UNIVERSITÄT LEIPZIG

Synaptisches Rauschen auf Grundlage der Poissonverteilung: Systementwurf und Integration in FPGA

Bachelor Arbeit

vorgelegt von

Azanzar Youssef
azanzar@gmx.de
Studiengang: Informatik

Betreuer:

Prof. Dr. Martin Bogdan

Universität Leipzig
Fakultät für Mathematik und Informatik
Institut für Informatik
Abteilung Technische Informatik

Leipzig, Oktober 2008

Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann.

Ort: **Datum:** **Unterschrift:**

Inhaltsverzeichnis

Abbildungsverzeichnis

Algorithmenverzeichnis

1 Einleitung

Unter Rauschen versteht man allgemein eine Störgröße. Bei der Übertragung von Nachrichtensignalen ist das Rauschen meist die größte Störquelle. Die Rauschquellen treten dabei im gesamten Übertragungssystem, also im Sender, im Empfänger und auf dem Übertragungsweg auf. Man unterscheidet dabei zwischen der durch äußere und innere Rauschquellen erzeugten Rauschleistung. Rauschen kann signifikante Auswirkung auf die Antwortdynamik nichtlinearer Systeme haben. Für Neuronen ist die primäre Quelle des Rauschens synaptische Hintergrundaktivität; bezeichnet als synaptisches Rauschen.

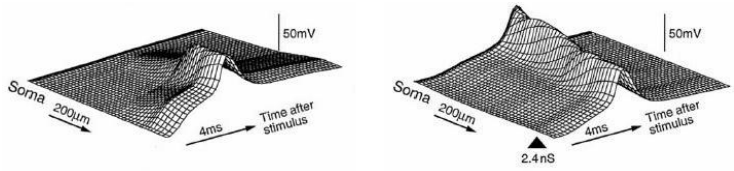
In dieser Arbeit wird die Möglichkeit untersucht, dass bereits in [1] in C++ programmiertes, synaptisches Rauschen in einem Xilinx Viertix4 Chip zu implementieren. Als Beschreibungssprache wird VHDL und als Entwicklungsumgebung wird „Xilinx ISE 9.2i“ mit dem integrierte Simulationsprogramm „ISE Simulator“ benutzt. Das Ziel dieser Untersuchung ist die Integration in ein biologisch motiviertes, künstliches, neuronales Netz. Ausgehend von der Motivation und der Vorstellung von biologischen Grundlagen, Modellen von künstlichen Neuronen und mathematischen Grundlagen wird dann ein Modell unter der Vorstellung des gesamten Problems und der Lösungsvorschläge beschrieben und diskutiert. Es werden Algorithmen beschrieben und implementiert, soweit es möglich ist.

2 Motivation

Eine wichtige Frage in der Neurologie ist, wie Neuronen Informationen verarbeiten oder kodieren. Hier muss zunächst geklärt werden auf welche Art und Weise der sich weit ausbreitende dendritische Baum von Neuronen synaptische Signale verarbeitet und integriert. Mit fortgeschrittenen Techniken ist bereits viel der Funktionsweise von Neuronen aufgedeckt worden. Betrachten wir das Problem der Kodierung eines Analogsignals in einer Bevölkerung von Neuronen. Einige Faktoren müssen in Betracht genommen, wenn man diesen Punkt anspricht. Zuerst ist die Feuerungsrate der Neuronen häufig hoch und unregelmäßig. Zweitens ist die Feuerungsrate der kortikalen Neuronen manchmal verhältnismäßig langsam, im Vergleich mit vielen anderen Signalen benötigen diese Decodierungen. Daher kommt die Frage wie neokortikal Neurone synaptische Signale in einem intakten Gehirn integrieren, jedoch ist diese bisher nur rudimentär beantwortet, was hauptsächlich darin begründet ist, dass die zur Verfügung stehende Messtechniken die Erregungsverteilung über ein Neuron zeitlich und räumlich bisher nicht hoch genug auflösen können, als dass Ableitungen in vivo aussagekräftige Ergebnisse hervorbringen konnten. Eine klare Aussage, die sich nach heutigem Wissensstand allerdings mit gutem Gewissen tätigen lässt, ist, dass in vivo betrachtete Neurone

eine wesentlich höhere spontane Feuerrate aufweisen, als solche, deren Aktivität in vitro gemessen wird. Worin liegt diese Beobachtung begründet? Neben der Tatsache, dass in Nährlösung gehaltene Gehirnschnitte zur Untersuchung von Neuronen in einem ihrer natürlichen Umgebung lediglich nachempfundenen Medium gehalten werden, kommt zum Tragen, dass die Neurone innerhalb dieser Schnitte den überwiegenden Teil ihrer weit reichenden Verbindungen zu Neuronen in kortikalen und subkortikalen Bereichen verloren haben. Ein einzelnes Pyramidal-Neuron beispielsweise besitzt auf seinem Dendritenbaum und dem Soma in seiner natürlichen Umgebung zwischen 5000 bis 60000 synaptische Verbindungen, die ihren Ursprung sowohl im cerebralen Kortex selbst haben, aber auch aus tiefer liegenden Regionen wie dem Hirnstamm und dem Thalamus aufsteigen. Intrazellulär Ableitungen von kortikalen Neuronen in vivo konnten zeigen, dass diese ob ihrer mannigfaltigen synaptischen Eingänge einem intensiven Bombardement synaptischer Aktivität unterliegen, woraus eine ständige Depolarisation und ein verringerter Eingangswiderstand über die neuronale Membran resultieren. Dies wird im Allgemeinen als 'high-conductance state' bezeichnet, einem Zustand erhöhter Leitfähigkeit entlang der dendritischen, erregbaren Membran.

In diesem Zustand erreichen postsynaptische Signale in Form sich ausbreitender Erregung über die Dendritenmembran das Soma mit wesentlich höherer Wahrscheinlichkeit [1].
Abbildungen 1a und *1b* verdeutlichen dieses Phänomen.



(a) Isolation (b) Population

Abbildung 1: a) Simulation in Vitro, b) Simulation in Vivo

Auf den beiden Graphiken [1] ist die räumliche und zeitliche Ausbreitung eines etwa 200 μm vom Soma entfernt gesetzten Reizes zu sehen. *Abbildung 1a* bezieht sich auf ein Neuron in vitro und zeigt, dass die Amplitude der Erregung in proximaler Richtung schnell abfällt und eine Depolarisation, in diesem Fall um 50mV, am Soma kaum zu registrieren ist. *Abbildung 1b* zeigt, wie der gleiche Reiz in einem Neuron propagiert, das sich in seiner natürlichen Umgebung, also einer Population zentralnervöser Neuronen befindet. In diesem Fall erreicht der gesetzte Reiz das Soma sogar mit leicht erhöhter Intensität.

3 Biologische Grundlagen

Künstliche neuronale Netze sind in ihrer Struktur und Arbeitsweise dem Nervensystem und speziell dem Gehirn von Tieren und Menschen nachempfunden.

Das Nervensystem von Lebewesen besteht aus dem Gehirn bzw. Zentralnervensystem (bei so genannten *primitiven* Lebewesen), den verschiedenen sensorischen Systemen, die Informationen aus den verschiedenen Körperteilen sammeln, und den motorischen System, das Bewegungen steuert. Zwar findet der größte Teil der Informationsverarbeitung im Gehirn/Zentralnervensystem statt, doch ist manchmal auch die außerhalb des Gehirns durchgeführte Vorverarbeitung beträchtlich, z.B. in der Retina (der Netzhaut) des Auges.

Im Bezug auf die Verarbeitung von Informationen sind die Neuronen die wichtigsten Bestandteile des Nervensystems. Nach gängigen Schätzungen gibt es in einem menschlichen Gehirn etwa 10^{11} Neuronen von denen ein ziemlich großer Teil gleichzeitig aktiv ist. Neuronen verarbeiten Informationen im wesentlichen durch Interaktionen miteinander.

Ein Neuron ist eine Zelle, die elektrische Aktivität sammelt und weiterleitet. Neuronen gibt es in vielen verschiedenen Formen und Größen. Dennoch kann man ein *prototypisches* Neuron angeben, dem alle Neuronen mehr oder weniger gleichen. Dieser Prototyp ist schematisch in *Abbildung 2[2]* dargestellt.

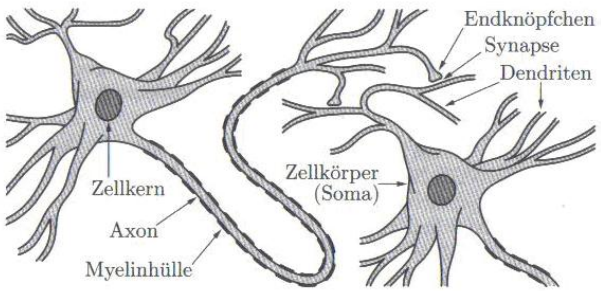


Abbildung 2: Prototypischer Aufbau biologischer Neuronen

Die Axone sind die Leitungen, auf denen Neuronen miteinander kommunizieren. Das Axon eines Neurons führt zu den Dendriten anderer Neuronen. An seinem Ende ist das Axon stark verzweigt und besitzt an den Enden der Verzweigungen so genannte Endknöpfchen (engl.: terminal boutons). Jedes Endknöpfchen berührt fast einen Dendriten oder dem Zellkörper eines anderen Neurons. Die Lücke zwischen den Endknöpfchen und einem Dendriten oder dem Zellkörper heißt synaptischer Spalt (10 - 50 nm breit).

Die typischste Form der Kommunikation zwischen Neuronen ist, dass ein Endknöpfchen des Axons bestimmte Chemikalien, die so genannte Neurotransmitter, freisetzt, die auf die Membran des empfangenden Dendriten einwirken und seine Polarisation (sein elektrisches Potential) ändern. Dann ist das Innere der Zellmembran, die das gesamte Neuron umgibt, normalerweise etwa 70mv negativer als sein Äußeres, da innerhalb des Neurons die Konzentration negativer Ionen und außerhalb die Konzentration positiver Ionen größer ist. Abhängig von der Art des ausgeschütteten Neurotransmitters kann sich die Potentialdifferenz verringern, nennt man sie exzitatorisch (erregend), oder solche, die sie erhöhen, inhibitorisch(hemmend). Wenn der erregende Nettoeinfluss groß genug ist, kann die Potentialdifferenz im Zellkörper stark abfallen. Ist die Verringerung des elektrischen Potentials groß genug, wird der Axonansatz depolarisiert. Diese Depolarisierung wird durch ein Eindringen positiver Natrium-Ionen in das Zellinnere hervorgerufen. Dadurch wird das Zellinnere vorübergehend (für etwa eine Millisekunde) positiver als seine Außenseite. Anschließend wird durch den Austritt von positiven Kalium-Ionen die Potentialdifferenz wieder aufgebaut. Die ursprüngliche Verteilung der Natrium- und Kalium-Ionen wird schließlich durch spezielle Ionen-Pumpen *(Abbildung 3) [3]* in der Zellmembran wieder hergestellt. Das Aktionspotenzial pflanzt sich entlang des Axons fort. Die Fortpflanzungsgeschwindigkeit beträgt je nach den Eigenschaften des Axons zwischen 0,5 und 130m/s. Insbesondere hängt sie davon ab, wie stark das Axon mit einer Myelin-Hülle umgeben ist (je stärker die Myelinisierung, desto schneller die Fortpflanzung des Aktionspotentials).

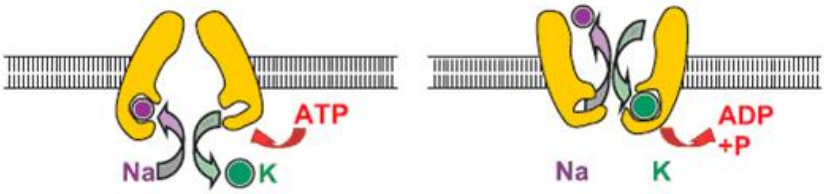


Abbildung 3: Ionenpumpe

Wenn dieser Nervenimpuls das Ende des Axon erreicht, bewirkt er an den Endknöpfchen die Ausschüttung von Neurotransmittern, wodurch das Signal weitergegeben wird. Die Anzahl der Nervenimpulse, die ein Neuron

pro Sekunde weiterleitet, werden als Feuerrate des Neurons bezeichnet.

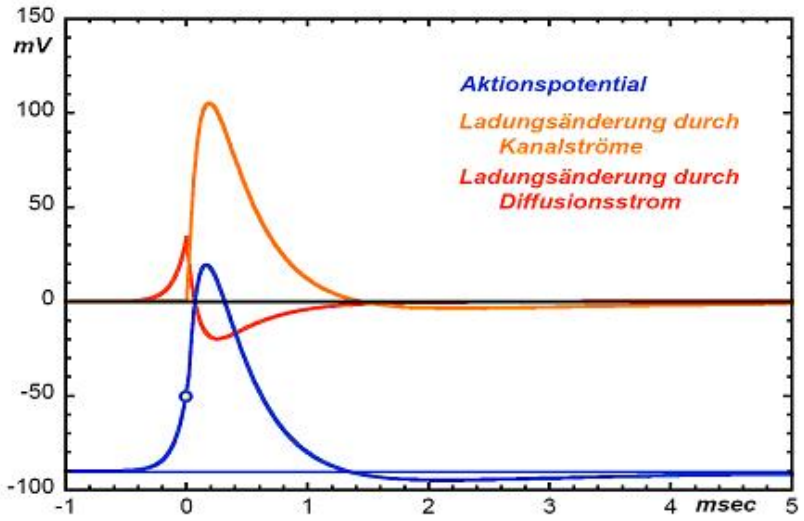


Abbildung 4: Aktionspotenzial *[3]*

4 Künstliche Neuronen

Modellbildung im Bereich neuronaler Systeme kann auf recht unterschiedlichen Ebenen geschehen, je nach Zielsetzung oder Herkunft. Das Ziel kann ein Verständnis biologischer Vorgänge sein, spezieller Gehirnfunktionen, Verarbeitung sensorische Signale oder Steuerung von Bewegungsvorgängen, aber auch technische Anwendungen, z.B. in der Mustererkennung oder Steuerung von Prozessen, oder auch die Suche nach möglichen Prinzipien neuronaler Datenverarbeitung. Auch die Modellierung einzelner Neuronen kann sehr unterschiedlich sein. *[4]*

Schwellenwertelement ist ein mathematisches und phänomenologisches Modell für Neuronen. Erhält ein Neuron genügend erregende Impulse, so wird es aktiv und sendet ein Signal an andere Neuronen. Dieses Modell wurde schon sehr früh von McCulloch und Pitts (1943) genauer untersucht. Schwellenwertelemente nennt man daher auch McCulloch-Pitts-Neuronen.

Ein Schwellenwertelement ist eine Verarbeitungseinheit für reelle Zahlen mit n Eingänge $x_1, \ldots, x_n$ und einem Ausgang y. Der Einheit als Ganzes ist ein Schwellenwert θ und jedem Eingang x_i ein Gewicht ω_i zugeordnet. Ein Schwellenwertelement *Abbildung:5[3]* berechnet die Funktion:

$$y = \begin{cases} 1, falls \sum_{i=1}^{n} \omega_i x_i \geq \theta \\ 0, sonst \end{cases} \quad (1)$$

.

Oft fasst man die Eingänge zu einem Eingangsvektor $\overrightarrow{E}$ und gewichte zu einem Gewichtsvektor $\overrightarrow{G}$. Dann kann man unter Verwendung von Skalarprodukt die von einem Schwellenwertelement geprüfte Bedingung auch $\overrightarrow{E}.\overrightarrow{G} \geq \theta$ schreiben.

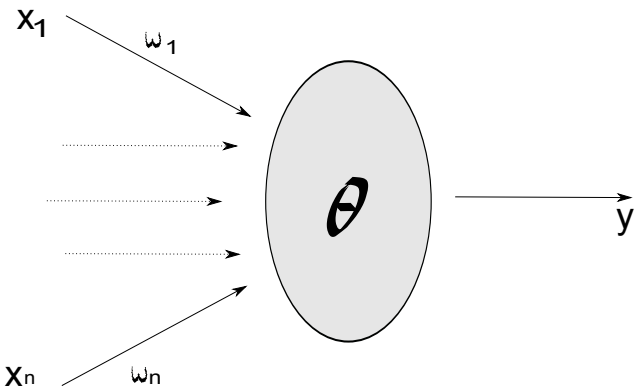


Abbildung 5: Darstellung eines Schwellenwertelement

Hodgkin-Huxley Modell: In diesem Modell sind die morphologischen und die Funktionen verschiedener Kanaltypen einbezogen. Mehrere Kompartimente dienen dazu detaillierte Beschreibungen von Dendriten, Soma und Axon zu emulieren. Diese Modellierung ist aufwändig und auf einzelne Neuronen oder eine geringe Zahl wechselwirkender Neuronen beschränkt. Dieses Modell wird auch häufig als Grundmodell, Elektrisches Modell oder Kabelmodell bezeichnet.

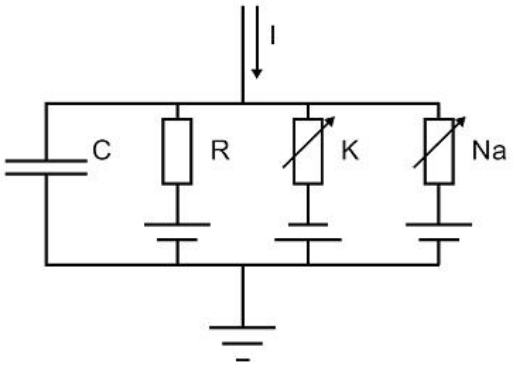


Abbildung 6: Darstellung von Hodgkin-Huxley Modell [13]

Punktneuronen: In Modellen so genannter Punktneuronen wird die räumliche Struktur nicht berücksichtigt, aber auch hier kann die Modellierung unterschiedlich komplex sein. In einer Beschreibung anhand Spannungs- oder transmitterabhängiger Leitfähigkeiten sind Konzentrationen und Ströme verschiedener Ionen zu berücksichtigen.

Integrate-and-fire-Modell berücksichtigen Änderungen des Membranpotentials und die Auslösung von Spikes. In einem verwandten, aber etwas einfacherem, Modell wird die Wirkung eines eintreffenden präsynaptischen Spikes durch eine zeitlich abklingende Erregung bzw. Hemmung beschrieben, wobei die Auslösung eines Spikes durch Überschreiten einer Schwelle geschieht.

