Ein VHDL Koprozessorkern für das exakte Skalarprodukt

Zur Erlangung des akademischen
Grades eines

DOKTORS DER
NATURWISSENSCHAFTEN

von der Fakultät für Mathematik der
Universität Karlsruhe (TH)

genehmigte

DISSERTATION

von

Dipl.-math.Norbert Bierlox
aus Neuenburg am Rhein

Tag der mündlichen Prüfung:	8. November 2002
Referent:	Prof. Dr. U. Kulisch
Korreferent:	Priv.-Doz. Dr. R. Lohner

Vorwort

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Angestellter am Lehrstuhl II des Institutes für Angewandte Mathematik der Universität Karlsruhe (TH). Ziel meiner Arbeit war es einen mathematischen Koprozessor zur Berechnung exakter Skalarprodukte für einfach genaue IEEE Zahlen (single precision) zu entwerfen.

An dieser Stelle möchte ich mich bei allen Mitarbeiterinnen und Mitarbeitern des Institutes für Angewandte Mathematik für die gute Zusammenarbeit und Hilfsbereitschaft bedanken. Sie haben damit sehr zum Entstehen dieser Arbeit beigetragen.

Ein besonderer Dank gilt meinem Referenten Herrn Prof. Dr. Ulrich Kulisch, der mir die Möglichkeit gegeben hat an seinem Institut zu arbeiten und diese Arbeit anzufertigen. Ein herzlicher Dank geht auch an Herrn Dr. habil. Rudolf Lohner für die Übernahme des Korreferats.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Bezeichnungen, Abkürzungen

ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
CLB	**C**onfigurable **L**ogic **B**lock
CSA	**C**arry **S**elect **A**dder
DLL	**D**elay **L**ocked **L**oop
EDA	**E**lectronic **D**esign **A**utomation
EDIF	**E**lectronic **D**esign **I**nterchange **F**ormat
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
GE	**G**ate **E**quivalent
HDL	**H**ardware **D**escription **L**anguage
IEEE	**I**nstitut of **E**lectrical and **E**lectronics **E**ngineers
LUT	**L**ook **U**p **T**able
MPGA	**M**ask **P**rogrammable **G**ate **A**rray
NaN	**N**ot **a** **N**umber
RTL	**R**egister **T**ransfer **L**evel
SCA	**S**tore **C**arry **A**dder
VHDL	**V**HSIC **H**ardware **D**escription **L**anguage
VHSIC	**V**ery **H**igh **S**peed **I**ntegrated **C**ircuit

Einleitung

“ A thought which does not result in an action is nothing much, and an action which does not proceed from a thought is nothing at all. ”

Georges Bernanos

Seit über 20 Jahren gehören Computer in Form von PCs und anderen Geräten zu unserem Leben. Die Geschwindigkeit und der Speicherumfang ist seit den Anfängen rapide gewachsen. Waren die PCs der achtziger Jahre noch mit knapp 5 MHz getaktet und hatten einen Hauptspeicher von durchschnittlich 64 kB, so arbeiten die heute in PCs üblichen Prozessoren bei Frequenzen von über 1 GHz und der Hauptspeicher beträgt meist 256 MB und mehr. Obwohl die Geschwindigkeit der Prozessoren auf das über 200-fache gestiegen ist (im Jahr 2002 wird die 2 GHz Grenze überschritten sein), rechnen die Computer noch genauso wie in ihren Anfangszeiten und das bedeutet leider all zu oft – sie rechnen falsch. Das einfache Beispiel[1]

$$1 \cdot 10^{17} + 1,23 - 1 \cdot 10^{17} = 0$$

zeigt, daß die Computer und damit ihre Prozessoren nichts dazu gelernt haben. In vielen Bereichen, in denen digitale Signal-Prozessoren verwendet werden, ist schon das Ergebnis der Rechnung[2]

$$1 \cdot 10^{7} + 1,23 - 1 \cdot 10^{7} = 0$$

falsch[3]. Das richtige Ergebnis bei beiden Rechnungen wäre offensichtlich 1,23. In dieser Arbeit wird gezeigt, daß so etwas heute nicht mehr notwendig ist. Auch moderne Prozessoren könnten durch moderaten Hardware-Aufwand dazu befähigt werden, diese Operationen in Hardware exakt zu berechnen. Dazu wird die Implementierung eines exakten Skalarproduktes für IEEE 754 single precision Zahlen vorgestellt. Es wurde für die Umsetzung die sogenannte Summationsmatrix gewählt, die hauptsächlich aus einem langen Addierer (über 555 Bit) und einem Shifter (64 bis

[1] bei Verwendung des IEEE 754 double precision Zahlenformates

[2] bei Verwendung des IEEE 754 single precision Zahlenformates

[3] Das ist zwar immer noch genauso falsch wie in den achtziger Jahren, aber immerhin circa 200 mal schneller berechnet

128 Bit) besteht. Das heute in PCs übliche Zahlenformat IEEE 754 double precision würde etwa den 8-fachen Aufwand an Hardware gegenüber der hier vorgestellten IEEE 754 single precision Lösung benötigen. Dieser Aufwand von circa 1 Million Gatter-Äquivalenten bei Prozessoren deren gesamter Hardwarebedarf die 10 Millionengrenze überschreitet, ist sicher vertretbar, auf alle Fälle aber überdenkenswert.

Die Wahl eine Hardware-Umsetzung für IEEE 754 single precision Zahlen, anstatt für die heute weitverbreiteten IEEE 754 double precision Zahlen zu untersuchen, geschah aus zweierlei Gründen. Erstens ist die Summationsmatrix dadurch kleiner (576 Bit gegenüber 4224 Bit) und daher auch in anderen Technologien als den High-End Technologien der Industrie umsetzbar. Zweitens gibt es noch viele Anwendungsgebiete (zum Beispiel die digitale Signalverarbeitung) in denen das IEEE 754 single precision Zahlenformat weit verbreitet ist. Zu Beginn der Arbeit (1997), war das verwendete FPGA mit circa 800.000 Gatter-Äquivalenten eines der komplexesten auf dem Markt. In dieser Arbeit wird gezeigt, daß schon in einem solchen Baustein, ohne aufwendige Technologie spezifische Handoptimierung (lediglich durch Verwendung der automatischen Werkzeuge der Hersteller) eine Taktfrequenz von 20 MHz erzielbar ist. Allein die Übertragung auf aktuelle FPGAs führt bereits auf Taktfrequenzen von über 50 MHz.

Zum Abschluß sei hier betont, daß der Schwerpunkt dieser Arbeit bei der Untersuchung der Summationsmatrix und ihrer Realisierung in Hardware liegt, jedoch nicht bei der Umsetzung der Summationsmatrix in dem zum Testen gewählten FPGA.

Die vorliegende Arbeit gliedert sich in sechs Kapitel. Im ersten Kapitel führen wir in die Grundlagen der **Rechnerarithmetik** ein. Nach einer kurzen Darstellung von Fest- und Gleitkomma-Arithmetik, erarbeiten wir die Voraussetzungen für die Berechnung eines exakten Skalarproduktes für Binärzahlen nach dem IEEE 754 Standard. Im zweiten Kapitel wird eine der möglichen Realisierungen eines solchen exakten Skalarproduktes in Hardware, die **Summationsmatrix**, vorgestellt. Dieses Kapitel beschäftigt sich allgemein mit diesem Konzept und zeigt die verschiedenen Möglichkeiten und Schwierigkeiten dieses Ansatzes auf. Im dritten Kapitel werden die **verwendeten Technologien**, beispielsweise die Hardware-Beschreibungssprache VHDL und das verwendete FPGA der Firma Xilinx vorgestellt. Das vierte Kapitel **Implementierung** beschreibt ausführlich, wie mit den Hilfsmitteln aus Kapitel 3 eine Summationsmatrix für das IEEE 754 single precision Zahlenformat in Hardware erstellt wurde. Es werden unterschiedliche Summationsmatrix-Umsetzungen vorgestellt und miteinander verglichen. Im fünften Kapitel diskutieren wir die erhaltenen **Ergebnisse** der Implementierung. Wir vergleichen sowohl den Platzbedarf (Ressourcenaufwand), als auch die Geschwindigkeit der einzelnen Umsetzungen. Abschließend werden Entscheidungshilfen zur Auswahl einer der vorgestellten Lösungen gegeben. Das letzte Kapitel **Bemerkungen** zeigt einige Anwendungsgebiet für den vorgestellten Entwurf auf. Neben der Möglichkeit auch Skalarprodukte von IEEE 754 double precision Zahlen in einer Summationsmatrix für IEEE 754 single precision exakt zu berechnen, wären dies beispielsweise eine höher genaue Arithmetik mit Hilfe von Staggered Zahlen oder eine Intervall-Arithmetik. In den Anhängen werden sowohl die Befehlsworte als auch der Quelltext und einige Hilfsprogramme für den Entwurf aus Kapitel 4 vorgestellt.

Kapitel

1

Rechnerarithmetik

> *“ It makes me nervous to fly on airplanes since I know they are designed using floating-point arithmetic ”*
>
> Alston S. Householder

1.1 Einleitung und Historische Bemerkungen

1.1.1 Zählen, Zahlen, Rechnen

1.1.2 Vom „Computer“ zum „Calculator“

Der Begriff *Computer* stammt, unschwer zu erkennen, aus dem angelsächsischen Sprachraum und wurde ursprünglich für rechnende Menschen benutzt. Umfangreiche Berechnungen wurden lange Zeit von Hand, gewissermaßen mit Papier und Bleistift durchgeführt. Häufig wurden dazu an den Universitäten Mathematikstudenten jüngerer Semester eingesetzt, beispielsweise um die großen Tafelwerke für trigonometrische Funktionen oder Logarithmentafeln zu erstellen. Auf diese Weise waren Berechnungen zumindest potentiell fehlerfrei durchführbar.

Mit Aufkommen mechanischer, ursprünglich analoger, Rechenanlagen wurden solche Berechnungen mehr und mehr automatisiert. Diese, sogenannten *Calculator*, stellten Zahlen durch drehbar gelagerte Wellen dar, wobei der Drehwinkel der Welle

mit einem Zahlwert identifiziert wurde. Gerechnet wurde mit Hilfe komplizierter mechanischer Konstruktionen. Durch den beschränkten Drehwinkelbereich war zwar nur ein beschränkter Zahlbereich verfügbar, aber zumindest theoretisch, war jede reelle Zahl in diesem Bereich darstellbar. Praktisch war aber sowohl die Einstell- und Ablesegenauigkeit der Zahlen, als auch die Genauigkeit der Rechenmechanik nur recht gering, so daß letztlich nur mit geringer Genauigkeit gerechnet werden konnte.

Erstmals in der Geschichte der Mathematik wurde nun mehr oder weniger bewußt ungenau — und somit letztlich falsch — gerechnet. Dies hat unmittelbar zur Konsequenz, daß sich offensichtlich mit diesen Rechenanlagen keine Mathematik im eigentlichen Sinne durchführen läßt sobald sie auf so berechneten Zwischenergebnissen beruht.

Ein prinzipiell anderer Ansatz zur Konstruktion einer automatischen Rechenmaschine wurde 1936 von Konrad Zuse vorgeschlagen. Mit der Z1 konstruierte er den ersten digitalen Rechner. Hier wurde also eine diskrete, endliche Zahlenmenge verwendet und somit sowohl die Unbeschränktheit, als auch die unendliche Feinheit der reellen Zahlen schon vom Ansatz her aufgegeben.

Obwohl heutzutage die „Calculator“ längst wieder als „Computer“ bezeichnet werden und man in modernen Vertretern dieser Art kaum noch deren Ursprünge zu erkennen vermag, rechnen sie doch immer noch mit einer endlichen Teilmenge R der reellen Zahlen $\mathbb{R}$ und somit zwangsläufig falsch, sobald eine Zahl außerhalb von R auftritt.

In den folgenden Abschnitten wollen wir näher untersuchen wie Computer rechnen, daß heißt, wie eventuell außerhalb R liegende Zwischenergebnisse behandelt werden und wie überhaupt die uns vertrauten Rechenoperationen auf den Computer übertragen werden.

1.1.3 Rundungen und Gerundete Arithmetik

Sei also R eine endliche und vorerst beliebige Teilmenge von $\mathbb{R}$. Versucht man zunächst naiv die Grundoperationen $\{+, -, \cdot, /\}$ von $\mathbb{R}$ nach R zu übertragen, etwa durch

$$\forall a, b \in \mathrm{R}: \quad a * b := i(a) * i(b) \qquad * \in \{+, -, \cdot, /\},$$

wobei $i : \mathrm{R} \to \mathbb{R}$ die kanonische Einbettung von R in $\mathbb{R}$ ist, so stößt man schnell auf Schwierigkeiten. Addiert man beispielsweise $\max\{\mathrm{R}\}$ zu sich selbst oder subtrahiert $\min\{\mathrm{R}\}$ von sich selbst, so liegt mindestens eines der beiden Ergebnisse außerhalb von R (wenn nicht der praktisch irrrelevante Fall $\mathrm{R} = \{0\}$ vorliegt, den wir im folgenden ausschließen).

Aber nicht nur die Beschränktheit von R ist problematisch. Weil R endlich ist, existieren sozusagen Lücken, in die man schon bei einfacher Mittelwertbildung geraten kann.

Um nun doch in R rechnen zu können, muß das Ergebnis einer Rechnung, das eventuell nicht mehr in R liegt, wieder dorthin zurück abgebildet werden. Dazu definiert man eine Funktion $\bigcirc : \mathbb{R} \to \mathrm{R}$, wobei sinnvollerweise folgende Forderungen gestellt werden [23]:

- wenn eine Zahl bereits in R liegt, so soll sie durch $\bigcirc$ nicht geändert werden und
- die von $\mathbb{R}$ induzierte Ordnung soll erhalten bleiben.

Formal schreiben sich diese Forderungen wie folgt:

$$\begin{array}{lll} \text{(R1)} & \forall a \in \mathrm{R}: & \bigcirc(a) = a, \qquad \text{(Projektion)} \\ \text{(R2)} & \forall a, b \in \mathrm{R}: & a \leq b \Rightarrow \bigcirc(a) \leq \bigcirc(b), \qquad \text{(Monotonie)} \end{array}$$

Eine Abbildung, die (R1) und (R2) erfüllt, heißt *Rundung*. Allerdings ist eine Rundung durch eben diese Forderungen noch nicht eindeutig festgelegt. Sinnvolle Rundungen sind beispielsweise die folgenden:

$$\square : \mathbb{R} \to \mathrm{R}, \quad a \mapsto \operatorname*{argmin}_{b \in \mathrm{R}} \{|b - a|\}, \tag{1.1}$$

$$\bigtriangledown : \mathbb{R} \to \mathrm{R}, \quad a \mapsto \max\{b \in \mathrm{R} : b \leq a\}, \tag{1.2}$$

oder

$$\bigtriangleup : \mathbb{R} \to \mathrm{R}, \quad a \mapsto \min\{b \in \mathrm{R} : b \geq a\}. \tag{1.3}$$

Hierbei beschreibt (1.1) die Rundung zur nächstgelegenen Zahl[1], (1.2) die Rundung zur nächstkleineren und (1.3) die zur nächstgrößeren Zahl. Die letzten beiden Rundungen heißen auch *gerichtet*. Diese Eigenschaft wird formal durch

$$\begin{array}{lll} \text{(R3)} & \forall a \in \mathrm{R}: & \bigcirc(a) \leq a, \quad \text{oder} \quad a \leq \bigcirc(a) \qquad \text{(gerichtet)} \\ & \forall a \in \mathrm{R}: & \end{array}$$

definiert. Die Rundung zur nächstgelegenen Zahl (1.1) ist überdies antisymmetrisch, das heißt, sie erfüllt

$$\text{(R4)} \quad \forall a \in \mathrm{R}: \quad \bigcirc(-a) = -\bigcirc(a). \qquad \text{(Antisymmetrie)}$$

Eine vierte, häufig verwendete Rundung, die Rundung zur Null, läßt sich leicht aus den beiden gerichteten Rundungen konstruieren:

$$\boxtimes : \mathbb{R} \to \mathrm{R}, \quad a \mapsto \begin{cases} \bigtriangledown(a) & \text{falls } a \geq 0 \\ \bigtriangleup(a) & \text{sonst.} \end{cases} \tag{1.4}$$

Diese Rundungen können nun dazu verwendet werden, um eine Arithmetik über R zu definieren. Die Grundoperationen werden dabei als Verknüpfung der entsprechenden Grundoperation in $\mathbb{R}$ mit einer nachgeschalteten Rundung definiert:

$$\text{(RG)} \quad \forall a, b \in \mathrm{R}: \quad a \circledast b := \bigcirc(a * b), \quad \text{mit } * \in \{+, -, \cdot, /\}$$

wobei $\bigcirc$ eine Rundung ist. Damit ist nun sichergestellt, daß $\circledast$ auch tatsächlich eine Abbildung von $\mathrm{R} \times \mathrm{R}$ nach R ist.

[1] Wobei die Funktion $\operatorname{argmin}_{b \in \mathbb{R}} \{|b - a|\}$ dasjenige b liefert, für das $|b - a|$ minimal ist. Der Fall, daß a genau zwischen zwei Zahlen in $\mathbb{R}$ liegt muß allerdings noch genauer definiert werden (und wird es auch in Abschnitt 1.4.3).

1.1.4 Skalarprodukte

Prinzipiell kann natürlich jede reelle Funktion nach R übertragen werden, indem die entsprechende Funktion mittels einer Rundung nach R abgebildet wird. In der Praxis, also auf dem Computer, erweist sich diese Methode aber oft als schwierig, da die meisten Funktionen mit Hilfe der Grundoperatoren ausgewertet werden und somit schon jedes Zwischenergebnis gerundet wird. Ein gutes Beispiel für dieses Vorgehen ist das Skalarprodukt zweier Vektoren $\boldsymbol{a}$, $\boldsymbol{b}$ aus dem $\mathbb{R}^n$. Unter Verwendung der oben definierten gerundeten Grundoperationen berechnen wir

$$\boldsymbol{a} \bigodot \boldsymbol{b} := a_1 \bigodot b_1 \ \oplus \ a_2 \bigodot b_2 \ \oplus \ \ldots \ \oplus \ a_n \bigodot b_n.$$

Insgesamt fallen also $2n-1$ Rundungen an. Würde man aber das Skalarprodukt mittels (RG) definieren, also nach der Formel

$$\boldsymbol{a} \bigodot \boldsymbol{b} := \bigcirc(\boldsymbol{a} \ \cdot \ \boldsymbol{b}) = \bigcirc(a_1 \cdot b_1 \ + \ a_2 \cdot b_2 \ + \ \ldots \ + \ a_n \cdot b_n),$$

so hätten wir lediglich eine einzige Rundung. Besonders deutlich wird der Unterschied zwischen diesen beiden Varianten, wenn sich im unteren Fall betragsmäßig große Summanden zu einem kleinen Ergebnis akkumulieren, dies aber im oberen Fall, aufgrund der gerundeten Zwischenergebnisse, unter Umständen nicht tun[2]. So *berechnen* beispielsweise fast alle Computer und Taschenrechner $10^{20} + 1 - 10^{20} = 0$. Das liegt daran, daß schon das erste Zwischenergebnis $10\,000\,000\,000\,000\,000\,001$ zu 10^{20}, also $10\,000\,000\,000\,000\,000\,000$ gerundet wird. Dieser relativ kleine Fehler wird dann bei der anschließenden Subtraktion von 10^{20} derart verstärkt, daß das Gesamtergebnis schon in der ersten Stelle falsch ist: eben 0 statt 1.

1.2 Festkomma-Arithmetik

In diesem und den beiden folgenden Abschnitten werden wir mögliche Strukturen von R näher untersuchen. Zunächst betrachten wir sogenannte *Festkommazahlen*, das sind Zahlen der Form

$$\pm d_1 d_2 \ldots d_m \, , d_{m+1} \ldots d_n, \tag{1.5}$$

wobei die d_i einzelne hintereinander geschriebene Ziffern sind. Im nächsten Abschnitt werden dann Zahlen eingeführt, bei denen das Komma nicht durch seine Lage innerhalb der Ziffern festgelegt ist, sondern durch einen zusätzlichen Parameter gesteuert wird. Im Abschnitt 1.4 schließlich erläutern wir den IEEE Standard 754 bzw. 854, der das heute gebräuchlichste Zahlenformat beschreibt.

Zunächst aber zu den Festkommazahlen. Ihre Form haben wir bereits in (1.5) notiert. Dabei sind die d_i aus der Menge $\{0, \ldots, b-1\}$, wobei b die *Basis* der Festkommazahl ist. Übliche Basen sind etwa 2, 8, 10 oder 16. Die Basis 10 ist wohl die gängigste Basis in der Menschen normalerweise rechnen. Zahlen zur Basis 10 werden auch *Dezimalzahlen* genannt. Sobald jedoch Zahlen auf dem Computer dargestellt werden finden sich fast ausschließlich Vielfache von 2 als Basis. Dies ist darin

[2] also wenn $\sum_{i=1}^{n} |a_i b_i| \gg |\sum_{i=1}^{n} a_i b_i|$ gilt.

begründet, daß digitale Rechner in ihren kleinsten Informationseinheiten nur zwei Zustände kodieren können, welche hier mit 0 und 1 dargestellt werden. Fassen wir nun mehrere dieser sogenannten Bits (**bi**nary digi**ts**, engl. für binäre Ziffer) zusammen so ergeben sich größere Basen. Häufig werden drei Bits verwendet, also Basis 8 (Oktalzahlen) oder 4 Bits (Hexadezimalzahlen) [3].

Zusätzlich zu den Ziffern, hat eine Festkommazahl noch ein Vorzeichen (+ oder −) und eben ein Komma. Dieses Komma kann hinter jeder beliebigen, für ein Festkommasystem aber festen, Ziffer $m \in \{1, \ldots, n\}$ stehen. Bei der Realisierung eines Festkommasystems auf einem Computer ist also die Anzahl der Ziffern n, die Stelle des Kommas m und die Basis b fest vorgegeben. Man bezeichnet ein Festkommasystem deshalb auch mit $\mathcal{D}(b, n, m)$.

Tabelle 1.1 und Abbildung 1.1 stellen das Festkommasystem $\mathcal{D}(2, 4, 1)$ dar.

$0{,}000_2$	$0{,}001_2$	$0{,}010_2$	$0{,}011_2$	$0{,}100_2$	$0{,}101_2$	$0{,}110_2$	$0{,}111_2$
$0{,}0_{10}$	$0{,}125_{10}$	$0{,}25_{10}$	$0{,}375_{10}$	$0{,}5_{10}$	$0{,}625_{10}$	$0{,}75_{10}$	$0{,}875_{10}$
$1{,}000_2$	$1{,}001_2$	$1{,}010_2$	$1{,}011_2$	$1{,}100_2$	$1{,}101_2$	$1{,}110_2$	$1{,}111_2$
$1{,}0_{10}$	$1{,}125_{10}$	$1{,}25_{10}$	$1{,}375_{10}$	$1{,}5_{10}$	$1{,}625_{10}$	$1{,}75_{10}$	$1{,}875_{10}$

Tabelle 1.1: Die positiven Zahlen des Festkommasystems $\mathcal{D}(2, 4, 1)$

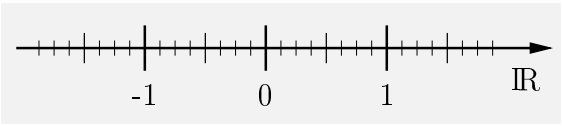


Abbildung 1.1: Das Festkommasystem $\mathcal{D}(2, 4, 1)$.

1.3 Gleitkomma-Arithmetik

Bei den Gleitkommazahlen sitzt das Komma immer an einer festen Stelle, in unserer Darstellung hinter der ersten Ziffer. Diese *Mantisse* kann nun aber durch einen zusätzlichen Faktor skaliert werden. Dieser Faktor hat die Form b^e, wobei b die bereits eingeführte Basis ist und e der sogenannte *Exponent*. Insgesamt ergibt sich also folgende Darstellung für eine Gleitkommazahl:

$$\pm d_1{,}d_2 \ldots d_n \cdot b^e = \pm \sum_{i=1}^{n} d_i b^{e-i}. \tag{1.6}$$

Der Exponent e muß dabei üblicherweise in einem festen Intervall $[e_1, e_2]$ liegen.

[3] Um Verwechslungen zu vermeiden notiert man manchmal die Basis als Index an die Zahl, also etwa $1101_2 = 15_8 = C_{16}$. An diesem Beispiel wird auch die Problematik bei Basen > 10 deutlich, die einzelnen Ziffern sind unter Umständen nicht mehr einstellig. Deshalb „zählt“ man ab 10 mit Buchstaben weiter (siehe Tabelle A.3)

Die Menge der Gleitkommazahlen ist also durch vier Parameter bestimmt: die Basis b, die Anzahl der Ziffern n, den minimalen Exponenten e_1 und den maximalen Exponenten e_2. Kurz bezeichnen wir eine solche Menge deshalb auch mit $\mathcal{F}(b, n, e_1, e_2)$. In Tabelle 1.2 und Abbildung 1.2 ist das Gleitkommasystem $\mathcal{F}(2, 3, -2, 1)$ dargestellt.

		normalisierte Zahlen			
		$1{,}00_2$	$1{,}01_2$	$1{,}10_2$	$1{,}11_2$
Exponent	1	$2{,}0_{10}$	$2{,}5_{10}$	$3{,}0_{10}$	$3{,}5_{10}$
	0	$1{,}0_{10}$	$1{,}25_{10}$	$1{,}5_{10}$	$1{,}75_{10}$
	-1	$0{,}5_{10}$	$0{,}625_{10}$	$0{,}75_{10}$	$0{,}875_{10}$
	-2	$0{,}25_{10}$	$0{,}3125_{10}$	$0{,}375_{10}$	$0{,}4375_{10}$
		denormalisierte Zahlen			
		$0{,}00_2$	$0{,}01_2$	$0{,}10_2$	$0{,}11_2$
	-2	$0{,}0_{10}$	$0{,}0625_{10}$	$0{,}125_{10}$	$0{,}1875_{10}$

Tabelle 1.2: Die positiven Zahlen des Gleitkommasystems $\mathcal{F}(2, 3, -2, 1)$ mit denormalisierten Zahlen.

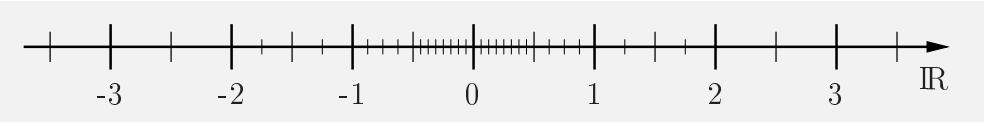


Abbildung 1.2: Das Gleitkommasystem $\mathcal{F}(2, 3, -2, 1)$ mit denormalisierten Zahlen.

Um Redundanzen zu vermeiden, wird üblicherweise gefordert, daß die Ziffer vor dem Komma (d_1) ungleich 0 ist. Solche Zahlen werden dann *normalisiert* genannt. Lediglich für den kleinsten Exponenten ($e = e_1$), sind auch *denormalisierte* Zahlen zugelassen, solche mit $d_1 = 0$. Damit erweitert sich die Menge der Gleitkommazahlen und wird um die 0 *dichter*.

Betrachten wir die Abbildungen und Tabellen 1.1 und 1.2, so sehen wir, daß Gleitkommazahlen im Gegensatz zu Festkommazahlen nicht äquidistant verteilt sind. Sie häufen sich um die Null und liegen mit wachsendem Betrag immer weiter auseinander. Dadurch decken sie einen relativ großen Bereich der reellen Zahlen ab, allerdings eben mit unterschiedlicher Feinheit. Die Festkommazahlen hingegen haben zwischen dem größten und kleinsten Element eine äquidistante Verteilung, dafür decken sie einen im allgemeinen deutlich kleineren Bereich der reellen Zahlen ab. Dies hat zur Folge, das Berechnungen mit Festkommazahlen sehr schnell in den Überlaufbereich geraten (über die größte Zahl des Festkommasystems hinaus) und deshalb stets aufwendig skaliert werden müssen.

1.4 IEEE 754 / 854

Um das Rechnen mit Gleitkommazahlen zu vereinheitlichen wurde nach mehrjährigen Anstrengungen und Diskussionen im Jahr 1985 ein Standard für Gleitkomma-

zahlen in binärer Darstellung verabschiedet. Dieser Standard IEEE 754-1985 (siehe [2]) definiert zwei unterschiedlich große Gleitkommazahlensysteme, die IEEE *single precision* Zahlen und die IEEE *double precision* Zahlen. Im Jahre 1987 wurde ein erweiterter Standard verabschiedet, der die schon bestehenden Vereinbarungen für Binärzahlen auf Zahlen mit beliebiger Basis erweitert (siehe [3]).

Wir wollen uns hier nur mit dem Standard für die Binärdarstellung beschäftigen, da dies die gebräuchliche Zahlendarstellung in den heutigen Computern ist. Die IEEE 754 single precision Zahlen bestehen aus einer 32 Bit Darstellung, die IEEE 754 double precision Zahlen verwenden 64 Bit.

1.4.1 Single Precision

Die 32 Bit der IEEE 754 single precision Zahlen teilen sich wie folgt auf:

Vorzeichen	s	das am weitesten links stehende (32. Bit) repräsentiert das Vorzeichen. (eine Eins bedeutet negativ)
Exponent	E	die nächsten 8 Bit (von links) bilden den Exponenten
Mantisse	$b_1 \cdots b_{23}$	die restlichen 23 Bit sind die Mantissenbits

Tabelle 1.3: IEEE 754 single Precision Zahl

Der Exponent wird mit einem sogenannten *bias* von +127 dargestellt, so daß die Darstellung des Exponenten mit dem Wert 0 die binäre Zahl `01111111` (dezimal 127) entspricht. Der Exponent bewegt sich für normalisierte Zahlen zwischen −126 (Binärdarstellung mit bias: `00000001`) und +127 (Binärdarstellung mit bias: `11111110`). Die Exponentendarstellung `00000000` wird für die Darstellung der Null (dabei sind die Mantissenbits alle Null) und für die Darstellung der denormalisierten Zahlen (mindestens ein Mantissenbit ist ungleich Null), verwendet. Mit einem Exponenten der Binärdarstellung `11111111` (mit bias) werden ±infinity (Mantissenbits alle Null) und die sogenannten NaN (**N**ot **a** **N**umber) dargestellt. ±infinity wird zur Darstellung von Zahlen die größer oder kleiner als die darstellbaren Werte sind, verwendet. Ein NaN tritt immer dann auf, wenn die Operation zu keinem sinnvollen Ergebnis führt, wie zum Beispiel bei $\frac{0}{0}$ oder $0 \times \infty$. Die IEEE 754 single precision Zahlen entsprechen dem Gleitkommasystem F(2,24,-126,127) mit denormalisierten Zahlen.

Die IEEE 754 single precision Zahlen lassen sich wie folgt aus ihrer Binärdarstellung berechnen:

$$(-1)^s \times 2^{E-127} \times b_0, b_1 \cdots b_{23}$$

wobei E die Dezimalzahl der entsprechenden Binärdarstellung ist und b_0 ist das sogenannte *hidden bit*. Das *hidden bit* b_0 ist bei den normalisierten Zahlen immer Eins (Exponent von `00000001` bis `11111110`) und bei den denormalisierten Zahlen (Exponent ist `00000000`) immer Null. Die darstellbaren Zahlen bei dem IEEE 754 single precision Zahlenformat sind der Tabelle 1.4 zu entnehmen.

	Binärdarstellung	dezimale Näherung
betragsgrößte normalisierte Zahl	s11111111111111111111111111111111 $= (-1)^s \times 2^{127} \times 1,1\cdots1_2$	$\pm 3,402823466E+038$
betragskleinste normalisierte Zahl	s00000001000000000000000000000000 $= (-1)^s \times 2^{-126} \times 1,0\cdots0_2$	$\pm 1.175494351E-038$
betragsgrößte denormalisierte Zahl	s00000000111111111111111111111111 $= (-1)^s \times 2^{-126} \times 0,1\cdots1_2$	$\pm 1.175494211E-038$
betragskleinste denormalisierte Zahl	s00000000000000000000000000000001 $= (-1)^s \times 2^{-126} \times 0,0\cdots01_2$	$\pm 1.401298464E-045$

Tabelle 1.4: Zahlenbereich der single precision Zahlen

1.4.2 Double Precision

Der prinzipielle Unterschied zwischen den IEEE 754 single precision und IEEE 754 double precision Zahlen ist der, daß wir im double-Fall 64 Bit zur Darstellung zur Verfügung haben. Diese 64 Bit teilen sich auf in 1 Vorzeichen-, 11 Exponenten- und 52 Mantissenbits. Auch hier gelten alle Darstellungen wie NaN und ±infinity entsprechend mit dem einzigen Unterschied, daß jetzt alle elf Exponenten-Bits Eins sind. Der Exponenten-bias bei IEEE 754 double precision Zahlen ist 1023 und der Exponentenbereich bewegt sich von -1022 (Binärdarstellung mit bias: `00000000001`) bis $+1023$ (Binärdarstellung mit bias: `11111111110`). Der darstellbare Zahlenbereich erstreckt sich also (genäherte Werte) von

$$-8.9884656743E+307 \quad \text{bis} \quad +8.9884656743E+307$$

die betragsmäßig kleinste darstellbare (denormalisierte) Zahl ist näherungsweise

$$\pm 9.8813129168E-324$$

Die IEEE 754 double precision Zahlen entsprechen dem Gleitkommazahlensystem F(2,53,-1022,1023) mit denormalisierten Zahlen.

1.4.3 Rundungen

Wie wir schon in Absatz 1.1.3 gesehen haben, benötigen wir bei der Rechnung in einem endlichen Zahlensystem Rundungen um eventuell außerhalb liegende Ergebnisse wieder in das Zahlensystem abzubilden. Selbst zur Konvertierung von einfachen Dezimalzahlen in ihre Binärdarstellung benötigen wir Rundungen. Die Dezimalzahl 0,1 ist binär mit endlicher Mantisse nicht exakt darstellbar. Die beiden nächst gelegenen Zahlen im IEEE double precision Format entsprechen etwa den Dezimalwerten

$0,1000000000000000555\ldots$ nächst größere darstellbare Zahl
oder
$0.0999999999999999167\ldots$ nächst kleinere darstellbare Zahl .

Aus diesen Gründen verlangen die IEEE 754/854 Standards vier verschiedene Rundungsarten. Das ist zum einen die Rundung nach oben (*round toward* $+\infty$), die Rundung nach unten (*round toward* $-\infty$), die Rundung zur Null (*round toward* 0) und die Rundung zur nächsten Zahl (*round to nearest*). Die Rundungen haben wir schon in Abschnitt 1.1.3 definiert, so daß wir hier nur noch den Sonderfall bei der Rundung zur nächsten Zahl behandeln müssen. Falls ein zu rundender Wert genau zwischen zwei darstellbaren Zahlen des Gleitkommasystems liegt so verlangen die IEEE Standards, daß von den beiden Zahlen diejenige gewählt wird, deren letzte Ziffer gerade ist. Deswegen wird diese Rundung auch *round to nearest even* genannt. Speziell bedeutet dies im binären Fall (IEEE 754), daß die Zahl gewählt wird deren letzte Ziffer eine Null ist.

1.4.4 Skalarprodukt

In Abschnitt 1.1.4 wurde schon festgestellt, daß speziell bei Skalarprodukten die mehrfache Anwendung einer Rundung zu beliebig falschen Ergebnissen führen kann. Als Abhilfe wurde in Abschnitt 1.1.4 die exakte Berechnung des Skalarproduktes mit nur **einer** abschließenden Rundung vorgestellt. Wir müssen also (beliebig viele) Produkte von Gleitkommazahlen exakt aufaddieren. Dies geht am einfachsten, wenn wir eine große (Festkomma-)Zahl haben, in die wir alle diese Produkte akkumulieren. Diese Festkommazahl muß den gesamten möglichen Exponentenbereich der Gleitkommaprodukte abdecken. Das betragsmäßig kleinste Produkt (Produkt der kleinsten denormalisierten Zahl mit sich selbst, siehe 1.4) hat den Exponenten $2\times e_1$, das betragsmäßig größte Produkt den Exponenten $2\times e_2$. Alle Produkte haben eine Mantisse die doppelt so breit ist wie die der verwendeten Gleitkommazahlen, also $2\times n$. Daraus ergibt sich, daß das Festkommasystem in dem der lange Akkumulator für das exakte Skalarprodukt von Gleitkommazahlen liegt, eine Ziffernanzahl von $2\times(e_2-e_1)+2\times n$ benötigt. Da die mehrfache Addition des größten Produktes ein Überlauf in diesem Festkommasystem zur Folge hätte, wählt man noch eine ausreichende Anzahl von k Ziffern zusätzlich, um eventuelle Überläufe aufzufangen. Zur exakten Berechnung von Skalarprodukten im Gleitkommasystem $\mathcal{F}(b,n,e_1,e_2)$ (siehe 1.3), benötigen wir ein Festkommasystem $\mathcal{D}(b,l,m)$ mit

$$\begin{aligned} l &:= 2\times(e_2-e_1)+2\times n+k \\ m &:= 2\times|e_2|+2 \\ k &> 1 \quad \text{ausreichend.} \end{aligned}$$

Für IEEE 754 single precision Zahlen erhalten wir daher

$$l = 2\times(127+126)+2\times 24+k = 554+k \tag{1.7}$$

wenn jetzt $k=22$ gewählt wird, ergibt sich eine Ziffernanzahl von $576 = 9\times 64$. Auch bei IEEE 754 double precision Zahlen wird k so groß gewählt, daß wir wieder ein für die Umsetzung in digitale Schaltungen günstige Ziffernanzahl erhalten. Zum Beispiel wird in [5] $k=92$ gewählt, da dann $l = 4288 = 67\times 64$ ist. Die Ziffernanzahl l bei IEEE 754 double precision Zahlen muß nach obiger Formel mindestens

$$l = 2\times(1023+1022)+2\times 53+k = 4196+k \tag{1.8}$$

betragen.

1.5 Zusammengesetzte Zahlenformate

Reicht die Rechengenauigkeit zur Durchführung einer bestimmten Aufgabe nicht aus, ist es oft hilfreich, die Genauigkeit der zugrundeliegenden Arithmetik zu verbessern. Da vom Prozessor in Hardware bereitgestellte Arithmetiken um ein vielfaches schneller sind, als per Software simulierte Arithmetiken, wird meist versucht, vorhandene Datentypen zu neuen zusammenzusetzen um, wenigstens in Teiloperationen die schnelle Hardwarearithmetik auszunutzen.

In den folgenden Abschnitten wollen wir zwei solche zusammengesetze Zahlenformate beschreiben. Das erste erhöht lediglich die Präzision bei der Gleitkommarechnung, das zweite liefert sogar in gewissem Sinne die mathematische Schärfe zurück, die ja vermeintlich beim Übergang auf eine endlich genaue Arithmetik verloren gegangen war. Dazu werden bei jeder Operation anstelle eines möglicherweise falschen Ergebnisses gleich zwei falsche Ergebnisse zurückgegeben, allerdings eines, das garantiert kleiner ist als der exakte Wert und eines, daß garantiert größer ist.

1.5.1 Staggered Zahlen

Staggered (engl. angehäuft) Zahlen können durch eine Summe gewöhnlicher Gleitkommazahlen dargestellt werden. Die wesentliche Voraussetzung hierbei ist allerdings, daß diese Summe exakt ausgerechnet wird, also nicht mit Hilfe der gewöhnlichen Additionsroutinen für die Summanden. Um diese verschiedenen Additionen zu unterscheiden, werden wir in diesem Kapitel für gerundete Operationen ausschließlich die Notation gemäß Abschnitt 1.1.3 verwenden.

Definition 1.1 *Gegeben seien l Gleitkommazahlen $x^{(1)}, \ldots, x^{(l)}$. Eine Staggered Zahl x der Länge l ist definiert durch*

$$x := \sum_{k=1}^{l} x^{(k)}.$$

Um maximale Genauigkeit zu erhalten, sollen sich die Sumanden nicht überlappen, d.h. es soll gelten $|x^{(1)}| > \ldots > |x^{(l)}|$ und die Exponenten zweier aufeinanderfolgender Sumanden sollen sich mindestens um die Anzahl der Ziffern pro Sumand n unterscheiden. In diesem Fall stellt x eine Gleitkommazahl mit mindestens $n \cdot l$ Ziffern, aber dem selben Exponentenbereich des zugrundeliegenden Gleitkommasystems, dar.

Angenommen, wir hätten ein Gleitkommasystem mit drei dezimalen Ziffern und vier Zahlen daraus: $x^{(1)} = 1.65 \cdot 10^3$, $x^{(2)} = 3.94 \cdot 10^0$, $x^{(3)} = 5.75 \cdot 10^{-5}$ und $x^{(4)} = 2.24 \cdot 10^{-8}$, so können wir damit eine Staggered Zahl der Länge 4 mit mindestens 12, in diesem Falle sogar 14 Ziffern darstellen:

$$x = 1.65 \cdot 10^3 + 3.94 \cdot 10^0 + 5.75 \cdot 10^{-5} + 2.24 \cdot 10^{-8} = 1.6539400575224 \cdot 10^3.$$

Arithmetische Grundoperationen

Beim Entwurf einer Staggered-Arithmetik müssen wir zunächst entscheiden, mit welcher Genauigkeit die Ergebnisse einzelner Operationen zurückgegeben werden

sollen. Wollten wir stets das exakte Ergebnis zurückgeben, so ergäben sich schnell sehr lange Zahlen und wir würden bereits an der Division scheitern, wo wir ja häufig unendlich viele Ziffern erhalten. Aus diesen Gründen definieren wir die Genauigkeit, d.h. die Anzahl der Gleitkommazahlen des Ergebnisses, als das Maximum der Genauigkeit der Operanden.

Unter Verwendung des Exakten Skalarproduktes lassen sich die Algorithmen für die Addition, Subtraktion und Multiplikation leicht angeben. Es wird lediglich das Ergebnis in einem langen Akkumulator berechnet und anschließend werden Schritt für Schritt die Gleitkomma-Sumanden der Ergebnis-Staggered-Zahl ausgelesen (siehe Algorithmus 1.1). Im Weiteren verwenden wir diesen Algorithmus mit der Schreibweise $z = \mathit{round\_to\_staggered}(\mathit{accu})$, wobei z eine Staggered-Zahl ist und accu ein langer Akkumulator (siehe 1.4.4).

Gegeben sei ein langer Akkumulator accu
$l = \mathit{staggered\_length\_of}(z)$
for $k = 1, \ldots, l$
  $z^{(k)} = \mathit{round\_towards\_zero}(\mathit{accu})$
  $\mathit{accu} = \mathit{accu} - z^{(k)}$

Algorithm 1.1: Schrittweises Ausrunden eines langen Akkumulators accu in die Summanden der Staggered-Zahl z.

Als einfaches Beispiel für die Grundarithmetik mit Staggered-Zahlen, geben wir hier die Subtraktion an. Gegeben seien zwei Staggered-Zahlen $x := \sum_{k=1}^{l_1} x^{(k)}$ und $y := \sum_{k=1}^{l_2} y^{(k)}$. Algorithmus 1.2 beschreibt die Berechnung von $z := \sum_{k=1}^{\max\{l_1,l_2\}} z^{(k)} := x - y$.

Gegeben seien zwei Staggered-Zahlen x und y
$l_1 = \mathit{staggered\_length\_of}(x)$; $l_2 = \mathit{staggered\_length\_of}(y)$
$\mathit{accu} = 0$
for $k = 1, \ldots, l_1$
  $\mathit{accu} = \mathit{accu} + x^{(k)}$
for $k = 1, \ldots, l_2$
  $\mathit{accu} = \mathit{accu} - y^{(k)}$
$\mathit{set\_length}(z, \max\{l_1, l_2\})$
$z = \mathit{round\_to\_staggered}(\mathit{accu})$

Algorithm 1.2: Subtraktion zweier Staggered-Zahlen x und y. Das exakte Zwischenergebnis wird im langen Akkumulator accu gespeichert.

Eine Arithmetik für Staggered-Zahlen ist beispielsweise in den XSC Sprachen [17] implementiert und beruht massiv auf der Verfügbarkeit des exakten Skalarproduktes.

Skalarprodukte von Staggered Vektoren

Da wir bei der Berechnung von Skalarprodukten bei Staggered-Vektoren ohnehin für die Teilprodukte und zur Akkumulation ein Exaktes Skalarprodukt einsetzen

müssen, läßt sich die Berechnung eines Staggered Skalarproduktes deutlich vereinfachen, wenn wir alle Teilprodukte in einem einzigen Akkumulator zusammenzählen. Durch Vermeidung des Rundens der Zwischenergebnisse, wird der Algorithmus sogar noch schneller. Algorithmus 1.3 beschreibt dieses Skalarprodukt für Staggered-Vektoren.

Gegeben seien zwei Staggered-Vektoren $\boldsymbol{x} = \left(x_i\right)_{i=1}^{n}$ und $\boldsymbol{y} = \left(y_i\right)_{i=1}^{n}$
$l_1 = \mathit{staggered\_length\_of}(\boldsymbol{x})$
$l_2 = \mathit{staggered\_length\_of}(\boldsymbol{y})$
$accu = 0$
for $i = 1, \ldots, n$
  for $j = 1, \ldots, l_1$
    for $k = 1, \ldots, l_2$
      $accu = accu + x_i^{(j)} \cdot y_i^{(k)}$
$z = \mathit{round\_to\_staggered}(accu)$

Algorithm 1.3: Das Skalarprodukt zweier Staggered-Vektoren $\boldsymbol{x}$ und $\boldsymbol{y}$. Das exakte Zwischenergebnis wird im langen Akkumulator *accu* gespeichert.

1.5.2 Intervalle

Um verläßliche, garantierte Ergebnisse zu erhalten, etwa zur Berechnung garantierter Fehlerschranken, genügt es nicht, lediglich höhergenau zu rechnen. Für diese Art von Aufgaben muß die Ungenauigkeit, die zwangsläufig durch das Runden entsteht, verläßlich kontrolliert werden, siehe [1], [21]. Wie wir bereits im Abschnitt 1.3 gesehen haben, kann jede Grundoperation mit Gleitkommazahlen ein Ergebnis liefern, daß nicht mehr im entsprechenden Gleitkommaformat exakt darstellbar ist. Aus diesem Grunde wurden im Abschnitt 1.1.3 Rundungen eingeführt. Üblicherweise wird die Rundung zur nächstgelegenen Zahl verwendet, weil so durch das Runden der kleinste Fehler entsteht. Allerdings geht dabei schon nach kurzen Rechnungen die Information verloren, ob das exakte Ergebnis oberhalb oder unterhalb des erhaltenen gerundeten Wertes liegt. Die Idee bei der Intervallrechnung ist es nun, anstelle eines Wertes zwei zurück zu liefern, wobei ein Wert garantiert oberhalb des exakten Ergebnisses liegt und der andere unterhalb. Damit erhalten wir als Datentyp Intervalle, welche wir formal mit folgender Definition beschreiben.

Definition 1.2 *Seien $\underline{x} \leq \bar{x}$ zwei Gleitkommazahlen, dann nennen wir die Menge $[x] := [\underline{x}, \bar{x}] := \{\xi \in \mathbb{R} \mid \underline{x} \leq \xi \leq \bar{x}\}$ ein (Gleitkomma-)Intervall.*

Bemerkung: Das Intervall $[\underline{x}, \bar{x}]$ enthält *alle reellen* Zahlen zwischen $\underline{x}$ und $\bar{x}$, nicht nur die Gleitkommazahlen.

Da Intervalle Elemente der Potenzmenge über $\mathbb{R}$ sind, übertragen sich die Operationen restriktiv via

$$[\underline{x}, \bar{x}] * [\underline{y}, \bar{y}] := \{\xi * \eta \mid \underline{x} \leq \xi \leq \bar{x} \wedge \underline{y} \leq \eta \leq \bar{y}\} \text{ mit } * \in \{+, -, \cdot, /\} \quad (\text{und } 0 \notin [\underline{y}, \bar{y}] \text{ für } * = /). \tag{1.9}$$

Unter Ausnutzung der Monotonie und Stetigkeit reduzieren sich diese unendlich vielen Operationen zur Berechnung der rechten Seite in (1.9) auf einige wenige. Beispielsweise berechnet sich die Addition zu $[\underline{x},\bar{x}]+[\underline{y},\bar{y}]=[\underline{x}+\underline{y},\bar{x}+\bar{y}]$. Unglücklicherweise sind aber die Schranken des Ergebnisintervalls unter Umständen nicht im Gleitkommaformat darstellbar, müssen also gerundet werden. Um sicherzustellen, das wir hierbei die exakte Lösung nicht verlieren, muß $\underline{x}+\underline{y}$ nach unten gerundet werden und $\bar{x}+\bar{y}$ nach oben [20, 22]. Zusammengefaßt müssen wir also die Addition folgendermaßen definieren:

$$[\underline{x},\bar{x}]\diamondsuit[\underline{y},\bar{y}] := [\nabla(\underline{x}+\underline{y}),\triangle(\bar{x}+\bar{y})], \tag{1.10}$$

wobei ∇ und $\triangle$ die gerichteten Rundungen aus Abschnitt 1.1.3 sind. Manchmal wird auch die etwas kürzere Notation

$$[\underline{x},\bar{x}]\diamondsuit[\underline{y},\bar{y}] := \diamondsuit([\underline{x}+\underline{y},\bar{x}+\bar{y}])$$

verwendet, oder allgemein nach (RG)

$$\begin{aligned}[\underline{x},\bar{x}]\diamondsuit[\underline{y},\bar{y}] &:= [\nabla(\inf\{[\underline{x},\bar{x}]*[\underline{y},\bar{y}]\}),\triangle(\sup\{[\underline{x},\bar{x}]*[\underline{y},\bar{y}]\})]\\ &= \diamondsuit([\underline{x},\bar{x}]*[\underline{y},\bar{y}])\end{aligned}$$

Es gibt zahlreiche Softwarebibliotheken, welche Intervall-Datentypen zur Verfügung stellen, etwa die XSC-Sprachen [13, 14, 17], oder `Profil/BIAS` [18, 19] und Intlab [27]. Neuerdings gibt es auch kommerzielle Fortran/C/C++ Compiler von Sun-Microsystems [29]. Darüberhinaus existiert sogar eine Intervall-Staggered Erweiterung zu Pascal-XSC [25].

Exkursion: Einschließung von Gleitkomma-Berechnungen

Insbesondere bei der Einschließung linearer Ausdrücke erweist sich das exakte Skalarprodukt als sehr nützlich. Haben wir beispielsweise ein lineares Gleichungssystem $\boldsymbol{Ax}=\boldsymbol{b}$ mit einer Gleitkomma-Matrix $\boldsymbol{A}$, und zwei Gleitkomma-Vektoren $\boldsymbol{b}$ (rechte Seite) und $\tilde{\boldsymbol{x}}$ (Näherungslösung), so können wir den Residuenvektor $\boldsymbol{r} := \boldsymbol{b}-\boldsymbol{A}\tilde{\boldsymbol{x}}$ auf zwei Weisen einschließen.

for $i=1,\ldots,n$
  $r_i = \diamondsuit b_i$
  for $j=1,\ldots,n$
    $r_i = r_i \diamondsuit a_{i,j} \diamondsuit \tilde{x}_j$

a) mit Intervallarithmetik

for $i=1,\ldots,n$
  $accu = b_i$
  for $j=1,\ldots,n$
    $accu = accu - a_{i,j}\cdot\tilde{x}_j$ /* exakt */
  $r_i = \diamondsuit accu$

b) mit dem Exakten Skalarprodukt

Algorithm 1.4: Berechnung einer Einschließung des Residuenvektors mit zwei unterschiedlichen Techniken.

Entweder wandeln wir (mit dem $\diamondsuit$-Operator) implizit alle Gleitkommazahlen in Punktintervalle um[4] und verwenden dann gewöhnliche Intervallarithmetik ($\diamondsuit$, $\diamondsuit$),

[4] $x \mapsto [x,x]$

siehe Algorithmus 1.4a, oder wir berechnen das Residuum mit Hilfe des exakten Skalarproduktes exakt und runden anschließend den Akkumulator einmal nach unten und einmal nach oben zu einem (sehr engen) Gleitkomma-Intervall.

Die Qualität der Einschließung (also der Durchmesser von $\boldsymbol{r}$) hängt dabei im ersten Fall ganz entscheidend von den Daten ab. Der relative Fehler in b) ist stets die Maschinengenauigkeit ϵ, während jedoch der relative Fehler in a) lediglich durch

$$\frac{(n+1)\epsilon}{1-(n+1)\epsilon} \cdot \left(|r_i| + \sum_{j=1}^{n} |a_{i,j}\tilde{x}_j| \right),$$

abgeschätzt werden kann (siehe [11]). Dies kann beliebig schlecht sein, wenn $|r_i| + \sum_{j=1}^{n} |a_{i,j}\tilde{x}_j|$ viel größer ist als $|r_i + \sum_{j=1}^{n} a_{i,j}\tilde{x}_j|$, wenn also die Summe der Beträge groß ist, gegenüber dem Betrag der Summe.

Kapitel

2

Summationsmatrix

“ the ferret, hunting eyes on the ground,
never hears footsteps of the hawk ”

Andrew Vachss: Footsteps of the Hawk

In diesem Kapitel werden wir die einzelnen Komponenten und Strategien betrachten die für eine Hardwarerealisierung eines exakten Skalarproduktes notwendig sind. Desweiteren werden wir auch die Möglichkeiten und Probleme, die sich aus den einzelnen Ansätzen ergeben, erörtern. Eine Diskussion über die tatsächliche Realisierung und die dabei auftretenden Probleme wird in einem späteren Kapitel stattfinden (siehe Kapitel 4.1), nachdem wir auch die Werkzeuge und Technologien, die zur Erstellung einer solchen Hardware notwendig sind, vorgestellt haben.

2.1 Implementierung des exaktes Skalarprodukt

Wie wir schon in 1.4.4 gesehen haben werden für die Berechnung eines exakten Skalarproduktes in IEEE 754 double precision 4196 Bits und für IEEE 754 single precision 554 Bits benötigt. In Registern dieser Breiten müssen dann die Produkte der entsprechenden Formate addiert werden. Da die Produkte in Abhängigkeit des Exponenten in verschiedene Bereiche des Registers addiert werden, ist auch ein Shifter notwendig. Im weiteren Verlauf dieses Kapitels wird von einer IEEE 754

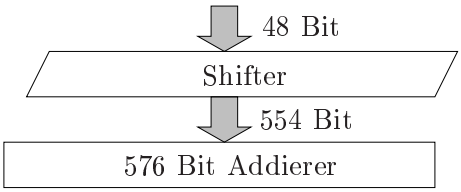


Abbildung 2.1: Langer Shifter, langer Addierer

single precision Lösung ausgegangen. Deswegen ist die Breite des Registers für das exakte Skalarprodukt $554 + k$ Bits (k Schutzbits). Hier bietet sich an, $k = 22$ zu wählen, so daß die Gesamtbreite $554 + 22 = 64 * 9 = 576$ beträgt. Die Breite eines single precision Produktes beträgt $24 * 2 = 48$ Bit.

2.1.1 Langer Addierer

Die erste Idee, das exakte Skalarprodukt für IEEE 754 single precision in Hardware zu realisieren, wäre ein 576 Bit breiter Addierer und ein fast ebenso breiter Shifter. (s. Abbildung 2.1)

Dabei muß ein links-Shift über die gesamten 554 Bits (ohne die Schutzziffern) möglich sein. Der Nachteil dieser Lösung ist zum einen der lange Addierer und zum anderen ein so breiter Shifter, der sowohl viele Ressourcen als auch eine lange Ausführungszeit benötigt. Natürlich würde ein so breiter Shifter mehrstufig aufgebaut werden, so daß die Ausführungszeit nicht mehr störend ist (kleiner als die Ausführungszeit eines 32 Bit Addierers), dafür werden aber mehrere Pipeline-Stufen eingeführt. Auch der 576 Bit breite Addierer wird mehrstufig aufgebaut, zum Beispiel aus 9 hintereinander geschaltenen 64-bit Addierern. Dabei ergeben sich Überträge zwischen den Addierern die später aufgelöst werden müssen.

Wenn wir annehmen, daß der Shifter 3 Takte verbraucht und im Addierer bis zu 8 Überträge zwischen den Addierern auftreten können, so ergeben sich maximal 11 Takte. Das heißt, daß bei diesem Konzept nicht nur ein großer Gatteraufwand sondern auch ein großer Zeitaufwand bezüglich der Ausführungszeit vorhanden ist.

2.1.2 Speicherlösung

Eine andere Möglichkeit der Implementierung ist die sogenannte Speicherlösung. Dieses Konzept wurde 1996 am Institut für Angewandte Mathematik der Universität Karlsruhe (TH) für das IEEE 754 double precision Format in Hardware umgesetzt [5].

Bei dieser Methode besteht der 4288 Bit lange Akkumulator für das Datenformat double precision aus einem entsprechend großen Speicher. Es werden nur die Speicherworte in einen Addierer geladen, in die das Produkt (in Abhängigkeit von seinem Exponenten) hinein addiert wird. Dieser Addierer muß nur so viele Speicherworte breit sein, wie von einem Produkt berührt werden. Auch der dazu gehörige Shifter braucht nur die Breite des Addierers zu haben. Bei einer IEEE 754 single precision Implementierung wäre ein Addierer (und der Shifter) bei einer angenommenen Speicherwortbreite von 32 Bit nur 96 Bit breit, da ein 48 Bit breites Produkt auf höchstens 3 Speicherworte trifft.

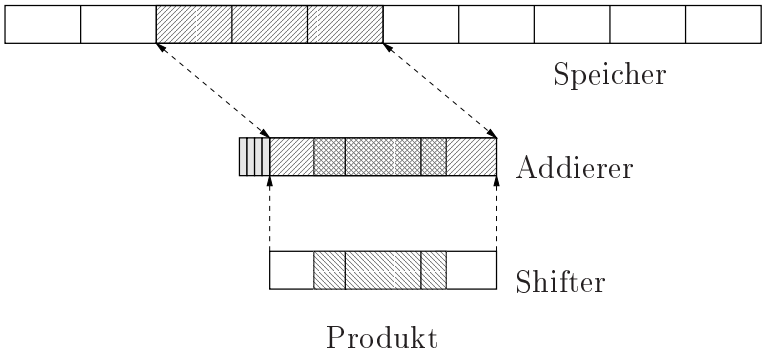


Abbildung 2.2: Speicherlösung

Der Vorteil dieser Lösung liegt darin, daß nur ein „kurzer“ Addierer und ein ebenso kurzer Shifter notwendig sind. Stattdessen wird Speicher in der Größe des Akkumulators benötigt, was aber vom Gatteraufwand sicher geringer ist als ein entsprechend großer Addierer.

Der Nachteil dieser Lösung liegt in der Zyklenanzahl die für eine Operation notwendig sind, wobei eine Parallelisierung nicht oder nur mit erheblichem Mehraufwand möglich ist. Die folgenden Schritte sind für die Akkumulation eines einzelnen Produktes notwendig:

1. Ermitteln der Adresse der betroffenen Speicherzellen
2. Auslesen der Speicherzellen und Laden in den Addierer
3. Zurechtschieben (shiften) des Produktes
4. Addieren des Produktes zum ausgelesenen Speicherinhalt
5. Rückschreiben der Summe in den Speicher
6. Behandlung eventuell auftretender Überträge

Dabei kann der Punkt 3 (shiften) parallel zu Punkt 2 (auslesen) ausgeführt werden. Für die Übertragsbehandlung sind spezielle Mechanismen notwendig, eine elegante und schnelle Lösung ist unter [5] zu finden.

2.1.3 Matrixanordnung

Eine weitere Möglichkeit (auf die in dieser Arbeit ausführlich eingegangen wird) ist die sogenannte Matrixanordung im weiteren Verlauf als Summationsmatrix (siehe [16]) bezeichnet. Dabei wird der 576 Bit breite Addierer in Zeilen untereinander angeordnet. Der Vorteil dieser Anordnung ist, daß der Shifter nur so breit wie die Matrixzeile sein muß. Jede Matrixzeile besteht aus mindestens zwei von einander unabhängig ansteuerbaren Addierern. Eine Steuerlogik generiert die Ansteuersignale für die einzelnen Addierer, so daß nur die richtigen Addierer auf den Summanden (bzw. einen Teil davon) zugreifen. Die Wahl der Breite einer Matrixzeile, die Anzahl der Zeilen und die Länge der Teiladdierer der Matrixzeilen sind wichtige Kenngrössen dieses Modelles.

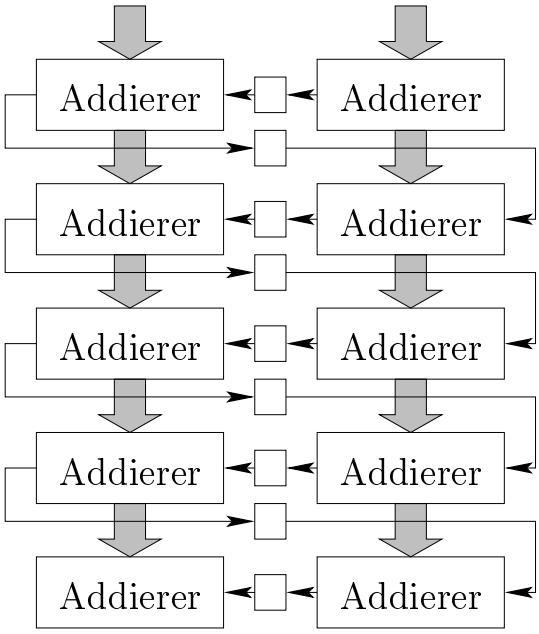


Abbildung 2.3: Die Summationsmatrix

In Abbildung 2.3 handelt es sich um das Beispiel einer 5-zeiligen Matrix wobei jede Zeile aus zwei getrennten Addierern besteht. Zwischen den Addierern sind die 1-bittigen Übertragsspeicher zu erkennen. Nach der Multiplikation wird das Produkt entsprechend seines Exponenten so geshiftet, daß es für die Addierer, die diesen Exponentenbereich bearbeiten, an der richtigen Stelle liegt. Die zuständigen Addierer akkumulieren dann die Daten zu ihrem Inhalt. Nach dieser Addition kann das nächste zurechtgeschobene Produkt angelegt werden. Nachdem alle notwendigen Produkte akkumuliert wurden, kann nach der Auflösung aller Überträge (das kann mehrere Takte beanspruchen; s. Abschnitt 2.4) mit dem Auslesen des Skalarproduktes begonnen werden. Dies geschieht im einfachsten Fall durch das Auslesen aller Matrixzeilen (je eine pro Takt) und anschließender Bearbeitung des Skalarproduktes, wie z.B. einer Rundung. Dabei muß nicht der ganze Inhalt der Summationsmatrix bearbeitet werden, sondern nur die Stellen die für die Ausgabe notwendig sind.

