweise der einzelnen Stufen gewährleistet. Dieses ist notwendig, da das Operationswerk eine *pipelineartige*[1] Struktur aufweist. Dies bedeutet, daß ein vorhergehender Block mit einer neuen Berechnung beginnen kann, während der nachfolgende Block seine Berechnung erst beginnt. Hierdurch kann eine wesentlich höhere Arbeitsgeschwindigkeit erzielt werden.

Von *Stufe 4* werden die Signale zur Ansteuerung eines externen Bildspeichers bereitgestellt. Diese Stufe arbeitet ebenfalls mit dem vollen Systemtakt und wird wie die Stufen 1 bis 3 von der vorhergehenden Stufe gestartet.

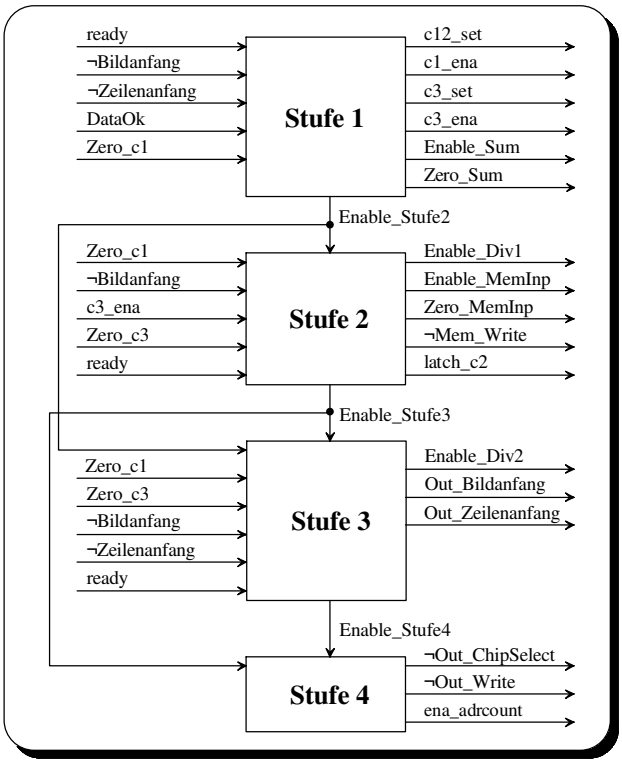


Bild 4.12: FPGA-Steuerwerk (Steuerautomaten)

Weiterhin benötigt das Steuerwerk als Zustandsinformation drei Zähler (Bild 4.13). Die Zähler *c1* und *c3* zählen dabei die Pixel innerhalb eines zusammenzufassenden Pixelquadrates, wobei *c1* die Anzahl der Spalten des Pixelquadrates zählt und *c3* die Anzahl der Zeilen. Beide Zähler besitzen eine Breite von 10 Bit. Ihr Wert wird allein durch die Anzahl der erforderlichen Reduzierstufen bestimmt, wobei gilt:

$$c1,c3=2^R-1, \text{ mit } R=1,2...10.$$

Es sei hier noch einmal daran erinnert, daß die zusammenzufassenden Pixelquadrate eine Kantenlänge von 2^R-Pixeln haben.

[1] *engl. pipeline* ≈ Rohrleitung

Der Zähler *c2* dient zur Erzeugung der Adresse für den Zwischenspeicher. Analog dazu dient der Zähler *address_count* zur Erzeugung der Adresse für einen externen Bildspeicher.

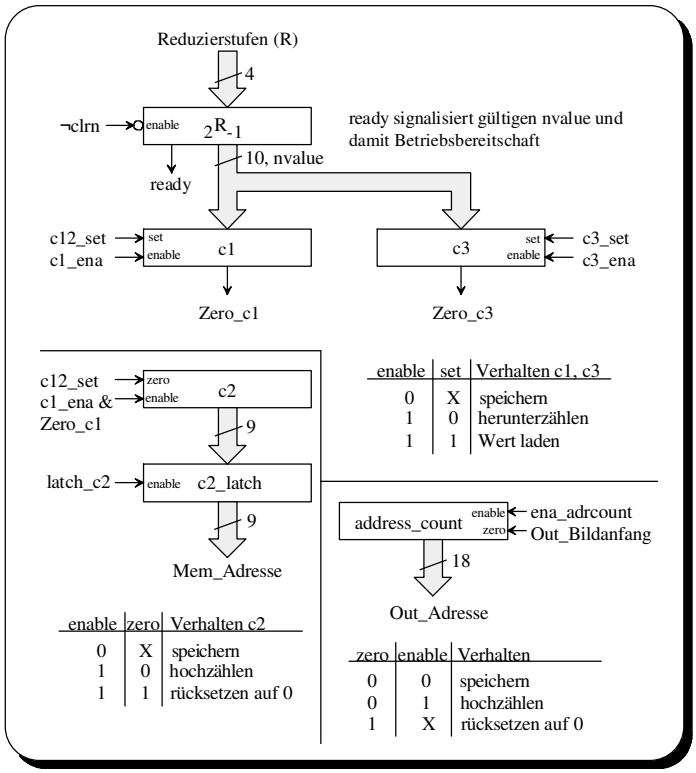


Bild 4.13: FPGA-Steuerwerk (Zähler)

5 Spezifikation und Lösungsansätze

In diesem Kapitel wird die der vorliegenden Diplomarbeit zugrunde liegende Problemstellung mit den sich daraus ergebenden Anforderungen an den NeNEB-Chip dargestellt. Des weiteren erfolgt die Darstellung einiger möglicher Realisierungsmöglichkeiten der Aktivierungsfunktion. Abschließend wird von Experimenten zur Bestimmung der erforderlichen Rechengenauigkeiten des neuronalen Netzwerkes berichtet.

5.1 Problemstellung

Ausgehend von dem in Kapitel 1 dargestellten Gesamtsystem zur Echtzeit-Klassifikation von Bildern, liegt der Schwerpunkt dieser Arbeit in der Hardware-Realisierung eines neuronalen Netzwerkes. Kennzeichnend ist dabei, daß die Bilder nicht als Ganzes vorliegen, sondern jeweils zeilenweise Pixel für Pixel vom Digitalisiersystem geliefert werden. Dies ist durch die Funktionsweise einer Kamera bedingt, die jeden Bildpunkt einzeln abtasten muß. Aufgrund des seriellen Pixeldatenstroms macht es daher keinen Sinn, das neuronale Netzwerk hochparallel zu realisieren. Es ist also nicht erforderlich, daß das Netzwerk in hoher Geschwindigkeit die Ausgabe für den gesamten Eingabevektor (bzw. das gesamte Bild) berechnet. Statt dessen genügt es, wenn die Berechnung des Anteils eines Pixelwertes für die Gesamterregung für alle Neuronen einer Schicht parallel erfolgt, die Berechnung für die einzelnen Pixelwerte jedoch seriell. Die Zeit, die für diese Berechnung zur Verfügung steht, entspricht dem reziproken Wert der Pixelfrequenz, d. h. der Zeit zwischen dem Abtasten zweier Pixel.

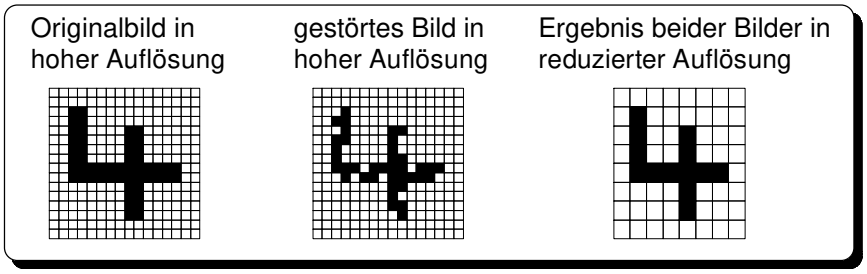


Bild 5.1: Vergleich von Bildern unterschiedlicher Auflösung

Neuronale Netzwerke haben die Eigenschaft, einem gelernten Muster ähnliche Muster zu erkennen, d. h. das zu erkennende Muster unterscheidet sich nur in wenigen Bildpunkten vom gelernten Muster. Bis zu einem bestimmten Grad, der problemabhängig ist, ist es daher möglich, die Bildauflösung zu reduzieren. Hierdurch verringern sich Unterschiede, die zwischen zwei Mustern bei hoher Bildauflösung noch erkennbar sind, unter Umständen derart, daß diese bei geringerer Bildauflösung nicht mehr erkennbar sind (Bild 5.1). Dies bedeutet, daß die Toleranz des Netzwerkes gegenüber nicht gelernten Mustern durch Verringern der Bildauflösung

erhöht werden kann. Die minimale Bildauflösung wird dabei durch die Art und die Anzahl der zu klassifizierenden Muster festgelegt. Bezogen auf die Richtungserkennung von Fahrbahnen oder Eisenbahnschienen (vgl. Kapitel 1) bedeutet dies, daß die Fahrbahn nur als Gesamtes erkennbar sein muß, nicht jedoch ihre genaue Struktur (z. B. kleine Schlaglöcher, die im Bild dunkler erscheinen). Dies erklärt die Entwicklung des im vorigen Kapitel beschriebenen Mosaik-FPGA zur Auflösungsreduktion, das für eine quadratische Region von Pixeln das arithmetische Mittel der Grauwerte dieser Region ermittelt, welches dann als reduziertes Pixel interpretiert wird.

Neben der Reduktion der Anzahl der Bildpunkte und somit der erforderlichen Eingänge des neuronalen Netzwerkes bedeutet eine Auflösungsreduktion auch eine Verringerung der Pixelfrequenz. Bei der im vorigen Kapitel beschrieben Mosaikbildung reduziert sich die Pixelfrequenz für jede Reduzierstufe um den Faktor 4, d. h. eine Reduktion der Auflösung von 512*512 Pixel auf 256*256 Pixel bei jeweils 50 Bildern pro Sekunde läßt die Pixelfrequenz von

$$f_1 = 512 \cdot 512 \cdot 50 \,\frac{1}{\mathrm{s}} = 13,1072 \text{ MHz}$$

auf

$$f_2 = \frac{1}{2} \cdot 512 \cdot \frac{1}{2} \cdot 512 \cdot 50 \frac{1}{\mathrm{s}} = 256 \cdot 256 \cdot 50 \,\frac{1}{\mathrm{s}} = 3,2768 \text{ MHz} = \frac{1}{4} \cdot f_1$$

sinken. Für das neuronale Netzwerk bedeutet dies aber auch eine Erhöhung der zur Verfügung stehenden Rechenzeit um den Faktor 4, d. h. mit einer leichten Auflösungsreduktion läßt sich das Bildformat einer Kamera bzw. des Digitalisiersystems an die Geschwindigkeit und die Kapazität des neuronalen Netzwerkes anpassen.

Aus dem Vorangegangen ist erkennbar, daß für die im Rahmen dieser Diplomarbeit erfolgte Hardware-Realisierung des neuronalen Netzwerkes nicht unbedingt Geschwindigkeitsgründe, d. h. eine signifikante Beschleunigung gegenüber einer Softwarelösung auf einer Workstation, im Vordergrund standen, sondern vor allem die Schaffung eines kompakten Klassifikationssystems, das sich, dank weniger Hardwarekomponenten, direkt in einer Kamera unterbringen läßt. Daher erfolgt auch nur die Implementation der Ausführungsphase (*feed forward*, vgl. Kapitel 2) eines neuronalen Netzwerkes, während das Trainieren des Netzwerkes zur Bestimmung der synaptischen Gewichte separat auf einer Workstation erfolgt, da hierfür ein wesentlich größerer Rechenaufwand (ca. Faktor 3) sowie größere Rechengenauigkeiten erforderlich sind. Das Trainieren des Netzwerkes auf der Workstation bietet zudem den Vorteil, daß die Trainingsdatenmenge, d. h. die Menge verschiedener Bilder, einen wesentlich größeren Umfang haben kann, als dies bei einem Trainieren in Echtzeit im Pixeldatenstrom möglich wäre. Zudem können zwecks Lernens aufgenommene Bilder geeignet variiert werden, damit vom Netzwerk eine größere Vielfalt an Bildern gelernt werden kann (vgl. *structured noise* in Kapitel 3.1 und [Pom93]).

5.2 Anforderungen an den NeNEB-Chip

In diesem Kapitel werden die Anforderungen an den NeNEB-Chip aufgezeigt. Hierzu gehören neben der Wahl eines geeigneten Netzwerktyps und der damit zu implementierenden Berechnungsfunktionen sowie der Netzwerk-Architektur auch die Festlegung der erforderlichen Ein-

und Ausgabeschnittstellen. Ebenso werden die Leistungsanforderungen wie minimale Eingangsbildgröße und Bildfrequenz dargelegt.

5.2.1 Wahl des Netzwerktyps

Wie der Motivation in Kapitel 1 und der Beschreibung in Kapitel 3 zu entnehmen ist, existiert in den USA ein Forschungsprojekt namens ALVINN. Hierbei wird erfolgreich die Ausgabe eines relativ kleinen Backpropagation-Netzwerkes dazu verwendet, ein Fahrzeug einem beliebigen Fahrbahnverlauf folgen zu lassen. Als Eingabe erhält das Netzwerk ausschließlich Bildinformationen des Fahrbahnverlaufs. Eine ähnliche Anwendung wurde für das in Kapitel 1 geschilderte Gesamtsystem zur Echtzeitbildklassifikation vorgesehen, wobei es sich hierbei nicht um die Erkennung eines Straßenverlaufs wie bei ALVINN handelt, sondern um die Erkennung des Verlaufs bzw. des Zustands (belegt, frei) von Eisenbahnstrecken.

Durch ALVINN motiviert, wurde für das zu implementierende Netzwerk ein Backpropagation-Netzwerk ausgewählt. Ein Vorteil eines BPG-Netzwerkes ist die gute Eignung für Aufgaben aus dem Bereich der Mustererkennung. Dies resultiert aus der Tatsache, daß jedes Neuron seinen Eingaberaum linear teilt, wodurch auch der Eingaberaum des Netzwerkes linear durch Hyperebenen geteilt wird. Hiermit können bestimmte Gruppen von Mustern, die für das Netzwerk eine Menge von Vektoren sind, in verschieden Klassen geteilt werden.

Ein weiterer Vorteil von Backpropagation-Netzwerken ist, daß hierfür ein allgemeingültiger Lernalgorithmus für beliebig große Netze existiert. Bei vielen Problemen weist dieser Algorithmus ein gutes Konvergenzverhalten auf. Zur Verbesserung der Konvergenzgeschwindigkeit sind darüber hinaus zahlreiche Abwandlungen des einfachen Backpropagation-Algorithmus entwickelt worden (siehe Kapitel 2 und [Roj93]).

5.2.2 Anforderungen an die Netzwerkarchitektur

In Anlehnung an ALVINN wird die Realisierung eines zweischichtigen Netzwerkes für ausreichend erachtet. Für die verborgene Schicht sollen mindestens 4, aus Flexibilitätsgründen besser 8 Neuronen vorgesehen werden. In der Ausgabeschicht sollen mindestens 8 Neuronen zur Verfügung stehen. In der Ausführungsphase sind für ein Backpropagation-Netzwerk drei Berechnungsfunktionen notwendig:

1. Multiplikation zur Berechnung der Teilerregung des Neurons: $\omega_{ij} \cdot x_j$
2. Addition zur Berechnung der Gesamterregung des Neurons: $\sum_j (\omega_{ij} \cdot x_j)$
3. Berechnung der Aktivierungsfunktion: $g(x)$.

Diese drei Funktionen sind für jedes Neuron geeignet zu implementieren, wobei alle drei Funktionen für jedes Neuron sowohl der verborgenen als auch der Ausgabeschicht gleich sein sollen. Die Art der Aktivierungsfunktion ist nicht festgelegt, soll aber nach Möglichkeit der stetigen Sigmoidfunktion ähnlich sein.

5.2.3 Schnittstellen und Datenformate

- **Bilddatenschnittstelle**

  Die Bilddaten werden von einem 8 Bit breiten Datenbus bereitgestellt. Die 256 möglichen Graustufen werden von den Neuronen als Eingaben normiert auf den Bereich $[0...(1-2^{-8})]$ interpretiert, d. h.

  $$\begin{array}{lcll} \text{Schwarz} & \triangleq & 00000000_2 & \triangleq & 0{,}0_{10}{}^{1} \\ & & 00000001_2 & \triangleq & 0{,}00390625_{10} \\ \text{Weiß} & \triangleq & 11111111_2 & \triangleq & 0{,}99609375_{10}. \end{array}$$

  Gültige Bilddaten werden durch ein taktsynchrones Gültigkeitssignal (*Strobe*) gekennzeichnet. Zusätzlich stehen zwei weitere taktsynchrone Signale zur Verfügung, die einen Bildanfang und einen Zeilenanfang kennzeichnen. Der Gesamtsystemtakt steht zur Verfügung.

- **Ausgabeschnittstelle**

  Die Ausgaben der Neuronen sollen im selben Datenformat wie die Eingaben erfolgen, d. h. normiert auf den Bereich $[0...(1-2^{-8})]$ mit 8 Bit. Die einzelnen Ausgabewerte sollen wahlfrei abgerufen werden können.

- **Programmierschnittstelle**

  Die Programmierschnittstelle zur Initialisierung des Chips mit den erforderlichen synaptischen Gewichten und anderen Parametern wurde zunächst nicht spezifiziert.

Weitere Anforderungen hinsichtlich der Datenformate bzw. der erforderlichen Bitbreiten wurden, wie in Kapitel 5.4 beschrieben wird, experimentell ermittelt.

5.2.4 Leistungsanforderungen

- Der NeNEB-Chip soll echtzeitfähig sein, d. h. die Berechnungen müssen im seriellen Pixeldatenstrom durchgeführt werden, ohne daß ein Bild zwischengespeichert werden muß.
- Alle Neuronen einer Schicht sollen ihre Berechnungen parallel ausführen.
- Die Berechnung der Ausgabeneuronen kann erfolgen, nachdem die Berechnung der verborgenen Neuronen abgeschlossen ist, d. h. nach einem kompletten Bild.
- Das Netzwerk soll eine Eingangsbildgröße von mindestens 32*32 Pixeln verwenden, wobei für diese Bildgröße mindestens 50 Bilder pro Sekunde zu erwarten sind.
- Alle synaptischen Gewichte des Netzwerkes müssen frei programmierbar sein.
- Es sollen verschiedene Steigungen der Aktivierungsfunktion gewählt werden können.
- Aufgrund der zu erwartenden großen Anzahl synaptischer Gewichte für die verborgene Schicht und der damit verbunden großen (teuren) Chipfläche sollen diese Gewichte in einem externen Speicher untergebracht werden.

[1] Die Darstellung xy_n bedeutet die Zahl xy zur Basis n, wobei n=2 das Binärsystem und n=10 das Dezimalsystem kennzeichnet.

5.3 Realisierungsmöglichkeiten der Aktivierungsfunktion

Zunächst wurden Überlegungen zu Realisierungsmöglichkeiten der in Kapitel 2 angeführten Sigmoidfunktion (2.8 und 2.17)

$$g(x) = \frac{1}{1 + e^{-cx}} \tag{5.1}$$

angestellt. Eine Möglichkeit wäre die Realisierung mittels *look up table*[1], wobei alle Ergebniswerte für ein bestimmtes Intervall von x in einem Speicher stehen. Dieser Speicher wird dann durch die x-Werte adressiert. Bei einem vorgesehenen Intervall der x-Werte von [-8..+8] mit 7 binären Nachkommastellen wären insgesamt 1024 Worte[2] bei einer Speicherwortbreite von 8 Bit notwendig. Da alle Neuronen parallel rechnen, müßte für jedes Neuron eine eigene *look up table* bereitgestellt werden. Diese Lösung wurde daher aufgrund des zu erwartenden großen Flächenbedarfs der *look up tables* verworfen.

Als weiterer Lösungsansatz wurde die Sigmoidfunktion durch eine Taylorreihe $t(x)$ angenähert. Um der Nichtlinearität der Sigmoidfunktion Rechnung zu tragen, sind mindestens drei Glieder der Taylorreihe notwendig (Herleitung siehe Anhang A.3):

$$t(x) = \frac{1}{2} + \frac{1}{4} \cdot c \cdot x - \frac{1}{48} \cdot c^3 \cdot x^3 \ . \tag{5.2}$$

Mit diesen drei Gliedern konnte jedoch keine ausreichende Annäherung an die Sigmoidfunktion im vorgesehenen Intervall erreicht werden (siehe Bild 5.2). Somit ist mindestens ein weiteres Glied der Taylorreihe notwendig. Bei jedem weiteren Glied der Taylorreihe nimmt die Potenz von x jedoch um zwei zu, da nur ungerade Potenzen in der Reihenentwicklung enthalten sind. Dieser Lösungsansatz wurde daher ebenfalls verworfen, da hierfür zur Berechnung der Potenzen aufwendige Multiplikationen erforderlich gewesen wären, die in keinem Verhältnis zum restlichen Berechnungsaufwand der Neuronen gestanden hätten.

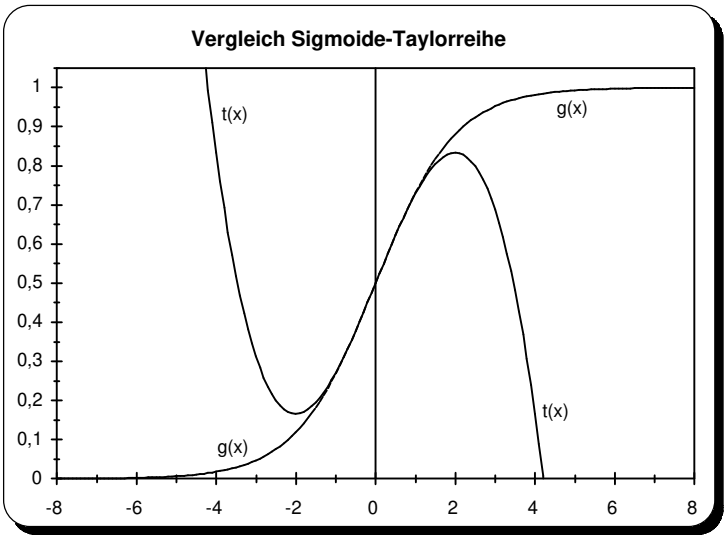


Bild 5.2: Taylor-Reihenentwicklung der Sigmoidfunktion nach (5.2) mit $c = 1$

Letztendlich fiel die Wahl auf eine Annäherung der Sigmoidfunktion durch drei Geraden (Bild 5.3). Dieser Lösungsansatz wurde unter anderem in [Clo94] untersucht, wobei

[1] *engl. look up table* ≈ Nachschlagetabelle
[2] $1024 = 2^{10}$, 10 = 3 Vorkommastellen + 7 Nachkommastellen

festgestellt wurde, daß die Näherung keinen nennenswerten Einfluß auf die Erkennungsrate des Netzwerkes hat. Selbst beim Trainieren eines Backpropagation-Netzwerkes kann die Geradenannäherung ohne nennenswerte Einbußen beim Konvergenzverhalten verwendet werden. Diese Ergebnisse wurden durch eigene Experimente bestätigt.

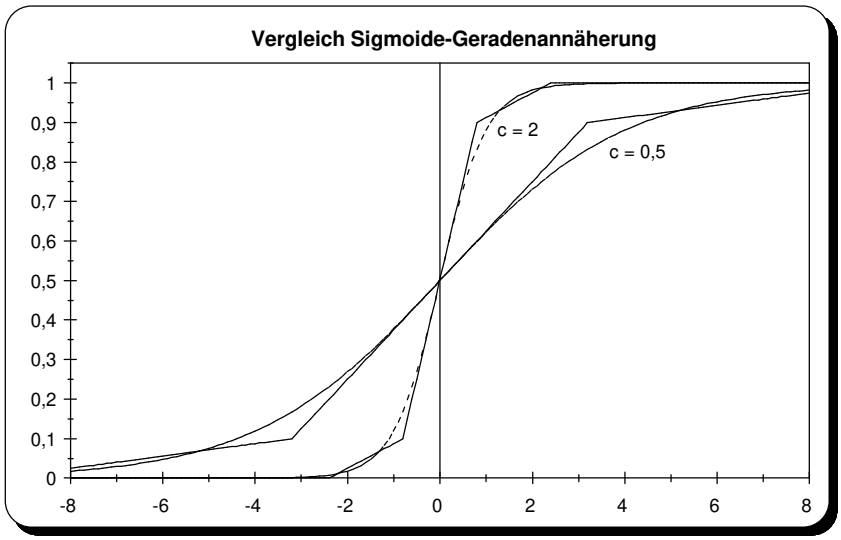


Bild 5.3: Vergleich Sigmoidfunktion - Geradenannäherung

Die Geradenannäherung bietet einen guten Kompromiß zwischen Geschwindigkeit und Aufwand der Berechnung. Zur Berechnung einer Gerade ist nur eine Multiplikation und eine Addition erforderlich. Über zwei Komparatoren[1] läßt sich anhand der x-Werte prüfen, welche der drei Geraden berechnet werden muß. Die Berechnung einer Gerade erfolgt nach der bekannten Geradengleichung

$$y = m \cdot x + b \tag{5.3}$$

wobei m die Steigung der Gerade bedeutet und b den Schnittpunkt der Gerade mit der y-Achse. Ein *clipping*[2]-Mechanismus sorgt dafür, daß y nicht kleiner als Null bzw. größer als Eins werden kann.

