INAUGURAL - DISSERTATION
zur
Erlangung der Doktorwürde
der
Naturwissenschaftlich - Mathematischen
Gesamtfakultät
der Ruprecht - Karls - Universität
Heidelberg

vorgelegt von
Dipl.-Phys. Gerhard Lienhart
aus Bühl

Tag der mündl. Prüfung: 19. Juli 2004

Beschleunigung Hydrodynamischer Astrophysikalischer Simulationen mit FPGA-Basierten Rekonfigurierbaren Koprozessoren

Gutachter: Prof. Dr. Reinhard Männer
Prof. Dr. Volker Lindenstruth

Beschleunigung Hydrodynamischer Astrophysikalischer Simulationen mit FPGA-Basierten Rekonfigurierbaren Koprozessoren

Zusammenfassung

Diese Dissertation befasst sich mit der Anwendung rekonfigurierbarer Koprozessoren zur Beschleunigung astrophysikalischer Simulationsalgorithmen, ausgehend von einer hybriden Plattform aus Standardrechner und einem Rechenbeschleuniger für die Gravitationssimulation (GRAPE). Für Simulationen, die eine Berücksichtigung der Hydrodynamik erforderlich machen, schränkt die dazu eingesetzte Simulationsmethode Smoothed Particle Hydrodynamics (SPH) die erzielbare Rechenleistung des Gesamtsystems stark ein. Es wurde der Ansatz verfolgt, durch den Einsatz einer FPGA-basierten Koprozessorplattform das SPH-Verfahren zu beschleunigen. Analysen der Simulationscodes ergaben, dass die SPH-Berechnungen unter Verwendung von Gleitkommazahlen mit 16 Mantissenbits ausreichend genau sind. Um den Ansatz zu realisieren, wurde ein FPGA-Koprozessor in Form einer PCI-Einsteckkarte verwendet, ausgestattet mit einem modernen Virtex-II-3000-FPGA von Xilinx. Es wurden FPGA-Designs entwickelt, welche für die umfangreichen aber einfach strukturierten SPH-Berechnungen bei ausreichend hoher Rechengenauigkeit eine Rechenleistung von über 3 GFlops erreichen. Dazu wurde eine Bibliothek arithmetischer Module für die rekonfigurierbare Logik entwickelt. Alle Module sind bezüglich der Rechengenauigkeit parametrisiert, und es wurden für verschiedene numerische Randbedingungen spezialisierte Operatoren entwickelt. Damit konnten optimal an die Problemstellung angepasste Rechenwerke in Form einer Pipeline aufgebaut werden. Für die SPH-Pipelines konnten 50-60 Gleitkommaoperationen unter Aufwendung von etwa 50 % der FPGA-Ressourcen implementiert werden, mit einer resultierenden Geschwindigkeit von 66 MHz. Die Schaltungen sind in der Lage, die Berechnungen synchron zur maximalen Datenrate von Speicher und PCI-Interface durchführen. Um das Beschleunigungspotential (etwa Faktor 10) effektiv auszuschöpfen, wird eine tiefgehende Umstrukturierung des Simulationsalgorithmus erforderlich, was Gegenstand der weiteren Forschung sein wird.

Acceleration of Astrophysical Hydrodynamics Simulations with FPGA-Based Reconfigurable Coprocessors

Abstract

This dissertation deals with the application of reconfigurable coprocessors for accelerating astrophysical simulation systems, where a simulation system consisting of a host workstation and an accelerator platform for the gravity part of the simulation (i.e. GRAPE) was presumed as given. For simulation algorithms which also deal with hydrodynamics, it shows, that the method of smoothed particles hydrodynamics (SPH) which is usually used for this purpose causes a bottleneck for the overall performance of the system. Therefore the approach was chosen to accelerate the time-critical calculation steps of SPH by an FPGA-based coprocessor. By studying the astrophysical code it was shown, that calculations based on floating-point numbers with 16 mantissa bits lead to a sufficient precision for SPH. To realize the accelerator a PCI-based FPGA-coprocessor featuring a modern Xilinx Virtex-II-3000 FPGA was used. FPGA designs were developed which are able to deal with the extensive but simple structured SPH calculations at sufficient precision. With these designs a performance of more than 3 GFlops has been achieved. For the arithmetics a library of modules parameterized in precision and specialized for different numerical situations has been developed. With this library it was possible to synthesize calculation units as a pipeline, optimally matched for the problem. The SPH pipelines, consisting of 50 to 60 operators, were successfully implemented in about 50 % of the FPGA resources with a resulting speed of 66 MHz. The design is able to perform the calculations synchronously with the data rate of the PCI bus and memory. A Speedup of 10 for SPH seems within reach, but for efficient utilization of the calculation power a deep re-design of the simulation algorithm will be necessary which is subject to further research.

Inhaltsverzeichnis

1 Einführung **1**

1.1 Problemstellung . . . 1

1.2 Aufbau der Arbeit . . . 2

2 Astrophysikalische Simulationsverfahren **3**

2.1 Allgemeine Simulationsmethodik . . . 3

2.1.1 Numerische Simulation der zeitlichen Entwicklung . . . 4

2.1.2 Klassifizierung der Dynamik astrophysikalischer Systeme . . . 4

2.2 Simulation der Gravitation . . . 5

2.2.1 Direkte N-Körper-Simulation . . . 5

2.2.2 Baumbasierte Verfahren . . . 6

2.2.3 Gitterbasierte Verfahren . . . 8

2.3 Smoothed Particle Hydrodynamics . . . 9

2.3.1 Allgemeine Methode von SPH . . . 9

2.3.2 Weitverbreitete spezielle SPH-Formulierung . . . 12

2.3.3 Variationen des SPH-Verfahrens . . . 18

2.3.4 Einbettung in Simulation der Gravitation . . . 19

2.4 Simulationsplattformen . . . 20

2.4.1 Hochleistungsrechner . . . 20

2.4.2 Spezialrechner . . . 21

3 Rekonfigurierbare Rechnerplattformen **27**

3.1 Rekonfigurierbare Logikbausteine . . . 27

3.1.1 Allgemeine Architektur von FPGAs . . . 27

3.1.2 Details zur verwendeten FPGA-Serie Virtex-II . . . 29

3.1.3 Programmierung von FPGAs . . . 36

3.2 Allgemeine Systemarchitektur . . . 37

3.2.1 Rekonfigurierbare Rechensysteme . . . 38

3.2.2 Rekonfigurierbare Koprozessoren . . . 38

3.3 Verwendete rekonfigurierbare Plattform . . . 38

4 Computerarithmetik **41**
4.1 Ganzzahlen . . . 42
4.1.1 Darstellungsmöglichkeiten ganzer Zahlen . . . 42
4.1.2 Additionsalgorithmen . . . 45
4.1.3 Multiplikationsverfahren . . . 51
4.1.4 Divisionsalgorithmen . . . 53
4.1.5 Quadratwurzel . . . 61
4.2 Festkommazahlen . . . 66
4.3 Gleitkommazahlen . . . 66
4.3.1 Darstellung von Gleitkommazahlen . . . 66
4.3.2 Genauigkeit von Gleitkommaoperationen . . . 67
4.3.3 Operationen auf Gleitkommazahlen . . . 70
4.4 Logarithmische Zahlen . . . 79
4.4.1 Darstellung im logarithmischen Zahlensystem . . . 79
4.4.2 Operationen auf logarithmischen Zahlen . . . 80
4.4.3 Semilogarithmische Zahlen . . . 81
4.5 Berechnung von Funktionen . . . 82
4.5.1 Table-Look-Up-basierte Methoden . . . 82
4.5.2 Digit-Recurrence-Algorithmen . . . 85
4.5.3 Iterative Näherungsverfahren . . . 86
4.5.4 Kombination verschiedener Verfahren . . . 88
4.6 Stand der Forschung zu Implementierungen auf FPGAs . . . 89

5 Anforderungen an eine Beschleunigerarchitektur **91**
5.1 Rechenleistung . . . 91
5.2 Kommunikationsbandbreite . . . 94
5.3 Erforderliche Rechengenauigkeit von SPH . . . 96
5.3.1 Testsimulationen mit künstlicher Reduktion der Genauigkeit . . . 96
5.3.2 Simulation der Rechenwerke . . . 97
5.4 Folgerungen für die Systemarchitektur . . . 105
5.4.1 Grundlegende Struktur der Rechenwerke . . . 107
5.4.2 Datenfluss . . . 107
5.4.3 Geeignetes Zahlenformat für die Rechenwerke . . . 109

6 Implementierung der Arithmetik **111**
6.1 Festkommaarithmetik auf FPGAs . . . 111
6.1.1 Addition und Subtraktion . . . 111
6.1.2 Multiplikation . . . 113
6.1.3 Division . . . 118
6.1.4 Quadratwurzel . . . 124
6.2 Gleitkommaarithmetik auf FPGAs . . . 127
6.2.1 Ausnahmebehandlung . . . 127
6.2.2 Addition . . . 128

6.2.3 Multiplikation 150
6.2.4 Division 156
6.2.5 Quadratwurzel 160
6.2.6 Zusammenfassung der Implementierungsergebnisse 164

7 Implementierung des Rechenbeschleunigers 169
7.1 Implementierung der Rechenwerke für SPH 169
7.1.1 Spline-Kernel 169
7.1.2 Designmethode für die Rechenwerke 176
7.1.3 Dichteberechnung 177
7.1.4 Kraftberechnung 179
7.2 Einbettung in den FPGA-Prozessor 182
7.3 Datenfluss und Steuerung 184
7.4 Ergebnisse für den Rechenbeschleuniger 185

8 Zusammenfassung und Diskussion 187

Kapitel 1

Einführung

1.1 Problemstellung

Durch die Entwicklung leistungsfähiger Rechnerplattformen in den letzten Jahren wurde das wissenschaftliche Rechnen zu einem wichtigen Standbein der Forschung. Insbesondere die Simulation komplexer physikalischer Systeme vermag eine Brücke zwischen theoretischen Modellen und experimenteller Observation zu schlagen. Für viele Systeme korreliert die Qualität der Aussagen direkt mit der zur Verfügung stehenden Rechenkraft und so gibt es einen nicht zu sättigenden Bedarf an Rechenleistung in den Naturwissenschaften. Insbesondere im Bereich der Astrophysik ist die Rechenleistung das entscheidende Kriterium, ob eine physikalische Fragestellung gelöst werden kann oder nicht. Die observierten Phänomene des Weltalls sind beliebig komplex, selbst wenn die diese Phänomene beschreibenden Gleichungen der Dynamik einfach sind, wie z.B. im Fall der Gravitation. Bereits die Dynamik von Kugelsternhaufen aus wenigen tausend Sternen wirft ungelöste Fragen auf, deren Beantwortung entscheidend für das Verständnis der Entstehung unserer Welt sein kann. Die Entwicklung der Rechenleistung der Mainstream-Technologie ist einigermaßen vorhersehbar und folgt weitgehend dem Moorschen Gesetz des Fortschritts der Transistorendichte. Damit ist jedoch auch vorhersehbar, dass die allgemeine Rechnerentwicklung für viele numerische Fragestellungen in absehbarer Zeit nicht die nötige Rechenleistung erbringen kann. Daraus motiviert sich die große Nachfrage nach Spezialrechnerplattformen, die mit aktueller Technologie eine weit höhere Leistung erreichen können als Standardsysteme.

Diese Arbeit befasst sich mit dem Ansatz, rekonfigurierbare Plattformen für numerische Simulationsaufgaben im Bereich Astrophysik einzusetzen. Der Ansatz wird für eine spezielle Klasse von Algorithmen ausgeführt - die Simulation von Systemen, für welche Nahwechselwirkungen einen entscheidenden Einfluss auf die Dynamik haben. Darunter fallen insbesondere die hydrodynamischen Phänomene, wie sie für eine Vielzahl von astrophysikalischen Problemstellungen berücksichtigt werden müssen. Dazu gehören beispielsweise Fragen zur Entstehung der Strukturen im Kosmos, zur Dynamik von Galaxien oder zur Sternentstehung. Die Konzentration auf diese Phänomene ist deshalb interessant, da hier der Einsatz von Rechenbeschleunigern sehr vielversprechend ist, denn die Behandlung der Gasdynamik ist derzeit die rechenzeitkritische Komponente, wenn die Berechnung der Gravitationskräfte durch bereits existierende Spezial-

rechnerplattformen beschleunigt wird. Solche Systeme werden in Abschnitt 2.4.2 vorgestellt. Für diese Arbeit wird davon ausgegangen, dass eine solche Plattform vorliegt. Ein aktueller Rechenbescheuniger (GRAPE-6) erreicht 1 TFlop Spitzenrechenleistung in einem Gehäuse der Größe einer Workstation. Damit reduziert sich die Rechenzeit für die Gravitationswechselwirkung auf einen Bruchteil der Gesamtrechenzeit. Wird auch der hydrodynamische Anteil der Simulationen durch eine zweite Spezialrechnerarchitektur beschleunigt, lässt sich mit relativ geringem Hardwareaufwand ein hoher Speedup erreichen, wie im Verlaufe dieser Arbeit gezeigt wird. Die Berechnungen der Hydrodynamik sind wesentlich komplexer als die der Gravitation und zudem in der genauen Formulierung applikationsabhängig. Die Struktur der Berechnungen ist jedoch sehr einfach. In dieser Arbeit wird gezeigt, dass aktuelle rekonfigurierbare Rechensysteme für hydrodynamische Berechnungen geeignet sind und eine im Vergleich zu Standardprozessoren sehr hohe Rechenleistung erreichen.

1.2 Aufbau der Arbeit

In Kapitel 2 werden die wissenschaftlichen und algorithmischen Grundlagen der wichtigsten astrophysikalischen Simulationsverfahren vorgestellt. Dabei werden die Vorzüge und Einschränkungen der verschiedenen Herangehensweisen motiviert und gegen die dieser Arbeit zugrunde liegenden Algorithmen abgegrenzt. Es wird in diesem Kapitel auch eine kurze Übersicht über die bisher eingesetzten Rechensysteme gegeben.

Das Kapitel 3 wird in die hardwareseitigen Grundlagen dieser Arbeit einführen. Da sich der Hauptteil dieser Arbeit mit der Implementierung von Algorithmen auf FPGA-basierten Systemen beschäftigt, wird hier detailliert auf die technologischen Aspekte solcher Systeme eingegangen. Insbesondere wird das in dieser Arbeit verwendete System beschrieben.

In Kapitel 4 werden die für den Implementierungsteil der Arbeit benötigten Grundlagen der Computerarithmetik eingeführt. Es werden die elementaren Zahlensysteme vorgestellt und deren Eigenschaften in Bezug auf die Umsetzung von Rechenwerken beleuchtet.

Eine Analyse der für eine Implementierung maßgeblichen Randbedingungen an die Rechenleistung und Rechengenauigkeit sowie der sich aus den Algorithmen ergebenden Strukturen und Datenflüsse wird in Kapitel 5 diskutiert.

Kapitel 6 und 7 beschäftigen sich mit der Implementierung der Simulationsalgorithmen auf rekonfigurierbaren Systemen. Den Hauptteil bildet die Diskussion zur Implementierung der Arithmetik. Für einen modernen Algorithmus wird die Implementierung auf der vorhandenen Plattform beschrieben und analysiert.

In Kapitel 8 werden die Ergebnisse der Arbeit zusammengefasst und diskutiert und es wird eine Ausblick auf die zukünftigen Schritte und Möglichkeiten des in dieser Arbeit verfolgten Ansatzes gegeben.

Kapitel 2

Grundlagen Astrophysikalischer Simulationen

In diesem Kapitel werden die grundlegenden Konzepte der Simulation astrophysikalischer Systeme vorgestellt. Es kann hier kein vollständiger Abriss aller Varianten der Simulationsmethodik gegeben werden. Vielmehr sollen die gebräuchlichen Konzepte motiviert werden, um den Rahmen der speziellen Formulierung, mit der sich der Implementierungsteil der Arbeit befasst, abzustecken.

2.1 Allgemeine Simulationsmethodik

Ziel astrophysikalischer Simulationen ist es, das dynamische Verhalten von astrophysikalischen Systemen numerisch zu studieren. Dies motiviert sich aus dem fehlenden experimentellen Zugang zur zeitlichen Entwicklung von astrophysikalischen Systemen, vor allem aufgrund der extremen Zeitskalen der Evolution solcher Systeme, aber auch aus der Unzugänglichkeit von inneren Größen beobachtbarer Systeme. Andererseits können die sich aus den theoretischen Modellen für diese Systeme ergebenden komplexen dynamischen Prozesse in den meisten Fällen analytisch nicht gelöst werden. Deshalb ist die numerische Simulation das wichtigste Bindeglied zwischen Observation und Theorie.

Die Behandlung der Gravitation ist für die Modellierung astrophysikalischer Systeme die wichtigste Komponente. Die Gravitation ist eine Fernwechselwirkung, denn sie wird mit dem Abstand r der Objekte quadratisch schwächer – die Oberfläche einer Kugelschale mit Radius r wächst jedoch mit r^2 und somit auch die Anzahl an Objekten, die sich ungefähr in einer Entfernung r befinden können. Dies hat zur Folge, dass die Gravitationskräfte von weit entfernten Objekten nicht vernachlässigt werden können. Für die Kraftberechnung eines jeden simulierten Massepunktes muss also die Masseverteilung des kompletten simulierten Systems herangezogen werden. Abhängig von der Art der zu simulierenden Systeme (siehe Abschnitt 2.1.2) gibt es eine Vielzahl verschiedener Algorithmen. Ein Überblick dazu folgt in Abschnitt 2.2. Die Simulation zusätzlicher Physik, insbesondere auch die in dieser Arbeit im Mittelpunkt stehende Hydrodynamik, ist eng an die durch die Gravitationssimulation vorgegebene Codearchitektur gekoppelt, da

alle Komponenten des Systems der Gravitationskraft unterworfen sind. Dies ist der Grund dafür, dass in dieser Arbeit so detailliert auf die Simulation der Gravitation eingegangen wird.

Im Folgenden werden einige grundlegende Aspekte astrophysikalischer Simulationen diskutiert.

2.1.1 Numerische Simulation der zeitlichen Entwicklung

Im Allgemeinen wird eine numerische Simulation zeitlich diskretisiert, indem Zeitschritte definiert werden. Für jeden Zeitschritt t_i werden die Wechselwirkungen der Systembestandteile ermittelt und daraus über einen so genannten *Integrator* die Veränderungen der Zustandsgrößen des Systems gegenüber dem Zeitschritt t_{i-1} berechnet. Beispielsweise sind für ein System aus Punktmassen die Zustandsgrößen gegeben durch die Teilchenpositionen $\vec{r}_i$, die Geschwindigkeiten $\vec{v}_i$ und die Massen m_i. Die Wechselwirkungen sind hier die Gravitationskräfte, die in jedem Zeitschritt t_i ermittelt werden. Aufgrund der berechneten Kräfte ermittelt der Integrator die neuen Positionen und Geschwindigkeiten für den Zeitpunkt t_{i+1}. Dazu werden die newtonschen Bewegungsgleichungen integriert.

Die in der Astrophysik verwendeten Integratoren unterscheiden sich im Wesentlichen in der Ordnung der Propagation von Zustandsvariablen und den dazu erforderlichen Wechselwirkungstermen, die gegebenenfalls in höherer Ordnung zu berechnen sind. Die Verfahren legen insbesondere auch die Anzahl der Zeitschritte $(t_i, t_{i-1} \ldots)$ fest, die in die Propagation der Zustandsvariablen auf t_{i+1} einbezogen werden. Weitverbreitete Verfahren sind beispielsweise Runge-Kutta-Verfahren verschiedener Ordnung, Leap-Frog und Hermite-Algorithmen (siehe z.B. [107]). Die Auswahl des Integrationsschemas hängt ab von der erwarteten Dynamik der Anwendung und der Art der Berechnung der Wechselwirkungsterme.

Eine wichtige Optimierungsmethode, um Wechselwirkungsberechnungen einzusparen, besteht darin, variable Zeitschritte einzusetzen. Entsprechend eines Kriteriums zur Dynamik einer Variablen können die Zeitschritte, für welche die entsprechenden Wechselwirkungsterme berechnet werden, angepasst werden. Dies hat zur Folge, dass die Bestandteile des Systems mit hoher Aktivität öfter neu berechnet werden als die Teile mit geringer Dynamik.

2.1.2 Klassifizierung der Dynamik astrophysikalischer Systeme

Es soll im Folgenden davon ausgegangen werden, dass die gravitativ wechselwirkenden Systembestandteile durch Massepunkte dargestellt werden (Teilchen). Astrophysikalische Systeme werden grundsätzlich danach unterschieden, ob sie als stoßfrei betrachtet werden können oder nicht. Stöße bedeuten in diesem Zusammenhang, dass sich Elemente des Systems so nahe kommen, dass eine starke Änderung der Impulse aufgrund der Wechselwirkung zwischen zwei oder mehreren Elementen resultiert. Ein Maß dafür, ob ein stoßfreies oder stoßdominiertes System vorliegt, ist der Verleich der Relaxationszeit t_{relax} mit der so genannten Crossing-Time t_{cross}. Letztere beschreibt die Zeitspanne für ein Teilchen mit einer typischen Geschwindgkeit v_t, sich einmal durch das System zu bewegen. Für ein kugelförmiges System aus N Teilchen mit Radius R und Gesamtmasse $N \cdot m$ kann über das Virialtheorem folgende Crossing-Time angesetzt werden (sie-

he z.B. [107]):

$$t_{cross} = \frac{R}{v_t} \approx \sqrt{\frac{R^3}{G\,N\,m}}, \tag{2.1}$$

mit der Gravitationskonstante G.

Die Relaxationszeit ergibt sich aus dem Zeitraum, in dem sich durch die Wechselwirkung mit dem System senkrecht zur Geschwindigkeit eines Teilchens im Mittel eine Änderung in der Größenordnung von v_t ergibt. Es lässt sich für die Relaxationszeit folgende Beziehung finden (siehe [14], vergleiche auch z.B. mit [1]):

$$t_{relax} = t_{cross}\frac{N}{8\ln N}. \tag{2.2}$$

Als stoßfrei können nun solche Systeme angesehen werden, für die $t_{cross} \ll t_{relax}$ gilt. In diesen Fällen sieht ein Teilchen nur ein geglättetes Gravitationspotential der anderen Teilchen. Beispiele dafür sind Galaxien und kosmologische Simulationen ohne starke Cluster-Bildung. Gilt dagegen $t_{cross} \geq t_{relax}$ müssen die Wechselwirkungen zwischen einzelnen Teilchen exakt simuliert werden. Dies ist beispielsweise für Kugelsternhaufen der Fall.

2.2 Algorithmen zur Simulation der Gravitationswechselwirkung

In diesem Unterkapitel werden nun die grundlegenden Simulationsmethoden zur Gravitation vorgestellt. Die verschiedenen Methoden haben gemeinsam, dass die betrachteten Systeme als diskrete Verteilung von Massepunkten angenommen werden. Dabei kann es mit Ausnahme von sehr kleinen Systemen keine Eins-zu-Eins-Zuordnung von Sternen zu Massepunkten geben. Noch weniger können die Moleküle oder Staubpartikel des interstellaren Mediums im Einzelnen simuliert werden. Deshalb werden ganze Regionen von Sternen bzw. dunkler Materie zu einem Massepunkt zusammengefasst. Die Masse dieser Pseudoteilchen beträgt viele Sonnenmassen (typischerweise 10^6 – 10^8 $M_\odot$).

2.2.1 Direkte N-Körper-Simulation

Die naheliegendste Variante der Gravitationskraftberechnung auf ein Teilchen i mit Position $\vec{r}_i$ und Masse m_i besteht darin, die Beiträge aller anderen Punktmassen m_j des Systems aufzusummieren. Diese Methode wird Direct Summation oder auch PP (**P**article-**P**article-Methode) genannt und führt auf folgende Gleichung:

$$\vec{F}_{grav,i} = -Gm_i \sum_{j \neq i} \frac{m_j \left(\vec{r}_i - \vec{r}_j\right)}{\left(\varepsilon^2 + |\vec{r}_i - \vec{r}_j|^2\right)^{\frac{3}{2}}}, \tag{2.3}$$

wobei G die Gravitationskonstante ist. Die Gleichung unterscheidet sich von der bekannten Formel für die Gravitation zwischen Punktmassen um den Term ε^2. Dieser zusätzliche Term vermeidet unphysikalisch hohe Kräfte, wenn sich Pseudoteilchen, welche viele Sternenmassen als Massepunkt repräsentieren, sehr nahe kommen.

Um die wechselseitigen Kräfte in einem System aus N Punktmassen zu berechnen, müssen $\frac{N}{2}(N-1)$ Paarwechselwirkungen berechnet werden. Wie weiter unten vorgestellt wird, gibt es weit schnellere Methoden. Bei Systemen, wo eine sehr hohe Genauigkeit der Kraftberechnung notwendig ist, wie bei der Betrachtung nicht-kollisionsfreier Systeme, zum Beispiel bei der Dynamik von Sternhaufen, wird PP nach wie vor eingesetzt (siehe z.B. [107], [118]). Seit einigen Jahren wird diese Methode wieder vermehrt eingesetzt, da extrem leistungsfähige Spezialrechner verfügbar wurden. In Abschnitt 2.4.2 wird beschrieben, wie mit der PP-Methode über den Einsatz der GRAPE (**GRA**vity **P**ip**E**) genannten Systeme ein enormes Leistungspotential erreicht werden kann. Eine Optimierung von PP folgt aus der Beobachtung, dass bei vielen astrophysikalischen Konstellationen extreme Kontraste der Masseverteilungen auftreten und entsprechend Regionen unterschiedlicher Dynamik existieren. Die Idee ist dabei, die Simulationen mit unterschiedlichen Zeitschritten durchzuführen, wie es bereits in Abschnitt 2.1 erwähnt wurde. Für Teilchen in Regionen hoher Dynamik werden die Wechselwirkungsterme öfter neu berechnet als in Regionen niedriger Dynamik. Dies kann in Situationen mit hohen strukturellen Kontrasten zu einer drastischen Reduktion des Rechenaufwandes führen. In [73] wurde demonstriert, wie dies zu einer Reduktion des Rechenaufwandes $\propto N$ führen kann.

2.2.2 Baumbasierte Verfahren

In Simulationsanwendungen, bei denen die Gravitation leicht fehlerbehaftet berechnet werden darf, sind baumbasierte Verfahren (Tree-Algorithmen) eine weitverbreitete Methode, den Rechenaufwand stark zu reduzieren. Das Prinzip besteht darin, die Kräfte auf weit entfernte Teilchen darüber zu berechnen, dass eine ganze Gruppe von Teilchen als ein einzelnes Wechselwirkungsteilchen betrachtet wird, welches im Schwerpunkt dieser Gruppe liegt und durch Multipolmomente geringer Ordnung beschrieben wird. Dies jedoch nur, wenn ein Akzeptanzkriterium (**M**ultipole **A**cceptance **C**riterion = MAC) für die Teilchengruppe erfüllt ist. Dieses Kriterium begrenzt den Fehler, der durch die Vernachlässigung höherer Multipolmomente entsteht. Ein Beispiel für ein solches Kriterium ist der Öffnungswinkel, unter dem diese Teilchengruppe erscheint. Ist der Winkel geringer als ein Schwellenwert, wird die Wechselwirkung anhand der Multipolmomente der Teilchengruppe berechnet, ansonsten muss die Teilchengruppe in Untergruppen aufgelöst werden. Die Wahl des Akzeptanzkriteriums ist kritisch für die Genauigkeit der Simulation, siehe z.B. [99]. Die Erzeugung der Interaktionslisten, welche sowohl aus Teilchen als auch aus den Repräsentanten von Teilchengruppen bestehen, geschieht über eine Baumstruktur, in der die Teilchen über ein Abstandskriterium oder durch eine rekursive Aufteilung des Raumes angeordnet werden. Mit zunehmender Entfernung von dem Teilchen, für das die Gravitationskraft berechnet werden soll, steigt die Anzahl der Zweige des Baumes, die als Wechselwirkungspartner gesehen werden. Dies führt dazu, dass die baumbasierten Algorithmen mit $O(N\ logN)$ in der Rechenzeit für die Kräfte skalieren. Der absolute Rechenaufwand hängt stark von der Rechengenauigkeit ab, die über das MAC gesteuert werden kann, (siehe auch [99]). Angewendet werden baumbasierte Verfahren beispielsweise für die Simulation von Galaxiendynamik und Sternentstehung. Für Simulationen mit sehr hohen Präzisionsanforderungen sind Tree-Algorithmen weniger geeignet, da mit sehr kleinem Öffnungswinkel die Kosten höher als bei Direct Summation werden können (z.B. bei Sternhaufen).

Im Folgenden werden die wichtigsten Tree-Algorithmen kurz vorgestellt werden.

Barnes-Hut-Tree-Algorithmus Diesem von Barnes und Hut entwickelten Top-Down-Verfahren [5] liegt für dreidimensionale Simulationen ein Oct-Tree zugrunde. Dies bedeutet, dass von jedem Knoten des Baumes acht Zweige ausgehen. Jeder Knoten entspricht einer kubischen Zelle des Raumes. Das Verfahren führt eine hierarchische Unterteilung des Raumes durch, indem – ausgehend von einer den ganzen Raum umgebenden Zelle – die Zellen jeweils in 8 kleinere Kuben unterteilt werden. Die Unterteilung wird so weit fortgesetzt, bis in jeder Zelle höchstens noch ein einziges Teilchen liegt. Für alle Knoten werden die Gesamtmasse, der Schwerpunkt und gegebenenfalls höhere Multipolmomente berechnet.

Die Interaktionsliste für die Kraftberechnung auf Teilchen i wird durch Absteigen in den Baum ermittelt. Wenn eine angetroffene Zelle weit genug von $\vec{r}_i$ entfernt ist, also z.B. das Öffnungswinkelkriterium erfüllt ist, wird diese Zelle in die Interaktionsliste aufgenommen, ansonsten wird eine Ebene tiefer in die Subzellen abgestiegen.

Die Methode, den Raum rekursiv geometrisch zu unterteilen führt zu einer einfachen Parallelisierungsstrategie, wenn global eine gleichmäßige Masseverteilung vorliegt wie bei kosmologischen Simulationen. Bei starker Konzentration der Verteilung auf unregelmäßige Strukturen wird eine Parallelverarbeitung jedoch sehr ineffizient. Überdies repräsentiert die Baumstruktur in solchen Fällen die Massestrukturen nur schlecht. Für solche Situationen eignet sich die nachfolgend vorgestellte Baumstruktur besser.

Zu modernen Implementierungen paralleler Codes basierend auf den Barnes-Hut-Trees siehe z.B. [98], [24] und [123].

Press Tree Es handelt sich hierbei um einen Binärbaum, welcher Bottom-Up aufgebaut wird. Unter den Teilchen wird das Paar wechselseitig nächster Nachbarn gesucht und durch einen Knoten abgebildet, wobei die beiden Teilchen die Zweige des Knotens bilden. Dann wird unter den verbliebenen ungepaarten Teilchen und den Knoten wieder das Paar wechselseitig nächster Nachbarn gesucht und erneut durch einen Knoten und zwei Zweige ersetzt. Dieser Prozess wird so lange wiederholt, bis nur noch ein Knoten, die Wurzel des Baumes, übrig ist. Das Verfahren hat den Vorteil, dass sich die räumliche Gruppierung der Teilchen in der Baumstruktur wiederfindet und passt sich deshalb gut an die physikalischen Strukturen einer Verteilung an. Wie beim Barnes-Hut-Algorithmus bekommen die Knoten die Summe der Masse, den Schwerpunkt und eventuell höhere Multipolmomente zugewiesen und die Erzeugung der Interaktionslisten kann auf die gleiche Weise geschehen. Zu modernen Implementierungen basierend auf Binärbäumen siehe z.B. [111] und [126]. Ein moderner Code mit Binärbaum, der auf einer rein geometrischen Raumaufteilung basiert, wird in [121] vorgestellt.

Fast Multipole Method (FMM) Dies ist ein Tree-Code, dessen Baumstruktur dem Barnes-Hut-Tree entspricht. Die Baumstruktur wird jedoch anstatt zur Berechnung der Kräfte zur Bestimmung des Gravitationspotentials für beliebige Positionen benutzt. Dazu werden auch Multipolmomente höherer Ordnung berücksichtigt und es wird zudem eine lokale Entwicklung des Feldes für die Knoten berechnet. Die Methode kann eine hohe Genauigkeit erreichen und eignet

sich gut für Systeme, bei denen alle Teilchen die gleichen Zeitschritte bekommen. Sie ist jedoch asymptotisch um einen Faktor 2 langsamer als der Barnes-Hut-Tree-Algorithmus und führt zu Schwierigkeiten bei Systemen mit hohen Dichtekontrasten [19]. Sie ist deshalb wenig verbreitet.

2.2.3 Gitterbasierte Verfahren

Bei Systemen, wo angenommen werden kann, dass sich die Bestandteile nahezu kollisionsfrei bewegen, sind Gittermethoden ein weitverbreitetes Werkzeug (siehe z.B. [47]). Es wird angenommen, dass jedes Teilchen nur das geglättete Potential einer großen Zahl anderer Teilchen spürt. Dieses Potential wir mit Hilfe eines Gitters berechnet, indem die gegebene Masseverteilung durch eine Verteilung auf einem Gitter approximiert wird.

Particle-Mesh-Methode PM

Dies ist eine Methode, die ursprünglich für die Elektrostatik und Plasmaphysik verwendet wurde. Eine Übersicht über die klassische PM-Methode gibt Sellwood [102]. Die grundsätzliche Vorgehensweise besteht darin, dass über den Raum ein Gitter mit N_g Gitterpunkten gelegt wird. Diesen Punkten wird eine Masse abhängig von den Teilchen in der Umgebung der Gitterpunkte zugewiesen, wobei es für diese Zuordnung verschiedene Verfahren gibt. Mit dem so erhaltenen Massepunktgitter wird dann das Potential ermittelt, indem die Poisson-Gleichung für das Gitter gelöst wird. Dies geschieht über die Anwendung der FFT (**F**ast **F**ourier **T**ransformation). Die Methode führt im dreidimensionalen Fall zu einem von der FFT dominierten Rechenaufwand $O(N_g log N_g)$. Neuere Implementierungen nutzen Techniken zur adaptiven Gitterverfeinerung (engl. AMR = **A**daptive **M**esh **R**efinement). Hier werden, falls nötig, feinere Untergitter hinzugefügt oder Teile des Gitters vergröbert (siehe z.B. [34]). Wichtigster Vorteil der PM-Methode ist ihre hohe Geschwindigkeit. Die Auflösung des Potentials ist mit einigen Gitterabständen jedoch nur sehr gering.

P^3M

Die P^3M-Methode (**P**article-**P**article **P**article-**M**esh) ist ein Mischverfahren von PM und PP (Direct Summation). Hierbei wird die Wechselwirkung mit nahen Teilchen über Direct Summation berechnet, während die Kräfte aufgrund des Potentials entfernter Teilchen durch eine PM-Methode berücksichtigt werden. Damit gewinnt man eine sehr hohe Genauigkeit für Wechselwirkungen auf kurzen Längenskalen, kann aber weiterhin von der hohen Rechengeschwindigkeit von PM profitieren. Die Methode eignet sich für Simulationen mit hohen Dichtekontrasten, wo bei reinem PM für die gleiche Genauigkeit ein wesentlich feineres Gitter erforderlich wäre. Periodische Randbedingungen ergeben sich ohne zusätzlichen Aufwand. Das Verfahren kann durch adaptive Gitterverfeinerung wie bei PM optimiert werden (AP^3M-Methode). Weite Verbreitung findet diese Methode bei kosmologischen Simulationen (siehe z.B. [19]).

2.3 SPH-Algorithmen zur Simulation der hydrodynamischen Kräfte

In der Astrophysik hat sich zur Behandlung hydrodynamischer Phänome das Simulationsverfahren **S**moothed **P**article **H**ydrodynamics (SPH) als Standardmethode etabliert. Es wurde aus der Motivation heraus entwickelt, nicht-achsensymmetrische astrophysikalische Systeme effizient behandeln zu wollen ([70],[36]). Im Gegensatz zu den in anderen wissenschaftlichen Disziplinen häufig eingesetzten gitterbasierten Methoden, wie beispielsweise die Finite-Elemente-Methoden, handelt es sich dabei um eine teilchenbasierte Methode ohne Verwendung eines Gitters. So wird eine Lagrange'sche Beschreibung der Hydrodynamik möglich. Dadurch überwindet die Methode Einschränkungen der Geometrie, Dichteverteilung und Dynamik, denen andere Simulationsmethoden unterliegen. Für dreidimensionale Simulationen mit Berücksichtigung der Gravitation durch eine Teilchenmethode hat sich SPH als überaus erfolgreiche Methode erwiesen und ist entsprechend weit verbreitet.

Im folgenden Abschnitt wird die Methode allgemein dargestellt. In 2.3.2 wird eine weitverbreitete spezielle Formulierung für die Hydrodynamik astrophysikalischer Systeme gegeben, auf die sich der Großteil dieser Arbeit beziehen wird. Dies ist jedoch nicht die einzig mögliche Darstellung, weshalb in 2.3.3 auf diverse Variationen eingegangen wird. Abschnitt 2.3.4 zeigt schließlich, wie sich die Methode in die Simulation der Gravitation einfügt.

2.3.1 Allgemeine Methode von SPH

Der Grundgedanke des SPH-Verfahrens besteht darin, eine Lagrange'sche Beschreibung der Hydrodynamik zu formulieren, wobei gasförmige Materie in diskretisierter Form durch ein Ensemble von statistisch entsprechend der Dichte verteilten Teilchen repräsentiert wird. Diese Fluid-Teilchen sind durch Position, Geschwindigkeit und Masse parametrisiert und die für die Hydrodynamik wichtigen inneren Größen wie Druck, Temperatur, innere Energie und Entropie etc. werden damit berechnet. Die Wechselwirkungskräfte auf ein Fluid-Element werden ausschließlich durch die Verteilung benachbarter Teilchen und deren innere Größen bestimmt. Durch diese Wechselwirkungskräfte angetrieben, bewegen sich die Fluid-Teilchen gemäß der Newton'schen Bewegungsgleichungen. Einen guten Überblick über diese Methode geben die Veröffentlichungen von W. Benz [11] und J. J. Monaghan [77].

Glättungskernmethode

Der Übergang von der Teilchendarstellung zu den hydrodynamischen Gleichungen, welche naturgemäß für kontinuierliche Verteilungen definiert sind, kann als Teil der Interpolationstheorie verstanden werden. Die Idee dabei ist, eine kontinuierliche Wahrscheinlichkeitsverteilung durch eine bekannte diskrete Verteilung zu approximieren. Als statistisches Verfahren wird hierzu die Glättungskernmethode verwendet, welche als Näherung eines Integrals durch eine Monte-Carlo-Methode verstanden werden kann [36]. Die Methode geht von der Glättung von ortsabhängigen

Funktionen $f(\vec{r})$ durch eine Kernfunktion W (engl. Smoothing Kernel) aus:

$$\langle f(\vec{r})\rangle = \int_V W\left(\vec{r}-\vec{r}',h\right)\, f(\vec{r}')\, d^3r'. \tag{2.4}$$

Der Kern W ist eine Funktion mit scharfem Peak bei $\vec{r}=\vec{r}'$ und ist parametrisiert durch h, das die Breite der Kernfunktion bestimmt. Für W wird folgende Normierungsbedingung verlangt:

$$\int_V W(\vec{r},h)\, d^3r = 1. \tag{2.5}$$

Die Kernfunktion muss außerdem die Beziehung

$$\langle f(\vec{r})\rangle \rightarrow f(\vec{r}) \text{ für } h \rightarrow 0 \tag{2.6}$$

garantieren, was bedeutet, dass für $h \rightarrow 0$ der Kern W gegen die δ-Funktion konvergiert. Die Funktion W wird weiterhin sphärisch-symmetrisch gewählt ($W(\vec{r},h) = W(|\vec{r}|,h)$), was dazu führt, dass die Näherung $\langle f\rangle$ in zweiter Ordnung genau bezüglich h ist:

$$\langle f(\vec{r})\rangle = f(\vec{r}) + O(h^2). \tag{2.7}$$

Geeignete Kernfunktionen sind Gaußfunktionen wie z.B.

$$W(\vec{r},h) = \frac{1}{h^3\pi^{3/2}} e^{-(|\vec{r}|/h)^2} \tag{2.8}$$

und bestimmte Spline-Funktionen. Für letztere wird im nächsten Unterkapitel eine explizite Formel gegeben. Spline-Kerne haben gegenüber Gaußfunktionen den Vorteil, dass sie auf einem kompakten Träger definiert werden können und außerhalb dieses Bereichs identisch 0 sind. Dadurch muss bei der Berechnung der Interpolationsintegrale nach Gleichung 2.4 nur eine Sphäre um $\vec{r}$ berücksichtigt werden.

Diskretisierung

Es soll nun davon ausgegangen werden, dass die Funktion f nur an N diskreten Punkten $\vec{r}_j$ bekannt ist, die entsprechend folgender Teilchenzahldichte verteilt sind:

$$n(\vec{r}) = \sum_{j=1}^{N} \delta\,(\vec{r}-\vec{r}_j). \tag{2.9}$$

Die Teilchenzahldichte kann wie andere skalare Funktionen durch die Glättungskernmethode genähert werden, wobei für die Anwendung von Gleichung 2.4 für $\langle n(\vec{r})\rangle$ nach Gleichung 2.7 dann folgende Beziehung gilt:

$$\lim_{h\to 0}\langle n(\vec{r})\rangle = \lim_{h\to 0}\sum_{j=1}^{N} W(|\vec{r}-\vec{r}_j|,h) = n(\vec{r}). \tag{2.10}$$

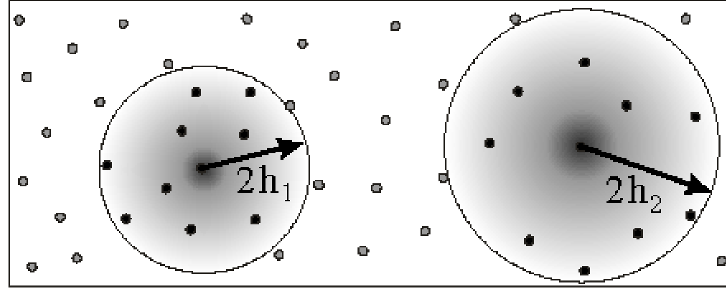


Abbildung 2.1: Veranschaulichung der Berechnung von SPH-Größen durch Gewichtung der Beiträge aus einer Nachbarschaftsumgebung mit kompaktem Träger. Zu den variablen Glättungslängen siehe Abschnitt 2.3.2.

Durch Durchmultiplizieren des Integrands von Gleichung 2.4 mit $n(\vec{r}')/\langle n(\vec{r}')\rangle$, wonach die Gleichung in $O(h^2)$ genau bleibt, und Ersetzen von $1/\left\langle n(\vec{r}_j)\right\rangle$ durch $m_j/\rho(\vec{r}_j)$ (m_j ist die Masse von Teilchen j und $\rho(\vec{r}_j)$ ist die Dichte an der Position von Teilchen j) kann folgender Ausdruck hergeleitet werden, welcher weiterhin eine mit $O(h^2)$ genaue Näherung von f ist:

$$\langle f(\vec{r})\rangle = \sum_{j=1}^{N} f(\vec{r}_j)\,\frac{m_j}{\rho(\vec{r}_j)}\,W(|\vec{r}-\vec{r}_j|,h). \tag{2.11}$$

Wird diese Formel angewendet, um aus der diskreten Verteilung der Teilchen eine kontinuierliche Dichteverteilung $\rho(\vec{r})$ zu gewinnen, ergibt sich folgender Ausdruck:

$$\langle \rho(\vec{r})\rangle = \sum_{j=1}^{N} m_j W(|\vec{r}-\vec{r}_j|,h). \tag{2.12}$$

Anschaulich ausgedrückt, wird die diskrete Verteilung der Fluid-Teilchen durch Faltung mit der Glättungsfunktion W zu einer kontinuierlichen Verteilung verschmiert, woraus sich der Name der Methode ableitet. Die auf diese Weise numerisch erhaltene Dichte wird zur Bestimmung weiterer Größen nach Gleichung 2.11 dort für $\rho(\vec{r}_j)$ eingesetzt.

Behandlung von Ableitungen

Die SPH-Methode wird erst dadurch interessant, dass nicht nur Funktionswerte $f(\vec{r})$ sondern auch deren Ortsableitungen $\vec{\nabla} f(\vec{r})$ an den Teilchenpositionen $\vec{r}=\vec{r}_i$ aus einer diskreten Verteilung bekannter Funktionswerte $f(\vec{r}_j)$ bestimmt werden können. Dazu betrachte man folgende Gleichung für die Approximation von $\vec{\nabla} f(\vec{r}_i)$ welche analog zu Gleichung 2.4 ist:

$$\left\langle \vec{\nabla} f(\vec{r}_i)\right\rangle = \int_V \vec{\nabla} f(\vec{r}')\, W\left(|\vec{r}_i-\vec{r}'|,h\right)\, d^3r'. \tag{2.13}$$

Durch partielle Integration erhält man:

$$\left\langle \vec{\nabla} f(\vec{r}_i)\right\rangle = \int_S f(\vec{r}')\, W\left(|\vec{r}_i-\vec{r}'|,h\right)\, d\vec{S} + \int_V f(\vec{r}')\, \vec{\nabla}_i W\left(|\vec{r}_i-\vec{r}'|,h\right)\, d^3r', \tag{2.14}$$

wobei “$\vec{\nabla}_i$” bedeutet, dass der Gradient nach den Koordinaten $\vec{r}_i$ des Teilchens i gebildet wird. Unter der Annahme, dass das Integrationsvolumen so weit ausgedehnt ist, dass die Funktion $f(\vec{r}')$ oder $W(|\vec{r}_i - \vec{r}'|, h)$ an dessen Oberfläche verschwinden, kann der Oberflächenterm vernachlässigt werden. Diese Bedingung ist bei den meisten astrophysikalischen Simulationsproblemen, für die SPH eingesetzt wird, erfüllt und soll für diese Arbeit stets vorausgesetzt werden. Dann kann wie bei der Ableitung von Gleichung 2.11 diskretisiert werden und man erhält:

$$\left\langle \vec{\nabla} f(\vec{r}_i) \right\rangle = \sum_{j=1}^{N} f(\vec{r}_j) \, \frac{m_j}{\rho(\vec{r}_j)} \, \vec{\nabla}_i W(|\vec{r}_i - \vec{r}_j|, h). \tag{2.15}$$

Es sei hier angemerkt, dass dies nicht unbedingt die numerisch beste Formulierung für $\left\langle \vec{\nabla} f(\vec{r}_i) \right\rangle$ ist. Wie in [77] gezeigt, kann es sinnvoll sein, die Dichte in die Operatoren einzubeziehen. Damit kann alternativ folgende Näherung für $\vec{\nabla} f(\vec{r}_i)$ hergeleitet werden:

$$\left\langle \vec{\nabla} f(\vec{r}_i) \right\rangle = \frac{1}{\rho(\vec{r}_i)} \sum_{j=1}^{N} m_j \, \left(f(\vec{r}_j) - f(\vec{r}_i) \right) \, \vec{\nabla}_i W(|\vec{r}_i - \vec{r}_j|, h). \tag{2.16}$$

Hier seien schließlich noch zwei mögliche numerische Formulierungen zur Approximation von $\vec{\nabla} \times \vec{g}$ gegeben:

$$\left\langle \vec{\nabla} \times \vec{g}(\vec{r}_i) \right\rangle = \sum_{j=1}^{N} \frac{m_j}{\rho(\vec{r}_j)} \, \vec{\nabla}_i W(|\vec{r}_i - \vec{r}_j|, h) \times \vec{g}(\vec{r}_j), \tag{2.17}$$

$$\left\langle \vec{\nabla} \times \vec{g}(\vec{r}_i) \right\rangle = \frac{1}{\rho(\vec{r}_i)} \sum_{j=1}^{N} m_j \, \left(\vec{g}(\vec{r}_i) - \vec{g}(\vec{r}_j) \right) \times \vec{\nabla}_i W(|\vec{r}_i - \vec{r}_j|, h). \tag{2.18}$$

Da wir hier mit sphärisch-symmetrischen Glättungskernen arbeiten, ergibt sich für den Gradienten von W folgende Rechenvorschrift:

$$\vec{\nabla}_i W(|\vec{r}_i - \vec{r}_j|, h) = \frac{\partial W(x,h)}{\partial x} \, \frac{\vec{r}_i - \vec{r}_j}{|\vec{r}_i - \vec{r}_j|} \tag{2.19}$$

Gemäß der SPH-Methode können nun beliebige gasdynamische Variablen und ebenso deren Ableitungen über eine einfache Summierung einer Funktion der Teilchenparameter, multipliziert mit dem Glättungskern oder einer Ableitung davon, berechnet werden. Dies führt zu einem sehr einfachen Rechenschema für die zu betrachtenden hydrodynamischen Gleichungen.

2.3.2 Weitverbreitete spezielle SPH-Formulierung

Es wir nun eine seit einigen Jahren bewährte Formulierung des SPH-Algorithmus im Detail ausgeführt, wie sie insbesondere für den Implementierungsteil dieser Arbeit verwendet wurde. Die Formeln entsprechen weitgehend der in [11] beschriebenen Methode. Zu Implementierungsdetails und numerischen Eigenschaften siehe beispielsweise auch [111], [126], [6].

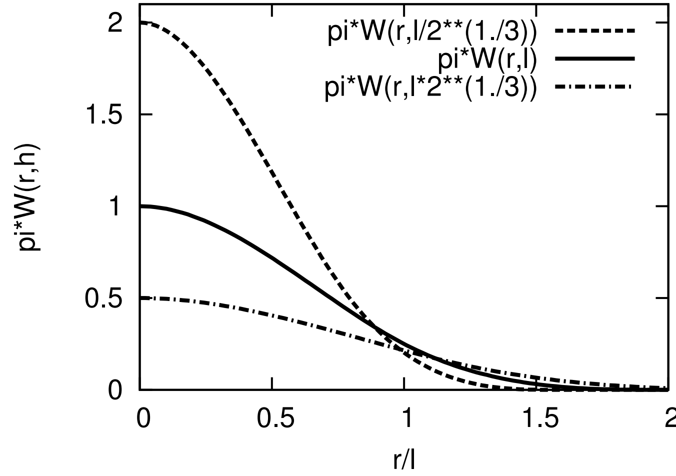


Abbildung 2.2: Kubische Splinefunktion des Standard-Glättungskernes nach Gleichung 2.20. Dargestellt ist der Verlauf für drei verschiedene Werte der Glättungslänge h ($l/\sqrt[3]{2}$, l, $l \cdot \sqrt[3]{2}$).

Spline-Kern

Für die Wahl der SPH-Glättungsfunktion $W(r,h)$ gibt es verschiedene Möglichkeiten. So werden bei theoretischen Betrachtungen über die SPH-Methode vor allem Gaußfunktionen als Kernfunktion verwendet. Diese sind in vielen Fällen aber auch für numerische Simulationen anwendbar. Weiter verbreitet sind jedoch kubische Spline-Funktionen. Diese haben den Vorteil, dass sie auf kompaktem Träger definiert werden können, sodass sie außerhalb einer durch h vorgegebenen Umgebung um den Nullpunkt identisch Null sind. Entsprechend ist für die Berechung von physikalischen Größen für ein Teilchen an der Position $\vec{r}$ eine SPH-Summation nur über die Teilchen, für die sich $(\vec{r}' - \vec{r})$ in dieser Umgebung befindet, auszuführen. Dies reduziert den Rechenaufwand von $O(N^2)$ auf $O(N \cdot N_n)$, wobei N_n die durchschnittliche Zahl von Nachbarteilchen ist, für welche die SPH-Summe gebildet werden muss. Gleichung 2.20 zeigt den inzwischen zum Standard gewordenen Spline-Kern, wie er von Monaghan und Lattanzio vorgestellt wurde [79]. Der Graph dieser Kurve ist in Abbildung 2.2 zu sehen, wobei die Kernfunktion für drei verschiedene Werte von h dargestellt ist. Die in dieser Arbeit beschriebene Implementierung wird auf dieser Standard-Formulierung basieren.

$$W(r,h) = \frac{1}{\pi h^3} \begin{cases} 1 - \frac{3}{2}v^2 + \frac{3}{4}v^3 & : \ 0 \leq v < 1 \\ \frac{1}{4}(2-v)^3 & : \ 1 \leq v < 2 \\ 0 & : \ sonst \end{cases} \qquad \text{mit } v = \frac{r}{h}. \tag{2.20}$$

Variable Glättungslänge

Gängige Praxis in der Anwendung von SPH bei astrophysikalischen Simulationen ist es, für jedes Teilchen mit Index i eine eigene Glättungslänge h_i zu verwenden (siehe z.B. [43] und [11]). Diese

Glättungslänge hängt von der Dichte des Gases an der Stelle $\vec{r}_i$ ab und variiert mit der Dynamik des Ensembles. Damit lässt sich erreichen, dass durch die Faltung mit der Glättungsfunktion trotz extremer Dichtekontraste stets eine glatte kontinuierliche Verteilung resultiert. Im Simulationscode geschieht dies, indem die Zahl der SPH-Nachbarn für jedes Teilchen näherungsweise konstant gehalten wird, was durch Anwendung folgender Gleichung erreicht werden kann (siehe [11]):

$$\frac{dh}{dt} = \frac{1}{3} h \, \vec{\nabla} \cdot \vec{v}. \tag{2.21}$$

Dadurch werden die numerischen Eigenschaften des Codes weitgehend unabhängig von der Geometrie der Masseverteilung, was einer der wichtigsten Vorteile von SPH gegenüber anderen Methoden ist. Zudem kann der Rechenaufwand pro SPH-Teilchen nahezu konstant gehalten werden. Um die Impulserhaltung zu garantieren, ist es nun wichtig, dass die Kräfte von Teilchen i auf Teilchen j und umgekehrt von Teilchen j auf Teilchen i übereinstimmen. Dies kann von den Interpolationsformeln nur gewährleistet werden, wenn eine Symmetrisierung der Formeln in h erfolgt. In dieser Arbeit wird deshalb stets vom Ansatz ausgegangen, dass h in allen Gleichungen durch die Form

$$h(\vec{r}, \vec{r}') = \frac{h(\vec{r}) + h(\vec{r}')}{2} \tag{2.22}$$

ersetzt wird, was für die diskrete Formulierung die Verwendung von h_{ij} entsprechend folgender Gleichung bedeutet:

$$h_{ij} = \frac{h(\vec{r}_i) + h(\vec{r}_j)}{2} = \frac{h_i + h_j}{2}. \tag{2.23}$$

In Abschnitt 2.3.3 wird eine alternative Möglichkeit zur Sicherstellung der Antisymmetrie der Kräfte im Fall variabler Glättungslängen gezeigt. Benz weist in [11] darauf hin, dass die dynamische Adaption von h für die exakte Berechnung von $\vec{\nabla}_i W(|\vec{r}_i - \vec{r}_j|, h)$ zu einem Korrekturterm $\partial W / \partial h \, \vec{\nabla} h$ führt, welcher sehr schwierig zu berechnen ist. Dieser Term kann jedoch vernachlässigt werden, solange $dh/dt \ll h$ gilt, was für diese Arbeit vorausgesetzt werden soll. Bei bestimmten Problemstellungen gilt diese Voraussetzung jedoch nicht, beispielsweise bei Systemen, die starke Schockwellen enthalten, jedoch über globale Zeitschritte integriert werden. Für solche Fälle zeigen Nelson und Papaloizou in [84], wie durch Einfügen der $\vec{\nabla} h$-Korrekturen drastische Verbesserungen der SPH-Simulationen erreicht werden können.

Es sei an dieser Stelle noch angemerkt, dass die Verwendung variabler Glättungslängen mit der Symmetrisierungsvorschrift nach Gleichung 2.23 dazu führt, dass für ein Teilchen mit Index i nicht mehr alle Teilchen in der Sphäre $|\vec{r} - \vec{r}_i| \leq 2 h_i$ SPH-Nachbarn sind, sondern anstatt dessen die Beziehung $|\vec{r}_j - \vec{r}_i| \leq 2 h_{ij}$ die Zugehörigkeit von Teilchen j zur Nachbarschaftsliste von Teilchen i definiert.

Schritt 1: Berechnung der Dichte

Wie in Gleichung 2.11 und 2.15 zu sehen ist, benötigt man zur Berechnung von Funktionswerten durch die SPH-Methode stets die Dichten $\rho(\vec{r}_j)$ an den Positionen der Fluid-Teilchen. Deshalb

besteht der erste Schritt einer Berechnung nach der SPH-Methode darin, diese Dichten zu bestimmen. Dies geschieht nach folgender Formel entsprechend Gleichung 2.12:

$$\rho_i = \langle \rho(\vec{r}_i) \rangle = \sum_{j=1}^{N} m_j W(|\underbrace{\vec{r}_i - \vec{r}_j}_{\vec{r}_{ij}}|, h_{ij}). \tag{2.24}$$

Um im zweiten Schritt die hydrodynamische Druckkraft berechnen zu können, müssen zuvor die Drücke an den Positionen der Teilchen bestimmt werden. Dazu können abhängig von den Eigenschaften der zu simulierenden Gase verschiedene thermodynamische Zustandsgleichungen angewendet werden. Im einfachsten Fall ist der Druck P_i an den Teilchenpositionen $\vec{r}_i$ nur von ρ_i abhängig, wie bei isothermen oder isentropen Gasen. Es können jedoch auch sehr komplexe Zustandsgleichungen auftreten, beispielsweise im Zusammenhang mit Phasenübergängen. Bei der Berechnung der Zustandsgleichungen handelt es sich um einen mit $O(N)$ skalierenden laufzeitunkritischen Teil des Algorithmus, der somit nicht Bestandteil der zu beschleunigenden Berechnungen ist. Deshalb soll hier nicht näher darauf eingegangen werden.

Schritt 2: Kraft- und Energieberechnung

Die wichtigste Gleichung der Hydrodynamik ist die der Impulserhaltung, welche als Euler-Gleichung folgende Gestalt hat[1]:

$$\frac{d\vec{v}}{dt} = \frac{\partial \vec{v}}{\partial t} + \left(\vec{v} \cdot \vec{\nabla}\right) \vec{v} = -\frac{\vec{\nabla} P}{\rho}. \tag{2.25}$$

Aus dieser Gleichung ergibt sich die Beschleunigung der Fluid-Elemente. Durch Aufstellen der Faltungsgleichungen nach 2.4 und Näherungen mit der Genauigkeit der SPH-Methode kann nach W. Benz [11] folgende Gleichung für die approximierte Beschleunigung hergeleitet werden:

$$\frac{d}{dt} \langle \vec{v} \rangle = -\vec{\nabla}_r \left\langle \frac{P}{\rho} \right\rangle - \left(\frac{P}{\rho^2} \right) \vec{\nabla}_r \langle \rho \rangle . \tag{2.26}$$

Mit dem SPH-Verfahren zur Diskretisierung dieser Integrale analog zu Gleichung 2.15 ergibt sich diese Gleichung zu:

$$\frac{d\vec{v}_i}{dt} = -\sum_{j=1}^{N} m_j \left(\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2} \right) \vec{\nabla}_i W \left(\left| \vec{r}_{ij} \right|, h_{ij} \right). \tag{2.27}$$

Dies ist die am häufigsten verwendete Form der Bestimmung der Beschleunigung aus der Druckverteilung des Gases. Diese Gleichung sichert sowohl die Impulserhaltung als auch die lokale Erhaltung des Drehmoments.

[1] Die Berücksichtigung des Gravitationspotentials führt auf der rechten Seite von Gleichung 2.25 zu einem zusätzlichen Term $-\vec{\nabla}\Phi$, welcher aus Gründen der Übersichtlichkeit an dieser Stelle nicht berücksichtigt werden soll.

Die einfache Euler-Gleichung 2.25 ist nicht in der Lage, Dissipation von kinetischer Energie in Wärme zu beschreiben, wofür Viskosität des Fluids notwendig wäre. Sollen jedoch Schockwellen simuliert werden, wird die ansonsten vernachlässigbar geringe molekulare Viskosität astrophysikalischer Gaskomponenten signifikant und die Vernachlässigung der Dissipation führt zu unphysikalischem Verhalten (z.B. starke Oszillationen des Systems). Deshalb wird die Euler-Gleichung für solche Simulationsprobleme um einen Term $\vec{P}^{visc}$ für eine *künstliche Viskosität* erweitert, welcher als zusätzlicher künstlicher viskoser Druck verstanden werden kann:

$$\frac{d\vec{v}}{dt} = -\frac{1}{\rho}\vec{\nabla}_r\left(P + P^{visc}\right). \tag{2.28}$$

Üblicherweise werden zwei verschiedene Arten von viskosem Druck verwendet, die so genannte künstliche Volumenviskosität

$$P_\alpha = -\alpha\,\rho\,l\,c_s\,\vec{\nabla}\cdot\vec{v}, \tag{2.29}$$

und die so genannte von Neumann-Richtmyer-Viskosität

$$P_\beta = \beta\,\rho\,l^2\left(\vec{\nabla}\cdot\vec{v}\right)^2, \tag{2.30}$$

wobei α und β freie Parameter sind, welche die Stärke der Viskosität kontrollieren, l eine typische Längenskala angibt, über die sich eine Schockfront ausbreitet und c_s die Schallgeschwindigkeit ist[2]. Würde man diese Formel derart auswerten, dass $\vec{\nabla}\cdot\vec{v}$ über die SPH-Methode approximiert wird und die Korrekturterme dann analog zu 2.27 berechnet werden, könnten die auftretenden unerwünschten Oszillationen nicht in ausreichendem Maße abgeschwächt werden. Die Ursache dafür ist, dass auf diese Weise Geschwindigkeitsfluktuationen auf Skalen kleiner als h nicht gedämpft werden können. Deshalb wird üblicherweise das von Monaghan und Gingold [78] vorgeschlagene Verfahren verwendet, statt der Größe $\vec{\nabla}\cdot\vec{v}$ einen Term μ_{ij} zu verwenden, welcher als Schätzwert für den Beitrag von Teilchen j zur Divergenz der Geschwindigkeit am Ort $\vec{r}_i$ verstanden werden kann:

$$\mu_{ij} = \frac{h_{ij}\left(\vec{v}_i - \vec{v}_j\right)\cdot\left(\vec{r}_i - \vec{r}_j\right)}{\left|\vec{r}_i - \vec{r}_j\right|^2 + \eta^2 h_{ij}^2}. \tag{2.31}$$

Der Term $\eta^2 h_{ij}^2$ sorgt dafür, dass μ_{ij} für kleine Teilchenabstände nicht divergiert. Dann nimmt die Berechnung von $d\vec{v}/dt$ folgende Gestalt an:

$$\frac{d\vec{v}_i}{dt} = -\sum_{j=1}^{N} m_j\left(\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2} + \Pi_{ij}\right)\vec{\nabla}_i W\left(\left|\vec{r}_{ij}\right|, h_{ij}\right) \tag{2.32}$$

$$\Pi_{ij} = \begin{cases} \dfrac{\left(-\alpha c_{ij}\mu_{ij} + \beta\mu_{ij}^2\right)}{\rho_{ij}} & : \quad \vec{v}_{ij}\vec{r}_{ij} \leq 0 \\ 0 & : \quad \vec{v}_{ij}\vec{r}_{ij} > 0 \end{cases} \tag{2.33}$$

$$c_{ij} = \frac{c_s(\vec{r}_i) + c_s(\vec{r}_j)}{2},\ \rho_{ij} = \frac{\rho_i + \rho_j}{2},\ \vec{v}_{ij} = \vec{v}_i - \vec{v}_j. \tag{2.34}$$

[2] Die Schallgeschwindigkeit kann in Schritt 1 für jedes Teilchen aus der Dichte und dem Druck berechnet werden

Diese Art der Formulierung hat sich als sehr erfolgreich erwiesen. Es hat sich jedoch gezeigt (Balsara [4]), dass bei Scherströmen viel Entropie erzeugt wird. Dieser unerwünschte Effekt wird vermieden, wenn statt Gleichung 2.31 folgende von Balsara vorgeschlagene Formel verwendet wird:

$$\mu_{ij} = \frac{h_{ij}\vec{v}_{ij}\vec{r}_{ij}}{\vec{r}_{ij}^2 + \eta^2 h_{ij}^2} \frac{f_i + f_j}{2} \tag{2.35}$$

$$f_i = \frac{|\langle\vec{\nabla}\cdot\vec{v}\rangle_i|}{|\langle\vec{\nabla}\cdot\vec{v}\rangle_i| + |\langle\vec{\nabla}\times\vec{v}\rangle_i| + \eta' c_i/h_i}. \tag{2.36}$$

Dies ist die Formulierung, die für diese Arbeit verwendet wird. Man erkennt, dass zur Berechnung der Balsara-Koeffizienten f_i die Erwartungswerte für die Divergenz und Rotation von v für das Teilchen i erforderlich sind. Diese können in Schritt 1 zusätzlich zur Dichteberechnung bestimmt werden, wobei analog zu den Formeln 2.16 und 2.18 gerechnet werden kann:

$$\rho_i\,(\vec{\nabla}\cdot\vec{v})_i = \sum_{j=1}^{N} m_j\,(\vec{v}_j - \vec{v}_i)\,\vec{\nabla}_i W(|\vec{r}_{ij}|, h_{ij}), \tag{2.37}$$

$$\rho_i\,(\vec{\nabla}\times\vec{v})_i = \sum_{j=1}^{N} m_j\,(\vec{v}_i - \vec{v}_j)\times\vec{\nabla}_i W(|\vec{r}_{ij}|, h_{ij}). \tag{2.38}$$

Auf diese Weise brauchen die Dichten ρ_i für die Summation noch nicht bekannt sein. So lässt sich eine weitere Schleife über alle Nachbarteilchen zwischen Schritt 1 und Schritt 2 vermeiden, die bei Anwendung der Gleichungen 2.15 und 2.17 erforderlich sein würde.

Im Allgemeinen reichen die Gleichungen zur Impulserhaltung (2.25) und die Zustansgleichung des Gases nicht aus, um ein hydrodynamisches System vollständig zu beschreiben. Während für isotherme und isentrope Zustandsgleichungen keine weiteren Größen zu berechnen sind, wird im Fall der idealen Gasgleichung für adiabatische Zustandsänderungen beispielsweise die Temperatur oder innere Energie benötigt. Das System von Gleichungen kann vervollständigt werden, wenn die Gleichung für die Energieerhaltung einbezogen wird. Für ein reibungsfreies ideales Gas lässt sich die hydrodynamische Formulierung der adiabatischen Energieerhaltung wie folgt aufstellen:

$$\frac{\partial u}{\partial t} + (\vec{v}\cdot\vec{\nabla})u = -\frac{p}{\rho}\vec{\nabla}\cdot\vec{v}. \tag{2.39}$$

Daraus ergibt sich analog der Herleitung von Gleichung 2.15 folgende Formel:

$$\frac{du_i}{dt} = \frac{P_i}{\rho_i^2}\sum_{j=1}^{N} m_j \vec{v}_{ij}\cdot\vec{\nabla}_i W(|\vec{r}_{ij}|, h_{ij}). \tag{2.40}$$

Die oben eingeführte künstliche Viskosität führt an Schockfronten zusätzlich zu einer Erhitzung des Gases. Aus diesem Grund wird Gleichung 2.40 ein Term hinzugefügt, welcher die Entropieänderung aufgrund der künstlichen Viskosität berücksichtigt. Für diese Arbeit wird deshalb

folgende SPH-Form für die Energiegleichung verwendet, welche die Erhaltung der Gesamtenergie sichert:

$$\frac{du_i}{dt} = \frac{P_i}{\rho_i^2} \sum_{j=1}^{N} m_j \vec{v_{ij}} \cdot \vec{\nabla}_i W(|\vec{r}_{ij}|, h_{ij}) + \frac{1}{2} \sum_{j=1}^{N} m_j \Pi_{ij} \vec{v_{ij}} \cdot \vec{\nabla}_i W(|\vec{r}_{ij}|, h_{ij}). \tag{2.41}$$

Der erste Summand der rechten Gleichungsseite kann direkt aus dem bereits für die Berechnung von f_i ermittelten Wert für $\langle \vec{\nabla} \cdot \vec{v} \rangle_i$ durch Multiplikation mit P_i/ρ_i bestimmt werden. Der zweite Summand kann leicht in die Schleife zur Berechnung von $d\vec{v}/dt$ einbezogen werden.

Zusammenfassung des SPH-Algorithmus

Es sollen hier noch einmal die wesentlichen Punkte des eben gegebenen SPH-Formalismus zusammengefasst werden, um den Ablauf der Berechnungen zu verdeutlichen.

- Es wird ein Spline-Glättungskern mit kompaktem Träger verwendet. Dies führt dazu, dass nur die Beiträge einer Anzahl nächster Nachbarn um ein Teilchen in die Berechnung von SPH-Größen einfließen. Für jedes Teilchen wird also eine Nachbarschaftsliste erstellt und für die SPH-Formeln wird nur über diese Liste summiert.
- Für jedes Teilchen i wird in einer ersten SPH-Schleife über die Nachbarn durch die Summenformel 2.24 die Dichte ρ_i bestimmt. In der gleichen Schleife werden die Summationen für $\vec{\nabla} \cdot \vec{v}$ (Gleichung 2.37) und $\vec{\nabla} \times \vec{v}$ (Gleichung 2.38) durchgeführt.
- Für jedes Teilchen wird nun der Druck, die Schallgeschwindigkeit und der Balsarakoeffizient f_i (Gleichung 2.36) bestimmt.
- Für alle Teilchen können dann in einer zweiten SPH-Schleife über die Nachbarn die Kräfte (Gleichung 2.32 mit 2.35 für die Berechnung von μ_{ij}) und Energien (Gleichung 2.41) berechnet werden.

Es gibt also zwei Schleifen über die Nachbarschaftslisten der SPH-Teilchen, in welchen eine Summation von SPH-Beiträgen der Nachbarn durchgeführt wird.

2.3.3 Variationen des SPH-Verfahrens

Die Formeln zur Berechnung von SPH-Erwartungswerten, wie sie im letzten Abschnitt entwickelt wurden, sind keineswegs die einzig mögliche Variante. Es gibt insbesondere unterschiedliche Ansätze, eine Symmetrie der Beiträge von Teilchen j zu einem Erwartungswert von Teilchen i und umgekehrt zu erreichen. Diese Symmetrie ist notwendig, damit eine antisymmetrische paarweise Wechselwirkung zwischen Teilchen möglich wird, was grundlegende Voraussetzung für die Impulserhaltung ist. So können beispielsweise die Kernel-Beiträge bei unterschiedlichen Glättungslängen $h_i \neq h_j$ anstatt durch

$$W\left(|\vec{r}_i - \vec{r}_j|, \frac{h_i + h_j}{2}\right)$$

auch durch

$$\frac{1}{2}\left(W(|\vec{r}_i - \vec{r}_j|, h_i) + W(|\vec{r}_i - \vec{r}_j|, h_j)\right)$$

symmetrisiert werden. Ein anderes Beispiel ist die Symmetrisierung von P/ρ^2, wie sie für die Berechnung der Beschleunigung gebraucht wird. Anstatt das arithmetische Mittel

$$\frac{1}{2}\left(\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2}\right)$$

zu berechnen, kann auch das geometrische Mittel verwendet werden:

$$\frac{\sqrt{P_i\,P_j}}{\rho_i\,\rho_j}.$$

Die eben beschriebenen Symmetrisierungsvarianten wurden in [43] eingeführt und diskutiert.

Besonders viele Variationen gibt es bei der Behandlung der künstlichen Viskosität. Diese Größe ist nicht direkt physikalisch motiviert, sondern soll lediglich unphysikalisches Verhalten der SPH-Simulation bei starken Schockwellen und Turbulenz vermeiden. Sie hat den ungewollten Seiteneffekt einer unphysikalisch hohen Energiedissipation. Deshalb gibt es zahlreiche Modifikationen, welche für verschiedene Simulationsszenarien optimiert sind. Es soll an dieser Stelle nicht näher darauf eingegangen werden.

2.3.4 Einbettung in Simulation der Gravitation

Auf den ersten Blick unterscheidet sich die algorithmische Struktur von SPH sehr stark von den Verfahren zur Berechnung der Gravitationskraft, und es erscheint schwierig, eine Verbindung herzustellen. Jedoch basieren alle bisher beschriebenen Methoden auf der Repräsentation des astrophysikalischen Systems durch Teilchen. Deshalb können die Fluidteilchen direkt in die Gravitationsbehandlung für das Gesamtsystem einbezogen werden. Umgekehrt werden für die wichtigsten N-body-Codes auch Nachbarschaftsbeziehungen in die Berechnung einbezogen, was sich ebenso für die Erzeugung von Nachbarschaftslisten für SPH ausnutzen lässt. So wenden die P^3M-Codes für kurze Distanzen eine *PP*-Methode an und die Ermittlung der dafür benötigten Nahbereichsteilchen leistet bereits viel Vorarbeit für den Aufbau der SPH-Nachbarschaftslisten (siehe z.B. [20]). Und bei baumbasierten Verfahren beinhaltet die hierarchische Datenstruktur bereits die Information über die Nachbarschaftsbeziehungen. Hernquist und Katz beschreiben in [43] ihren TREESPH-Code, der auf einem Barnes-Hut-Tree aufbaut und für die Nachbarschaftssuche die Baumstruktur ähnlich wie für den Aufbau der Gravitations-Interaktionslisten verwendet. Ein aktuelleres Beispiel für einen mit einem Barnes-Hut-Tree gekoppelten SPH-Code ist GADGET [106]. Ein Beispiel für die Verwendung eines Binärbaumes in Verbindung mit SPH ist der GASOLINE-Code [121].