2.2 Kenngrößen

Um im weiteren Verlauf dieser Arbeit eine eindeutige Bezeichnung der verschiedenen Elemente und Größen der Summationsmatrix verwenden zu können definieren wir die folgenden Begriffe:

p_l	Breite des Multipliziererausgangs; bei IEEE 754 single precision 48 Bit
s_l	Breite des Shifterausganges = Zeilenbreite
n	Anzahl der Teiladdierer einer Matrixzeile
m	Anzahl aller Teiladdierer der Summationsmatrix
Axy	x-ter Teiladdierer der y-ten Matrixzeile
Axy_n	Breite des Teiladdierers Ayx; falls alle Addierer gleich breit sind verwenden wir A$_n$

a	1. Faktor, oder Faktor a
b	2. Faktor, oder Faktor b
mul	Multiplizierer
p	Produkt
s	Shifter
B1	Shifterteilausgang 1
B0	Shifterteilausgang 0
A00	Teiladdierer der 0-ten Matrixzeile
A01	Teiladdierer der 0-ten Matrixzeile
A10	Teiladdierer der 1-ten Matrixzeile
A11	Teiladdierer der 1-ten Matrixzeile

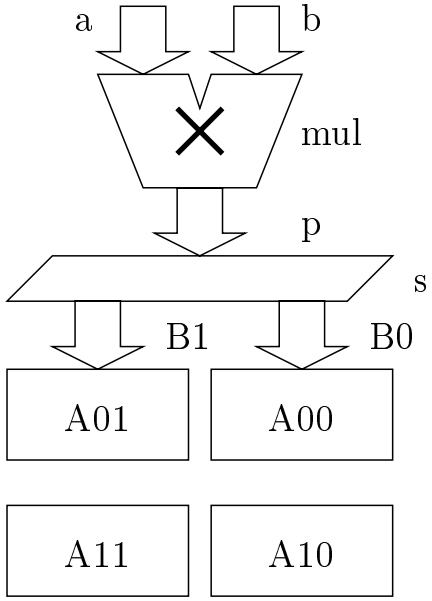


Abbildung 2.4: Begriffe der SuMa

Die Bezeichnung Teiladdierer (oder nur Addierer) ist in diesem Zusammenhang irreführend, da es sich bei diesen Elementen eigentlich um einen Addierer/Subtrahierer mit daran angeschlossenem Akkumulationsspeicher handelt (siehe Abbildung 2.5).

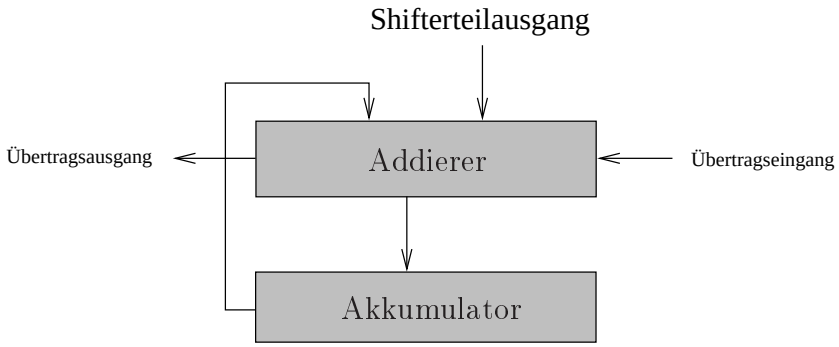


Abbildung 2.5: Aufbau eines Teil-„Addierers“

2.3 Funktionsweise

In Abbildung 2.6 wird die Funktionsweise der Summationsmatrix und die Zusammenarbeit mit dem Shifter verdeutlicht. In diesem Beispiel besteht die Zeile einer Summationsmatrix aus zwei gleich großen Addierern wobei die Addiererbreite größer als die Produktbreite ist ($A_n > p_l$ Abbildung 2.6, 1. Zeile). Der Shifter schiebt das Produkt dem Produktexponenten entsprechend zurecht. Danach wird der Shifterausgang s an die Addierer angelegt. Die dem Exponenten des Produktes entsprechenden Addierer addieren das Produkt. Dabei können sich folgende Fälle ergeben.
Das Produkt fällt vollständig in einen Teiladdierer (Abbildung 2.6, 2. Zeile), dadurch kann am Ausgang dieses Teiladdierers eventuell ein Übertrag auftreten.

Das Produkt verteilt sich auf zwei Addierer der gleichen Matrixzeile (Abbildung 2.6, 3. Zeile), es können am Ausgang beider Addierer Überträge auftreten. Das Produkt verteilt sich auf zwei Addierer zweier aufeinanderfolgenden Matrixzeilen (Abbildung 2.6, 4. Zeile), es können am Ausgang beider Addierer Überträge auftreten.

2.3.1 Zeilenbreite

Die Breite einer Zeile der Summationsmatrix beeinflußt folgende Größen:

1. Breite des benötigten Shifters
2. maximale Breite der einzelnen Addierer
3. maximale Breite des Produktes
4. die Zeilenanzahl

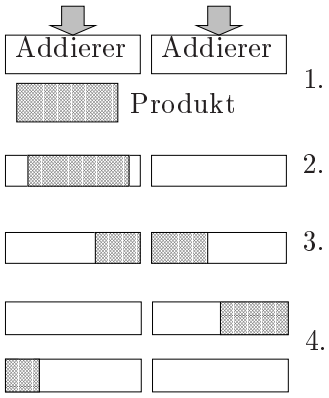


Abbildung 2.6: Das geshiftete Produkt

Shifterbreite

Da der Shifter das Produkt über die gesamte Breite schieben muß, ist er genauso breit wie eine Matrixzeile. Wie in Abbildung 2.6 4. Zeile zu sehen ist, muß es ein rotierender Shifter sein. Das bedeutet, daß er die Bit des Produktes, die er an der einen Seite rausschiebt (die Bits die aus dem Shifter „fallen“) auf der anderen Seite (in gleicher Reihenfolge) wieder einfügen muß.

Zeilenanzahl

Die Zeilenanzahl ist direkt von der Länge des Akkumulators und der Zeilenlänge abhängig, so ergibt sich bei der IEEE 754 single precision Lösung mit einem 576 Bit langen Akkumulator für das exakte Skalarprodukt und einer Zeilenlänge von 128 Bit eine Zeilenanzahl von 4,5 und damit insgesamt 5 Zeilen, wobei eine der Zeilen nicht vollbesetzt ist.

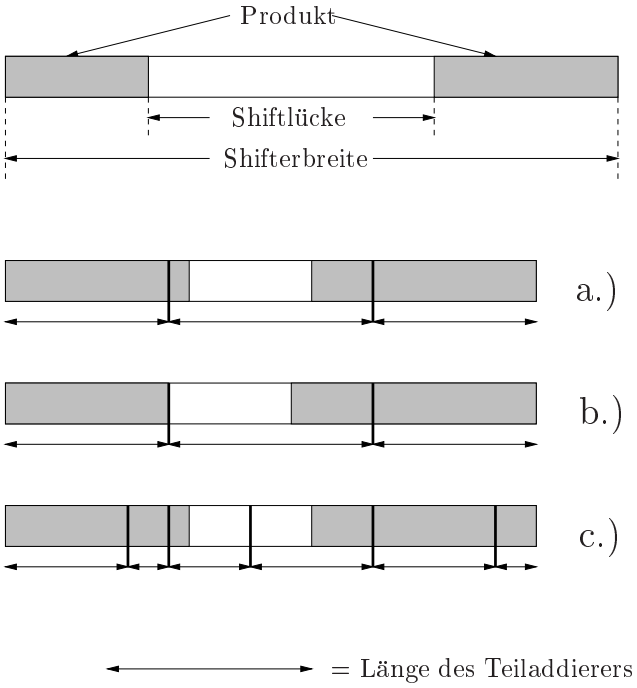


Abbildung 2.7: Shiftlücke

Addiererbreite

Da jede Matrixzeile aus mindestens zwei von einander unabhängig ansteuerbaren Addierern bestehen muß wird die Addiererbreite auch durch die Zeilenlänge der Matrix begrenzt. Dabei ist es nicht notwendig, daß alle Addierer einer Zeile gleich lang sind. Die einzige Bedingung ist, daß die Shiftlücke bis zu einem (linker oder rechter) Rand eines Teiladdierers reicht. Dies muß für alle möglichen Shiftweiten gelten. In Abbildung 2.7 a.) reicht die Shiftlücke an keinen Rand eines Teiladdierers sondern befindet sich innerhalb des mittleren Teiladdierers und daher ist dies **keine** zulässige Aufteilung der Matrixzeile. In Teil b.) der selben Abbildung reicht der linke Rand der Shiftlücke zwar an den linken Rand des zweiten Teiladdierers dies gilt aber (wie in Teil a.) gesehen) nicht für alle Shiftweiten und daher ist diese Aufteilung auch **nicht** zulässig. Im Teil c.) der Abbildung gibt es keinen Teiladdierer, der breiter als die Shiftlücke ist, und daher reicht die Shiftlücke für alle Shiftweiten bis zum Rand (oder darüber hinaus) eines Teiladdierers. Deswegen ist diese ungleichmäßige Aufteilung eine zwar ungewöhnliche aber **zulässige** Zerlegung in Teiladdierer. Es muß also für alle k Teiladdierer einer Zeile y gelten:

$$s_l - p_l + 1 \geq \max_{1 \leq y \leq k} \mathrm{A}xy_n \quad \textit{für alle } x \tag{2.1}$$

bzw. *(falls alle Addierer gleich breit sind)*

$$s_l - p_l + 1 \geq \mathrm{A}_n \tag{2.2}$$

Normalerweise werden alle Teiladdierer gleich breit gewählt. Die Länge ist meist ein Vielfaches von 2, da dadurch die digitale Ansteuerlogik, die bestimmt ob ein Teiladdierer das anliegende Produkt addieren muß oder nicht, vereinfacht wird.

Produktbreite

Die mögliche Produktbreite p_l wird auch durch die Gleichung 2.1 beziehungsweise 2.2 bestimmt. So kann ein Produkt bei einer fixen Shiftbreite von $s_l = 128$ Bit je nach (gleichmässiger) Aufteilung der Matrixzeile in k gleich grosse Teiladdierer folgende Werte annehmen:

$k = 2 \quad \Rightarrow \quad A_n = 64 \quad$ Bit und $p_l \leq 65$ Bit
$k = 4 \quad \Rightarrow \quad A_n = 32 \quad$ Bit und $p_l \leq 97$ Bit
$k = 8 \quad \Rightarrow \quad A_n = 16 \quad$ Bit und $p_l \leq 113$ Bit
$k = 16 \quad \Rightarrow \quad A_n = 8 \quad$ Bit und $p_l \leq 121$ Bit

2.4 Überträge

Beim Studium der Tabelle in 2.3.1 kommt man zum Schluß, daß die beste Teiladdiererlänge 16 Bit oder kürzer wäre. Damit läßt man aber vollkommen außer Acht, daß sich zwischen den Teiladdierern Überträge ansammeln, und zwar je kürzer ein Teiladdierer ist um so größer ist die Anzahl der Addierer und dadurch auch die Gesamtanzahl der Überträge. Wir werden später zwei Ansätze betrachten, diese Überträge zu behandeln. Bei dem einen Ansatz werden die auftretenden Überträge zwischen den Addierern in einem extra Register gespeichert, bei dem anderen Ansatz werden sie sofort aufgelöst.

2.4.1 Zwischenspeichern

Das Problem mit zwischengespeicherten Überträgen ist, daß sie nachträglich aufgelöst werden müssen. Das geschieht dadurch, daß in jedem Takt ein etwa anliegender Übertrag mit berücksichtigt wird. Der Addierer akkumuliert also nicht nur den für ihn zuständigen Teil des Produktes (oder Null falls er nicht am Produkt akkumulieren beteiligt ist) sondern auch noch einen eventuell vorhandenen Übertrag dazu. In jedem Takt werden die bestehenden Überträge verarbeitet, doch können dadurch und durch den neuen Summanden wieder Überträge entstehen. In einer Summationsmatrix mit 16 Teiladdierern gibt es 15 mögliche Überträge. Selbst wenn nur zwischen den zwei niederwertigsten Addierern ein Übertrag gespeichert ist, so kann sich durch die Auflösung dieses Übertrages an einem höherwertigen Addierer ein neuer Übertrag bilden. Es werden also bis zu m - *1* Takte benötigt, bis ein Ergebnis ausgelesen werden kann.

2.4.2 Carry Select

Um dieses Nachtakten zu vermeiden, kann die sogenannte Carry-Select Technik angewandt werden (*engl.* carry: Übertrag, select: Auswahl). Dabei wird nicht nur das eigentliche Ergebnis der Addition berechnet, sondern auch das um eins erhöhte Ergebnis. Dies geschieht je nach Technologie entweder durch einen speziellen Addierer (einem carry-select adder) oder durch zwei getrennte Addierer. Da Addierer normalerweise schon einen Übertragseingang haben, muß dazu nur dieser Eingang bei dem einen Addierer auf 0 und bei dem anderen auf 1 gesetzt werden. Wenn alle Addierer (parallel) ihre zwei Ergebnisse ermittelt haben, wird in Abhängigkeit der möglichen

Überträge bei allen Addierer gleichzeitig lokal das richtige dieser zwei Ergebnisse ausgewählt und gespeichert. Danach besteht keine Notwendigkeit irgendwelche Überträge zwischen den Addierern zu speichern und der Inhalt des langen Akkumulators kann sofort ohne notwendige Nachtakte ausgelesen werden. Diese Methode benötigt nicht nur einen erhöhten Aufwand bei den Addierern (150 - 200 % Gatteraufwand) sondern auch noch Zeit nach dem Addieren zur Auswahl und Abspeicherung des richtigen Wertes. Da viele Entwürfe von symmetrischem Takt ausgehen, und in der zweiten Takthälfte nur das Abspeichern des errechneten Wertes stattfindet, nimmt diese Lösung nur wenig oder gar keine zusätzliche Zeit in Anspruch. Nähere Einzelheiten dazu werden im Abschnitt 4.13, wenn es um die Umsetzung einer Summationsmatrix geht, besprochen.

Auch diese Lösung profitiert von wenigen (also „großen“) Addierern, da zum sofortigen Auflösen aller Überträge an allen Addieren die lokalen Überträge der niederwertigeren Addierer notwendig sind. Das bedingt zum einen den Transport vieler Signale über die ganzen Addierer und zum anderen wird die Auflösungslogik an den höherwertigen Addierern komplexer und damit immer zeitintensiver, so daß sich der erhoffte Gewinn durch das Einsparen der Nachtakte in eine Verlängerung (Verlangsamung) des einzelnen Taktes wandelt. Die Verdopplung des Platzbedarfes durch Einsetzen von Carry-Select ist in den heutigen Technologien meist nicht so dramatisch, wie die Notwendigkeit vieler zusätzlicher Signale über die ganzen Addierer. Wir dürfen dabei nicht vergessen, daß wir schon einen 128 Bit breiten Bus (Shifterausgang) an die Addierer haben.

2.5 Kontextspeicher

Es ist je nach Anwendung vorteilhaft oder gar notwendig, daß mehrere exakte Skalarprodukte gleichzeitig vorhanden sind. Dies ist zum einen natürlich durch die Vervielfachung der beschriebenen Hardware, also der Bereitstellung mehrerer Summationsmatrizen, oder durch das Verwenden von zusätzlichem Speicher, möglich. Der Vorteil des zusätzlichen Speichers ist, daß mit wenig (Gatter-)Aufwand mehrere Skalarprodukte bereitgestellt werden. Der Nachteil dieser Lösung ist, daß es nicht möglich ist, mehrere Skalarprodukte (zeitlich) parallel zu berechnen, da der Multiplizierer, Shifter und die Addierer nur einmal vorhanden sind. Wenn durch die Verwendung eines solchen Speichers mehrere Skalarprodukte vorhanden sind, muß diesen zur Unterscheidung noch eine Speicheradresse zugeordnet werden.

Diesen Speicher und seine Adressen wollen wir im weiteren Verlauf als **Kontextspeicher** bzw. **Kontextadressen** bezeichnen. Der Inhalt des Kontextspeichers an einer Kontextadresse wollen wir hier kurz mit **Kontext** bezeichnen. Dieser Speicher ist direkt an jedem Teiladdierer (eigentlich ja ein Akkumulator, s. 2.2) angesiedelt. Dabei darf nicht vergessen werden, daß das Laden und das Zurückschreiben des entsprechenden Speicherwortes auch wieder Zeit erfordert. Das Zurückschreiben des Speicherinhaltes ist natürlich nur dann erforderlich wenn im nächsten Takt die Bearbeitung eines anderen Kontextes (und damit einer anderen Speicherstelle) vorgesehen wird. Deswegen ist die Verwendung eines Speichers mit mehreren von einander unabhängigen Schnittstellen (z. B. dual port RAM) vorteilhaft. Dadurch kann das Zurückspeichern des einen Kontextes und das Laden eines anderen gleichzeitig erfolgen. Somit braucht die Ansteuerung des Speichers keine weitere Taktzeit,

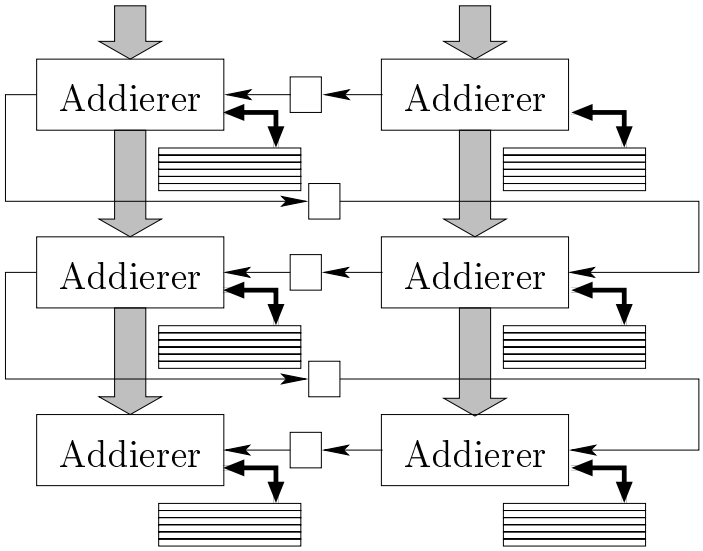


Abbildung 2.8: Die Summationsmatrix mit Kontextspeicher

da der Wechsel des Kontextes schon vor dem Addieren (zur Multiplikationszeit oder zur Shiftzeit) feststeht und deswegen das Laden des neuen Kontextes am Ende des Taktes vor der Addition (parallel zum Abspeichern) stattfinden kann.

2.5.1 Parallelbearbeitung

Mit solchem Kontextspeicher ist es möglich gleichzeitig mehrere Skalarprodukte zu bearbeiten, zwar kann pro Verarbeitungstakt immer nur ein Produkt akkumuliert werden, aber dafür in ein beliebiges Skalarprodukt bzw. in den Kontext der dieses Skalarprodukt repräsentiert. Dies ist für Algorithmen die mehrere Skalarprodukte (gleichzeitig) benötigen, wie zum Beispiel bei einem Intervall Skalarprodukt, von nutzen.

Wenn der Benutzer mehrere Programme gleichzeitig ausführt, werden unter Umständen auch mehrere Skalarprodukte gleichzeitig benötigt. Die gleiche Situation ist gegeben, wenn die Summationsmatrix-Hardware in einer Mehrbenutzerumgebung (multi user enviroment) installiert ist. Auch dabei benötigen dann ein oder mehrere Programme ein oder mehrere Skalarprodukte. Da die Abbildung der vorhandenen Skalarprodukte auf die Programme vom darüber liegenden Betriebssystem eines Rechners vorgenommen wird, müssen wir uns hier nicht mit den unterschiedlichen Fällen

- ein Programm; mehrere Skalarprodukte
- mehrere Programme eines Benutzers; jeweils ein oder mehrere Skalarprodukte
- mehrere Benutzer; jeweils ein oder mehrere Skalarprodukte
- mehrere Benutzer, mehrere Programme; jeweils ein oder mehrere Skalarprodukte

beschäftigen. Die einzige Frage, die wir hier betrachten wollen ist, was für Möglichkeiten haben wir, wenn die Anzahl der vorhandenen Skalarprodukte nicht ausreicht.

Natürlich ist durch eine Vervielfachung der Skalarprodukt-Hardware also der Installation mehrerer Summationsmatrizen in einer Rechnerumgebung dieses Problem zu umgehen. In diesem Fall hat das Betriebssystem des Rechners die zusätzliche Aufgabe, nicht nur die einzelnen Skalarprodukte auf die Kontexte sondern auch noch auf die unterschiedlichen Summationsmatrizen zu verteilen. Da dies ausschließlich ein Verwaltungsproblem des Betriebssystems ist werden wir hier nicht weiter darauf eingehen.

Was aber soll geschehen, wenn mehr Skalarprodukte benötigt werden, als physikalisch vorhanden sind? Eine einfache Strategie wäre solange alle Anforderungen nach einem neuen Skalarprodukt in eine Warteschleife zu stellen, bis wieder einer der benutzten Kontexte freigegeben wird. Dieser freie Kontext kann dann einer der Anforderungen der Warteschleife zugeordnet werden. Auch diese Strategie findet nur auf der Betriebssystemsebene statt.

Was aber soll geschehen, wenn das Betriebssystem eine der üblichen Strategien des Auslagerns (swapping oder paging bei Speicherverwaltung) auf Skalarprodukte anwenden möchte? Eigentlich sind ja die Werte der Skalarprodukte (der Inhalt des Kontextspeichers) auch nur Bitmuster oder lange „Zahlen“. In diesem Fall muß die Hardware eine Möglichkeit des Auslesens des Kontextspeichers und das Laden eines ausgelagerten Kontextes ermöglichen.

2.5.2 Auslagerung

Falls die Summationsmatrix nicht immens viel Speicher (und damit Kontexte) zur Verfügung hat, wird es immer den Fall geben, daß zu wenig Kontexte zur Verfügung stehen. Eventuell weitere vorhandene Information über einen Kontext, wie z.B. die Matrixzeile in der sich das erste von Null verschiedene Bit des Skalarproduktes oder das Vorzeichen des Skalarproduktes befindet, müssen für jeden der vorhandenen Kontexte zusätzlich abgespeichert werden. Deswegen sollen alle vorhandenen Informationen, nicht nur das eigentliche Skalarprodukt, als Kontext bezeichnet werden und zum Kontextspeicher gehören. Um dann einen oder mehrere Kontexte auszulagern, ist es notwendig, den Inhalt des Kontextspeichers auszulesen. Dies kann zum einen über die gleiche Schnittstelle über die die Summationsmatrix ausgelesen wird erfolgen, oder über eine spezielle Schnittstelle zum Kontextspeicher.

Wichtig ist, daß dieses Auslesen schnell erfolgt, da ein IEEE 754 single precision Kontext über 576 Bit (also mehr als 72 Byte) umfaßt. Die Strategie, welche Kontexte und wieviele Kontexte ausgelagert werden, soll dabei gänzlich dem Betriebssystem (bzw. dem Treiber der Summationsmatrix) überlassen werden. Davon ausgehend, daß das Auslesen eines Kontextes mit der gleichen Taktrate und Breite geschieht wie das Akkumulieren der Produkte, benötigt das Auslagern eines Kontextes in unserem Beispiel 5 bzw. 10 Takte. Dabei wurde beim ersten Fall (5 Takte) angenommen, daß eine 128 Bit breite Schnittstelle (= Shifterbreite = Matrixzeilenlänge) vorhanden ist. Wenn von einer $2*32 = 64$ Bit breiten (zwei IEEE 754 single precision Operanden) oder einer 48 Bit breiten (= Produktbreite = Shiftereingangsbreite) Schnittstelle ausgegangen wird, so ergeben sich 10 oder gar 14 Takte um den gesamten Kontext auszulesen. Dabei kann, falls das Auslagern über dieselbe Schnittstelle

wie das Eingeben der Produkte bzw. Summanden erfolgt, die Summationsmatrix in dieser Zeit keine Akkumulationen ausführen.

2.5.3 Kontext laden

Irgendwann muß auch wieder mit einem der ausgelagerten Kontexte weiter gearbeitet werden. Dafür muß der ausgelesene Kontextspeicherinhalt wieder in die Summationsmatrix zurückgeschrieben werden. Eventuell ist vor diesem Zurückschreiben erst ein Auslagern eines anderen Kontextes notwendig, um für den einzulesenden Kontext Platz zu schaffen. Das Zurückschreiben erfordert dann mindestens genauso viele Takte wie das Einlesen. Das heißt, daß ein Auslagern eines Kontextes nur dann Sinn ergibt, wenn die Lebensdauer (in Takten gerechnet) erheblich größer ist als die Aus- und Einlesezeit. Damit nicht kurzlebige Skalarprodukte ausgelagert werden, wäre eine Markierung (durch Compiler oder Benutzer) denkbar die dem Betriebssystem mitteilt, daß es keinen großen Gewinn erzielt, wenn es ein so markiertes Skalarprodukt auslagert.

2.6 Fensterlösung

Wie wir schon in Abschnitt 1.4.4 gesehen haben, benötigt ein IEEE 754 double precision Skalarprodukt über 4196 Bit, also fast das 8-fache der über 554 Bit eines IEEE 754 single precision Skalarproduktes. Die wirklich ausgenutzte Länge bei IEEE 754 double precision Skalarprodukten ist im Normalfall um einiges geringer. In den meisten Fällen ist sie sogar so klein, daß sie in eine Realisierung eines IEEE 754 single precision, wie zum Beispiel die vorgestellte Summationsmatrix passen würde. Das legt die Idee nahe ein IEEE 754 double precision Skalarprodukt auch wirklich mit einer IEEE 754 single precision Hardware zu berechnen. Dabei kann natürlich nur ein kleiner Ausschnitt des gesamten Bereiches, ein **Fenster**, der 4288 Bit (mit $k = 22$ Schutzziffern) eines exakten IEEE 754 double precision Skalarproduktes in einer 576 Bit Summationsmatrix berechnet werden. Doch was passiert wenn dieses Fenster bei einer Anwendung nicht ausreicht?

Eine einfache Möglichkeit wäre, ein neues Fenster in dem Bereich, der benötigt wird, anzulegen. Das ist mit einem weiteren Kontext sehr leicht zu bewerkstelligen. Diese Idee, ein grosses Skalarprodukt in einer Hardware die nicht den gesamten Bereich des Skalarproduktes abdeckt, zu berechnen, wollen wir im weiteren Verlauf mit **Hardware-Fenster** bezeichnen. Da wir uns in dieser Arbeit hauptsächlich mit Hardwarelösungen beschäftigen, sprechen wir nur von **Fenster** oder **Fensterlösung**. Wir wollen die Fenster anhand eines IEEE 754 double precision in einer IEEE 754 single precision Hardware diskutieren. Dabei ist es meistens unerheblich, auf welche Art (Summationsmatrix, Speicherlösung, Langer Addierer) das Skalarprodukt in Hardware berechnet wird. Die verschiedenen Fensterlösungen können auch als Speicherlösung mit „überdimensioniertem“Addierer betrachtet werden. Da bei der Fensterlösung der Addierer unnötig groß (mehr als 550 Bit) ist, gegenüber den 96 Bits (siehe 2.1.2), die eine Speicherlösung für 48 Bit (oder auch 64 Bit) Produkte benötigt, wollen wir hier diesen Ansatz gesondert betrachten. Bei der Untersuchung einer Fensterlösung ergeben sich die folgenden Probleme:

1. welchen Bereich soll das Start-Fenster abdecken?

2. welchen Bereich sollen die Zusatz-Fenster abdecken?

3. Wie wird aus mehreren Fenstern das Ergebnis ermittelt?

4. reicht eine IEEE 754 single precision Hardware um IEEE 754 double precision Skalarprodukte effektiv zu berechnen?

Wenden wir uns als erstes den Punkten 1 und 2 zu.

2.6.1 Starre Fenster

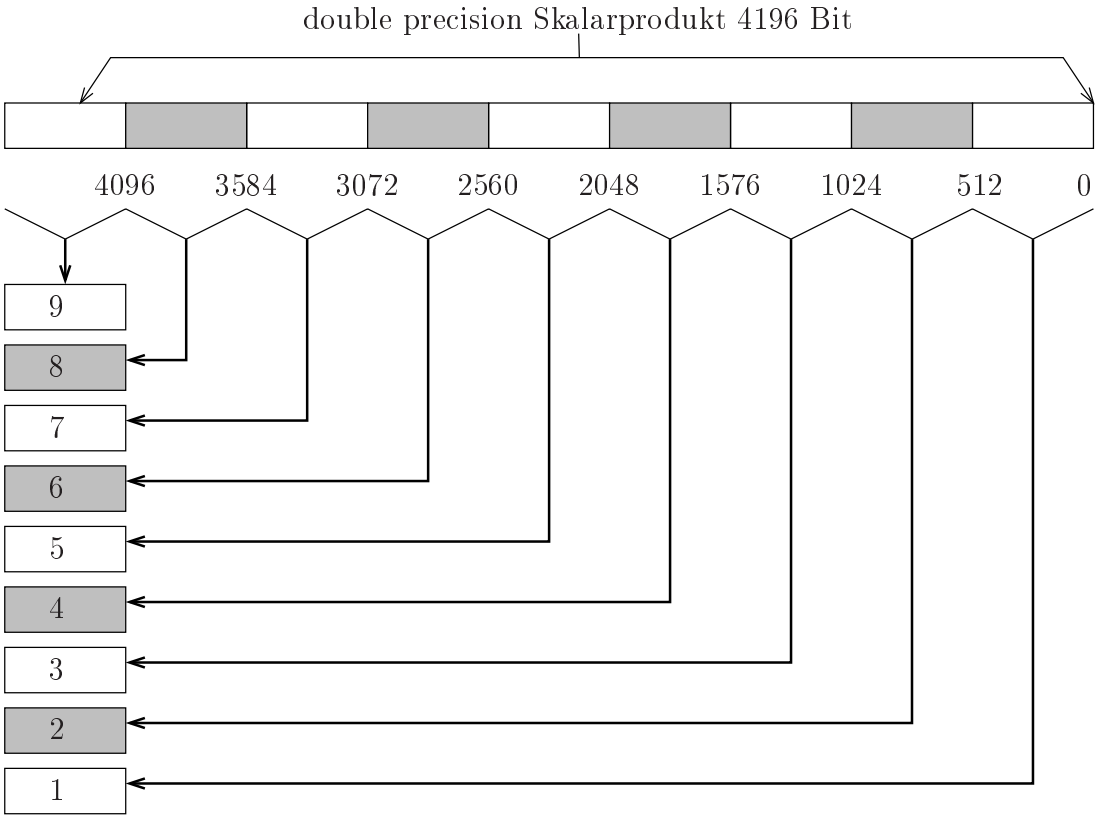


Abbildung 2.9: Starre Aufteilung

Eine Möglichkeit wäre die 4196 Bit des IEEE 754 double precision Skalarproduktes in $2^9 = 512$ Bit große Fenster aufzuteilen. Diese feste Aufteilung werden wir im weiteren Verlauf als **starre** Fenster bezeichnen. Dann werden 9 Kontexte benötigt um die gesamte Breite abzudecken. Die Vorteile dieser Lösung sind:

- einfache Zuordung der Produkte zu den Kontexten anhand des Produktexponenten
- einfache Berechnung des Kontextes da Fensterbreite eine Zweierpotenz
- feststehender Resourcenbedarf von 9 Kontexten

Dabei ist der letzte Punkt auch gleichzeitig ein Nachteil. Selbst wenn nur ein oder zwei Kontexte wirklich benötigt werden sind alle 9 Kontexte für dieses Skalarprodukt reserviert (s. Abbildung 2.10). Um diese Resourcenverschwendung zu umgehen, könnten die einzelnen Kontexte nur bei Bedarf verwendet werden. Das bedingt die Notwendigkeit der Kommunikation mit dem Betriebssystem. Denn erst wenn der Produktexponent feststeht, kann ermittelt werden ob ein neuer Kontext benötigt wird. Danach muß das Betriebssystem bestimmen, welcher der Kontexte für das neue Fenster verwendet werden soll. Anschließend findet unter Umständen noch ein Auslagern statt, falls der Kontext für das neue Fenster schon belegt ist.

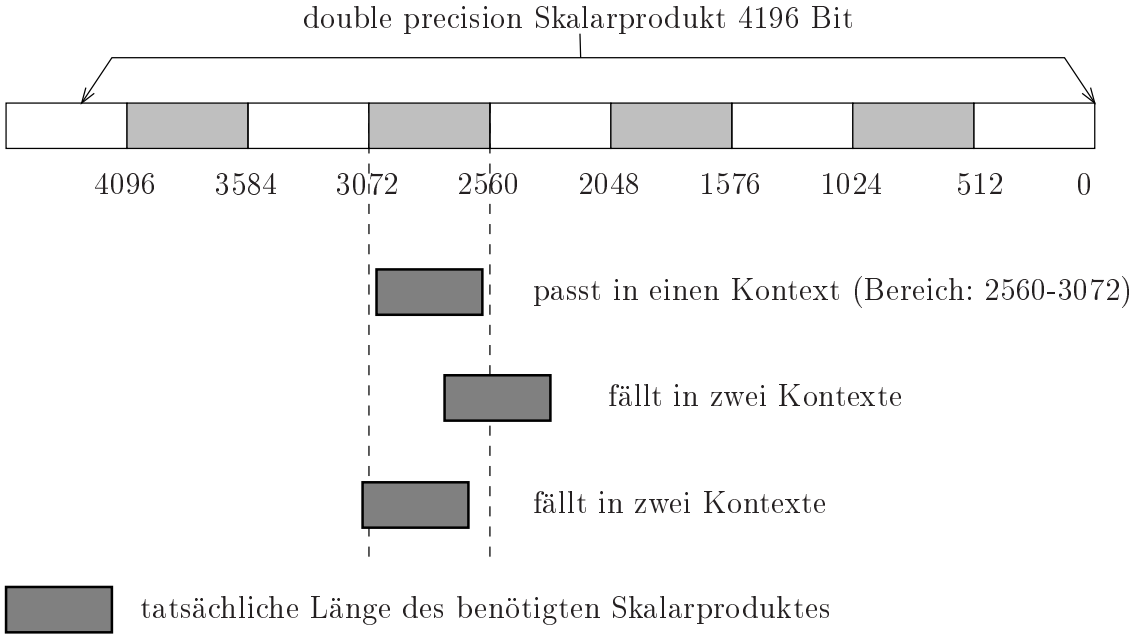


Abbildung 2.10: Effektive Breite $\leq$ 512 Bit

Bei vielen Programmen, die IEEE 754 double precision als Zahlenformat verwenden, sind die effektiven Breiten der IEEE 754 double precision Skalarprodukte meist kleiner als 500 Bit. Das würde aber bedeuten, daß solche Skalarprodukte in einen IEEE 754 single precision Kontext passen würden. In der Praxis werden wahrscheinlich trotzdem zwei Kontexte notwendig, da der benötigte Bereich von zwei Kontexten abgedeckt wird. Da beim Start der ersten Multiplikation noch nicht klar ist, ob das Skalarprodukt (falls es nur zwei Kontexte benötigt) noch den Kontext oberhalb oder den unterhalb des ersten (in den das erste Produkt fällt) benötigt, wird es am besten sein, wenn gleich 3 Kontexte für ein IEEE 754 single precision Skalarprodukt reserviert werden (s. Abbildung 2.11).

Der Vorteil liegt dabei in der Reduzierung der notwendigen Kommunikation zwischen der Summationsmatrix und dem Betriebssystem (bzw. dem Treiber). Das Betriebssystem kann automatisch 3 Kontexte für dieses Skalarprodukt reservieren und die Hardware plaziert das erste Produkt immer im „mittleren“ dieser Kontexte. Eigentlich muß sich die Hardware nur die Reihenfolge, in der sie die Produkte in diese Kontexte rein addiert, merken, so daß beim späteren Auslesen diese Kontexte ihrer Wertigkeit entsprechend behandelt werden.

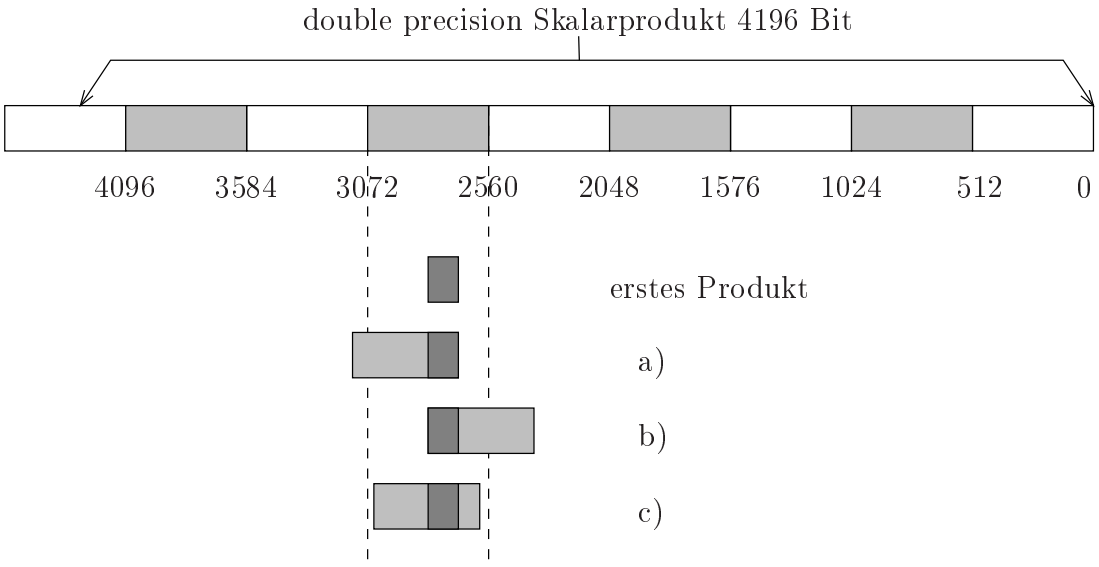


Abbildung 2.11: Effektive Breite $\leq$ 512 Bit, 3 mögliche Kontexte

2.6.2 Dynamische Fenster

In Abbildung 2.11 erkennt man, daß es in diesem Beispiel möglich ist, das ganze Skalarprodukt in einen Kontext abzuspeichern (s. 2.11 c), falls zu Beginn der Akkumulation Informationen über die Lage des Kontextes vorhanden sind. Eine denkbare Strategie wäre die Annahme, daß das erste Produkt immer in der Mitte des benötigten Kontextes liegt. Eine andere Möglichkeit ist es, dem „Benutzer“ die Möglichkeit zu geben, die Fensterlage zu bestimmen. Dabei wollen wir in diesem Falle unter Benutzer sowohl den Programmierer als auch den Compiler oder Treiber verstehen. Diese beiden Möglichkeiten wollen wir als das **dynamische** Fenster bezeichnen um die variablen Exponentenbereiche der Kontexte zu betonen, im Gegensatz zu den starren Fenstern. Sollte der Exponentenbereich vom Benutzer (im obigen Sinne) bestimmt werden und wir wollen diesen Aspekt besonders betonen, so sprechen wir nicht von dynamischen sondern von **gesteuerten** Fenstern.

Um das Konzept der dynamischen Fenster zu verwirklichen, sind mehrere Erweiterungen des Konzeptes notwendig, die wir an der Summationsmatrix betrachten wollen. Notwendige Erweiterungen für Fenster:

1. Anpassung der Multiplikation (und Matrixzeilenbreite, Addiererlänge)
2. Möglichkeit zwei Kontexte versetzt zu addieren

Diese Punkte wollen wir im weiteren Verlauf noch ausführlich diskutieren. Die zusätzlichen Erweiterungen, die für dynamische Fenster notwendig sind, werden wir dabei nicht ausführlich diskutieren, da sie je nach Umsetzung in Hardware oder auch in Software (Betriebssystem, Treiber) denkbar sind. Für dynamische Fenster werden zusätzlich noch benötigt:

1. Erweiterung des Kontextes um ein Bereichsregister
2. Steuerung der Shiftweite in Abhängigkeit des Bereichsregisters

3. Signalisierung falls die reservierten 3 Kontexte nicht ausreichen
4. Markierung der zusammengehörigen Kontexte

Bereichsregister

Damit beim Addieren und Auslesen nachvollziehbar ist, welcher Bereich des gesamten (IEEE 754 double precision) Skalarproduktes in einem Kontextspeicher ist, muß dies mit zum Kontext gehörend abgespeichert werden. Dabei ist es nebensächlich von welchem der Bits des Kontextes der Exponent abgespeichert wird, solange es eindeutig ist.

Shiftweite

Damit das Produkt richtig zurecht geschoben wird, muss der Shifter in Abhängigkeit des Bereichsregisters noch einen zum Produktexponenten zusätzlichen Wert erhalten, damit die entsprechenden Produktbits auch in die entsprechenden Kontextbits akkumuliert werden. Die Shiftweite wird also vom Produktexponenten und vom Inhalt des Bereichsregisters bestimmt. Der Produktexponent bestimmt außerdem noch in welchen Kontext das Produkt addiert wird.

Signalisierung

Sollte das Produkt oder ein Teil davon, außerhalb der drei reservierten Kontexte liegen, so muß die Hardware der Verwaltungseinheit für die Kontexte (in unserer Betrachtung dem Betriebssystem oder dem Treiber) die Notwendigkeit eines weiteren Kontextes mitteilen. Nachdem diese Einheit einen weiteren Kontext bereitgestellt hat, kann das Produkt akkumuliert werden.

Markierung

Da ein IEEE 754 double precision Skalarprodukt in einer IEEE 754 single precision Einheit mehrere Kontexte belegen kann, müssen diese als zusammengehörig markiert werden. Die Wertigkeit der einzelnen Kontexte ergibt sich aus ihren jeweiligen Bereichsregistern.

2.7 Multiplikation bei der Fensterlösung

Bei der Fenstertechnik wird das Errechnen eines exakten IEEE 754 double precision Skalarproduktes in einer Hardware für IEEE 754 single precision durchgeführt. Der größte Unterschied zwischen diesen beiden Datenformaten ist die Mantissenlänge und der Exponentenbereich, woraus auch die unterschiedliche Länge des Akkumulators für das exakte Skalarprodukt resultiert. Bei den Exponenten sind anstatt der 8 Bit einer IEEE 754 single precision Zahl bei IEEE 754 double precision 11 Bit vorhanden (siehe 1.4). Aus diesem Grund wird anstatt der Addition zweier 8 Bit Zahlen die Addition zweier 11 Bit Zahlen notwendig. Dies erhöht den Hardwareaufwand nicht wesentlich. Viel dramatischer ist, daß ein Produkt zweier IEEE 754 double precision Mantissen 106 Bit breit ist, gegenüber den 48 Bit eines IEEE 754

single precision Produktes. Dadurch wird der Shiftereingang von 48 Bit auf 106 Bit Breite angehoben. Solange der Ausgang des Shifters nicht größer wird, verändert sich daher nicht allzu viel am Hardwareaufwand für den Shifter. Ganz anders verhält es sich mit der Größe des Multiplizierers. Der ohnehin schon aufwendige 24×24 Multiplizierer, müßte durch einen 53×53 Multiplizierer, mit annähernd vierfachem Hardware-Aufwand, ersetzt werden.

Außer dem immensen Hardwarebedarf des Multiplizierers ist auch eine Umorganisation der Matrixzeile notwendig. Wie wir in 2.3.1 gesehen hatten, ist es für ein IEEE 754 single precision Produkt ausreichend, wenn wir zwei 64 Bit breite Addierer pro Matrixzeile verwenden. Der Vorteil dieser Addierer war, daß wir nicht allzu viele Überträge haben, die wir entweder durch Nachtakten (siehe 2.4.1) oder durch die Carry-Select Technik (siehe 2.4.2) auflösen müssen. Wenn wir von der gleichen Matrixzeilenbreite von 128 Bit ausgehen, so müssen wir aber nach der Tabelle in 2.3.1 mindestens 8 einzelne Addierer pro Matrixzeile verwenden, falls wir Addierer mit Breiten die durch Zweierpotenzen darstellbar sind verwenden möchten. Natürlich könnten wir auch nur 6 Addierer wählen deren Breiten kleiner als 24 Bit sind und zusammen eine Länge von 128 Bit erreichen (zum Beispiel 4×21 und 2×22 Bits). Der Nachteil einer solchen Aufteilung liegt darin, daß solche Addierer eine aufwendigere Auswahllogik benötigen, da ihre Breite keine Zweierpotenz ist.

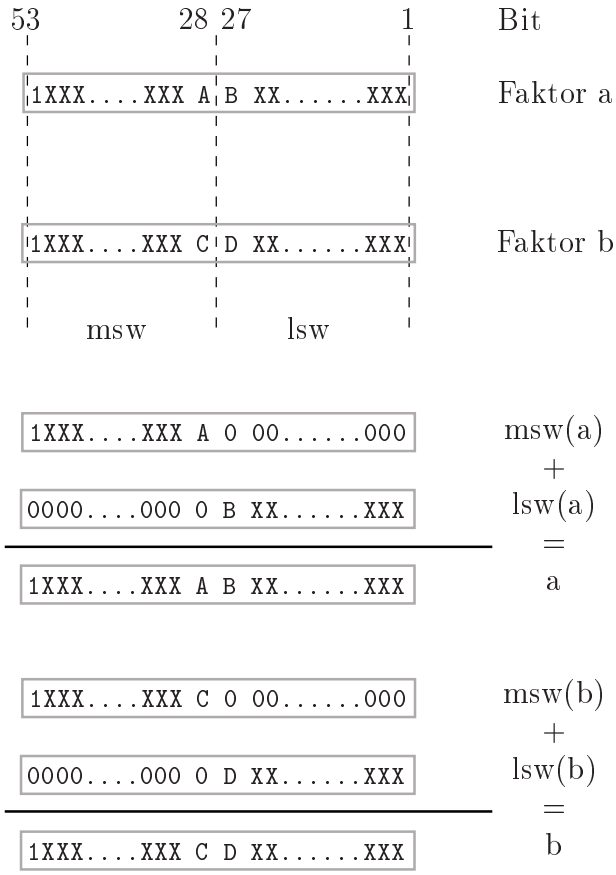


Abbildung 2.12: IEEE 754 double precision Faktoren

Wie wir sehen, ist also der Multiplizierer die Komponente, die von der Fenster-

technik am meisten betroffen wird. Wir dürfen bei dieser Diskussion nicht aus den Augen verlieren, daß wir in einer IEEE 754 single precision Einheit auch IEEE 754 double precision Skalarprodukte berechnen wollen. Der Schwerpunkt einer solchen Einheit sind also die single precision Skalarprodukte. Daher ist eine Vervierfachung des Hardwareaufwandes für den Multiplizierer nicht vertretbar. Vor allen Dingen auch deshalb nicht, da der Multiplizierer auch schon in seiner IEEE 754 single precision Variante circa ein Zehntel des gesamten Hardwarebedarfes benötigt.

2.7.1 Partialprodukte

Aufgrund des immensen Platzbedarfes des Multiplizierers werden wir bei der Multiplikation auf die sequentielle Multiplikation zurückgreifen. Diese benötigt zwar mehr Takte, dafür aber auch einen kleineren Multiplizierer (siehe [28]). Bei unserer Problemstellung wäre eine Aufteilung der 53×53 Bit Multiplikation in vier Partialprodukte sinnvoll. Dabei werden zwar vier Takte für eine IEEE 754 double precision Multiplikation benötigt, dafür aber auch nur ein „etwas“ größerer Multiplizierer. Dieser 27×27 Multiplizierer benötigt circa 25 % mehr Resourcen (Fläche) als der 24×24 Multiplizierer.

Die Aufteilung in die vier Partialprodukte geschieht durch die Zerlegung der Faktoren a und b in ihr 26 Bit langes oberes Wort (msw: **m**ost **s**ignificant **w**ord) und ihr 27 Bit langes unteres Wort (lsw: **l**east **s**ignificant **w**ord) wie in Abbildung 2.12 dargestellt.

$$\begin{aligned} a \times b &= (msw(a) + lsw(a)) \times (msw(b) + lsw(b)) \\ &= msw(a) \times lsw(b) + lsw(a) \times msw(b) \\ & \quad msw(a) \times msw(b) + lsw(a) \times lsw(b) \end{aligned}$$

Operation				Ergebnis	#vN	#hN
msw(a)	×	msw(b)	=	XXX...XXX...XXX000...000...000	0	54
msw(a)	×	lsw(b)	=	000...000XXX...XXX000000...000	26	27
lsw(a)	×	msw(b)	=	000...000XXX...XXX000000...000	26	27
lsw(a)	×	lsw(b)	=	000...000...000XXX...XXX...XXX	52	0

#vN: Anzahl **v**orderer **N**ullen
#hN: Anzahl **h**interer **N**ullen

Wenn jetzt noch die führenden Nullen bei lsw(a) und lsw(b) und die hinteren Nullen bei msw(a) und msw(b) gelöscht werden, haben wir vier 27×27 Multiplikationen. Eigentlich sind es eine 27×27, eine 26×26, eine 26×27 und eine 27×26 Multiplikation. Der Einfachheit halber erweitern wir die 26 Bit Worte um ein führendes Nullbit, so daß wir nur 27×27 Bit Multiplikationen betrachten müssen.

Diese Produkte müßten nun mit den hinteren Nullbits rechtsbündig addiert werden, um das Produkt von $a \times b$ zu erhalten. Diese Addition verlegen wir in die Summationsmatrix ($a \times b$ soll ja akkumuliert werden) und die Anzahl der hinteren Nullbits (#hN in der obigen Tabelle) der einzelnen Produkte werden zum Exponenten des Partialproduktes dazu addiert, also als zusätzliche Shiftweite benutzt. Damit haben wir mit einem 27×27 Bit großen Multiplizierer in vier Takten ein

IEEE 754 double precision Produkt akkumuliert, ohne den Hardwareaufwand zu sehr zu vergrößern (wir haben einen um circa 25% grösseren Multiplizierer). Ein Nachteil zeigt sich bei der Erhöhung des (Partialprodukt-)Exponenten durch die hinteren Nullen. Sowohl 27 als auch 54 sind keine Zweierpotenzen und daher benötigen wir eine zusätzliche Addition um die wirkliche Shiftweite zu ermitteln. Dieses Problem könnten wir dadurch entschärfen, daß wir bei der Zerlegung der Faktoren für das lsw die letzten 32 Bits verwenden, so daß das msw nur noch 21 Bit groß ist. Dadurch erhalten wir bei der Exponentenerhöhung der Partialprodukte die Werte 32 bzw. 64 (anstatt 27 bzw. 54) was wiederum Zweierpotenzen wären und daher leichter zu addieren sind. Bei dieser Aufteilung benötigen wir aber einen 32×32 Bit Multiplizierer was den Hardwareaufwand gegenüber dem ursprünglichen 24×24 Bit Multiplizierer um circa 90% erhöht. Desweiteren können wir auch nicht mehr nur 4,5 Zeilen (= 576 Bit) in der Summationsmatrix verwenden, da wir sonst nur $576 - 512 - (64 - 1) = 1$ Schutzziffer zur Verfügung hätten. Zahl 63 $(= 64 - 1)$ resultiert aus der Möglichkeit, daß das Partialprodukt gerade noch in einen Kontext akkumuliert wird. Aus diesem Grund benötigen wir dann auch fünf volle Zeilen, also zehn anstatt nur neun 64 Bit Addierer. Dies erhöht den Hardwareaufwand wieder zusätzlich. Auch bei einem 27×27 Multiplizierer ist dieser Schritt zu überlegen, da wir dann auch nur 11 Schutzziffern hätten.

2.8 Zwischenregister

Jetzt ergibt sich das Problem aus mehreren Kontexten den wirklichen Wert des IEEE 754 double precision Produktes zu berechnen. Die einzelnen Kontexte können sich durch Überträge beeinflußen. Es ist auch möglich, daß die Kontexte unterschiedliche Vorzeichen haben. Deswegen müssen diese Überträge vom niederwertigsten Kontext aufwärts aufgelöst werden. Betrachten wir einmal zwei aufeinander folgende Kontexte genauer. Dabei wollen wir von einer Summationsmatrix mit einer Matrixzeilenlänge von 128 Bit, bestehend aus zwei 64 Bit Addierern ausgehen. Bei der Unterteilung des gesamten IEEE 754 double precision Skalarprodukte in IEEE 754 single precision Kontexte gehen wir von den 512 Bit Schritten, wie in Abbildung 2.9 vorgestellt, aus. Das heißt, daß sobald der Exponent eines Partialproduktes die 512 Bit Grenze überschreitet wird das Partialprodukt in den nächsten Kontext akkumuliert. Nehmen wir an, daß ein 54 Bit breites Partialprodukt am vorletzten Bit unseres niederwertigen Kontextes beginnt. Dann ragen die 53 oberen Bits dieses Partialproduktes über die 512 Bitgrenze und müssen am Ende zum darüberliegenden Kontext addiert werden. Das Problem ist nur, die verschiedenen Kontexte sind an unterschiedlichen Addressen des Kontextspeichers.

2.8.1 Fensterauflösung

Bei der angenommenen Architektur der Summationsmatrix (Breite 128 bit, 4,5 Zeilen) ist dieses Problem durch ein 128 Bit Register, das am Summationsmatrixdatenbus angeschlossen ist, lösbar. Zuerst wird die obere Zeile des unteren Kontextes (die nur halb besetzte Zeile) in das Register geladen (die fehlenden 64 Bit werden mit Nullbits aufgefüllt) wie in Abbildung 2.14 a) dargestellt. Danach wird der Registerinhalt zur niederwertigsten Zeile des oberen Kontextes dazu addiert bzw. subtrahiert

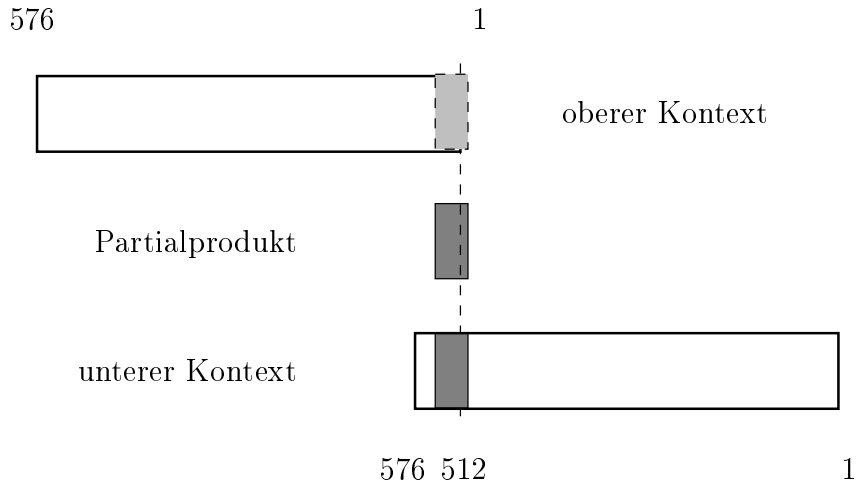


Abbildung 2.13: Überschneidung zweier Kontexte

(s. Abbildung 2.14 b)) und schon beinhaltet der obere Kontext alle notwendigen Überträge.

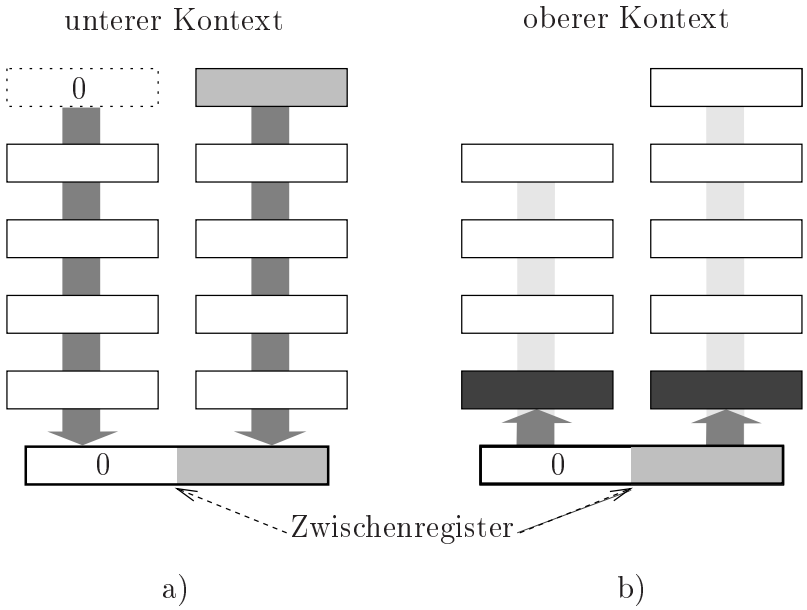


Abbildung 2.14: Überschneidung auflösen mit Hilfe des Zwischenregisters

2.8.2 Skalarprodukt-Arithmetik

Alle Matrixzeilen haben Zugriff auf die Eingangsdaten, die vom Shifter geliefert werden, damit sie die Produkte einlesen können. Außerdem ist es allen Matrixzeilen (bzw. Teiladdierern) möglich ihren Inhalt auszugeben. Wenn nun das im vorherigen Abschnitt angesprochene Zwischenregister an diesen (Eingang/Ausgang) Schnittstellen liegt, haben wir die Möglichkeit, mehrere Kontexte zu addieren.

Das bedeutet im Falle eines *sp*-Skalarproduktes, daß wir zwei exakte Skalarprodukte addieren bzw. subtrahieren können. Im Falle von *dp*-Skalarprodukten ist dies nur dann ohne zusätzliche Shiftstufe möglich, wenn wir die starre Fensterlösung gewählt haben. Im Falle von dynamischen Fenstern müssen die Kontexte erst entsprechend zurecht geshiftet und gegebenenfalls auch noch erweitert werden. Diese Addition zweier Kontexte würde in unserer Beispielmatrix (128 Bit Zeilenbreite, 4,5 = 5 Zeilen) 15 Takte benötigen.

$$\begin{aligned} Kontext_a + Kontext_b &= Kontext_c \\ Zeile1_a + Zeile1_b &= Zeile1_c \\ Zeile2_a + Zeile2_b &= Zeile2_c \\ Zeile3_a + Zeile3_b &= Zeile3_c \\ Zeile4_a + Zeile4_b &= Zeile4_c \\ Zeile5_a + Zeile5_b &= Zeile5_c \end{aligned} \tag{2.3}$$

Die ersten 5 Takte werden dabei zum Kopieren von $Kontext_a$ nach $Kontext_c$ benötigt. Anschließend wird die $Zeile1_b$ zur $Zeile1_c$ (= $Zeile1_a$ aufgrund des Kopierens) addiert. Dazu wird zuerst die $Zeile1_b$ in das Zwischenregister geladen (1 Takt) und danach zur $Zeile1_c$ addiert (1 Takt). Also benötigt jede der 5 Addition in Gleichung 2.3 zwei Takte. Wichtig dabei ist, daß $Zeile1$ die niederwertigste Zeile ist, auf die dann $Zeile2$ folgt, darauf folgt $Zeile3$, und so weiter.

Zu Bedenken dabei ist, daß zwischen den einzelnen Zeilen Überträge auftreten können. Diese müssen natürlich an die entsprechende Zeile geleitet werden, bevor die Addition dieser Zeile startet. Damit kann dann der Übertrag mit der Addition zusammen berücksichtigt werden.

2.9 Anwendungen

Wir wollen in diesem Abschnitt noch einmal die Verwendungsmöglichkeiten einer Summationsmatrix für IEEE 754 single precision Zahlen aufzeigen. Dabei gehen wir von einer Summationsmatrix mit 10 Addierern der Breite 64 Bit, die in 5 Zeilen der Breite 128 Bit organisiert sind, aus. Desweiteren wollen wir annehmen, daß wir 8 Kontexte (s. 2.5) und ein wie in Abschnitt 2.8 beschriebenes 128 Bit großes Zwischenregister haben. Der Multiplizierer sei 27×27 Bit oder 32×32 Bit groß.