Die Multiplikation läßt sich entscheidend vereinfachen, wenn für die Steigung m nur Zweierpotenzen 2^n verwendet werden. In diesem Fall läßt sich die Multiplikation durch eine Schiebeoperation ersetzen, wobei je nach Vorzeichen von n eine Verschiebung des Operanden x nach links oder rechts um $|n|$-Schritte erfolgt. Der Nachteil dieser Lösung ist, daß bei einer Einschränkung der Steigung der Geraden auf Zweierpotenzen nur eine geringe Anzahl von Sigmoidfunktionen angenähert werden können. Es sei jedoch hier noch einmal erwähnt, daß die Lernparameter beim Trainieren des Netzes, wie die Lernkonstante und die Steigung der Sigmoidfunktion, je nach Problemfall empirisch bestimmt werden, wobei stets gewisse Freiheiten existieren. Diese Freiheiten führen dazu, daß sich, wie durch Simulation feststellbar, bei einer geringfügig größeren oder kleineren Steigung der Sigmoidfunktion keine wesentlichen Einflüsse auf die Konvergenz des Backpropagation-Algorithmus ergeben. Wichtig ist jedoch, daß sowohl beim Lernen als auch beim Ausführen des Netzwerkes eine Sigmoidfunktion mit gleicher Steigung verwendet wird.

[1] *Komparator* - Vergleicher, eingedeutscht von *engl. compare* ≈ vergleichen
[2] *engl. clipping* ≈ Abschneiden

5.4 Parameterbestimmung

Vor der eigentlichen Implementation des NeNEB-Chips wurden zunächst einige Simulationen durchgeführt. Diese hatten zum Ziel, die erforderlichen Rechengenauigkeiten und Zahlendarstellungen zu ermitteln. Aufgrund der geforderten Echtzeitberechnung des neuronalen Netzwerkes kam eine Gleitkommaarithmetik wegen des großen Implementationsaufwandes und der damit benötigten großen Chipfläche nicht in Betracht. Es war daher notwendig, alle Rechenfunktionen in Festkommaarithmetik mit begrenzter Stellenanzahl zu implementieren, wobei reine binäre Arithmetik verwendet wird.

Für die Eingangswerte des neuronalen Netzwerkes ist die Zahlendarstellung sowie die Stellenanzahl durch die Spezifikation von vornherein festgelegt (siehe 5.2.3). Hierfür werden nur vorzeichenlose Werte zwischen 0 und 1 verwendet, wobei nur die Nachkommastellen mit 8 Bit kodiert werden, da die Werte stets kleiner als 1 sind.

Für die Darstellung der Gewichte sind in der Literatur ([Sak93], [Rey91], [Clo94]) unterschiedliche Angaben zu finden, wobei aufgrund der Konvergenzprobleme beim Lernen stets einige Stellen mehr als beim Ausführen benötigt werden. In [Rey91] werden für ausreichende Konvergenz des Lernvorgangs mindestens 14 Binärstellen empfohlen. In [Clo94] werden für die Gewichte Festkommazahlen mit 16 Stellen sowie 12 Nachkommastellen verwendet. Dabei ist zu beachten, daß bei sehr vielen Gewichten bzw. sehr vielen Eingabestellen für ein Neuron (max. 65536 beim NeNEB-Chip) die Werte der einzelnen Gewichte relativ klein werden, da jeder Eingabewert nur einen kleinen prozentualen Anteil an der Gesamterregung des Neurons hat. Dies erfordert daher eine höhere Genauigkeit bzw. mehr Nachkommastellen für die Gewichte auch für die Ausführungsphase des Netzwerkes.

Um die Erkennungsrate eines BPG-Netzwerkes mit begrenzter Stellenzahl bei Verwendung von Festkommaarithmetik zu testen, wurden mit Hilfe einer Videokamera von einer am Arbeitsbereich installierten Modelleisenbahnanlage einige Bilder aufgenommen. Es wurden dabei einige Bilder eines geraden freien Streckenabschnitts sowie einige Bilder des gleichen Strekkenabschnitts mit zwei verschiedenen fahrenden Loks aufgenommen. Die Videobilder wurden anschließend am Arbeitsbereich "Kognitive Systeme" des Fachbereichs Informatik der Universität Hamburg mit einer Größe von 256*256 Punkten digitalisiert. Da ein Videobild wie ein Fernsehbild aus zwei Halbbildern besteht, die zeitlich in kurzem Abstand aufeinanderfolgen, war in den digitalisierten Bildern der Zeitversatz zwischen beiden Halbbildern erkennbar. Es wurde daher mit Hilfe eines Konvertierungsprogrammes jedes digitalisierte Bild wieder in die ursprünglichen zwei Halbbilder aufgespalten, wobei zur Erhaltung der originalen Bildgröße jede Zeile verdoppelt wurde (Programm *interla*). Anschließend wurden die Bilder mit einem Programm zum Histogrammausgleich (Programm *hist*), zur Erzielung eines gleichmäßigen Helligkeitsniveaus aller Bilder, bearbeitet. Dabei wurde das in Kapitel 3 geschilderte, bei ALVINN verwendete Verfahren angewendet. Im Anschluß daran wurden die Bilder mit Hilfe des beim Mosaik-FPGA (siehe Kapitel 4) verwendeten Auflösungsreduktions-Algorithmus auf eine Bildgröße von 32*32 Punkten gebracht (Programm *mosaik*).

Mittels eines Softwareprogramms zur Simulation des Backpropagation-Algorithmus (*bpg*) wurde dann ein zweischichtiges Netzwerk mit 4 verdeckten Neuronen und zwei Ausgabeneuronen darauf trainiert, zwischen Bildern mit freier Strecke und Bildern mit besetzter Strecke zu unterscheiden. Dabei wurde für die Ausgabeneuronen eine binäre 1-aus-N-Codierung (siehe Kapitel 3) gewählt, wobei ein Ausgabewert kleiner als 0,5 als logische 0 bzw. inaktiv und ein Ausgabewert größer oder gleich 0,5 als logische 1 bzw. aktiv, definiert wurde.

Das Simulationsprogramm verwendet für die Berechnungen ausschließlich Gleitkommaarithmetik und ist in der Lage, eine sogenannte Netzdatei zu erzeugen, in der alle Daten des trainierten Netzwerkes inklusive der Gewichte enthalten sind. Diese Netzdatei wurde dann in ein eigenes Simulationsprogramm (*bpgsim*) eingelesen. Dieses Programm berechnet die Ausführungsphase des in der Netzdatei beschriebenen Netzwerkes, wobei alle internen Berechnungen mit einer wählbaren binären Nachkommastellenanzahl erfolgen. In *bpgsim* können für die Gewichte und die Signale, d. h. die Ein- bzw. Ausgabewerte der Neuronen, getrennte Nachkommastellenanzahlen angegeben werden. Ebenso können in *bpgsim* zwei verschiedene Aktivierungsfunktionen, die Sigmoidfunktion und die im vorhergehenden Abschnitt beschriebene Geradenannäherung, gewählt werden.

Nach erfolgreichem Training des Netzwerkes wurde das Netzwerk mit unbekannten Bildern sowohl mit *bpg* als auch mit *bpgsim* berechnet und die Ergebnisse verglichen. In *bpgsim* wurde dabei mit beiden Aktivierungsfunktionen und verschiedenen Genauigkeiten, d. h. verschiedenen Nachkommastellenanzahlen für die Gewichte und die Signale, gearbeitet. Auf diese Weise war ein Vergleich der Erkennungsrate des Netzwerkes bei Berechnung mit Gleitkommaarithmetik und mit Festkommaarithmetik möglich. Es zeigte sich hierbei, daß bei genügend vielen Nachkommastellen (16 Nachkommastellen für die Gewichte, 8 für die Signale) die gleiche Erkennungsrate mit Festkommaarithmetik wie mit Gleitkommaarithmetik erzielt werden kann. Erst bei stark reduzierter Stellenanzahl (9 Nachkommastellen für die Gewichte, 6 für die Signale) traten bei kritischen Bildern Erkennungsprobleme auf. Unter kritischen Bildern wurden dabei solche Bilder verstanden, die bei einer Berechnung mit *bpg* gerade noch richtig erkannt wurden, d. h. deren Ausgabewerte in der Nähe von 0,5 waren. Hierbei kam es oft zum "Umkippen" der Ausgabeneuronen, d. h. war die Ausgabe mit *bpg* noch (1,0), so wurde von *bpgsim* (0,1) ausgegeben. Tabelle 5.1 veranschaulicht dieses Verhalten. Es sind dort die Ausgaben des Simulationsprogramms *bpgsim* angegeben, wobei für jedes Neuron sowohl die Erregung als auch die Ausgabe angegeben wird. Dabei werden zunächst die verdeckten Neuronen (*H x*) und dann die Ausgabeneuronen (*O x*) ausgegeben, wobei zur Kontrolle mit dem NeNEB-Chip die entsprechenden Hexadezimalwerte mit angegeben werden.

	Gewichte: 12 Nachkommastellen Signale: 8 Nachkommastellen				**Gewichte: 9 Nachkommastellen Signale: 6 Nachkommastellen**			
	Erregung (*E*)		Ausgabe (*y*)		Erregung (*E*)		Ausgabe (*y*)	
Sigmoid-funktion	H 1 0.09375000	018	0.54296875	8B	H 1 -0.03125000	FF8	0.48437500	7C
	H 2 -5.77343750	A3A	0.00000000	00	H 2 -5.70312500	A4C	0.00000000	00
	H 3 3.49218750	37E	0.99609375	FF	H 3 3.37500000	360	0.98437500	FC
	H 4 2.58203125	295	0.99218750	FE	H 4 2.54687500	28C	0.98437500	FC
	O 1 0.14453125	025	**0.57031250**	92	O 1 -0.01562500	FFC	**0.48437500**	7C
	O 2 -0.13671875	FDD	**0.42968750**	6E	O 2 0.01562500	004	**0.50000000**	80
Geraden-annäherung	H 1 0.09375000	018	0.54687500	8C	H 1 -0.03125000	FF8	0.48437500	7C
	H 2 -5.77343750	A3A	0.00000000	00	H 2 -5.70312500	A4C	0.00000000	00
	H 3 3.49218750	37E	0.99609375	FF	H 3 3.37500000	360	0.99609375	FF
	H 4 2.58203125	295	0.99609375	FF	H 4 2.54687500	28C	0.99609375	FF
	O 1 0.15234375	027	**0.57421875**	93	O 1 -0.03125000	FF8	**0.48437500**	7C
	O 2 -0.14843750	FDA	**0.42578125**	6D	O 2 0.03125000	008	**0.51562500**	84

Tabelle 5.1: Ausgaben des Simulationsprogramms *bpgsim*

Die Verwendung der Geradenannäherung als Aktivierungsfunktion in *bpgsim* hatte bei den vorhandenen Bildern keinen Einfluß auf die Erkennungsrate. Bei gleichen Stellenanzahlen konnte sowohl mit der Sigmoidfunktion als auch mit der Geradenannäherung die gleiche Erkennungsrate erzielt werden, wenngleich die genauen Ausgabewerte aufgrund des unterschiedlichen Berechnungsverfahrens geringfügig unterschiedlich waren.

In Bild 5.4 ist eines der aufgenommenen Bilder sowie die Ergebnisse nach den einzelnen Bearbeitungsschritten zu sehen. Mit dem Bild auf der rechten Seite von Bild 5.4 wurden die Ergebnisse in Tabelle 5.1 berechnet.

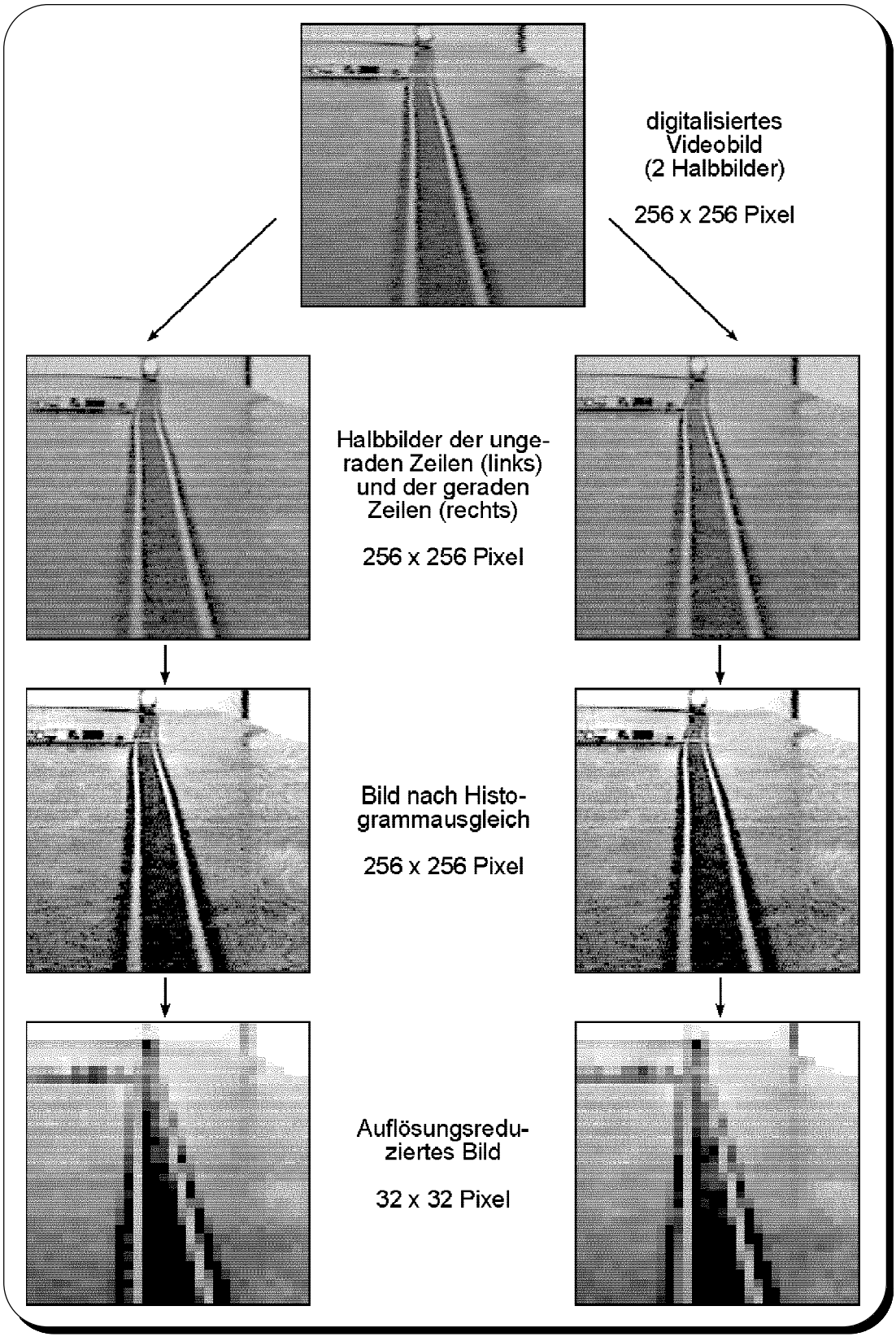


Bild 5.4: Eisenbahnbilder nach den einzelnen Bearbeitungsschritten

6 NeNEB-Chip

Dieses Kapitel beschreibt die Architektur und die Funktionsweise des NeNEB-Chips. Zunächst erfolgt ein Überblick über die Struktur des implementierten Netzwerkes. Anschließend erfolgt eine Beschreibung der einzelnen Funktionsblöcke und deren Aufbau.

6.1 Netzwerkarchitektur

Das im NeNEB-Chip implementierte neuronale Netzwerk besteht aus drei Schichten, die jeweils untereinander voll verbunden sind (Bild 6.1):

- **Eingabeschicht**

  Die Eingabeschicht kann maximal 2^{16} = 65536 Eingabestellen enthalten. Somit können Bilder mit einer maximalen Auflösung von 256*256 Pixeln direkt vom Netzwerk verarbeitet werden.

- **verdeckte Schicht**

  Für die verdeckte Schicht werden 8 Neuronen verwendet.

- **Ausgabeschicht**

  In der Ausgabeschicht stehen insgesamt 32 Neuronen zur Verfügung.

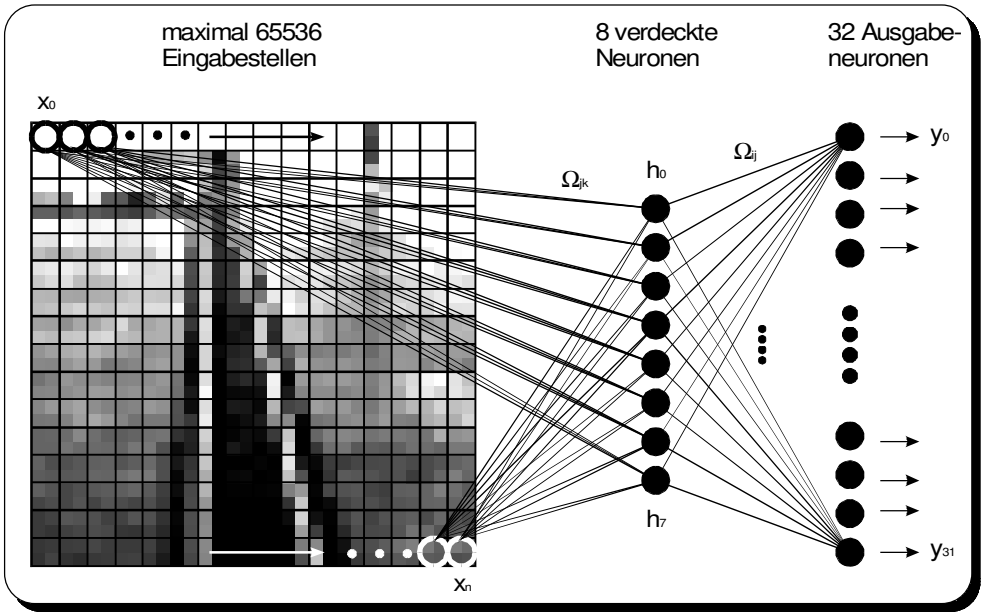


Bild 6.1: Netzwerkarchitektur des NeNEB-Chips

6.2 Blockschaltbild und Signalfluß

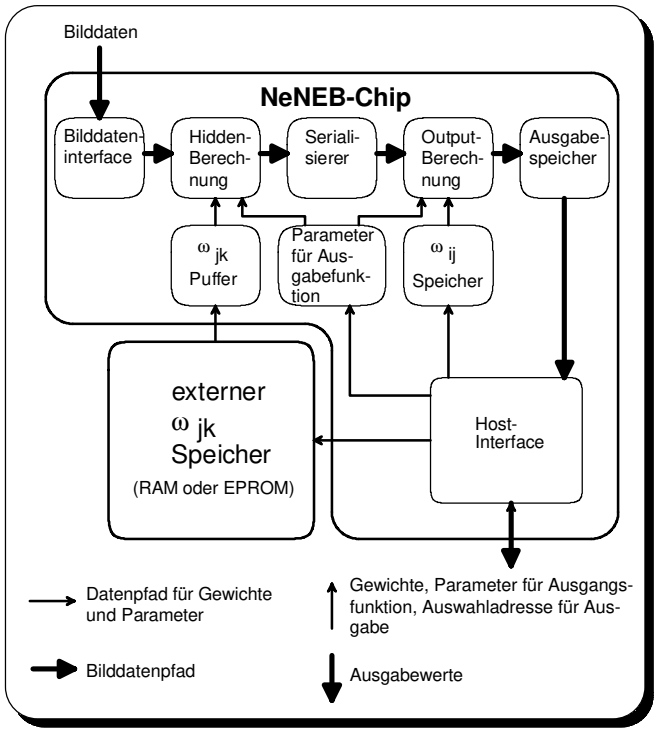


Bild 6.2: Blockschaltbild NeNEB-Chip

Im Blockschaltbild in Bild 6.2 sind alle Funktionsblöcke und Datenpfade des NeNEB-Chips zu sehen. Auf die Steuereinheit mit den erforderlichen Steuersignalen ist in dem Bild der Übersichtlichkeit wegen verzichtet worden. Die Bilddaten werden zunächst im *Bilddateninterface* für die Dauer eines Pixels zwischengespeichert, so daß sie während der Zeitdauer einer Multiplikation den verborgenen Neuronen (*Hidden-Berechnung*) zur Verfügung stehen. Nachdem alle verborgenen Neuronen ihre Ausgabe berechnet haben, d. h. nachdem ein ganzes Bild verarbeitet wurde, werden die Ausgaben der verborgenen Neuronen der Reihe nach den Ausgabeneuronen (*Output-Berechnung*) zugeführt. Hierzu werden die parallelen Ausgaben der verborgenen Neuronen in einem *Serialisierer* serialisiert. Diese Vorgehensweise erlaubt für die Ausgabeneuronen die gleiche serielle Arbeitsweise, wie bei den verborgenen Neuronen. Die Ausgaben der Ausgabeneuronen werden nach abgeschlossener Berechnung in einen *Ausgabespeicher* geschrieben, in dem die Ausgabedaten bis zum Abschluß der Berechnung für das nächste Bild erhalten bleiben und über das *Host-Interface* wahlfrei abgerufen werden können.

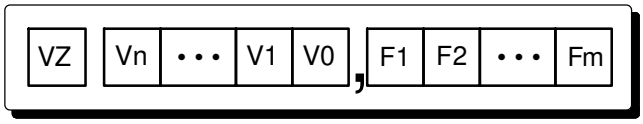
Über das *Host-Interface* erfolgt auch die anfängliche Initialisierung des NeNEB-Chips. Hierbei werden die internen Register mit den Parametern für die Ausgabefunktion (*Parameter für Ausgabefunktion*) sowie die Gewichte der Ausgabeneuronen (ω_{ij}*-Speicher*) programmiert. Sofern als externer Gewichtespeicher (*externer* ω_{jk}*-Speicher*) für die verborgenen Neuronen ein RAM[1] anstelle eines EPROM[2] verwendet wird, erfolgt dessen Programmierung ebenfalls über das *Host-Interface*. Das *Host-Interface* stellt die Verbindung zur "Außenwelt" dar, d. h. hier kann der Anschluß eines PC, einer Workstation o. ä. erfolgen.

[1] *RAM - engl. Random Access Memory* ≈ Schreib-/Lesespeicher mit wahlfreiem Zugriff auf die einzelnen Datenworte
[2] *EPROM - engl. Erasable and Progammable Read Only Memory* ≈ lösch- und programmierbarer nur lesbarer Speicher

Alle in Bild 6.2 im Bilddatenpfad skizzierten Funktionsblöcke sind pipelineartig hintereinandergeschaltet. Das Kennzeichen einer Pipelinestruktur ist die Pufferung der Daten am Ein- und/oder Ausgang jeder Komponente der Pipeline für die Verarbeitungszeitdauer einer Komponente, wodurch die einzelnen Komponenten der Pipeline bis zu einem gewissen Grad parallel arbeiten können. Dies bedeutet, daß z. B. die *Hidden-Berechnung* schon neue Daten aufnehmen kann, während der *Serialisierer* noch mit den vorhergehenden Daten beschäftigt ist.