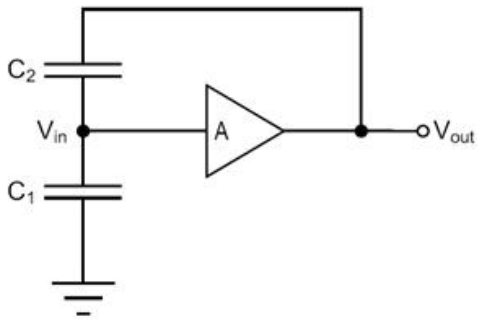


Abbildung 7: Carver Meads basic Integrate and Fire Neuron [13]

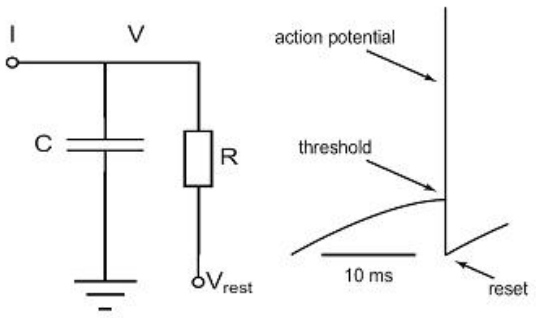


Abbildung 8: Lapicques original Integrate and Fire Neuron [13]

Spikende Neuronen: Hier kann die mögliche Bedeutung der präzisen Synchronisation einlaufender Spikes untersucht werden, aber auch der Einfluss des Rauschens aufgrund der diskreten Natur der Signale.

5 Mathematische Grundlagen

Ein Rechner verfügt über endliche Ressourcen wie Speicher, Takt-Zyklus, Recheneinheiten und begrenzte Anzahl von Registern. Deswegen kann man nicht, auf einem Rechner, alle Zahlen von $\mathbb{R}$ darstellen, und darauf arbeiten, sondern nur eine Teilmenge $\mathbb{M}_{b,l}$ von $\mathbb{R}$.

5.1 Zahlendarstellung

Zu einer Basis $b \in \mathbb{N}$ kann man jede beliebige Zahl $x \in \mathbb{R}$ darstellen mit der folgenden Formel:

$$x = \pm \sum_{i=1}^{\infty} d_i b^{r-i} \tag{2}$$

$$mit\ d_i \in \{\ 0,\ \ldots,\ b\text{-}1\ \}$$

Die sogenannte Normalisierte Darstellung ist wie folgt definiert:

$$x = v.m.b^e \tag{3}$$

$$\text{mit v} \in \{\pm 1\},\ \text{m} = \sum_{i=1}^{\infty} d_i b^{r-e-i}$$

v-Vorzeichen, b-Basis, m-Mantisse ,e-Exponent

Bei b=2 ist die erste Stelle nach dem Komma gleich 1 mit Ausnahme die 0, wobei alle Stellen in der Mantisse gleich 0 sind. Die erste Stelle in einer Mantisse ist stets gleich 1 und braucht damit nicht dargestellt zu werden.

Auf einem Rechner muss entscheidbar sein, wann zwei Zahlen gleich sind. Eine wichtige Frage ist daher, in wie weit normalisierte Darstellungen von reellen Zahlen eindeutig sind. Jedes x $\in \mathbb{R}\backslash\{0\}$ besitzt höchstens zwei normalisierte Darstellungen der Form (2). Besitzt x zwei Darstellungen, so ist die eine abbrechend, d.h. in der Form:

$$x = \pm \sum_{i=1}^{s} d_i b^{r-i}, d_i \neq 0 \tag{4}$$

Und die andere ist gegeben durch:

$$x = \pm \sum_{i=1}^{s} d_i b^{r-i} \mp b^{r-s} \pm \sum_{i-s+1}^{\infty} (b-1) b^{r-i} \tag{5}$$

Wobei letztere eventuell noch nicht normalisiert ist.

Die auf einem (idealisierten) Rechner verfügbaren Zahlen (sog. Maschinenzahlen) seien gegeben durch:

$$\mathbb{M}_{b,l} = \{\pm \sum_{i=1}^{l} d_i b^{r-1} | d_i \in \{0, \ldots, b-1\}, d_1 \neq 0, r \in \mathbb{Z} \cup \{0\}\} \subseteq \mathbb{R}. \tag{6}$$

Wobei $b \in \mathbb{N} \setminus \{1\}$und $l \in \mathbb{N}$.

Man nennt $\mathbb{M}_{b,l}$ die Menge der normalisierten Fließkommazahlen zur Basis b mit Mantissenlänge l.

Um mit einem gegebenen $x \in \mathbb{R}$ auf einem Rechner arbeiten zu können, muss x durch einen $\tilde{x} \in \mathbb{M}_{b,l}$ ersetzt werden. Man spricht dann von Rundung. Rundung ist eine Abbildungsfunktion $fl: \mathbb{R} \to \mathbb{M}_{b,l}$ mit :

$$fl(x) = x \text{ für alle } x \in \mathbb{M}_{b,l}, \text{ (Projektionseigenschaft)}$$

$$fl(x) \leq fl(y) \text{ für alle } x,y \in \mathbb{R} \text{ mit } x \leq y, \text{ (Monotonie)}$$

Die IEEE *(Institute of Electrical and Electronics Engineers)* hat die Darstellung von Gleitkommazahlen in ihrem Standard IEEE 754 seit 1985 reglementiert; beinahe alle modernen Prozessoren folgen diesem Standard. Ausnahmen sind einige IBM-Großrechnersysteme, die VAX-Architektur und einige Supercomputer, etwa von Cray. Die tatsächliche Darstellung im Computer besteht also aus einem Vorzeichen-Bit, einigen Mantissen-Bits und einigen Exponenten-Bits. Wobei die Mantisse meistens normiert ist und Zahlen im Intervall $[1; 2)$ darstellt. (Da in diesem Intervall das erste Bit mit der Wertigkeit Eins stets gesetzt ist, wird es meistens implizit angenommen und nicht gespeichert.) Der Exponent wird meistens im Biased-Format, oder auch im Zweierkomplement dargestellt. Des Weiteren werden zur Darstellung besonderer Werte (Null, Unendlich, keine Zahl) meistens einige Exponentenwerte, zum Beispiel der größtmögliche und der kleinstmögliche Exponent, reserviert. Durch die unterschiedliche binäre Darstellung der Zahlen kann es in beiden Systemen zu Artefakten kommen. Das heißt: Zahlen, die im Dezimalsystem „rund“ erscheinen, zum Beispiel 12,45, können im Binärsystem nicht exakt dargestellt werden. Stattdessen wird ihre Binär-Darstellung abgeschnitten, so dass man bei der Rückumwandlung ins Dezimalsystem den Wert 12,44999999900468785 erhält. Dieses kann in nachfolgenden Berechnungen zu unvorhergesehenen Ab- oder Aufrundungsfehler führen. *[6]*

5.2 Zufallszahlen

Eine Zufallszahl ist eine Zahl, die unabhängig von seinen erzeugenden Kriterien produziert wird. Es ist schwer und sehr aufwendig eine reine Zufallszahl zu erzeugen. Echte Zufallszahlen können nur durch zufällige Phänomene gewonnen werden wie Würfeln, Roulette, Rauschen elektronischer Bauelemente (z.B. Z-Diode), radioaktive Zerfallsprozesse oder quantenphysikalische Effekte. Man nennt diese Verfahren: physikalische Zufallszahlengeneratoren. Sie sind jedoch zeitlich oder technisch recht aufwändig. In der Praxis genügt häufig eine Folge von Pseudozufallszahlen, das sind scheinbar zufällige Zahlen. Sie sind also nicht zufällig, da sie sich vorhersagen lassen, haben aber ähnliche statistische Eigenschaften (gleichmäßige Häufigkeitsverteilung, geringe Korrelation) wie echte Zufallszahlenfolgen. Solche Verfahren nennt

man Pseudozufallszahlengeneratoren. Um solche Zahlen zu produzieren, greift man auf verschieden Wahrscheinlichkeitsverteilungen zurück, die man aus der Stochastik wie der Normalverteilung (wird auch Gaußverteilung genannt), Geometrischeverteilung (nennt man auch Uniformverteilung), Exponentialverteilung, Poissonverteilung etc ... kennt. Es werden die verschiedene für uns relevanten Verteilungen kurz in den Abschnitten 5.3 bis 5.5 geschildert.

5.3 Uniformverteilung

Die Wahrscheinlichkeit des Auftauchens einer Zufallsvariable mittels der Uniformverteilung in einem Interval $[a, b]$ ist definiert mit der Wahrscheinlichkeitsdichte $f(x)$ und der Verteilungsfunktion $F(x)$:

$$f(x) = \begin{cases} 0, x \leq a \\ \frac{1}{a+b}, a < x < b \\ 0, x \geq b \end{cases} \tag{7}$$

$$F(x) = \begin{cases} 0, x \leq a \\ \frac{x-a}{b-a}, a < x < b \\ 1, x \geq b \end{cases} \tag{8}$$

Mit dem Mittelwert $\mu = \frac{a+b}{2}$ und der Standardverteilung $\sigma_x = \frac{b-a}{2\sqrt{3}}$.
In dieser Arbeit interessieren uns die Werte in dem Bereich (0,1).

5.4 Gaußverteilung

Die Gaußverteilung ist definiert durch die Wahrscheinlichkeitsdichte
f: $\mathbb{R} \rightarrow \mathbb{R}_{>}0$, x$\mapsto$f(x), so dass:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} exp(-\frac{1}{2}(\frac{x-\mu}{\sigma})^2) \tag{9}$$

Wird auch geschrieben als : X $\sim \mathcal{N}$ (μ, σ^2), wobei μ der Erwartungswert und σ die Standardabweichung sind.

Die Verteilungsfunktion der Normalverteilung ist gegeben durch:

$$F(x) = \frac{1}{\sigma\sqrt{2}\pi} \int_{-\infty}^{x} exp(-\frac{1}{2}(\frac{t-\mu}{\sigma})^2)dt. \tag{10}$$

Um standardnormalverteilte Zufallszahlen zu erzeugen, braucht man (nach Box-Muller-Methode) zwei Standardzufallszahlen u_1 und u_2, diese lassen

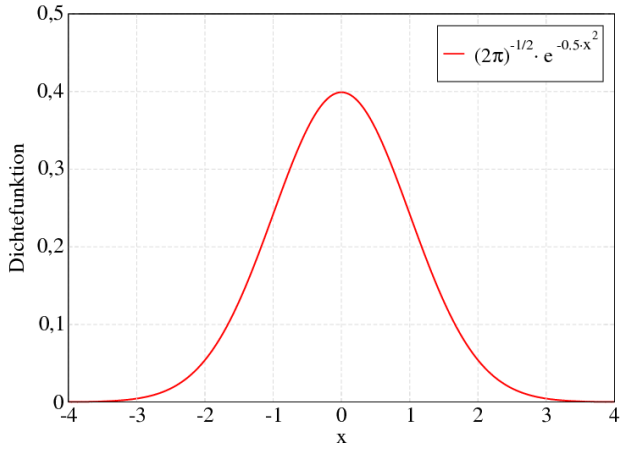


Abbildung 9: Dichtefunktion der Standardnormalverteilung [6]

sich z.B. durch Uniformverteilung erzeugen und dann in die Polarkoordinaten (r,φ) transformieren, wobei r=$\sqrt{-2.\ln(1-u_1)}$ und φ=$2\pi u_2$. Nun kann man die standardnormalverteilte Zufallszahlen z_1 und z_2 wie folgt berechnen:

$$z_1 = r.\cos(\varphi) = \sqrt{-2\ln(1-u_1)}.\cos(2\pi.u_2)$$
$$z_2 = r.\sin(\varphi) = \sqrt{-2\ln(1-u_1)}.\sin(2\pi.u_2)$$

Um mit der Box-Muller-Methode Normalverteilung mit beliebigen Parametern zu erzeugen, lassen sich die erhaltenen z_i nach dem Muster $x_i = \mu + \sigma \cdot z_i$ transformieren. *[6]*

Dies auf einem Rechner zu implementieren, wäre sehr aufwendig, da die Funktionen $\sin(x)$, $\cos(x)$, $\ln(x)$ und $\sqrt{x}$ sehr komplex sind und man sie nicht genau berechnen kann. Eine weitere Methode ist die Polar-Methode. Die Polar-Methode ist schneller als die Box-Muller-Methode, weil die Polar-Methode nur die $\ln(x)$ Funktion benutzt.

Algorithm 1: Gaußverteilung durch die Polar-Methode

Eingabe: Uniforme Zufallszahlen u_1 und u_2
Ausgabe: Gaußverteile Zahlen x_1 und x_2
(1) Berechne $v = (2u_1 - 1)^2 + (2u_2 - 1)^2$;
(2) falls $v \geq 1$ wiederhole (1).;
$x_1 = (2u_1 - 1)(\sqrt{\frac{-2\ln(v)}{v}})$;
$x_2 = (2u_2 - 1)(\sqrt{\frac{-2\ln(v)}{v}})$;

5.5 Exponentialverteilung

Die Exponentialverteilung ist definiert mit der Wahrscheinlichkeitsdichte

$$f_\lambda(x) = \begin{cases} \lambda \cdot \exp(-\lambda x), x \geq 0 \\ 0, x < 0 \end{cases} \tag{11}$$

mit dem reellen Parameter λ und dem Erwartungswert $\frac{1}{\lambda}$. Für $x = 0$ nimmt die Dichtefunktion ihren maximalen Wert $f_{max} = \lambda$.

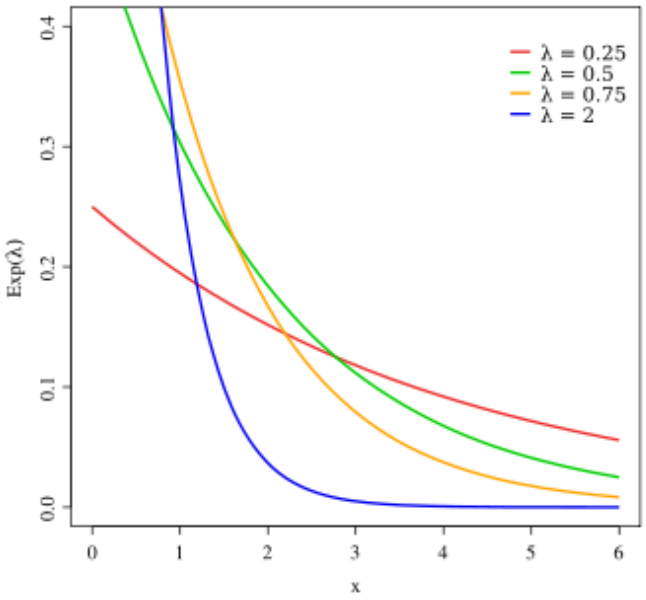


Abbildung 10: Dichte der Exponentialverteilung mit verschiedenen Werten λ. [6]

Sei X eine Zufallsvariable in (0,1) standardverteilt (z.B. mittels Uniformverteilung) und $\lambda = 1$, dann ist die Exponentialverteilung mit dem Parameter λ wie folgt definiert:

$$F(X) = -\frac{1}{\lambda} \ln(X) \tag{12}$$

5.6 Poissonverteilung

Die Poissonverteilung wird manchmal als die Verteilung der seltenen Ereignisse bezeichnet. Bezeichnet wird die Wahrscheinlichkeitsfunktion mit P_λ. Allgemein wird die Poisson-Wahrscheinlichkeitsfunktion wie folgt definiert:

$$P(X = k) = \frac{\lambda^k}{k!} \exp(-\lambda) \quad mit \quad k \in \mathbb{N},\ \lambda \in \mathbb{R}_+^* \tag{13}$$

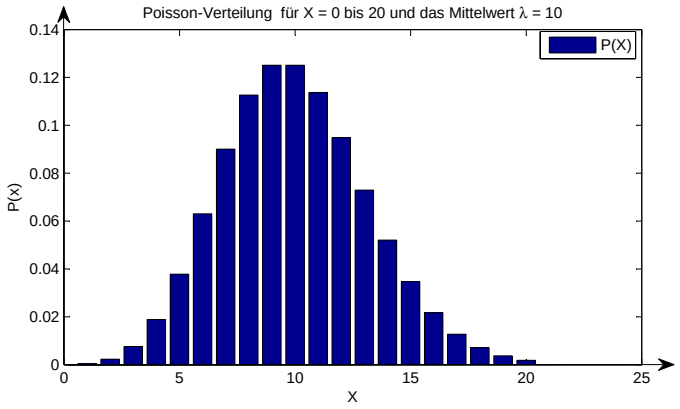


Abbildung 11: Verteilungsfunktion der Poissonverteilung mit $\lambda = 10$

- k ist die Variable
- λ bestimmt Varianz und Erwartungswert gleichzeitig

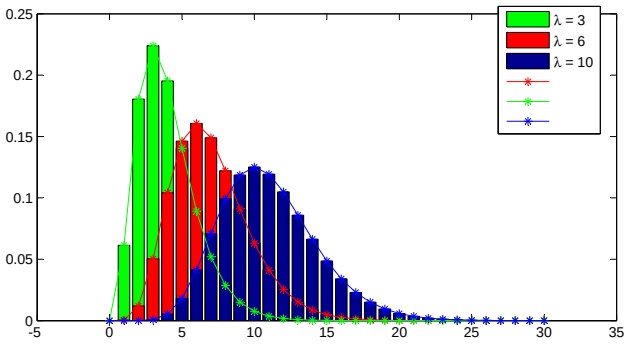


Abbildung 12: Poisson-Verteilungsfunktion mit λ =3, 6 und 10

Poisson-Verteilungsfunktion ist komplett durch λ charakterisiert.

6 Entwurf

6.1 Modellvorstellung

Es wird anfangs von einem Blackbox-Modell ausgegangen und langsam in die Spezifikation von einzelnen Blöcke in die Richtung Verhalten und Strukturalle Modellierung gegangen. Als Eingabewerte sind gegeben:

Schwellenwert: S, der überschritten werden muss, um ein Neuron als erregt zu bezeichnen.

Rauschintensität: n $|n < S$ besagt, wie groß der Einfluss des Rauschens auf das Signal ist.

INT_{max}**:** Obergrenze des Mittelwerts für die Poisson-Verteilung bzw. Anzahl Neuronen.

Als Ausgabe wird eine poissonverteilte Zahl ausgegeben. Mit Hilfe der Rauschintensität wird bei der Initialisierung des Mittelwert λ die Poissonverteilung berechnet. Hierzu wird ein Interval [INT_{min},INT_{max}] eingeführt, wo der Mittelwert λ liegen darf; dieses Interval ist proportional zu dem Interval [r,s], wo die Rauschintensität liegen darf, wobei r das Ruhepotential des Neurons darstellt (hier wird zur Einfachheit r und INT_{min} auf 0 gesetzt).