Diese Arbeit orientiert an der Implementierung eines Codes basierend auf dem Press-Tree wie in [111] und [126]. Die Gravitation wird sowohl für die N-body-Teilchen[3] als auch für SPH-Teilchen gerechnet. Zur Beschleunigung kann ein GRAPE-System verwendet werden (siehe Abschnitt 2.4.2, [110] und [126]). Für die SPH-Teilchen werden dann Korrekturen notwendig, da

[3] N-body-Teilchen sind rein gravitativ wechselwirkende Teilchen.

die SPH-Teilchen zur Vermeidung von Inkonsistenzen ein nach dem SPH-Formalismus geglättetes Gravitationspotential bewirken sollten. Dazu kann im SPH-Formalismus folgende Formel angewendet werden, welche sich daraus ableitet, dass für ein Teilchen i die Glättungskernfunktion $W(|\vec{r}-\vec{r}_i|,h_{ij})$ als Dichteverteilungsfunktion interpretiert wird (nach [11]):

$$\begin{aligned} -\vec{\nabla}\Phi_i &= -G\sum_{j=1}^{N}\frac{M(|\vec{r_{ij}}|)}{r_{ij}^2}\frac{\vec{r_{ij}}}{|\vec{r_{ij}}|} \\ M(x) &= 4\pi\int_0^x u^2 W(u,h_{ij})du. \end{aligned} \tag{2.42}$$

2.4 Rechnerplattformen für astrophysikalische Simulationen

Da die Qualität astrophysikalischer Simulationen von der Auflösung und damit von der simulierten Teilchenzahl abhängt, wird in der Regel mit der bei vertretbarer Rechenzeit maximal möglichen Anzahl von Teilchen gerechnet. Dazu wird vorzugsweise auf Hochleistungsrechnern gearbeitet. Grundsätzlich können sowohl Rechner-Cluster als auch Supercomputer eingesetzt werden, und beide Rechnerklassen werden abhängig von der Verfügbarkeit solcher Rechensysteme verwendet. In Abschnitt 2.4.1 werden einige Aspekte der Implementierung von Simulationscodes auf solchen Mainstream-Plattformen umrissen. Abschnitt 2.4.2 gibt einen Überblick über Spezialrechner, die für astrophysikalische Simulationen eingesetzt werden.

2.4.1 Hochleistungsrechner

Supercomputer sind die klassische Plattform für astrophysikalische Simulationen. Kann die Anwendung gut parallelisiert werden, wie es beispielsweise bei kosmologischen Simulationen mit P^3M bei relativ homogener Verteilung der Massen der Fall ist, skaliert die Rechengeschwindigkeit sehr gut mit der Anzahl der Prozessoren. Hier kommen die Vorteile eines Supercomputers zum tragen, wie die hohe Rechenleistung der Knoten und die schnelle Verbindung mit kurzer Latenzzeit zwischen Rechenknoten. Sobald jedoch auch hydrodynamische Phänomene durch SPH berücksichtigt werden oder starke Inhomogenitäten durch die Zusammenballung von Massen auftreten, skaliert die Rechenleistung des Codes für mehr als ca. 30 Prozessoren schlecht (siehe z.B. [90]). Mehr und mehr werden auch Rechnercluster eingesetzt. Diese Systeme sind sehr populär geworden, seit schnelle Verbindungsnetze für Standard-PCs existieren und gute Standardprozessoren eine bei vielen Anwendungen vergleichbar hohe Leistung wie bei einem Supercomputer-Knoten erzielen. Durch den Einsatz von Standardkomponenten ergibt sich eine wesentlich bessere Price-Performance als bei Supercomputern. Mit Clustern ist eine sehr hohe Speicherkapazität pro Rechenknoten möglich, was insbesondere für hydrodynamische Codes wichtig ist. Zu aktuellen Beispielen von astrophysikalischen Simulationen auf Clustern siehe z.B. [121] und [25].

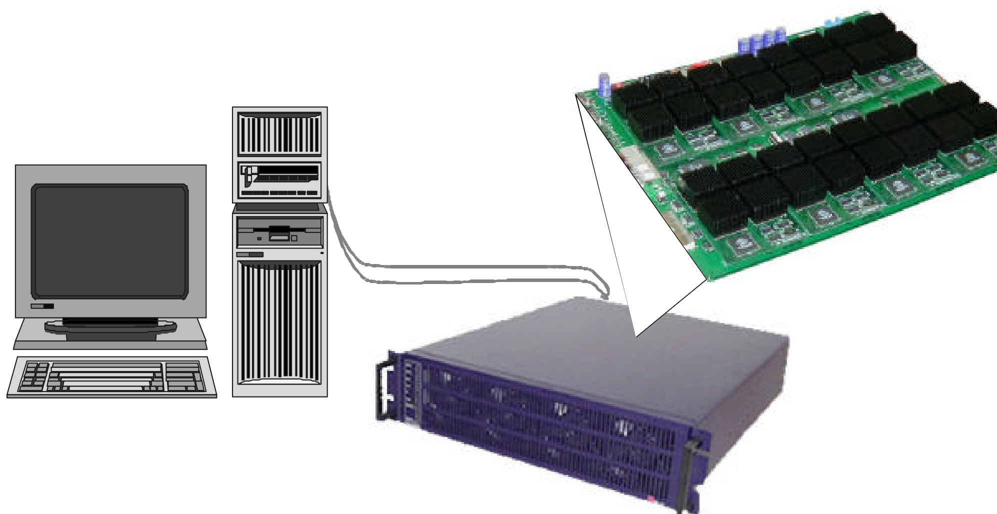


Abbildung 2.3: Simulationssystem, bestehend aus Host-Rechner und GRAPE-6-Spezialrechner zur Beschleunigung der Gravitationsberechnung.

2.4.2 Spezialrechner

GRAPE

Die Behandlung der Gravitation ist der bei weitem rechenaufwändigste Teil von astrophysikalischen N-Körper-Simulationen und limitiert deshalb die Teilchenzahl, die simuliert werden kann. Aus dem wissenschaftlichen Bedarf nach höheren Teilchenzahlen (um die Auflösung und Aussagekraft der Simulationen zu steigern) motivierte sich das GRAPE-Projekt der Universität Tokio [26]. Ende der 1980er-Jahre begonnen, wurde eine Generationenfolge von Spezialrechnern entwickelt, welche auf der gemeinsamen Idee beruhen, die Gravitationskraft durch ein maßgeschneidertes Rechenwerk in der Form einer Pipeline (GRAvity Pipe = *GRAPE*) massiv parallel zu berechnen (siehe z.B. [113], [75] und [50]). Dieses Konzept bietet sich an, da die Gravitationskraft über eine einzige Formel, bestehend aus wenigen Operationen berechnet werden kann. Ein Simulationssystem besteht bei diesem Konzept aus einem Standard-Host-System und angeschlossenem GRAPE-Rechenbeschleuniger, wie in Abbildung 2.3 gezeigt wird. Dieser Rechenbeschleuniger ist im Wesentlichen aus parallelen Spezialprozessoren aufgebaut, in welchen die Rechenpipelines implementiert sind. Die aktuelle Plattform ist GRAPE-6 [74]. Die Konzepte, die bei dieser Architektur angewandt werden, sollen im Folgenden etwas genauer beleuchtet werden.

Abbildung 2.4 zeigt eine schematische Darstellung der GRAPE-6-Architektur aus der Sicht des Anwenders. Das grundsätzliche Verarbeitungsschema ist folgendes: Die Positionen $\vec{r}_i$ und Geschwindigkeiten $\vec{v}_i$ der Teilchen, für welche die Kräfte berechnet werden sollen, werden lokal in den Pipelines gespeichert, die Daten der Wechselwirkungspartner werden dagegen in einem externen Speicher abgelegt. Wird die Berechnung gestartet, werden die benötigten Teilchendaten der Interaktionspartner automatisch, gesteuert über einen Adressgenerator, den Rechenwer-

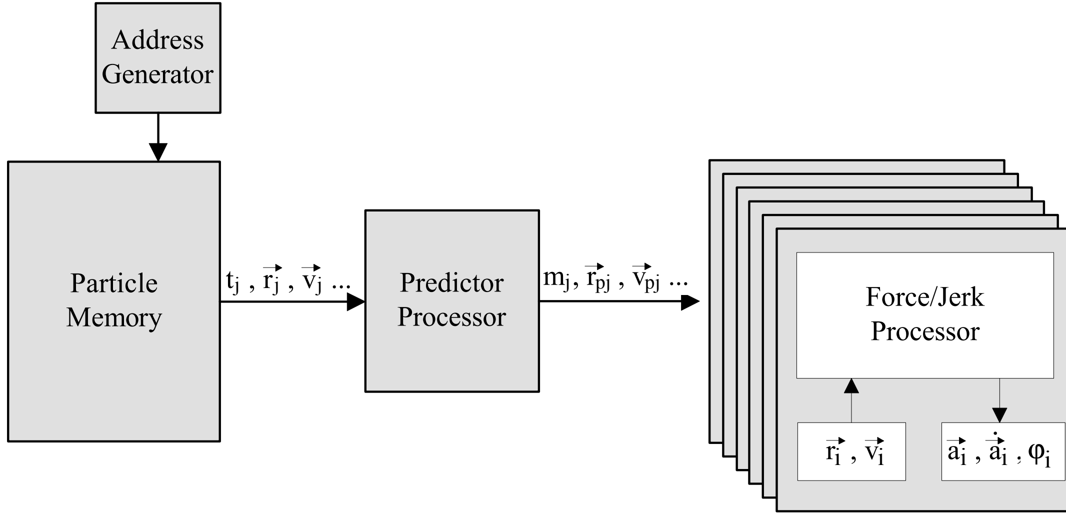


Abbildung 2.4: Architektur des GRAPE-6-Spezialrechners aus Anwendersicht (nach [72]).

ken zugeführt. Die Architektur erlaubt neben der reinen Berechnung der Kräfte zwei wichtige algorithmische Optimierungen. Den Pipelines vorgeschaltet ist ein so genannter *Predictor Processor*. Dieser ermöglicht die Extrapolation von Teilchenvariablen früherer Zeitpunkte t_j auf den aktuellen Zeitschritt t der Simulation:

$$\begin{aligned} \vec{r}_j(t_j) &\rightarrow \vec{r}_{pj} \approx \vec{r}_j(t) \\ \vec{v}_j(t_j) &\rightarrow \vec{v}_{pj} \approx \vec{v}_j(t). \end{aligned}$$

Damit können auf einfache Weise Algorithmen mit individuellen Zeitschritten implementiert werden. Als zweite Optimierung wird neben dem Potential die Teilchenbeschleunigung und deren Ableitung berechnet. Dies erlaubt es, im Integrator ein Hermite-Schema zur Bestimmung der Teilchenvariablen anzuwenden. Die Formel, die in den GRAPE-6-Plattformen berechnet wird, ist dementsprechend:

$$\begin{aligned} \vec{a}_i &= \sum_j G m_j \frac{\vec{r}_{ij}}{\left(r_{ij}^2+\varepsilon^2\right)^{3/2}} \\ \dot{\vec{a}} &= \sum_j G m_j \left[\frac{\vec{v}_{ij}}{\left(r_{ij}^2+\varepsilon^2\right)^{3/2}} - \frac{3\left(\vec{v}_{ij}\cdot\vec{r}_{ij}\right)\vec{r}_{ij}}{\left(r_{ij}^2+\varepsilon^2\right)^{5/2}} \right] \\ \phi_i &= \sum_j G m_j \frac{1}{\left(r_{ij}^2+\varepsilon^2\right)^{1/2}}. \end{aligned} \tag{2.43}$$

Die GRAPE-6-Maschinen basieren auf eigens konstruierten ASICs, den GRAPE-6-Chips, welche in einem 0.25-μm-Prozess gefertigt sind. Die Bausteine sind so konstruiert, dass damit massiv parallele Systeme aufgebaut werden können. Ein GRAPE-6-Chip beinhaltet einen Pipeline-Prozessor, der die Netzwerkschnittstelle, das Speicherinterface und die Prediction-Einheit implementiert sowie sechs parallele Pipelines, welche selbst logisch in acht virtuelle Pipelines

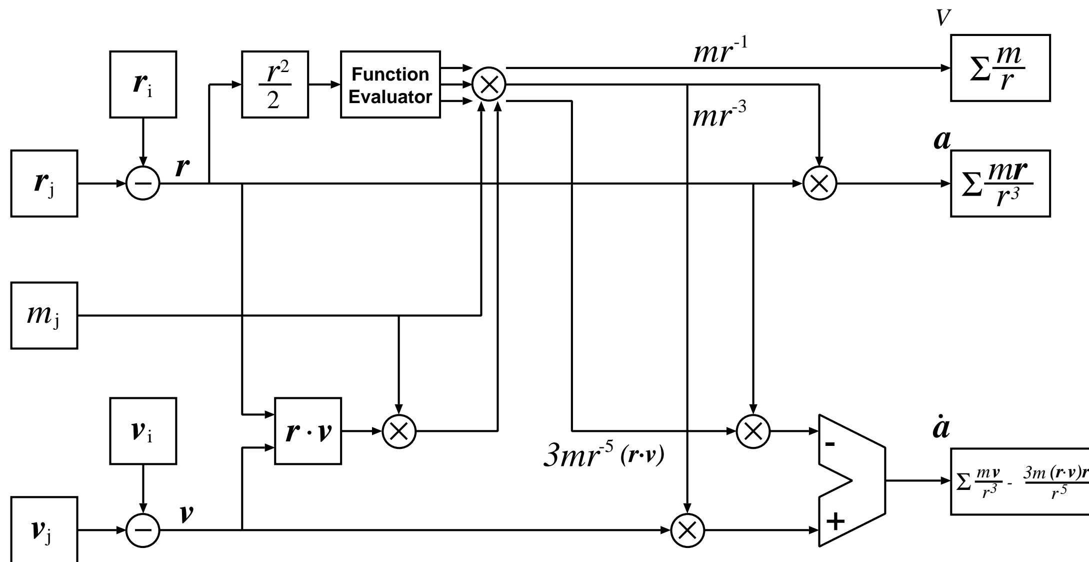


Abbildung 2.5: Architektur der Pipeline zur Berechnung der Formeln 2.43 (entnommen aus [75]).

aufgeteilt sind. Aus Anwendersicht beinhaltet ein solcher Chip also 48 Pipelines, welche zur parallelen Berechnung der Kräfte aller im Speicher gehaltenen Teilchen auf 48 Teilchen verwendet werden. Ein Blockschaltbild der Implementierung einer der sechs GRAPE-6-Pipelines zur Berechnung der Formeln 2.43 ist in Abbildung 2.5 dargestellt.

Ein GRAPE-6-Board besteht aus 32 GRAPE-6-Chips. Die sechs Pipelines pro Chip arbeiten mit einer Taktfrequenz von 90 MHz. Pro Pipeline kann in jedem Taktzyklus eine Teilchen-Teilchen Wechselwirkung (ϕ, $\vec{a}$ und $\dot{\vec{a}}$) berechnet werden, wozu 57 Gleitkommaoperationen angesetzt werden. Für das gesamte GRAPE-6-Board ergibt dies eine Rechenleistung von 985 GFlops und es können pro Sekunde $17.3 \cdot 10^9$ Kraftberechnungen durchgeführt werden. Alle GRAPE-6-Chips berechnen die Wechselwirkungen auf einem gemeinsamen Satz von 48 Teilchen, aber jeweils mit unterschiedlichen Listen von Interaktionspartnern. Die Daten dieser Wechselwirkungspartner werden in lokalen Speichern abgelegt, welche direkt an die Chips angeschlossen sind. Das gesamte Board hat also genauso wie ein einzelner Chip 48 virtuelle Pipelines. Die 32 Teilergebnisse zu einer dieser Pipelines des GRAPE-6-Boards werden in einem über FPGAs implementierten Addiererbaum auf dem Board zu *einem* Rechenergebnis pro virtueller Pipeline reduziert und an den Host gesandt.

Die GRAPE-Architektur erreicht bei Simulationen von Systemen, deren Entwicklung fast ausschließlich durch die Gravitationskraft determiniert ist, eine sehr hohe Effizienz. So wurde in Tokio ein paralleles System aus 64 GRAPE-6-Boards mit einer Spitzenleistung von 63.4 TFlops aufgebaut und für eine echte Simulationsanwendung eine durchschnittliche Leistung von über 33 TFlops nachgewiesen [74]. Mehrfach wurde für die GRAPE-Systeme bereits der Gordon-Bell-Preis verliehen.

Tabelle 2.1 zeigt die Entwicklung der GRAPE-Plattformen. Es werden zwei unterschiedli-

che Zweige verfolgt, die sich in der Rechengenauigkeit der GRAPE-Pipelines unterscheiden. Die Plattformen mit niedriger Genauigkeit haben ungerade, die mit hoher Genauigkeit gerade Versionsnummern.

Plattform	Jahr	Prozess	Peakperf.	Genauigkeit	# Pipelines	Frequenz
GRAPE-1	1989		240 MFlops	niedrig	1	8 MHz
GRAPE-1A	1990		240 MFlops	niedrig	1	8 MHz
GRAPE-2	1990		40 MFlops	hoch	1/3	4 MHz
GRAPE-2A	1992		180 MFlops	hoch	1	10 MHz
GRAPE-3	1991		14.4 GFlops	niedrig	48	10 MHz
GRAPE-3A	1992		2.4 GFlops	niedrig	4	20 MHz
GRAPE-4	1995	1 μm	1 TFlops	hoch	1692	30 MHz
GRAPE-5	1999	0.5 μm	38.4 GFlops	niedrig	16	80 MHz
GRAPE-6	2002	0.25 μm	63 TFlops	hoch	12288	90 MHz

Tabelle 2.1: Entwicklung der GRAPE-Systeme.

Mittlerweile ist neben dem aktuellen GRAPE-6-Spezialrechner mit 32 GRAPE-6-Chips auch ein kleineres System, basierend auf einer 64-Bit-PCI-Einsteckkarte, verfügbar. Dieses enthält mit vier GRAPE-6-Chips nur $1/8$ der Ressourcen des großen Boards, kostet jedoch auch nur $1/8$ davon.

MDM

Die Molecular Dynamics Machine (MDM) ist wie GRAPE eine japanische Entwicklung und wird am Institut für Physikalische und Chemische Forschung, RIKEN, betrieben. Das System besteht aus zwei Spezialrechnerplattformen, dem MDGRAPE-2-System zur Berechnung von Van-der-Waals- und Coulombkräften im Ortsraum und dem WINE-2-System zur Berechnung der Coulombkräfte im Fourierraum [83]. Das MDGRAPE-2-System hat prinzipiell die gleiche Funktionsweise wie GRAPE, kann aber verschiedene Kraftgesetze implementieren. Die WINE-2-Boards wurden ebenfalls basierend auf massiv parallelen Pipelines innerhalb spezieller ASICs konstruiert und ermöglichen unter anderem die schnelle Berechnung der DFT und IDFT. In [82] wird die Leistungsfähigkeit eines Systems bestehend aus 96 Boards mit jeweils 16 MDGRAPE-2-Chips, 144 Boards mit jeweils 16 WINE-2-Chips und diversen Host-Workstations präsentiert. Das Gesamtsystem hat eine Peak-Performance von etwa 78 TFlops und für eine Simulation der Verfestigung von NaCl mit 33 Millionen Teilchen wurde eine Rechenleistung von 8.61 TFlops erreicht [82]. Die MDGRAPE-2 Plattform ist auch für die astrophysikalische Anwendung interessant, da damit auch spezielle baumbasierte Verfahren zur Gravitationsberechnung, für welche GRAPE ungeeignet ist, effizient umgesetzt werden können ([51]).

PROGRAPE

Das PROGRAPE-System wurde als Erweiterung der GRAPE-Plattform entwickelt. Ziel war es, wie bei den GRAPE-Systemen eine massiv parallele Rechnerarchitektur aufzubauen, welche je-

doch andere Wechselwirkungen als die Gravitation simulieren kann. Insbesondere sollte die Architektur für SPH-Berechnungen verwendet werden können. Es wurde der Ansatz verfolgt, bei ansonsten unveränderter Systemarchitektur die GRAPE-Chips der GRAPE-Boards durch rekonfigurierbare Chips zu ersetzen. In [40] wird beschrieben, wie ein Prototyp dieser Plattform für die Gravitationswechselwirkung eine Rechenleistung von knapp 1 GFlop erreicht. Es existiert bereits ein PROGRAPE-2-System mit einem moderneren FPGA (Altera APEX20K400). Dieser ist in der Lage, die Pipelines für eine etwas einfacherere SPH-Formulierung als sie in Abschnitt 2.3.2 gegeben wurde aufzunehmen. Es wird mit einem logarithmischen Zahlenformat geringer Genauigkeit gerechnet (14 Bit). Dem Autor sind jedoch noch keine diesbezüglichen Veröffentlichungen bekannt. Im Rahmen dieser Arbeit wurde 2002 in Japan auf dem PROGRAPE-2-System ein Prototyp für die SPH-Formulierung nach Abschnitt 2.3.2 implementiert. Es wurde evaluiert, dass das PROGRAPE-2-System für unsere Genauigkeitsanforderungen noch nicht geeignet ist.

Für zukünftige Entwicklungen von GRAPE-Systemen räumt man in Japan den rekonfigurierbaren Rechensystemen einen hohen Stellenwert ein. Bereits jetzt werden FPGAs für die Verbindung der GRAPE-6-Chips zu parallelen Recheneinheiten eingesetzt. Für die nahe Zukunft ist geplant, ein paralleles System mit FPGAs als zentralen Rechenelementen aufzubauen, um damit massiv parallel SPH-Simulationen durchzuführen.

Kapitel 3

FPGA-basierte, rekonfigurierbare Rechnerplattformen

3.1 Rekonfigurierbare Logikbausteine

In diesem Kapitel werden die Grundlagen rekonfigurierbarer Logik erläutert, wobei ausschließlich auf die FPGA-Technologie eingegangen werden soll. Rekonfigurierbare Logik bedeutet, dass durch die Konfiguration solcher Chips elektronische Schaltungen erzeugt werden können. Diese Bausteine sind nicht nur einmal konfigurierbar sondern rekonfigurierbar. So können mit ein und demselben Chip viele verschiedene elektronische Schaltungen implementiert und jederzeit ausgetauscht werden. Daher sind diese Bauelemente ideal für Anwendungen, bei denen mit Spezialarchitekturen eine hohe Leistungsfähigkeit erreicht werden soll, aber die Flexibilität einer programmierbaren Lösung zwingend notwendig ist.

Diese Arbeit beruht auf dem Ansatz, rekonfigurierbare Logik in Form von FPGAs zur Implementierung von Rechenbeschleunigern einzusetzen. In diesem Kapitel wird entsprechend detailliert auf die Architektur von FPGAs eingegangen. Damit wird das Fundament für alle weiteren Ausführungen zur Implementierung der Spezialarchitektur gelegt, mit der astrophysikalische Simulationsalgorithmen beschleunigt werden.

In Abschnitt 3.1.1 wird die allgemeine Architektur von FPGAs erläutert und in Abschnitt 3.1.2 werden Details zum FPGA-Typ, der in dieser Arbeit hauptsächlich verwendet wurde, vorgestellt. Die Hochleistungs-FPGAs anderer Hersteller weisen ähnliche Eigenschaften auf, und so ist vieles aus Abschnitt 3.1.2 übertragbar. Für eine Übersicht über FPGAs anderer Hersteller sei auf [122] und Referenzen verwiesen.

3.1.1 Allgemeine Architektur von FPGAs

Die Abkürzung 'FPGA' steht für **F**ield **P**rogrammable **G**ate **A**rray. Der Name weist auf die allen FPGAs gemeinsame Struktur hin, dass sie im Wesentlichen aus einer Matrix programmierbarer Logikelemente bestehen (siehe Abb. 3.1). In diesem Kontext ist Programmierung gleichbedeutend mit Konfiguration der Eigenschaften der FPGA-Ressourcen. Durch die Konfiguration

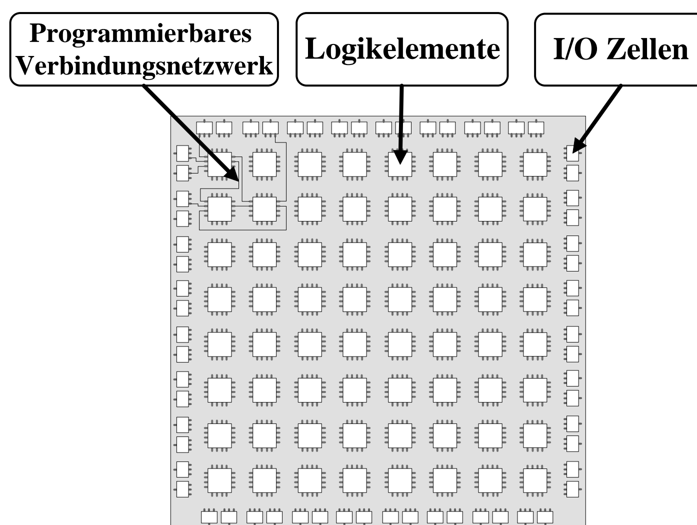


Abbildung 3.1: Prinzipieller Aufbau eines FPGAs.

bekommen die Logikelemente eines FPGAs ein bestimmtes digitalelektronisches Verhalten zugewiesen. Die Logikelemente sind verbunden durch ein ebenfalls konfigurierbares Verbindungsnetzwerk. Durch Programmieren des Chips kann eine beliebige digitalelektronische Schaltung erzeugt werden – natürlich im Rahmen der Logikressourcen. FPGAs werden programmiert, indem ein so genannter Konfigurationsbitstrom auf den Chip transferiert wird. Nach der Konfiguration verhält sich der Baustein wie eine für eine Anwendung speziell konstruierte Integrierte Schaltung. Im Unterschied zum ASIC (**A**pplikation **S**pecific **I**ntegrated **C**ircuit) kann das elektronische Verhalten eines FPGAs jederzeit durch erneute Konfiguration geändert werden. Ein Schaltungsdesign, auf dem der Konfigurationsbitstrom für den FPGAs basiert, wird im Folgenden FPGA-Design genannt.

Die am weitesten verbreiteten FPGAs benutzen auf SRAM-Zellen basierende Logikelemente. Hier werden logische Funktionen einer festen Anzahl von Signalen mittels kleiner vorprogrammierter Speicherelemente (ROM) implementiert. Die Eingangssignale einer logischen Verknüpfung werden als Adressen eines solchen Speichers geschaltet. Durch eine in das Speicherelement programmierte Look-Up-Tabelle (LUT) kann eine beliebige logische Verknüpfung der Eingangssignale gebildet werden. Diese Look-Up-Tabellen werden deshalb auch Funktionsgeneratoren genannt. Die programmierbaren Logikelemente der gängigen FPGA-Typen beinhalten eine oder mehrere LUTs, die jeweils über vier Eingangssignale adressiert werden. Logische Funktionen mit mehr als vier Eingangssignalen werden durch das Zusammenschalten mehrerer LUTs gebildet. Moderne FPGAs erweitern die Möglichkeiten der Logikzellen durch zusätzlichen Schaltungsaufwand. So sorgt eine Carry-Logik mit speziellen schnellen Carry-Verbindungen zwischen benachbarten Logikelementen für schnelle Addierer und Zähler. Die Logikzellen lassen sich meist auch als kleine RAM-Blöcke ansteuern. Die Funktionsgeneratoren werden dabei nicht als ROM-Elemente sondern als SRAM konfiguriert. Damit können beispielsweise Zwischenergebnisse innerhalb des FPGAs gespeichert oder FIFO-Elemente aufgebaut werden.

Um sequenzielle Logik zu ermöglichen, beinhalten die Logikelemente eines FPGAs ein oder

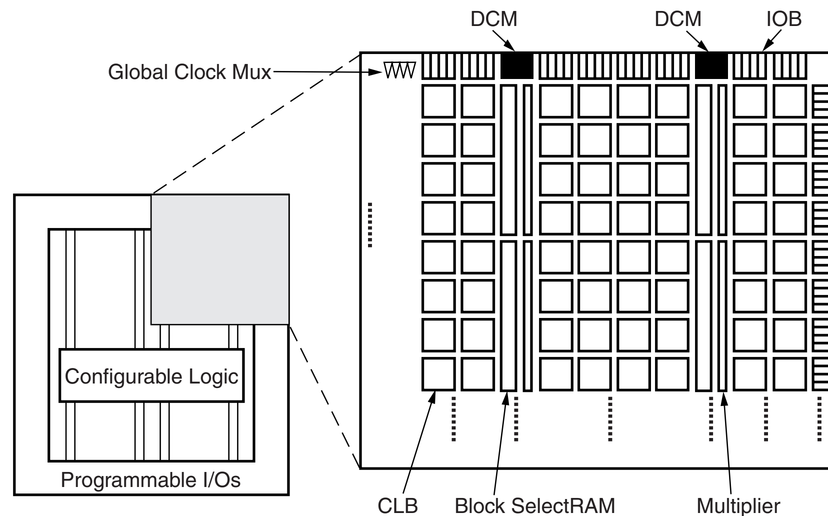


Abbildung 3.2: Struktureller Aufbau eines Virtex-II-FPGAs von Xilinx (nach [127]).

mehrere programmierbare Registerelemente (Flip-Flops). Typischerweise entspricht die Zahl der Register der Zahl von Funktionsgeneratoren. FPGAs sind demnach sehr reich an Registern, was sich direkt im Programmierstil von FPGA-Designs niederschlägt - dazu mehr in Abschnitt 3.1.3. Es sei hier im Voraus erwähnt, dass in einem FPGA-Design die maximale Signalverzögerung der kombinatorischen Logik zwischen Register-Ausgang (oder FPGA-Eingangspad) und Register-Eingang (oder FPGA-Ausgangspad) die Geschwindigkeit eines FPGA-Designs determiniert. Je kürzer diese maximale Verzögerung ist, desto höher kann die Taktfrequenz des Designs gewählt werden.

3.1.2 Details zur verwendeten FPGA-Serie Virtex-II

In diesem Abschnitt werden die wichtigsten Eigenschaften der verwendeten FPGA-Serie dargestellt. Es wird eine Übersicht über die FPGA-Architektur gegeben, wie sie zum Verständnis der Implementierung und Optimierung der in dieser Arbeit betrachteten Schaltungen notwendig ist. Obwohl moderne CAD-Software für das Schaltungsdesign eine Beschreibung der Hardware ohne genaue Kenntnisse der internen Struktur ermöglicht, ist es unerlässlich, diese Strukturen zu kennen, um ressourcenschonende Schaltungsvarianten entwickeln zu können. Deshalb wird besonders detailliert auf die Schaltungsmöglichkeiten für arithmetische Module eingegangen. Für eine tiefergehendere Betrachtung der FPGA-Technologie sei auf das Datenblatt [127] verwiesen. Zur Schaltungstechnik siehe auch [27].

Abbildung 3.2 zeigt den Aufbau eines Virtex-II FPGAs von Xilinx auf struktureller Ebene. Die IOB genannten Elemente (IOB bedeutet **I**nput-**O**utput-**B**lock), bilden die Schnittstellen zwischen der FPGA-internen Schaltungslogik und den Anschlusspins des Bausteins. Die IOBs können für 25 verschiedene elektrische Verbindungsstandards konfiguriert werden wie beispielsweise PCI, LVTTL, AGP-2X oder LVDS. Damit können diese FPGAs in einer Vielzahl von digitalelektronischen Umgebungen eingesetzt werden, was das Design von FPGA-basierten Ko-

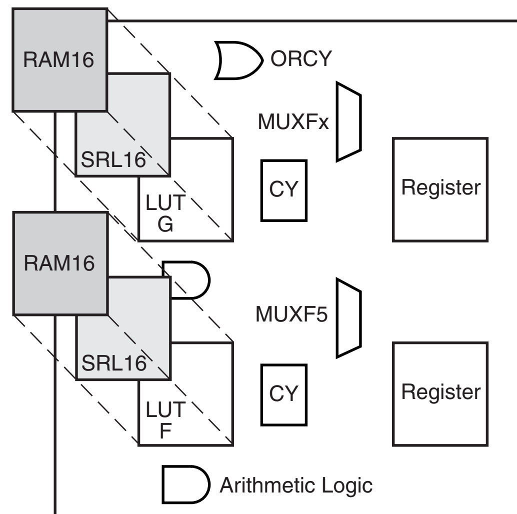


Abbildung 3.3: Struktur einer Slice-Einheit des Virtex-II-FPGAs (nach [127]).

prozessoren sehr vereinfacht. Weiter sind in Abb. 3.2 die als Quadrate dargestellten konfigurierbaren Logikblöcke (CLBs) zu sehen. Auf Details zu den CLBs wird weiter unten noch ausführlich eingegangen. Die Matrix der CLBs wird durchbrochen von zusätzlichen Modulen - den so genannten Block-RAMs und Multiplizierer-Elementen (auch Block-Multiplizierer genannt). Die Block-RAM-Elemente sind SRAM-Elemente, deren Eigenschaften wie Single-Port- oder Dual-Port-Modus, Bitbreite und Speichertiefe konfiguriert werden können. Je nach Größe des FPGAs gibt es vier bis 168 Block-RAM-Elemente mit jeweils 18 KBit Speicherkapazität. Sie bieten damit lokale Speicherressourcen, die weit über die Speicherfähigkeiten der CLBs hinausgehen. Sie eignen sich gut für die Implementierung von Zwischenspeichern für wenige hundert Datenworte. Die Multiplizierer-Elemente können 18-Bit-Integer-Zahlen multiplizieren. Da diese Multiplizierer durch festverdrahtete Logik aufgebaut sind, ist deren Flächenbedarf auf dem Chip weit geringer als bei gleichartigen Multiplizierern, die durch Verwendung von CLBs implementiert werden. Die Multiplizierer wurden in die Virtex-II-Bausteine integriert, um vor allem die Schaltungsmöglichkeiten für DSP-Anwendungen (**D**igital **S**ignal **P**rocessing), die das Hauptanwendungsgebiet für FPGAs darstellen, zu erweitern. In dieser Arbeit spielen diese Elemente eine wichtige Rolle für die Implementierung der arithmetischen Module.

Nun zurück zu den CLBs, welche die wichtigsten rekonfigurierbaren Ressourcen eines FPGAs sind. Jede CLB besteht aus vier gleichartigen Einheiten, den so genannten Slices. Um eine Vorstellung von der Logikressourcen aktueller FPGAs zu vermitteln sei an dieser Stelle angemerkt, dass der in dieser Arbeit verwendete FPGA (Xilinx XC2V3000) 14336 Slices enthält. Diese Slices sind innerhalb der CLBs durch besonders schnelle konfigurierbare Leitungen verbunden. Der Aufbau eines Slice wird in Abb. 3.3 schematisch umrissen. Es sind zwei Funktionsgeneratoren *G* und *F* mit jeweils vier Eingängen und zwei Register vorhanden. Die Elemente *G* und *F* können wahlweise als Look-Up-Tabellen, als serielle Shiftregister oder RAM konfiguriert werden. Mit einem Funktionsgenerator lässt sich eine beliebige logische Funktion von vier Eingängen bilden. Die beiden Multiplexer-Elemente *MUXFx* und *MUXF5* erlauben die Kombination der Funkti-

onseinheiten sowie die Verknüpfung logischer Zwischenergebnisse mit anderen Slices der CLB. Dadurch wird es möglich, in einem Slice beliebige logische Funktionen mit bis zu fünf Eingangssignalen, in einer CLB mit bis zu sieben Eingängen aufzubauen. Durch Einbeziehen der Nachbar-CLB können beliebige logische Funktionen mit 8 Eingängen erzeugt werden. Es lassen sich auch bestimmte logische Funktionen von mehr als sieben Eingängen in einer CLB erzeugen (z.B. logisches OR von 32 Eingängen). Die Kenntnisse über die Erzeugung von Logikelementen sind wichtig für die Optimierung von FPGA-Designs. So determiniert die Anzahl der beteiligten LUTs die Schaltungsverzögerung einer logischen Funktion. Ist eine hohe Schaltgeschwindigkeit erforderlich, sollten logische Verknüpfungen von mehr als 8 Signalen vermieden werden.

Zur Implementierung von schnellen arithmetischen Operatoren ist in den Slices zusätzliche Logik vorhanden, was durch die Symbole *ORCY*, *CY* und *Arithmetic Logic* angedeutet wird. Es handelt sich einerseits um eine Fast-Carry-Logik zur Beschleunigung von Schaltungen wie Integer-Addierern oder Zählern. So kann mit einer CLB ein sehr schneller 8-Bit-Ripple-Carry-Addierer aufgebaut werden. Die zusätzliche Logik ermöglicht außerdem die effiziente Implementierung von Multiplizier-Addier-Elementen. So kann in einer CLB eine 8-Bit-Zahl mit einem Bit multipliziert und danach zu einer zweiten 8-Bit-Zahl addiert werden. Dies ist wichtig für die Konstruktion von Multiplizierern und Akkumulatoren. Aufgrund der Bedeutung für diese Arbeit wird am Ende dieses Abschnitts detaillierter auf die Eigenschaften der CLBs bezüglich der Implementierung arithmetischer Operationen eingegangen.

Die Option, die Funktionsgeneratoren *F* und *G* als 16x1 Bit RAM-Elemente zu konfigurieren ist insbesondere für die Konstruktion von FIFO-Speichern wichtig. Dabei sind sowohl synchrone Single-Port als auch synchrone oder asynchrone Dual-Port RAM-Elemente möglich. Zur Unterscheidung von den Block-RAM-Elementen soll dieser Speicher im Folgenden CLB-RAM genannt werden. Die Funktionsgeneratoren können schließlich noch als Shiftregister-Elemente konfiguriert werden. Es handelt sich dabei um einen speziellen RAM-Modus. Diese Elemente haben dann das Verhalten, dass ein Eingangsbit synchron auf die Adresse 0x0 der CLB-RAM-Zelle geschrieben wird und in jedem Takt der Speicherinhalt um eine Speicherstelle weitergeschoben wird. Nach N Takten befindet sich ein geschriebenes Datenbit in der Speicherzelle $N-1$. Durch Lesezugriff auf eine Adresse 0x0..0xf erhält man somit eine Verzögerung um 1..16 Taktzyklen. Eine Verzögerungsschaltung für ein Bit um 16 Taktperioden kostet so nur einen Funktionsgenerator (1/8 CLB), während eine registerbasierte Implementierung 16 Flip-Flops (2 CLBs) aufwenden würde. Für Verzögerungsglieder werden deshalb vorzugsweise Shiftregister-Elemente eingesetzt.

Zur Illustration der komplexen Schaltungsmöglichkeiten der CLBs ist in Abb. 3.7 ein schematischer Schaltplan eines halben Slice dargestellt. Die Multiplexer des Schaltbilds, die ohne Select-Eingang dargestellt sind, schalten ein festes Signal durch, welches bei der Konfiguration des FPGAs festgelegt wird. Zu Details sei wiederum auf das Datenblatt der Virtex-II Bausteine verwiesen [127].

Wie oben erwähnt, wird an dieser Stelle noch etwas ausführlicher auf die Implementierung arithmetischer Funktionen durch die FPGA-Ressourcen eingegangen. Besonders wichtig sind hier Addierer. Für die Implementierung von Addierern gibt es zahlreiche Architekturen (siehe Abschnitt 4.1.2). Wichtigster Bestandteil für FPGA-Implementierungen sind jedoch meist Halbaddierer und Volladdierer, wie sie in Abbildung 3.4 dargestellt sind. Für die 1-Bit-Eingangssignale

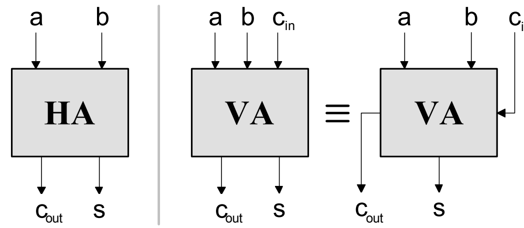


Abbildung 3.4: Halbaddierer (links) und Volladdierer (rechts).

a, b und c_{in}(Carry-Input für Volladdierer) und die resultierenden Signale s (Summe) und c_{out} (Carry Output) ergeben sich die folgenden Gleichungen:

$$\text{Halbaddierer:} \quad \begin{aligned} s &= a \oplus b \\ c_{out} &= a \cdot b \end{aligned} \tag{3.1}$$

$$\text{Volladdierer:} \quad \begin{aligned} s &= a \oplus b \oplus c_{in} \\ c_{out} &= (\overline{c_{in}} \cdot a \cdot b) + (c_{in} \cdot (a + b)) \,. \end{aligned} \tag{3.2}$$

Würde man diese Gleichungen durch jeweils zwei LUTs im FPGA umsetzen, betrüge der Ressourcenbedarf ein Slice pro Halbaddierer oder Volladdierer. Außerdem müssten in diesem Fall die Carry-Signale über allgemeine Routing-Ressourcen geleitet werden. Um diese Situation zu verbessern, sind in den meisten handelsüblichen FPGAs, so auch in den Virtex-II-FPGAs, zusätzliche Ressourcen zur schnellen Carry-Behandlung und -Verteilung vorhanden. Gleichung 3.2 kann folgendermaßen umformuliert werden, was sich leicht anhand der Logiktabelle 3.4 nachvollziehen lässt:

$$\begin{aligned} s &= a \oplus b \oplus c_{in} \\ c_{out} &= ((a \oplus b) \cdot c_{in}) + \left(\overline{(a \oplus b)} \cdot b\right) . \end{aligned} \tag{3.3}$$

c_{in}	a	b	$a \oplus b$	s	c_{out}
0	0	0	0	0	0
0	0	1	1	1	0
0	1	0	1	1	0
0	1	1	0	0	1
1	0	0	0	1	0
1	0	1	1	0	1
1	1	0	1	0	1
1	1	1	0	1	1

(3.4)

Gleichung 3.3 führt zu einer Implementierung eines Volladdierers, wie sie in Abbildung 3.5 gezeigt wird. Ist das Signal S des Multiplexers *MUXCY* auf logisch 1 (wenn genau eines der Signale a oder b gleich 1 ist, also $a \oplus b = 1$) wird c_{in} an c_{out} weitergeleitet (propagate), ansonsten wird c_{out} generiert ($a = b = 1$) oder gelöscht ($a = b = 0$). Die Carry-Signale der Slices

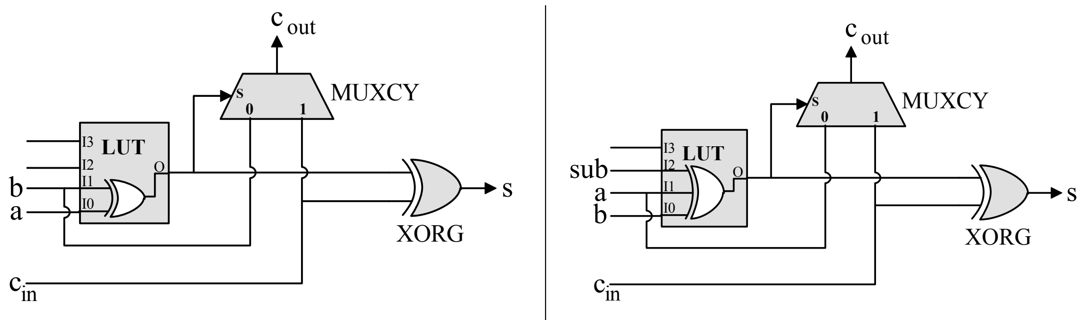


Abbildung 3.5: Implementierung eines Volladdierers (links) oder Addier/Subtrahier-Elements (rechts) durch 1/2 Slice eines Virtex-II-FPGA.

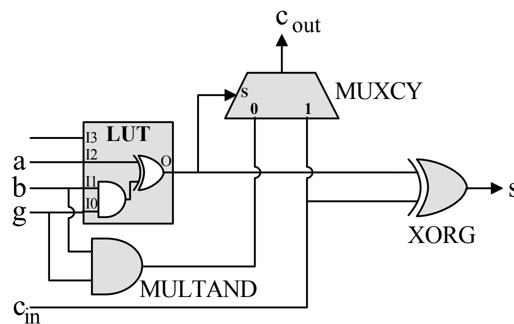


Abbildung 3.6: Implementierung eines Volladdierers mit Selektion des Eingangs *b* durch *g* mit 1/2 Slice eines Virtex-II-FPGA.

sind durch sehr schnelle Carry-Chains verbunden (siehe Abbildung 3.8). Damit können auf einfache und ressourcensparende Weise Ripple-Carry-Addierer mit sehr kurzen Laufzeiten für die Propagation der Carry-Signale erzeugt werden. Dies erübrigt in den meisten Fällen den Einsatz von schnelleren Addiererarchitekturen wie Carry-Save- oder Carry-Lookahead-Addierer. Mit dem gleichen Ressourcenaufwand können auch Addier/Subtrahier-Elemente aufgebaut werden, was die rechte Seite von Abbildung 3.5 zeigt. Diese Elemente werden genauso miteinander verbunden wie im Fall des Ripple-Carry-Addierers. Das Carry-Signal c_{in} des Addierers für das niedrigstwertige Bit muss dann mit dem Signal *sub* verbunden sein.

Für effiziente Umsetzungen von Multiplizierern und Akkumulatoren wurde in die Slices außerdem die MULTAND-Logik eingebaut. Damit lässt sich mit einem halben Slice die Addition von c_{in}, a und $(g \cdot b)$ erreichen, also die durch g (Gate) selektierte Addition von b zu a. Formel 3.5 beschreibt die erforderliche Schaltungslogik. Die resultierende Schaltung ist in Abbildung 3.6 dargestellt.

$$
\begin{aligned}
s &= a \oplus (b \cdot g) \oplus c_{in} \\
c_{out} &= (\overline{c_{in}} \cdot a \cdot b \cdot g) + (c_{in} \cdot (a + b \cdot g)) \\
&= ((a \oplus b \cdot g) \cdot c_{in}) + \left(\overline{(a \oplus b \cdot g)} \cdot (b \cdot g)\right)
\end{aligned} \tag{3.5}
$$

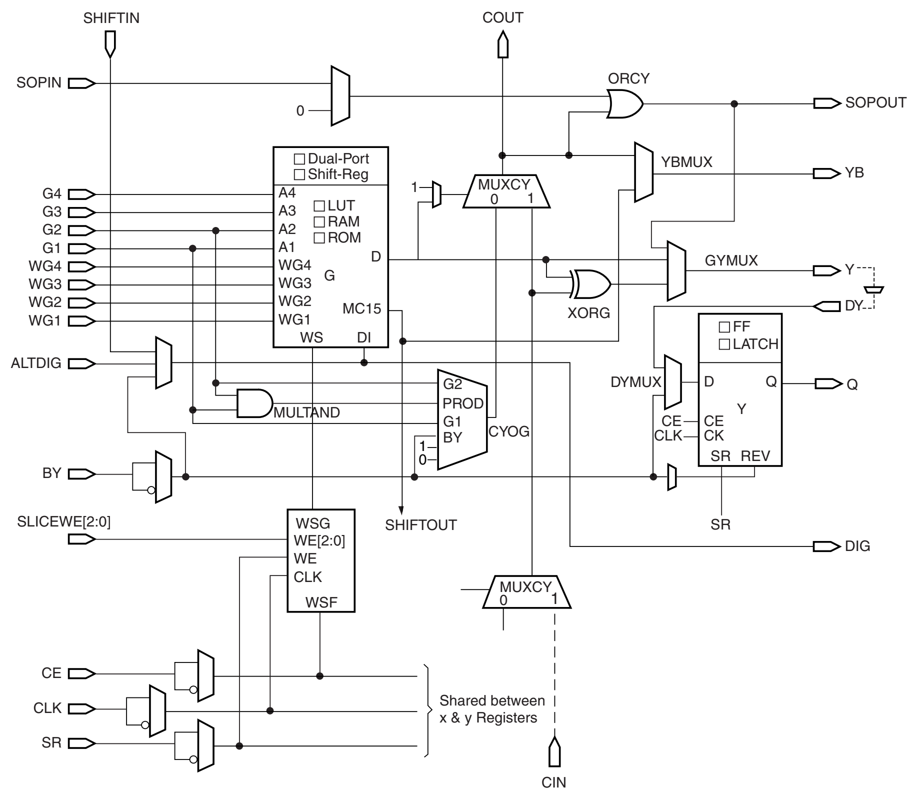


Abbildung 3.7: Schaltung einer halben Slice-Einheit des Xilinx Virtex-II (obere Hälfte). Das Element MUXFx von Abb. 3.3 ist hier nicht dargestellt (nach [127]).

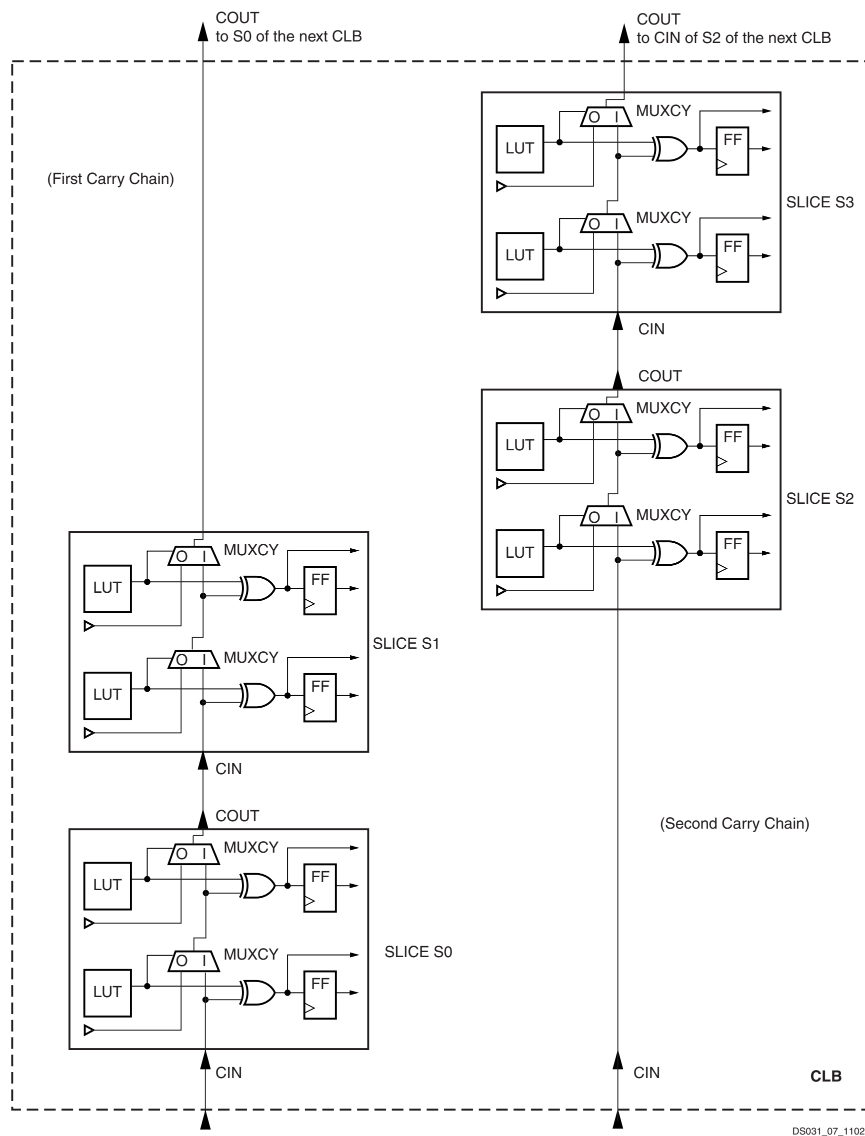


Abbildung 3.8: Verschaltung der Carry-Signale innerhalb einer CLB des Xilinx Virtex-II (nach [127]).

3.1.3 Programmierung von FPGAs

Es wurde oben bereits erwähnt, wie ein FPGA durch die Konfiguration bzw. Programmierung seiner Logikelemente und des Verbindungsnetzwerks seine Funktion als digitalelektronische Schaltung erhält. Zwischen FPGA-Design und FPGA-Konfiguration gibt es durch hochentwickelte Designwerkzeuge eine ähnlich hohe Abstraktion der Programmierung wie zwischen Hochsprachenprogrammierung und Maschinencode. Um zu verdeutlichen, wie in der Praxis FPGAs programmiert werden, soll hier kurz auf den Design-Flow programmierbarer Logik eingegangen werden.

Das Analogon zur Hochsprache bei der Mikroprozessorprogrammierung ist im FPGA-Design die Hardwarebeschreibungssprache (**H**ardware **D**escription **L**anguage, HDL). Besonders weit verbreitet sind die Sprachen *VHDL* und *Verilog*. Auf dieser Ebene geschieht das Schaltungsdesign mit den gleichen Techniken wie beim ASIC-Design, jedoch mit etwas unterschiedlichen Design-Kriterien, worauf weiter unten eingegangen wird. Diese Sprachen ermöglichen sowohl ein strukturelles Design der Schaltungen, also das hierarchische Zusammenbauen von Schaltungen aus gegebenen Bausteinen, als auch das verhaltensmäßige Beschreiben von Schaltungsbausteinen. Letzteres war ursprünglich für die reine Simulation von Schaltungsideen gedacht. Die aktuellen Designwerkzeuge unterstützen jedoch inzwischen bei den meisten verhaltensmäßig beschriebenen Konstrukten (bei allen, wenn gewisse syntaktische Regeln eingehalten werden) die Synthese in eine Schaltung. Da im programmierten FPGA die Fehlersuche nur begrenzt möglich ist, muss vor der Synthese eine gründliche Simulation auf HDL-Ebene durchgeführt werden. Die Transformation der in der HDL formulierten Schaltung in eine Netzliste, in der die schaltungstechnische Struktur festgehalten wird, wird meist Synthese genannt. Das Synthesewerkzeug führt bereits eine Abbildung der Schaltungslogik auf die elementaren Elemente des FPGAs durch, jedoch noch ohne die FPGA-Implementierung festzulegen. Dies ist Aufgabe des Place&Route-Werkzeugs, welches in der Regel vom Hersteller des verwendeten FPGAs bereitgestellt wird. Dort wird die Netzliste eingelesen, die Logikelemente werden auf die FPGA-Ressourcen platziert und die logische Verbindungsstruktur wird durch ein automatisches Routing-Verfahren auf die physikalischen FPGA-Verbindungselemente abgebildet.

Grundsätzlich werden, bis auf wenige Ausnahmen, FPGA-Schaltungen nach dem Prinzip der synchronen Verarbeitung entwickelt (synchrones Design). Hierbei werden sämtliche Signale auf ein oder mehrere Taktsignale bezogen. Die maximale Signallaufzeit zwischen zwei Registern, die synchron zu einem Takt getriggert werden, oder zwischen solchen Registern und den Ein- oder Ausgängen des FPGAs, bestimmt dann die maximale Frequenz dieses Taktes, bis zu der die Schaltung vorhersehbar korrekt arbeitet. Diese Vorhersehbarkeit ist nur beim synchronen Design gegeben, da im Allgemeinen die Funktion von asynchronen Designs von den relativen Verzögerungen der Logikpfade abhängen kann. Diese Signalverzögerungen in rein kombinatorischer Logik lassen sich jedoch durch die im Place&Route-Prozess unterschiedlich zugeordneten Verbindungslängen zwischen den Logikelementen nur schwer kontrollieren. Die Anwendung asynchroner Techniken ist deshalb sehr fehlerträchtig.

Während im VLSI-Design Register viel Chipfläche verbrauchen und deshalb nach Möglichkeit vermieden werden, sind FPGAs reich an Registern. Wie oben gesehen, ist in der Regel jeder LUT ein Register zugeordnet - es braucht also nicht an Registern gespart zu werden. Damit wird

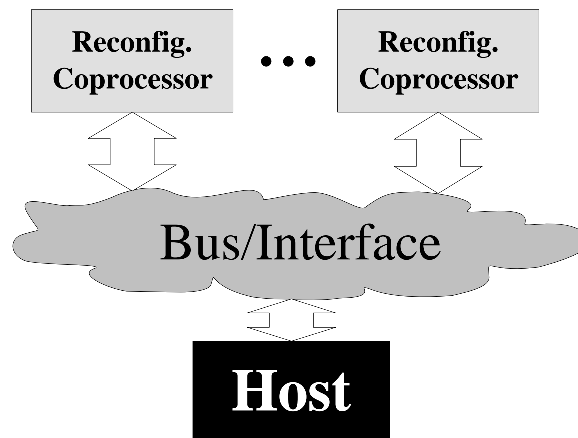


Abbildung 3.9: Grundlegende Systemarchitektur eines rekonfigurierbaren Rechensystems aus Steuerrechner (Host) und rekonfigurierbaren Koprozessoren.

die Technik des Pipelinings für das Design von FPGA-Schaltungen zu einem Schlüsselprinzip. Die Konstruktion einer Pipeline für eine Schaltung beruht darauf, die Verarbeitung in Schritte aufzuteilen, die zeitlich nacheinander ausgeführt werden können. Zwischen den Verarbeitungsschritten können Register eingeführt werden. Der Datendurchsatz verändert sich dadurch nicht, es entsteht jedoch eine Verzögerung der Ausgangssignale relativ zu den Eingangssignalen um eine Anzahl von Taktzyklen, die der Anzahl von Registerstufen entspricht.

Die maximale Taktfrequenz einer derart aufgebauten Schaltung wird durch die Signalverzögerung der Logik zwischen den Registern bestimmt. Ein tiefes Pipelining bedeutet eine feine Aufteilung der Schaltung in Verarbeitungsschritte mit kurzen Signalverzögerungen. Mit der Tiefe des Pipelinings steigt die erreichbare Taktfrequenz und damit die Geschwindigkeit des Designs. Dagegen vergrößert sich die Anzahl der Taktzyklen zwischen Signaleingang und Ausgang. Diese Zeit soll im Folgenden Latenzzeit (engl. Latency) oder Latenz genannt werden.

3.2 Allgemeine Systemarchitektur rekonfigurierbarer Plattformen

In diesem Abschnitt wird die Struktur von rekonfigurierbaren Plattformen umrissen. Es kann hier kein allgemeiner Überblick über alle Varianten des rekonfigurierbaren Rechnens gegeben werden. Der Abschnitt soll eher eine Idee von den möglichen Rechnerarchitekturen, die auf rekonfigurierbaren Systemkomponenten basieren, vermitteln. Einen weiterführenden Überblick geben beispielsweise [17], [117], [3] und [41].

3.2.1 Rekonfigurierbare Rechensysteme

Rekonfigurierbare Rechensysteme werden in der Regel als Hybridsysteme aus Standardcomputerplattform und rekonfigurierbaren Koprozessoren zusammengesetzt. Dabei bilden Mikroprozessorbasierte Computer den/die Steuerungsrechner (Host) für die rekonfigurierbare Plattform. Dieser Aufbau wird in Abb. 3.9 dargestellt. Die rekonfigurierbaren Systeme werden entweder direkt über ein Bus-System (z.B. PCI-Bus) oder indirekt über Schnittstellenkarten des Host mit diesem verbunden. Die rekonfigurierbaren Koprozessoren können jedoch darüber hinaus direkt untereinander verbunden sein.

Dieser hybride Aufbau aus Host und Koprozessoren ist aus folgenden Gründen zweckmäßig:

- FPGAs erhalten ihre Schaltungsfunktionalität erst durch die Programmierung mit einem Konfigurationsbitstrom. In der Regel geschieht die Konfigurierung durch einen Steuerrechner. Moderne FPGAs unterstützen zwar auch die Konfigurierung über ROM-Bausteine, allgemeiner verwendbare rekonfigurierbare Rechensysteme benötigen jedoch die Option, häufig und schnell die FPGA-Designs wechseln zu können.

- Das rekonfigurierbare Rechensystem muss mit Daten aus Standardquellen wie Festplatten oder Netzwerkressourcen versorgt werden. Die Nutzung von Standardcomputern ist die einfachste Möglichkeit, solche Datenquellen bereitzustellen.

- Eine rekonfigurierbare Plattform eignet sich nur zur Beschleunigung laufzeitkritischer Teile eines Algorithmus wie z.B. innere Schleifen eines Kraftberechnungsverfahrens. Simulationsalgorithmen beinhalten jedoch oft komplexe, aber laufzeitunkritische Programmteile, die besser über sequentielle instruktionsbasierte Ausführung in einem Mikroprozessor ausgeführt werden.

3.2.2 Rekonfigurierbare Koprozessoren

Den prinzipiellen Aufbau FPGA-basierter Koprozessoren zeigt Abb. 3.10. Die zentralen Bausteine sind ein oder mehrere FPGAs, die entweder durch direkte Punkt-zu-Punkt-Verbindungen kommunizieren oder über einen oder mehrere Busse Daten austauschen. Die FPGAs haben meist direkten Anschluss an Speicherressourcen. Viele rekonfigurierbare Koprozessoren verfügen überdies über weitere Elektronikressourcen wie z.B. Signalprozessoren sowie über Anschlussmöglichkeiten an periphere Hardware, beispielsweise durch Stecker für Aufsteckplatinen. Diese Ressourcen können durch einen Bus oder direkten Anschluss an einen FPGA in den Koprozessor eingebunden sein.

3.3 Verwendete rekonfigurierbare Plattform

Im laufe dieser Arbeit wurde mit vier verschiedenen FPGA-Plattformen gearbeitet. Der Hauptteil der Arbeit wurde jedoch mit der MPRACE-Plattform umgesetzt, weshalb hier nur diese beschrieben werden soll. Abb. 3.11 zeigt das MPRACE-Board [55]. Dieses System wurde 2001/2002

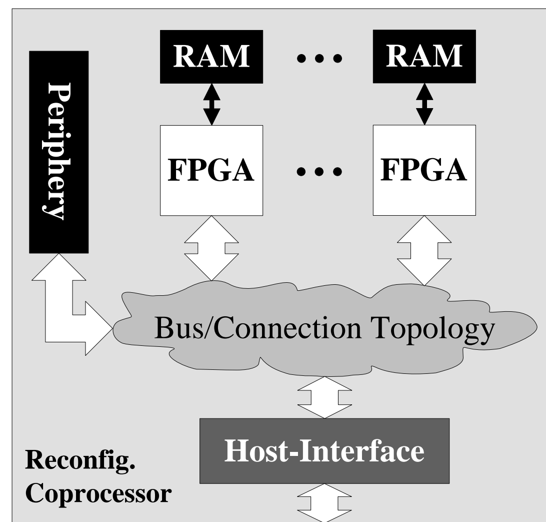


Abbildung 3.10: Grundlegende Architektur eines rekonfigurierbaren Koprozessors.

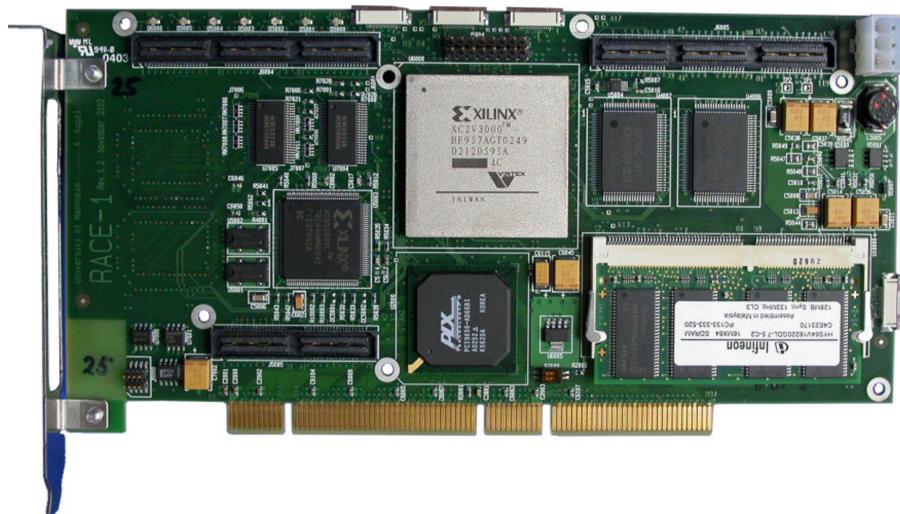


Abbildung 3.11: Der rekonfigurierbare Koprozessor MPRACE.

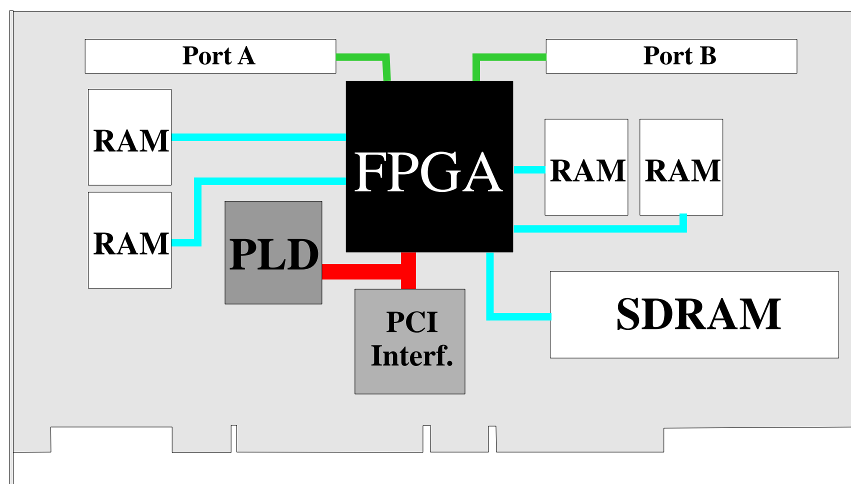


Abbildung 3.12: Schematischer Aufbau des MPRACE Koprozessors.

an der Universität Mannheim als Prototypplattform für verschiedene Anwendungen im Bereich der Hochenergiephysik, Bildverarbeitung und des Hochleistungsrechnens entwickelt. Es handelt sich um eine Koprozessorkarte die über den 64-Bit-PCI-Bus in einen handelsübliches PC-System eingebaut werden kann. In Abb. 3.12 ist der Aufbau der Karte schematisch dargestellt. Im Zentrum steht ein moderner und leistungsfähiger FPGA der Virtex-II-Serie von Xilinx. Die Boards können sowohl mit einem Virtex-XC2V3000-FPGA als auch mit einem Virtex-XC2V6000, der gegenüber dem XC2V3000 etwa die doppelte Zahl an Logikressourcen hat, bestückt werden. Zur Verdeutlichung der Logikkapazität sei erwähnt, dass der XC2V3000-FPGA über 28672 4-Input-LUTs und ebensoviele 1-Bit-Register[1] verfügt. Hinzu kommen 96 18×18-Bit-Multiplizierer und ebensoviele 18-KBit-Block-RAM-Elemente. Der FPGA ist mit vier voneinander unabhängigen SRAM-Modulen mit jeweils 36 Bit Datenbreite und einem Notebook-SDRAM-Speicherriegel verbunden. Zwei Erweiterungsstecker *PortA* und *PortB* öffnen den Koprozessor für den Anbau von Erweiterungskarten. Als PCI-Interface-Chip wurde der PCI-9656 von PLX verwendet. Der Datenaustausch zwischen PCI-Interface und FPGA geschieht über einen lokalen 32-Bit-66-MHz-Bus. Die Übertragungsbandbreite zum FPGA ist demnach maximal 264 MByte/sec. Über den PLD erfolgt die Konfiguration des Boards. So ermöglicht dieser Baustein unter anderem die Kontrolle der FPGA-Konfigurierung und das Clock-Management auf dem Board.

Der sich auf dem MPRACE-System befindliche FPGA von Xilinx wurde ab Herbst 2001 kommerziell vertrieben. Die Reihe der Virtex-II-FPGAs war die erste Serie von FPGAs mit Block-Multiplizierer-Ressourcen. Erst damit wurde es möglich, über ein PCI-basiertes System in der Art des MPRACE-Boards ein rekonfigurierbares Rechensystem in der Komplexität aufzubauen, wie es für diese Arbeit geschehen ist.

[1] Genau genommen sind es noch mehr, wenn man die Register in den IO-Zellen hinzurechnet.

Kapitel 4

Grundlagen der Computerarithmetik

In diesem Kapitel wird der Stand der Forschung zur Computerarithmetik zusammengefasst, jedoch konzentriert auf die Grundlagen, die für die Implementierung von Rechenwerken im Rahmen dieser Arbeit benötigt werden. Alternative Verfahren werden nur kurz umrissen, um eine Einordnung in das weite Feld der Architekturen für arithmetische Operationen zu ermöglichen. Das Kapitel ist entsprechend den wichtigsten Darstellungsmöglichkeiten von Zahlen unterteilt in die Abschnitte zu Ganzzahlen, Festkommazahlen, Gleitkommazahlen und logarithmischen Zahlen. Im Abschnitt für Ganzzahlen werden auch kurz Restklassensysteme und im Abschnitt über logarithmische Zahldarstellungen die Klasse der semilogarithmischen Zahlensysteme umrissen. Die Eigenschaften der verschiedenen Zahlenformate werden herausgestellt und die Auswirkungen der verschiedenen Darstellungsmöglichkeiten auf die Implementierung von Operatoren erläutert. Dabei wird die Implementierung von Operationen auf ganzen Zahlen in Abschnitt 4.1 besonders eingehend betrachtet, da diese den Kern der Implementierung von Operationen – auch in anderen Zahlensystemen – bilden. Die elementaren Operationen auf Festkommazahlen werden in Abschnitt 4.2 auf die Verarbeitung von Ganzzahlen zurückgeführt. Darauf aufbauend wird in Abschnitt 4.3 die Verarbeitung von Gleitkommazahlen erläutert. Als Alternativen zur Gleitkommaarithmetik werden in Abschnitt 4.4 die Arithmetik auf logarithmischen und semilogarithmischen Zahlen diskutiert. Bis zu dieser Stelle beschränken sich die Beschreibungen auf Implementierungskonzepte der elementaren Operatoren. In Abschnitt 4.5 werden ergänzend dazu verschiedene Methoden zur Berechnung von Funktionen einer Variablen ausgeführt.

Die vorgestellten Rechenverfahren sind inzwischen weitgehend Standard und werden in der Auswahl und Formulierung präsentiert, wie sie für die Implementierung auf FPGAs Verwendung findet. Einen guten Überblick über die gebräuchlichen Techniken der Computerarithmetik gibt das Buch [89]. Weitere Standardwerke zur Computerarithmetik sind [114], [53], [101], [125], [104] und [56].

Das Kapitel schließt mit einem Überblick über den Stand der Forschung zur Implementierung der Arithmetik für wissenschaftliches Rechnen auf FPGAs in Abschnitt 4.6.

4.1 Ganzzahlen

4.1.1 Darstellungsmöglichkeiten ganzer Zahlen

Die elementare Form, ganze Zahlen darzustellen, ist die eines Stellen- oder Positionssystems von k Ziffern x_i, deren Wertigkeit Potenzen einer Basiszahl (Radix) r sind, wie in Gleichung 4.1 ausgedrückt wird.

$$(x_{k-1}\cdots x_0)_r = \sum_{i=0}^{k-1} x_i r^i,\ \ x_i \in [-\alpha,\beta] = \{-\alpha,-\alpha+1,\ldots,\beta-1,\beta\}. \tag{4.1}$$

Beispielsweise handelt es sich bei $(r=2,\alpha=0,\beta=1)$ um das Dual- oder Binärsystem, bei $(r=10,\alpha=0,\beta=9)$ um das Dezimalsystem und bei $(r=16,\alpha=0,\beta=15)$ um Hexadezimalzahlen. Für $(\beta-\alpha\geq r)$ ergeben sich redundante Zahlendarstellungen, bei denen Zahlwerte mit unterschiedlichen Ziffernkombinationen dargestellt werden können. Im Bereich der Digitalelektronik sind die Binärzahlen besonders ausgezeichnet, da die elektronische Repräsentation von Zahlen als Folge von Bits die Interpretation der Bits als Binärziffern nahe legt. Zahldarstellungen mit höherer Basis spielen jedoch eine wichtige Rolle bei der elektronischen Implementierung von schnellen Operatoren auf Ganzzahlen. Hier werden die Ziffern jeweils durch mehrere Bits kodiert. Die folgende Diskussion verschiedener Darstellungsmöglichkeiten ganzer Zahlen bezieht sich auf Erweiterungen der elementaren Zahldarstellung nach Gleichung 4.1, um auch negative Zahlen elektronisch darstellen zu können und die elementaren arithmetischen Operationen in diesen Zahlensystemen zu optimieren. In dieser Arbeit werden ganze Zahlen gelegentlich auch mit dem englischen Begriff *Integer* und die Untermenge der positiven ganzen Zahlen mit *Unsigned-Integer* bezeichnet. Am Ende dieses Abschnitts wird kurz auf Restklassensysteme eingegangen, welche sich grundsätzlich von den Positionssystemen unterscheiden.

Vorzeichen-Betrag-Darstellung

Bei dieser Darstellung wird zum Zahlenformat von Gleichung 4.1 eine Vorzeicheninformation hinzugefügt. Im Fall von Binärzahlen wird dem höchstwertigen Bit x_{k-1} ein Vorzeichenbit s vorangestellt:

$$(s\,x_{k-1}\cdots x_0)_r = (-1)^s \cdot \sum_{i=0}^{k-1} x_i r^i. \tag{4.2}$$

Vorteile dieser Darstellung sind die konzeptuelle Einfachheit, der symmetrische Umfang des darstellbaren Zahlenbereichs und die einfache Operation des Invertierens, indem lediglich das Vorzeichenbit zu negieren ist. Die grundlegende Operation Addition ist dagegen aufwändiger zu realisieren als für Zahlen in Komplement-Darstellung, wie weiter unten gezeigt wird.