6.3 Signalrepräsentation

In diesem Abschnitt erfolgt eine Beschreibung aller intern und extern verwendeten Datenformate des NeNEB-Chips. Für die Darstellung wird dabei das folgende einheitliche Format verwendet, das die Bedeutung und die Anzahl der einzelnen Bits widerspiegelt.



Zahlen mit Betrag und Vorzeichen: $-[2^{n+1} - 2^{-m}] \le x \le +[2^{n+1} - 2^{-m}]$

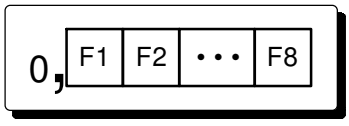
Zweierkomplementwerte: $-[2^{n+1}] \le x \le +[2^{n+1} - 2^{-m}]$

Wertebereich (Beispiel n=3, m=8): $\boxed{-15,9375 \le x \le +15,9375}$

Bit	Beschreibung
VZ	Vorzeichenbit, wird bei Zweierkomplementwerten direkt an den Vorkommastellen plaziert
Vn	Vorkommastelle, Wertigkeit 2^n
V1	Vorkommastelle, Wertigkeit 2^1
V0	Vorkommastelle, Wertigkeit 2^0
F1	Dezimalstelle, Wertigkeit 2^{-1}
F2	Dezimalstelle, Wertigkeit 2^{-2}
Fm	Dezimalstelle, Wertigkeit 2^{-m}

Bild 6.3: Darstellung der Datenformate

6.3.1 Ein-/Ausgabesignale der Neuronen



$\boxed{0 \le x \le 0,99609375}$

Bild 6.4: Darstellung der Ein-/Ausgabesignale *x*

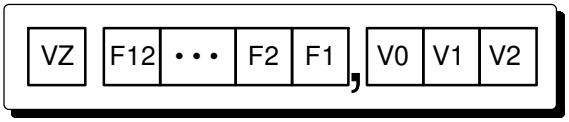
Wie durch die Spezifikation des Chips festgelegt und in Kapitel 5.2.3 beschrieben, erfolgt die Darstellung der Bilddaten mit 8 Bit. Dies erlaubt die Kodierung von maximal 256 Helligkeits-

bzw. Grauwerten der Pixel. Die 256 möglichen Grauwerte werden dabei auf den Bereich $[0...(1-2^{-8})]$ normiert. Die Zuordnung von Grauwerten, Binärwerten und äquivalenten Dezimalwerten gibt Tabelle 6.1 wieder, wobei die Binärwerte der besseren Übersicht wegen in Tetraden[1] aufgeteilt sind. Diese Kodierung wird für die Ein- und Ausgabesignale x aller Neuronen verwendet. Da die Dezimalwerte stets kleiner als Eins sind, werden nur die Nachkommastellen kodiert. Alle Werte sind vorzeichenlose positive Zahlen.

Grauwert	Dezimalwert	Binärwert
100 % Schwarz 0 % Weiß	0,00000000	0000 0000
254/255 % Schwarz 1/255 % Weiß	0,00390625	0000 0001
⋮	⋮	⋮
1/255 % Schwarz 254/255 % Weiß	0,9921875	1111 1110
0 % Schwarz 100 % Weiß	0,99609375	1111 1111

Tabelle 6.1: Kodierung der Bilddaten

6.3.2 Gewichte und Schwellwerte

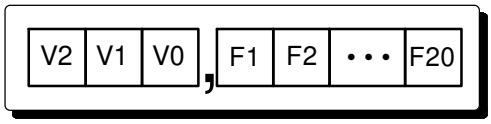


$$-7,999755859 \le \omega \le +7,999755859$$

Bild 6.5: Darstellung der Gewichte ω und der Schwellwerte θ

Für die Gewichte ω und die Schwellwerte θ werden vorzeichenbehaftete 16-Bit-Werte mit 12 Nachkommastellen verwendet, wobei die Darstellung mit Betrag und Vorzeichen erfolgt. Diese Darstellung wurde in Anlehnung an [Clo94] und aus den in Kapitel 5.4 genannten Gründen (viele Gewichte = kleine Werte) gewählt.

6.3.3 Multiplikationsergebnisse



$$0 \le M \le 7,999999046$$

Bild 6.6: Darstellung der Multiplikationsergebnisse M

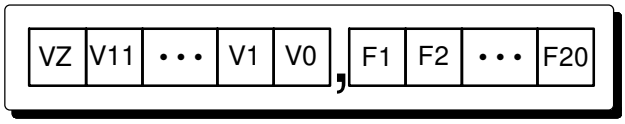
[1] *Tetraden - von griech. Tetra* ≈ vier

Die Multiplikationsergebnisse *M*

$$M = |\omega| \cdot x \tag{6.1}$$

werden mit 23 Bit vorzeichenlos kodiert. Dabei werden, wie für die Gewichte, 3 Bit für die Vorkommastellen benutzt. Für die Anzahl Dezimalstellen wird die Summe der Dezimalstellenanzahlen der Signale (8) und der Gewichte (12) verwendet.

6.3.4 Additionsergebnisse



$$-2048,0 \le a \le +2047,999999046$$

Bild 6.7: Darstellung der Additionsergebnisse *a*

Bei den Additionsergebnissen *a* handelt es sich um die aufsummierten Ergebnisse der Multiplikationen *M* und dem Schwellwert θ (vgl. Integrationsfunktion (2.3))

$$a = \theta + \sum_i M_i \cdot \mathrm{sgn}(\omega_i) = \theta + \sum_i \omega_i \cdot x_i \ .^{1} \tag{6.2}$$

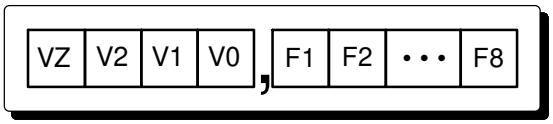
Deren Darstellung erfolgt im Zweierkomplement mit insgesamt 32 Bit sowie 20 Dezimalstellen, wie bei den Multiplikationsergebnissen *M*. Die relativ große Zahl an Vorkommastellen (11) erklärt sich dadurch, daß trotz relativ kleiner Multiplikationsergebnisse bei vielen Eingabestellen bei der Aufsummierung relativ große Zwischenergebnisse auftreten können, obwohl das Endergebnis wesentlich kleiner ist. Hierzu folgendes Beispiel:

> a) Zunächst wird ein Addierer ohne Bereichsbegrenzung verwendet. Es werden *n* Multiplikationsergebnisse *M* aufsummiert, die zu einem Zwischenergebnis $a_1 = +10$ führen. Nun werden weitere *m* Multiplikationsergebnisse aufsummiert, die zu einem Zwischenergebnis $a_2 = +30$ führen. Anschließend werden *o* Multiplikationsergebnisse aufsummiert, die zu einem Endergebnis $a_3 = +10$ führen. Das Resultat ist ein voll erregtes Neuron.
>
> b) Nun wird ein Addierer mit einer Bereichsbegrenzung auf ± 10 verwendet. Nach den ersten *n* Multiplikationsergebnissen stimmt das Zwischenergebnis $\tilde{a}_1 = +10$ überein. Jedoch bleibt dieses Zwischenergebnis ($\tilde{a}_2 = +10$) nach den weiteren *m* Multiplikationsergebnissen erhalten, da der Addierer keine größeren Additionsergebnisse als $+10$ aufgrund der Bereichsbegrenzung zuläßt. Werden nun die letzten *o* Multiplikationsergebnisse aufsummiert, so führt dies zu einem Endergebnis $\tilde{a}_3 = -10$, was einem gehemmten Neuron entspricht.

Das vorstehende Beispiel zeigt ein Umkippen des Ausgabewertes des Neurons, das durch die Bereichsbegrenzung des Addierers hervorgerufen wird. Um ein solches Umkippen beim NeNEB-Chip möglichst zu vermeiden, wurde die Anzahl der Vorkommastellen der Additionsergebnisse *a* großzügig bemessen.

[1] *Signum-(sgn)-Funktion* = liefert das Vorzeichen einer Zahl (+1 für positive Zahl, -1 für negative Zahl)

6.3.5 Erregung der Neuronen

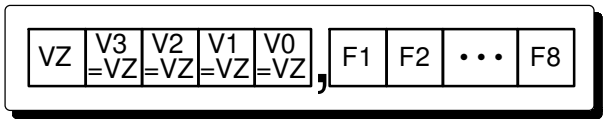


$$-8,0 \le E \le +7,99609375$$

Bild 6.8: Darstellung der Erregung *E* der Neuronen

Die Gesamterregung *E* der Neuronen verwendet nur eine Teilmenge des Zahlenbereichs der Additionsergebnisse *a*. Der Zahlenbereich für die Gesamterregung *E* wurde so gewählt, daß auch noch eine relativ flache Sigmoidfunktion (c = 0,5) über den ganzen abhängigen Wertebereich (0 bis 1) berechnet werden kann. Die Darstellung erfolgt ebenfalls im Zweierkomplement.

6.3.6 Schwellwert der Ausgabefunktion

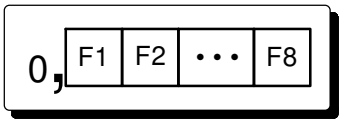


$$-16,0 \le SW \le +15,99609375 \quad \text{theoretisch}$$
$$-0,5 \le SW \le +0,5 \quad \text{erlaubt}$$

Bild 6.9: Darstellung des Schwellwertes *S* der Ausgabefunktion

Der erlaubte Wertebereich des Schwellwertes *SW* ist deutlich kleiner, als der theoretisch mögliche Wertebereich. Die Bits *V3-V0* der Vorkommastellen haben, wie in Bild 6.9 zu sehen, immer den Wert des Vorzeichenbits *VZ* (Erklärung siehe Kapitel 6.5). Die Darstellung der Schwellwerte *SW* erfolgt im Zweierkomplement.

6.3.7 Offsets der Ausgabefunktion



$$0 \le O \le +0,99609375$$

Bild 6.10: Darstellung der Offsets *O* der Ausgabefunktion

Die Offsets *O* der Ausgabefunktion verwenden das gleiche Darstellungsschema wie die Ein-/Ausgabesignale.

6.4 Erregungsfunktion

Die Erregungsfunktion wird in zwei Teilen berechnet:

1. Vorzeichenlose Berechnung der Multiplikationen nach (6.1): $M = |\omega| \cdot x$
2. Aufaddierung der Multiplikationsergebnisse und Berücksichtigung des Schwellwertes θ nach (6.2): $a = \theta + \sum_i M_i \cdot \mathrm{sgn}(\omega_i)$

Beide Berechnungen werden in den folgenden beiden Kapiteln ausführlich dargestellt. Weiterhin erfolgt eine Beschreibung des Speicherschemas der Gewichte für die verborgenen Neuronen und die Ausgabeneuronen.

6.4.1 Erregungsfunktion Teil 1 (Multiplikation)

Die Berechnung der Multiplikationen erfolgt, zunächst ohne Berücksichtigung des Vorzeichens der Gewichte ω , nach dem Hornerschema [Bro85]. Das Hornerschema dient dazu, Polynome der Form

$$y = a_0 \cdot x^0 + a_1 \cdot x^1 + ... + a_n \cdot x^n \tag{6.3}$$

einfach und ohne Potenzierung zu berechnen. Statt dessen wird das Polynom durch fortlaufende Addition und Multiplikation nach dem folgenden Schema berechnet:

$$y = (((a_n \cdot x + a_{n-1}) \cdot x + a_{n-2}) \cdot x + ... + a_1) \cdot x + a_0 \ . \tag{6.4}$$

Die Multiplikation zweier Zahlen im Binärsystem läßt sich ebenfalls durch ein Polynom der Form

$$c = a \cdot b = a \cdot b[0] \cdot 2^0 + a \cdot b[1] \cdot 2^1 + ... + a \cdot b[n] \cdot 2^n \tag{6.5}$$

ausdrücken. Der Ausdruck $b[n]$ bezeichnet dabei das n-te Bit der Binärzahl b mit Wertigkeit 2^n. Somit kann auch die Multiplikation zweier Zahlen nach dem Hornerschema wie folgt durchgeführt werden, wobei ein Operand (b) seriell der Berechnung zugeführt wird, der andere (a) hingegen parallel vorliegt:

$$c = (((a \cdot b[n] \cdot 2 + a \cdot b[n-1]) \cdot 2 + a \cdot b[n-2]) \cdot 2 + ... + a \cdot b[1]) \cdot 2 + a \cdot b[0] \ . \tag{6.6}$$

Im Binärsystem ist diese Berechnung nach dem Hornerschema relativ einfach durchzuführen, da die Multiplikation $a \cdot b[n]$ durch eine UND-Verknüpfung realisiert werden kann und die Multiplikation mit 2 durch eine Schiebeoperation nach links. Somit wird außer der UND-Verknüpfung und dem Schieberegister nur noch ein Addierer in der Breite der Summe der Stellenzahlen beider Operanden benötigt.

Die Realisierung der Multiplikation nach dem Hornerschema beim NeNEB-Chip zeigt Bild 6.11. Hier wurden für den parallel anliegenden Operand (= Operand a in (6.6)) die Signale x gewählt, da diese z. B. am Eingang des Chips ohnehin in Form der Bilddaten byteweise zur Verfügung stehen. Die Gewichte ω (= Operand b in (6.6)) werden somit der Berechnung seriell zugeführt. Dies bietet Vorteile hinsichtlich der benötigten Verbindungsleitungen zum externen Gewichtespeicher, wie in Kapitel 6.4.3 gezeigt wird.

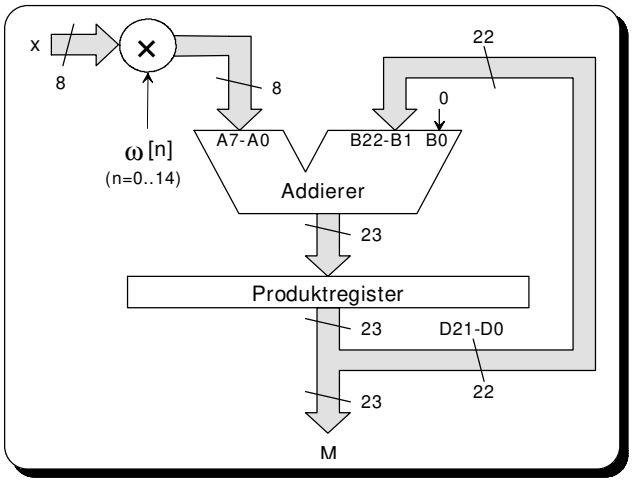


Bild 6.11: 1. Teil der Erregungsfunktion zu Berechnung von $|\omega| \cdot x$ nach dem Hornerschema

In Analogie zu (6.5) berechnen die Neuronen im NeNEB-Chip die Gleichung

$$M = (x \cdot |\tilde{\omega}|) \cdot 2^{-m} = (x \cdot \omega[0] \cdot 2^0 + x \cdot \omega[1] \cdot 2^1 + ... + x \cdot \omega[n] \cdot 2^n) \cdot 2^{-m} \qquad (6.7)$$

mit (vgl. Zahlendarstellung der Gewichte in Kapitel 6.3.2)

$$\tilde{\omega} = (\omega[0] \cdot 2^{-m} + \omega[1] \cdot 2^{-m+1} + ... + \omega[m-1] \cdot 2^{-1} + \omega[m] \cdot 2^0 + ... + \omega[n+m] \cdot 2^n) \cdot 2^m \quad . \qquad (6.8)$$

Für die Berechnung nach dem Hornerschema ergibt sich somit in Analogie zu (6.6)

$$M = ((((x \cdot \omega[\tilde{n}] \cdot 2 + x \cdot \omega[\tilde{n}-1]) \cdot 2 + x \cdot \omega[\tilde{n}-2]) \cdot 2 + ... + x \cdot \omega[1]) \cdot 2 + x \cdot \omega[0]) \cdot 2^{-m} \qquad (6.9)$$

mit

$$\tilde{n} = n + m .$$

Die Multiplikationen mit 2^m bzw. 2^{-m} in den Gleichungen (6.7) bis (6.9) werden nicht als Rechenoperation ausgeführt. Sie dienen nur der Interpretation des Kommas an der richtigen Stelle.

Auf ein Schieberegister wurde beim NeNEB-Chip verzichtet, da in jedem Schritt nach der Multiplikation und der Addition nur eine Multiplikation mit 2 durchgeführt werden muß, was jedoch nur eine Verschiebung um eine Stelle nach links bedeutet. Diese Verschiebung wird durchgeführt, indem die Summe aus dem Addierer um eine Stelle nach links versetzt wieder als zweiter Operand (*B*) für die Addition verwendet wird. Das niederwertigste Bit des Operanden *B* wird permanent mit logisch Null besetzt. Somit wird außer dem Addierer (*Addierer*) und einem Zwischenregister (*Produktregister*) nur noch eine UND-Verknüpfung der Signale mit dem jeweiligen Bit der Gewichte benötigt (Bild 6.11).

6.4.2 Erregungsfunktion Teil 2 (Addition)

Im zweiten Teil der Erregungsfunktion werden die Multiplikationsergebnisse *M* unter Berücksichtigung des Vorzeichens der Gewichte ω zusammen mit dem jeweiligen Schwellwert θ des Neurons aufsummiert. Die Realisierung dieses zweiten Teils der Erregungsfunktion zeigt Bild Bild 6.12. Auch hier ist wieder eine pipelineartige Struktur, wie in Kapitel 6.2 bereits erläutert, erkennbar. In dem Summandenregister wird das Ergebnis der vorhergehenden Multiplikation bis zum Vorliegen eines neuen Multiplikationsergebnisses

gespeichert, so daß dem Addierer die gesamte Zeit zwischen zwei Multiplikationen zur Verfügung steht.

Die Berechnung beginnt, indem zunächst der vorzeichenlose Schwellwert θ (*Thetaregister*) über einen Multiplexer (*Mux1*) in den höherwertigen Teil des Summandenregisters (*Summandenregister1*) geladen wird. Gleichzeitig wird der niederwertige Teil des Summandenregisters (*Summandenregister2*) sowie das Summenregister (*Summenregister*) auf Null gesetzt. Über den Addierer (*Addierer2*) wird dann aus dem vorzeichenlosen Schwellwert ein Zweierkomplementwert gebildet und im Summenregister gespeichert, indem der vorzeichenlose Schwellwert in Abhängigkeit vom Vorzeichen des Schwellwertes zu Null addiert oder von Null subtrahiert wird. Das Vorzeichen des Schwellwertes gelangt über einen Multiplexer (*Mux2*) an den entsprechenden Funktionswahleingang (*Add/Sub*) des Addierers.

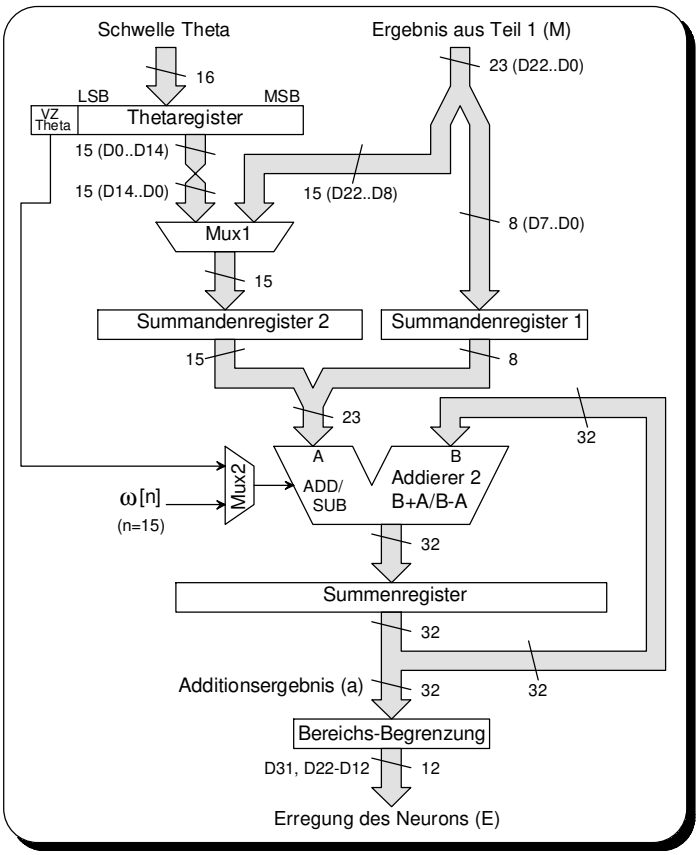


Bild 6.12: 2. Teil der Erregungsfunktion zur Berechnung von $a = \theta + \sum_i M_i \cdot \mathrm{sgn}(\omega_i)$

Die Aufsummierung der Multiplikationsergebnisse *M* geht in ähnlicher Weise vonstatten. Zunächst werden die vorzeichenlosen Multiplikationsergebnisse *M* in das Summandenregister geladen, wobei der Multiplexer *Mux1* nun auf den rechten Datenpfad geschaltet ist. Der Multiplexer *Mux2* sorgt nun dafür, daß das Vorzeichen des Gewichtes ω an den Funktionswahleingang *Add/Sub* des Addierers gelangt. Die Multiplikationsergebnisse werden nun in Abhängigkeit vom Vorzeichen des Gewichtes ω zum Summenregister addiert oder subtrahiert.

Nach Abschluß der Berechnung, d. h. nach Addition aller Multiplikationsergebnisse *M*, wird das Additionsergebnis *a* auf den Wertebereich der Erregung *E* reduziert (*Bereichs-Begrenzung*). Die Bereichsbegrenzung schneidet zum einen einige Dezimalstellen ab, zum anderen werden der maximale und minimale Wert auf den Wertebereich $[-8... + (8 - 2^{-8})]$ (vgl.

Kapitel 6.3.5) begrenzt. Der Begrenzungsalgorithmus kann in Form einer VHDL-ähnlichen Beschreibung wie folgt beschrieben werden:

```
IF (D23 or D24 or ... or D30 = 1) and (D31 = 0) THEN
 setze D12 bis D22 auf 1 (maximaler Wert); -- zu große pos. Zahl
ELSIF (D23 or D24 or ... or D30 = 0) and (D31 = 1) THEN
 setze D12 bis D22 auf 0 (minimaler Wert); -- zu große neg. Zahl
ELSE
 lasse D12 bis D22 unverändert; -- Normalfall, keine Änderung
END IF;
```

Der erste Vergleich prüft, ob das Additionsergebnis positiv zu groß ($\geq +8$) ist. Dieses ist der Fall, wenn eines oder mehrere der höherwertigen Bits D23 bis D30 den Wert 1 hat. Die für die Erregung E relevanten Bits D12 bis D22 werden dann auf 1, d. h. den größten positiven Wert ($8 - 2^{-8}$), gesetzt, wobei das Vorzeichenbit D31 unverändert übernommen wird. Analog dazu prüft der zweite Vergleich, ob das Additionsergebnis negativ zu groß ist (< -8), was durch eine 0 in einem oder mehreren der höherwertigen Bits gekennzeichnet ist. In diesem Fall wird die Erregung auf den größten negativen Wert (-8) gesetzt, wobei wiederum das Vorzeichenbit unverändert übernommen wird. Einige Beispiel für die Wertebereichsbegrenzung der Erregung sind in Tabelle 6.2 zu finden. Der Grund für die Begrenzung der Erregung E wird in Kapitel 6.5 bei der Beschreibung der Ausgabefunktion deutlich.