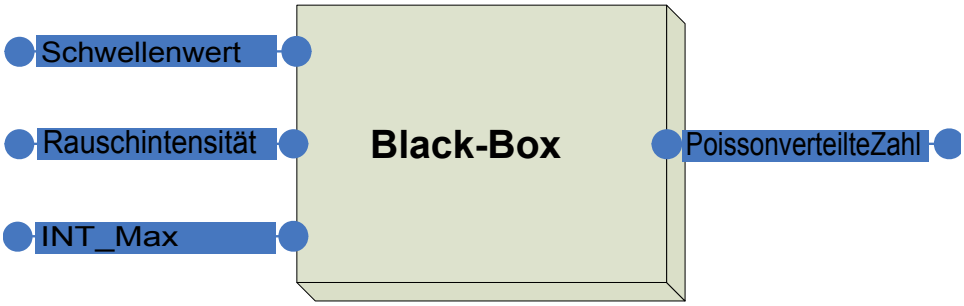


Abbildung 13: Modelldarstellung als Blackbox

Der Mittelwert wird aus n, s und INT_{max} anhand der Formel 15 ermittelt:

$$\frac{n-r}{s-r} = \frac{\lambda - INT_{min}}{INT_{max} - INT_{min}} \tag{14}$$

und daraus ergibt sich wegen $r = 0$ und $INT_{min} = 0$:

$$\lambda = \frac{\lambda}{s} \cdot INT_{max} \tag{15}$$

Mit diesem λ werden poissonverteilte Zufallszahlen generiert, die dann als λ^* bezeichnet werden und wiederum mit 14 die tatsächlichen Werte berechnet werden, die dem entsprechenden Neuron als Rauschen zugeordnet und als n^* bezeichnet werden.

$$n^* = \frac{\lambda^* \cdot s}{INT_{max}} \tag{16}$$

Zur Erzeugung von Poissonzahlen mit $k \leq 10$ wird direkt mit 13 berechnet, indem man die k! in einer LUT (Look-Up-Table) zuvor berechnet und speichert. Für größere k wird (13) approximiert mittels Stirling-Approximation.

$$k! \approx \sqrt{2\pi k}\ \left(\frac{k}{\mathrm{e}}\right)^k$$

$\ln(k!) = k\ln(k) - k + \frac{1}{2}\ln(2\pi k) + \frac{1}{12k} - \frac{1}{360k^3} + \frac{1}{1260k^5} - \frac{1}{1680k^7} + \cdots$

Für $k > 1000$ genügt der Ausdruck $\ln(k!) \approx k\ln(k) - k$, um den relativen Fehler kleiner als 1 % zu halten. Für sehr große n, $k > 10^{44}$ genügt $\ln(k!) \approx k \cdot \ln(k)$.
Der resultierende Ausdruck:

$$P(X = k) = \frac{1}{\sqrt{2\pi k}} \exp(k\ln(1+v) - (\lambda - k) - \delta), \tag{17}$$

wobei $v = \frac{\lambda - k}{k}$ und $\delta = \frac{1}{12k} - \frac{1}{360k^3} + \frac{1}{1260k^5}$.

$k\ln(1+v) - (\lambda - k) - \delta$ kann für $|v| \leq 0.25$ erweitert werden zu $kv^2\phi(v)$, wobei $\phi(v)$ durch das Polynom:

$$\phi(v) = -\frac{1}{2} + \frac{v}{3} - \frac{v^2}{4} + \frac{v^3}{5} - \cdots \approx \sum_{i=0}^{n} a_i v^i \tag{18}$$

approximiert wird. Die Koeffizienten a_i mit einer Genauigkeit von 8 bis 10 Stellen werden in einer Tabelle gespeichert und bei Bedarf entnommen.

Was jetzt in der Black-Box abläuft, soll anhand des Algorithmus von [1] *Kapitel 4 Abschnitt 4.3.1 Algorithmus im Stil von Donald E.Knuth, Seite 21* näher erläutert werden.

Fall (A). Eingabe: Mittelwert $\lambda \geq 10$. Ausgabe: Poissonverteilte Zahl $\mathcal{K}$. Fall (A) benötigt eine Tabelle der Koeffizienten a_i und eine Tabelle von k! (für k = 0, 1, . . . , 9). Bei der Initialisierung werden die drei von λ abhängigen Variablen $s := \sqrt{\lambda}$, $d := 6\lambda^2$und $L := \lfloor \lambda - 1.1484 \rfloor$ berechnet.

N (Normalverteilte Variable):
Generiere $\mathcal{T}$ (standardnormalverteilt) und setze $G := \lambda + s.\mathcal{T}$. Wenn $G \geq 0$, setze $\mathcal{K} := \lfloor G \rfloor$. In dem seltenen Fall $G < 0$ ist sofortige Ablehnung indiziert, also: Wenn $G < 0$, gehe zu **P**.

L (L-Kriterium):
Wenn $\mathcal{K} \geq L$ return $\mathcal{K}$.

Q1 (Quetschfunktion):
Generiere $\mathcal{U}$ ((0,1)-gleichverteilt).
Wenn $d \cdot \mathcal{U} \geq (\lambda - \mathcal{K})^3$ return $\mathcal{K}$.

P (Vorbereitung fur die Schritte Q2 und D):
Die folgenden acht von λ abhängigen Variablen werden berechnet:
$w := \frac{1}{s\sqrt{2\pi}}$, $b_1 := \frac{1}{24\lambda}$, $b_2 := \frac{3}{10}b_1^2$, $c_3 := \frac{1}{7}b_1 b_2$, $c_2 := b_2 - 15c_3$, $c_1 := b_1 - 6b_2 + 45c_3$, $c_0 := 1 - b_1 + 3b_2 - 15c_3$ und $c := \frac{0.1069}{\lambda}$.
Wenn $G \geq 0$, Aufruf von Prozedur **F** zur Berechnung von p_x, p_y, f_x und f_y.
Wenn $G < 0$, gehe zu **E** (**Q2** wird übersprungen).

Q2 (Quotientenkriterium):
Wenn $f_x(1 - \mathcal{U}) \leq p_y e^{(p_x - f_x)}$, return $\mathcal{K}$.

E (Doppelt exponentialverteilte Instanz):
Generiere E (standard exponentialverteilt) und $\mathcal{U}$ ((0,1)-gleichverteilt).
Setzte $\mathcal{U} := \mathcal{U} + \mathcal{U} - 1$ und $\mathcal{T} := 1.8 + Esgn(\mathcal{U})$.
Wenn $\mathcal{T} \leq -0.6744$, starte Schritt **E** von vorn. Andernfalls setze $\mathcal{L} := \lfloor \lambda + s\mathcal{T} \rfloor$ und rufe Prozedur **F** auf, um px, py und fx, fy zu berechnen.

D (Differenzfunktion):
Wenn $c|\mathcal{U}| \geq p_y e^{(p_x - E)} - f_y e^{(f_x + E)}$, gehe zurück zu Schritt **E**. Andernfalls, return $\mathcal{K}$.

F Prozedur **F**:
1.Poissonwahrscheinlichkeiten $p_{\|} = P_x e p_y$
Falls $\mathcal{K} < 10$: Setze $px := -\lambda$ und $p_y := \frac{\lambda^{\mathcal{K}}}{\mathcal{K}!}$ mit Hilfe der Tabelle k!.
Falls $\mathcal{K} \geq 10$: Setze $\delta := \frac{1}{12\mathcal{K}}, \delta = \delta - 4.8\delta^3$ und $V := (\frac{\lambda - \mathcal{K})}{\mathcal{K}}$. Dann $p_x := \mathcal{K}\ln(1 + V) - (\lambda - \mathcal{K}) - \delta$. Für den Fall, dass $|V| \leq 0.25$, substituiere $p_x := \mathcal{K}V^2 \sum_i a_i \cdot V^i$ unter Benutzung der Koeffizienten a_i für höheren Genauigkeitsgrad. Setze $p_y := \frac{1}{\sqrt{2\pi} \cdot \sqrt{\mathcal{K}}}$.
2. Diskrete Normalwahrscheinlichkeiten:$f_{\mathcal{K}} = f_y e^{fx}$

Setze $\mathcal{X} := \frac{(\mathcal{K}-\lambda+0.5)}{s}, f_x := 0.5\mathcal{X}^2$ und $f_y := w(((c_3\mathcal{X}^2 + c_2)\mathcal{X}^2 + c_1)\mathcal{X} + c_0)$.

Fall (B). Eingabe: Mittelwert $\lambda < 10$. Ausgabe: Poissonverteilte Zahl $\mathcal{K}$.

Fall (B) Wird durch eine Tabellen-unterstützte Inversionsmethode behandelt. Die Tabelle enthält die kumulativen Wahrscheinlichkeiten $P_{\mathcal{K}}(\mathcal{K} = 1, 2, \cdots, 35)$ für eine Genauigkeit von neun signifikanten Stellen. Bei der Initialisierung werden die fünf Variablen $\mathcal{M} := max(1, \lfloor\lambda\rfloor), \mathcal{L} := 0, p := e^{-\lambda}, q := p$ und $p_0 := p$ gesetzt.

U (gleichverteilte Variable) Generiere $\mathcal{U}((0,1) - gleichverteilt)$. Setze $\mathcal{K} := 0$. Wenn $\mathcal{U} \leq P_0$, return $\mathcal{K}$.

T (Vergleich mit existierender Tabelle) Wenn $\mathcal{L} = 0$(leere Tabelle der $P_{\mathcal{K}}$), gehe zu Schritt **G**. Andernfalls setze J:=1, aber wenn $\mathcal{U} > 0.458$, setze $J := min(\mathcal{L}, \mathcal{M})$ (weil, wenn $\mathcal{U} > 0.458 > P_9$(bei $\lambda = 10$), wird $\mathcal{K}$ niemals kleiner sein als $\lfloor\lambda\rfloor$).
Für $\mathcal{K} := J, J+1, \cdots, \mathcal{L}$ tue: Sobald $\mathcal{U} \leq P_{\mathcal{K}}$, return $\mathcal{K}$. Ist diese Suche erfolglos und $\mathcal{L} = 35$, gehe zurück zu Schritt **U**.

G (Generierung der Tabelle $P_{\mathcal{K}}$ und Vergleich mit $\mathcal{U}$) Für $\mathcal{K} = \mathcal{L}+1, \mathcal{L}+2, \cdots, 35$ tue: Setze $p := \frac{p\lambda}{\mathcal{K}}, q := q + p, P_{\mathcal{K}}q$ und sobald $\mathcal{U} \leq q$, setze $\mathcal{L} := \mathcal{K}$ und return $\mathcal{K}$. Ist diese Suche erfolglos, setzte $\mathcal{L} := 35$ und gehe Zurück zu Schritt **U**.

Zur Realisierung des System werden Zahlen in Form von Floating-Point-Zahlen gebraucht, da VHDL reale Zahlen nicht synthetisieren kann. Dazu werden Bauteile für Grundoperationen wie Multiplikation, Division, Addition und Substraktion gebraucht. Die Bauteile werden in Form von Entitäten entworfen. Es müssen auch bestimmte Funktionen implementiert werden, wie $exp(x)$, $ln(x)$ und $\sqrt{}(x)$. Zur Erzeugung von Zufallszahlen muss für jeden Typ (Gauss-, Uniform- und Exponentialverteilte Zahlen) ein Generator entworfen und implementiert werden. Die Grundidee ist, dass Funktionen und Generatoren als Procedure in einem Paket implementiert werden. Im Laufe des Programms werden diese dann aufgerufen. Es wird dann versucht das ganze System in Blöcke zu teilen. Jeder Block hat eine bestimmte Aufgabe wie es im Algorithmus ablaufen soll. Es werden noch Komparatoren und Look-up-Tables benötigt. Als Paket werden die von VHDL.org Floating-Point-Pakete benutzt, dort sind ein paar Funktionen vordefiniert wie `+ , - , * , /`. Die Floating-Point-Pakete sind synthetisierbar (laut VHDL.org). Nach ein paar Tests wurde festgestellt, dass die Division an sich nicht korrekt abläuft und nicht synthetisierbar ist. Nach einer kleinen Änderung wurde das Problem behoben.

6.2 Floating-Point

Die behandelten Zahlen sind von Typ Floating-Point und werden wie in der Formel 3 dargestellt. Jede Zahl wird definiert mit seinem Vorzeichen v, Exponenten e, Mantisse m und Basis $b = 2$

$$x = v.2^e.m \tag{19}$$

Floating-Point-Zahlen sind in ***IEEE-754*** (32 und 64 Bit Länge) und in ***IEEE-854*** (variable Länge) definiert. Zur Realisierung dieser Zahlenformate wird im dem Fall der 16 Bit-Formate einen Array der Länge 16 gebraucht, wobei das erste Bit das Vorzeichen v repräsentiert("0"für Positiv und "1"für Negativ), die nachfolgenden 5 Bits präsentieren den Exponenten e und die restlichen 9 Bits präsentieren die Mantisse m.

V	E	E	E	E	E	E	M	M	M	M	M	M	M	M	M
Vorzeichen	Exponent						Mantisse								

Der Datentyp Float im VHDL Paket ist wie folgendes deklariert :

```
subtype float16 is float (6 downto -9);
subtype float32 is float (8 downto -23);
subtype float64 is float (11 downto -52);
```

Die verschiedenen Floating-Point-Formate von ***IEEE-754-Standard*** werden in der Tabelle 1 vorgestellt.

Floating-Point Formate				
Parameter	Format			
	Singel	Singel Extended	Double	Double Extended
Präzision	24	≥32	53	≥64
e_{max}	+127	≥1023	1023	≥16383
e_{min}	-126	-1022	-1022	-≤-16382
Exponent-bias	127	nicht definiert	1023	nicht definiert
Exponentenlängein Bit	8	≥ 11	11	≥ 15
Formatlänge	32	≥ 43	64	≥ 79

Tabelle 1: Floating-Point Formate nach dem Standard IEEE-754 [9]

Zu unserem Entwurf in VHDL, werden Pakete, die diese Formate unterstützen, benutzt. Die Pakete sind von ***EDA Industry Working Groups*** unter der Web-Seite Adresse[10] zur Verfügung gestellt worden. Es wird zusätzlich ein Paket angefertigt, das die entworfene Funktionen und Konstanten, die man zur Implementierung des Modells benötigt, beinhaltet. Die Pakete befinden sich in einer Bibliothek:"***IEEE_proposed"***.

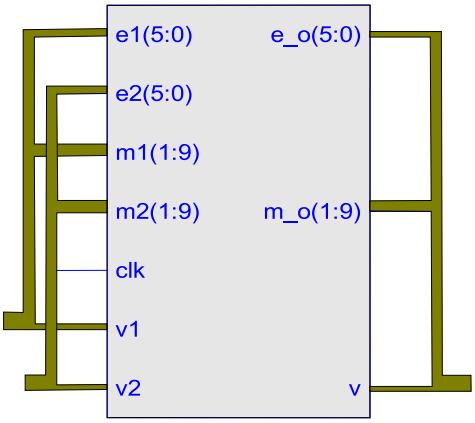


Abbildung 14: Beispiel eines Floating-Point Entity

6.3 Elementar Funktionen

Zu den elementaren Funktionen, die zum Entwurf des System notwendig sind, gehören die folgenden Funktionen:

- Addition/Substraktion
- Multiplikation/Division
- Exponential-Funktion: $\exp(x)$
- Der Natürliche Logarithmus $\ln(x)$
- Quadratwurzel-Funktion : $\sqrt{x}$.

Es gibt verschiedene Verfahren zum Lösen solcher Funktionen z.B. Reihenentwicklung, Konvergenzverfahren, direkte Methoden, tabellenbasierte Verfahren; im Allgemeinen braucht man eine höhere Rechenleistung. Man denkt zuerst an die Reihenentwicklung, aber die Unendlichkeit der Reihe macht die Sache kompliziert. Man kann bis auf eine Stelle $i = n$ berechnen und dann die Fehler für den Abbruch bei n berechnen, was zu einen ineffizienten und höheren Rechenaufwand führt. Um eine Reihenentwicklung in der Hardware zu berechnen, benötigt man schnelle Gleitkomma-Multiplizierer, - Divider, Addierer, Subtrahierer und Tabellen für die vorgerechneten Werte, was eigentlich einen höheren Hardwareaufwand erzielt. Als Alternative kann man mit Konvergenzverfahren nur mit primitiven Operationen, wie Addition, Subtraktion, Verschiebung um n Bits nach rechts (entspricht die Division bei 2^n) und die Verschiebung um n Bits nach links (entspricht die Multiplikation bei 2^n), rechnen. Die Idee hinter dem Konvergenzverfahren

basiert auf der Approximation von einer Funktion $y = f(x)$ durch zwei rekursive Formeln: x_{i+1} und y_{i+1}. Man initialisiert x_i mit $x_0 = x$ und lässt x_i nach einem Wert w (typisch 0 oder 1) unter Benutzung von Hilfswerten b_i (gespeichert in einer Tabelle) konvergieren. Die zweite Formel verwendet die Hilfswerte b_i, um y zu approximieren.

6.3.1 Floating-Point Add/Sub

Um zwei Fließkommazahlen zu addieren bzw. zu substrahieren, müssen sie zu erst so transformiert werden, dass die beiden Zahlen den gleichen Exponenten haben ***(eng:alignment)*** und dann kann man die Addition/Substraktion wie gewöhnlich durchführen. Um die Exponenten anzugleichen, kann man auf Shift-Operationen zugreifen.

Hier ist ein Beispiel in dezimaler Darstellung :

$1234.56 = 1.23456.10^3$
$18.7654 = 1.87654.10^1 \quad = 0.0187654.10^3$

Addition/Substraktion:

$$\begin{aligned} 1234.56 &= 1.23456.10^3 \pm \ 1.87654.10^1 \\ &= 1.23456.10^3 \pm \ 0.0187654.10^3 \\ &= (1.23456 \pm 0.0187654).10^3 \end{aligned}$$

Zum Entwurf einer Add/Sub Entity wird folgender Algorithmus benutzt: Es wird eine Variable op gewählt, um festzulegen, was für eine Operation durchzuführen ist. Für die Addition sei op = 0 und für die Substraktion 1.

Algorithm 2: Floating-Point Add/Sub

Eingabe: $A = v_1.e_1.m_1$ und $B = v_2.e_2.m_2$
Ausgabe: $A \pm B$
if *op=1* **then**
  $v_2 := 1 - v_2$;
if $e_1 < e_2$ **then**
  $swap(v_1, v_2)$; $swap(m_1, m_2)$; $swap(e_1, e_2)$;
$e := e1$; $m_2 := m_2/2^{(e_1 - e_2)}$; $v := v_1$;
if v *xor* $v_2 = 0$ **then**
  $m := m_1 + m_2$;
  if $m > 2$ **then**
    $e = e + 1$;
    $m = m/2$;
else
  if $e_1 = e_2$ *and* $m_1 < m_2$ **then**
    $swap(m_1, m_2)$;
    $v := 1 - v$;
  $m := m_1 - m_2$;
  leadingzeroes(m, n);
  $m := m * 2^n$;
  $e := e - n$;
$m := round(m)$;
if $m \geq 2$ **then**
  $e := e + 1$; $m := m/2$

$$\begin{aligned} A \pm B &= v_1 \cdot e_1 \cdot m_1 \pm v_2 \cdot e_2 \cdot m_2 \\ &= v \cdot e \cdot m \end{aligned}$$

Es kann sein, dass m sehr klein und somit mit vielen führenden Nullen versehen ist. In diesem Fall muss das Ergebnis normalisiert werden. Um m zu normalisieren, wird eine Prozedur leadingzeroes definiert.

procedure leadingzeroes(M: in float; k: out natural)

Diese Prozedur gibt die Anzahl der führenden Nullen in der Mantisse an, diese werden dann von dem Exponenten e subtrahiert.