Folgende Anwendungsmöglichkeiten ergeben sich für eine solche Hardware:

1. Berechnung von bis zu 8 beliebig großen, exakten Skalarprodukten für IEEE 754 single precision Zahlen, pro Takt eine Akkumulation

2. Berechnung eines beliebig großen, exakten Skalarproduktes für IEEE 754 double precision Zahlen, 4 Takte für eine Akkumulation; mit starren Fenstern

3. Addition und Multiplikation von Staggered Zahlen sowohl des Grunddatentyps *sp* als auch *dp*; die Anzahl der benötigten Takte ist von der Staggered-Länge l abhängig (siehe 1.5.1)

4. Addition zweier exakter *sp* (oder *dp*) Skalarprodukte

Kapitel

3

Verwendete Technologien

“ Time to get up, Lucky Eddie. Remember, the early bird gets the worm!
But can I have bacon and eggs instead? ”

Dik Browne: Hagar The Horrible

In diesem Kapitel wollen wir die notwendigen Werkzeuge und Technologien vorstellen, die benötigt werden, um die Umsetzung der Summationsmatrix in Hardware zu realisieren. Dabei werden wir kurz die verwendete Hardwarebeschreibungssprache VHDL und die dazu benötigten Programme vorstellen. Ein weiterer wichtiger Punkt ist die Vorstellung der verwendeten Hardware-Testumgebung in Form der Spyder-Virtex-X2 PCI-Karte und des daraufbefindlichen FPGAs Xilinx Virtex XCV800-4.

3.1 HDL

Eine Hardwarebeschreibungssprache (*HDL: hardware description language*) dient dazu, wie ihr Name schon sagt, um eine Hardware (bestehende oder zu entwickelnde) zu beschreiben. Die Hardware kann auf verschiedenen Abstraktionsebenen beschrieben werden. So kann bei einer Verhaltensbeschreibung für die Multiplikation zweier Zahlen oder Bitvektoren einfach ein $*$ geschrieben werden, wohin gegen bei einer RTL-Beschreibung (*register transfer level*) der Multiplizier zum Beispiel als

Wallace-Baum mit allen Addierern und Verbindungen ausformuliert wird. Diese Beschreibung wird dann von einem Compiler in eine sogenannte Netzliste überführt, die die Hardware beschreibt. Im Falle der Verhaltensbeschreibung wird bei unserem Beispiel die Wahl des Multiplizierer-Designs dem HDL-Compiler überlassen. Was für Hardwarebeschreibungssprachen sind üblich?

3.1.1 Schema

Das Schema ist eine der einfachen und nur für kleine Schaltungen praktizierbare Eingabe der Hardwarebeschreibung. Dabei werden die Schaltungen aus den vorhandenen Grundelementen zusammengestellt, in dem die Grundelemente entsprechend verbunden werden. In der Abbildung 3.1 sehen wir ein UND-Gatter mit vier Eingängen das aus 3 einfacheren UND-Gattern mit jeweils zwei Eingängen gebildet wurde. Diese Art der Eingabe ist sehr anschaulich, doch wird sie bei größeren Entwürfen schnell unübersichtlich.

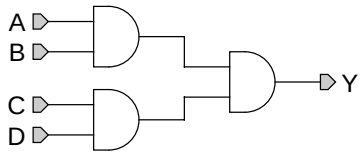


Abbildung 3.1: 4 Eingänge-UND-Gatter als Schema

Aus diesem Grund führten die Halbleiterhersteller und EDA-Werkzeughersteller Hardwarebeschreibungssprachen ein. Mit einer solchen Sprache können die gleichen (und mehr) Schaltungen beschrieben werden wie durch die Schema-Eingabe. Leider hatte jeder Hersteller seine eigene Sprache entwickelt, manchmal sogar für jede Bausteinfamilie oder Technologie eine eigene. Bis Ende der 80er Jahre setzten sich dann hauptsächlich zwei Sprachen durch: Verilog und VHDL.

3.1.2 Verilog

Verilog wurde 1983 von der Firma Gateway Design Automation eingeführt. 1989 kaufte die Firma Cadence die Firma Gateway und damit auch Verilog. Seitdem hat sich diese an die Programmiersprache C angelehnte Sprache, vor allem im Bereich des ASIC-Entwurfs einen Platz erobert. Im Dezember 1995 wurde Verilog als IEEE 1364 genormt (siehe [28], [30]). Unser Beispiel des vierfach UND-Gatters kann in Verilog als Beschreibung einer UND-Verknüpfung aller vier Eingänge vorgenommen werden.

```
Y = A & B & C & D;
```

Dabei wird dem Entwurfswerkzeug die Abbildung auf ein 4-fach UND-Gatter (falls in der Zieltechnologie vorhanden) oder die Bildung durch drei UND-Gatter wie in Abbildung 3.1 überlassen. Es ist aber auch möglich den Aufbau durch drei UND-Gatter anzugeben.

```
UND2(F,A,B);
```

```
UND2(G,C,D);
UND2(Y,F,G);
```

Dabei wurde dem Entwurfswerkzeug die Entscheidung wie dieses 4-fach UND-Gatter aufzubauen ist abgenommen, aber auch die Möglichkeit ein eventuell in der Zieltechnologie vorhandenes 4-fach UND-Gatter zu verwenden (dieses einfache Beispiel hindert heutige Compiler nicht, trotzdem durch Optimierung ein 4-fach UND-Gatter einzusetzen.) Verilog ist in den USA sehr verbreitet. In Europa wird zur Zeit Verilog zu circa 40% eingesetzt. Die anderen 60% der Hardwarebeschreibungen werden in VHDL ausgeführt.

3.1.3 VHDL

Da wir für die Realisierung unseres Entwurfes VHDL gewählt hatten, wollen wir diese Sprache hier etwas ausführlicher vorstellen. Für eine richtige Einführung in die Sprache sei auf das Buch „The Designer's Guide to VHDL“ [4] oder auf das Buch „Schaltungsdesign mit VHDL“ [24] verwiesen. Die Bücher [9], [8], [6] und [7] sind zur Vertiefung und für spezielle Fragen zu empfehlen. In dem Buch „HDL Chip Design“ [28] sind Vergleiche zwischen VHDL und Verilog zu finden.

VHDL wurde 1980 in einem Projekt des amerikanischen Verteidigungsministerium (*DoD: Department of Defense*) entwickelt. 1987 wurde VHDL als IEEE 1076 genormt und ist seitdem weit verbreitet. VHDL ist eine universelle Programmiersprache, die für den Entwurf von digitalen Schaltungen (neuere Bestrebungen versuchen VHDL auch für analoge Schaltungen zu erweitern) optimiert ist. VHDL ist an die Programmiersprache Ada (ebenfalls Anfang der 80er Jahre vom DoD unterstützt) angelehnt. Deswegen kann VHDL benutzt werden zur:

- Entwurfeingabe
- Spezifikation
- Simulation
- Synthese

von digitalen Schaltungen. Da VHDL auch die abstrakte Beschreibung von Schaltungen erlaubt kann es sehr gut zur Spezifikation verwendet werden. VHDL unterstützt das modulare Beschreiben durch die Möglichkeit der Kapselung einer Einheit. Dabei wird eine Schnittstelle (*interface*) dieser Einheit definiert. Zu dieser Schnittstelle kann dann die Realisierung der Funktionen der Einheit auf unterschiedliche Weise erfolgen, ohne daß sich die Schnittstelle ändert. Zum Beispiel kann auf einer hohen Abstraktionsebene die Multiplikation nur als $*$ ausgeführt sein, was für die Simulation einer gesamten Schaltung vollständig ausreicht. Soll diese Schaltung dann synthetisiert, also in Hardware umgesetzt werden, so kann diese Einheit durch einen fertigen Hardware-Multiplizierer ersetzt werden oder die Hardwarebeschreibung zum Beispiel als Wallace-Baum erfolgen. Die Simulation der gesamten Schaltung mit einem Multiplizierer, der explizit als Wallace-Baum ausgeführt wurde, erfordert viel mehr Rechenaufwand vom Simulator, da er jetzt nicht nur die Multiplikation zweier Zahlen ausführen muss. Jetzt muß er jeden einzelnen Addierer

des Wallace-trees simulieren. Deswegen werden für die Simulation von Schaltungen nicht alle Einheiten in ihrer Hardware-Realisierung eingesetzt, sondern es werden abstraktere VHDL-Beschreibungen (die nicht synthetisierbar sein müssen) mit gleichem Verhalten verwendet. So muss nur die gerade getestete Komponente in ihrer Hardware-Realisierung simuliert werden.

VHDL unterstützt nicht nur Modularisierung eines Entwurfes, sondern kann auch für nebenläufig (*concurrent*) ablaufende Schaltungen verwendet werden. Prinzipiell sind in VHDL alle Anweisungen nebenläufig. Um sequentielle Anweisungen anzugeben müssen die gewünschten Anweisungen in einem `process` zusammengefasst werden.

Wir wollen hier noch einmal auf unser Beispiel des vierfach UND-Gatters zurückkommen. In VHDL ist es natürlich auch möglich alle vier Eingänge über eine UND-Funktion zu verknüpfen.

```
Y <= A AND B AND C AND D;
```

Dabei bedeutet `<=` eine Zuweisung für Signale. Signale sind Elemente in VHDL (im Gegensatz zu Variablen), die eine Entsprechung in Hardware haben. Zum Beispiel werden die Verbindungsleitungen im nächsten Beispiel durch Signale angegeben.

Wenn wir annehmen, daß uns als Grundfunktion ein UND-Gatter mit zwei Eingängen zur Verfügung steht, so kann die Schaltung aus Abbildung 3.1 auch in VHDL beschrieben werden. Um auch die vorher angesprochene Modularisierbarkeit zu demonstrieren, werden wir auch die Vereinbarung dieses UND-Gatters mit zwei Eingängen vorstellen.

```
ENTITY und2 IS
PORT (i1, i2 : IN  STD_LOGIC;
      x      : OUT STD_LOGIC);
END und2;
ARCHITECTURE beispiel_und2 OF und2 IS
BEGIN
  x <= i1 AND i2;
END beispiel_und2;
```

Mit Hilfe dieser Vereinbarung werden wir jetzt drei dieser UND-Gatter instanziieren und miteinander verbinden. Um die Beispiele einfach und kurz zu halten, geben wir keine fertigen VHDL-Programme, sondern nur die daraus relevanten Teile an. Beim folgenden Beispiel verwenden wir auch das in VHDL verwendete Kommentarzeichen `--` um Erläuterungen zu geben. Sobald in VHDL innerhalb einer Zeile das Kommentarzeichen erscheint, ist der Rest der Zeile keine VHDL-Anweisung mehr.

```
...
ARCHITECTURE beispiel_und4 OF und4 IS
  SIGNAL draht1, draht2 : STD_LOGIC;
BEGIN
  und2(A, B, draht1);                 -- Instanziierung des
                                      -- ersten UND-Gatters
  und2(x => draht2, i1 => C, i2 => D) -- Instanziierung des
                                      -- zweiten UND-Gatters
```

```
                                    -- explizite Zuweisung
  und2(draht1, draht2, Y);          -- Instanziierung des
                                    -- dritten UND-Gatters
END beispiel_und4;
```

Bei der Instanziierung des ersten und dritten UND-Gatters wurden die Verbindungen der Signale zu den Eingängen und Ausgängen implizit, aufgrund der Reihenfolge der aufgeführten Signale durchgeführt. Im Falle des zweiten UND-Gatters wurde eine explizite Zuweisung, die nicht notwendig in der definierten Reihenfolge erfolgen muss, vorgenommen.

3.2 ASIC

Die mit einer Hardwarebeschreibungssprache beschriebene Schaltung kann dann in Hardware umgesetzt werden. Dies geschieht meist in einem sogenannten ASIC (*application specific integrated circuit*). Bei einem ASIC wird eine Schaltung nicht aus Standardbausteinen wie zum Beispiel aus der 74xx-TTL-Reihe aufgebaut, sondern die Schaltung wird selbst auf einem Chip, also in Silizium realisiert. Dabei werden verschiedene Ansätze unterschieden, die sich in ihrer Komplexität und damit auch im Preis (gemessen in Personen-Jahren) unterscheiden.

3.2.1 Full-Custom

Beim Full-Custom-Entwurf hat die Entwicklerin alle Freiheitsgrade. Das geht soweit, daß sie die Struktur des Transistors auf dem Silizium bestimmen kann, bzw. muß. Dadurch können erfahrene Entwickler alle notwendigen Optimierungen und Besonderheiten einer Schaltung auf unterster Ebene beeinflussen. Dieser Vorteil ist aber auch gleichzeitig ein Nachteil. Die Entwicklerin muss genau wissen was sie macht, um eine funktionierende Schaltung bzw. das Optimum zu erhalten. Beim Full-Custom-Entwurf müssen alle Schritte wie z.B. die einzelnen Belichtungsmasken für die Chipherstellung extra für diesen einen Entwurf hergestellt werden. Da die Prozesse in diesen einzelnen Schritte sehr zeit- und kostenaufwendig sind, lohnt sich ein Full-Custom-Entwurf nur für Chips mit sehr hoher Stückzahl.

3.2.2 Semi-Custom

Um die Nachteile des Full-Custom-Entwurfes zu vermeiden, wurden Semi-Custom-Techniken entwickelt. Bei diesen Techniken werden einige der „Nachteile“ des Full-Custom-Entwurfes dadurch reduziert, daß dem Entwickler nicht mehr alle Freiheiten zur Verfügung stehen. Teile des Entwurfes werden ihm vorgegeben, so daß er sich weder um die Funktion noch um den Test dieser Teile kümmern muss. Prinzipiell werden zwei Arten des Semi-Custom-Entwurfes unterschieden.

Standardzellen

Da ist als erstes der Entwurf mit Standardzellen. Bei dieser Technik verwendet der Entwickler Zellen die vom Halbleiterhersteller optimiert und getestet wurden. Der Entwickler ist dann ähnlich wie beim Aufbau einer Platine für die Auswahl,

Platzierung und Verbindung der einzelnen Komponenten (Standardzellen) zu einer Schaltung verantwortlich. Auch bei dieser Technik müssen für jeden Schritt die Belichtungsmasken hergestellt werden, der einzige Unterschied ist, daß die Zellen schon getestet und optimiert und in einer Bibliothek vorhanden sind. Ein Entwurf mit Standardzellen kann jederzeit in kritischen Teilen zu einem „Full-Custom-Entwurf" werden, sobald extra für diese Schaltung neue Zellen entwickelt werden.

MPGA / Sea-of-Gates

Bei einem maskenprogrammierten Gate Array (*MPGA*) werden die Siliziumscheiben mit einer regelmäßigen Struktur von Transistoren, Logikgattern (*gates*) oder Logikblöcken vorproduziert. Zwischen diesen Elementen befinden sich freie Flächen für die Verdrahtung. Die Sea-of-Gates Technologie ist eine Abwandlung dieses Aufbaues. Dabei fehlen die freien Flächen und die Verdrahtung wird ausschließlich durch die Verschaltung der „Gatter" in darüber liegenden Verdrahtungsebenen ausgeführt. Da bei dieser Technologie nicht mehr die einzelnen Transistoren sondern „nur noch" verhältnismäßig einfache Verbindungen (Verdrahtungen) individuell ausgeführt werden, ist sie nicht so zeit- und kostenintensiv wie die Full-Custom-Technik. Wenn jetzt auch schon die Verdrahtungen vorbereitet werden, so daß die Verbindungsstellen nur noch „programmiert" werden müssen, erhalten wir FPGAs.

3.3 FPGA

Die Abkürzung FPGA steht für **F**ield **P**rogrammable **G**ate **A**rray. Dabei bedeutet „Feld programmierbar", daß im Gegensatz zu den Gate Arrays bei den ASICs (3.2.2) diese außerhalb eines Reinraumes (im Feld) programmierbar sind. Im Wesentlichen besteht ein FPGA aus einer regelmäßigen Struktur (Array) programmierbarer Logik-Blöcke. In diesen Logik-Blöcken werden die gewünschten Funktionen realisiert. Durch das Verschalten dieser Logikblöcke wird die gesamte Schaltung erstellt. Dieses Verschalten geschieht nicht wie bei den Gate Arrays beim Halbleiterhersteller sondern sie werden vom Anwender programmiert. Dafür sind im FPGA Leitungen vorhanden, die durch Programmierung mit den Ein- und Ausgängen bestimmter Logikblöcke und mit anderen Verbindungsleitungen verknüpfbar sind. Meist sind auch die einzelnen Logikblöcke selbst verschieden konfigurierbar, so daß zum Beispiel ein Block als RAM oder ROM arbeiten kann. Die einzelnen FPGAs unterscheiden sich sowohl in der Menge und Funktion der einzelnen Logikblöcke als auch in der Art, wie die Konfigurationsdaten (das „Programm" des FPGAs) gespeichert werden. FPGAs erlauben in den einzelnen Blöcken meist nur eine Logikfunktion mit wenigen Eingängen (4-10). Komplexere Funktionen werden aus mehreren solchen Grundfunktionen zusammengebaut. Da bei vielen Schaltungen auch Speicherelemente (Register, Flip-flops) notwendig sind, besitzt solch ein Logikblöck meist auch noch wenige Speicherelemente um die erzeugten bzw. erhaltenen Daten zu speichern. Dabei sind jetzt **nicht** die Konfigurationsdaten gemeint die das Verhalten des Logikblockes bestimmen, sondern die Eingänge der mit diesem Block realisierten Logikfunktion. Im folgenden wollen wir ganz kurz ein paar prinzipielle Unterschiede der FPGAs kennenlernen für weitere Informationen sei zum Beispiel auf *Das FPGA-Kochbuch* [30] verwiesen.

3.3.1 LUT

Ein Konzept beim internen Aufbau eines Logikblockes ist die sogenannte **L**ook **U**p **T**able. Dabei stellen die Eingangssignale im Grunde die Adressleitungen eines kleinen 1-Bit-Speichers dar, der Inhalt der so adressierten Speicherzelle ist das Ausgangssignal der Logikfunktion. Als Beispiel nehmen wir eine LUT mit vier Eingängen. Damit wollen wir eine einfache Logik(grund)funktion ein UND-Gatter mit 2 Eingängen A und B und dem Ausgang Y realisieren.

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Tabelle 3.1: Wahrheitstabelle UND-Gatter

Wie schon bemerkt, ist unsere LUT ein Speicher mit vier Adresseingängen. Damit können wir sechzehn Speicherzellen adressieren. Da wir nur zwei Eingänge benötigen legen wir zwei der Adressleitungen auf Null. Jetzt können wir nur noch die benötigten 4 Speicherzellen auswählen. Wenn wir in alle Speicherzellen eine Null schreiben, und nur in die durch A=1 und B=1 adressierte eine Eins so haben wir die von uns gewünschte UND-Gatter Funktion (siehe Tabelle 3.2).

Adresse1	Adresse2	Adresse3	Adresse4	Speicherinhalt
0	0	**0**	**0**	**0**
0	0	**0**	**1**	**0**
0	0	**1**	**0**	**0**
0	0	**1**	**1**	**1**
die folgenden Speicherzellen können nicht adressiert werden, da Adresse1 und Adresse2 durch die Konfiguration schon auf 0 gesetzt wurden				
0	1	0	0	X
0	1	0	1	X
...	...	...	...	...
1	1	1	1	X

Tabelle 3.2: UND-Gatter in 4er LUT

Wenn wir jetzt Adresse3 als A, Adresse4 als B und den Ausgang des Speichers als Y interpretieren, so erhalten wir unser UND-Gatter.

3.3.2 Multiplexer

Ein anderer Ansatz Logikfunktionen zu realisieren geschieht mit Multiplexern. Ein einfacher Multiplexer (sogenannter 2:1 Multiplexer) besteht aus zwei Dateneingängen D0 und D1, einem Steuereingang S und einem Ausgang Y. Dabei funktioniert der Multiplexer als Umschalter. Wenn am Steuereingang S eine Null anliegt wird der

Eingang D0 auf den Ausgang Y durchgeschaltet, wenn an S eine Eins anliegt so wird der Eingang D1 auf den Ausgang Y durchgeschaltet. Multiplexerfunktionen werden in vielen Entwürfen benötigt, so daß sie sich als Grundfunktion anbieten. Aber wie kann ein solcher Umschalter unser UND-Gatter realisieren? Wie wir in der folgenden Tabelle sehen, muß einfach der Dateneingang D0 dauerhaft auf Null, S als der Eingang A unseres UND-Gatters, D1 als der Eingang B unseres UND-Gatters interpretiert werden und schon haben wir an Y die gewünschte UND-Funktion von A und B.

D0	D1	S	Y
0	**0**	**0**	**0**
0	**1**	**0**	**0**
1	0	0	1
1	1	0	1
0	**0**	**1**	**0**
0	**1**	**1**	**1**
1	0	1	0
1	1	1	1

Tabelle 3.3: UND-Gatter mit 2:1 Multiplexer

3.3.3 Technologien

Die FPGAs unterscheiden sich aber nicht nur durch ihre Logikblöcke, sondern auch durch ihre Speichertechnologie. Prinzipiell kann dabei erstmal zwischen reversiblen und irreversiblen Technologien unterschieden werden. Bei irreversiblen Technologien wird die Konfiguration des FPGAs einmal in den Baustein „gebrannt“. Danach ist keine Rekonfiguration des Bausteines mehr möglich. Das Programmierelement ist dabei zum Beispiel eine sogenannte *Antifuse*. Dabei wird beim „Brennen“ (meist durch eine höhere Spannung als im normalen Betrieb) eine Isolationsschicht aufgeschmolzen und dadurch eine dauerhafte Verbindung an diesem Punkt hergestellt.

Bei den reversiblen Technologien gibt es die SRAM, EPROM, PROM, Flash und EEPROM Technologie. Da die großen FPGAs heutzutage noch alle in SRAM Technologie ausgeführt sind, wollen wir nur diese hier kurz betrachten. SRAM Zellen sind Speicherzellen, die ihren Inhalt nach Entfernen der Versorgungsspannung wieder verlieren. Deswegen muss jedesmal wenn ein Gerät mit einem solchen FPGA gestartet wird, erst wieder die Konfiguration geladen werden. Wenn es keine Alternative in einer nicht-flüchtigen Speichertechnologie für das FPGA gibt, wird die Konfiguration in einem nichtflüchtigen Speicher gehalten und von einem Mikrocontroller nach dem Anlegen der Spannung geladen.

3.3.4 Virtex

Auf der von uns zum Entwickeln und Testen des Entwurfs benutzten Karte (siehe 3.5.1) befand sich ein FPGA der Firma Xilinx. Aus diesem Grund wollen wir hier diese Baustein Familie näher betrachten. Es gibt natürlich noch andere FPGAs

anderer Firmen mit denen unser Design verwirklichbar ist, doch auf der uns zur Verfügung stehenden Hardware befindet sich ein Xilinx Virtex XCV800. Wie wir im späteren Kapitel Implementierung (siehe 4.1) sehen werden, ist der in dieser Arbeit vorgestellte Entwurf aufgrund seiner VHDL-Beschreibung nicht von einem Xilinx-Baustein abhängig.

Aufbau des FPGA

Bei den FPGAs der Xilinx Virtex Familie handelt es sich um SRAM FPGAs. Die Logikblöcke (*CLB:* **C***onfigurable* **L***ogic* **B***lock*) sind matrixförmig angeordnet, wobei die CLBs einer Spalte durch eine sogenannte „carry-chain“ miteinander verbindbar sind. Dies ermöglicht zum Beispiel die Konstruktion von schnellen Addierern. Neben jedem CLB befindet sich eine Verbindungseinheit (*Switching Matrix*), mit der alle Ein- und Ausgänge des CLBs verbunden sind. Die einzelnen Verbindungseinheiten wiederum sind mit ihren Nachbarn verbunden. Desweiteren hat jede dieser Verbindungseinheiten Zugriff auf mehrere Leitungen, die sich über Teile oder gar über die ganze Matrix erstrecken. Jede Verbindungseinheit kann jeden ihrer Eingänge mit jedem anderen ihrer Eingänge verbinden. Über diese Mechanismen werden die gewünschten Verdrahtungen realisiert. Bei jedem CLB befinden sich außerdem noch zwei Tristate-Buffer die zur Realisierung von internen Bussen oder ähnlichem verwendet werden können. Natürlich existieren auch noch spezielle sich über den ganzen Chip ausdehnende Leitungen die zur Verteilung von Takten angelegt sind. Die Signal-Verzögerungszeiten von Verbindungen zwischen zwei beliebigen Punkten auf dem FPGA hängt davon ab, welche und vor allen Dingen wieviele Verbindungsressourcen benötigt werden. Das heißt die Verzögerungen werden hauptsächlich durch die Anzahl der programmierten Verbindungen bestimmt. Daher hat eine Verbindung von einem Ende des Chips zum anderen bei der Verwendung einer „longline“ (Verbindungsleitung die über den ganzen Chip geht) eine geringere Signalverzögerung als eine Verbindung über 5-8 Verbindungseinheiten. Dies gilt auch dann, wenn die zwei verbundenen Punkte auf dem FPGA dadurch viel enger zusammenliegen. Das bestimmende Maß ist hierbei die Anzahl der konfigurierbaren Verbindungen, die im ersten Fall ca. 4 (vom CLB zur Switching Matrix, zur longline, über die Switching Matrix zum CLB) gegenüber mehr als 12 im zweiten Fall (von der ersten Swichting Matrix, zur zweiten, ... zur achten) beträgt. Genauere Angaben über die FPGAs der Virtex Familien sind zum einen im Datenblatt [32] und allgemein auf der WWW-Seite der Firma Xilinx [31] zu finden. Wir beschränken uns hier auf die Informationen, die wir später zum Interpretieren der Ergebnisse unseres Entwurfes benötigen.

Aufbau eines CLBs

Jedes CLB besteht aus zwei identischen Einheiten den sogenannten Slices die wir im weiteren Verlauf als Logikblock bezeichnen wollen. In Abbildung 3.2 ist erkennbar, daß jeder dieser Logikblöcke aus zwei gleichen Einheiten besteht, die übereinander angeordnet sind. Als erstes ist leicht erkennbar, daß der Logikblock aus einer LUT mit 4 Eingängen besteht. Der Ausgang dieser Look Up Table wird an den Eingang des Speicherelementes geführt. Zwischen LUT und Speicherelement befindet sich

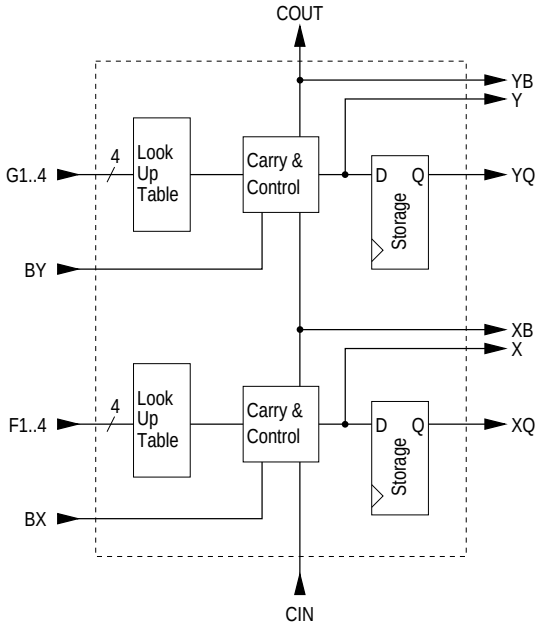


Abbildung 3.2: Slice eines Virtex FPGAs

noch eine Kontrollogik die für die Verwendung der carry-chain oder die Zusammenfassung der beiden Look Up Tables notwendig ist. Die Eingänge BX und BY sind für den Aufbau komplexerer Logikfunktionen bzw. zur Steuerung der carry-chain notwendig. Die Ausgänge YB und XB sind nur dann notwendig, wenn in diesem Logikblock eine „carry-chain“ endet, und bilden dann zum Beispiel den Übertragsausgang (*carry*) eines Addierers. Der Eingang CIN bzw. der Ausgang COUT ist zur Kaskadierung mehrerer übereinander liegender Slices zu einer „carry-chain“ notwendig. Dieser Ein- bzw. Ausgang ist nicht direkt sondern nur über die Ein- bzw. Ausgänge des darunter bzw. darüber liegenden Slices zugänglich. Für die Implementierung von Volladdierern ist extra ein Exklusiv-Oder-Gatter, zur Unterstützung von Multiplizierer ist extra ein UND-Gatter in die carry-chain integriert.

Die LUT ist aber nicht nur zur Implementierung von logischen Funktionen verwendbar sondern kann auch als 16 Bit Schieberegister konfiguriert werden. Wie wir in 3.3.1 gesehen haben, ist eine Look Up Table mit vier Eingängen im Prinzip nichts anderes als ein Speicher mit 16 Speicherplätzen. Aus diesem Grund ist es nicht weiter verwunderlich, daß eine LUT eines Virtex Slices auch als 16×1 Bit synchrones RAM konfigurierbar ist. Die zwei LUTs eines Logikblockes können aber auch als 16×2 Bit oder als 32×1 Bit synchrones RAM Verwendung finden. Innerhalb eines Logikblockes ist mit den zwei LUTs aber auch ein 16×1 Bit synchrones RAM mit zwei Ein- bzw. Ausgängen (dual ported) möglich.

Auch das Speicherelement ist konfigurierbar. Dabei ist sowohl der Grundzustand (ob nach dem Einschalten bzw. Zurücksetzen eine 1 oder eine 0 im Speicher ist) als auch die Flanke (ansteigend oder fallend) des Taktsignales auf die es reagieren soll konfigurierbar. Das Element kann sowohl synchron als auch asynchron zurückgesetzt (*reset*) werden.

sonstige Resourcen

Wichtig für das FPGA sind natürlich auch die Eingangs- bzw. Ausgangs-Anschlüsse, da diese erst die Kommunikation mit anderen Bausteinen ermöglichen. Vier dieser Anschlüsse sind speziell für den Anschluß externer Taktquellen ausgelegt. Das besondere dabei ist, daß diese Anschlüsse jeweils Verbindung zu einem der vier Taktnetze haben. Die Taktnetze erstrecken sich über den ganzen Chip und sind so aufgebaut, daß der Takt an allen Logikblöcken „zur gleichen Zeit“ ankommt. Für die Taktbehandlung befindet sich auch noch bei jedem dieser Takteingänge eine sogenannte DLL (*delay locked loop*). Mit dieser Komponente kann unter anderem der einkommende Takt um 90°, 180° oder 270° verschoben werden. Auch eine Frequenzverdopplung (durch Verwendung einer zweiten DLL auch eine Vervierfachung) ist möglich. Außerdem kann der einkommende Takt (genauer seine Frequenz) durch verschiedene Teiler von 1,5 bis 16 dividiert werden.

Da viele Entwürfe auch Speicher benötigen besitzen die FPGAs der Virtex Familie ein oder mehrere 4096 Bit grosse Speicherblöcke, die als verschiedene Speicher von 1×4096 bis 32×128 Bit als single-ported oder von 8×512 bis 16×256 Bit als dual-ported RAM konfigurierbar sind. Der Unterschied gegenüber der 32×1 Bit bzw. 16×1 Bit Blöcke bei Verwendung der LUT ist hauptsächlich in der Größe und in der Lokalisierung des Speichers zu sehen. Die Zusatz-Speicherblöcke (*block RAM*) befinden sich am linken bzw. rechten Rand des FPGAs. Dadurch müssen Logikkomponenten die diesen Speicher ansprechen wollen entweder am Rand plaziert werden, oder unter Verwendung vieler Verbindungsresourcen (damit auch längerer Signallaufzeiten) verbunden werden. Die als Speicher konfigurierten LUTs sind dagegen in einem Logikblock in der Nähe der zugehörigen Logik plazierbar.

Gatter-Äquivalente

Der Begriff Gatter-Äquivalente (*gate equivalent = GE*) sagt aus, wieviel Grundgatter (zum Beispiel NAND-Gatter mit zwei Eingängen) notwendig wären um eine bestehende Schaltung nachzubilden. Im Vergleich von Schaltungen auf Chipebene sagt diese Größe etwas über den „Platzverbrauch“, genauer gesagt über die Komplexität der Schaltungen aus. Wie kann aber festgestellt werden, ob eine Schaltung mit den Ressourcen eines FPGAs auskommt? Oder wie können zwei verschiedene FPGAs verglichen werden? Dafür geben die Hersteller an, wieviele Gatter-Äquivalente ein Baustein bei einer typischen Anwendung zur Verfügung stellt. Unter einer typischen Anwendung versteht jeder Hersteller etwas Anderes, so daß einem Anwender im Zweifelsfall nur die Analyse seiner Schaltung oder ein Versuch-und-Irrtum-Vorgehen übrig bleibt. Bei der Analyse einer Schaltung ist zu bedenken, daß zum Beispiel die LUTs der 8 Slices, die für einen 16 Bit Volladdierer benötigt werden, **nicht** mehr als RAM-Speicher oder Logikgenerator zur Verfügung stehen. Müssen aber die Ausgänge dieses Addierers gespeichert werden (zum Beispiel in einer Pipeline) so sind die Speicherelemente der 8 Slices nicht nur frei, sondern für diese Verwendung auch vorgesehen, da sie schon direkt mit den „Ausgängen“ des Addierers verbunden sind.

Kapazität

Wieviele CLBs bzw. Slices bzw. Logikblöcke sind in so einem FPGA? Wir wissen, daß pro CLB zwei Slices und pro Slices zwei der oben vorgestellten Logikblöcke vorhanden sind. Die Anzahl der CLBs ist von der Größe des FPGAs abhängig. Die CLBs sind matrixförmig in Zeilen und Spalten angeordnet. Xilinx bietet bei der Virtex Familie folgende Bausteingrößen an (diese Tabelle ist nicht vollständig, für weitere Angaben siehe Xilinx [31]):

Baustein Name	CLB Matrix	**Slices** $= Zeilen \times Spalten$	LUTs $= Slices \times 2$	**RAM-Blöcke**	Blockspeicher in Bits $= Blöcke \times 4096$	max. LUT Speicher $= LUTs \times 16$
XCV50	16×24	**768**	1536	**8**	32768	24576
XCV100	20×30	**1200**	2400	**10**	40960	38400
XCV150	24×36	**1728**	3456	**12**	49152	55296
XCV200	28×42	**2352**	4704	**14**	57344	75264
XCV400	40×60	**4800**	9600	**20**	81920	153600
XCV600	48×72	**6912**	13824	**24**	98304	221184
XCV800	56×84	**9408**	18816	**28**	114688	301056
XCV1000	64×96	**12288**	24576	**32**	131072	393216

Tabelle 3.4: Virtex Familie

Wie kommt ein Baustein wie der XCV800 zu seinem Namen? Der Namen dieses FPGAs „verspricht“ 800 Tausend Gatter Äquivalente[1]. Wie kommt der Hersteller zu dieser Behauptung? Xilinx rechnet für jede LUT die als Logikgenerator verwendet wird 12 GE. Für jede LUT, die als Speicher verwendet wird (16×1 Bit) pro Speicherbit 4 GE also insgesamt 64 GE. Jedes Flip-Flop wird je nach Optionen (set/reset, synchron/asynchron, ...) als 6 bis 12 GE gerechnet. Bei der Annahme, daß 20% der LUTs als Speicher und die anderen 80% als Logikgenerator verwendet werden, erhalten wir so pro Slice:

Logik	12×2	80%	$\approx$	20 GE
Speicher	64×2	20%	$\approx$	25 GE
Flip-Flop	10×2		$\approx$	20 GE
Ctrl-Logik (s. 3.2)			$\approx$	5 GE
Insgesamt			$\approx$	70 GE

Das ergibt bei 9408 Slices mit je 70 GE eine Summe von 658560 GE falls alle Slices voll ausgenutzt werden. Jedes Blockspeicher-Bit entspricht noch einmal 4 GE, so daß wir noch einmal $4 \times 114688 = 458752$ verfügbare Gatter Äquivalente erhalten. Damit erhalten wir ohne die zusätzlichen Elemente wie Anschlüsse, Tristate-Buffer und anderen Elementen, schon eine Gesamtkapazität von über 1,1 Millionen GE unter der Annahme, daß 20% der LUTs als Speicher-Bausteine benutzt werden. Dieser Abschnitt sollte **nicht** die Rechenweise der Firma Xilinx vorführen (genauere Angaben dazu siehe [31]) sondern ein Gefühl für die Komplexität eines solchen FPGAs geben.

[1] Abkürzung: GE für *gate equivalent*

Geschwindigkeit

Die FPGAs der Virtex-Familie gibt es in unterschiedlichen Geschwindigkeitsstufen den sogenannten *speedgrades*. Eine Konfigurationsdatei, die für einen Baustein des einen speedgrades erstellt wurde, kann auch für einen identischen Baustein (Größe **und** Gehäuse) mit einem anderen speedgrade verwendet werden. Die FPGAs der einzelnen speedgrades unterscheiden sich nur in ihrer physikalischen Ausführung und damit in ihren Schaltzeiten. Zum Beispiel ist das Minimum an Zeit die ein 16 Bit Addierer in einem Virtex Baustein mit speedgrade -6 (schnellster Baustein) benötigt 5.0 ns, was eine maximale Frequenz von annähernd 200 Mhz bedeutet. Der gleiche Addierer braucht bei dem langsamsten speedgrade -4 circa 6.5 ns und erreicht dadurch eine theoretische Maximalfrequenz von 160 MHz. Natürlich sind diese Geschwindigkeiten in einem endgültigen Design selten zu erreichen, da noch die Verzögerungen der Verbindungslinien und etwaiger Logik vor oder nach dem Addierer hinzugezählt werden müssen. Normalerweise werden Schaltungen in den Bausteinen des langsamsten speedgrades realisiert, da diese Bausteine billiger sind. Auch die Markteinführung der langsameren Bausteine ist meist früher, da sie in der Fertigung nicht ganz so aufwendig sind.

Viele (vor allen Dingen die großen) FPGAs werden für Tests oder Prototyping verwendet und es kommen daher hier die billigeren und damit langsameren Bausteine zum Einsatz. Beim Test bzw. Prototyping ist es nicht so wichtig, ob der Entwurf mit 12 oder 16 MHz „läuft“, viel wichtiger ist, daß er überhaupt (und fehlerfrei) „läuft“.

3.4 Entwurfszyklus

3.4.1 Eingabe

In diesem Abschnitt wollen wir nicht den schwierigen Vorgang von der Idee zum „fertigen“ Entwurf in Form einer HDL-Beschreibung besprechen, sondern nur grob die Schritte die notwendig sind, bis eine (hoffentlich) funktionierende Schaltung auf dem Chip (speziell auf einem FPGA) vorliegt. Als allererstes muß natürlich die Grundidee strukturiert und in eine geeignete Form gebracht werden. Wie wir schon weiter oben gesehen haben, eignen sich dafür Hardwarebeschreibungssprachen vorzüglich, besonders VHDL. Im weiteren gehen wir davon aus, daß die gewählte Sprache VHDL ist.

3.4.2 Simulation

Wenn der Entwurf vollständig in VHDL vorliegt, so kann er simuliert werden. Natürlich werden komplexere Entwürfe nicht vollständig simuliert, sondern wie in Abschnitt 3.1.3 gesehen, nur stückweise. Die beim jeweiligen Test nicht relevanten Teile werden durch einfachere (aber funktionskompatible) Beschreibungen ersetzt.

Die Simulation erfolgt mit einem Simulatorprogramm, das die gewählte Sprache versteht. Meistens ist es bei Simulatoren auch möglich, Module aus anderen Hardwarebeschreibungssprachen oder allgemeinen Programmiersprachen einzubinden. Durch solche Simulationen können logische Abläufe überprüft werden. Durch mehrere Iterationen entsteht so eine funktional korrekte Beschreibung der gewünschten Funktion. Da es VHDL auch ermöglicht Verzögerungen und andere Zeitverhal-

ten zu beschreiben, ist auch eine Simulation des gewünschten zeitlichen Ablaufes möglich. Durch „Klammerung“ mit entsprechenden Direktiven kann der nur simulatorspezifische Teil für die Synthese ausgeblendet werden.

```
-- synthesis translate_off
  assert (adder_cell_width=64)
    report "only Addercells of 64 bit width supported"
    severity FAILURE;
-- synthesis translate_on
```

Diese so geklammerten Anweisungen werden nur vom Simulator gelesen und entsprechend umgesetzt.

3.4.3 Synthese

Nachdem der Entwurf in einer synthetisierbaren Form vorliegt (IEEE 1076.6) kann dieser Entwurf von einem Synthesewerkzeug bearbeitet werden. Diese Programme untersuchen dabei die Beschreibung in der Hochsprache und erzeugen daraus eine sogenannte Netzliste. In dieser Netzliste ist die gesamte in Hardware umzusetzende Funktion aus einfachen Bausteinen aufgebaut beschrieben. Dabei werden schon, falls notwendig und möglich, Eigenheiten der Zieltechnologie (z.B. bei FPGA-Entwürfen die Größe der LUT) berücksichtigt. Um diese Netzliste für verschiedene Werkzeuge verständlich und austauschbar zu halten, wurde ein einheitliches Format entwickelt: EDIF (*electronic design interchange format* siehe [10]). Diese Netzliste wird jetzt von Zieltechnologie abhängigen Programmen, in unserem Falle von Programmen der Firma Xilinx weiterverarbeitet.

3.4.4 Mapping

Beim sogenannten „mapping“ (*engl: Abbildung*) werden die Elemente der Netzliste endgültig auf die Zieltechnologie abgebildet. In diesem Stadium werden auch redundante Elemente entfernt und falls notwendig andere Elemente vervielfacht. Eine Vervielfachung von Elementen ist zum Beispiel dann notwendig, wenn der Ausgang eines Elementes zu viele Eingänge anderer Elemente treiben muß. In dieser Stufe wird auch offensichtlich, ob überhaupt genügend Ressourcen wie Anschlüsse, CLBs, Speicher, Flip-Flops und andere Elemente die benötigt werden, vorhanden sind. Nachdem die Abbildung aller Elemente der Netzliste auf Elemente des FPGAs stattgefunden hat, muß die physikalische Abbildung dieser zugeordneten Elemente auf die einzelnen FPGA-Elemente erfolgen. Der Mapper hat nur festgelegt, was in eine LUT oder in ein Flip-Flop abgebildet wird aber nicht auf welches der tatsächlich vorhandenen (physikalischen) Elemente.

3.4.5 Placing

Diese Aufgabe übernimmt der Placer. Beim „placing“ (*engl: Platzierung*) werden die Elemente der Netzliste, die der mapper geschrieben hat, auf die vorhandenen Elemente im FPGA verteilt. Bei diesem Verteilungsprozeß werden eventuell vorhandene „timing constraints“ oder „placing constraints“ berücksichtigt. Diese Einschränkungen (*constraints*) wurden entweder vom Benutzer oder aber von den Werkzeugen

der vorherigen Verarbeitungsstufen in den Entwurf (bzw. die Netzliste) eingeführt. Aufgrund dessen, daß der Placer Elemente die „zusammengehören“ in leicht erreichbarer Nähe platzieren muß, ist das „placing“ eine zeitaufwendige Stufe. Nachdem der Placer mit seiner Arbeit fertig ist, sind entweder alle „placing constraints“ erfüllt, oder aber nicht (für das Programm) erfüllbar. Jetzt müssen nur noch die „timing constraints“ erfüllt werden.

3.4.6 Routing

Nachdem der Placer die Elemente platziert hat, müssen die Verbindungen gelegt werden. Diese „Verdrahtung“ übernimmt der Router. Auch der Router muß sich um Nebenbedingungen kümmern, nämlich um die noch nicht erfüllten „timing constraints“. Erst nach dem Routing steht fest, wie lange ein Signal von einem Punkt auf dem FPGA bis zu einem anderen Punkt, durch die ganzen Verbindungsleitungen und Logikelemente hindurch, benötigt. Das heißt, die eigentliche Signallaufzeit addiert sich aus den Verzögerungszeiten der Verbindungen (die erst der Router erstellt), und den zu durchlaufenden Logikstufen (die schon dem Mapper bekannt sind). Aus diesem Grund kann es sein, daß schon in der mapping-Stufe eine Verletzung der Zeitbedingungen (*timing constraints*) erkannt wird, nämlich genau dann, wenn schon die Summe der Zeitverzögerungen aller Logikstufen die insgesamt zur Verfügung stehende Zeit überschreitet. Sollte das der Fall sein muß der Entwurf neu gestaltet werden. Dies kann zum Beispiel durch das Einfügen einer oder mehrerer (zusätzlicher) Pipeline-Stufen erfolgen.

Falls der Router genügend Ressourcen zum „Verdrahten“ des ganzen Entwurfs hat, steht am Ende des Programmes fest, ob die (*timing constraints*) erfüllt sind oder nicht. Falls sie nicht erfüllt sind, so kann durch ein anderes Placing und anschließendem erneutem Routing ein weiterer Versuch unternommen werden alle Bedingungen zu erfüllen.

3.4.7 Floorplanning

Es ist aber auch möglich, daß der Benutzer sich von einem Programm die Platzierungen und die Verbindungen des Entwurfes anzeigen läßt. Dies geschieht mit dem sogenannten Floor-planner. Er ermöglicht dem Benutzer nicht nur eine Visualisierung des Entwurfes auf dem FPGA, sondern auch die Möglichkeit selbst Elemente neu zu platzieren. Durch diese Platzierung kann, falls genügend routing-Ressourcen vorhanden sind, ein optimaleres Zeitverhalten (oder Platzausnutzung) erreicht und somit eventuell alle Bedingungen erfüllt werden. Wenn alle Bedingungen erfüllt sind, kann aus dem platzierten und verdrahteten Entwurf eine Konfigurationsdatei erzeugt werden, die dann in das FPGA geladen wird. Nach dem Laden steht die Hardwarefunktion des Entwurfs im FPGA zur Verfügung.

3.5 Verwendete Werkzeuge

In diesem Abschnitt werden wir kurz die von uns verwendeten Hardware- und Softwarewerkzeuge vorstellen, die bei der Realisierung der im nächsten Kapitel vorgestellten Implementierungen eingesetzt wurden.

3.5.1 Hardware

Als FPGA wurde ein Xilinx Virtex XCV800-4 im BG432 Gehäuse verwendet. Dieses FPGA befindet sich auf der Entwicklungskarte „Spyder-Virtex-X2“ vom Forschungszentrum Informatik in Karlsruhe (siehe [12], bzw. Abbildung C.2). Diese Karte ist sowohl als PCI-Einsteckkarte als auch „stand-alone“ betreibbar. Wir haben sie als PCI-Buskarte verwendet.

Als PCI-Interface befindet sich auf dieser Karte ein PCI 9080 Baustein der Firma PLX Technology (siehe [26]). Über diesen Baustein erfolgt sowohl das Programmieren des FPGAs (laden der Konfigurationsdatei) als auch die Kommunikation mit dem implementierten Entwurf. Es werden 32 Bit breite Datenworte über den PCI-Baustein auf den lokal auf der Karte vorhandenen internen Bus gebracht. Von diesem Bus kann der FPGA Baustein die 32 Bit in 4 Bustakten (36 MHz) byteweise einlesen. Daten die der FPGA-Baustein ausgibt, werden über den gleichen Bus an das entsprechende Programm gesendet. Auf der Karte befinden sich zwei Quarz-Oszillatoren. Der erste Oszillator stellt den 36 MHz Systemtakt für den lokalen Bus und den PCI-Baustein zur Verfügung. Der zweite Oszillator ist ausschließlich für den Betrieb des Entwurfes auf dem FPGA zuständig. In unserem Fall wählten wir einen 5 MHz Oszillator.

Die Kommunikation des Benutzers mit dem FPGA läuft ausschließlich über ein in C geschriebenens Programm namens `call_suma`. Die PCI-Bus Anbindung erfolgt dabei mit Hilfe einer Funktionsbibliothek für den PCI 9080 Baustein von PLX Technology. Um die Auswertung der Daten von und zu dem Entwurf für den Benutzer zu vereinfachen wurde ein php3-Programm geschrieben, daß die Daten über einen normalen Internet-Browser zur Verfügung stellt (siehe dazu Anhang C).

Von der Möglichkeit auf der Karte bis zu 2 MB SRAM-Speicher zu installieren und über ein zu implementierendes Speicherinterface durch den FPGA anzusprechen wurde kein Gebrauch gemacht. Aller im Entwurf verwendeter Speicher befindet sich im FPGA und wurde entweder durch den in 3.3.4 vorgestellten Blockspeicher oder durch LUT als Speicherelemente realisiert.

3.5.2 Software

Als Software für die Entwicklung des C-Programmes `call_suma` kam Visual C++ 4.2 der Firma Microsoft zum Einsatz.

Für die anfängliche Simulation der VHDL-Dateien wurde der VHDL-Simulator Model*Sim* EE/Plus Version 5.1g der Firma Modeltech verwendet.

Die Synthese (Umwandlung der VHDL-Beschreibung in eine EDIF-Netzliste) erfolgte hauptsächlich mit *Synplify Version 5.2.2* bzw. *Synplify Pro Version 6.1.3* der Firma Synplicity. Auch der FPGA-Compiler der Firma Synopsys (in der Foundation 2.1i von Xilinx enthalten) wurde anfänglich für diesen Schritt verwendet.

Für die restlichen Bearbeitungsschritte bis zur Konfigurationsdatei wurde die Foundation 2.1i Software der Firma Xilinx verwendet. Alle Programme wurden auf einem 400 MHz Pentium II mit 512 MB unter Windows NT 4.0 SP6 ausgeführt. Als Editor für die VHDL-Dateien fand die Windowsversion des GNU Emacs 20.4.1 mit VHDL-mode 3.30 Verwendung.

Kapitel

4

Implementierung

“ Daß man dasselbe zweimal tut, ist unvermeidlich, aber ein drittes Mal nie. Wenn du dem Gegner mit einer Taktik kommst und keinen Erfolg hast, wird bei einem zweiten Versuch die Wirkung noch geringer sein. ”

Miyamoto Musashi: Das Buch der fünf Ringe

In diesem Kapitel stellen wir die Realisierung einer Summationsmatrix für das IEEE 754 single precision Zahlenformat vor. Wir werden die Funktion und das Zusammenspiel der einzelnen Komponenten beschreiben, ohne jedoch auf die Umsetzung dieser Funktionen in VHDL einzugehen. Alle VHDL Quelltexte des vorgestellten Entwurfes sind im Anhang D zu finden. Anschließend an die Funktionsbeschreibung werden wir Erfahrungen und Daten von der Umsetzung des Entwurfes in das vorgestellte FPGA beschreiben.

Bei der vorgestellten Summationsmatrix handelt es sich um ein aus zehn 64 Addierern (eigentlich Addierer-Subtrahierer) mit einem Kontextspeicher, so daß mehrere (normalerweise 16) verschiedene exakte Skalarprodukte bearbeiten werden können.

Eine gewisse Verwirrung erzeugt die Verwendung des Begriffes *Summationsmatrix* da wir zum einen darunter den im Kapitel 2 vorgestellten langen Akkumulator in Matrixanordnung, zum anderen aber den ganzen Komplex bestehend aus Multiplizierer, Shifter und dem langen Akkumulator verstehen. Wenn wir den ganzen

Komplex als Summationsmatrix bezeichnen, so verwenden wir zur Verdeutlichung für den langen Akkumulator in Matrixform den Begriff Summationsmatrix-Kern. Als Abkürzung für diese Begriffe verwenden wir auch SuMa bzw. SuMa-Core.

4.1 Blockschaltbild

In der Abbildung 4.1 sind die einzelnen Komponenten des Entwurfs zu sehen. Der Datenfluss verläuft von oben nach unten, das heißt, von der Eingabe (`INPUT`) über die Dekodier-Einheit (`Decode`), durch diverse andere Einheiten und Pipelinestufen (`Pipes`) in die eigentliche Summationsmatrix (`SuMa Core`).

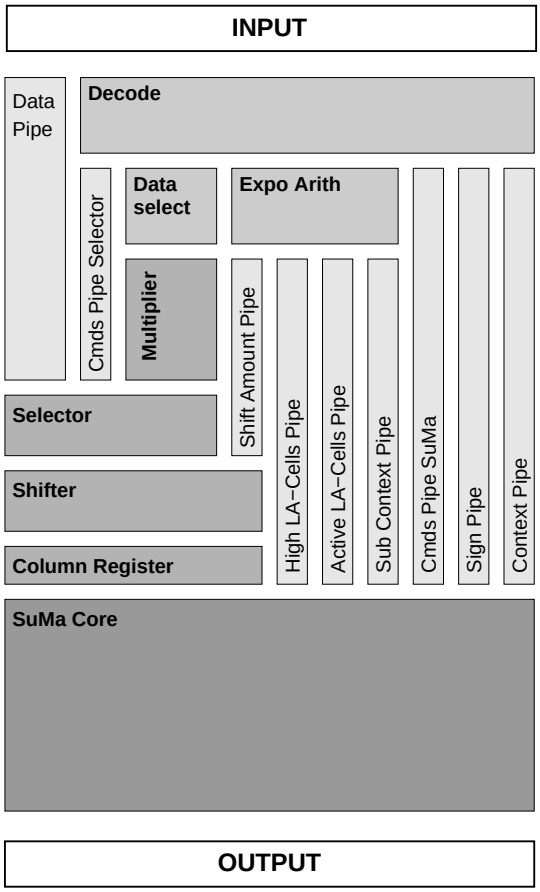


Abbildung 4.1: Blockschaltbild des Entwurfes

Die verschiedenen Pipelinestufen (`Pipes`) sind nur dazu da um alle Daten solange zu verzögern, daß sie zur gleichen Zeit (im gleichen Takt) an der Summationsmatrix oder an einer anderen Einheit ankommen wie parallel laufende Daten, die einen anderen Verarbeitungsweg durchlaufen. So wird die Shiftweite durch mehrere Registerstufen in der `Shift Amount Pipe` solange zwischengespeichert, bis das dazu gehörende Produkt mit der Shiftweite am Shifter ankommt. Die `Data Pipe` wiederum verzögert die unbehandelten Eingangsdaten solange, bis sie mit dem aus ihnen erzeugten Produkt, das erst die Dekodier-Einheit (`Decode`), die Datenauswahl-Einheit (`Data select`) und den Multiplizierer (`Multiplier`) durchlaufen mußte, an

der Selektier-Einheit (`Selektor`) ankommt. Erst die Selektier-Einheit entscheidet dann, welche Daten (Eingangsdaten oder Produkt) an den Shifter und damit an die Summationsmatrix weitergegeben werden. Die Länge (Tiefe) der einzelnen Stufen wird im VHDL-Text über Konstanten gesteuert. Aus dem Blockschaltbild Abbildung 4.1 ist durch die Darstellung ersichtlich, welche Daten zur gleichen Zeit an welcher Einheit ankommen. Deswegen werden wir nicht mehr weiter auf diese Elemente eingehen.

Alle dargestellten Einheiten (mit Ausnahme des Multiplizierers), also die Dekodier-Einheit (`Decode`), die Datenauswahl-Einheit (`Data select`), die Exponenten-Arithmetik-Einheit (`Expo Arith`), die Selektier-Einheit (`Selektor`), die Schiebe-Einheit (`Shifter`) und die Zeilenregister-Einheit (`Columns Register`) benötigen einen Takt um ihre Daten zu bearbeiten. Das heißt, jede dieser Einheiten hat eine „Pipeline-Tiefe“ von einem Takt. Der Multiplizierer mit dem wir unsere Tests durchgeführt hatten, hat eine Pipeline-Tiefe von 4 Takten. Natürlich wäre auch ein kombinatorischer Multiplizierer mit nur einem Takt Pipeline-Tiefe möglich gewesen. Die Gründe für bzw. gegen einen solchen Multiplizierer werden wir an entsprechender Stelle diskutieren.

4.2 Eingabe-Einheit

Die Eingabe-Einheit (`INPUT`) dient der Kommunikation des lokalen Busses der Spyderkarte (siehe 3.5.1) mit dem eigentlichen Entwurf. Der lokale Bus wird mit einem Takt von 36 MHz betrieben, wobei pro Takt nur ein Byte (8 Bit) an Daten übertragen wird. Von dem Adressraum den diese Entwicklungskarte zur Verfügung stellt, werden von uns nur 4 Bit also 16 Adressen verwendet. Diese 16 Adressen sprechen 16 verschiedene 32 Bit Register an, wobei 5 davon der Eingabe dienen. Die fünf Eingaberegister unterteilen sich in zwei Register für IEEE 754 single precision Zahlen (*REG0* und *REG1*), ein Befehlsregister (*REG2*) und zwei Zusatzregister (*REG10* und *REG11*). Mit Hilfe der beiden Zusatzregister können auch IEEE 754 double precision Zahlen eingegeben werden, wobei dann die oberen 32 Bit der ersten double-Zahl in *REG0* und die unteren 32 Bit in *REG10* geschrieben werden. Für die zweite double-Zahl steht dann *REG1* (für die oberen 32 Bit) und *REG11* (für die unteren 32 Bit) zur Verfügung. Obwohl der Entwurf für IEEE 754 single precision Zahlen ausgelegt wurde, haben wir auch die in einem früheren Kapitel (siehe 2.6) angesprochene Möglichkeit, IEEE 754 double precision Zahlen in einer IEEE 754 single precision Summationsmatrix zu akkumulieren, teilweise implementiert.

Sobald die benötigten Daten in den zwei bzw. vier Registern anliegen, kann durch das Laden eines Befehls in *REG2* die Übergabe der Daten an die Summationsmatrix erfolgen. Die Summationsmatrix ist mit dem zweiten, vom internen Bus unabhängigen Takt betrieben, wobei dieser in der vorliegenden Implementierung nicht größer als 36 MHz sein darf. Diese Begrenzung ist darin begründet, daß die Eingabe implizit davon ausgeht, daß der Bustakt größer ist als der Takt mit dem die Summationsmatrix betrieben wird. Für die Datenübergabe hat die Summationsmatrix eine 64 Bit große Daten- und eine 32 Bit große Befehlsschnittstelle (`MISCmd`). Die Datenschnittstelle besteht aus den zwei 32 Bit breiten Zugängen `MID0` und `MID1`. Sobald ein gültiger Befehl in *REG2* geschrieben wurde, können die Eingaberegister nicht mehr verändert werden, bis alle Daten nach dem in Tabelle 4.1 dargestellten

Ablauf an die Summationsmatrix übergeben wurden. In dieser Tabelle bedeutet *NULL*, daß alle 32 Bit auf logisch 0 gesetzt werden. Sobald der aktuelle Befehl abgearbeitet ist, was 1 oder 4 Takte dauert, wird *REG2* wieder auf *NULL* gesetzt.

Wie aus der Tabelle ersichtlich ist, werden bei dem Befehl „akkumuliere double Produkt“ im Gegensatz zu allen anderen Befehlen 4 Takte benötigt. Dabei bedeutet die Angabe zweiter, dritter bzw. vierter Takt die Taktnummer nach Anlegen des „akkumuliere double“-Befehls. Im ersten Takt werden die beiden höherwertigen 32 Bit Teile der IEEE 754 double precision Zahlen, im zweiten Takt die anderen 32 Bit Teile an die Summationsmatrix übertragen. Im dritten und vierten Takt werden keine weiteren Daten übertragen. Um die vier notwendigen Partialprodukte zu bilden, werden die zuvor übertragenen Daten in der Datenauswahl-Einheit zwischengespeichert. Das Befehlswort im zweiten bis vierten Takt unterscheidet sich jeweils um 2 Bit vom Befehlswort des ersten Taktes. In diesen zwei Bit wird die Taktnummer binär mit Null beginnend kodiert. Also im ersten `00`, im zweiten `01`, im dritten `10` und im vierten Takt `11`. Dieser „Zähler“ ist notwendig, um der Datenauswahl-Einheit (`Data select`) und der Exponentenarithmetik-Einheit (`Expo Arith`) mitzuteilen, welches der vier Partialprodukte gerade bearbeitet wird.

Bei dem Befehl „addiere single Zahl“ wird auch noch der Inhalt von *REG1* an die Summationsmatrix übertragen. Diese Daten werden von ihr ignoriert, so daß stattdessen auch eine *NULL* übertragen werden könnte.

Die Eingabe-Einheit gibt in jedem Takt 64 Bit Daten und einen 32 Bit Befehl an die Summationsmatrix weiter, solange aber kein expliziter Befehl vom Benutzer gesendet wurde (in *REG2* geladen wurde), sind diese 96 Bit alle Null. Die Summationsmatrix interpretiert dies als „akkumuliere 0×0 auf den aktuellen Kontext“.

4.3 Ausgabe-Einheit

Obwohl die Ausgabe in Abbildung 4.1 erst ganz zum Schluß dargestellt ist, wollen wir sie hier schon vorstellen. Die Ausgabe-Einheit besteht nur aus mehreren 32 Bit Registern, die über den lokalen Bus und den PCI-Baustein genauso wie *REG0, REG1, ...* angesprochen werden (siehe Tabelle 4.2). Dabei ist im Gegensatz zu den Eingabe-Registern nur ein lesender Zugriff möglich. Jeder Lesebefehl (siehe Tabelle A.4) aktualisiert immer alle Ausgabe Register.

4.4 Dekodier-Einheit

Nachdem die Daten von der Eingabe-Einheit an die Summationsmatrix übergeben wurden, gelangen diese an die Dekodier-Einheit (`Decode`). Diese Einheit interpretiert den Befehl und generiert daraus Steuerungssignale für die darunterliegenden Einheiten. Außer diesen Steuersignalen werden noch weitere Aufgaben von dieser Einheit übernommen.