Additions-ergebnis a	0 00010101 01101100011 001101100101 D31 D22 D12	positiv zu groß
Erregung E	0 11111111111	maximaler pos. Wert
Additions-ergebnis a	1 10100101 10101101011 101001000111 D31 D22 D12	negativ zu groß
Erregung E	1 00000000000	maximaler neg. Wert
Additions-ergebnis a	0 00000000 01101100011 001101100101 D31 D22 D12	innerhalb des Wertebereichs von E
Erregung E	0 01101100011	unveränderter Wert

Tabelle 6.2: Beispiele für die Bereichsbegrenzung der Erregung E

Der Addierer enthält ebenfalls eine Bereichsbegrenzung, die bei einem Zweierkomplement-Über-/Unterlauf das Ergebnis auf den maximalen bzw. minimalen Wert setzt. Der vorzeichenlose Operand A wird für die folgenden Erläuterungen als Zweierkomplementwert aufgefaßt. Er ist positiv, wenn der Addierer auf Addieren geschaltet ist, und negativ, wenn der Addierer über den Funktionswahleingang *Add/Sub* auf Subtrahieren geschaltet ist. Somit wird die Subtraktion auf eine Zweierkomplementaddition zurückgeführt. Ein Zweierkomplementüberlauf entsteht dabei, wenn beide Operanden des Addierers positiv sind, das Ergebnis aber negativ. In diesem Fall wird das Ergebnis auf den maximalen positiven Wert ($2048 - 2^{-20}$) gesetzt. Analog entsteht ein Zweierkomplementunterlauf, wenn beide Operanden negativ sind, das Ergebnis aber positiv. In diesem Fall wird das Ergebnis auf den maximalen negativen Wert (-2048) gesetzt. In Form einer VHDL-ähnlichen Beschreibung läßt sich die Behandlung des Zweierkomplementüberlaufs wie folgt ausdrücken:

```
IF (A positiv) and (B positiv) and (Summe negativ) THEN
 setze Summe auf maximalen positiven Wert; -- Summe pos. zu groß
ELSIF (A negativ) and (B negativ) and (Summe positiv) THEN
 setze Summe auf maximalen negativen Wert; -- Summe neg. zu groß
ELSE
 lasse Summe unverändert; -- Normalfall, keine Änderung
END IF;
```

Die folgende Tabelle 6.3 enthält einige Beispiele für die Zweierkomplementbehandlung des Addierers.

Operand A		0	000	0000	0011	1111	1010	1000	1001	0100	positiv
Operand B	+	0	111	1111	1111	1111	1001	1000	0000	0010	positiv
Summe S	=	1	000	0000	0011	1111	0100	0000	1001	0110	negativ
korrigierte Summe S	->0		111	1111	1111	1111	1111	1111	1111	1111	maximaler positiver Wert
Operand A		1	111	1111	1100	0000	0101	0111	0110	1011	negativ
Operand B	+	1	000	0000	0000	0000	0110	0111	1111	1111	negativ
Summe S	=	0	111	1111	1100	0000	1011	1111	0110	1010	positiv
korrigierte Summe S	->1		000	0000	0000	0000	0000	0000	0000	0000	maximaler negativer Wert

Tabelle 6.3: Beispiele zum Zweierkomplementüberlauf im Addierer (*Addierer2*)

6.4.3 Speicherung der Gewichte der verborgenen Neuronen

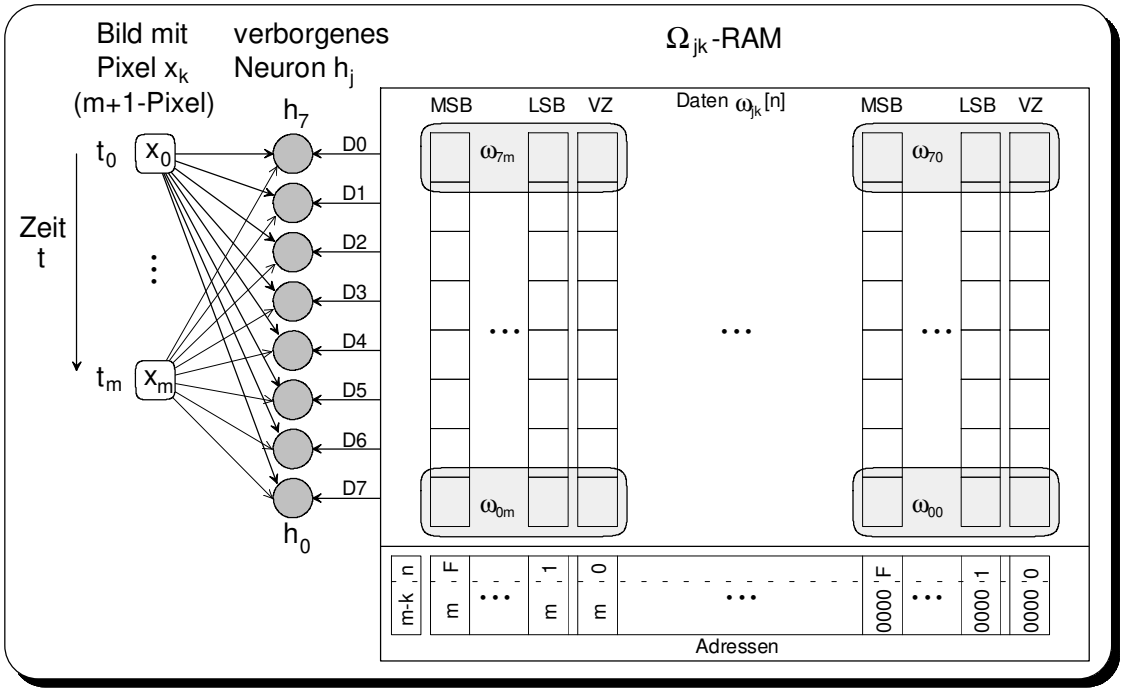


Bild 6.13: Speicherung der Gewichte ω_{jk} im externen Gewichtespeicher

Im externen Gewichtespeicher (Bild 6.13) werden die Gewichte ω_{jk} der verborgenen Neuronen gespeichert. Der Grund für die externe Speicherung im Gegensatz zur ON-Chip Speicherung liegt in der hohen Zahl der möglichen Eingabestellen und der damit verbundenen hohen Zahl von Gewichten für die verborgenen Neuronen. Als externer Speicher kann sowohl ein statisches *RAM* als auch ein *EPROM* direkt angesteuert werden. Die Art der Speicherorganisation ist direkt von der Art der Berechnung der Multiplikationen in der Erregungs- bzw. Integrationsfunktion abhängig. Die gewählte serielle Zuführung der Gewichte bietet große Vorteile hinsichtlich der Speicherung der Gewichte der verborgenen Neuronen. Diese müssen aus Geschwindigkeitsgründen alle parallel rechnen, d. h. für jedes Bilddatenbyte werden acht

Gewichte auf einmal benötigt. Würden die Gewichte der Berechnung der verborgenen Neuronen parallel zugeführt werden, so wäre hierfür eine Speicherwortbreite von

$$\text{Anzahl der Hidden} \cdot \text{Stellenzahl der Gewichte} = 8 \cdot 16 = 128 \text{ Bit}$$

und damit 128 Datenleitungen nötig. Hinzu kämen bei maximal 65536 Gewichten 16 Adressleitungen, wodurch insgesamt 144 Leitungen zur Speicheransteuerung nötig wären. Durch die serielle Zuführung der Gewichte hingegen wird in einem Zeittakt immer nur ein Bit aller Gewichte benötigt. Dies bedeutet, daß in einem Speicherwort von 8 Bit für jedes verborgene Neuron ein Bit des jeweiligen Gewichtes untergebracht werden kann. Hierdurch ist nur eine Speicherwortbreite von der Anzahl der verborgenen Neuronen (8) erforderlich. Die Erhöhung der Anzahl der Speicherworte um den Faktor 16 (der Stellenzahl der Gewichte) ist dabei unkritisch, da hierfür nur vier zusätzliche Adressleitungen, d. h. insgesamt 20, benötigt werden. Bei gleicher Speichergröße, allerdings anderer Organisation (statt 64 K[1] * 128 Bit, 1 M[2] * 8 Bit), sind somit 28 Leitungen, d. h. nur ein Fünftel, für die Speicheransteuerung ausreichend. Nicht berücksichtigt sind hierbei die Steuerleitungen für die Speicheransteuerung, die in beiden Beispielen in gleicher Anzahl vorhanden sind.

Da jedes Speicherwort ein Bit gleicher Wertigkeit aller Gewichte der verborgenen Neuronen enthält, ist jedes Gewicht ω_{jk} in 16 aufeinanderfolgenden Speicherworten gespeichert. Das höchste Speicherwort (Adresse $xxxxF_{16}$) eines 16er Blocks enthält das höchstwertige Bit (*MSB*[3], *M2* vgl. 6.3.2), das zweite Speicherwort (Adresse $xxxx1_{16}$) des 16er Blocks enthält das niederwertige Bit (*LSB*[4], *F12* vgl. 6.3.2) und das erste Speicherwort (Adresse $xxxx0_{16}$) des 16er Blocks enthält das Vorzeichenbit (*VZ*). Somit werden mit den vier niederwertigen Adreßleitungen die einzelnen Bits der Gewichte aller verborgenen Neuronen adressiert. Die weiteren Adreßleitungen dienen zur Auswahl der entsprechenden Gewichte für ein bestimmtes Bildpixel x_k. Bei insgesamt (m+1)-Pixeln stehen die Gewichte für das erste Pixel x_0 an der höchsten Adresse, die für das letzte Pixel x_m an Adresse $0000x_{16}$. Diese Adressierungsreihenfolge ist durch rückwärts zählende Zähler für die Adreßgenerierung bedingt. Der Vorteil dieser Arbeitsweise der Zähler wird in Kapitel 6.8 bei der Beschreibung des Steuerwerkes deutlich.

Die Speichergröße bzw. die Anzahl der Speicherworte ist auf $2^{20} = 1048576 = 1\text{ M}$ limitiert. Dies erlaubt die Speicherung von maximal

$$\frac{\text{Anzahl Speicherworte}}{\text{Anzahl Bits der Gewichte}} \cdot \text{Speicherwortbreite} = \frac{2^{20}}{16} \cdot 8 = 2^{19} = 524288 = 512\text{ K}$$

Gewichten. Da jedes Pixel (bzw. jede Eingabestelle) mit jedem verborgenen Neuron verbunden ist, beträgt somit die maximale Anzahl der Eingabestellen

$$\frac{2^{19}}{8} = 65536 = 64\text{ K}.$$

Dies ermöglicht dem NeNEB-Chip, Bilder mit einer Bildgröße von 256*256 Pixeln (sofern quadratische Bilder verwendet werden) direkt ohne Auflösungsreduktion zu verarbeiten. Bei kleineren Bildgrößen kann ein kleinerer Speicher verwendet werden, da stets nur die Adressen 00000_{16} bis $(\text{Bildgröße m} \cdot 16)_{16}$ verwendet werden.

---

[1] *1 K* - 1 Kilo = 2^{10} (binäres Vielfaches)
[2] *1 M* - 1 Mega = 2^{20} (binäres Vielfaches)
[3] *MSB - engl. most significant bit* ≈ höchstwertiges Bit
[4] *LSB - engl. least significant bit* ≈ niederwertiges Bit

6.4.4 Speicherung der Gewichte der Ausgabeneuronen

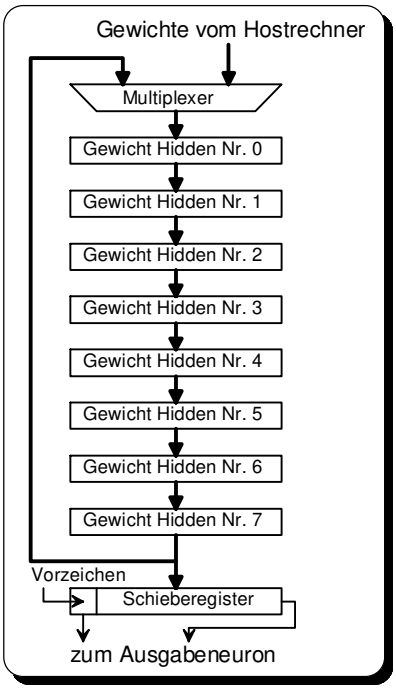


Bild 6.14: Ringspeicher zur Speicherung der Gewichte ω_{ij} der Ausgabeneuronen

Für die Speicherung der Gewichte ω_{ij} der Ausgabeneuronen wurde für jedes Ausgabeneuron ein Ringspeicher (Bild 6.14) mit 8 Speicherzellen, eine für jedes verborgene Neuron, und anschließendem Schieberegister zur Serialisierung der Gewichte implementiert. Da hier der Speicheraufwand aufgrund der wesentlich geringeren Anzahl von Gewichten weitaus kleiner ist, wurde auf einen externen Speicher verzichtet. Hierdurch kann die Anzahl der benötigten Anschlußpins des NeNEB-Chips in vertretbaren Grenzen gehalten werden.

Über den Multiplexer vor der ersten Speicherzelle kann zwischen dem Ausgang der letzten Speicherzelle und dem Eingang des Ringspeichers gewählt werden. Dies ermöglicht zwei unterschiedliche Betriebsarten des Ringspeichers. In der Initialisierungsphase werden die Gewichte der Reihe nach über den Eingang in den Speicher nach dem *Fifo*[1]-Prinzip hineingeschoben. In der Berechnungsphase hingegen werden die Gewichte ringartig geschoben, d. h. der Inhalt der untersten Speicherzelle wandert in die oberste Speicherzelle und der Inhalt aller anderen Speicherzellen wandert eine Position tiefer (Bild 6.15). Nach Abschluß eines Berechnungsvorganges (im Bild durch die neun Zeitpunkte t_1-t_9 angedeutet) stehen somit die Daten wieder in der anfänglichen Reihenfolge.

Das Schieberegister nach dem achten Speicherregister in Bild 6.14 sorgt dafür, daß die Gewichte der Berechnung der Multiplikation seriell, d. h. Bit für Bit, zugeführt werden. Durch diese Vorgehensweise können die verborgenen Neuronen und die Ausgabeneuronen völlig identisch aufgebaut werden, was eine wesentliche Vereinfachung des Implementationsaufwandes darstellt. Das jeweilige Vorzeichen des Gewichtes steht in einem separaten Register und wird nicht mitgeschoben. Der Grund dafür ist, daß das Vorzeichen erst zu einem späteren Zeitpunkt in der Berechnung benötigt wird, als der Betrag des Gewichtes.

[1] *Fifo - engl. first in first out* ≈ zuerst hinein, zuerst hinaus

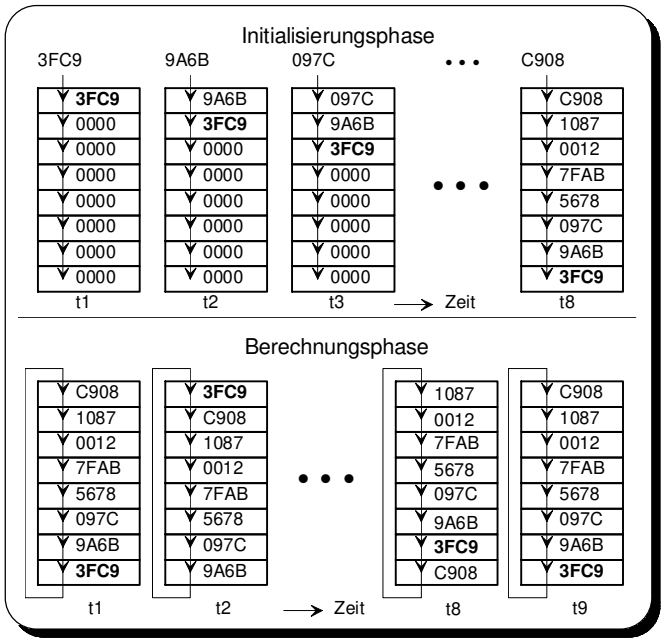


Bild 6.15: Funktionsprinzip des Ringspeichers

6.5 Aktivierungsfunktion

In diesem Kapitel werden die Implementation der durch Geradenannäherung approximierten Aktivierungsfunktion sowie die Bestimmung der dazu notwendigen Parameter beschrieben.

6.5.1 Parameterbestimmung

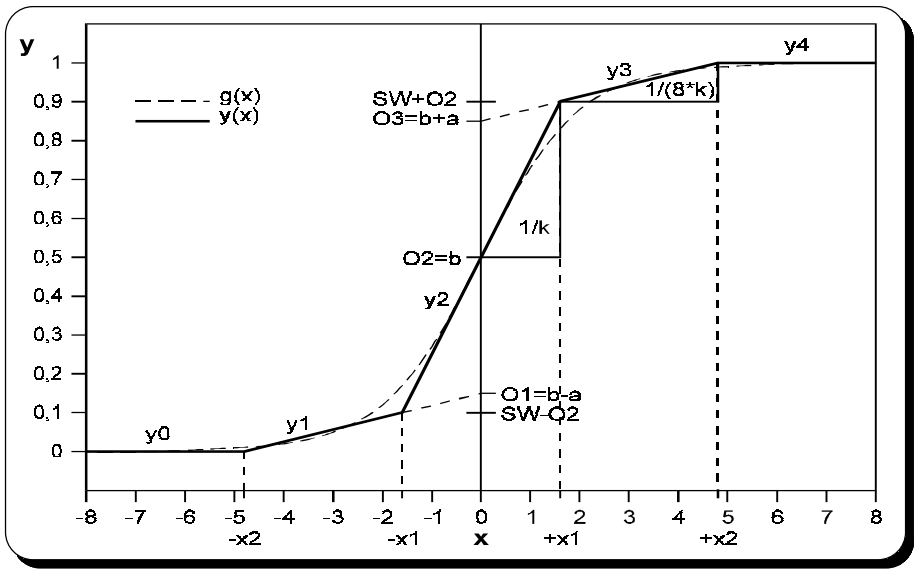


Bild 6.16: Aktivierungsfunktion aus fünf Geradenstücken mit Parametern

Das Bild 6.16 zeigt die aus fünf Geradenstücken bestehende Annäherung der Sigmoidfunktion mit allen wichtigen Parametern. Die stückweise Geradenannäherung wurde gewählt, um den

Implementationsaufwand und die Ausführungszeit der Berechnung in vertretbaren Grenzen zu halten.

Für die Berechnung ergeben sich fünf Bereiche mit unterschiedlichen Geradengleichungen:

- Bereich 1 [-8...-x_2]: $y_0 = 0$ (6.10)
- Bereich 2 [-x_2...-x_1]: $y_1 = \frac{x}{8 \cdot k} + (b - a)$ (6.11)
- Bereich 3 [-x_1...+x_1]: $y_2 = \frac{x}{k} + b$ (6.12)
- Bereich 4 [+x_1...+x_2]: $y_3 = \frac{x}{8 \cdot k} + (b + a)$ (6.13)
- Bereich 5 [+x_2...8]: $y_4 = 1$ (6.14)

Für eine optimale Annäherung an die Sigmoidfunktion

$$g(x, c) = \frac{1}{1 + e^{-cx}} \tag{6.15}$$

können die Parameter a, x_1 und x_2 sowie die Grundsteigung k der Geradenannäherung variiert werden. Der Parameter b ist durch die Sigmoidfunktion festgelegt auf den Wert (vgl. (A.11) und (A.23))

$$b = g(0) = 0,5\,. \tag{6.16}$$

Die Grundsteigung k der Geraden y_2 wird gleich der Steigung der Sigmoidfunktion im Nullpunkt $g'(0, c)$ gewählt. Dies ergibt nach (A.8) bis (A.10)

$$k = \frac{4}{c} = \frac{1}{g'(0, c)} \tag{6.17}$$

Da für die Grundsteigung k nur Zweierpotenzen der Form

$$k = 2^{S-1} \quad \text{mit} \quad S = 0, 1, ..., 7 \tag{6.18}$$

zur Verfügung stehen, muß das Ergebnis aus (6.17) auf den nächsten zulässigen Wert von k auf- oder abgerundet werden. Der Wert S wird als Schiebefaktor bezeichnet und ist im folgenden Kapitel beschrieben.

Der Punkt $\pm x_1$ kennzeichnet den Übergang von Bereich 2 nach Bereich 3 bzw. Bereich 3 nach Bereich 4. An diesem Punkt haben die Geraden y_1 und y_2 bzw. y_2 und y_3 den gleichen Wert. x_1 wird am besten so bestimmt, daß die Fläche zwischen der Sigmoidfunktion $G(x_m)$ und den fünf Geraden $Y(x_1, x_m)$ innerhalb eines Intervalls [0...x_m]

$$F(x_1) = \left| G(x_m) - Y(x_1, x_m) \right| \tag{6.19}$$

minimal wird (vgl. (A.13)). Hierzu werden sowohl die Sigmoidfunktion als auch die Geraden über das Intervall integriert. Dabei reicht es aufgrund der Symmetrie der Sigmoidfunktion aus, wenn ausschließlich über den positiven Bereich integriert wird. Es ergibt sich für die Flächenfunktion (vgl. (A.14) bis (A.53))

$$F(x_1) = \left| \frac{7}{2 \cdot k} \cdot x_1^2 - \frac{7}{2} \cdot x_1 + k + \frac{\ln 0,5}{c} \right| . \tag{6.20}$$

Die Nullstelle der Flächenfunktion $F(x_1)$ bezüglich x_1 ist die gesuchte Lösung. Hierfür ergibt sich (vgl. (A.54) bis (A.59))

$$x_1 = \frac{k}{2} - \sqrt{\frac{k^2}{4} - \frac{2 \cdot k}{7} \cdot \left(k + \frac{\ln 0,5}{c} \right)} \ . \tag{6.21}$$

Der Parameter a der Geraden y_1 und y_3 ist nach (A.24) bis (A.27) direkt von x_1 abhängig. Es ergibt sich dafür

$$a = \frac{7}{8 \cdot k} \cdot x_1 \ . \tag{6.22}$$

Sofern für die Grundsteigung k direkt das Ergebnis aus (6.17) verwendet wird, d. h. nicht auf die nächste passende Zweierpotenz nach (6.18) gerundet wird, hat der Parameter a den konstanten Wert

$$a = 0,3347 , \tag{6.23}$$

da nach (A.60) bis (A.63) der Ausdruck

$$\frac{x_1}{k} = \frac{1}{2} - \sqrt{\frac{1}{4} - \frac{2}{7}\left(1 + \frac{\ln 0,5}{4} \right)} = 0,3825 \tag{6.24}$$

ebenfalls konstant ist.