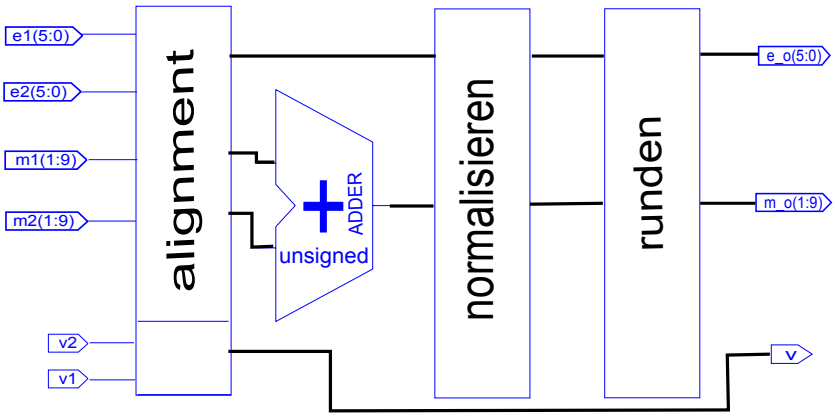


Abbildung 15: Floating-Point Adder Modell

6.3.2 Floating-Point Mult/Div

Seien zwei Zahlen A und B vom Typ Float $A = v_1 \cdot 2^{e1} \cdot m_1$ und $B = v_2 \cdot e_2 \cdot m_2$

$$\begin{aligned} A * B &= v_1 \cdot e_1 \cdot m_1 * v_2 \cdot e_2 \cdot m_2 \\ &= min(v_1, v_2) \cdot (e_1 + e_2) \cdot m_1 \cdot m_2 \\ &= v \cdot e \cdot m \end{aligned}$$

$m_1 \cdot m_2$ ergibt ein m mit doppelter Länge als m_1 und m_2. Um in demselben Format zu bleiben, muss m normalisiert werden, indem man m durch $m/2$ und den Exponenten e durch $e + 1$ substituiert. Das Resultat muss dann gerundet werden, dafür ist die Funktion ***round(m)*** zuständig. Der Übergang von einem Format zum anderen erfolgt in ***"float_pkg"*** durch die Funktion ***resise*** oder ***to_float(Zahl, Exponent_länge, Mantissen_länge)***.

Die Multiplikation $A * B$ kann durchgeführt werden, indem man die Vorzeichen von beiden Zahlen mit einem XOR-Gatter verbindet, das Resultat ist dann das Vorzeichen von dem Ergebnis. Die Exponenten werden addiert und die Mantissen werden multipliziert.

Algorithm 3: Floating-Point Multiplikation

Eingabe: $A = v_1.e_1.m_1$ und $B = v_2.e_2.m_2$
Ausgabe: $A * B$
$v := v_1 xor v_2$;
$m := m_1 * m_2$;
$e := e_1 + e_2$;
if $m >= 2$ **then**
  $e := e + 1$; $m := m/2$;
$m := round(m)$;
if $m >= 2$ **then**
  $e := e + 1$; $m := m/2$;

Beispiel: Sei das 10er System und eine Genauigkeit 10^{-4}, so dass die Zahlen in der Form $v.m.10^e$ mit $1 \leq m \leq 9.9999$ darstellbar sind.
Berechnung von $R = (3.5372 * 10^3) * (2.4481 * 10^{-1})$:

R= 8.65941932$*10^2$,
8.65941932 < 10,
rounding:m$\simeq$ 8.7574,
8.6594< 10,
R= 8.6594$*10^{-2}$.

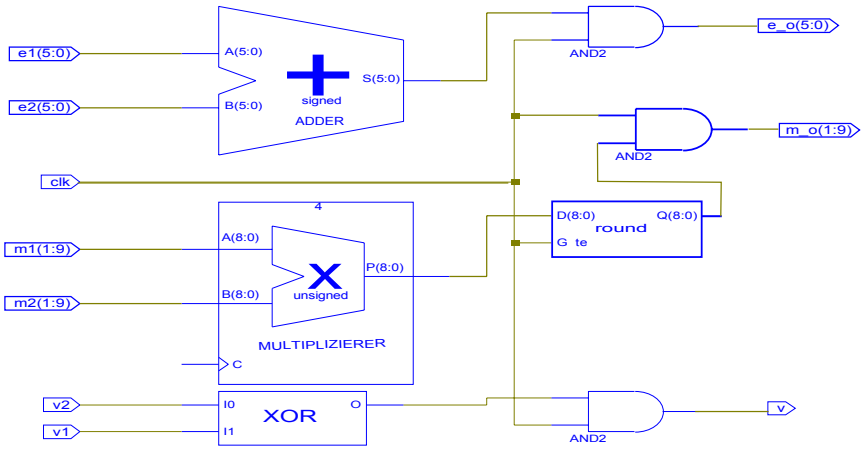


Abbildung 16: Floating-Point Multiplizierer Modell

Die Division A/B wird sich ergeben, indem man die Vorzeichen mit XOR-Gatter verbindet, die Mantisse von A durch die Mantisse von B dividiert und der Exponent ergibt sich durch die Substraktion: (Exponent von A)-(Exponent von B).

Algorithm 4: Floating-Point Division

Eingabe: $A = v_1.e_1.m_1$ und $B = v_2.e_2.m_2$
Ausgabe: $A \div B$
$v := v_1 xor v_2$;
$m := m_1/m_2$;
$e := e_1 - e_2$;
if $m < 1$ **then**
  $e := e - 1$; $m := m * 2$;
$m := round(m)$;

Beispiel: Sei wieder das 10er System

Berechnung $Q = (3.5372 * 10^3)/(2.4481 * 10^{-1})$:

`Q=` $1.444875618 * 10^4$,
$1.444875618 \geq 1$,
`rounding:m`$\simeq 1.4448$,
`Q=` $1.4448 * 10^4$.

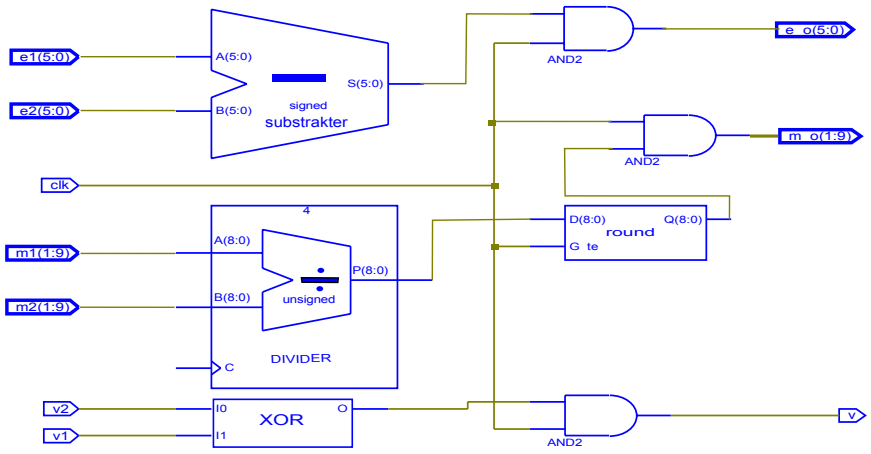


Abbildung 17: Floating-Point Divider Modell

6.3.3 Berechnung von Floating-Point exp(x) und ln(x)

Für die Exponential-Funktion $exp(x)$ werden die folgende Rekursionen verwendet:

$$x_{i+1} = x_i - \ln(b_i) \quad (20)$$
$$y_{i+1} = y_i \cdot b_i, \quad (21)$$

wobei b_i so gewählt ist, dass bei m Schritten (wenn eine Genauigkeit von m Stellen erzielt werden soll) der Wert $w = 0$ erreicht wird. Es gilt:

$$y_{i+1}.e^{x_{i+1}} = y_i \cdot e^{x_i - \ln b_i} = y_i.e^{x_i}$$

und somit auch

$$y_m.e^{x.m} = y_0.e^{x_0}$$

für $x_m = w = 0$ und $y_0 = 1$ ergibt sich das Resultat:

$$y_m = e^{x0} = e^x$$

und jetzt wählt man $b_i = 1 + s_i.2^{-i}$ mit $s_i \in \{-1, 0, 1\}$; und dabei ergeben sich die Formeln 20 und 21 wie folgt neu:

$$x_{i+1} = x_i - \ln(1 + s_i.2^{-1})$$
$$y_{i+1} = y_i.(1 + s_i.2^{-1})$$

Die Werte $\ln(1 + s_i.2^{-1})$ und $(1 + s_i.2^{-1})$ werden zuvor berechnet, dann in einer Tabelle gespeichert.

Die Tabelle wird als Look-up Table in einem ROM der Größe $2n$ x n gespeichert. Um $exp(x)$ zu berechnen, ist jetzt ein Vector $s = (s_0, s_1, \cdots, s_{m-1})$ zu finden, so dass:

$$x_m = x_0 - \sum_{l=0}^{m-l} \ln(1 + s_l.2^{-l}) = 0$$

$$\Longrightarrow \sum_{l=0}^{m-l} \ln(1 + s_l.2^{-l}) = x_0 \quad ,$$

mit der Annahme, dass $x > 0$, so ist $s = 0$ oder 1 und

$$x_0 \leq \sum_{i=0}^{m-l} \ln(1 + 2^{-i}) \approx 1.56.$$

i	$1+2^{-i}$	$ln(1+2^{-i})$	$1-2^{-i}$	$ln(1-2^{-i})$
0	10.0000000000	0.1011000110	0	-
1	1.1000000000	0.0110011111	0.1000000000	-.1011000110
2	1.0100000000	0.0011100100	0.1100000000	-.0100100111
3	1.0010000000	0.0001111001	0.1110000000	-.0010001001
4	1.0001000000	0.0000111110	0.1111000000	-.0001000010
5	1.0000100000	0.0000100000	0.1111100000	-.0000100001
6	1.0000010000	0.0000010000	0.1111110000	-.0000010000
7	1.0000001000	0.0000001000	0.1111111000	-.0000001000
8	1.0000000100	0.0000000100	0.1111111100	-.0000000100
9	1.0000000010	0.0000000010	0.1111111110	-.0000000010
10	1.0000000001	0.0000000001	0.1111111111	-.0000000001

Tabelle 2: $ln(1 \pm 2^{-i})$ Werte-Tabelle mit 10 Bit Präzision [4]

Die Berechnung läuft gemäss folgenden Algorithmus:

Algorithm 5: Berechnung von $exp(x)$ nach [4]

Eingabe: x
Ausgabe: $exp(x)$
For i in 0 to n loop
$d_i = x_i - \ln(1 + 2^{-i})$;
if $d_i \geq 0$ **then**
  $s_i = 1$; $x_{i+1} = d_i$;
else
  $s_i = 0$; $x_{i+1} = x_i$;

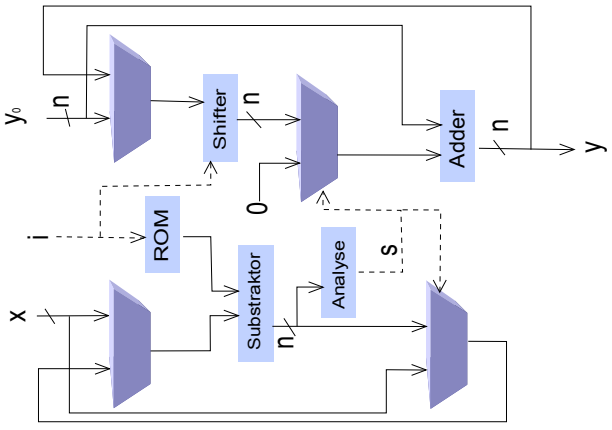


Abbildung 18: Hardware-Modell zur Berechnung von $exp(x)$

Zur Berechnung von einer $\ln(x)$ Funktion wird das gleiche Verfahren der $exp(x)$ Funktion mit ein paar Änderungen benutzt. Die Berechnung von $\exp(x)$ basiert auf der Summation von den Werten $\ln(1+s_l.2^{-i})$, um mit der Konvergenz den Wert x nach Null zu zwingen. Bei $\ln(x)$ wird die Konvergenz nach 1 und statt der Summation wird die Multiplikation mit vorgerechneten Faktoren verwendet.

Die rekursive, allgemeine Formel zur Berechnung von $\ln(x)$ lautet:

$$x_{i+1} = x_i \cdot b_i \tag{22}$$
$$y_{i+1} = y_i - \ln(b_i) \tag{23}$$

Die Werte b_i werden so gewählt, dass $x_{i+1} = \sum_{l=0}^{i} b_l$ nach 1 in m Schritten (an der Stelle $x_m = 1$) konvergiert. Der Faktor b_i ist gegeben in der Form $1 + s_i \cdot 2^{-i}$ und $s_i \in \{-1, 0, 1\}$.

$$x_m = x_0 \cdot \prod_{l=0}^{m-1} b_l = 1 \Longrightarrow \prod_{l=0}^{m-1} b_l = \frac{1}{x_0}$$

aus : $$y_{i+1} = y_i - \ln(b_i) \Longleftrightarrow y_{i+1} = y_0 - \sum_{j=0}^{i} \ln(b_j)$$
folgt $$y_{i+1} = y_0 - \prod_{j=0}^{i} b_j = y_0 + \ln(x_0) \text{ da } \ln(\tfrac{1}{x_0}) = -\ln(x_0)$$

mit $y_m = 0$ ergibt sich das gewünschte Resultat $y_m = \ln(x_0)$.

Anhand der Formeln 22 und 23 ist jetzt die Berechnung der Funktion $F(x_0) = \ln(x_0)$ nachvollziehbar. Es ist zu beachten, dass die Werte $\ln(1 + s_i \cdot 2^{-i})$ nicht neu berechnet werden müssen, weil die Tabelle LUT von $exp(x)$ diese Werte schon beinhaltet.

Algorithm 6: Berechnung von $ln(x)$ nach [4]

Eingabe: x
Ausgabe: exp(x)
For i in 0 to n loop
if $x_i(1 + 2^{-i}) \leq 1$ **then**
  $s_i = 1$;
  $x_{i+1} = x_i(1 + 2^{-i})$;
else
  $s_i = 0$;
  $x_{i+1} = x_i$;
end loop;

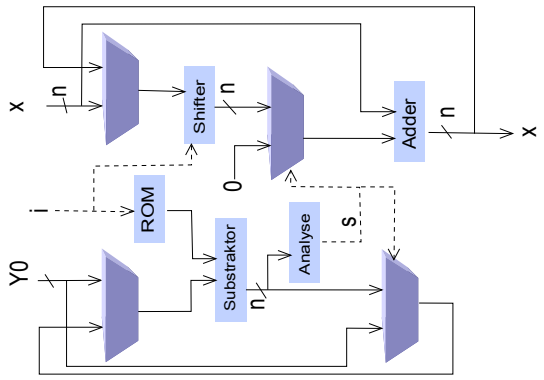


Abbildung 19: Hardware-Modell zur Berechnung von ln(x)

Mit dem oben geführten Algorithmus für $\exp(x)$ und $\ln(x)$ und den oben dargestellten Tabellen werden nur Zahlen zwischen 0 und 1.25 (*für* $\exp(x)$) und 0.21 und 3.45 *(für* $\ln(x)$*)* mit höherer Genauigkeit berechnet. Für andere Werte außerhalb der obengenannten Intervalle können Näherungsverfahren benutzt werden, da diese Funktionen nach bestimmten Werten fast linear werden. Es werden Intervalle definiert und ihre Grenzwerte werden vorgerechnet. Zu jeden Intervall wird eine Funktion $f(x) = m \cdot x + n$ zugeordnet, es handelt sich um die Funktion einer Geraden, die beide Endpunkte des Intervalls beinhaltet, wobei m der Anstieg der Gerade und n die Ordinate des Schnittpunktes der Gerade mit der Ordinatenachse X ist. Es muss nun die eingegebene Zahl zu einem Intervall zugeordnet und dann die Approximation durch den Einsatz in die Funktion der Gerade des jeweiligen Interval ermittelt werden. Diese Methode weist aber grosse Abweichung zwischen dem realen Wert und dem approximativen Wert auf. Um eine höhere Genauigkeit zu erreichen, müssen die Intervalle sehr klein gehalten werden, was wiederum einen höheren Aufwand an Speicher und Hardware-Komplexität benötigt, da immer geprüft werden muss, ob ein Wert in einem Intervall liegt.

Durch die Ausnutzung von den Exponential- und Logarithmus-Funktionen, den Rechenregeln und ihren Eigenschaften kann der Aufwand niedrig und die Genauigkeit hoch gehalten werden. Die wichtigsten Eigenschaften werden hier aufgelistet:

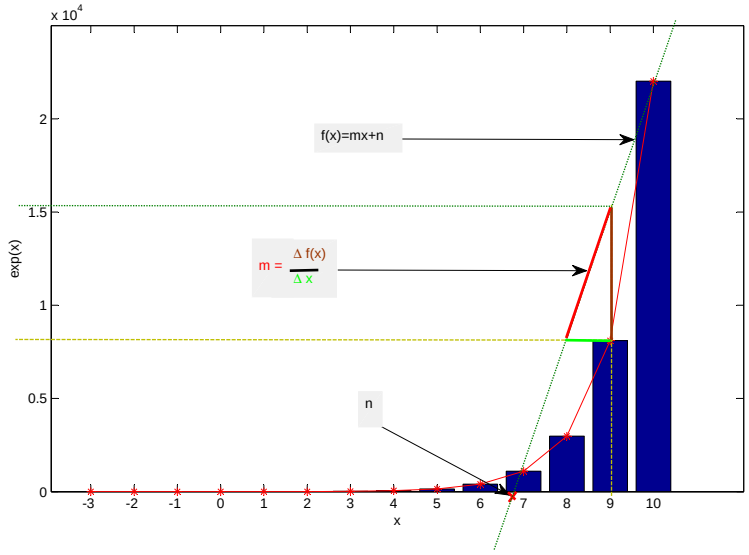


Abbildung 20: Exp(x) Funktion Näherungsmethode

- $x = e^{\ln(x)}$
- $e^x = 2^{x \cdot log_2(e)}$, wobei e die eulersche Zahl ist.
- $log_2(a) = \frac{\log(a)}{\log(2)}$
- $ln(a) = \frac{\log(a)}{\log(e)}$
- $log_2(e) \cdot ln(2) = \frac{\log(e)}{\log(2)} \cdot \frac{\log(2)}{\log(e)} = 1$

Jede Zahl x kann wie folgt dargestellt werden:

$$x = x \cdot log_2 e \cdot ln2 = I + F, \tag{24}$$

wobei I der Integer-Teil und F der Fraktion-Teil ist (und $0 \leq F < 1$), dann folgt:

$$e^x = e^{I \cdot ln2 + F \cdot ln2} = 2^I \cdot e^{F.ln2} \tag{25}$$

Bemerkung : $F \cdot ln2 \in [0, 0.69)$ also $e^{F.ln2} \simeq (f * ln(2)) + 1$.