Da wäre als erstes die Erzeugung des Vorzeichens (`sign`). Je nach dem welcher Befehl aktiv ist, wird entweder das Vorzeichen des Produktes oder das Vorzeichen der einzelnen Zahl (bei den Additionsbefehlen) erzeugt. Hierbei ist zu bedenken, daß der Summationsmatrix-Kern in Abhängigkeit des Vorzeichens entweder addiert (positives Vorzeichen) oder subtrahiert (negatives Vorzeichen). Das heißt alle Befehle,

Befehl	1. Takt	2. Takt	3. Takt	4. Takt
addiere single Zahl	$\Leftarrow$ *REG0* $\Leftarrow$ *REG1*# $\Leftarrow$ *REG2*			
akkumuliere single Produkt	$\Leftarrow$ *REG0* $\Leftarrow$ *REG1* $\Leftarrow$ *REG2*			
addiere double Zahl	$\Leftarrow$ *REG0* $\Leftarrow$ *REG1* $\Leftarrow$ *REG2*			
akkumuliere double Produkt	$\Leftarrow$ *REG0* $\Leftarrow$ *REG1* $\Leftarrow$ *REG2*	$\Leftarrow$ *REG10* $\Leftarrow$ *REG11* $\Leftarrow$ *REG2*⋆	$\Leftarrow$ *NULL* $\Leftarrow$ *NULL* $\Leftarrow$ *REG2*⋆	$\Leftarrow$ *NULL* $\Leftarrow$ *NULL* $\Leftarrow$ *REG2*⋆
andere Befehle	$\Leftarrow$ *NULL* $\Leftarrow$ *NULL* $\Leftarrow$ *REG2*			

der Inhalt von *REG1* wird von der Summationsmatrix ignoriert (siehe 4.2)

⋆ mit Partialproduktzähler: `01`, `10`, `11` (siehe 4.2)

Tabelle 4.1: Datenübergabe an Summations-Matrix

Register	I/O	Funktion	siehe
REG0	Eingabe	single Operand 1 msw von double Operand 1	4.2
REG1	Eingabe	single Operand 2 msw von double Operand 2	4.2
REG2	Eingabe	Befehlseingabe	4.2
REG3	Ausgabe	Kontextinformationen: Bit(32 ... 30): ist konstant `101` Bit(29): Overflow Bit Bit(28): Vorzeichen Bit (`0` = positiv) Bit(27): sticky-Bit Bit(26 ... 17): `msw_lacell` Bit(16 ... 5) konstant `0` Bit(4): `all_zero` der oberen 64 Ausgabebits Bit(3): `all_one` der oberen 64 Ausgabebits Bit(2 ... 1): Bit(128 ... 127) der Ausgabe	 4.12 4.12 4.12 4.12 4.11 4.11 4.12
REG4	Ausgabe	Bit(32 ... 1): Bit(126 ... 95) der Ausgabe	4.12
REG5	Ausgabe	Bit(32 ... 3): Bit(94 ... 65) der Ausgabe Bit(2): `all_zero` der unteren 64 Ausgabebits Bit(1): `all_one` der unteren 64 Ausgabebits	4.12 4.11 4.11
REG6	Ausgabe	Bit(32 ... 1): Bit(64 ... 33) der Ausgabe	4.12
REG7	Ausgabe	Bit(32 ... 1): Bit(32 ... 1) der Ausgabe	4.12
REG8	Ausgabe	nur für Tests	
REG9	Ausgabe	nur für Tests	
REG10	Eingabe	lsw von double Operand 1	4.2
REG11	Eingabe	lsw von double Operand 2	4.2
REG12	Ausgabe	nur für Tests	
REG13	Ausgabe	nur für Tests	
REG14	Ausgabe	nur für Tests	
REG15	Ausgabe	Datum des Entwurfs, Versionsnummer, Architektur (CSA, SCA, Tranfer-Register), ...	Anhang D*

* Datei `versionnumber.vhd`

Tabelle 4.2: Interface Register des Entwurfes

die das Vorzeichen beeinflussen, wie zum Beispiel der Subtraktionsbefehl (Umkehrung des Vorzeichens) oder der Negationsbefehl (Null minus Akku) wirken schon in der Dekodier-Einheit auf das Vorzeichen und bestimmen somit die Aktion (Addition oder Subtraktion) der Addierer im Summationsmatrix-Kern. Eine Ausnahme von diesem Verhalten ist die bedingte Negation da sie das aktuelle Vorzeichen des Skalarproduktes benötigt. Deswegen wird für bei diesem Befehl das Vorzeichen noch einmal im Summationsmatrix-Kern verändert. Das aktuelle Vorzeichen wird solange gespeichert bis ein neuer Befehl ein anderes Vorzeichen erzeugt.

Die Dekodier-Einheit extrahiert die Exponenten der entsprechenden IEEE 754 Zahlen (single oder double) je nach Befehl aus den Eingangsdaten und erzeugt auch das eventuell vorhandene „hidden bit“.

Ein weiterer wichtiger Punkt ist die Extraktion der Kontextaddresse aus dem Befehl und die Speicherung derselben. Die Kontextaddresse wird solange gespeichert, bis ein neuer Befehl eine andere Kontextaddresse angibt.

4.5 Datenauswahl-Einheit

Die Datenauswahl-Einheit (`Data Select`) blendet sowohl das Vorzeichen als auch den Exponenten aus den 32 Bit Daten aus und bildet zusammen mit dem „hidden bit“ die Mantisse einer IEEE 754 single precision Zahl. Die gleiche Aufgabe übernimmt sie für IEEE 754 double precision Zahlen, falls es sich um die 32 höherwertigen Bits handelt. Im double Fall speichert sie auch noch die in den ersten beiden Takten ankommenden Daten, so daß sie die Daten für die Partialprodukte des dritten und vierten Taktes weitergeben kann. Wie in Tabelle 4.3 zu sehen ist, werden bei den Steuer- und Lesebefehlen die aufbereiteten Daten an den Multiplizierer weitergegeben. Dies ist unkritisch, da die Selektiereinheit nur bei solchen Befehlen, die wirklich etwas in den langen Akkumulator übernehmen müssen, Daten durchläßt. In allen anderen Fällen wird das Produkt verworfen und ein Nullprodukt weitergegeben. Die Datenauswahl-Einheit ist eigentlich nur für den IEEE 754 double precision Betrieb notwendig, ihre Funktion könnte auch von der Dekodier-Einheit übernommen werden. In der Tabelle 4.3 bedeutet die Angabe zweiter, dritter oder vierter Takt wie schon bei der Tabelle 4.1 die Taktnummer nach Anlegen des (akkumuliere double) Befehls.

4.6 Exponenten-Arithmetik-Einheit

Die Exponenten-Arithmetik-Einheit (`Expo Arith`) berechnet zuallererst, wie ihr Name schon vermuten läßt, den Exponenten des aktuellen (Partial-)Produktes. Dabei werden im vorliegenden Entwurf die Exponenten der Teilprodukte bei einem IEEE 754 double precision Zahlen Produkt so berechnet, daß sie in einen IEEE 754 single precision Akkumulator akkumulierbar sind. Das bedeutet, daß das Produkt 1.0×1.0 im single- wie im double-Fall auf das gleiche Bit fällt. Dadurch kann in ein bestehendes IEEE 754 single precision Skalarprodukt ein IEEE 754 double precision Produkt akkumuliert werden, solange das Produkt aus den double-Zahlen nicht den entsprechenden Exponentenbereich verläßt. Dieses Verhalten wurde von uns aus zweierlei Gründen gewählt. Einmal weil wir die eigentliche Behandlung von IEEE 754 dou-

ADx aufbereitete Daten: kein Exponent, kein Vorzeichen, hidden bit
SADx gespeicherte aufbereitete Daten aus Takt eins
UDx unveränderte Eingangsdaten
SDx gespeicherte Daten aus Takt zwei

Befehl	1. Takt	2. Takt	3. Takt	4. Takt
addiere single Zahl	**AD0** **NULL**			
akkumuliere single Produkt	**AD0** **AD1**			
addiere double Zahl	**AD0** **UD1**			
akkumuliere double Produkt	**AD0** **AD1**	**AD0** **AD1**	**SD1** **SAD0**	**SAD1** **SD0**
andere Befehle	**AD0** **AD1**			

Tabelle 4.3: Datenweitergabe der Datenauswahl-Einheit

ble precision Zahlen mit den verschiedenen Fensterkonzepten, wie sie in Abschnitt 2.6 beschrieben haben, nicht implementiert haben, da die Wahl des entsprechenden Konzeptes sehr stark vom Einsatzgebiet des Entwurfes abhängt.

Der zweite Grund für dieses Verhalten ist die Tatsache, daß es mit dem vorhandenen Interface unmöglich ist, dem Entwurf in jedem Takt ein neues Operandenpaar anzubieten. Wie wir in Abschnitt 3.5.1 gesehen haben, sind der Summationsmatrix-Takt und der PCI-Bus getrennt. Mit der Spyderkarte ist es nicht möglich, der Summationsmatrix pro (Summationsmatrix-)Takt 64 Bit (zwei Operanden) oder gar 96 Bit (Operanden und Befehl) zu übertragen. Eigentlich ist aber genau die Möglichkeit pro Takt ein Operandenpaar zu verarbeiten, die Stärke der Summationsmatrix. Um den Nachweis zu erbringen, daß unser Entwurf sehr wohl genau diese Stärke umsetzt, haben wir den double-Fall wie beschrieben implementiert, so daß dieses Verhalten für immerhin vier aufeinanderfolgende Takte realisiert wird.

Die nächste wichtige Aufgabe für die Exponenten-Arithmetik-Einheit ist die Erzeugung von jeweils 3 Signalen für jeden der 10 Addierer des langen Akkumulators. Diese Signale werden sowohl von dem aktuellen Befehl als auch vom aktuellen Exponenten bestimmt. In der Tabelle 4.4 ist die Bedeutung der einzelnen Signale aufgelistet. Die dabei verwendeten Namen, entsprechen den Signalbezeichnungen in den VHDL-Dateien. Dabei steht die Bezeichnung `lacell` für Lokale Addierer-Zelle (**l**ocal **a**dder **c**ell). Außerdem ist noch angegeben, für welche Befehle diese Signale hauptsächlich relevant sind.

Für IEEE 754 double precision Zahlen erzeugt diese Einheit auch noch die sogenannte `sub_context` Nummer. Dieses 4 Bit breite Signal würde bei den in Abschnitt 2.6 vorgestellten Fensterlösungen die Nummer des benötigten IEEE 754 single precision Kontextes für das aktuelle IEEE 754 double precision Produkt (Unter-Kontext des gesamten IEEE 754 double precision Kontextes) angeben. In der vorliegenden Implementierung wird dieses Signal aus oben diskutierten Gründen nicht benötigt, und daher von den Synthese-Werkzeugen wegoptimiert.

activ_lacell aktive Zelle	Addiererzelle ist an der Addition aktiv beteiligt Akkumulation, Einlesen, Negieren
high_lacell obere Zelle	Addiererzelle ist oberhalb der unteren aktiven Zelle Akkumulation
tpt_lacell „transparente“ Zelle	Addiererzelle lädt nicht den gespeicherten Wert, sondern eine Null Negieren, Initialisieren

Tabelle 4.4: Steuersignale für Addiererzellen

4.7 Multiplizierer

Bei dem in unserem Entwurf verwendeten Multiplizierer handelt es sich um einen 32×32 Bit Multiplizierer von Xilinx. Die Firma Xilinx bietet für ihre FPGAs optimierte Module verschiedener Funktionen an, unter anderem auch Multiplizierer.

Unsere Wahl fiel auf eine vorgefertigte Einheit, da es heutzutage wohl in jeder Zieltechnologie speziell dafür optimierte Multiplizierer gibt.

Wichtig bei der Wahl des Multiplizierers ist die Größe. Für ausschließlich IEEE 754 single precision Produkte würde ein 24×24 Bit großer Multiplizierer ausreichen. Für IEEE 754 double precision Produkte die durch 4 Partialprodukte berechnet werden wird eigentlich nur ein 27×27 Bit großer Multiplizierer benötigt, da IEEE 754 double precision Mantissen 53 Bit groß sind und somit die benötigten Partialprodukte 27×27 bzw. 26×27 Bit breit wären. Da dafür aber schon beim ersten Partialprodukt mindestens 26 Mantissenbits benötigt werden, wäre die von uns gewählte Datenübergabe aufgeteilt in die oberen und unteren 32 Bit der IEEE 754 double precision Zahl nicht möglich, da bei Übergabe der oberen 32 Bit nur 21 Mantissenbits vorhanden sind. Natürlich könnten erst die unteren 32 Bit übertragen werden, so daß nach anschließender Übertragung der oberen 32 Bit für jedes Partialprodukt die benötigten Mantissenbits vorhanden wären. Dabei ergäbe sich aber das „Problem“, daß der Exponent des ersten Partialproduktes erst ein Takt später bestimmt werden kann. Aus diesen Gründen haben wir uns für einen 32×32 Bit Multiplizierer entschieden. Der einzige Nachteil bei dieser Wahl ist der größere Ressourcenverbrauch.

4.8 Selektier-Einheit

Die grundlegende Aufgabe der Selektier-Einheit (`Selektor`) ist es die direkte Übergabe der Eingangsdaten in den Summationsmatrix-Kern zu ermöglichen. Diese Möglichkeit ist zur direkten Belegung des langen Akkumulators, zum Beispiel zum Laden eines vorher ausgelagerten Kontextes, notwendig. Die Selektier-Einheit wird immer dann gebraucht, wenn nicht das Produkt des Multiplizierers sondern etwas anderes an den Summationsmatrix-Kern weitergegeben werden soll, wie dies auch bei den Addier-Befehlen der Fall ist. Die Selektier-Einheit ist nichts weiteres als ein 64 Bit breiter 4:1 Multiplexer der unter vier Möglichkeiten auswählt.

1. Eingangsdaten: für das Laden von Kontexten
2. eine single bzw. double Zahl: für Addierbefehle
3. Produkt: für Akkumulationsbefehle
4. Nullprodukt: für alle anderen Befehle

4.9 Schiebe-Einheit / Shifter

Die Schiebe-Einheit (`Shifter`) nimmt den 64 Bit Ausgang der Selektier-Einheit und schiebt (*engl. shift*) diesen um bis zu 127 Bit nach links, wobei links herausfallende Bits rechts wieder reingeschoben werden. Bei den Befehlen mit denen ein Kontext in den langen Akkumulator geschrieben wird, ist die Shiftweite entweder 0 oder 64 Bit, je nachdem ob der rechte oder der linke Addierer einer Summationsmatrix-Zeile beschrieben werden soll. Bei allen anderen Daten hängt die Shiftweite von den Exponenten der Eingabedaten ab. Dies gilt auch für das Nullprodukt das je nach Eingabedaten geshiftet wird, obwohl es gar nicht notwendig wäre. Der Vorteil bei

dieser Vorgehensweise ist, daß im Befehlswort bestimmt werden kann, welcher der vier Eingänge der Selektier-Einheit auf den Eingang der Schiebe-Einheit geschaltet werden soll.

Die Schiebe-Einheit ist also ein 64 zu 128 Bit Links-Rotations-Shifter.

4.10 Zeilenregister

Die letzte Einheit vor der eigentlichen Summationsmatrix ist das 128 Bit breite Zeilenregister. Dieses Register ist direkt mit dem Ausgang des Summationsmatrix-Kerns verbunden, so daß eine 128 Bit breite Zeile eines beliebigen Kontextes (samt Vorzeichen, also 129 Bit) in das Register geladen und im nächsten Takt auf eine beliebige Zeile eines Kontextes addiert werden kann. Dadurch kann z.B. ein IEEE 754 single precision Kontextes dupliziert (kopiert) werden, oder zwei exakte single Skalarprodukte werden addiert bzw. subtrahiert. Um zum Beispiel den Kontext CA vom Kontext CB zu subtrahieren, sind die folgenden 12 Befehle notwendig, dabei bedeutet CBX bzw. CAX die X-te Zeile des Kontextes CB bzw. CA:

1. Negiere Kontext CA
2. Lade CA1 ins Zeilenregister
3. Addiere Zeilenregister zu CB1
4. Lade CA2 ins Zeilenregister
5. Addiere Zeilenregister zu CB2
6. Lade CA3 ins Zeilenregister
7. Addiere Zeilenregister zu CB3
8. Lade CA4 ins Zeilenregister
9. Addiere Zeilenregister zu CB4
10. Lade CA5 ins Zeilenregister
11. Addiere Zeilenregister zu CB5
12. Negiere Kontext CA

Nach diesen zwölf Befehlen (die auch nur 12 Takte benötigen) steht in Kontext CB die Differenz von CB−CA, und der Kontext CA ist (wieder) unverändert. Dies gilt natürlich nur, wenn zwischenzeitlich keine anderen Befehle auf die Kontexte CA und CB angewandt wurden. Da wir mit zehn 64 Bit Addierern für ein IEEE 754 single precision Skalarprodukt anstatt der notwendigen 554 Bit mit 640 Bit über 80 Schutzbits haben, tritt bei dieser Operation kein Überlauf des Akkumulators auf. Die 80 Schutzziffern reichen für über $1023^8 \geq 10^{24}$ Takte, was selbst bei einer Frequenz von 1000 GHz (= 1 Terra-Hertz = 10^{12} Hz) über 30000 Jahre dauern würde. Dies gilt natürlich für alle Operationen in einem 640 Bit großen Akkumulator für IEEE 754 single precision Zahlen.

Dieses Zeilenregister wird ebenfalls benötigt, um einen IEEE 754 double precision Kontext mit mehreren IEEE 754 single precision Kontexten zu realisieren. Je nachdem, welches Fensterkonzept (z.B. dynamischen Konzept) realisiert wurde, ist noch ein Zurechtschieben der Daten notwendig. Dieser Shift könnte bei entsprechender Steuerlogik und Beschaltung auch von der schon vorhandenen Schiebe-Einheit vorgenommen werden.

4.11 Addiererzelle

Als nächstes wollen wir die wichtigste Komponente des Summationsmatrix-Kerns vorstellen, die Addiererzelle (*lacell = local adder cell*). Die Addiererzelle besteht aus einem 64 Bit breiten Addierer/Subtrahierer, den wir im weiteren Verlauf nur als Addierer bezeichnen wollen, mit zweimal 64 Bit breiten Eingängen und dem 64 Bit breiten Kontextspeicher mit vier Addressleitungen und daher 16 verschiedenen Kontextspeicherplätzen. Der 64 Bit breite Addierer kann auch optional aus zwei identischen 32 Bit Addierern aufgebaut sein. Die Addierer (64 oder 32 Bit) sind genauso wie der Multiplizierer von Xilinx optimierte Module.

Aufgrund zweier unterschiedlicher Konzepte die Überträge zwischen den Addierern aufzulösen bzw. zu behandeln, benötigen wir auch zwei etwas unterschiedliche Addiererzellen. Die Grundaddierer sind aber gleich. Bei den Addierern ist zu berücksichtigen, daß sie negative Zahlen im sogenannten Zweierkomplement darstellen. Das bedeutet natürlich, daß ein negativer Inhalt eines Kontextes für die weiter Verarbeitung eventuell erst wieder in die Vorzeichen-Betrag-Darstellung gewandelt werden muß. Genau dafür ist der Befehl „bedingtes Negieren“ zuständig.

Die beiden Übertrags-Konzepte wollen wir im weiteren Verlauf als CSA (**C**arry **S**elect **A**dder) und SCA (**S**tore **C**arry **A**dder) bezeichnen. Dabei werden wir der Lesbarkeit wegen von CS- bzw. SC-Addierern sprechen.

Gemeinsamkeiten beider Addiererzelltypen

Als erstes wollen wir die gemeinsamen Eigenschaften der beiden Addiererzell-Typen aufzeigen. Wie wir schon in Tabelle 4.4 gesehen haben, wird das Verhalten der Addiererzelle von der Exponenten-Arithmetik-Einheit gesteuert. Aufgrund der Signale `activ_lacell` und `high_lacell` von der Exponenten-Arithmetik-Einheit weiß jede Addiererzelle, ob sie das Produkt (oder einen Teil davon) akkumulieren muß, also eine **aktive** Zelle ist, oder sich darüber oder darunter befindet. Bei den möglichen zwei aktiven Zellen wissen diese auch, ob sie die höher- oder niederwertige der beiden Zellen sind. Jede Zelle ist also genau von einem der in Tabelle 4.5 vorgestellten vier Zelltypen.

Die Steuersignale der Exponenten-Arithmetik-Einheit bestimmen auch, welche Daten an die zwei 64 Bit Eingänge *op1* und *op2* des Addierers gelangen sollen. In Tabelle 4.6 bedeutet *NULL* wieder 64 Null-Bits, *OLD* steht für die im Kontextspeicher der aktuellen Addresse befindlichen Daten und *DATA* für die vom Shifter erhaltenen Daten. Im Subtraktionsfall führt der Addierer $op1 - op2$ aus.

Jede Addiererzelle bestimmt auch, ob alle 64 Bit ihres aktuellen Ergebnisses Null oder Eins sind und speichert dies in zwei Bits (`all_zero, all_one`) zusätzlich zu den 64 Datenbits im Kontextspeicher ab. Das bedeutet also, daß der sich lokal

Zelltyp	Beschreibung
`cell_up`	oberhalb der aktiven Zellen
`cell_high`	obere aktive Zelle
`cell_low`	untere aktive Zelle
`cell_down`	unterhalb der aktiven Zellen

Tabelle 4.5: Mögliche Typen der Addiererzellen

Befehle	*op1*	*op2*
Initialisiere Kontext	*NULL*	*NULL*
Addition/Subtraktion aktive Zellen	*OLD*	*DATA*
Addition/Subtraktion nicht aktive Zellen	*OLD*	*NULL*
Kontext laden	*NULL*	*DATA*
Negiere Kontext	*NULL*	*OLD*

Tabelle 4.6: Operanden der Addierer

bei jeder Addiererzelle befindliche 16 „Worte“ umfassende Kontextspeicher nicht 64 sondern 66 Bit breit ist.

Wenn ein Lesebefehl an eine Addiererzelle angelegt ist, so gibt sie den Inhalt ihres Kontextspeichers der aktuellen Addresse an den 66 Bit breiten Zellenausgang weiter. Sobald der Lesebefehl beendet ist, wird der Addiererzellen-Ausgang wieder hochohmig (*tristate*). Dieses Verhalten wird für die später vorgestellte Buslösung benötigt, da dort die Ausgänge von jeweils fünf Addiererzellen verbunden werden. Da die Ausleselogik aber sicher stellt, daß immer nur eine dieser fünf Zellen ihre Daten auf die Ausgänge gibt, können keine Konflikte entstehen.

Carry Select Addiererzelle

Bei den CS-Addierern haben wir nicht nur einen 64 Bit Addierer/Subtrahierer sondern gleich zwei davon. Dabei wird der Übertragseingang (*carry in*) des einen Addierers auf logisch Null, der Eingang des anderen auf logisch gesetzt. Dadurch werden jeweils zwei Ergebnisse berechnet. Durch diese zwei Ergebnisse erhalten wir auch

	Addition	Subtraktion
1. Addierer	$op1 + op2$	$op1 - op2 - \mathbf{1}$
2. Addierer	$op1 + op2 + \mathbf{1}$	$op1 - op2$

zwei Übertragsausgänge. Wie wir weiter oben schon gesehen haben, ist im Entwurf der Aufbau eines 64 Bit Addierers durch zwei 32 Bit Addierer möglich, so daß wir von der Addiererseite her nicht nur zwei sondern vier Übertragssignale haben. Diese Signale werden zu drei Signalen in Abhängigkeit des Zelltypus umgesetzt. Die drei Übertragssignale sind:

`co_low`: wird nur von der unteren aktiven Zelle erzeugt, falls bei ihr ein Übertrag (*carry*) auftritt

`co_high1`: wird nur von der oberen aktiven Zelle erzeugt, wenn bei ihr durch ein Übertrag der unteren Zelle ein Übertrag (*carry*) auftreten würde

`co_high2`: wird nur von der oberen aktiven Zelle erzeugt, falls bei ihr auf alle Fälle ein Übertrag (*carry*) auftritt

Tabelle 4.7 zeigt, wie diese Signale aus den vier Addiererüberträgen und den Zelltypen aus Tabelle 4.5 erzeugt werden. Dabei bedeutet `c00` Übertragsausgang (das c) des unteren 32 Bit Addierers (erste Null), der den 64 Bit Addierer mit Übertragseingang gleich Null (zweite Null) bildet. Nach diesem Vorgehen bedeutet `c11` Übertragsausgang (das c) des oberen 32 Bit Addierers (erste Eins), der den 64 Bit Addierer mit Übertragseingang gleich Eins (zweite Eins) bildet. Sollte der 64 Bit Addierer aus einem 64 Bit und nicht aus zwei 32 Bit Addierern bestehen, so wird `c00` konstant auf Null und `c10` konstant auf Eins gesetzt. Ein `x` in der Tabelle bedeutet, daß dieser Wert für die Erzeugung des entsprechenden Übertrages nicht relevant ist. Für jeden Übertrag gibt es pro Operation mehr als eine Zeile, dabei sind die Zeilen als logisch „ver**oder**t“ zu lesen. Aus den drei Signalen `co_low`,

Übertrag	Zelltyp	Operation	c00	c01	c10	c11
co_low	cell_low	Addition	1	x	x	1
			x	1	x	x
		Subtraktion	x	0	x	0
			0	0	0	1
co_high1	cell_high	Addition	x	x	1	1
			x	1	0	x
		Subtraktion	1	1	1	1
			0	0	1	1
co_high2	cell_high	Addition	1	x	x	1
			x	1	x	x
		Subtraktion	x	0	x	0
			0	0	0	1

Tabelle 4.7: Übertragsbildung einer CSA-Zelle

`co_high1` und `co_high2` aller 10 Addiererzellen wird dann der eigentliche Übertrag `carry` erzeugt, der an jeder Addiererzelle anliegt. Die Ausgänge `co_low` werden außerdem jeweils direkt an die darüber liegende Zelle weitergegeben. Nachdem alle Addierer ihre Additionen beendet haben und der eigentliche Übertrag `carry` erzeugt wurde, kann jede Addiererzelle anhand des `carry` Signals (und eventuellen `ci_low` Signals), der lokalen Überträge und ihres Zelltypes das richtige der zwei vorhandenen Ergebnisse auswählen und abspeichern.

Aufgrund der Tatsache, daß bei dieser Addiererart das Ergebnis in Abhängigkeit der Überträge (*carry, carries*) ausgewählt (*select*) wird, heißt diese Version

Carry**S**elect-Addierer(zelle). Anstatt die Überträge in jedem Takt aufzulösen, was den Vorteil hat, daß nach jedem Takt das endgültige Ergebnis vorliegt, können die Überträge auch gespeichert und im nächsten Takt verarbeitet werden. Dies führt uns zu der SC-Addiererzelle.

Store Carry Addiererzelle

Die SC-Addiererzelle benötigt für die Additions- bzw. Subtraktionsüberträge pro Kontext noch einen 4 Bit Speicher. Da die Addierer bei der Subtraktion im Zweierkomplement rechnen, brauchen wir für die Subtraktion einen anderen Übertragsspeicher als für die Addition. Der Übertrag bei der Subtraktion heißt borrow (*engl: borgen*). Bei der Subtraktion bedeutet eine Eins am Übertragsausgang es tritt **kein** *borrow* auf, eine Null hingegen bedeutet es tritt ein *borrow* auf. Dies ist gerade umgekehrt wie bei der Addition, bei der eine Null bedeutet es tritt **kein** *carry* auf, eine Eins wiederum bedeutet es tritt ein *carry* auf.

Der Vorteil der Store Carry Addiererzelle ist, daß nach der Addition bzw. Subtraktion sofort das Ergebnis und die Überträge gespeichert werden können, ohne erst eine Übertrags- und Auswahllogik dazwischen zu schalten. Außerdem benötigen wir nur einen 64 Bit Addierer gegenüber den zweien bei der CS-Addiererzelle. Dafür müssen bei dieser Lösung bevor das Ergebnis vorhanden ist erst alle Überträge aufgelöst werden. Das kann im Extremfall bedeuten, daß bei k Einzeladdierern (**nicht** Addiererzellen!) $k-1$ Takte notwendig sind um alle Überträge vorheriger Subtraktion und Addition aufzulösen. Desweiteren ist eine Logik notwendig, die die notwendigen Null-Additionen und Subtraktionen durchführt. Sollten zwischen zwei Addierern sowohl ein *carry* als auch ein *borrow* auftreten, so heben sich diese gegenseitig auf. Es sind also wirklich höchsten $k-1$ Nachtakte notwendig, wobei jeder dieser Nachtakte ein Additions- oder ein Subtraktionstakt sein kann. Um die dazu notwendige Logik zu umgehen, können auch einfach immer $k-1$ Additions- und $k-1$ Subtraktionsnachtakte vor dem Auslesen des Inhaltes erfolgen.

4.12 Summationsmatrix-Kern

Der Summationsmatrix-Kern oder ab jetzt die Summationsmatrix genannt, ist die Anordnung eines langen Akkumulator in Matrixform (siehe 2.1.3). Wir stellen hier die Umsetzung einer solchen Summationsmatrix mit einer Zeilenbreite von 128 Bit und 5 Zeilen, aufgebaut aus jeweils zwei 64 Bit Addiererzellen, vor. Die wichtigste Funktion dieser Entwurfseinheit ist das Versorgen der zehn Addiererzellen mit den Eingabedaten und den Steuersignalen. Desweiteren werden in dieser Einheit die Signale für das Abspeichern des Ergebnisses in den Kontextspeicher der Addiererzellen erzeugt. Auch die Weitergabe des Kontextspeicherinhaltes bei den Auslesebefehlen erfolgt an dieser Stelle.

Der Übertragsausgang der höchsten Addiererzelle bestimmt zusammen mit dem abgespeicherten Wert des Kontextvorzeichens das aktuelle Kontextvorzeichen und einen eventuellen Kontextüberlauf. Wie wir schon in Abschnitt 4.10 gesehen haben, ist das Überlaufen eines IEEE 754 single precision Kontextes durch normale Operationen nicht möglich, selbst bei einem IEEE 754 double precision Fensterkontext haben wir noch mindestens $640-512-64+1=65$ Bits als Schutzbits übrig. Diese

Anzahl Schutzbits würde bei einer wie in 4.10 angenommenen Arbeitsfrequenz von 1000 GHz immer noch über ein Jahr benötigen um einen Überlauf zu erzeugen. Diese beiden Bits werden zusammen mit den zehn Bits des sogenannten `msw_lacell` in einem zusätzlichen Kontextspeicher abgelegt.

Die zehn Bits `msw_lacell` sind bis auf zwei immer alle Null. Die zwei von Null verschiedenen (aufeinanderfolgenden) Bits zeigen an, in welchen beiden Addierern die höchstwertigen Bits des aktuellen Kontextes liegen. Diese Information wird aus den jeweils zehn Signalen `all_zero` und all_one, unter Verwendung des aktuellen Vorzeichens des Kontextes, bestimmt. Damit wird ein einfacher Lesebefehl „read msw“ möglich, der immer mindestens die 65 signifikantesten Bits ausgibt. Die Summationsmatrix gibt 128 Bit des aktuellen Kontextes, und die Information, ob die 64 oberen bzw. unteren Bits davon alle Null bzw. Eins sind, zusammen mit einem sogenannten `sticky` Bit aus. Das `sticky` Bit gibt darüber Auskunft, ob unterhalb der ausgegebenen 128 Bits (davon sind mindestens 65 relevant) noch weitere Bits gesetzt sind. Mit diesen Daten ist es möglich, jede der im IEEE 754 Standard vorkommende Rundung zu realiseren.

Jetzt ist eigentlich nur noch offen, wie die Eingangsdaten (Produkt, ...) an die Addiererzellen gelangen.

4.12.1 Transferregister-Lösung

Die einfachste Möglichkeit den einzelnen Matrixzeilen die Eingangsdaten zukommen zu lassen ist, die Daten an ein 128 Bit breites Register vor der niederwertigsten Matrixzeile anzulegen. Die Ausgänge dieses Registers führen zum einen an den Dateneingang der Addiererzellen dieser Matrixzeile, zum anderen an die Eingänge eines äquivalenten Registers vor der nächsthöheren Matrixzeile. Jeder Addierer der ersten Zeile entscheidet dann aufgrund der Steuersignale ob die Daten für ihn bestimmt sind oder nicht. Im nächsten Takt werden die Daten der ersten Zeile von den Registern der zweiten Zeile übernommen. Während die Addiererzellen der zweiten Zeile überprüfen, ob die Daten aus dem ersten Takt für sie bestimmt sind, verarbeiten die Addiererzellen schon wieder die nächsten an ihr Register angelegten Daten. Dieser Mechanismus zieht sich durch alle Zeilen durch, so daß jede Zeile in jedem Takt andere Daten (vielleicht auch einen anderen Kontext) bearbeitet bzw. überprüft. Bei einem Lesebefehl übergibt die Matrixzeile anstatt der Eingangsdaten ihre Ausgangsdaten an die nächste Zeile weiter. Dadurch gelangen diese Daten nach mehreren Takten an den Ausgang der letzten (höchstwertigen) Matrixzeile und können von dort aus nach außen weitergegeben werden.

Aufgrund der Übertragungsregister (Transferregister) vor jeder Zeile bezeichnen wir diese Anordnung als Transferregister-Lösung. Eine ausführliche Betrachtung dieses Modells ist in [15] zu finden. Der Vorteil dieser Anordnung ist der, vor allen Dingen für einen FPGA-Entwurf, daß nur von einer Matrixzeile bis zur nächsten Verbindungen notwendig sind. Die Transferregister sind bei dem von uns benutzten Virtex FPGA in Form der Speicherelemente in jedem Slice (siehe Abbildung 3.2) schon „kostenlos“ vorhanden. Der Nachteil dieser Lösung ist, daß im Extremfall jede Matrixzeile in jedem Takt an einem anderen Produkt bzw. an einem Kontext arbeitet. Dadurch ist nur die Verwendung von SC-Addiererzellen sinnvoll.

Durch die Wahl dieser Addiererzellen haben wir auch die Nachtakte für die

Übertragsauflösung bei der Transferregister-Lösung erhalten. Zusätzlich wird bei dieser Anordnung auch noch die Bestimmung der `msw_lacell`, also die Bestimmung der höchstwertigen Bits eines Kontextes, erschwert. Dazu können am einfachsten mehrere Nachtakte (z.B. in Form von Null-Produkten) zur Übertragsauflösung und anschließend ein Takt zur Bestimmung von `msw_lacell` mit anschließendem Auslesen der entsprechenden Addierer erfolgen. Wenn k die Anzahl der Matrixzeilen ist, so können durch die Nachtakte für die Übertragsauflösung und die $k-1$ Takte bis das Ergebnis am Ausgang anliegt, mehr als $2k$ Takte benötigt werden, um die erste Matrixzeile eines Kontextes auszulesen. Damit benötigen wir für das Beispiel aus Abschnitt 4.10 (Subtraktion zweier Kontexte) nicht nur die angegeben 12 Takte. Bei dem obigen Beispiel gingen wir implizit von einer CSA-Bus-Lösung (siehe 4.12.2) aus.

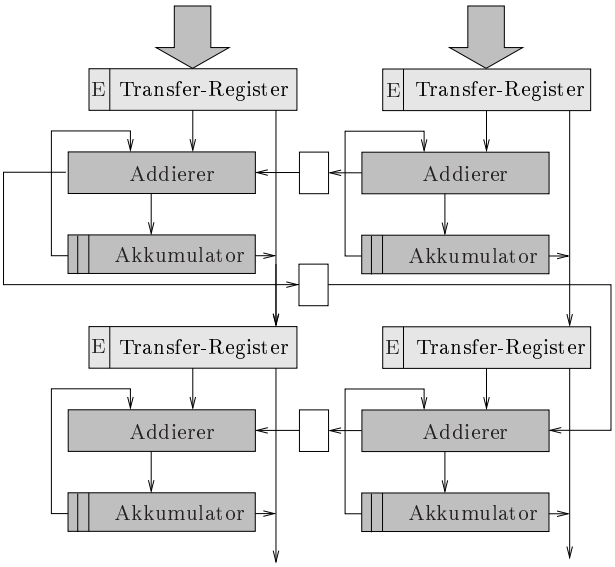


Abbildung 4.2: Die Summationsmatrix mit Transfer-Registern

Wir haben wieder 5 Matrixzeilen mit jeweils zwei Addiererzellen. Daher benötigen wir zur Übertragsauflösung bis zu 9 Takte. Danach mindestens einen Takt für die Bestimmung von `msw_lacell`. Im nächsten Takt können wir den Kontext `CA` negieren. Dies geschieht zeilenweise in insgesamt fünf Takten, aber nach einem Takt ist `CA1` (erste Zeile von Kontext A) negiert. Jetzt können wir alle fünf Zeilen auslesen. Sobald der Lesebefehl für die erste Matrixzeile sich in der Summationsmatrix befindet, können wir wieder mit dem Negieren von Kontext A beginnen. Nach fünf Takten erhalten wir die erste Zeile von Kontext A am Ausgang des Summationsmatrix-Kerns so daß wir sie wieder (über das Zeilenregister) in den Kontext B addieren können. Nach weiteren fünf Takten sind alle Matrixzeilen von Kontext A in die Summationsmatrix eingegeben. Nach noch einmal fünf Takten befindet sich in Kontext B die gewünschte Differenz. Kontext A ist durch das zweite Negieren auch wieder in seinem Ursprungszustand. Das heißt, wir müssen 9 Nachtakte + 1 Bestimmungstakt + 1 Negationstakt + 5 Auslesetakte + 1 Negationstakt + 5 Additionstakte = 22 Takte warten, bis sowohl Kontext A als auch Kontext B die gewünschten Daten enthalten. Wir können Befehle für Kontext A schon wieder

nach 17 für Kontext B nach 18 Takten anlegen, da sich danach alle den jeweiligen Kontext betreffenden Befehle schon in der Summationsmatrix befinden. Wir sehen also, daß die Transfer-Register-Lösung zwar Ressourcen spart, aber einiges an Aufwand bei der Ablaufsteuerung benötigt und das Ergebnis erst einige Takte später zur Verfügung stellt.

Um den Inhalt einer Addiererzelle schneller, nämlich genau in dem Takt in dem der Lesebefehl an ihr anliegt, zu bekommen, können die Ausgänge der Addiererzellen spaltenweise (jeweils die fünf linken und die fünf rechten) zu einem Bus zusammengefaßt werden. Damit reduziert sich in der obigen Rechnung die Taktanzahl um 4, da sobald wir den Lesebefehl an der ersten Matrixzeile anliegen haben, wir im nächsten Takt den Inhalt von Kontext A (über das Zeilenregister) wieder zur Addition in den Kontext B an die Summationsmatrix anlegen können.

4.12.2 Buslösung

Wenn wir nicht nur die Ausgänge sondern auch die Eingänge an einen Bus legen, das heißt alle Addiererzellen bekommen gleichzeitig in jedem Takt die gleichen Daten, so sprechen wir von einer Buslösung.

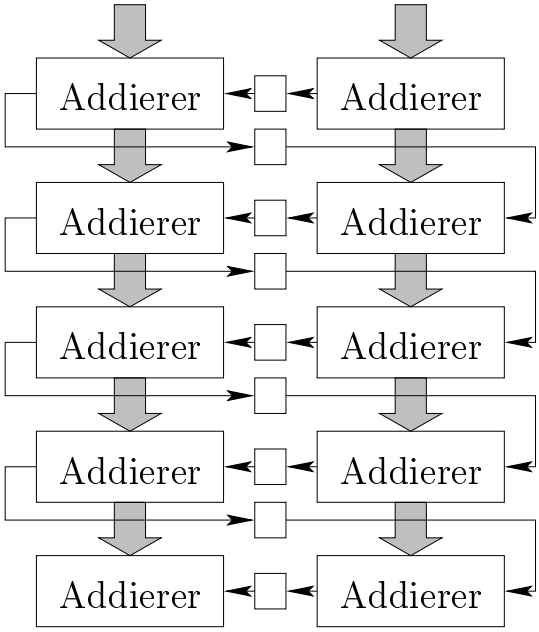


Abbildung 4.3: Die Summationsmatrix mit Busanbindung

Store-Carry Lösung

Eine Buslösung mit SC-Addiererzellen hat gegenüber der Transfer-Register-Lösung den Vorteil, daß die gesamte Summationsmatrix in jedem Takt den gleichen Kontext und die gleiche Operation bearbeitet. Damit fällt die „Wartezeit“ bis ein Befehl zu seiner entsprechenden Matrixzeile gelangt ist weg. Dies kann sehr wohl bei Matrixformen die mehr als unsere 5 Zeilen tief sind von Vorteil sein. Der andere

wichtige Vorteil ist, daß die Transfer-Register (in unserem Fall immerhin 640 Ein-Bit-Register) wegfallen. Der Wegfall dieser Register wird aber durch einen erhöhten Aufwand an Verbindungsleitungen erkauft. Wir dürfen nicht vergessen, daß jede Addiererzelle sowohl an einem 64 Bit (dazu kommen noch circa 10 Bit Steuersignale und ein 4 Bit breiter Kontextaddress-Bus) breiten Ein- als auch an einem 64 Bit (+2 Bit `all_zero, all_one`) breiten Ausgangsbus liegt. Diese Verbindungsleitungen können je nach verwendeter Technologie um einiges teurer (aufwendiger, mehr Platz notwendig) sein als die zusätzlichen Register.

Wie wir schon in Abbildung 3.2 gesehen haben, bekommen wir bei den Xilinx Virtex FPGAs die Transfer-Register „geschenkt“. Geschenkt bekommen wir sie in sofern, als daß sie sowieso vorhanden sind, wenn wir sie nicht verwenden liegen sie einfach brach. Die zusätzlichen Verbindungen über alle Matrixzeilen hinweg, anstatt von einer Zeile zur nächsten, fallen in einem FPGA (und damit auch bei den Mitgliedern der Virtex Familie) stark ins Gewicht. Sobald wir aber in einem FPGA die Verbindungsressourcen (*routing resources*) stark ausnutzen, geht dies auf Kosten der erreichbaren Taktzeit.

An dieser Stelle möchten wir noch einmal anmerken, daß wir unseren Entwurf zwar in einem FPGA auf seine Funktion getestet haben da dies eine kosten- und (Arbeits-)Zeit günstige Methode ist, wir haben unseren Entwurf **nicht** speziell für FPGAs (oder eine andere Technologie) erstellt. Viel mehr war unser Ansinnen die unterschiedlichen Topologien zu testen und zu vergleichen.

Ihre Stärke spielt die Buslösung erst dann aus, wenn wir anstatt der Store-Carry-Addiererzellen die größeren Carry-Select-Addiererzellen verwenden.

Carry-Select Lösung

Wenn wir die gesamte Matrix aus Carry-Select-Addiererzellen aufbauen, so haben wir zwar die doppelte Anzahl an Addierern und das aufwändige Routing der Buslösung, dafür brauchen wir keinerlei Nachtakte zur Übertragsauflösung. Natürlich ist die Carry-Select Lösung immer langsamer als die anderen beiden Lösungen, selbst bei optimaler Verbindung, da bei CS-Addierern vor dem Abspeichern noch die Auswahl des richtigen Ergebnisses erfolgen muß.

In Tabelle 4.8 haben wir noch einmal die Unterschiede der drei Lösungen aufgezeigt. Dabei ist bei der Addiereranzahl der CS-Lösung aber zu berücksichtigen, daß ein Carry-Select-Addierer nur in unserem FPGA den doppelten Aufwand (also zwei anstatt einem Addierer) bedeutet. In anderen Technologien (z.B. ASIC) benötigt ein CS-Addierer nur das 1,5-fache an Logik (Transistoren und Platz) eines „normalen“ Addierers. Dann ist der Logikbedarf der Carry-Select-Lösung ungefähr so groß wie der der Transfer-Register-Lösung. Die Aussage über die Takthöhe soll nur zeigen, daß die erste Lösung die schnellste ist, bei ausreichenden Routing-Ressourcen ist die zweite Lösung annähernd oder genauso schnell wie die erste. Die dritte Lösung ist aus den oben genannten Gründen immer etwas langsamer als die anderen beiden Lösungen. Im nächsten Abschnitt wollen wir uns diesen Umstand noch einmal etwas genauer ansehen.

	Logikbedarf	Routing	Takt	Pipeline-Tiefe
Transfer Register	640 Addierer 640 Register cb-Speicher	hauptsächlich zwischen zwei Matrixzeilen	hoch	groß
Store-Carry (Buslösung)	640 Addierer 0 Register cb-Speicher	über die ganze Matrix	mittel bis hoch	beim Auslesen Nachtakte notwendig
Carry-Select (Buslösung)	1280 Addierer 0 Register kein cb-Speicher	über die ganze Matrix	mittel	1

Tabelle 4.8: Vergleich der Summationsmatrix-Lösungen

4.13 Zeitverhalten

Wir wollen uns jetzt den zeitlichen Ablauf der einzelnen Aktionen für eine Akkumulation ansehen. Dabei betrachten wir nur die Abläufe innerhalb des Summationsmatrix-Kerns. Wir haben eine fünfzeilige Summationsmatrix mit einer Zeilenbreite von 128 Bit wobei jede Zeile aus zwei gleich breiten Addiererzellen besteht. Jede dieser Addiererzellen hat einen Kontextspeicher mit 16 verschiedenen Kontexten.

Als erstes wollen wir einmal die einzelnen Aktionen betrachten, die für den Ablauf einer Akkumulation notwendig sind. Dabei unterscheiden wir nicht zwischen den einzelnen Lösungen. Aktionen, die nur bei bestimmten Lösungen auftreten, werden bei Bedarf als solche gekennzeichnet.

1. Eingangsdaten, Steuersignale und Kontextaddresse liegen an
2. Kontextspeicherdaten liegen an
3. Beginn der Addition
4. Addition ist beendet: Ergebnis(se) und Überträge sind vorhanden
5. Vorzeichen, Überlauf und `msw_lacell` sind erzeugt
6. Carry-Select-Logik ist fertig: endgültiges Ergebnis ist vorhanden
7. Daten liegen am Eingang des Kontextspeichers an
8. Daten sind im Kontextspeicher gespeichert

Wenn alle diese Aktionen beendet sind, kann ein neuer Zyklus (Takt) beginnen. Alle diese Aktionen können kombinatorisch (d.h. sie benötigen keine Taktflanke zur Steuerung) ablaufen, mit einer Ausnahme: das Abspeichern der Kontextdaten. Normalerweise braucht ein Speicher einen Taktimpuls um den Speichervorgang auszulösen. Nachdem die Daten eine speicherabhängige Zeit an den Speichereingängen angelegen sind, sind sie sicher in den Speicher übernommen und die Speichereingänge können verändert werden. Der zweite (zeitlich der erste) Taktimpuls, den wir benötigen, ist der Anfang (gleichzeitig das Ende des vorhergehenden) eines solchen Zyklus.

Wir benötigen also für die Steuerung unserer Summationsmatrix zwei Ereignisse (Impulse) die zeitlich nacheinander auftreten.

Die einfachste Möglichkeit dabei ist, dafür die ansteigende und die abfallende Taktflanke unseres Taktes zu verwenden.

4.13.1 Zeitverlauf bei SCA-Bus-Lösung

Jetzt wollen wir die Aktionen für die SCA-Bus-Lösung noch einmal genauer betrachten:

t_0	Eingangsdaten, Steuersignale und Kontextaddresse liegen an
t_1	Kontextspeicherdaten liegen an
t_2	Beginn der Addition
t_3	Addition ist beendet: Ergebnis(se) und Überträge sind vorhanden
t_4	Vorzeichen, Überlauf und `msw_lacell` sind erzeugt
t_5	Daten liegen am Eingang des Kontextspeichers an
t_6	Daten sind im Kontextspeicher gespeichert

Tabelle 4.9: Aktionen bei der SCA-Bus-Lösung

Kontextspeicher: t_1, t_6

Die Zeit, die vom Zeitpunkt t_0 bis zum Zeitpunkt t_1 vergeht, wird benötigt, um den Kontextspeicherinhalt (gewünschtes Skalarprodukt, auf das akkumuliert werden soll) der neuen Kontextaddresse (liegt erst bei t_0 an) an die Addiererzellen zu führen. Wenn wir sicher stellen, daß sich die Kontextaddresse zwischen zwei Befehlen nicht ändert, so könnte das gewünschte Skalarprodukt zum Zeitpunkt t_0 schon anliegen, da es ja seit dem Zeitpunkt t_6 des voran gegangenen Taktes schon existiert. Diese Möglichkeit bedingt aber dann, daß ein Kontextwechsel einen extra Takt benötigt, in dem dann die Umschaltung von einem auf den anderen Kontextspeicherinhalt erfolgt. Dieses Vorgehen wird nur bei großen (im Verhältnis zum Takt) Umschaltzeiten des Kontextspeichers eine signifikant kürzere Taktzeit erbringen. Sie hat auch dann keinen Sinn, wenn aufgrund der Benutzung der Summationsmatrix oft ein Kontextwechsel notwendig ist. Im Extremfall (in jedem Takt wird ein neuer Kontext bearbeitet) halbiert sich die Anzahl der ausgeführten Akkumulationen pro Sekunde fast. Zwei „verkürzte“ Takte (ohne t_1) pro Akkumulation gegenüber einem „langem“ Takt (mit t_1).

Die Zeit zwischen t_5 und t_6 ist die Zeit, die der Speicher benötigt, um seine Eingangsdaten sicher abzulegen. Diese Zeitspanne zwischen Anliegen der Ergebnisse zum Abspeichern und Abrufmöglichkeit von gespeicherten Kontextdaten kann durch die Verwendung eines sogenannten „dual ported RAM“ einem Speicher mit getrennten Lese- und Schreibaddress-Eingängen gelöst werden. Dabei sind die Addresse in die die anliegenden Daten gespeichert werden und die Addresse die gespeicherte Daten an die Ausgänge legt von einander unabhängig. Da es ohne weiteres möglich ist, daß wir im nächsten Takt auf die im aktuellen Takt erzeugten Daten zugreifen wollen, hat es natürlich keinen Sinn die Kontextdaten für den nächsten Takt

vor dem Anlegen der aktuellen Daten an den Speicher abzurufen. Wenn Schreib- und Leseaddresse an einem solchen Speicher identisch sind, so liegen am Ausgang „sofort“ die Eingangsdaten an. Die Verzögerung ist minimal und normalerweise nur von den Leitungen innerhalb des Speichers abhängig.

Vorzeichen, Überlauf, ...: t_4

Die Zeit zwischen dem Vorliegen der Ergebnisse (t_3) und dem Abspeichern aller Daten (t_5) kann zu Null verkürzt werden, indem die Daten (Vorzeichen, Überlauf, `msw_lacell`) die bis t_4 erzeugt werden zu einem späteren Zeitpunkt, von den Ergebnissen getrennt, abgespeichert werden. Dazu betrachten wir, wann bzw. für was wir diese Daten überhaupt benötigen. Dabei stellen wir fest, daß alle diese Daten (auch das Vorzeichen, da Kontext im Zweierkomplement abgespeichert wird!) erst zum Auslesen eines Kontextes benötigt werden. Also frühestens zum Anfang des nächsten Taktes (nach der Akkumulation), oder falls noch Nachtakte zur Übertragsauflösung notwendig sind, erst mehrere Takte später. Daher ist es ausreichend, wenn wir diese Daten erst am Ende des Taktes (Beginn des neuen Taktes) abspeichern. Natürlich muß dabei sichergestellt sein, daß auch diese Daten lange genug unverändert an den Speichereingängen anliegen.

Addition: t_2 *bis* t_3

Die Addition der Eingabedaten und Kontextdaten ist der wichtigste und auch am längsten dauernde Vorgang innerhalb eines Taktes unserer Summationsmatrix. Diese Zeit können wir nur durch die Verwendung kürzerer Addierer verringern. Damit erkaufen wir uns aber mehr Übertragsspeicher (nicht besonders dramatisch) und vor allen Dingen mehr Nachtakte. Außerdem ist das Skalieren der Addiererbreite nicht linear, das heißt also nicht, daß wir bei nur halb so breiten Addierern auch nur die Hälfte der Zeit für die Addition benötigen. Dies ist bei dem zum Testen verwendeten Virtex FPGA besonders deutlich. Die Angaben in Tabelle 4.10 sind aus dem Datenblatt von Xilinx [32] und beziehen sich auf die schnellste Version mit *speedgrade* -6.

Addiererbreite	Zeit
16 Bit	5,0 ns
64 Bit	7,2 ns

Tabelle 4.10: Additionszeiten bei einem Virtex-6 FPGA

4.13.2 Zeitverlauf bei CSA-Bus-Lösung

Wie wir gesehen haben, kommt zu den Zeitpunkten in Tabelle 4.9 noch die Auswahllogik (*select logic*) dazu, so daß sich die Tabelle zu der Tabelle 4.11 erweitert. Da die Auswahllogik nur „wenige“ Daten von jeder Addiererzelle benötigt und auch nur ein Ausgangssignal erzeugt, ist diese Zeitspanne relativ kurz. Es werden (je nach Logikimplementierung) circa 5 Gatterlaufzeiten und die Umschaltzeit für den Auswahlmultiplexer benötigt.

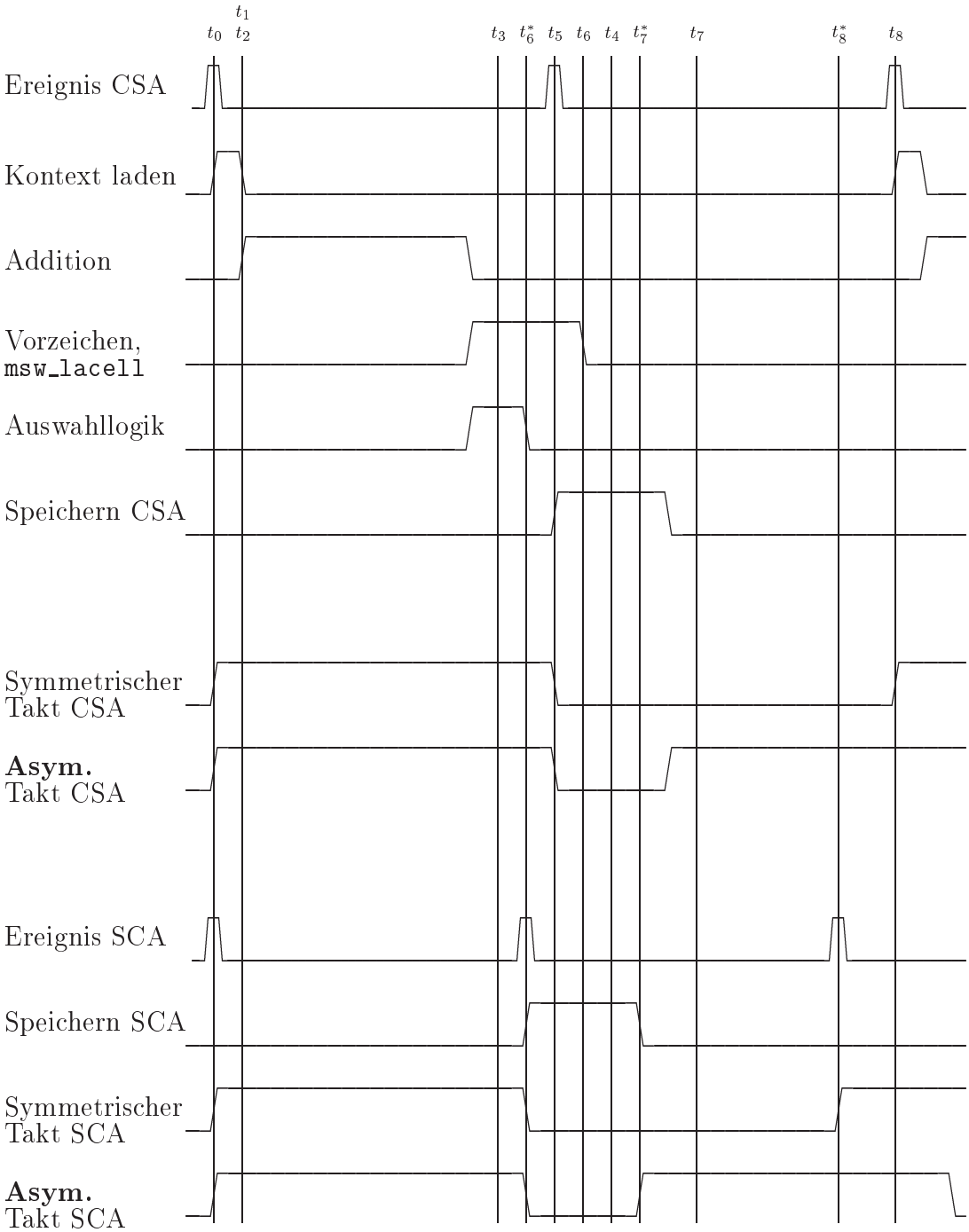


Abbildung 4.4: Zeitpunkte und Takte

Für alle anderen Zeitpunkte gilt das schon im obigen Abschnitt 4.13.1 gesagte.

t_0	Eingangsdaten, Steuersignale und Kontextaddresse liegen an
t_1	Kontextspeicherdaten liegen an
t_2	Beginn der Addition
t_3	Addition ist beendet: Ergebnis(se) und Überträge sind vorhanden
t_4	Vorzeichen, Überlauf und `msw_lacell` sind erzeugt
t_5	Auswahllogik selektiert die richtigen Ergebnisse
t_6	Daten liegen am Eingang des Kontextspeichers an
t_7	Daten sind im Kontextspeicher gespeichert

Tabelle 4.11: Aktionen bei der SCA-Bus-Lösung

4.13.3 Zeitverlauf bei Transfer-Register-Lösung

Die Transfer-Register-Lösung hat, wie wir schon weiter oben gesehen haben, gegenüber der SCA-Bus-Lösung nur den Vorteil der lokalen Verbindungen. Außerdem ist es nicht möglich den Zeitpunkt t_4 in den normalen Ablauf zu integrieren, da jede Addiererzelle innerhalb eines Taktes an einem von den anderen Zellen unabhängigen Kontext arbeiten kann. Wie schon in Abschnitt 4.13.1 bemerkt, verlängert der Zeitpunkt t_4 bei den Buslösungen die Ausführungszeit nicht.

4.13.4 Zeitverlauf

Im Zeit-Diagramm (Abbildung 4.4) sind alle Zeitpunkte wie in Tabelle 4.11 bezeichnet. Für die SC-Addiererlösungen fällt natürlich der Zeitpunkt t_5 weg, so daß nach t_3 und t_4 sofort t_6 folgt. Die Zeilen *Ereignis CSA* und *Ereignis SCA* beschreiben die in Abschnitt 4.13 besprochenen Impulse. Als Impuls zum Speichern verwenden wir dabei die fallende Flanke unseres Taktsignales.

Die symmetrischen Takte sind solche Takte, bei denen die Zeitabstände zwischen steigenden und fallenden Taktflanken immer gleich lang sind. Wenn wir einen Takt mit kurzer Low-Zeit (Zeit zwischen fallender und darauffolgender steigender Taktflanke) verwenden, so bekommen wir einen höheren Takt (kürzere Zykluszeit). Dies resultiert daraus, daß die Addition die zeitaufwendigste Funktion ist. Die Zeitabschnitte in Abbildung 4.4 zeigen keine tatsächlichen Größen (die Auswahllogik braucht nicht zwingend doppelt solange wie das Laden des Kontextes), sondern nur ungefähre Größenordnungen. Natürlich sind die tatsächlichen Zeiten sowohl von der verwendeten Technologie als auch vom Entwurf (*design*) selbst abhängig.

Im Diagramm haben wir auch einen Zeitpunkt t_8 eingeführt. Wie leicht zu sehen ist, bedeutet dieser Zeitpunkt den Beginn des folgenden Taktes (also t_0 vom nachfolgenden Takt). Alle mit * gekennzeichneten Zeitpunkte sind die um $t_5 - t_3$ nach links versetzten Zeitpunkte für die Store-Carry-Lösungen. Bei der zweiten und dritten Zeile (*Kontext laden* und *Addition*) gilt der Beginn dieser Aktionen am Ende der Zeile natürlich nur noch für die CSA-Bus-Lösung mit symmetrischem Takt. Das gleiche gilt für die Zeilen *Ereignis XXX* entsprechend.

Kapitel

5

Ergebnisse

> *Bei allem, was man tut, kommt es auf den richtigen Zeitpunkt und den richtigen Rhytmus an.*

Miyamoto Musashi: Das Buch der fünf Ringe

In diesem Kapitel wollen wir die Ergebnisse der Synthese unseres Entwurfes bzw. seiner Variationen wie Transfer-Register-Lösung, SCA-Bus-Lösung und CSA-Bus-Lösung vorstellen. Obwohl wir, wie schon mehrmals betont, unseren Entwurf **nicht** für irgendeine Technologie optimiert haben, sind die beim Umsetzen in ein Virtex XCV800-4 FPGA gewonnen Ergebnisse von einer gewissen Aussagekraft. Natürlich sind die hier vorgestellten Zeiten mit anderen Technologien oder FPGAs ohne weiteres zu überbieten, aber trotzdem wird eine CSA-Bus-Lösung immer einen niedrigeren Takt als eine Transfer-Register-Lösung ermöglichen.

Auch bei den benötigten Ressourcen darf nicht vergessen werden, daß es Technologie-abhängige Werte gibt. Es sei hier noch einmal darauf verwiesen, daß ein Carry-Select-Addier normalerweise nicht den doppelten sondern nur den anderthalb-fachen Ressourcenbedarf hat (siehe 4.12.2). Trotzdem sind die Angaben in Gatter-Äquivalenten (*GE*) mit Ausnahme des CS-Addierers schon als allgemein gültige Größenordnungen zu verstehen.

Das bedeutet, daß alle Zeitangaben (oder Frequenzen) höchstens im Verhältnis zu den anderen Lösungen (z.B. Transfer-Register-Lösung verglichen mit CSA-Bus-

Lösung) gesehen werden sollten, die Logik-Ressourcen hingegen können (mit obiger Einschränkung) als absolute Aussage gesehen werden. Die FPGA bzw. Virtex-FPGA abhängigen Größen wie Slice-Anzahl und ähnliches können natürlich nur für Entwürfe in diesen Technologien gelten. Alle Angaben, sowohl die Zeit als auch die Logikangaben, sind von der Xilinx Software erstellt.

Wir haben uns nur auf die Xilinx Virtex Familie beschränkt, da wir zum Einen zum Testen einen Vertreter dieser Familie zur Verfügung hatten, zum Anderen die jüngeren Familienmitglieder (Virtex-E) von der Version unserer Software nicht unterstützt wurden. Natürlich ist uns klar, daß die Einschränkungen von Ressourcen (und damit kleine Taktfrequenzen) beim gesamten Entwurf mit einem XCV3200E (Virtex-E mit 104×156 CLBs; circa 30% schneller als die verwendete Familie) nicht vorhanden wären und daher die erreichbare Taktfrequenz tatsächlich nur von der langsamsten Einheit (die Addiererzelle) abhängen würde.