Darstellungen mit Bias

Eine weitere Möglichkeit, negative Zahlen darzustellen, besteht darin, auf alle darzustellenden Zahlen x einen konstanten Wert *bias* hinzuzuaddieren, sodass das Resultat positiv ist. Die kleinste

darstellbare Zahl ist demnach $-bias$. Die größte darstellbare Zahl reduziert sich gegenüber der Darstellung ohne Bias um $bias$. Darstellungen mit Bias führen zu einem leicht erhöhten Aufwand für die Addition und Subtraktion, wie man an Gleichung 4.3 sehen kann (a und b sind Zahlen in der Darstellung mit Bias) . Der Wert für $bias$ kann jedoch so gewählt werden, dass der Aufwand für die zusätzliche Addition oder Subtraktion eines Bias vernachlässigbar gering ist (z.B. 2^{k-1} für Dualzahlen mit k Stellen).

$$\begin{aligned}
\underbrace{x+y+bias}_{a+b} &= \underbrace{(x+bias)}_{a}+\underbrace{(y+bias)}_{b}-bias \\
\underbrace{x-y+bias}_{a-b} &= \underbrace{(x+bias)}_{a}-\underbrace{(y+bias)}_{b}+bias.
\end{aligned} \tag{4.3}$$

Multiplikation und Division sind hier wesentlich komplexer als bei anderen Ganzzahlrepräsentationen. Diese Darstellung wird deshalb nur in Spezialfällen eingesetzt, wo nur Addition und Subtraktion benötigt und durch die spezielle Wahl für $bias$ einfache Operationen und Größenvergleiche möglich werden. Die Darstellung mit Bias wird beispielsweise für die Exponenten von Gleitkommazahlen verwendet.

Komplement-Darstellungen

In der Komplement-Darstellung werden negative Zahlen wie bei der Darstellung mit Bias in positive Zahlen überführt, indem eine hinreichend große Konstante M addiert wird. Positive Zahlen bleiben dagegen unverändert. Um die Überschneidung von positiven und negativen Zahldarstellungen zu vermeiden, muss $M \geq N+P+1$ gelten, wobei P und N die betragsmäßig größten darstellbaren positiven und negativen Zahlen sind. Für $M = N+P+1$ ergibt sich die maximale Codierungseffizienz der Darstellung. In der Komplement-Darstellung werden Additionen unabhängig vom Vorzeichen der Argumente als Unsigned-Integer-Addition der Komplement-Zahlen modulo M durchgeführt. Dies funktioniert, da in der Modulo-M-Arithmetik die Addition von $M-x$ identisch mit der Subtraktion von x ist. Subtraktionen werden erzeugt, indem vom Subtrahenden das Komplement gebildet wird.

Im Spezialfall einer Darstellung zur Basis $r = 2$ mit k Binärstellen ergeben sich die Zweierkomplement- und Einserkomplement-Darstellung auf folgende Weise.

1er-Komplement-Darstellung Hier wird $M = 2^k - 1$ gewählt. Dies führt für die Berechnung von $M - |x|$ zur Darstellung einer Zahl $x < 0$ zur einfachen Rechenvorschrift, alle Bits von x zu invertieren. Die Addition zweier 1er-Komplement-Zahlen a und b modulo M nimmt folgende Gestalt an:

$$\begin{aligned}
(a+b)\ mod\ M &= \begin{cases} a+b+1 & : \quad (a+b) > 2^k-1 \\ a+b & : \quad (a+b) \leq 2^k-1 \end{cases} \\
&= \begin{cases} a+b+1 & : \quad (a+b) \geq 2^k \\ a+b & : \quad (a+b) < 2^k. \end{cases}
\end{aligned} \tag{4.4}$$

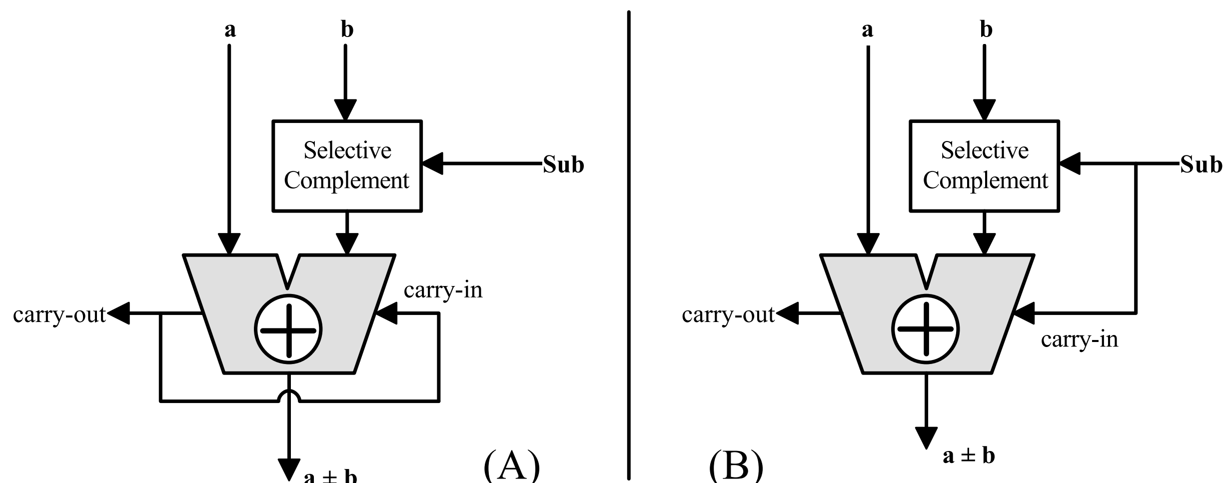


Abbildung 4.1: Aufbau eines Addier/Subtrahier-Operators für Zahlen in der 1er-Komplement-Darstellung (A) und 2er-Komplement-Darstellung (B).

Der Wert 2^k entspricht dem Carry-Out einer Hardwareschaltung des Addierers. Die Fallunterscheidung bedeutet in Hardware, dass ein Carry-Out sofort in das Carry-In des Addierers zurückgeführt wird, was auch *End-Around Carry* genannt wird. Es sei noch erwähnt, dass für den Ergebniswert 0 die Darstellung $(1\,1 \cdots 1)_2$ auftreten kann, was als zweite Repräsentation der 0 interpretiert werden kann. Abbildung 4.1 zeigt auf der linken Seite den schematischen Aufbau eines kombinierten Addier/Subtrahier-Operators für 1er-Komplement-Zahlen.

2er-Komplement-Darstellung In dieser Darstellung wird $M = 2^k$ gewählt. Die Addition modulo M kann durch einen einfachen Unsigned-Integer-Addierer umgesetzt werden, bei dem das Carry-Out mit der Wertigkeit 2^k ignoriert wird. Die Vorschrift, bei der Subtraktion vom Subtrahenden b das Komplement zu bilden, kann zerlegt werden in die Erzeugung des 1er-Komplements $\bar{b}$ und der Addition von 1. Letzteres kann ohne zusätzlichen Hardwareaufwand durch ein Carry-In in einen Unsigned-Integer-Addierer erfolgen. Abbildung 4.1 zeigt auf der rechten Seite die Realisierung eines kombinierten Addier/Subtrahier-Operators.

Redundante Darstellung

Redundante Darstellungen von Zahlen ergeben sich wie oben erwähnt aus der Wahl eines Satzes von Ziffern, für den entsprechend der Notation in Formel 4.1 die Eigenschaft $\beta - \alpha \geq r$ gilt. Hauptmotivation für die Einführung dieser Zahlendarstellung ist, dass damit Addierer realisiert werden können, die ohne Carry-Fortpflanzung über viele Logikstufen hinweg auskommen. Dazu ist grob gesagt die Redundanz hoch genug zu wählen, damit bei der Addition Stellenüberträge in die nachfolgende Stelle dort aufgefangen werden können, ohne dabei einen erneuten Übertrag hervorzurufen. Da Additionen grundlegender Bestandteil der Implementierung aller anderen arithmetischen Grundoperationen sind, können diese Operationen mit Hilfe redundanter Zahlenformate ebenfalls durch Schaltungen mit verkürzten Carry-Ketten erzeugt werden. Redundante

Zahlendarstellungen gehen einher mit einem erhöhten Aufwand für die Speicherung und den Datentransport und einem hohen Aufwand der Konvertierung in nicht-redundante Darstellungen. Die Vor- und Nachteile redundanter Zahlendarstellungen hängen stark von den Eigenschaften der physikalischen Schaltungstechnik und der Art der Operatoren ab. Für Implementierungen auf FPGA-basierten Systemen ist die Verwendung redundanter Zahldarstellungen wenig verbreitet.

Restklassensysteme (Residue Number Systems)

Ganze Zahlen lassen sich in einem endlichen Zahlenbereich eindeutig durch Reste x_i darstellen, welche bei der Division durch ganzzahlige Moduln m_i entstehen. Voraussetzung ist, dass die m_i relativ prim zueinander sind. Der Zahlenbereich ist bei der Darstellung von positiven Zahlen durch $0 \leq x < \prod_{i=1}^{n} m_i$ gegeben, wenn n die Zahl der Moduln ist. Restklassensysteme (engl. **R**esidue **N**umber **S**ystems = RNS) erlauben eine übertragsfreie Berechnung von Summen und Produkten, da Addition und Multiplikation stellenweise modulo m_i durchgeführt werden können. Die Grundoperation Division ist dagegen sehr schwierig durchzuführen (siehe z.B. [44]) und für die Berechnung der Quadratwurzel existiert kein Verfahren. Überdies sind Größenvergleiche nicht direkt möglich. Während die Überführung einer Zahl von einem Positionssystem in ein Restklassensystem leicht umzusetzen ist, ist die Rückumwandlung in ein Stellensystem sehr aufwändig. Die Implementierung von Rechenwerken basierend auf Restklassensystemen ist vor allem im Bereich der digitalen Signalverarbeitung verbreitet, wo bei vielen Aufgabenstellungen ausschließlich Multiplikationen und Additionen durchgeführt werden müssen (z.B. FIR-Filter [95], Wavelet Transformation [96]). Es lassen sich dann sehr schnelle und ressourcensparende Rechenwerke aufbauen. Aufgrund der genannten Nachteile wurden Restklassensysteme für diese Arbeit nicht weiter in Betracht gezogen.

4.1.2 Additionsalgorithmen

Im letzten Abschnitt wurde bereits die Implementierung von Addierern für Zahlen in der 1er-Komplement- und 2er-Komplement-Darstellung allgemein skizziert, ohne auf die Hardware für die Unsigned-Integer-Addierer einzugehen. Bevor die Schaltungstechnik dieser elementaren Addiererelemente beschrieben wird, soll noch auf die Implementierung eines Additions/Subtraktions-Operators für Zahlen in der Vorzeichen-Betrags-Darstellung eingegangen werden. Den Aufbau eines solchen Operators zur Berechung von $a \pm b$ (ein Signal Sub zeigt an, ob subtrahiert werden soll) zeigt Abbildung 4.2.

Diese Schaltung wird hier ausführlich erklärt, da sie in ähnlicher Form auch Bestandteil der Gleitkommaaddierer ist. Kern der Architektur ist ein Unsigned-Integer-Addierer, der, wie oben bereits beschrieben, geeignet ist, um Zahlen in der Komplement-Darstellung zu addieren. Unterscheiden sich die Argumente a und b im Vorzeichen und ist $Sub = 0$ oder haben a und b das gleiche Vorzeichen und ist $Sub = 1$, gilt die Beziehung $|a \pm b| = ||b| - |a||$. Entsprechend wird in diesem Fall das Komplement der Zahl a berechnet. Wird das 2er-Komplement verwendet, kann dies dadurch geschehen, dass $|a|$ negiert wird und das Carry-In des Addierers auf 1 gesetzt wird. Das höchstwertige Bit nach der Addition gibt das Vorzeichen des Resultats an. Bei Vorliegen einer negativen Zahl wird erneut das Komplement davon gebildet, um $|a \pm b|$ zu erhalten. Das

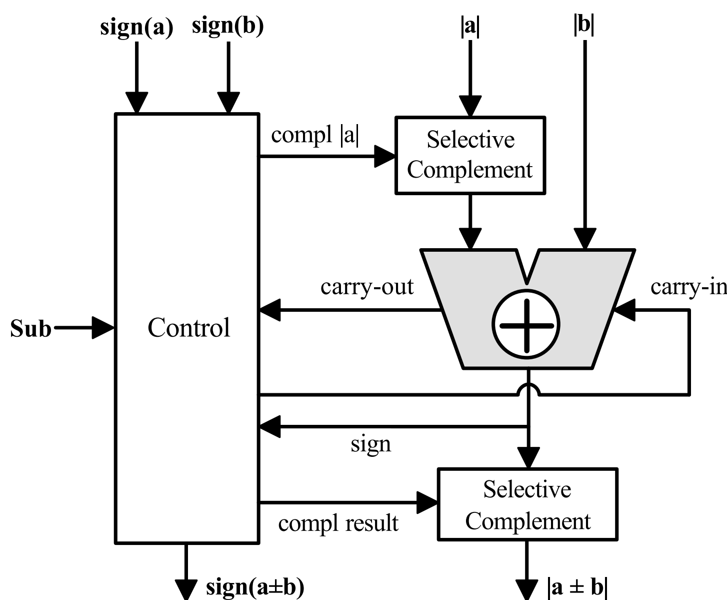


Abbildung 4.2: Aufbau eines Addier/Subtrahier-Operators für Zahlen in der Vorzeichen-Betrag-Darstellung.

Vorzeichen des Ergebnisses kann aus den Vorzeichen von a und b, dem Signal *Sub* und dem Signal *sign* ermittelt werden.

Für die elementare Unsigned-Integer-Addition sollen hier die wichtigsten Prinzipien kurz vorgestellt werden. Welches Verfahren bei einer Anwendung gewählt wird, hängt vor allem von der verwendeten Technologie ab. In der Regel müssen die konkurrierenden Faktoren Geschwindigkeit und Ressourcenbedarf gegeneinander abgewogen werden.

Ripple-Carry-Addierer

Die einfachste Weise, elementare Volladdierer zu N-Bit-Addierern zusammenzuschalten, besteht im Aufbau eines Ripple-Carry-Addierers. Hier werden die c_{out}-Signale der Volladdierer (Übertragsausgang) in die Eingänge c_{in} der nächsthöheren Stufe gegeben. Dieses Additionsschema ergibt sich aus folgender, in logische Operationen auf den Binärstellen der Argumente aufgeschlüsselten mathematischen Formulierung für $s = a + b + c_{in}$:

$$\begin{aligned}
(s_{k-1} \cdots s_0) &= (a_{k-1} \cdots a_0) + (b_{k-1} \cdots b_0) + (0 \cdots 0\, c_{in}) \\
s_i &= a_i \oplus b_i \oplus c_i \\
g_i &= a_i \cdot b_i \\
p_i &= a_i \oplus b_i \\
c_{i+1} &= g_i + c_i \cdot p_i, \quad c_0 = c_{in}, \; c_{out} = c_k.
\end{aligned} \tag{4.5}$$

Die Signale g_i und p_i geben an, dass in Stufe i ein Carry-Signal c_{i+1} generiert ($g_i = 1$) oder propagiert ($p_i = 1$) wird. Diese Formulierung für die Carry-Signale wird sich im nachfolgenden Abschnitt als nützlich erweisen.

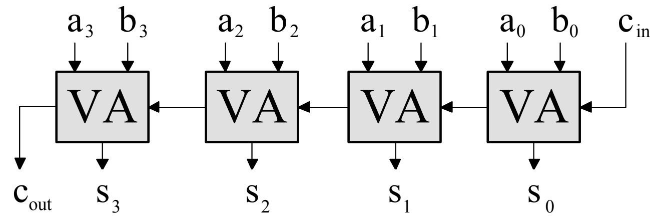


Abbildung 4.3: Addierer für 4-Bit-Ganzzahlen als Ripple-Carry-Schaltung.

Abbildung 4.3 zeigt einen solchen Addierer für binäre 4-Bit-Zahlen. Die Signallaufzeit verhält sich bei dieser Addiererkette linear zur Breite des Addierers.

Carry-Lookahead-Addierer

Carry-Lookahead-Addierer ergeben sich aus dem Prinzip, die Gleichung 4.5 für c_i abzurollen, was in folgender Formel beschrieben wird:

$$\begin{aligned}
c_i &= g_{i-1} + c_{i-1} \cdot p_{i-1} \\
&= g_{i-1} + g_{i-2} \cdot p_{i-1} + c_{i-2} \cdot p_{i-2} \cdot p_{i-1} \\
&= g_{i-1} + g_{i-2} \cdot p_{i-1} + g_{i-3} \cdot p_{i-2} \cdot p_{i-1} + c_{i-3} \cdot p_{i-3} \cdot p_{i-2} \cdot p_{i-1} \\
&= \underbrace{g_{i-1} + g_{i-2} \cdot p_{i-1} + g_{i-3} \cdot p_{i-2} \cdot p_{i-1} + g_{i-4} \cdot p_{i-3} \cdot p_{i-2} \cdot p_{i-1}}_{g_{[i-4,i-1]}} \\
&\quad + c_{i-4} \cdot \underbrace{p_{i-4} \cdot p_{i-3} \cdot p_{i-2} \cdot p_{i-1}}_{p_{[i-4,i-1]}} \\
&\vdots
\end{aligned} \tag{4.6}$$

Für einen 4-Bit-Addierer ergeben sich somit folgende Gleichungen:

$$\begin{aligned}
c_1 &= g_0 + c_0 \cdot p_0 \\
c_2 &= g_1 + g_0 \cdot p_1 + c_0 \cdot p_0 \cdot p_1 \\
c_3 &= g_2 + g_1 \cdot p_2 + g_0 \cdot p_1 \cdot p_2 + c_0 \cdot p_0 \cdot p_1 \cdot p_2 \\
c_4 &= \underbrace{g_3 + g_2 \cdot p_3 + g_1 \cdot p_2 \cdot p_3 + g_0 \cdot p_1 \cdot p_2 \cdot p_3}_{g_{[0,3]}} \\
&\quad + c_0 \cdot \underbrace{p_0 \cdot p_1 \cdot p_2 \cdot p_3}_{p_{[0,3]}} .
\end{aligned} \tag{4.7}$$

Abbildung 4.4 zeigt die entsprechende Schaltung für einen solchen **L**ookahead-**C**arry-**G**enerator (LCG). Die Ergebnisbits ergeben sich unter Verwendung des LCG durch die einfache Beziehung $s_i = p_i \oplus c_i$. Die Latenz eines damit erzeugten 4-Bit-Addierers entspricht der Verzögerung von vier Logikgattern (jeweils eines zur Erzeugung von g_i und p_i, zwei Gatter für die Generierung von c_i und ein Gatter für s_i). Die Lookahead-Technik wird vor allem deshalb interessant, da die LCG-Elemente in einer Baumstruktur zu größeren LCG-Bausteinen gruppiert werden können. Abbildung 4.5 zeigt, wie aus fünf 4-Bit-LCG-Elementen ein 16-Bit-LCG erzeugt werden

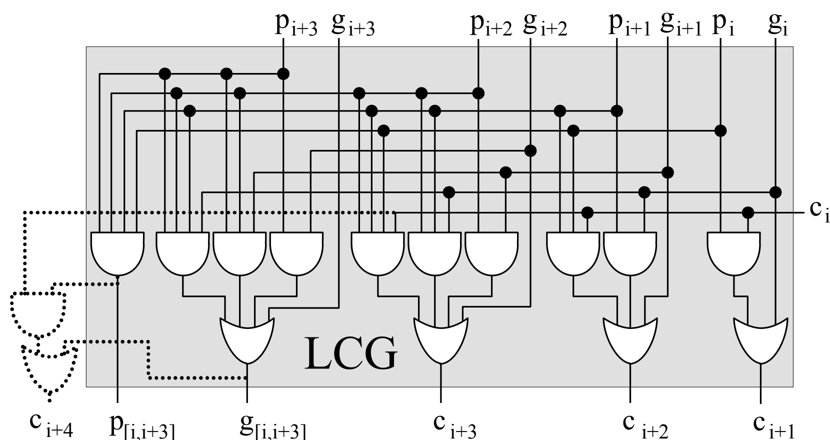


Abbildung 4.4: Lookahead-Carry-Generator (LCG) für die Erzeugung der Carry-Signale aus den Generate- (g) und Propagate-Signalen (p) aus den Summanden bzw. aus der vorausgehenden Ebene eines LCG-Baumes. Die Signale $g_{[i,i+3]}$ und $p_{[i,i+3]}$ können in die nächsthöhere Ebene eines LCG-Baumes gegeben oder im Fall der Wurzel eines solchen Baumes zur Erzeugung des Carry-Out-Signals der Schaltung verwendet werden (gepunktete Elemente).

kann. Zusammen mit den Logikgattern zur Erzeugung der Signale p_i, g_i und s_i ergibt sich der dargestellte 16-Bit-Addierer. Vier solche Addierer können mit einem weiteren LCG-Element zu einem 64-Bit-Addierer verbunden werden. Für jede weitere Stufe des LCG-Baumes erhöht sich die Latenz um die Verzögerung von 4 Logikgattern (2 Gatter für die Erzeugung der Signale $g_{[\cdot,\cdot]}$ und $p_{[\cdot,\cdot]}$ für die nächste Stufe und 2 Gatter für die Rückgabe der Carry-Signale an die vorausgehende Stufe). Die Baumstruktur der LCG-Elemente führt damit dazu, dass sich die Latenz des Addierers logarithmisch zu dessen Breite verhält, im Gegensatz zum linearen Anstieg bei Ripple-Carry-Addierern.

Es sei hier nur erwähnt, dass es optimierte Varianten zum eben beschriebenen Verfahren für Carry-Lookahead-Addierer gibt. Beispielsweise ergibt sich der Ling-Addierer aus einer modifizierten Formulierung, die für eine VLSI-Implementierung zur Ersparnis von Logikgattern gegenüber den hier gezeigten Schaltungen führt.

Carry-Save-Addierer

Ein Carry-Save-Addierer ist eine 3-2-Reduktionsschaltung, welche drei Eingangszahlen zu zwei Ausgangwerten reduziert. Dies ist mit einer einfachen Aneinanderreihung von Volladdierern möglich, wie in Abbildung 4.6 gezeigt. Die Latenzzeit ist durch die Signalverzögerung in einem der Volladdierer gegeben. Solche Addierer werden vorwiegend in Addiererbäumen eingesetzt, wo viele Argumente summiert werden müssen. Die internen Signale in Carry-Save-Darstellung können als redundante Zahlen verstanden werden. Zur Rückgewinnung von Binärzahlen am Ende des Baumes wird ein Carry-Propagate-Addierer erforderlich, wie z.B. ein Ripple-Carry- oder Carry-Lookahead-Addierer.

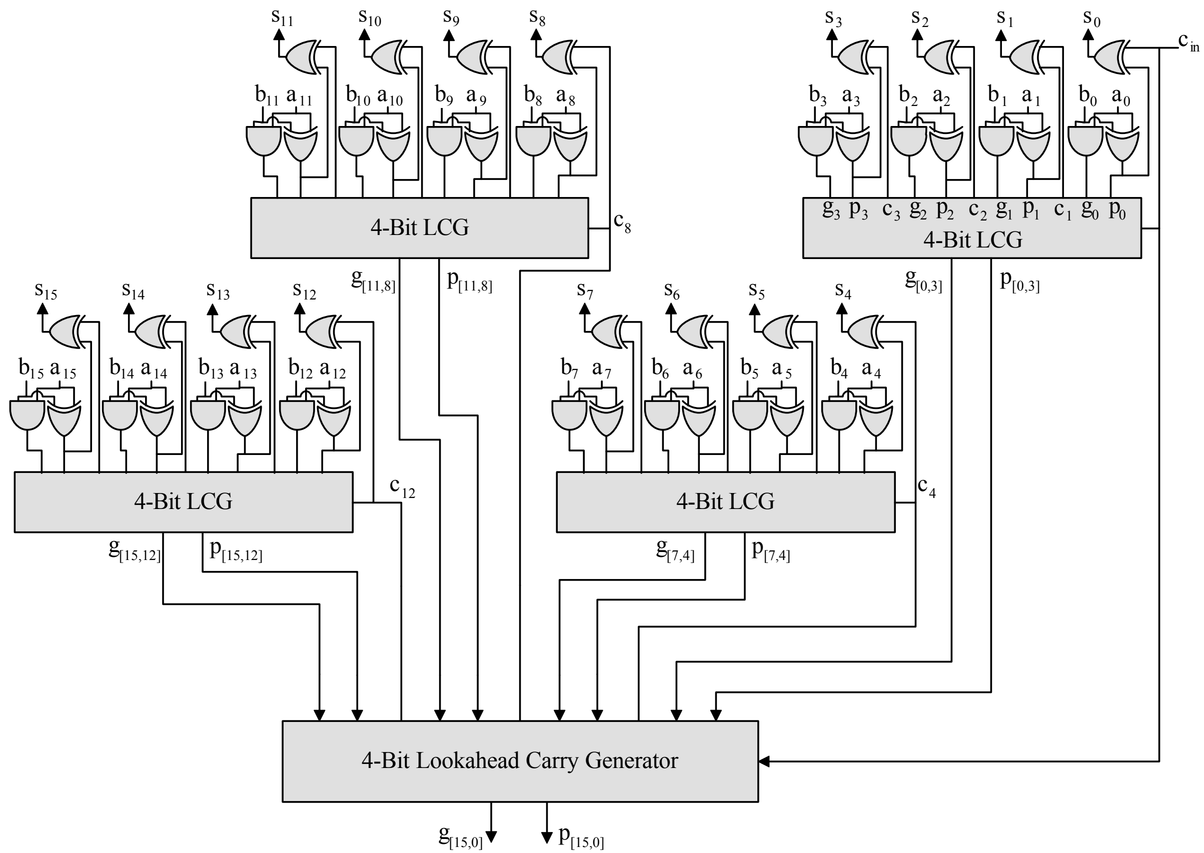


Abbildung 4.5: Carry-Lookahead-Addierer für 16-Bit-Ganzzahlen unter Verwendung eines zweistufigen LCG-Baumes (5 LCG-Elemente entsprechend Abbildung 4.4).

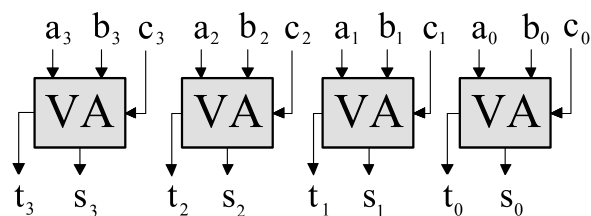


Abbildung 4.6: Addierer für Ganzzahlen als Carry-Save-Schaltung.

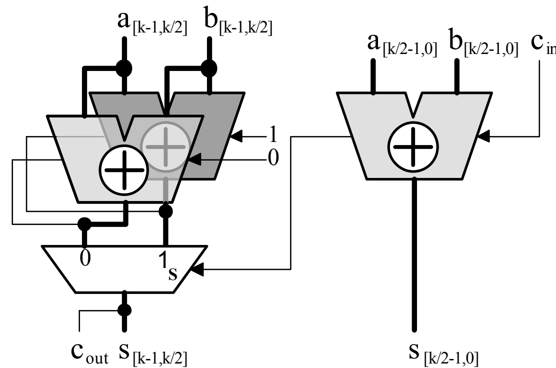


Abbildung 4.7: Addierer für Ganzzahlen als einstufige Carry-Select-Schaltung.

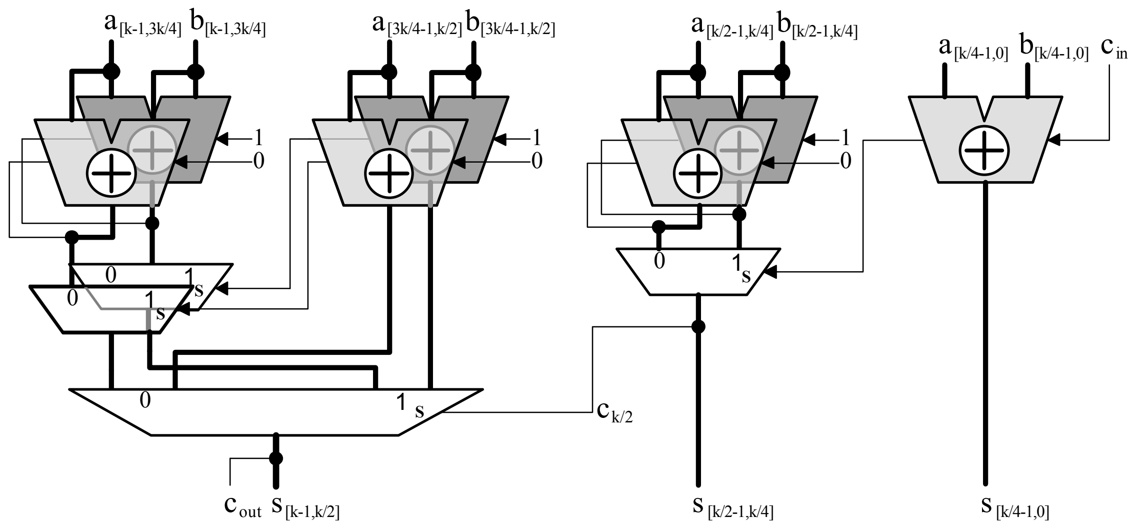


Abbildung 4.8: Zweistufiger Carry-Select-Addierer.

Carry-Select-Addierer

Als letzte Variante von Addierer-Designs soll noch kurz auf die Carry-Select-Technik eingegangen werden, die vor allem bei der Konstruktion von hybriden Addiererschaltungen eingesetzt wird. Hier werden die k-Bit-Summanden a und b in l-Bit-Stücke aufgeteilt:

$$a = a_{[k-1,k-l]} \ldots a_{[l-1,0]}\ ,\ b = b_{[k-1,k-l]} \ldots b_{[l-1,0]}.$$

Die Summe $a_{[l-1,0]} + b_{[l-1,0]}$ wird einfach ausgewertet, die übrigen Teilsummen der l-Bit-Stücke werden zweifach berechnet, einmal für $c_{in} = 0$ und einmal für $c_{in} = 1$. Das Resultat wird durch einen Baum von Multiplexern anhand der Signale c_{out} der Teilsummen zusammengesetzt. Die Abbildungen 4.7 und 4.8 zeigen zwei Beispiele für Carry-Select-Addierer. Die Latenz verringert sich gegenüber der eines k-Bit-Addierers auf die Latenz der l-Bit-Addierer plus der logarithmisch mit der Tiefe skalierenden Latenz des Multiplexer-Baumes.

4.1.3 Multiplikationsverfahren

Die elementare Rechenvorschrift zur Bildung des Produkts zweier k-Bit-Radix-r-Ganzzahlen x und y ist gegeben durch die Formel

$$x \cdot y = (x_{k-1} \cdots x_0) \cdot (y_{k-1} \cdots y_0) = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} x_i\, y_j\, r^{i+j}. \tag{4.8}$$

Es ist also die Summe aus k^2 1-Position-Produkten zu bilden (1-Bit-Produkte für $r = 2$). Die verschiedenen Implementierungsstrategien unterscheiden sich im Wesentlichen in der Art und Weise, wie die Partialprodukte gebildet und aufsummiert werden.

Shift/Add-Methode

Die Shift/Add-Methode entspricht der Schulmethode für die Multiplikation . Hier sind die Partialprodukte die um j Stellen nach links verschobenen Produkte von x mit einer Stelle y_j

$$x \cdot y = \sum_{j=0}^{k-1} (x_{k-1} \cdots x_0) \cdot y_j \cdot r^j. \tag{4.9}$$

Dieses Rechenschema kann direkt mit einer Hardwarearchitektur zur sequentiellen Multiplikation, wie in Abb. 4.9 dargestellt, umgesetzt werden. Dazu wird Formel 4.9 durch folgendes Iterationsschema für das Zwischenergebnis s ausgewertet:

$$\begin{aligned} s(0) &= (x_{k-1} \cdots x_0) \cdot y_0 \cdot r^{k-1} \\ s(i) &= (x_{k-1} \cdots x_0) \cdot y_i \cdot r^{k-1} + s(i-1) \cdot r^{-1} \\ x \cdot y &= s(k-1). \end{aligned} \tag{4.10}$$

Für den Fall $r = 2$ kann das k-Bit-Partialprodukt $x \cdot y_j$ durch einen k-Bit-Multiplexer oder ein k-Bit-*AND*-Element gebildet werden. Das $2k$-Bit-Produkt aus k-Bit-Zahlen wird hier in k Takten berechnet.

Für $r > 2$ ergibt sich das gleiche Hardwareschema, jedoch ist die Erzeugung des $(k + \log_2 r)$ Bit breiten Partialprodukts $x \cdot y_j$ aufwändiger. Das Resultat ist in diesem Fall nach $\lceil k/\log_2 r \rceil$ Takten verfügbar.

Tree-Methode

Bei dieser Technik werden die parallel berechneten Partialprodukte durch einen Addiererbaum aufsummiert. Dies ist in Abbildung 4.10 dargestellt.

Zur Begrenzung der Signallaufzeiten im Addiererbaum können redundante Zahlendarstellungen gewählt werden, um die Anwendung von schnellen Carry-Save-Addierelementen zu ermöglichen. Ist die Latenz des Multiplizierers zweitrangig, kann die Schaltung durch Einfügen von Pipeline-Registern zwischen den Addiererstufen weiter beschleunigt werden.

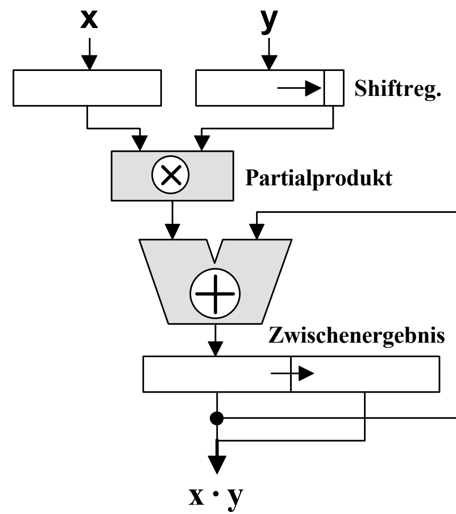


Abbildung 4.9: Hardwareumsetzung der sequentiellen Multiplikation nach der Shift/Add-Methode.

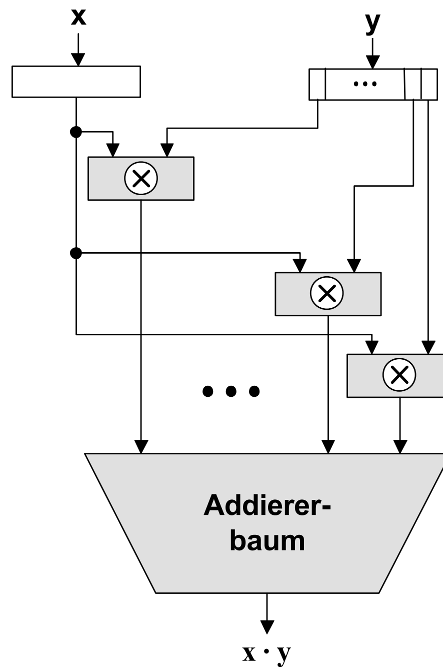


Abbildung 4.10: Hardwareumsetzung der Multiplikation nach der Tree-Methode.

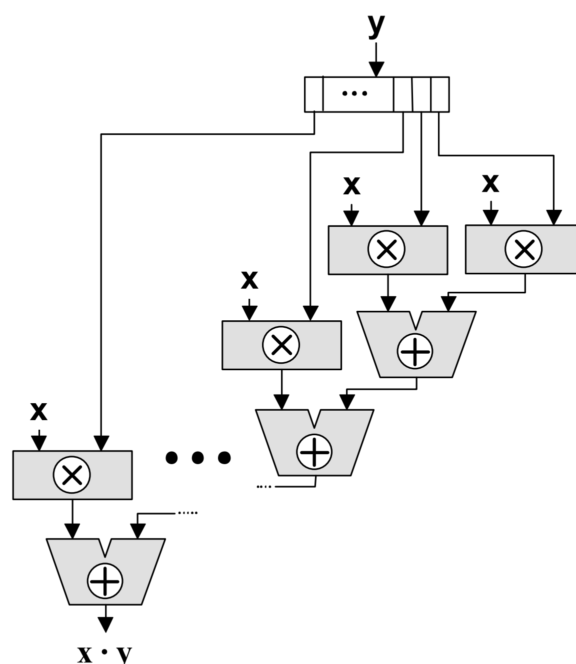


Abbildung 4.11: Hardwareumsetzung der Multiplikation als Array-Multiplizierer.

Array-Multiplizierer

Multiplizierer dieser Art können als Tree-Multiplizierer mit besonderer Struktur des Addiererbaums gesehen werden. Es handelt sich hier um einen Addiererbaum mit maximaler Höhe, bei dem auf der Stufe $i < k$ das Partialprodukt $P_i = x \cdot y_i \cdot r^i$ zur Summe der Partialprodukte $\sum_{j=0}^{i} P_j$ addiert wird. Dies ist die langsamste Variante für einen Addiererbaum, jedoch mit sehr regulärer Struktur, was eine effiziente Umsetzung als integrierte elektronische Schaltung ermöglicht. Die Addierer für die Partialprodukte können als schnelle Carry-Save-Addierer ausgeführt werden, was einen abschließenden Addierer zur Rückführung in eine nicht-redundante Integer-Zahl erforderlich macht. Abbildung 4.11 zeigt die schematische Darstellung eines Array-Multiplizierers. In Abbildung 4.12 ist die detaillierte Schaltung für einen 4×4-Bit-Multiplizierer mit Volladdierer-Elementen dargestellt. Durch Pipeline-Register kann diese Architektur ebenso beschleunigt werden wie bei der Tree-Methode.

4.1.4 Divisionsalgorithmen

Es gibt eine Vielzahl von Verfahren für die digitalelektronische Umsetzung der Division. In diesem Abschnitt wird eine kurze Übersicht über die gängigen Algorithmen gegeben. Die für diese Arbeit geeignet erscheinenden Methoden werden eingehend beschrieben und bezüglich der Schaltungstechnik diskutiert. Andere Verfahren werden nur erwähnt, um sie gegen die verwendeten Verfahren abzugrenzen. Da in dieser Arbeit nur Divisionsoperatoren für vorzeichenlose Ganzzahloperanden benötigt werden, werden auch nur solche Operatoren beschrieben. Es sei

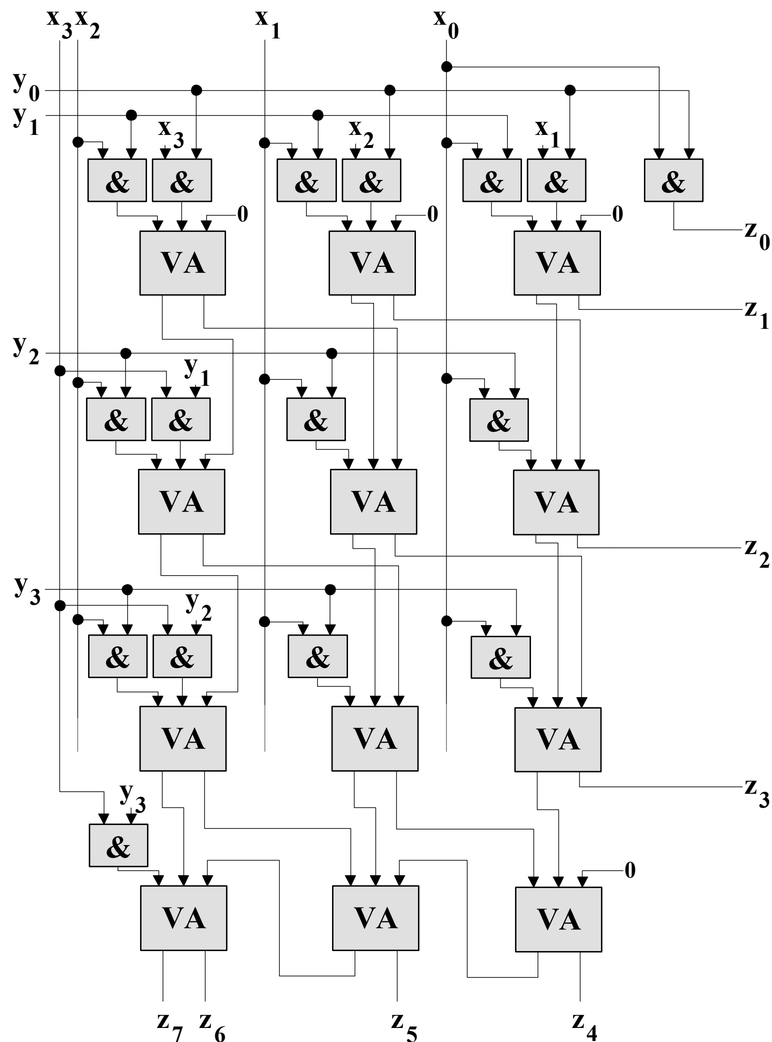


Abbildung 4.12: Detailschaltbild eines Array-Multiplizierers für 4-Bit-Zahlen bestehend aus Volladdierern (VA), welche als dreistufiger Carry-Save-Addierer mit nachfolgender Ripple-Carry-Stufe geschaltet sind. Jedes Und-Element bildet ein 1×1-Bit-Produkt.

jedoch erwähnt, dass die meisten hier beschriebenen Verfahren mit relativ wenig Aufwand auf Signed-Integer-Operanden erweitert werden können. Eine gute Übersicht über moderne Divisionsalgorithmen, wie sie für Mikroprozessoren angewandt werden, geben Obermann und Flynn [87] und Obermann [85].

Ein elementares Divisionsverfahren leitet sich daraus ab, dass die Division $q = x/d$ die Umkehrung der Multiplikation $x = d \cdot q$ ist. Die Schritte der Shift/Add-Methode für die Multiplikation lassen sich umkehren, was zur Schulmethode für die Division führt. Damit kann die Division durch bedingte Subtraktion und Schiebeoperationen durchgeführt werden.

Es sei x eine Ganzzahl mit $2k$ Radix-r-Stellen. Der Divisor d und der resultierende Quotient q seien Ganzzahlen mit k Stellen. Die Zahlen $s(i)$ seien die Restbeträge nach der Iteration i.

$$\begin{aligned}
x &= x_{2k-1} \cdots x_0 \\
d &= d_{k-1} \cdots d_0 \\
q &= q_{k-1} \cdots q_0 \\
s(i) &= s(i)_{2k-i-1} \cdots s(i)_0.
\end{aligned}$$

Es müssen bei dieser Festlegung der Zahlendarstellungen vorab zwei Ausnahmebedingungen abgefangen werden:

1. Division durch Null ($d = 0$)
2. Überlauf von q im Fall $x \geq r^k \cdot d$

Die Iteration der Schulmethode ($i = 1, \ldots, k$) verläuft dann nach folgendem Schema:

$$\begin{aligned}
s(0) &= x \\
q_{k-i} &= \max\left(m \in [0, r-1] : \left(s(i-1) - m \cdot r^{k-i} d\right) \geq 0\right) \\
s(i) &= s(i-1) - q_{k-i} \cdot r^{k-i} d.
\end{aligned} \tag{4.11}$$

Der Rest der Division ist gegeben durch $s(k)$. Im Folgenden werden weit verbreitete Verfahren vorgestellt, die sich schaltungstechnisch effizient umsetzen lassen.

Non-Performing und Restoring Division

Dies sind einfache Verfahren, die Schulmethode nach Gleichung 4.11 für $r = 2$ umzusetzen. Das Iterationsschema ist dann folgendes:

$$\begin{aligned}
s(0) &= x \\
t(i) &= s(i-1) - 2^{k-i} d \\
q_{k-i} &= \begin{cases} 1 & : \ t(i) \geq 0 \\ 0 & : \ t(i) < 0 \end{cases} \\
s(i) &= \begin{cases} t(i) & : \ t(i) \geq 0 \\ s(i-1) = t(i) + 2^{k-i} d & : \ t(i) < 0. \end{cases}
\end{aligned} \tag{4.12}$$

Hier wird also in jedem Iterationsschritt i der um $k-i$ Stellen linksverschobene Divisor d vom Rest $s(i-1)$ subtrahiert, was im Falle eines positiven Ergebnisses $t(i) \geq 0$ als neuer Rest $s(i)$ übernommen wird. Im Falle eines negativen Ergebnisses wird diese Differenz dagegen verworfen (Non-Performing Division) und der alte Restwert weiterverwendet, bzw. der frühere Rest durch eine Addition des verschobenen Divisors wieder restauriert (Restoring Division). Non-Performing Division und Restoring Division sind äquivalent und werden deshalb in der Literatur synonym verwendet. In der praktischen Ausführung wird nicht der um $k-i$ Stellen verschobene Divisor subtrahiert, sondern der Restwert vor jeder Iteration um eine Stelle nach links verschoben, wie an folgender äquivalenter Rechenvorschrift sichtbar wird:

$$\begin{aligned} s(0) &= x \\ t(i) &= 2 \cdot s(i-1) - 2^k \cdot d \\ q_{k-i} &= \begin{cases} 1 & : \; t(i) \geq 0 \\ 0 & : \; t(i) < 0 \end{cases} \\ s(i) &= \begin{cases} t(i) & : \; t(i) \geq 0 \\ 2 \cdot s(i-1) = t(i) + 2^k \cdot d & : \; t(i) < 0. \end{cases} \end{aligned} \tag{4.13}$$

Damit kann in einer seriellen Architektur zur Division der Divisor in einem Register gespeichert werden, während nur der Restwert, der sich in jedem Takt ändern kann, linksverschoben oder durch den linksverschobenen neuen Restwert ersetzt wird. Dies kann beispielsweise über einen einfachen Multiplexer, der einem Restwert-Register vorgeschaltet ist, geschehen. Eine alternative Implementierung wäre ein parallel ladbares Shiftregister.

Abbildung 4.13 zeigt eine Hardwareimplementierung dieses Verfahrens. Vor der Operation wird das Restwertregister S mit dem Dividenden x geladen und das Register D mit d. In jedem Zeitschritt wird der Inhalt von S um eine Position nach links verschoben. Das MSB (**M**ost **S**ignificant **B**it) geht dabei in das Flip-Flop M. Der Addierer bildet die Differenz der Bits $2k-1\ldots k$ von S mit dem Divisor. Ein Carry $c_{out} = 1$ signalisiert, dass das Ergebnis der Subtraktion positiv ist, da Werte mit unterschiedlichen Vorzeichen addiert wurden (die versteckten Vorzeichenbits würden sich mit dem Carry zu 0 addieren). Das Ergebnis kann ebenfalls als positiv interpretiert werden, wenn das Bit M logisch 1 ist, da dann in jedem Fall der Rest größer als der Divisor ist. In beiden Fällen werden die Bits $2k-1\ldots k$ von S mit dem Subtraktionsergebnis geladen (die Bits $k-1\ldots 0$ bleiben unverändert) und q_{k-i} als logisch 1 ausgegeben. Ansonsten wird S nicht neu geladen (Non-Performing-Schritt) und q_{k-i} als logisch 0 ausgegeben. Die Ergebnisbits q_{k-i} werden seriell in das Register Q geladen. Dieses Register kann auch mit S überlagert werden, da beide Register synchron linksverschoben werden. Nach k Iterationen ist die Division beendet.

Das Verfahren kann durch eine Pipeline parallelisiert werden, in der für jede Iteration eine eigene Schaltungslogik vorhanden ist. Dazu sind die Subtrahier-Elemente zu vervielfachen und in einer Pipeline anzuordnen. Die Auswahl, ob mit dem neuen oder alten Restwert weitergearbeitet wird, geschieht dann nicht über das Laden oder Nicht-Laden eines Registers, sondern durch Auswahl und Weiterleiten eines der Werte über einen Multiplexer. Die im nächsten Abschnitt beschriebene Divisionsmethode ermöglicht bei ähnlicher Struktur der Verarbeitung die Parallelisierung ohne den Einsatz von Multiplexern.

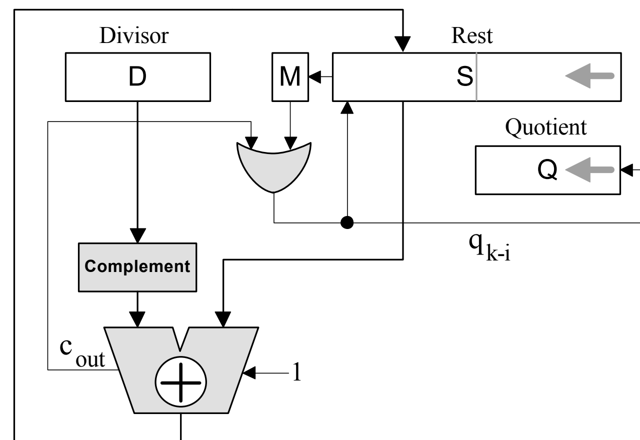


Abbildung 4.13: Serieller Dividierer nach der Non-Performing-Methode.

Non-Restoring Division

Dies ist ebenfalls ein Verfahren für $r = 2$. Es wird bei diesem Verfahren der Umstand ausgenutzt, dass die Restaurations-Addition von $(2^{k-i}d)$ in Gleichung 4.12 bei negativem $t(i)$ und nachfolgende Subtraktion von $(2^{k-(i+1)}d)$ im nächsten Iterationsschritt äquivalent ist mit der Addition von $(2^{k-(i+1)}d)$. Die Restauration eines früheren Restwertes und die testweise Subtraktion eines verschobenen Divisors für den nächsten Iterationsschritt können deshalb zusammengefasst werden, indem der verschobene Divisor im nächsten Iterationsschritt addiert wird. Die folgende Gleichung zeigt die Formulierung dieses Verfahrens ähnlich der Darstellung in Gleichung 4.13:

$$\begin{aligned} s(0) &= x \\ s(i) &= \begin{cases} 2 \cdot s(i-1) - 2^k \cdot d & : \quad s(i-1) \geq 0 \\ 2 \cdot s(i-1) + 2^k \cdot d & : \quad s(i-1) < 0 \end{cases} \\ q_{k-i} &= \begin{cases} 1 & : \quad s(i) \geq 0 \\ 0 & : \quad s(i) < 0. \end{cases} \end{aligned} \tag{4.14}$$

Die Non-Restoring-Methode kann ähnlich in einer seriellen Hardwarearchitektur umgesetzt werden, wie oben für die Non-Performing-Methode beschrieben wurde. Es soll hier jedoch die parallelisierte Implementierungsvariante vorgestellt werden. In Abbildung 4.14 ist ein paralleler Dividierer für 8-Bit-Dividenden x und 4-Bit-Divisoren d gezeigt ($k = 4$). Die Signale, die von einer Addiererreihe an die nächste weitergegeben werden, entsprechen den verschobenen Restwerten $s(i)$. Durch die den linken Eingängen der Addierer vorangehenden XOR-Elemente werden Addier/Subtrahier-Elemente gebildet, wobei ein Eingang der XOR-Bausteine Addition oder Subtraktion selektiert. Die Breite der Addiererreihen muss $k+1$ betragen, da einerseits die Summe oder Differenz des um eine Stelle linksverschobenen Restwertes mit dem k-Bit-Divisor berechnet werden muss, andererseits keine Vorzeichen berücksichtigt werden müssen. Letzteres ergibt sich daraus, dass die Vorzeichen der Summanden immer verschieden sind und das Carry-Out einer Addiererreihe identisch ist mit dem Carry-Out, das entstünde, wenn die Vorzeichenbits

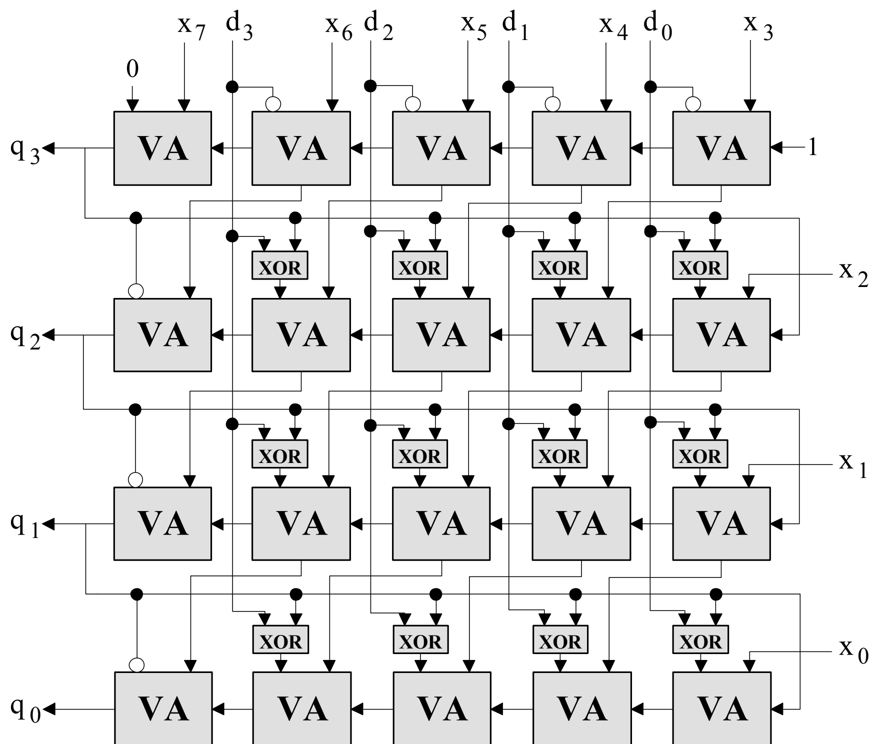


Abbildung 4.14: Detailschaltbild eines Non-Restoring Array-Dividierers für 8-Bit-Zahlen x und 4-Bit-Divisoren d, bestehend aus Volladdierern (VA), welche mit Hilfe der XOR-Elemente als Addier/Subtrahier-Elemente in Ripple-Carry-Schaltung aufgebaut sind. Die Volladdierer der ersten Spalte und der untersten Zeile können durch Logik für c_{out} ersetzt werden, da deren Additionsergebnis nicht benötigt wird.

in einem $(k+2)$-Bit-Addierer mitverarbeitet würden. Zugleich zeigt das Carry-Out ein positives Resultat an und gibt damit neben dem Quotientenbit die Verarbeitung in der folgenden Addiererreihe vor.

SRT-Division

Bei VLSI-Implementierungen sehr weit verbreitet ist die so genannte SRT-Division. Diese Methode wurde benannt nach Sweeney, Robertson und Tocher. Die Methode basiert auf einer Verallgemeinerung der Non-Restoring-Methode für Radix-$r \geq 2$ und Anwendung einer redundanten Darstellung der Quotientenstellen. Durch Erhöhung von r kann die Zahl der Iterationen zur Berechnung der Division verringert werden, da eine Stelle dann aus mehreren Bits besteht, nach wie vor aber pro Iterationsschritt eine Ergebnisstelle resultiert. Die redundante Darstellung der Quotientenstellen ermöglicht die Vereinfachung des Auswahlverfahrens für diese Stellen.

Für die SRT-Division hat die Iteration über die Restwerte $s(i)$ mit gleicher Nomenklatur wie

bei Gleichung 4.14 folgende Gestalt:

$$\begin{aligned} s(0) &= x \\ s(i) &= r \cdot s(i-1) - q_{k-i}\, d. \end{aligned} \tag{4.15}$$

Die Quotientenstellen $q_{k-i} \in (-(r-1), r-1)$ liegen nun in redundanter Radix-r-Zahlendarstellung vor und können beispielsweise über einen Carry-Propagate-Addierer in ein nicht-redundantes Ergebnis umgewandelt werden. Das Schlüsselproblem bei der Division ist, die Stellen q_{k-i} mit geringem Hardwareaufwand zu finden. Die Idee der SRT-Division besteht nun darin, die durch die redundante Darstellung von q gewonnene Freiheit in der Wahl der Quotientenstellen dahingehend auszunutzen, dass nur wenige Bits von $s(i)$ und $d(i)$ ausgewertet werden müssen, um die q_{k-i} mit ausreichender Genauigkeit zu bestimmen, sodass die Iteration konvergiert.

Abbildung 4.15 zeigt das Prinzipschaltbild eines Dividierers nach der SRT-Methode. Die Auswahl der redundanten Quotientenstelle q_{k-i} geschieht im Baustein *Select* q_{k-i}. Im Block *Multiple Generation/Selection* werden die Produkte $q_{k-i}\, d$ auf möglichst effiziente Weise gebildet. Durch geschickte Wahl der q_{k-i} kann z.B. bei Radix-4-Dividierern erreicht werden, dass der Divisor lediglich verschoben und danach eventuell das Komplement gebildet werden muss, was beides durch einen geringen Hardwareaufwand zu erreichen ist.

Während die SRT-Methode für serielle Dividierer-Architekturen wie in Abbildung 4.15 eine Vervielfachung der Rechenleistung erbringen kann, verschwinden diese Vorteile bei einer parallelen FPGA-Implementierung. Einfache Abschätzungen des Implementierungsaufwands pro Ergebnisbit zeigen, dass die Non-Restoring-Methode bei geringerer Komplexität des Algorithmus mindestens ebenso gut abschneidet wie beispielsweise die SRT-Implementierung eines Radix-4-Dividierers. Voraussetzung dafür ist, dass zwei Addier/Subtrahier-Rechenwerke für die Non-Restoring-Methode genauso ressourceneffizient implementiert werden können wie ein Multiplexer und 1er-Komplement-Generator, wie sie bei der SRT-Methode für die Auswahl der Vielfachen des Divisors (*Multiple Generation/Selection*) notwendig werden und ein nachfolgender Addierer. Dies ist beispielsweise beim verwendeten Virtex-II-FPGA der Fall. Gegen die Anwendung der SRT-Methode bei FPGA-Implementierungen spricht außerdem, dass die Geschwindigkeit und Latenz eines parallelen SRT-Dividierers wesentlich schwieriger über das Einfügen von Pipeline-Registern kontrolliert werden kann, als es beispielsweise bei Array-Dividierern nach Abbildung 4.14 geschehen kann. Aus diesen Gründen wurde die Implementierung von Dividierern nach der SRT-Methode für diese Arbeit nicht weiter in Betracht gezogen.

Weitere Divisionsverfahren

Für die Berechnung der Division gibt es zahlreiche weitere Algorithmen. Eine Übersicht über Hardwarealgorithmen für Divisionsoperatoren in modernen Prozessoren geben z.B. [85] und [87]. Für Verfahren zur Berechnung der Division mit doppelter Genauigkeit siehe auch [60] und [92]. Die bisher beschriebenen Verfahren gehören alle zur Klasse der *Digit-Recurrence-Verfahren* mit einer Ergebnisstelle pro Iteration, also linearer Konvergenz. Ein Verfahren mit quadratischer Konvergenz ist die Division durch wiederholte Multiplikation (auch unter dem Namen Goldschmidt-Verfahren bekannt [38]; siehe auch [28]). Hierbei werden in mehreren Durchläufen $i = 0, \ldots, m-1$ jeweils sowohl der aktuelle Dividend $x(i)$, $x(0) = x$ als auch der Divisor

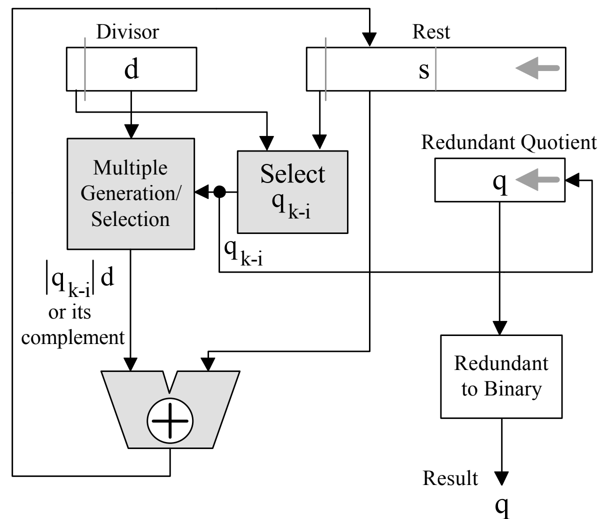


Abbildung 4.15: Blockschaltbild eines einfachen seriellen Dividierers nach der SRT-Methode.

$d(i)$, $d(0) = d$ mit Zahlen t_i multipliziert, und zwar so, dass $d(i)$ gegen 1 konvergiert. Dann konvergiert $x(i)$ gegen $q = x/d$, denn es gilt:

$$q = \frac{x}{d} = \frac{\overbrace{x \cdot t_0 \cdot t_1 \cdots t_{m-1}}^{x(m-1)}}{\underbrace{d \cdot t_0 \cdot t_1 \cdots t_{m-1}}_{\approx 1}}.$$

Die Faktoren t_i werden gewählt zu $t_i = 2 - d_i$. Damit ergibt sich die Iteration zu:

$$\begin{aligned} x(i+1) &= x(i) \cdot (2 - d_i) \\ d(i+1) &= d(i) \cdot (2 - d_i). \end{aligned} \tag{4.16}$$

Die quadratische Konvergenz folgt direkt aus der Beziehung $1 - d(i+1) = (1 - d(i))^2$. Für eine Genauigkeit von k Bits müssen $\lceil log_2\, k \rceil$ Iterationen berechnet werden, wobei zur Vermeidung der Akkumulation von Rechenfehlern mit höherer Genauigkeit als k Bits gerechnet werden muss (mindestens $log_2\, m$ Zusatzbits). Für $k = 24$ ist $m = 5$. Es müssen also $2m - 1 = 9$ Multiplikationen durchgeführt werden ($d(m-1)$ braucht nicht berechnet zu werden). Wie im vorangegangenen Abschnitt gezeigt, benötigt eine Hardwareimplementierung eines Dividierers nach der Non-Restoring-Methode in voll parallelisierter Ausführung weniger als doppelt so viele Ressourcen wie ein Array-Multiplizierer und damit etwa fünfmal weniger Ressourcen als die parallele Implementierung der Division durch wiederholte Multiplikation. Letzteres Verfahren kommt deshalb für eine FPGA-Implementierung nicht in Frage.

Einige weitere Verfahren zur Berechnung der Division basieren auf dem Ansatz, zuerst die reziproke Zahl $1/d$ zu approximieren und dann den Quotienten durch eine nachfolgende Multiplikation mit x zu bilden. Die Berechnung von $I(d) = 1/d$ gehört zur Problemstellung der

Funktionsberechnung. Zur Lösung können beispielsweise Näherungsverfahren wie die Newton-Raphson-Methode angewandt werden. Näheres dazu wird in Abschnitt 4.5 folgen. Da jedes dieser Verfahren mehrere Multiplikationen und Additionen erforderlich macht, spielt diese Herangehensweise für eine parallele FPGA-Implementierung ebenfalls keine Rolle. Diese Methoden sind jedoch wichtig für die schnelle Berechnung von Divisionen und anderen Operationen mit hoher Genauigkeit (z.B. Double-Precision) in Mikroprozessoren. Die ebenfalls in 4.5 beschriebenen Verfahren, welche auf Table-Look-Up[1] basieren, könnten ebenfalls in Betracht gezogen werden. Aufgrund der großen, exponentiell mit den Genauigkeitsanforderungen steigenden Speicher, die für die Look-Up-Tables benötigt werden, wurde dieser Ansatz ebenfalls nicht umgesetzt. Diese Verfahren hatten jedoch für FPGA-Implementierungen bis vor wenigen Jahren noch eine große Bedeutung, als die Implementierung durch Logikressourcen noch nicht möglich war.

4.1.5 Quadratwurzel

Die Ganzzahl-Quadratwurzel q einer Zahl x mit Rest s ist durch folgende Gleichung definiert:

$$x = q^2 + s, \tag{4.17}$$

wobei für s die Beziehung $s \leq 2q$ gilt. Letzteres folgt aus dem Widerspruch

$$s > 2q \Rightarrow x \geq q^2 + 2q + 1 = (q+1)^2.$$

Daraus ergibt sich, dass für die Repräsentation von s maximal eine Stelle mehr benötigt wird als für q.

Für die Quadratwurzel existiert ein ähnliches iteratives Verfahren wie jenes, das bei der Division vorgestellt wurde. Die Papier-und-Bleistift-Methode dazu ist nicht allgemein bekannt und wird deshalb hier kurz hergeleitet. Es sei x der Radikand mit $2k$ Radix-r-Stellen. Die resultierende Quadratwurzel q sei eine Ganzzahl mit k Stellen und die Zahlen $q(i)$ (k Bits) und $s(i)$ ($2k$ Bits) seien die vorläufigen Quadratwurzeln und Restbeträge nach der Iteration i:

$$\begin{aligned} x &= x_{2k-1} \cdots x_0 \\ q &= q_{k-1} \cdots q_0 \\ q(i) &= q(i)_{k-1} \cdots q(i)_0 \\ s(i) &= s(i)_{2k-i-1} \cdots s(i)_0 \\ s &= s(k) = s_k \cdots s_0. \end{aligned}$$

Für die vorläufigen Quadratwurzeln $q(i)$ seien die führenden i Bits bestimmt, die übrigen Bits seien identisch Null. Die Zahlen $q(i)$ und die Restwerte $s(i)$ sind über die Beziehung

$$s(i) = x - q(i)^2$$

[1] Mit Table-Look-Up ist ein Verfahren gemeint, das Tabellen (LUTs = **L**ook-**U**p-**T**ables) zur schnellen Bestimmung von Funktionsergebnissen verwendet.

miteinander verknüpft. Für $q(i)$ gilt:

$$\begin{aligned}
q(i) &= q_{k-1}\cdots q_{k-i}\,0\cdots 0 \\
&= q(i-1)+r^{k-i}q_{k-i} \\
q(0) &= 0\cdots 0 \\
q(i)^2 &= \big(q(i-1)+r^{k-i}q_{k-i}\big)^2 \\
&= q(i-1)^2+2q(i-1)q_{k-i}r^{k-i}+q_{k-i}^2r^{2k-2i},
\end{aligned} \tag{4.18}$$

woraus sich für $s(i)$ auf folgende Weise eine Iterationsgleichung herleiten lässt:

$$\begin{aligned}
s(i) &= x-q(i)^2 \\
&= x-\big(q(i-1)^2+2q(i-1)q_{k-i}r^{k-i}+q_{k-i}^2r^{2k-2i}\big) \\
&= s(i-1)-\big(2q(i-1)q_{k-i}r^{k-i}+q_{k-i}^2r^{2k-2i}\big) \\
&= s(i-1)-\big(2q(i-1)r^{k-i}+q_{k-i}r^{2k-2i}\big)\cdot q_{k-i}.
\end{aligned} \tag{4.19}$$

Das neue Quadratwurzelbit q_{k-i} wird im Iterationsschritt i festgelegt durch die Bedingung der Minimierung von $s(i) > 0$. Es ergibt sich somit folgendes Iterationsschema:

$$\begin{aligned}
s(0) &= x \\
q_{k-i} &= \max\big(m\in[0,r-1]: s(i-1)-\big(2q(i-1)r^{k-i}+mr^{2k-2i}\big)\cdot m\geq 0\big) \\
s(i) &= s(i-1)-\big(2q(i-1)r^{k-i}+q_{k-i}r^{2k-2i}\big)\cdot q_{k-i}.
\end{aligned} \tag{4.20}$$

Dieses Iterationsverfahren lässt sich intuitiver als Papier-und-Bleistift-Methode erkennen, wenn man $s(i)$ durch $s'(i) = r^{-2k+2i}s(i)$ ersetzt und $q(i)$ durch $q'(i) = r^{-k+i}q(i)$. Dann folgt:

$$\begin{aligned}
s'(0) &= x\cdot r^{-2k} \\
q'(i) &= 0\cdots 0\,q_{k-1}\cdots q_{k-i} \\
q_{k-i} &= \max\big(m\in[0,r-1]: r^2s'(i-1)-(2rq'(i-1)+m)\cdot m\geq 0\big) \\
s'(i) &= r^2s'(i-1)-(2rq'(i-1)+q_{k-i})\cdot q_{k-i} \\
s &= s'(k).
\end{aligned} \tag{4.21}$$

Die Zahlen $q'(i)$ sind weiterhin Ganzzahlen, während $s'(i)$ Festkommazahlen sind. Durch die Multiplikation von $s(i-1)$ mit r^2 wird ausgedrückt, dass in jedem Iterationsschritt zwei weitere Stellen von x in die Berechnung einbezogen werden (die Nachkommastellen von $s'(i-1)$ beeinflussen die Wahl von q_{k-i} nicht). Abbildung 4.16 zeigt ein Beispiel zu dieser Methode mit $r = 10$ und $k = 3$, wobei $q = \sqrt{123456}$ berechnet wird.

Für $r = 2$ können wie bei der Division Non-Performing-, Restoring- und Non-Restoring-Methoden zur Berechnung angewandt werden.

$$
\begin{array}{lcccccccl}
s'(0) = 10^{-6}x & = & .1 & 2 \vdots 3 & 4 \vdots 5 & 6 \vdots & & & q(0) = 0 \\
100\, s'(0) & = & 1 & 2 \vdots .3 & 4 \vdots 5 & 6 \vdots & & & (20 \cdot 0 + 3) \cdot 3 = 9 \leq 12 \\
 & & & 9 \vdots & \vdots & \vdots & & & \rightarrow q_2 = 3 \quad \rightarrow \quad q'(1) = 3 \\
100\, s'(1) & = & & 3 & 3\ \ 4 \vdots .5 & 6 \vdots & & & (20 \cdot 3 + 5) \cdot 5 = 325 \leq 334 \\
 & & & 3 & 2\ \ 5 \vdots & \vdots & & & \rightarrow q_1 = 5 \quad \rightarrow \quad q'(2) = 35 \\
100\, s'(2) & = & & & 9\ \ 5 & 6 \vdots & & & (20 \cdot 35 + 1) \cdot 1 = 701 \leq 956 \\
 & & & & 7\ \ 0 & 1 \vdots & & & \rightarrow q_0 = 1 \quad \rightarrow \quad q'(3) = 351 \\
s = s'(3) & = & & & 2\ \ 5 & 5 & & & q = q'(3) = 351
\end{array}
$$

Abbildung 4.16: Beispiel zur Berechnung der Quadratwurzel nach der Papier-und-Bleistift-Methode.

Non-Performing- und Restoring-Methode

Diese beiden Methoden leiten sich daraus ab, dass Gleichung 4.21 für $r = 2$ in folgender Form geschrieben werden kann:

$$
\begin{aligned}
s'(0) &= x \cdot 2^{-2k} \\
t(i) &= 4 \cdot s'(i-1) - (4q'(i-1) + (01)_2) \\
q_{k-i} &= \begin{cases} 1 & : \ t(i) \geq 0 \\ 0 & : \ t(i) < 0 \end{cases} \\
q'(i) &= 2 \cdot q'(i-1) + q_{k-i} \\
s'(i) &= \begin{cases} t(i) & : \ t(i) \geq 0 \\ 4 \cdot s'(i-1) = t(i) + (4q'(i-1) + (01)_2) & : \ t(i) < 0. \end{cases}
\end{aligned}
\tag{4.22}
$$

Das Iterationsschema ist also dem der Non-Performing- oder Restoring-Methode für die Division sehr ähnlich. Hier wird bei der testweisen Subtraktion eine aus der vorläufigen Quadratwurzel q' durch Linksverschieben um zwei Stellen und Anhängen der Bits 01 gebildete Zahl verwendet. Statt wie bei der Division in jedem Iterationsschritt den vorläufigen Restwert um eine Stelle nach links zu verschieben, ist hier eine Verschiebung um zwei Stellen erforderlich.