Man kann das $ln(x)$ auf einer anderen Weg berechnen. Jede Zahl kann in der Form (19) geschrieben werden. Eine wichtige Eigenschaft von der Logarithmus-Funktion ist die Umwandlung des Produktes zweier Zahlen in eine Addition ihrer Logarithmen $ln(a.b) = ln(a) + ln(b)$. Wenn man diese Eigenschaft ausnutzt und in die Formel (19) einsetzt, bekommt man:

$$ln(2^e.m) = ln(2^e) + ln(m) = e.ln(2) + ln(m) \tag{26}$$

x	e^x	$x * log_2 e$	i	f	$2^i * e^{f.ln2}$	$2^i * (f.ln2 + 1)$
-2	0.135335283	-2.885390083	-2	-0.885390083	0.135335283	0.09657359
-1	0,367879441	-1,442695042	-1	-0,442695042	0,367879441	0,34657359
0	1	0	0	0	1	1
1	2.718281828	1.442695042	1	0.442695042	2.71828183	2.61370564
2	7.389056099	2.885390083	2	0.885390083	7.38905611	6.45482256
3	20.08553692	4.328085125	4	0.328085125	20.08553696	19.63858047
4	54.59815003	5.770780167	5	0.770780167	54.59815017	49.09645118
5	148.4131591	7.213475209	7	0.213475209	148.4131595	146.9401266
6	403.4287935	8.65617025	8	0.65617025	403.428795	372.4345751
7	1096.633158	10.09886529	10	0.098865292	1096.633163	1094.172875
8	2980.957987	11.54156033	11	0.541560334	2980.958001	2816.780326
9	8103.083928	12.98425538	12	0.984255375	8103.083972	6890.429802
10	22026.46579	14.42695042	14	0.426950417	22026.46592	21232.6724

Tabelle 3: Vergleich Berechnungsmethoden von $exp(x)$

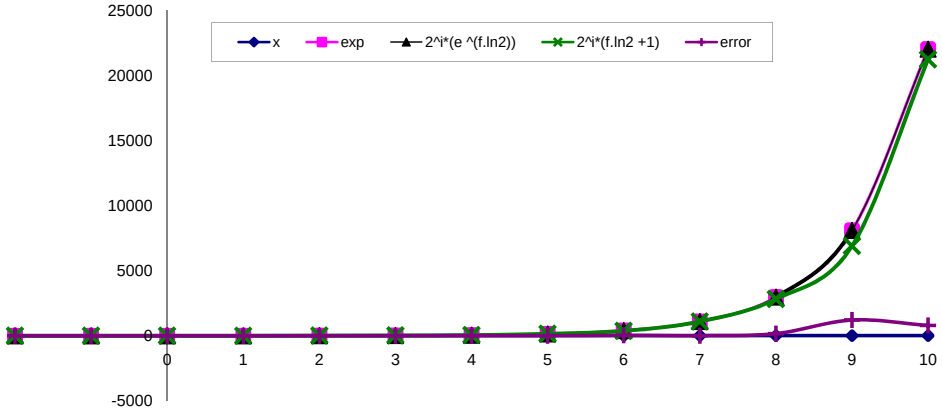


Abbildung 21: Vergleich Berechnungsmethoden von $exp(x)$ Grafische Darstellung

Da es für m immer $0 < m \leq 1$ gilt,hilft der Einsatz von Taylor-Entwicklung für den Logarithmus bis Grad 2, um eine gute Genauigkeit zu erhalten.

$$ln(1+x) = \sum_{n=1}^{\infty} (-1)^{n+1} \frac{x^n}{n} mit(-1 < x \leq +1) \tag{27}$$

also (26) kann erweitert werden zu:

$$ln(x) \simeq e.ln(2) + (m-1) - \frac{(m-1)^2}{2} \tag{28}$$

Auf diesen Einsatz wird die Implementierung von $ln(x)$ basieren.

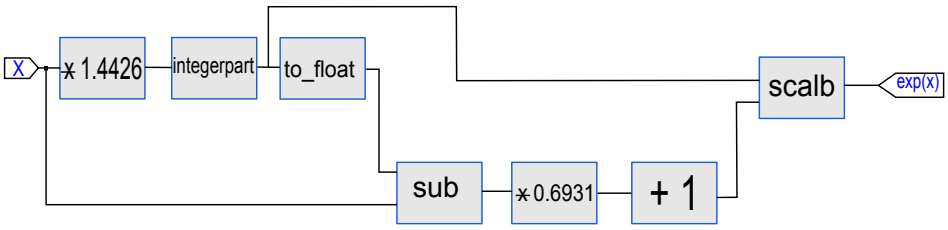


Abbildung 22: Floating-Point $exp(x)$ Modell

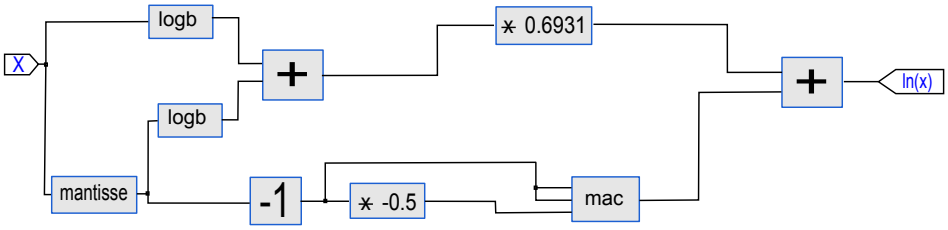


Abbildung 23: Floating-Point $ln(x)$ Modell

scalb: berechnet $y \cdot 2^k$, ***mac:*** berechnet $a * b + c$, ***logb:*** ermittelt das unbiased Exponent, $ln2 = 0.6931$, $log_2 e = 1.4426$

6.3.4 Berechnung von $\sqrt{x}$ Funktion

Die Quadratwurzel ist wegen der Komplexität des Algorithmus schwer auf FPGAs zu implementieren. Es wird hier eine Nicht-Wiederherstellung des Quadratwurzel-Algorithmus *(non-restoring square root algorithm)* und ein andere Algorithmus präsentiert, um eine Floating-Point Quadratwurzel zu implementieren.

Die Quadratwurzelalgorithmen sind in drei Methoden geteilt: Newton-Raphson Methode, Redundante SRT-Methode und Nicht-Redundante Methode. Die Newton-Raphson-Methode wurde in vielen Implementierungen etabliert. Zur Berechnung von $y = \sqrt{x}$ wird ein Approximationswert durch Iterationen berechnet. Zum Beispiel können wir die Newton-Raphson-Methode auf $f(T) = 1/T^2 - X$ zur Herleitung der Iterationsgleichung $T_{i+1} = T_i \times (3 - {T_i}^2 \times X)/2$ verwenden, wobei T_i ein Näherungswert von $1/\sqrt{X}$ ist. Nach n-Iterationen lässt sich ein Näherungswert der Quadratwurzel durch die Gleichung $Y = \sqrt{X} \simeq T_n \times X$ bestimmen. Der Algorithmus braucht einen Generator zur Erzeugung von T_0, eine ROM-Tabelle kann auch eingesetzt werden. In jeder Iteration sind eine Multiplikation und Addition oder Subtraktion erforderlich. Um die Multiplikation zu beschleunigen, braucht man einen schnelleren, parallelen Multiplikator.

Die klassische radix-2 SRT-Redundante Methode basiert auf der rekursiven Formel: $X_{i+1} = 2X_i - 2Y_{iy_{i+1}} - y_{i+1}^2 2^{-(i+1)}, Y_{i+1} = Y_i + y_{i+1}2^{-(i+1)}$, y_i und ist das i-ten Bit der Quadratwurzel und $y_i \in (-1, 0, 1)$. In jeder Iteration gibt es vier Operationen:

1. Shift nach links(Xi), um $2X_i$ zu produzieren.
2. Bestimmung von y_{i+1}.
3. Bildung von $F = -2Y_{iy_{i+1}} - y_{i+1}^2 2^{-(i+1)}$.
4. Addition von F und $2X_i$, um X_{i+1} zu produzieren.

Die Nicht-Redundante Methode ist ähnlich der SRT Methode, aber sie nutzt die zweier-komplementäre Darstellung für die Quadratwurzel. Die klassische Nicht-Redundante Methode basiert auf der Berechnungen von: $R_{i+1} = X - Y_i^2, Y_{i+1} = Y_i + y_{i+1}2^{-(i+1)}$, mit R_i als i-ten Teil-Rest, Y_i als i-ten Teil der Quadratwurzel mit $Y_1 = 0.1$, und y_i ist der i-te Bit in der Quadratwurzel. Die sich daraus ergebenden Werte werden ausgewählt, anhand der Vorzeichen vom Rest R_{i+1}, wenn $R_{i+1} \geq 0, y_{i+1} = 1$, sonst $y_{i+1} = -1$. Die Berechnungen können vereinfacht werden, indem man die Variablen ersetzt: $X_{i+1} = (X - Y_i^2)2^i$ durch $X_{i+1} = 2X_i - 2Y_{iy_i} + y_i^2 2^{-i}, Yi + 1 = Y_i + y_{i+1}2^{-(i+1)}$, mit X_i als i-ten Teil-Rest (X_1 ist Radikand). ***(Übersetzung aus [12])***.

Man kann die Quadratwurzel mit einer anderen Methode berechnen, indem man die beiden Funktionen $ln(x)$ und $exp(x)$ ausnutzt. Die Quadratwurzel eines Wertes A :$\sqrt{A}$ kann auch wie folgt geschrieben: $A^{1/2}$. Der natürliche Logarithmus von $\sqrt{A}$: $ln(\sqrt{A})$ ist gleich $lnA^{1/2} = 1/2ln(A)$. Jetzt kann die Exponential-Funktion eingesetzt werden, um das Endresultat zu ermitteln: $exp(1/2ln(A)) = \sqrt{A}$. Der Vorteil an diesem Einsatz ist die Einsparung an Ressourcen durch die Ausnutzung von den zuvor entwickelten Funktionseinheiten $ln(x)$ und $exp(x)$.

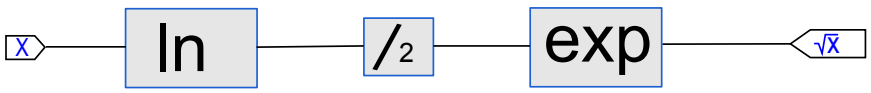


Abbildung 24: Floating-Point $\sqrt{x}$ Modell

6.4 Zufallsvariable

Das einfachste Modell in Sachen Zufallsvariable ist ein Schieberegister mit Rückkopplung auf NAND oder XOR-Gatter. Man unterscheidet zwischen linear rückgekoppelten (eng: linear feedback schift register LFSR) und nicht lineare rückgekoppelten Schieberegistern.

Man kann ein Polynom spezifizieren z.B. $P(x) = x^{18} + x^5 + x^2 + x^1 + x^0$, die + Operation kann durch XOR Operation ersetzt werden. Eine Zufallsvariable a_0 kann also folgend berechnet: $a_0 = a_{18} \oplus a_5 \oplus a_2 \oplus a_1$ (mit $\oplus$ als XOR Operator). Das Ganze kann durch ein Schieberegister und paar XOR-Gatter realisiert werden. Im Fall vom 4 Bit Zufallsgenerator braucht man ein Polynom von Grad 4 z.B: $x^4 + x^3 + 1$.

Im Fall einer Uniform-Verteilung wird folgende Rekursion benutzt:
$X_{n+1} = (aX_n + c) mod\ m$ mit $0 < m$, $0 \leq a < m$ und $0 \leq c < m$
Die Auswahl der vier Parametern, X_0, a, c und m, hat einen grossen Einfluss auf die Statistik und die Zyklen der erzeugten Zahlen. Park und Miller*[11]* haben die vier Parametern festgelegt, um gute Ergebnisse zu erzielen, das heißt lange Zyklen und möglichst viele Zahlen.

$$a = 7^5 = 16807,\ m = 2^{31} - 1 = 2147483647.$$

Um m zu approximieren, wird eine Faktorisierung durch:

$$\text{m} = \text{aq} + \text{r};\ \text{q} = [\text{m/a}];\ \text{r} = \text{m mod a}$$

und so bekommt man die endgültige Formel:

$$\text{az mod m} = \begin{cases} \text{a(z mod q)-r[z/q]}, falls > 0 \\ \text{a(z mod q)-r[z/q]+m}, sonst \end{cases} \tag{29}$$

Gauss-verteilte Zahlen können mittels der Polar Methode erzeugt werden *(sehe Algorithmus 1 im Kapitel 5.4)*.

7 Implementierung, Simulation und Synthese

In diesem Abschnitt werden die Implementierungen von verschiedenen Funktionen, ihre Synthesen und ihre Simulationen präsentiert sowie die Probleme, die damit verbunden sind, und die Lösungsmöglichkeiten.

VHDL ist die Implementierungssprache und die Entwicklungsumgebung ist Xilinx ISE 9.2i. Die Standardbibliothek ist IEEE, dazu wird eine zusätzliche Bibliothek definiert IEEE"_proposed. IEEE_Proposed enthält vier Pakete: float_pkg, fixed_pkg, math_utility_pkg und uniform_pkg.
Als erstes wird ein neues Projekt erstellt und die notwendigen Bibliotheken eingebunden. IEEE ist Standardbibliothek und wird automatisch gebunden, wenn man ein neues Projekt erstellt. Eigene Bibliotheken müssen extra erstellt und dann die dazugehörigen Pakete dort hinein verschoben werden.

Bibliothekenaufruf:

```
Library IEEE, IEEE_proposed;
USE  IEEE.STD_LOGIC_1164.ALL;
USE  IEEE.STD_LOGIC_UNSIGNED.ALL;
USE  IEEE_PROPOSED.math_utility_pkg.ALL;
USE  IEEE_PROPOSED.float_pkg.ALL;
USE  IEEE_PROPOSED.fixed_pkg.ALL;
USE  IEEE_PROPOSED.uniform_pkg.ALL;
```

Entity-Deklaration:

```
entity <entity_name> is
    port (<port_name>  :<mode>  <type> ; <andere ports>  ... );
end} <entity_name> ;
```

Architektur-Deklaration:

```
architecture < arch_name > of <entity_name> is
(signal Deklaration, Komponente Deklaration, etc.)
begin
 architecture Köper
end <arch_name>;
```

Ganz am Anfang sind die Grundoperationen Addition, Substraktion, Multiplikation und Division in Float_pkg zu testen und ihre Synthese zu analysieren. Um diese Operationen zu testen, wurden die entsprechenden Entitäten entworfen. Als Eingabe sind zwei Signale von Typ float16 definiert und als Ausgabe ebenfalls ein Signal vom Typ float16. float16 ist ein float-Datentyp der Länge 16 Bit, wobei der Exponent 6 Bit lang und die Mantisse

9 Bit lang ist. Der Datentyp float16 ist im Packet uniform_pkg definiert, das ist eigentlich ein Subtyp vom Datentyp float:

```
subtype float16 is float(6 downto -9).
```

7.1 Adder, Multiplizierer, Divider

Es wird hier mit Absicht die Substraktion weggelassen, da diese Operation als Addition betrachtet werden kann.

```
-------------     Adder       -------------
entity adder is port(
  A  :in  float(6 downto -9);
  B  :in  float(6 downto -9);
  S  :out float(6 downto -9) );
 end adder ;
architecture bhv of adder is begin
   S<=A+B;
end adder;
```

```
-------------   Multiplier     ------------
entity mult is port (
  A  :in  float(6 downto -9);
  B  :in  float(6 downto -9);
  R  :out float(6 downto -9) );
end mult ;
architecture bhv of mult is
begin
   R<=A*B;
end mult;
```

```
-------------    Divider     ------------
entity div is port (
  A  :in  float(6 downto -9);
  B  :in  float(6 downto -9);
  Q  :out float(6 downto -9) );
end div ;
architecture bhv of div is begin
   Q<=A/B;
end div;
```

Der gewöhnliche Divisionsoperator verursacht manchmal eine Fehlermeldung bei der Synthese. Als Alternative ist die Multiplikation mit dem Reziprok zu verwenden.

Betrachten wir eine Menge von Zahlen und prüfen wir die Funktionalität unserer Funktionseinheiten. Als Eingabe sind zwei Signale, A und B,

von Typ *float(6 downto -9)*. Die Ausgabe Q ist ebenfalls ein Signal von Typ *float(6 downto -9)*. Um die Werte leserlich darzustellen, werden diese Werte durch die Funktion *to_real*, in den real-Datentyp konvertiert, ra: real von A, rb: real von B und rq: real von Q.