5.1 Zeiten

In der Tabelle 5.1 sehen wir die unterschiedlichen Frequenzen, die die einzelnen Entwurfs-Einheiten in Virtex FPGAs erzielen. Die Angabe des Bausteines ist in sofern von Bedeutung, daß wir versucht haben immer einen kleinen, das heißt sowenig als möglich CLBs beinhaltenden, zu verwenden. Auf der anderen Seite war bei vielen Einheiten ein größerer Baustein notwendig, um die entsprechende Anzahl an Ein- bzw. Ausgangsanschlüssen zur Verfügung zu haben.

Anhand dieser Tabelle kann festgestellt werden, wie schnell diese Einheiten prinzipiell in der verwendeten Technologie (Xilinx Virtex Familie) sind. Um bei dem Summationsmatrix-Kern so viel Verbindungsressourcen wie möglich zu erhalten, haben wir dabei den größten Vertreter (XCV1000) der Virtex Familie gewählt. Wie die Tabelle zeigt erreicht der Summationsmatrix-Kern (Transfer-Register-Lösung; Zeile 11 in Tabelle 5.1) auch ungefähr die gleichen Zeiten wie eine einzelne Addiererzelle (Zeile 10).

Die Eingabe-Einheit wurde bei diesen Untersuchungen aus mehreren Gründen nicht berücksichtigt.

1. sie hat zwei Takte, wobei der PCI-Takt 36 MHz beträgt
2. sie ist nur ein Interface zur eigentlichen Einheit
3. sie besteht hauptsächlich aus Registern

Auch die Ausgabe-Einheit, die nur aus Registern und den dazu gehörigen Verbindungen besteht, wurde nicht als getrennte Einheit berücksichtigt.

Betrachten wir die Taktfrequenzen zwischen dem Speedgrade -4 und dem Speedgrade -6 so stellen wir fest, daß die höhere Frequenz immer ungefähr das 1,27-fache der niedrigeren Frequenz ist. Das heißt also, daß der Baustein in Speedgrade -6 circa 25% schneller als der Baustein mit dem Speedgrade -4 ist (dafür ist er auch um einiges teurer). Wenn wir jetzt noch die Aussage des Datenblattes der Virtex-E Familie hinzu nehmen, die besagt, daß diese Familie circa 30% schneller als die Virtex Familie ist, können wir abschätzen, daß die Einheiten in einem Baustein der Virtex-E Familie mehr als das eineinhalb-fache schneller wären als zum Beispiel

bei einem XCV800-4 BG432. Bei diesen Betrachtungen müssen wir außerdem noch berücksichtigen, daß die angegebenen Frequenzen „worst-case“ Angaben sind, also Frequenzen, die auf alle Fälle erzielt werden.

Nr.	Entwurfs-Einheit	Baustein	Grade -4	Grade -6
0	Dekodier-Einheit	XCV150 BG352	163,29 MHz	210,35 MHz
1	Dekodier-Einheit	XCV150 BG352	91,67 MHz	118,06 MHz
2	Datenauswahl-Einheit	XCV150 BG352	68,01 MHz	87,61 MHz
3	Exp.-Arithmetik-Einheit	XCV150 BG352	24,17 MHz	32,8 MHz
4	komb. Multiplizierer	XCV150 BG352	25,71 MHz	32,73 MHz
5	4-stufiger Multiplizierer	XCV150 BG352	111,73 MHz	146,35 MHz
6	Selektier-Einheit	XCV400 BG560*	79,17 MHz	101,98 MHz
7	Schiebe-Einheit / Shifter	XCV150 BG352	49,08 MHz	63,22 MHz
8	Zeilenregister	XCV400 BG560*	58,55 MHz	75,41 MHz
9	CS†-Addiererzelle	XCV400 BG560*	22,38 MHz	28,84 MHz
10	SC‡-Addiererzelle	XCV400 BG560*	29,22 MHz	37,63 MHz
11	SuMa-Kern TR¶	XCV1000 BG560**	28,85 MHz	36,63 MHz
12	SuMa-Kern§ SC‡	XCV1000 BG560**	18,36 MHz	24,02 MHz
13	SuMa-Kern§ CS†	XCV1000 BG560**	9,87 MHz	12,71 MHz
14	Summationsmatrix#TR¶	XCV800 BG432***	19,5 MHz	25,07 MHz
15	Summationsmatrix#SC‡	XCV800 BG432***	12,84 MHz	16,54 MHz
16	Summationsmatrix#CS†	XCV800 BG432***	6,70 MHz	8,63 MHz

* XCV150 BG352 hatte zu wenig Anschlüsse
† Carry-Save
‡ Store-Carry
** XCV1000 damit genug Verbindungsressourcen vorhanden sind
¶ Transfer-Register
§ Buslösung
# **Gesamter Entwurf** mit Ein-/Ausgabe
*** XCV800-4 BG432 auf der Spyderkarte vorhanden (siehe 3.5.1)

Tabelle 5.1: Frequenzen der Entwurfseinheiten in Virtex FPGA

Betrachten wir die Tabelle 5.1 einmal genauer: als erstes fällt wohl der Frequenzunterschied der Zeile 0 und 1 auf. Der Entwurf von Zeile 0 ist über 1,75 mal so schnell wie der von Zeile 1. Dabei handelt es sich aber nicht um zwei unterschiedliche Designs, sondern ausschließlich um unterschiedliche Platzierungen und daraus resultierend, unterschiedlichen Verbindungen. Die zwei Entwürfe wurden mit unterschiedlichen Zeitbedingungen gestartet, einmal mit 66 MHz (Zeile 1) und das andere mal mit 100 MHz (Zeile 0). Sobald die „place and route“ Software die vorgegebenen Zeiten erfüllt hat, hört sie auf den Entwurf zu optimieren. Dieses Beispiel soll nur zeigen, daß durch intelligentes (eventuell von Hand) Placing und Routing sehr viele Möglichkeiten gegeben sind. Dies trifft vor allen Dingen dann zu, wenn genug Ressourcen dazu vorhanden sind. Diese Problematik ist vor allen Dingen sichtbar wenn man die Zeile 11 mit der Zeile 14 (oder 12 mit 15 bzw. 13 mit 16) vergleicht. In Zeile 11 handelt es sich um den Summationsmatrix-Kern (1636 Slices) in einem XCV1000

(13% Ausnutzung) der bei der Frequenz ziemlich nahe an die Frequenz seiner langsamsten Komponente, der (SC-)Addiererzelle (siehe Zeile 10) herankommt. Bei dem Entwurf in Zeile 14 handelt es sich um den gleichen Summationsmatrix-Kern (andere Platzierung) an den noch die anderen Einheiten (alle schneller!) angeschlossen sind. Dieser Entwurf benötigt 4119 Slices und benutzt damit 43% aller in einem XCV800 vorhandenen Slices. Die langsamere Taktfrequenz resultiert ausschließlich aus der gedrängten Anordnung und den damit verbundenen eingeschränkten Routing-Möglichkeiten. Der gleiche Entwurf würde in einem XCV3200E mit seinen 4119 Slices auch nur circa 13% (von 32448 Slices) benötigen und hätte damit genügend Verbindungs-Ressourcen. Wenn wir jetzt noch berücksichtigen, daß der langsamste Speedgrade, den es bei der Virtex-E Familie gibt, schon -6 ist (der schnellste ist -8) so würden wir in so einem Baustein auch für den Entwurf aus Zeile 14 ungefähr 40 MHz erhalten.

Als letzte Betrachtung wollen wir die langsamsten zwei Einheiten (der kombinatorische Multiplizierer wird natürlich durch den 4-stufigen ersetzt) betrachten. Das ist zum einen die Exponenten-Arithmetik-Einheit in Zeile 3 und die Addiererzellen in Zeile 9 bzw. 10 (die Zeilen 11-16 bauen sich aus Einheiten der Zeilen 1-10 auf). Wenn wir bedenken, daß es sich bei der Exponenten-Arithmetik-Einheit hauptsächlich um einen Addierer handelt, so ist es nicht weiter verwunderlich, daß dieser ähnlich schnell (oder langsam) wie die 64 Bit Addiererzellen ist. Wir hatten schon in Abschnitt 3.3.4 gesehen, daß ein 16 Bit Addierer (Exponenten Addition) nicht viermal so schnell wie ein 64 Bit Addierer ist. Trotz allem müßte der Exponenten Addierer eine viel höhere Frequenz erreichen. Da wir aber vor der Addition erst noch Vergleiche und Auswahllogik haben, erreicht die Exponenten-Arithmetik-Einheit nicht die erwarteten Frequenzen. In einem Entwurf, in dem die Exponenten-Arithmetik-Einheit die langsamste Einheit ist, müßte man sie einfach durch Einführung einer (oder mehrerer) Pipelinestufen beschleunigen. Die Addiererzellen sind durch so eine Entwurfsänderung nicht zu beschleunigen, da die Summationsmatrix ihren Sinn nur dann erfüllt, wenn sie in jedem Takt ein Produkt akkumuliert. Dies ist mit mehrstufigen Addiererzellen nicht möglich.

Diese Betrachtungen sollen nur zeigen, daß die Arbeitsfrequenz eines Entwurfes in einem FPGA sehr stark von den vorhandenen Ressourcen abhängt.

5.2 Logik-Ressourcen

In der Tabelle 5.2 handelt es sich bei den entsprechenden Zeilen um die gleichen Entwürfe wie in der Tabelle 5.1. Natürlich gibt es in dieser Tabelle keine Zeile 0, da sich dieser Entwurf ausschließlich durch ein anderes „Placing and Routing“ und damit Zeitverhalten von dem in Zeile 1 unterscheidet. In der Spalte Slices handelt es sich um die Anzahl der Slices (1 Slice = 2 CLBs, siehe 3.3.4) die benötigt werden. Bei der Spalte FF und LUT um die Anzahl der benutzten Speicherelemente und Look Up Tables (als Logikgeneratoren) der CLBs. Bei der Spalte RAM handelt es sich um die Anzahl der LUT die als 16×1 Speicher verwendet wurden (siehe 3.3.4). Die Anzahl in der Spalte LUT enthält die als Speicher benutzen LUTs aus der Spalte RAM nicht. In der Spalte GE handelt es sich um die Angabe der Gatter-Äquivalente des Entwurfes, wobei die einzelnen Komponenten mit den schon früher besprochenen Anteilen in die Gesamtanzahl eingehen (siehe 3.3.4).

Bei der letzten Spalte handelt es sich um die Prozentzahl aller im benutzten

Baustein (siehe dazu Spalte Baustein in Tabelle 5.1) vorhandenen Slices und damit um eine ungefähre Angabe wieviele Verbindungs-Ressourcen noch vorhanden sind. Dabei ist zu beachten, daß die Buslösungen in Zeile 15 und 16 wie schon in Abschnitt 4.12.2 bemerkt um einiges Verbindungs-Ressourcen intensiver sind als die Transfer-Register-Lösung in Zeile 14.

Nr.	Entwurfs-Einheit	Slices	FF	LUT	RAM	GE	%
1†	Dekodier-Einheit	39	22	40	0	1104	2%
2†	Datenauswahl-Einheit	169	192	332	0	3528	9%
3†	Exp.-Arithmetik-Einheit	108	63	212	0	1960	6%
4†	komb. Multiplizierer	544	0	1008	0	13491	31%
5†	4-stufiger Multiplizierer	568	1088	1008	0	22195	32%
6§	Selektier-Einheit	65	64	130	0	1484	1%
7†	Schiebe-Einheit / Shifter	339	128	674	0	5143	19%
8§	Zeilenregister	139	266	267	0	3730	2%
9§	CS-Addiererzelle	231	0	382	66	11709	4%
10§	SC-Addiererzelle	156	0	242	70	10994	3%
11*	SuMa-Kern TR	1636	13	2528	716	113145	13%
12*	SuMa-Kern SC	1627	13	2519	716	113085	13%
13*	SuMa-Kern CS	2373	13	3924	676	120298	19%
14#	Summationsmatrix TR	4119	3299	5464	713	164294	43%
15#	Summationsmatrix SC	4111	3299	5458	713	164258	43%
16#	Summationsmatrix CS	4909	3306	6969	673	172175	52%
17#	Summationsmatrix CS**	4220	2493	6196	673	161736	44%
18‡	Ein-/Ausgabe-Einheit	689	813	773	0	10439	8%

† XCV150 24×36 CLBs siehe Tabelle 3.4
§ XCV400 40×60 CLBs siehe Tabelle 3.4
# XCV800 56×84 CLBs siehe Tabelle 3.4
* XCV1000 64×96 CLBs siehe Tabelle 3.4
** Ganzer Entwurf **ohne** Ein-/Ausgabe-Einheit
‡ Differenz von Zeile 16 und 17

Tabelle 5.2: Ressourcenverbrauch der Entwurfseinheiten in Virtex FPGA

Wenn wir die Tabelle 5.2 betrachten, so fällt uns auf, daß der gesamte Entwurf (Zeile 16) 3 Speicherblöcke (16×1) weniger hat, als der dazu gehörige Kern (Zeile 13). Dies liegt daran, daß diese 3 Blöcke im Gesamt-Entwurf von der Software „weg“-optimiert werden konnten. Dabei handelt es sich um die Bits 32 bis 30 von *REG3* die wie in Tabelle 4.2 angegeben immer konstant `101` sind. Warum wurden die nur im Gesamtentwurf entfernt? Im Entwurf aus Zeile 13 wird der Ausgang des Kontextspeichers direkt mit Ausgabeanschlüssen des FPGAs verbunden. Beim Entwurf in Zeile 16 wird dieser Ausgang mit dem *REG3* verbunden. Dadurch hat die Software die Möglichkeit festzustellen, daß sich die Eingänge dieses Registers nicht ändern und damit diese drei Speicherzellen redundant sind. Da die Software aber nicht die Möglichkeit hat, einfach drei Ausgänge weg zulassen, wird der Speicher (mit konstantem Inhalt) mit diesen verbunden.

Gegenüber der Tabelle 5.1 haben wir jetzt zwei Zeilen (Nr. 17 und 18) zusätzlich. Der Entwurf der Zeile 17 ist der gleiche wie in Zeile 16, nur mit dem Unterschied, daß die ganze Ein- und Ausgabe-Einheit nicht vorhanden ist. Dadurch war es uns möglich, in Zeile 18, als Differenz von Zeile 16 und 17, den Ressourcenverbrauch der Ein-/Ausgabe-Einheit anzugeben.

Als nächstes wollen wir uns den Ressourcenverbrauch der beiden Multiplizier (Zeile 4 und 5) ansehen. Das auf den ersten Blick erstaunliche daran ist, daß obwohl wir beim 4-stufigen Multiplizierer 1088 Speicherelemente mehr und dadurch auch 65% mehr Gatter-Äquivalente haben als beim kombinatorischen Multiplizierer, sich die Anzahl der verwendeten Slices um nicht einmal 5% erhöht. Das bedeutet, daß der 4-stufige Multiplizierer unser FPGA nur unwesentlich mehr „belastet“, aber wie wir in Tabelle 5.1 gesehen haben das Vierfache an Arbeitsfrequenz ermöglicht. Wenn wir uns noch einmal das Design eines Virtex CLBs vor Augen führen (siehe Abbildung 3.2), so wird uns klar, daß wir auch hier ähnlich wie bei der Transfer-Register-Lösung (siehe 4.12.2) von den in jedem Slice vorhandenen Speicherelement profitieren. Wenn wir stattdessen nur den kombinatorischen Multiplizierer verwenden, so liegen die 1008 Speicherelemente der Slices, deren LUT wir benötigen, brach. Daher haben wir nur einen minimalen Mehraufwand an Slices (+20), um den viermal so schnellen Multiplizierer zu erhalten.

Ein weiterer interessanter Aspekt zeigt sich beim Vergleich der beiden Addiererzellen (Zeile 9 und 10). Obwohl wir bei der Carry-Select-Addiererzelle fast das 1,5-fache an Slices benötigen (231 anstatt 156), erhöht sich die Anzahl der Gatter-Äquivalente gerade mal um 7%. Bei näherer Betrachtung erkennen wir, daß wir für die Store-Carry-Addiererzelle 242 LUTs +70 LUTs (für die 70 RAM) = 312 LUTs benötigen, was genau die Anzahl von LUTs ist, die uns 156 Slices (pro Slice 2 LUTs) zur Verfügung stellen. Das bedeutet, daß die Store-Carry-Addiererzelle alle LUTs ihrer Slices benötigt und nur die Speicherelemente brachliegen. Bei der Carry-Select-Addiererzelle hingegen benötigen wir eigentlich nur $382 + 66 = 448$ LUTs die wir schon in 224 Slices finden. Das PAR-Programm (Place And Route) hat aber zugunsten eines besseren Routings (und daher eines schnelleren Taktes) 14 der Slices nur halb genutzt. Das heißt, bei der Carry-Select-Addiererzelle liegen nicht nur alle Speicherelemente der benötigten Slices sondern auch 14 LUTs brach. Wenn wir jetzt noch bedenken, daß eine LUT, die als Logikgenerator verwendet wird (Spalte LUT), weniger als ein Fünftel (siehe 3.3.4) an Gatter-Äquivalenten erbringt als eine LUT die als Speicher (Spalte RAM) verwendet wird, so ist die geringe Differenz in Gatter-Äquivalenten (7%) gegenüber der großen Differenz an benötigten Slices (48%) bzw. an Logikelementen (Spalte LUT 57%) erklärlich. Bei der Berechnung der Gatter-Äquivalente für Addierer bilden auch immer die in dieser Tabelle nicht aufgeführt speziellen „carry-chains“ zusätzliche Ressourcen (siehe 3.3.4).

Wie wir gesehen haben, sind die Gatter-Äquivalente eine Maßzahl, die ziemlich technologie-unabhängige Aussagen über die Komplexität eines Entwurfes machen. z.B. 22195 GE gegenüber 13491 GE bei den beiden Addierern. Wogegen die Anzahl der benötigten Slices oder CLBs, ausschließlich Aussagekraft haben beim Vergleich zweier Entwürfe innerhalb ähnlich aufgebauter FPGA-Familien. Die 544 benötigten Slices des kombinatorischen Multiplizierers beinhalten schon alle für den 4-stufigen Multiplizierer benötigten Ressourcen (544 Slices = 1088 LUTs und 1088 Speicherelemente = FF).

5.3 Floorplan

Die Abbildung 5.1 veranschaulicht noch einmal die Verteilung des Entwurfes aus Zeile 16 in Tabelle 5.2. Die farbigen Markierungen bezeichnen alle die Slices, die von den entsprechenden Einheiten benutzt werden. In dieser Abbildung ist an der Blockgestalt sehr gut zu erkennen, daß der Multiplizierer ein (von Xilinx) optimiertes Makro ist. Das gleiche erkennt man an den vierzig 32-Bit Addierern (zehn 64-Bit CS-Addiererzellen = zwanzig 64-Bit Addierer = vierzig 32-Bit Addierer) und an den zwanzig 32-Bit Kontextspeichern. Das Wort LA-Zellen in der Legende der Abbildung steht für **L**okale **A**ddierer-Zellen. Bei diesem Bild wird, besser als durch die einfache Angabe, daß 52% aller Slices belegt sind, ersichtlich, wie gedrängt es in und um die Addiererzellen herum zugeht. Dabei muß man berücksichtigen, daß jeder Ausgang einer 1-Bit Speicherzelle an zwei 1-Bit Addierer bzw. ihre Eingangsauswahl-Logik, an den Ausgangsbus und an die Erkennung für `all_zero` bzw. `all_one` gelangen muß. Jeweils fünf der zehn 64-Bit Addiererzellen befinden sich an einem gemeinsamen Ein- und Ausgangsbus.

5.4 Entwurfsentscheidungen

In diesem Abschnitt wollen wir noch einmal die erarbeiteten Erkenntnisse aus der Implementierung einer IEEE 754 single precision Summationsmatrix und ihres Tests in einem Xilinx Virtex XCV800-4 FPGA zusammen fassen. Dabei dürfen wir nicht vergessen, daß die Stärke der Summationsmatrix die Akkumulation eines Produktes pro Takt ist. Daher macht es wenig Sinn, die Addiererzellen zur Beschleunigung des Taktes mehrstufig auszulegen.

5.4.1 Transfer-Register oder Bus

Die wichtigste und grundsätzliche Entscheidung bei der Umsetzung einer Summationsmatrix in Hardware ist die Wahl zwischen der Transfer-Register-Lösung und der Buslösung. Dabei ist das Augenmerk auf die Vor- und Nachteile der beiden Lösungen und die Auswirkung derselben auf die angestrebte Technologie zu richten. Deswegen hier noch einmal die Vorteile und Nachteile der Transfer-Register-Lösung gegenüber der Carry-Select-Buslösung. Die Store-Carry-Buslösung wird bei dieser Betrachtung aus dem in Abschnitt 4.12.2 erörterten Grund nicht weiter beachtet.

Aus Tabelle 5.3 ist ersichtlich, daß die Wahl der Lösung sowohl abhängig von der gewünschten Technologie als auch dem gewünschten Einsatzgebiet ist. Die Vorteile der Buslösung liegen darin, daß sie das Ergebnis nach jedem Takt zur Verfügung hat und im nächsten Takt schon ausgelesen werden kann. Die Vorteile der Transfer-Register-Lösung liegen in der hauptsächlich lokalen Verbindungsart und der höheren Taktfrequenz.

5.4.2 Multiplizierer

Der Multiplizierer dürfte heutzutage keine Probleme mehr bereiten, da es in jeder Technologie schnelle 24×24 (bzw. 32×32), eventuell mehrstufige, Multiplizierer gibt.

Kriterium	Transfer-Register	CS-Buslösung
Addiereranzahl	mindestens 560 +	doppelt soviele
Carry-Logik	keine +	relativ aufwendig
Übertragsspeicher	2 Bit zwischen jeweils zwei Addierern	keine +
Nachtakte für Übertragsauflösung	mehrere (abhängig von Addiereranzahl)	keine +
Register	genauso viele wie Addiererbits	keine +
Verbindungen	lokal: von Zeile zu Zeile	global an alle Zeilen
bearbeitete Kontexte	jede Zelle unabhängig	alle Zellen bearbeiten den gleichen Kontext
Auslesen	mehrere Takte später	im gleichen Takt +
Taktfrequenz	Addiererabhängig +	immer etwas niedriger

+ markiert einen Vorteil der jeweiligen Lösung

Tabelle 5.3: Vergleich Transfer-Register gegenüber CSA-Bus

Die eigentliche Taktbeschränkung (falls wir mehrstufige Multiplizierer zulassen) liegt in den Addierern der Addiererzellen.

5.4.3 Addierer

Wie wir in früheren Abschnitten gesehen haben (siehe 2.3.1, 4.13.1) begrenzt die Addiererbreite die höchste Taktfrequenz des gesamten Entwurfes. Daher ist die Wahl der Addiererbreite immer direkt von der angestrebten Frequenz innerhalb der gewünschten Technologie abhängig. Dabei muß aber berücksichtigt werden, daß bei kürzeren Addierern mehr Addierer benötigt werden und daher die Nachteile der Transfer-Register-Lösung noch stärker ins Gewicht fallen. Je mehr Teil-Addierer vorhanden sind, umso mehr Übertragsspeicher und Nachtakte zur Auflösung werden benötigt. Bei der CS-Buslösung ist die einzige Größe, die Addiererbreiten abhängig ist, die Komplexität der Carry-Logik. Jedoch ist zu berücksichtigen, daß je kürzer die Addierer werden um so kürzer werden die Additionszeiten und umso gravierender sind die Verzögerungen durch die (auch noch größer gewordene) Carry-Logik bzw. die Ergebnisauswahl (siehe Tabelle 4.11, bzw. Abbildung 4.4).

5.4.4 Exponenten-Arithmetik-Einheit

Die Exponenten-Arithmetik-Einheit hat nur sehr kurze (< 16 Bit) Addierer. Bevor die Addierer ihre Arbeit aufnehmen können müssen verschiedene Vergleiche und Auswahlen stattfinden. Deswegen wird es nötig sein die Exponenten-Arithmetik-Einheit spätestens bei Addiererbreiten von ungefähr 20 Bit zwei- oder mehrstufig auszulegen. Das ist außer der zusätzlichen Pipeline-Stufe weiter kein Problem.

5.4.5 Ein-/Ausgabe-Einheit

Die Ein-/Ausgabe-Einheit beinhaltet keinerlei Technologie-kritische Elemente, dafür ist sie aber sehr stark von dem gewünschten Einsatzgebiet abhängig. Dabei ist immer zu berücksichtigen, daß ein Einsatzgebiet, das uns nicht mindestens ein Operandenpaar pro Takt liefern kann, keinen Sinn macht, da die Summationsmatrix ihre Stärke in der Akkumulation eines Produktes pro Takt besitzt. Für „langsamere“ Einsatzgebiete sind andere Lösungen wie die Speicherlösung sinnvoller. Wichtig ist, unsere Summationsmatrix braucht pro Takt ein Operandenpaar, das ist zum Beispiel auch dann gewährleistet, wenn wir in 2 Takten 4 Operanden bekommen und dann 2 Takte keine Operanden, falls es sich um eine IEEE 754 double precision Anwendung (siehe 6.3.1) oder um eine Intervallanwendung (siehe 6.3.2) handelt. Wie wir in Abschnitt 6.3.3 gesehen haben, ist das Verhältnis von zu übertragenden Operanden zu auszuführenden Operation bei einer Arithmetik oder einem Skalarprodukt für Staggered IEEE 754 single precision Zahlen noch kleiner als $\frac{1}{2}$.

5.4.6 Zusätzliche Einheiten

Natürlich werden alle nicht benötigten Einheiten wie etwa das Zeilenregister oder der 32×32 anstatt des 24×24 Multiplizierers weggelassen, wenn sie im gewünschten Einsatzgebiet nicht benötigt werden. Dafür können noch andere, hier in ihrem Zeit- und Ressourcenverhalten nicht weiter untersuchten Einheiten wie eine Rundung (siehe 6.2) oder eine Intervall-Einheit (siehe 6.3.2) dazu kommen. Auch ein mehrere Register umfassender Speicher und eine entsprechende Steuerungslogik für Staggered Zahlen sind denkbar. Der Schwerpunkt dieser Arbeit lag auf der Untersuchung der Summationsmatrix und ihrer Realisierung in Hardware und weniger in der Anwendung und Einbettung einer solchen Einheit und deswegen wollen wir diese auch nicht näher betrachten.

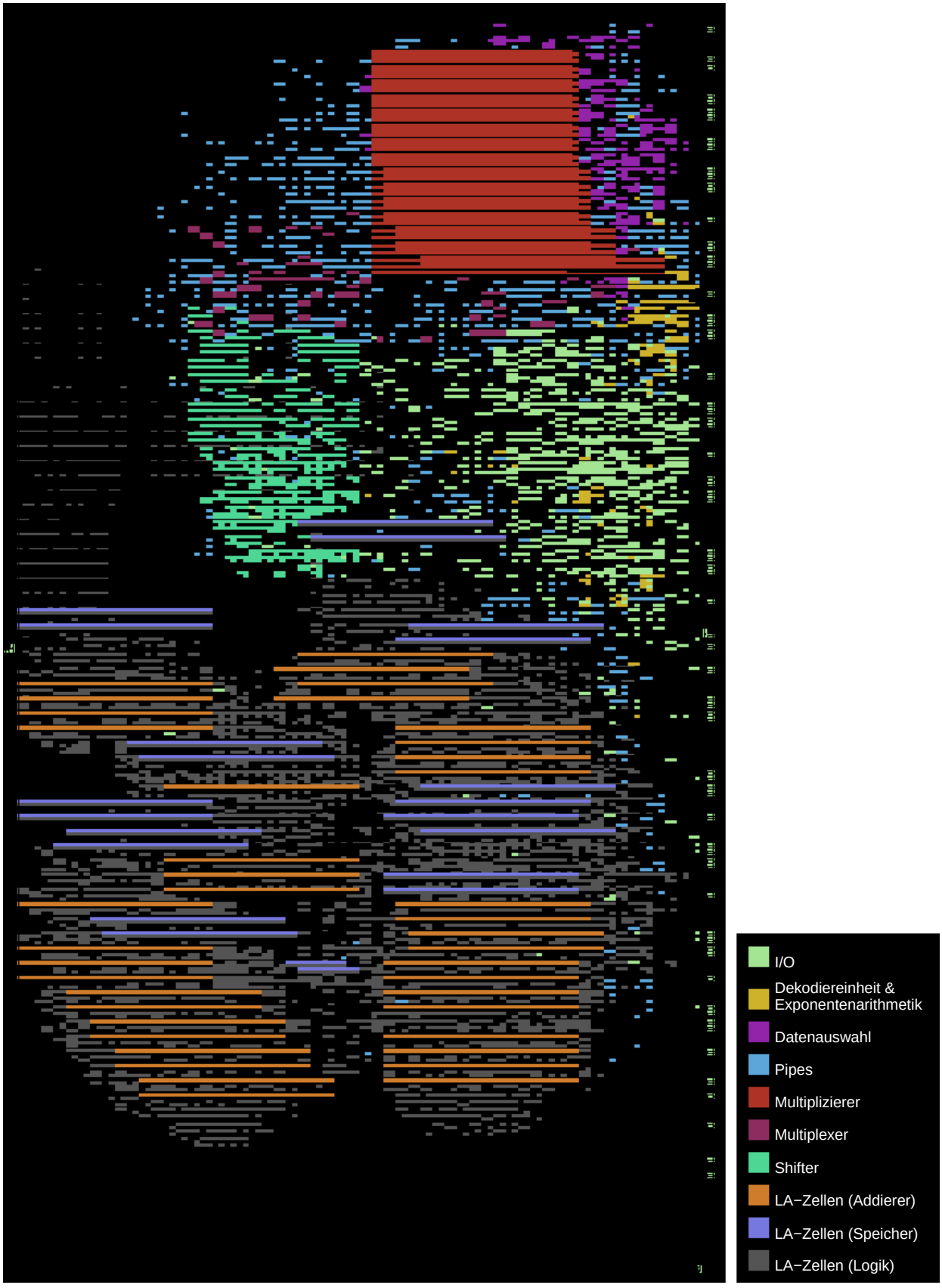


Abbildung 5.1: Gesamter Entwurf in einem XCV800

Kapitel

6

Bemerkungen

“ Calvin: *You can't just turn on creativity like a faucet.*
You have to be in the right mood.
Hobbes: *What mood is that?*
Calvin: *Last-minute panic.* ”

Bil Waterson: Calvin and Hobbes

In diesem Kapitel wollen wir mögliche Anwendungen und Erweiterungen des vorgestellten Entwurfes der Summationsmatrix betrachten. Dabei gehen wir davon aus, daß außer der Summationsmatrix noch ein Prozessor vorhanden ist, der die Daten zur Summationsmatrix schickt und die Ausgaben bei Bedarf abholt. Wir betrachten die Summationsmatrix also nur als das, was sie ist, eine Möglichkeit zum exakten Berechnen von IEEE 754 single precision Skalarprodukten. Im ersten Teil werden wir auf die Weiterverarbeitung der Ausgabe eingehen, die Rundung. Der Entwurf verarbeitet zwar als Eingabe **gültige** IEEE Zahlen, seine Ausgabe ist aber nicht in diesem Format.

Des weiteren werden wir dann mögliche Erweiterungen betrachten, die die Einheit befähigen, wie in früheren Kapiteln schon vorgestellt, IEEE 754 double precision Zahlen zu verarbeiten, bzw. für diese ein exaktes Skalarprodukt zur Verfügung zu stellen. Als weitere Anwendungen werden Intervall-Skalarprodukte von IEEE 754 single precision Zahlen und die Anwendung für ein exaktes Skalarprodukt von Staggered Zahlen, die auf dem IEEE 754 single precision Datentyp beruhen, vorgestellt.

Doch zu allererst wollen wir uns der Behandlung bzw. fehlenden Bearbeitung der IEEE Sonderfälle widmen.

6.1 IEEE Sonderfälle

Bis jetzt haben wir uns noch nirgends mit den Sonderfällen des IEEE 754 single precision Zahlenformates wie NaN (**N***ot* **a N***umber*) und ±Infinity beschäftigt. Das liegt zum einen daran, daß wir bis jetzt immer implizit davon ausgegangen sind, daß wir als Operanden „gültige“ (de)normalisierte Zahlen haben. Diese Annahme ist darin begründet, daß die Summationsmatrix ihre Daten von einem anderen Prozessor bezieht und nur ihre Spezialaufgabe des exakten Skalarproduktes durchführt.

Damit überlassen wir diesem externen Prozessor (im weiteren Verlauf dieses Kapitels auch Host genannt) die Aufgabe der Sonderfallbehandlung. Wenn wir uns die anfallenden Aufgaben genauer ansehen, so werden wir feststellen, daß diese Arbeit auch durch eine kleine parallel zur Summationsmatrix laufende Einheit erfüllt werden kann. Die Aufgabe dieser Einheit ist nur, für jeden Kontext mit zu protokollieren, ob der Kontext ein NaN oder ein Infinity ist. Dabei betrachten wir nur ein ±Infinity als Ergebnis von ±Infinity-Operanden ,nicht als Ergebnis von zu vielen Akkumulationen, die das Ergebnis außerhalb des darstellbaren Bereiches treiben. Diesen Fall betrachten wir im nächsten Abschnitt wenn es um die Weiterverarbeitung der Summationsmatrix-Ausgabe geht (siehe 6.2).

Bei der Summationsmatrix treten nur die Operationen Multiplikation und Addition auf, so daß wir nur die in Tabelle 6.1 vorgestellten Ausnahmen berücksichtigen müssen. In dieser Tabelle verstehen wir unter einer normalen Zahl eine IEEE 754 single precision Zahl die ungleich NaN oder ±Infinity ist, das gleiche gilt auch, wenn wir $X > 0$ oder $X < 0$ schreiben. Wenn nicht von einer „normalen Zahl“ gesprochen wird, so ist jede mögliche IEEE 754 single precision Zahl (auch NaN oder ±Infinity) gemeint.

Operation	Bemerkung	Ergebnis
$0 \times \pm\infty$	Null multipliziert mit Unendlich	NaN
$X \times \pm\infty$	$X > 0$ multipliziert mit Unendlich	$\pm\infty$
$X \times \pm\infty$	$X < 0$ multipliziert mit Unendlich	$\mp\infty$
$X \times$ NaN	Multiplikation mit NaN	NaN
$X + \pm\infty$	Addition von Unendlich zu einer normalen Zahl	$\pm\infty$
$X - \pm\infty$	Subtraktion von Unendlich von einer normalen Zahl	$\mp\infty$
$X \pm$ NaN	Addition/Subtraktion mit NaN	NaN
$\pm\infty + \mp\infty$	Addition von Unendlich mit verschiedenen Vorzeichen	NaN
$\pm\infty - \pm\infty$	Subtraktion von Unendlich mit gleichem Vorzeichen	NaN
$\pm\infty + \pm\infty$	Addition von Unendlich mit gleichem Vorzeichen	$\pm\infty$
$\pm\infty - \mp\infty$	Subtraktion von Unendlich mit verschiedenen Vorzeichen	$\pm\infty$

Tabelle 6.1: Mögliche Ausnahme-Operationen

Wir sehen, daß diese Einheit (nennen wir sie IEEE-Einheit) die Operanden nur auf Null, NaN und ±Infinity untersuchen muß und dann nach Tabelle 6.1 den ent-

sprechenden Wert NaN, $+\infty$ oder $-\infty$ für den aktuellen Kontext abspeichern kann. Dabei ist der aktuelle Inhalt der Summationsmatrix unwichtig. Beim Auslesen eines Kontextes muß die IEEE-Einheit dann nur noch überprüfen, ob sie für diesen Kontext einen Ausnahmewert gespeichert hat und diesen dann gegebenenfalls anstatt des Summationsmatrix-Inhaltes weitergeben.

Da nur vier verschiedene Zustände gespeichert werden müssen, nämlich NaN, $+\infty$, $-\infty$ oder GUELTIG sind pro Kontext gerade mal zwei Bit notwendig. Der Hardware-Aufwand für diese Einheit ist also sehr gering. Die einzige Möglichkeit noch ein Ausnahmeergebnis zu erhalten ist, wenn der Inhalt des Kontextes aufgrund seiner Größe nicht mehr als IEEE 754 single precision Zahl darstellbar ist. Dann wird das Ergebnis $\pm$Infinity. Dies festzustellen ist (unter anderem) Aufgabe des Hosts oder der im nächsten Abschnitt (siehe 6.2) vorgestellten Rundungseinheit.

6.2 Rundung

Bei der Ausgabe der Summationsmatrix werden (falls vorhanden) mindestens 65 relevante Mantissenbits ausgegeben (siehe 4.12), so daß auch eine Darstellung des Ergebnisses in IEEE 754 double precision Zahlen möglich ist. Da die Ausgabe der Summationsmatrix immer 128 Bit breit ist, auch wenn nur die unteren 47 Bit des IEEE 754 single precision Kontextes belegt sind, sind die Signale `all_zero` und `all_one`, die für jeweils die unteren bzw. oberen 64 Bit mitgeliefert werden, eine Hilfe für die Erkennung der führenden Nullen. Die Rundungseinheit muß also nur ein 64 Bit breites Datenwort auf die Anzahl der führenden Nullen (*engl: leading zeros*) untersuchen, um die Ausgabe zu normalisieren.

Damit die Rundungseinheit auch wirklich alle Rundungsmodi, die im IEEE 754 Standard beschrieben sind, ausführen kann, wird ihr von der Summationsmatrix über das `sticky`-Bit mitgeteilt, ob unterhalb der ausgegebenen 128 Bits noch zusätzlich mindestens ein Bit gesetzt ist. Das Vorzeichen des Kontextinhaltes wird als extra Bit auch an die Ausgabe geliefert. Sollte die Rundungseinheit keine Möglichkeit der Zweierkomplement-Bildung haben (negative Inhalte werden in der Summationsmatrix als Zweierkomplement dargestellt), so ist über den bedingten Negationsbefehl die Ausgabe in einer Vorzeichen-Betrag-Darstellung möglich.

Mit Hilfe all dieser Informationen ist es einer Rundungseinheit oder dem Host-Prozessor möglich, das Ergebnis der Summationsmatrix in eine IEEE 754 single precision oder IEEE 754 double precision Zahl umzuwandeln.

6.3 Anwendungen

In diesem Abschnitt wollen wir ein paar mögliche Verwendungen dieser IEEE 754 single precision Summationsmatrix vorstellen. Dabei benötigen alle Vorschläge außer der vorhanden Logik noch zusätzliche Logik. Die notwendigen Zusätze wollen wir hier skizzenhaft aufführen, um die möglichen Einsätze des vorliegenden Entwurfes aufzuzeigen.

6.3.1 Double Precision Skalarprodukte

Die Möglichkeit mit unserer IEEE 754 single precision Summationsmatrix auch exakte Skalarprodukte von IEEE 754 double precision Zahlen zu berechnen, wurden ausführlich in dem Abschnitt 2.6 und auch 4.10 angesprochen. Dabei haben wir gesehen, daß insbesondere für das dynamische Fensterkonzept eine Interaktion zwischen der Summationsmatrix und dem Host notwendig ist. Der Host muß nämlich die Zugehörigkeit der einzelnen IEEE 754 single precision-Kontexte zu den entsprechenden IEEE 754 double precision-Kontexten verwalten. Wie wir gesehen haben, sind für manche dieser Lösungen nicht nur mehrere Befehle zum Addieren der einzelnen Teile sondern auch Zusätze zum Beispiel in der Exponenten-Arithmetik-Einheit notwendig.

Für den einfachsten Fall des starren Fensterkonzeptes (siehe 2.6.1) genügt es, mit dem schon vorhandenen Zeilenregister die einzelnen Kontexte zusammen zu setzen. Über die beiden Bits `all_zero` der Ausgabe eines Kontextes ist es leicht möglich festzustellen, ob einer der beteiligten IEEE 754 single precision Kontexte nur aus Nullen besteht. Nach dem Zusammensetzen der Kontexte (und der Übertragsauflösungen) ist das Ergebnis die Ausgabe des höchsten von Null verschiedenen Kontextes, eventuell mit der Ausgabe des darunterliegenden Kontextes kombiniert. Die Ausgabe des darunterliegenden Kontextes ist nur dann notwendig, wenn der höchste Kontext weniger als die benötigten Mantissenbits beinhaltet. Sollte beim höchsten Kontext das `sticky`-Bit Null sein, so sind die darunterliegenden Kontexte auf etwaige Inhalte zu untersuchen. Auch dazu reicht wieder die Untersuchung der jeweiligen Informationsbits `all_zero` bzw. `all_one` aus.

Für eine solche Anwendung ist es vorteilhafter, wenn die Informationen `all_zero`, `all_one` und `sticky` der einzelnen Kontexte nicht hintereinander sondern nebeneinander abgespeichert werden. Beim vorgestellten Entwurf sind diese Informationen „hintereinander“ abgespeichert, so daß für das Auslesen der Information jedes Teil-Kontextes ein Takt notwendig ist. Wenn alle diese Informationen in einem einzigen Register „nebeneinander“ zugänglich wären, so könnten sie in einem Takt (mit einem speziellen Befehl) ausgelesen und anschließend ausgewertet werden.

Diese Anmerkungen zeigen, daß der vorliegende Entwurf zwar die grundlegenden Ideen wie mehrere IEEE 754 single precision Kontexte, Partialprodukt-Steuerung und Zeilenregister schon mitbringt, aber nicht auf eine IEEE 754 double precision Anwendung hin optimiert ist.

6.3.2 Intervall Skalarprodukt

Eine weitere Anwendung des vorliegenden Entwurfes ist das IEEE 754 single precision Intervall-Skalarprodukt. Wie wir schon in Abschnitt 1.5.2 gesehen haben, benötigen wir für Intervall-Skalarprodukte zwei Skalarprodukte (siehe auch [21]).

Das eine dieser Skalarprodukte ist die Darstellung der unteren das andere die Darstellung der oberen Grenze des Ergebnisintervalles. Die Multiplikation zweier Intervalle können wir ähnlich wie die schon vorhandenen double Partialprodukte in vier Teilmultiplikationen zerlegen. Wenn wir nicht erst das erste und dann das zweite Intervall zur Summationsmatrix übertragen, sondern im ersten Takt zum Beispiel die beiden unteren Grenzen und im zweiten Takt die beiden oberen Grenzen der beiden Intervalle $[a_1, a_2]$ und $[b_1, b_2]$ so benötigen wir nur zwei Übertragungstakte für

insgesamt vier Multiplikationstakte. Sobald die ersten Operanden an der Summationsmatrix anliegen, kann diese mit den Operationen beginnen. Ähnlich wie bei der double Zahlen Verarbeitung werden die übertragenen Zahlen zwischengespeichert, um alle vier Partialprodukte nacheinander auszuführen.

$$\begin{aligned}[a_1, a_2] \times [b_1, b_2] &= [min(a_1 \times b_1, a_1 \times b_2, a_2 \times b_1, a_2 \times b_2), \\ &\quad max(a_1 \times b_1, a_1 \times b_2, a_2 \times b_1, a_2 \times b_2)]\end{aligned}$$

Die vier notwendigen Teilprodukte sind also: $\begin{matrix} a_1 \times b_1 \\ a_1 \times b_2 \\ a_2 \times b_1 \\ a_2 \times b_2 \end{matrix}$

Anschließend ist noch die Minimum- und Maximumbildung über die Teilprodukte für die Bestimmung der unteren und oberen Grenze notwendig. Wenn wir jetzt davon ausgehen, daß die Operanden für die Teilprodukte nacheinander an den Multiplizierer gebracht werden und die Kontextaddressen für das Skalarprodukt der unteren und der oberen Grenze abgespeichert und damit abrufbar ist, so kann mit Hilfe der in Abbildung 6.1 vorgestellten Logik eine Intervall-Einheit aufgebaut werden. Dabei ist über die Logik auch sichergestellt, daß sich zwei aufeinanderfolgende Intervall-Skalarprodukte nicht gegenseitig beeinflussen.

Als erstes ist zu bemerken, daß `t_start` ein Bit ist, das nur beim ersten Partialprodukt auf Eins gesetzt ist, um etwaige aufeinanderfolgende Intervall-Skalarprodukte von einander unterscheidbar zu machen. Das Bit `t_int` wiederum ist bei allen vier Teilprodukten eines Intervall-Skalarproduktes gesetzt, damit Intervall-Skalarprodukte von anderen Skalarprodukten unterscheidbar sind. Bei der vorliegenden Lösung gehen wir davon aus, daß nur ein Intervall-Skalarprodukt möglich ist, da wir anderenfalls auch noch eine Kennung für die unterschiedlichen Intervall-Skalarprodukte mitführen müssten. Alle in der Abbildung vorhandenen Register werden mit der fallenden Taktflanke des Summationsmatrix-Taktes gesteuert (`nclk`).

Der in der Abbildung vorhandene Komparator wird eigentlich zweimal benötigt, aber der Übersichtlichkeit wegen haben wir ihn nur einmal gezeichnet. Am Eingang `B` des Komparators (*Vergleicher*) befindet sich ein Register, das ein Teilprodukt, sein Vorzeichen und seinen Exponenten abspeichern kann. Die Teilprodukte werden nämlich mit samt ihres Vorzeichens und ihres Exponenten an den Vergleicher geliefert, so daß er auch wirklich das Maximum bzw. Minimum bestimmen kann. Der Ausgang `M` des Maximum-Komparators ist genau dann Eins, wenn die Daten am Eingang `A` größer als die am Eingang `B` sind. Beim Minimum-Komparator ist der Ausgang genau dann Eins, wenn die Daten am Eingang `A` kleiner als die am Eingang `B` sind. Mit diesem Ausgang wird zum Einen über den Steuereingang `store_enable` das Register am Eingang `B` veranlaßt, die aktuellen Daten am Eingang `A` für die weiteren Vergleiche als momentanes Maximum zu speichern. Zum Anderen wird über die paar UND- und ODER-Gatter in den vier Registern immer dem aktuellen

Teilprodukt eine Eins und allen anderen eine Null an den Eingang gelegt, so daß in diesen vier Registern parallel zu den vier links befindlichen Teilprodukt-Registern immer genau bei dem Teilprodukt eine Eins steht, das bis jetzt das Maximum ist. Das heißt, außer dem Komparator müssen auch die vier Register und ihre UND- und ODER-Gatter für die Minimum-Ermittlung doppelt vorhanden sein.

Nachdem die vier Teilprodukte die beiden Komparatoren passiert haben, liegen uns am Ende der Pipelines die Partialprodukte (eines nach dem anderen) zusammen mit der Information „ist Maximum“ bzw. „ist Minimum“ an. Am Ausgang der `t_int`-Pipeline erkennen wir, daß es sich überhaupt um ein Intervall-Skalarprodukt handelt und somit die Maximums- bzw. Minimums-Information überhaupt relevant ist. Wenn wir diese Einheit in Abbildung 4.1 zwischen Multiplizierer und Selektier-Einheit platzieren (natürlich werden alle Pipelines um die entsprechende Tiefe verlängert) so haben wir die Möglichkeit in Abhängigkeit der vorhandenen Information den Summationsmatrix-Kern entsprechend zu füttern. Wenn es sich beim anliegenden Teilprodukt weder um das Minimum noch um das Maximum handelt, so muß die Selektier-Einheit eine `NULL` an den Shifter übergeben. Diese Möglichkeit ein `NULL`-Produkt zu übergeben ist schon in die Selektier-Einheit implementiert. Falls es sich beim anliegenden Teilprodukt um das Maximum handelt, so muß die entsprechende Kontextaddresse an dieser Stelle in die Kontextaddress-Pipeline gegeben werden und das Teilprodukt wird ansonsten wie alle anderen Produkte behandelt. Entsprechendes gilt für das Minimum. Bei allen anderen Produkten (nicht Intervall-Skalarprodukten) verhält sich der Entwurf mit Ausnahme der jetzt um vier Stufen längeren Pipelines wie sonst auch.

Das heißt, wenn wir alle Pipelines um vier Stufen verlängern, noch zwei 58 Bit[1] Vergleicher, zwei Kontextaddress-Register und etwas Zusatzlogik und die notwendigen vier Pipeline-Stufen spendieren, so haben wir eine Intervall-Skalarprodukt-Einheit. Genauer gesagt eine Einheit die exakte Skalarprodukte für IEEE 754 single precision Zahlen und für Intervalle die aus solchen Zahlen gebildet sind, berechnet.

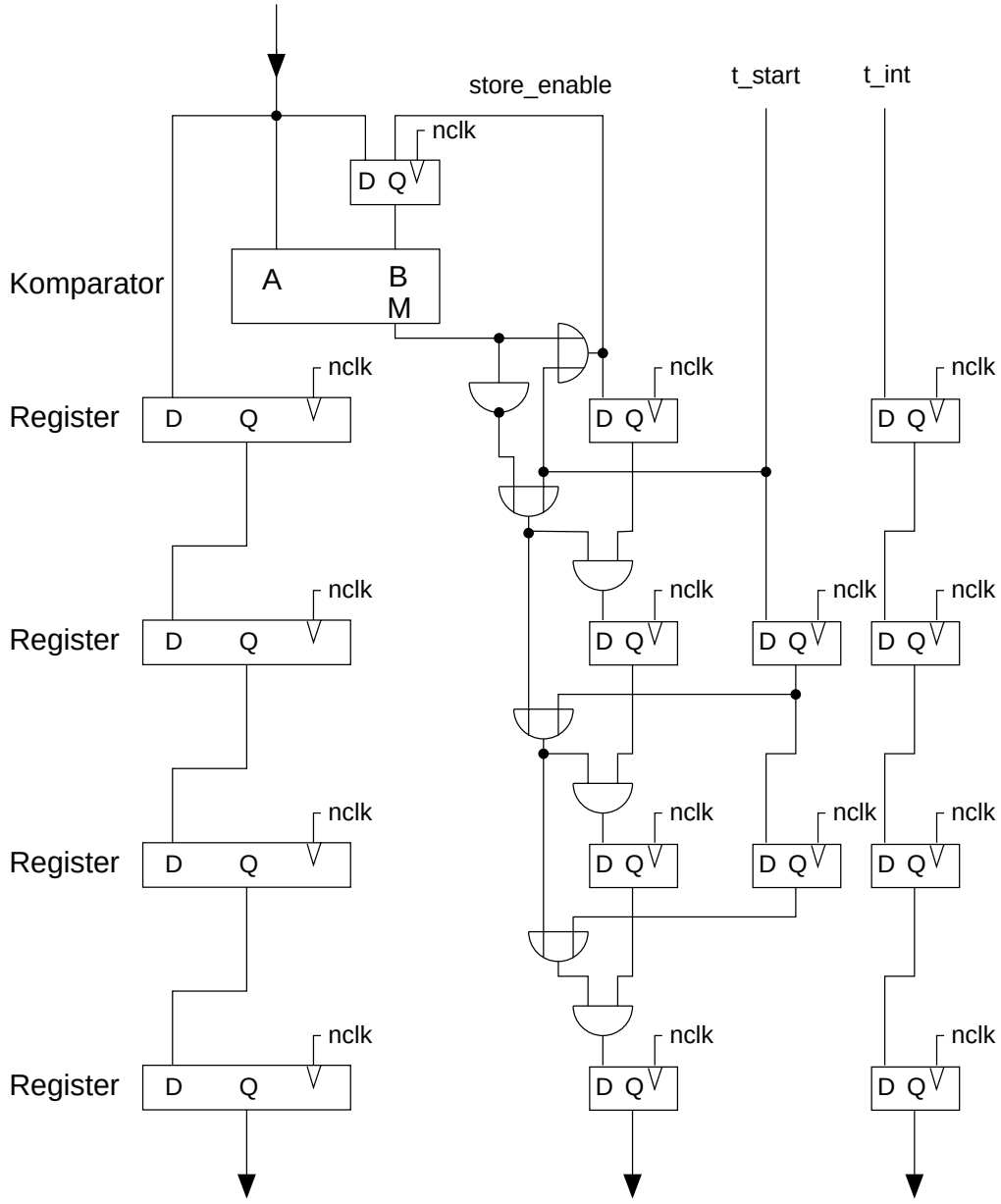
6.3.3 Staggered Arithmetik

Als letzte Anwendung wollen wir die Verwendung für eine Staggered-Arithmetik aufzeigen. Dabei gehen wir wieder davon aus, daß der Grunddatentyp IEEE 754 single precision Zahlen sind. Dann kann mit dieser Einheit nicht nur ein exaktes Skalarprodukt berechnet werden, sondern auch die Multiplikation, Addition und Subtraktion von Staggered Zahlen.

Staggered Auslese

Wie wir in Abschnitt 1.5.1 gesehen haben, ist das Skalarprodukt zweier Staggered-Vektoren nichts anderes als die Akkumulation von mehreren IEEE 754 single precision Produkten. Genauer gesagt ist das Skalarprodukt $\boldsymbol{x} \cdot \boldsymbol{y}$ der beiden Staggered-Vektoren $\boldsymbol{x} = \left(x_i\right)_{i=1}^{n}$ und $\boldsymbol{y} = \left(y_i\right)_{i=1}^{n}$ bei einer Staggered-Länge $l_1, l_2 = 5$ und der Vektorlänge $n = 10$ eine Akkumulation von $n \times l^2 = 250$ Einzelprodukten. Um diese 250 Teilprodukte zu bilden, benötigen wir nur 100 IEEE 754 single precision

[1] 1 Vorzeichen + 9 Produktexponenten + 48 Produkt Bits



D bezeichnet die Eingänge der Register
Q bezeichnet die Ausgänge der Register
nclk ist der (negierte) Takt (siehe 6.3.2)

Abbildung 6.1: Vierstufige Intervall-Einheit

Zahlen, nämlich die 10×5 Komponenten des Vektors $\boldsymbol{x}$ und die 50 Komponenten des Vektors $\boldsymbol{y}$. Auch hier haben wir ähnlich wie beim Intervall-Skalarprodukt mehr „Teil“-Produkte als Daten. Dies bedingt wiederum, daß eine Staggered-Zahlen-Einheit am besten mehrere Register hat um die Daten zwischenzuspeichern. Mit Ausnahme dieses Speichers (der nicht notwendig ist, falls der Host die Operanden für jedes Teilprodukt neu überträgt) und der Steuerung des Operandenabrufs aus dem Speicher ist der vorliegende Entwurf vollkommen für die genannten Staggered Operationen (natürlich nicht für die Division) ausreichend.

Erst wenn wir die Daten aus der Summationsmatrix auslesen wollen, ergibt sich ein Problem. Wie wir gesehen haben werden mindestens 65 Bit des vorhandenen Skalarproduktes nach aussen gegeben. Für das obige Beispiel benötigen wir aber mindestens (falls vorhanden) $5 \times 24 = 120$ Bits. Jetzt besteht natürlich die Möglichkeit die weiteren Bits (falls vorhanden) mit den entsprechenden Befehlen auszulesen und anschließend aus diesen Daten die Staggered Darstellung des exakten Skalarproduktes zu ermitteln. Diese Ermittlung wird durch die folgenden vier Schritte beschrieben:

1. Daten auslesen
2. das erste Bit ungleich Null feststellen
3. dieses und die folgenden 23 Bits zur Bildung der ersten IEEE 754 single precision Mantisse verwenden
4. den Exponenten dieser Mantisse bestimmen
5. am Ende dieser Mantisse wieder mit Schritt 1 beginnen

Dabei sind die Schritte 1-4 bei jedem Runden (Rundung zur Null) einer Summationsmatrix-Ausgabe notwendig. Der einzige Aufwand besteht in dem in Schritt 5 beschrieben Vorgehen.

Dafür können wir aber die schon vorhandene Hardware nutzen. Zu beachten ist, daß das beschriebene Verfahren den Inhalt des Summationsmatrix-Kontextes zerstört (Schritt 6). Falls dies nicht erwünscht ist, muß erst eine Kopie des Kontextes erstellt werden (siehe 2.8.2). Um den Kontext (oder seine Kopie) als staggered Zahl auszulesen, gehen wir wie folgt vor:

1. `msw_lacell` auslesen
2. das erste Bit ungleich Null feststellen (falls eine Rundungs-Einheit vorhanden ist, ist diese Hardware schon vorhanden)
3. von dieser Stelle aus 24 Bit nach rechts gehen
4. alle weiteren Bits auf Null setzen
5. die so modifizierten 128 Bit (führende Nullbits, 24 Nutzbits, in Schritt 4 erzeugte Nullen) in das Zeilenregister laden
6. das Zeilenregister von den zwei durch `msw_lacell` bestimmten Addiererzellen subtrahieren

7. gleichzeitig den Exponenten der in Schritt 3 ermittelten 24 Bit Mantisse bestimmen

8. wieder mit Schritt eins beginnen, bis wir $l-1$ Iterationen durchgeführt haben (l = Staggered-Länge des gewünschten Ergebnisses)

9. die letzte Komponente (die l-te) der Staggered Darstellung des Ergebnisses ermitteln wir durch das „normale“ Auslesen der Summationsmatrix mit anschließender Rundung.

Wie wir sehen, sind die Schritte 1-3 bei beiden Beschreibungen die gleichen. Der Schritt 5 des ersten Verfahrens (*am Ende dieser Mantisse wieder mit Schritt 1 beginnen*) wird in der Beschreibung des Summationsmatrix-Verfahrens in die Schritte 4, 5 und 6 aufgegliedert. Bei beiden Verfahren muß natürlich vor jeder weiteren Iteration geprüft werden, ob überhaupt noch Daten für weitere Mantissen vorhanden sind.

Wie wir sehen, benötigen wir, falls schon eine Rundungs-Einheit vorhanden ist, für die Staggered Ausgabe nur noch die Möglichkeit die Bits hinter den Mantissenbits auf Null zu setzen, was auch bei der Übernahme in das Zeilenregister geschehen kann.

Anhang

A

Befehle

In diesem Abschnitt werden die 32 Bit des Befehlswortes, das in das Register *REG2* geladen wird, vorgestellt und ihre Funktion erklärt. Dabei sind nicht alle Bits für den normalen Betrieb notwendig, manche Bits werden nur zum Testen und zur Fehlersuche benötigt, deswegen sind sie als sogenannte Debug-Bits gekennzeichnet.

Alle Bits werden innerhalb des Entwurfes nur über ihre Konstantennamen angesprochen, so daß die eigentliche Position im Befehlswort unwichtig ist. Manche Bits bilden zusammen eine Addresse bzw. einen Zähler und müssen deswegen in einer definierten Reihenfolge nebeneinander stehen.

In Tabelle A.2 sind alle Bits mit ihrem in den VHDL-Dateien verwendeten Namen und ihrer von `call_suma` (siehe B) verwendeten Position aufgeführt. In der Spalte `INTERFACE` ist aufgeführt, ob die Eingabe-Einheit dieses Bit bearbeitet. Alle Bits werden von der Dekodier-Einheit bearbeitet und in wenigen Fällen auch noch zusätzlich von der Eingabe-Einheit. Ein X in den Spalten `EXPO` bis `SUMA` zeigt an, daß die entsprechende Einheit durch dieses Bit beeinflußt wird (über von `DECODE` erzeugte Steuersignale). Die Bedeutung dieser Abkürzungen und der Abschnitt in denen die Einheit vorgestellt wurde, sind aus Tabelle A.1 zu entnehmen. Alle Bits mit einem X in der Spalte „sign“ beeinflussen (neben den eigentlichen Daten) auch noch das Vorzeichen und damit die Operation (Addition oder Subtraktion) des Summationsmatrix-Kerns.

`EXPO`	Exponenten-Arithmetik-Einheit	siehe 4.6
`DATA`	Datenauswahl-Einheit	siehe 4.5
`SELECT`	Selektier-Einheit	siehe 4.8
`COLREG`	Zeilenregister-Einheit	siehe 4.10
`SUMA`	Summationsmatrix-Kern	siehe 4.12

Tabelle A.1: Abkürzungen in Tabelle A.2

Name	Bitnr.	Position	INTERFACE	*Debug*	sign	EXPO	DATA	SELECT	COLREG	SUMA	Abschnitt
cmd_ok_flag4	31	fest	X								A.1
cmd_add_tpt	30					X				X	A.2
cmd_in_out	29					X			X	X	A.3
cmd_double_res	28					X			X	X	A.4
cmd_mode_tpt	27					X	X	X			A.5
cmd_mode_add	26					X	X	X			A.5
cmd_mode_dbl	25					X	X	X			A.5
cmd_ok_flag3	24	fest	X								A.1
cmd_ctxt_bit4	23									X	A.6
cmd_ctxt_bit3	22									X	A.6
cmd_ctxt_bit2	21									X	A.6
cmd_ctxt_bit1	20									X	A.6
probe_flag1	19			X							A.7
probe_flag2	18			X							A.7
cmd_substract	17				X						A.8
cmd_cb_res	16					X				X	A.9
reserve4	15										A.10
reserve3	14										A.10
reserve2	13										A.10
reserve1	12										A.10
cmd_cell_bit04	11	links von cmd_cell_bit03				X					A.11
cmd_cell_bit03	10	links von cmd_cell_bit02				X					A.11
cmd_cell_bit02	9	links von cmd_cell_bit01				X					A.11
cmd_cell_bit01	8					X					A.11
cmd_ok_flag2	7	fest	X								A.1
cmd_def_prod	6			X				X			A.12
cmd_dbl_cnt2	5	links von cmd_dbl_cnt1	X			X	X				A.13
cmd_dbl_cnt1	4		X			X	X				A.13
cmd_cond_neg	3									X	A.14
cmd_init_ctxt	2					X		X		X	A.15
cmd_neg_ctxt	1				X					X	A.14
cmd_ok_flag1	0	fest	X								A.1

Tabelle A.2: Befehlbits

A.1 Die OK Flag Bits: cmd_ok_flagX

Diese vier Bit werden dazu verwendet, um einen gültigen Befehl im *REG2* der Eingabe-Einheit zu erkennen und die Datenregister (*REG0, REG1, REG10, REG11*) bis zur Übergabe der Daten und des Befehls zu sperren (siehe 4.2). Da über den lokalen Bus das 32 Bit Befehlswort in 8 Bit Portionen übertragen wird, soll die Lage der OK Flag Bits sicherstellen, daß das gesamte Befehlswort im *REG2* ist, bevor mit der Weitergabe an die Summationsmatrix begonnen wird. Aus diesem Grunde sollten mindestens die beiden äußern Bits (`cmd_ok_flag1, cmd_ok_flag4`) bestehen bleiben. Die OK Flag Bits sind die einzigen Bits im Befehlswort die aufgrund ihrer Funktion eine feste Position benötigen.