Non-Restoring-Methode

Für $r = 2$ kann genau wie im Fall der Division auch für die Quadratwurzel eine Non-Restoring-Methode formuliert werden. Die Herleitung ist hier jedoch etwas komplizierter. Dazu betrachte man den Term, welcher in Gleichung 4.20 zur Berechnung von $s(i)$ im Fall $q_{k-i} = 1$ von $s(i-1)$ zu subtrahieren ist. Dieser Term wird hier $u(i)$ genannt:

$$
\begin{aligned}
u(i) &= \left(2q(i-1)r^{k-i} + q_{k-i}r^{2k-2i}\right) \cdot q_{k-i} \\
&= 2q(i-1)2^{k-i} + (01)_2 \cdot 2^{2k-2i}.
\end{aligned}
\tag{4.23}
$$

Wird testweise $t(i) = s(i-1) - u(i)$ berechnet, entscheidet das Vorzeichen von $t(i)$ über q_{k-i}. Es gilt für $q_{k-i} = 1$ die Beziehung $s(i) = t(i)$, für $q_{k-i} = 0$ jedoch $s(i) = t(i) + u(i)$. Wie bei der Non-Restoring-Methode bei der Division kann für letzteren Fall die Restauration des Restwertes und die testweise Subtraktion der nächsten Iteration zusammengefasst werden, was sich aus folgender Gleichung erschließt:

$$
\begin{aligned}
t(i+1) &= s(i) - u(i+1) \\
&= t(i) + \underbrace{u(i) - u(i+1)}_{v(i+1)} \\
v(i+1) &= u(i) - u(i+1) \\
&= (2 \quad \overbrace{q(i-1)}^{q((i+1)-1) \Leftrightarrow q_{k-i}=0} \quad 2^{k-i} + (01)_2 \cdot 2^{2k-2i}) \\
&\quad -(2q((i+1)-1)2^{k-(i+1)} + (01)_2 \cdot 2^{2k-2(i+1)}) \\
&= 2q((i+1)-1)\left(2^{k-i} - 2^{k-(i+1)}\right) + (01)_2 \cdot \left(2^{2k-2i} - 2^{2k-2(i+1)}\right) \\
&= 2q((i+1)-1)2^{k-(i+1)} + (11)_2 \cdot 2^{2k-2(i+1)} \\
v(i) &= 2q(i-1)2^{k-i} + (11)_2 \cdot 2^{2k-2i}.
\end{aligned}
\tag{4.24}
$$

Anstatt zur Berechnung von $t(i+1)$ vom restaurierten $s(i)$ den Wert $u(i+1)$ zu subtrahieren, kann also $v(i+1)$ zum nicht-restaurierten Restwert $t(i)$ addiert werden, wobei sich $u(i+1)$ und $v(i+1)$ nur um ein Bit unterscheiden. Damit ergibt sich folgendes Iterationsschema, für welches $t(i)$ in $s(i)$ umbenannt wurde:

$$
\begin{aligned}
s(0) &= x \\
s(i) &= \begin{cases} s(i-1) - \left(2q(i-1)2^{k-i} + (01)_2 \cdot 2^{2k-2i}\right) & : \quad s(i-1) \geq 0 \\ s(i-1) + \left(2q(i-1)2^{k-i} + (11)_2 \cdot 2^{2k-2i}\right) & : \quad s(i-1) < 0 \end{cases} \\
q_{k-i} &= \begin{cases} 1 & : \quad s(i) \geq 0 \\ 0 & : \quad s(i) < 0. \end{cases} \\
q(i) &= q_{k-1} \cdots q_{k-i}\, 0 \cdots 0 \\
q &= q(k).
\end{aligned}
\tag{4.25}
$$

Ersetzt man hier wie beim Übergang von Gleichung 4.20 zu Gleichung 4.21 $s(i)$ durch $s'(i) = 2^{-2k+2i}s(i)$ und $q(i)$ durch $q'(i) = 2^{-k+i}q(i)$, erhält man die einfacher zu lesende Iterationsfor-

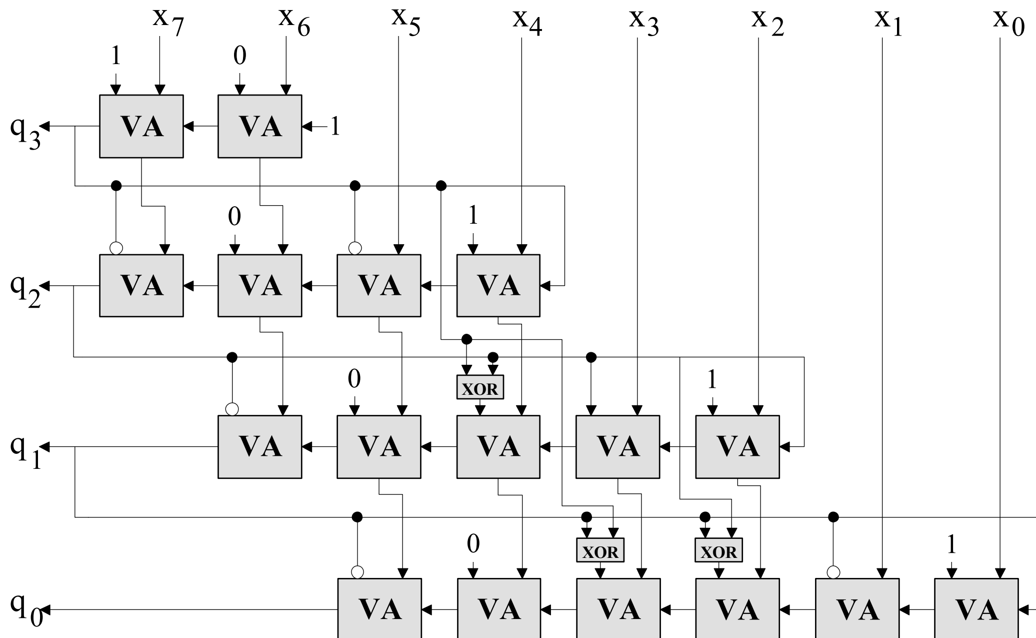


Abbildung 4.17: Detailschaltbild eines Quadratwurzel-Operators im Non-Restoring-Verfahren für 8-Bit-Zahlen x und 4-Bit-Ergebnisse q bestehend aus Volladdierern (VA) welche als Addier/Subtrahier-Elemente in Ripple-Carry Schaltung aufgebaut sind.

mel:

$$
\begin{aligned}
s'(0) &= x \cdot 2^{-2k} \\
s'(i) &= \begin{cases} 4s'(i-1) - (4q'(i-1) + (01)_2) & : \quad s'(i-1) \geq 0 \\ 4s'(i-1) + (4q'(i-1) + (11)_2) & : \quad s'(i-1) < 0 \end{cases} \\
q_{k-i} &= \begin{cases} 1 & : \quad s'(i) \geq 0 \\ 0 & : \quad s'(i) < 0. \end{cases} \\
q'(i) &= 0 \cdots 0\, q_{k-1} \cdots q_{k-i} \\
q &= q'(k).
\end{aligned}
\tag{4.26}
$$

Abbildung 4.17 zeigt die Schaltung für einen Quadratwurzel-Operator für 8-Bit-Zahlen. Hier wurde das Iterationsschema auf die gleiche Weise angewandt wie beim Array-Dividierer gezeigt wurde (siehe Abbildung 4.14).

Weitere Verfahren

Bei der Quadratwurzel handelt es sich um die Berechnung einer elementaren Funktion. Es kann eine Vielzahl von Algorithmen, welche zur Approximation solcher Funktionen entwickelt wurden, angewandt werden. Dazu wird in Abschnitt 4.5 ein allgemeiner Überblick gegeben. Es sei hier bereits erwähnt, dass für eine parallele Implementierung die in den vorangegangenen Ab-

schnitten beschriebenen *Digit-Recurrence*-Verfahren im Hinblick auf den Ressourcenverbrauch den in Abschnitt 4.5 vorgestellten Implementierungsalternativen überlegen sind (wie auch im Fall der Division). Diese alternativen Verfahren erlangen ihre Bedeutung vor allem im Design programmgesteuerter Prozessoren, wo die Latenzzeit entscheidend für die Rechenleistung ist.

4.2 Festkommazahlen

Eine Festkomma- oder Fixpunkt-Zahl ist durch einen Ganzzahl-Teil (k Stellen) und Nachkomma-Teil (l Stellen) gegeben. Die Darstellung ergibt sich aus 4.1 durch Erweiterung der Stellenwerte auf Potenzen von r mit negativen Exponenten, wie in folgender Gleichung gezeigt wird:

$$\underbrace{x_{k-1}\cdots x_0}_{Ganzzahl-Anteil} \;.\; \underbrace{x_{-1}\cdots x_{-l}}_{Nachkomma-Teil} = \sum_{i=-l}^{k-1} x_i r^i. \tag{4.27}$$

Die größte darstellbare Zahl hängt exponentiell von der Ganzzahl-Bitbreite k ab, die kleinste darstellbare Zahl exponentiell von der Anzahl der Nachkommastellen l. Die relative Genauigkeit, mit der eine Zahl darstellbar ist, ist von deren Größe abhängig und ist maximal für die größte darstellbare Zahl und minimal für die kleinste. Diese Darstellung eignet sich zur Funktionsauswertung nur, wenn der Wertebereich der Variablen sehr eng und a priori bekannt ist.

In Festkommaarithmetik können die grundlegenden binären Operatoren auf Fixpunktzahlen a und b als Integer-Operatoren auf den Ganzzahlen $(a\,r^l)$ und $(b\,r^l)$ implementiert werden.

$$a \pm b = (a\,r^l \pm b\,r^l)\cdot r^{-l} \tag{4.28}$$
$$a \cdot b = (a\,r^l \cdot b\,r^l)\cdot r^{-2\,l} \tag{4.29}$$
$$a/b = \left((a\,r^l)\cdot r^{k+l}/(b\,r^l)\right)\cdot r^{-k-l}. \tag{4.30}$$

Je nach Operator hat für das Integer-Resultat aus den als Integer-Werte interpretierten Fixpunktargumenten also lediglich eine Positionsverschiebung um eine feste Zahl von Stellen zu erfolgen, was für eine Hardwareimplementierung keinerlei zusätzliche Ressourcen gegenüber einer Integer-Implementierung erfordert. Die Addition führt zu einem exakten Ergebnis. Im Fall der Multiplikation und Division ist gegebenenfalls eine Rundungsoperation zu vollziehen.

4.3 Gleitkommazahlen

4.3.1 Darstellung von Gleitkommazahlen

Die Gleitpunktdarstellung von Zahlen beruht auf der multiplikativen Zerlegung eines Zahlenwertes in ein Vorzeichen $sign \in \{-1,+1\}$, eine Potenz b^{exp} zu einer Basis b und einer normierten Mantisse $signif$ im Intervall $[1,b)$:

$$z = sign \cdot b^{exp} \cdot signif. \tag{4.31}$$

Der Exponent *exp* gibt somit die Größenordnung vor, in der die Zahl liegt und die Mantisse legt den Wert mit der Präzision der Mantissenbreite fest. Wir schränken uns ein auf die Basis $b = 2$. Die Mantisse $signif = 1.f$ ist dann eine Festkommazahl im Intervall $[1,2)$. Die Bitbreite dieser Mantisse sei M:

$$signif = 1 + f_{M-2}\, 2^{-1} + ... + f_0\, 2^{-M+1}. \tag{4.32}$$

Da das führende Bit einer normierten Mantisse immer 1 ist, werden nur die Nachkommastellen gespeichert:

$$\begin{aligned} f &= signif - 1 \\ &= f_{M-2}\, 2^{-1} + .. + f_0\, 2^{-M+1}. \end{aligned} \tag{4.33}$$

Die niedrigstwertige Stelle $f_0\, 2^{-M+1}$ wird im Folgenden auch mit LSB (**L**east **S**ignificant **B**it) und die Wertigkeit 2^{-M+1} mit *ulp* (**U**nit of **L**east **P**recision) bezeichnet.

Der Exponent *exp* wird oft durch Addition eines Offsets (*bias*) zu einer vorzeichenlosen Zahl transformiert, im Folgenden e genannt. Der Wert für *bias* wird wie folgt gewählt, wobei E die Bitbreite ist, mit der *exp* dargestellt wird:

$$\begin{aligned} exp &= e - bias \\ bias &= 2^{E-1} - 1. \end{aligned} \tag{4.34}$$

Diese Wahl des Wertes für *bias* führt zu sehr einfachen Routinen zur Verarbeitung der Exponenten bei arithmetischen Operationen. Das Vorzeichen *sign* wird kodiert durch ein Bit $s \in \{0,1\}$ und die Vorschrift $(s = 0 \rightarrow sign = +1,\ s = 1 \rightarrow sign = -1)$. Durch das Tripel (s,e,f) wird folgende Zahl dargestellt:

$$z = (-1)^s \cdot 2^{e-bias} \cdot (1.f). \tag{4.35}$$

Der ANSI/IEEE-754-Standard [48] gibt für Gleitkommazahlen einfacher und doppelter Genauigkeit die in Tabelle 4.1 aufgelisteten Parameter vor. In der Tabelle sind weitere wichtige Eckdaten wie kleinste und größte darstellbare Zahlen aufgeführt.

4.3.2 Genauigkeit von Gleitkommaoperationen

Der IEEE-754-Standard für Gleitkommazahlen gibt strenge Vorschriften für die Genauigkeit von Rechenoperationen auf diesen Zahlen vor. So wird vorgeschrieben, dass das Ergebnis einer Fließkommaoperation mit dem gerundeten Ergebnis der exakt durchgeführten Operation übereinstimmen muss. Für das Runden selbst sind die nachfolgend aufgestellten Modi vorgesehen. Hier wird die gerundete Mantisse zusammen mit dem Vorzeichen des Ergebnisses als Ganzzahl in Vorzeichen-Betrag-Darstellung ($s\ 1\ f_{M-2} \cdots f_0$) und die Mantisse der exakten Operation als Fixpunktzahl r mit unendlich vielen Nachkommastellen interpretiert ($s\ 1\ r_{M-2} \cdots r_0.r_{-1} \cdots$). Diese verschiedenen Rundungs-Modi sind in Abbildung 4.18 dargestellt.

- *Round To Nearest Even* - dieser Modus bedeutet, dass aufgerundet wird, wenn der Wert der Nachkommastellen des exakten Ergebnisses größer als $ulp/2$ ist. Abgerundet wird, wenn diese Stellen einen Wert kleiner als $ulp/2$ haben. Ist der Wert exakt gleich $ulp/2$, wird zur nächsten geraden Zahl gerundet.

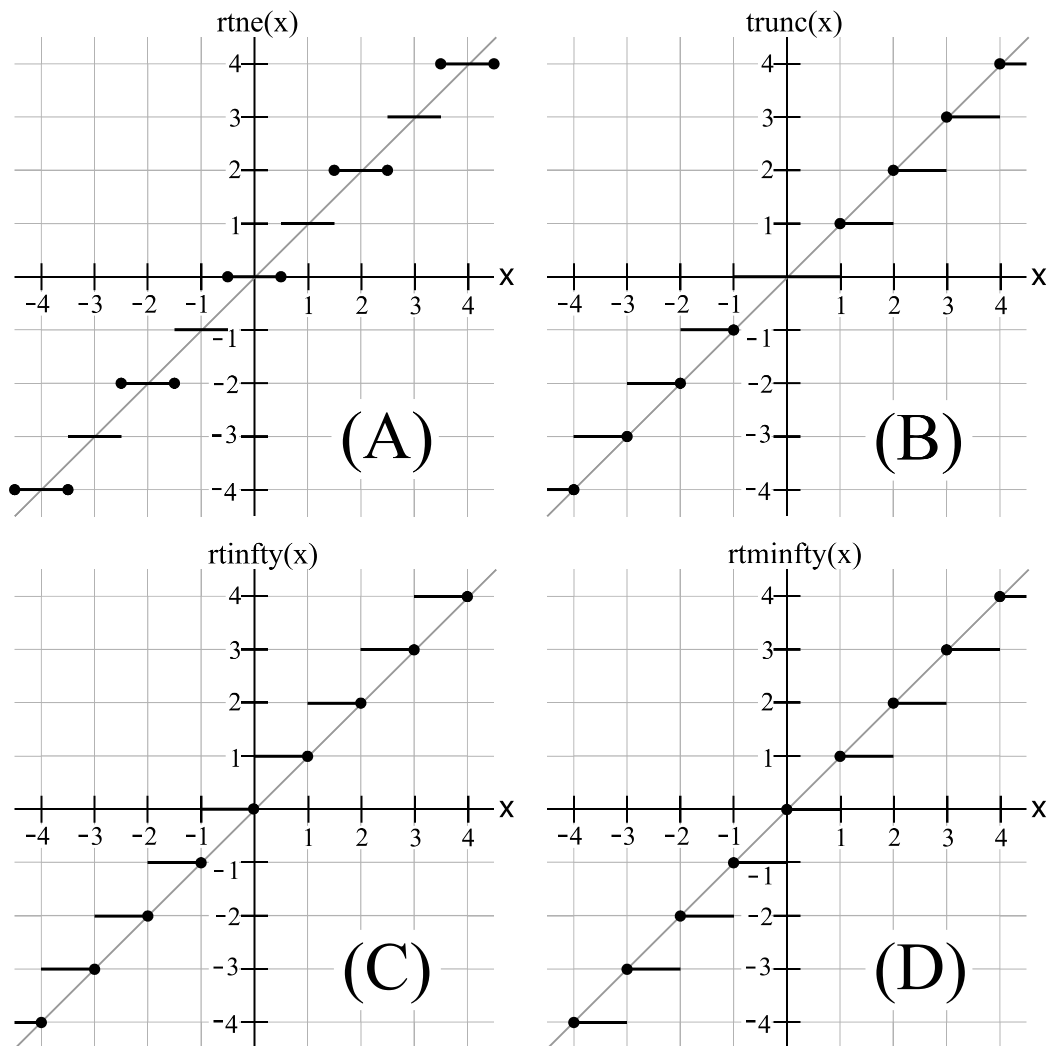


Abbildung 4.18: Rundungsmodi des IEEE-754-Standards für Gleitkommazahlen: (A) *Round To Nearest Even*, (B) *Round Toward 0*, (C) *Round Toward* ∞, (D) *Round Toward* $-\infty$.

	einfache Genauigkeit	doppelte Genauigkeit
Wortbreite	32	64
Mantissenbits	23 + 1 verborgenes Bit	52 + 1 verborgenes Bit
Exponentenbits	8	11
Exponenten-Bias	127	1023
Null	$e + bias = 0$, $f = 0$	$e + bias = 0$, $f = 0$
Unendlich	$e + bias = 255$, $f = 0$	$e + bias = 2047$, $f = 0$
Keine Zahl	$e + bias = 255$, $f \neq 0$	$e + bias = 2047$, $f \neq 0$
Normale Zahl	$e + bias \in [1, 254]$ $e \in [-126, 127]$ repräsentiert $1.f \cdot 2^e$	$e + bias \in [1, 2046]$ $e \in [-1022, 1023]$ repräsentiert $1.f \cdot 2^e$
min	2^{-126}	2^{-1022}
max	$\approx 2^{128}$	$\approx 2^{1024}$

Tabelle 4.1: Parameter und Eigenschaften des ANSI/IEEE-754-Standards für Gleitkommazahlen einfacher und doppelter Genauigkeit.

- *Round Toward 0* - hier werden alle Nachkommastellen des exakten Ergebnisses abgeschnitten (*Truncation*).
- *Round Toward* ∞ - dieser Modus bedeutet Runden der Ergebnismantisse zur nächstgrößeren Ganzzahl. Für negative Zahlen bedeutet dies das Abschneiden der Nachkommastellen des exakten Ergebnisses, für Positive jedoch Aufrunden.
- *Round Toward* $-\infty$ - dieser Modus bedeutet Runden der exakten Ergebnismantisse zur nächstkleineren Ganzzahl. Für positive Zahlen bedeutet dies das Abschneiden der Nachkommastellen des exakten Ergebnisses, für negative Zahlen Aufrunden des Betrages.

Der Modus *Round To Nearest Even* ist der Standardmodus. Dieser Modus gewährleistet, dass der Fehler der Ergebnismantisse kleiner oder gleich $ulp/2$ ist. Die sich abhängig von der Mantissenbreite daraus ergebenden maximalen relativen Rechenfehler sind in Abbildung 4.19 dargestellt. Ein Modus *Round To Nearest Integer*, der ebenfalls einen Ergebnisfehler kleiner oder gleich $ulp/2$ garantieren würde, ist nicht vorgesehen, da ein solches Rundungsverfahren eine Akkumulation von Rundungsfehlern begünstigen würde. Dies wird an folgendem Beispiel verständlich, bei dem die exakte Ergebnismantisse einer Operation nur zwei Nachkommastellen $(r_{-1}r_{-2})$ ungleich 0 hat:

$$\begin{aligned} r_{-1}r_{-2} &= 00 \text{ Abrunden} \Rightarrow \text{Fehler} = 0 \\ r_{-1}r_{-2} &= 01 \text{ Abrunden} \Rightarrow \text{Fehler} = -0.25 \\ r_{-1}r_{-2} &= 10 \text{ Aufrunden} \Rightarrow \text{Fehler} = 0.5 \\ r_{-1}r_{-2} &= 11 \text{ Aufrunden} \Rightarrow \text{Fehler} = 0.25. \end{aligned}$$

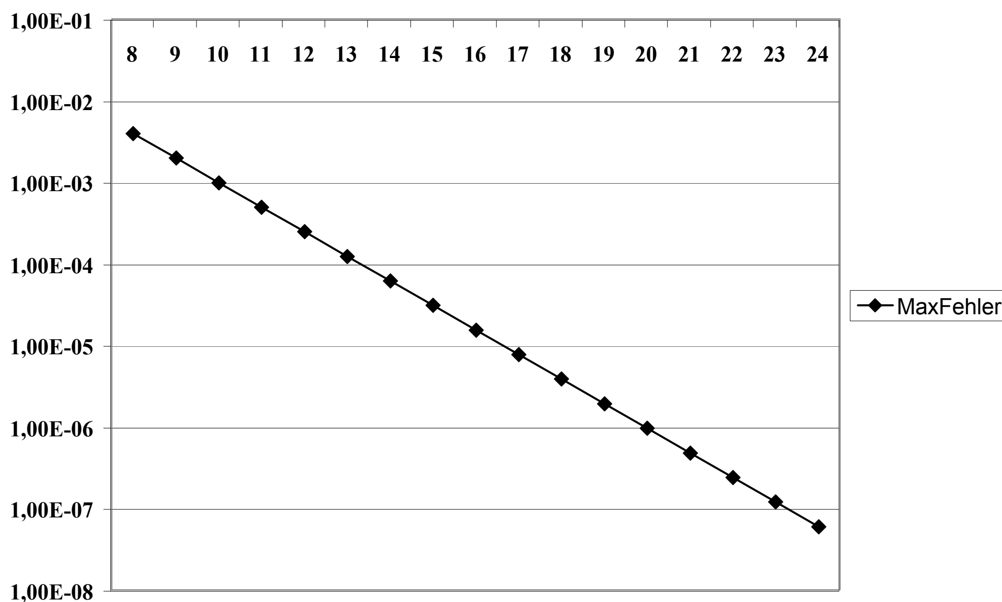


Abbildung 4.19: Maximaler relativer Fehler von Gleitkommaoperationen abhängig von der Mantissenbreite.

Kommen alle vier Fälle mit gleicher Wahrscheinlichkeit vor, ergibt sich ein durchschnittlicher Rundungsfehler von $1/4 \cdot (0 - 0.25 + 0.5 + 0.25) = 0.125$, welcher sich akkumulieren kann.

4.3.3 Operationen auf Gleitkommazahlen

In diesem Abschnitt wird die Umsetzung der elementaren Operationen Addition, Multiplikation, Division und Quadratwurzel vorgestellt. Bevor die Architekturen zur Implementierung dieser Gleitkommaoperatoren erläutert werden, wird zuerst die grundlegende Struktur, der diese Umsetzungen folgen, beschrieben. Motivation dabei ist, die Ausführungen entsprechend der gemeinsamen Struktur zu untergliedern und die Beziehung zwischen den Realisierungen verschiedener Gleitkommaoperationen herzustellen.

Allgemeine Struktur von Gleitkommaoperationen

Abbildung 4.20 zeigt den strukturellen Aufbau eines Gleitkommaoperators. Es lassen sich drei logische Verarbeitungsblöcke separieren. Im *Preparation* genannten Block werden die Eingangsdaten aufbereitet, sodass die Verarbeitung auf Integer-Operationen auf den Exponenten und Festkomma-Operationen auf den Mantissen zurückgeführt werden kann. Es wird ebenfalls ein vorläufiges Ergebnisvorzeichen ermittelt. Im Block *Operation* werden die zentralen Rechenoperationen des Operators ausgeführt. Die damit berechnete vorläufige Ergebnismantisse und der dazugehörige Exponent werden in der Stufe *Normalization* schließlich in eine normalisierte und gerundete Gleitkommazahl überführt. Dabei können sich abhängig von der vorläufigen Mantisse sowohl der Exponent als auch das Vorzeichen (bei der Addition) ändern.

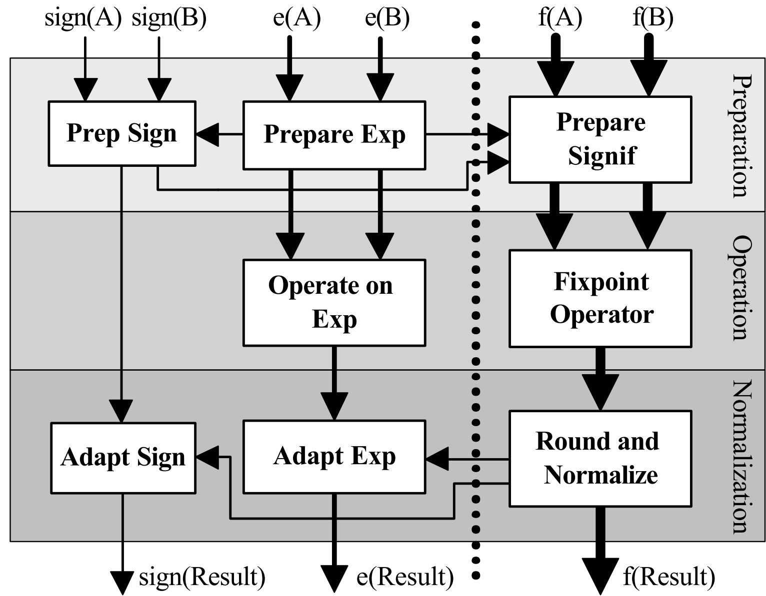


Abbildung 4.20: Allgemeine Struktur eines Gleitkommaoperators.

Addition

Anhand unten stehender Gleichung 4.36 für die Addition zweier Gleitkommazahlen A und B lassen sich die erforderlichen Verarbeitungsschritte eines Fließkomma-Addierers leicht einsehen. Die Mantissen der Zahlen A und B werden hier aus Gründen der Übersichtlichkeit mit s_1 und s_2 bezeichnet, entgegen der Bezeichnung in 4.35. Die Vorzeichen werden ebenfalls vereinfacht als '$\pm$' dargestellt.

$$\begin{aligned}\overbrace{\pm s_{sum}\, 2^{e_{sum}-bias}}^{R=A+B} &= \overbrace{\left(\pm s_1\, 2^{e_1-bias}\right)}^{A} + \overbrace{\left(\pm s_2\, 2^{e_2-bias}\right)}^{B} \\ &= \pm \underbrace{\left(\pm s_1 + s_2\, 2^{e_2-e_1}\right)}_{\text{Festkomma-Operator} \rightarrow s_{add}} 2^{e_1-bias}\end{aligned} \tag{4.36}$$

Preparation Sofort ersichtlich, können die Mantissen erst nach einer erfolgten Ausrichtung entsprechend der Differenz der Exponenten e_2 und e_1 durch einen Fixpunkt-Operator addiert werden. Es sei angenommen, dass $e_1 \geq e_2$ gelte. Diese Bedingung kann dadurch erreicht werden, dass im Fall $e_1 < e_2$ die Zahlen A und B vertauscht werden. Unter dieser Voraussetzung bedeutet $s_2' = s_2 2^{e_2-e_1}$ eine Rechtsverschiebung von s_2 um die Differenz der Exponenten. Haben A und B unterschiedliche Vorzeichen, ist zudem vor der Addition das Komplement einer der beiden Mantissen zu bilden. Ohne Einschränkung kann in diesem Fall das Komplement s_1' der unverschobenen Mantisse s_1 gebildet werden, ansonsten gilt $s_1' = s_1$. Als vorläufiges Vorzeichen v des Ergebnisses wird dann das Vorzeichen des zu s_2' gehörigen Operanden genommen, welches im Schritt *Normalization* abhängig vom Ergebnis der Integer-Operation gegebenenfalls

korrigiert wird.

Operation Nach der Vorbereitung der Mantissen (s'_1 und s'_2) ist in dem Integer-Operator lediglich die Summe dieser Zahlen s_{add} zu berechnen. Für den vorläufigen Ergebnisexponenten wird der größere der Exponenten e_1 und e_2 ausgewählt, welcher als e'_1 bezeichnet werden soll (dies kann bereits in der *Preparation*-Stufe geschehen).

Normalization Bis zu dieser Stelle haben wir als Ergebnis ein vorläufiges Vorzeichen v, eine nicht-normierte, eventuell negative Mantisse s_{add} und den vorläufigen Ergebnisexponenten e'_1. Was zu tun bleibt, ist die Rückführung des Ergebnisses in eine normierte Gleitkommazahl. Dazu ist im Fall $s_{add} < 0$ das Komplement dieser Zahl zu bilden, die Zahl so zu normieren, dass das führende Bit die Wertigkeit 2^0 hat und das Ergebnis zu runden. Wenn für die Mantissen vor der Addition galt:

$$\begin{aligned} s_1 &\in [1,2) \\ s'_1 &\in (-2,2) \\ s_2 &\in [1,2) \\ s'_2 &\in [0,2), \end{aligned} \tag{4.37}$$

dann gilt für das Additionsergebnis:

$$\begin{aligned} s_{add} &\in (-2,4) \\ |s_{add}| &\in [0,4). \end{aligned} \tag{4.38}$$

Um s_{add} zu normieren, kann ein Rechtsshift um eine Stelle auftreten. Da bei ungleichen Vorzeichen von s'_1 und s'_2 eine Auslöschung von Bits stattfinden kann, muss $|s_{add}|$ eventuell linksverschoben werden, wobei dann die Anzahl führender Nullen dieser Zahl ab der Position 2^0 die Verschiebungsweite festlegt. Nach der Normierung erfolgt die Rundung der Fixpunktzahl auf die Anzahl von Mantissestellen des Ergebnisses. Beim Runden kann ein Übertrag entstehen, weshalb eine zweite Normierungsstufe erforderlich ist, die den Übertrag feststellt und gegebenenfalls die Mantisse um eine Position nach rechts verschiebt. Für jede Positionsverschiebung in den Normierungsstufen muss der vorläufige Ergebnisexponent entsprechend korrigiert werden. Die Bildung des Komplements für $s_{add} < 0$ kann mit der Rundung zusammengefasst werden. Da die Normierung dann vor der Komplementlogik liegt, muss die Ermittlung der Verschiebungsweite auch bei negativen Zahlen möglich sein. Dazu werden anstatt der Nullen die führenden Einsen gezählt. Bei Vorliegen einer negativen Zahl ist keine Rechtsverschiebung notwendig, da $s_{add} > -2$ gilt (Bit mit Wertigkeit 2^1 ist 1). Die Zahl der führenden Einsen ab der Bitposition 2^0 gibt die Verschiebungsweite der erforderlichen Linksverschiebung vor. Das Vorzeichen des Ergebnisses wird aus dem vorläufigen Vorzeichen v bestimmt, welches negiert wird, falls $s_{add} < 0$ gilt.

Blockschaltbild Ein Blockschaltbild eines Addierers für Gleitkommazahlen ist in Abbildung 4.21 gezeigt. Das Element *Unpack* zerlegt die eingehenden Fließkommazahlen in ihre Bestandteile Vorzeichen, Exponent und Mantisse. Bei den Mantissen wird die verborgene führende 1

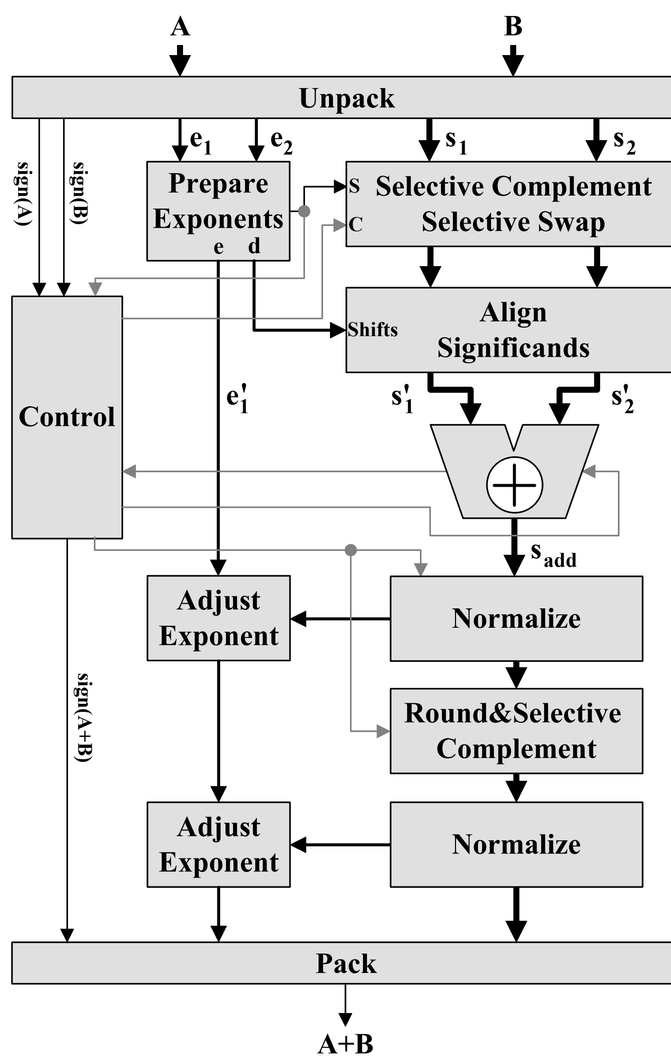


Abbildung 4.21: Funktioneller Aufbau eines Addierers für Gleitkommazahlen.

vorangestellt. Das Modul *Prepare Exponent* bildet die Differenz der eingehenden Exponenten von A und B. Bei negativer Differenz wird im rechts nebenstehenden Block die Funktion *Selective Swap* aktiviert, um die Mantissen von A und B zu vertauschen. Die Funktion *Selective Complement* dieses Blocks wird durch den Block *Control* aktiviert, wenn die Vorzeichen von A und B verschieden sind. Anschließend erfolgt im Modul *Align Significands* die Verschiebung der Mantisse mit zugehörigem größeren Exponenten um $|e_2 - e_1|$ Stellen. An den Ausgängen dieses Moduls liegen dann die für die Addition vorbereiteten Mantissen s_1' und s_2' vor. Beim Integer-Addierer ist zu sehen, dass ein Signal von *control* an den Carry-Eingang geht. Dieses Signal ist genau dann gesetzt, wenn auch die oben erwähnte Funktion *Selective Complement* aktiviert ist. Damit braucht dort nur das 1er-Komplement gebildet zu werden. Diese Schaltung entspricht damit genau der Anordnung von Abbildung 4.2. Im Fall der Subtraktion zeigt ein aktiviertes Carry-Out des Addierers dem Block *control* an, dass der Ausgang des Addierers positiv ist, bei Carry-Out gleich 0 ist die ausgehende Fixpunktzahl negativ. Letzteres führt dazu, dass *Control* die Bildung des Komplements der normierten Ergebnismantisse veranlasst. Die zweistufige Normierung mit zwischenliegendem Modul für die bedingte Komplementbildung und Rundung (*Round & Selective Complement*) arbeitet wie oben beschrieben. Die Anpassung des vorläufigen Ergebnisexponenten e_1' entsprechend der Verschiebungen in den Normierungsstufen geschieht in den *Adjust Exponent* genannten Modulen. Am Ende der Schaltung werden das in *Control* ermittelte Ergebnisvorzeichen, der Ergebnisexponent und die Ergebnismantisse im Modul *Pack* zu einer Gleitkommazahl $A+B$ zusammengefasst.

Multiplikation

Ähnlich wie bei der Addition wird in diesem Abschnitt anhand der Gleichung 4.39 die Verfahrensweise bei der Gleitpunktmultiplikation entwickelt. Die Gleichung zeigt, wie die Multiplikation als Operationen auf Fixpunktzahlen formuliert werden kann. Es lässt sich sofort sehen, dass Exponenten und Mantissen unabhängig voneinander verarbeitet werden können.

$$\begin{aligned} \overbrace{\pm s_{prod}\, 2^{e_{prod}-bias}}^{R=A\cdot B} &= \overbrace{\left(\pm s_1\, 2^{e_1-bias}\right)}^{A} \cdot \overbrace{\left(\pm s_2\, 2^{e_2-bias}\right)}^{B} \\ &= \pm \underbrace{(s_1 \cdot s_2)}_{\text{Festkomma-Operator}} \quad 2^{(e_1+e_2-bias)-bias}. \end{aligned} \tag{4.39}$$

Preparation Die *Preparation*-Stufe beinhaltet lediglich das Entpacken der Gleitkommazahlen in Vorzeichen, Exponenten und Mantissen sowie die Erzeugung des Ergebnisvorzeichens $sign(A\cdot B) = sign(A) \oplus sign(B)$.

Operation Die Mantissen können direkt in einem Festkommamultiplizierer, welcher nach Abschnitt 4.2 identisch mit einem Ganzzahlmultiplizierer ist, zur vorläufigen Ergebnismantisse s_{mult} verarbeitet werden. Liegen Mantissen der Breite k vor, hat s_{mult} $2k$ Stellen. Bei der Multiplikation müssen die Exponenten addiert werden. Da die Exponenten in Bias-Darstellung vorliegen, muss

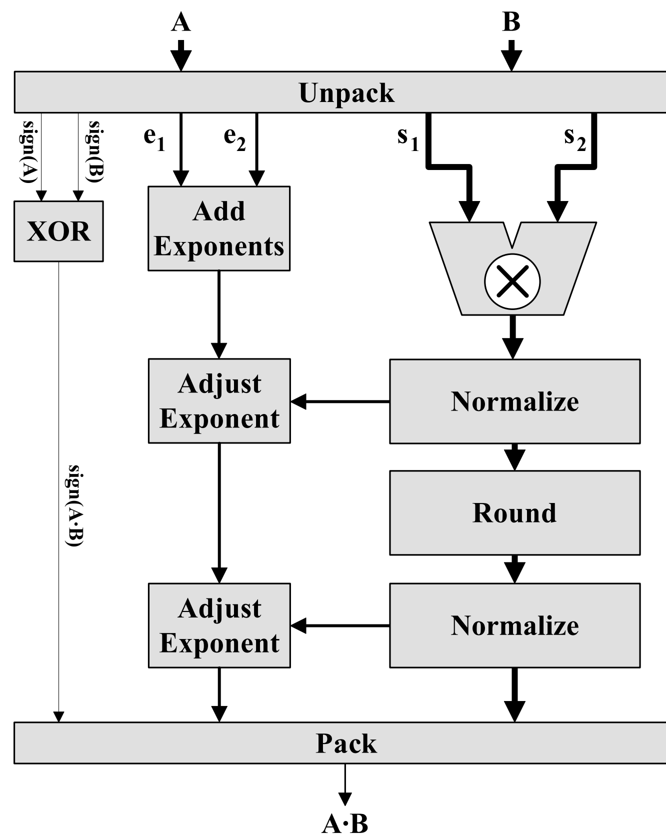


Abbildung 4.22: Funktioneller Aufbau eines Multiplizierers für Gleitkommazahlen.

nach der Addition der Wert von Bias subtrahiert werden, um den vorläufigen Ergebnisexponenten zu bekommen.

Normalization Für die Fixpunktmantisse s_{mult} gilt folgende Beziehung:

$$s_{mult} \quad \in \quad [1,4). \tag{4.40}$$

Im Normalisierungsschritt kann also eine Rechtsverschiebung um eine Position erforderlich werden. Da im Rundungsschritt ein Überlauf stattfinden kann, muss eine zweite Normalisierung folgen.

Blockschaltbild Abbildung 4.22 zeigt das Blockschaltbild für die Gleitkommamultiplikation. Weil die zu verknüpfenden Mantissen nicht wie bei der Addition abhängig von den Vorzeichen und Exponenten vorverarbeitet werden müssen, ergibt sich eine wesentlich einfachere Schaltung als in Abbildung 4.21. Ausschließlich die Korrektur des vorläufigen Exponenten durch die Normalisierung der Mantisse verknüpft die Pfade von Exponenten- und Mantissenverarbeitung.

Division

Das Verfahren zur Berechnung der Gleitkommadivision ergibt sich aus folgender Gleichung:

$$\begin{aligned}\overbrace{\pm s_{quot}\, 2^{e_{quot}-bias}}^{R=A/B} &= \overbrace{\left(\pm s_1\, 2^{e_1-bias}\right)}^{A} / \overbrace{\left(\pm s_2\, 2^{e_2-bias}\right)}^{B} \\ &= \pm \underbrace{(s_1/s_2)}_{\text{Festkomma-Operator}} \quad 2^{(e_1-e_2+bias)-bias}.\end{aligned} \tag{4.41}$$

Wie bei der Multiplikation folgt, dass die Verarbeitung von Exponenten und Mantissen unabhängig voneinander erfolgen kann (abgesehen von der Normalisierung).

Preparation Die *Preparation*-Stufe beinhaltet lediglich das Entpacken der Gleitkommazahlen in Vorzeichen, Exponenten und Mantissen, sowie die Erzeugung des Ergebnisvorzeichens $sign(A \cdot B) = sign(A) \oplus sign(B)$.

Operation Die Mantissen können in einem Festkommadividierer zur vorläufigen Ergebnismantisse s_{div} verarbeitet werden. Dazu wird die Mantisse des Operanden A erweitert, indem nach dem LSB Nullen angefügt werden. Der Dividierer muss abhängig vom gewählten Rundungsmodus mehr als k Ergebnisbits erzeugen und ebenfalls den Rest der Division berücksichtigen.

Bei der Division müssen die Exponenten addiert werden. Da die Exponenten in Bias-Darstellung vorliegen, muss nach der Subtraktion der Wert von Bias addiert werden, um den vorläufigen Ergebnisexponenten zu bekommen.

Normalization Für die Fixpunktmantisse s_{div} gilt folgende Beziehung:

$$s_{div} \in (1/2, 2). \tag{4.42}$$

Im Normalisierungsschritt kann also eine Linkssverschiebung um eine Position erforderlich werden. Da im Rundungsschritt ein Überlauf stattfinden kann, muss eine zweite Normalisierung folgen. Für beide Normalisierungsschritte muss eine Anpassung des Exponenten erfolgen.

Blockschaltbild Abbildung 4.23 zeigt den Aufbau eines Gleitkommazahl-Dividierers. Es ergibt sich eine Struktur, die sehr ähnlich der des Multiplizierers ist. Die Pfade zur Verarbeitung der Exponenten und Mantissen sind lediglich aufgrund der Normalisierungsblöcke miteinander verbunden.

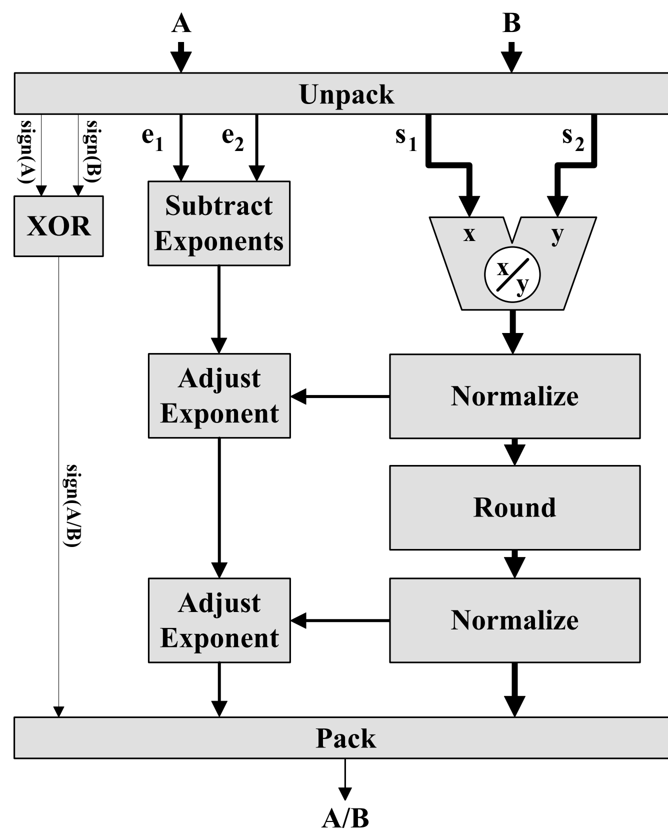


Abbildung 4.23: Funktioneller Aufbau eines Dividierers für Gleitkommazahlen.

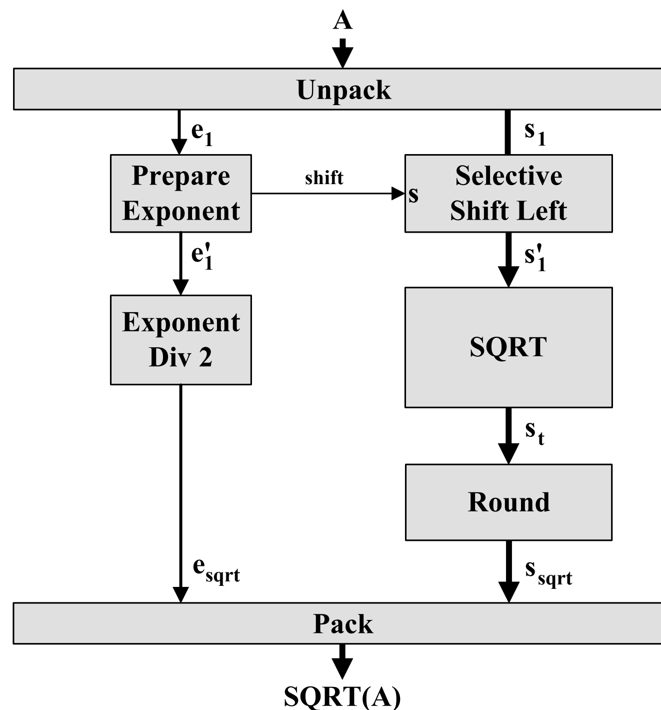


Abbildung 4.24: Funktioneller Aufbau eines Quadratwurzeloperators für Gleitkommazahlen.

Quadratwurzel

Die Verfahrensweise bei der Berechnung der Quadratwurzel einer Gleitkommazahl ergibt sich aus folgender Gleichung:

$$\overbrace{\pm s_{sqrt}\, 2^{e_{sqrt}-bias}}^{R=\sqrt{A}} = \Big(\overbrace{\pm s_1\, 2^{e_1-bias}}^{A}\Big)^{\frac{1}{2}} = \pm \underbrace{\sqrt{s'_1}}_{\text{Festkomma-Operator}} \quad 2^{(e'_1-bias)/2} \tag{4.43}$$

$$s'_1 = \begin{cases} s_1 & : \ e_1 - bias \quad \text{gerade} \\ s_1\, 2 & : \ e_1 - bias \quad \text{ungerade} \end{cases}$$

$$e'_1 = \begin{cases} e_1 & : \ e_1 - bias \quad \text{gerade} \\ e_1 - 1 & : \ e_1 - bias \quad \text{ungerade} \end{cases}$$

Eine Architektur für eine Hardwareimplementierung dieser Gleichungen ist in Abbildung 4.24 gezeigt und wird im Folgenden beschrieben.

Preparation Bei der Quadratwurzel ist der Exponent zu halbieren. Da ungeradzahlige Exponenten nicht ganzzahlig durch zwei geteilt werden können, wird in diesem Fall die Normierung der Zahl aufgehoben, indem die Mantisse um eine Stelle linksverschoben und im Gegenzug der

Exponent um Eins verringert wird. Da der Wert von *bias* ungeradzahlig ist, veranlasst ein geradzahliges e_1 ($e_1 - bias$ ungerade), dass sich s'_1 aus der um eine Stelle nach links verschobenen Mantisse s_1 ergibt, ansonsten gilt $s'_1 = s_1$. Ist e_1 gerade, wird dementsprechend $e'_1 = e_1 - 1$, ansonsten $e'_1 = e_1$ gesetzt. Es sei hier erwähnt, dass die Verringerung des Exponenten um Eins mit der in der *Operation*-Stufe erfolgenden Weiterverarbeitung des Exponenten zusammengefasst werden kann.

Operation In dieser Stufe ist die Quadratwurzel aus der Festkommazahl s'_1 mit Ergebnis s_t zu berechnen. Der Exponent e'_1 ist durch zwei zu dividieren, wobei berücksichtigt werden muss, dass der resultierende Exponent wieder in Darstellung mit Bias vorliegen muss. Dazu wird $e_{sqrt} = (e'_1 + bias)/2$ berechnet.

Normalization Lag die Mantisse s'_1 nach Voraussetzung im Intervall $[1,4)$, so gilt $s_t \in [1,2)$. Es ist also keine Normalisierungsverschiebung notwendig. Nach dem Runden wird ebenfalls keine erneute Normalisierung notwendig, denn es gilt sogar $s'_1 \in [1, 4 - 2ulp]$ und damit:

$$(2 - ulp)^2 = 4 - 2ulp + ulp^2 > 4 - 2ulp \geq s'_1$$

Also kann die gerundete Wurzel s_{sqrt} in keinem Fall größer als $2 - ulp$ werden und ist deshalb bereits die Mantisse in der normalisierten Darstellung.

Blockschaltbild Abbildung 4.24 zeigt die Struktur des Quadratwurzel-Operators. Der Implementierungsaufwand fast vollständig durch die Festkommaoperation bestimmt.

4.4 Logarithmische Zahlen

Eine wichtige Alternative zur Verwendung von Gleitkommzahlen ist die Anwendung eines logarithmischen Zahlensystems. Dieses System basiert auf der Idee, dass Multiplikationen und Divisionen sich durch die einfache Addition und Subtraktion von Logarithmen exakt berechnen lassen. So lässt sich beispielsweise in Anwendungen mit einem hohen Anteil von Multiplikationen eine drastische Reduzierung der erforderlichen Hardwareressourcen erreichen [115]. In diesem Abschnitt werden die Definition des logarithmischen Zahlensystems (engl. **L**ogarithmic **N**umber **S**ystem = LNS) und seine Implikationen bezüglich arithmetischer Operationen diskutiert. Es wird insbesondere untersucht, unter welchen Voraussetzungen dieses Zahlensystem für die Hardwareimplementierung von Spezialanwendungen einen Vorteil gegenüber der Gleitkommaarithmetik bringt.

4.4.1 Darstellung im logarithmischen Zahlensystem

Eine Zahl x wird im logarithmischen Zahlensystem als Festkommazahl e des Logarithmus von x zu einer Basis b dargestellt. Im Folgenden soll immer von $b = 2$ ausgegangen werden (dies ist

auch die am häufigsten verwendete Basis). Um auch negative Zahlen darstellen zu können, wird ein Vorzeichen s hinzugefügt. Die Darstellung (s,e) repräsentiert also folgenden Wert:

$$x = (-1)^s \cdot b^e. \tag{4.44}$$

Die Festkommazahl e sei eine 2er-Komplementzahl mit k Ganzzahlstellen und l Nachkommastellen:

$$\begin{aligned} e &= e_{k-1} \cdots e_0 . e_{-1} \cdots e_{-l} \\ &= -2^{k-1} e_{k-1} + \sum_{i=-l}^{k-2} e_i \cdot 2^i \end{aligned}$$

Die Anzahl der Vorkommastellen legt die Größenordnung der kleinsten und größten darstellbaren Zahl fest, ähnlich wie der Exponent bei Gleitkommazahlen. Die Anzahl der Nachkommastellen gibt die Präzision der Darstellung vor, analog zur Mantissenbreite bei Gleitkommazahlen. Hier jedoch ist die Intervallbreite aufeinanderfolgender Zahlen von allen Bits von e abhängig (bei Gleitkommazahlen nur vom Exponenten). Es kann gezeigt werden, dass ein LNS mit Logarithmen in 32-Bit-Festkommazahldarstellung eine höhere Genauigkeit ermöglicht als ein Single-Precision-Gleitkommaformat [16].

4.4.2 Operationen auf logarithmischen Zahlen

In einem LNS nehmen die wichtigsten arithmetischen Operationen auf den Zahlen $x = 2^s$ und $y = 2^t$ folgende Gestalt an:

$$\log_2(x+y) = s + \log_2(1+2^{t-s}) \tag{4.45}$$
$$\log_2(x-y) = s + \log_2(1-2^{t-s}) \tag{4.46}$$
$$\log_2(x \cdot y) = s + t \tag{4.47}$$
$$\log_2(x/y) = s - t \tag{4.48}$$
$$\log_2(\sqrt[n]{x}) = \frac{s}{n} \tag{4.49}$$
$$\log_2(x^n) = n \cdot s \tag{4.50}$$

Die Gleichungen wurden für positive Zahlen aufgestellt. Im allgemeinen Fall wird eine zusätzliche Logik geringer Komplexität für die Verarbeitung der Vorzeichen benötigt. Wie an obigen Gleichungen zu sehen ist, können Multiplikation und Division im LNS also durch einfache Addition oder Subtraktion der logarithmischen Zahlen ausgeführt werden - mit einem Rechenfehler identisch Null. Auch Wurzeln und ganzzahlige Potenzen nehmen im LNS eine sehr einfache Form an, denn es müssen nur Divisionen und Multiplikationen mit ganzzahligen Werten durchgeführt werden. Die Berechnung von Summen und Differenzen ist jedoch ungleich schwieriger. Diese Operationen erfordern die Auswertung einer nichtlinearen Funktion $F(v) = \log_2(1 \pm 2^v)$. Hierfür können Interpolationsverfahren, insbesondere Table-Look-Up-basierte Verfahren, angewandt werden, wie sie in Abschnitt 4.5 vorgestellt werden. Mit der Genauigkeit skaliert die Grö-

ße der Look-Up-Tabellen exponentiell. Coleman und Chester [16] berichten für Single-Precision-Operationen von Tabellengrößen $O(100)$ KBits für die Addition und $O(300)$ KBits für die Subtraktion bei Anwendung einer Taylor-Näherung ersten Grades, optimiert durch eine variierende Intervallbreite für die Interpolation und einen Fehlerkorrekturmechanismus.

Die Skalierung der LUT-Größen für die Addition und Subtraktion schränkt die Anwendbarkeit des logarithmischen Zahlensystems bei FPGA-Implementierungen auf Anwendungen mit geringen Genauigkeitsanforderungen ein. Hinzu kommt ein hoher Aufwand für die Konvertierung zwischen Gleitkommazahlen und logarithmischen Zahlen (wird in Abschnitt 4.5 diskutiert). Da es in diesem Zahlensystem für die Implementierung von Addierern auf FPGAs gegenüber einer LUT-basierten Implementierung keine praktikable Alternative gibt, eignet sich der Ansatz eines logarithmischen Zahlensystems nur dann, wenn die zugrundeliegende Anwendung einen geringen Anteil von Additionen aufweist oder der FPGA-Typ reich an RAM-Ressourcen ist.

4.4.3 Semilogarithmische Zahlen

Bei dieser Zahlendarstellung handelt es sich um eine Mischung aus logarithmischer und Gleitkommazahldarstellung. Wie J.-M. Muller in [80] zeigt, ermöglicht diese Darstellung einen Kompromiss zwischen Rechengeschwindigkeit und der Größe der benötigten LUTs für die arithmetischen Grundoperationen. Eine Zahl x wird repräsentiert durch:

$$x = s \cdot m_{k,x} \cdot 2^{e_{k,x}}, \tag{4.51}$$

mit einem ganzzahligen Parameter k, wobei $e_{k,x}$ ein Vielfaches von 2^{-k} ist, mit:

$$2^{e_{k,x}} \leq |x| < 2^{e_{k,x}+2^{-k}}. \tag{4.52}$$

Dann kann x durch folgende Darstellung repräsentiert werden:

$$x = s \cdot 1.\overbrace{\underbrace{000\ldots000}_{k\text{ Nullen}}\text{XXX}\ldots\text{XXX}}^{n\,Bits} \cdot 2^{e_{k,x}} \tag{4.53}$$

Die Zahl $e_{k,x}$ kann als Näherung von $\log_2 |x|$ auf k Nachkommastellen verstanden werden. Mit k verringert sich die Breite der zu speichernden Mantisse von n auf $n-k$. Für $k=0$ ergibt sich die Gleitkommadarstellung, für $k \geq n$ das logarithmische Zahlenformat. Für die Diskussion der arithmetischen Operatoren in diesem Zahlensystem sei auf [80] verwiesen. Als besonders interessant stellen sich die Zahlenformate mit $k \approx n/2+2$ heraus. In diesem Fall bleibt der Aufwand für Multiplikation und Division ähnlich gering wie bei den logarithmischen Zahlen. Für Addition und Subtraktion werden dagegen wesentlich kleinere LUTs (wenige 10 KBits für Single-Precision), ein $(n-k+1) \times n$-Bit-Multiplizierer, zwei Shift-Operationen und eine geringe Menge zusätzlicher Logik benötigt. Die Umwandlung zwischen Gleitkommadarstellung und semilogarithmischer Darstellung und umgekehrt kann mit ähnlichem Aufwand wie bei der Addition durchgeführt werden.

Diese Zahlendarstellung ist eine interessante Alternative gegenüber der Verwendung von Gleitkommazahlen. Die Komplexität von Multiplikationen und Divisionen wird stark verringert. Addition und Subtraktion benötigen dagegen ähnlich viele Logikressourcen wie bei den entsprechenden Gleitkommaoperationen, zusätzlich sind hier jedoch Look-Up-Tables erforderlich, die beispielsweise bei Virtex-II-FPGAs durch wenige Block-RAMs abgebildet werden können. Für Anwendungen, bei denen deutlich mehr Multiplikationen und Divisionen als Additionen durchgeführt werden müssen, kann eine signifikante Verringerung des Logikressourcenbedarfs erreicht werden, vorausgesetzt, der FPGA-Typ hat genügend RAM-Ressourcen. Sind hingegen ähnlich viele Additionen wie Multiplikationen zu berechnen, ergibt sich grob abgeschätzt keine Ressourcenersparnis. Es findet dann lediglich eine Verlagerung von Multiplizierern auf RAM-Ressourcen statt.

4.5 Berechnung von Funktionen

Für transzendente Funktionen wie die trigonometrischen Funktionen, die Exponentialfunktion und den Logarithmus gibt es keine exakten Berechnungsverfahren wie bei den Grundoperationen. Diese Funktionen können jedoch durch Konvergenzmethoden numerisch beliebig genau berechnet werden. Einige dieser Methoden eignen sich auch für eine direkte Hardwareumsetzung und werden damit für eine FPGA-Implementierung interessant. Alternativ dazu können in jedem Fall Funktionswerte unter Benutzung vorausberechneter Tabellenwerte gewonnen werden, wobei zur Verringerung der Tabellengröße in den meisten Fällen eine Interpolation durch Polynome angewandt wird. Im Folgenden werden verschiedene Tabellenbasierte Verfahren und Konvergenzmethoden beschrieben und hinsichtlich des Implementierungsaufwands diskutiert. Die beschriebenen Verfahren sind die Grundlage für Rechenwerke in modernen Prozessoren zur Auswertung der elementaren Funktionen wie Quadratwurzel und Kehrwertbildung mit doppelter Genauigkeit. Ziel dieses Abschnittes ist es, die Verfahren im Hinblick darauf zu diskutieren, ob sie für eine Implementierung auf FPGAs interessant sind.

4.5.1 Table-Look-Up-basierte Methoden

Diese Methoden basieren darauf, Look-Up-Tables (LUTs) zu verwenden, in denen Funktionswerte oder Interpolationsparameter gespeichert sind, mit dem Ziel, durch die Vorausberechnung einen großen Teil des Rechenaufwands für die Funktionsauswertung einzusparen.

Direktes Table-Look-Up

Das Verfahren, Funktionswerte durch direkte Abbildung des Arguments auf eine Zahl über eine LUT zu ermitteln, eignet sich nur für Anwendungen mit sehr geringen Genauigkeitsanforderungen. Die Tabellengröße steigt exponentiell ($\propto 2^k \cdot k$) mit der zu erzielenden Genauigkeit von k Bits. Für eine Ergebnisgenauigkeit in der Größenordnung von Single-Precision ist diese Methode deshalb in der Regel nicht anwendbar, da alleine die LUT zur Bestimmung einer 24-Bit-Mantisse bereits 48 MBytes ($2^{24} \cdot 24$ Bits) groß sein müsste.

LUT-basierte polynomiale Approximation

Eine Kombination von Table-Look-Up und einer stückweisen Näherung der Funktion durch Polynome geringen Grades erlaubt es, für eine vorgegebene Genauigkeit wesentlich kleinere LUTs als im Fall des direkten Table-Look-Ups zu verwenden. Durch den Grad der Interpolationspolynome kann die LUT-Größe in weiten Bereichen variiert werden.

Stückweise lineare Approximation Eine einfache Methode ist die stückweise lineare Approximation einer Funktion durch eine Taylorentwicklung ersten Grades zwischen den durch die führenden $k/2$ Bits des Funktionsarguments gegebenen Stützstellen. Das Argument x wird also unterteilt in zwei $k/2$-Bit-Segmente x_1 und x_2, $x = (x_1, x_2)$. Die Berechnung der linearen Interpolation geschieht dann auf folgende Weise:

$$f(x) \approx C_1(x_1) + x_2 \cdot C_2(x_1), \tag{4.54}$$

wobei für die einfache Taylornäherung $C_1(x_1) = f(x_1)$ und $C_2(x_1) = f'(x_1)$ gilt. Durch die Modifikation von C_1 und C_2 kann eine bessere Approximation erreicht werden. Für die Approximationskoeffizienten werden zwei LUTs benötigt, die für k Bits Genauigkeit mit $2^{k/2} \cdot (k/2 + k)$ in der Größe skalieren (≈ 150 KBit für $k = 24$) gegenüber $2^k \cdot k$ (≈ 390 MBit für $k = 24$) beim direkten Table-Look-Up. Zusätzlich werden ein $k/2 \times k/2$-Bit-Multiplizierer und ein k-Bit-Addierer benötigt.

In [22] wird anhand der Kehrwertbildung demonstriert, wie bei gleich bleibender Genauigkeit und der Verwendung eines $(k/2+3) \times (k/2+3)$-Bit-Multiplizierers sowie eines $(k+2)$-Bit-Addierers eine Kompression der LUT erreicht werden kann (104 KBit für Single-Precision).

In [116] wird eine ebenfalls auf der stückweise linearen Approximation basierende Methode vorgeschlagen, welche die Näherung verschiedener Potenzen mit Hilfe einer LUT ähnlicher Größe und einem $(k+1) \times (k+1)$-Bit-Multiplizierer durchführt. Die Methode eignet sich für Potenzen $x^{\pm 2^p}$(p ganzzahlig) und $x^{\pm 2^{p_1} + 2^{-p_2}}$($p_1$ ganzzahlig, $p_2 \geq 0$ ganzzahlig) und wird anhand der Berechnung von Kehrwerten, Quadratwurzeln und Potenzen mit $p = 3$ demonstriert. Je nach Operation wird von erforderlichen LUT-Größen von einigen 10 KBit berichtet (z.B. 50 KBit für Kehrwertbildung und 200 KBit für Potenzen mit $p = 3$).

Die moderaten LUT-Größen und die geringe Komplexität der Berechnung machen diese Methoden sehr interessant für eine Implementierung auf Virtex-II-FPGAs, da diese über Block-Ram- und Block-Multiplizierer-Elemente verfügen (siehe Abschnitt 3.1.2). Denn in diesem Fall können solche LUTs in einigen wenigen Block-RAM-Elementen (18 KBit pro Element) gespeichert werden. Da die Multiplikation für die Virtex-II-FPGAs durch die Block-Multiplizierer übernommen werden kann, lassen sich damit Operationen wie Quadratwurzel- und Kehrwertbildung fast gänzlich ohne die allgemeinen Logikressourcen implementieren.

Näherungen höheren Grades Mit Näherungen höheren Grades kann zwar eine weitere Reduktion der LUT-Größe erreicht werden. Gleichzeitig steigt jedoch die Anzahl der Multiplikationen und Additionen, die berechnet werden müssen. In [93] wird beispielsweise ein Interpolator

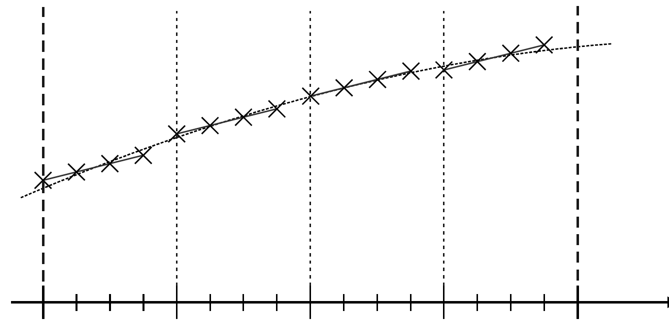


Abbildung 4.25: Lineare Approximation nach der Bipartite-Table-Methode.

zweiten Grades für eine Minimax-Approximation vorgestellt, für den zur Berechnung der elementaren Funktionen mit Single-Precision eine LUT-Größe von $O(20\text{ KBits})$ genügt. Für eine detaillierte Diskussion siehe z.B. [91].

Multipartite-Table-Methoden Die bisher beschriebenen Verfahren zur stückweisen polynomialen Approximation benötigen alle mindestens einen Multiplizierer. Multipartite-Table-Methoden basieren zwar auch auf dem Prinzip der stückweisen Approximation durch Taylorpolynome, ermöglichen jedoch eine Implementierung der stückweise linearen Interpolation ohne Multiplikationen. Die Idee bei diesen Methoden besteht darin, die Steigungen für die lineare Interpolation über mehrere durch die Stützpunkte vorgegebenen Intervalle unverändert zu lassen. Bei den Methoden mit zwei LUTs (Bipartite-Table-Methode) wird das Funktionsargument x unterteilt in drei Stücke x_1, x_2 und x_3 mit n_1, n_2 und n_3 Bits Breite, $x = (x_1, x_2, x_3)$. Anstelle der Gleichung 4.54 für die lineare Interpolation, wird folgende Gleichung ausgewertet:

$$f(x) \approx C_1(x_1, x_2) + C_2(x_1, x_3), \tag{4.55}$$

wobei $C_1 = f((x_1, x_2, 0) + \xi(x_1, x_2))$ und $C_2 = (0, 0, x_3) \cdot f'((x_1, 0, 0) + \chi(x_1))$. Die Werte ξ und χ dienen der Verbesserung der Approximation. Beispielsweise entspricht ξ in [112] der halben durch x_3 gegebenen Intervallbreite, χ der halben durch x_2 gegebenen Intervallbreite. Die Größe der Tabelle für C_2 wird bei diesem Ansatz aufgrund der Symmetrie der linearen Approximation halbiert. Abbildung 4.25 veranschaulicht dieses Verfahren für $n_1 = n_2 = n_3 = 2$.

Es ist möglich, mit der Bipartite-Table-Methode die Tabellengröße für die LUTs gegenüber der Methode des direkten Table-Look-Up deutlich zu verringern. So ergab sich in [21] für den Fall der Kehrwertbildung mit einer zweiteiligen Tabelle und einer redundanten Darstellung der Ergebnisse eine Tabellengröße von $2^{2/3\,k} \cdot k + 2^{2/3\,k} \cdot 2/3\ k$ Bit ($\approx$ 2600 KBits für Single-Precision). Die Methode aus [112] skaliert in der LUT-Größe ähnlich, führt aber zu einer besseren Kompression der Tabellen. So wird bei Single-Precision für die Kehrwertbildung eine LUT-Größe von etwa 1900 KBit erreicht (variiert abhängig von der Anzahl verwendeter Guard-Bits). Eine weitere Verringerung der Größe kann durch eine Unterteilung der Tabelle für C_2 in kleinere Tabellen erreicht werden, jedoch mit einem erhöhten Aufwand für die Addierer. Solche Multipartite-Table-Verfahren wurden in [112] ebenfalls diskutiert und für verschiedene Funktionen angewandt. Die Verwendung von 5 Tabellen für die Single-Precision-Berechnung von $1/x$

führt zu einem Speicheraufwand von insgesamt 620 KBit. Für eine vergleichende Diskussion der verschiedenen Multipartite-Table-Verfahren sei auf [23] verwiesen.

4.5.2 Digit-Recurrence-Algorithmen

Digit-Recurrence-Algorithmen sind iterative Verfahren, welche in jedem Schritt eine feste Zahl von Ergebnisstellen produzieren. Die Restoring/Non-Performing- und Non-Restoring-Division und -Quadratwurzel sowie die SRT-Division, wie sie in den Abschnitten 4.1.4 und 4.1.5 beschrieben wurden, fallen in diese Kategorie von Algorithmen.

Für serielle Architekturen ergibt sich eine sehr geringe Schaltungskomplexität. Durch eine Parallelisierung durch Abrollen der Iterationen in eine Pipeline kann der Datendurchsatz vervielfacht werden, mit proportional dazu skalierendem Ressourcenaufwand. Für die Division und Quadratwurzel wurden bereits Beispiele vollständig parallelisierter Implementierungen gegeben.

Da Digit-Recurrence-Algorithmen Verfahren mit linearer Konvergenz sind, ergeben sich hohe Latenzzeiten. Während dies für Recheneinheiten in Mikroprozessoren fatal ist, spielt die Latenzzeit bei Implementierungen von Spezialpipelines für fest vorgegebene Rechenverfahren, wie es in dieser Arbeit der Fall ist, keine Rolle. Hier kommt es nur auf den Datendurchsatz an. Deshalb sind Digit-Recurrence-Algorithmen bei FPGA-Implementierungen sehr weit verbreitet.

Im nächsten Abschnitt wird kurz auf eine weit verbreitete Technik zur Berechnung der nichtalgebraischen Funktionen – die CORDIC-Methode – eingegangen, welche ebenfalls zu den Digit-Recurrence-Algorithmen gehört.

CORDIC-Verfahren

Es soll hier nicht detailliert auf diese Methode eingegangen werden - als Standardmethode ist sie Teil jedes Lehrbuchs der Computerarithmetik. Prinzipiell beruht die Methode auf der Idee, dass durch die Rotation des Einheitsvektors $(0,1)$ um einen Winkel ϕ der Ergebnisvektor die Koordinaten $(cos(\phi), sin(\phi))$ hat. Durch eine solche Rotation können also die trigonometrischen Funktionen *Sinus* und *Kosinus* berechnet werden. Die Methode stellt im Wesentlichen ein geschicktes Verfahren bereit, solche Rotationen iterativ mit wenig Hardwareaufwand zu berechnen – es werden nur Additionen, Shift-Operationen und Table-Look-Ups benötigt. Mathematisch gehört die Iterationsmethode zu den Verfahren der additiven Normalisierung. Das verallgemeinerte CORDIC-Verfahren erlaubt es, iterativ mit einer festen, seriell arbeitenden Hardwareschaltung praktisch alle elementaren transzendenten Funktionen zu erzeugen. Dieses Verfahren ist deshalb bei Taschenrechnern sehr weit verbreitet. Eine CORDIC-Einheit kann mit drei Addier-Subtrahier-Elementen und einer kleinen Lookup-Tabelle erzeugt werden, das Abrollen der Iterationsschleifen für eine parallele Implementierung führt zu einem Hardwareaufwand, der 2-3 parallelen Dividierern nach der Digit-Recurrence-Methode entspricht.

Logarithmus

Für die Berechnung des Logarithmus wird nun ein Konvergenzverfahren beschrieben, das auf der multiplikativen Normalisierung beruht. Es wird sich zeigen, dass dieses Verfahren mit weni-

ger Hardwareressourcen als eine Implementierung mit der CORDIC-Methode umgesetzt werden kann. Folgende Formel beschreibt das Iterationsschema zur Berechnung von $q = ln(x)$:

$$\begin{aligned} x^{(0)} &= x \\ y^{(0)} &= 0 \\ x^{(i+1)} &= x^{(i)} \cdot c^{(i)} = x^{(i)} \cdot \left(1 + d_i 2^{-i}\right) \quad d_i \in \{-1, 0, 1\} \\ y^{(i+1)} &= y^{(i)} - ln(c^{(i)}) = y^{(i)} - ln\left(1 + d_i 2^{-i}\right). \end{aligned} \tag{4.56}$$

Die Werte $c^{(i)} = ln\left(1 + d_i 2^{-i}\right)$ werden dabei aus einer Tabelle gelesen. Die Auswahl für d_i geschieht so, dass $x^{(k)}$ gegen 1 konvergiert. Dann konvergiert $y^{(k)}$ gegen $ln(x)$, was sich aus folgenden Gleichungen einsehen lässt:

$$\begin{aligned} x^{(k)} &= x \prod c^{(i)} \approx 1 \Rightarrow \prod c^{(i)} \approx \frac{1}{x} \\ y^{(k)} &= -\sum ln(c^{(i)}) = -ln\left(\prod c^{(i)}\right) \approx ln(x). \end{aligned} \tag{4.57}$$

Die Methode konvergiert für folgenden Wertebereich für x:

$$\frac{1}{\prod\left(1 + 2^{-i}\right)} \leq x \leq \frac{1}{\prod\left(1 - 2^{-i}\right)} \quad \Leftrightarrow \quad 0.21 \leq x \leq 3.45. \tag{4.58}$$

Für Werte von x außerhalb dieses Intervalls kann mit $x = 2^q\, s$ – wobei q so gewählt wird, dass $1 \leq s < 2$ gilt – folgende Beziehung zur Berechnung des Logarithmus herangezogen werden:

$$ln(x) = ln\left(2^q\, s\right) = q\, ln(2) + ln(s). \tag{4.59}$$

Bei dieser Methode werden k Iterationen benötigt, um ein Ergebnis $y^{(k)}$ mit k Stellen Genauigkeit zu erzeugen. Der Grund dafür ist, dass für große i die Näherung $ln(1 \pm 2^{-i}) \approx \pm 2^{-i}$ gilt. Dies hat zur Folge, dass die k-te Iteration den Wert von y um höchstens *ulp* ändert.

Pro Iteration müssen nur $x^{(i)}$ und $y^{(i)}$ durch Addition/Subtraktion eines Wertes aus einer Look-Up-Tabelle verändert werden. Bei CORDIC wären pro Iteration drei Additions/Subtraktions-Operationen notwendig. Außerdem müssten zur Sicherstellung der Konvergenz bei CORDIC manche Iterationen wiederholt werden, weshalb für eine Genauigkeit von k Bits mehr als k Iterationen erforderlich wären. Diese Faktoren führen zu einem klaren Vorteil der vorgestellten Methode gegenüber einer CORDIC-Realisierung im Fall des Logarithmus.