Der Punkt x_2 braucht nicht explizit bestimmt zu werden, da er direkt von x_1 abhängig ist. Er kommt aufgrund einer Bereichsbegrenzung im Addierer der Aktivierungsfunktion zustande (siehe folgendes Kapitel) und muß daher nicht explizit programmiert werden. Für die Ermittlung von x_1 nach (6.21) ist er jedoch notwendig, da hiervon die Fläche unter den Geraden und damit auch $F(x_1)$ abhängig ist. Es ergibt sich für x_2

$$x_2 = 4 \cdot k - 7 \cdot x_1 \tag{6.25}$$

Die bisherigen Gleichungen (6.17) bis (6.25) liefern passende Rechenwerte für die Parameter der fünf Geraden. Auf dem NeNEB-Chip werden, wie aus dem folgenden Kapitel ersichtlich, hiervon abweichende Parameter programmiert, die sich aus den berechneten wie folgt ergeben (vgl. Bild 6.16):

$$S = 1 + \mathrm{ld}\ k \quad , \quad SW = \frac{x_1}{k} \quad , \quad O_1 = b - a \quad , \quad O_2 = b \quad , \quad O_3 = b + a \ . \tag{6.26}$$

Die folgende Tabelle 6.4 gibt für alle zuvor erläuterten Parameter die passenden Werte im Zusammenhang mit ausgewählten Werten des Parameters c der Sigmoidfunktion wieder.

	berechnete Parameter					**programmierbare Parameter**				
c	***k***	x_1	x_2	***a***	***b***	$S=1+\mathrm{ld}\ k$	$SW=x_1/k$	$O_1=b-a$	$O_2=b$	$O_3=b+a$
8	0,5	0,19	0,66			0				
4	1	0,38	1,32			1				
2	2	0,77	2,64			2				
1	4	1,53	5,29	0,33	0,5	3	0,38	0,17	0,5	0,83
0,5	8	3,06	10,58			4				
0,25	16	6,12	21,16			5				
0,125	32	12,24	42,31			6				
0,0625	64	24,48	84,62			7				

Tabelle 6.4: Parameter *c* der Sigmoidfunktion mit passenden Parametern der Geraden

6.5.2 Realisierung der Aktivierungsfunktion

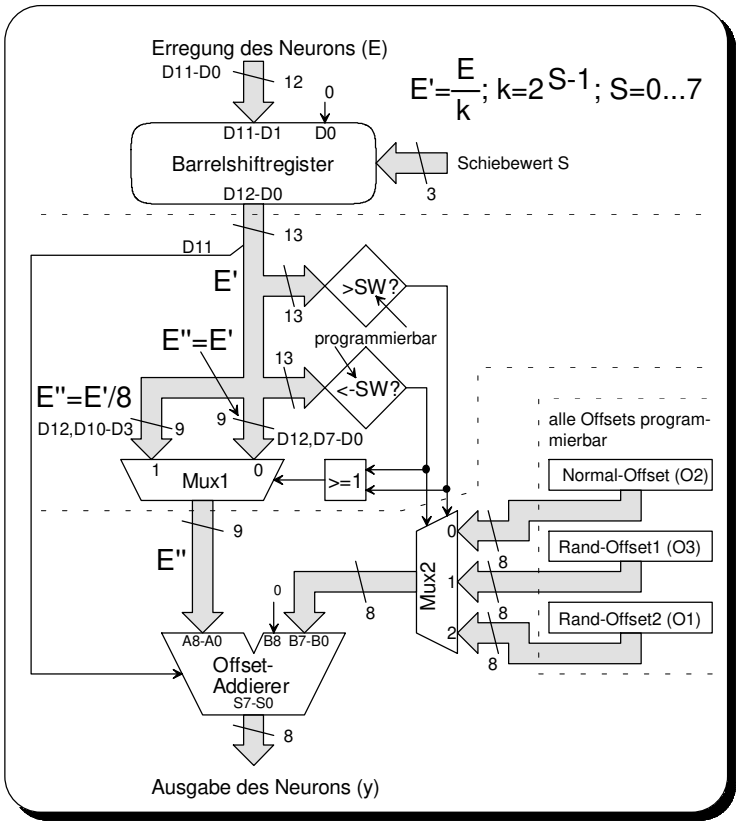


Bild 6.17: Realisierung der Aktivierungsfunktion

Das Bild 6.17 zeigt die Realisierung der Aktivierungsfunktion auf dem NeNEB-Chip. Ebenso wie die zwei Teile der Erregungsfunktion unabhängig voneinander, d. h. parallel arbeiten, erfolgt auch die Berechnung der Aktivierungsfunktion parallel zur Erregungsfunktion, d. h. während bereits eine neue Erregung berechnet wird, wird für die vorhergehende Erregung noch die Aktivierungsfunktion berechnet. Die Berechnung läßt sich grob in drei Teile unterteilen (im Bild durch die Trennlinien angedeutet):

- Teil 1: Division der Erregung E durch die Grundsteigung k:

$$E' = \frac{E}{k} = \frac{E}{2^{S-1}} \tag{6.27}$$

- Teil 2: Vergleich des ersten Divisionsergebnisses E' mit dem Schwellwert SW sowie eventuell zweite Division durch 8:

$$E'' = \begin{cases} E'/8 & \text{wenn} \quad |E'| > SW \\ E' & \text{wenn} \quad |E'| \le SW \end{cases} \tag{6.28}$$

- Teil 3: Offsetaddition gemäß Vergleichsergebnis aus Teil 2 liefert Ausgabe y:

$$y = \begin{cases} E'' + O_3 & \text{wenn} \quad E' > SW \\ E'' + O_1 & \text{wenn} \quad E' < SW \\ E'' + O_2 & \text{sonst} \end{cases} \tag{6.29}$$

Die Division nach (6.27) wird durch ein Barrel-Schieberegister (*Barrelshiftregister*) durchgeführt. Das Barrel-Schieberegister ist dabei ähnlich aufgebaut, wie in Bild 4.11 gezeigt. Der Unterschied ist, daß das Barrel-Schieberegister der Aktivierungsfunktion gleiche Bitbreiten auf der Ein- und Ausgangsseite hat und das hier die Division im Zweierkomplement durchgeführt wird. Damit die Verschiebung immer nur in eine Richtung - nach rechts in Richtung niedriger Wertigkeit - durchgeführt werden kann, wird die Erregung E an das Barrel-Schieberegister um eine Stelle nach links versetzt, d. h. mit 2 multipliziert, herangeführt. Hierbei sei daran erinnert, daß für eine sehr steile Sigmoidfunktion (c=8) die Grundsteigung k der Geraden den Wert 0,5 hat (vgl. Tabelle 6.4). Eine Division durch 0,5 bedeutet aber eine Multiplikation mit 2, was einer Verschiebung nach links entspräche, d. h. das Barrel-Schieberegister müßte eine Verschiebung in beide Richtungen ermöglichen.

Nach erfolgter Division durch die Grundsteigung k wird das Divisionsergebnis E' mit dem Schwellwert SW verglichen. Hierzu werden zwei Komparatoren verwendet, die einen Vergleich auf größer $+SW$ bzw. kleiner $-SW$ durchführen. Der erlaubte Wertebereich der Schwellwerte SW beträgt $\pm 0,5$ (vgl. Kapitel 6.3.6), da bei einem Divisionsergebnis $|E'| \le SW$ stets die Grundgerade y_2 mit Offset $O_2 = 0,5$ berechnet wird, d. h. die Offsetaddition führt bei Werten $|E'| \le 0,5$ stets zu einem Ergebnis $y \le 1$. Somit ist gewährleistet, daß die Sigmoidfunktion stets mit allen fünf Geraden angenähert wird.

Das Vergleichsergebnis der zwei Komparatoren wird zum einen dazu verwendet, eine zusätzliche Division durch 8 zur Berechnung der Geraden y_1 bzw. y_3 durchzuführen, zum anderen dient es dazu, den richtigen Offset für die Offsetaddition gemäß (6.29) auszuwählen. Die zusätzliche Division durch 8 wird allein von einem Multiplexer (*Mux1*) erledigt, der entweder die höherwertigen 9 Bit (D10-D3, inkl. Vorzeichen D12) oder die niederwertigen 9 Bit (D7-D0,D12) der 13 Bit (D12-D0) des ersten Divisionsergebnisses E' auswählt. Das Bit D11 findet bei beiden Datenpfaden keine Verwendung, es wird direkt an den Offsetaddierer geführt.

Der Offsetaddierer (*Offset-Addierer*) addiert zu dem Ergebnis E'' aus (6.28) den durch die Komparatoren ausgewählten Offset gemäß (6.29). Zusätzlich ist in dem Addierer eine Bereichsbegrenzung integriert. Diese Bereichsbegrenzung sorgt dafür, daß das Ergebnis der Addition nicht kleiner als Null bzw. größer als der maximale Wert der Signale ($1 - 2^{-8}$) wird. Die Addition erfolgt dabei im Zweierkomplement, aufgrund der Bereichsbegrenzung werden jedoch nur vorzeichenlose positive Werte gemäß Kapitel 6.3.1 ausgegeben. Da die Offsets nur vorzeichenlose positive Werte annehmen können, ist das Vorzeichenbit B8 des Operanden B des Offsetaddierers permanent mit Null belegt.

Zusätzlich besitzt der Addierer einen Überlaufeingang für das Bit D11 des ersten Divisionsergebnisses E'. Dieses Bit kennzeichnet Werte von E', die auch nach der Offsetaddition nicht zu einer Ausgabe im erlaubten Wertebereich führen. Der Grund dafür ist, daß das Bit D11 einen Wert $|E'| > 8$ bzw. $\left|E'' = \frac{E'}{8}\right| > 1$ kennzeichnet, die Offsets aber stets Werte zwischen Null und kleiner Eins haben. In diesem Fall ist das Bit D11 immer komplementär zum Vorzeichenbit D12. Das Bit D11 braucht nur berücksichtigt zu werden, wenn die zweite Division durch 8 erforderlich ist, da ansonsten durch die Komparatoren der Wert $|E'| < 0,5$ garantiert wird. Weiterhin hat das Bit D11 nur Aussagekraft, wenn der Schiebewert S=0 verwendet wird. Bei allen anderen Schiebewerten hat das Bit D11 immer den Wert des Vorzeichenbits D12. Aus diesen Gründen wurde das Bit D11 nicht direkt als zusätzliches Bit mit in die Addition einbezogen, wodurch der als *ripple-carry* realisierte Addierer um eine Stufe kleiner wurde.

Der Begrenzungsalgorithmus im Addierer läßt sich in Pseudo-VHDL-Code wie folgt formulieren:

```
IF (A positiv) and ((Summe negativ) or (E'>=+8)) THEN
 setze Summe auf maximalen Wert (1-2^-8); -- Zweierkomplementüberlauf
                                          oder E' ist positiv zu groß
ELSIF (A negativ) and ((Summe negativ) or (E'<=-8)) THEN
 setze Summe auf Null; -- Summe ist nach Addition immer noch negativ
                          oder E' ist negativ zu groß
ELSE
 lasse Summe unverändert; -- Normalfall, keine Änderung
END IF;
```

Ein Zweierkomplementüberlauf tritt demnach nur auf, wenn der Operand *A* des Addierers positiv ist und das Ergebnis der Addition negativ. In diesem Fall, wie auch wenn der Wert *E'* größer als +8 ist, ist das tatsächliche Ergebnis größer oder gleich Eins (Bereich 5 der Geradenannäherung) und wird somit auf den maximalen Wert $(1-2^{-8})$ gesetzt. Sofern der Operand *A* und das Ergebnis negativ sind bzw. der Wert *E'* kleiner als -8 (Bereich 1 der Geradenannäherung), wird das Ergebnis auf Null gesetzt.

6.6 Serialisierung der Ausgaben der verborgenen Neuronen

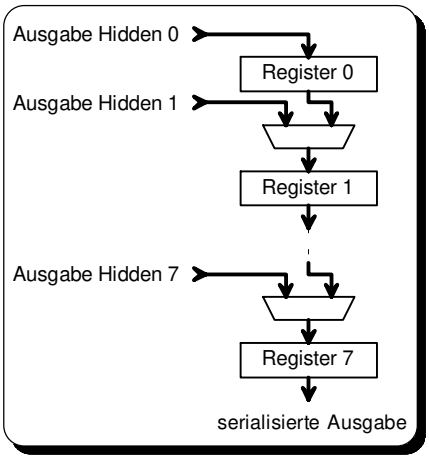


Bild 6.18: Parallel-/Serienumsetzer zur Serialisierung der Ausgabe der verborgenen Neuronen

Die verborgenen Neuronen bekommen ihre Eingabewerte seriell in Form der Bilddaten, gewichten diese und liefern dann ihre Ausgabe. Eine parallele Arbeitsweise, wie sie bei biologischen Neuronen zu beobachten ist, macht hier keinen Sinn, da die Bilddaten seriell von einer Kamera geliefert werden. Bei den Ausgabeneuronen hingegen würde eine parallele Berechnung möglich sein, da alle verborgenen Neuronen ihre Ausgabe parallel liefern. Jedoch wären dann allein für ein Ausgabeneuron insgesamt acht Multiplizierer nach Kapitel 6.4.1 erforderlich. Dies würde bei 32 vorgesehenen Ausgabeneuronen zu einem unvertretbar hohen Hardwareaufwand führen. Daher wurde beschlossen, die Ausgaben der verborgenen Neuronen in einem Parallel-/Serienumsetzer nach Bild 6.18 zu serialisieren. Somit können die Ausgabeneuronen identisch zu den verborgenen Neuronen aufgebaut und implementiert werden. Hierdurch wurde der Implementationsaufwand stark verringert, da auch die Steuerung der Rechenwerke für beide Neuronentypen weitgehend identisch ist.

6.7 Schnittstellen und Signalbeschreibungen

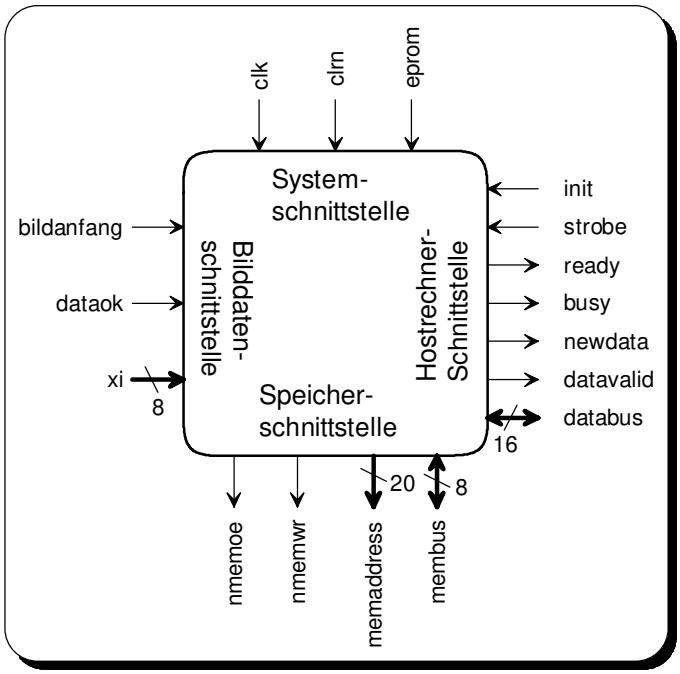


Bild 6.19: Signal- und Schnittstellenübersicht des NeNEB-Chips

In diesem Kapitel werden alle Schnittstellen und Signale, die dem NeNEB-Chip die Kommunikation mit der "Außenwelt" ermöglichen, beschrieben. Die Schnittstellen werden in vier Gruppen aufgeteilt (vgl. Bild 6.19):

- **Systemschnittstelle**

  In dieser Schnittstelle sind alle Signale enthalten, die sich in keine der nachfolgenden Gruppen einordnen lassen.

- **Hostrechnerschnittstelle**

  In der Hostrechnerschnittstelle sind alle Signale und Datenleitungen zusammengefaßt, die der Kommunikation mit einem externen PC oder einer Workstation dienen. Hierüber erfolgt einerseits die Programmierung des NeNEB-Chips, andererseits können über die Hostrechnerschnittstelle die Ausgabedaten des neuronalen Netzwerkes abgerufen werden.

- **Speicherschnittstelle**

  Die Speicherschnittstelle enthält alle Signale und Datenleitungen zur Ansteuerung des externen Gewichtespeichers.

- **Bilddatenschnittstelle**

  Über die Bilddatenschnittstelle ist der NeNEB-Chip mit den anderen Komponenten im Gesamtsystem (Bild 1.1) verbunden. Hierüber werden dem NeNEB-Chip die Bilddaten entweder direkt vom Digitalisiersystem oder von der Auflösungsreduktion geliefert.

Zusätzlich zu der Beschreibung der Schnittstellen wird die Synchronisation der in den NeNEB-Chip führenden Signale erläutert.

6.7.1 Systemschnittstelle und interne Takterzeugung

Signalname	Rich-tung	Beschreibung
clrn	in	asynchroner low-aktiver Rücksetzeingang
clk	in	Takteingang
eprom	in	asynchroner Auswahleingang für externen Speicher
Legende: in - Eingang		

Tabelle 6.5: Signale der Systemschnittstelle

In Tabelle 6.5 sind alle Signale der Systemschnittstelle aufgeführt. Über den Eingang *eprom* läßt sich festlegen, ob für den externen Gewichtespeicher ein EPROM (*eprom = 1*) oder ein RAM (*eprom = 0*) verwendet wird. Der asynchrone Rücksetzeingang *clrn* dient dazu, alle Register auf dem NeNEB-Chip zu löschen und den Chip den Anfangszustand zu versetzen. An den Takteingang *clk* kann direkt der Systemtakt des Klassifikationssystems (vgl. Bild 1.1) oder ein entsprechend heruntergeteilter Systemtakt angeschlossen werden. Intern werden aus diesem Systemtakt insgesamt sechs Taktsignale gemäß Bild 6.20 und Tabelle 6.6 gebildet, wobei Operationswerk und Steuerwerk jeweils um eine halbe Taktperiode versetzte Takte verwenden (Zweiflankensteuerung). Der Grund für jeweils 3 unterschiedliche Takte für Steuerwerk und Operationswerk ist die pipelineartige Struktur des NeNEB-Chips, wobei die einzelnen Teile der Pipeline unterschiedliche Verarbeitungsgeschwindigkeiten haben.

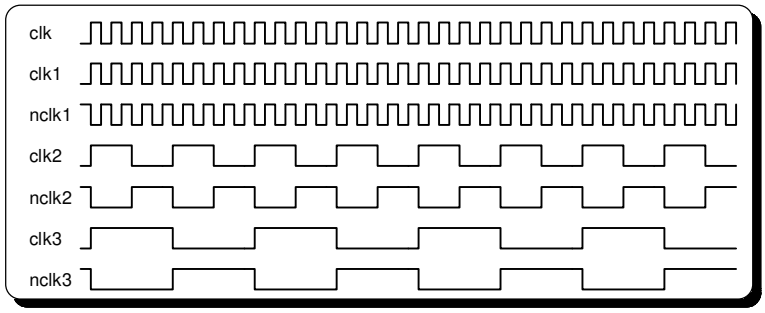


Bild 6.20: interne Takte des NeNEB-Chips in Relation zum Systemtakt *clk*

Parameter	Symbol	Wert
Steuerwerktakt 1	clk1	clk
Operationswerktakt 1	nclk1	¬clk
Steuerwerktakt 2	clk2	¼ clk1
Operationswerktakt 2	nclk2	¬clk2
Steuerwerktakt 3	clk3	½ clk2
Operationswerktakt 3	nclk3	¬clk3

Tabelle 6.6: Taktraten der internen Taktsignale

Der Vorteil der versetzten Takte besteht in der zeitlichen Entkopplung von Steuerwerk und Operationswerk, d. h. Signale, die bei der steigenden Flanke des Steuerwerktaktes im Steuerwerk generiert werden, werden erst bei der fallenden Flanke des Steuerwerktaktes bzw. der zeitgleichen steigenden Flanke des Operationswerktaktes vom Operationswerk ausgewertet. Sofern das Operationswerk wieder eine Rückmeldung auf das erhaltene Signal an das Steuerwerk liefert, kann das Steuerwerk dieses bereits bei der nächsten steigenden Flanke des

Steuerwerktaktes auswerten. Dies bedeutet, daß das Steuerwerk mit jedem Takt einen Zustandsübergang ausführen kann und nicht erst zwei Takte später, wie dies bei nicht versetzten Takten der Fall wäre (Bild 6.21). Signale zwischen Steuerwerk und Operationswerk müssen deshalb nach maximal einer halben Taktperiode stabil sein, da das Gesamtsystem (Steuerwerk und Operationswerk) wegen der Ausnutzung beider Taktflanken effektiv mit dem doppelten Takt arbeitet.

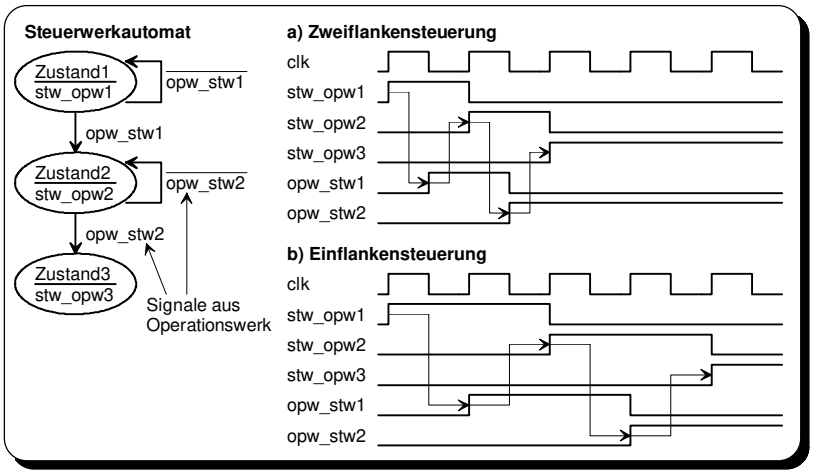


Bild 6.21: Zweiflankensteuerung vs. Einflankensteuerung

Die Erzeugung der sechs Taktsignale erfolgt durch einen asynchronen Taktteiler (Bild 6.22) aus vorderflankengesteuerten D-Flip-Flops. Hierdurch werden *Hazards*[1] auf den Taktsignalen vermieden. Solche Hazards können bei einem synchronen Taktteiler, der auch als Zähler aufgefaßt werden kann, durch Laufzeitverzögerungen entstehen. Als Abhilfe gegen die Hazards bei einem synchronen Teiler könnte eine einschrittige Zustandskodierung verwendet werden. Da jedoch ein synchroner Teiler einen größeren Hardwareaufwand bedeutet als ein asynchroner Teiler, wurde der asynchronen Variante der Vorzug gegeben, wenngleich hier die einzelnen Taktflanken aufgrund der Durchlaufverzögerung der einzelnen Flip-Flops geringfügig versetzt sind (< 0,5 ns).

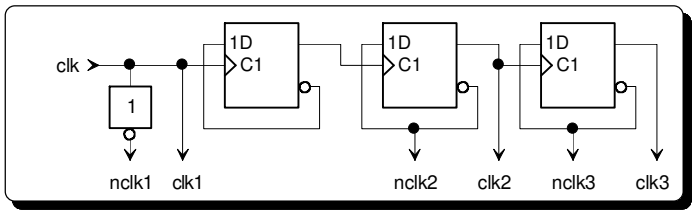


Bild 6.22: asynchroner Taktteiler zur Takterzeugung

6.7.2 Synchronisation externer Signale

Alle in den NeNEB-Chip führenden externen Signale werden mit dem entsprechenden internen Verarbeitungstakt synchronisiert. Dieses ist erforderlich, da die Signale in der Regel von Steuerwerken mit mehreren Flip-Flops als Speicherglieder verarbeitet werden. Aufgrund unterschiedlicher Signallaufzeiten auf dem Chip besteht andernfalls die Gefahr, daß ein aktives Signal von einem Flip-Flop bei der aktiven Taktflanke bereits erkannt wird, von einem anderen

[1] *engl. Hazard* ≈ die Gefahr, das Wagnis

Flip-Flop desselben Steuerwerkes jedoch nicht. Dieses würde zu einem völlig unvorhersehbaren und inkonsistenten Verhalten des Steuerwerkes führen.

Die Synchronisation geschieht durchweg über vorderflankengesteuerte D-Flip-Flops. Diese übernehmen den am Daten-(D)-Eingang anliegenden Wert nur bei der steigenden Flanke des Taktsignals. Als Taktsignal wurde für die Synchronisiereinheit der Operationswerktakt gewählt, da dieser um eine halbe Taktperiode phasenverschoben zum Steuerwerktakt ist. Hierdurch ist gewährleistet, daß die Signale bei Auswertung durch das Steuerwerk stabile Pegel haben. Das folgende Bild 6.23 zeigt die synchronisierten Eingangssignale in Relation zum Hauptsystemtakt *clk* sowie dem jeweiligen Synchronisationstakt. Tabelle 6.7 enthält die dazugehörigen Timingdaten.

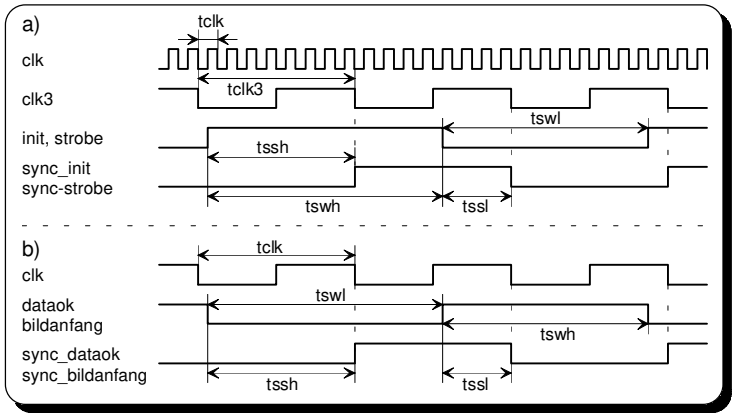


Bild 6.23: Synchronisationstiming der externen Signale

Parameter	Symbol	a)		b)	
		min.	max.	min.	max.
Clock cycle time	tclk	-	-	-	-
Clock 3 cycle time	tclk3	-	-	-	-
Signal width high time	tswh	tclk3	-	tclk	8 tclk
Signal width low time	tswl	tclk3	-	tclk	-
Time from synchronized signal high to signal high	tssh	-	tclk3	-	tclk
Time from synchronized signal low to signal low	tssl	-	tclk3	-	tclk

Tabelle 6.7: Timingdaten zu Bild 6.23

6.7.3 Hostrechnerschnittstelle

Die Signale der Hostrechnerschnittstelle zeigt Tabelle 6.8. Es gibt zwei Betriebsarten der Hostrechner-Schnittstelle:

1. Programmierung des NeNEB-Chips (Schreiben der Initialisierungsdaten)
2. Lesen der Ausgabedaten nach vorheriger Auswahl.