A.2 Das Addiere-bit: cmd_add_tpt

Mit diesem Bit unterscheidet die Exponenten-Arithmetik-Einheit und die Summationsmatrix zwischen dem Akkumulationsmodus ($add = 0$) und dem Lademodus ($tpt = 1$ für transparent). Die Exponenten-Arithmetik-Einheit benötigt dieses Signal um den richtigen Exponenten zu ermitteln.

A.3 Das Eingabe-bit: cmd_in_out

Das Eingabe-bit `cmd_in_out` bestimmt, ob Daten in die Summationsmatrix eingelesen oder ausgelesen werden. Dabei bedeutet eine Null den Einlese- (*in*) und eine Eins den Auslese-Modus (*out*).

A.4 Das Zeilenregister-bit: cmd_double_res

Durch das Zeilenregister-bit wird bestimmt, ob die Daten aus der Summationsmatrix in das Zeilenregister eingelesen werden (`cmd_in_out` = 1, `cmd_double_res` = 1), oder ob die Daten, die sich im Zeilenregister befinden, in die Summationsmatrix akkumuliert werden (`cmd_in_out` = 0, `cmd_double_res` = 1).

A.5 Die Mode-bits: cmd_mode_xxx

Die Mode-bits haben nur für die Einheiten Exponenten-Arithmetik, Datenauswahl und Selektieren eine Bedeutung. Durch das `cmd_mode_add` Bit wird diesen Einheiten mitgeteilt, wenn es sich um eine Akkumulation (`cmd_mode_add` = 1) handelt. Durch `cmd_mode_dbl` = 1 wird signalisiert, daß es sich bei den anliegenden Daten um IEEE 754 double precision Zahlen beziehungsweise Teile davon handelt. Das `cmd_mode_tpt` Bit dient dazu die Eingabedaten unverändert (mit Ausnahme eines von der Exponenten-Arithmetik-Einheit gesteuerten, eventuellen 64 Bit Shiftes) an die Summationsmatrix zu übergeben.

A.6 Die Kontext-Address-bits: `cmd_ctxt_bitX`

Diese Address-bits sind ausschließlich für den Kontextspeicher zuständig und werden daher unverändert an die Summationsmatrix weitergeleitet. Die Dekodier-Einheit speichert diese Addresse bis zum nächsten Befehl, so daß die Summationsmatrix immer auf dem zuletzt bearbeiteten Kontext agiert. Dies hat den Vorteil, daß die Summationsmatrix im „Leerlauf"(kein neuer Befehl, daher Akkumulation des Default-Produktes) bei SC-Addiererzellen schon mit der Übertragsauflösung beschäftigt ist. Aus dem gleichen Grund wird von der Dekodier-Einheit auch das Vorzeichen gespeichert, so daß nach einer Addition eine Carry- und nach einer Subtraktion eine Borrow-Auflösung stattfindet. Natürlich sind diese Funktionen nur für SCA-Lösungen relevant.

A.7 Die Probe-bits: `probe_flagX`

Diese Bits werden nur für die Fehlersuche oder für Testläufe des Entwurfes benötigt. Mit Hilfe dieser Bits ist es möglich die einzelnen Befehle im Entwurf wieder zufinden, so daß an unterschiedlichen Stufen der Pipeline bestimmte Befehle erkannt werden.

A.8 Das Subtraktions-bit: `cmd_substract`

Das Subtraktions-bit kehrt das aktuelle Vorzeichen der Daten um. Normalerweise wird das Vorzeichen aus den Vorzeichen der Daten bestimmt. Durch dieses Bit ist nicht nur eine Addition (positiver oder negativer Produkte) sondern auch eine Subtraktion der Daten möglich.

A.9 Das Übertrags-Auflösungs-bit: `cmd_cb_res`

Bei gesetztem `cmd_cb_res` Bit (= 1) werden im Zusammenspiel mit dem in Abschnitt A.8 vorgestellten `cmd_substract` Bit ein Takt lang die Carry (`cmd_substract` = 0) beziehungsweise die Borrow (`cmd_substract` = 1) des ausgewählten Kontextes aufgelöst. Dies geschieht dadurch, daß ein Takt lang zu diesem Kontext eine Null addiert bzw. subtrahiert wird. Es besteht im vorliegenden Entwurf **keine** Möglichkeit festzustellen, ob schon alle Überträge aufgelöst sind.

A.10 Die Reserve-bits: `reserveX`

Diese Bits stehen zusammen mit den Probe-bits und dem Default-Produkt-bit für Befehlserweiterungen zur Verfügung, da sie im normalen Betrieb keine Funktion haben.

A.11 Die Addiererzellen-bits: `cmd_cell_bit0X`

Diese vier Bits bestimmen bei den entsprechenden Befehlen (siehe Tabelle A.4) welche zwei Addiererzellen angesprochen werden. Dabei gibt eine Zahl n (gebildet durch die vier Bit) von 2 bis 10 an, daß die n-te und die $(n-1)$-te Addiererzelle

gemeint ist. Bei Eingabe einer eins (*binär* 0001) werden genauso wie bei einer zwei (*binär* 0010) die zweite und erste Addiererzelle angesprochen.

A.12 Das Default-Produkt-bit: cmd_def_prod

Mit diesem Bit ist es möglich anstatt des Null-Produktes im „Leerlauf“ ein anderes implementiertes Produkt auf den aktuellen Kontext zu addieren. Diese Bit ist nur für Testzwecke notwendig und steht deswegen für etwaige Befehlserweiterungen zur Verfügung.

A.13 Die Double-Zähler-bits: cmd_dbl_cntX

In diesen zwei Bits kodiert die Eingabe-Einheit die Nummer des aktuellen Teilproduktes eines IEEE 754 double precision Produktes (siehe Tabelle 4.3).

A.14 Die Negiere-bits: cmd_cond_neg, cmd_neg_ctxt

Das Negationsbit (cmd_neg_ctxt) wird zur Negation (= Multiplikation mit -1) benötigt. Dadurch ist es mit Hilfe des Zeilenregisters nicht nur möglich zwei Kontexte zu addieren, sondern auch zu subtrahieren. Da die Summationsmatrix negative Zahlen im Zweierkomplement abspeichert (siehe 4.11) müssen negative Zahlen vor einer Weiterverarbeitung (Auslesen, Runden) eventuell erst in eine Vorzeichen-Betrags-Darstellung gewandelt werden. Dies geschieht mit der bedingten Negation (cmd_cond_neg = 1 und cmd_neg_ctxt = 1). Dabei wird im Gegensatz zur normalen Negation nicht das Vorzeichen mit negiert. Die bedingte Negation hat nur bei negativen Kontexten eine Wirkung.

Die Negation wird durch eine Subtraktion des gesamten Kontextes von *NULL* bewerkstelligt. Deswegen müssen nach einer Negation eventuelle Überträge erst aufgelöst werden.

A.15 Das Initialisiere-Kontext-bit: cmd_init_ctxt

Ist das Bit cmd_init_ctxt gesetzt (= 1) so wird der eigentliche Befehl ignoriert und der gesamte Kontext initialisiert. Dabei wird der Inhalt des Kontextspeichers, das Vorzeichen, das Überlaufbit und alle Übertragsspeicher auf ihre Grundwerte (= 1 für Borrow, = 0 sonst) gesetzt.

A.16 Die wichtigsten Befehle

In diesem Abschnitt stellen wir die wichtigsten Befehle (die einzig gültigen) in ihrer hexadezimalen Schreibweise vor. Dabei verwenden wir außer den hexadezimalen Ziffern 0-F noch die beiden Ziffern X und Y zur Darstellung des Befehls. Diese beiden zusätzlichen Ziffern kommen jeweils nur an einer bestimmten Stelle im Befehlswort vor. Dabei steht X für die zu bearbeitende Kontextaddresse (Kontextnummer auf

Hex	Binär	dezimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Tabelle A.3: Hexadezimal Ziffern

die der Befehl wirkt). `X` steht also für eine der sechzehn möglichen Kontextaddressen Null bis Fünfzehn. Das `Y` steht nur in Befehlen die bestimmte Addiererzellen auswählen. Die Hexadezimal-Zahlen 1 bis A (binär `0001` bis `1010`) wählen dabei die n-te und $n-1$-te Addiererzelle aus, also bedeutet eine 5 (binär `0101`) an dieser Stelle, daß die fünfte und die vierte Addiererzelle angesprochen wird. Die einzige Ausnahme davon ist die 1. Da es keine nullte Addierzelle gibt, hat die 1 die gleiche Funktion wie die 2. Alle anderen Bitkombinationen, also 0 und B-F haben keine Auswirkung.

Steht an einer dieser Stellen anstatt `X` oder `Y` eine `0`, so bedeutet das, daß die entsprechende Angabe für diesen Befehl irrelevant ist und damit auch ignoriert wird. Zum Beispiel ist bei einer Produkt-Akkumulation die Angabe von Addiererzellen unnötig, da die entsprechenden Addiererzellen bei diesem Befehl von der Exponenten-Arithmetik-Einheit bestimmt werden.

Befehl	Beschreibung	Kommentar
8 **9**X**C**0081	Addiere single Produkt	REG0, REG1
8 **9**X**E**0081	Subtrahiere single Produkt	REG0, REG1
8 **B**X**C**0081	Addiere double Produkt	REG0, REG1, REG10, REG11
8 **B**X**E**0081	Subtrahiere double Produkt	REG0, REG1, REG10, REG11
8 **D**X**C**0081	Addiere single Zahl	REG0
8 **D**X**E**0081	Subtrahiere single Zahl	REG0
8 **F**X**C**0081	Addiere double Zahl	REG0, REG1
8 **F**X**E**0081	Subtrahiere double Zahl	REG0, REG1
8 1X**D**0081	Carry auflösen	addiert Null auf alle Addiererzellen
8 9X**F**0081	Borrow auflösen	Subtrahiert Null von allen Addiererzellen
8 1XC0085	Initialisiere Kontext	Kontext wird zu Null
8 1XC0083	Negiere Kontext	
8 1XC008**B**	bedingtes Negieren	negiert nur wenn Kontext negativ ist
A 1XC0081	Lese Kontext	Es werden die obersten 2 Addierzellen ausgelesen
87XC0Y81	Schreibe transparent	in die beiden Addierzellen Y, Y−1
E1XC0Y81	Lese transparent	die beiden Addierzellen Y, Y−1
B1XC0Y81	Lade in Zeilenregister	die beiden Addierzellen Y, Y−1
91XC0Y81	Addiere Zeilenregister	in die beiden Addierzellen Y, Y−1
8 10C0081	Wechsle Default-Produkt	anstatt alles Null zu addieren im „Leerlauf“

Tabelle A.4: Befehle

Anhang

B

call_suma Programm

Dieser Anhang stellt das C-Programm vor, mit dessen Hilfe eine Kommunikation mit unserem Entwurf auf der Spyderkarte überhaupt erst möglich ist. Es ist eine Abwandlung des Multiplikationsprogramm das zum Umfang der Spyderkarte gehörte. Auch das in Anhang C vorgestellte Web Interface verwendet zur Kommunikation mit der Spyderkarte ausschließlich das Programm `call_suma`. Ein Aufruf dieses Programmes führt natürlich nur dann zu sinnvollen Ergebnissen, falls ein entsprechender Entwurf in das Xilinx XCV800-4 FPGA auf der Spyderkarte geladen wurde. Das `call_suma` Programm kann auf unterschiedliche Art aufgerufen werden. Als Parameter sind nur Zeichenketten in hexadezimaler Form sinnvoll.

B.1 Kein Parameter

Wird das Programm ohne Parameter aufgerufen, so gibt es alle vorhandenen Informationen über den im Xilinx FPGA geladenen Entwurf aus. Als erstes wird der Inhalt des Registers 15 (siehe 4.2) ausgewertet und teilweise in Klartext übersetzt. In diesem Register sind das Datum des Entwurfes und einige Eigenschaften kodiert. Bei den Eigenschaften handelt es sich um die verwendete Addiererart (CS- oder SC-Addiererzellen), Bus- oder Transferregister-Lösung, ob die Contextaddresse abgespeichert wird und ob ein Zeilenregister vorhanden ist. Danach wird der Inhalt der Register 0, 1, 2, 10 und 11 ausgegeben (siehe 4.2). Im Anschluß daran folgen in 32 Bit-Portionen der Shifter-Ausgang (= Summationsmatrix-Kern-Eingang) und die Ausgabe des Summationsmatrix-Kerns. Alle Registerinhalte (`REG0 - REG14`) werden in hexadezimaler Schreibweise (siehe A.3) dargestellt.

B.2 Ein Parameter

Wird das Programm `call_suma` mit einem Parameter aufgerufen (in dezimaler Schreibweise), so wird der Inhalt des entsprechenden Registers ausgegeben (in hexadezimaler Schreibweise). Da der Entwurf nur 16 Register besitzt (`REG0 - REG15`)

ergeben als Parameter nur Werte zwischen Null und Fünfzehn einen Sinn. Sollte der Entwurf erweitert werden, oder die Registeranzahl verringert werden, so verändert sich die Menge der sinnvollen Parameter entsprechend. Es findet keine Überprüfung des Parameters statt, so daß die Reaktion auf andere als die vom jeweiligen Entwurf abgedeckten Werte vom Verhalten des PCI-Bausteines auf der Karte abhängig sind.

B.3 Zwei Parameter

Beim Aufruf mit zwei Parametern wird der erste Parameter als Registeraddresse interpretiert. Der zweite Parameter wird als 32 Bit Word in hexadezimaler Schreibweise erwartet, es wird versucht dieses Wort in das durch den ersten Parameter angegebene Register zu schreiben. Aus diesem Grund sind als Registeraddresse nur die Werte 0, 1, 2, 10 und 11 sinnvoll, da die anderen Register nicht beschreibbar sind.

B.4 Quelltext

```
/******************************************************************************
 * Copyright (c) 1997 - 1998 PLX Technology, Inc.
 *
 * PLX Technology Inc. licenses this software under specific terms and
 * conditions.  Use of any of the software or derviatives thereof in any
 * product without a PLX Technology chip is strictly prohibited.
 *
 * PLX Technology, Inc. provides this software AS IS, WITHOUT ANY WARRANTY,
 * EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, ANY WARRANTY OF
 * MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.  PLX makes no guarantee
 * or representations regarding the use of, or the results of the use of,
 * the software and documentation in terms of correctness, accuracy,
 * reliability, currentness, or otherwise; and you rely on the software,
 * documentation and results solely at your own risk.
 *
 * IN NO EVENT SHALL PLX BE LIABLE FOR ANY LOSS OF USE, LOSS OF BUSINESS,
 * LOSS OF PROFITS, INDIRECT, INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES
 * OF ANY KIND.  IN NO EVENT SHALL PLX'S TOTAL LIABILITY EXCEED THE SUM
 * PAID TO PLX FOR THE PRODUCT LICENSED HEREUNDER.
 *
 ******************************************************************************/

/**                              adapted from Norbert Bierlox; 1998 - 2001  **/

#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include "e:\Plx\pcitypes.h"
#include "e:\Plx\plxtypes.h"
#include "e:\Plx\PlxError.h"
#include "e:\Plx\plx.h"
#include "e:\Plx\PciApi.h"

/*****************************************************/
//      PCI Data
```

```
// Used to work with a PCI device
/*******************************************************/
RETURN_CODE rc;
DEVICE_LOCATION device;
HANDLE myPlxDevice;
VIRTUAL_ADDRESSES virtualAddresses;

int openVirtex() {
/*************************************************************/
//                       PCI9080 Device Open
/*************************************************************/
// opens PCI access to the Spyder-Virtex Board
// You must close the PCI access by closeVirtex before exit !

  device.BusNumber  = MINUS_ONE_LONG;
  device.SlotNumber = MINUS_ONE_LONG;
  device.DeviceId   = MINUS_ONE_LONG;
  device.VendorId   = MINUS_ONE_LONG;
  strcpy (device.SerialNumber, "pci9080-0");
  rc = PlxPciDeviceOpen (&device, &myPlxDevice);
  if (rc != ApiSuccess) {
    printf ("ERROR: in opening device. Returned code is %d\n",rc);
    return 0;
  }
  return 1;
}

int closeVirtex() {
/*************************************************************/
//                        PCI9080 Device Close
/*************************************************************/
// closes PCI access to the Spyder-Virtex Board

  rc = PlxPciDeviceClose(myPlxDevice);
  if (rc != ApiSuccess) {
    printf ("ERROR: in closing device. Returned code is %d\n",rc);
    return 0;
  } else
    return 1;
}

/*************************************************************/
/*                         Data and Functions
*/
/*************************************************************/

  volatile PU32 pBase = 0x0;

int findBoard() {
/*************************************************************/
//                       Get PCI Virtual Base Addresses
/*************************************************************/
// determines the virtual addresses of the memory
// sets the global vars: virtualAddresses and pBase
// can be used only when  PCI9080 Device is open
```

```
  rc = PlxPciBaseAddressesGet(myPlxDevice, &virtualAddresses);
  if ( rc != ApiSuccess ) {
    printf ("ERROR: Failed to get virtual address with error code %08X",rc);
    return 0;
  } else {
    pBase = (PU32) virtualAddresses.Va3;
    return 1;
  }
}

int writeMemCell (U32 addrOff, U32 value32) {
  if ( openVirtex() && findBoard() ) {
    pBase[addrOff] = value32;
    if ( !closeVirtex() ) {
      return 0;
    }
    return 1;
  }
  return 0;
}

int readMemCell(U32 addrOff, U32* value32){
  if ( openVirtex() && findBoard() ) {
    *value32 = pBase[addrOff];
    if ( !closeVirtex() ) {
      return 0;
    }
    return 1;
  }
  return 0;
}

void main ( int argc, char* argv[] ) {
  unsigned char regAddr;
  U32 value;
  U32 value2;
  int i;
  if ( argc==1 ) {
    readMemCell((unsigned char)15, &value);
    printf("\n%02d.%02d.2001 Version: %01d  %05x\n",
    (value&0x01f00000)>>20,(value&0xf0000000)>>28,value&0x00000003,value&0x000fffff);
    if ( 0<(value&0x00000200) ) {
      printf("SCB-Architektur (Store Carry Borrow)\n");
    } else {
      printf("CSA-Architektur (Carry Select Adder)\n");
    }
    if ( 0<(value&0x08000000) ) {
      printf("STR-Architektur (Transfer Register)\n");
    } else { printf("\n"); }
    if ( 0<(value&0x00080000) ) {
      printf("TEST VERSION   !!!!!!!!!!!\n\n");
    } else { printf("\n"); }
    if ( 0<(value&0x00000010) ) {
      printf("double resolve Register    : vorhanden\n");
    } else {
      printf("double resolve Register    : keins\n");
```

```
    }
    if ( 0<(value&0x04000000) ) {
      printf("Context-Adresse gespeichert: JA\n");
    } else {
      printf("Context-Adresse gespeichert: NEIN\n");
    }
    printf("\n");
    for ( i=0; i<2; ++i ) {
      readMemCell((unsigned char)i, &value);
      readMemCell((unsigned char)i+10, &value2);
      printf("Reg %2d : %08x   %08x  <-- Reg %2d\n", i, value, value2, i+10);
    }
    readMemCell((unsigned char)2, &value);
    printf("Reg %2d : %08x           <-- Command\n", 2, value);
    readMemCell((unsigned char)3, &value);
    printf("Reg %2d : %08x           <-- \n", 3, value);
    readMemCell((unsigned char)4, &value);
    printf("Reg %2d : %08x           <-- Shifter out (127 downto  96)\n", 4, value);
    readMemCell((unsigned char)5, &value);
    printf("Reg %2d : %08x           <-- Shifter out ( 95 downto  64)\n", 5, value);
    readMemCell((unsigned char)6, &value);
    printf("Reg %2d : %08x           <-- Shifter out ( 63 downto  32)\n", 6, value);
    readMemCell((unsigned char)7, &value);
    printf("Reg %2d : %08x           <-- Shifter out ( 31 downto   0)\n", 7, value);
    printf("\n");
    readMemCell((unsigned char)8, &value);
    printf("Reg %2d : %08x           <-- SuMa out \n", 8, value);
    readMemCell((unsigned char)9, &value);
    printf("Reg %2d : %08x           <-- SuMa out (127 downto  96)\n", 9, value);
    readMemCell((unsigned char)12, &value);
    printf("Reg %2d : %08x           <-- SuMa out ( 95 downto  64)\n", 12, value);
    readMemCell((unsigned char)13, &value);
    printf("Reg %2d : %08x           <-- SuMa out ( 63 downto  32)\n", 13, value);
    readMemCell((unsigned char)14, &value);
    printf("Reg %2d : %08x           <-- SuMa out ( 31 downto   0)\n", 14, value);
  } else {
    if ( argc>1 ) {
      regAddr = atoi( argv[1] );
      if ( argc>2 ) {                       // write mode
        sscanf(argv[2],"%x",&value);
        if ( writeMemCell(regAddr, value) ) {
          exit(0);
        } else exit(1);
      } else {                              // read mode
        if ( readMemCell(regAddr, &value) ) {
          printf("%08x\n",value);
          exit(0);
        } else exit(1);
      }
    }
  }
}
```

Anhang

C

Web Interface

Beim `call_suma` Programm ist die Ein- und Ausgabe nur an dem Computer möglich ist, der die Spyderkarte an seinem PCI-Bus angeschlossen hat. Mit dem hier vorgestellten Web-Interface ist der Test des Entwurfes von jedem Rechner aus möglich. Dieses Web-Interface besteht aus einem php3-Skript das auf dem Computer mit der Spyderkarte gestartet wird. Dazu muß auf diesem Rechner ein Webserver[1] mit php3-Modul installiert sein.

Die Eingabe der Operanden erfolgt entweder in binärer oder hexadezimaler Schreibweise. Es ist auch eine Eingabe in dezimaler Schreibweise möglich, wobei die Konvertierung in die binäre Form dann vom php3-Skript übernommen wird. In Abbildung C.1 ist die Konvertierung der dezimalen 0,1 in die binäre Darstellung zu sehen. Dabei wird die am einfachsten zu implementierende Rundungsart (round to zero) verwendet. Das Befehlswort ist entweder über eine Auswahlliste oder durch Setzen der einzelnen Bits einzugeben. Normalerweise sind die Eingaben über die entsprechenden Auswahllisten (Operandenformat, Befehl, Kontextaddresse) vorzunehmen und die binäre Schreibweise wird nur als zusätzliche Information ausgegeben. Nachdem alle Eingaben getätigt sind, werden durch Anklicken des Start-„Knopfes“ die entsprechenden Daten zum FPGA übertragen. Die Übertragung der Daten werden vom php3-Skript durch mehrfachen Aufruf des `call_suma` Programmes (siehe B) vorgenommen. Nach der Übertragung aller Daten und des Befehlswortes [2] wird die Summationsmatrix vollständig ausgelesen. Das bedeutet also, daß durch das Betätigen des Start-„Knopfes“ 3-5 Schreib- und 10 Leseaufrufe des `call_suma` Programmes erfolgen.

Der Inhalt der Summationsmatrix wird in hexadezimaler Schreibweise ausgegeben. Dabei wird die Information welcher Addierer die führenden Nutzbits enthält (`msw_lacell`, siehe 4.12) genauso wie ein eventuell gesetztes sticky Bit (siehe 4.12) durch unterschiedliche Farbgebung der Addiererzellen signalisiert. Das Komma wird

[1] z.B. Apache 1.3.6 mit PHP 3.0.7

[2] Beschreiben von `REG0, REG1, REG2` und im double-Fall auch noch `REG10, REG11`

durch die farbliche Markierung der ersten (hexadezimalen) Vorkommaziffer[3] angedeutet.

Das abgebildete Beispiel zeigt, daß ein Konvertierungsfehler (siehe 1.4.3) auch durch anschließendes exaktes Rechnen nicht mehr behoben werden kann. Im abgebildeten Beispiel wird $0,1 \times 10$ im IEEE 754 single precision Format berechnet. Das exakte Ergebnis dieser Rechnung $= 1$ wäre in der Summationsmatrix auch darstellbar. Durch die Konvertierung von 0,1 wird aber $0,0999\ldots \times 10$ berechnet, so daß das Ergebnis $0,\mathrm{fffffff}_{16}$ ist, anstatt des korrekten[4] Wertes $1,0_{16}$.

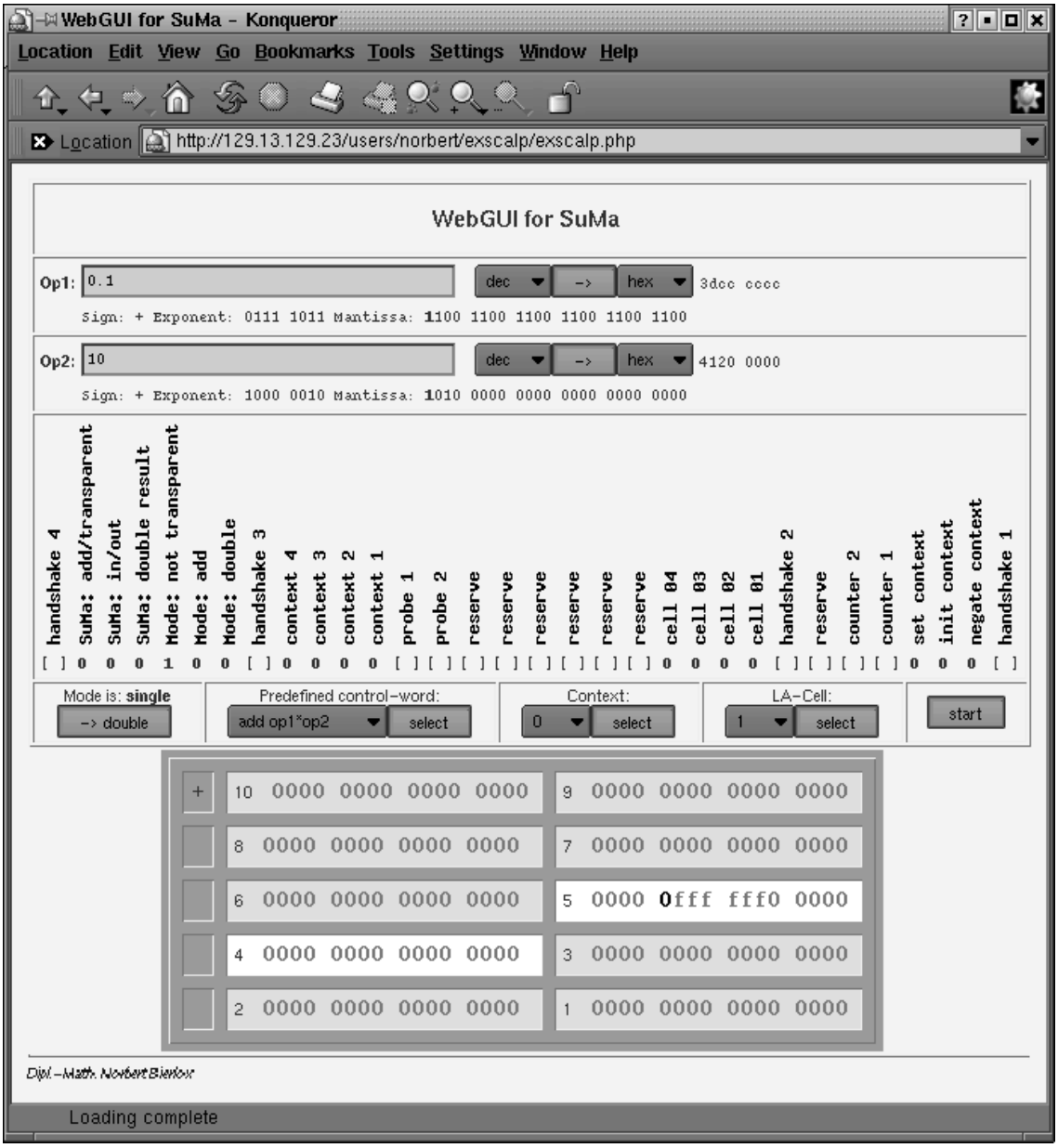


Abbildung C.1: Grafisches Interface für den Entwurf in der Spyder-Karte

[3] im Bespiel die Null vor den sechs fs

[4] Die Summationsmatrix rechnet korrekt, nur die Werte die sie als Eingabe erhält sind nicht exakt

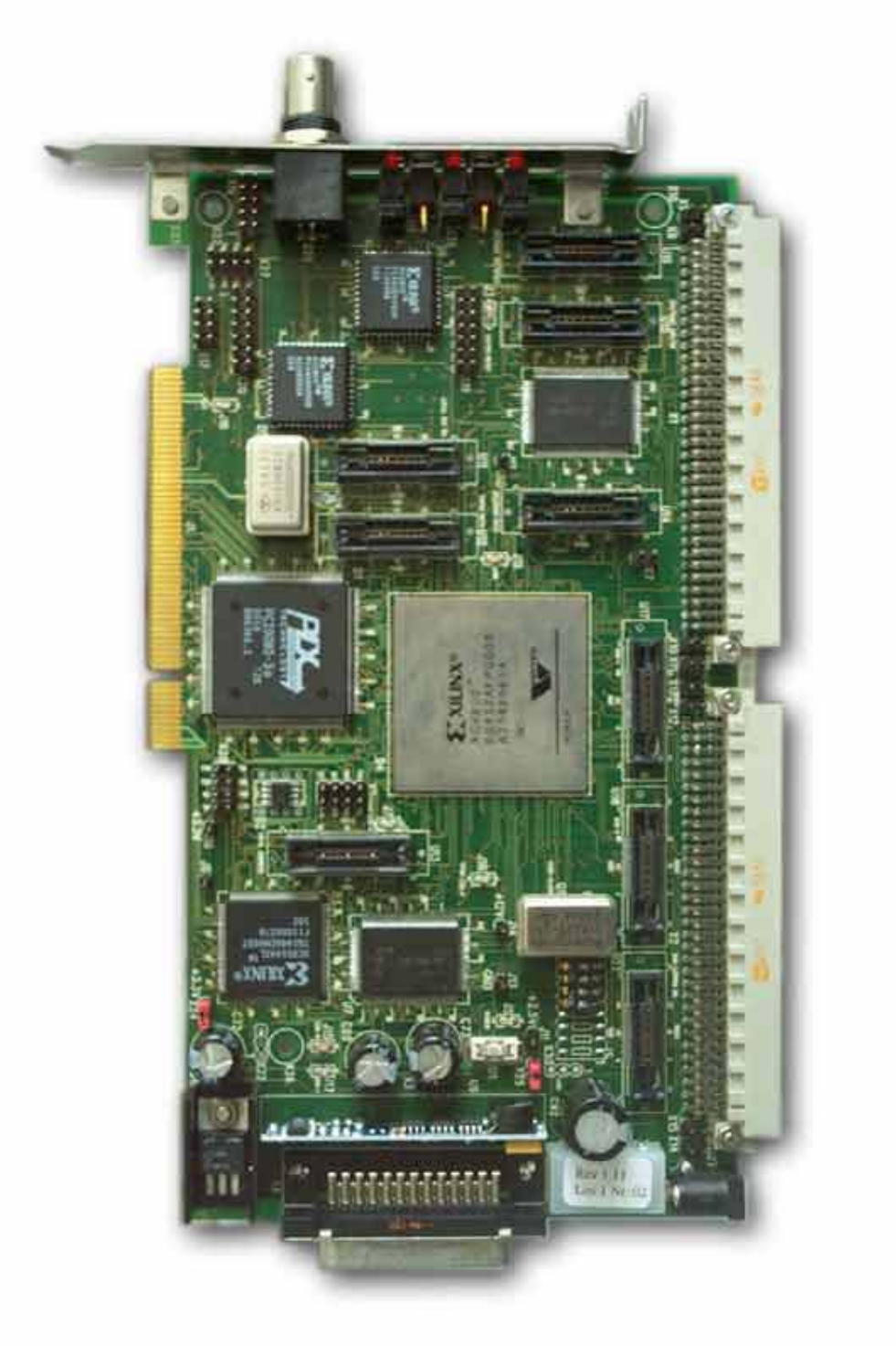


Abbildung C.2: Foto der verwendeten Entwicklungskarte Spyder-Virtex-X2

Anhang

D

VHDL Quelltexte

D.1 Datei: constants.vhd

```
--------------------------------------------------------------------------------
--
-- needed for data_select.vhd
-- double products are produced with 4 partial products in the following order:
--    double_counter
-- 1.  00             msw1 x msw2 = 42 bit partial product    64 bit left shift
-- 2.  01             lsw1 x lsw2 = 64 bit partial product    no shift
-- 3.  10             lsw1 x msw2 = 53 bit partial product    32 bit left shift
-- 4.  11             msw1 x lsw2 = 53 bit partial product    32 bit left shift
--
--------------------------------------------------------------------------------

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

package constants is

  constant context_store  : integer := 1;   -- needed in Decode for storing
                                            --   the context
  constant s_transfer_reg : integer := 0;   -- =1 transfer register
                                            --   architecture;
                                            -- testversion only
  constant carry_store    : integer := 1;   -- carry_store=1 carry store
                                            --   architecture
                                            -- carry_store=0 carry select
                                            --   architecture
  constant double_resolve : integer := 1;   -- =1 with double resolve register
                                            -- =0 without double resolve
                                            --   register
  constant context_b_ram  : integer := 0;   -- =1 context RAM is XILINX Block
                                            --   RAM
                                            -- =0 context RAM is XILINX
                                            --   distributed RAM

  constant default_prod : std_logic_vector(63 downto 0) := x"0000000000000000";
  constant const_prod   : std_logic_vector(63 downto 0) := x"0000000000000001";


  constant mode_mxm : std_logic_vector(2 downto 0) := "000";  -- msw1 x msw2
  constant mode_lxl : std_logic_vector(2 downto 0) := "001";  -- lsw1 x lsw2
  constant mode_lxm : std_logic_vector(2 downto 0) := "010";  -- lsw1 x msw2
  constant mode_mxl : std_logic_vector(2 downto 0) := "011";  -- msw1 x lsw2
  constant mode_sad : std_logic_vector(2 downto 0) := "100";  -- single add
                                                              -- mode
  constant mode_dad : std_logic_vector(2 downto 0) := "101";  -- double add
                                                              -- mode
  constant mode_tpt : std_logic_vector(2 downto 0) := "110";  -- transparent
                                                              -- mode
  constant mode_spn : std_logic_vector(2 downto 0) := "111";  -- single
                                                              -- precision mode

  constant sad_bias  : std_logic_vector(12 downto 0) := "0000001110110";
  -- single add mode shift bias 127-9 bias for "missed" second operand
  constant dad_bias  : std_logic_vector(12 downto 0) := "0000011111001";
  -- double add mode shift bias for "missed" second operand
  constant mxm_bias  : std_logic_vector(12 downto 0) := "0000100000110";
  -- mode_mxm shift bias
constant mxl_bias  : std_logic_vector(12 downto 0) := "0000011100110";
  -- mode_lxm and mode_mxl shift bias 32
constant lxl_bias  : std_logic_vector(12 downto 0) := "0000011000110";
  -- mode_lxl shift bias

  constant multi_depth : integer := 4;         -- how many pipe stages has
                                               -- the multiplier
  constant shift_depth : integer := 1;         -- how many pipe stages has
                                               -- the shifter
  constant dblres_depth: integer := double_resolve;
  constant decod_depth : integer := 1;         -- how many pipe stages has
                                               -- the decode unit
  constant dsel_depth  : integer := 1;         -- how many pipe stages has
                                               -- the data_select unit
  constant select_depth: integer := 1;         -- how many pipe stages has
                                               -- the selector

  constant context_depth : integer := 4;       -- the real depth is
                                               -- 2^context_depth
  constant lacell_width  : integer := 64;      -- width of a SuMa Addercell
  constant adder_width   : integer := 32;      -- width of a Partialadder in
                                               -- the SuMa Addercell
  constant num_cell      : integer := 10;      -- width of a SuMa Addercell
  constant SuMa_row      : integer := 5;       -- how much rows have the SuMa
  constant SuMa_col      : integer := 2;       -- how much Addercells are in
                                               -- one SuMa row

  -- due to the permant changing of the command-bits meaning we need constants
```

```
--
-- give command to suma
constant cmd_ok_flag4   : integer := 31;
-- commands for suma
constant cmd_add_tpt    : integer := 30;
constant cmd_in_out     : integer := 29;
constant cmd_double_res : integer := 28;
-- modes for expari/data_select
constant cmd_mode_tpt   : integer := 27;  -- if 0 then transparent
constant cmd_mode_add   : integer := 26;
constant cmd_mode_dbl   : integer := 25;
-- give command to suma
constant cmd_ok_flag3   : integer := 24;
-- context address
constant cmd_ctxt_bit4  : integer := 23;
constant cmd_ctxt_bit3  : integer := 22;
constant cmd_ctxt_bit2  : integer := 21;
constant cmd_ctxt_bit1  : integer := 20;
-- probe flags / reserve bits
constant probe_flag1    : integer := 19;
constant probe_flag2    : integer := 18;
-- cell address
constant cmd_substract  : integer := 17;
constant cmd_cb_res     : integer := 16;
constant reserve4       : integer := 15;
constant reserve3       : integer := 14;
constant reserve2       : integer := 13;
constant reserve1       : integer := 12;
constant cmd_cell_bit04 : integer := 11;
constant cmd_cell_bit03 : integer := 10;
constant cmd_cell_bit02 : integer :=  9;
constant cmd_cell_bit01 : integer :=  8;
-- give command to suma
constant cmd_ok_flag2   : integer :=  7;
constant cmd_def_prod   : integer :=  6;
-- double counter
constant cmd_dbl_cnt2   : integer :=  5;
constant cmd_dbl_cnt1   : integer :=  4;
-- context commands
constant cmd_cond_neg   : integer :=  3;
constant cmd_init_ctxt  : integer :=  2;
constant cmd_neg_ctxt   : integer :=  1;
-- give command to suma
constant cmd_ok_flag1   : integer :=  0;

  constant debug1         : integer := 159;
end constants;
```

D.2 Datei: versionnumber.vhd

```
--------------------------------------------------------------------------------
-- Constants needed to created the Version string in reg15
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

library work;
use work.constants.all;

package versionnumber is

  constant month            : std_logic_vector(3 downto 0) := "0110";
  constant str_arch         : std_logic_vector(0 downto 0) :=
                                        conv_std_logic_vector(s_transfer_reg,1);
  constant ctxt_store       : std_logic_vector(0 downto 0) :=
                                         conv_std_logic_vector(context_store,1);
  constant ctxt_bram        : std_logic_vector(0 downto 0) :=
                                         conv_std_logic_vector(context_b_ram,1);
  constant day              : std_logic_vector(4 downto 0) := "00001";
  constant test_version     : std_logic := '1' or str_arch(0);
  constant decode           : std_logic := '1';
  constant data_select      : std_logic := '1';
  constant expari           : std_logic := '1';
  constant multiplier       : std_logic := '1';
  constant selector         : std_logic := '1';
  constant pipes            : std_logic := '1';
  constant shifter          : std_logic := '1';
  constant suma             : std_logic := '1';
  constant def_product      : std_logic := '1';
  constant def_prod_bit     : integer := 10;   -- bitnumber of def_product
                                               -- (for spyder2suma)
  constant suma_carry       : std_logic_vector(0 downto 0) :=
                                          conv_std_logic_vector(carry_store,1);
  constant suma_read        : std_logic := '1';
  constant suma_write       : std_logic := '1';
  constant suma_msw         : std_logic := '1';
```

```
  constant suma_context     : std_logic := '1';
  constant suma_double_res  : std_logic_vector(0 downto 0) :=
                                    conv_std_logic_vector(double_resolve,1);
  constant suma_complenment : std_logic := '1';
  constant iteration3       : std_logic := '0';
  constant iteration2       : std_logic := '0';
  constant iteration1       : std_logic := '1';

  constant versionstring : std_logic_vector(31 downto 0) := month&
                           str_arch&ctxt_store&ctxt_bram&day&test_version&
                           decode&data_select&expari&multiplier&selector&pipes&
                           shifter&suma&def_product&suma_carry&
                           suma_read&suma_write&suma_msw&suma_context&
                           suma_double_res&suma_complenment&
                           iteration3&iteration2&iteration1;

end versionnumber;
```

D.3 Datei: components.vhd

```
-- synopsys translate_off
-- this library is only needed to avoid the black box warnings in synplify
-- synthesis translate_on
library synplify;
use synplify.attributes.all;
-- synopsys translate_on

library ieee;
use ieee.std_logic_1164.all;

library ieee;
use ieee.std_logic_unsigned.all;

library work;
use work.constants.all;

package components is

component reg is
  generic ( width            : integer := 8);
  port (    d                : in  std_logic_vector(width-1 downto 0);
            clk              : in  std_logic;
            rst              : in  std_logic;
            enable           : in std_logic;
            q                : out std_logic_vector(width-1 downto 0)  );
end component reg;

component nreg is
  generic ( width            : integer := 8);
  port (    d                : in  std_logic_vector(width-1 downto 0);
            clk              : in  std_logic;
            rst              : in  std_logic;
            enable           : in std_logic;
            q                : out std_logic_vector(width-1 downto 0)  );
end component nreg;

component breg is
  generic ( width            : integer := 8);
  port (    d                : in  std_logic_vector(width-1 downto 0);
            clk              : in  std_logic;
            rst              : in  std_logic;
            enable           : in std_logic;
            q                : out std_logic_vector(width-1 downto 0)  );
end component breg;

component bnreg is
  generic ( width            : integer := 8);
  port (    d                : in  std_logic_vector(width-1 downto 0);
            clk              : in  std_logic;
            rst              : in  std_logic;
            enable           : in std_logic;
            q                : out std_logic_vector(width-1 downto 0)  );
end component bnreg;

component pipe is
  generic ( width            : integer := 8 ;
            depth            : integer := 1   );
  port (    dpipe            : in  std_logic_vector(width-1 downto 0);
            clk              : in  std_logic;
            rst              : in  std_logic;
            enable           : in std_logic;
            qpipe            : out std_logic_vector(width-1 downto 0));
end component pipe;

component shift064_128 is
  port (   shift_amount      : in  std_logic_vector (  6 downto 0);
```

```
          shifter_in        : in  std_logic_vector ( 63 downto 0);
          clk, rst          : in  std_logic;
          shifter_out       : out std_logic_vector (127 downto 0)  );
end component shift064_128;

component selector is
  port (   data_in, data_prod,
          data_sel,data_def : in  std_logic_vector ( 63 downto 0);
          clk, rst          : in  std_logic;
          select0, select1  : in  std_logic;
          data_out          : out std_logic_vector ( 63 downto 0) );
end component selector;

component dbl_resolve is
  port (shifter_in   : in  std_logic_vector(127 downto 0);
        suma_in      : in  std_logic_vector(127 downto 0);
        clk          : in  std_logic;
        rst          : in  std_logic;
        sign_in      : in  std_logic;
        sign_suma    : in  std_logic;
        sign_out     : out std_logic;
        cmds_in      : in  std_logic_vector(6 downto 0);
        cmds_out     : out std_logic_vector(6 downto 0);
        data_to_suma : out std_logic_vector(127 downto 0));
end component dbl_resolve;

component Decode is
  generic ( registered  :     integer := 1 );
  port (    clk, rst    : in  std_logic;
            DecI_D0     : in  std_logic_vector (31 downto 0);
            DecI_D1     : in  std_logic_vector (31 downto 0);
            DecI_Cmd    : in  std_logic_vector (31 downto 0);
            DecO_D0     : out std_logic_vector (31 downto 0);
            DecO_D1     : out std_logic_vector (31 downto 0);
            DecO_expo0  : out std_logic_vector (10 downto 0);
            DecO_expo1  : out std_logic_vector (10 downto 0);
            DecO_hidden : out std_logic_vector ( 1 downto 0);
            DecO_ctxt   : out std_logic_vector ( 3 downto 0);
            DecO_dblcnt : out std_logic_vector ( 1 downto 0);
            DecO_cmds   : out std_logic_vector ( 6 downto 0);
            DecO_mode   : out std_logic_vector ( 2 downto 0);
            DecO_cells  : out std_logic_vector ( 3 downto 0);
            DecO_e_cmd  : out std_logic_vector ( 5 downto 0);
            DecO_probes : out std_logic_vector ( 1 downto 0);
            DecO_select : out std_logic_vector ( 1 downto 0);
            DecO_sign   : out std_logic
    );
end component Decode;


component expari is
  generic ( registered     :     integer := 1 );
  port (    clk, rst       : in  std_logic;
            mode_lacells   : in  std_logic_vector( 5 downto 0);
            ea, eb         : in  std_logic_vector(10 downto 0);
            data_mode      : in  std_logic_vector( 2 downto 0);
            double_counter : in  std_logic_vector( 1 downto 0);
            cells_to_read  : in  std_logic_vector( 3 downto 0);
            sub_context    : out std_logic_vector( 3 downto 0);
            shift_amount   : out std_logic_vector( 6 downto 0);
            activ_lacells  : out std_logic_vector( 9 downto 0);
            tpt_lacells    : out std_logic_vector( 9 downto 0);
            high_lacells   : out std_logic_vector( 9 downto 0)  );
end component expari;


component data_select is
  generic ( registered           :     integer := 1 );
  port (    clk, rst             : in  std_logic;
            data0_in,  data1_in  : in  std_logic_vector(31 downto 0);
            hidden_bit           : in  std_logic_vector( 1 downto 0);
            data_mode            : in  std_logic_vector( 2 downto 0);
            double_counter       : in  std_logic_vector( 1 downto 0);
            data0_out, data1_out : out std_logic_vector(31 downto 0)  );
end component data_select;

component context_ram is
  generic ( depth          : integer := context_depth;
            cell_width     : integer := lacell_width   );
  port (   address         : in  std_logic_vector(context_depth-1 downto 0);
           clk             : in  std_logic;
           write_en        : in  std_logic;
           data_in         : in  std_logic_vector(lacell_width+1 downto 0);
           data_out        : out std_logic_vector(lacell_width+1 downto 0)  );
end component context_ram;

component context_bram is
  generic ( depth          : integer := context_depth;
```

```
            cell_width     : integer := lacell_width   );
  port (   address         : in  std_logic_vector(context_depth-1 downto 0);
           clk             : in  std_logic;
           write_en        : in  std_logic;
           data_in         : in  std_logic_vector(lacell_width+1 downto 0);
           data_out        : out std_logic_vector(lacell_width+1 downto 0)  );
end component context_bram;

component csa64 is
  generic (partadder_width : integer := adder_width);
  port (   op_a            : in  std_logic_vector(lacell_width-1 downto 0);
           op_b            : in  std_logic_vector(lacell_width-1 downto 0);
           addsub          : in  std_logic;
           cout00          : out std_logic;
           cout01          : out std_logic;
           cout10          : out std_logic;
           cout11          : out std_logic;
           out_xl          : out std_logic_vector(31 downto 0);
           out_yl          : out std_logic_vector(31 downto 0);
           out_xh          : out std_logic_vector(31 downto 0);
           out_yh          : out std_logic_vector(31 downto 0));
end component csa64;

component cs64 is
  generic (partadder_width : integer := adder_width);
  port (   op_a            : in  std_logic_vector(lacell_width-1 downto 0);
           op_b            : in  std_logic_vector(lacell_width-1 downto 0);
           addsub          : in  std_logic;
           cin_l           : in  std_logic;
           cin_h           : in  std_logic;
           cout_l          : out std_logic;
           cout_h          : out std_logic;
           sum_l           : out std_logic_vector(31 downto 0);
           sum_h           : out std_logic_vector(31 downto 0));
end component cs64;

component lacell is
  generic (partadder_width : integer := adder_width );
  port (   a               : in  std_logic_vector(lacell_width-1 downto 0);
           context         : in  std_logic_vector(context_depth-1 downto 0);
           clk_store       : in  std_logic;
           addsub          : in  std_logic;
           store           : in  std_logic;
           ci_low          : in  std_logic;
           carry           : in  std_logic;
           activ_cell      : in  std_logic;
           high_cell       : in  std_logic;
           enable_out      : in  std_logic;
           load_data       : in  std_logic;
           neg_context     : in  std_logic;
           all_zero        : out std_logic;
           all_one         : out std_logic;
           co_low          : out std_logic;
           co_high1        : out std_logic;
           co_high2        : out std_logic;
           sum             : out std_logic_vector(lacell_width+1 downto 0));
end component lacell;

component lacell_cs is
  generic (partadder_width : integer := adder_width );
  port (   a               : in  std_logic_vector(lacell_width-1 downto 0);
           context         : in  std_logic_vector(context_depth-1 downto 0);
           clk_store       : in  std_logic;
           init_ctxt       : in  std_logic;
           addsub          : in  std_logic;
           store           : in  std_logic;
           carry_in        : in  std_logic;
           carry_out       : out std_logic;
           activ_cell      : in  std_logic;
           high_cell       : in  std_logic;
           enable_out      : in  std_logic;
           load_data       : in  std_logic;
           neg_context     : in  std_logic;
           all_zero        : out std_logic;
           all_one         : out std_logic;
           sum             : out std_logic_vector(lacell_width+1 downto 0));
end component lacell_cs;

component SuMa is
  generic (partadder_width : integer := adder_width );

  port (   data_in         : in  std_logic_vector(lacell_width*SuMa_col-1
                                                                  downto 0);
           sign            : in  std_logic;
           activ_lacells   : in  std_logic_vector(num_cell-1 downto 0);
           high_lacells    : in  std_logic_vector(num_cell-1 downto 0);
           tpt_lacells     : in  std_logic_vector(num_cell-1 downto 0);
           clk             : in  std_logic;
```

```
          clk_store       : in  std_logic;
          rst             : in  std_logic;
          context         : in  std_logic_vector(context_depth-1 downto 0);
          cb_resolve      : in  std_logic;
          neg_context     : in  std_logic;
          cond_neg        : in  std_logic;
          add_tpt         : in  std_logic;
          in_out          : in  std_logic;
          double_res      : in  std_logic;
          init_ctxt       : in  std_logic;
          data_out        : out std_logic_vector(131 downto 0);
          context_out     : out std_logic_vector( 15 downto 0);
          debug_out       : out std_logic_vector( 15 downto 0)  );
end component SuMa;


component M_Spez_SuMa is
    port ( clk       : in  std_logic;
          clk_store : in  std_logic;
          MIDefprod : in  std_logic;
          MII       : in  std_logic;
          MID0      : in  std_logic_vector ( 31 downto 0);
          MID1      : in  std_logic_vector ( 31 downto 0);
          MISCmd    : in  std_logic_vector ( 31 downto 0);
          MODExpo   : out std_logic_vector ( 15 downto 0);
          MOVData   : out std_logic_vector (  1 downto 0);
          MOData    : out std_logic_vector (131 downto 0);
          MODebug   : out std_logic_vector (debug1 downto 0)  );
end component M_Spez_SuMa;


-------------------------------------------------------------------------------
-- Xilinx Core-Generator Moduls
-------------------------------------------------------------------------------



   ---------------------------------------------------------------------------
   -- Xilinx Memory (distributed RAM)
   ---------------------------------------------------------------------------


component ramsuu64x16 is
  port (A       : in  std_logic_vector(context_depth-1 downto 0);
        CLK     : in  std_logic;
        D       : in  std_logic_vector(lacell_width-1 downto 0);
        WE      : in  std_logic;
        SPO     : out std_logic_vector(lacell_width-1 downto 0) );
end component ramsuu64x16;
-- synopsys translate_off
-- synthesis translate_on
attribute syn_black_box of ramsuu64x16: component is true;
-- synopsys translate_on

component ramsuu2x16 is
  port (A       : in  std_logic_vector(context_depth-1 downto 0);
        CLK     : in  std_logic;
        D       : in  std_logic_vector(1 downto 0);
        WE      : in  std_logic;
        SPO     : out std_logic_vector(1 downto 0) );
end component ramsuu2x16;
-- synopsys translate_off
-- synthesis translate_on
attribute syn_black_box of ramsuu2x16: component is true;
-- synopsys translate_on

-- initialised with all '1' for borrows
component ramsuu2x16borrow is
  port (A       : in  std_logic_vector(context_depth-1 downto 0);
        CLK     : in  std_logic;
        D       : in  std_logic_vector(1 downto 0);
        WE      : in  std_logic;
        SPO     : out std_logic_vector(1 downto 0) );
end component ramsuu2x16borrow;
-- synopsys translate_off
-- synthesis translate_on
attribute syn_black_box of ramsuu2x16borrow: component is true;
-- synopsys translate_on

component ramsuu16x16 is
  port (A       : in  std_logic_vector(context_depth-1 downto 0);
        CLK     : in  std_logic;
        D       : in  std_logic_vector(15 downto 0);
        WE      : in  std_logic;
        SPO     : out std_logic_vector(15 downto 0) );
end component ramsuu16x16;
-- synopsys translate_off
-- synthesis translate_on
attribute syn_black_box of ramsuu16x16: component is true;
```

```
-- synopsys translate_on

component ramsuu32x16 is
  port (A       : in  std_logic_vector(context_depth-1 downto 0);
        CLK     : in  std_logic;
        D       : in  std_logic_vector(31 downto 0);
        WE      : in  std_logic;
        SPO     : out std_logic_vector(31 downto 0) );
end component ramsuu32x16;
-- synopsys translate_off
-- synthesis translate_on
attribute syn_black_box of ramsuu32x16: component is true;
-- synopsys translate_on

   ------------------------------------------------------------------------
   -- Xilinx Memory (distributed RAM)
   ------------------------------------------------------------------------

component brams32x128 is
  port (addr    : in  std_logic_vector(6 downto 0);
        clk     : in  std_logic;
        di      : in  std_logic_vector(31 downto 0);
        we      : in  std_logic;
        rst     : in  std_logic;
        do      : out std_logic_vector(31 downto 0) );
end component brams32x128;
-- synopsys translate_off
-- synthesis translate_on
attribute syn_black_box of brams32x128: component is true;
-- synopsys translate_on

-- borrow RAM: initialised with "ffffffff"
component brams32x128b is
  port (addr    : in  std_logic_vector(6 downto 0);
        clk     : in  std_logic;
        di      : in  std_logic_vector(31 downto 0);
        we      : in  std_logic;
        rst     : in  std_logic;
        do      : out std_logic_vector(31 downto 0) );
end component brams32x128b;
-- synopsys translate_off
-- synthesis translate_on
attribute syn_black_box of brams32x128b: component is true;
-- synopsys translate_on
```

```
   ------------------------------------------------------------------------
   -- Xilinx Adders
   ------------------------------------------------------------------------

component addsub32uu_reg is
  port (A       : in  std_logic_vector(31 downto 0);
        B       : in  std_logic_vector(31 downto 0);
        C_IN    : in  std_logic;
        C_OUT   : out std_logic;
        ADD     : in  std_logic;
        S       : out std_logic_vector(31 downto 0)  );
end component addsub32uu_reg;
-- synopsys translate_off
-- synthesis translate_on
attribute syn_black_box of addsub32uu_reg: component is true;
-- synopsys translate_on

component addsub64uu_reg is
  port (A       : in  std_logic_vector(63 downto 0);
        B       : in  std_logic_vector(63 downto 0);
        C_IN    : in  std_logic;
        C_OUT   : out std_logic;
        ADD     : in  std_logic;
        S       : out std_logic_vector(63 downto 0)       );
end component addsub64uu_reg;
-- synopsys translate_off
-- synthesis translate_on
attribute syn_black_box of addsub64uu_reg: component is true;
-- synopsys translate_on


component addsub32uu is
  port (A       : in  std_logic_vector(31 downto 0);
        B       : in  std_logic_vector(31 downto 0);
        C_IN    : in  std_logic;
        C_OUT   : out std_logic;
        ADD     : in  std_logic;
        S       : out std_logic_vector(31 downto 0)  );
end component addsub32uu;
-- synopsys translate_off
```

```
-- synthesis translate_on
attribute syn_black_box of addsub32uu: component is true;
-- synopsys translate_on

component addsub64uu is
  port (A       : in  std_logic_vector(63 downto 0);
        B       : in  std_logic_vector(63 downto 0);
        C_IN    : in  std_logic;
        C_OUT   : out std_logic;
        ADD     : in  std_logic;
        S       : out std_logic_vector(63 downto 0)       );
end component addsub64uu;
-- synopsys translate_off
-- synthesis translate_on
attribute syn_black_box of addsub64uu: component is true;
-- synopsys translate_on

   ------------------------------------------------------------------------------
   -- Xilinx Multipliers
   ------------------------------------------------------------------------------

component mul32x32u_pipe is
  port (
        A       : in std_logic_vector(31 downto 0);
        B       : in std_logic_vector(31 downto 0);
        CLK     : in std_logic;
        P       : out std_logic_vector(63 downto 0));
end component mul32x32u_pipe;
-- synopsys translate_off
-- synthesis translate_on
attribute syn_black_box of mul32x32u_pipe: component is true;
-- synopsys translate_on

component mul32x32u_cmb
        port (
        a: IN std_logic_VECTOR(31 downto 0);
        b: IN std_logic_VECTOR(31 downto 0);
        p: OUT std_logic_VECTOR(63 downto 0));
end component;
-- synopsys translate_off
-- synthesis translate_on
attribute syn_black_box of mul32x32u_cmb: component is true;
-- synopsys translate_on

end components;
```