4.5.3 Iterative Näherungsverfahren

Die Anwendung iterativer Näherungsverfahren ist eine Standardmethode zur Berechnung transzendenter Funktionen in Software, wenn nur die elementaren Operationen zur Verfügung stehen. Aufgrund der schnellen Konvergenz (oft quadratisch) lässt sich das Funktionsergebnis durch wenige Iterationen mit gewünschter Genauigkeit bestimmen. Seit wenigen Jahren basieren auch die Rechenwerke einiger Mikroprozessoren zur Berechnung elementarer Funktionen wie Kehrwert und Quadratwurzel auf iterativen Näherungsverfahren (siehe z.B. [86]). In diesem Abschnitt werden einige gebräuchliche Verfahren vorgestellt und hinsichtlich ihrer Eignung für eine FPGA-Implementierung untersucht.

Newton-Raphson-Methode

Diese Methode basiert auf dem Newton-Verfahren zur Berechung der Nullstellen einer Funktion $f(z) = 0$. Die Iterationsgleichung lautet:

$$z(i) = z(i-1) - \frac{f(z(i-1)}{f(z(i-1))'}, \tag{4.60}$$

wobei $z(0)$ ein Startwert ist, welcher abhängig von der Funktion zu wählen ist. Die Newton-Raphson-Methode wird vor allem zur Berechnung von $z = 1/x$ verwendet, wobei z in diesem Fall über die Nullstelle von $f(z) = 1/z - x$ gefunden werden kann. Dann wird die Iterationsvorschrift 4.60 zu:

$$z(i) = z(i-1)\,(2 - z(i-1)\,x)\,. \tag{4.61}$$

Das Verfahren konvergiert quadratisch. Pro Iteration müssen zwei Multiplikationen und eine Addition ausgeführt werden. Der Rechenaufwand ist also vergleichbar mit der bereits vorgestellten Methode der Division durch wiederholte Multiplikation und damit gilt auch hier, dass eine Implementierung der Division ausschließlich über dieses Verfahren nicht sinnvoll ist. Eine Kombination mit einem LUT-basierten Verfahren zur Bestimmung eines Startwertes kann jedoch eine Alternative zur Digit-Recurrence-Methode sein. Zur Implementierung der Quadratwurzel nach der Newton-Raphson-Methode siehe z.B. [100]. Hier sind pro Iteration drei Multiplikationen erforderlich.

Spezielle Verfahren mit multiplikativer Konvergenz

Für die Division/Kehrwertberechnung wurde bereits das Verfahren der wiederholten Multiplikation (Goldschmidt-Verfahren) vorgestellt. Ein ähnliches Verfahren existiert für die Quadratwurzel und inverse Quadratwurzel von $x \in [1,2)$ (siehe z.B. [28]). Es soll hier nun auch für diese Methode untersucht werden, ob sie für FPGA-Anwendungen in Frage kommt. Das Prinzip ist hier, eine Sequenz von Faktoren t_i zu finden, sodass für eine Iterationsvariable $s(m)$ gilt:

$$\begin{aligned} s(m) &= x \cdot t_0^2 \cdot t_1^2 \cdots t_{m-1}^2 \rightarrow 1 \; (m \rightarrow \infty) \\ &\Rightarrow t_0^2 \cdot t_1^2 \cdots t_{m-1}^2 \rightarrow 1/x \; (m \rightarrow \infty). \end{aligned} \tag{4.62}$$

Dann folgt:

$$\begin{aligned} q(m) &= \prod\nolimits_{i=0..m-1} t_i \rightarrow \tfrac{1}{\sqrt{x}} \; (m \rightarrow \infty) \\ z(m) &= x \prod\nolimits_{i=0..m-1} t_i \rightarrow \sqrt{x} \; (m \rightarrow \infty). \end{aligned} \tag{4.63}$$

Die Wahl von t_i mit Startwert $S_{1/sqrt}$ zu:

$$\begin{aligned} t_0 &= S_{1/sqrt} \\ t_i &= \left(1 + \tfrac{1-s(m)}{2}\right) \end{aligned} \tag{4.64}$$

führt zu quadratischer Konvergenz, was sich aus folgender Beziehung ergibt:

$$\begin{aligned} \varepsilon_i &= 1 - s(i) \\ \varepsilon_{i+1} &= 1 - s(i+1) = \tfrac{3}{4}\varepsilon_i^2 + \tfrac{1}{4}\varepsilon_i^3. \end{aligned} \tag{4.65}$$

Zur Berechung von $z = \sqrt{x}$ werden $s(m)$ und $z(m)$ iterativ berechnet ($m = 1, \ldots, M$), für $z = 1/\sqrt{x}$ werden dagegen $s(m)$ und $q(m)$ iteriert. Der Ausgangswert $t_0 = S_{1/sqrt}$ kann als 1 angesetzt werden. Alternativ können zur Beschleunigung des Verfahrens t_0 und t_0^2 durch ein Table-Look-Up für $1/\sqrt{x}$ und dessen Quadrat mit geringer Präzision bestimmt werden ([28]). In [91] wird demonstriert, wie durch Anwendung einer Minimax-Approximation zweiten Grades und einer einzigen Iteration mit quadratischer Konvergenz die Operationen Kehrwertbildung, Quadratwurzel und Kehrwert der Quadratwurzel mit Double-Precision berechnet werden können. In diesem Fall braucht für die Quadratwurzel nur $z(2)$ ausgewertet zu werden, was durch drei Multiplikationen und zwei Additionen geschehen kann:

$$\begin{aligned} z(1) &= x\, S_{1/sqrt} \\ z(2) &= z(1) + z(1) \cdot \tfrac{1 - z(1)\, S_{1/sqrt}}{2} \end{aligned} \tag{4.66}$$

Für eine FPGA-Implementierung mit einer Genauigkeit geringer als Single-Precision würde eine einfache lineare Approximation für $S_{1/sqrt}$ mit einer einzigen nachfolgenden Iteration nach Gleichung 4.66 genügen, wofür einige wenige Block-RAMs, zwei Addierer und mehrere Block-Multiplizierer erforderlich sind. Gegenüber einer parallelen Implementierung nach der Digit-Recurrence-Methode ist eine solche Schaltung nur sinnvoll, wenn im FPGA abzüglich der für andere Operationen verwendeten Block-Multiplizierer noch mehrere solche Elemente ungenutzt blieben.

4.5.4 Kombination verschiedener Verfahren

Die im letzten Abschnitt beschriebene Kombination eines Table-Look-Up-Verfahrens zur Generierung eines Startwerts für ein iteratives Näherungsverfahren mit einer geringen Zahl nachfolgender Iterationen lässt sich genauso für die Division nach dem Goldschmidt-Verfahren und für die Verfahren nach der Newton-Raphson-Methode anwenden. Besonders attraktiv ist dabei die Variante, nur eine einzige Iteration durchzuführen. Der Startwert muss dann eine Genauigkeit von etwa der Hälfte der Ergebnisstellen erreichen. Soll z.B. eine Ergebnisgenauigkeit von Single-Precision erreicht werden, sollte der Startwert dann mit etwa 12 Bit Genauigkeit ermittelt werden, was durch eine Bipartite-Table mit $O(10)$ KBit erreicht werden kann. Gegenüber einer Digit-Recurrence-Implementierung werden hauptsächlich Block-RAM-Elemente und Block-Multiplizierer benötigt, dagegen nur eine geringe Zahl von allgemeinen Logikressourcen für die Addierer. Der Nachteil einer solchen Methode ist die Komplexität der Implementierung aufgrund der Verbindung unterschiedlicher Berechnungsmethoden. Insbesondere die Look-Up-Tables für die Bestimmung des Startwertes müssen für jede gewünschte Ergebnisgenauigkeit neu bestimmt werden und unterscheiden sich stark in der Größe. Letzteres führt zum Problem der effizienten Abbildung dieser Tabellen auf die vorgegebenen Block-RAMs des FPGAs. Schließlich entsteht auch das nicht unerhebliche Problem der Verifizierung der Genauigkeit für alle Implementierungen eines in der Rechengenauigkeit parametrisierten Operatormoduls. Spielen die letztgenannten Faktoren jedoch eine untergeordnete Rolle, wenn etwa die erforderliche Operatorgenauigkeit einen a priori festgelegten Wert hat, kann durch ein solches Kombinationsverfahren der Res-

sourcenverbrauch von den knappen allgemeinen Logikressourcen hin zu den ansonsten eventuell brach liegenden Block-Multiplizierern und Block-RAMs verschoben werden.

4.6 Stand der Forschung zu Implementierungen auf FPGAs

Seit die Technik rekonfigurierbarer Logik mit den ersten leistungsfähigen FPGAs Ende der 1980er-Jahre einem breiteren Anwenderkreis bekannt wurde, wird versucht, diesen Ansatz für wissenschaftliches Rechnen zu verwenden. Aufgrund der im Vergleich zu heute geringen Menge von Logikressourcen früher FPGAs beschränkte sich die Anwendung anfangs hauptsächlich auf Aufgaben des **D**igital-**S**ignal-**P**rocessing (DSP) wie Bild- und Ton-Verarbeitung, wo häufig eine geringe Genauigkeit der Arithmetik genügt. Frühe Versuche, FPGAs für Gleitkommaarithmetik nutzbar zu machen, wurden von Fagin und Renard [30], Shirazi et al. [103] und Louca et al. [69] veröffentlicht. Zu dieser Zeit war es nicht möglich, auch nur einen einzigen Single-Precision-Multiplizierer auf einem FPGA zu implementieren, weshalb die Rechengenauigkeit eingeschränkt (in [103]), die Berechnung in mehrere Iterationen zerlegt (in [69]) oder in mehreren FPGAs implementiert wurde (in [30]). Kompliziertere Operatoren wie Dividierer wurden wie in [103] durch Table-Look-Up aus synchronen Speicherelementen umgesetzt. Für Anwendungen mit geringerem Datendurchsatz wurden serielle Digit-Recurrence-Algorithmen vorgeschlagen. Louie und Ercegovac zeigen in [68] und [67] solche Implementierungen für die Division (siehe auch [66]) und Quadratwurzel. Wenige Jahre später mit der Verfügbarkeit weit größerer FPGAs war es bereits möglich, vollständig parallele Single-Precision-Operatoren zu implementieren. Ligon et al. demonstrierten in [65] die Implementierung von Addierern und Multiplizierern. Li und Chu publizierten in [57] ihre Umsetzung von Quadratwurzeloperatoren (siehe auch [58]).

Die heute verfügbaren FPGAs ermöglichen es bereits, eine große Zahl von Gleitkommaoperationen auf einem Chip zu implementieren. Entsprechend schnell wächst derzeit die Anzahl von Veröffentlichungen zu diesem Thema. Beispiele für neuere Implementierungen parametrisierbarer Gleitkommamodule sind [49], [8], [45] und [59]. Die Veröffentlichungen [49], [8] und [45] beschränken sich auf die Diskussion von Gleitkommaaddierern und -multiplizierern. In [59] vergleichen Liang et al. verschiedene Verfahren, um die Latenzzeit von Gleitkommaaddierern auf Kosten des Ressourcenverbrauchs zu reduzieren. Roesler und Nelson diskutieren in [97] die Anwendung der Block-Multiplizierer und Shift-Register, welche ab der Generation der Virtex-II-FPGAs verfügbar sind, zur Optimierung von Gleitkommaaddierern, -multiplizierern und -dividierern. Für die Dividierer wurde ein Newton-Raphson-Iterationsschema mit vorangehender Bestimmung eines Startwertes durch eine Look-Up-Table gewählt. In [13] untersuchen Beuchat und Tisserand Verfahren zur Optimierung von Multiplizierern und verschiedenen SRT-Dividierern unter Verwendung der Virtex-II-Block-Multiplizierer, einschließlich dem Skalierungsverhalten der beschriebenen Methoden mit der Breite der Operanden.

Die Entwicklung von FPGA-Designs für Gleitkommaoperatoren durch den Autor dieser Arbeit verlief parallel zur Forschung anderer Gruppen in diesem Bereich ([49], [8], [45], [59], und [97], wie oben bereits diskutiert), da die im Verlaufe der Arbeit neu verfügbar gewordenen FPGAs neue Implementierungsmöglichkeiten eröffneten. In [64] veröffentlichte der Autor die Implementierung einer Bibliothek von vollständig parallelen und parametrisierbaren Gleit-

kommaoperatoren, welche Addition, Multiplikation, Division und Quadratwurzel einschließt, und demonstrierte die Leistungsfähigkeit anhand der Implementierung eines Teiles des SPH-Formalismus.

Eine sehr aktuelle Darstellung der Performance von Gleitkommaarithmetik auf FPGAs im Vergleich zu CPUs wird in [120] gegeben.

Zur Problemstellung der Optimierung der Rechengenauigkeit einzelner Operatoren in komplexen Rechenwerken seien die Arbeiten von Gaffar et al. ([32] und [33]) erwähnt. Die in diesen Veröffentlichungen vorgestellten Optimierungsmethoden werden anhand einfacher Anwendungen im Bereich DSP, bei denen ausschließlich Additionen und Multiplikationen erforderlich sind, demonstriert.

Eine Implementierung eines Rechenwerks auf einem FPGA, basierend auf dem logarithmischen Zahlensystem wurde in [76] publiziert.

Im Bereich der astrophysikalischen Simulationsalgorithmen gab es 1995 von Cook et al. [18] eine Veröffentlichung über die Implementierung der Gravitationsberechnung auf einem FPGA-Prozessor bestehend aus acht FPGAs, die über einen Bus verbunden sind und zusammengenommen etwa 3–4 % der Logikressourcen des in dieser Arbeit verwendeten FPGAs haben. Es wurden Gleitkommazahlen in Single-Precision verarbeitet, wobei die Arithmetik durch serielle Addierer und Multiplizierer sowie eine lineare Interpolation für die Berechnung von $x^{-3/2}$ implementiert wurde. Mit dieser Architektur wurde eine Rechenleistung von 40 MFlops erreicht.

Eine Implementierung der Gravitationskraftberechnung unter Verwendung der in Abschnitt 2.4.2 vorgestellten PROGRAPE-Plattform wurde in [40] veröffentlicht. Die PROGRAPE-1-Plattform verfügt über etwa 30 % der Logikressourcen des MPRACE-Boards. Damit wurde unter Verwendung eines logarithmischen 12-Bit-Zahlenformats eine Rechenleistung von 0.96 GFlops erreicht, wobei für die Formel zur Gravitationskraftberechnung (nach Gleichung 2.3 in Abschnitt 2.2.1) 30 Operationen gezählt wurden.

Ho et al. verwendeten in [46] die Gravitationsberechnung als Testproblem für ihre Entwicklung der Symmetric Table Addition Method zur Näherung von Funktionen. Diese Approximationsmethode wurde für die Bestimmung von $x^{-3/2}$ eingesetzt. Auf ihrer Plattform basierend auf einem Virtex-FPGA von Xilinx wurde unter Aufwendung von 10260 Slices für die Berechnung mit Single-Precision (vgl.: der in dieser Arbeit verwendete FPGA enthält 14336 Slices, siehe Abschnitt 3.1.2 bzw. 3.3) wurde eine Peak-Performance von 1.5 GFlops angegeben, wobei auch hier 30 Operationen für eine Berechnung einer Paarwechselwirkung angesetzt wurden.

In [54] wurden die Vorarbeiten für eine SPH-Implementierung auf einem FPGA-Prozessor veröffentlicht. Mit einer Prototypimplementierung der Formel für die SPH-Dichteberechnung (entsprechend Gleichung 2.24 in Abschnitt 2.3.2) auf einem System bestehend aus 16 FPGAs (mit zusammen etwa 10 % der Logikressourcen des für diese Arbeit verwendeten FPGAs) wurde unter Verwendung eines Gleitkommaformats mit 20 Mantissenbits eine Peak-Performance von 208 MFlops erreicht. Die Division und Quadratwurzel wurden über eine lineare Interpolation berechnet. Das in dieser Dissertation beschriebenen Projekt ist die Fortführung des in [54] vorgestellten Ansatzes.

Kapitel 5

Analyse der Astrophysikalischen Algorithmen bezüglich der Anforderungen an eine Beschleunigerplattform

In diesem Kapitel werden die diese Arbeit betreffenden astrophysikalischen Algorithmen nach notwendigen Kriterien für eine Beschleunigung durch eine Koprozessorplattform untersucht. Dies sind insbesondere die Rechenleistung und die Kommunikationsbandbreite. Weiter wird analysiert, welche Rechengenauigkeiten gegeben sein müssen, damit ein korrektes numerisches Verhalten der Algorithmen gewährleistet ist. Daraus wird die grundlegende Beschleunigerarchitektur abgeleitet.

Es wird von einem System bestehend aus einem Rechnerknoten und einem Beschleuniger für die Gravitationsberechnung ausgegangen, wie es in Abschnitt 2.4.2 in Abbildung 2.3 gezeigt wurde. Zu diesem System soll eine zweite Spezialrechnerarchitektur hinzugefügt werden, um die rechenzeitkritischen Teile des Algorithmus zu beschleunigen. Es wird von einer modernen Implementierung eines astrophysikalischen Simulationscodes des Max-Planck-Instituts für Astrophysik, Heidelberg (*WINE*, siehe [126]) ausgegangen, für den detaillierte Analysen angefertigt werden konnten[1]. Die Gravitationswechselwirkung wird bei diesem Code über einen baumbasierten Algorithmus berechnet, mit der Option, einen GRAPE-Rechenbeschleuniger anzuwenden. Die SPH-Berechnungen sind in der Formulierung umgesetzt, wie sie in Abschnitt 2.3.2 gegeben wurde. Da viele Codes anderer Arbeitsgruppen eine grundsätzlich ähnliche Struktur aufweisen, sind die Ergebnisse weitgehend übertragbar.

5.1 Erforderliche Rechenleistung

In diesem Abschnitt wird die Skalierung des Rechenaufwands mit der simulierten Teilchenzahl, wie sie bereits in den Grundlagen erwähnt wurde, präzisiert. Anhand einer Analyse der Performance für eine Testrechnung werden konkrete Zahlen zum Rechenaufwand gegeben. Hierbei

[1] Die Messungen an den Simulationscodes wurden von T. Naab, O. Kessel-Deynet und M. Wetzstein am Max-Planck-Institut für Astronomie in Heidelberg ausgeführt.

ist insbesondere von Interesse, welchen Anteil an der Gesamtrechenzeit die Berechnung der SPH-Formeln verursacht. Denn dieses Verhältnis ist entscheidend für den maximal erreichbaren Speedup.

Skalierung des Rechenaufwandes

Wie erwähnt basiert der untersuchte Code auf einem Baumverfahren für die Behandlung der Gravitation. Es wird ein Bottom-Up-Binärbaum verwendet, wie er unter der Bezeichnung *Press Tree* in Abschnitt 2.2.2 beschrieben wurde. Ist N_{grav} die Anzahl der gravitativ wechselwirkenden Teilchen, skaliert die Rechenzeit für die Gravitationssimulation mit:

$$t_{grav,Host} = \alpha \, N_{grav} \log_2 N_{grav}, \tag{5.1}$$

mit einer Poportionalitätskonstante α, welche durch die Rechenleistung der Simulationsplattform bestimmt ist. Die Zahl N_{grav} setzt sich zusammen aus der Anzahl an SPH-Teilchen N_{SPH} und der Anzahl rein gravitativ wechselwirkender Teilchen N_{Nbody}. Es gilt also: $N_{grav} = N_{Nbody} + N_{SPH}$. Typischerweise liegt N_{SPH} in der gleichen Größenordnung wie N_{Nbody}.

Die Gravitationssimulation dominiert den Gesamtrechenaufwand, wenn dafür kein Rechenbeschleuniger eingesetzt wird. Der Einsatz eines Rechenbeschleunigers bewirkt folgende Veränderung der Rechenzeit:

$$t_{grav,GRAPE} = \beta \, N_{grav} \log_2 N_{grav} + \gamma N_{grav}, \tag{5.2}$$

mit $\beta \ll \alpha$ und einem durch die Kommunikationsbandbreite mit dem Rechenbeschleuniger bestimmten Faktor γ.

Für SPH ergibt sich folgende Skalierung, wenn N_{neigh} die durchschnittliche Anzahl von Nachbarn ist, über die in den SPH-Formeln summiert werden muss:

$$t_{SPH} = \delta \, N_{SPH,active} \, N_{neigh}. \tag{5.3}$$

Hier wurde anstatt N_{SPH} die Zahl $N_{SPH,active}$ verwendet, was ausdrückt, dass bei Variable-Timestep-Verfahren in manchen Zeitschritten nur eine Untermenge der SPH-Teilchen neu berechnet wird.

Analyse eines Simulationscodes

Während die Peak-Performance moderner Prozessoren im GFlops-Bereich liegt, wird bei wissenschaftlichen Simulationsanwendungen in der Regel nur ein Bruchteil dieser Rechenleistung auch tatsächlich erzielt. Um die für die SPH-Formeln auf einem Standardprozessor erreichbare Rechenleistung abzuschätzen, wurde die innerste Schleife der Druckkraftberechnung nach Gleichung 2.32 separat implementiert und die Performance dieses Codefragments gemessen. Es ergab sich auf einem Pentium-III-System mit 1266 MHz abhängig von der Reihenfolge der Teilchen, für welche die SPH-Formeln ausgewertet wurden, eine Rechenleistung von etwa 180–320 MFlops. In echten astrophysikalischen Codes kann die Performance noch weit darunter liegen, da

sich durch verteilte Datenstrukturen sehr komplexe Zugriffsmuster auf den Hauptspeicher ergeben können. Es wird im Folgenden anhand des *WINE*-Codes diskutiert, welcher Speedup für das Gesamtsystem erreicht werden kann, wenn die laufzeitkritischen Formeln des SPH-Algorithmus beschleunigt werden.

Durch Laufzeitmessungen wurde ermittelt, wie sich die Rechenzeit auf die Behandlung von Gravitation, SPH und den Rest aufteilt. Für den Fall, dass alles auf einem einzigen Rechnerknoten gerechnet wird, ergab sich, dass für Teilchenzahlen in der Größenordnung 10^5 etwa 70 % der Rechenzeit für die Ermittlung der Gravitationskräfte aufgewendet wird, etwa 25 % der Zeit für SPH benötigt wird und alles übrige nur wenige Prozent des Rechenaufwandes verursacht. Wird die Gravitationsrechnung durch eine GRAPE-5-Plattform beschleunigt, reduziert sich die Rechenzeit dafür auf etwa 14 %, wobei die Kommunikation mit GRAPE-5 die Rechengeschwindigkeit limitiert. Bei Anwendung der neuen 64-Bit-PCI-basierten GRAPE-6-Plattform kann eine weitere deutliche Reduktion dieses Rechenzeitanteils erwartet werden.

Durch eine genauere Analyse der Rechenzeit für den SPH-Teil des Codes zeigte sich, dass etwa 70 % der Rechenzeit auf die inneren Schleifen fällt[2] und etwa 30 % für die Nachbarschaftssuche[3] verwendet wird, wobei die Nachbarschaftslisten für den zweiten Schritt des SPH-Algorithmus erneut berechnet werden. Für die inneren SPH-Schleifen wurde eine Rechenleistung von etwa 70 MFlops erzielt, wobei für die Messung ein System mit Pentium-III-Prozessor, 500 MHz, verwendet wurde. Für ein aktuelles System kann dagegen von einer Rechenleistung von etwa 200 MFlops ausgegangen werden.

Um ein hohes Beschleunigungspotential für die Gesamtapplikation zu ermöglichen, muss der Anteil der Rechenzeit, der außerhalb der SPH-Formeln aufgewendet wird, möglichst niedrig werden. Anhand dieser Bedingung lassen sich folgende Designkriterien für ein beschleunigtes System finden:

- Sehr viel an Rechenzeit kann eingespart werden, wenn die Nachbarschaftslisten für den ersten SPH-Schritt zwischengespeichert werden, so dass sie für Schritt 2 nicht mehr neu berechnet werden müssen. Der Speicherbedarf dafür ist jedoch $N_{SPH} \cdot N_{neigh}$ (für 10^5 Teilchen und durchschnittlich 50 Nachbarn sind das etwa 20 MByte). Die Zwischenspeicherung wurde bisher vermieden, um neben der Baumstruktur nicht noch andere dynamische Datenstrukturen verwalten zu müssen.

- Die Gravitationsberechnung sollte simultan mit der Erzeugung der Nachbarschaftslisten für SPH geschehen, damit sich die Rechenzeiten beider Teile nicht summieren.

- Die Generierung der Nachbarschaftslisten kann unter Einsparung von Rechenzeit "vergröbert" werden, beispielsweise durch gemeinsame Listen für mehrere aktive Teilchen. Die Extraktion der entgültigen Nachbarschaftslisten kann dann vom Rechenbeschleuniger übernommen werden (sofern dafür noch Ressourcen verfügbar sind).

[2]In diesen Schleifen werden die Berechnungen der SPH-Summen wie in der Formulierung nach Abschnitt 2.3.2 durchgeführt (siehe Gleichung 2.24, 2.32 und 2.41).

[3]Nur Teilchen aus einer lokalen Umgebung tragen zu den SPH-Summen bei, siehe Abschnitt 2.3.2. Die Nachbarschaftslisten werden durch eine lokale Suche in einer verketteten Liste bestimmt. Siehe dazu [126].

Werden diese Punkte berücksichtigt, erscheint eine Reduktion des Rechenaufwands aller Codeteile außerhalb der inneren SPH-Schleifen auf unter 10 % erreichbar. Es ergibt sich dann ein Beschleunigungspotential um einen Faktor $O(10)$. Um einen Speedup in dieser Größenordnung zu realisieren, muss die Beschleunigerplattform eine Dauerrechenleistung von mehr als 2 GFlops erreichen.

5.2 Erforderliche Kommunikationsbandbreite für einen Rechenbeschleuniger

In diesem Abschnitt wird die Kommunikation für eine Architektur aus Steuerungsrechner und Rechenbeschleuniger diskutiert. Es soll davon ausgegangen werden, dass nur die zeitkritischen inneren Schleifen des Simulationscodes auf dem Rechenbeschleuniger implementiert werden. Diese Schleifen summieren die Beiträge der Teilchen aus einer Nachbarschaftsliste zu den SPH-Näherungen für die Dichte ρ, die Divergenz und Rotation von $\vec{v}$, die Druckkraftkomponenten und die Energien. Dann müssen für den ersten Schritt des SPH-Algorithmus folgende Daten transferiert werden:

- Für die Dichteberechnung werden die Positionsvektoren $\vec{r}_i$, Glättungslängen h_i und Massen m_i benötigt.
- Für die Berechnung von $(\vec{\nabla} \cdot \vec{v})_i$ und $(\vec{\nabla} \times \vec{v})_i$ sind zusätzlich die Geschwindigkeitsvektoren $\vec{v}_i$ zum Beschleuniger zu übertragen.
- Die Endergebnisse ρ_i, $(\vec{\nabla} \cdot \vec{v})_i$ und $(\vec{\nabla} \times \vec{v})_i$ werden zurück zum Steuerungsrechner geschrieben.

Ist N_{SPH} wie oben eingeführt die Anzahl der SPH-Teilchen, die insgesamt in die Berechnungen involviert sind, werden für die erste SPH-Schleife also $N_{SPH} \cdot 8$ Gleitkommazahlen zum Rechenbeschleuniger übertragen. Für $N_{SPH,active}$ SPH-Teilchen, für welche die SPH-Summen ausgeführt werden, ergibt sich ein Rücktransfer von $N_{SPH,active} \cdot 5$ Gleitkommazahlen. Für die eigentlichen Berechnungen wird für jedes aktive Teilchen zusätzlich seine Nachbarschaftsliste benötigt. Bei einer mittleren Zahl an SPH-Nachbarn N_{neigh}, wofür $N_{neigh} \approx 50$ ein typischer Wert ist, ergibt sich ein zusätzliches Volumen von $N_{active} \cdot N_{neigh}$ Datenworten, das zum Rechenbeschleuniger übertragen werden muss.

Für den zweiten Schritt der SPH-Berechnungen ergibt sich folgender zusätzlicher Kommunikationsaufwand:

- Es müssen die auf dem Steuerungsrechner erzeugten Drücke P_i, Balsarakoeffizienten f_i und die Schallgeschwindigkeiten c_i zum Rechenbeschleuniger übertragen werden[4].

[4] Es wurde hier angenommen, dass die in Gleichung 2.32 auftretenden Konstanten α und β fest sind. Bei manchen Codes werden diese Parameter individuell für jedes Teilchen berechnet und müssen dann ebenfalls zum Rechenbeschleuniger übertragen werden.

- Als Ergebnisse werden die Druckkräfe und Entropieänderung durch die künstliche Viskosität zum Steuerungsrechner transferiert.

Hier ergibt sich also ein Datenaufkommen von $N_{SPH} \cdot 3$ Worten hin zum Steuerungsrechner und $N_{SPH,active} \cdot 4$ Datenworten zurück. Falls die Nachbarschaftsliste nicht auf der Beschleunigerplattform gespeichert werden kann, ist diese für den zweiten Schleifendurchlauf erneut zu übertragen. Falls auch die Teilchendaten aus Schritt 1 nicht gespeichert werden können, beispielsweise wenn der Speicher des Rechenbeschleunigers nur einen Teil der SPH-Teilchendaten aufnehmen kann, sind diese Daten ebenfalls ein zweites Mal zu übertragen.

Zusammengerechnet ergibt sich für die Berechnung der SPH-Formeln pro Zeitschritt ein minimales Datenaufkommen von:

$$\begin{aligned} Com_{Host \to Accel} &= N_{SPH} \cdot 11 + N_{active} \cdot N_{neigh} \\ Com_{Accel \to Host} &= N_{SPH} \cdot 9, \end{aligned} \tag{5.4}$$

wobei das Transfervolumen zwischen Steuerungsrechner und Beschleuniger $Com_{Host \to Accel}$ und in umgekehrter Richtung $Com_{Accel \to Host}$ genannt wurde. Falls auf dem Beschleuniger keinerlei Daten aus der ersten Schleifenberechnung gespeichert werden können, ergibt sich folgendes Volumen für den Upload zum Beschleuniger:

$$Com_{Host \to Accel} = N_{SPH} \cdot 19 + 2 \cdot N_{active} \cdot N_{neigh}. \tag{5.5}$$

Es soll nun eine grobe Abschätzung für die Mindestbandbreite $B_{com,min}$ gemacht werden, ab der eine Rechenleistung von über 2 GFlops erreicht werden kann. Für die SPH-Berechnungen werden pro aktivem Teilchen 100 Gleitkommaoperationen angesetzt. Es wird der Fall $N_{SPH} = N_{active}$ und $N_{neigh} = 50$ angenommen. Dann ergibt sich bei einer Übertragungsbandbreite B_{com} für den Datentransfer zwischen Steuerungsrechner und Beschleuniger folgende Zeit, wenn vom maximalen Datentransfervolumen aus Gleichung 5.5 ausgegangen wird:

$$\begin{aligned} t_{com} &= (N_{SPH} \cdot 28 + 2 \cdot N_{SPH} \cdot 50) \cdot 4\,Byte/B_{com} \\ &= \frac{128 \cdot N_{SPH} \cdot 4\,Byte}{B_{com}}. \end{aligned} \tag{5.6}$$

Demgegenüber wird folgende Zeit für eine SPH-Berechnung mit 2 GFlops benötigt:

$$t_{2GFlops} = \frac{N_{SPH} \cdot 50 \cdot 100}{2 \cdot 10^9 \frac{1}{sec}}. \tag{5.7}$$

Für $t_{com} = t_{2GFlops}$ ergibt sich dann:

$$B_{com,min} \approx 200\,MByte/sec. \tag{5.8}$$

Wird davon ausgegangen, dass die Nachbarschaftsliste und alle Teilchendaten auf dem Rechenbeschleuniger zwischengespeichert werden können, reduziert sich $B_{com,min}$ auf etwa 140 MByte/sec. Soll eine signifikante Beschleunigung der SPH-Berechnungen erreicht werden, muss die Kommunikationsbandbreite deutlich über diesen Werten liegen. Vor allem aber dürfen die Berechnungen selbst nicht viel länger als die Kommunikationszeit dauern, was insbesondere bedeutet, dass während der Übertragung der Daten gerechnet werden muss (zumindest beim Nachbarschaftslistentransfer, da dieser den größten Kommunikationsaufwand verursacht).

5.3 Erforderliche Rechengenauigkeit von SPH

Die benötigte Rechengenauigkeit ist der entscheidende Faktor dafür, ob eine Rechenbeschleunigung mit rekonfigurierbaren Rechensystemen möglich ist, und, falls dies der Fall ist, welcher Speedup erwartet werden kann. Dies lässt sich wie folgt begründen: Zur Beschleunigung müssen die Rechenwerke hoch parallelisiert implementiert werden, denn rekonfigurierbare Prozessoren erreichen konstruktionsbedingt wesentlich geringere Taktfrequenzen als Standardprozessoren. Der Ressourcenaufwand der einzelnen Operatoren hängt wiederum stark von der zu erzielenden Rechengenauigkeit ab. Je geringer also die Genauigkeitsvorgaben sind, desto mehr Operationen können parallel implementiert werden und desto höher ist deshalb das Potential für eine Beschleunigung.

Die Analyse der erforderlichen Rechengenauigkeit wurde in zwei Schritten durchgeführt. Erstens wurde ein künstlich in der Rechengenauigkeit eingeschränkter Simulationscode auf ein typisches Simulationsproblem angewandt, um die Genauigkeitsschwelle zu finden, ab der sich Abnormalitäten in der Simulation zeigen. Es wurde hier nicht jeder einzelne Rechenschritt in der Genauigkeit beschnitten. Nur die wichtigsten Zwischenergebnisse wurden verändert, damit der modifizierte Code ohne große Leistungseinbußen verwendet werden konnte. Nur so war es möglich, ein realistisches Simulationsproblem in vertretbarer Zeit zu untersuchen. Um das numerische Verhalten bei reduzierter Genauigkeit detaillierter zu analysieren, wurden in einem zweiten Schritt die SPH-Berechnungen mit künstlichen Eingangsdaten anhand einer Simulation eines Spezialrechenwerkes untersucht. Beide Schritte werden in den folgenden Unterabschnitten eingehend erläutert.

5.3.1 Testsimulationen mit künstlicher Reduktion der Genauigkeit

Es wurde der Kollaps einer Gaswolke simuliert. Es handelt sich um die gleiche Simulation, die auch im Zusammenhang mit den Tests eines neuen Simulationscodes in [126] beschrieben wird[5]. Gestartet wurde mit einer zu Beginn ruhenden sphärischen isothermen Gasverteilung mit Masse M, Radius R und folgendem Dichteprofil:

$$\rho(r) = \frac{M}{2\pi R^2}\,\frac{1}{r}, \tag{5.9}$$

die unter dem Einfluss der Eigengravitation kollabiert. Das System wurde mit 20000 SPH-Teilchen simuliert, wobei eine durchschnittliche Zahl an Nachbarteilchen von 50 eingestellt wurde.

Zur Beurteilung der Qualität der Simulationen wurden die Energieanteile des Gesamtsystems mit den Resultaten für die volle Rechengenauigkeit über den Zeitraum der Simulation miteinander verglichen. Die Energie wurde dabei aufgeschlüsselt in kinetische Energie, Gravitationsenergie und innere Energie. Dies ist das übliche Vorgehen, neue astrophysikalische Codes auf korrektes physikalisches Verhalten zu testen. Fehler in der Simulation führen zu Abweichungen

[5] Die Messungen wurden von Markus Wetzstein am Max-Planck-Institut für Astronomie, Heidelberg durchgeführt

in der Dynamik des Systems, welche am einfachsten über den zeitlichen Verlauf der Energiebestandteile nachzuweisen sind.

Es wurden vier Simulationen durchgeführt, in denen mit verkürzten Mantissenbreiten von 8, 12, 16 und 20 Bit gerechnet wurde. Abbildung 5.1 zeigt den Vergleich der Energien der 8-Bit-Rechnung mit der Single-Precision-Rechnung. Es zeigen sich bei der 8-Bit-Version zwar geringe aber dennoch signifikante Abweichungen. Besser zu sehen sind diese Abweichungen in den Plots für die relativen Fehler, wie sie in Abbildung 5.2 auf der linken Seite gegeben sind. Es tritt neben Fehlern im Prozentbereich auch eine deutliche Akkumulation von Rechenfehlern auf. Bereits bei 12-Bit Rechengenauigkeit ist keine signifikante Fehlerakkumulation mehr zu sehen, wovon man sich anhand der rechten Seite von Abbildung 5.2 überzeugt. Die Fehler in der Energie bewegen sich durchweg in einem akzeptablen Bereich ($O(0.1\%)$). Bei den Simulationen mit höheren Bitbreiten ergibt sich kaum noch eine Verbesserung des Verhaltens. Abbildung 5.3 zeigt die Ergebnisse im Fall von 20-Bit-Mantissen.

Zur Auswertung der Simulationsdaten ist zu erwähnen, dass der Simulationscode bei unterschiedlichen Rechengenauigkeiten die Daten für unterschiedliche Zeitschritte ausgab. Für die Berechnung der relativen Rechenfehler mussten die Daten deshalb auf die Zeitschritte der Referenzsimulation interpoliert werden. Es wurden drei Interpolationsverfahren angewandt, eine lineare Approximierung, eine Spline-Interpolation, und das Verfahren der kleinsten quadratischen Fehler. Durch den Vergleich zwischen den Verfahren ergab sich, dass die Rechenfehler bereits bei 12-Bit-Mantissen etwa im Rahmen der Interpolationsfehler lagen.

5.3.2 Simulation der Rechenwerke

Um einen detaillierten Einblick in das numerische Verhalten eines Rechenwerks für die SPH-Formeln zu erhalten, in dem wie bei der späteren FPGA-Implementierung jede Rechenoperation mit beschränkter Genauigkeit arbeitet, wurde ein Simulationssystem für Gleitkommaoperationen mit limitierter Genauigkeit entwickelt. Dazu wurden Simulatoren für die Operationen Addition, Multiplikation, Division und Quadratwurzel erzeugt, welche das Verhalten dieser Operatoren für beliebige Mantissenbreiten auf Bit-Ebene abbilden, einschließlich verschiedener Rundungsmodi[6]. Mit diesen Operatoren wurden Rechenwerke zusammengesetzt, wie sie als Pipeline in Hardware implementiert werden können. Als Eingangsdaten für die Genauigkeitssimulationen wurden fünf stark unterschiedliche künstlich erzeugte Teilchenverteilungen verwendet:

1. Gleichverteilung in alle Raumrichtungen.
2. Linear von einer Ecke eines Würfels in Richtung einer Diagonale des Würfels abfallend.
3. In X-Richtung Gaußverteilung, in Y- und Z-Richtung Gleichverteilung.
4. Um einen Punkt in X-, Y- und Z-Richtung gaußverteilt.
5. Stufenförmige Verteilung in X-, Y- und Z-Richtung.

[6]Es wurden die Modi *Round To Nearest Even*, *Round To Nearest Integer* und *Truncation* implementiert.

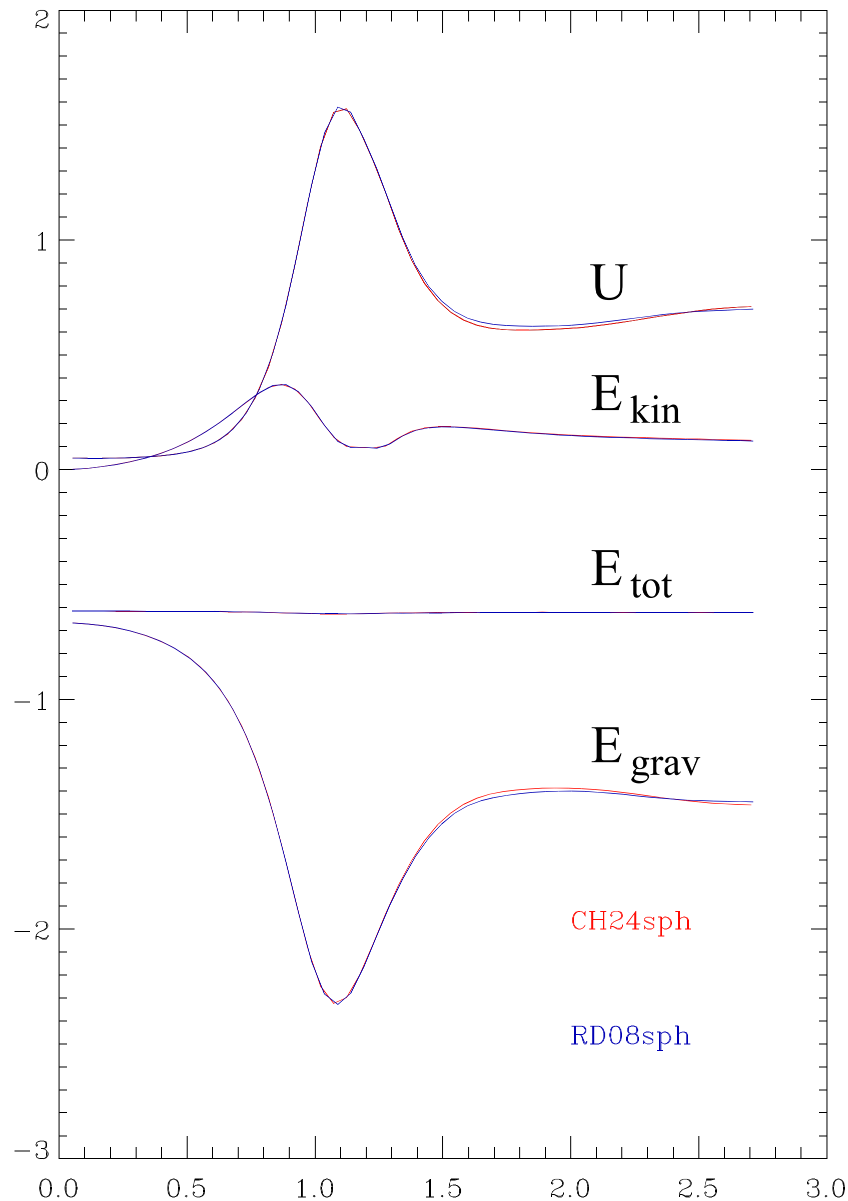


Abbildung 5.1: Vergleich der Energien bei SPH-Berechnung mit Rundung der Zwischenergebnisse auf 8-Bit-Mantissen (RD08sph) mit den Ergebnissen des Originalcodes (CH24sph, Single-Precision). Es ist die zeitliche Entwicklung der inneren Energie (U), kinetischen Energie (E_{kin}), Gravitationsenergie (E_{grav}) und der Gesamtenergie (E_{tot}) dargestellt. Es zeigen sich signifikante Abweichungen zwischen den Kurven für hohe und niedrige Rechengenauigkeit.

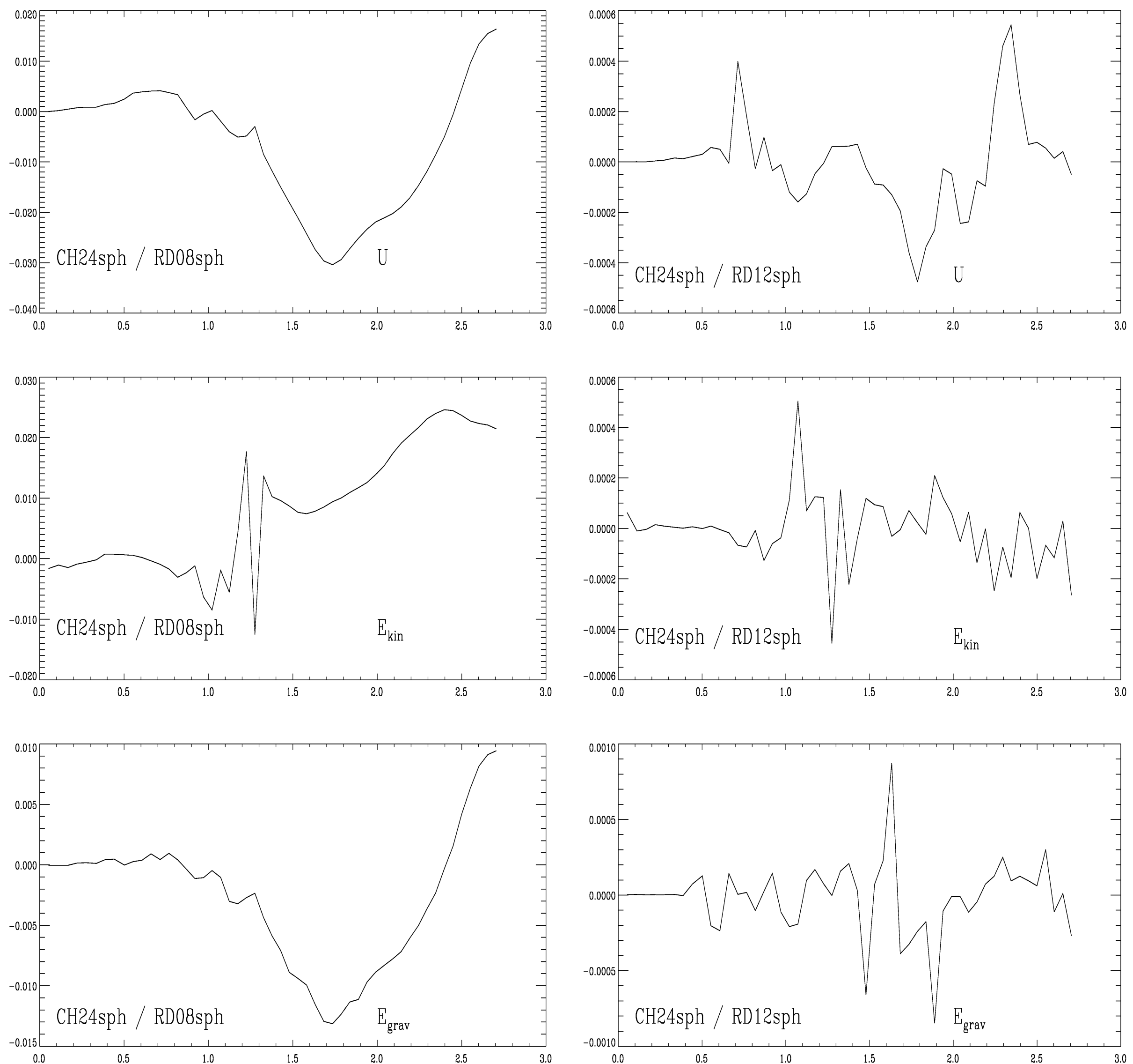


Abbildung 5.2: Zeitliche Entwicklung der relativen Fehler in der Berechnung der Energien bei Rundung der Zwischenergebnisse auf 8-Bit-Mantissen (links) und 12-Bit-Mantissen (rechts). Links ist deutlich die Akkumulation von Rechenfehlern zu sehen, während rechts die Fehler um Null pendeln. Man beachte auch die links und rechts unterschiedlichen Skalen.

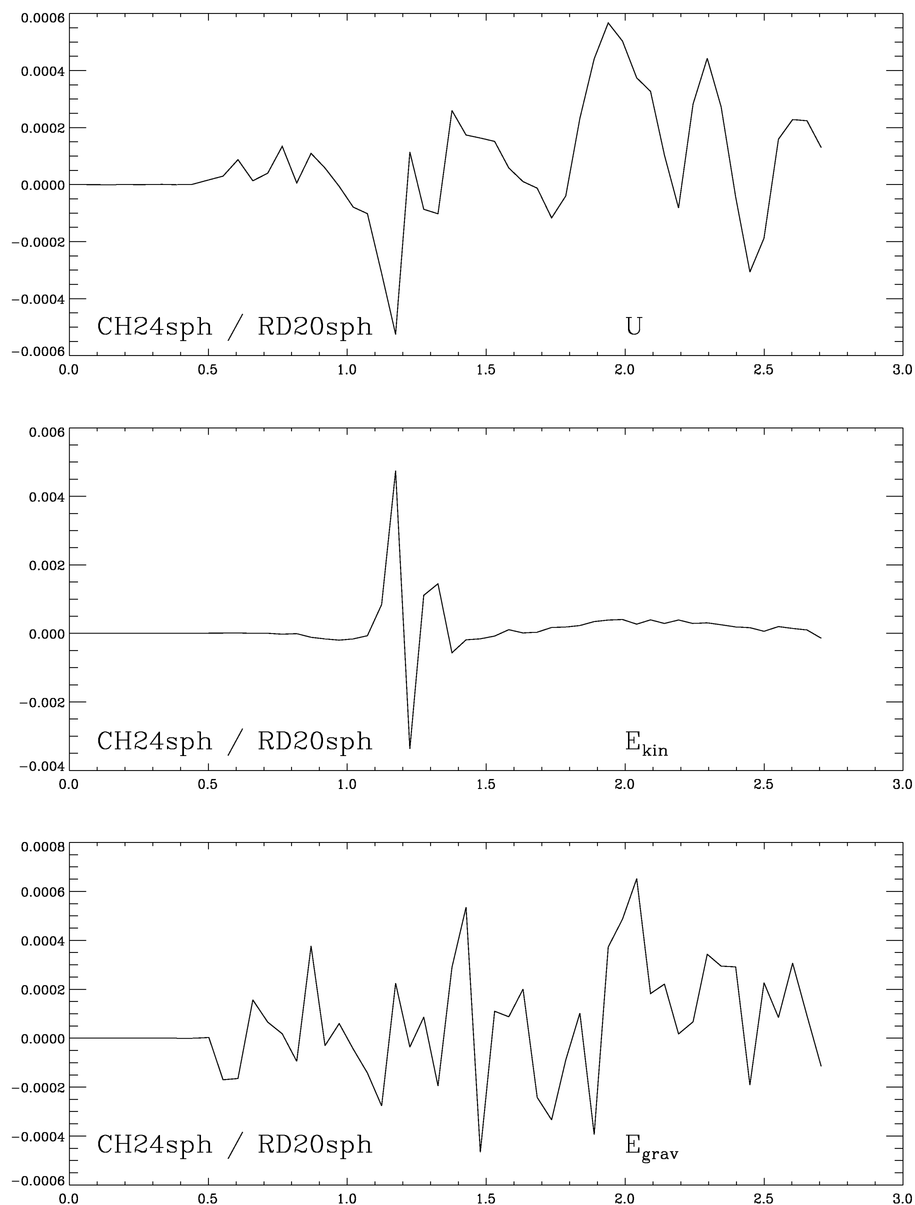


Abbildung 5.3: Relative Fehler in der Berechnung der Energien bei Rundung der Zwischenergebnisse auf 20-Bit-Mantissen.

Die Verteilungen wurden in einem $3 \times 3 \times 3$-Würfel erzeugt, wobei für die Teilwürfel eine Kantenlänge von $a = 4$ festgesetzt wurde[7]. Im Zentrum des innersten Würfels war ein Teilchen mit Glättungsparameter $h = 0.5$ vorgegeben. Nun wurden so lange Teilchen entsprechend einer Wahrscheinlichkeitsdichtefunktion hinzugefügt, bis das Zentralteilchen in seiner durch die Sphäre mit Radius $2h$ gegeber Nachbarschaft 50 SPH-Wechselwirkungspartner erhielt. In Abbildung 5.4 sind mit Ausnahme der Gleichverteilung die sich ergebenden Teilchenkonstellationen dargestellt. Die Simulationen wurden für alle Teilchen im innersten Würfel durchgeführt ($O(10^3)$ Teilchen). In allen Fällen wurde die Geschwindigkeitsverteilung durch gaußverteilte Komponenten der Geschwindigkeitsvektoren angesetzt.

Es wurden folgende Formeln entsprechend den Berechnungen von Schritt 1 und Schritt 2 der in Abschnitt 2.3.2 beschriebenen speziellen SPH-Formulierung berechnet:

$$\rho_i = \sum_{j=1}^{N} m_j W(|\vec{r_{ij}}|, h_{ij}) \tag{5.10}$$

$$\rho_i \, (\vec{\nabla} \cdot \vec{v})_i = -\sum_j m_j \, \vec{v_{ij}} \, \vec{\nabla}_i W(|\vec{r_{ij}}|, h_{ij}) \tag{5.11}$$

$$\rho_i \, (\vec{\nabla} \times \vec{v})_i = \sum_j m_j \, \vec{v_{ij}} \times \vec{\nabla}_i W(|\vec{r_{ij}}|, h_{ij}) \tag{5.12}$$

$$\frac{d\vec{v}_i}{dt} = -\sum_j m_j \left(\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2} + \Pi_{ij} \right) \vec{\nabla}_i W \left(|\vec{r}_{ij}|, h_{ij} \right) \tag{5.13}$$

$$\frac{du_{visc,i}}{dt} = \frac{1}{2} \sum_j m_j \Pi_{ij} \vec{v_{ij}} \cdot \vec{\nabla}_i W(|\vec{r}_{ij}|, h_{ij}). \tag{5.14}$$

Erwartungsgemäß zeigten die Berechnungen für die in drei Achsrichtungen gaußverteilte haufenförmige Teilchenkonstellation, welche die größten Dichtekontraste der getesteten Verteilungen hat, die größten Rechenfehler. Die mittleren relativen Fehler unterschieden sich jedoch selbst hier nicht um mehr als einen Faktor 2 von den Resultaten für die anderen Verteilungen. In Abbildung 5.5 sind die gemittelten Ergebnisse für verschiedene Mantissenbreiten dargestellt. Für alle Operatoren wurde hier mit dem Standardrundungsmodus *Round To Nearest Even* gerechnet.

Die wichtigste Beobachtung ist, dass sich für die Dichten mittlere Fehler im Bereich der Darstellungsgenauigkeit der Gleitkommazahlen ergeben. Bereits ab 12 Bit erhält man so eine Rechengenauigkeit in der Größenordnung 10^{-3}–10^{-4}, was vergleichbar ist mit den Fehlern in der Energie aus Abbildung 5.2 (rechte Seite). Für 16-Bit-Mantissen ist der mittlere relative Fehler bereits so niedrig, dass keinerlei signifikante Fehlerfortpflanzung auf die weiteren SPH-Berechnungen zu befürchten ist. Die Werte für die Divergenz- und Rotationsberechnung liegen etwa einen Faktor 50 über dem Fehler der Dichteberechnung. Etwa genauso stark weichen die Ergebnisse für die Berechnung der Kräfte und Entropieänderung ab. Verantwortlich dafür ist das hohe Maß an Auslöschung signifikanter Stellen bei der Akkumulation der Zwischenergebnisse. Der Faktor 50 läßt auf einen durchschnittlichen Verlust von fünf bis sechs führenden Mantissenbits schließen.

[7] Es interessieren in diesem Zusammenhang nur relative Größen, so dass der Betrag von a beliebig gewählt wurde.

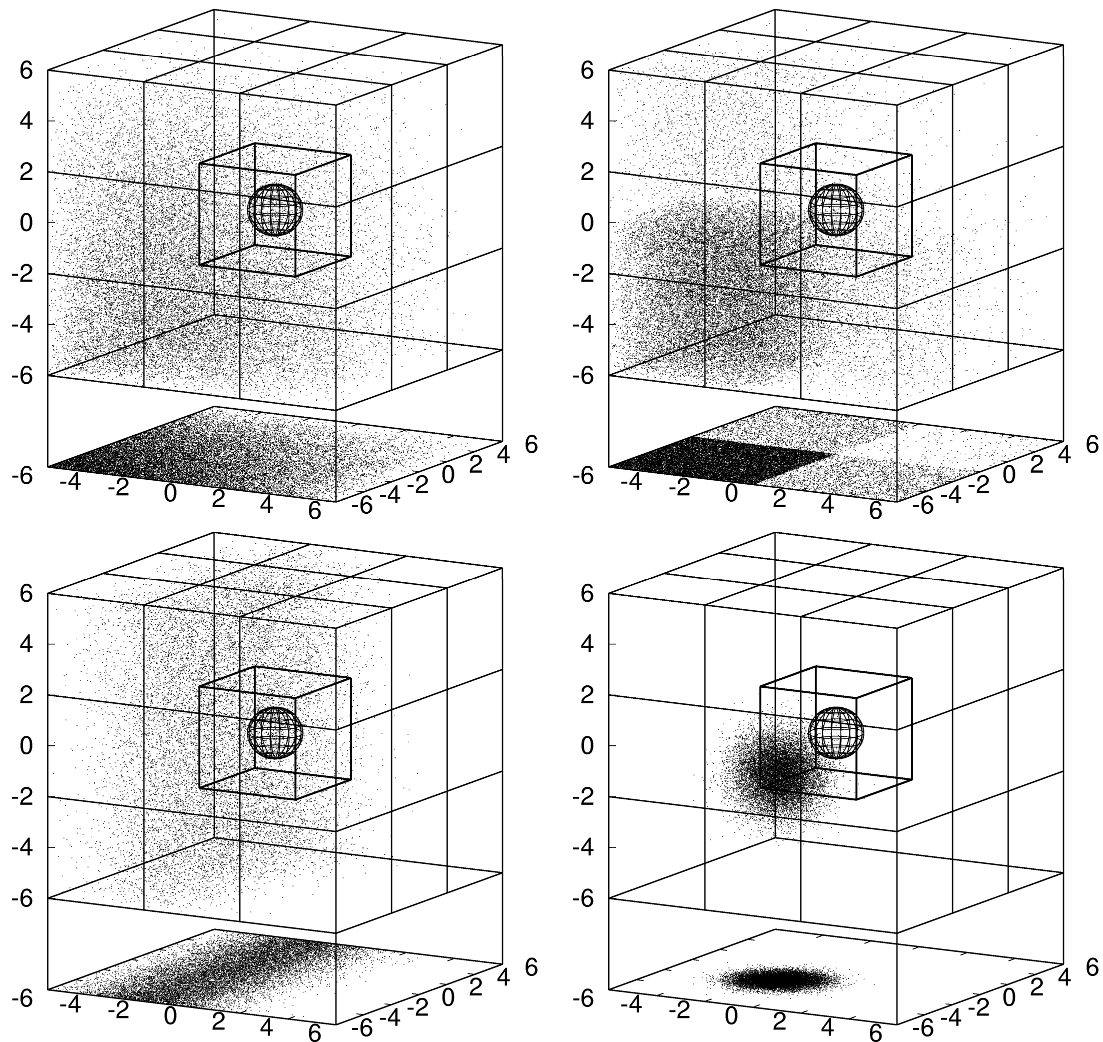


Abbildung 5.4: Simulierte Teilchenverteilungen zur Analyse der Randbedingungen an die Rechenwerke. Es ist die dreidimensionale Verteilung innerhalb eines 3x3x3-Kubus und deren Projektion auf die Grundfläche des Kubus zu sehen. Die Sphäre im Zentrum des inneren Kubus entspricht der Nachbarschaftsumgebung des Teilchens in der Mitte, welche etwa 50 Nachbarteilchen einschließt. Links oben ist eine linear in alle Richtungen abfallende Verteilung, rechts oben eine Stufenverteilung, links unten eine in eine Richtung normalverteilte und rechts unten eine in alle Achsrichtungen normalverteilte Teilchenkonstellation gezeigt.

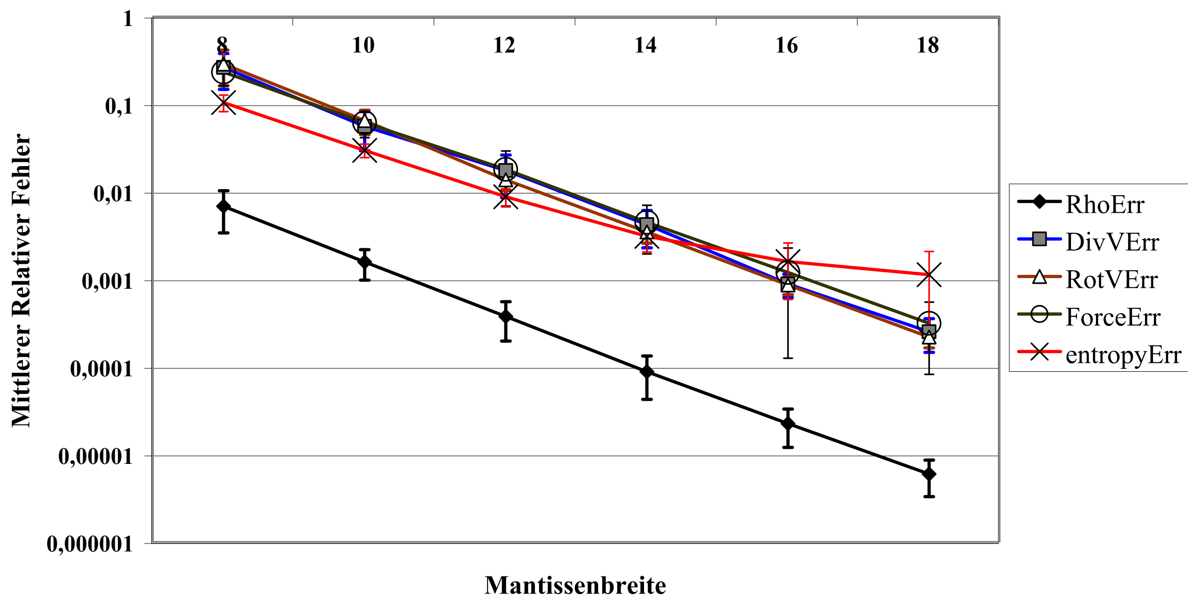


Abbildung 5.5: Abhängigkeit des mittleren relativen Fehlers der Berechnungen der SPH-Formeln für die Dichte (RhoErr), Divergenz (DivVErr), Rotation (RotVErr), Kraft (ForceErr) und Entropieänderung aufgrund der künstlichen Viskosität (EntropyErr) von der Genauigkeit der verwendeten Gleikommaoperationen.

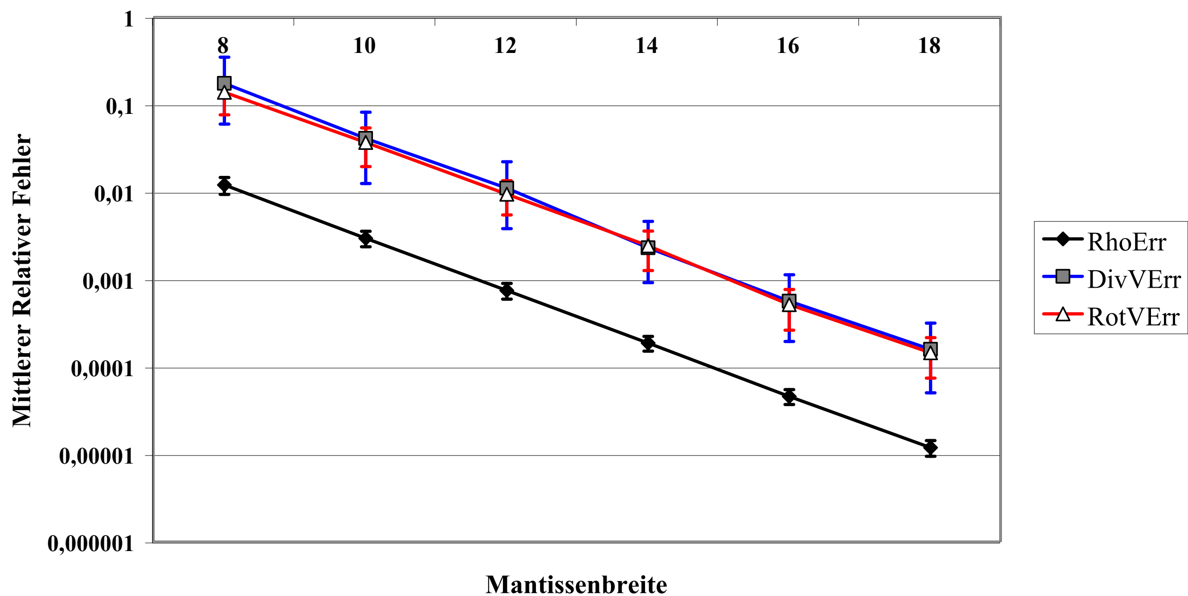


Abbildung 5.6: Abhängigkeit des mittleren relativen Fehlers der Berechnungen der SPH-Formeln für die Dichte (RhoErr), Divergenz (DivVErr) und Rotation (RotVErr), wenn nur die Eingangsgrößen und einige Zwischenergebnisse in der Genauigkeit eingeschränkt werden, wie es für die Messungen bei echten astrophysikalischen Testsimulationen gemacht wurde.

Um beurteilen zu können, ob die Ergebnisse zu den Simulationen der Rechenwerke vergleichbar mit den Testrechnungen aus Abschnitt 5.3.1 sind, wurden ebenfalls Simulationen durchgeführt, bei denen nur die Eingangsdaten und einige Zwischenergebnisse in der Genauigkeit reduziert wurden, wie es für die Testrechnungen geschehen ist. In Abbildung 5.6 sind die Ergebnisse dazu gezeigt. Es ergab sich, dass sich die relativen Fehler der Dichten nicht signifikant von den Fehlern für die detaillierte Rechenwerkssimulation unterschieden, die der übrigen Größen im Mittel ebenfalls nur wenig unterschiedlich waren bei etwas erhöhter Streuung der Genauigkeiten (siehe Fehlerbalken). Im einzelnen wichen die Genauigkeiten bis etwa auf einen Faktor 2–3 ab. Eine Erhöhung der Rechengenauigkeit um zwei Mantissenbits gegenüber den Testrechnungen sollte also zu einer vergleichbaren Genauigkeit der SPH-Simulation führen.

Es bleibt die Ungewissheit, ob das in Abschnitt 5.3.1 beschriebene astrophysikalische System repräsentativ für eine Vielzahl anderer astrophysikalischer Simulationen bezüglich des Verhaltens bei reduzierter Genauigkeit ist. Deshalb soll im Folgenden von einer Mantissenbreite von 16 Bit ausgegangen werden, um eine Prototypimplementierung mit einigen Sicherheitsreserven bezüglich der Numerik aufzubauen. Die mögliche Variation der Genauigkeitsanforderungen, abhängig von der Applikation, motiviert den Aufbau von in der Rechengenauigkeit skalierbaren FPGA-Designs. Diese Skalierbarkeit wird im Folgenden ein wichtiges Designkriterium für die Architektur der Rechenwerke sein.

Neben der Rechengenauigkeit der Operationen spielt auch das Rundungsverhalten und vor allem die Ausnahmebehandlung eine wichtige Rolle, weshalb diesbezüglich einige Designrichtlinien für die SPH-Rechenwerke gegeben werden sollen.

Runden Im SPH-Formalismus werden alle Größen durch die Akkumulation von Beiträgen aus Nachbarwechselbeziehungen gebildet. Um zu vermeiden, dass sich systematische Rechenfehler akkumulieren, sollten die Berechnungen korrekt gerundet werden. Bei Testrechnungen konnte kein signifikanter Unterschied in den Rechengenauigkeiten zwischen dem Standardrundungsmodus *Round To Nearest Even* und dem einfacheren Modus *Round To Nearest Integer* festgestellt werden. Es ist jedoch nicht auszuschließen, dass es Testszenarien gibt, für die sich doch Unterschiede zeigen. Deshalb sollte für die Implementierung der Rechenwerke stets erwogen werden, ob für einen Operator die Werte der Ergebnisstellen nach dem LSB nicht statistisch gleichverteilt sind, was vor allem bei Additionen auftreten kann. In Abschnitt 4.3.2 wurde bereits ein Beispiel für das Ungleichgewicht von Rundungsfehlern gegeben. In solchen Fällen sollte der Standard-Rundungsmodus *Round To Nearest Even* dem einfacheren Modus *Round To Nearest Integer* vorgezogen werden. Für Operationen wie Multiplikation, Division und Quadratwurzel, bei denen mit hoher Wahrscheinlichkeit nach dem Rundungsbit noch weitere Bits ungleich Null folgen, kann davon ausgegangen werden, dass der Modus *Round To Nearest Integer* zur Vermeidung der Fehlerakkumulation ausreichend genau ist.

Ausnahmebehandlung Es kann vorausgesetzt werden, dass die Berechnungen im Verlaufe einer Simulation nicht zu Überläufen von Zwischenergebnissen führen, da ansonsten die Software-Implementierung ebenfalls nicht korrekt arbeiten würde. Dagegen sind Underflow-Bedingungen, also die Unterschreitung der kleinsten darstellbaren Zahl, häufige Ereignisse. Sie können bei-

spielsweise bei der Subtraktion gleicher Zahlen oder Multiplikation sehr kleiner Zahlen auftreten. Die für die Simulation einzig sinnvolle Antwort auf Zahl-Unterläufe ist die Ausgabe von Null, beziehungsweise die Ausgabe der kleinsten darstellbaren Zahl, falls Null durch das verwendete Zahlsystem nicht repräsentiert werden kann. Einen Underflow nicht abzufangen, wäre fatal, da anstatt Null dann eine der größten darstellbaren Zahlen ausgegeben werden könnte.

5.4 Folgerungen für die Rechenbeschleunigerarchitektur

Die in den letzten Abschnitten diskutierten Randbedingungen an einen Rechenbeschleuniger lassen sich wie folgt zusammenfassen:

- Für eine deutliche Beschleunigung muß eine Dauerrechenleistung von etwa 2 GFlops bereitgestellt werden.
- Die Kommunikationsbandbreite zwischen Beschleuniger und Host sollte über 200 MByte/sec betragen.
- Es soll eine Rechengenauigkeit entsprechend der Verwendung eines Gleitkommaformats mit 16 Mantissenbits erreicht werden.

Die hohen Anforderungen an die Übertragungsbandbreite lassen sich am besten erfüllen, wenn die Beschleunigerplattform möglichst eng mit dem Host-System verbunden ist. Die geforderte Kommunikationsbandbreite kann insbesondere durch eine Bus-basierte Lösung umgesetzt werden. Ein 64-Bit-PCI-Bus mit 66 MHz hat beispielsweise eine maximale Bandbreite von über 500 MByte/sec. Eine Implementierung über eine Standard-Netzwerkschnittstelle (z.B GBit-Ethernet) kommt nicht in Frage, da damit keine Dauerübertragungsbandbreite von 200 MByte/sec erreichbar ist. Andere Schnittstellen, wie z.B. Myrinet oder proprietäre Lösungen, könnten schnell genug sein, sind aber mit einem hohen Implementierungsaufwand verbunden.

Für den Aufbau einer Koprozessorplattform mit einer Dauerrechenleistung von über 2 GFlops können verschiedene Strategien in Erwägung gezogen werden. Bereits zu Beginn dieser Arbeit wurde der Ansatz motiviert, den Rechenbeschleuniger basierend auf der FPGA-Technologie zu entwickeln. Dieser Ansatz soll an dieser Stelle im Hinblick auf die nun präzisierten Anforderungen an die Rechenleistung im Vergleich zu alternativen Strategien diskutiert werden.

DSPs Im Bereich der Signalverarbeitung in der Industrie werden als Rechenbeschleuniger typischerweise digitale Signalprozessoren (DSPs) verwendet. Diese lassen sich leicht parallel betreiben und erreichen für Spezialanwendungen pro Prozessor Rechenleistungen im GFlops-Bereich. Solche Anwendungen sind beispielsweise digitale Filter für die Bild- und Tonverarbeitung. Hier werden viele Multiplizier-Akkumulier-Operationen benötigt, wofür die Rechenwerke der DSPs optimiert sind. Für die SPH-Berechnung werden Signalprozessoren jedoch kaum bessere Leistungen erzielen als Standardprozessoren, da sie über ein ähnlich enges Speicherinterface verfügen und die Rechenwerksstruktur dieser Prozessoren für die SPH-Summationsschleifen keine Vorteile bringt. Ein Leistungsgewinn kann also nur durch den parallelen Einsatz mehrerer DSPs

erreicht werden. Für eine Rechenleistung von 2 GFlops müssten schätzungsweise 5–10 Signalprozessoren verwendet werden, wobei völlig unklar ist, wie diese effizient mit den Teilchendaten versorgt werden können. Überdies ist die Implementierbarkeit eines Bus-basierten Koprozessors mit so vielen Hochleistungs-DSPs fraglich. Der Ansatz eines Standardrechners als Host müsste aufgegeben werden zugunsten eines Industrie-PC-Formfaktors oder einer proprietären Systemstruktur wie beispielsweise bei [35], was die Kosten vervielfachen würde[8].

ASICs Die größte Rechenleistung ließe sich erzielen, wenn ein anwendungsspezifischer Spezialchip (*ASIC*) entwickelt würde, der für die SPH-Berechnungen maßgeschnittene Rechenwerke in Form von Pipelines enthält. Dies könnte ähnlich wie bei der Entwicklung der GRAPE-Plattformen aus Abschnitt 2.4.2 geschehen. Für eine spezielle SPH-Formulierung ließen sich damit enorme Rechenleistungen erzielen. Wie in Abschnitt 2.3.3 erwähnt wurde, können die SPH-Algorithmen jedoch deutlich variieren. Sollen zusätzliche physikalische Effekte berücksichtigt werden, wie beispielsweise magnetohydrodynamische Kräfte, werden zudem Erweiterungen der Berechnungsformeln erforderlich. Eine für viele Forschergruppen anwendbare und für den Fortschritt der Simulationsalgorithmen offene Architektur kann also nicht über diesen Ansatz feststrukturierter Spezialchips geschaffen werden.

FPGAs Die Verwendung von FPGAs kann als Kompromiss zwischen der Flexibilität von Prozessoren und der extrem hohen Leistungsfähigkeit von ASICs gesehen werden. Ihre Struktur ermöglicht es, wie in den nächsten Kapiteln ausführlich beschrieben wird, hochspezialisierte Rechenpipelines aufzubauen. Trotz deutlich niedrigerer Taktfrequenz als bei Prozessoren (etwa Faktor 10–20) lässt sich damit für viele Anwendungen ein wesentlich höherer Datendurchsatz erreichen als bei Standardprozessoren. Die Rekonfigurierbarkeit sichert die Flexibilität des Ansatzes. Für unterschiedliche Formulierungen von SPH brauchen nur unterschiedliche FPGA-Designs aufgespielt zu werden. Es ist jedoch a priori nicht klar, ob die Logikressourcen aktueller FPGAs genügen, um die komplexen Rechenwerke für die SPH-Algorithmen zu implementieren. Dies zu zeigen wird ein elementarer Gedanke der weiteren Ausführungen dieser Arbeit sein.