In der ersten Betriebsart wird auf die Hostrechnerschnittstelle nur geschrieben (siehe Timing in Bild 6.24 und Tabelle 6.9), d. h. der Datenbus ist als Eingang geschaltet. Die 16-Bit-Initialisierungsdaten werden sequentiell auf den Datenbus (*databus*) gegeben und die Gültigkeit der Daten durch ein Quittungssignal (*strobe*) angezeigt. Das Quittungssignal wird intern, wie im vorigen Kapitel beschrieben, synchronisiert (*sync_strobe*, vgl. Timing in Bild 6.23 und

Tabelle 6.7). Anschließend zeigt der NeNEB-Chip durch das *busy*-Signal an, daß er beschäftigt ist. Dieses Signal wird maximal eine Taktperiode nach Aktivierung des Quittungssignals aktiviert. Ein erneutes Datenwort darf frühestens eine halbe Taktperiode nach Deaktivierung des Quittungssignals angelegt werden, da die Daten stets bei der Vorderflanke des Taktsignals (*clk3*) übernommen werden (gestrichelte Linien in Bild 6.24). Der Algorithmus zum Schreiben der Initialisierungsdaten könnte etwa wie folgt lauten:

```
aktiviere init
warte mindestens eine Taktperiode tclk3;
deaktiviere init;
warte mindestens eine Taktperiode tclk3;
REPEAT
 lege Daten auf den Datenbus;
 aktiviere strobe;
 warte mindestens eine Taktperiode tclk3;
 deaktiviere strobe;
 warte mindestens eine halbe Taktperiode tclk3;
 REPEAT UNTIL busy = 0
UNTIL alle Daten gesendet;
```

Signalname	Richtung	Beschreibung
init	in	bringt den NeNEB-Chip in den Initialisierungsmodus
strobe	in	kennzeichnet am *databus* anliegende gültige Daten
busy	out	signalisiert den Zustand "beschäftigt", d. h. der Chip kann zur Zeit keine neuen Daten am *databus* empfangen
ready	out	signalisiert den Abschluß des Initalisierungsvorgangs und die Bereitschaft zum Empfang von Bilddaten
newdata	out	signalisiert, daß neue Ausgabedaten vorliegen
datavalid	out	signalisiert gültige Ausgabedaten auf dem *databus*
databus	bidir	bidirektionaler 16-Bit-Datenbus
Legende: in - Eingang out - Ausgang bidir - bidirektional		

Tabelle 6.8: Signale der Hostrechner-Schnittstelle

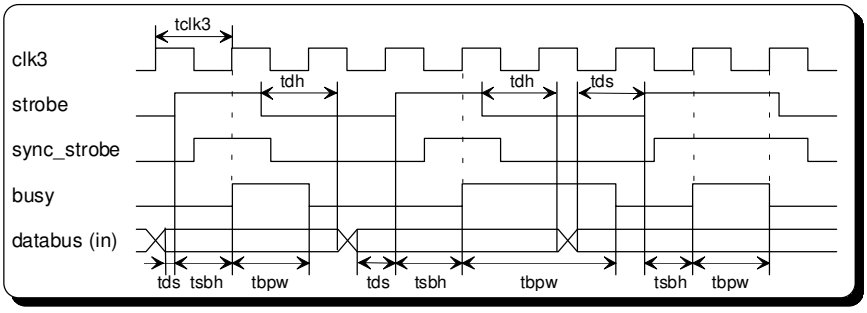


Bild 6.24: Timing der Hostrechner-Schnittstelle (Schreiben der Initialisierungsdaten)

Parameter	Symbol	min.	max.
Clock 3 cycle time	tclk3	-	-
Data setup time	tds	0	-
Data hold time	tdh	½ tclk3	-
Time from busy high to strobe high	tsbh	-	tclk3
Busy pulse width time	tbpw	tclk3	-

Tabelle 6.9: Timingdaten zu Bild 6.24

Das folgende Bild 6.25 zeigt einen Ausschnitt aus dem Eingangsteil der Hostrechner-Schnittstelle mit D-Flip-Flop (*strobe_sync*) zur Synchronisierung des Quittungssignals *strobe*. Während das D-Flip-Flop mit dem Operationswerktakt *nclk3* arbeitet, wird das nachfolgende Eingangsregister *datareg* vom Steuerwerktakt *clk3* getaktet. Der Grund für diese Taktung des Eingangsregisters liegt darin, daß alle nachfolgenden Register des Operationswerkes, die alle mit dem Operationswerktakt getaktet werden, bei der aktiven Taktflanke an ihren Eingängen stabile Signale benötigen. Daher wird der Wert auf dem Datenbus *databus* durch das Eingangsregister bereits eine halbe Taktperiode früher übernommen, womit die Stabilität der Signale für die nachfolgenden Register des Operationswerkes gewährleistet ist.

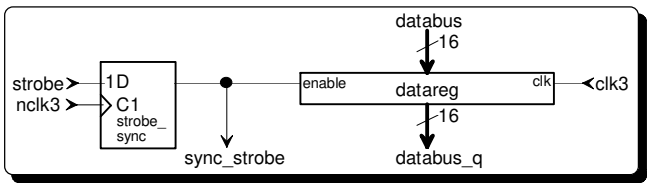


Bild 6.25: Ausschnitt aus dem Eingangsteil der Hostrechner-Schnittstelle

Nach Abschluß der Initialisierung aktiviert der NeNEB-Chip das Signal *ready* und zeigt so seine Bereitschaft zur Entgegennahme von Bilddaten an. Der niederwertige Teil des Datenbusses (D7-D0) wird nun bis zum Vorliegen der ersten Berechnungsergebnisse hochohmig (*tristate*) geschaltet, während der höherwertige Teil des Datenbusses (D15-D8) stets als Eingang konfiguriert bleibt. Dieser höherwertige Teil fungiert jetzt in der zweiten Betriebsart (Bild 6.26 und Tabelle 6.11) als Adreßeingang zur Auswahl des jeweiligen Ausgaberegisters bzw. Ausgabeneurons. Da der NeNEB-Chip nur über 32 Ausgabeneuronen verfügt, werden nur die fünf (2^5 = 32) Bits D12-D8 zur Auswahl eines Ausgaberegisters verwendet (vgl. Tabelle 6.10). Die Belegung der verbleibenden drei Bits D15-D13 ist ohne Bedeutung.

Adresse		**Ausgabe-**
binär	**hexadezimal**	**neuron Nr.**
xxx00000	00	0
xxx00001	01	1
. . .	. . .	. . .
xxx11111	3F	31

Tabelle 6.10: Kodierung der Ausgaberegister

Sofern die ersten Berechnungsergebnisse vorliegen, aktiviert der NeNEB-Chip das Signal *newdata*, welches bis zum Anliegen und Bestätigen einer neuen Adresse aktiviert bleibt. Wenn die Daten auf dem Datenbus gültig sind, wird vom NeNEB-Chip das Signal *datavalid* aktiviert. Auf dem Datenbus liegt nun der Wert der zuletzt angelegten Adresse. Sobald auf dem Datenbus eine neue Adresse anliegt und mit dem Quittungssignal *strobe* bestätigt wird, wird das Signal *datavalid* deaktiviert und der Datenbus hochohmig geschaltet. Falls das Signal *newdata* noch aktiviert ist, wird dieses eine Taktperiode nach Deaktivierung von *datavalid* ebenfalls deaktiviert.

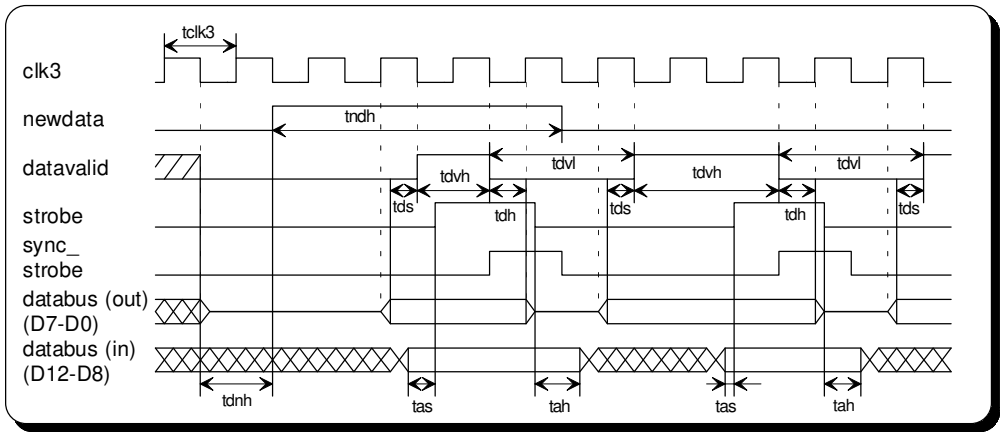


Bild 6.26: Timing der Hostrechner-Schnittstelle (Lesen der Ausgabedaten)

Parameter	Symbol	min.	max.
Clock 3 cycle time	tclk3	-	-
Time from datavalid low to newdata high	tdnh	tclk3	
Newdata high time	tndh	4 tclk3	-
Data setup time	tds	-	½ tclk3
Data hold time	tdh	½ tclk3	-
Datavalid high time	tdvh	tclk3	-
Datavalid low time	tdvl	2 tclk3	-
Address setup time	tas	0	-
Address hold time	tah	½ tclk3	-

Tabelle 6.11: Timingdaten zu Bild 6.26

6.7.4 Speicherschnittstelle

Signalname	Rich-tung	Beschreibung
nmemoe	out	low-aktives Lesesignal
nmemwr	out	low-aktives Schreibsignal
memaddress	out	20-Bit-Adressbus
membus	bidir	bidirektionaler 8-Bit-Datenbus
Legende: out - Ausgang bidir - bidirektional		

Tabelle 6.12: Signale der Speicherschnittstelle

Tabelle 6.12 zeigt die Signale der Speicherschnittstelle. Mit den vorhandenen Signalleitungen läßt sich ein statisches RAM oder ein EPROM direkt ansteuern. Die beiden Steuersignale *nmemoe* und *nmemwr* sind, wie bei handelsüblichen Speicherbausteinen in der Regel auch, low-aktiv.

Sofern als externer Speicher ein RAM verwendet wird, wird dieser in der Initialisierungsphase mit den vom Hostrechner empfangenen Gewichten programmiert. Bild 6.27 und Tabelle 6.13 zeigen hierzu das Schreibtiming der Speicherschnittstelle. Der Speicherdatenbus *membus* ist in diesem Fall während der gesamten Initialisierungsphase als Ausgang konfiguriert. Während der

Berechnungsphase hingegen werden die Daten aus dem externen Speicher nur gelesen. Der Speicherdatenbus *membus* ist daher ständig als Eingang geschaltet, und die Leseleitung *nmemoe* des Speichers ist ständig aktiv. Die Daten aus dem Speicher werden jeweils bei der fallenden Flanke des Taktsignals übernommen, und im Anschluß daran wird eine neue Adresse erzeugt. Das Lesetiming zeigen Bild 6.28 und Tabelle 6.14.

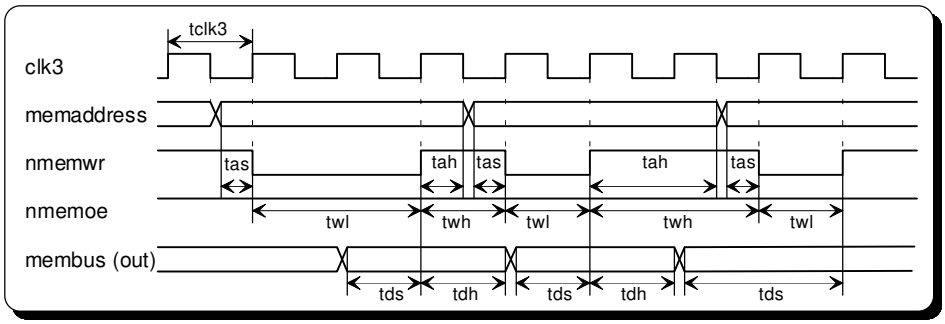


Bild 6.27: Timing der Speicherschnittstelle (Schreiben)

Parameter	Symbol	min.	max.
Clock 3 cycle time	tclk3	-	-
Address setup time	tas	-	½ tclk3
Address hold time	tah	½ tclk3	-
Data setup time	tds	½ tclk3	-
Data hold time	tdh	tclk3	-
Write enable low time	twl	tclk3	-
Write enable high time	twh	tclk3	-

Tabelle 6.13: Timingdaten zu Bild 6.27

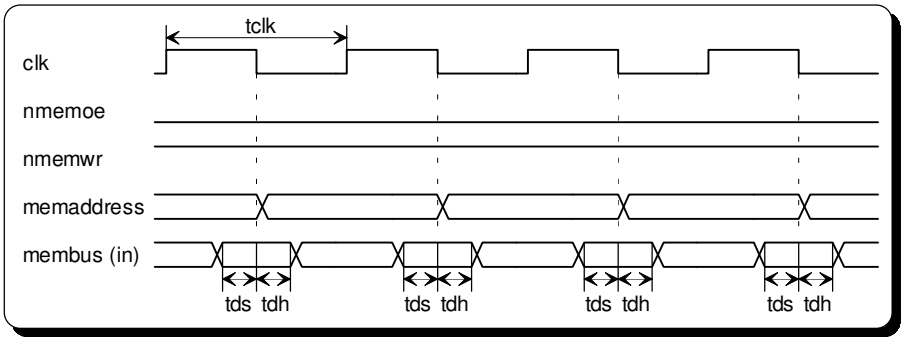


Bild 6.28: Timing der Speicherschnittstelle (Lesen)

Parameter	Symbol
Clock cycle time	tclk
Data setup time	tds
Data hold time	tdh

Tabelle 6.14: Timingdaten zu Bild 6.28

6.7.5 Bilddatenschnittstelle

Signalname	Richtung	Beschreibung
bildanfang	in	low-aktives Signal zur Kennzeichnung des Bildanfangs
dataok	in	low-aktives Signal zur Kennzeichnung eines gültigen Pixelwertes
xi	in	8-Bit-Datenbus für die Pixelwerte
Legende: in - Eingang		

Tabelle 6.15: Signale der Bilddatenschnittstelle

Die Signale der Bilddatenschnittstelle sind in Tabelle 6.15 aufgeführt. Die Arbeitsweise der Bilddatenschnittstelle ähnelt stark der der Hostrechnerschnittstelle. Das Signal *bildanfang* ist vergleichbar mit dem Signal *init*, und das Signal *dataok* ist vergleichbar mit dem Signal *strobe*. Der Unterschied zur Hostrechnerschnittstelle ist, daß die Bilddatenschnittstelle kein *busy*-Signal besitzt, da der Bilddatenstrom einer Kamera nicht einfach angehalten werden kann. Der NeNEB-Chip muß daher in der Lage sein, die Bilddaten mit der von der Kamera gelieferten Rate in Echtzeit zu verarbeiten, wobei die maximale Pixeldatenrate durch die Spezifikation des NeNEB-Chips festgelegt ist. Hiernach benötigt der NeNEB-Chip zur Verarbeitung eines Bilddatenbytes eine Zeitdauer von 17 Taktzyklen *tclk* (*tdp*). Diese Zeitdauer wird durch die benötigte Zeitdauer für eine Multiplikation der verborgenen Neuronen bestimmt (vgl. Kapitel 6.8.3 und Anhang B.3.3). Die Zeitdauer zwischen zwei Bildern (*tbp*) wird durch die langsamste Komponente (vgl. Kapitel 6.8.5 und Anhang B.3.5) der Berechnungseinheiten im Bilddatenpfad (vgl. Bild 6.2) bestimmt und beträgt 1106 Taktzyklen. Somit können Bilder mit einer minimalen Größe von

$$\frac{tbp}{tdp} = \frac{1106}{17} = 65,06 \approx 66 \text{ Pixeln}$$

bei maximaler Pixeldatenrate direkt aufeinanderfolgen. Bei kleineren Bildern und maximaler Pixeldatenrate muß eine Wartepause eingelegt werden, um den NeNEB-Chip nicht mit Daten zu "überfahren". Das Timing der Bilddatenschnittstelle zeigt das folgende Bild 6.29 sowie Tabelle 6.16.

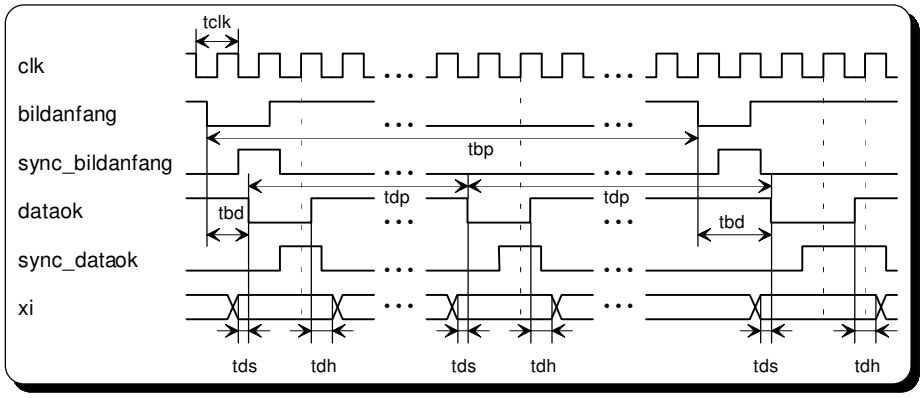


Bild 6.29: Timing der Bilddatenschnittstelle

Parameter	Symbol	min.	max.
Clock cycle time	tclk	-	-
Time between *bildanfang* pulses	tbp	1106 tclk	-
Time from *bildanfang* high to *dataok* high	tbd	tclk	-
Time between *dataok* pulses	tdp	17 tclk	-
Data setup time	tds	0	-
Data hold time	tdh	½ tclk3	-

Tabelle 6.16: Timingdaten zu Bild 6.29

Bild 6.30 zeigt die Realisierung der Bilddatenschnittstelle sowie das Pufferregister (*xireg*) für die Eingabewerte der verborgenen Neuronen. Das Eingangsregister (*inputreg*) der Bilddatenschnittstelle wird wie das Eingangsregister der Hostrechnerschnittstelle vom Steuerwerktakt getaktet, während das nachfolgende *xireg* vom Operationswerktakt getaktet wird. Die Freigabe des Pufferregisters erfolgt durch den Steuerautomaten für den ersten Teil der Erregungsfunktion der verborgenen Neuronen. Somit ist gewährleistet, daß der Eingabewert für die gesamte Zeitdauer einer Multiplikation zur Verfügung steht.

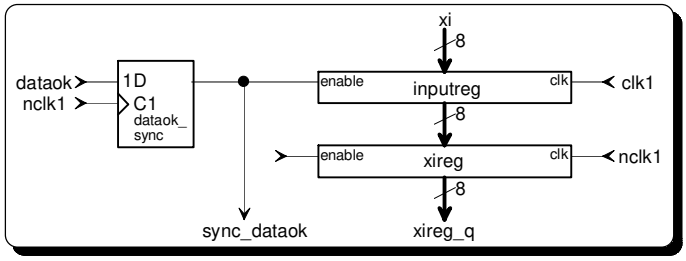


Bild 6.30: Realisierung der Bilddatenschnittstelle

6.8 Steuerwerk

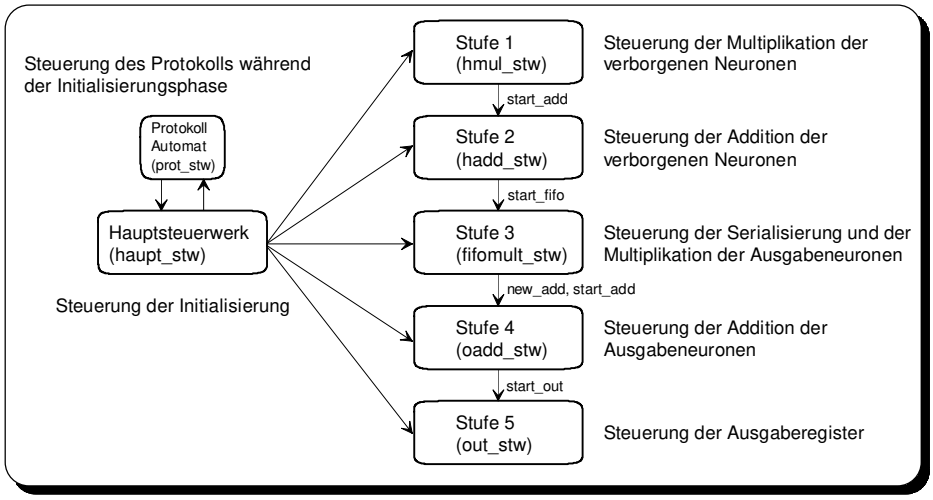


Bild 6.31: Übersicht über die einzelnen Teile des Steuerwerkes beim NeNEB-Chip

Um eine möglichst hohe Verarbeitungsgeschwindigkeit zu erzielen, wurde für den NeNEB-Chip eine Pipelinearchitektur gewählt. Der Vorteil der Pipelinearchitektur ist, daß die einzelnen Komponenten der Pipeline weitgehend parallel arbeiten können. Dementsprechend wurde das

Steuerwerk des NeNEB-Chips in mehrere voneinander unabhängig arbeitende Stufen aufgeteilt. Diese Stufen steuern jeweils bestimmte funktional zusammenhängende Teile des Operationswerkes, wobei die Taktraten der einzelnen Steuerwerksstufen mit denen der entsprechenden Operationswerkteile übereinstimmt. Die Festlegung der Taktraten erfolgte dabei anhand der geschätzten Verzögerungszeiten der synthetisierten VHDL-Modelle, wobei die Modelle jeweils hinsichtlich einer möglichst geringen Chipfläche synthetisiert wurden. Ausnahmen hiervon bilden nur die beiden ersten Stufen zur Berechnung der Multiplikation und der Addition der verborgenen Neuronen, da hiervon die Gesamtgeschwindigkeit des NeNEB-Chips bestimmt wird.

6.8.1 Synchronisation der einzelnen Steuerwerksstufen

Wie aus Bild 6.31 ersichtlich, besteht das Steuerwerk des NeNEB-Chips aus mehreren voneinander unabhängigen Stufen, die zum Teil mit unterschiedlichen Takten arbeiten. Daher sind verschiedene Synchronisationsmaßnahmen der in die Steuerwerke führenden Signale notwendig. Es kann dabei zwischen drei verschiedenen Fällen unterschieden werden:

- **Synchronisation von Signalen zwischen zwei Steuerwerken mit gleichem Takt**

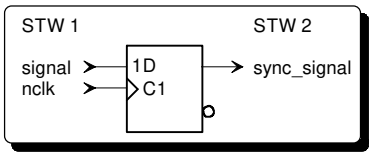


Bild 6.32: Synchronisation von Signalen zwischen zwei Steuerwerken (*STW 1* und *STW 2*) mit gleichem Takt

In diesem Fall dient die Synchronisation nur der korrekten zeitlichen Entkopplung beider Steuerwerke, indem dafür gesorgt wird, daß sich das Signal nur während der fallenden (nicht aktiven) Taktflanke des Steuerwerkes ändert. Hierfür wird zur Synchronisation ein D-Flip-Flop verwendet, das mit dem negierten Steuerwerktakt bzw. dem Operationswerktakt getaktet wird (Bild 6.32).

- **Synchronisation von Signalen zwischen zwei Steuerwerken mit unterschiedlichem Takt**

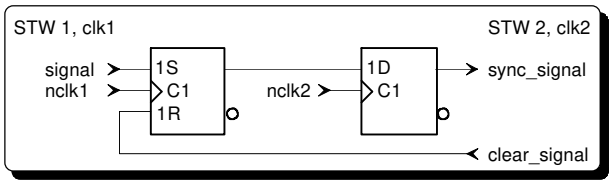


Bild 6.33: Synchronisation von Signalen zwischen zwei Steuerwerken (*STW 1* und *STW 2*) mit unterschiedlichem Takt (*clk1* und *clk2*)

Bei diesem Fall ist die Taktfrequenz von Steuerwerk 1 immer höher als die von Steuerwerk 2. Je nachdem in welchem Zustand Steuerwerk 2 sich gerade befindet, besteht hierbei die Gefahr, daß das Signal von Steuerwerk 1 nicht registriert wird. Daher wird dieses Signal in einem RS-Flip-Flop, das mit dem negierten Takt von Steuerwerk 1 arbeitet, zwischengespeichert. Dieses Flip-Flop wird von Steuerwerk 2 erst gelöscht, wenn das

Signal verarbeitet worden ist. Zusätzlich wird das in dem RS-Flip-Flop zwischengespeicherte Signal noch, wie im vorigen Absatz beschrieben, mit dem Takt von Steuerwerk 2 synchronisiert (Bild 6.33).