D.4 Datei: `cells.vhd`

```
library ieee;
use ieee.std_logic_1164.all;

library work;
use work.components.all;

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

architecture rtl_reg of reg is
begin
  dffsrst: process (clk)
  begin  -- process dffsrst
    if clk'event and clk = '1' then
      if rst='0' then
        q <= (others => '0');
      elsif (enable='1') then
        q <= d;
      end if;
    end if;
  end process dffsrst;
end rtl_reg;


library ieee;
use ieee.std_logic_1164.all;

library work;
use work.components.all;


entity nreg is
  generic ( width  : integer := 8);
  port (    d      : in  std_logic_vector(width-1 downto 0);
```

```
            clk    : in  std_logic;
            rst    : in  std_logic;
            enable : in  std_logic;
            q      : out std_logic_vector(width-1 downto 0)  );
end nreg;

architecture rtl_reg of nreg is
begin
  dffsrst: process (clk)
  begin  -- process dffsrst
    if clk'event and clk = '0' then
      if rst='0' then
        q <= (others => '0');
      elsif (enable='1') then
        q <= d;
      end if;
    end if;
  end process dffsrst;
end rtl_reg;



library ieee;
use ieee.std_logic_1164.all;

library work;
use work.components.all;

entity breg is
  generic ( width  : integer := 8);
  port (    d      : in  std_logic_vector(width-1 downto 0);
            clk    : in  std_logic;
            rst    : in  std_logic;
            enable : in  std_logic;
            q      : out std_logic_vector(width-1 downto 0)  );
end breg;

architecture rtl_reg of breg is
begin
  dffsnrst: process (clk)
  begin  -- process dffsrst
    if clk'event and clk = '1' then
      if rst='0' then
        q <= (others => '1');
      elsif (enable='1') then
        q <= d;
      end if;
    end if;
  end process dffsnrst;
end rtl_reg;


library ieee;
use ieee.std_logic_1164.all;

library work;
use work.components.all;

entity bnreg is
  generic ( width  : integer := 8);
  port (    d      : in  std_logic_vector(width-1 downto 0);
            clk    : in  std_logic;
            rst    : in  std_logic;
            enable : in  std_logic;
            q      : out std_logic_vector(width-1 downto 0)  );
end bnreg;

architecture rtl_reg of bnreg is
begin
  dffsnrst: process (clk)
  begin  -- process dffsrst
    if clk'event and clk = '0' then
      if rst='0' then
        q <= (others => '1');
      elsif (enable='1') then
        q <= d;
      end if;
    end if;
  end process dffsnrst;
end rtl_reg;


library ieee;
use ieee.std_logic_1164.all;

library work;
use work.components.all;
```

```
entity pipe is
  generic ( width  : integer := 8 ;
            depth  : integer := 1  );
  port (    dpipe  : in  std_logic_vector(width-1 downto 0);
            clk    : in  std_logic;
            rst    : in  std_logic;
            enable : in  std_logic;
            qpipe  : out std_logic_vector(width-1 downto 0));
end pipe;

architecture structural of pipe is
  signal data_connect : std_logic_vector(((depth+1)*width)-1 downto 0);
begin
  data_connect(width-1 downto 0) <= dpipe;
  pipeline: for index in 0 to depth-1 generate
  fexp_pipe: block
                                -- fexp insists on a block statement
    signal data_in  : std_logic_vector(width-1 downto 0);
    signal data_out : std_logic_vector(width-1 downto 0);
                                -- Synplicity do nonsense
                                -- without these signals
    begin
      data_in  <= data_connect(((index+1)*width)-1 downto (index*width));
      data_out <= data_connect(((index+2)*width)-1 downto ((index+1)*width));
      stageX : reg
        generic map ( width )
           port map (d      =>
                     data_connect(((index+1)*width)-1 downto (index*width)),
                     clk    => clk,
                     rst    => rst,
                     enable => enable,
                     q      =>
                     data_connect(((index+2)*width)-1 downto
                                                          ((index+1)*width)));
    end block;
  end generate pipeline;
  qpipe <= data_connect(((depth+1)*width)-1 downto (depth*width));
end structural;


library ieee;
use ieee.std_logic_1164.all;

library ieee;
use ieee.std_logic_unsigned.all;

library work;
use work.components.all;

entity shift64_128 is
  port (   shift_amount : in  std_logic_vector (  6 downto 0);
           shifter_in   : in  std_logic_vector ( 63 downto 0);
           clk, rst     : in  std_logic;
           shifter_out  : out std_logic_vector (127 downto 0)  );
end shift64_128;

architecture shift_cohen of shift64_128 is
  signal enable : std_logic;
  signal wires_in, wires0, wires1, wires_out : std_logic_vector(127 downto 0);
begin
  enable <= '1';
  wires_in(127 downto shifter_in'length) <= (others => '0');
  wires_in(shifter_in'length-1 downto 0) <=  shifter_in;

  reg_in:  reg generic map ( width => 128 )
               port map    ( d     => wires_in,
                             clk   => clk, rst => rst, enable => enable,
                             q     => wires0);
  reg_out: reg generic map ( width => 128 )
               port map    ( d     => wires_out,
                             clk   => clk, rst => rst, enable => enable,
                             q     => shifter_out);

  barrel: process(wires0, shift_amount)
    variable DIn_v  : Std_Logic_vector(shifter_out'length - 1 downto 0);
    variable amount : integer;
  begin
    amount := conv_integer(shift_amount);
    DIn_v  := wires0;
    if amount=0 then wires_out <= DIn_v;
    else
      for lp in DIn_v'length-1 downto 1 loop
        if lp = amount then
          wires_out <= DIn_v((DIn_v'left - lp) downto 0) &
                       DIn_v(DIn_v'left downto (DIn_v'left + 1 - lp));
        end if;
      end loop;
    end if;
```

```
  end process barrel;

end shift_cohen;


library ieee;
use ieee.std_logic_1164.all;

library ieee;
use ieee.std_logic_unsigned.all;

library work;
use work.components.all;


entity shift064_128 is
  port (   shift_amount : in  std_logic_vector (  6 downto 0);
           shifter_in   : in  std_logic_vector ( 63 downto 0);
           clk, rst     : in  std_logic;
           shifter_out  : out std_logic_vector (127 downto 0)  );
end shift064_128;

architecture shift_chang of shift064_128 is
  constant N : integer := 128;
  constant M : integer := 7;
  type arytype is array (M downto 0) of std_logic_vector(N-1 downto 0);
  signal intsig, left_s, pass_s : arytype;
  signal enable : std_logic;
  signal wires_in, wires_out : std_logic_vector(127 downto 0);
begin
  enable <= '1';
  wires_in(127 downto  shifter_in'length) <= (others => '0');
  wires_in(shifter_in'length-1 downto  0) <= shifter_in;

  reg_out: reg generic map ( width => 128 )
               port map    ( d     => wires_out,
                             clk   => clk, rst => rst, enable => enable,
                             q     => shifter_out);
  intsig(0) <= wires_in;

  muxgen: for j in 1 to M generate
    pass_s(j) <= intsig(j-1);
    left_s(j) <= intsig(j-1)(N-2**(j-1)-1 downto 0) &
                 intsig(j-1)(N-1 downto N-2**(j-1));
    intsig(j) <= pass_s(j) when shift_amount(j-1)='0' else left_s(j);
  end generate muxgen;
  wires_out   <= intsig(M);
end shift_chang;


library ieee;
use ieee.std_logic_1164.all;

library ieee;
use ieee.std_logic_unsigned.all;

library work;
use work.components.all;

library work;
use work.constants.all;

entity selector is
  port (   data_in, data_prod,
           data_sel,data_def   : in  std_logic_vector ( 63 downto 0);
           clk, rst            : in  std_logic;
           select0, select1    : in  std_logic;
           data_out            : out std_logic_vector ( 63 downto 0) );
end selector;


architecture struct of selector is
   signal data0, data1, data2,
          data3, data_res : std_logic_vector ( 63 downto 0);
   signal VCC : std_logic;

begin
   VCC   <= '1';
   data3 <= data_def;
   data2 <= data_sel;
   data1 <= data_prod;
   data0 <= data_in;

   data_res <=  data0 when (select1='1' and select0='0') else
                data1 when (select1='0' and select0='1') else
                data2 when (select1='1' and select0='1') else
                data3;
```

```
   reg_data_out : reg generic map ( width  => 64)
                      port map    ( d      => data_res,
                                    clk    => clk,
                                    rst    => rst,
                                    enable => VCC,
                                    q      => data_out);

end struct;


library ieee;
use ieee.std_logic_1164.all;

library ieee;
use ieee.std_logic_unsigned.all;

library work;
use work.components.all;

library work;
use work.constants.all;


entity dbl_resolve is

  port (shifter_in   : in  std_logic_vector(127 downto 0);
        suma_in      : in  std_logic_vector(127 downto 0);
        clk          : in  std_logic;
        rst          : in  std_logic;
        sign_in      : in  std_logic;
        sign_suma    : in  std_logic;
        sign_out     : out std_logic;
        cmds_in      : in  std_logic_vector(6 downto 0);
        cmds_out     : out std_logic_vector(6 downto 0);
        data_to_suma : out std_logic_vector(127 downto 0));

end dbl_resolve;

architecture struct of dbl_resolve is
  signal suma_reg_in    : std_logic_vector(128 downto 0);
  signal suma_reg_out   : std_logic_vector(128 downto 0);
  signal to_suma_in     : std_logic_vector(128 downto 0);
  signal to_suma_out    : std_logic_vector(128 downto 0);
  signal suma_in_en     : std_logic_vector(  0 downto 0);
  signal suma_in_enable : std_logic_vector(  0 downto 0);
  signal wire_enable    : std_logic;
  signal VCC            : std_logic;
begin  -- struct

  VCC            <= '1';
  suma_reg_in    <= suma_in&(not sign_suma);
  suma_in_en(0)  <= '1'          when cmds_in(1)='1' and cmds_in(3)='1' else
                    '0';
  to_suma_in     <= suma_reg_out when cmds_in(1)='1' and cmds_in(3)='0' else
                    shifter_in&sign_in;
  sign_out       <= to_suma_out(0);
  data_to_suma   <= to_suma_out(128 downto 1);
  wire_enable    <= suma_in_enable(0);

  suma_in : nreg  -- Attention nreg activ if clk'event and clk='0'
    generic map (width => 129)
    port map ( d      => suma_reg_in,
               clk    => clk,
               rst    => rst,
               enable => wire_enable,
               q      => suma_reg_out );

  suma_enable_pipe : pipe generic map (width => 1, depth => 1 )
  port map (dpipe  => suma_in_en,
            clk    => clk,
            rst    => rst,
            enable => VCC,
            qpipe  => suma_in_enable);

  to_suma : reg
    generic map (width => 129)
    port map ( d      => to_suma_in,
               clk    => clk,
               rst    => rst,
               enable => VCC,
               q      => to_suma_out );

  cmds_out : reg
    generic map (width => 7)
    port map ( d      => cmds_in,
               clk    => clk,
               rst    => rst,
               enable => VCC,
```

```
              q      => cmds_out );

end struct;
```

D.5 Datei: context.vhd

```
library ieee;
use ieee.std_logic_1164.all;

library work;
use work.constants.all;

library work;
use work.components.all;



entity context_ram is
  generic ( depth      : integer := context_depth;
            cell_width : integer := lacell_width  );
  port (   address      : in  std_logic_vector(context_depth-1 downto 0);
           clk          : in  std_logic;
           write_en     : in  std_logic;
           data_in      : in  std_logic_vector(lacell_width+1 downto 0);
           data_out     : out std_logic_vector(lacell_width+1 downto 0)  );
end context_ram;


architecture xilinx_struct of context_ram is
  signal VCC, GND            : std_logic;
  signal in_data, out_data   : std_logic_vector(lacell_width-1 downto 0);
  signal in_data0, out_data0 : std_logic_vector(adder_width-1 downto 0);
  signal in_data1, out_data1 : std_logic_vector(adder_width-1 downto 0);
  signal in_ctrl, out_ctrl   : std_logic_vector(1 downto 0);
begin  -- xilinx_struct
  VCC <= '1';
  GND <= '0';

  in_ctrl <= data_in(lacell_width+1 downto lacell_width);

  data_out(lacell_width+1 downto lacell_width) <= out_ctrl;

  width_64: if (depth=4 and adder_width=64) generate
    in_data <= data_in(lacell_width-1 downto 0);
    data_out(lacell_width-1 downto 0) <= out_data;
  ctxt_ram_data : ramsuu64x16
            port map ( A       => address,
                       CLK     => clk,
                       D       => in_data,
                       WE      => write_en,
                       SPO     => out_data );
  ctxt_ram_ctrl : ramsuu2x16
            port map ( A       => address,
                       CLK     => clk,
                       D       => in_ctrl,
                       WE      => write_en,
                       SPO     => out_ctrl );
  end generate width_64;


  width_32: if (depth=4 and adder_width=32) generate
    in_data0 <= data_in(adder_width-1  downto 0);
    in_data1 <= data_in(lacell_width-1 downto adder_width);
    data_out(adder_width-1  downto 0)           <= out_data0;
    data_out(lacell_width-1 downto adder_width) <= out_data1;
  ctxt_ram_data0 : ramsuu32x16
            port map ( A       => address,
                       CLK     => clk,
                       D       => in_data0,
                       WE      => write_en,
                       SPO     => out_data0 );
  ctxt_ram_data1 : ramsuu32x16
            port map ( A       => address,
                       CLK     => clk,
                       D       => in_data1,
                       WE      => write_en,
                       SPO     => out_data1 );
  ctxt_ram_ctrl : ramsuu2x16
            port map ( A       => address,
                       CLK     => clk,
                       D       => in_ctrl,
                       WE      => write_en,
                       SPO     => out_ctrl );
  end generate width_32;



-- synopsys translate_off
```

```
-- synthesis translate_off
  assert (depth=4)
    report "no more contexts then 16 implemented. only Context_depth=4 is
            valid"
    severity FAILURE;
  assert (lacell_width=64)
    report "only SuMa Addercells of 64 bit width supported"
    severity FAILURE;
-- synthesis translate_on
-- synopsys translate_on

end xilinx_struct;


------------------------------------------------------------------------------
-- context memory with xilinx block ram
------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

library work;
use work.constants.all;

library work;
use work.components.all;


entity context_bram is
  generic ( depth      : integer := context_depth;
            cell_width : integer := lacell_width  );
  port (   address     : in  std_logic_vector(context_depth-1 downto 0);
           clk         : in  std_logic;
           write_en    : in  std_logic;
           data_in     : in  std_logic_vector(lacell_width+1 downto 0);
           data_out    : out std_logic_vector(lacell_width+1 downto 0)  );
end context_bram;


architecture xilinx_struct of context_bram is
  signal VCC, GND            : std_logic;
  signal in_data, out_data   : std_logic_vector(lacell_width-1 downto 0);
  signal in_data0, out_data0 : std_logic_vector(adder_width-1 downto 0);
  signal in_data1, out_data1 : std_logic_vector(adder_width-1 downto 0);
  signal in_ctrl, out_ctrl   : std_logic_vector(1 downto 0);
  signal long_address         : std_logic_vector(6 downto 0);
begin  -- xilinx_struct
  VCC <= '1';
  GND <= '0';

  long_address    <= "000" & address;


  in_ctrl <= data_in(lacell_width+1 downto lacell_width);

  data_out(lacell_width+1 downto lacell_width) <= out_ctrl;

  width_32: if (lacell_width=64) generate
    in_data0 <= data_in(adder_width-1  downto 0);
    in_data1 <= data_in(lacell_width-1 downto adder_width);
    data_out(adder_width-1  downto 0)           <= out_data0;
    data_out(lacell_width-1 downto adder_width) <= out_data1;
  ctxt_ram_data0 : brams32x128
             port map ( addr    => long_address,
                        clk     => clk,
                        di      => in_data0,
                        we      => write_en,
                        rst     => VCC,
                        do      => out_data0 );
  ctxt_ram_data1 : brams32x128
             port map ( addr    => long_address,
                        clk     => clk,
                        di      => in_data1,
                        we      => write_en,
                        rst     => VCC,
                        do      => out_data1 );
  ctxt_ram_ctrl : ramsuu2x16
             port map ( A       => address,
                        CLK     => clk,
                        D       => in_ctrl,
                        WE      => write_en,
                        SPO     => out_ctrl );
  end generate width_32;



-- synopsys translate_off
-- synthesis translate_off
```

```
  assert (depth=<7)
    report "no more contexts then 128 implemented. only Context_depth=<7 is
            valid"
    severity FAILURE;
  assert (lacell_width=64)
    report "only SuMa Addercells of 64 bit width supported"
    severity FAILURE;
-- synthesis translate_on
-- synopsys translate_on

end xilinx_struct;
```

D.6 Datei: addierer.vhd

```
library ieee;
use ieee.std_logic_1164.all;

library work;
use work.components.all;

library work;
use work.constants.all;

entity csa64 is

  generic (partadder_width : integer := adder_width );

  port (op_a   : in  std_logic_vector(63 downto 0);
        op_b   : in  std_logic_vector(63 downto 0);
        addsub : in  std_logic;         -- '1'=add '0'=sub
        cout00 : out std_logic;
        cout01 : out std_logic;
        cout10 : out std_logic;         -- if partwidth=32
        cout11 : out std_logic;         -- we need 4 carry out
        out_xl : out std_logic_vector(31 downto 0);  -- lsbs a+b
        out_yl : out std_logic_vector(31 downto 0);  -- lsbs a+b+1
        out_xh : out std_logic_vector(31 downto 0);  -- msbs a+b
        out_yh : out std_logic_vector(31 downto 0)); -- msbs a+b+1
end csa64;


architecture xilinx_struct of csa64 is
  signal xsum, ysum         : std_logic_vector(lacell_width-1 downto 0);
  signal xsuml, xsumh       : std_logic_vector(31 downto 0);
  signal ysuml, ysumh       : std_logic_vector(31 downto 0);
  signal op_al, op_ah       : std_logic_vector(31 downto 0);
  signal op_bl, op_bh       : std_logic_vector(31 downto 0);
  signal VCC, GND           : std_logic;
begin  -- xilinx_struct
  VCC <= '1';
  GND <= '0';

  op_al <= op_a(31 downto  0);
  op_ah <= op_a(lacell_width-1 downto 32);
  op_bl <= op_b(31 downto  0);
  op_bh <= op_b(lacell_width-1 downto 32);

  partwidth_32: if (partadder_width=32) generate
    add_a_b_0l : addsub32uu_reg
            port map (  A       => op_al,
                        B       => op_bl,
                        C_IN    => GND,
                        C_OUT   => cout00,
                        ADD     => addsub,
                        S       => out_xl);
    add_a_b_0h : addsub32uu_reg
            port map (  A       => op_ah,
                        B       => op_bh,
                        C_IN    => GND,
                        C_OUT   => cout01,
                        ADD     => addsub,
                        S       => out_xh);
    add_a_b_1l : addsub32uu_reg
            port map (  A       => op_al,
                        B       => op_bl,
                        C_IN    => VCC,
                        C_OUT   => cout10,
                        ADD     => addsub,
                        S       => out_yl);
    add_a_b_1h : addsub32uu_reg
            port map (  A       => op_ah,
                        B       => op_bh,
                        C_IN    => VCC,
                        C_OUT   => cout11,
                        ADD     => addsub,
                        S       => out_yh);

  end generate partwidth_32;
```

```
  partwidth_64: if (partadder_width=64) generate
    cout00 <= GND;    -- with this initialisation there is
    cout10 <= VCC;    -- LaCell for the partadder_width=32

    add_a_b_0 : addsub64uu_reg
            port map (  A       => op_a,
                        B       => op_b,
                        C_IN    => GND,
                        C_OUT   => cout01,
                        ADD     => addsub,
                        S       => xsum);
    add_a_b_1 : addsub64uu_reg
            port map (  A       => op_a,
                        B       => op_b,
                        C_IN    => VCC,
                        C_OUT   => cout11,
                        ADD     => addsub,
                        S       => ysum);

    out_xl <= xsum(31 downto  0);
    out_xh <= xsum(lacell_width-1 downto 32);
    out_yl <= ysum(31 downto  0);
    out_yh <= ysum(lacell_width-1 downto 32);
  end generate partwidth_64;

-- synopsys translate_off
-- synthesis translate_off
  assert (partadder_width=64 or partadder_width=32)
    report "only partial adders of width 64 or 32 are supported"
    severity FAILURE;
-- synthesis translate_on
-- synopsys translate_on

end xilinx_struct;

--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
-- carry store Adder
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

library work;
use work.components.all;

library work;
use work.constants.all;

entity cs64 is

  generic (partadder_width : integer := adder_width);

  port (op_a   : in  std_logic_vector(lacell_width-1 downto 0);
        op_b   : in  std_logic_vector(lacell_width-1 downto 0);
        addsub : in  std_logic;         -- '1'=add '0'=sub
        cin_l  : in  std_logic;
        cin_h  : in  std_logic;
        cout_l : out std_logic;
        cout_h : out std_logic;
        sum_l  : out std_logic_vector(31 downto 0);
        sum_h  : out std_logic_vector(31 downto 0));

end cs64;


architecture xilinx_struct of cs64 is
  signal xsum             : std_logic_vector(lacell_width-1 downto 0);
  signal xsum_l, xsum_h   : std_logic_vector(31 downto 0);
  signal op_al, op_ah     : std_logic_vector(31 downto 0);
  signal op_bl, op_bh     : std_logic_vector(31 downto 0);
  signal carry_l, carry_h : std_logic;
  signal VCC, GND         : std_logic;
begin  -- xilinx_struct
  VCC <= '1';
  GND <= '0';

  op_al <= op_a(31 downto  0);
  op_ah <= op_a(lacell_width-1 downto 32);
  op_bl <= op_b(31 downto  0);
  op_bh <= op_b(lacell_width-1 downto 32);


  partwidth_32: if (partadder_width=32) generate
    add_a_b_l : addsub32uu_reg
            port map (  A       => op_al,
```

```
                        B       => op_bl,
                        C_IN    => cin_l,
                        C_OUT   => carry_l,
                        ADD     => addsub,
                        S       => xsum_l);
    add_a_b_h : addsub32uu_reg
            port map (  A       => op_ah,
                        B       => op_bh,
                        C_IN    => cin_h,
                        C_OUT   => carry_h,
                        ADD     => addsub,
                        S       => xsum_h);
  end generate partwidth_32;

  partwidth_64: if (partadder_width=64) generate
    carry_l <= not addsub;
    xsum_l  <= xsum(31 downto  0);
    xsum_h  <= xsum(63 downto 32);
    add_a_b : addsub64uu_reg
            port map (  A       => op_a,
                        B       => op_b,
                        C_IN    => cin_l,
                        C_OUT   => carry_h,
                        ADD     => addsub,
                        S       => xsum);
  end generate partwidth_64;

  cout_h  <= carry_h;
  cout_l  <= carry_l;
  sum_h   <= xsum_h;
  sum_l   <= xsum_l;

-- synopsys translate_off
-- synthesis translate_off
  assert (partadder_width=64 or partadder_width=32)
    report "only partial adders of width 64 or 32 are supported"
    severity FAILURE;
-- synthesis translate_on
-- synopsys translate_on

end xilinx_struct;
```

D.7 Datei: `lacell.vhd`

```
library ieee;
use ieee.std_logic_1164.all;

library work;
use work.constants.all;

library work;
use work.components.all;

entity lacell is

  generic (partadder_width : integer := adder_width );

  port (a           : in  std_logic_vector(lacell_width-1 downto 0);
        context     : in  std_logic_vector(context_depth-1 downto 0);
        clk_store   : in  std_logic;
        addsub      : in  std_logic;      -- '1'=add '0'=sub
        store       : in  std_logic;      -- write selected sum to ram
        ci_low      : in  std_logic;
        carry       : in  std_logic;
        activ_cell  : in  std_logic;
        high_cell   : in  std_logic;
        enable_out  : in  std_logic;
        load_data   : in  std_logic;
        neg_context : in  std_logic;
        all_zero    : out std_logic;
        all_one     : out std_logic;
        co_low      : out std_logic;
        co_high1    : out std_logic;
        co_high2    : out std_logic;
        sum         : out std_logic_vector(lacell_width+1 downto 0));

end lacell;


architecture struct of lacell is
  signal op_2, op_1         : std_logic_vector(lacell_width-1 downto 0);
  signal old_data           : std_logic_vector(lacell_width+1 downto 0);
  signal res                : std_logic_vector(lacell_width+1 downto 0);
  signal hlp                : std_logic_vector(lacell_width-1 downto 0);
  signal c00, c01, c10, c11 : std_logic;
  signal xl, xh, yl, yh     : std_logic_vector(31 downto 0);
```

```
  signal cell_down          : std_logic;  -- cell under the active cells
  signal cell_low           : std_logic;  -- low activ cell
  signal cell_high          : std_logic;  -- high activ cell
  signal cell_up            : std_logic;  -- cell above the active cells
  signal enable_ctxt        : std_logic;
  signal VCC, GND           : std_logic;
  signal not_clk_store      : std_logic;

begin
  VCC     <= '1';
  GND     <= '0';

  loading: process(activ_cell, high_cell, load_data, neg_context, old_data, a)
  variable sel : std_logic_vector(1 downto 0);
  begin
    sel := neg_context&activ_cell;
    cell_down <= not activ_cell and not high_cell;
    cell_low  <=     activ_cell and not high_cell;
    cell_high <=     activ_cell and     high_cell;
    cell_up   <= not activ_cell and     high_cell;
    case sel is
      when "00"   => op_2 <= (others => '0');
      when "01"   => op_2 <= a;
      when others => op_2 <= old_data(63 downto 0);
    end case;
    if (load_data='0') then
      op_1  <= old_data(63 downto 0);
    else
      op_1 <= (others => '0');
    end if;
  end process;


  load_out: process(enable_out, old_data)
  begin
    if (enable_out='1') then
      sum <= old_data;
    else
      sum <= (others => 'Z');
    end if;
  end process;


  add64: csa64
         generic map (partadder_width => partadder_width)
         port map (op_a   => op_1,
                   op_b   => op_2,
                   addsub => addsub,
                   cout00 => c00,
                   cout01 => c01,
                   cout10 => c10,
                   cout11 => c11,
                   out_xl => xl,
                   out_xh => xh,
                   out_yl => yl,
                   out_yh => yh );

  -- low cell produce a carry
  co_low   <= '1' when ((cell_low='1' and addsub='1') and
                        ((c00='1' and c11='1') or (c01='1'))) or
                        ((cell_low='1' and addsub='0') and
                        ((c01='0' and c11='0') or
                         (c00='0' and c01='0' and c10='0' and c11='1')))
              else '0';

  -- if low cell produce a carry than high cell produce a carry too
  -- conditional carry
  co_high1 <= '1' when ((cell_high='1' and addsub='1') and
                        ((c10='1' and c11='1') or (c01='1' and c10='0'))) or
                        ((cell_high='1' and addsub='0') and
                        (
--                        (c00='1' and c01='1' and c10='1' and c11='1') or
                         (c00='0' and c01='0' and c10='1' and c11='1')
                         ))
              else '0';

  -- high cell produce a carry (independent from a low cell carry)
  -- absolute carry
  co_high2 <= '1' when ((cell_high='1' and addsub='1') and
                        ((c00='1' and c11='1') or (c01='1'))) or
                        ((cell_high='1' and addsub='0') and
                        ((c01='0' and c11='0') or
                         (c00='0' and c01='0' and c10='0' and c11='1')))
              else '0';

  hlp(31 downto  0) <= xl when (addsub='1' and (cell_down='1' or
                                  cell_low='1'  or
                                 (cell_high='1' and ci_low='0') or
```

```
                          (cell_up='1'   and carry ='0'))) or
                         (addsub='0' and (
                          (cell_high='1' and ci_low='1') or
                          (cell_up='1'   and carry ='1')))
                      else yl;
hlp(63 downto 32) <= xh when (addsub='1' and (cell_down='1' or
                          (cell_low='1'  and c00='0') or
                          (cell_high='1' and ci_low='0' and c00='0') or
                          (cell_high='1' and ci_low='1' and c10='0') or
                          (cell_up='1'   and carry ='0' and c00='0') or
                          (cell_up='1'   and carry ='1' and c10='0'))) or
                         (addsub='0' and (
                          (cell_low='1' and c10='0') or
                          (cell_high='1' and c00='0' and c01='1' and
                           c10='0' and c11='1') or
                          (cell_high='1' and ci_low='1' and c00='0') or
                          (cell_high='1' and ci_low='0' and c10='0') or
                          (cell_up='1'   and carry ='1' and c00='0' and
                           c01='0' and c10='1' and c11='1')
                          ))
                      else yh;

-- purpose: computing allzero and allone
-- type   : combinational
-- inputs : hlp
-- outputs: res, all_zero, all_one
allzerone: process (hlp, old_data)
  variable all0, all1 : std_logic;
begin  -- process allzerone
  all0 := hlp(63) or hlp(62) or hlp(61) or hlp(60) or
          hlp(59) or hlp(58) or hlp(57) or hlp(56) or hlp(55) or
          hlp(54) or hlp(53) or hlp(52) or hlp(51) or hlp(50) or
          hlp(49) or hlp(48) or hlp(47) or hlp(46) or hlp(45) or
          hlp(44) or hlp(43) or hlp(42) or hlp(41) or hlp(40) or
          hlp(39) or hlp(38) or hlp(37) or hlp(36) or hlp(35) or
          hlp(34) or hlp(33) or hlp(32) or hlp(31) or hlp(30) or
          hlp(29) or hlp(28) or hlp(27) or hlp(26) or hlp(25) or
          hlp(24) or hlp(23) or hlp(22) or hlp(21) or hlp(20) or
          hlp(19) or hlp(18) or hlp(17) or hlp(16) or hlp(15) or
          hlp(14) or hlp(13) or hlp(12) or hlp(11) or hlp(10) or
          hlp( 9) or hlp( 8) or hlp( 7) or hlp( 6) or hlp( 5) or
          hlp( 4) or hlp( 3) or hlp( 2) or hlp( 1) or hlp( 0);

  all1 := hlp(63) and hlp(62) and hlp(61) and hlp(60) and
          hlp(59) and hlp(58) and hlp(57) and hlp(56) and hlp(55) and
          hlp(54) and hlp(53) and hlp(52) and hlp(51) and hlp(50) and
          hlp(49) and hlp(48) and hlp(47) and hlp(46) and hlp(45) and
          hlp(44) and hlp(43) and hlp(42) and hlp(41) and hlp(40) and
          hlp(39) and hlp(38) and hlp(37) and hlp(36) and hlp(35) and
          hlp(34) and hlp(33) and hlp(32) and hlp(31) and hlp(30) and
          hlp(29) and hlp(28) and hlp(27) and hlp(26) and hlp(25) and
          hlp(24) and hlp(23) and hlp(22) and hlp(21) and hlp(20) and
          hlp(19) and hlp(18) and hlp(17) and hlp(16) and hlp(15) and
          hlp(14) and hlp(13) and hlp(12) and hlp(11) and hlp(10) and
          hlp( 9) and hlp( 8) and hlp( 7) and hlp( 6) and hlp( 5) and
          hlp( 4) and hlp( 3) and hlp( 2) and hlp( 1) and hlp( 0);
  res(63 downto 0) <= hlp;
  res(64)          <= not all0;
  res(65)          <= all1;
  all_zero         <= old_data(64);
  all_one          <= old_data(65);
end process allzerone;

enable_ctxt   <= store;
not_clk_store <= not clk_store;

ctxt_ram_block: if (context_b_ram=1) generate
  context_mem: context_bram
    generic map (depth           => context_depth  )
    port map (   address         => context,
                 clk             => not_clk_store,
                 write_en        => enable_ctxt,
                 data_in         => res,
                 data_out        => old_data       );
end generate ctxt_ram_block;

ctxt_ram_distributed: if (context_b_ram/=1) generate
context_mem: context_ram
  generic map (depth           => context_depth  )
  port map (   address         => context,
               clk             => not_clk_store,
               write_en        => enable_ctxt,
               data_in         => res,
               data_out        => old_data       );
end generate ctxt_ram_distributed;
```

```
end struct;

--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
-- carry store version
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

library work;
use work.constants.all;

library work;
use work.components.all;

entity lacell_cs is

  generic (partadder_width : integer := adder_width );

  port (a           : in  std_logic_vector(lacell_width-1 downto 0);
        context     : in  std_logic_vector(context_depth-1 downto 0);
        clk_store   : in  std_logic;
        init_ctxt   : in  std_logic;
        addsub      : in  std_logic;     -- '1'=add '0'=sub
        store       : in  std_logic;     -- write sum to ram
        carry_in    : in  std_logic;
        carry_out   : out std_logic;
        activ_cell  : in  std_logic;
        enable_out  : in  std_logic;
        load_data   : in  std_logic;
        neg_context : in  std_logic;
        all_zero    : out std_logic;
        all_one     : out std_logic;
        sum         : out std_logic_vector(lacell_width+1 downto 0));

end lacell_cs;


architecture struct of lacell_cs is
  signal op_2, op_1    : std_logic_vector(lacell_width-1 downto 0);
  signal old_data      : std_logic_vector(lacell_width+1 downto 0);
  signal res           : std_logic_vector(lacell_width+1 downto 0);
  signal hlp           : std_logic_vector(lacell_width-1 downto 0);
  signal xl, xh        : std_logic_vector(31 downto 0);
  signal wire_cout_l   : std_logic_vector(0 downto 0);
  signal wire_cout_h   : std_logic_vector(0 downto 0);
  signal wire_cin_h    : std_logic_vector(0 downto 0);
  signal wire_bin_h    : std_logic_vector(0 downto 0);
  signal wire_bcin_h   : std_logic_vector(0 downto 0);
  signal carry_out_h   : std_logic_vector(0 downto 0);
  signal borrow_out_h  : std_logic_vector(0 downto 0);
  signal c_in, c_out   : std_logic_vector(1 downto 0);
  signal b_in, b_out   : std_logic_vector(1 downto 0);
  signal enable_carry  : std_logic;
  signal enable_borrow : std_logic;
  signal enable_ctxt   : std_logic;
  signal VCC, GND      : std_logic;
  signal not_clk_store : std_logic;

begin
  VCC     <= '1';
  GND     <= '0';

  loading: process(activ_cell, load_data, neg_context, old_data, a)
  variable sel : std_logic_vector(1 downto 0);
  begin
    sel := neg_context&activ_cell;
      case sel is
        when "00"   => op_2 <= (others => '0');
        when "01"   => op_2 <= a;
        when others => op_2 <= old_data(63 downto 0);
      end case;
      if (load_data='0') then
        op_1  <= old_data(63 downto 0);
      else
        op_1 <= (others => '0');
      end if;
  end process;


  load_out: process(enable_out, old_data)
  begin
    if (enable_out='1') then
      sum <= old_data;
    else
      sum <= (others => 'Z');
```

```
  end if;
end process;


add64: cs64
       generic map (partadder_width => partadder_width)
       port map (op_a   => op_1,
                 op_b   => op_2,
                 addsub => addsub,
                 cout_l => wire_cout_l(0),
                 cout_h => wire_cout_h(0),
                 cin_l  => carry_in,
                 cin_h  => wire_bcin_h(0),
                 sum_l  => xl,
                 sum_h  => xh );

hlp(31 downto  0) <= xl;
hlp(63 downto 32) <= xh;

-- purpose: computing allzero and allone
-- type   : combinational
-- inputs : hlp
-- outputs: res, all_zero, all_one
allzerone: process (hlp)
  variable all0, all1 : std_logic;
begin  -- process allzerone
  all0 := hlp(63) or hlp(62) or hlp(61) or hlp(60) or
          hlp(59) or hlp(58) or hlp(57) or hlp(56) or hlp(55) or
          hlp(54) or hlp(53) or hlp(52) or hlp(51) or hlp(50) or
          hlp(49) or hlp(48) or hlp(47) or hlp(46) or hlp(45) or
          hlp(44) or hlp(43) or hlp(42) or hlp(41) or hlp(40) or
          hlp(39) or hlp(38) or hlp(37) or hlp(36) or hlp(35) or
          hlp(34) or hlp(33) or hlp(32) or hlp(31) or hlp(30) or
          hlp(29) or hlp(28) or hlp(27) or hlp(26) or hlp(25) or
          hlp(24) or hlp(23) or hlp(22) or hlp(21) or hlp(20) or
          hlp(19) or hlp(18) or hlp(17) or hlp(16) or hlp(15) or
          hlp(14) or hlp(13) or hlp(12) or hlp(11) or hlp(10) or
          hlp( 9) or hlp( 8) or hlp( 7) or hlp( 6) or hlp( 5) or
          hlp( 4) or hlp( 3) or hlp( 2) or hlp( 1) or hlp( 0);

  all1 := hlp(63) and hlp(62) and hlp(61) and hlp(60) and
          hlp(59) and hlp(58) and hlp(57) and hlp(56) and hlp(55) and
          hlp(54) and hlp(53) and hlp(52) and hlp(51) and hlp(50) and
          hlp(49) and hlp(48) and hlp(47) and hlp(46) and hlp(45) and
          hlp(44) and hlp(43) and hlp(42) and hlp(41) and hlp(40) and
          hlp(39) and hlp(38) and hlp(37) and hlp(36) and hlp(35) and
          hlp(34) and hlp(33) and hlp(32) and hlp(31) and hlp(30) and
          hlp(29) and hlp(28) and hlp(27) and hlp(26) and hlp(25) and
          hlp(24) and hlp(23) and hlp(22) and hlp(21) and hlp(20) and
          hlp(19) and hlp(18) and hlp(17) and hlp(16) and hlp(15) and
          hlp(14) and hlp(13) and hlp(12) and hlp(11) and hlp(10) and
          hlp( 9) and hlp( 8) and hlp( 7) and hlp( 6) and hlp( 5) and
          hlp( 4) and hlp( 3) and hlp( 2) and hlp( 1) and hlp( 0);
  res(63 downto 0) <= hlp;
  res(64)          <= not all0;
  res(65)          <= all1;
--   all_zero         <= old_data(64);
--   all_one          <= old_data(65);
   all_zero         <= not all0;
   all_one          <= all1;
end process allzerone;

enable_ctxt <= store;

not_clk_store <= not clk_store;

b_in            <= "11" when init_ctxt='1' else wire_cout_h&wire_cout_l;
enable_borrow   <= (not addsub or init_ctxt) and store;
borrow_out_h(0) <= b_out(1);
wire_bin_h(0)   <= b_out(0);

c_in            <= "00" when init_ctxt='1' else wire_cout_h&wire_cout_l;
enable_carry    <= (addsub or init_ctxt) and store;
carry_out_h(0)  <= c_out(1);
wire_cin_h(0)   <= c_out(0);

wire_bcin_h(0)  <= wire_cin_h(0)  when addsub='1' else wire_bin_h(0);
carry_out       <= carry_out_h(0) when addsub='1' else borrow_out_h(0);

ctxt_ram_block: if (context_b_ram=1) generate
  context_mem: context_bram
    generic map (depth           => context_depth  )
    port map (   address         => context,
                 clk             => not_clk_store,
                 write_en        => enable_ctxt,
                 data_in         => res,
                 data_out        => old_data       );
end generate ctxt_ram_block;
```

```
  ctxt_ram_distributed: if (context_b_ram/=1) generate
  context_mem: context_ram
    generic map (depth           => context_depth  )
    port map (   address         => context,
                 clk             => not_clk_store,
                 write_en        => enable_ctxt,
                 data_in         => res,
                 data_out        => old_data       );
  end generate ctxt_ram_distributed;

  context_carry: ramsuu2x16
    port map (   A          => context,
                 CLK        => not_clk_store,
                 WE         => enable_carry,
                 D          => c_in,
                 SPO        => c_out );

  context_borrow: ramsuu2x16borrow
    port map (   A          => context,
                 CLK        => not_clk_store,
                 WE         => enable_borrow,
                 D          => b_in,
                 SPO        => b_out );

end struct;
```

D.8 Datei: Decode.vhd

```
library ieee;
use ieee.std_logic_1164.all;

library work;
use work.components.all;

library work;
use work.constants.all;


entity Decode is
  generic (
    registered  : integer := 1 );
  port (
    clk, rst    : in  std_logic;
    DecI_D0     : in  std_logic_vector (31 downto 0);
    DecI_D1     : in  std_logic_vector (31 downto 0);
    DecI_Cmd    : in  std_logic_vector (31 downto 0);

    DecO_D0     : out std_logic_vector (31 downto 0);
    DecO_D1     : out std_logic_vector (31 downto 0);
    DecO_expo0  : out std_logic_vector (10 downto 0);
    DecO_expo1  : out std_logic_vector (10 downto 0);
    DecO_hidden : out std_logic_vector ( 1 downto 0);
    DecO_ctxt   : out std_logic_vector ( 3 downto 0);
    DecO_dblcnt : out std_logic_vector ( 1 downto 0);
    DecO_cmds   : out std_logic_vector ( 6 downto 0);
    DecO_mode   : out std_logic_vector ( 2 downto 0);
    DecO_cells  : out std_logic_vector ( 3 downto 0);
    DecO_e_cmd  : out std_logic_vector ( 5 downto 0);
    DecO_probes : out std_logic_vector ( 1 downto 0);
    DecO_select : out std_logic_vector ( 1 downto 0);
    DecO_sign   : out std_logic
    );
end Decode;

architecture struct of Decode is

procedure expand_sp(single_prec : in  std_logic_vector(31 downto 0);
                    hidden      : OUT std_logic;
                    exponent    : OUT std_logic_vector(10 downto 0) )
is
  variable hidden_bit : std_logic;
begin
  hidden_bit := (single_prec(30) OR single_prec(29) OR single_prec(28) OR
                 single_prec(27) OR single_prec(26) OR single_prec(25) OR
                 single_prec(24) OR single_prec(23));
  if (hidden_bit='1') then
    exponent := single_prec(30 downto 23) & "000";
  else
    exponent := "00000001000";
  end if;
  hidden := hidden_bit;
end expand_sp;

procedure expand_dp(double_prec : in  std_logic_vector(31 downto 0);
                    hidden      : OUT std_logic;
                    exponent    : OUT std_logic_vector(10 downto 0) )
is
```

```
  variable hidden_bit : std_logic;
begin
  hidden_bit := (double_prec(30) OR double_prec(29) OR double_prec(28) OR
                 double_prec(27) OR double_prec(26) OR double_prec(25) OR
                 double_prec(24) OR double_prec(23) OR double_prec(22) OR
                 double_prec(21) OR double_prec(20) );
  if (hidden_bit='1') then
    exponent := double_prec(30 downto 20);
  else
    exponent := "00000000001";
  end if;
  hidden := hidden_bit;
end expand_dp;



signal old_sign         : std_logic;
signal enable           : std_logic;
signal hid0_sp, hid1_sp : std_logic;
signal hid0_dp, hid1_dp : std_logic;
signal exp0_sp, exp1_sp : std_logic_vector(10 downto 0);
signal exp0_dp, exp1_dp : std_logic_vector(10 downto 0);

signal Dec0_D0_cmb      : std_logic_vector(31 downto 0);
signal Dec0_D1_cmb      : std_logic_vector(31 downto 0);
signal Dec0_ctxt_cmb    : std_logic_vector( 3 downto 0);
signal old_ctxt         : std_logic_vector( 3 downto 0);
signal Dec0_dblcnt_cmb  : std_logic_vector( 1 downto 0);
signal Dec0_cmds_cmb    : std_logic_vector( 6 downto 0);
signal Dec0_probes_cmb  : std_logic_vector( 1 downto 0);
signal Dec0_select_cmb  : std_logic_vector( 1 downto 0);
signal Dec0_mode_cmb    : std_logic_vector( 2 downto 0);
signal Dec0_cells_cmb   : std_logic_vector( 3 downto 0);
signal Dec0_expo0_cmb   : std_logic_vector(10 downto 0);
signal Dec0_expo1_cmb   : std_logic_vector(10 downto 0);
signal Dec0_hidden_cmb  : std_logic_vector( 1 downto 0);
signal Dec0_e_cmd_cmb   : std_logic_vector( 5 downto 0);

signal Dec0_sign_cmb_vector : std_logic_vector( 0 downto 0);
signal Dec0_sign_vector     : std_logic_vector( 0 downto 0);

signal select_mode          : std_logic_vector( 2 downto 0);

begin         -- struct of Decode
```

```
expand_expos: process (DecI_D0, DecI_D1) is
  variable v_exp0_sp, v_exp1_sp : std_logic_vector(10 downto 0);
  variable v_exp0_dp, v_exp1_dp : std_logic_vector(10 downto 0);
  variable v_hid0_sp, v_hid1_sp : std_logic;
  variable v_hid0_dp, v_hid1_dp : std_logic;
begin
    expand_sp(DecI_D0,v_hid0_sp,v_exp0_sp);
    expand_sp(DecI_D1,v_hid1_sp,v_exp1_sp);
    expand_dp(DecI_D0,v_hid0_dp,v_exp0_dp);
    expand_dp(DecI_D1,v_hid1_dp,v_exp1_dp);
    exp0_sp <= v_exp0_sp;
    exp1_sp <= v_exp1_sp;
    exp0_dp <= v_exp0_dp;
    exp1_dp <= v_exp1_dp;

    hid0_sp <= v_hid0_sp;
    hid1_sp <= v_hid1_sp;
    hid0_dp <= v_hid0_dp;
    hid1_dp <= v_hid1_dp;
end process;

sign_store: process (DecI_D0, DecI_D1, DecI_Cmd, clk, rst)
begin  -- process sign_store
  if (rst='0') then
    old_sign <= '1';
  elsif (clk'event and clk='1') then
    if ((DecI_Cmd(cmd_ok_flag4)='1') and (DecI_Cmd(cmd_ok_flag1)='1')) then
      if ((DecI_Cmd(cmd_dbl_cnt2 downto cmd_dbl_cnt1) = "00") and
          not (select_mode="011") and (DecI_Cmd(cmd_init_ctxt)='0') and
          (DecI_Cmd(cmd_cb_res)='0') and
          not ((DecI_Cmd(cmd_mode_dbl)='1') and
               (DecI_Cmd(cmd_mode_add)='1')) and
          DecI_Cmd(cmd_double_res)='0')
      then
        if (DecI_Cmd(cmd_substract)='0') then
          old_sign <= not((DecI_D0(31) xor DecI_D1(31)) or
                      DecI_cmd(cmd_neg_ctxt));
        else
          old_sign <= ((DecI_D0(31) xor DecI_D1(31)) or
                       DecI_cmd(cmd_neg_ctxt));
        end if;
      elsif ((DecI_Cmd(cmd_mode_dbl)='1') and (DecI_Cmd(cmd_mode_add)='1'))
      then
```

```
        if (DecI_Cmd(cmd_substract)='0') then
          old_sign <= not DecI_D0(31);
        else
          old_sign <= DecI_D0(31);
        end if;
      else
        if (DecI_Cmd(cmd_substract)='0') then
          old_sign <= '1';
        else
          old_sign <= '0';
        end if;
      end if;
    end if;
  end if;
end process sign_store;

ctxt_store: process(DecI_Cmd, clk, rst)
begin
  if (rst='0') then
    old_ctxt <= (others => '0');
  elsif (clk'event and clk='0') then
    if (DecI_Cmd(cmd_ok_flag3)='1' and DecI_Cmd(cmd_ok_flag2)='1') then
       old_ctxt <= DecI_Cmd(cmd_ctxt_bit4  downto cmd_ctxt_bit1);
    end if;
  end if;
end process ctxt_store;

ctxt_direct: if (context_store/=1) generate
  DecO_ctxt_cmb   <= DecI_Cmd(cmd_ctxt_bit4  downto cmd_ctxt_bit1);
end generate ctxt_direct;
ctxt_stored: if (context_store =1) generate
  DecO_ctxt_cmb   <= old_ctxt;
end generate ctxt_stored;


DecO_sign               <= old_sign;

DecO_expo0_cmb          <= exp0_dp when DecI_Cmd(cmd_mode_dbl)='1' else
                           exp0_sp;
DecO_expo1_cmb          <= exp1_dp when DecI_Cmd(cmd_mode_dbl)='1' else
                           exp1_sp;
DecO_hidden_cmb(0)      <= hid0_dp when DecI_Cmd(cmd_mode_dbl)='1' else
                           hid0_sp;
DecO_hidden_cmb(1)      <= hid1_dp when DecI_Cmd(cmd_mode_dbl)='1' else
                           hid1_sp;

DecO_e_cmd_cmb(5 downto 4) <= "11" when (DecI_Cmd(cmd_cell_bit01)='0' and
                                         select_mode="011") else
                              "01" when  DecI_Cmd(cmd_double_res)='1' else
                              "00";
DecO_e_cmd_cmb(3 downto 2) <= "01" when (select_mode="011" or
                                         DecI_Cmd(cmd_in_out)='1' or
                                         DecI_Cmd(cmd_double_res)='1') else
                              "10" when DecI_Cmd(cmd_neg_ctxt)='1' else
                              "11" when DecI_cmd(cmd_init_ctxt)='1' or
                                        DecI_cmd(cmd_cb_res)='1' else
                              "00";
DecO_e_cmd_cmb(1 downto 0) <= "00" when (DecI_cmd(cmd_init_ctxt)='1' or
                                         DecI_Cmd(cmd_neg_ctxt)='1') else
                              "01" when (select_mode="011") else
                              "10" when (DecI_Cmd(cmd_add_tpt)='1'  and
                                         DecI_Cmd(cmd_in_out)='0') else
                              "11";

DecO_cells_cmb  <= DecI_Cmd(cmd_cell_bit04 downto cmd_cell_bit01);

DecO_D0_cmb     <= DecI_D0;
DecO_D1_cmb     <= DecI_D1;
DecO_dblcnt_cmb <= DecI_Cmd(cmd_dbl_cnt2   downto cmd_dbl_cnt1);
DecO_mode_cmb   <= DecI_Cmd(cmd_mode_tpt)  & DecI_Cmd(cmd_mode_add)   &
                   DecI_Cmd(cmd_mode_dbl);
DecO_cmds_cmb   <= DecI_Cmd(cmd_cb_res)    & DecI_Cmd(cmd_cond_neg)   &
                   DecI_Cmd(cmd_add_tpt) &
                   DecI_Cmd(cmd_in_out)    & DecI_Cmd(cmd_init_ctxt)  &
                   DecI_Cmd(cmd_double_res) &
                   DecI_Cmd(cmd_neg_ctxt);
DecO_probes_cmb <= DecI_Cmd(probe_flag1) & DecI_Cmd(probe_flag2);
select_mode     <= DecI_Cmd(cmd_mode_tpt) & DecI_Cmd(cmd_mode_add)    &
                   DecI_Cmd(cmd_mode_dbl);
                   -- accumulate product
DecO_select_cmb <= "10" when (select_mode="100" or select_mode="101") else
                   -- add operand
                   "11" when (select_mode="110" or select_mode="111") else
                   -- transparent
                   "01" when  select_mode="011"                       else
                   "00";

output_register: if (registered=1) generate
```

```
enable <= '1';
reg00: pipe generic map( width => 32)
          port map    ( dpipe => Dec0_D0_cmb,
                        clk   => clk, rst => rst, enable => enable,
                        qpipe => Dec0_D0);
reg01: pipe generic map( width => 32)
          port map    ( dpipe => Dec0_D1_cmb,
                        clk   => clk, rst => rst, enable => enable,
                        qpipe => Dec0_D1);
reg02: pipe generic map( width => 11)
          port map    ( dpipe => Dec0_expo0_cmb,
                        clk   => clk, rst => rst, enable => enable,
                        qpipe => Dec0_expo0);
reg03: pipe generic map( width => 11)
          port map    ( dpipe => Dec0_expo1_cmb,
                        clk   => clk, rst => rst, enable => enable,
                        qpipe => Dec0_expo1);
reg04: pipe generic map( width =>  2)
          port map    ( dpipe => Dec0_hidden_cmb,
                        clk   => clk, rst => rst, enable => enable,
                        qpipe => Dec0_hidden);
reg05: pipe generic map( width =>  4)
          port map    ( dpipe => Dec0_ctxt_cmb,
                        clk   => clk, rst => rst, enable => enable,
                        qpipe => Dec0_ctxt);
reg06: pipe generic map( width =>  2)
          port map    ( dpipe => Dec0_dblcnt_cmb,
                        clk   => clk, rst => rst, enable => enable,
                        qpipe => Dec0_dblcnt);
reg07: pipe generic map( width =>  7)
          port map    ( dpipe => Dec0_cmds_cmb,
                        clk   => clk, rst => rst, enable => enable,
                        qpipe => Dec0_cmds);
reg08: pipe generic map( width =>  3)
          port map    ( dpipe => Dec0_mode_cmb,
                        clk   => clk, rst => rst, enable => enable,
                        qpipe => Dec0_mode);
reg09: pipe generic map( width => 4)
          port map    ( dpipe => Dec0_cells_cmb,
                        clk   => clk, rst => rst, enable => enable,
                        qpipe => Dec0_cells);
reg10: pipe generic map( width => 6)
          port map    ( dpipe => Dec0_e_cmd_cmb,
                        clk   => clk, rst => rst, enable => enable,
                        qpipe => Dec0_e_cmd);
    reg11: pipe generic map( width => 2)
              port map    ( dpipe => Dec0_select_cmb,
                            clk   => clk, rst => rst, enable => enable,
                            qpipe => Dec0_select);
    reg12: pipe generic map( width => 2)
              port map    ( dpipe => Dec0_probes_cmb,
                            clk   => clk, rst => rst, enable => enable,
                            qpipe => Dec0_probes);

    end generate output_register;
    output_combinatorial: if (registered/=1) generate
      Dec0_D0     <= Dec0_D0_cmb;
      Dec0_D1     <= Dec0_D1_cmb;
      Dec0_expo0  <= Dec0_expo0_cmb;
      Dec0_expo1  <= Dec0_expo1_cmb;
      Dec0_hidden <= Dec0_hidden_cmb;
      Dec0_ctxt   <= Dec0_ctxt_cmb;
      Dec0_dblcnt <= Dec0_dblcnt_cmb;
      Dec0_cmds   <= Dec0_cmds_cmb;
      Dec0_probes <= Dec0_probes_cmb;
      Dec0_e_cmd  <= Dec0_e_cmd_cmb;
      Dec0_select <= Dec0_select_cmb;
      Dec0_mode   <= Dec0_mode_cmb;
      Dec0_cells  <= Dec0_cells_cmb;
    end generate output_combinatorial;

end struct;   -- Decode
```

D.9 Datei: data_select.vhd

```
library ieee;
use ieee.std_logic_1164.all;

library ieee;
use ieee.std_logic_unsigned."+";

library work;
use work.components.all;

library work;
use work.constants.all;

entity data_select is
```

```
  generic ( registered           : integer := 1 );
  port    ( clk, rst             : in  std_logic;
            data0_in,  data1_in  : in  std_logic_vector(31 downto 0);
            hidden_bit           : in  std_logic_vector( 1 downto 0);
            data_mode            : in  std_logic_vector( 2 downto 0);
            double_counter       : in  std_logic_vector( 1 downto 0);
            data0_out, data1_out : out std_logic_vector(31 downto 0) );
end data_select;

--------------------------------------------------------------------------------
--
--                                  +hidden bit | 52 bit mantissa      |
-- DOUBLE precision number:   63 62 61 60 59 .. 52 51 50 .. 32 31.. 2 1 0
--                          sign| 11 bits exponent| 21 bit msw|32 bit lsw|
--
-- double products are produced with 4 partial products in the following order:
--    double_counter
-- 1.  00             msw1 x msw2 = 42 bit partial product    64 bit left shift
-- 2.  01             lsw1 x lsw2 = 64 bit partial product    no shift
-- 3.  10             lsw1 x msw2 = 53 bit partial product    32 bit left shift
-- 4.  11             msw1 x lsw2 = 53 bit partial product    32 bit left shift
--
-- data_mode: 100 single precision SuMa
--            101 double precision SuMa
--            110 single add  (no second operand)
--            111 double add  (lsw and msw added in one clock cycle)
--            011 transparent mode data_out == data_in
--
-- double_counter: for the 4 partial products (double input) we need different
--                 shift_amounts. this signal shows in which stage we are
--
--------------------------------------------------------------------------------

architecture struct of data_select is
  signal aligned_data0, aligned_data1   : std_logic_vector(31 downto 0);
  signal data0_reg, data1_reg           : std_logic_vector(31 downto 0);
  signal data0_cmb, data1_cmb           : std_logic_vector(31 downto 0);
  signal data0_msw_reg, data1_msw_reg   : std_logic_vector(31 downto 0);
  signal data0_lsw_reg, data1_lsw_reg   : std_logic_vector(31 downto 0);
  signal choose_data                    : std_logic_vector( 2 downto 0);
  signal msw_enable, lsw_enable, VCC    : std_logic;

begin
  VCC            <= '1';

choose_data    <= "0" & double_counter when data_mode(1 downto 0)="01"
                  else mode_tpt        when data_mode="011"
                  else mode_dad        when data_mode="111"
                  else mode_sad        when data_mode="110"
                  else mode_spn;
aligned_data0  <= "00000000000" & hidden_bit(0) & data0_in(19 downto 0) when
                     data_mode(0)='1'
                  else "00000000" & hidden_bit(0) & data0_in(22 downto 0);
aligned_data1  <= "00000000000" & hidden_bit(1) & data1_in(19 downto 0) when
                     data_mode(0)='1'
                  else "00000000" & hidden_bit(1) & data1_in(22 downto 0);

msw_enable     <= '1' when choose_data=mode_mxm else '0';
lsw_enable     <= '1' when choose_data=mode_lxl else '0';

data0_msw: nreg generic map ( width  => 32 )
                port map    ( d      => aligned_data0,
                              clk    => clk, rst => rst,
                              enable => msw_enable,
                              q      => data0_msw_reg);
data1_msw: nreg generic map ( width  => 32 )
                port map    ( d      => aligned_data1,
                              clk    => clk, rst => rst,
                              enable => msw_enable,
                              q      => data1_msw_reg);
data0_lsw: nreg generic map ( width  => 32 )
                port map    ( d      => data0_in,
                              clk    => clk, rst => rst,
                              enable => lsw_enable,
                              q      => data0_lsw_reg);
data1_lsw: nreg generic map ( width  => 32 )
                port map    ( d      => data1_in,
                              clk    => clk, rst => rst,
                              enable => lsw_enable,
                              q      => data1_lsw_reg);

data0_cmb <=      data1_msw_reg when choose_data=mode_mxl
             else data1_lsw_reg when choose_data=mode_lxm
             else data0_in      when choose_data=mode_tpt or
                                     choose_data=mode_lxl
             else aligned_data0; -- when choose_data=mode_mxm or
                                 --      choose_data=mode_spn
data1_cmb <=      data0_msw_reg   when choose_data=mode_lxm
             else data0_lsw_reg   when choose_data=mode_mxl
```

```
            else data1_in        when choose_data=mode_tpt or
                                      choose_data=mode_dad
                                   or choose_data=mode_lxl
            else (others => '0') when choose_data=mode_sad
            else aligned_data1; -- when choose_data=mode_mxm or
                                --      choose_data=mode_spn
  output_register: if (registered=1) generate
    reg0: pipe generic map ( width  => 32)
            port map    ( dpipe  => data0_cmb,
                          clk    => clk, rst => rst,
                          enable => VCC,
                          qpipe  => data0_out);
    reg1: pipe generic map ( width  => 32 )
            port map    ( dpipe  => data1_cmb,
                          clk    => clk, rst => rst,
                          enable => VCC,
                          qpipe  => data1_out);
    end generate output_register;
    output_combinatorial: if (registered/=1) generate
      data0_out <= data0_cmb;
      data1_out <= data1_cmb;
    end generate output_combinatorial;
end struct;
```