Die FPGA-Koprozessorkarte MPRACE, wie sie in Abschnitt 3.3 vorgestellt wurde, genügt den wesentlichen Anforderungen, wie sie in diesem Abschnitt gegeben wurden. Als Einsteckkarte für den 64-Bit-PCI-Bus stellt sie eine hohe Kommunikationsbandbreite bereit (maximal 264 MByte/sec, limitiert durch den lokalen Bus auf dem Board, siehe auch Abschnitt 5.4.2). Der moderne FPGA beinhaltet, wie sich zeigen wird, genügend Logikressourcen, um die Rechenwerke mit ausreichend hoher Geschwindigkeit und Rechengenauigkeit zu implementieren. Deshalb wurde dieses Board als Prototypplattform verwendet und alle weiteren Ausführungen beziehen sich auf diesen Koprozessor.

---

[8] Bei einer proprietären Lösung wären zumindest die Initialkosten sehr hoch und zudem die Entwicklungszeit beträchtlich.

5.4.1 Grundlegende Struktur der Rechenwerke

Wie bereits motiviert wurde, kann eine Rechenleistung im GFlops-Bereich nur über die Parallelisierung der Rechenwerke erreicht werden. Dabei ist die vollständig auf Operatorebene parallelisierte Struktur als Rechenpipeline die naheliegendste Variante[9]. Nimmt man für die SPH-Formeln 100 Operationen an, und kalkuliert mit einer sehr konservativen Schätzung von 50 MHz Taktfrequenz im FPGA, ergibt sich für diese Stategie eine Rechenleistung von 5 GFlops. Dies entspricht sehr gut den Anforderungen and den Rechenbeschleuniger, weshalb diese Parallelisierungsstrategie für die Implementierung gewählt wurde.

Die Struktur der SPH-Berechnungen ist immer gleich. Für ein Teilchen mit Index i wird eine Summation über die SPH-Beiträge der Teilchen j einer Nachbarschaftsumgebung von Teilchen i durchgeführt. Die SPH-Beiträge ergeben sich aus den Teilchendaten mit Index i und j. Wird die zu berechnende SPH-Größe S genannt und werden die Summanden mit F bezeichnet, ergibt sich folgendes Schema:

$$S_i = \sum_{j \in Neighborlist(i)} F(x_{i1}, \ldots, x_{iN}, x_{j1}, \ldots, x_{jN}), \tag{5.15}$$

wobei die verwendeten Teilchendaten für Teilchen l mit $x_{l1}, \ldots, x_{lN}$ bezeichnet wurden. Dieses Rechenschema kann in eine Hardwarearchitektur übersetzt werden, wie sie in Abbildung 5.7 gezeigt ist. In dieser Schaltung wird davon ausgegangen, dass im Speicher an der Adresse j die Daten zu Teilchen j liegen. Zu Beginn einer Summation wird das Register *i-data* geladen und der Akkumulator wird zurückgesetzt. In jedem weiteren Takt wird ein neuer Datensatz der Nachbarteilchen j aus dem Speicher gelesen und über das Register *j-data* in die Pipeline gegeben. Wurde die gesamte Nachbarschaftsliste abgearbeitet, kann das Endresultat aus dem Akkumulator ausgelesen werden. Sollen mehrere Formeln gleichzeitig ausgewertet werden, beispielsweise die Berechnung der Dichte, Geschwindigkeitsdivergenz und Rotation des Geschwindigkeitsfeldes, werden im Modul *Arithmetic Unit* bei unveränderter äußerer Beschaltung einfach mehrere Summationen parallel ausgeführt.

5.4.2 Datenfluss

Eine hohe mittlere Datentransferrate über den PCI-Bus kann nur erreicht werden, wenn große Datenblöcke über *Direct Memory Access* (DMA) auf den Rechenbeschleuniger übertragen werden. Denn Einzelzugriffe über den PCI-Bus führen zu Übertragungsbandbreiten von nur etwa 5 MByte/sec für Lesezugriffe und ca. 30 MByte/sec für Schreibzugriffe, wie für das MPRACE-Board gemessen wurde. Werden DMA-Transfers eingesetzt können ab einer Blockgröße von 8 KByte sowohl für Lese- als auch für Schreibzugriffe Datenraten von 200 MByte/sec gemessen werden, wie in Abbildung 5.8 gezeigt wird. Dass gegenüber einer theoretischen Bandbreite von 512 MByte/sec nur etwa die Hälfte davon erreicht wird, folgt aus dem nur halb so schnellen lokalen Bus auf der MPRACE-Plattform.

[9] Andere Ansätze wären z.B., instruktionsbasierte Rechenwerke, gesteuert durch ein Mikroprogramm, auf dem FPGA zu implementieren oder eine Reihe von Operatoren parallel zu implementieren und die Verarbeitung über dynamische Verbindungsstrecken zwischen diesen Operatoren, kontrolliert durch eine State-Machine, zu steuern.

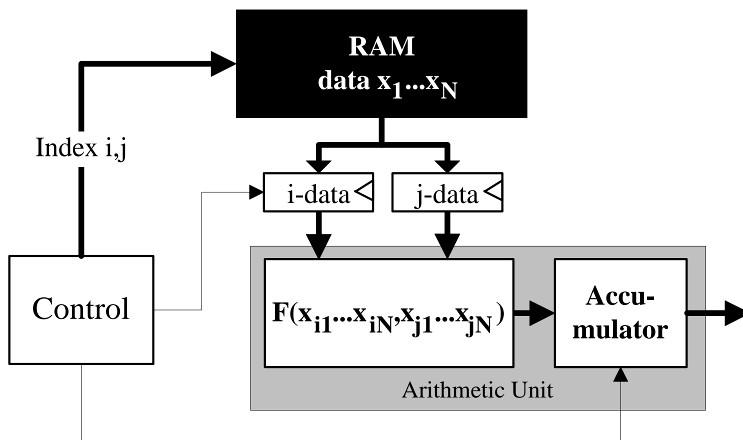


Abbildung 5.7: Hardwarearchitektur die sich aus dem abstrakten Rechenschema der SPH-Formeln ergibt.

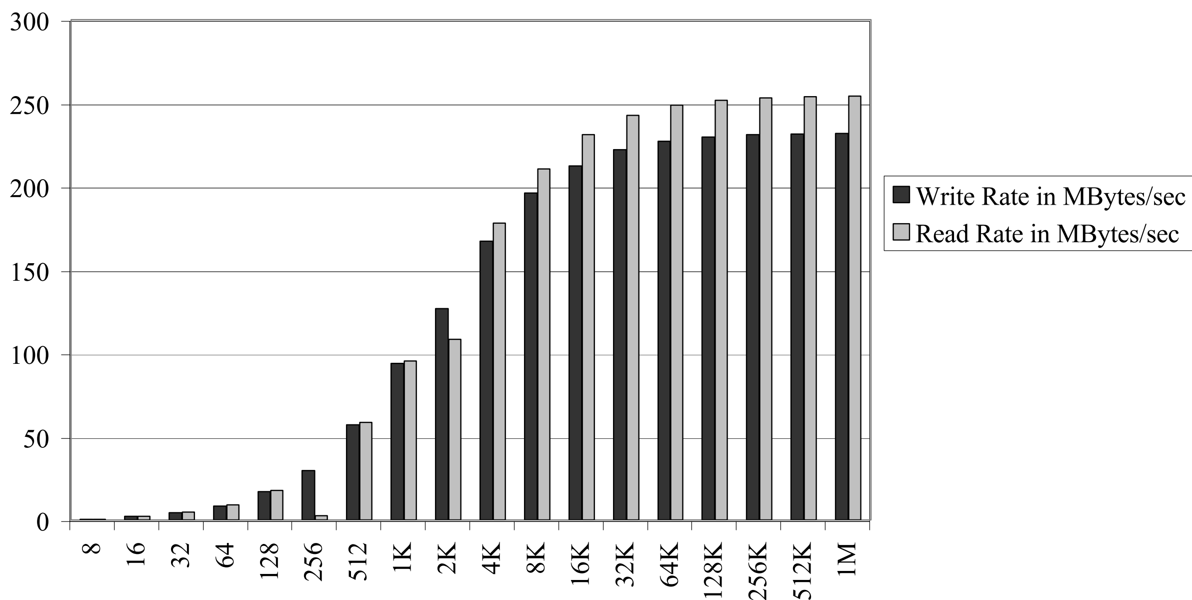


Abbildung 5.8: Datentransferleistung über den 64-Bit 66 MHz PCI-Bus für das MPRACE-Board.

5.4.3 Geeignetes Zahlenformat für die Rechenwerke

Die Gleitkommaarithmetik ist in jedem Fall ein numerisch geeignetes Zahlenformat für die Simulationen. Unter Verwendung dieses Zahlenformats wurden schließlich alle bisherigen Tests durchgeführt, insbesondere auch die Genauigkeitssimulationen aus Abschnitt 5.3.2. Entscheidend dafür, ob diese Zahlendarstellung auch für eine FPGA-Implementierung eine gute Lösung darstellt, ist die zu erwartende Performance bei einer Umsetzung im FPGA. In diesem Abschnitt erfolgt deshalb anhand der Grundlagen aus Kapitel 4 eine vergleichende Diskussion für die Gleitkommaarithmetik und alternative Rechentechniken.

Die sich typischerweise über viele Größenordnungen erstreckenden Werte astrophysikalischer Zustandsgrößen führen zur Grundvoraussetzung, dass das verwendete Zahlenformat einen weiten Zahlenbereich darstellen kann. Eine Verarbeitung alleine unter Verwendung von Festkommazahlen scheidet also aus. Es kann höchstens partiell mit Festkommazahlen gerechnet werden, wie beispielsweise bei der Berechnung der Glättungskernfunktionen. Diese haben als Eingangsgröße eine Zahl aus dem Intervall $[0,2]$, können also prinzipiell durch ein Rechenwerk mit Festkommazahlen umgesetzt werden, wobei dies für kleine Eingangsgrößen mit einem Verlust an Genauigkeit für die Ergebnisse einhergeht. Da es noch keine Untersuchungen dazu gibt, wie sich solche Rechenfehler in den Simulationen auswirken, soll vorerst auf die Verwendung von Festkommazahlen gänzlich verzichtet werden.

Neben dem Gleitkommaformat sind vor allem die logarithmische und semilogarithmische Zahlendarstellung interessante Kandidaten für eine Implementierung. Es wird davon ausgegangen, dass eine logarithmische Darstellung mit 24-Bit-Zahlen eine ähnliche Rechengenauigkeit erlaubt wie eine Gleitkommadarstellung mit 16-Bit-Mantissen und 8-Bit-Exponenten (siehe dazu [16]).

Wie in Abschnitt 4.4 erläutert wurde, können in logarithmischer Darstellung Multiplikationen, Divisionen, Potenzen und Wurzeln sehr einfach implementiert werden. Dagegen muss für Additionen und Subtraktionen eine nichtlineare Funktion ausgewertet werden, was in der Regel über eine LUT-basierte Approximation geschieht. Erschwerend kommt hinzu, dass für die Addition und Subtraktion verschiedene Approximationen anzuwenden sind. Ein Addier/Subtrahier-Operator, wie er für die Summation vorzeichenbehafteter Zahlen erforderlich ist, erfordert also die Implementierung von zwei Operatoren (wobei alles außer den LUTs für die Approximation gemeinsam implementiert werden kann). Das Erfordernis einer Funktionsapproximation für Addierer und Subtrahierer ist das Kernproblem bei logarithmischen Zahldarstellungen. Wird diese Näherung beispielsweise durch eine wie in Abschnitt 4.5.1 beschriebene lineare Approximation durchgeführt, werden bereits bei logarithmischen 18-Bit-Zahlen pro Addition mindestens 2 Block-RAM-Elemente eines Virtex-II-FPGAs benötigt. Wie in Abschnitt 5.3 angemerkt wurde, kann es für die Simulationsrechnungen durchaus erforderlich werden, mit logarithmischen 24-Bit-Zahlen (entsprechend 16-Bit-Mantissen einer Gleitkommadarstellung) zu arbeiten, wofür bereits mehr als 16 solcher Block-RAM-Elemente für eine einzige Addition benötigt werden. Bei genauerer Betrachtung der SPH-Formeln findet man ein Verhältnis von Addierern zu Multiplizierern von etwa 2 zu 3. Es werden für eine Implementierung mit logarithmischen Zahlen bereits bei einer 18-Bit-Darstellung deutlich mehr Block-RAM-Elemente benötigt als Block-Multiplizierer

im Fall einer Implementierung durch Gleitkommaarithmetik[10]. Dabei ist die Verdoppelung der LUT-Anzahl für Addier/Subtrahier-Bausteine noch nicht eingerechnet. Bei Erhöhung der Genauigkeit steigt der Bedarf an Block-RAM-Elementen zudem dramatisch an. Da die entgültige Rechengenauigkeit der Rechenbeschleunigerarchitektur nicht auf die geringe Genauigkeit einer logarithmischen 18-Bit-Darstellung festgelegt werden kann, scheidet das logarithmische Zahlensystem aus.

Als letzte abzuwägende Alternative zum Gleitkommaformat bleiben die semilogarithmischen Zahlen. Wie in Abschnitt 4.4.3 beschrieben wurde, kann mit diesem Zahlenformat für den hier vorliegenden Fall, dass ähnlich viele Addierer wie Multiplizierer benötigt werden, eine Verschiebung von Multiplizierer- auf RAM-Ressourcen erreicht werden. Der übrige Logikressourcenverbrauch wird jedoch nur unwesentlich verringert. Dividierer können im Vergleich mit einer Gleitkommaimplementierung wesentlich ressourcensparender aufgebaut werden, jedoch ist unklar, wie es sich für den Quadratwurzeloperator verhält. Für FPGA-Architekturen, die Block-RAM-Elemente enthalten, jedoch nicht über Block-Multiplizierer verfügen, ist das semilogarithmische Zahlensystem sehr vielversprechend. In unserem Fall bei Verwendung eines Virtex-II-FPGAs sind jedoch genügend Block-Multiplizierer vorhanden. Damit überwiegen die durch den LUT-basierten Ansatz entstehenden Nachteile bezüglich der Skalierbarkeit in der Rechengenauigkeit die Vorteile dieses Zahlensystems.

[10]Hier wurde bereits im Vorgriff auf Abschnitt 6.2.3 verwendet, dass für einen Gleitkommamultiplizierer bis zu einer Mantissenbreite von 18 Bit die interne Festkommaoperation auf einen Block-Multiplizierer des Virtex-II-FPGAs abgebildet werden kann.

Kapitel 6

Implementierung der Arithmetik auf FPGAs

In diesem Kapitel wird die Entwicklung der elementaren Bausteine für eine Pipeline-Implementierung diskutiert, mit der die astrophysikalischen SPH-Algorithmen simuliert werden können.

In Kapitel 5 wurde auf die Anforderungen an die Arithmetik eingegangen. Wie gezeigt, muss das verwendete Zahlenformat sowohl eine hohe Präzision garantieren als auch viele Größenordnungen darstellen können und sollte zudem in der Rechengenauigkeit leicht skalierbar sein. Geeignet dazu ist ein Gleitkommaformat mit parametrisierter Mantissenbreite, wie es in Abschnitt 4.3 dargestellt wurde. Wie in Abschnitt 5.4.3 motiviert, ist dieses Format im Fall einer SPH-Implementierung auf einer modernen FPGA-Plattform anderen Zahlenformaten überlegen.

Deshalb wird im Hauptteil dieser Arbeit das Gleitkommaformat verwendet, entsprechend konzentriert sich dieses Kapitel auf die Implementierung der zugehörigen Operatoren. Alle elementaren Gleitkommaoperationen, wie in 4.3.3 gesehen, beinhalten eine Fixpunkt-Operation. Deshalb wird in Abschnitt 6.1 zuerst auf die Implementierung von Festkommaoperatoren eingegangen. Darauf aufbauend, werden in Abschnitt 6.2 die für diese Arbeit erforderlichen Gleitkommaoperatoren entwickelt.

6.1 Festkommaarithmetik auf FPGAs

In diesem Abschnitt wird die Implementierung der wichtigsten Operatoren auf FPGAs vorgestellt: Addition/Subtraktion, Multiplikation, Division und Quadratwurzel. Die Diskussion von nicht-elementaren Operatoren wurde auf die Implementierung der Quadratwurzel beschränkt, da nur diese Bausteine für die Implementierung des Rechenbeschleunigers benötigt wurden.

6.1.1 Addition und Subtraktion

Zentrale Rechenelemente für sämtliche Festkomma-Operatoren sind Integer-Addierer beziehungsweise Addier-Subtrahier-Elemente. Entsprechend gründlich werden die Eigenschaften verschiedener Implementierungsstrategien, wie sie in Abschnitt 4.1.2 vorgestellt wurden, analysiert.

Bei der Konstruktion von schnellen Addierer-Rechenwerken für Mikroprozessoren oder anwendungsspezifische Spezialchips (ASICs) ist es ein weitverbreitetes Standardverfahren, Carry-Lookahead-Addierer einzusetzen. Diese haben ein Zeitverhalten, das logarithmisch mit der Addiererbreite skaliert. Für sehr breite Operanden werden darüber hinaus Carry-Select-Addierertechniken eingesetzt. Sollen mehr als zwei Argumente summiert werden, werden die dazu erforderlichen Addiererbäume typischerweise durch Carry-Save-Addierer mit nachfolgender Carry-Lookahead-Stufe implementiert. Zur Untersuchung, ob für FPGAs ähnliche Design-Kriterien gelten, wurde eine Vergleichsstudie angefertigt, welche die Eigenschaften von Carry-Lookahead-Addierern gegenüber der Implementierung als einfache Ripple-Carry-Addierer für verschiedene Bitbreiten gegenüberstellt. Die Carry-Lookahead-Addierer wurden nach dem in Abbildung 4.5 gezeigten Bauprinzip umgesetzt und die Ripple-Carry-Addierer entsprechend Abbildung 3.5 implementiert. Ergänzend wurde auch die Implementierung von Carry-Select-Addierern untersucht (implementiert, wie in Abbildung 4.7 und 4.8 gezeigt mit Teiladdierern in Ripple-Carry-Bauweise). Tabelle 6.1 gibt die Resultate wieder.

	Carry-Lookahead			Ripple-Carry			Carry-Select		
Breite	LUTs	f/MHz	WP	LUTs	f/MHz	WP	LUTs	f/MHz	WP
4	11	269	4 L	4	425	1 L			
						+ 3 C			
8	28	206	5 L	8	390	1 L			
	31	263	4 L			+ 7 C			
16	57	140	7 L	16	334	1 L			
	69	166	6 L			+ 15 C			
32	117	106	9 L	32	261	1 L	65	249	2 L + 16 C
	153	121	9 L			+ 31 C	86	202	3 L + 8 C
64	246	75	13 L	64	181	1 L	129	189	2 L + 32 C
	281	97	11 L			+ 63 C	167	169	3 L + 16 C
128	497	58	17 L	128	112	1 L	257	138	2 L + 64 C
	527	80	13 L			+ 127 C	330	137	3 L + 32 C
256	984	50	19 L	256	64	1 L	513	87	2 L + 128 C
	1026	68	15 L			+ 255 C	652	102	3 L + 64 C
512	1982	51	18 L	512	34	1 L	1025	54	2 L + 256 C
						+ 511 C	1316	74	3 L + 128 C

Tabelle 6.1: Leistung verschiedener Addierertechniken bei Implementierung in Virtex-II-FPGA XC2V3000-4. Abhängig von der Breite der Addierer wurden die Logikressourcen (LUTs), die maximale Taktfrequenz f und die Länge des zeitkritischen Logikpfades (WP) in LUTs (L) und Carry-Multiplexer (C) bestimmt.

Es wird deutlich, dass die einfache, aber durch die FPGAs mittels spezieller Carry-Propagation-Netzwerke unterstützte Implementierung von Ripple-Carry-Addierern über alle für Gleitkommaoperationen auftretende Bitbreiten deutlich schneller und zugleich um einen Faktor 3-4 ressourcensparender ist als eine Carry-Lookahead-Lösung. Erst bei sehr hohen Breiten über 256 Bit kommt das logarithmische Zeitverhalten zum Tragen. Doch auch für diesen Bereich gibt es

bessere Design-Strategien, wie die Spalte für Carry-Select-Addierer zeigt. Diese erreichen bei deutlich geringerem Logikaufwand wesentlich höhere Taktfrequenzen als die Carry-Lookahead-Addierer. Es sei hier noch angemerkt, dass durch Pipelining die Geschwindigkeit der Ripple-Carry-Addierer ohne zusätzlichen Aufwand an LUTs, jedoch unter Aufwendung zusätzlicher Register erhöht werden kann, indem die Operationen zerteilt werden und die Addition der Stücke nacheinander erfolgt, entsprechend dem Erscheinen der Carry-Out-Signale der Teiladdierer.

Bis jetzt wurden nur Integer-Addierer betrachtet. Daraus ergeben sich jedoch auf sehr einfache Weise auch die Festkomma-Addierer. Wie im Abschnitt 4.2 bereits gezeigt wurde, kann die Beziehung zwischen der Addition auf Festkommazahlen a und b mit Resultat c, wobei a, b und c jeweils l Nachkommastellen haben und der Implementierung durch einen Integer-Addierer durch folgende Gleichung beschrieben werden:

$$c = a \pm b = (a\, r^l \pm b\, r^l) \cdot r^{-l}. \tag{6.1}$$

Die Zahlen a und b können also als Integer-Werte $a\, r^l$ und $b\, r^l$ verarbeitet werden und es muss lediglich eine Kommaverschiebung des Resultats um $-l$ Stellen erfolgen, um das Festkomma-Resultat c zu erhalten.

6.1.2 Multiplikation

Wie die Addition kann auch die Multiplikation $c = a \cdot b$ von Fixpunkt-Zahlen a, b auf eine Operation auf Integer-Zahlen zurückgeführt werden, wie bereits in Abschnitt 4.2 gezeigt wurde:

$$c = a \cdot b = (a\, r^l \cdot b\, r^l) \cdot r^{-2l}. \tag{6.2}$$

Hier muss für das Fixpunkt-Resultat also eine Kommaverschiebung um $2\, l$ Stellen erfolgen. Im weiteren Verlauf dieses Abschnitts kann die Diskussion deshalb auf die Implementierung von Integer-Multiplizierern beschränkt werden.

Der für den größten Teil der Arbeit verwendete FPGA der Virtex-II-Serie ermöglicht die Implementierung von Multiplizierern sowohl mit programmierbaren Logikelementen, wie sie in jedem FPGA vorhanden sind, als auch über die vorhandenen Block-Multiplizierer, wie sie in Abschnitt 3.1.2 erwähnt wurden. Im Verlauf der Implementierung der Arithmetik wurde mit verschiedenen FPGA-Typen gearbeitet, weshalb beide Varianten implementiert wurden. Da die Ergebnisse dieser Arbeit auch für andere FPGA-Typen anwendbar sein sollen, wird nicht nur die einfachere Methode mit den Block-Multiplizierern beschrieben, sondern auch auf die allgemeiner verwendbare Implementierung von Multiplizierern ausführlich eingegangen. Letztere wird zwar ebenfalls in einer auf die interne Struktur von Virtex-II-FPGAs optimierten Form diskutiert, die Methode ist jedoch auf die meisten neueren FPGA-Typen mit ähnlichen Ergebnissen übertragbar.

Implementierung durch Logikressourcen des FPGAs

In Abschnitt 4.1.3 wurden die Array-Multiplizierer vorgestellt. Wie dort in Abbildung 4.12 zu sehen ist, bestehen diese Multiplizierer im Wesentlichen aus einem Netzwerk von Volladdierern,

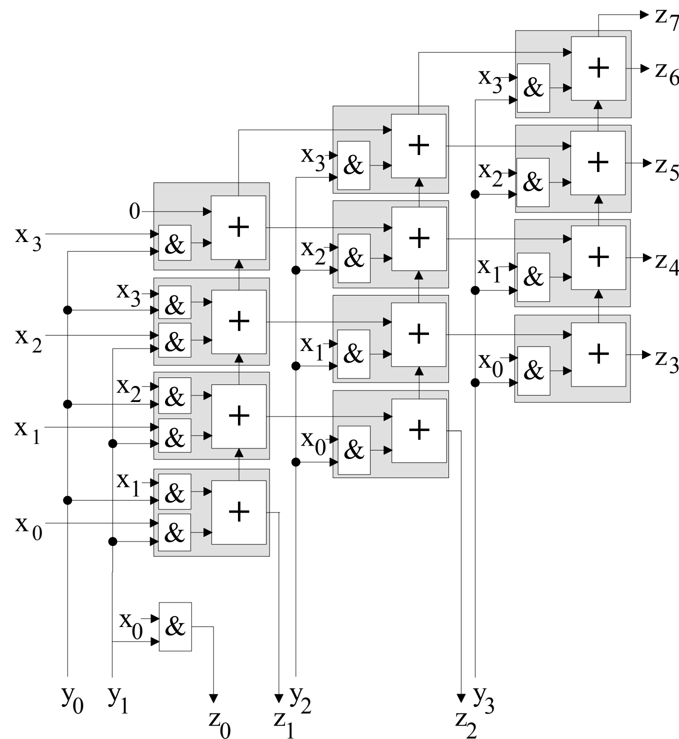


Abbildung 6.1: Multipliziererschaltung für 4-Bit Integer-Zahlen mit Multiplizier-Addier-Primitiven (graue Blöcke).

denen UND-Gatter vorgeschaltet sind. Diese Struktur lässt sich sehr effizient auf die FPGA-Ressourcen eines Virtex-FPGAs abbilden. Wie in Abschnitt 3.1.2 beschrieben und in Abbildung 3.6 illustriert wurde, lässt sich ein 1-Bit-Volladdierer für die Bits a und b und Carry-Eingang c_{in} mit einer UND-Verknüpfung auf dem Eingangsbit b mit dem Signal g durch ein halbes Slice des FPGAs implementieren. Da die verwendete LUT in diesem Fall noch einen freien Eingang besitzt, kann in derselben LUT auch das Bit a mit einem weiteren Eingangsbit h verknüpft werden, ohne zusätzliche Logikressourcen zu benötigen. Aus solchen Multiplizier-Primitiven können $N \times 1$-Bit-Multiplizierer zusammengesetzt werden, welche über das gleiche schnelle Carry-Netzwerk verbunden sind wie die im letzten Abschnitt besprochenen Ripple-Carry-Addierer. Daher verhält sich die Geschwindigkeit solcher Multiplizier-Elemente mit der Breite genau so wie bei den Addierern. Die Schaltung nach Abbildung 4.12 verwendet außer in der letzten Stufe Carry-Save-Addierer. Bei der FPGA-Implementierung ist eine Umsetzung mit den eben beschriebenen $N \times 1$-Bit-Multiplizierern besser geeignet, da wie bei den Addierern die Vorteile der Nutzung der im FPGA vorhandenen speziellen Carry-Leitungen den Gewinn durch weniger tiefe Logikketten überwiegt. Abbildung 6.1 zeigt eine nach diesen Kriterien aufgebaute Schaltung eines 4×4-Bit Multiplizierers, der in 6 $1/2$ Slices (13 LUTs) implementiert werden kann.

Tabelle 6.2 zeigt den Ressourcenverbrauch und die Geschwindigkeit von Array-Multiplizier-Schaltungen für Breiten der Operanden x und y (gleiche Bezeichnung wie in Abbildung 6.1) von vier Bit bis 24 Bit. Ebenfalls sind die Ergebnisse für eine feste Breite von 24 Bit für x bei

variierender Breite von zwei bis acht Bit für y dargestellt. Insbesondere daran ist gut zu sehen, dass die Geschwindigkeit vor allem von der Breite des Operanden y abhängig ist, während z.B. ein 24×4-Multipizierer nicht sehr viel langsamer als ein 4×4-Multiplizierer ist. Die Daten zu den Operatoren, die wie in Abbildung 6.1 ohne Register implementiert wurden, sind in der Tabelle durch die Pipelinetiefe ∞ angedeutet.

Der starken Abnahme der Geschwindigkeit mit der Breite von y kann durch Einfügen von Pipeline-Registern entgegengewirkt werden, wie in Tabelle 6.2 zu sehen ist. Die Latenzzeit des Operators steigt für einen solchen Aufbau dann linear mit der Breite von y an, wenn die Geschwindigkeit bzw. die Pipelinetiefe konstant gehalten wird. Außerdem werden zusätzliche Pipeline-Register für x, Teile von y und für die bereits ermittelten Ergebnisbits erforderlich. Während Pipeline-Register, welche die Ausgabe von LUTs (z.B. das Ergebnisbit einer Multiplizier-Addier-Primitiven) speichern, in der Regel in die gleichen Slices gepackt werden wie diese LUTs, ist dies für die Pipeline-Register für x, y und die Ergebnisbits nicht der Fall. Der Logikzellenverbrauch steigt also insgesamt deutlich mit der Verkürzung der Pipelinestufen (nicht in der Tabelle dargestellt, da dies erst nach dem Place&Route-Prozess sichtbar wird).

Ein anderer Weg, die Geschwindigkeit für breite Multiplizierer zu erhöhen, besteht darin, einen Multiplizierer nach der Tree-Methode, wie in Abschnitt 4.1.3 beschrieben, aufzubauen. Die Partialprodukte können durch Array-Multiplizierer mit geringer Tiefe berechnet werden und werden über einen Addiererbaum reduziert. Auf diese Weise können Multiplizierer verschiedener Breite mit einer logarithmisch entsprechend der Höhe des Addiererbaumes skalierenden Latenzzeit aufgebaut werden (für die hier betrachteten Operandenbreiten kann die Latenz sogar konstant gehalten werden).

Die Multiplizierer wurden in dieser Arbeit auf diese Weise implementiert. Abbildung 6.2 zeigt als Beispiel einen 16-Bit-Multiplizierer. Tabelle 6.3 zeigt die Implementierungsergebnisse für Integer-Multiplizierer bis zu einer Breite von 24 Bit. Wie zu sehen ist, ergibt sich im Vergleich mit den Array-Multiplizierern bei einem geringen Mehrverbrauch von etwa 10 Prozent an LUTs eine hohe Geschwindigkeit bei kleiner Latenz. Alle Register sind vorausgehenden LUTs assoziiert und können damit ohne zusätzliche Logikzellen des FPGAs implementiert werden (im Gegensatz zu den Pipeline-Registern im Fall der Array-Multiplizierer).

Implementierung durch Block-Multiplizierer

Da die Multiplikation von Festkommazahlen bei vielen Anwendungen, für die FPGAs eingesetzt werden, wichtiger Bestandteil ist, werden in einigen neueren FPGA-Generationen Multiplizier-Elemente bereits als feste Schaltung integriert. So auch in den FPGAs der Virtex-II-Serie (von Xilinx Block-Multiplizierer genannt), wie in Abschnitt 3.1.2 bereits erwähnt wurde. Festverdrahtete Multiplizierer nehmen auf dem FPGA nur einen Bruchteil der Fläche ein, die für einen gleichwertigen Operator, implementiert durch die programmierbare Logik, benötigt wird. Im FPGA der Serie Virtex-II sind je nach Größe des FPGAs bis zu 168 18-Bit-Multiplizierer (für Signed-Integer-Zahlen) vorhanden. Integer-Zahlen, die breiter als 18-Bit sind, können durch Verwendung mehrerer Block-Multiplizierer mit anschließenden Additionsstufen für die Partialprodukte implementiert werden. Moderne Synthese-Werkzeuge wie z.B. das in dieser Arbeit verwendete Programm Synplify der Firma Synplicity erzeugen aus der rein verhaltensmäßigen

Breite x	Breite y	Pipeline-tiefe	Register	LUTs	Geschwindigkeit /MHz	Latenz
4	4	∞	0	13	112	0
8	8	∞	0	57	54.2	0
8	8	4	40	56	85.5	2
12	12	∞	0	133	36.7	0
12	12	4	96	132	82.4	3
12	12	3	120	137	101	4
12	12	2	165	143	131	6
16	16	∞	0	241	27.5	0
16	16	4	161	246	80.0	4
16	16	3	190	251	97.5	6
16	16	2	281	259	125	8
20	20	∞	0	381	21.9	0
20	20	4	239	393	77.9	5
20	20	3	314	401	94.4	7
20	20	2	429	407	120	10
24	24	∞	0	553	18.2	0
24	24	4	331	573	76.1	6
24	24	3	422	580	91.8	8
24	24	2	609	587	115	12
24	2	∞	0	25	156	0
24	3	∞	0	49	115	0
24	4	∞	0	73	91.8	0
24	5	∞	0	97	76.1	0
24	6	∞	0	121	65.0	0
24	7	∞	0	145	56.7	0
24	8	∞	0	169	50.3	0

Tabelle 6.2: Ergebnisse für Ressourcenverbrauch und Geschwindigkeit von Array-Multiplizierern bei Virtex-II-FPGA XC2V3000-4 nach der Synthese mit Synplify. Eine Pipelinetiefe von ∞ bedeutet, dass keine Register verwendet wurden.

Breite	Register	LUTs	Geschwindigkeit/MHz	Latenz
4	8	12	113	1
8	40	64	137	2
12	72	146	129	2
16	112	264	123	2
20	160	410	96.6	2
24	216	592	92.8	2

Tabelle 6.3: Ressourcenverbrauch, Geschwindigkeit und Latenz von Integer-Multiplizierern bei Virtex-II-FPGA XC2V3000-4 nach Synthese mit Synplify.

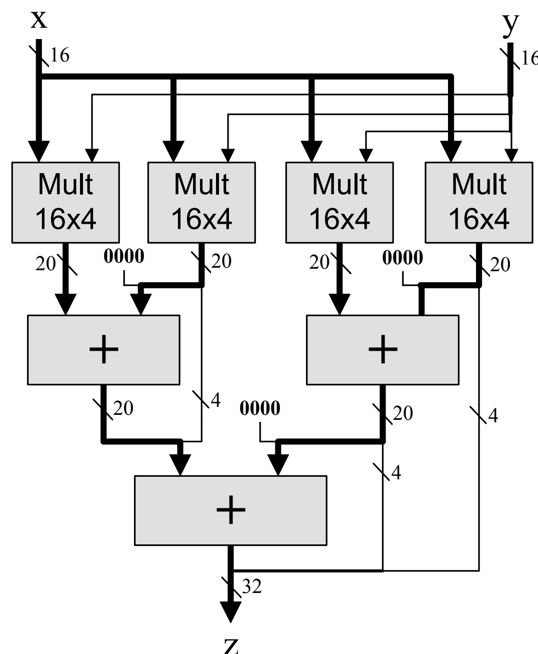


Abbildung 6.2: Multipliziererschaltung für 16-Bit-Integer-Zahlen mit Array-Multiplizierern zur Berechnung der fünf 16-Bit×4-Bit-Partialprodukte.

Beschreibung der Multiplikation in der Hardwarebeschreibungssprache automatisch die Umsetzung mit den Block-Multiplizierern. Zum Beispiel wird dann ein auf diese Weise erzeugter 24-Bit-Multiplizierer nach der Divide-And-Conquer-Methode aus vier Block-Multiplizierern und zwei Addierern zusammengesetzt. Wie in [13] aufgezeigt wird, kann ein solcher Multiplizierer mithilfe der Methode nach Karatsuba und Ofman aus nur drei Block-Multiplizierern, drei Addierern und zwei Subtrahierern zusammengesetzt werden. Dies ergibt sich aus folgender Zerlegung der Multiplikation für $2k$-Bit-Operanden x und y, wobei x und y in k-Bit Stücke x_h, x_l, y_h und y_l zerteilt werden:

$$\begin{aligned} x \cdot y &= (x_h\, 2^k + x_l) \cdot (y_h\, 2^k + y_l) \\ &= x_h \cdot y_h \cdot (2^{2k} - 2^k) + (x_h + x_l) \cdot (y_h + y_l)\, 2^k + x_l \cdot y_l \cdot (1 - 2^k). \end{aligned} \tag{6.3}$$

Übersteigt die Breite der zu implementierenden Multiplizierer die Breite der Block-Multiplizierer nur wenig, ist es sinnvoll, Block-Multiplizierer und FPGA-Logikelemente für die Partialprodukte zu mischen[1], was dann jedoch manuell durch strukturellen HDL-Code erzeugt werden muss.

Quadrierung

Soll nicht eine allgemeine Integer-Multiplikation $x \cdot y$, sondern lediglich eine Quadratur x^2 berechnet werden, kann diese Operation wesentlich ressourceneffizienter als beim oben beschrie-

[1] Beispielsweise auf die in [13] beschriebene Weise.

benen Multiplizierer implementiert werden. Das liegt vor allem daran, dass viele Partialsummen des Produktes identisch sind und deshalb gemeinsam berechnet werden können, wie an folgender Gleichung deutlich wird:

$$\begin{aligned} z = (x_{k-1} \cdots x_0)_r^2 &= \textstyle\sum_{i=0}^{k-1} \sum_{j=0}^{k-1} x_i x_j \, r^{i+j} \\ &= \textstyle\sum_{i=0}^{k-1} x_i^2 \, r^{2i} \, + \, 2 \cdot \sum_{i=0}^{k-1} \sum_{j=i+1}^{k-1} x_i x_j \, r^{i+j}. \end{aligned} \tag{6.4}$$

Damit müssen statt k^2 nur noch $k + \frac{k^2-k}{2}$ Partialprodukte summiert werden. Für $r = 2$ ergibt sich:

$$z = \sum_{i=0}^{k-1} x_i \, 2^{2i} \, + \sum_{i=0}^{k-1} \sum_{j=i+1}^{k-1} (x_i \, x_j) \, 2^{i+j+1}. \tag{6.5}$$

In Abbildung 6.3 ist auf der linken Seite ein auf diese Weise aufgebauter Quadrierer für 4-Bit-Ganzzahlen gezeigt. Im Vergleich zur Implementierung des 4-Bit-Multiplizierers wie in Abbildung 6.1 ergibt sich eine Ressourcenersparnis von über 50 Prozent.

Eine weitere Möglichkeit, einen Quadrierer zu bauen, besteht darin, eine Look-Up-Table (LUT) dafür anzuwenden. Das Ergebnis der Operation ist nur von den k Bits des Arguments abhängig. Es gibt also bei Binärzahlen nur 2^k verschiedene Quadratzahlen, im Gegensatz zu 2^{2k} möglichen Produkten bei der Multiplikation. Bei einem 4-Bit-Quadrierer genügt beispielsweise eine Tabelle mit 16 Einträgen, um alle Resultate zu speichern. Werden die 4-Bit-Argumente als 4-Bit-Adressen interpretiert und ist an der damit adressierten Speicherstelle einer LUT der Tabellenwert des Resultats gespeichert, erfüllt diese LUT die Funktion des Quadrierers. In Abbildung 6.3 ist auf der rechten Seite die Implementierung des 4-Bit-Quadrierers nach dieser Methode gezeigt und man erkennt, dass sich die gleiche Menge an Ressourcen wie für den auf Multiplizier-Addier-Elementen basierenden Quadrierer ergibt (eine 1-Bit-16-Adress-LUT benötigt genauso wie ein 1-Bit-Mult-Add-Element 1/2 Slice des Virtex-II-FPGAs). Diese Variante, einen Quadrierer für 4-Bit-Zahlen zu bauen, ist in diesem Fall die bevorzugte, da sich für die FPGA-Implementierung eine Logikverzögerung ergibt, die der einer einzigen LUT entspricht und deshalb extrem kurz ist.

Größere Quadrierer werden nach der Divide-And-Conquer-Methode aus kleineren Quadrierern und Array-Multiplizierern über einen anschließenden Addiererbaum zusammengesetzt, ähnlich wie beim Aufbau größerer Multiplizierer, jedoch mit der Maßgabe, den Anteil der Array-Multiplizierer zu minimieren. Dies führt zu einer relativ komplexen Implementierung solcher Operatoren. Abbildung 6.4 zeigt als Beispiel den Aufbau eines Quadrierers für 16-Bit-Binärzahlen. Der sich ergebende Ressourcenverbrauch, die Geschwindigkeit und die Latenz sind für verschiedene Breiten der Operanden in Tabelle 6.4 aufgestellt.

6.1.3 Division

Die Division wurde in dieser Arbeit basierend auf der in Abschnitt 4.1.4 erläuterten Non-Restoring-Methode implementiert. Gegenüber der dort ebenfalls erwähnten Non-Performing-Methode zeichnet sich dieser Algorithmus dadurch aus, dass anstatt des Aufbaus aus Multiplexern und Addierern bei einer parallelen Implementierung lediglich Stufen von Addier-Subtrahier-Elementen

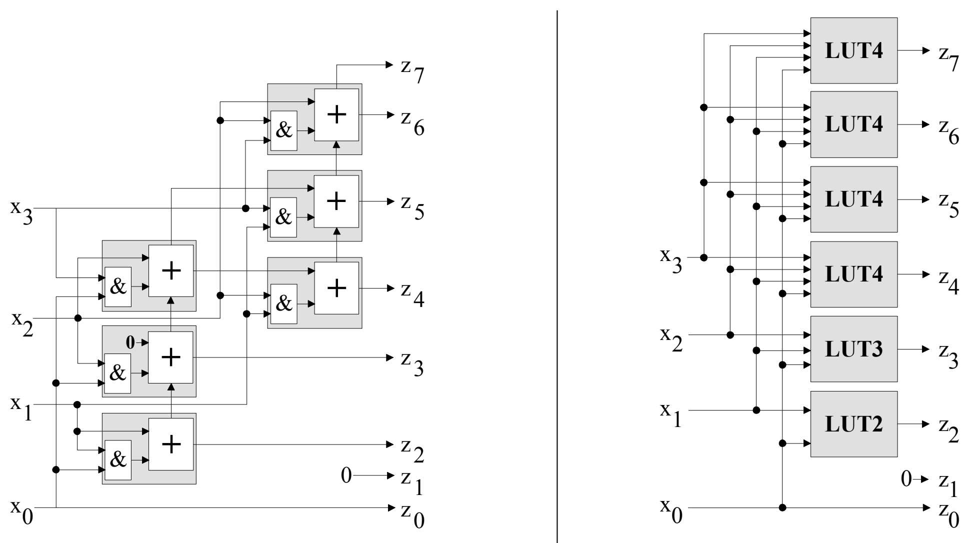


Abbildung 6.3: Schaltungsvarianten zur Quadrierung einer 4-Bit-Zahl. Links mit Multiplizier-Addier-Primitiven, rechts mit Look-Up-Tables.

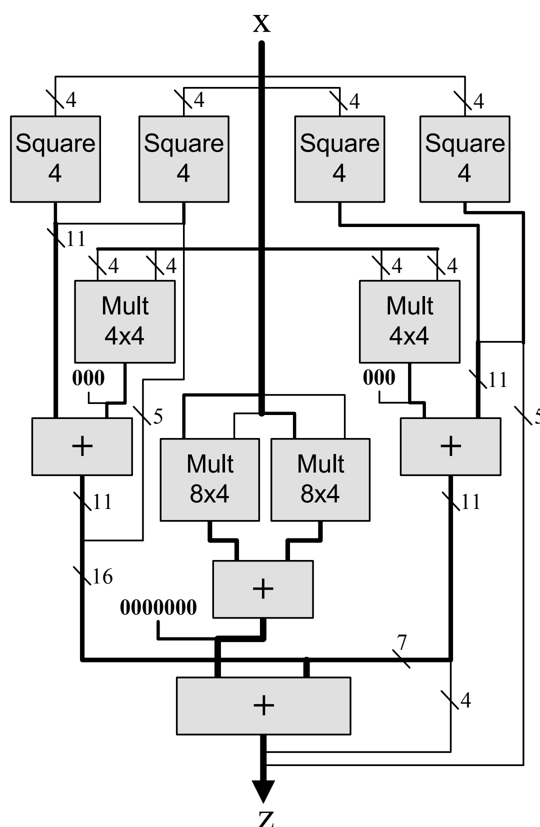


Abbildung 6.4: Quadriererschaltung für 16-Bit-Integer-Zahlen aus kleinen Quadrierern und Array-Multiplizierern zur Berechnung der Partialprodukte.

Breite	Register	LUTs	Geschwindigkeit/MHz	Latenz
4	0	6	355	0
8	15	45	124	1
12	64	89	136	2
16	84	177	124	2
20	119	262	124	2
24	160	328	98.3	3

Tabelle 6.4: Ressourcenverbrauch, Geschwindigkeit und Latenz von Integer-Quadrierern bei Virtex-II-FPGA XC2V3000-4 nach Synthese mit Synplify.

notwendig werden. Wie in Abschnitt 3.1.2 gezeigt (siehe inbesondere Abbildung 3.5), ermöglicht der Virtex-II-FPGA die Implementierung eines 1-Bit-Addierer/Subtrahierer-Elements in einem halben Slice, während ein 1-Bit-Multiplexer und Addierer den doppelten Ressourcenaufwand benötigen. Die Gründe, warum die SRT-Divisionsmethode nicht verwendet wurde, wurden bereits bei der Vorstellung des Verfahrens in Abschnitt 4.1.4 diskutiert.

Die Non-Restoring-Methode für Festkommazahlen soll nun in etwas modifizierter Form aufgestellt werden, da wir hier daran interessiert sind, Dividenden, Divisoren und Quotienten mit gleicher Breite der Fixpunktdarstellung zu verwenden. Dazu nehmen wir an, dass der Dividend e (engl. enumerator), der Divisor d und der Quotient q binäre Festkommazahlen mit einer Vorkommastelle und l Nachkommastellen sind:

$$\begin{aligned} e &= e_0.e_{-1}\cdots e_{-l} \\ d &= d_0.d_{-1}\cdots d_{-l} \\ q &= q_0.q_{-1}\cdots q_{-l}. \end{aligned}$$

Durch eine Kommaverschiebung in davon abweichend dargestellten Festkommazahlen e und d kann diese Form in jedem Fall erreicht werden (z.B. bei nach Gleichung 4.27 dargestellten Zahlen um $k-1$ Stellen). Die Größe des Quotienten ist von dieser Verschiebung unabhängig, solange e und d um die gleiche Anzahl von Stellen verschoben werden. Das Resultat q kann nicht mehr mit einer Vorkommastelle dargestellt werden, wenn die Bedingung $e \geq 2d$ zutrifft – es liegt dann also ein Überlauf vor. Deshalb muss gelten:

$$e < 2d. \tag{6.6}$$

Die Iterationsformel für die Non-Restoring-Methode der Division ergibt sich nun ensprechend

Gleichung 4.14 wie folgt:

$$
\begin{aligned}
s^{(0)} &= e \\
s^{(i)} &= \begin{cases} 2 \cdot s^{(i-1)} - 2 \cdot d & : \quad s^{(i-1)} \geq 0 \\ 2 \cdot s^{(i-1)} + 2 \cdot d & : \quad s^{(i-1)} < 0 \end{cases} \\
q_{1-i} &= \begin{cases} 1 & : \quad s^{(i)} \geq 0 \\ 0 & : \quad s^{(i)} < 0 \end{cases} \\
i &= 1 \ldots (l+1).
\end{aligned}
\tag{6.7}
$$

Die Bedingung 6.6 garantiert, dass auch $|s^{(i)}| < 2d$ gilt, was man wie folgt induktiv schließen kann:

$$
\begin{aligned}
s^{(0)} &= e < 2d \\
\text{Für } s^{(i)} \geq 0,\ |s^{(i)}| < 2d \Rightarrow s^{(i+1)} &= \underbrace{2\, s^{(i)}}_{<4d} - 2\, d \ < \ 2d \\
\text{Für } s^{(i)} < 0,\ |s^{(i)}| < 2d \Rightarrow s^{(i+1)} &= \underbrace{2\, s^{(i)}}_{>-4d} + 2\, d \ > -2d
\end{aligned}
$$

$\square$

Die Beziehung $|s^{(i)}| < 2d$ impliziert, dass $s^{(i)}$ dargestellt werden kann durch die Festkommarepräsentation $(\pm s_1^{(i)} s_0^{(i)} . s_{-1}^{(i)} \cdots s_{-l}^{(i)})$. Wird der Restwert der Division benötigt, wird bei negativem Wert von $s^{(l+1)}$ eine Korrekturaddition notwendig:

$$
rem = \begin{cases} 2^{-(l+1)}\, s^{(l+1)} & : \quad s^{(l+1)} \geq 0 \\ 2^{-(l+1)} (s^{(l+1)} + 2 \cdot d) & : \quad s^{(l+1)} < 0 \end{cases}
\tag{6.8}
$$

Dies entspricht dem Übergang von der Non-Restoring-Methode zur Restoring-Methode der Division im letzten Iterationsschritt.

Abbildung 6.5 zeigt die Implementierung eines Dividierers für 4-Bit-Festkommazahlen nach diesem Schema, einschließlich der Erzeugung des Restwertes *rem*. Man beachte, dass es in den Stufen zur Erzeugung von $q_{-1} \ldots q_{-3}$ nicht notwendig ist, die Vorzeichenbits von $s^{(i)}$ zu verarbeiten. Der Grund dafür ist, dass in jeder Stufe zwei Binärzahlen unterschiedlichen Vorzeichens addiert werden. Ein Carry-Out aus dem Addierer mit Wertigkeit 2^1 würde zu einem Vorzeichenbit identisch Null führen, und zeigt damit bereits ein positives Ergebnis an.

Die Einführung von Pipeline-Registern zur Reduzierung der Logiklaufzeiten kann bei einem solchen Array-Dividierer sehr einfach durch Einfügen solcher Register zwischen die Addier-Subtrahier-Elemente geschehen. Die Pipeline-Register für die Ergebnisse der Addier-Subtrahier-Stufen können effizient in die gleichen Logikelemente für diese arithmetischen Funktionsblöcke gepackt werden und führen deshalb zu keinem erhöhten FPGA-Flächenverbrauch. Dagegen sind die Register, die für die Weiterleitung des Divisors gebraucht werden, nicht mit vorausgehenden

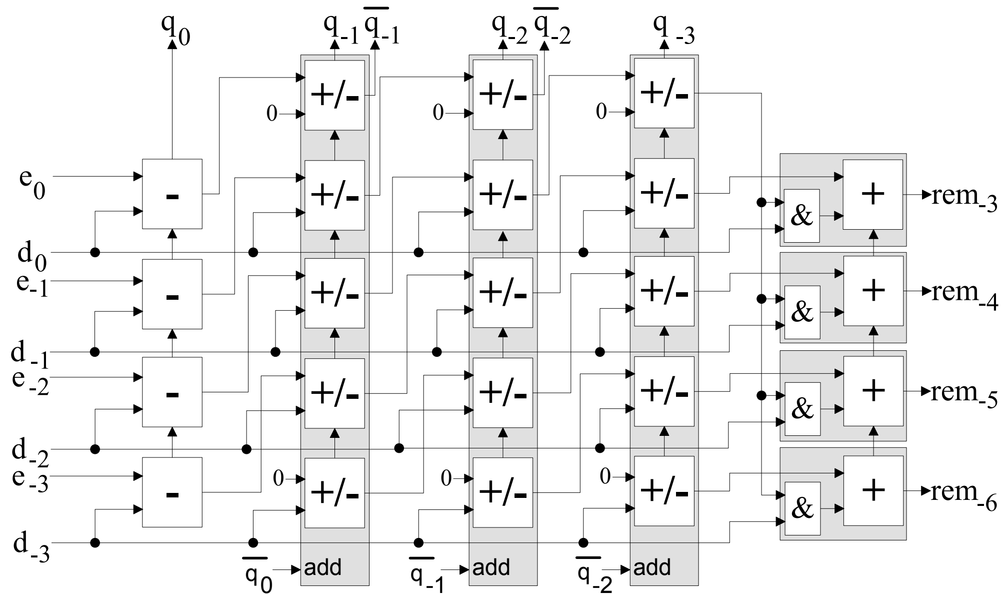


Abbildung 6.5: Dividiererschaltung für 4-Bit-Festkommazahlen mit Addier-Subtrahier-Primitiven (graue Blöcke), $q = e/d, rem = e - q \cdot d$.

Logikblöcken verbunden und benötigen zu deren Implementierung deshalb zusätzliche Logikelemente. Im Fall der Multiplizierer wurde das bei den Array-Multiplizierern in gleicher Weise bestehende Problem, eine ressourceneffiziente Pipeline aufzubauen, dadurch gelöst, dass nur kleine Array-Multiplizierer für die Partialprodukte verwendet und diese über einen Addiererbaum reduziert wurden. Dieser Ansatz ist bei der Division nicht möglich, da die iterative Berechnung des Quotienten nicht wie bei der Multiplikation in mehrere nebenläufig lösbare Teilprobleme (dort war dies die Bestimmung der Partialprodukte) zerlegt werden kann. Trotz ähnlichen Umfangs im Ressourcenverbrauch an arithmetischen Logikbausteinen benötigen deshalb Dividierer wesentlich mehr Logikelemente des FPGAs als Multiplizierer gleicher Operandenbreite.

Der Ressourcenverbrauch und die Geschwindigkeit der so erzeugten Dividierer für einen Virtex-II-FPGA, abhängig von der Breite der Festkommazahlen und der Pipelinetiefe, ist in Tabelle 6.5 aufgestellt. Die Pipelinetiefe ist als Anzahl von Quotientenbits pro Pipelinestufe angegeben und entspricht damit der Anzahl von Addier-Subtrahier-Stufen zwischen zwei Registern. Es ist zu sehen, dass sich für den Fall von Registern nach jeweils zwei Stufen bei akzeptabler Geschwindigkeit ein gutes Verhältnis von Registern zu LUTs ergibt[2]. Durch noch tieferes Pipelining kann zwar die Geschwindigkeit deutlich erhöht werden, der Ressourcenverbrauch wird jedoch aufgrund der hohen Zahl von Registern etwa verdoppelt.

[2]Es sei hier daran erinnert, dass es im FPGA gleich viele 1-Bit-Register und LUTs gibt.

Breite	Quotientenbits pro Pipelinestufe	Register	LUTs	Geschwindigkeit/MHz	Latenz
4	4	4	25	67.0	1
4	2	15	26	98.9	2
4	1	36	26	160.8	4
8	4	8	73	51.5	2
8	2	65	76	91.7	4
8	1	130	79	151.2	8
12	4	74	158	47.2	3
12	2	143	164	85.2	6
12	1	290	167	142.7	12
16	4	133	278	45.2	4
16	2	153	284	80.2	8
16	1	514	287	135.8	16
20	4	196	430	41.7	5
20	2	395	436	76.2	10
20	1	807	441	130.0	15
24	4	281	614	36.6	6
24	2	569	620	72.7	12
24	1	1157	630	125.0	24

Tabelle 6.5: Ressourcenverbrauch und Geschwindigkeit von Festkomma-Dividierern bei Virtex-II-FPGA XC2V3000-4 ohne Erzeugung des Divisionsrestes.

6.1.4 Quadratwurzel

Wie in Abschnitt 4.1.5 herausgestellt wurde, kann die Quadratwurzel nach ähnlichen Schemata wie die Division berechnet werden. Von den verschiedenen Verfahren wurde aus gleichen Gründen wie bei der Division die Non-Restoring-Methode ausgewählt. Es soll nun auch hier von einem Festkommaoperanden, nun x genannt, mit einer Vorkommastelle und l Nachkommastellen ausgegangen werden ($x = x_0.x_{-1}\cdots x_{-l}$). Dann kann die iterative Lösungsmethode für $q = \sqrt{x}$ ähnlich wie Gleichung 4.26 auf folgende Weise aufgestellt werden:

$$
\begin{aligned}
s^{(0)} &= x\,2^{-1} \\
q^{(0)} &= 0 \\
s^{(i)} &= \begin{cases} 4s^{(i-1)} - \left(4q^{(i-1)} + (0\,1)_2\right) & : \ s^{(i-1)} \geq 0 \\ 4s^{(i-1)} + \left(4q^{(i-1)} + (1\,1)_2\right) & : \ s^{(i-1)} < 0 \end{cases} \\
q_{1-i} &= \begin{cases} 1 & : \ s^{(i)} \geq 0 \\ 0 & : \ s^{(i)} < 0. \end{cases} \\
q^{(i)} &= 0\cdots 0\,q_0\cdots q_{1-i} \\
q &= q^{(l+1)}\cdot 2^{-l} = q_0.q_{-1}\cdots q_{-l}
\end{aligned}
\tag{6.9}
$$

Äquivalent dazu ist folgende Formulierung, welche die Verarbeitung der Daten etwas anschaulicher ausdrückt:

$$
\begin{aligned}
s^{(0)} &= 0 \\
s^{(1)} &= ((x_0 x_{-1})_2 - (0\,1)_2) \\
q^{(1)} &= \begin{cases} 1 & : \ s^{(1)} \geq 0 \\ 0 & : \ s^{(1)} < 0 \end{cases} \\
s^{(2)} &= \begin{cases} (s^{(1)} x_{-2} x_{-3})_2 - (q^{(1)}\,0\,1)_2 & : \ s^{(1)} \geq 0 \\ (s^{(1)} x_{-2} x_{-3})_2 + (q^{(1)}\,1\,1)_2 & : \ s^{(1)} < 0 \end{cases} \\
q^{(2)} &= \begin{cases} (q^{(1)}\,1)_2 & : \ s^{(2)} \geq 0 \\ (q^{(1)}\,0)_2 & : \ s^{(2)} < 0 \end{cases} \\
s^{(i)} &= \begin{cases} (s^{(i-1)} x_{-2i+2} x_{-2i+1})_2 - (q^{(i-1)}\,0\,1)_2 & : \ s^{(i-1)} \geq 0 \\ (s^{(i-1)} x_{-2i+2} x_{-2i+1})_2 + (q^{(i-1)}\,1\,1)_2 & : \ s^{(i-1)} < 0 \end{cases} \\
q^{(i)} &= \begin{cases} (q^{(i-1)}\,1)_2 & : \ s^{(i)} \geq 0 \\ (q^{(i-1)}\,0)_2 & : \ s^{(i)} < 0 \end{cases} \\
i &= 3\ldots(l+1) \\
q &= q^{(l+1)}\cdot 2^{-l}
\end{aligned}
\tag{6.10}
$$

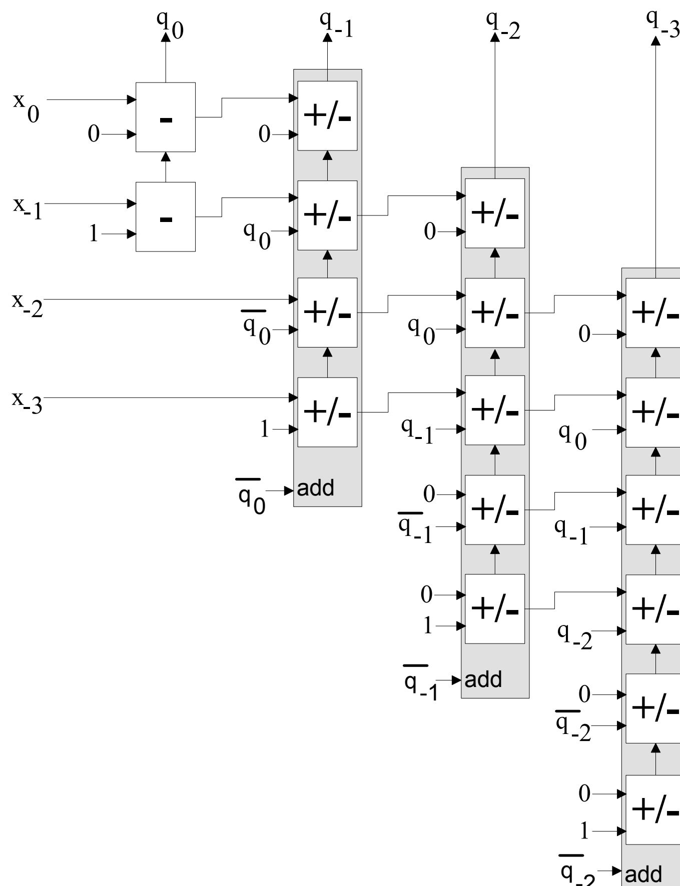


Abbildung 6.6: Schaltung zur Berechnung der Quadratwurzel für 4-Bit-Festkommazahlen mit Addier-Subtrahier-Primitiven (graue Blöcke), $q = \sqrt{x}$.

Abbildung 6.6 zeigt die parallele Implementierung eines Quadratwurzel-Operators für 4-Bit-Zahlen nach diesem Verfahren. Der Ressourcenverbrauch und die Geschwindigkeit der so erzeugten Quadratwurzel-Operatoren für einen Virtex-II-FPGA abhängig von der Breite der Festkommazahlen und der Pipelinetiefe ist in Tabelle 6.6 aufgestellt[3]. Die Pipelinetiefe ist wie bei den Dividiererschaltungen als Anzahl von Ergebnisbits pro Pipelinestufe angegeben und entspricht damit der Anzahl von Addierer/Subtrahierer-Stufen zwischen zwei Registern. Die Geschwindigkeiten sind geringfügig höher als bei den Dividierern. Der Ressourcenverbrauch entspricht etwa 70% des Aufwandes für einen Dividierer gleicher Breite und liegt sogar deutlich unter dem Aufwand für die Multiplizierer (ohne Block-Multiplizierer). Der Bedarf an LUTs entspricht etwa dem Aufwand für die Quadrierer.

[3]Für die erzeugten Operatoren wurde vorausgesetzt, dass eines der führenden beiden Operandenbits den Wert Eins hat, was bei der Verwendung innerhalb eines Gleitkommaoperators immer der Fall ist, wie weiter unten gezeigt wird. Dies führt zu einer Ressourcenersparnis von mehreren LUTs.

Breite	Ergebnisbits pro Pipelinestufe	Register	LUTs	Geschwindigkeit/MHz	Latenz
4	4	0	16	102	0
4	2	9	15	177	2
4	1	12	14	186	4
8	4	18	52	69	2
8	2	39	53	106	4
8	1	67	53	172	8
12	4	42	107	52.7	3
12	2	89	107	92.2	6
12	1	155	109	162	12
16	4	77	177	49.9	4
16	2	159	177	86.6	8
16	1	271	183	153	16
20	4	123	263	47.0	5
20	2	243	265	82.0	10
20	1	419	273	146	20
24	4	177	365	44.5	6
24	2	339	371	78.1	12
24	1	599	379	140	24

Tabelle 6.6: Ressourcenverbrauch und Geschwindigkeit von Implementierungen des Quadratwurzel-Operators für Festkommazahlen bei Virtex-II-FPGA XC2V3000-4.

6.2 Gleitkommaarithmetik auf FPGAs

In dieser Arbeit wird für alle Operanden davon ausgegangen, dass sie normale Zahlen sind, was insbesondere bedeutet, dass deren führendes Mantissebit stets den Wert Eins hat. Wie in Abschnitt 4.3.1 in Tabelle 4.1 illustriert, erlaubt der ANSI/IEEE-754-Standard für Gleitkommazahlen neben der Darstellung normaler Zahlen auch die Spezialfälle $\pm\infty$, Null und NaN (Not a Number). Zugunsten einfacherer Schaltungen wird auf die Spezialfälle verzichtet, da sie für die Auswertung der SPH-Formeln in Simulationsberechnungen keine Bedeutung haben. Es wird also insbesondere auch auf die Darstellung und Verarbeitung von Nullen verzichtet, was keine Einschränkung für die Genauigkeit der zu berechnenden Formeln bedeutet. Tritt bei den Berechnungen ein Wert identisch Null auf, wird statt einer Null die kleinste darstellbare Zahl propagiert. Die physikalisch relevanten Rechenergebnisse (z.B. Dichte, Beschleunigung) werden dadurch in keiner Weise verfälscht.

Im Verlaufe dieses Abschnitts wird detailliert auf die Umsetzung der Gleitkommaoperatoren Addition, Multiplikation, Division und Quadratwurzel auf FPGAs eingegangen. Kern der Implementierungen werden die Festkommaoperatoren sein, wie sie in Abschnitt 6.1 dargestellt wurden. Eine allgemeine Darstellung der Verarbeitung von Gleitkommazahlen wurde in Abschnitt 4.3.3 bereits gegeben. Aufbauend auf diesen Grundlagen werden die Schaltungstechniken zur Implementierung auf FPGAs konkretisiert. Die Implementierungen werden darüber hinaus abhängig von verschiedenen Randbedingungen an die Schaltungen spezialisiert, sodass die FPGA-Ressourcen möglichst effizient eingesetzt werden. Die allen implementierten Operatoren gemeinsame Strategie zur Behandlung von Ausnahmen wird im folgenden Abschnitt diskutiert. In den weiteren Abschnitten zur Implementierung der Operatoren wird die Ausnahmebehandlung dann nicht weiter berücksichtigt, denn es wird davon ausgegangen, dass die Verarbeitungseinheiten so gestaltet sind (insbesondere durch interne Verbreiterung der Exponenten, s.u.), dass in den Rechenwerken keine Ausnahmen entstehen und erst in den *Pack*-Modulen die Ausnahmebehandlung erfolgen muss.

6.2.1 Behandlung von Ausnahmen

Aufgrund der Beschränkung der Operanden auf normale Zahlen reduziert sich die Ausnahmebehandlung für die Gleitkommaoperatoren auf die Erkennung von Overflow- und Underflow-Bedingungen, die sich aus der Verarbeitung der Exponenten ergeben. Auf die Erfassung der Overflow-Situationen wird verzichtet, da der Algorithmus so eingestellt ist, dass keine Überläufe auftreten[4]. Es bleibt also die Behandlung der Underflow-Situationen. Bei der Quadratwurzel können keine Underflow-Ausnahmen auftreten. Bei den übrigen Grundoperationen kommen diese Ausnahmen dagegen relativ häufig vor und können unbehandelt extreme Rechenfehler hervorbringen. Die Rechenwerke können gegen Underflows während der Exponentenverarbeitung abgesichert werden, wenn die Exponenten e zuvor um ein Bit erweitert werden zu $\hat{e}$. Die Erweiterung geschieht mit Berücksichtigung des neuen Bias-Wertes $bias'$ für den breiteren Exponenten.

[4]Dies ist bereits bei der Softwareimplementierung, auf der diese Arbeit aufsetzt, eine zwingende Voraussetzung für das Funktionieren des Algorithmus.

Ist die Breite vor der Erweiterung E Bit und danach $\hat{E} = E + 1$ Bit, ergibt sich:

$$\begin{aligned} \hat{e} &= e - bias + bias' \\ &= e - (2^{E-1} - 1) + 2^{\hat{E}-1} - 1 \\ &= e + 2^{E-1}. \end{aligned}$$

Die Addition braucht nicht durchgeführt zu werden, denn das Bit $[\hat{E} - 1]$ von $\hat{e}$ kann einfach auf den Wert von Bit $[E - 1]$ von e gesetzt werden, das Bit $[\hat{E} - 2]$ auf das Inverse davon; die übrigen Bits entsprechen den Bits von e. Umgekehrt kann eine Verkürzung des Exponenten um ein Bit ausgeführt werden, indem das MSB weggelassen und das Bit $[E - 1]$ invertiert wird. Die Underflow-Bedingung liegt vor, wenn beide führenden Bits des zu verkürzenden Exponenten identisch Null sind (sind diese Bits identisch Eins, liegt ein Überlauf vor). Denn es gibt keinen kürzeren Exponenten, der durch eine Erweiterung um ein Bit in diesen Zustand kommen könnte. Diese Verkürzung der Exponenten auf die ursprüngliche Breite mit Abfangen der Ausnahmen geschieht wie bereits erwähnt in der *Pack*-Einheit der Operatoren und wird im Folgenden stets implizit angenommen.

6.2.2 Implementierung von Gleitkomma-Addierern auf FPGAs

Die Gleitkomma-Addierer sind die komplexesten Operatoren, die in dieser Arbeit behandelt werden. Während der Kern der Operation - die Addition von Festkommazahlen - sehr leicht und effizient in FPGAs umgesetzt werden kann, wie wir in 6.1.1 gesehen haben, ist die *Preparation*- und *Normalization*-Stufe des Operators sehr aufwändig. Wie in Abschnitt 4.3.3 deutlich wurde, sind dazu Verschiebungen von Festkommazahlen um eine variable Anzahl von Stellen notwendig (siehe dazu insbesondere Gleichung 4.36). Wie solche Shift-Operationen implementiert werden, zeigt der folgende Abschnitt. Im daran anschließenden Abschnitt wird kurz auf die für die Normalisierung benötigten Elemente zur Bestimmung der Anzahl aufeinanderfolgender Nullen oder Einsen eingegangen. Daran anschließend folgt die detaillierte Diskussion verschiedener Spezialisierungen von Gleitkomma-Addierern. Die Motivation dazu besteht darin, dass durch die Spezialisierung eine Reduzierung des Ressourcenverbrauchs erreicht werden kann. So reicht es in manchen Fällen aus, Additionen ohne Berücksichtigung von Vorzeichen durchzuführen, beispielsweise wenn Quadrate summiert werden sollen. Zuerst wird die reine Addition von vorzeichenfreien Zahlen beschrieben. Dann wird auf den allgemeinen Fall der Addition von Zahlen mit Vorzeichen eingegangen. Schließlich wird noch ausführlich die Umsetzung von Akkumulatoren für Gleitkommazahlen diskutiert, welche elementare Bausteine für die in dieser Arbeit betrachteten Algorithmen sind. Innerhalb der Abschnitte wird auf weitere Spezialisierungsmöglichkeiten eingegangen, insbesondere die Implementierung der Rundungsmodi.

Shift-Elemente

In diesem Abschnitt wird nun die Implementierung von Verschiebungsbausteinen beschrieben, wie sie für Gleitkomma-Addierer benötigt werden. Die wichtigste Eigenschaft, die diese Elemente erfüllen müssen, ist, einen Bitvektor um eine variable Zahl von Stellen verschieben zu

können. Dazu kommt eine weitere Eigenschaft, die im Zusammenhang mit dem Runden des Resultats der Addition erforderlich wird. Wie später eingehend beschrieben wird, benötigt die Rundungseinheit im Fall des Standard-Rundungsmodus die Information, ob an irgendeiner vorhergehenden Stelle der Schaltung Bits ungleich Null vernachlässigt wurden. Deshalb müssen die Verschiebungsbausteine registrieren, ob durch die Shift-Operation Bits verloren gehen. Das Flag, welches einen solchen Verlust von Bits anzeigt, wird im Folgenden entsprechend dem im Englischen gebräuchlichen Begriff Sticky-Bit genannt.

Das übliche Verfahren, ein Shift-Modul für eine variable Zahl von Verschiebungen aufzubauen, basiert auf der Idee, eine Kaskade von Multiplexern aufzubauen, die in Stufe $i \in [0, n-1]$ eine wahlweise Verschiebung um 2^i Stellen erlauben. Bei n Stufen ergibt sich eine maximale Verschiebung um $2^n - 1$ Stellen. Die Anzahl der Multiplexerstufen skaliert logarithmisch mit der maximalen Verschiebung, wobei sich die Breite dieser Stufen linear zu der Breite der zu verschiebenden Bitvektoren verhält. Mit der Breite k der Bitvektoren skaliert der Ressourcenverbrauch an Multiplexern also mit $k\lceil log_2 k\rceil$. Zur Erzeugung des Sticky-Bits kann bei einem solchen Aufbau eine sehr einfache Schaltung verwendet werden. Alle Signale, die in einer Stufe jenseits des LSBs verschoben werden könnten (2^i Stück), müssen nur mit einer ODER-Verknüpfung versehen und im Fall der Verschiebung mit dem Sticky-Bit aus einer eventuell vorhergehenden Stufe (für $i \geq 1$) über ein ODER-Gatter verknüpft werden. Falls in Stufe i keine Verschiebung erfolgt, wird lediglich das frühere Sticky-Bit weitergeleitet.

Abbildung 6.7 zeigt eine solche Schaltung für die Verschiebung einer 32-Bit-Zahl x um bis zu 31 Stellen (Verschiebungsweite s) mit Ergebnis y und Sticky-Bit t. Soll die Schaltung zusätzlich die Möglichkeit bieten, veranlasst durch ein Signal *zeroOut* alle Ausgangsbits auf Null zu setzten, lässt sich dies ohne zusätzlichen Aufwand an FPGA-Ressourcen erreichen. Dazu können die Multiplexer der letzten Stufe ersetzt werden durch LUTs, welche das Multiplexer-Signal noch mit *zeroOut* über ein UND-Gatter verknüpfen. Da die LUTs für die 2-auf-1-Multiplexer noch einen Eingang frei haben, lässt sich dort *zeroOut* eingeben und in den LUTs das gewünschte Verhalten programmieren. Soll bei $zeroOut = 1$ ein korrektes Sticky-Bit erzeugt werden, müssen die Signale s_0 bis s_4 auf Eins geschaltet sein und t ist bei der vorliegenden Schaltung zusätzlich mit x_{31} über ein ODER-Gatter zu verknüpfen. Es sei hier bereits erwähnt, dass für die Gleitkomma-Addierer der Wert des Sticky-Bits für $zeroOut = 1$ unerheblich ist, da in einem solchen Fall das Rundungsbit nach der Addition in jedem Fall Null ist.