- **Synchronisation von Signalen zwischen Operationswerk und Steuerwerk mit unterschiedlichem Takt**

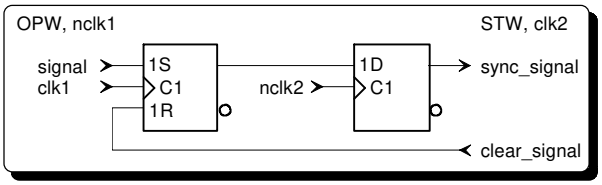


Bild 6.34: Synchronisation von Signalen zwischen Operationswerk und Steuerwerk (*OPW* und *STW*) mit unterschiedlichem Takt (*clk1* und *clk2*)

Dieser Fall ist nahezu identisch mit dem vorhergehenden Fall. Der Unterschied ist nur, daß das Signal hierbei vom Operationswerk erzeugt wird. Daher wird das RS-Flip-Flop mit dem entsprechenden negierten Takt des Operationswerkes betrieben (Bild 6.34).

6.8.2 Hauptsteuerwerk und Protokollautomat

lfd. Nr.	Beschreibung
1	Anzahl der Eingabestellen (16 Bit)
2	Schiebewert *S* der Ausgabefunktion
3	positiver Schwellwert +*SW* der Ausgabefunktion
4	negativer Schwellwert -*SW* der Ausgabefunktion
5	Offset 1 der Ausgabefunktion
6	Offset 2 der Ausgabefunktion
7	Offset 3 der Ausgabefunktion
8	Schwellwert θ des Ausgabeneurons Nr. 31
9	Gewicht $\omega_{31,7}$ vom verborgenen Neuron Nr. 7 zum Ausgabeneuron Nr. 31
⋮	⋮
16	Gewicht $\omega_{31,0}$ vom verborgenen Neuron Nr. 0 zum Ausgabeneuron Nr. 31
17	Schwellwert θ des Ausgabeneurons Nr. 30
18 - 25	Gewichte des Ausgabeneurons Nr. 30
⋮	⋮
287	Schwellwert θ des Ausgabeneurons Nr. 0
288 - 295	Gewichte des Ausgabeneurons Nr. 0
296	Schwellwert θ des verborgenen Neurons Nr. 7
⋮	⋮
303	Schwellwert θ des verborgenen Neurons Nr. 0
304 - 319	Gewichte für Eingabestelle Nr. 0 (x_0)
⋮	⋮

Tabelle 6.17: Reihenfolge der Initialisierungsdaten

Das Hauptsteuerwerk steuert mit Hilfe des Protokollautomaten die Initialisierung des NeNEB-Chips. Der Protokollautomat sorgt dabei für die Abwicklung des Protokolls auf der Hostrechnerschnittstelle, während das Hauptsteuerwerk die ankommenden Daten zu den einzelnen Registern auf dem NeNEB-Chip bzw. an den externen Gewichtespeicher leitet. Die Reihenfolge der Initialisierungsdaten (Tabelle 6.17) sowie deren Format ist dabei fest vorgegeben. Das Format der einzelnen Daten entspricht direkt den Zahlenformaten aus Kapitel 6.3. Die Reihenfolge der Daten ist angelehnt an das Format der Netzdatei, die von dem in Kapitel 5.4 beschriebenen Programm *bpg* erzeugt wird.

Bis zur lfd. Nr. 303 in Tabelle 6.17 müssen immer alle Initialisierungsdaten vorhanden sein. Die Gewichte und Schwellwerte nicht benötigter Ausgabeneuronen müssen auf Null gesetzt werden. Die Daten für den externen Gewichtespeicher (ab lfd. Nr. 304) sind nur notwendig, wenn das Auswahlsignal *eprom* eine Null führt, andernfalls wird von einem bereits programmierten Speicher ausgegangen. Auch hierbei müssen die entsprechenden Bits nicht verwendeter verborgener Neuronen auf Null gesetzt werden. Eine binäre 16-Bit-Datendatei mit allen Initialisierungsdaten kann mit dem Programm *conv-nn* (Anhang D.1.2) direkt aus einer angegebenen Netzdatei, wie sie vom Programm *bpg* erzeugt wird, erzeugt werden.

Nach Beendigung der Initialisierung gibt das Hauptsteuerwerk die anderen fünf Steuerwerksstufen frei und signalisiert mit dem Signal *ready* die Rechenbereitschaft des NeNEB-Chips.

Das Hauptsteuerwerk wie auch der Protokollautomat arbeiten mit dem langsamsten Systemtakt *clk3*, da die meisten der zu initialisierenden Komponenten mit diesem Takt arbeiten. Nur die für die Adreßerzeugung des externen Speichers benötigten Zähler *c1* und *c2* (siehe folgendes Kapitel) arbeiten normalerweise mit dem vollen Systemtakt *clk* und besitzen deshalb eine steuerbare Taktumschaltung.

6.8.3 Stufe 1 (*hmul_stw*)

Stufe 1 des Steuerwerkes steuert die Multiplikation der verborgenen Neuronen, d. h. die Erregungsfunktion Teil 1 gemäß Kapitel 6.4.1. Dieser Teil des NeNEB-Chip ist der zeitkritischste Teil und wird daher mit dem vollen Systemtakt *clk* betrieben. Um dies möglich zu machen, müssen bei der Synthese des Addierers Vorgaben für die maximale Durchlaufverzögerung gemacht werden (siehe Kapitel 7.3). Zur Durchführung einer Multiplikation nach dem Hornerschema benötigt diese Steuerwerksstufe 15 Taktzyklen (siehe Anhang B.3.3), entsprechend den 15 Bits des Betrages eines Gewichtes. Hinzu kommen noch zwei Steuertakte, wodurch insgesamt 17 Takte für einen Zyklus erforderlich sind. Für eine vorgesehene maximale Bildgröße von 65536 Pixeln und 50 Bildern pro Sekunde wäre somit eine Taktfrequenz von mindestens

$$17 \text{ Takte} \cdot 50 \text{ Bilder} \cdot 65536 \text{ Pixel} = 55,7 \text{ MHz} \approx 60 \text{ MHz} \tag{6.30}$$

notwendig. Nach den vorläufigen Abschätzungen mit den Werten aus der Synthese kann diese Taktfrequenz auch erzielt werden (vgl. Kapitel 7.3).

Die Erzeugung der Adresse für den externen Gewichtespeicher wird ebenfalls durch Stufe 1 kontrolliert, da im entsprechenden Teil des Operationswerkes die Daten aus dem externen Gewichtespeicher benötigt werden. Wie in Kapitel 6.4.3 bereits angedeutet, wird die Adresse zweiteilig erzeugt. Ein Zähler (*c1*) erzeugt dabei die niederwertigen 4 Bits der Adresse, die direkt den einzelnen Bits der Gewichte entsprechen. Das Zählerablaufsignal von *c1* wird dabei in

Stufe 1 des Steuerwerkes ausgewertet und kennzeichnet das Ende der Multiplikation, d. h. alle Bits eines Gewichtes sind verarbeitet. Der zweite Zähler (*c2*) erzeugt die höherwertigen 16 Bits der Adresse, die mit der Pixelnummer korrespondieren. Dessen Zählerablaufsignal wird erst von Stufe 2 des Steuerwerkes ausgewertet.

Die beiden Zähler *c1* und *c2* sowie alle weiteren Zähler auf dem NeNEB-Chip arbeiten als Rückwärtszähler. Der Grund dafür ist, daß alle Zähler entweder von außen oder innerhalb der VHDL-Modelle mit zunächst unbekannten Werten geladen werden. Die Zähler sollen dabei ein Signal (Zählerablaufsignal) abgeben, wenn diese Anzahl Zählimpulse gezählt sind. Würden die Zähler vorwärts zählen, so müßte für das Ablaufsignal ein Vergleicher in der Breite des Zählers realisiert werden. Läßt man die Zähler hingegen vom gewünschten Wert rückwärts bis Null zählen, so braucht für das Ablaufsignal nur ein Vergleich für den Nulldurchgang aus einer Anzahl ODER-Glieder bzw. einem ODER-Glied mit entsprechend vielen Eingängen realisiert zu werden.

Nach Beendigung einer Multiplikation wird mit dem Signal *start_add* die Stufe 2 des Steuerwerkes zur Addition der Multiplikationsergebnisse gestartet. Gleichzeitig wird das Ergebnis der Multiplikation in das Summandenregister von Teil 2 der Erregungsfunktion der verborgenen Neuronen geschrieben. Danach ist die Stufe 1 wieder bereit für eine neue Multiplikation bzw. für den Beginn eines neuen Bildes, unabhängig davon, ob der Zähler *c2* abgelaufen ist oder nicht. Dies bedeutet, daß die Stufe 1 in der Lage ist, sich jeweils nach Ablauf einer Multiplikation auf einen neuen Bildanfang zu synchronisieren. Ein Bild kann vom NeNEB-Chip jedoch nur klassifiziert werden, wenn alle in Zähler *c2* programmierten Pixel empfangen wurden, da ansonsten die weiteren Stufen des Steuerwerkes zur Steuerung der Ausgabeneuronen nicht aktiviert werden.

6.8.4 Stufe 2 (*hadd_stw*)

Von Stufe 2 wird die Addition der verborgenen Neuronen, d. h. die Erregungsfunktion Teil 2 gemäß Kapitel 6.4.2 gesteuert. Diese Stufe hat gemäß den Vorgaben im vorigen Kapitel für ihre Arbeit mindestens 17 Taktzyklen Zeit, jedoch muß hier nur jeweils eine Addition durchgeführt werden. Daher wird diese Stufe mit einem Viertel des Systemtaktes bzw. *clk2* betrieben.

Vor Beginn der Additionen sorgt diese Stufe aufgrund des Signals für den Bildanfang (*bildanfang*) dafür, daß das Summenregister der Erregungsfunktion Teil 2 mit dem Schwellwert θ vorbesetzt wird. Anschließend werden die Multiplikationsergebnisse aus Teil 1 der Erregungsfunktion, gesteuert von dem Signal *start_add* aus Stufe 1, solange aufsummiert, bis der Zähler *c2* abgelaufen ist, oder ein neuer Bildanfang registriert wird. Erst wenn der Zähler *c2* abgelaufen ist, wird das Additionsergebnis in das Barrelshiftregister der Ausgabefunktion der verborgenen Neuronen geladen und die Stufe 3 des Steuerwerkes mit dem Signal *start_fifo* gestartet.

6.8.5 Stufe 3 (*fifomult_stw*)

Die Stufe 3 ist vergleichbar mit Stufe 1, jedoch wird hier die Multiplikation der Ausgabeneuronen gesteuert. Zusätzlich steuert diese Stufe den Serialisierer zur Serialisierung der Ausgaben der verborgenen Neuronen sowie den Ringspeicher mit den Gewichten der Ausgabeneuronen. Die Arbeitsweise dieser Stufe ist nahezu identisch mit Stufe 1. Ebenso wie bei dieser werden

hier zwei Zähler, einer für die Zählung der Bits der Gewichte (*c3*) und einer für die Zählung der verborgenen Neuronen (*c4*), verwendet. Der Unterschied ist, daß hier die Erregung stets mit den Ausgaben aller verborgenen Neuronen berechnet wird. Dies hat zur Folge, daß die Stufe 3 insgesamt 137 Takte[1] für einen kompletten Rechenzyklus benötigt (vgl. Anhang B.3.5). Da diese Stufe im Vergleich zu Stufe 1 jedoch relativ wenig Rechenoperationen, 8 Multiplikationen gegenüber bis zu 65536 Multiplikationen, durchführen muß, erfolgt die Taktung mit dem langsamsten Systemtakt *clk3* bzw. einem Achtel von *clk1*. Daher beträgt der Gesamtzeitbedarf von Stufe 3 insgesamt 1096 Takte *clk1*, weshalb diese Stufe die langsamste Komponente im NeNEB-Chip ist. Dies trifft jedoch nur die Gesamtrechenzeit für ein Bild, d. h. Stufe 3 bestimmt die minimale Zeitdauer zwischen zwei Bildern. Bei größeren Bildern (> 66 Pixel) überwiegt die Rechenzeit der Stufen 1 und 2.

6.8.6 Stufe 4 (*oadd_stw*)

Vergleichbar mit Stufe 2 steuert Stufe 4 die Addition der Ausgabeneuronen. Diese Stufe wird ebenso wie die vorhergehende Stufe 3 mit dem langsamen Systemtakt *clk3* betrieben. Die Initialisierung des Summenregisters erfolgt hier durch das von Stufe 3 gelieferte Startsignal *new_add*, während die Additionen der Multiplikationsergebnisse vom Signal *start_add* aus Stufe 3 veranlaßt werden. Nach Abschluß aller Additionen wird von Stufe 4 mit dem Signal *start_out* die letzte Stufe 5 gestartet.

6.8.7 Stufe 5 (*out_stw*)

Diese letzte Stufe des Steuerwerkes steuert die wahlfreie Ausgabe der Ausgabewerte der Ausgabeneuronen. Zum einen wird hier das entsprechende Protokoll der Hostrechnerschnittstelle für das Lesen der Ausgabedaten abgewickelt, zum anderen lädt diese Stufe auf Veranlassung von Stufe 4 die Ausgaberegister mit neuen Ergebnissen.

[1] 17 Takte/Multiplikation · 8 verborgene Neuronen + 1 Steuertakt = 137 Takte

7 VHDL-Realisierung, Simulation und Synthese

In diesem Kapitel wird zunächst ein grober Abriß über die Sprache VHDL sowie die Realisierung einer integrierten Schaltung mit Hilfe dieser Hardwarebeschreibungssprache gegeben. Anschließend wird über das Vorgehen und die Ergebnisse der funktionalen Simulation der VHDL-Modelle berichtet. Zum Abschluß werden einige Ergebnisse aus der Synthese der VHDL-Modelle vorgestellt, bei denen es sich aber vorläufig nur um Abschätzungen des Synopsys-Systems handelt.

7.1 VHDL-Realisierung eines digitalen Systems

VHDL ist eine Hardwarebeschreibungssprache, die 1983 vom amerikanischen Verteidigungsministerium initiiert wurde und seit Ende 1987 als IEEE Standard 1076 genormt ist [IEE88]. Die prozedurale Sprache dient zur Beschreibung und Simulation digitaler Systeme und deren Umgebung. Alle beim Systementwurf notwendigen Beschreibungen werden dabei vom Sprachumfang abgedeckt. Die Sprache unterstützt die hierarchische Gliederung eines Entwurfs bzw. Designs entsprechend dem *top-down*-Konzept. Hierbei werden die Elemente eines zunächst sehr abstrakten Designs schrittweise immer weiter verfeinert, bis am Ende nur noch elementare strukturelle Einheiten zu finden sind. Für die Beschreibung einer Designeinheit stehen in VHDL drei verschiedene Methoden zur Verfügung:

- **Verhaltensbeschreibung**

  Die Verhaltensbeschreibung eines Designs erfolgt in algorithmischer Form wie in einer höheren Programmiersprache, völlig losgelöst von einer konkreten Schaltungsstruktur.

- **Strukturbeschreibung**

  Bei der Strukturbeschreibung werden die strukturellen Verbindungen der einzelnen Elemente des Designs nach der Art einer "Gatternetzliste" beschrieben.

- **Datenflußbeschreibung**

  Diese Beschreibungsform stellt eine Mischform der beiden vorgenannten Formen dar.

Am Beispiel der nachfolgenden VHDL-Modelle für ein Barrel-Schieberegister und für ein einfaches Register variabler Breite sollen die wesentlichen Merkmale dieser Hardwarebeschreibungssprache kurz vorgestellt werden. Die nachstehenden Ausführungen sollen nur einen kurzen Überblick über die Hardwarebeschreibungssprache VHDL geben und erheben keineswegs einen Anspruch auf Vollständigkeit, da der gesamte Sprachumfang von VHDL aufgrund der vielen möglichen Einsatzbereiche und Anwendungsfälle außerordentlich komplex ist.

```
-------------------------------------------------------------------------
-- Barrelshiftregister variabler Breite (Parameter "breite")
-- asynchroner low-aktiver-Reset-Eingang ("clrn")
-- synchroner Load-Enable-Eingang ("enable")
-- Parameter sel bestimmt die Shiftweite nach rechts um 0 bis 7 Stufen
-------------------------------------------------------------------------

library ieee;
use IEEE.std_logic_arith.conv_std_logic_vector;
use IEEE.std_logic_1164.all;

entity barrel is
 generic (
 breite : integer:= 13 -- legt die Bitbreite des Barrelshifters fest
);
 port (
 clrn  : in std_logic;
 clk   : in std_logic;
 enable: in std_logic;
 sel   : in std_logic_vector (2 downto 0);
 d     : in std_logic_vector ((breite-1) downto 0);
 q     : out std_logic_vector ((breite-1) downto 0);
end barrel;

architecture barrel_arc of barrel is

component reg -- Deklaration des Registers
 generic (
 breite : integer:= 13);
 port (
 clrn  : in std_logic;
 clk   : in std_logic;
 enable: in std_logic;
 reset : in std_logic;
 d     : in std_logic_vector ((breite-1) downto 0);
 q     : out std_logic_vector ((breite-1) downto 0));
end component;

signal int_d : std_logic_vector ((breite-1) downto 0);
signal constnull : std_logic := '0';

begin

 with sel select -- Festlegung der Verschiebung gemäß Prameter "sel"
  int_d <= d when "000",
   (d (breite-1) & d (breite-1 downto 1)) when "001",
   (d (breite-1) & d (breite-1) & d (breite-1 downto 2)) when "010",
   (d (breite-1) & d (breite-1) & d (breite-1)
    & d (breite-1 downto 3)) when "011",
   (d (breite-1) & d (breite-1) & d (breite-1) & d (breite-1)
    & d (breite-1 downto 4)) when "100",
   (d (breite-1) & d (breite-1) & d (breite-1) & d (breite-1)
    & d (breite-1) & d (breite-1 downto 5)) when "101",
   (d (breite-1) & d (breite-1) & d (breite-1) & d (breite-1)
    & d (breite-1) & d (breite-1) & d (breite-1 downto 6)) when "110",
   (d (breite-1) & d (breite-1) & d (breite-1) & d (breite-1)
    & d (breite-1) & d (breite-1) & d (breite-1)
    & d (breite-1 downto 7)) when "111",
   d when others;

 barrelreg : reg generic map (breite) -- Instanziierung des Registers
        port map (clrn => clrn, clk => clk, enable => enable,
                  reset => constnull, d => int_d, q => q);

end barrel_arc;
```

```
-------------------------------------------------------------------------
-- Register variabler Breite (Parameter "breite")
-- asynchroner low-aktiver-Reset-Eingang ("clrn")
-- synchroner Load-Enable-Eingang ("enable")
-- synchroner high-aktiver Reset-Eingang mit Vorrang ("reset")
-------------------------------------------------------------------------

library ieee;
use IEEE.std_logic_arith.conv_std_logic_vector;
use IEEE.std_logic_1164.all;

entity reg is
 generic (
 breite : integer:= 8);
 port (
 clrn  : in std_logic;
 clk   : in std_logic;
 enable: in std_logic;
 reset : in std_logic;
 d     : in std_logic_vector ((breite-1) downto 0);
 q     : out std_logic_vector ((breite-1) downto 0);
end reg;

architecture reg_arc of reg is
begin
 reg_proc : process (clrn, clk, enable, reset, d)
  begin
   if clrn = '0' then -- asynchronen Reset abfragen
    q <= conv_std_logic_vector (0,breite);
   elsif clk'event and clk = '1' then -- Abfrage auf steigende
                                      Taktflanke
    if reset = '1' then -- synchronen Reset abfragen
     q <= conv_std_logic_vector (0,breite);
    elsif enable = '1' then -- synchrones Enable abfragen
     q <= d;
    end if;
   end if;
  end process;
end reg_arc;
```

Eine VHDL-Beschreibung bzw. ein VHDL-Modell oder allgemein ein Design beginnt meist mit der Angabe spezifischer Bibliotheken (*library*- und *use*-Anweisung), die von diesem Design verwendet werden. Anschließend erfolgt die Beschreibung der Schnittstelle des Designs (*entity*-Anweisung). In der Schnittstelle sind alle Signale sowie deren Richtung (*in*-Eingang, -*out*-Ausgang, *inout*-bidirektional) und deren Typ (*integer*, *bit*, *boolean*, *std_logic* etc.) aufgeführt, mit denen das Design mit anderen Designs in Kontakt treten kann (*port*-Anweisung). Zusätzlich kann ein Design innerhalb der Schnittstellendefinition generische Werte zur Parametrisierung enthalten (*generic*-Anweisung). Über solche generischen Parameter kann z. B. wie im Falle des Barrel-Schieberegisters die Breite des Registers von außen bzw. durch ein Design höherer Ebene festgelegt werden. Weitere Verwendungsmöglichkeiten generischer Parameter bestehen in der Festlegung der Verzögerungszeiten eines Designs, wie z. B. eines Speichermodells mit verschiedenen Zugriffszeiten.

Den Hauptteil eines VHDL-Modells bildet die *architecture*, wobei eine *entity* mehrere *architectures* haben kann. Hier wird das Design in Form einer Verhaltens- oder Strukturbeschreibung bzw. einer Kombination der beiden beschrieben. Die Zuordnung einer *architecture* zu einer *entity* erfolgt durch eine Konfigurationsanweisung (*configuration*-Anweisung).

Vor der eigentlichen Verhaltens- oder Strukturbeschreibung können Typen (Schlüsselworte *type* bzw. *subtype*), Konstanten (Schlüsselwort *constant*) und Signale (Schlüsselwort *signal*) für

die Verwendung innerhalb des Anweisungsteils der *architecture* definiert werden. Zusätzlich werden hier die Schnittstellen von benötigten anderen Designs deklariert, wobei diese Deklarationen mit der *entity* der deklarierten Designs übereinstimmen muß (*component*-Anweisung). Die Beschreibung des eigentlichen Designs erfolgt nun im Anweisungsteil der *architecture*, wobei zunächst alle Anweisungen parallel ausgeführt werden. Anweisungen, die seriell ausgeführt werden sollen, lassen sich innerhalb eines Prozesses zusammenfassen (*process*-Anweisung). Dabei werden alle Anweisungen innerhalb des Prozesses seriell ausgeführt, mehrere Prozesse innerhalb der *architecture* jedoch parallel. Die einzelnen Prozesse können nur über die im Deklarationsteil der *architecture* deklarierten Signale kommunizieren.

Ein Prozeß hat im Regelfall eine Sensivitätsliste (*sensitivy-list*), in der alle Eingangssignale des Prozesses aufgeführt sind. Innerhalb des Deklarationsteils eines Prozesses (zwischen den Schlüsselworten *process* und *begin*) lassen sich wieder Konstanten und Typen sowie Variablen (Schlüsselwort *variable*) deklarieren, jedoch keine Signale. Hierbei wird auch der Unterschied zwischen Variablen und Signalen deutlich. Zuweisungen an Variablen werden entsprechend der seriellen Ausführungsfolge im Prozeß sofort wirksam. Hingegen werden Zuweisungen an Signale erst bei einer *wait*-Anweisung oder im Falle einer Sensitivitätsliste am Ende des Prozesses wirksam, wobei nur eine der beiden Möglichkeiten erlaubt ist. Die *wait*-Anweisung dient dabei zur zeitlichen Kontrolle des Prozesses. Mit ihr kann auf ein bestimmtes Ereignis gewartet werden (*wait on signal_list*), auf die Erfüllung einer Bedingung gewartet werden (*wait until condition*), für eine bestimmte Zeit gewartet werden (*wait for time_expression*) oder der Prozeß dauerhaft angehalten werden (*wait* alleine), da Prozesse stets immer aktiv sind. Eine weitere Möglichkeit für die zeitliche Kontrolle innerhalb eines Prozesses bzw. auch innerhalb der *architecture* ist die verzögerte Signalzuweisung

```
signal_name <= expression after time_expression; .
```

Sowohl innerhalb der Prozesse als auch innerhalb der *architecture* stehen verschiedene leistungsfähige Anweisungen wie bei einer höheren Programmiersprache zur Verfügung. Die Syntax der Anweisungen ist jedoch verschieden, je nachdem, ob es sich dabei um eine Anweisung für einen Prozeß oder um eine Anweisung innerhalb der *architecture* handelt. Das folgende Beispiel einer bedingten Verzweigung verdeutlicht dies:

```
-- im Prozess
IF condition THEN
 sequential_statements
{ELSIF
 sequential_statements}
{ELSE
 sequential_statements}
END IF;

-- in der architecture
signal_name <= expression WHEN condition ELSE
               {expression WHEN condition ELSE}
               expression;
```

Sofern im Deklarationsteil der *architecture* andere Designs deklariert worden sind, müssen diese zur endgültigen Verwendung innerhalb des Anweisungsteils noch instanziiert werden. Die Instanziierung geschieht dabei ähnlich wie Prozeduraufrufe in höheren Programmiersprachen. Die generischen Parameter und die Signale werden dabei in getrennten Listen unter Verwendung der jeweiligen Schlüsselworte (*generic map* bzw. *port map*) aufgeführt. Die Übergabe der Parameter kann entweder nach Reihenfolge nur über die aktuellen Parameter erfolgen oder wie im Beispiel implizit durch Zuweisung des aktuellen Parameters an den formalen Parameter.