D.10 Datei: expo_arith.vhd

```
library ieee;
use ieee.std_logic_1164.all;

library ieee;
use ieee.std_logic_unsigned."+";

library work;
use work.components.all;

library work;
use work.constants.all;

entity expari is
  generic ( registered     : integer := 1 );
  port (    clk, rst       : in  std_logic;
            mode_lacells   : in  std_logic_vector( 5 downto 0);
            ea, eb         : in  std_logic_vector(10 downto 0);
            data_mode      : in  std_logic_vector( 2 downto 0);
            double_counter : in  std_logic_vector( 1 downto 0);
            cells_to_read  : in  std_logic_vector( 3 downto 0);
            sub_context    : out std_logic_vector( 3 downto 0);
            shift_amount   : out std_logic_vector( 6 downto 0);
            activ_lacells  : out std_logic_vector( 9 downto 0);
            tpt_lacells    : out std_logic_vector( 9 downto 0);
            high_lacells   : out std_logic_vector( 9 downto 0)  );
end expari ;

--------------------------------------------------------------------------------
-- ea, ab: bit 31 downto 20 from the input data words. contains the exponent
--         in single and double case.
--
--                                    +hidden bit | 52 bit mantissa      |
-- DOUBLE precision number:   63 62 61 60 59 .. 52 51 50 .. 32 31.. 2 1 0
--                          sign| 11 bits exponent| 21 bit msw|32 bit lsw|
--
-- double products are produced with 4 partial products in the following order:
-- 1. msw1 x msw2 = 42 bit partial product    64 bit left shift
-- 2. lsw1 x lsw2 = 64 bit partial product    no shift
-- 3. lsw1 x msw2 = 53 bit partial product    32 bit left shift
-- 4. msw1 x lsw2 = 53 bit partial product    32 bit left shift
--
-- data_mode: 100 single precision SuMa
--            101 double precision SuMa
--            110 single add  (no second operand)
--            111 double add  (lsw and msw added in one clock cycle)
--            011 transparent mode data_out == data_in
--
-- double_counter: for the 4 partial products (double input) we need different
--                 shift_amounts. this signal shows in which stage we are
--
-- activ_lacells:  the 2 lacells which are inferred in the addition are '1'
-- high_lacells:   all lacells which are above the low activ_lacells are '1',
--                 because they can catch a carry.
--------------------------------------------------------------------------------

architecture struct of expari is
  signal erg, exp_reg_a, exp_reg_b           : std_logic_vector(12 downto 0);
  signal aligned_expo_a, aligned_expo_b      : std_logic_vector(12 downto 0);
  signal expo_a, expo_b                      : std_logic_vector(12 downto 0);
  signal choose_exponent                     : std_logic_vector( 2 downto 0);
  signal VCC, enable_exponent                : std_logic;
  signal sub_context_cmb                     : std_logic_vector( 3 downto 0);
```

```
  signal shift_amount_cmb                   : std_logic_vector( 6 downto 0);
  signal activ_lacells_cmb, high_lacells_cmb : std_logic_vector( 9 downto 0);
  signal activs_to_read, highs_to_read      : std_logic_vector( 9 downto 0);
  signal activ_lacells_out                  : std_logic_vector( 9 downto 0);
  signal high_lacells_out                   : std_logic_vector( 9 downto 0);
  signal expo_tpt_cmb                       : std_logic_vector( 9 downto 0);

  function exponent(ea, eb, choose  : std_logic_vector)
           return std_logic_vector is
    variable e0, e1, e2, e3, e4, e5, e6, e7, e8 :
                                            std_logic_vector(12 downto 0);
  begin
    e0 := "0000000000000";
    e1 := ea;
    e2 := eb;
    e3 := e1 + e2;
    e4 := e3 + mxm_bias;
    e5 := e3 + mxl_bias;
    e6 := e3 + lxl_bias;
    e7 := e1 + sad_bias;
    e8 := e1 + dad_bias;

    if    (choose=mode_spn) then return e3;  -- single product
    elsif (choose=mode_mxm) then return e4;  -- msw1 x msw2
    elsif (choose=mode_mxl or
           choose=mode_lxm) then return e5;  -- lsw1 x msw2, msw1 x lsw2
    elsif (choose=mode_lxl) then return e6;  -- lsw1 x lsw2
    elsif (choose=mode_sad) then return e7;  -- single add
    elsif (choose=mode_dad) then return e8;  -- double add
    else  return e0;                         -- transparent mode needs no shift
    end if;
  end;

begin
  VCC             <= '1';
  choose_exponent <= "0" & double_counter when data_mode="101"
                     else mode_tpt        when data_mode="011"
                     else mode_dad        when data_mode="111"
                     else mode_sad        when data_mode="110"
                     else mode_spn;
  aligned_expo_a  <= "00" & ea when data_mode(0)='1' else "00000" &
                             ea(10 downto 3);
  aligned_expo_b  <= "00" & eb when data_mode(0)='1' else "00000" &
                             eb(10 downto 3);

enable_exponent <= '1' when choose_exponent=mode_mxm else '0';

exp_a: nreg generic map ( width  => 13 )
            port map    ( d      => aligned_expo_a,
                          clk    => clk, rst => rst,
                          enable => enable_exponent,
                          q      => exp_reg_a);
exp_b: nreg generic map ( width  => 13 )
            port map    ( d      => aligned_expo_b,
                          clk    => clk, rst => rst,
                          enable => enable_exponent,
                          q      => exp_reg_b);

expo_a          <= exp_reg_a
                   when (choose_exponent=mode_lxl or
                         choose_exponent=mode_lxm or
                         choose_exponent=mode_mxl )
                   else aligned_expo_a;
expo_b          <= exp_reg_b
                   when (choose_exponent=mode_lxl or
                         choose_exponent=mode_lxm or
                         choose_exponent=mode_mxl )
                   else aligned_expo_b;
erg             <= "0000001000000" when (mode_lacells(5 downto 4)="11") else
                   exponent(expo_a, expo_b, choose_exponent);

sub_context_cmb   <= erg(12 downto 9);
shift_amount_cmb  <= erg( 6 downto 0);

activs_to_read <= "1100000000" when cells_to_read(3 downto 0)="1010" else
                  "0110000000" when cells_to_read(3 downto 0)="1001" else
                  "0011000000" when cells_to_read(3 downto 0)="1000" else
                  "0001100000" when cells_to_read(3 downto 0)="0111" else
                  "0000110000" when cells_to_read(3 downto 0)="0110" else
                  "0000011000" when cells_to_read(3 downto 0)="0101" else
                  "0000001100" when cells_to_read(3 downto 0)="0100" else
                  "0000000110" when cells_to_read(3 downto 0)="0011" else
                  "0000000011" when cells_to_read(3 downto 0)="0010" else
                  "0000000011" when cells_to_read(3 downto 0)="0001" else
                  "0000000000";
highs_to_read  <= "1000000000" when cells_to_read(3 downto 0)="1010" else
                  "1100000000" when cells_to_read(3 downto 0)="1001" else
                  "1110000000" when cells_to_read(3 downto 0)="1000" else
```

```
                "1111000000" when cells_to_read(3 downto 0)="0111" else
                "1111100000" when cells_to_read(3 downto 0)="0110" else
                "1111110000" when cells_to_read(3 downto 0)="0101" else
                "1111111000" when cells_to_read(3 downto 0)="0100" else
                "1111111100" when cells_to_read(3 downto 0)="0011" else
                "1111111110" when cells_to_read(3 downto 0)="0010" else
                "1111111110" when cells_to_read(3 downto 0)="0001" else
                "0000000000";

activ_lacells_cmb(9) <= '0';
activ_lacells_cmb(8) <= '1' when  erg(8)='1' and erg(7)='1' and
                                              erg(6)='1'  else '0';
activ_lacells_cmb(7) <= '1' when  erg(8)='1' and erg(7)='1'  else '0';
activ_lacells_cmb(6) <= '1' when (erg(8)='1' and erg(7)='0' and
                                              erg(6)='1') or
                                 (erg(8)='1' and erg(7)='1' and
                                              erg(6)='0') else '0';
activ_lacells_cmb(5) <= '1' when  erg(8)='1' and erg(7)='0'  else '0';
activ_lacells_cmb(4) <= '1' when (erg(8)='0' and erg(7)='1' and
                                              erg(6)='1') or
                                 (erg(8)='1' and erg(7)='0' and
                                              erg(6)='0')  else '0';
activ_lacells_cmb(3) <= '1' when  erg(8)='0' and erg(7)='1'   else '0';
activ_lacells_cmb(2) <= '1' when (erg(8)='0' and erg(7)='0' and
                                              erg(6)='1') or
                                 (erg(8)='0' and erg(7)='1' and
                                              erg(6)='0')  else '0';
activ_lacells_cmb(1) <= '1' when  erg(8)='0' and erg(7)='0'   else '0';
activ_lacells_cmb(0) <= '1' when  erg(8)='0' and erg(7)='0' and
                                              erg(6)='0'   else '0';

activ_lacells_out <= activ_lacells_cmb when mode_lacells(3 downto 2)="00"
                     else
                     activs_to_read    when mode_lacells(3 downto 2)="01"
                     else
                     (others => '1')   when mode_lacells(3 downto 2)="10"
                     else
                     (others => '0');
expo_tpt_cmb      <= (others => '1')   when (mode_lacells(1 downto 0)="00")
                     else
                     activ_lacells_cmb when (mode_lacells(3 downto 0)="0010")
                     else
                     activs_to_read   when ((mode_lacells(3 downto 0)="0110")
                                            or
                                                mode_lacells(1 downto 0)="01")
                     else
                     (others => '0');

high_lacells_out  <= activs_to_read    when mode_lacells(1 downto 0)="01"
                     else
                     activs_to_read    when mode_lacells(5 downto 4)="01"
                     else
                     high_lacells_cmb;

high_lacells_cmb(9)  <= '1';
high_lacells_cmb(8)  <= '1';
high_lacells_cmb(7)  <= '0' when  erg(8)='1' and erg(7)='1' and erg(6)='1'
                        else '1';
high_lacells_cmb(6)  <= '1' when (erg(8)='0') or
                                 (erg(8)='1' and erg(7)='0')
                        else '0';
high_lacells_cmb(5)  <= '1' when (erg(8)='1' and erg(7)='0' and erg(6)='0')
                                 or (erg(8)='0')
                        else '0';
high_lacells_cmb(4)  <= '1' when  erg(8)='0'
                        else '0';
high_lacells_cmb(3)  <= '1' when (erg(8)='0' and erg(7)='0') or
                                 (erg(8)='0' and erg(7)='1' and erg(6)='0')
                        else '0';
high_lacells_cmb(2)  <= '1' when  erg(8)='0' and erg(7)='0'
                        else '0';
high_lacells_cmb(1)  <= '1' when  erg(8)='0' and erg(7)='0' and erg(6)='0'
                        else '0';
high_lacells_cmb(0)  <= '0';

output_register: if (registered=1) generate
  reg0: pipe generic map ( width  => 4)
             port map    ( dpipe  => sub_context_cmb,
                           clk    => clk, rst => rst,
                           enable => VCC,
                           qpipe  => sub_context);
  reg1: pipe generic map ( width  => 7)
             port map    ( dpipe  => shift_amount_cmb,
                           clk    => clk, rst => rst,
                           enable => VCC,
                           qpipe  => shift_amount);
  reg2: pipe generic map ( width  => 10)
             port map    ( dpipe  => activ_lacells_out,
```

```
                          clk    => clk, rst => rst,
                          enable => VCC,
                          qpipe  => activ_lacells);
    reg3: pipe generic map ( width  => 10)
               port map    ( dpipe  => high_lacells_out,
                          clk    => clk, rst => rst,
                          enable => VCC,
                          qpipe  => high_lacells);
    reg4: pipe generic map ( width  => 10)
               port map    ( dpipe  => expo_tpt_cmb,
                          clk    => clk, rst => rst,
                          enable => VCC,
                          qpipe  => tpt_lacells);

  end generate output_register;
  output_combinatorial: if (registered/=1) generate
    sub_context   <= sub_context_cmb;
    shift_amount  <= shift_amount_cmb;
    activ_lacells <= activ_lacells_out;
    high_lacells  <= high_lacells_out;
    tpt_lacells   <= expo_tpt_cmb;
  end generate output_combinatorial;
end struct;
```

D.11 Datei: SuMa_core.vhd

```
library ieee;
use ieee.std_logic_1164.all;

library work;
use work.constants.all;

library work;
use work.components.all;

-- library work;
-- use work.cells.all;


entity SuMa is
  generic (partadder_width : integer := adder_width );

  port (   data_in       : in  std_logic_vector(127 downto 0);
           sign          : in  std_logic;
           activ_lacells : in  std_logic_vector(num_cell-1 downto 0);
           high_lacells  : in  std_logic_vector(num_cell-1 downto 0);
           tpt_lacells   : in  std_logic_vector(num_cell-1 downto 0);
           clk           : in  std_logic;
           clk_store     : in  std_logic;
           rst           : in  std_logic;
           context       : in  std_logic_vector(  3 downto 0);
           cb_resolve    : in  std_logic;  -- only for carry store architecture
                                           -- to resolve carries and borrows
           neg_context   : in  std_logic;  -- '1' neg/twos complement of
                                           --     context
           cond_neg      : in  std_logic;  -- '1' negate ctxt only if negative
           add_tpt       : in  std_logic;  -- '0' means add, '1' means
                                           --     transparent
           in_out        : in  std_logic;  -- '0' means in, '1' means out
           double_res    : in  std_logic;  -- '1' means
                                           -- if in_out='1' then
                                           --    load lacell(activ_lacells) to
                                           --    resolve register
                                           -- else
                                           --    add resolve register
           init_ctxt     : in  std_logic;  -- set actual context all zero
           -- maybe only 128 bits are necessary
           -- data_out(131)='1' if data_out(129 downto 66) all are '1'
           -- data_out(130)='1' if data_out(129 downto 66) all are '0'
           -- data_out( 65)='1' if data_out( 63 downto  0) all are '1'
           -- data_out( 64)='1' if data_out( 63 downto  0) all are '0'
           data_out      : out std_logic_vector(131 downto 0);
           -- context_out(15)   reserve
           -- context_out(14)   reserve
           -- context_out(13)   reserve
           -- context_out(12) ='1' means overflow
           -- context_out(11) sign of the context; ='1' negative, ='0' positive
           -- context_out(10) stickybit: if any lacell under the msw has a '1'
           --                            stickybit='1' else stickybit='0'
           -- context_out(num_cell-1 downto 0) the bits of the two msw are '1'
           -- others '0'
           --                          if the result is in the last lacell the
           --                          first msw are all zero
           context_out   : out std_logic_vector( 15 downto 0);
           debug_out     : out std_logic_vector( 15 downto 0)   );
end SuMa;

architecture struct of SuMa is
```

```
signal cout1, cout2, clow, carry : std_logic_vector(num_cell-1 downto 0);
signal write_enable              : std_logic_vector(num_cell downto 0);
signal transparent               : std_logic_vector(num_cell-1 downto 0);
signal all_zeros, all_ones       : std_logic_vector(num_cell-1 downto 0);
signal wire                      : std_logic_vector(num_cell   downto 0);
signal enable_outs               : std_logic_vector(num_cell-1 downto 0);
signal ctxt_data_i, ctxt_data_o  : std_logic_vector( 15 downto 0);
signal msw_context               : std_logic_vector( 10 downto 0);
signal msw_lacells               : std_logic_vector(num_cell-1 downto 0);
signal data_in_low, data_in_high : std_logic_vector( 63 downto 0);
signal data_out0, data_out1      : std_logic_vector( 65 downto 0);
signal data_sign                 : std_logic;
signal VCC, GND                  : std_logic;
signal ctxt_overflow, ctxt_sign  : std_logic;
signal reset_write               : std_logic;
signal oldsign, oldover          : std_logic;
signal not_clk_store             : std_logic;
signal c_sign, c_overflow        : std_logic;
signal sel                       : std_logic_vector(4 downto 0);
begin

  VCC <= '1';
  GND <= '0';

wire0_cs: if (carry_store=1) generate
  wire(0) <= not data_sign;
end generate wire0_cs;

wire0_csa: if (carry_store/=1) generate
  wire(0) <= '0';
end generate wire0_csa;

  ------------------------------------------------------------------------------
  -- add_tpt | in_out | double_res | init_ctxt | command
  ------------------------------------------------------------------------------
  --    0    |   0    |     0      |     0     | normal add
  --    0    |   1    |     0      |     0     | normal read suma needs
  --         |        |            |           |    msw_lacells
  --    1    |   0    |     0      |     0     | set the cell declared in
  --         |        |            |           |    activ_lacells
  --    1    |   1    |     0      |     0     | read the cell declared in
  --         |        |            |           |    activ_lacells
  --    x    |   1    |     1      |     0     | load the 2 highest lacells of
  --         |        |            |           | context into resolve register
  --    x    |   0    |     1      |     0     | add resolve register in the 2
  --         |        |            |           | lowest lacells of context
  --    x    |   x    |     x      |     1     | set the whole context to zero
  ------------------------------------------------------------------------------

data_sign    <= sign or (cond_neg and not oldsign);
data_in_low  <= data_in( 63 downto  0);
data_in_high <= data_in(127 downto 64);
transparent  <= tpt_lacells;
enable_outs  <= activ_lacells when (in_out='1' and (double_res='1' or
                                                    add_tpt='1')) else
                ctxt_data_o(9 downto 0) when add_tpt='0' and in_out='1' else
                (others => '0');


writing: process (in_out, init_ctxt, add_tpt, cb_resolve,
                  neg_context, activ_lacells, high_lacells)
  variable write_en_mask : std_logic_vector(num_cell downto 0);
begin  -- process writing
  if (init_ctxt='1') or (neg_context='1') or (cb_resolve='1')then
    write_en_mask := (others => '1');
  elsif (in_out = '0' and add_tpt = '0') then
    write_en_mask := "1"&(activ_lacells or high_lacells);
  else
    write_en_mask := (others => '0');
  end if;
    write_enable <= write_en_mask;
end process writing;

carry_res_csa: if (carry_store/=1) generate
carry_res: process (clk,
                    cout1, cout2, clow, all_ones, all_zeros,
                    sign, neg_context, high_lacells, activ_lacells)
  variable carry_resolve, carry0, carry1,
           carry2, carry3, carry4, carry5,
           carry6, carry7, carry8, carry9 : std_logic;
  variable all_hlp                        : std_logic_vector(9 downto 0);
begin  -- process carry_res
  if (sign='1') then
    all_hlp := all_ones or not high_lacells or activ_lacells;
  else
    all_hlp := all_zeros or not high_lacells or activ_lacells;
  end if;
```

```
  carry_resolve:= (cout2(0) or cout2(1) or cout2(2) or cout2(3) or
                   cout2(4) or cout2(5) or cout2(6) or cout2(7) or
                   cout2(8) or cout2(9)) or
                  ((clow(0) or clow(1) or clow(2) or clow(3) or clow(4) or
                    clow(5) or clow(6) or clow(7) or clow(8) or clow(9)) and
                   (cout1(0) or cout1(1) or cout1(2) or cout1(3) or
                    cout1(4) or cout1(5) or cout1(6) or cout1(7) or
                    cout1(8) or cout1(9)));
    carry0 := '0';
    carry1 := all_hlp(0);
    carry2 := all_hlp(0) and all_hlp(1);
    carry3 := all_hlp(0) and all_hlp(1) and all_hlp(2);
    carry4 := all_hlp(0) and all_hlp(1) and all_hlp(2) and all_hlp(3);
    carry5 := all_hlp(0) and all_hlp(1) and all_hlp(2) and all_hlp(3)
                         and all_hlp(4);
    carry6 := all_hlp(0) and all_hlp(1) and all_hlp(2) and all_hlp(3)
                         and all_hlp(4) and all_hlp(5);
    carry7 := all_hlp(0) and all_hlp(1) and all_hlp(2) and all_hlp(3)
                         and all_hlp(4) and all_hlp(5) and all_hlp(6);
    carry8 := all_hlp(0) and all_hlp(1) and all_hlp(2) and all_hlp(3)
                         and all_hlp(4) and all_hlp(5) and all_hlp(6)
                         and all_hlp(7);
    carry9 := all_hlp(0) and all_hlp(1) and all_hlp(2) and all_hlp(3)
                         and all_hlp(4) and all_hlp(5) and all_hlp(6)
                         and all_hlp(7) and all_hlp(8);

    carry(0) <= carry0 and carry_resolve;
    carry(1) <= carry1 and carry_resolve;
    carry(2) <= carry2 and carry_resolve;
    carry(3) <= carry3 and carry_resolve;
    carry(4) <= carry4 and carry_resolve;
    carry(5) <= carry5 and carry_resolve;
    carry(6) <= carry6 and carry_resolve;
    carry(7) <= carry7 and carry_resolve;
    carry(8) <= carry8 and carry_resolve;
    carry(9) <= carry9 and carry_resolve;
end process carry_res;
end generate carry_res_csa;


-- purpose: computes the sign of the actual context depending
--          on the input sign and the carry out of the last lacell
-- type   : combinational
-- inputs : sign, cout1, cout2, clow
  -- outputs: ctxt_sign, ctxt_overflow
  context_sign: process (sign, carry, oldover, oldsign, -- clk,
                         neg_context, cond_neg)
--     variable c_sign, c_overflow : std_logic;
--     variable sel                : std_logic_vector(4 downto 0);
  begin  -- process context_sign
    sel <= neg_context&cond_neg&carry(9)&sign&oldsign;
    case sel is
      when "10111"|"10101"|"00111"
                   => c_sign     <= '0';
                      c_overflow <= oldover;
      when "10110"|"10100"|"11101"|"00100"
                   => c_sign     <= '1';
                      c_overflow <= oldover;
      when "00001" => c_sign     <= '0';
                      c_overflow <= oldover;
      when others  => c_sign     <= oldsign;
                      c_overflow <= oldover;
      end case;
  end process context_sign;

debug_out <= "0000"&neg_context&cond_neg&carry(9)&sign&data_sign&"000"&
             ctxt_sign&ctxt_overflow&oldsign&oldover;

sign_over: process (clk, c_sign, c_overflow)
  begin  -- process sign_over
    if (clk'event and clk='0') then
      if (init_ctxt='1') then
        ctxt_sign     <= '0';
        ctxt_overflow <= '0';
      else
        ctxt_sign     <= c_sign;
        ctxt_overflow <= c_overflow;
      end if;
    end if;
  end process sign_over;

  -- purpose: computes the msw of the actual context
  -- type   : combinational
  -- inputs : all_zero, all_one
  -- outputs: msw_lacells, msw_context
  msw: process (clk, all_zeros, all_ones, carry)
    variable all_hlp, zero_hlp : std_logic_vector(num_cell-1 downto 0);
    variable msw_hlp           : std_logic_vector(num_cell-1 downto 0);
```

```
  variable sticky            : std_logic;
begin  -- process msw
  if (carry(9)='1') then
    all_hlp := all_ones;
  else
    all_hlp := all_zeros;
  end if;
  zero_hlp  := all_zeros;

  if all_hlp(2)='1' and all_hlp(3)='1' and all_hlp(4)='1' and
     all_hlp(5)='1' and all_hlp(6)='1' and all_hlp(7)='1' and
     all_hlp(8)='1' and all_hlp(9)='1' then
    msw_hlp := "0000000011";
    sticky  := '0';
  end if;
  if all_hlp(2)='0' and all_hlp(3)='1' and all_hlp(4)='1' and
     all_hlp(5)='1' and all_hlp(6)='1' and all_hlp(7)='1' and
     all_hlp(8)='1' and all_hlp(9)='1' then
    msw_hlp := "0000000110";
    sticky  := not (zero_hlp(0));
  end if;
  if all_hlp(3)='0' and all_hlp(4)='1' and all_hlp(5)='1' and
     all_hlp(6)='1' and all_hlp(7)='1' and all_hlp(8)='1' and
     all_hlp(9)='1' then
    msw_hlp := "0000001100";
    sticky  := not (zero_hlp(1) and zero_hlp(0));
  end if;
  if all_hlp(4)='0' and all_hlp(5)='1' and
     all_hlp(6)='1' and all_hlp(7)='1' and all_hlp(8)='1' and
     all_hlp(9)='1' then
    msw_hlp := "0000011000";
    sticky  := not (zero_hlp(2) and zero_hlp(1) and zero_hlp(0));
  end if;
  if all_hlp(5)='0' and
     all_hlp(6)='1' and all_hlp(7)='1' and all_hlp(8)='1' and
     all_hlp(9)='1' then
    msw_hlp := "0000110000";
    sticky  := not (zero_hlp(3) and zero_hlp(2) and zero_hlp(1) and
                    zero_hlp(0));
  end if;
  if all_hlp(6)='0' and all_hlp(7)='1' and all_hlp(8)='1' and
     all_hlp(9)='1' then
    msw_hlp := "0001100000";
    sticky  := not (zero_hlp(4) and zero_hlp(3) and zero_hlp(2) and
                    zero_hlp(1) and zero_hlp(0));
  end if;
  if all_hlp(7)='0' and all_hlp(8)='1' and all_hlp(9)='1' then
    msw_hlp := "0011000000";
    sticky  := not (zero_hlp(5) and zero_hlp(4) and zero_hlp(3) and
                    zero_hlp(2) and zero_hlp(1) and zero_hlp(0));
  end if;
  if all_hlp(8)='0' and all_hlp(9)='1' then
    msw_hlp := "0110000000";
    sticky  := not (zero_hlp(6) and zero_hlp(5) and zero_hlp(4) and
                    zero_hlp(3) and zero_hlp(2) and zero_hlp(1) and
                    zero_hlp(0));
  end if;
  if all_hlp(9)='0' then
    msw_hlp := "1100000000";
     sticky  := not (zero_hlp(7) and zero_hlp(6) and zero_hlp(5) and
                     zero_hlp(4) and zero_hlp(3) and zero_hlp(2) and
                     zero_hlp(1) and zero_hlp(0));
  end if;


  if (clk'event and clk='0') then
    msw_lacells <= msw_hlp;
    msw_context <= sticky & msw_hlp;
  end if;
end process msw;


lacells_cs_arch: if (carry_store=1) generate
lacells_cs_even: for i in 0 to 4 generate
lacell : lacell_cs
  generic map ( partadder_width => partadder_width)
  port map    ( a               => data_in_low,
                context         => context,
                clk_store       => clk_store,
                init_ctxt       => init_ctxt,
                addsub          => data_sign,
                store           => write_enable(2*i),
                carry_in        => wire(2*i),
                carry_out       => carry(2*i),
                activ_cell      => activ_lacells(2*i),
                enable_out      => enable_outs(2*i),
                load_data       => transparent(2*i),
                neg_context     => neg_context,
```

```
                all_zero        => all_zeros(2*i),
                all_one         => all_ones(2*i),
                sum             => data_out0 );

  wire(2*i+1) <= carry(2*i);
end generate lacells_cs_even;

lacells_cs_odd: for i in 0 to 4 generate
lacell : lacell_cs
  generic map ( partadder_width => partadder_width)
  port map    ( a               => data_in_high,
                context         => context,
                clk_store       => clk_store,
                init_ctxt       => init_ctxt,
                addsub          => data_sign,
                store           => write_enable(2*i+1),
                carry_in        => wire(2*i+1),
                carry_out       => carry(2*i+1),
                activ_cell      => activ_lacells(2*i+1),
                enable_out      => enable_outs(2*i+1),
                load_data       => transparent(2*i+1),
                neg_context     => neg_context,
                all_zero        => all_zeros(2*i+1),
                all_one         => all_ones(2*i+1),
                sum             => data_out1 );

  wire(2*i+2) <= carry(2*i+1);
end generate lacells_cs_odd;
end generate lacells_cs_arch;

lacells_csa_arch: if (carry_store/=1) generate
lacells_csa_even: for i in 0 to 4 generate
lacell : lacell
  generic map ( partadder_width => partadder_width)
  port map    ( a               => data_in_low,
                context         => context,
                clk_store       => clk_store,
                addsub          => data_sign,
                store           => write_enable(2*i),
                ci_low          => wire(2*i),
                carry           => carry(2*i),
                activ_cell      => activ_lacells(2*i),
                high_cell       => high_lacells(2*i),
                enable_out      => enable_outs(2*i),
                load_data       => transparent(2*i),
                neg_context     => neg_context,
                all_zero        => all_zeros(2*i),
                all_one         => all_ones(2*i),
                co_low          => clow(2*i),
                co_high1        => cout1(2*i),
                co_high2        => cout2(2*i),
                sum             => data_out0 );

  wire(2*i+1) <= clow(2*i);
end generate lacells_csa_even;

lacells_csa_odd: for i in 0 to 4 generate
lacell : lacell
  generic map ( partadder_width => partadder_width)
  port map    ( a               => data_in_high,
                context         => context,
                clk_store       => clk_store,
                addsub          => data_sign,
                store           => write_enable(2*i+1),
                ci_low          => wire(2*i+1),
                carry           => carry(2*i+1),
                activ_cell      => activ_lacells(2*i+1),
                high_cell       => high_lacells(2*i+1),
                enable_out      => enable_outs(2*i+1),
                load_data       => transparent(2*i+1),
                neg_context     => neg_context,
                all_zero        => all_zeros(2*i+1),
                all_one         => all_ones(2*i+1),
                co_low          => clow(2*i+1),
                co_high1        => cout1(2*i+1),
                co_high2        => cout2(2*i+1),
                sum             => data_out1 );

  wire(2*i+2) <= clow(2*i+1);
end generate lacells_csa_odd;
end generate lacells_csa_arch;


  context_out   <= ctxt_data_o(15 downto 0);

  oldsign       <= ctxt_data_o(11);
  oldover       <= ctxt_data_o(12);
```

```
  ctxt_data_i(15 downto 0)  <= (others => '0') when (init_ctxt='1') else
                               "101" & ctxt_overflow & ctxt_sign & msw_context;

  not_clk_store <= not clk_store;

  ctxt_ram_log : ramsuu16x16
             port map ( A       => context,
                        CLK     => clk_store,
                        D       => ctxt_data_i,
                        WE      => VCC, -- write_enable(10),
                        SPO     => ctxt_data_o );

  data_out <= data_out1 & data_out0;

end struct;
```

D.12 Datei: M_spez_suma_sd.vhd

```
library ieee;
use ieee.std_logic_1164.all;

library work;
use work.constants.all;

library work;
use work.components.all;

entity M_Spez_SuMa is
  port (
        clk              :  in std_logic;
        clk_store        :  in std_logic;
        MIDefprod        :  in std_logic;
        MII              :  in std_logic;
        MID0             :  in std_logic_vector ( 31 downto 0);
        MID1             :  in std_logic_vector ( 31 downto 0);
        MISCmd           :  in std_logic_vector ( 31 downto 0);
        MODExpo          : out std_logic_vector ( 15 downto 0);
        MOVData          : out std_logic_vector (  1 downto 0);
        MOData           : out std_logic_vector (131 downto 0);
        MODebug          : out std_logic_vector (debug1 downto 0)
       );
end M_Spez_SuMa;

architecture struct of M_Spez_SuMa is

signal VCC, GND, rst           : std_logic;

signal wire_DecO_D0            : std_logic_vector ( 31 downto 0);
signal wire_DecO_D1            : std_logic_vector ( 31 downto 0);
signal wire_data0_out          : std_logic_vector ( 31 downto 0);
signal wire_data1_out          : std_logic_vector ( 31 downto 0);
signal data_sel_in             : std_logic_vector ( 63 downto 0);

signal wire_DecO_cmds          : std_logic_vector (  6 downto 0);
signal wire_DecO_ctxt          : std_logic_vector (  3 downto 0);
signal wire_context            : std_logic_vector (  3 downto 0);
signal wire_sub_context        : std_logic_vector (  3 downto 0);
signal wire_sub_context_pipe   : std_logic_vector (  3 downto 0);
signal wire_DecO_dblcnt        : std_logic_vector (  1 downto 0);
signal wire_DecO_expo0         : std_logic_vector ( 10 downto 0);
signal wire_DecO_expo1         : std_logic_vector ( 10 downto 0);
signal wire_DecO_hidden        : std_logic_vector (  1 downto 0);
signal wire_DecO_mode          : std_logic_vector (  2 downto 0);
signal wire_DecO_sign          : std_logic;
signal wire_DecO_sign_vector   : std_logic_vector (  0 downto 0);
signal wire_sign_vector        : std_logic_vector (  0 downto 0);
signal wire_sign               : std_logic;
signal wire_sign_suma          : std_logic;
signal wire_sign_suma_in       : std_logic;
signal wire_DecO_cells         : std_logic_vector (  3 downto 0);
signal wire_DecO_e_cmd         : std_logic_vector (  5 downto 0);
signal wire_DecO_probes        : std_logic_vector (  1 downto 0);

signal wire_activ_lacells      : std_logic_vector (  9 downto 0);
signal wire_activ_lacells_pipe : std_logic_vector (  9 downto 0);
signal wire_high_lacells       : std_logic_vector (  9 downto 0);
signal wire_high_lacells_pipe  : std_logic_vector (  9 downto 0);
signal wire_tpt_pipe           : std_logic_vector (  9 downto 0);
signal wire_expo_tpt           : std_logic_vector (  9 downto 0);

signal wire_cmds               : std_logic_vector (  6 downto 0);
signal wire_cmds_in            : std_logic_vector (  6 downto 0);
signal wire_select             : std_logic_vector (  1 downto 0);
signal wire_probes             : std_logic_vector (  1 downto 0);
signal wire_DecO_select        : std_logic_vector (  1 downto 0);
signal wire_neg_context        : std_logic;
signal wire_cb_resolve         : std_logic;
signal wire_add_tpt            : std_logic;
```

```
signal wire_in_out             : std_logic;
signal wire_in_out_vec         : std_logic_vector (  0 downto 0);
signal wire_data_valid1        : std_logic_vector (  0 downto 0);
signal wire_data_valid2        : std_logic_vector (  1 downto 0);
signal wire_double_res         : std_logic;
signal wire_init_ctxt          : std_logic;
signal wire_cond_neg           : std_logic;

signal valid1, valid2          : std_logic;

signal wire_sel0, wire_sel1    : std_logic;

signal wire_p                  : std_logic_vector ( 63 downto 0);
signal mid0_mid1, wire_data_in : std_logic_vector ( 63 downto 0);
signal wire_shift_amount       : std_logic_vector (  6 downto 0);
signal wire_shift_amount_pipe  : std_logic_vector (  6 downto 0);
signal wire_shifter_in         : std_logic_vector ( 63 downto 0);
signal wire_shifter_out        : std_logic_vector (127 downto 0);
signal wire_data_suma          : std_logic_vector (127 downto 0);
signal wire_suma_out           : std_logic_vector (127 downto 0);
signal wire_data_out           : std_logic_vector (131 downto 0);
signal wire_context_out        : std_logic_vector ( 15 downto 0);
signal wire_debug_out          : std_logic_vector ( 15 downto 0);
signal wire_data_sel           : std_logic_vector ( 63 downto 0);
signal defprod                 : std_logic_vector ( 63 downto 0);

begin

  -- jetzt kommen die pipeline stufen ...
  -- gelatched werden : mode context mantissen exponenten
  --  der produktexponent wird in exponent0(7) und exponent1 geschrieben
  --  dabei ist exponent0(7) das msb des produktexponenten!!
  --   expont1 expont0 mantisse1               mantisse0
  --   |------||------||----------------------||----------------------|
  --   f..............0f..............0f..............0f..............0

  VCC     <= '1';
  GND     <= '0';

  rst       <= MII;
  mid0_mid1 <= MID0 & MID1;


decoding : Decode port map ( clk          => clk,
                             rst          => rst,
                             DecI_D0      => MID0,
                             DecI_D1      => MID1,
                             DecI_Cmd     => MISCmd,
                             DecO_D0      => wire_DecO_D0,
                             DecO_D1      => wire_DecO_D1,
                             DecO_expo0   => wire_DecO_expo0,
                             DecO_expo1   => wire_DecO_expo1,
                             DecO_hidden  => wire_DecO_hidden,
                             DecO_ctxt    => wire_DecO_ctxt,
                             DecO_dblcnt  => wire_DecO_dblcnt,
                             DecO_cmds    => wire_DecO_cmds,
                             DecO_mode    => wire_DecO_mode,
                             DecO_cells   => wire_DecO_cells,
                             DecO_e_cmd   => wire_DecO_e_cmd,
                             DecO_probes  => wire_DecO_probes,
                             DecO_select  => wire_DecO_select,
                             DecO_sign    => wire_DecO_sign );

data : data_select port map (clk            => clk,
                             rst            => rst,
                             data0_in       => wire_DecO_D0,
                             data1_in       => wire_DecO_D1,
                             hidden_bit     => wire_DecO_hidden,
                             data_mode      => wire_DecO_mode,
                             double_counter => wire_DecO_dblcnt,
                             data0_out      => wire_data0_out,
                             data1_out      => wire_data1_out );

expoari : expari port map ( clk            => clk,
                            rst            => rst,
                            mode_lacells   => wire_DecO_e_cmd,
                            ea             => wire_DecO_expo0,
                            eb             => wire_DecO_expo1,
                            data_mode      => wire_DecO_mode,
                            double_counter => wire_DecO_dblcnt,
                            cells_to_read  => wire_DecO_cells,
                            sub_context    => wire_sub_context,
                            shift_amount   => wire_shift_amount,
                            activ_lacells  => wire_activ_lacells,
                            tpt_lacells    => wire_expo_tpt,
                            high_lacells   => wire_high_lacells);

multiplier : mul32x32u_pipe port map (a   => wire_data0_out,
```

```
                                    b   => wire_data1_out,
                                    clk => clk,
                                    p   => wire_p);

  -- multiplexor data_in or product or def_prod or data_sel to shifter_in
    wire_sel0 <= wire_select(1);
    wire_sel1 <= wire_select(0);

  defprod <= default_prod when MIDefprod='0' else const_prod;

multiplex : selector
     port map ( data_in   => wire_data_in,
                data_prod => wire_p,
                data_sel  => wire_data_sel,
                data_def  => defprod,
                clk       => clk,
                rst       => rst,
                select0   => wire_sel0,
                select1   => wire_sel1,
                data_out  => wire_shifter_in );

  wire_DecO_sign_vector(0) <= wire_DecO_sign;

sign_pipe : pipe generic map (width => 1,
                              depth => dsel_depth+multi_depth+select_depth+
                                       shift_depth )
    port map (dpipe  => wire_DecO_sign_vector,
              clk    => clk,
              rst    => rst,
              enable => VCC,
              qpipe  => wire_sign_vector);

  wire_sign                <= wire_sign_vector(0);


context_pipe : pipe generic map (width => 4,
                                 depth => dsel_depth+multi_depth+select_depth+
                                          shift_depth+dblres_depth )
  port map (dpipe  => wire_DecO_ctxt,
            clk    => clk,
            rst    => rst,
            enable => VCC,
            qpipe  => wire_context);
```

```
data_pipe : pipe generic map (width => 64,
                              depth => decod_depth+dsel_depth+multi_depth )
  port map (dpipe  => mid0_mid1,
            clk    => clk,
            rst    => rst,
            enable => VCC,
            qpipe  => wire_data_in);

  data_sel_in <= wire_data0_out&wire_data1_out;

data_sel_pipe : pipe generic map (width => 64,
                                  depth => multi_depth )
  port map (dpipe  => data_sel_in,
            clk    => clk,
            rst    => rst,
            enable => VCC,
            qpipe  => wire_data_sel);

select_pipe : pipe generic map (width => 2,
                                depth => dsel_depth+multi_depth )
  port map (dpipe  => wire_DecO_select,
            clk    => clk,
            rst    => rst,
            enable => VCC,
            qpipe  => wire_select);

cmds_pipe : pipe generic map (width => 7,
                              depth => dsel_depth+multi_depth+select_depth+
                                       shift_depth )
  port map (dpipe  => wire_DecO_cmds,
            clk    => clk,
            rst    => rst,
            enable => VCC,
            qpipe  => wire_cmds);

shift_amount_pipe : pipe generic map (width => 7,
                                      depth => multi_depth+select_depth )
  port map (dpipe  => wire_shift_amount,
            clk    => clk,
            rst    => rst,
            enable => VCC,
            qpipe  => wire_shift_amount_pipe);
```

```
activ_lacells_pipe : pipe generic map (width => 10,
                                       depth => multi_depth+select_depth+
                                                shift_depth+dblres_depth )
  port map (dpipe  => wire_activ_lacells,
            clk    => clk,
            rst    => rst,
            enable => VCC,
            qpipe  => wire_activ_lacells_pipe);

high_lacells_pipe : pipe generic map (width => 10,
                                      depth => multi_depth+select_depth+
                                               shift_depth+dblres_depth )
  port map (dpipe  => wire_high_lacells,
            clk    => clk,
            rst    => rst,
            enable => VCC,
            qpipe  => wire_high_lacells_pipe);

  tpt_pipe : pipe generic map (width => 10,
                               depth => multi_depth+select_depth+shift_depth+
                                        dblres_depth )
    port map (dpipe  => wire_expo_tpt,
              clk    => clk,
              rst    => rst,
              enable => VCC,
              qpipe  => wire_tpt_pipe);


sub_context_pipe : pipe generic map (width => 4,
                                     depth => multi_depth+select_depth+
                                              shift_depth+dblres_depth )
  port map (dpipe  => wire_sub_context,
            clk    => clk,
            rst    => rst,
            enable => VCC,
            qpipe  => wire_sub_context_pipe);

shifter : shift064_128 port map (shift_amount => wire_shift_amount_pipe,
                                 shifter_in   => wire_shifter_in,
                                 clk          => clk,
                                 rst          => rst,
                                 shifter_out  => wire_shifter_out);

probes_pipe : pipe generic map (width => 2,
                                depth => dsel_depth+multi_depth+select_depth+
                                         shift_depth+dblres_depth )
    port map (dpipe  => wire_Dec0_probes,
              clk    => clk,
              rst    => rst,
              enable => VCC,
              qpipe  => wire_probes);

out_valid_pipe1 : pipe generic map (width => 1,
                                    depth => 1 )
  port map (dpipe  => wire_in_out_vec,
            clk    => clk,
            rst    => rst,
            enable => VCC,
            qpipe  => wire_data_valid1);


out_valid_pipe2 : pipe generic map (width => 2,
                                    depth => 1 )
  port map (dpipe  => wire_probes,
            clk    => clk,
            rst    => rst,
            enable => VCC,
            qpipe  => wire_data_valid2);


wire_cb_resolve    <= wire_cmds_in(6);
wire_cond_neg      <= wire_cmds_in(5);
wire_add_tpt       <= wire_cmds_in(4);
wire_in_out_vec(0) <= wire_cmds_in(3);
wire_in_out        <= wire_cmds_in(3);
wire_init_ctxt     <= wire_cmds_in(2);
wire_double_res    <= wire_cmds_in(1);
wire_neg_context   <= wire_cmds_in(0);

  dblresreg: if (double_resolve=1) generate
    double_reg : dbl_resolve port map (shifter_in   => wire_shifter_out,
                                       suma_in      => wire_suma_out,
                                       clk          => clk,
                                       rst          => rst,
                                       sign_in      => wire_sign,
                                       sign_suma    => wire_sign_suma,
                                       sign_out     => wire_sign_suma_in,
                                       cmds_in      => wire_cmds,
```

```
                                    cmds_out     => wire_cmds_in,
                                    data_to_suma => wire_data_suma);

    wire_suma_out  <= wire_data_out(129 downto 66)&wire_data_out(63 downto 0);
    wire_sign_suma <= wire_context_out(11);
  end generate dblresreg;

  nodblresreg: if (double_resolve/=1) generate
    wire_data_suma    <= wire_shifter_out;
    wire_sign_suma_in <= wire_sign;
    wire_cmds_in      <= wire_cmds;
  end generate nodblresreg;


  SuMa_Core : SuMa port map (
    data_in       => wire_data_suma,
    sign          => wire_sign_suma_in,
    activ_lacells => wire_activ_lacells_pipe,
    high_lacells  => wire_high_lacells_pipe,
    tpt_lacells   => wire_tpt_pipe,
    clk           => clk,
    clk_store     => clk_store,
    rst           => rst,
    context       => wire_context,
    cb_resolve    => wire_cb_resolve,
    neg_context   => wire_neg_context,
    cond_neg      => wire_cond_neg,
    add_tpt       => wire_add_tpt,
    in_out        => wire_in_out,
    double_res    => wire_double_res,
    init_ctxt     => wire_init_ctxt,
    data_out      => wire_data_out,
    context_out   => wire_context_out,
    debug_out     => wire_debug_out
  );

MODExpo <= wire_context_out;
MOData  <= wire_data_out;

MOVData(0) <= wire_data_valid2(1);
MOVData(1) <= wire_probes(1);
MODebug <= wire_context&
           wire_in_out&wire_add_tpt&wire_neg_context&wire_init_ctxt&
           wire_sign_suma_in&wire_double_res&"000000"&wire_debug_out&
--         wire_sign_suma_in&wire_double_res&wire_high_lacells_pipe&
--         wire_cond_neg&"0"&wire_activ_lacells_pipe&              -- reg3
           wire_data_suma;                                        -- reg4-reg7

end struct;
```

D.13 Datei: spyder2suma.vhd

```
library ieee;
use ieee.std_logic_1164.all;

library work;
use work.versionnumber.all;

library work;
use work.constants.all;

library work;
use work.components.all;

-- synopsys translate_off
-- this library is only needed to avoid the black box warnings in synplify
-- synthesis translate_on
library synplify;
use synplify.attributes.all;
-- synopsys translate_on

-------------------------------------------------------------------------------
-- reg0  msw from double precision or single precision OPERAND0
-- reg10 lsw from double precision                     OPERAND0
-- reg1  msw from double precision or single precision OPERAND1
-- reg11 lsw from double precision                     OPERAND1
-- reg2  Command word
-- reg3(31 downto 16) <= suma_expo;
-- reg3(3 downto 0)   <= suma_data(131 downto 128);
-- reg4               <= suma_data(127 downto  96);
-- reg5               <= suma_data( 95 downto  64);
-- reg6               <= suma_data( 63 downto  32);
-- reg7               <= suma_data( 31 downto   0);
-------------------------------------------------------------------------------

entity spyder is
  port ( ISCLK          : in    std_logic;
         IUCLK          : in    std_logic;
```

```
        GNET2_GRESET_0 : in    std_logic;
        FRD_0          : in    std_logic;
        FWR_0          : in    std_logic;
        LA             : in    std_logic_vector(31 downto 2) ;
        LBE0_0         : in    std_logic;
        LBE1_0         : in    std_logic;
        LBE2_0         : in    std_logic;
        LBE3_0         : in    std_logic;
        LD             : inout std_logic_vector(31 downto 0) );
end spyder;

architecture struct of spyder is

type statetype is (waiting, single, mxm, lxl, lxm, mxl, reset);

signal clk          : std_logic;
signal SCLK         : std_logic;
signal clk_store    : std_logic;
signal regSpaceCS   : std_logic;
signal reg0_suma    : std_logic_vector( 31 downto 0);
signal reg1_suma    : std_logic_vector( 31 downto 0);
signal reg2_suma    : std_logic_vector( 31 downto 0);
signal reg0         : std_logic_vector( 31 downto 0);
signal reg1         : std_logic_vector( 31 downto 0);
signal reg2         : std_logic_vector( 31 downto 0);
signal reg3         : std_logic_vector( 31 downto 0);
signal reg4         : std_logic_vector( 31 downto 0);
signal reg5         : std_logic_vector( 31 downto 0);
signal reg6         : std_logic_vector( 31 downto 0);
signal reg7         : std_logic_vector( 31 downto 0);
signal reg8         : std_logic_vector( 31 downto 0);
signal reg9         : std_logic_vector( 31 downto 0);
signal reg10        : std_logic_vector( 31 downto 0);
signal reg11        : std_logic_vector( 31 downto 0);
signal reg12        : std_logic_vector( 31 downto 0);
signal reg13        : std_logic_vector( 31 downto 0);
signal reg14        : std_logic_vector( 31 downto 0);
signal reg15        : std_logic_vector( 31 downto 0);  -- Version/Properties
                                                       -- Register
signal reg_expo     : std_logic_vector( 15 downto 0);
signal reg_data     : std_logic_vector(131 downto 0);
signal regs_locked  : std_logic;
signal regs_unlock  : std_logic;
signal double_ready : std_logic;
```

```
signal currentstate : statetype;
signal nextstate    : statetype;
attribute syn_state_machine of currentstate: signal is true;
attribute syn_encoding      of currentstate: signal is "safe,onehot";


signal probe1_wire    : std_logic_vector( 31 downto 0);
signal probe_on       : std_logic;
signal probe_valid    : std_logic;
signal select_default : std_logic;
signal data_valid     : std_logic_vector(  1 downto 0);
signal debug_data     : std_logic_vector(debug1 downto 0);

signal GND : std_logic;

begin

  SCLK      <= ISCLK;
  clk_store <= IUCLK;
  clk       <= IUCLK;

  regSpaceCS <= '1' when LA(31 downto 6) = "00001000000000000000000000"
                else '0';


  reg0WriteP : process(GNET2_GRESET_0, SCLK, FWR_0, regs_locked,
                       LBE3_0, LBE2_0, LBE1_0, LBE0_0,  LA, regSpacecs, LD)
  begin
    if (GNET2_GRESET_0 = '0') then
      reg0 <= (others => '0');
    elsif (SCLK'event and SCLK = '1') then
      if (regSpaceCS = '1' and LA(5 downto 2) = "0000" and
          FWR_0 = '0' and regs_locked='0') then
        if (LBE3_0 = '0') then
          reg0(31 downto 24) <= LD(31 downto 24) ;
        end if;
        if (LBE2_0 = '0') then
          reg0(23 downto 16) <= LD(23 downto 16);
        end if;
        if (LBE1_0 = '0') then
          reg0(15 downto 8) <= LD(15 downto 8);
        end if;
        if (LBE0_0 = '0') then
          reg0(7 downto 0) <= LD(7 downto 0);
```

```
      end if;
    end if;
  end if;
end process;


reg10WriteP : process(GNET2_GRESET_0, SCLK, FWR_0, regs_locked,
                      LBE3_0, LBE2_0, LBE1_0, LBE0_0,  LA, regSpacecs, LD)
begin
  if (GNET2_GRESET_0 = '0') then
    reg10 <= (others => '0');
  elsif (SCLK'event and SCLK = '1') then
    if (regSpaceCS = '1' and LA(5 downto 2) = "1010" and
        FWR_0 = '0' and regs_locked='0') then
      if (LBE3_0 = '0') then
        reg10(31 downto 24) <= LD(31 downto 24) ;
      end if;
      if (LBE2_0 = '0') then
        reg10(23 downto 16) <= LD(23 downto 16);
      end if;
      if (LBE1_0 = '0') then
        reg10(15 downto 8) <= LD(15 downto 8);
      end if;
      if (LBE0_0 = '0') then
        reg10(7 downto 0) <= LD(7 downto 0);
      end if;
    end if;
  end if;
end process;


reg1WriteP : process(GNET2_GRESET_0, SCLK, FWR_0, regs_locked,
                     LBE3_0, LBE2_0, LBE1_0, LBE0_0,  LA, regSpacecs, LD)
begin
  if (GNET2_GRESET_0 = '0') then
    reg1 <= (others => '0');
  elsif (SCLK'event and SCLK = '1') then
    if (regSpaceCS = '1' and LA(5 downto 2) = "0001" and
        FWR_0 = '0' and regs_locked='0') then
      if (LBE3_0 = '0') then
        reg1(31 downto 24) <= LD(31 downto 24) ;
      end if;
      if (LBE2_0 = '0') then
        reg1(23 downto 16) <= LD(23 downto 16);
      end if;
      if (LBE1_0 = '0') then
        reg1(15 downto 8) <= LD(15 downto 8);
      end if;
      if (LBE0_0 = '0') then
        reg1(7 downto 0) <= LD(7 downto 0);
      end if;
    end if;
  end if;
end process;


reg11WriteP : process(GNET2_GRESET_0, SCLK, FWR_0, regs_locked,
                      LBE3_0, LBE2_0, LBE1_0, LBE0_0,  LA, regSpacecs, LD)
begin
  if (GNET2_GRESET_0 = '0') then
    reg11 <= (others => '0');
  elsif (SCLK'event and SCLK = '1') then
    if (regSpaceCS = '1' and LA(5 downto 2) = "1011" and
        FWR_0 = '0' and regs_locked='0') then
      if (LBE3_0 = '0') then
        reg11(31 downto 24) <= LD(31 downto 24) ;
      end if;
      if (LBE2_0 = '0') then
        reg11(23 downto 16) <= LD(23 downto 16);
      end if;
      if (LBE1_0 = '0') then
        reg11(15 downto 8) <= LD(15 downto 8);
      end if;
      if (LBE0_0 = '0') then
        reg11(7 downto 0) <= LD(7 downto 0);
      end if;
    end if;
  end if;
end process;


reg2WriteP : process(GNET2_GRESET_0, SCLK, FWR_0, regs_locked,
                     LBE3_0, LBE2_0, LBE1_0, LBE0_0,  LA, regSpacecs, LD)
begin
  if (GNET2_GRESET_0 = '0') then
    reg2      <= (others => '0');
  elsif (SCLK'event and SCLK = '1') then
    if (regs_unlock='1') then
```

```
      reg2      <= (others => '0');
    elsif (regSpaceCS = '1' and LA(5 downto 2) = "0010" and
        FWR_0 = '0' and regs_locked='0') then
      if (LBE3_0 = '0') then
        reg2(31 downto 24) <= LD(31 downto 24) ;
      end if;
      if (LBE2_0 = '0') then
        reg2(23 downto 16) <= LD(23 downto 16);
      end if;
      if (LBE1_0 = '0') then
        reg2(15 downto 8) <= LD(15 downto 8);
      end if;
      if (LBE0_0 = '0') then
        reg2(7 downto 0) <= LD(7 downto 0);
      end if;
    end if;
  end if;
end process;


fsm_seq: process (clk, GNET2_GRESET_0)
begin
  if GNET2_GRESET_0 = '0' then
    currentstate <= waiting;
  elsif clk'event and clk = '1' then
    if (regs_locked='1') then
      currentstate <= nextstate;
    end if;
  end if;
end process fsm_seq;


fsm_comb: process (reg2, currentstate)
begin
  case currentstate is
    when mxl     => nextstate <= reset;
    when waiting => if (reg2(cmd_mode_tpt downto cmd_mode_dbl)="101") then
                      nextstate <= mxm;
                    else
                      nextstate <= single;
                    end if;
    when single  => nextstate <= reset;
    when lxm     => nextstate <= mxl;
    when lxl     => nextstate <= lxm;
    when mxm     => nextstate <= lxl;
    when others  => nextstate <= waiting;
  end case;
end process fsm_comb;


with currentstate select
  reg0_suma <=
  reg0            when mxm|single,
  reg10           when lxl,
  (others => '0') when others;

with currentstate select
  reg1_suma <=
  reg1            when mxm|single,
  reg11           when lxl,
  (others => '0') when others;

with currentstate select
  reg2_suma <=
  reg2(31 downto cmd_dbl_cnt2+1)&"00"&reg2(cmd_dbl_cnt1-1 downto 0) when
                                                                  mxm|single,
  reg2(31 downto cmd_dbl_cnt2+1)&"01"&reg2(cmd_dbl_cnt1-1 downto 0) when lxl,
  reg2(31 downto cmd_dbl_cnt2+1)&"10"&reg2(cmd_dbl_cnt1-1 downto 0) when lxm,
  reg2(31 downto cmd_dbl_cnt2+1)&"11"&reg2(cmd_dbl_cnt1-1 downto 0) when mxl,
  (others => '0')                                                    when
                                                                    others;


with currentstate select
  regs_unlock <=
  '1' when reset,
  '0' when others;


setlock: process (clk, GNET2_GRESET_0, reg2, regs_unlock)
begin
  if GNET2_GRESET_0 = '0' then
    regs_locked  <= '0';
  elsif clk'event and clk = '1' then
    if (regs_unlock='1') then
      regs_locked  <= '0';
    elsif ((reg2(cmd_ok_flag4) AND reg2(cmd_ok_flag3) AND
            reg2(cmd_ok_flag2) AND reg2(cmd_ok_flag1)) = '1') then
      regs_locked  <= '1';
```

```
    end if;
  end if;
end process;


readBufP : process (FRD_0, regSpaceCS,
                    reg0, reg1, reg10, reg11, reg15, reg8, reg9,
                    reg12, reg13, reg14,
                    reg2, reg3, reg4, reg5, reg6, reg7, LA)
begin
  if (regSpaceCS = '1' and FRD_0 = '0') then
    case LA(5 downto 2) is
          when "0000" => LD <= reg0;
          when "0001" => LD <= reg1;
          when "0010" => LD <= reg2;
          when "0011" => LD <= reg3;
          when "0100" => LD <= reg4;
          when "0101" => LD <= reg5;
          when "0110" => LD <= reg6;
          when "0111" => LD <= reg7;
          when "1000" => LD <= reg8;
          when "1001" => LD <= reg9;
          when "1010" => LD <= reg10;
          when "1011" => LD <= reg11;
          when "1100" => LD <= reg12;
          when "1101" => LD <= reg13;
          when "1110" => LD <= reg14;
          when "1111" => LD <= reg15;
    end case;
  else
    LD <= (others => 'Z');
  end if;
end process;

SuMatrix : M_Spez_SuMa port map (
  clk       => clk,
  clk_store => clk_store,
  MIDefprod => select_default,
  MII       => GNET2_GRESET_0,
  MID0      => reg0_suma,
  MID1      => reg1_suma,
  MISCmd    => reg2_suma,
  MODExpo   => reg_expo,
  MOVData   => data_valid,
  MOData    => reg_data,
  MODebug   => debug_data
);



probe_on    <= data_valid(1);
probe_valid <= data_valid(0);

reg3to7write: process (clk,GNET2_GRESET_0)
begin
  reg15 <= versionstring;
  if (GNET2_GRESET_0 = '0') then
    reg4  <= (others => '0');
    reg5  <= (others => '0');
    reg6  <= (others => '0');
    reg7  <= (others => '0');
  elsif (clk'event and clk = '1') then
    reg15(def_prod_bit) <= select_default;
    if (probe_on='1') then
      reg4  <= debug_data(127 downto  96);
      reg5  <= debug_data( 95 downto  64);
      reg6  <= debug_data( 63 downto  32);
      reg7  <= debug_data( 31 downto   0);
    end if;
  end if;
end process;

reg3to7write: process (clk,GNET2_GRESET_0)
begin
  if (GNET2_GRESET_0 = '0') then
    reg8  <= (others => '0');
  elsif (clk'event and clk = '0') then
    if (probe_on='1') then
      reg8  <= reg_expo(11)&reg_expo(12)&"000000"&
               "0"&reg_data(131)&"0"&reg_data(65)&"0"&reg_data(130)&"0"&
               reg_data(64)&"000"&reg_expo(10)&"00"&reg_expo(9 downto 0);
    end if;
  end if;
end process;

reg8to14write: process (clk,GNET2_GRESET_0)
begin
  if (GNET2_GRESET_0 = '0') then
```

```
    reg3  <= (others => '0');
    reg9  <= (others => '0');
    reg12 <= (others => '0');
    reg13 <= (others => '0');
    reg14 <= (others => '0');
  elsif (clk'event and clk = '0') then
    if (probe_on='1') then
      reg3  <= debug_data(159 downto 128);
      reg9  <= reg_data(129 downto  98);
      reg12 <= reg_data( 97 downto  66);
      reg13 <= reg_data( 63 downto  32);
      reg14 <= reg_data( 31 downto   0);
    end if;
  end if;
end process;
```

```
  defaultprod: process (clk, GNET2_GRESET_0, reg2)
    variable sel : std_logic;
  begin
    if GNET2_GRESET_0 = '0' then
      select_default <= '0';
    elsif clk'event and clk = '1' then
        sel := select_default;
      if (regs_locked='1' and reg2(cmd_def_prod)='1') then
        select_default <= not sel;
      end if;
    end if;
  end process defaultprod;

end struct;
```

Literaturverzeichnis

[1] Alefeld, G. und J. Herzberger: *Introduction to Interval Computations*. Academic Press, 1983.

[2] American National Standards Institute, Institute of Electrical and Electronics Engineers: *IEEE Standard for Binary Floating-Point Arithmetic*, New York (1985). ANSI/IEEE Std 754-1985.

[3] American National Standards Institute, Institute of Electrical and Electronics Engineers: *IEEE Standard for Radix-Independent Floating-Point Arithmetic*, New York (1987). ANSI/IEEE Std 854-1987.

[4] Ashenden, Peter J.: *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers, Inc., 1996.

[5] Baumhof, Christoph: *Ein Vektorarithmetik-Koprozessor in VLSI-Technik zur Unterstützung des Wissenschaftlichen Rechnens*. Doktorarbeit, Universität Karlsruhe (TH), 1996.

[6] Chang, Kou-Chuan: *Digital Design and Modeling with VHDL and Synthesis*. IEEE Computer Society Press Los Alamitos, California, 1997.

[7] Chang, Kou-Chuan: *Digital systems design with VHDL and Synthesis*. IEEE Computer Society Press Los Alamitos, California, 1999.

[8] Cohen, Ben: *VHDL Answers to Frequently Asked Questions*. Kluwer Academic Publishers Boston, 1997.

[9] Cohen, Ben: *VHDL Coding Styles and Methodologies*. Kluwer Academic Publishers Boston, 1997.

[10] Electronic Industries Alliance: *Electronic Design Interchange Format*, 1988. http://www.edif.org.

[11] Facius, Axel: *Iterative Solution of Linear Systems with Improved Arithmetic and Result Verification*. Doktorarbeit, Universität Karlsruhe (TH), July 2000. `www.uni-karlsruhe.de/~Axel.Facius` → Publications → Books → diss.

[12] FZI Embeddes System Design Group (FZI-ESDG): *Spyder-Virtex-X2 Board*, 1999. http://www.fzi.de/sim/people/weiss/sites/spyder.htm.

[13] Hammer, R., M. Hocks, U. Kulisch und D. Ratz: *Toolbox for Verified Computing.* Springer, Berlin Heidelberg, 1993.

[14] Hammer, R., M. Hocks, U. Kulisch und D. Ratz: *C++ Toolbox for Verified Computing.* Springer, Berlin Heidelberg, 1995.

[15] Kirchner, R. und U. Kulisch: *Arithmetic for Vector Processors.* In: *ARITH*, Band 8, Seiten 256 – 269. IEEE, 1987.

[16] Kirchner, R. und U. Kulisch: *Accurate Arithmetic for Vector Processors.* Journal of Parallel and Distributed Computing, 5:250–270, 1988.

[17] Klatte, R., U. Kulisch, A. Wiethoff, C. Lawo und M. Rauch: *C-XSC.* Springer Verlag, 1992.

[18] Knüppel, O.: *BIAS — basic interval arithmetic subroutines.* Technischer Bericht, Universität Hamburg-Harburg, Hamburg, Germany, 1993.

[19] Knüppel, O.: *PROFIL — Programmer's Runtime Optimized Fast Interval Library.* Technischer Bericht, Universität Hamburg-Harburg, Hamburg, Germany, 1993.

[20] Kulisch, U. (Herausgeber): *Wissenschaftliches Rechnen mit Ergebnisverifikation – Eine Einführung.* Akademie Verlag, Ost-Berlin, Vieweg, Wiesbaden, 1989.

[21] Kulisch, U.: *Advanced arithmetic for digital computer design of arithmetic units.* Electronic Notes in Theoretical Computer Science, 24:1–72, 1999. http://www.elsevier.nl/locate/entcs/volume24.html.

[22] Kulisch, U. und W. L. Miranker (Herausgeber): *A New Approach to Scientific Computation.* Academic Press, New York, 1983.

[23] Kulisch, Ulrich: *Grundlagen des Numerischen Rechnens.* Bibliographisches Institut Mannheim, Bibliographisches Institut Mannheim1976.

[24] Lehmann, Gunther, Bernhard Wunder und Manfred Selz: *Schaltungsdesign mit VHDL.* Franzis'-Verlag, Poing, 1994. http://www-itiv.etec.uni-karlsruhe.de/.

[25] Lohner, R.: *Interval Arithmetic in staggered correction format.* Scientific Computing with Automatic Result Verification, 8:301–321, 1993.

[26] PLX Technology, Inc.: *32 Bit PCI Master Chip*, 1999. http://www.plxtech.com.

[27] Rump, S. M.: *INTLAB — Interval Laboratory.* Technischer Bericht, Universität Hamburg-Harburg, 1998.

[28] Smith, Douglas J.: *HDL Chip Design.* Doone Publications, Madison, AL, USA, 1999.

[29] *Sun FORTE tools*. `www.sun.com/forte/developer/`.

[30] Wannemacher, Markus: *Das FPGA-Kochbuch*. International Thomson Publishing GmbH, Bonn, 1998.

[31] Xilinx: *Homepage der Firma Xilinx*, 2001. www.xilinx.com/.

[32] Xilinx: *Virtex 2.5 V Field Programmable Gate Arrays Data Sheet*, April 2001. www.xilinx.com/partinfo/ds003.pdf.

Lebenslauf

Name		Norbert Erich Karl Bierlox
Anschrift		Winterstrasse 30, 76137 Karlsruhe Tel: (07 21) 38 69 32 (privat) Tel: (07 21) 96 44 8 - 143 (Arbeit) email: Norbert.Bierlox@ieee.org
Geburtsdatum/-ort		18. Februar 1958 in Neuenburg am Rhein
Nationalität		deutsch
Familienstand		ledig
Eltern		Karl Bierlox und Edith Bierlox geb. Bartsch
Schulausbildung	1965 - 1968	Grundschule in Müllheim Baden
	1968 - 1974	Realschule in Müllheim Baden
	15.6.1974	Mittlere Reife
	1974 - 1978	Gewerbeschule II in Freiburg im Breisgau
Berufstätigkeit	9/74 - 6/76	Lehre zum Elektroanlageninstallateur bei der Firma Siemens AG in Freiburg im Breisgau
	7/76 - 1/78	Lehre zum Energieanlagenelektroniker bei der Firma Siemens AG in Freiburg im Breisgau
	2/78 - 8/79	Energieanlagenelektroniker bei der Firma Siemens AG in Freiburg im Breisgau
Wehrdienst	4/78 - 6/79	Elektronik-Bildgeräte-Mechaniker beim Aufklärungsgeschwader 51 in Eschbach Baden (15 monatiger Pflichtwehrdienst)
Schulausbildung	1979 - 1981	Technische Oberschule, Freiburg im Breisgau
	22.6.1981	Fachgebundene Hochschulreife
Berufstätigkeit	9/81 - 4/83	Elektriker bei der Firma Naef AG, Basel
	5/83 - 9/83	Elektriker bei der Firma Elektro-Schuler, Freiburg im Breisgau
Studium	1983 - 1996	Studium der Mathematik an der Universität Karlsruhe mit dem Nebenfach Informatik
	1989 - 1996	Wissenschaftliche Hilfskraft beim Modellversuch Informatik für Blinde (1995 umbenannt in Studienzentrum für Sehgeschädigte) an der Universität Karlsruhe
	1993-1994	Diplomarbeit am Institut für Angewandte Mathematik der Universität Karlsruhe mit dem Thema: „Parallelisierung von Algorithmen in Pascal-XSC unter Einsatz von PVM3“
Studienabschluß	27.9.1996	Diplom-Mathematiker
Promotion	10/96 - 7/01	Wissenschaftlicher Mitarbeiter am Institut für Angewandte Mathematik bei Prof. Kulisch, Arbeit an der Dissertation mit dem Thema: „Ein VHDL Koprozessorkern für das exakte Skalarprodukt“
Berufstätigkeit	seit 8/01	Professional Services bei fun communications GmbH, Karlsruhe