Simulations Ergebnisse :

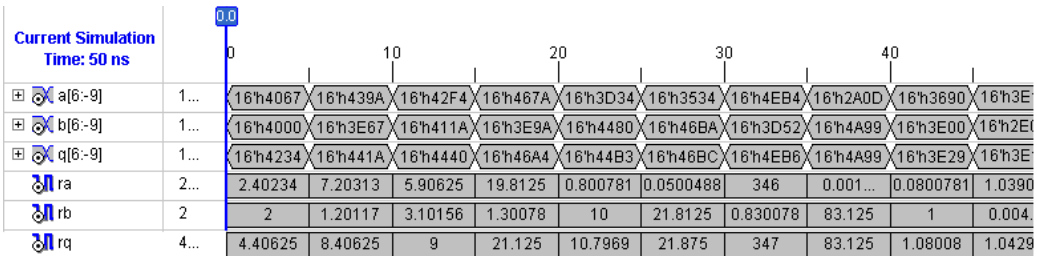


Abbildung 25: Simulationsergebnisse: Addition

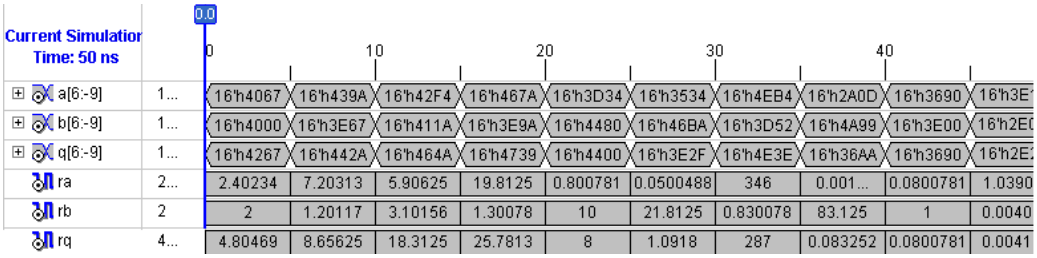


Abbildung 26: Simulationsergebnisse: Multiplikation

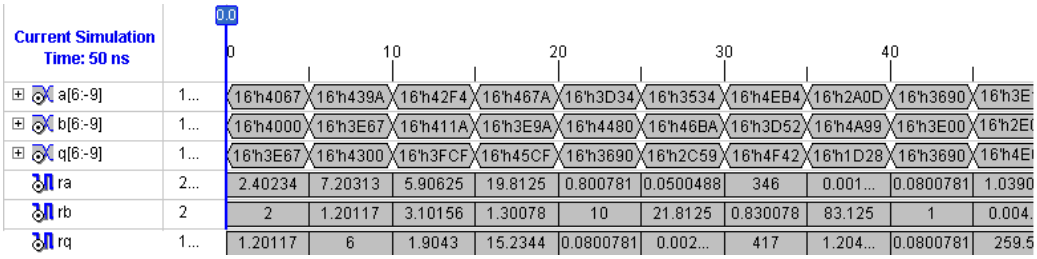


Abbildung 27: Simulationsergebnisse: Division

Synthese: Adder

```
# Adders/Subtractors                                : 12
 14-bit addsub                                     : 1
 5-bit subtractor                                  : 3
 6-bit adder                                       : 1
 6-bit subtractor                                  : 1
 7-bit adder                                       : 2
 7-bit subtractor                                  : 1
 8-bit adder                                       : 1
 9-bit adder                                       : 2
# Comparators                                       : 28
 14-bit comparator greater                         : 1
 7-bit comparator greater                          : 22
 7-bit comparator less                             : 2
 8-bit comparator greatequal                       : 1
 8-bit comparator greater                          : 1
 8-bit comparator lessequal                        : 1
# Multiplexers                                      : 2
 1-bit 14-to-1 multiplexer                         : 2
# Xors                                              : 1
 1-bit xor2                                        : 1
 ratio = 6
 Number of Slices:                      320  out of   6144     5%
 Number of 4 input LUTs:                568  out of  12288     4%
 Number of IOs:                          48
 Number of bonded IOBs:                  34  out of    240    14%
 Maximum combinational path delay: 26.272ns
 Total memory usage is 246212 kilobytes
```

Synthese: Multiplizierer

```
# Multipliers                                   : 1
 10x10-bit multiplier                          : 1
# Adders/Subtractors                            : 10
 6-bit adder                                   : 1
 6-bit subtractor                              : 4
 7-bit adder                                   : 2
 8-bit adder                                   : 1
 9-bit adder                                   : 2
# Comparators                                   : 3
 8-bit comparator greatequal                   : 1
 8-bit comparator greater                      : 1
 8-bit comparator lessequal                    : 1
# Xors                                          : 1
 1-bit xor2                                    : 1
 ratio = 3
 Number of Slices:                      195  out of   6144     3%
 Number of 4 input LUTs:                345  out of  12288     2%
 Number of IOs:                          48
 Number of bonded IOBs:                  34  out of    240    14%
 Number of DSP48s:                        1  out of     32     3%
 Maximum combinational path delay: 20.093ns
 Total memory usage is 239940 kilobytes
```

Synthese: Divider

```
# Adders/Subtractors                                  : 27
 15-bit subtractor                                   : 14
 5-bit adder                                         : 1
 5-bit subtractor                                    : 3
 6-bit adder                                         : 1
 6-bit subtractor                                    : 1
 7-bit adder                                         : 2
 7-bit subtractor                                    : 2
 8-bit adder                                         : 1
 9-bit adder                                         : 2
# Comparators                                         : 3
 8-bit comparator greatequal                         : 1
 8-bit comparator greater                            : 1
 8-bit comparator lessequal                          : 1
# Xors                                                : 1
 1-bit xor2                                          : 1
 ratio = 6
 Number of Slices:                      345  out of   6144     5%
 Number of 4 input LUTs:                614  out of  12288     4%
 Number of IOs:                          48
 Number of bonded IOBs:                  34  out of    240    14%
 Maximum combinational path delay: 50.744ns
 Total memory usage is 245512 kilobytes
```

7.2 $exp(x)$, $ln(x)$ und $\sqrt{x}$

Diese Funktionen befinden sich in uniform_pkg. Es gibt drei implementierte Funktionen für die Funktion $exp(x)$:

exp_f welche auf LUT basiert und für Zahlen von 0 bis 1.25 mit guter Genauigkeit gilt,

$exp\_16(x)$ für 16 Bit Zahlenlänge,

$exp\_32(x)$ für 32 Bit Zahlenlänge.

exp_16 und *exp_32* sind auf der Formel(25) basierend implementiert worden.

Die Simulation hat gezeigt, dass *exp_16* auf Eingabewerten kleiner oder gleich 23 beschränkte ist. Bei Eingaben grösser als 23 sind wegen Integer overflow die Ausgabewerte verfälscht, wenn die Funktion das $2^{integerpart}$ berechnet.

Die Funktionen sind im Packet uniform_pkg deklariert und implementiert. Der Aufruf einer Funktion von einem bestimmten Packet erfolgt, indem man die entsprechende Bibliothek und Pakete in die Beschreibung einbindet. Die Funktion kann dann mit ihren Namen aufgerufen werden. Die Logarithmus-Funktion ist unter dem Namen ln_16 deklariert, die Quadratwurzel unter sqrt_f und die Exponential-Funktion, wie schon oben bereits erwähnt, unter exp_f, exp_16 und exp_32.

Wichtig: Wenn man sich für ein Format entschieden hat, muss man die Exponent- und Mantissenlänge in float_pkg ändern.

```
--constant float_exponent_width : NATURAL    := 6;
--constant float_fraction_width : NATURAL    := 9;
float_exponent_width : NATURAL    := 8; --constant
float_fraction_width : NATURAL    := 23;--constant

function exp_16 (x:float(6 downto -9)) return float16;
function exp_32 (x:float(6 downto -9)) return float32;
function ln_16  (x:float(6 downto -9)) return float16;
function sqrt_f (x:float(6 downto -9)) return float16;
```

Simulation: exp_16

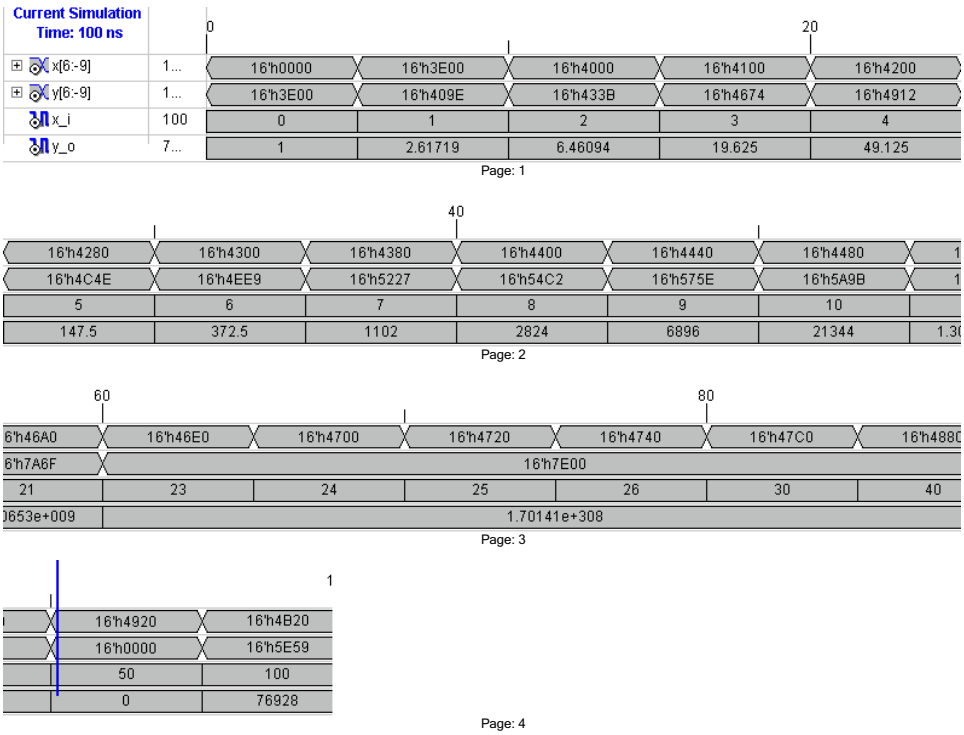


Abbildung 28: Simulationsergebnisse: Exponential-Funktion exp_16

Ab dem Zeitpunkt t=60, d.h. ab dem Wert 23, bekommt man ein Integer overflow.

Simulation: *exp_32*

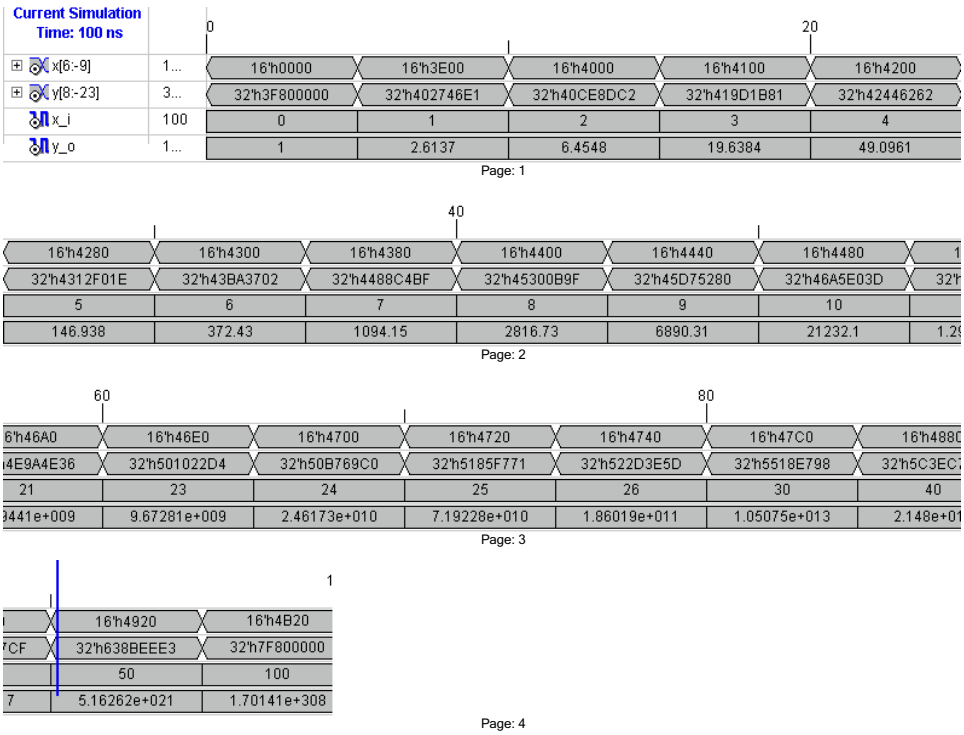


Abbildung 29: Simulationsergebnisse: Exponential-Funktion exp_32

Simulation: ln_16 ***und*** $\sqrt{x}$

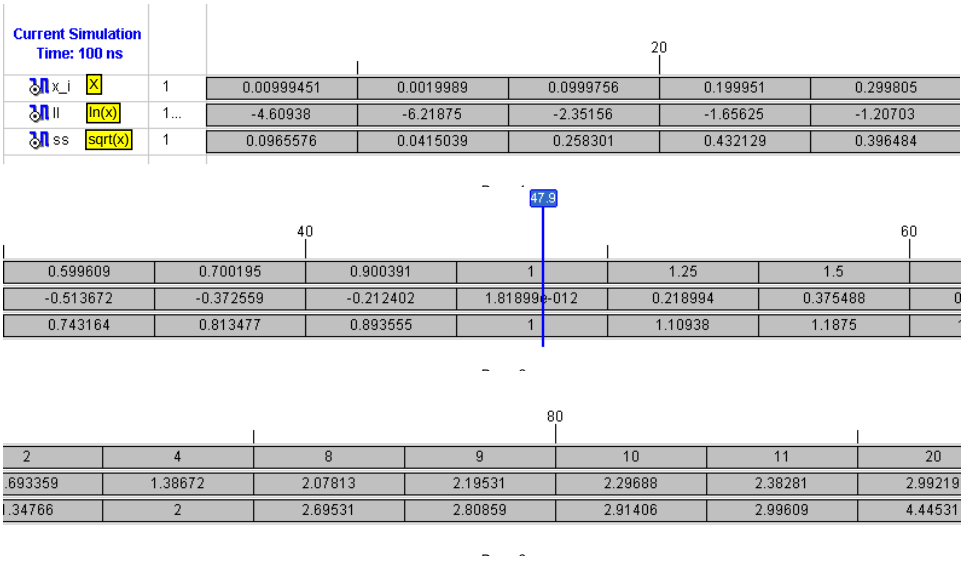


Abbildung 30: Simulationsergebnisse: Logarithmus- und Quadratwurzel-Funktion ln_16, sqrt_f

Synthese :*exp_16*

```
# ROMs                                                : 1
 16x14-bit ROM                                       : 1
# Multipliers                                         : 3
 10x10-bit multiplier                                : 2
 24x11-bit multiplier                                : 1
# Adders/Subtractors                                  : 58
 14-bit addsub                                       : 2
 16-bit adder                                        : 2
 5-bit subtractor                                    : 7
 6-bit adder                                         : 5
 6-bit subtractor                                    : 11
 7-bit adder                                         : 12
 7-bit subtractor                                    : 3
 8-bit adder                                         : 7
 9-bit adder                                         : 9
# Comparators                                         : 66
 14-bit comparator greater                           : 1
 14-bit comparator less                              : 1
 7-bit comparator greatequal                         : 1
 7-bit comparator greater                            : 35
 7-bit comparator less                               : 5
 8-bit comparator greatequal                         : 7
 8-bit comparator greater                            : 8
 8-bit comparator lessequal                          : 8
# Multiplexers                                        : 3
 1-bit 14-to-1 multiplexer                           : 3
# Logic shifters                                      : 1
 10-bit shifter logical right                        : 1
# Xors                                                : 1
 1-bit xor2                                          : 1
ratio = 22
 Number of Slices:                     1054  out of   6144    17%
 Number of 4 input LUTs:               1887  out of  12288    15%
 Number of IOs:                          32
 Number of bonded IOBs:                  25  out of    240    10%
 Number of DSP48s:                        1  out of     32     3%
 Maximum combinational path delay: 105.062ns
 Total memory usage is 291464 kilobytes
```

Synthese :*exp_32*

```
# Multipliers                                      : 3
 24x24-bit multiplier                              : 2
 33x25-bit multiplier                              : 1
# Adders/Subtractors                               : 60
 10-bit adder                                      : 8
 23-bit adder                                      : 9
 28-bit addsub                                     : 2
 31-bit adder                                      : 1
 6-bit subtractor                                  : 8
 7-bit subtractor                                  : 6
 8-bit adder                                       : 6
 8-bit subtractor                                  : 5
 9-bit adder                                       : 12
 9-bit subtractor                                  : 3
# Comparators                                      : 114
 10-bit comparator greatequal                      : 7
 10-bit comparator greater                         : 9
 10-bit comparator lessequal                       : 9
 28-bit comparator greater                         : 1
 28-bit comparator less                            : 1
 9-bit comparator greatequal                       : 1
 9-bit comparator greater                          : 79
 9-bit comparator less                             : 7
# Multiplexers                                     : 3
 1-bit 28-to-1 multiplexer                         : 3
# Logic shifters                                   : 2
 24-bit shifter logical right                      : 2
# Xors                                             : 1
 1-bit xor2                                        : 1
 ratio is 84
 Number of Slices:                     4293  out of   6144    69%
 Number of 4 input LUTs:               7798  out of  12288    63%
 Number of IOs:                          48
 Number of bonded IOBs:                  48  out of    240    20%
 Number of DSP48s:                       11  out of     32    34%
 Maximum combinational path delay: 158.275ns
Total memory usage is 360584 kilobytes
```

Synthese : $ln(x)$

```
# Multipliers                                      : 2
 10x10-bit multiplier                             : 2
# Adders/Subtractors                               : 85
 14-bit addsub                                    : 3
 15-bit subtractor                                : 14
 16-bit adder                                     : 2
 5-bit subtractor                                 : 14
 6-bit adder                                      : 8
 6-bit subtractor                                 : 9
 7-bit adder                                      : 12
 7-bit subtractor                                 : 4
 8-bit adder                                      : 8
 9-bit adder                                      : 11
# Comparators                                      : 77
 14-bit comparator greater                        : 2
 14-bit comparator less                           : 1
 7-bit comparator greater                         : 38
 7-bit comparator less                            : 10
 8-bit comparator greatequal                      : 8
 8-bit comparator greater                         : 9
 8-bit comparator lessequal                       : 9
# Multiplexers                                     : 4
 1-bit 13-to-1 multiplexer                        : 1
 1-bit 14-to-1 multiplexer                        : 3
# Logic shifters                                   : 1
 13-bit shifter logical right                     : 1
# Xors                                             : 2
 1-bit xor2                                       : 2
ratio is 23
 Number of Slices:                     1186  out of   6144    19%
 Number of 4 input LUTs:               2095  out of  12288    17%
 Number of IOs:                          32
 Number of bonded IOBs:                  25  out of    240    10%
 Number of DSP48s:                        1  out of     32     3%
 Maximum combinational path delay: 66.988ns
 Total memory usage is 282312 kilobytes
```

Synthese :$\sqrt{x}$

```
# Multipliers                                      : 5
 10x10-bit multiplier                             : 4
 24x11-bit multiplier                             : 1
# Adders/Subtractors                               : 166
 14-bit addsub                                    : 4
 14-bit subtractor                                : 1
 15-bit subtractor                                : 28
 16-bit adder                                     : 4
 5-bit subtractor                                 : 24
 6-bit adder                                      : 14
 6-bit subtractor                                 : 21
 7-bit adder                                      : 25
 7-bit subtractor                                 : 8
 8-bit adder                                      : 15
 9-bit adder                                      : 22
# Comparators                                      : 145
 14-bit comparator greater                        : 3
 14-bit comparator less                           : 2
 7-bit comparator greatequal                      : 1
 7-bit comparator greater                         : 74
 7-bit comparator less                            : 15
 8-bit comparator greatequal                      : 17
 8-bit comparator greater                         : 16
 8-bit comparator lessequal                       : 17
# Multiplexers                                     : 7
 1-bit 13-to-1 multiplexer                        : 1
 1-bit 14-to-1 multiplexer                        : 6
# Logic shifters                                   : 2
 10-bit shifter logical right                     : 1
 13-bit shifter logical right                     : 1
# Xors                                             : 3
 1-bit xor2                                       : 3
ratio is 72
 Number of Slices:                     3392  out of   6144    55%
 Number of 4 input LUTs:               5996  out of  12288    48%
 Number of IOs:                          32
 Number of bonded IOBs:                  25  out of    240    10%
 Number of DSP48s:                        2  out of     32     6%
Maximum combinational path delay: 191.940ns
Total memory usage is 345736 kilobytes
```

7.3 Zufallsvariable

Der Generator für uniform-verteilte Zufallsvariablen steht in diesem Abschnitt im Mittelpunkt, da die anderen Verteilungen auf der Uniformverteilung basieren.