Da VHDL eine prozedurale Sprache ist, können bestimmte Anweisungen zu Unterprogrammen zusammengefaßt werden. Hierfür stehen Funktionen mit genau einem Rückgabewert (Schlüsselwort *function*) und Prozeduren (Schlüsselwort *procedure*) zur Verfügung, wobei die Ausführung innerhalb der Unterprogramme sequentiell erfolgt. Die Unterprogramme, wie auch Typen- oder Konstantendeklarationen etc. können in separaten Einheiten (*packages*) zusammengefaßt werden.

7.2 Simulationsergebnisse

Der gesamte NeNEB-Chip wurde entsprechend dem *top-down*-Konzept in mehrere Teilmodelle aufgespalten (vgl. Design-Hierarchie in Anhang B.1). Die einzelnen Teilmodelle wurden dabei mit geeigneten Testmustern separat simuliert, wobei die Eingabewerte der Modelle interaktiv im Synopsys-VHDL-Debugger erzeugt und mit *waves*[1] überprüft wurden. Diese Testmethode wurde auch für komplexere Modelle wie die Erregungsfunktion (*errj_alu*) oder die Aktivierungsfunktion (*out_alu*), d. h. bis zur Ebene 3 der Design-Hierarchie angewendet. Sofern es sich bei den Teilmodellen um getaktete Modelle wie z. B. Register handelte, wurden für die Teilmodelle eigene kleine Simulationsmodelle geschrieben, in denen ein asynchrones Rücksetzsignal *clrn* sowie ein kontinuierlicher Takt *clk* erzeugt wurde. Zur leichteren interaktiven Erzeugung der Eingabewerte im VHDL-Debugger wurden jeweils kleine Kommandoprozeduren (Dateien mit Endung *.cms*) geschrieben. Mit Hilfe dieser Kommandoprozeduren konnten dann auch komplette Kommandosequenzen geschrieben werden, um die Simulation quasi automatisch ablaufen zu lassen.

Nachdem hiermit die Korrektheit der implementierten Algorithmen bis zur Ebene 3 überprüft war, wurde alle Teilmodelle zum Gesamtmodell des NeNEB-Chips (*nn*) in Ebene 6 zusammengefügt, wobei die Ebenen 4 (*hidden* und *output*) und 5 (*nn_stw* und *nn_opw*) aufgrund der vielen Steuersignale nicht separat simuliert wurden. Für dieses Gesamtmodell wurde dann ein Simulationsmodell (*nn_sim*) geschrieben, das sowohl den Hostrechner simuliert als auch die Bilddaten im korrekten Format liefert. Das Simulationsmodell benötigt zwei Dateien mit den Initialisierungsdaten und den Bilddaten. Beide Dateien müssen im *pgm-ascii*-Format[2] vorliegen und werden von dem VHDL-Modell *readfkt* eingelesen. Die Namen und Größen der Dateien werden dem Simulationsmodell über *generic*-Parameter übergeben. Für die Simulation des externen Gewichtespeichers wurde ein am Arbeitsbereich verfügbares VHDL-Modell eines statischen RAM verwendet (*sram*).

Die korrekte Funktion des Netzwerkes wurde dann zunächst mit dem XOR-Problem (vgl. Kapitel 2.2.6) getestet. Dabei, wie auch in den folgenden Simulationen, wurden neben den wichtigen Steuersignalen und Zuständen der Steuerautomaten die Erregungen und Ausgaben aller Neuronen überprüft. Als Referenz diente dabei das Simulationsprogramm *bgpsim* (Anhang D.1.1). Die vollständige Funktion des NeNEB-Chips ließ sich mit dem XOR-Problem jedoch noch nicht überprüfen, da dieses nur zwei Eingabestellen, zwei verborgene Neuronen und ein Ausgabeneuron benötigt. Außerdem haben die Netzgewichte des XOR-Problems relativ einfache Werte (vgl. Anhang D.1.3), bei denen Rechenfehler verborgen bleiben können, jedoch konnte mit der Simulation des XOR-Problems schon eine erste Überprüfung der grundsätzlichen Funktionsfähigkeit des NeNEB-Chips durchgeführt werden.

---

[1] *waves* - Darstellungsprogramm zum Überprüfen von Signalpegeln ähnlich Timingdiagrammen

[2] Das *pgm-ascii*-Format speichert Graustufenbilder (8-Bit-Farbtiefe) als eine Folge von ASCII-Zahlenwerten, d. h. in Form einer Textdatei. Am Anfang der Dateien steht die Bildgröße in Form von Zeilen- und Spaltenanzahl sowie die Farbtiefe .

Als nächstes wurde ein 1-aus-8-Decoder gemäß (Tabelle 7.1) simuliert. Für dieses Problem waren neben den 3 Eingabestellen und den 8 Ausgabeneuronen alle 8 verborgenen Neuronen nötig, zudem haben die Netzgewichte schön verteilte "krumme" dezimale Werte. Hier konnte geprüft werden, ob auch wirklich alle 8 verborgenen Neuronen korrekt arbeiten. Weiterhin konnte mit der Simulation des 1-aus-8-Decoders die korrekte Funktion des Serialisierers für die Ausgaben der verborgenen Neuronen und des Ringspeichers für die Gewichte der Ausgabeneuronen überprüft werden.

Eingabemuster	**Ausgabemuster**
0 0 0	0 0 0 0 0 0 0 1
0 0 1	0 0 0 0 0 0 1 0
0 1 0	0 0 0 0 0 1 0 0
0 1 1	0 0 0 0 1 0 0 0
1 0 0	0 0 0 1 0 0 0 0
1 0 1	0 0 1 0 0 0 0 0
1 1 0	0 1 0 0 0 0 0 0
1 1 1	1 0 0 0 0 0 0 0

Tabelle 7.1: Ein- und Ausgabe des 1-aus-8-Decoders

Zuletzt wurde noch ein Problem mit einer größeren Anzahl an Eingabestellen und Ausgabeneuronen simuliert. Hierfür wurde das Buchstabenerkennungsproblem (Bild 7.1) gewählt. Dieses bildet die Buchstaben auf einer 5*7 Eingabematrix auf ein Netzwerk mit 8 verborgenen Neuronen und 26 Ausgabeneuronen ab. Aus Zeitgründen wurden dabei zwar nur die Buchstaben "A" bis "H" getestet, jedoch zeigten sich dabei keine Abweichungen zwischen dem Simulationsprogramm und dem NeNEB-Chip. Mit den insgesamt 35 Eingabewerten für jedes Muster des Buchstabenerkennungsproblems ließ sich erstmalig auch die korrekte Funktion des Pipelining-Konzepts des NeNEB-Chips testen. Dazu wurde das bisherige Simulationsmodell (*nn_sim*) leicht abgeändert (*nn_sim2*), so daß insgesamt drei Eingangsmuster nacheinander simuliert werden konnten.

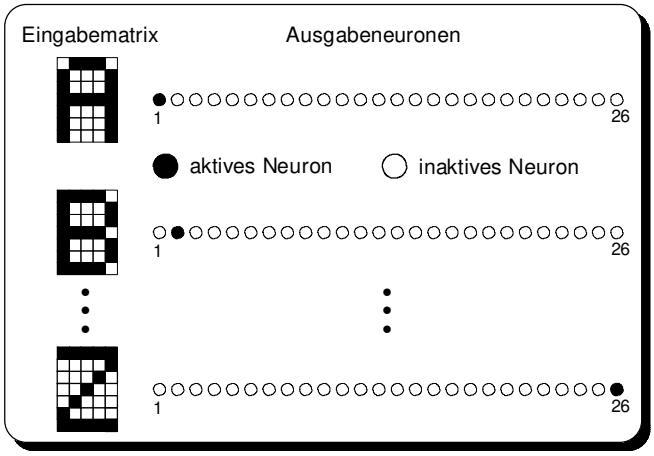


Bild 7.1: Buchstabenerkennungsproblem

7.3 Syntheseergebnisse

Zur Überprüfung der Synthetisierbarkeit der VHDL-Modelle wurden alle Teilmodelle bis zur 4. Ebene mit dem Synopsys-System unter Verwendung eines 0,7 µm CMOS[1]-Prozesses (ecpd07 von ES2) synthetisiert. Die Synthese der beiden höheren Ebenen 5 und 6 wurde aus Zeitgründen bisher nicht durchgeführt.

Die Abschätzungen des Synthesesystems für die Verzögerungszeiten wurde als Grundlage für die Festlegung der Taktraten der einzelnen Verarbeitungsstufen verwendet. Die VHDL-Modelle wurden dabei zunächst ohne Vorgaben bzw. Einschränkungen synthetisiert. Das Ergebnis dieser ersten Synthesen war im Regelfall eine Schaltung mit geringstem Flächenbedarf. Dies rührt daher, daß für die Elemente der synthetisierten Schaltung zum Teil komplexe optimierte Bibliothekselemente wie z. B. Volladdierer oder Multiplexer verwendet wurden. Die Hierarchie der VHDL-Modelle blieb auch in den synthetisierten Schaltungen erhalten. Allerdings hatten die auf diese Weise synthetisierten Schaltungen zum Teil recht hohe Verzögerungszeiten. Da das Ziel beim NeNEB-Chip, wie bei jeder integrierten Schaltung aus Kostengründen auch, ein möglichst geringer Chipflächenbedarf ist, waren diese ersten Syntheseergebnisse trotzdem schon sehr zufriedenstellend, da nur die vordersten Stufen für die Berechnung der verborgenen Neuronen zeitkritisch sind.

Die beiden zeitkritischsten Elemente bei der Berechnung der verborgenen Neuronen sind die beiden Addierer in Teil 1 und Teil 2 der Erregungsfunktion. Um zu überprüfen, welche maximale Geschwindigkeit bei diesen beiden Addierern erzielt werden kann, wurde zunächst der Addierer in Teil 1 der Erregungsfunktion (*adder1*) mit der Option *max_delay 0.0 all_outputs ()* synthetisiert. Dies bedeutet, daß das Synthesesystem eine Schaltung mit minimaler Verzögerungszeit bei allen Ausgängen synthetisieren soll. Das Ergebnis dieser Synthese zeigt Tabelle 7.2. Das Ergebnis zeigt eine Verringerung der Verzögerungszeit um den Faktor 4, aber nur eine Zunahme der Chipfläche um den Faktor 2, d. h. ein akzeptables Resultat, da die Zunahme der Chipfläche geringer als der Geschwindigkeitsgewinn ist.

	ohne Optionen	**Option minimale Verzögerungszeit**
Chipfläche	39276,8 µm²	81789,8 µm²
geschätzte maximale Verzögerungszeit	12,93 ns	3,31 ns

Tabelle 7.2: Syntheseergebnisse bei *adder1*

Für den Addierer in Teil 2 der Erregungsfunktion (*adder2*) steht mindestens die achtfache Verzögerungszeit von *adder1* zur Verfügung. Da dieser Wert bereits bei der Standardsynthese ohne Optionen erzielt wurde (vgl. Tabelle 7.3), waren hier keine weiteren Syntheseversuche notwendig. Die folgende Tabelle 7.3 enthält noch einige weitere Syntheseergebnisse zeitbestimmender Schaltungsteile.

In Tabelle 7.4 sind die Flächenabschätzungen für ein verborgenes Neuron sowie für ein Ausgabeneuron enthalten. Der Flächenbedarf des Gewichtespeichers eines Ausgabeneurons (*fifo2*) ist dabei getrennt aufgeführt, da er allein rund 40 % der Gesamtfläche des Neurons in Anspruch nimmt. Der leichte Unterschied zwischen den Flächen eines verborgenen Neurons (*hidden*) und eines Ausgabeneurons (*output*) erklärt sich durch die Tatsache, daß das Schieberegister zur

[1] *CMOS - engl. complementary metal oxide semiconductor* ≈ komplementärer Metall Oxyd Halbleiter

Serialisierung der Gewichte eines Ausgabeneurons nicht im Modell des Gewichtespeichers enthalten ist, sondern im Modell des Ausgabeneurons. Mit diesen Flächenabschätzungen von ca. 0,6 mm² für ein verborgenes Neuron und ca. 1,0 mm² für ein Ausgabeneuron ergibt sich für den NeNEB-Chip bei 8 verborgenen Neuronen und 32 Ausgabeneuronen eine Gesamtchipfläche von ca. 40 mm².

VHDL-Modell	Chipfläche	geschätzte maximale Verzögerungszeit
adder2	109057,4 µm²	23,11 ns
adder3	24966,0 µm²	6,41 ns
barrelreg	80438,4 µm²	4,6 ns Register-Eingänge 1,03 ns Register-Ausgänge
komp1	17328,0 µm²	9,99 ns
komp2	17240,6 µm²	10,11 ns
cntdwn (16 Bit)	128489,4 µm²	8,26 ns Flip-Flop-Eingänge 2,37 ns timeout-Ausgang
hmul_stw	16807,4 µm²	3,16 ns Flip-Flop-Eingänge 2,96 ns Ausgänge
hadd_stw	25292,8 µm²	3,96 ns Flip-Flop-Eingänge 2,63 ns Ausgänge
fifomult_stw	20573,0 µm²	3,22 ns Flip-Flop-Eingänge 3,08 ns Ausgänge

Tabelle 7.3: Syntheseergebnisse wichtiger zeitbestimmender Schaltungsteile

VHDL-Modell	Chipfläche
hidden	605598,4 µm²
output	656545,0 µm²
fifo2	410073,2 µm²

Tabelle 7.4: Chipflächenabschätzung für ein verborgenes Neuron und für ein Ausgabeneuron

8 Ergebnisse, Ausblick und Erweiterungsansätze

Als Vorstufe für den NeNEB-Chip wurde im Rahmen dieser Diplomarbeit zunächst ein FPGA zur Auflösungsreduktion (Mosaik-FPGA) entwickelt. Der Grund für die Entwicklung des Mosaik-FPGA war, daß zu Beginn der Arbeit noch keine genauen Vorstellungen über die erreichbare Geschwindigkeit des NeNEB-Chips vorlagen, weswegen der NeNEB-Chip zunächst als eine Art Machbarkeitsstudie gedacht war. Über das Mosaik-FPGA sollte die Möglichkeit geschaffen werden, die Geschwindigkeit und die Auflösung des Digitalisiersystems an die Geschwindigkeit und die Kapazität des NeNEB-Chips anzupassen, wobei ein einfacher Auflösungsreduktions-Algorithmus wie der gewählte völlig ausreichend ist.

Mit den jetzt vorliegenden Ergebnissen über die zu erwartende Geschwindigkeit und die Kapazität des NeNEB-Chips ist das Mosaik-FPGA nur noch bedingt notwendig, da die Geschwindigkeit des Mosaik-FPGAs selbst nur sehr begrenzt ist (vgl. Tabelle 8.1). Es wird nur noch bei größeren Bildern (> 256 * 256 Pixel) oder höheren Pixelraten (> 3,2 MHz) benötigt. Der Nachteil der relativ geringen Geschwindigkeit des Mosaik-FPGA ließe sich durch die Realisierung als VLSI-Chip bzw. durch direkte Implementation in den NeNEB-Chip überwinden. Dies hätte den Vorteil einer noch kompakteren Realisierung des in Kapitel 1 geschilderten Gesamtsystems zur Bildklassifikation.

	max. Taktfrequenz	**max. Pixelrate**	**max. Pixel/Zeile Eingang**	**max. Pixel/Zeile Ausgang**
Mosaik-FPGA	32 MHz	8 MHz	abhängig von der Anzahl der Reduktionsstufen	512
NeNEB-Chip	ca. 60 MHz	ca. 3,3 MHz	256 bei quadratischen Bildern	-

Tabelle 8.1: Vergleich der Leistungsdaten des NeNEB-Chips und des Mosaik-FPGAs

Der NeNEB-Chip selbst hat die zunächst vorgegebenen Minimalanforderungen weit übertroffen, sowohl in bezug auf die Größe des neuronalen Netzwerkes als auch im Hinblick auf die Geschwindigkeit (vgl. Tabelle 8.2). Entsprechend dem vorgesehenen Einsatzgebiet des NeNEB-Chips in einem kompakten (*stand-alone*) Bildklassifikationssystem wurde der Chip nicht skalierbar gestaltet. Das darin implementierte Netzwerk dürfte aufgrund der Erfahrungen von ALVINN für die verschiedensten Bilderkennungs- bzw. -klassifikationsaufgaben geeignet sein. Trotzdem wurden über die Skalierbarkeit Überlegungen angestellt. Hierbei kann zunächst zwischen zwei Arten der Skalierung unterschieden werden:

1. horizontale Erweiterung
2. vertikale Erweiterung.

max. Taktfrequenz	ca. 60 MHz
max. Pixelrate	ca. 3,3 MHz
max. Geschwindigkeit	ca. 26,2 Mcps
max. Bildgröße	65.536 Pixel
Bildformat	beliebig bis zur max. Bildgröße
neuronales Netzwerk	zweischichtiges Backpropagation-Netzwerk
Anzahl verborgener Neuronen	8
Anzahl Ausgabeneuronen	32
Anzahl Gewichte der verborgenen Neuronen	524.288 (8 · 65.536)
Anzahl Gewichte der Ausgabeneuronen	256 (32 · 8)
Genauigkeit der Eingabewerte	8 Bit
Genauigkeit der Gewichte	16 Bit (12 Nachkommastellen)
Aktivierungsfunktion	Geradenannäherung der Sigmoidfunktion aus fünf Geradenstücken mit 8 verschiedenen Steigungen

Tabelle 8.2: Merkmale des NeNEB-Chips

Bei der horizontalen Erweiterung wird die Anzahl der verborgenen Neuronen bzw. der Ausgabeneuronen durch Nebeneinanderschalten mehrerer NeNEB-Chips erweitert. Hierzu wäre es notwendig, den NeNEB-Chip durch einen bidirektionalen 8-Bit-Bus (*Databus*) zu erweitern, auf dem die Ausgaben der verborgenen Neuronen den Ausgabeneuronen der anderen Chips zur Verfügung gestellt werden können. Hinzu käme ein bidirektionales Datenfreigabesignal. Nach Art eines *Daisy-Chaining* belegt dabei jeder Chip nacheinander den gemeinsamen bidirektionalen Bus (Bild 8.1). Hierzu benötigt jeder Chip noch einen Ein- und Ausgang für ein *Chip-active*-Signal (*CAI*, *CAO*). Der Chip, der gerade den Bus treibt, zeigt dies durch eine Eins auf *CAO* an. Ein nachfolgender Chip darf den Bus nur belegen, wenn er einen Eins-Null-Übergang an *CAI* erkannt hat. Der Chip, der nach dem Einschalten als erstes den Bus belegen darf, wird durch eine Eins an seinem Master-Eingang (*MS*) gekennzeichnet.

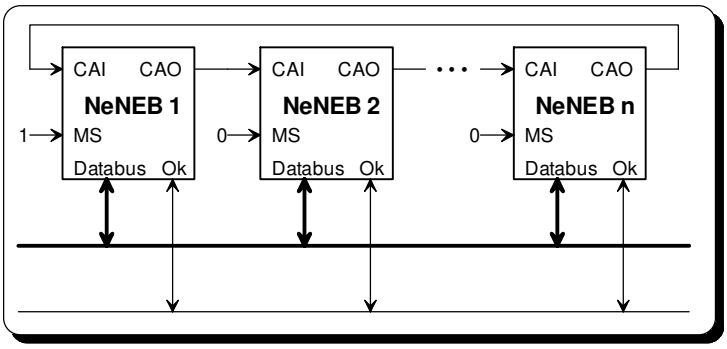


Bild 8.1: horizontale Skalierung mehrerer NeNEB-Chips

Die vertikale Erweiterung dient der Erhöhung der Anzahl der Netzwerkschichten, indem mehrere NeNEB-Chips hintereinander geschaltet werden. Hierfür müßten die Ausgaben der Ausgabeneuronen den nachfolgenden Chips serialisiert zugeführt werden, d. h. das wahlfreie Ausgaberegister müßte durch einen Serialisierer, wie er für die Ausgaben der verborgenen Neuronen verwendet wird, ersetzt werden.

9 Zusammenfassung

Im Rahmen der vorliegenden Diplomarbeit wurde die Ausführungsphase eines künstlichen neuronalen Backpropagation-Netzwerkes mit 8 verborgenen Neuronen und 32 Ausgabeneuronen als eigenständiges VLSI-Chipmodell (NeNEB-Chip) in Form einer Hardwarebeschreibungssprache (VHDL) realisiert. Das Ziel bei der Entwicklung des NeNEB-Chips war die Fähigkeit des Chips zur Echtzeitklassifikation von Bildern, die von einem Digitalisiersystem bzw. einer Auflösungsreduktionseinheit geliefert werden. Hierzu ist der NeNEB-Chip in ein Gesamtsystem aus Kamera, Digitalisiersystem und evtl. Auflösungsreduktionseinheit eingebunden. Die Berechnung des neuronalen Netzwerkes erfolgt in Echtzeit in einem seriellen Strom von Bilddaten. Daher konnte auch die Berechnung serialisiert werden, im Gegensatz zu üblichen hochparallelen Lösungen mit systolischen Arrays o. ä.

Der Vorteil des NeNEB-Chips ist seine Fähigkeit, mit einer sehr großen Anzahl von Eingabestellen bzw. Gewichten für jedes verborgene Neuron arbeiten zu können. Hierzu werden die Gewichte der verborgenen Schicht in einem externen Gewichtespeicher abgelegt, während die Gewichte der Ausgabeneuronen wegen der wesentlich geringeren Anzahl ON-Chip gespeichert werden. Die Multiplikation eines Gewichtes mit einem Eingabewert erfolgt nach dem Hornerschema, wobei die Gewichte der Berechnung seriell zugeführt werden. Da in jedem Zeittakt nur ein Bit jedes Gewichtes benötigt wird, enthält jedes Speicherwort ein Bit der entsprechenden Gewichte für alle verborgenen Neuronen. Bei insgesamt acht verborgenen Neuronen können somit handelsübliche Speicherbausteine mit 8 Bit Datenbusbreite verwendet werden. Die Größe des externen Speichers ist auf 1 M Byte begrenzt. Dies erlaubt die Speicherung von insgesamt 524288 Gewichten bzw. 65536 Gewichten pro verborgenem Neuron, da jedes Gewicht in 16 aufeinanderfolgenden Speicherworten gespeichert ist. Hiermit können Bildgrößen von maximal 256 * 256 Pixeln direkt vom NeNEB-Chip verarbeitet werden, wobei das Bildformat nicht zwingend quadratisch sein muß, da ein Bild vom NeNEB-Chip als ein einziger großer Vektor gesehen wird.

Aufgrund der vorläufigen Abschätzung mit den Syntheseergebnissen können bei einer Taktfrequenz von 60 MHz pro Sekunde 50 Bilder mit einer Bildgröße von 65536 Pixeln verarbeitet werden (ca. 26,2 Mcps). Der Chipflächenbedarf beträgt ca. 0,6 mm² für ein verborgenes Neuron und ca. 1,0 mm² für ein Ausgabeneuron (inkl. des Gewichtespeichers). Der Gesamtchip dürfte eine Fläche von ca. 40 mm² haben.