Die Implementierung der Verschiebeelemente geschah durch verhaltensmäßige Beschreibung der Schaltungsstruktur entsprechend Abbildung 6.7, um dem Synthesewerkzeug Optimierungsmöglichkeiten zu bieten. Tabelle 6.7 zeigt den sich ergebenden Ressourcenverbrauch für Shift-Elemente verschiedener Breite, sowohl mit Sticky-Bit-Generierung als auch ohne. Es wird deutlich, dass ein Verschiebebaustein bereits mehr FPGA-Ressourcen benötigt, als ein Addierer gleicher Breite. Die Geschwindigkeit kann im Bedarfsfall leicht durch Einfügen von Pipeline-Registern annähernd ohne Vergrößerung des Flächenverbrauchs erhöht werden. Deshalb sind Shift-Elemente für die Addierer nicht geschwindigkeitslimitierend.

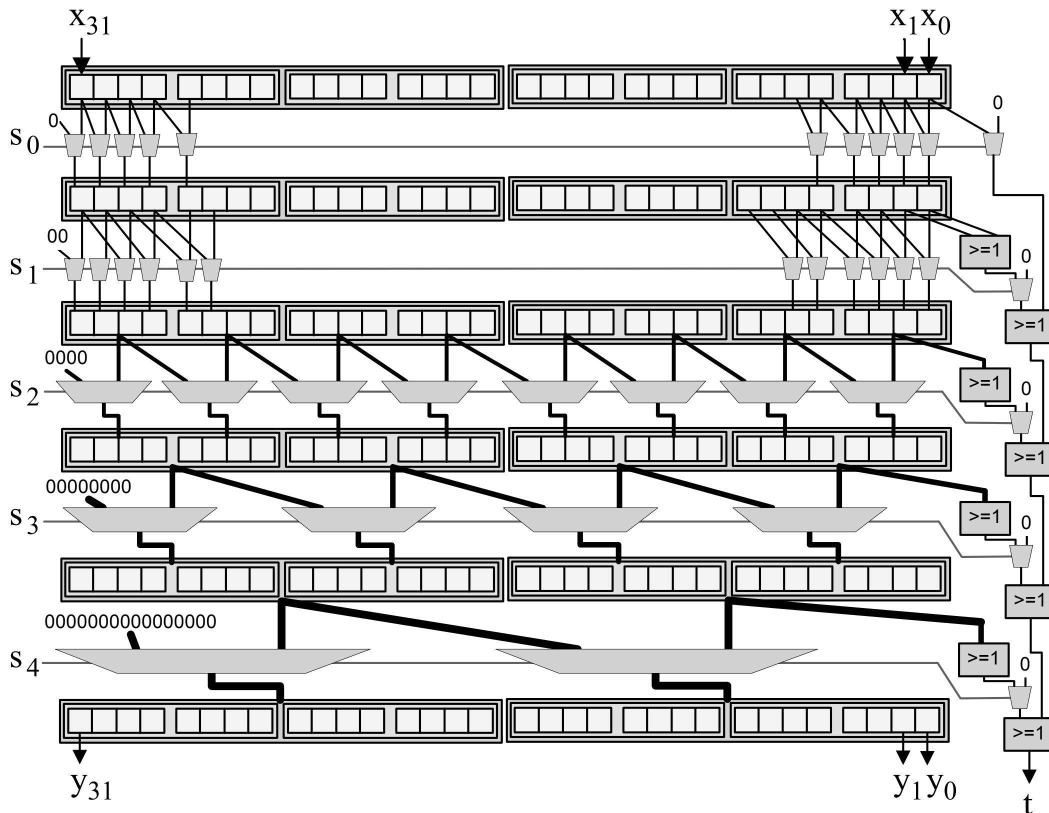


Abbildung 6.7: Schaltung zur Verschiebung eines 32-Bit-Vektors x um Null bis 31 Stellen nach rechts. Das Signal t zeigt an, ob ein Bit ungleich Null herausgeschoben wurde (Sticky-Bit).

	kein Sticky-Bit		mit Sticky-Bit	
Breite	LUTs	Geschwindigkeit/MHz	LUTs	Geschwindigkeit/MHz
8	20	199	26	172
10	30	174	43	141
12	39	161	51	141
14	48	157	60	125
16	55	151	67	136
18	73	139	98	105
20	83	136	107	105
22	96	134	120	105
24	106	133	127	105
26	117	132	139	106

Tabelle 6.7: Ressourcenverbrauch und Geschwindigkeit von Shift-Bausteinen ohne und mit Erzeugung des Sticky-Bits bei Virtex-II-FPGA XC2V3000-4.

	Zählen von Nullen		Nullen oder Einsen	
Breite	LUTs	Geschwindigkeit/MHz	LUTs	Geschwindigkeit/MHz
16	23	211	39	165
20	24	188	35	150
24	32	188	47	164
28	41	162	58	175
32	50	162	75	150
36	60	162	81	139
40	67	162	105	126

Tabelle 6.8: Ressourcenverbrauch und Geschwindigkeit von Bausteinen zum Zählen führender Nullen oder selektiver Zählung von Nullen und Einsen bei Virtex-II-FPGA XC2V3000-4.

Zählen führender Nullen oder Einsen

Sind bei einem Gleitkomma-Addierer die Vorzeichen der Summanden unterschiedlich, so werden die zuvor aufeinander ausgerichteten Mantissen subtrahiert. Dadurch kann es zur Auslöschung von Bits kommen, wie bereits in Abschnitt 4.3.3 beschrieben. Diese Auslöschung äußert sich bei positivem Ergebnis der Subtraktion in einer Reihe führender Nullen. Ist das Ergebnis negativ, tritt dagegen eine Reihe führender Einsen auf. Die Anzahl dieser führenden Bits mit gleichem Wert ist zu zählen, um die Weite der bei der Normalisierung erforderlichen Verschiebung nach links festzustellen. Die Module, die diese Aufgabe erfüllen, wurden in VHDL verhaltensmäßig beschrieben, wobei die Beschreibung bezüglich des Synthesewerkzeuges so weit optimiert wurde, dass die Implementierung vergleichbar effizient wie bei einer handoptimierten strukturellen Implementierung wurde. Tabelle 6.8 zeigt die erzielten Ergebnisse. Es wird deutlich, dass die Zählschaltung für führende Nullen bereits etwa 1.5-mal so viele FPGA-Ressourcen erfordert wie ein Addierer für Bitvektoren gleicher Breite. Sollen neben Nullen auch Einsen gezählt werden können, erhöht sich der Ressourcenverbrauch noch einmal um etwa 50 %.

Addition vorzeichenloser Zahlen

Dieser und die folgenden Abschnitte zur Implementierung von Addiererschaltungen sind wie der Grundlagenabschnitt 4.3.3 zu Gleitkomma-Addierern gegliedert in *Preparation*, *Operation* und *Normalization*. Dies soll es erleichtern, die Gemeinsamkeiten und Unterschiede der Implementierungen darzustellen.

Die Addition von vorzeichenlosen Zahlen unterscheidet sich vom allgemeinen Fall vor allem dadurch, dass im Festkomma-Addierer der *Operation*-Stufe keine Auslöschung von führenden Bits mit einhergehendem Verlust an Genauigkeit des Ergebnisses stattfinden kann. Für den Ressourcenverbrauch der Implementierung bedeutet dies insbesondere, dass das aufwändige Shift-Modul der Normalisierung entfallen kann.

Zur Erinnerung der Nomenklatur sei hier anlehnend an Gleichung 4.36 aus Abschnitt 4.3.3

die Formulierung der Addition positiver Gleitkommazahlen gegeben:

$$\begin{aligned}\overbrace{s_{sum}\,2^{e_{sum}-bias}}^{A+B} &= \overbrace{\left(s_1\,2^{e_1-bias}\right)}^{A} + \overbrace{\left(s_2\,2^{e_2-bias}\right)}^{B} \\ &= \underbrace{\left(s_1 + s_2\,2^{e_2-e_1}\right)}_{\text{Festkomma-Operator} \rightarrow s_{add}} \quad 2^{e_1-bias}.\end{aligned} \tag{6.11}$$

Preparation Die *Unpack*-Stufe aus Abbildung 4.21 ist bei allen Gleitkommaoperationen identisch und soll an dieser Stelle nur einmal kurz beschrieben werden. Einzige Aufgabe dieser Stufe ist, die eingehenden Gleitkommazahl-Operanden, welche hier entsprechend der Nomenklatur von Abschnitt 4.3.1 $A = (s^A, e^A, f^A)$ und $B = (s^B, e^B, f^B)$ genannt werden, in Vorzeichen $sign(A)$ und $sign(B)$ (das erübrigt sich hier), Mantissen s_1 und s_2 und Exponenten e_1 und e_2 zu zerlegen. Die Exponenten liegen in der Gleitkommadarstellung bereits in Ganzzahldarstellung (Breite E Bit) mit Bias $2^{E-1} - 1$ vor. Lediglich die als Fraction gegebenen Werte $f^A = f^A_{M-2} \dots f^A_0$ und $f^B = f^B_{M-2} \dots f^B_0$ müssen durch Ergänzen eines führenden Bits mit Wert Eins zu den Mantissen s_1 und s_2 ergänzt werden.

Der erste Schritt bei jeder Addition besteht darin, die Differenz $e_{12} = e_1 - e_2$ der Exponenten der eingehenden Operanden festzustellen. Daraus wird die Anzahl d der Stellen, um die eine der Mantissen vor der Festkomma-Addition verschoben werden muss und die Auswahl des vorläufigen Ergebnisexponenten e'_1 bestimmt. Ist e_{12} negativ, wird $e'_1 = e_2$, ansonsten $e'_1 = e_1$ gesetzt. Außerdem werden für $e_{12} < 0$ die Mantissen vertauscht, damit das Verschiebemodul nur für eine Mantisse implementiert werden muss. Der Wert von d wird dann durch den Betrag von e_{12} festgelegt. Im Allgemeinen ist die Breite E von $|e_{12}|$ größer als die Breite von d, welche durch $S = \lceil log_2(M+1) \rceil$ gegeben ist (für diese Arbeit wurde $S = 5$ festgesetzt). Deshalb ist der Wert der Verschiebung durch $d = min(2^S - 1, |e_{12}|)$ zu berechnen. Für den Aufbau einer Schaltung, welche die beschriebene Verarbeitung der Exponenten übernimmt, sind zwei Addierer zur Berechnung der Differenz e_{12} und des Betrags $|e_{12}|$ und zusätzliche Logik zur Auswertung der Überlaufbedingung für d erforderlich. Abbildung 6.8 zeigt die Implementierung einer solchen Schaltung[5].

Abbildung 6.9 zeigt die Gesamtschaltung eines Gleitkommazahl-Addierers als Blockschaltbild. Im durch das Rechteck *Preparation* umrandeten Teil ist das Modul *Prepare Exponents* zu sehen, welches der eben beschriebenen Schaltung entspricht. Das Signal *swap* weist die Einheit *Selective Swap* an, die Mantissen zu vertauschen. Dieses Modul besteht lediglich aus zwei Multiplexern. Um Ressourcen zu sparen, wird die Ergänzung der $(M-1)$-Bit-Signale f_1 und f_2 zu den Mantissen erst nach diesem Baustein durchgeführt. Des Weiteren wird ein Rundungsbit an die Mantissen angehängt, sodass Festkommazahlen mit $M+1$ Stellen resultieren. Bei der Mantisse, die durch ein Shift-Element verschoben werden kann, wird das führende Bit nur dann als Eins

[5]Genauso hätte die Bildung des 2er-Komplements über einen Addier-Subtrahier-Baustein geschehen können, mit einfacherer Feststellung des Überlaufs und Maximierung von d im Fall des Überlaufs durch ODER-Gatter. Diese Variante würde jedoch zu mehr Ressourcenverbrauch führen, da die vorgestellte Implementierung dem Synthesewerkzeug mehr Optimierungsmöglichkeiten bietet.

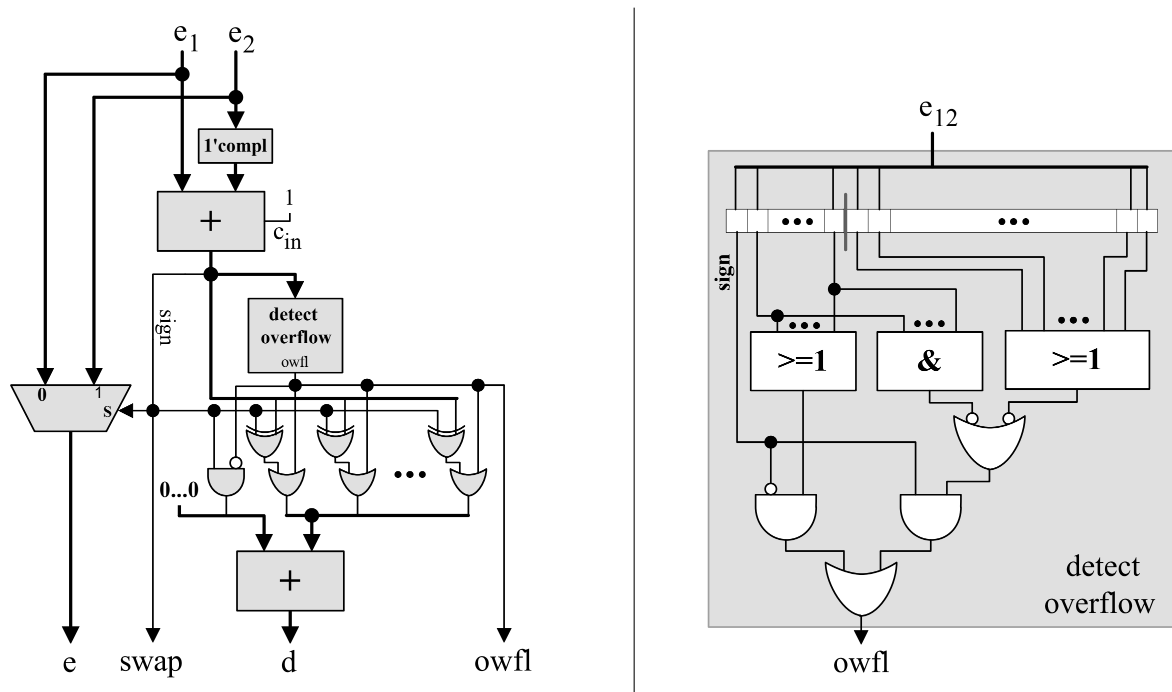


Abbildung 6.8: Schaltung zur Verarbeitung der Exponenten zur Bestimmung der Verschiebungsweite d und Auswahl des größeren der Exponenten e. Auf der rechten Seite ist die Schaltung zur Bestimmung des Überlaufs *owfl* bei Reduktion der Signed-Integer-Zahl e_{12} auf den Betrag dieser Zahl mit reduzierter Breite (Anzahl der Bits rechts vom grauen Balken) abgebildet.

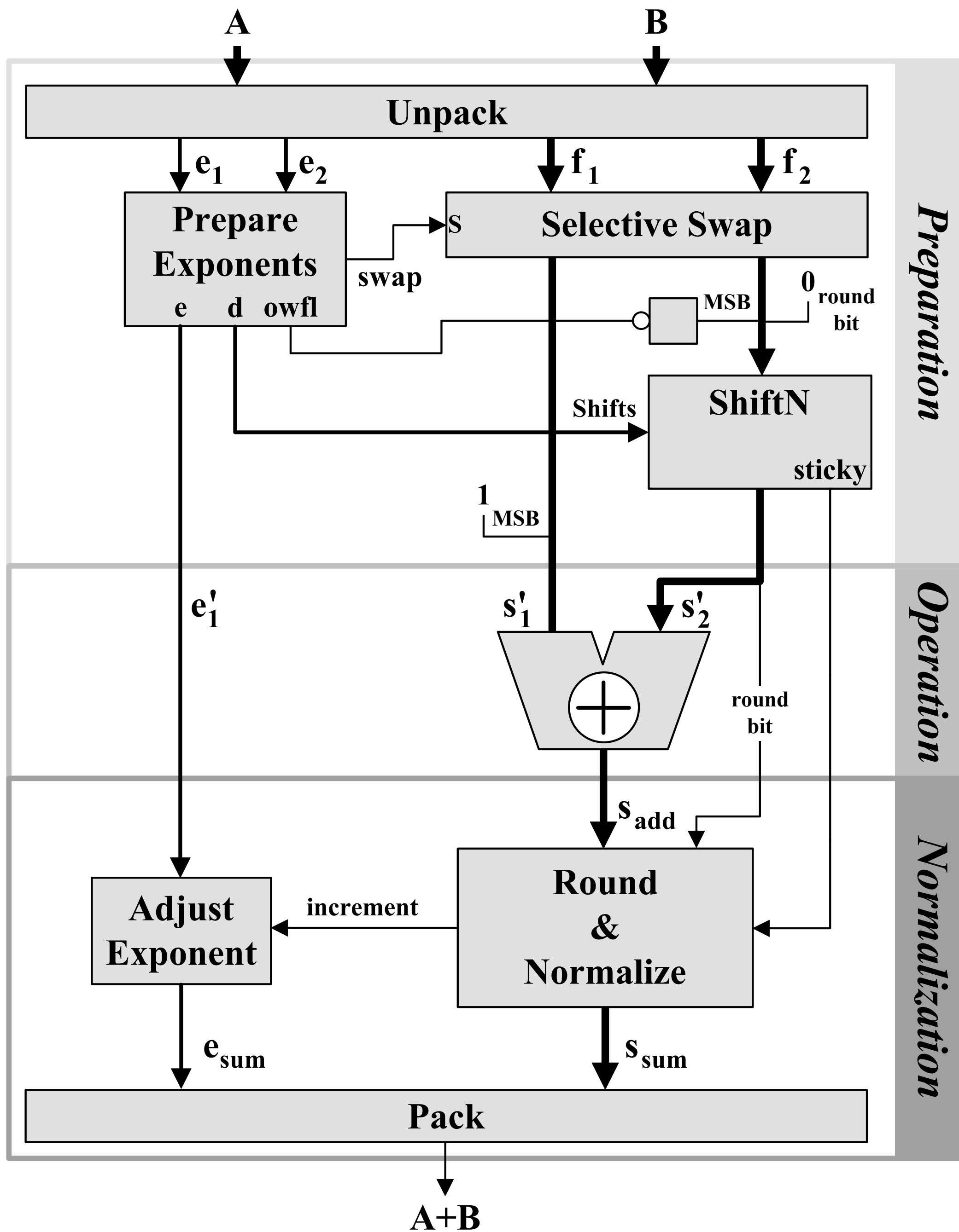


Abbildung 6.9: Blockschaltbild für einen Gleitkommazahl-Addierer für positive Zahlen.

ergänzt, wenn von *Prepare Exponents* kein Überlauf (Flag *owfl*) von der Verschiebeweite d signalisiert wird. Dadurch kann das im letzten Abschnitt beschriebene zusätzliche Signal *zeroOut* für das Verschiebemodul selbst für die maximal (für $S = 5$) erlaubte Mantissenbreite von $M = 31$ entfallen. Die *Preparation*-Stufe wird komplettiert durch den Verschiebebaustein *ShiftN*, welcher den eingehenden $(M+1)$-Bit-Vektor um maximal 31 Stellen verschieben kann. Für $M = 31$ und $owfl = 1$ ist das resultierende Sticky-Bit aufgrund der Maskierung des MSB zwar nicht korrekt, da das Rundungsbit in diesem Falle aber Null ist, spielt das für das Standardrundungsverfahren keine Rolle.

Operation Da die Auswahl des größeren der eingehenden Exponenten bereits in der *Preparation*-Stufe erledigt wurde, bleibt für die *Operation*-Stufe nur noch die Addition der Festkommazahlen s_1' und s_2'. Das Rundungsbit muss nicht in die Addition einbezogen werden. Deshalb genügt ein M-Bit-Addierer mit $(M+1)$-Bit-Ergebnis. Es wird ein einfacher Ripple-Carry-Addierer verwendet, der (wie in Abschnitt 6.1.1 diskutiert) für die hier vorliegenden Bitbreiten die effizienteste Implementierungsvariante ist.

Normalization Während im Fall der allgemeinen Addition von Gleitkommazahlen nach Formel 4.38 der Betrag des Ergebnisses der Festkommazahladdition im Intervall $[0,4)$ liegt, bleibt das Additionsresultat s_{add} für positive Gleitkommazahlen im Intervall $[1,4)$. Die Normalisierung kann also lediglich eine Verschiebung von s_{add} um eine Stelle nach rechts erforderlich machen - Linksverschiebungen sind nicht nötig. Ebenso kann nach dem Runden eine Verschiebung um eine Stelle nach rechts notwendig werden, wenn $s_{add} \in [1,2)$ gilt, durch das Runden jedoch eine Zahl größer oder gleich 2 entsteht. Niemals ist jedoch eine zweimalige Verschiebung notwendig, wovon man sich durch die Betrachtung der Addition der größtmöglichen Mantissen überzeugen kann:

$$\begin{array}{rcl} s_{max} & = & 1.\ 1\ 1\ \ldots\ 1\ 1\ \mid \underbrace{0}_{\text{Rundungsbit}} \\ s_{add,max} = s_{max} + s_{max} & = & 1\ 1.\ 1\ 1\ \ldots\ 1\ 0\ \mid 0 \\ s_{norm,max} = norm(s_{add,max}) & = & 0\ 1.\ 1\ 1\ \ldots\ 1\ 1\ \mid 0 \\ s_{sum,max} \leq s_{norm,max} + \frac{1}{2}\,ulp & = & 0\ 1.\ 1\ 1\ \ldots\ 1\ 1\ \mid 0 \\ & & +\ 0.\ 0\ 0\ \ldots\ 0\ 0\ \mid 1 \\ & = & 0\ 1.\ 1\ 1\ \ldots\ 1\ 1\ \mid 1 \end{array} \tag{6.12}$$

Deshalb können die beiden *Normalize*-Stufen und die Rundungsstufe aus Abbildung 4.21 zu einer Stufe zusammengefasst werden, die lediglich aus einem Addierer für das Runden, einem nachgeschalteten 2-auf-1-Multiplexer und etwas Logik zur Bestimmung des für das Runden zu addierenden Betrags besteht. Letztere wertet das führende und die letzten zwei Bits von s_{add}, das Rundungsbit und das Sticky-Bit aus, um festzustellen, ob zu einem der letzten beiden Bits von s_{add} eine Eins addiert werden muss.

Das Sticky-Bit liefert die Information, ob das exakte Additionsresultat über das LSB von s_{add} hinaus noch Bits ungleich Null hätte. Diese Information wird insbesondere für den Standardrundungsmodus *Round To Nearest Even* benötigt, da dann an der Schwelle zwischen Auf- und

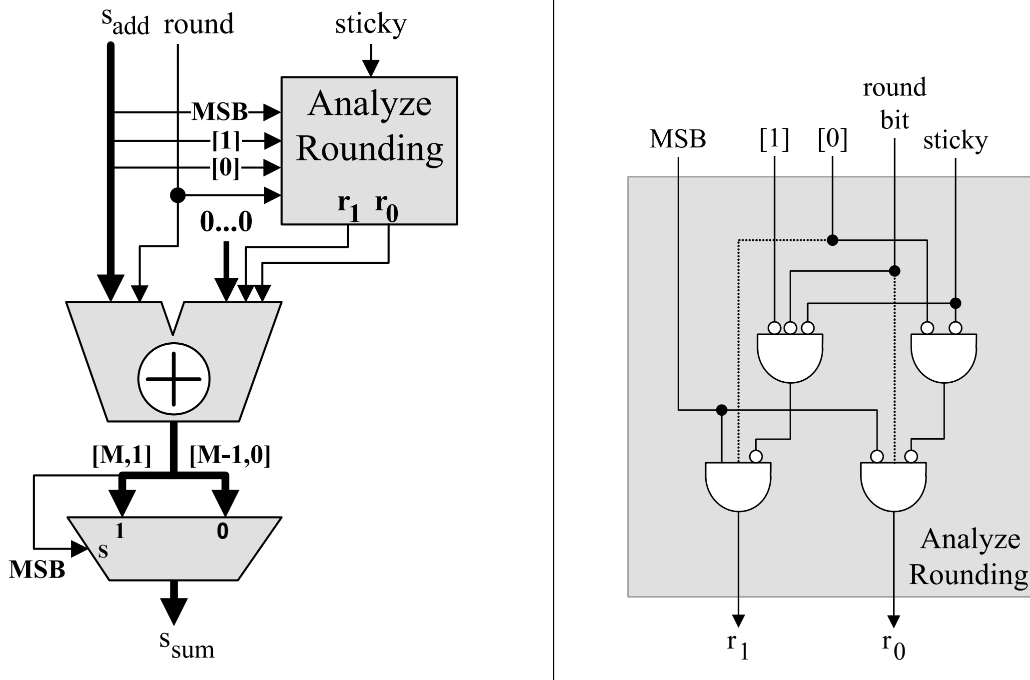


Abbildung 6.10: Schaltung zur Rundung und Normalisierung des Resultats für die Addition von positiven Gleitkommazahlen. Rechts ist die Logik zur Umsetzung des Standard-Rundungsmodus *Round To Nearest Even* abgebildet. Die gepunkteten Verbindungen werden für die Schaltung nach der linken Abbildung nicht benötigt, erleichtern aber das Verständnis.

Abrunden so gerundet wird, dass das LSB der Ergebnismantisse Null wird. Dies macht es erforderlich, zu bestimmen, ob s_{add} exakt auf der Rundungsschwelle liegt. Dies ist genau dann der Fall, wenn das Rundungsbit identisch Eins ist und alle darüber hinausgehenden Bits des exakten Additionsresultats identisch Null sind. Letztere Information wird gerade durch das Sticky-Bit angezeigt.

Die Schaltung zur Rundung und Normalisierung ist in Abbildung 6.10 gezeigt und entspricht dem Block *Round & Normalize* aus Abbildung 6.9. Das *MSB* in Abbildung 6.10 zeigt gleichzeitig an, ob der vorläufige Exponent e_1' inkrementiert werden muss, um den Exponenten des Resultats e_{sum} zu erhalten. Diese Anpassung des Exponenten wird im Block *Adjust Exponent* aus Abbildung 6.9 durchgeführt.

Ressourcenverbrauch und Geschwindigkeit In Tabelle 6.9 sind die Ergebnisse für den Ressourcenverbrauch der Gleitkomma-Addierer für positive Zahlen aufgestellt. Die Ergebnisse sind aufgeschlüsselt bezüglich der Anzahl an 4-Input-LUTs und 1-Bit-Registern nach der Synthese durch Synplify und der Anzahl an Virtex-II-Slices und der maximale Taktfrequenz, bestimmt durch die Implementierung des Designs auf einem Virtex-II-FPGA (XC2V3000, Speed Grade -4). Die im weiteren Verlauf dieses Kapitels angegebenen Zahlen zu Ressourcen wurden auf die gleiche Weise ermittelt. In Abbildung 6.11 ist der Ressourcenverbrauch grafisch in Abhängigkeit von der Breite der Mantisse dargestellt. Man erkennt ein nahezu lineares Ansteigen aller Grö-

Round To Nearest Integer					Round To Nearest Even				
Mantisse Bits	LUTs	Regs	Slices	Design Freq. (MHz)	Mantisse Bits	LUTs	Regs	Slices	Design Freq. (MHz)
12	147	91	81	91	12	157	92	88	89
14	167	103	91	91	14	177	104	99	87
16	187	115	102	88	16	211	117	117	90
18	213	128	116	89	18	230	129	126	87
20	232	140	128	94	20	253	141	141	87
22	253	152	139	90	22	273	153	151	86
24	276	164	152	90	24	296	165	165	81

Tabelle 6.9: Ressourcenverbrauch und Geschwindigkeit von Gleitkomma-Addierern für positive Zahlen bei Virtex-II-FPGA XC2V3000-4.

ßen mit der Mantissenbreite. Die zusätzliche logarithmische Komponente der Skalierung, die sich aus der Implementierung der Shifter-Elemente ergibt, ist an dem Sprung an der Anzahl von LUTs zwischen Bitbreite 14 und 16 zu sehen.

Die Geschwindigkeit der Operatoren ist durch die Verarbeitung der Exponenten im Modul *Prepare Exponents* begrenzt. Durch tieferes Pipelining kann die Geschwindigkeit noch stark erhöht werden. Dann werden jedoch auch zusätzliche, nicht mit vorangehenden LUTs gepaarte Registerstufen für die Mantissen notwendig. Dies würde zu einem Anstieg an aufzuwendenden Slices führen. Ohnehin werden die Festkomma-Dividierer und -Quadratwurzeloperatoren, wie in Abschnitt 6.1 gezeigt, die Geschwindigkeit der Gesamtschaltung limitieren. Damit erübrigt sich eine Optimierung durch tieferes Pipelining der Addierer.

Addition beliebiger Gleitkommazahlen

In Abschnitt 4.3.3 wurden die Verarbeitungsschritte der Gleitkommazahl-Addition anhand folgender Gleichung motiviert:

$$\begin{aligned}\overbrace{\pm s_{sum}\, 2^{e_{sum}-bias}}^{A+B} &= \overbrace{\left(\pm s_1\, 2^{e_1-bias}\right)}^{A} + \overbrace{\left(\pm s_2\, 2^{e_2-bias}\right)}^{B} \\ &= \pm \underbrace{\left(\pm s_1 + s_2\, 2^{e_2-e_1}\right)}_{\text{Festkomma-Operator} \rightarrow s_{add}} 2^{e_1-bias}.\end{aligned} \tag{6.13}$$

Bezugnehmend auf den letzten Abschnitt wird im Folgenden nun detailliert auf die Hardwareimplementierung dieser Berechnung eingegangen.

Preparation Die Vorverarbeitung der Exponenten ist im allgemeinen Fall der Addition von Gleitkommazahlen A und B mit beliebigem Vorzeichen identisch mit dem Verfahren, das im letzten Abschnitt beschrieben wurde und die dort in Abbildung 6.8 gezeigte Schaltung kann

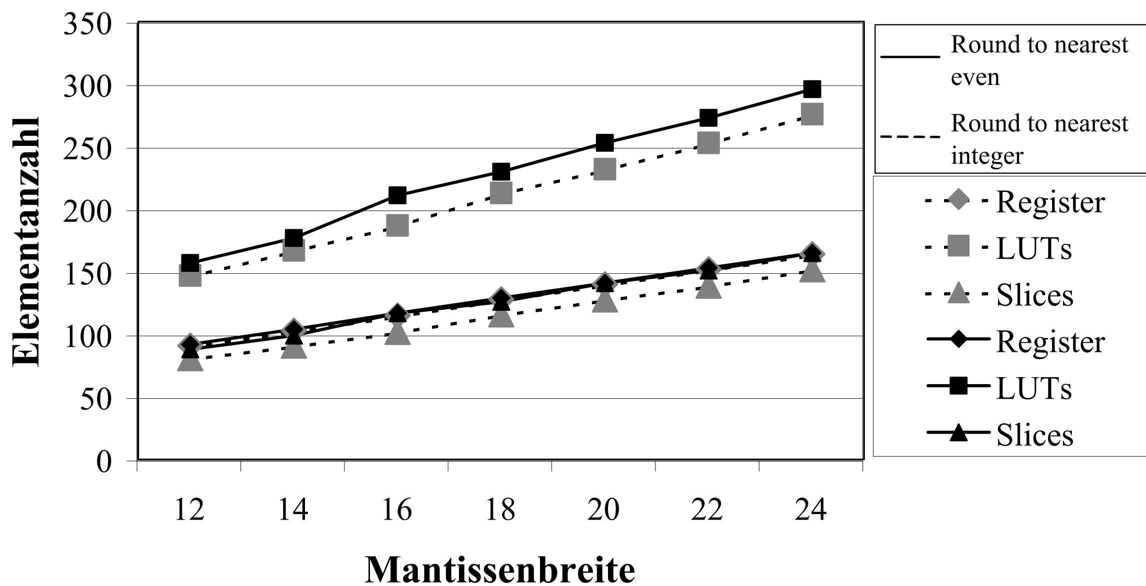


Abbildung 6.11: Verlauf des Ressourcenverbrauchs für die Gleitkomma-Addierer für positive Zahlen nach Tabelle 6.9.

auch hier unverändert übernommen werden. Genau wie dort werden die Mantissen vertauscht, wenn $e_2 > e_1$ gilt, damit das Shift-Modul nur für eines der Argumente implementiert werden muss. Hinzu kommt nun jedoch die Verarbeitung der Vorzeichen $sign(A)$ und $sign(B)$ und die Komplementbildung einer der Mantissen, falls die Vorzeichen unterschiedlich sind. Die Komplementbildung wird nur für die Mantisse aufgebaut, die nicht verschoben wird. Deshalb kann als vorläufiges Vorzeichen der Summe das Vorzeichen der in das Verschiebeelement geleiteten Zahl genommen werden. Dieses Vorzeichen soll hier $sign_2'$ genannt werden. Die Bildung des Komplements kann ohne zusätzlichen Aufwand an FPGA-Ressourcen als 1er-Komplement im Vertauschungsmodul implementiert werden. Dazu sind lediglich die LUTs für den Multiplexer für s_1' mit einem zusätzlichen Eingang *compl* zu versehen, welcher eine Negierung aller Ausgangssignale veranlasst. Die Addition von Eins zur Bildung des 2er-Komplements kann auf den Additionsvorgang von s_1' und s_2' verschoben werden.

Für ein im *Normalization*-Schritt durchzuführendes korrektes Runden nach dem Standard-Verfahren müssen die Mantissen am niederwertigen Ende um zwei Bits erweitert werden, ein Guard- und ein Round-Bit. Das Sticky-Bit wird ebenfalls benötigt. Die Begründung dafür erfolgt in der Beschreibung zur *Normalization*-Stufe. Die Erweiterung geschieht für s_1' durch Anfügen von Einsen, wenn eine Komplementbildung erfolgte, ansonsten durch Nullen. Wie wir später sehen werden, brauchen Round- und Guard-Bit für s_1' nicht weiterverarbeitet zu werden - es erleichtert aber das Verständnis der Schaltung, die Erweiterung anzunehmen. Für s_2' werden bereits vor dem Shift-Modul am niederwertigen Ende zwei Nullen angehängt. Da in der nachfolgenden *Operation*-Stufe Festkommazahlen in Komplementdarstellung verarbeitet werden, müssen die

Mantissen am höherwertigen Ende um zwei Stellen erweitert werden. Für s'_2 genügt es, Nullen voranzustellen, für s'_1 ergeben sich die voranzustellenden Bits aus dem Signal *compl*, welches die Komplementbildung veranlasst.

In Abbildung 6.12 ist die Gesamtschaltung des Gleitkomma-Addierers gezeigt und man vergleiche die gezeigte *Preparation*-Stufe mit der Schaltung aus Abbildung 6.9. Der Ressourcenverbrauch für die *Preparation*-Stufe ist nur geringfügig höher als für den Addierer für positive Zahlen. Der Mehrverbrauch entsteht vor allem im Modul *ShiftN*, da ein um eine Stelle breiterer Bitvektor verschoben werden muss.

Operation Diese Stufe wird wie beim Addierer für positive Zahlen durch die Anwendung eines einfachen Ripple-Carry-Addierers als Festkommazahl-Addierer implementiert. Die zusätzliche Addition eines LSB für den Fall, dass s'_1 eine 1er-Komplement-Zahl ist, wird durch ein Carry-In in den Addierer realisiert. Sowohl Sticky-Bit als auch Guard- und Round-Bit brauchen nicht in die Addition einbezogen zu werden. Diese Bits würde sich durch die Addition nicht verändern, da einerseits für $compl = 0$ nur Nullen addiert würden, andererseits für $compl = 1$ die LSBs 111 plus dem Carry-In addiert würden, was durch das Carry-In in den um diese Bits gekürzten Addierer erledigt werden kann.

Abhängig vom Vorzeichen der sich nach der Addition ergebenden Festkommazahl in 2er-Komplement-Darstellung ist das vorläufige Vorzeichen $sign'_2$ zu invertieren. Dies hätte man auch der *Normalization*-Stufe zuordnen können.

Normalization In dieser Stufe ist nun die 2er-Komplement-Festkommazahl s_{add} (einschließlich Guard-Bit g und Round-Bit r) zu einer normalisierten und korrekt gerundeten Mantisse zu überführen. Wie in Abschnitt 4.3.3 beschrieben, kann die Normalisierung bereits in der 2er-Komplement-Darstellung von s_{add} geschehen. Dies hat den Vorteil, dass der Addierer, der für die Umwandlung von s_{add} in den Betrag davon benötigt würde, entfallen kann. Die Betragsbildung kann mit der Rundungsschaltung, welche der Normalisierung folgt, verbunden werden.

Das führende Bit von s_{add} entspricht dem Vorzeichen und soll $sign_{add}$ genannt werden. Die Zahl s_{add} hat dann folgende Darstellung:

$$s_{add} = sign_{add}\, y_1 y_0 . y_{-1} \cdots y_{-M+1}\, g\, r. \tag{6.14}$$

Für s_{add} gilt, wie bereits in Gleichung 4.38 festgestellt wurde:

$$\begin{aligned} s_{add} &\in (-2,4) \\ |s_{add}| &\in [0,4). \end{aligned} \tag{6.15}$$

Deshalb kann bei positivem s_{add} eine Rechtsverschiebung um eine Stelle oder unabhängig vom Vorzeichen von s_{add} eine Linksverschiebung um die volle Breite von s_{add} erforderlich werden. Für $sign_{add} = 0$ gibt $y_1 = 1$ eine Rechtsverschiebung um eine Stelle vor. Für $y_1 = 0$ gibt die Anzahl der führenden Nullen in $y_0 . y_{-1} \cdots y_{-M+1}\, g\, r$ die Zahl der Stellen an, um die linksverschoben werden muss. Für $sign_{add} = 1$ sind die führenden Einsen in $y_0 . y_{-1} \cdots y_{-M+1}\, g\, r$ zu zählen, um die Weite der Linksverschiebung zu bestimmen. Damit ein Shift-Modul verwendet werden

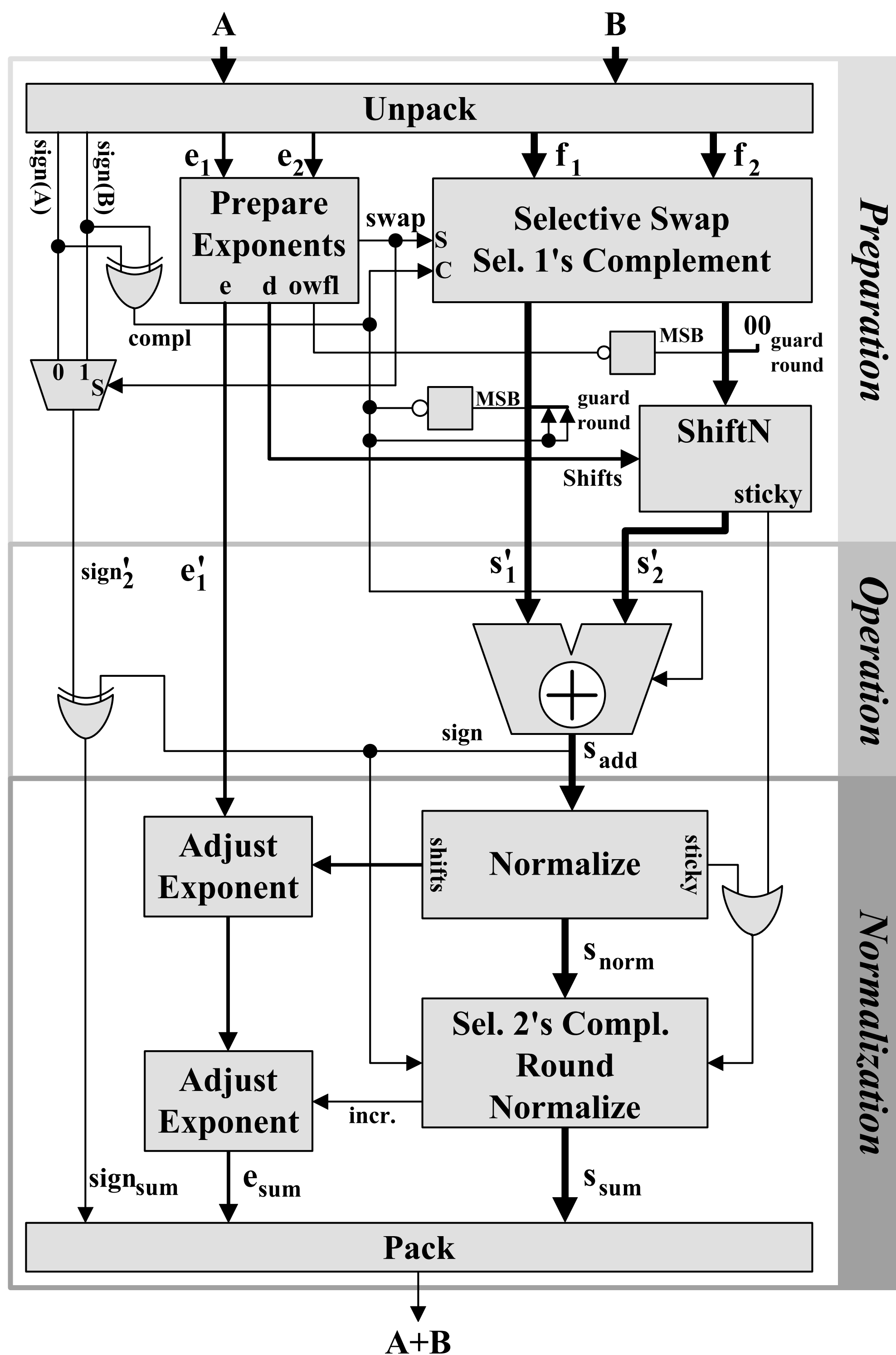


Abbildung 6.12: Blockschaltbild für einen Gleitkommazahl-Addierer für beliebige Gleitkommazahlen.

kann, das eine Verschiebung in nur eine Richtung durchführt, wird ein leicht abgewandeltes Verfahren angewandt. Die Zahl s_{add} wird in jedem Fall um eine Stelle rechtsverschoben zur Zahl $s'_{add} = sign_{add}\, y_1.y_0 \cdots y_{-M+1}\, g\, r$, was keinerlei Hardwareressourcen braucht. Danach wird durch die Anzahl der führenden Nullen für $sign_{add} = 0$ oder Einsen für $sign_{add} = 1$ in $y_1.y_0 \cdots y_{-M+1}\, g\, r$ die Weite *pShifts* der Linksverschiebung bestimmt. Als Resultat der Normalisierung ergibt sich die $(M+2)$-Bit-Zahl $s_{norm} = sign_{add}\, y'_0.y'_{-1} \cdots y'_{-M+1}\, r'$ mit dem neuen Rundungsbit r', also eine um zwei Bits kürzere Zahl als s_{add}. Aus den am niederwertigen Ende herausgefallenen Bits und dem alten Sticky-Bit wird über eine ODER-Verknüfung das neue Sticky-Bit gewonnen. Die Detailschaltung der ersten Normalisierungsstufe zeigt Abbildung 6.13. Entsprechend den Verschiebungen der Mantisse bei der Normalisierung muss der vorläufige Exponent e'_1 angepasst werden. Dazu ist von e'_1 die Zahl *pShifts* zu subtrahieren und aufgrund der vorausgegangenen Rechtsverschiebung eine 1 zu addieren.

Von der normierten Zahl s_{norm} muss nun der gerundete Betrag gebildet werden. Das Runden geschieht auf ähnliche Weise wie beim Addierer für positive Zahlen. Hier ist zusätzlich noch das 2er-Komplement zur Erzeugung des Betrags notwendig, falls s_{norm} negativ ist. Wird zum Runden anstatt des Addierers ein Addier-Subtrahier-Baustein verwendet, kann diese Komplementbildung ohne weitere Hardwareressourcen gemeinsam mit der Rundungsaddition geschehen. Die Schaltung für das Modul *Selective 2's Complement & Round & Normalize* des Addierers ist für das Standardrundungsverfahren in Abbildung 6.14 gezeigt. Die Schaltung *Analyze Rounding* benötigt wie bei der Addition ohne Vorzeichen nur 2 LUTs des FPGAs. Der Multiplexer zur abschließenden Normalisierung im Fall eines Überlaufs durch das Runden kann hier entfallen. Der Grund dafür ist, dass ein Übertrag auf das Bit mit Wertigkeit 2^1 nur stattfinden kann, wenn vor der Rundungsaddition alle Bits von der Wertigkeit 2^0 bis einschließlich des Rundungsbits den Wert Eins und folglich danach alle den Wert Null haben. Das MSB der normierten Mantisse ausgenommen, welches immer Eins ist und deswegen in der *Pack*-Stufe entfällt, haben die anderen Mantissebits, egal ob um eine Stelle rechtsverschoben oder nicht, bereits den richtigen Wert. Es muss also lediglich anhand des Bits mit der Wertigkeit 2^1 der Exponent des Ergebnisses korrigiert werden, was durch eine selektive Inkrementierung in der zweiten *Adjust-Exponent*-Einheit geschieht. Die beiden *Adjust-Exponent*-Module hätten mit Ersparnis eines Addierers zusammengefasst werden können. Da die Module auf verschiedenen Tiefen in der Pipeline liegen und eine Zusammenlegung deshalb zusätzliche Register erfordert hätte, welche die Ersparnis wieder aufwiegen, wurden sie getrennt implementiert.

Ressourcenverbrauch und Geschwindigkeit Die Ergebnisse zum Ressourcenverbrauch für die Addierer in Abhängigkeit von der Mantissenbreite und die erzielte Geschwindigkeit sind in Tabelle 6.10 und in Abbildung 6.15 aufgetragen. Es ergibt sich ein ähnlicher Verlauf wie im Fall des Addierers für positive Zahlen, allerdings auf einem um etwa 70% höheren Niveau. Der logarithmische Anteil der Skalierung der Shifter-Elemente zeigt sich hier bereits am überproportionalen Anstieg der LUT-Anzahl zwischen der Bitbreite 12 und 14, da ein zusätzliches Guard-Bit verschoben werden muss.

Für die Geschwindigkeit ergeben sich geringfügig schlechtere Werte als bei den Addierern für vorzeichenlose Zahlen. Auch hier limitiert die Schaltung für die Verarbeitung der Exponenten

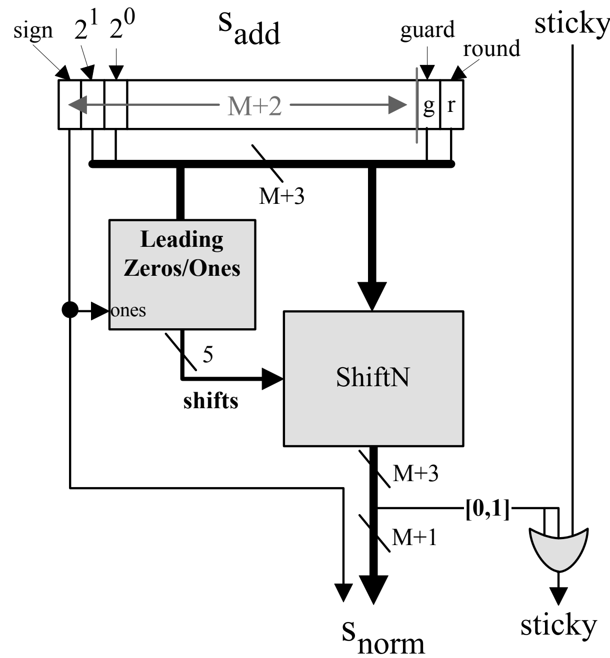


Abbildung 6.13: Normalisierung der Mantisse beim Gleitkommazahl-Addierer.

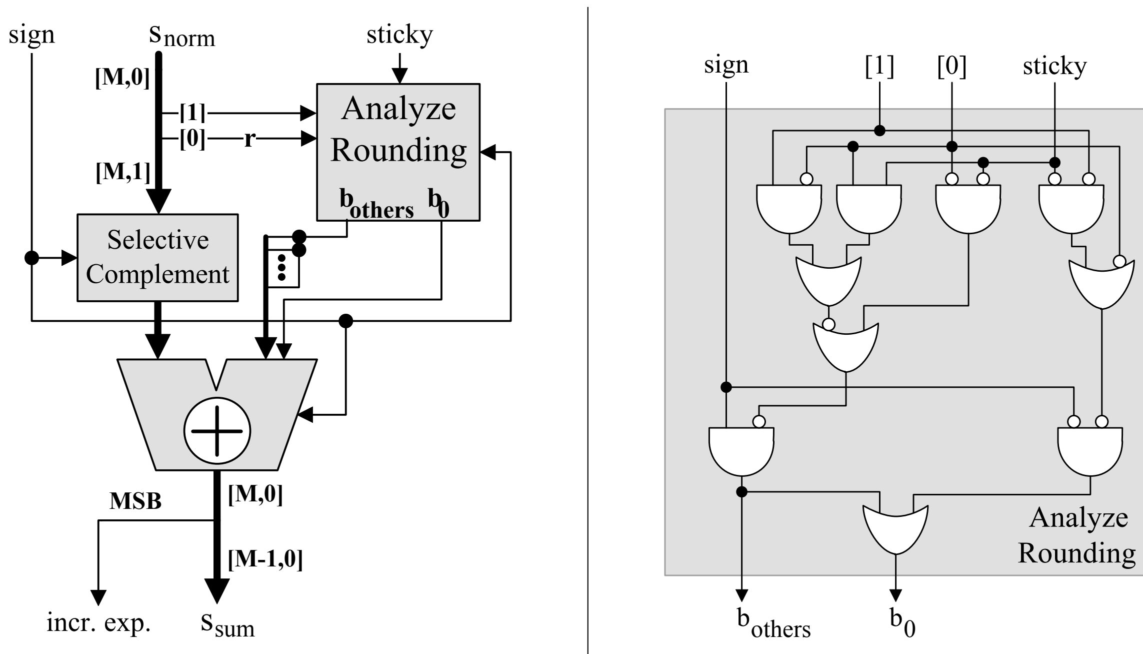


Abbildung 6.14: Schaltung zur 2er-Komplement-Bildung und Rundung der Mantisse beim Addierer für beliebige Gleitkommazahlen. Die abschließende Normalisierung beschränkt sich auf die Feststellung einer erforderlichen Inkrementierung des Exponenten (siehe Text). Rechts ist die Logik zur Umsetzung des Standardrundungsmodus *Round To Nearest Even* abgebildet.

Round To Nearest Integer					Round To Nearest Even				
Mantisse Bits	LUTs	Regs	Slices	Design Freq. (MHz)	Mantisse Bits	LUTs	Regs	Slices	Design Freq. (MHz)
12	232	143	139	83	12	275	157	162	83
14	270	160	158	88	14	318	166	183	85
16	312	177	181	83	16	346	183	199	87
18	339	193	197	85	18	382	198	219	85
20	374	209	214	88	20	419	214	240	78
22	403	225	233	86	22	447	231	257	82
24	437	241	251	85	24	480	247	276	78

Tabelle 6.10: Ressourcenverbrauch und Geschwindigkeit von Gleitkomma-Addierern bei Virtex-II-FPGA XC2V3000-4.

in der *Preparation*-Stufe die Geschwindigkeit der Addierer. Durch tieferes Pipelining kann die Geschwindigkeit noch stark erhöht werden, diese Optimierung wurde jedoch aus den gleichen Gründen wie beim Addierer für vorzeichenlose Zahlen nicht durchgeführt.

Akkumulation vorzeichenloser Zahlen

Die Akkumulation von Gleitkommazahlen, also die Addition aufeinanderfolgender Gleitkommazahlen, ist für die in dieser Arbeit betrachteten Berechnungen stets erforderlich, um die SPH-Größen aus den Einzelbeiträgen von Nachbarteilchen zu gewinnen. Bei der gegebenen Design-Strategie einer Rechen-Pipeline, die pro Takt eine Wechselwirkung berechnet, müssen die Akkumulatoren in der Lage sein, pro Takt eine Akkumulation durchzuführen.

Bei der Addition zweier Zahlen hängt das Ergebnis der Operation nur von den beiden Summanden ab. Spielt es keine Rolle, ob das Ergebnis erst nach mehreren Takten erscheint, kann deshalb bei der Implementierung der Operation beliebig tiefes Pipelining angewendet werden. So konnten bei den bisher vorgestellten Implementierungen zur Erhöhung der Taktfrequenz zwischen eingehenden Mantissen und dem Festkommazahl-Addierer zwei Pipelinestufen arbeiten.

Bei der Akkumulation von Zahlen verhält es sich grundlegend anders. Hier ist ein neuer Summand zur Summe aller vorangegangenen Summanden zu addieren. Das aktuelle Zwischenergebnis muss also in der *Operation*-Stufe in dem Moment korrekt ausgerichtet am Festkommaaddierer bereitstehen, wenn dort der neue Summand ansteht. Die Schwierigkeit dabei besteht darin, dass das Resultat der Festkommaaddition abhängig vom neuen Summanden um eine variable Anzahl von Stellen verschoben werden muss. Ein entscheidender Punkt für die Implementierung schneller Akkumulatoren ist, dass die zu den sich gerade in der Festkommaaddiererstufe befindlichen Zahlen gehörenden Exponenten frühzeitig berechnet werden können. Bereits bei Eingang eines neuen Summanden kann aus dessen Exponenten und den Exponenten der vorher eingegangenen Summanden der temporäre Exponent berechnet werden, der aktuell sein wird, wenn die Mantisse des neuen Summanden in der *Operation*-Stufe angelangt. Deshalb kann die *Preparation*-Stufe für die eingehenden Operanden durch Pipelinestufen genauso beschleunigt

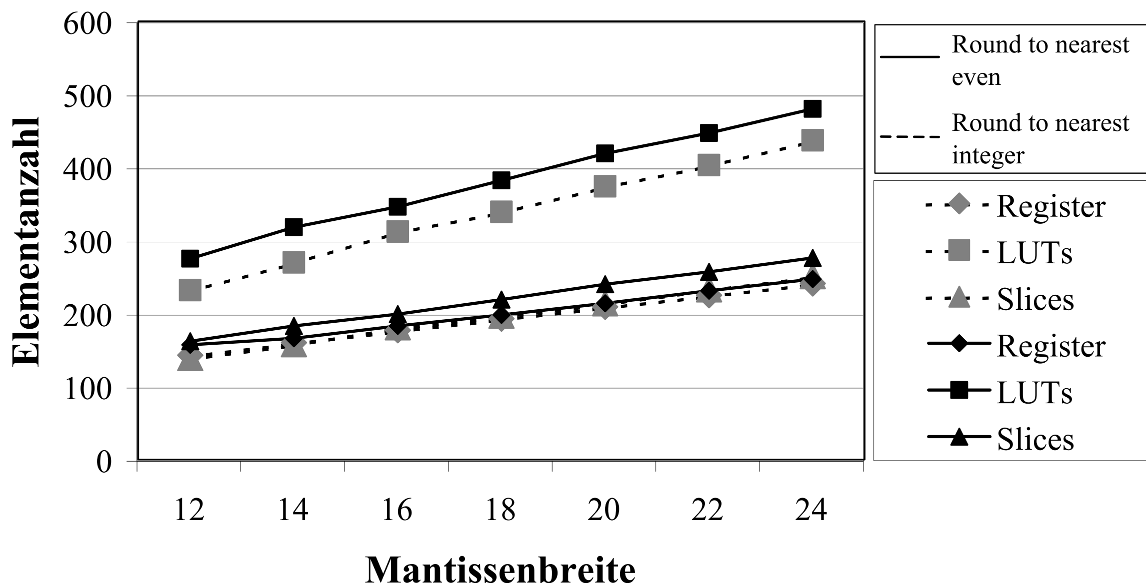


Abbildung 6.15: Verlauf des Ressourcenverbrauchs für die Gleitkomma-Addierer nach Tabelle 6.10.

werden, wie es bei den Addierern der Fall war. Es bleibt das Problem, dass das Zwischenergebnis der *Operation*-Stufe um eine variable Zahl von Stellen verschoben werden muss. Der Schlüssel für eine schnelle Implementierung liegt darin, die möglichen Verschiebeweiten einzuschränken, indem nur um ein Vielfaches einer ganzen Zahl P verschoben wird. Dies wird dadurch erreicht, dass intern die Exponenten auf Vielfache von P aufgerundet werden. Um einen Verlust von Rechengenauigkeit aufgrund der vergröberten Verschiebeweiten zu vermeiden, wird die Mantissenbreite am niederwertigen Ende um $P-1$ Bits erhöht. Für diese Arbeit wurde $P=8$ verwendet, was dazu führt, dass statt eines fünfstufigen Verschiebeelementes nur ein dreistufiges Element erforderlich wurde. Außerdem wurde anstatt eines Ripple-Carry-Addierers ein Carry-Save-Addierer verwendet. Damit konnte die Signalverzögerung auf vier Logikstufen begrenzt werden, was zu einer für diese Arbeit ausreichend schnellen Implementierung führt. Nachfolgend werden die drei Verarbeitungsstufen im Detail vorgestellt.

Preparation Im ersten Schritt der *Preparation*-Stufe wird der eingehende Exponente e_{in} im Vergleich zum internen temporären Exponenten e_t, welcher zur Zwischensumme der *Operation*-Stufe gehört, analysiert. Da intern auf einem vergröberten Exponentenbereich gearbeitet wird, ist zuerst e_{in} auf ein Vielfaches von $P=8$ aufzurunden. Der Exponent e_{in} liegt zwar in Bias-Darstellung vor - das spielt hier jedoch keine Rolle, da für die Verarbeitung bei der Addition nur die relativen Bezüge zwischen den Exponenten wichtig sind. Deshalb wird e_{in} ohne Einschränkung als positive Ganzzahl interpretiert. Das Aufrunden von $e_{in} = e_{in,E-1} \cdots e_{in,0}$ zu $e_{ru} = e_{ru,E-1} \cdots e_{ru,0}$ geschieht durch Null-Setzen der Bits $[2,0]$ und Addition einer Eins zu Bit

[3], wenn die ODER-Verknüpfung der drei Bits $e_{in,2} \cdots e_{in,0}$ Eins ergibt. Zur Feststellung der Verschiebungsweite der Mantisse des neuen Summanden s_{in} oder der Mantisse der Zwischensumme s_t wird die Differenz $v = e_t - e_{ru}$ gebildet. Ist v positiv, muss s_{in} rechtsverschoben werden und zwar um den Betrag $d = v + e_{ru} - e_{in}$. Für $v < 0$ muss s_{in} dagegen nur um $d = e_{ru} - e_{in}$ Stellen rechtsverschoben werden. Im letzteren Fall ist s_t um $|v|$ Stellen nach rechts zu verschieben. Ist $v < 0$ wird für die nächste Akkumulation $e_t = e_{ru}$ gesetzt, ansonsten bleibt e_t unverändert. Abbildung 6.16 zeigt die Umsetzung der Exponentenanalyse als Schaltung. In Abbildung 6.17 wird dieser Teil der Schaltung als *Prepare Exponents* bezeichnet. Es sind weiterhin der Verschiebebaustein für die Mantisse s_{in} und die beiden Verschiebebausteine für die in Carry-Save-Form vorliegende Zwischensumme s_{sum} gezeigt. Letztere verschieben s_t um das P-fache der angezeigten Verschiebeweite $shifts$ (hier gilt $P = 8$).

Die ursprünglich M Bit breiten Mantissen werden intern in einem erweiterten Format verarbeitet. Zum Ausgleich der Verschiebung durch das Aufrunden der Exponenten auf Vielfache von 8 werden $ext_{ru} = 7$ Stellen hinter das LSB angehängt. Zur Erhöhung der Genauigkeit der Akkumulation können weitere Guard-Bits (ext_{LSB}) angehängt werden. Da während der Akkumulation keine Analyse der Zwischensummen auf führende Nullen stattfindet, müssen die Mantissen auch am höherwertigen Ende erweitert werden. Sollen bis zu N Summanden akkumuliert werden, müssen $ext_{MSB} = \lceil log_2 N \rceil$ Bits hinzugefügt werden.

Die Flusssteuerung *Control* sichert das korrekte Arbeiten des Akkumulators, indem für eingangsseitiges $valid = 0$ die Registerinhalte der Exponentenvorverarbeitung und der verschobenen Mantissen fixiert werden. Durch das Signal $first$ getriggert, veranlasst die Steuerung den Reset der Register in *Prepare Exponents* und unterdrückt die Eingänge B und C des CSA. Das Signal $last$ induziert die Meldung eines fertigen Akkumulationsresultats durch Setzen des ausgangsseitigen $valid$-Signals.

Operation Wie bereits erwähnt, arbeitet der Addierer zur Akkumulation einer neuen Mantisse zur vorherigen Zwischensumme (beide in erweiterter Darstellung mit $ext_{MSB} + M + ext_{ru} + ext_{LSB}$ Bits Breite) als Carry-Save-Addierer. Damit wird der Addierer so schnell wie eine einfache LUT des FPGAs, erfordert jedoch doppelt so viele Logikressourcen wie ein Ripple-Carry-Addierer. Zudem liegt das Additionsergebnis s_t in Carry-Save-Form vor und muss nachfolgend in der *Normalization*-Stufe durch einen Carry-Propagate-Addierer in eine Binärdarstellung umgewandelt werden. Aufgrund der Verschiebeelemente im Logikpfad des Addierers wurde diese Implementierungsvariante zur Gewährleistung einer hohen Geschwindigkeit notwendig.

Normalization Wie in Abbildung 6.17 gezeigt, beginnt die *Normalization*-Stufe mit der Erzeugung einer Binärzahl für s_t. Dazu wird ein Ripple-Carry-Addierer verwendet. Die Zwischensumme kann maximal $ext_{MSB} + ext_{ru}$ führende Nullen aufweisen. Die Anzahl führender Null-Bits wird ähnlich wie beim Gleitkommazahl-Addierer gezählt, woraus die Verschiebeweite zur Normierung der Mantisse bestimmt wird. Für die Verschiebung um diese Verschiebeweite müssen nur $ext_{MSB} + M + ext_{ru}$ Bits einbezogen werden. Das führende Bit braucht nicht verschoben zu werden, da das MSB nach der Verschiebung nach Voraussetzung den Wert Eins haben wird, und deshalb ignoriert werden kann. Das letzte Bit ist das niederwertigste Bit, welches durch eine Ver-

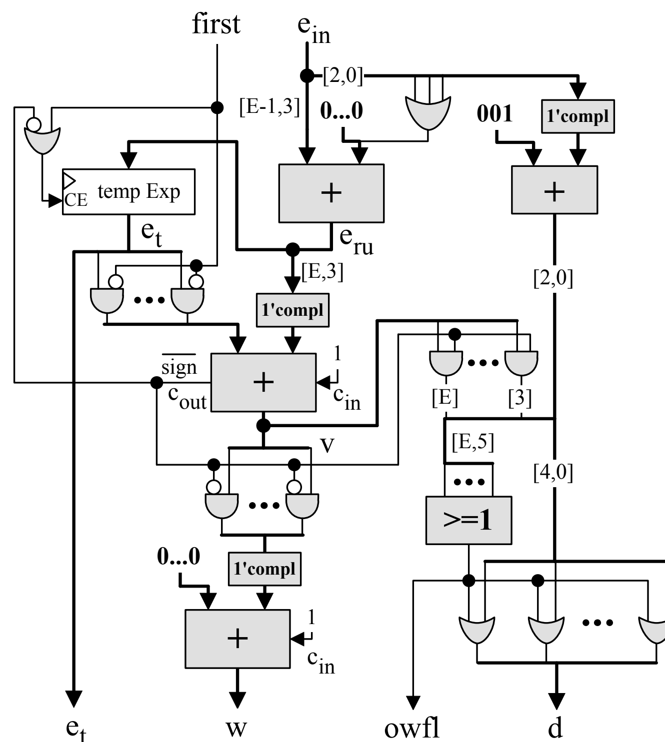


Abbildung 6.16: Schaltung zur Vorverarbeitung der Exponenten im Akkumulatorbaustein (*Prepare Exponents*). Ermittelt wird die erforderliche Verschiebeweite d der eingehenden Mantisse ($owfl = 1$ zeigt einen Überlauf der Verschiebung an) und die Weite w in Vielfachen von $P = 8$ der Rechtsverschiebung der Zwischensumme sowie der vorläufige Exponent der Zwischensumme e_t.

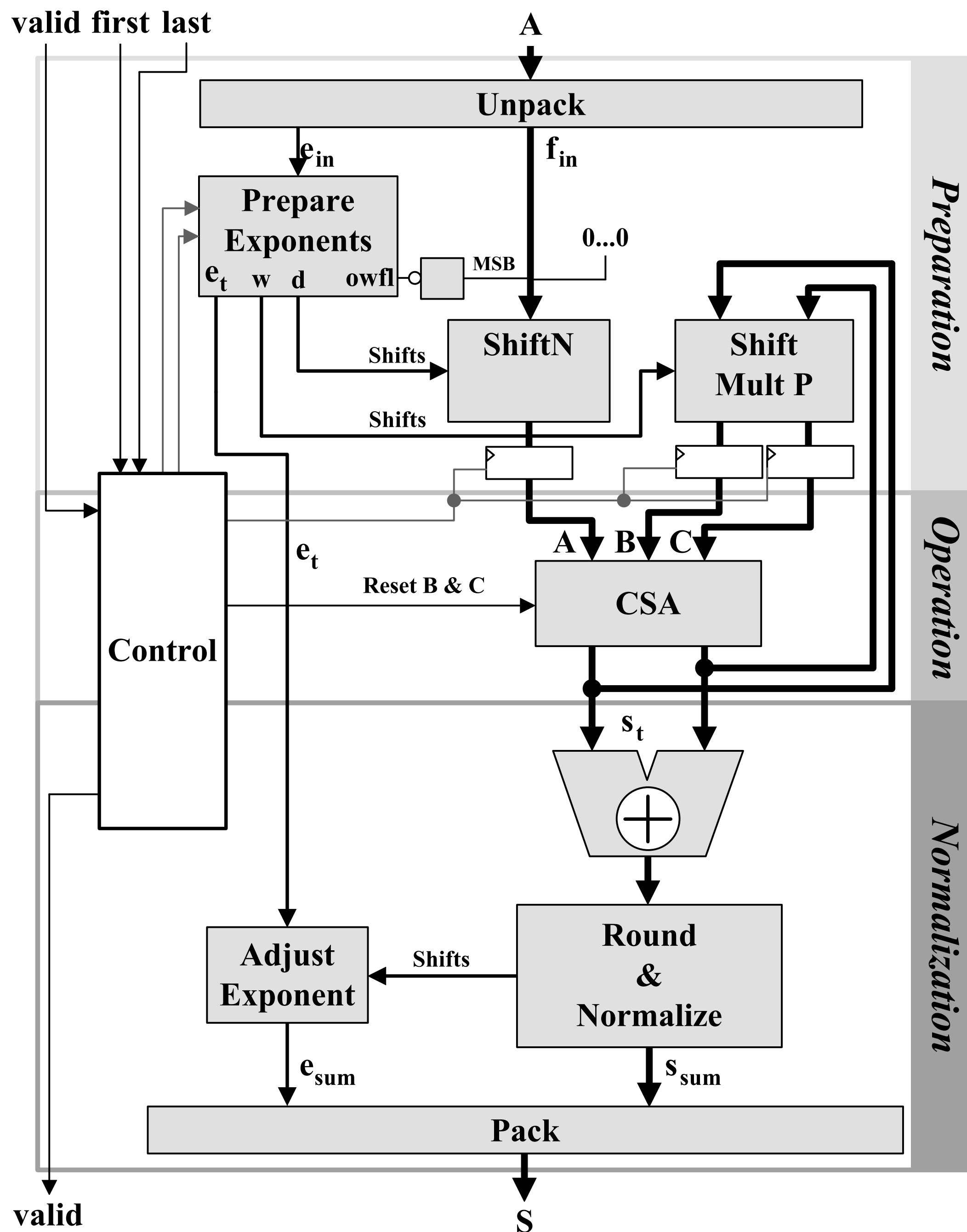


Abbildung 6.17: Blockschaltbild für einen Akkumulator für positive Gleitkommazahlen. Es sind nur die für die Funktion notwendigen Register gezeichnet. Zur Beschleunigung können – mit Ausnahme der Rückkopplungsschleife des CSA – weitere Pipeline-Register eingefügt werden. Die Steuerungssignale sind grau dargestellt.

positive Zahlen					beliebige Zahlen				
Mantisse Bits	LUTs	Regs	Slices	Design Freq. (MHz)	Mantisse Bits	LUTs	Regs	Slices	Design Freq. (MHz)
12	417	255	236	93	12	483	345	330	95
14	454	275	259	92	14	521	371	355	95
16	497	296	278	91	16	573	397	383	94
18	532	316	298	92	18	617	423	418	95
20	580	337	322	91	20	656	449	437	91
22	619	357	346	91	22	712	475	471	91
24	644	377	356	90	24	742	501	494	95

Tabelle 6.11: Ressourcenverbrauch und Geschwindigkeit von Akkumulatoren für Gleitkommazahlen bei Virtex-II-FPGA XC2V3000-4.

schiebung zum Rundungsbit werden kann. Der verschobene Bitvektor soll s_n genannt werden. Die Rundung selbst geschieht durch Addition einer Eins zu den führenden M Bits von s_n. Eine zweite Normierung nach der Rundung braucht wie beim Addierer nicht durchgeführt zu werden. Ein Überlauf kann nur stattfinden, wenn die führenden M Bits von s_n alle Eins sind, dann aber sind diese Bits nach der Rundung alle Null und die ausgegebene Mantisse (ohne führende 1) bleibt auch ohne Verschieben korrekt. Es muss also nur das Carry-Out des Rundungs-Addierers registriert werden, um den Exponenten des Ergebnisses entsprechend zu korrigieren. Die Anpassung des Exponenten geschieht anhand der Anzahl führender Nullen, der Zahl ext_{MSB} und des eben erwähnten Carry-Out-Signals.

Ressourcenverbrauch und Geschwindigkeit Tabelle 6.11 zeigt auf der linken Seite den Ressourcenverbrauch für die Akkumulatoren für positive Gleitkommazahlen in Abhängigkeit von der Mantissenbreite. Es ist zu sehen, dass etwa 40% mehr FPGA-Ressourcen benötigt werden als für den allgemeinen Gleitkommazahladdierer. Die Designs sind hier etwas schneller als bei den Addierern, was daran liegt, dass die Vorverarbeitung des eingehenden Exponenten durch tieferes Pipelining schneller als bei den Addierern wurde.

Akkumulation beliebiger Gleitkommazahlen

Sollen positive und negative Gleitkommazahlen akkumuliert werden, ergibt sich das Problem der Auslöschung führender Stellen in der *Operation*-Stufe. Zur Kompensation des damit einhergehenden Genauigkeitsverlustes müssten deutlich mehr Guard-Bits (ext_{LSB}) und wesentlich größere Verschiebeelemente für die Zwischenergebnisse s_t implementiert werden. Um dies zu vermeiden, wurde stattdessen die Idee verfolgt, positive und negative Summanden getrennt zu addieren.

Mit dieser Lösung kann der Aufbau des Akkumulators weitgehend gleich bleiben wie für die Akkumulation positiver Zahlen. Es liegt nämlich pro Takt immer nur ein Summand, positiv oder negativ, an. Es kann also so getan werden, als wären zwei Akkumulatoren, jeweils einer

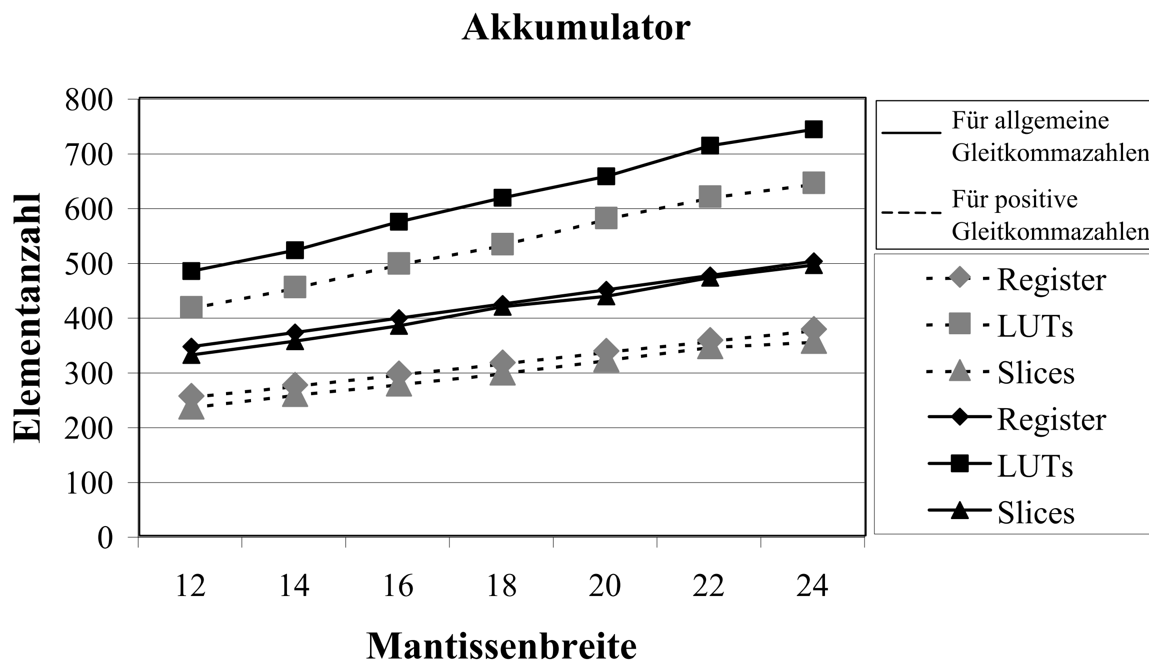


Abbildung 6.18: Verlauf des Ressourcenverbrauchs für die Gleitkomma-Akkumulatoren nach Tabelle 6.11.

für positive und negative Zahlen, vorhanden und abhängig vom Vorzeichen der eingehenden Summanden wird der eine oder andere aktiv. Es müssen dazu nur die Register zur Zwischenspeicherung der temporären Exponenten und Mantissen verdoppelt und über Multiplexer selektiert werden, die übrige Hardware kann für beide Akkumulatoren geteilt werden. Damit erhält man einen 2-Kanal-Akkumulator für Gleitkommazahlen ohne Vorzeichen.