Zur Erzeugung uniformverteilter Zufallszahlen wurden zwei Prozeduren in uniform_pkg implementiert. Die erste Prozedur heisst UNIFORM_F_16 (mit 16 Bit Präzision) und die zweite heisst UNIFORM_F_32 (32 Bit Präzision).

```
procedure UNIFORM_F_16(variable Seed:inout float16;
                       variable    X:outfloat16);
procedure UNIFORM_F_32 (variable Seed:inout float32;
                       variable    X:out float32);
```

Die Variable ***Seed*** muss einmal initialisiert werden.

```
variable seed: float16 :="0100011110000000";
```

Der Aufruf der Prozedur erfolgt über ihren Namen z.B. ***UNIFORM_F_16***

```
UNIFORM_F_16(seed,rand);
```

Die erzeugte Zufallszahl ist nun ***rand***.

```
architecture Behavioral of random is begin process(clk)
 variable s:float16 :="0100011110000000"; --  28
 variable rand:= float16_1;
begin
    UNIFORM16(s,rand);
    y<=rand;
end process; end Behavioral;
```

Die erzeugten Zahlen von den beiden Prozeduren unterscheiden sich in ihrer Periodizität sowie in ihrer Menge. Mit einem seed-Startwert von 759 und einem Takt von 5 ns hat sich bei der UNIFORM_F_16 Simulation ergeben, dass auf anfangs 126 Werte über eine Zeitspanne von 630 ns dann eine Reihe von Zahlen (101 Werte) mit einer Periode von 505 ns folgt. Diese Folge tritt immer wieder in derselben Reihenfolge auf.

Mit denselben Einstellungen *(selber Takt und selber Seed)* wurde bei UNIFORM_F_32 über 10000 ns keine Periodizität festgestellt.

Gaussverteilte-Zahlen können entweder mit der Box-Müller-Methode oder mit LFSR-Registern erzeugt werden. Das Ergebnis der Simulation in Multisim der 4 Bit Schaltung in der Abbildung 36 ist eine Folge von Zahlen: (0,8,12,14,7,3,9,4,2,1,8,4,10,5,10,13,14,15,7,6,13,6,6,5,2,9,12,6,3,1,0), die sich laufend wiederholt. Die erzeugten Zahlen unterscheiden sich, wenn man das

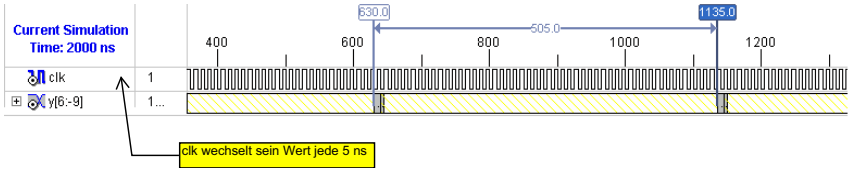


Abbildung 31: UNIFORM_F_16 mit Seed Startwert = 759

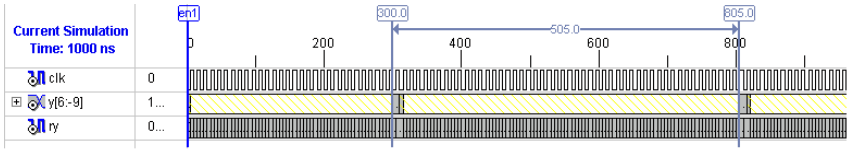


Abbildung 32: UNIFORM_F_16 mit Seed Startwert =28

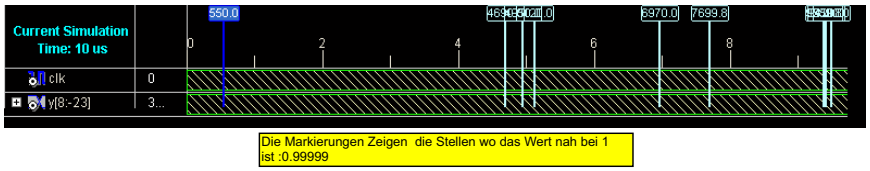


Abbildung 33: UNIFORM_F_16 mit Seed Startwert =28

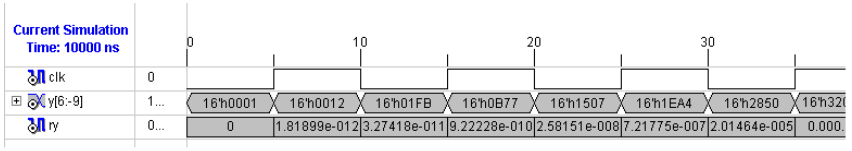


Abbildung 34: Beispiel-Werte: UNIFORM_F_16 mit Seed Startwert =28

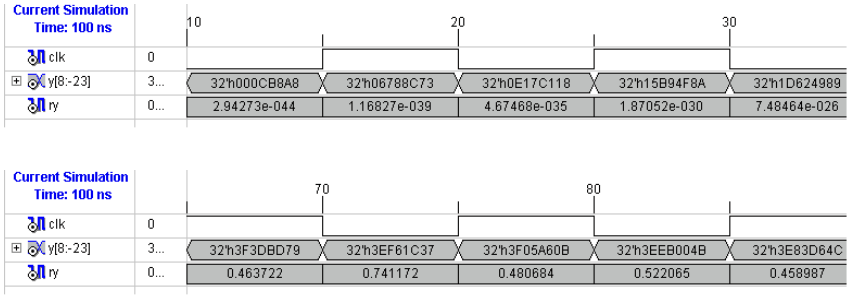


Abbildung 35: Beispiel-Werte: UNIFORM_F_32 mit Seed Startwert =28

Polynom ändert. In VHDL wurde noch ein weiteres Beispiel mit einem 9 Bit Register entworfen. Das resultierende Ergebnis kann als Mantisse in einen float Datentyp übertragen werden.

Sei *rand* die erzeugte Zufallszahl von Typ std_logic_vector und *rand_f* die erwünschte float Zufallszahl. Ein float Datentyp ist nichts weiter als ein Subtyp vom Datentyp std_logic_vector. Die erzeugte Zufallszahl *rand* ist 9 Bit lang. Wenn man eine 6 Bit langen std_logic_vector mit *rand* verkettet, bekommt man eine neue Variable der Länge 16 Bit, die immer noch vom Typ std_logic_vector ist. Die Umwandlung in float16 wäre jetzt nicht mehr so schwer, denn es ist schon ein float Datentyp. Man muss nur den Exponenten und die Mantisse zuweisen. Wenn man den Exponenten 0011111 wählt, dann sind die erzeugten Zahlen immer in dem Bereich (1,2). Wenn jetzt 1 daraus substrahiert wird, liegen die Ergebnisse nun im Bereich (0,1). Man kann auf die Substraktion verzichten, indem man anstatt 0011111, 0100000 wählt, aber die erzeugten Zahlen liegen im Bereich (0.5, 1).

$0 \leq rand$ d.h $1 \leq 0011111 \& rand < 2$, weil 0011111 der Multiplikation bei 2^0 und der expliziten Addition von 1 entspricht.
und somit $0 \leq 0011111 \& rand - 1 < 1$ ist.

$0.5 \leq 0100000 \& rand < 1$, weil 0100000 der Multiplikation bei 2^{-1} und der expliziten Addition von 1 entspricht.

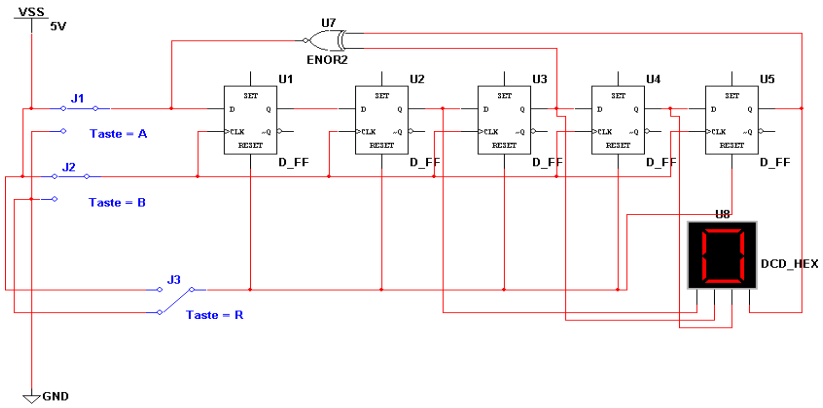


Abbildung 36: Linear rückgekoppeltes Schieberegisterschema

Synthese: UNIFORM_F_16

```
# Multipliers                                          : 3
 10x10-bit multiplier                                  : 3
# Adders/Subtractors                                   : 91
 14-bit addsub                                         : 3
 15-bit subtractor                                     : 14
 5-bit subtractor                                      : 12
 6-bit adder                                           : 7
 6-bit subtractor                                      : 16
 7-bit adder                                           : 15
 7-bit subtractor                                      : 3
 8-bit adder                                           : 7
 9-bit adder                                           : 14
# Comparators                                          : 87
 14-bit comparator greater                             : 2
 14-bit comparator less                                : 1
 6-bit comparator greater                              : 1
 7-bit comparator greater                              : 49
 7-bit comparator less                                 : 12
 8-bit comparator greatequal                           : 7
 8-bit comparator greater                              : 7
 8-bit comparator lessequal                            : 7
 9-bit comparator greater                              : 1
# Multiplexers                                         : 3
 1-bit 14-to-1 multiplexer                             : 3
  ratio is 33
 Number of Slices:                     1721  out of   6144    28%
 Number of 4 input LUTs:               3087  out of  12288    25%
 Number of IOs:                          17
 Number of bonded IOBs:                  16  out of    240     6%
Maximum combinational path delay: No path found
 Total memory usage is 292740 kilobytes
```

Synthese: UNIFORM_F_32

```
    Macro Statistics
# Multipliers                                          : 3
 24x24-bit multiplier                                  : 3
# Adders/Subtractors                                   : 105
 10-bit adder                                          : 7
 23-bit adder                                          : 14
 28-bit addsub                                         : 3
 29-bit subtractor                                     : 28
 6-bit subtractor                                      : 12
 7-bit subtractor                                      : 9
 8-bit adder                                           : 7
 8-bit subtractor                                      : 7
 9-bit adder                                           : 15
 9-bit subtractor                                      : 3
# Comparators                                          : 151
 10-bit comparator greatequal                          : 7
 10-bit comparator greater                             : 7
 10-bit comparator lessequal                           : 7
 23-bit comparator greater                             : 1
 28-bit comparator greater                             : 2
 28-bit comparator less                                : 1
 8-bit comparator greater                              : 1
 9-bit comparator greater                              : 113
 9-bit comparator less                                 : 12
# Multiplexers                                         : 3
 1-bit 28-to-1 multiplexer                             : 3
 ratio is 103 Optimizing  ratio 100 (+ 5) of 6144 slices : Area final
 ratio is 98.
 Number of Slices:                     5868  out of   6144    95%
 Number of 4 input LUTs:              10883  out of  12288    88%
 Number of IOs:                          33
 Number of bonded IOBs:                  32  out of    240    13%
 Number of DSP48s:                       12  out of     32    37%
 Maximum combinational path delay: No path found Total memory usage
 is 361860 kilobytes
```

Synthese: LFSR

```
# Latches                                          : 2
 10-bit latch                                      : 2
# Xors                                             : 1
 1-bit xor2                                        : 1
 ratio is 0
 Number of Slices:                       12  out of   6144     0%
 Number of Slice Flip Flops:             18  out of  12288     0%
 Number of 4 input LUTs:                 20  out of  12288     0%
 Number of IOs:                          12
 Number of bonded IOBs:                  12  out of    240     5%
    IOB Flip Flops:                       1
 Number of GCLKs:                         1  out of     32     3%
 Maximum combinational path delay: No path found
 Total memory usage is 220420 kilobytes
```

8 Ergebnisse

In diesem Abschnitt werden die gesamten Ergebnisse dieser Arbeit zusammengefasst. Das Ziel dieser Arbeit ist die Untersuchung der Möglichkeit den Algorithmus von [1] in einem FPGA zu implementieren. Die in dem Algorithmus geführten Funktionen sind sehr komplex. Die Komplexität dieser Funktionen erschwert die Integration in FPGA´s, insbesondere wenn man eine höhere Genauigkeit erzielen will.

Die Implementierung von der Exponential-Funktion verbraucht mehr als 80% der gesamten Ressourcen eines Virtex 4 Chip von Xilinx im Fall einer Genauigkeit von 32 Bit und 22% für eine 16 Bit Genauigkeit. Wenn man noch dazu den Verbrauch von der Logarithmusfunktion einbezieht, was eigentlich über 60% beträgt, so sind dann kaum noch Ressourcen vorhanden, um den Rest zu implementieren. Im Algorithmus sind noch viele Wertvergleiche notwendig, was viele Register verbraucht. Eine Entity, die die drei Funktionen: Exponential (exp_32), Logarithmus(ln_16) und Quadratwurzel(sqrt_f) berechnet, hat 99% des gesamten Chip-Ressourcen verbraucht. Ein Testen von der in [5] entworfene $exp(x)$ Funktion wurde durchgeführt. es wurde festgestellt, dass dieses Modell 98% des gesamten Chip verbraucht.

Alternativ für den Hardware-Entwurf des Synaptisches Rauschens wäre eine Software-Lösung oder der Einsatz von Digital-Signal-Prozessoren(DSP) und eines Mikroprogramm möglich. DSP´s haben eine optimierte Architektur und Funktionseinheiten für Signal- und Daten-Verarbeitung. Die DSP´s können zusätzlich Multiplikationen schnell durchführen. Durch einen Mikroprogramm kann man die Sprünge im Algorithmus realisieren. Die Operationen können sequentiell oder parallel verarbeitet werden. Dank integrierter Speicher können Zwischenergebnisse oder Vorberechnungen gespeichert werden.

Taschenrechner sind in der Lage mit geringen Ressourcen komplexe Funktionen wie cos, sin, $exp(x)$, $ln(x)$ und $\sqrt{x}$ zu berechnen. Es wäre interessant zu wissen, welche numerischen Methoden und Algorithmen von den Firmen verwendetet werden, denn diese wären von großen Nutzen für die Weiterentwicklung in diesem Gebiet. Jedoch sind diese Methoden häufig Firmengeheimnisse, die nicht gern herausgegeben werden.

9 Zusammenfassung

Angefangen mit der Motivation und der Einleitung wurden biologische Grundlagen präsentiert, um den Hintergrund zu verstehen. Dann wurden mathematische Grundlagen erläutert: Wahrscheinlichkeit, Verteilungsfunktionen, Zufallsvariablen und Zahlen-Darstellung im Floating-Point Format. Als nächstes wurde einen Modell zur Erzeugung synaptischen Rauschens vorgestellt und daraus wurden die wichtigen Funktionen hergeleitet, die man für einen Hardware-Entwurf benötigt: Exponential, Logarithmus und Quadratwurzel. Es wurden paar Funktionen von Floating-Point Paketen vorgestellt, die man braucht, um das Modell in VHDL zu beschreiben. Die relevanten, vordefinierten Funktionen in float_pkg wurden getestet, um ihre Synthesefähigkeit zu prüfen. Dabei wurde festgestellt, dass die Division manchmal Fehler verursacht und damit bricht die Synthese zusammen. Die Funktion *to_integer* funktioniert nicht, man muss zuerst die Zahl in Standard Logic umwandeln, dann in Integer.

Es wurden Modelle zur Berechnung von Exponential, Logarithmus und Quadratwurzel vorgestellt und implementiert. Die Entwürfe sind simuliert und synthetisiert. Die Ergebnisse der Simulation und der Synthese wurden vorgestellt. Zum Schluss wurden die Ergebnisse analysiert. Der Quellcode in VHDL ist im Anhang zu finden.

10 Anhang

```
library IEEE,ieee_proposed;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
use ieee.std_logic_arith.all;
use IEEE.Math_real.all;
use IEEE.math_complex.all;
use ieee_proposed.fixed_pkg.all;
use ieee_proposed.float_pkg16.all;
--use ieee_proposed.float_pkg.all;
use ieee_proposed.math_utility_pkg.all;

package uniform_pkg is
------------------------------------------------------------
-----  Subtype myfloat => float Zahlen mit 32 Bit Länge --
-----  Subtype float16 => float Zahlen mit 16 Bit Länge --
------------------------------------------------------------
subtype myfloat is float(8 downto -23);
subtype float16 is float (6 downto -9);
type tabel is array (0 to 7) of float(6 downto -9);
type faktor is array (0 to 9) of integer;
type LUT is array(0 to 16) of float(6 downto -9);


--------------------------------------------------------------------
-----  Konstanten Deklaration für die 32 Bit Zahlen   ------------
--------------------------------------------------------------------

constant float_0:myfloat:="00000000000000000000000000000000";
--constant float_1:myfloat:="00111111100000000000000000000000";
constant float_1:float(8 downto -23):="00111111100000000000000000000000";
constant float_E:myfloat:="01000000001011011111100001010100";
constant float_hig: myfloat:="01111111100000000000000000000000";
constant float_low: myfloat:="11111111100000000000000000000000";
constant float_PI: myfloat:= "01000000010010010000111111011011";
constant float_LOG2:myfloat:="00111111001100010111001000011000";
constant float_log10:myfloat:="01000000000100110101110110001110";

constant float_log10_E:myfloat:="00111110110111100101101111011001";
constant float_EPS:myfloat:="00110101100001100001011110111101";--0.000001
constant log2_e:float(8 downto -23):=to_float(real(1.4426950),8,23);
```

```
constant ln2:float(8 downto -23):=to_float(real(0.69315),8,23);--ln(2)

------------------------------------------------------------------
----- konstanten für uniform Diviate bei 32 Bit Zahlenlänge  -----
------------------------------------------------------------------
constant IM :myfloat :="01001111000000000000000000000000";
constant IM1:myfloat:="01001110111111111111111111111111";--2147483563
constant AM:myfloat:="00110000000000000000000000000001";--0,465661357E-10
constant IA1:myfloat :="01000111000111000100111000000000";--40014
constant IQ1:myfloat :="01000111010100011010010000000000";--53668
constant IR1:myfloat :="01000110001111101100110000000000";--12211
constant val_Max:myfloat:="00111111011111111111111101011000";--0.999990
-- val_Max:maximal annehmbarer Wert
------------------------------------------------------------------
----- Konstanten für uniform Diviate bei 16 Bit Zahlenlänge  -----
------------------------------------------------------------------

constant AM16:float(6 downto -9) :="0010000000000001";--3,0585E-05
constant IA16:float(6 downto -9) :="0100011110000000";--28
constant IQ16:float(6 downto -9) :="0101001001001001";--1170
constant IR16:float(6 downto -9) :="0100011100100000";--25
constant IM16:float(6 downto -9) :="0101110000000000";--32767
constant float16_0:float(6 downto -9):="0000000000000000";-- 0
constant float16_1:float(6 downto -9):="0011111000000000";--1
constant max_val16:float(6 downto -9):="0011110110011010";--0.9
constant log2_e_16:float(6 downto -9) :=to_float(real(1.44269),6,9);
constant ln2_16   :float(6 downto -9) :=
to_float(real(0.69315),6,9);--ln(2)

------------------------------------------------------------------
-- Koeffizienten für den Schritt F im Algorithmus             --
------------------------------------------------------------------
constant Table_F : tabel :=
 (
0 => to_float(-0.5),
1 => to_float(0.3333333),
2 =>to_float(0.2500068),
3 => to_float(0.2000118),
4 =>to_float(-0.1661269),
5 => to_float(0.1421878),
6 =>to_float(-0.1384794),
7 => to_float(0.1250060) );
-------------------------------------------
-- Faktoriel Tabelle : x! fuer x= 0 bis 9  --
```

```
-------------------------------------------
constant k_factor :faktor:=(
0 => 1,
1 => 1,
2 => 2,
3 => 6,
4 => 24,
5 => 120,
6 => 720,
7 => 5040,
8 => 40320,
9 => 36288 );

-------------------------------------------------
---    ln und exp Funktionen                 ------
-----    nach Computer Arithmetic Algorithms   -----
------     von Israel Koren                        ---
-------------------------------------------------------

-------------------------------------------
--   lut 1+2^-i für ln und exp Funktionen  --
-------------------------------------------
constant a_LUT : LUT :=
 (
0 => to_float(2.0),
1 => to_float(1.5),
2 => to_float(1.25),
3 => to_float(1.125),
4 => to_float(1.0625),
5 => to_float(1.03125),
6 => to_float(1.015625),
7 => to_float(1.0078125),
8 => to_float(1.00390625),
9 => to_float(1.001953125),
10=> to_float(1.000976563),
11=> to_float(1.000488281),
12=> to_float(1.000244141),
13=> to_float(1.00012207),
14=> to_float(1.000061035),
15=> to_float(1.000030518),
16=> to_float(1.000015259) );
-------------------------------------
---  lut ln(1+2^-i) für ln und exp --
-------------------------------------
```

```
constant b_LUT : LUT :=
 (
0 => to_float(0.69314718),
1 => to_float(0.405465108),
2 => to_float(0.223143551),
3 => to_float(0.117783035),
4 => to_float(0.060624621),
5 => to_float(0.030771658),
6 => to_float(0.015504186),
7 => to_float(0.00778214),
8 => to_float(0.00389864),
9 => to_float(0.00195122),
10=> to_float(0.000976086),
11=> to_float(0.000488161),
12=> to_float(0.000244111),
13=> to_float(0.000122062),
14=> to_float(0.000061033),
15=> to_float(0.000030517),
16=> to_float(0.000015258) );
-----------------------------------------------------------------
-- Konstanten zur Berechnung von Koeffizienten in Schritt (P)  --
-----------------------------------------------------------------
constant iw:float(6 downto -9) :="0011101100110001";--(0.3989423);
-- sqrt(2pi)^-1
constant ib1:float(6 downto -9):="0011010011111101";--(0.046667);
-- 1/24
constant ib2:float(6 downto -9):=to_float(real(0.3),6,9);-- 0.3;
-- 3/10
constant ic:float(6 downto -9) :="0011011101101100";--(0.1069);
constant ic3:float(6 downto -9):="0011100001001001";--(0.1428571);
--1/7

-----------------------------------------------------------------
-----               Prozeduren und Funktionen                -----
-----------------------------------------------------------------

  procedure UNIFORM_F_16 (variable Seed:inout float(6 downto -9);
                          variable X:out float(6 downto -9));
  procedure UNIFORM_F_M (variable Seed:inout myfloat;
                         variable X:out myfloat);
  procedure GAUSSIEN16  (variable seed:inout float(6 downto -9);
                        variable G: inout float(6 downto -9));
  function log_R(X : myfloat ) return integer;
  function log_R16(X:float(6 downto -9) ) return INTEGER;
```

```
  --function exp_f(x: float(6 downto -9)) return float16;
  function exp_16(x:float(6 downto -9)) return float16;
  function exp_32(x:float(6 downto -9)) return myfloat;
  function ln_16(x:float(6 downto -9)) return float16;
  function sqrt_f(x:float(6 downto -9)) return float16;
  function exp_f(x: myfloat) return myfloat;
  -- computer arithmetic algorithms
  function ln_f (x: myfloat) return myfloat;
  -- computer arithmetic algorithms

end uniform_pkg;

------------------------------------------------------------
-----           package body                           -----
------------------------------------------------------------

Package body uniform_pkg is

------------------------------------------------------------------
----- Procedure Uniform_F_M gibt eine Zufallszahl mittels     --
----- Uniformverteilung  zurück. Beim ersten Aufruf der       --
----- Prozedur muss "seed" initialisiert werden, danach wird  --
----- seed automatisch ein neuer Wert zugewiesen und damit    --
----- eine neuen Zufallszahl erzeugt.                         --
------------------------------------------------------------------

procedure UNIFORM_F_M (variable Seed:inout myfloat;
variable X:out myfloat)is

variable f:myfloat:=float_0;
variable temp:myfloat:=float_0;
begin
f:=seed/IQ1; seed:=IA1 * (seed mod IQ1)- IR1*abs(f);
if seed<float_0 then
  seed:=IA1 * (seed mod IQ1)- IR1*abs(f)+ IM;
end if;
temp:=seed*AM;
if temp>float_1 then temp:=val_max;
end if;
X:=temp;
     end UNIFORM_F_M;
------------------------------------------------------------------
-- Procedure Uniform_F16 gibt eine Zufallszahl der Länge 16 Bit--
------------------------------------------------------------------
```

```
procedure UNIFORM_F_16 (variable Seed:inout float(6 downto -9);
                variable X:out float(6 downto -9))is

variable f:float(6 downto -9):=float16_0;
variable temp:float(6 downto -9):=float16_0;

begin
f:=seed/IQ16; seed:=IA16 * (seed mod IQ16)- IR16*abs(f);
if seed<float16_0 then
        seed:=IA16 * (seed mod IQ16)- IR16*abs(f)+ IM16;
end if;
temp:=seed*AM16;
if temp>float16_1 then temp:=max_val16; end if;
X:=temp;
     end UNIFORM_F_16;
-----------------------------------------------------------------
-- Funktion gibt den unbased Exponenten einer Floating Zahl als --
-- integer aus                                                  --
-----------------------------------------------------------------

function log_R(X : myfloat ) return integer is variable
temp:integer:=0;
begin temp :=logb(X);
return temp; end log_R;
-------------------------------------------------------
--  im Fall dass X in 16 bit Darstellung ist        ---
-------------------------------------------------------
function log_R16(X : float(6 downto -9) ) return integer is
variable temp:integer:=0;
begin
temp :=logb(X);
return temp;
end log_R16;

------------------
-- lead_zero -----
------------------


----------------------------------------
procedure GAUSSIEN16 (variable seed:inout float(6 downto -9);
variable G: inout float(6 downto -9)) is

variable U1,U2:float(6 downto -9):=float16_0;
```

```
variable iset:boolean;
variable v1,v2,x1,x2,gset:float(6 downto -9):=float16_0;
variable rsq,tmp,fac:float(6 downto -9):=float16_1;
--variable tmp:integer;
    begin
    if (iset) then
            iset := False;
            G:=gset;
            return;
            end if;
while (True) loop
            uniform_f16(seed,U1);
                    v1:=2*U1-float16_1;
            uniform_f16(seed,U2);
                    v2:=2*U2-float16_1;
            rsq := v1 * v1 + v2* v2;
            if ( rsq <= float16_1 ) then
                tmp :=rsq;
                fac := sqrt(-2 *(tmp / rsq));
                x1 := v1 * fac;
                x2 := v2 * fac;
                iset := True;
                gset:= x2;
                G := x1;
                --return;
            end if;
            end loop;

    end Gaussien16; -- Gaussian
--------------------------------
---- exp_f Funktion-------------
--------------------------------

function exp_f(x: myfloat) return myfloat is

--Diese auskommentierte Version
--verbraucht mehr Ressourcen als
--die sequentielle Version.
--------------------------------
--------------------------------
--variable y:myfloat:=float_1;
--variable d:myfloat:=float_1;
--variable t:myfloat;
--begin --t:=X;
```

```
--for i in 0 to 9 loop
--d:=t-b_LUT(i);
--      if d>=0 then
--              t:=d;
--y:=y*a_LUT(i);
--      else
--              t:=t;
--          end if;
--end loop;
--return y;
--end exp_f;
-------------------------------
-------------------------------
variable t:myfloat:=float_1;
variable y_i:myfloat:=float_1;
variable d:myfloat:=float_1;

begin t:=float_1;
y_i:=float_1;
d:=float_1;
t:=x; d:=t-b_lut(0);
    if d>=float_0 then
        t:=d;
        y_i:=y_i*a_lut(0);
    end if ;
d:=t-b_lut(1);
    if d>=float_0 then
        t:=d;
        y_i:=y_i*a_lut(1);
    end if;
d:=t-b_lut(2);
    if d>=float_0 then
        t:=d;
        y_i:=y_i*a_lut(2);
    end if;
d:=t-b_lut(3);
    if d>=float_0 then
        t:=d;
        y_i:=y_i*a_lut(3);
    end if ;
d:=t-b_lut(4);
    if d>=float_0 then
        t:=d;
        y_i:=y_i*a_lut(4);
```

```
    end if;
d:=t-b_lut(5);
    if d>=float_0 then
        t:=d;
        y_i:=y_i*a_lut(5);
    end if;
d:=t-b_lut(6);
    if d>=float_0 then
        t:=d;
        y_i:=y_i*a_lut(6);
    end if;
d:=t-b_lut(7);
    if d>=float_0 then
        t:=d;
        y_i:=y_i*a_lut(7);
    end if;
d:=t-b_lut(8);
    if d>=float16_0 then
        t:=d;
        y_i:=y_i*a_lut(8);
    end if ;
d:=t-b_lut(9);
    if d>=float16_0 then
        t:=d;
        y_i:=y_i*a_lut(9);
  end if;
return y_i;
end exp_f;

--------------------------------
---- ln_f Funktion-------------
--------------------------------

function ln_f(x: myfloat) return myfloat is

variable y:myfloat:=float_0;
variable z:myfloat;
variable d:myfloat:=float_0;
variable t:myfloat;
begin t:=X; if t>0 then for i in 0 to 9 loop
d:= t*a_lut(i);
    if d<=1 then
                t:=d;
                y:=y+b_LUT(i);
```

```
                z:=-y;
                else
                t:=t;
            end if;
end loop;
else z:=float_0;
end if;
return z;
end ln_f;
-----------------------------------------------------
-- exp Funktion nach der Formel exp(x)= 2^I.e^(f.ln2)
-----------------------------------------------------
-----------------------------------------------------------
-- exp Funktion für 16 Bit Darstellungen             ----
--- !!! Nur für Zahlen bis 23, ansonsten              ---
---- wird ein integer overflow auftreten.              --
-----------------------------------------------------------
function exp_16(x:float(6 downto -9)) return float16 is
variable t,z,tmp,y:float(6 downto -9); variable k:natural;
begin
--  x*log_2(e)= Integert-Teil + Fraktion-Teil
    t:= x*to_float(real(1.44269),6,9);
--  Integert-Teil von x*log_2(e)
    k:=integerpart(t);
--  Fraktion-Teil von x*log_2(e)
    z:=t-to_float(to_signed (k,16));
--  Fraktion-Teil*ln(2)
    tmp:=z*to_float(real(0.69315),6,9);
--  2^Integert-Teil * e^Fraktion-Teil*ln2
    y:=scalb(tmp+float16_1,k);
                            --  approximiert!!!!
return y; end exp_16;
-----------------------------------------------------------------
-- exp Funktion für 32 Bit Darstellung !! kein Overflow  !!----
-----------------------------------------------------------------
function exp_32(x:float(6 downto -9)) return myfloat is

variable t, z, tmp, y :float(8 downto -23);
variable k:natural;

begin

    t:= x*to_float(real(1.44269),8,23);
    k:=integerpart(t);
```

```
    z:=t-to_float(to_signed (k,32));
    tmp:=z*to_float(real(0.69315),8,23);
    y:=scalb(tmp+1,k);

return y; end exp_32;
-----------------------------------------
-- ln Funktion --
-----------------------------------------
function ln_16(x:float(6 downto -9)) return float16 is

variable t, p, z, l,y :float(6 downto -9);

variable n,k:integer;
begin
t:=x;
n:=Logb(t);

z:="0011111" & t(-1 downto -9);

l:=(z - float16_1); -- mac function berechnet  y:=l*r +c --
p:=mac(l,l/to_float(real(-2.0),6,9),l);

y:=p+to_float(to_signed (n+logb(z),16))*ln2_16;
return y;
end ln_16;

-----------------------------------------
-- sqrt(x):= x^0.5 := exp (0.5*ln(x))
-----------------------------------------
function sqrt_f(x:float(6 downto -9)) return float16 is

variable t,z,y:float(6 downto -9);
begin
t:=x;
z:=ln_16(t)/2;
y:=exp_16(z);
return y;

end sqrt_f;

-------------------------------

end uniform_pkg;
```

```
#######################################
           Addierer
#######################################
library IEEE,ieee_proposed;
USE ieee.std_logic_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee_proposed.math_utility_pkg.all;
use ieee_proposed.float_pkg.all;
use ieee_proposed.fixed_pkg.all;
use ieee_proposed.uniform_pkg.all;

entity float_add is
     Port ( a : in  float16;
            b : in  float16;
            s : out  float16);
end float_add;

architecture Behavioral of float_add is

begin s<=a+b;

end Behavioral;
#######################################
           Multiplizierer
#######################################

library IEEE,ieee_proposed;
USE ieee.std_logic_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee_proposed.math_utility_pkg.all;
use ieee_proposed.float_pkg.all;
use ieee_proposed.fixed_pkg.all;
use ieee_proposed.uniform_pkg.all;

entity float_mult is
    Port ( a : in  float16;
           b : in  float16;
           r : out  float16);

end float_mult;

architecture Behavioral of float_mult is
begin
```

```
        r <= a*b;
end Behavioral;
#######################################
            Divider
#######################################
library IEEE,ieee_proposed;
USE ieee.std_logic_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee_proposed.math_utility_pkg.all;
use ieee_proposed.float_pkg.all;
use ieee_proposed.fixed_pkg.all;
use ieee_proposed.uniform_pkg.all;

entity float_DIV is
    Port ( R : in  float16;
           Q : in  float16;
           D : out  float16);
end float_DIV;

architecture Behavioral of float_DIV is
signal s:float16;
begin
s<= 1/Q;
D<=R*Q;
end Behavioral;

#######################################
            LFSR
#######################################

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity LFSR is
    Port (
     clk:in bit;
     start:in bit;
     y : out  STD_LOGIC_vector(0 to 9));
end LFSR;

architecture Behavioral of LFSR is
begin
```

```
process(clk)
variable k :STD_LOGIC_vector(0 to 9):="0100110101";
variable n :STD_LOGIC_vector(0 to 9):="0101010101";
begin
if start='0' then
 y<="0000000000";
 end if;
if start='1' and clk='1' then
n(0):='1';
n(2 to 9):=k(1 to 8);
n(1):=(k(7)xor k(9))nand '1';
k:=n;
y<=k;
end if;
end process;

end Behavioral;

######################################
          Main Modul exp,ln und sqrt
######################################

library IEEE,ieee_proposed;
USE ieee.std_logic_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.std_logic_arith.all;
use ieee.numeric_std.all;
use ieee_proposed.math_utility_pkg.all;
--use ieee_proposed.float_pkg.all;
use ieee_proposed.float_pkg16.all;
use ieee_proposed.fixed_pkg.all;
use ieee_proposed.uniform_pkg.all;

entity mainmodul is
    Port ( x : in   float(6 downto -9);
           e : out  float(6 downto -9);
           k : out  float(6 downto -9);
           s : out  float(6 downto -9));
end mainmodul;

architecture Behavioral of mainmodul is

signal t,z,n:float(6 downto -9);
begin
```

```
t  <= exp_16(x);
z  <= ln_16(x);
n  <= sqrt_f(x);

e<= t;
k<= z;
s<= n;

end Behavioral;
```

Literatur

[1] Synaptisches Rauschen auf Grundlage der Poissonverteilung, Studienarbeit von Jan Loderhose universität Tübingrn

[2] NeuroFuzzy-Systeme 3. Auflage vieweg verlag

[3] http://www.tphys.uni-heidelberg.de/ horner/NIV05.pdf

[4] Computer Arithmetik Algorithms Israel KOREN PRENTICE

[5] Table-Driven Floating Point Exponential Function

[6] wikipedia.de, Stand Juli 2008

[7] Physiologie des Menschen, 30.Auflage Spinger Verlag

[8] Floating Point package user´guide

[9] IEEE-754 IEEE Standard for Binary Floating-Point Arithmetic

[10] http://www.eda.org/fphdl/vhdl.html, Stand August 2008

[11] Numerical Recipes in C The Art of Scientific Computing Second Edition / William H. Press CAMBRIDGE UNIVERSITY PRESS

[12] Yamin Li and Wanming Chu Implementation of Single Precision Floating Point Square Root on FPGAs FCCM'97, IEEE Symposium on FPGAs for Custom Computing Machines, April 16 – 18, 1997, Napa, California, USA, pp.226-232 Aizu-Wakamatsu 965-80 Japan http://www.acsel-lab.com/arithmetic/

[13] Knut Frode Pettersson, Thesis Candidatus Scientiarum: Controlled Introduction of Noise to an Integrate & Fire Neuron, University of Oslo 2004.