Um daraus einen allgemeinen Gleitkommazahl-Akkumulator zu machen wird eine zusätzliche Kontroll-Logik erforderlich. Als Kontrollsignale von außen werden die Anzeige der ersten (*first*) und der letzten (*last*) zu akkumulierenden Gleitkommazahl benötigt. Das Signal *first* setzt die Zwischenergebnisse zurück, *last* triggert die Ausgabe der Ergebnisse. Damit der Akkumulator in der Lage ist, einen kontinuierlichen Zahlenstrom zu verarbeiten, können die beiden Kanäle nicht nach Vorzeichen geordnet werden. Stattdessen wird die eingehende Zahl für $first = 1$ in Kanal 1 verarbeitet und das Vorzeichen gespeichert. Alle folgenden Zahlen mit gleichem Vorzeichen kommen ebenfalls in Kanal 1, die mit abweichendem Vorzeichen werden in Kanal 2 geleitet. Für $last = 1$ wird das Ergebnis aus Kanal 1 ausgegeben, ein Takt danach das Ergebnis aus Kanal 2. Die ausgegebenen Vorzeichen ergeben sich aus dem gespeicherten Vorzeichen für den ersten Kanal.

In Abbildung 6.19 ist die Schaltung für einen solchen Akkumulator gezeigt. Es ist zu sehen, dass sowohl für die Zwischenergebnisse in der CSA-Rückkopplungschleife als auch für die vorläufigen Ergebnisse nach der Normalisierungsstufe zwei Registersätze eingefügt wurden, die über Multiplexer selektiert werden. Genauso ist das Register für die Zwischenspeicherung des temporären Exponenten im Modul *Prepare Exponents* zu verdoppeln. Dies wird durch die grau-

en Kontrollsignale zu diesem Modul hin angedeutet, welche zwei Clock-Enable-Signale und ein Reset-Signal darstellen. Die Kontrolle über die Clock-Enable-, Reset- und Select-Signale wird von der *Control State Machine* übernommen.

Tabelle 6.11 zeigt auf der rechten Seite den Ressourcenverbrauch und die Geschwindigkeit der auf diese Weise erzeugten Akkumulatoren in Abhängigkeit von der Mantissenbreite (siehe auch den grafischen Verlauf in Abbildung 6.18). Gegenüber dem Akkumulator für positive Zahlen ergibt sich ein Offset im Ressourcenverbrauch von etwa 40 %. Die Geschwindigkeit bleibt dagegen unverändert.

6.2.3 Implementierung von Gleitkomma-Multiplizierern auf FPGAs

In diesem Abschnitt wird zuerst detailliert auf den allgemeinen Fall der Multiplikation von zwei Zahlen eingegangen. Danach wird kurz der Spezialfall der Quadratur diskutiert.

Allgemeine Multiplikation

In Abschnitt 4.3.3 wurden die Verarbeitungsschritte eines Gleitkommazahl-Multiplizierers anhand Gleichung 4.39 motiviert, die hier zur Erinnerung noch einmal wiedergegeben wird:

$$\begin{aligned} \overbrace{\pm s_{prod}\, 2^{e_{prod}-bias}}^{R=A\cdot B} &= \overbrace{\left(\pm s_1\, 2^{e_1-bias}\right)}^{A} \cdot \overbrace{\left(\pm s_2\, 2^{e_2-bias}\right)}^{B} \\ &= \pm \underbrace{(s_1 \cdot s_2)}_{\text{Festkomma-Operator}} \quad 2^{(e_1+e_2-bias)-bias}. \end{aligned} \tag{6.16}$$

Durch Abbildung 4.22 und die zugehörige Erklärung ist bereits der grundsätzliche Aufbau eines Gleitkomma-Multiplizierers gezeigt worden. Dieser Aufbau wird nun etwas detaillierter, insbesondere in Bezug auf eine FPGA-Implementierung, ausgeführt. Die Struktur der sich ergebenden Schaltung ist in 6.20 skizziert.

Preparation Hier ändert sich nichts im Vergleich zu den Ausführungen in 4.3.3. Das Vorzeichen des Ergebnisses ergibt sich aus einer XOR-Verknüpfung der Operanden-Vorzeichen. Entsprechend der Bezeichnungen in Gleichung 6.16 sind e_1 und e_2 die Exponenten, s_1 und s_2 die Mantissen der entpackten Operanden.

Operation In dieser Stufe müssen die Exponenten addiert und die Mantissen multipliziert werden. Die Multiplikation von s_1 und s_2 geschieht über einen Festkomma-Multiplizierer in der Realisierung nach der Tree-Methode, wie in Abschnitt 6.1.2 vorgestellt.

Die Multiplikation von M-Bit-Mantissen mit MSB der Wertigkeit 2^0 und LSB der Wertigkeit 2^{-M+1} führt zu einem Zwischenergebnis s_{mult} mit $2M$ Stellen, wobei das MSB die Wertigkeit 2^1 hat ($s1, s2 \in [1,2) \Rightarrow s_1 \cdot s_2 \in [1,4)$).

Für die Addition der Exponenten ist zu berücksichtigen, dass sie in einer Darstellung mit Bias vorliegen. Nach der Addition von e_1 und e_2 muss deshalb der Wert *bias* subtrahiert werden,

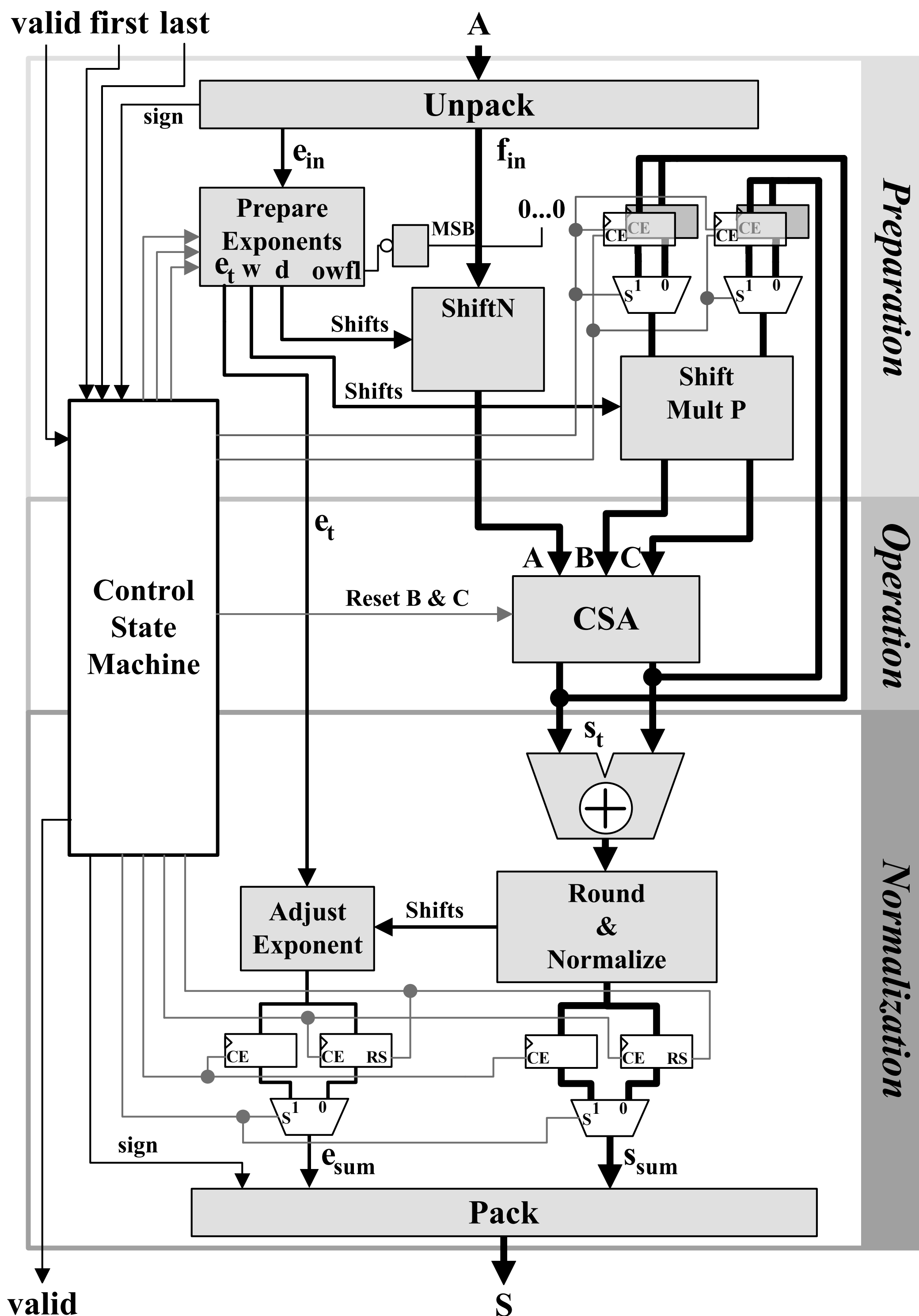


Abbildung 6.19: Blockschaltbild für einen Akkumulator für Gleitkommazahlen. Positive und negative Summanden werden getrennt akkumuliert. Die Kontrollsignale sind grau dargestellt.

um den vorläufigen Exponenten e_t zu erhalten. Da bei einer Exponentenbreite von E Bit nach Gleichung 4.34 gilt: $bias = 2^{E-1} - 1$, bedeutet eine Subtraktion dieser Zahl die Addition einer Eins zu Bit $[E-1]$ und $[0]$. Die Addition einer Eins zum LSB kann bereits durch ein Carry-In im Addierer für e_1 und e_2 umgesetzt werden und die Addition einer Eins zum MSB entspricht der Invertierung des MSB. Effektiv ist also der Hardwareaufwand für die Addition der Exponenten identisch mit der Addition von 2er-Komplement-Zahlen.

Normalization Dass die vorläufige Ergebnismantisse der *Operation*-Stufe im Wertebereich $[1,4)$ liegt, war bereits beim Addierer für positive Zahlen der Fall. Auch hier überzeugt man sich durch die Betrachtung der Operation auf den größtmöglichen darstellbaren Mantissen ($s_1 = s_2 = 2 - ulp$) leicht davon, dass das gerundete Ergebnis niemals größer oder gleich 4 wird:

$$\begin{aligned} (2-ulp)(2-ulp)+ulp &= 4-4\,ulp+\overbrace{ulp^2+ulp}^{<2\,ulp} \\ &< 2\cdot(2-ulp). \end{aligned} \tag{6.17}$$

Deshalb kann hier die gleiche Normalisierungs- und Rundungseinheit wie bei diesen Addierern verwendet werden, also die Schaltung nach Abbildung 6.10. Das Sticky-Bit ergibt sich aus der ODER-Verknüpfung aller Bits von s_{mult} mit niedrigerer Wertigkeit als 2^{-M+1} (Bits $[M-2,0]$), das Rundungsbit ist Bit $[M-1]$, die Bits $[2M-1,M]$ entsprechen dem Bitvektor s_{add} in dieser Abbildung. Auch hier muss der Exponent durch Inkrementieren korrigiert werden, falls eine Rechtsverschiebung der Mantisse notwendig wurde.

Ressourcenverbrauch und Geschwindigkeit Die Resultate für den Ressourcenverbrauch und die Geschwindigkeit der implementierten Gleitkomma-Multiplizierer sind in den Tabellen 6.12 und 6.13 differenziert nach Rundungsmodus und Verwendung von Block-Multiplizierern aufgelistet. Die Abbildungen 6.21 und 6.22 zeigen den grafischen Verlauf der Ergebnisse. Für den Fall, dass keine Virtex-II-Block-Multiplizierer verwendet werden, ergibt sich ein klarer quadratischer Zusammenhang zwischen Mantissenbreite und Ressourcenaufwand. Ausschlaggebend ist die Skalierung der Festkomma-Multiplikation. Durch die Exponentenverarbeitung und *Normalization*-Stufe ergibt sich ein Overhead von 15–40 %, abhängig von der Mantissenbreite.

Quadratur

Die Architektur zur Implementierung der Quadratur ist weitgehend identisch mit der der allgemeinen Multiplikation. Statt zweier Operanden ist nur A zu entpacken und die Vorzeichenverarbeitung entfällt komplett. Der Festkomma-Multiplizierer wird mit großer Ressourcenersparnis ersetzt durch eine Quadrierer-Einheit. Die Addition der Exponenten wird reduziert auf eine Verdoppelung des eingehenden Exponenten. Dazu wird der Exponent um eine Stelle linksverschoben und zur Bias-Korrektur das LSB auf Eins gesetzt und das MSB invertiert. Der Ressourcenaufwand dafür ist mit einem Inverter für das MSB also minimal. Die weitere Verarbeitung in der *Normalization*-Stufe ist identisch mit dem Multiplizierer für Gleitkommazahlen. Die Resultate sind in Tabelle 6.14 und in Abbildung 6.23 aufgetragen. Es wurde nur der Rundungsmodus

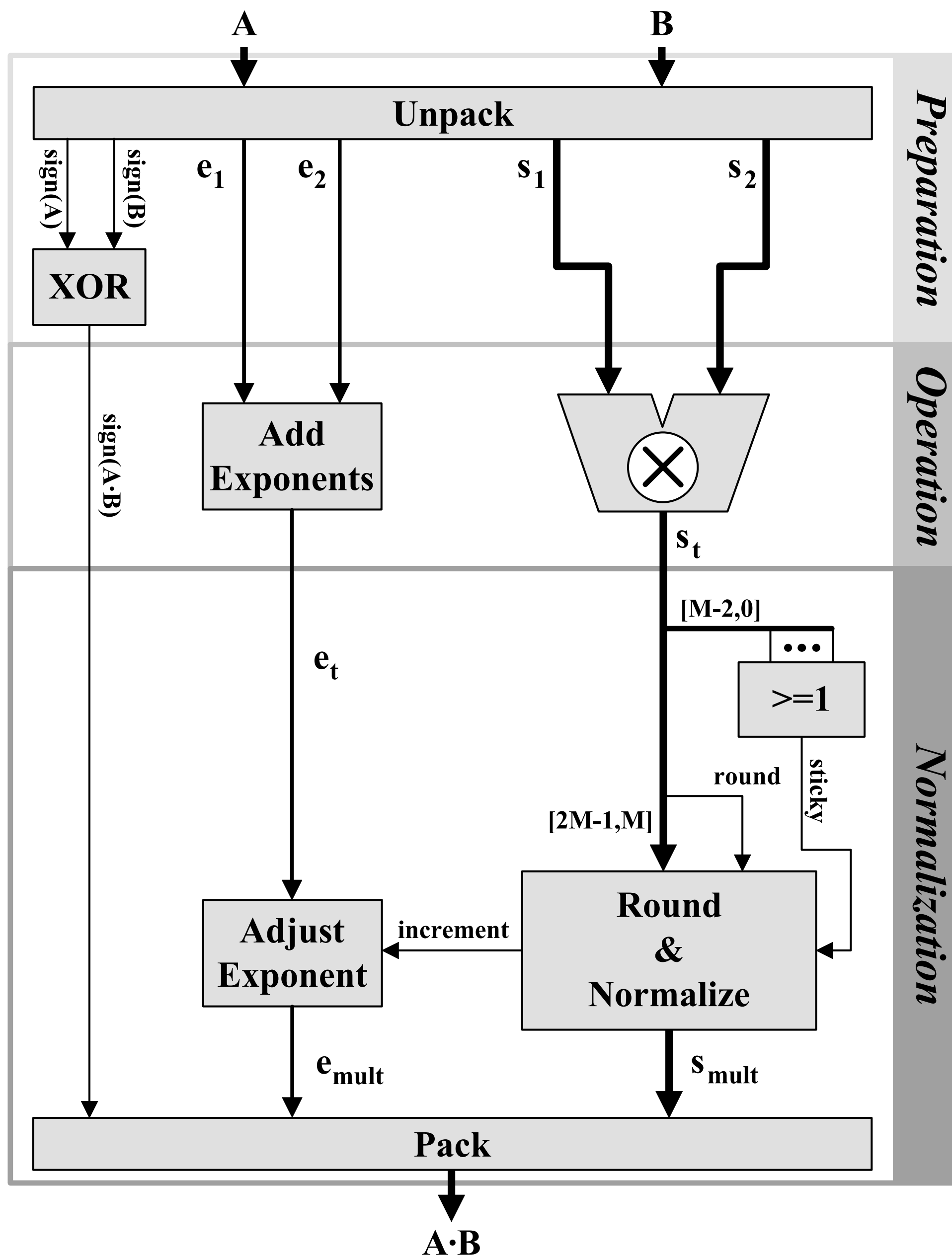


Abbildung 6.20: Blockschaltbild für einen Multiplizierer für Gleitkommazahlen.

Round To Nearest Integer					Round To Nearest Even				
Mantisse Bits	LUTs	Regs	Slices	Design Freq. (MHz)	Mantisse Bits	LUTs	Regs	Slices	Design Freq. (MHz)
12	46	54	34	104	12	50	54	39	95
14	50	58	36	92	14	55	58	42	88
16	57	62	40	91	16	60	62	45	81
18	119	124	92	76	18	126	128	101	76
20	118	82	68	73	20	125	100	84	73
22	133	88	76	71	22	138	108	91	71
24	145	94	82	70	24	150	116	97	70

Tabelle 6.12: Ressourcenverbrauch und Geschwindigkeit von Gleitkomma-Multiplizierern bei Virtex-II-FPGA XC2V3000-4 unter Benutzung der Block-Multiplizierer-Ressourcen des FPGAs. Bis zu einer Mantissenbreite von 18 Bit wird ein 18x18-Bit-Block-Multiplizierer verwendet, darüber hinaus vier dieser Elemente.

Round To Nearest Integer					Round To Nearest Even				
Mantisse Bits	LUTs	Regs	Slices	Design Freq. (MHz)	Mantisse Bits	LUTs	Regs	Slices	Design Freq. (MHz)
12	202	102	107	113	12	206	116	114	104
14	259	130	138	101	14	264	146	146	101
16	328	146	170	101	16	334	164	179	104
18	400	180	210	96	18	406	200	219	94
20	482	198	247	92	20	489	220	256	90
22	574	238	297	87	22	582	262	307	87
24	672	258	342	84	24	680	284	352	82

Tabelle 6.13: Ressourcenverbrauch und Geschwindigkeit von Gleitkomma-Multiplizierern bei Virtex-II-FPGA XC2V3000-4 ohne Benutzung der Block-Multiplizierer-Ressourcen des FPGAs.

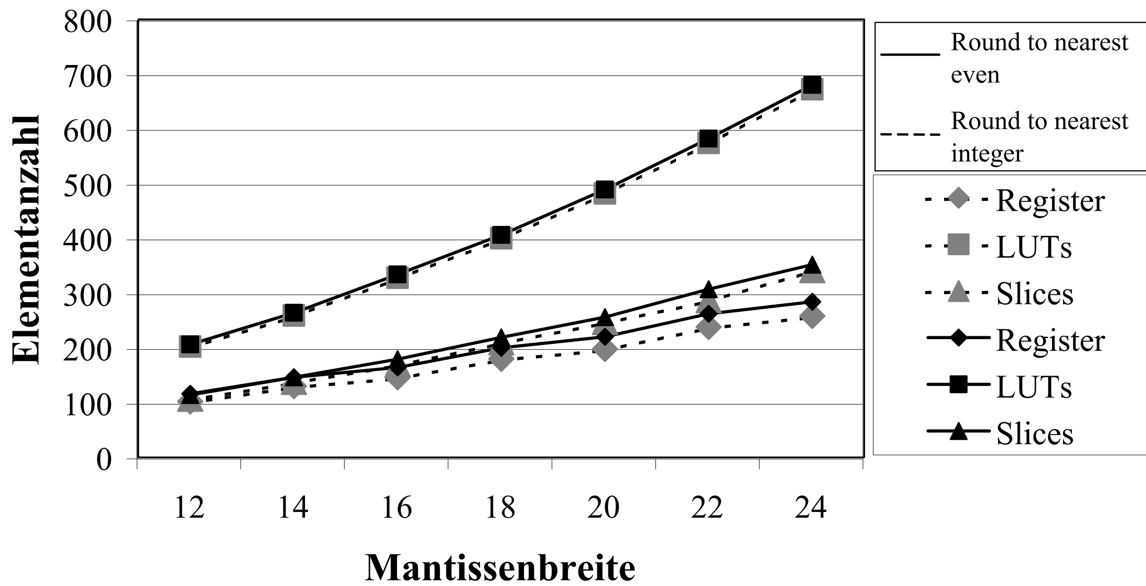


Abbildung 6.21: Verlauf des Ressourcenverbrauchs für die Gleitkomma-Multiplizierer nach Tabelle 6.13.

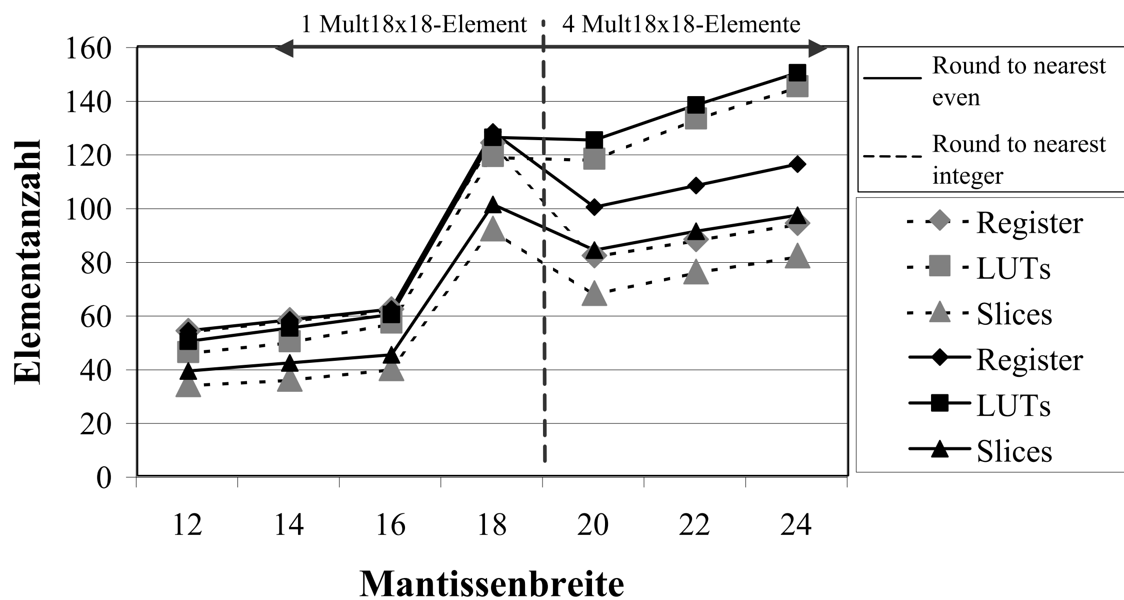


Abbildung 6.22: Verlauf des Ressourcenverbrauchs für die Gleitkomma-Multiplizierer nach Tabelle 6.12.

mit Block-Multiplizierern					ohne Block-Multiplizierer				
Mantisse Bits	LUTs	Regs	Slices	Design Freq. (MHz)	Mantisse Bits	LUTs	Regs	Slices	Design Freq. (MHz)
12	36	49	23	107	12	132	92	70	108
14	40	53	25	96	14	193	106	102	97
16	45	57	27	91	16	222	116	117	85
18	82	73	42	102	18	271	142	143	86
20	89	79	46	102	20	320	155	168	95
22	96	85	49	99	22	364	236	204	103
24	103	91	53	97	24	390	250	216	112

Tabelle 6.14: Ressourcenverbrauch und Geschwindigkeit von Gleitkomma-Quadrierern bei Virtex-II-FPGA XC2V3000-4 mit und ohne Benutzung der Block-Multiplizierer-Ressourcen des FPGAs. Bei Verwendung der Block-Multiplizierer werden bis zu einer Mantissenbreite von 18 Bit eines, darüber hinaus drei dieser Elemente implementiert.

Round To Nearest Integer implementiert. Der Fall, dass das Rundungsbit 1 ist und sich ein Sticky-Bit identisch Null ergibt, wird bei der Quadratur von physikalischen Größen nur extrem selten auftreten[6]. Es ist also kein signifikanter mittlerer Offset der Rechenergebnisse zu befürchten, wie er beispielsweise bei der Addition auftreten kann. Bei den Multiplizierern konnte diese Annahme nicht vorausgesetzt werden, da beispielsweise die Situation einer Multiplikation einer Konstante (z.B. 3.0) mit einer physikalischen Größe bei diesem Rundungsmodus durchaus zu einem mittleren Offset führen kann. Gegenüber den Multiplizierern ergibt sich für die Umsetzung ohne Block-Multiplizierer eine Ressourcenersparnis von 30–40 %. Der grafische Verlauf in Abbildung 6.23 zeigt kein quadratisches Verhalten, was darauf hindeutet, dass die Festkomma-Quadrierer nicht optimal implementiert sind. Der Ressourcenverbrauch liegt bei 12-Bit-Mantissen fast 50 %, bei 24-Bit-Mantissen dagegen nur etwa 19 % höher als bei den Festkommaquadrierern (vergleiche Tabelle 6.4). Die Geschwindigkeit der Designs ist vergleichbar mit den Ergebnissen für die Gleitkomma-Multiplizierer.

6.2.4 Implementierung von Gleitkomma-Dividierern auf FPGAs

Zur Erinnerung der Verarbeitungsschritte eines Dividierers für Gleitkommazahlen sei hier die bereits in Abschnitt 4.3.3 gegebene Gleichung für die Division (4.41) nochmals aufgestellt:

$$\begin{aligned} \overbrace{\pm s_{quot}\, 2^{e_{quot}-bias}}^{R=A/B} &= \overbrace{\left(\pm s_1\, 2^{e_1-bias}\right)}^{A} / \overbrace{\left(\pm s_2\, 2^{e_2-bias}\right)}^{B} \\ &= \pm \underbrace{(s_1/s_2)}_{\text{Festkomma-Operator}} \quad 2^{(e_1-e_2+bias)-bias}. \end{aligned} \tag{6.18}$$

[6]Dazu müssten bei einer Mantisse $1.x_{-1}\ldots x_{-M+1}$ mit ungerader Mantissenbreite M das Bit $x_{(M+1)/2}$ ungleich 0 und die Bits $x_{(M+1)/2-1}\ldots x_{-M+1}$ identisch 0 sein.

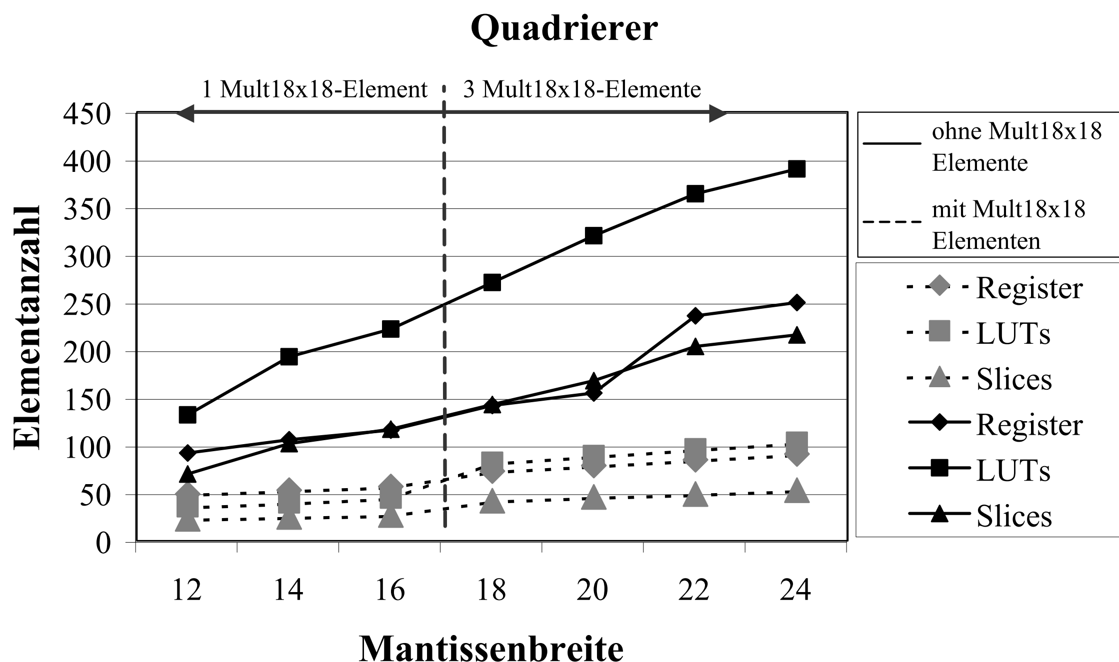


Abbildung 6.23: Verlauf des Ressourcenverbrauchs für die Gleitkomma-Quadrierer nach Tabelle 6.14.

Die schaltungstechnische Umsetzung dieser Berechnung, deren grundlegende Struktur bereits in Abschnitt 4.3.3 ausgeführt wurde (siehe Abbildung 4.23), wird nun konkretisiert. In Abbildung 6.24 ist die Struktur der FPGA-Implementierung für den Dividierer gezeigt, wie sie im Folgenden beschrieben wird.

Preparation Diese Stufe ist identisch mit der des Gleitkomma-Multiplizierers, da auch hier die Exponenten keine Vorverarbeitungsschritte der Mantissen bedingen. Es ist lediglich das Vorzeichen des Ergebnisses zu bilden.

Operation In dieser Stufe muss die Differenz der Exponenten e_1 und e_2 und der Quotient der M-Bit-Mantissen s_1 und s_2 gebildet werden. Für die Division der Mantissen wird ein Festkomma-Dividierer nach der Non-Restoring-Methode, wie er in Abschnitt 6.1.3 vorgestellt wurde, implementiert. Hierbei ist zu erwähnen, dass s_1 und s_2 im Intervall $[1,2)$ liegen und deshalb die Bedingung 6.6 in jedem Fall erfüllt ist - es kann also kein Überlauf auftreten. Um das Ergebnis korrekt runden zu können, ist es erforderlich, den vorläufigen Quotienten s_t auf mindestens $M+2$ Stellen zu berechnen. Dies folgt daraus, dass gilt: $s_t \in (0.5,2)$, und deshalb vor dem Runden eine Normalisierungsverschiebung um eine Stelle nach links erfolgen kann. Insbesondere für den Standardrundungsmodus muss auch ein Sticky-Bit erzeugt werden. Dieses kann aus dem Rest der Division ermittelt werden, denn das Sticky-Bit ist genau dann identisch Null, wenn der Rest Null ist. Es muss also ein Dividierer mit Ausgabe des Restwertes *rem* verwendet werden. Das

Sticky-Bit ergibt sich dann aus der ODER-Verknüpfung der Bits von *rem*.

Bei der Subtraktion der Exponenten, welche in der Darstellung mit Bias vorliegen, ist zu berücksichtigen, dass der Wert $bias = 2^{E-1} - 1$ hinzuaddiert werden muss, um den vorläufigen Exponenten e_t ebenfalls in der Bias-Darstellung zu erhalten. Für e_t gilt also:

$$\begin{aligned} e_t &= e_1 - e_2 + 2^{E-1} - 1 \\ &= e_1 + \overline{e_2} + 2^{E-1}. \end{aligned}$$

Es muss deshalb nur das 1er-Komplement von e_2 zu e_1 addiert und das MSB des Resultats invertiert werden, um e_t zu erhalten.

Normalization Wie bereits erwähnt, führt die Beziehung $s_t \in (0.5, 2)$ dazu, dass eine Linksverschiebung um eine Stelle zur Normalisierung der Mantisse erforderlich werden kann. Dass niemals eine gerundete Mantisse größer oder gleich 2 entstehen kann, wird aus der Betrachtung der Division der größtmöglichen Mantisse ($s_{max} = 2 - ulp$) durch die kleinstmögliche Mantisse ($s_{min} = 1$) ersichtlich:

$$\begin{aligned} \frac{s_{max}}{s_{min}} + \frac{ulp}{2} &= \frac{2 - ulp}{1} + \frac{ulp}{2} = 2 - \frac{ulp}{2} \\ &= 1.\ 1\ 1 \ldots 1\ 1\ |\underbrace{1}_{\text{Rundungsbit}}. \end{aligned} \tag{6.19}$$

Es wird nun s_t in jedem Fall um eine Stelle nach links verschoben, sodass sich $s'_t = 2 \cdot s_t \in (1, 4)$ ergibt. Die Mantisse s'_t kann nun mit der gleichen *Round & Normalize*-Schaltung normalisiert und gerundet werden, wie sie für den Addierer für positive Gleitkommazahlen und den Multiplizierer verwendet wurde (Abbildung 6.10). Zeigt diese Schaltung eine erfolgte Rechtsverschiebung um eine Stelle an, macht das die Linksverschiebung von s_t zu s'_t rückgängig und der vorläufige Exponent e_t ist auch der Exponent des Ergebnisses. Wurde dagegen keine Rechtsverschiebung vorgenommen, ist e_t um Eins zu verringern.

Ressourcenverbrauch und Geschwindigkeit Die Tabellen 6.15 und 6.16 zeigen die erzielten Ergebnisse bezüglich Ressourcenverbrauch und Geschwindigkeit der Gleitkomma-Dividierer für verschiedene Pipelinetiefen und Rundungsmodi. Abbildung 6.25 illustriert die Skalierung des Ressourcenverbrauchs für die Dividierer mit Rundungsmodus *Round To Nearest Even*. Wie bei der Diskussion der Festkomma-Dividierer deutlich wurde, sind Dividierer mit einer Pipelinetiefe von vier Stufen inakzeptabel langsam, bei einer Pipelinetiefe von 1 dagegen wird der Ressourcenverbrauch von den Registern dominiert. Deshalb wurden nur die Pipelinetiefen 2 und 3 näher untersucht. Wie bei der Muliplikation ergibt sich ein quadratisch skalierender Ressourcenaufwand. Während sich die Anzahl an LUTs nur wenig (< 10 %) von den Zahlen für die Mutiplizierer unterscheidet, ergibt sich für eine Pipelinetiefe von 2 ein Mehrverbrauch von etwa 50 %, welcher, wie bereits bei der Diskussion der Festkomma-Dividierer erwähnt, von der großen Zahl an Registern stammt, die nicht einer vorangehenden LUT zugeordnet werden können. Der Übergang von einer Pipelinetiefe von 3 zu einer Tiefe von 2 führt zu einem Mehrverbrauch an Slices

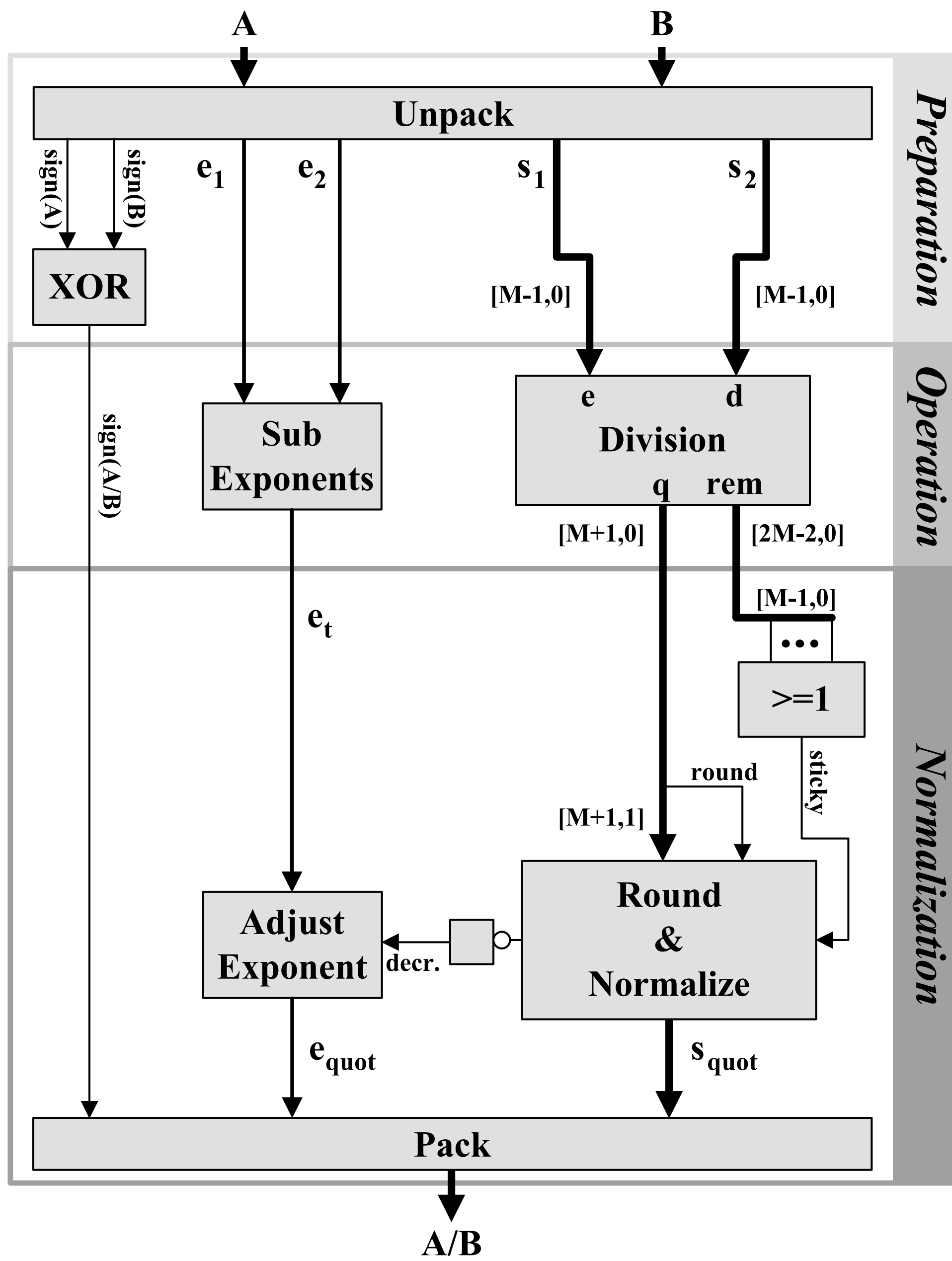


Abbildung 6.24: Blockschaltbild für einen Dividierer für Gleitkommazahlen.

3 Quotientenbits pro Pipelinestufe					2 Quotientenbits pro Pipelinestufe				
Mantisse Bits	LUTs	Regs	Slices	Design Freq. (MHz)	Mantisse Bits	LUTs	Regs	Slices	Design Freq. (MHz)
12	228	153	147	71	12	230	200	161	91
14	288	173	180	69	14	292	257	206	91
16	359	226	230	69	16	362	322	257	90
18	438	287	285	64	18	440	395	314	85
20	522	325	332	61	20	526	476	377	84
22	617	388	399	61	22	620	565	446	79
24	720	469	471	59	24	722	662	521	78

Tabelle 6.15: Ressourcenverbrauch und Geschwindigkeit von Gleitkomma-Dividierern ohne Rundung des Ergebnisses bei Virtex-II-FPGA XC2V3000-4.

in der Größenordnung 12–14 %. Die Implementierung des Rundungsmodus *Round To Nearest Even* führt gegenüber dem *Truncation*-Modus zu einem Mehraufwand von 15–20 %. Nicht aufgeführt wurden die Daten zum ebenfalls implementierten Rundungsmodus *Round To Nearest Integer*. Hier liegen die Zahlen zum Ressourcenverbrauch relativ genau zwischen den Werten der Tabellen 6.15 und 6.16.

6.2.5 Implementierung der Gleitkomma-Quadratwurzel auf FPGAs

Wie bei den vorangegangenen Abschnitten soll auch hier die in Abschnitt 4.3.3 angeführte Gleichung für die Operation auf Gleitkommazahlen wiedergegeben werden, um davon ausgehend die in jenem Abschnitt bereits grundlegend beschriebene Verarbeitung der Gleitkommaeinheit für die Quadratwurzel zu konkretisieren. Es handelt sich hier um die Wiedergabe von Gleichung 4.43:

$$\begin{aligned}
\overbrace{\pm s_{sqrt}\, 2^{e_{sqrt}-bias}}^{R=\sqrt{A}} &= \Big(\overbrace{\pm s_1\, 2^{e_1-bias}}^{A}\Big)^{\frac{1}{2}} \\
&= \pm \underbrace{\sqrt{s_1'}}_{\text{Festkomma-Operator}} \; 2^{\left(e_1'-bias\right)/2}
\end{aligned} \tag{6.20}$$

$$\begin{aligned}
s_1' &= \begin{cases} s_1 & : \; e_1 - bias \quad \text{gerade} \\ s_1\, 2 & : \; e_1 - bias \quad \text{ungerade} \end{cases} \\
e_1' &= \begin{cases} e_1 & : \; e_1 - bias \quad \text{gerade} \\ e_1 - 1 & : \; e_1 - bias \quad \text{ungerade.} \end{cases}
\end{aligned}$$

Abbildung 6.26 zeigt die Implementierung des Operators, wie sie im Folgenden erläutert wird.

3 Quotientenbits pro Pipelinestufe					2 Quotientenbits pro Pipelinestufe				
Mantisse Bits	LUTs	Regs	Slices	Design Freq. (MHz)	Mantisse Bits	LUTs	Regs	Slices	Design Freq. (MHz)
12	277	151	174	70	12	281	219	198	86
14	347	199	223	69	14	350	279	249	83
16	426	254	279	67	16	436	347	311	89
18	509	283	325	62	18	513	423	366	76
20	603	351	389	62	20	606	527	434	75
22	706	426	463	59	22	708	621	509	77
24	807	477	523	59	24	818	723	588	78

Tabelle 6.16: Ressourcenverbrauch und Geschwindigkeit von Gleitkomma-Dividierern mit Rundungsmodus *Round To Nearest Even* bei Virtex-II-FPGA XC2V3000-4.

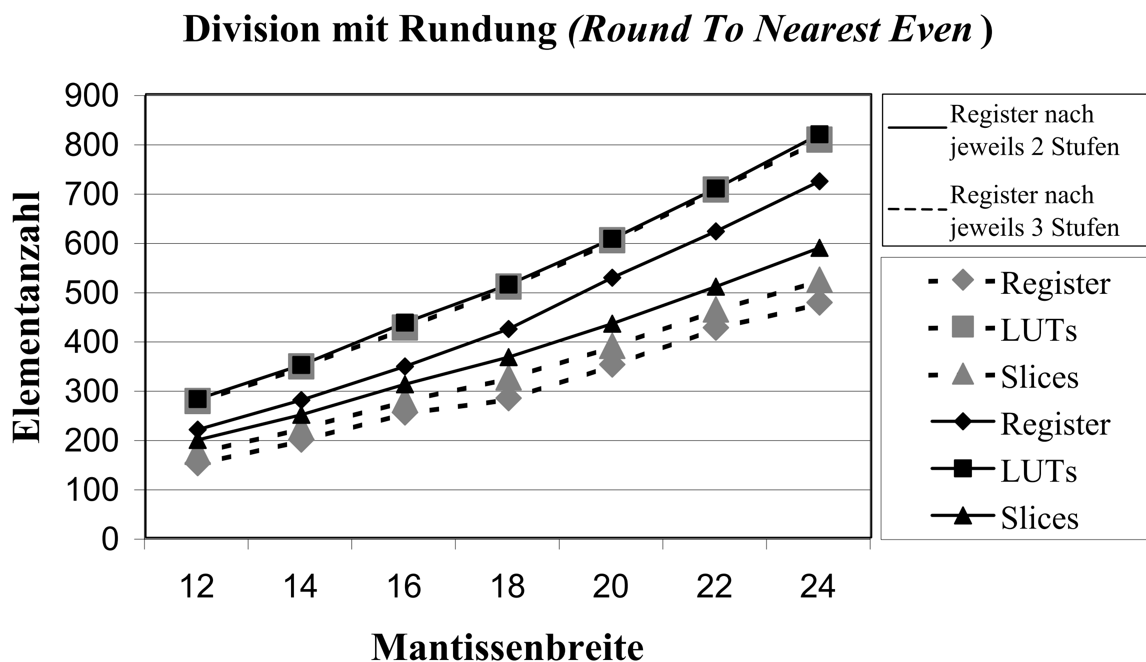


Abbildung 6.25: Verlauf des Ressourcenverbrauchs für die Gleitkomma-Dividierer nach Tabelle 6.16.

Preparation Dass bei der Quadratwurzel der Exponent ganzzahlig durch zwei dividiert werden muss, dieser jedoch nicht notwendigerweise geradzahlig ist, macht eine Vorverarbeitung der eingehenden Mantisse erforderlich. Ist e_1 gerade, was bedeutet, dass $e_1 - bias$ ungerade ist, wird s_1 um eine Stelle linksverschoben, ansonsten belassen. Die Schaltung dazu besteht aus einem einfachen Multiplexer, dessen Select-Signal durch das LSB von e_1 bestimmt ist. Die bearbeitete Mantisse ist s'_1. Für ein gerades e_1 müsste nun e'_1 durch Subtraktion von Eins berechnet werden. Um dazu nicht einen eigenen Subtrahierer aufzuwenden, wird diese Operation auf die *Operation*-Stufe verschoben, indem lediglich ein Signal $\overline{decr} = \overline{1} = 0$ weitergeleitet wird ($\overline{decr} = 1$ für e_1 ungerade).

Operation Der Exponent des Ergebnisses ergibt sich aus der Beziehung:

$$e_{sqrt} = (e'_1 - bias)/2 + bias = (e_1 + bias - decr)/2.$$

Diese Operation kann auch folgendermaßen formuliert werden:

$$e_{sqrt} = \begin{cases} \frac{e_1 - 1}{2} + \frac{bias - 1}{2} + 1 & : \; decr = 0 \\ \frac{e_1}{2} + \frac{bias - 1}{2} & : \; decr = 1. \end{cases}$$

Dies bedeutet, dass e_1 um eine Stelle nach rechts verschoben werden muss und dabei das LSB vernachlässigt werden kann. Die Addition von $\frac{bias-1}{2} = 2^{E-1} - 1$ und Eins im Falle $decr = 0$ geschieht über einen einfachen Addierer, dessen Carry-In mit $\overline{decr}$ verbunden ist.

Die Quadratwurzel der $(M+1)$-Bit-Zahl s'_1 (MSB mit Wertigkeit 2^1) muss auf $M+1$ Stellen berechnet werden (MSB mit Wertigkeit 2^0) - das LSB ist dann das Rundungsbit, welches in der *Normalization*-Stufe benötigt wird. Das Festkomma-Quadratwurzelergebnis ist s_t.

Normalization Diese Stufe beinhaltet lediglich die Addition von $ulp/2$ zu s_t, um das Resultat s_{sqrt} zu erhalten. Ein Sticky-Bit wie bei den bisher besprochenen Operationen ist im Fall der Quadratwurzel nicht erforderlich. Dies liegt daran, dass der Fall eines nichtverschwindenden Rundungsbits bei gleichzeitigem Verschwinden aller Bits eines exakten Ergebnisses über das Rundungsbit hinaus nicht auftreten kann. Wäre dies der Fall, stellte s_t die exakte Wurzel von s'_1 dar. Da die Quadrierung von s_t dann zu einem nichtverschwindenden Bit der Wertigkeit $ulp^2/2$ führt, welches nicht in s'_1 vorliegt, ergibt sich daraus jedoch ein Widerspruch.

Wie bereits in Abschnitt 4.3.3 festgestellt, bedarf es weder nach der Festkomma-Quadratwurzel noch nach der Rundung einer Normalisierungs-Verschiebung.

Ressourcenverbrauch und Geschwindigkeit In den Tabellen 6.17 und 6.18 sind die erzielten Werte für den Ressourcenverbrauch und die Geschwindigkeiten für die Gleitkomma-Quadratwurzelberechnung aufgeführt. Abbildung 6.27 zeigt den Verlauf des Ressourcenverbrauches aus Tabelle 6.18. Es ergibt sich qualitativ der gleiche Verlauf wie bei den Dividierern. Die Operatoren benötigen jedoch grob nur etwa 60 % der Ressourcen der Dividierer gleicher Mantissenbreite,

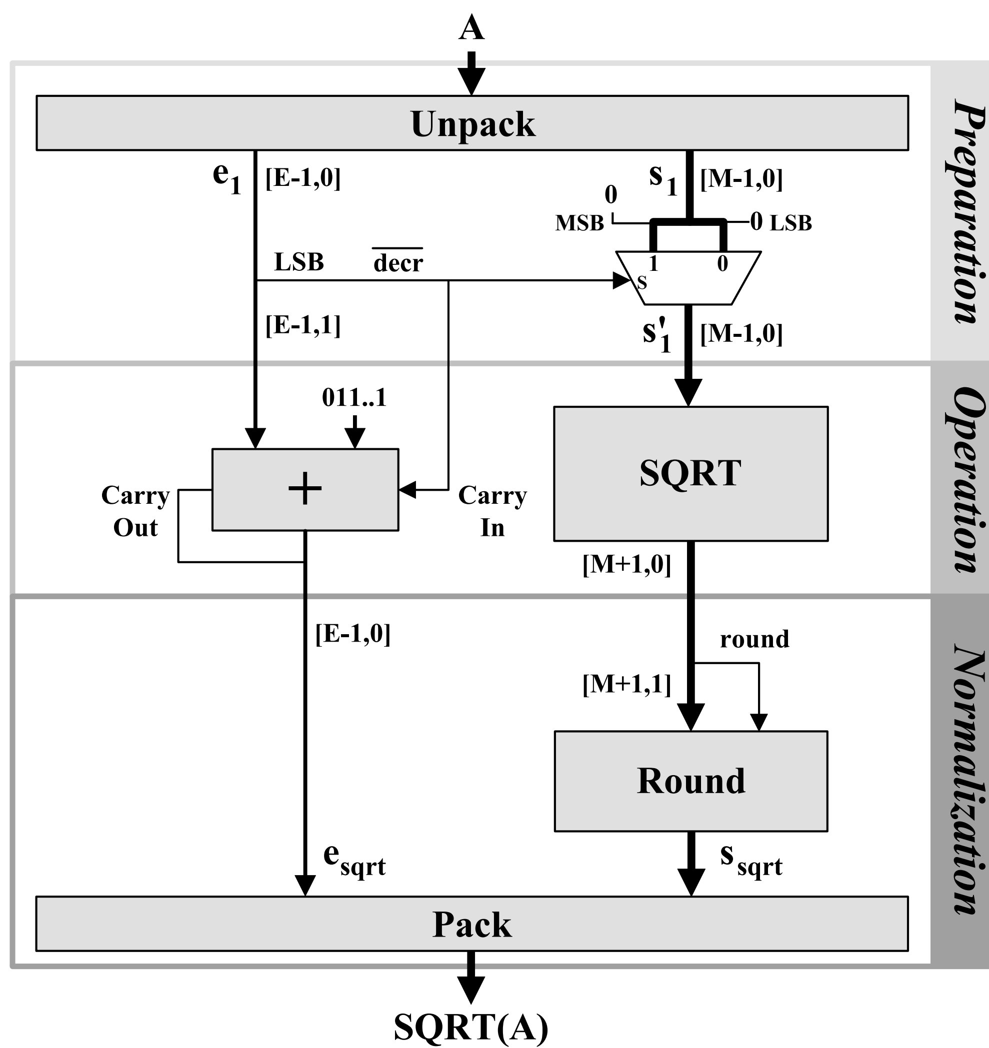


Abbildung 6.26: Blockschaltbild für einen Quadratwurzel-Operator für Gleitkommazahlen.

3 Ergebnisbits pro Pipelinestufe					2 Ergebnisbits pro Pipelinestufe				
Mantisse Bits	LUTs	Regs	Slices	Design Freq. (MHz)	Mantisse Bits	LUTs	Regs	Slices	Design Freq. (MHz)
12	134	79	92	70	12	134	112	102	96
14	171	114	119	67	14	172	144	129	93
16	210	124	144	66	16	215	183	161	95
18	253	164	178	66	18	258	216	191	89
20	303	208	213	64	20	309	264	230	87
22	356	212	243	61	22	362	310	269	86
24	413	262	284	62	24	420	366	313	87

Tabelle 6.17: Ressourcenverbrauch und Geschwindigkeit von Gleitkomma-Quadratwurzel-Operatoren ohne Rundungsfunktion bei Verwendung des Virtex-II-FPGAs XC2V3000-4.

was sich vor allem durch die gegenüber den Festkommadividierern um etwa 30 % geringere Größe der Festkommaquadratwurzeln ergibt. Die erzielten Geschwindigkeiten sind etwas höher als bei den Dividierern. Beim Übergang von Pipelinetiefe 3 zu 2 ergibt sich ein Mehraufwand an Slices in der Größenordnung 10 %.

6.2.6 Zusammenfassung und Vergleich der Implementierungsergebnisse zu den Operatoren

In den letzten Abschnitten wurden die wichtigsten Gleitkommaoperatoren, die im Rahmen dieser Arbeit entwickelt wurden, im Detail diskutiert. Wichtigstes Optimierungskriterium war, einen möglichst geringen Ressourcenaufwand bei akzeptabler Geschwindigkeit zu erreichen. Bei den Digit-Recurrence-Methoden für die Divisions- und Quadratwurzelberechnung wurde durch die Parametrisierung der Pipelinetiefe eine direkte Einflussmöglichkeit auf die Geschwindigkeit und den Ressourcenverbrauch der Schaltungen gegeben. Es wurden zahlreiche Spezialisierungen der Operatoren entwickelt, um abhängig von den numerischen Eigenschaften eines Algorithmus jeweils die optimale Implementierung wählen zu können. So wurde bei der Addition und Akkumulation unterschieden, ob die Operanden über ein Vorzeichen verfügen oder ausschließlich positiv sind. Es wurden getrennte Designs für die Multiplikation und die Quadratur aufgebaut. Außerdem wurde die Unterstützung verschiedener Rundungsmodi eingeführt.

Für die einzelnen Operatoren wurden bereits die Implementierungsergebnisse in Abhängigkeit von der Darstellungsgenauigkeit der verarbeiteten Gleitkommazahlen tabellarisch und grafisch dargestellt, wobei die Resultate bezüglich des Aufwands an 4-Input-LUTs, 1-Bit-Registern und Slices sowie der erzielten Geschwindigkeit aufgeschlüsselt wurden. Dies macht es möglich, bereits vor der Implementierung neuer Rechenwerke den Aufwand relativ genau abzuschätzen.

Um für Ressourcenabschätzungen den Vergleich zwischen verschiedenen Operationen zu erleichtern, sind in Abbildung 6.28 einige Ergebnisse bei Mantissenbreiten von 16 und 24 Bit ein-

3 Ergebnisbits pro Pipelinestufe					2 Ergebnisbits pro Pipelinestufe				
Mantisse Bits	LUTs	Regs	Slices	Design Freq. (MHz)	Mantisse Bits	LUTs	Regs	Slices	Design Freq. (MHz)
12	157	91	103	66	12	159	124	113	94
14	195	126	131	70	14	200	158	140	101
16	237	168	161	69	16	244	194	170	95
18	282	178	188	66	18	292	234	203	88
20	335	224	224	61	20	344	278	239	88
22	391	274	262	63	22	400	326	278	92
24	450	280	294	62	24	460	378	320	88

Tabelle 6.18: Ressourcenverbrauch und Geschwindigkeit von Gleitkomma-Quadratwurzel-Operatoren mit Rundungsmodus *Round To Nearest Even* bei Verwendung des Virtex-II-FPGAs XC2V3000-4.

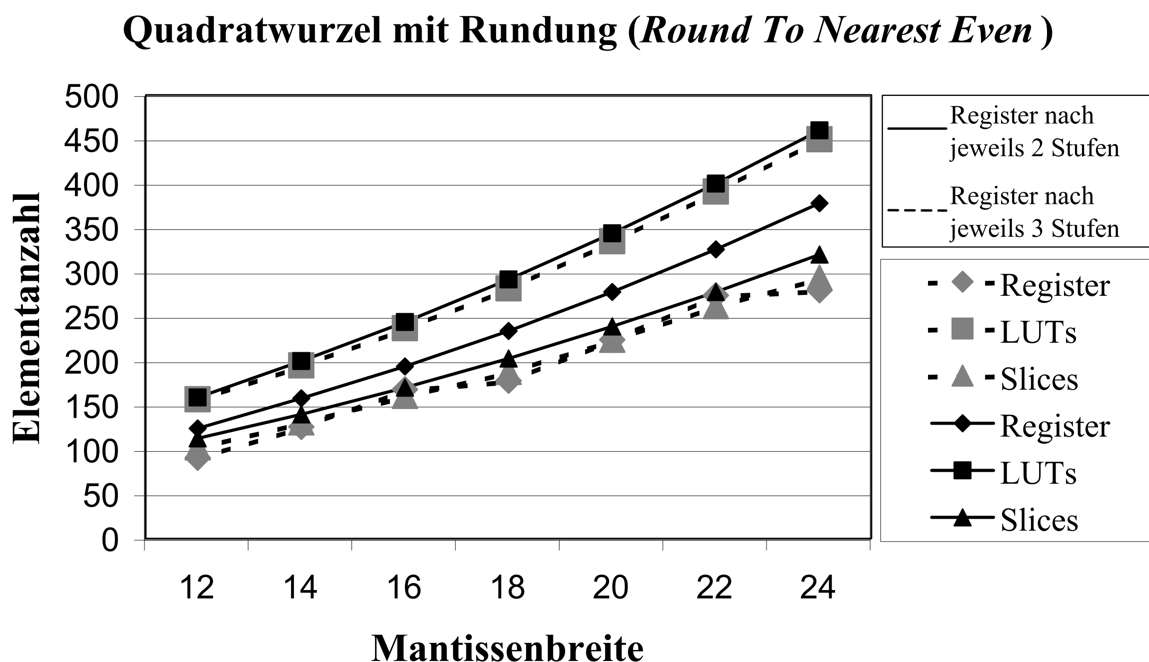


Abbildung 6.27: Verlauf des Ressourcenverbrauchs für die Quadratwurzel-Operatoren nach Tabelle 6.18.

Operator	Lienhart Slices	Lienhart Freq. (MHz)	Lienhart Latenz	Nallatech Slices	Nallatech Freq (MHz)	Nallatech Latenz	Quixilica Slices	Quixilica Freq. (MHz)	Quixilica Latenz
Add	276	78	6	290	152	14	365	137	10
Mult	97	70	3	126	113	6	358	128	7
Div	588	78	15	730	154	26	738	139	27
SQRT	320	88	14	330	151	29	675	175	27

Tabelle 6.19: Vergleich der Implementierungsergebnisse des Autors mit kommerziellen Produkten für den FPGA Virtex-II-4. Verglichen werden folgende Operatoren: Addierer für allgemeine Zahlen (Add), Muliplizierer (Mult), Dividierer (Div) und Quadratwurzel-Operator (SQRT), alle mit Single-Precision und Standardrundungsmodus.

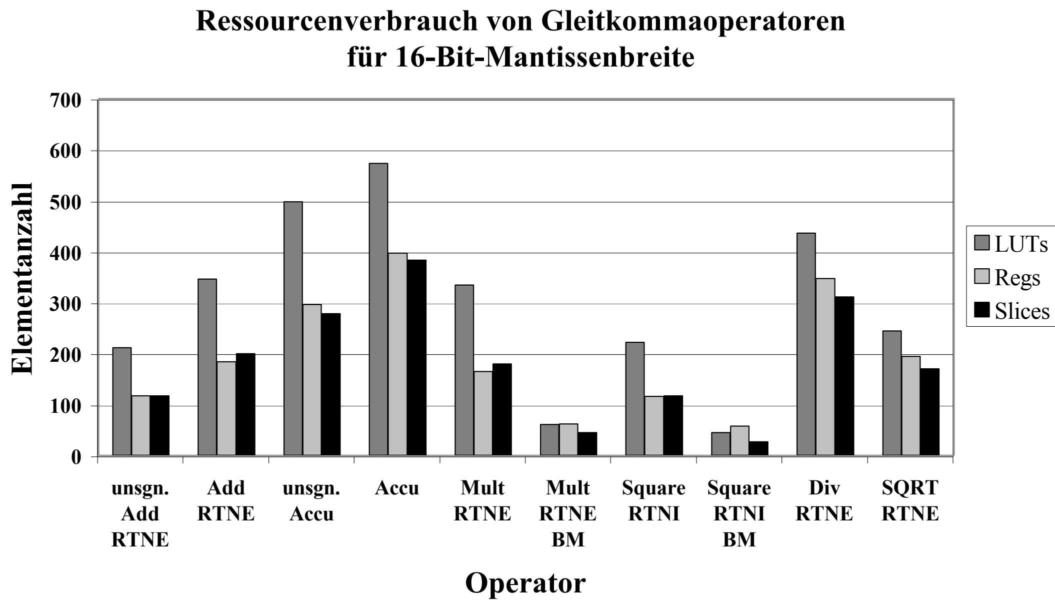
ander gegenübergestellt, wobei die Zahlen für den jeweils besten implementierten Rundungsmodus verglichen werden[7]. Für die Dividierer und Quadratwurzel-Operatoren wurde eine Pipelinetiefe von zwei Addier/Subtrahier-Stufen zwischen den Registerstufen gewählt. Damit erreichen alle aufgeführten Operatoren eine Geschwindigkeit von etwa 90 MHz.

Entscheidend für den Flächenverbrauch auf dem FPGA ist die Anzahl an Slices. Es sei an dieser Stelle daran erinnnert, dass eine Slice zwei 4-Input-LUTs und zwei 1-Bit-Register enthält. Liegt die Anzahl der Slices deutlich über der Hälfte der LUTs, bedeutet dies, dass viele Register nicht mit vorangehenden Logikstufen gepaart sind, beispielsweise durch direkt aufeinanderfolgende Pipelining-Register. In diesem Fall wird noch tieferes Pipelining den Flächenverbrauch weiter steigern. Sehr deutlich ist dies bei den Dividierern und Quadratwurzel-Operatoren zu sehen. Die Addierer und Multiplizierer haben dagegen ein sehr gutes Verhältnis von Slices zu LUTs. Hier gibt es noch Spielraum für eine Geschwindigkeitsoptimierung, ohne den Flächenverbrauch wesentlich zu erhöhen.

Besonders auffallend ist die Skalierung des Ressourcenaufwands. Während die Addierer und Akkumulatoren bei 24-Bit- gegenüber 16-Bit-Mantissen einen Mehraufwand an Slices von etwa 30–40 % verursachen, ergibt sich bei den übrigen Operatoren etwa eine Verdoppelung. Deutlich wird auch die enorme Einsparung von Logikressourcen für die Multiplizierer und Quadrierer, wenn die Block-Multiplizierer-Elemente des Virtex-II-FPGAs verwendet werden. Allerdings ist zu berücksichtigen, dass für 24-Bit-Mantissen vier 18×18-Bit-Multipliziererelemente für die Multiplizierer und drei dieser Elemente für die Quadrierer benötigt werden, während nur jeweils einer dieser Bausteine bei 16-Bit-Mantissen gebraucht wird.

In Tabelle 6.19 werden die Implementierungsergebnisse für die Operatoren mit 24-Bit-Mantissen und Rundungsmodus *Round To Nearest Even* mit den Leistungsdaten kommerzieller Produkte verglichen. Für den Vergleich wurden die FPGA-IP-Cores der Firmen *Nallatech Limited* [81] und *QinetiQ Limited* (Quixilica Cores, [94]) angeführt. Die Operatoren des Autors weisen in allen Fällen einen geringeren Ressourcenverbrauch auf als die kommerziellen Produkte. Al-

[7]Mit Ausnahme der Quadrierer wurde der Standardmodus *Round To Nearest Even* verwendet. Bis auf extrem seltene Ausnahmen führt der Modus *Round To Nearest Integer* im Fall der Quadrierer zum gleichen Rechenergebnis, siehe dazu die Diskussion der Gleitkomma-Quadrierer.



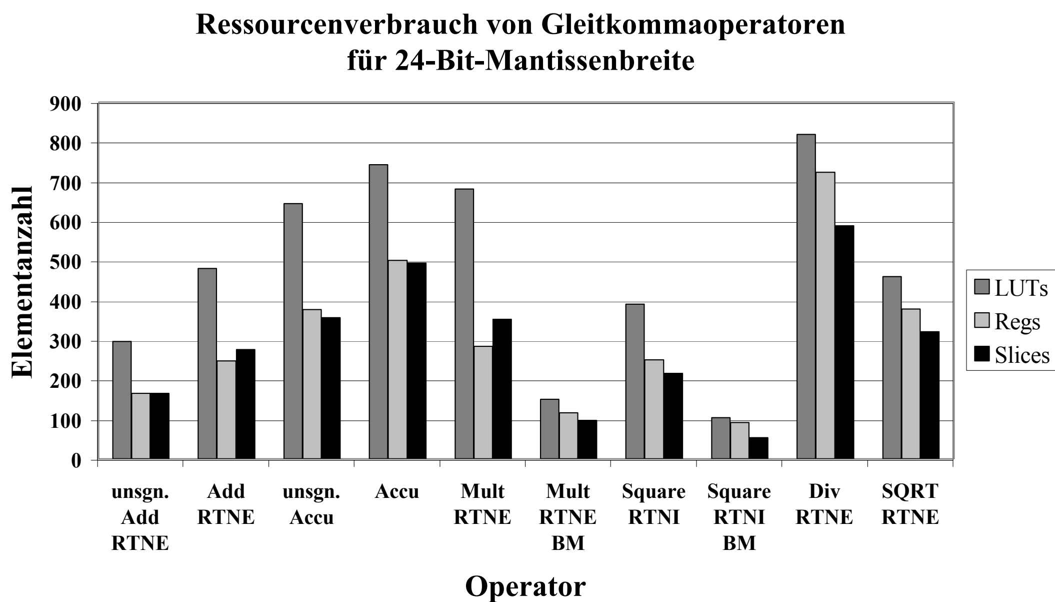


Abbildung 6.28: Vergleich des Ressourcenaufwands für verschiedene Gleitkommaoperatoren für 16-Bit- und 24-Bit-Mantissen für die Implementierung auf Virtex-II-FPGAs. “BM” bezeichnet die Verwendung von Block-Multplizierern des Virtex-II-FPGAs. Die Bezeichnungen “RTNE” und “RTNI” stehen für die Rundungsmodi *Round To Nearest Even* und *Round To Nearest Integer*.

lerdings werden auch deutlich niedrigere Taktfrequenzen erreicht. Wie an den Zahlen für die Latenzzeiten (in Taktzyklen) zu sehen ist, wurde in den kommerziellen Produkten ein etwa doppelt so tiefes Pipelining der Operatoren implementiert und so ergibt sich ein Geschwindigkeitsvorteil um bis zu einen Faktor 2. Für die Addierer und Quadratwurzel-Operatoren werden bei Nallatech trotz erheblich größerer Pipelinetiefe nur wenig mehr Logikressourcen benötigt. Der Grund dafür ist, dass Nallatech intern ein eigenes 38-Bit-Zahlenformat verwendet, sodass die Rundungs- und Normalisierungs-Logik vereinfacht werden kann. Die Quixilica-Designs sind in der Rechengenauigkeit skalierbar, wie die Designs des Autors. Die Module von Nallatech sind dagegen auf Single-Precision festgelegt.

Für diese Arbeit war vor allem wichtig, eine in der Rechengenauigkeit skalierbare Bibliothek von Gleitkommaoperationen mit möglichst geringem Logikressourcenaufwand für die Operatoren aufzubauen. Motivation dafür war, mit den gegebenen FPGAs auch sehr komplexe Formeln vollständig parallelisiert implementieren zu können. Dieses Ziel wurde sehr gut erreicht. Alle Operatoren des Autors weisen ein ausgewogenes Verhältnis von Geschwindigkeit und Ressourcenverbrauch auf und selbst mit einer Genauigkeit von Single-Precision kann mit voller Geschwindigkeit des PCI-Bus (66 MHz) gerechnet werden. Im folgenden Abschnitt wird sich zeigen, dass aufgrund der limitierten Speicherbandbreite auf der verwendeten Koprozessorplattform derzeit eine Erhöhung der Taktfrequenz für die Rechenwerke über 66 MHz hinaus ohnehin nicht sinnvoll erscheint.

Wird für zukünftige FPGA-Designs eine Erhöhung der Taktfrequenz erforderlich, bieten die derzeitigen FPGA-Designs des Autors noch reichlich Potential für eine Geschwindigkeitssteigerung.

Kapitel 7

Implementierung des Rechenbeschleunigers

7.1 Implementierung der Rechenwerke für SPH

In diesem Abschnitt werden die FPGA-Designs der Rechenwerke für eine Prototypimplementierung eines Rechenbeschleunigers beschrieben. Es wird von einer SPH-Formulierung ausgegangen, wie in Abschnitt 2.3.2 vorgestellt. Die für die SPH-Simulation zentralen Formeln der Dichteberechnung (Schritt 1) und Kraftberechnung (Schritt 2) wurden vollständig parallel implementiert. Details der Umsetzung werden in den Abschnitten 7.1.3 und 7.1.4 ausgeführt. Zuvor werden in Abschnitt 7.1.1 die Rechenwerke für den Spline-Kernel und dessen Gradienten vorgestellt, welche Kern der Implementierung von SPH-Formeln sind. Dieser Abschnitt wurde von der Beschreibung der SPH-Rechenwerke separiert, da für die Umsetzung der Kernfunktionen spezielle Ressourcen sparende Operatorbausteine verwendet wurden, auf welche kurz eingegangen werden soll. Demgegenüber sind die in den Abschnitten 7.1.3 und 7.1.4 beschriebenen Rechenwerke im Wesentlichen aus bereits diskutierten Bausteinen zusammengesetzt.

7.1.1 Rechenwerke für die Bestimmung der Spline-Kernel-Werte

Berechnung des SPH-Kerns $\mathbf{W(r,h)}$

Es wird hier von der Kernfunktion ausgegangen, wie sie in Abschnitt 2.3.2 in Formel 2.20 als Funktion von $v = \frac{r}{h}$ aufgestellt wurde und hier noch einmal wiedergegeben ist:

$$W(r,h) = \frac{1}{\pi h^3} \begin{cases} 1 - \frac{3}{2}v^2 + \frac{3}{4}v^3 & : \quad 0 \leq v < 1 \\ \frac{1}{4}(2-v)^3 & : \quad 1 \leq v < 2 \\ 0 & : \quad sonst. \end{cases} \tag{7.1}$$

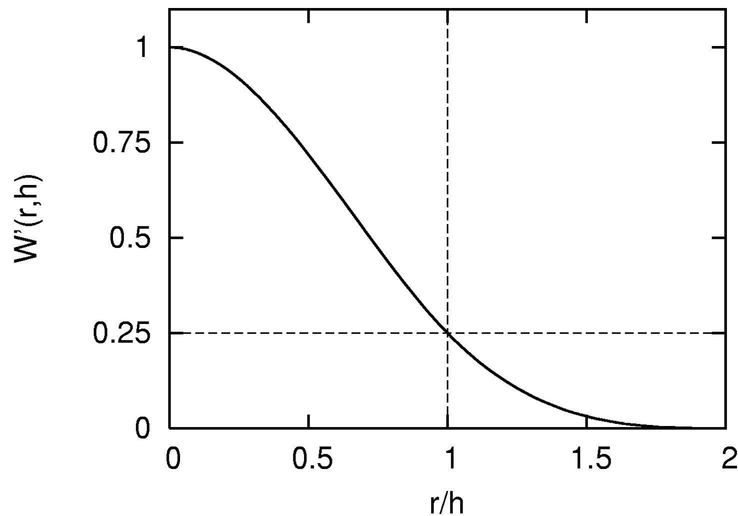


Abbildung 7.1: Spline-Kernfunktion für SPH entsprechend Formel 7.2.

Lässt man den Skalierungsfaktor $\frac{1}{\pi h^3}$ weg, ergibt sich folgende skalare Funktion, welche im Intervall $[0,2)$ ungleich Null ist:

$$W'(x) = \begin{cases} 1 - \frac{3}{2}x^2 + \frac{3}{4}x^3 & : \quad 0 \leq x < 1 \\ \frac{1}{4}(2-x)^3 & : \quad 1 \leq x < 2 \\ 0 & : \quad sonst \end{cases} \tag{7.2}$$

Abbildung 7.1 zeigt den Graphen dieser Funktion. Zur Implementierung können hier beispielsweise die Table-Look-Up-Methoden, welche in 4.5.1 diskutiert wurden, verwendet werden. Stattdessen wurde der Weg gewählt, eine direkte Implementierung durch Gleitkommaoperatoren durchzuführen. Dies hat folgende Vorteile:

- Architekturunabhängigkeit, da keine Block-RAM- oder externe RAM-Ressourcen benötigt werden.

- Gute Skalierbarkeit in der Rechengenauigkeit, da lediglich die Genauigkeit der Einzeloperatoren angepasst werden muss. Für eine LUT-basierte Implementierung müssten die Look-Up-Tables für jede Operandenbreite neu berechnet werden. Zudem vergrößert sich die LUT-Tiefe exponentiell mit der Genauigkeit.

- Leicht verifizierbare Korrektheit der Implementierung, da Schaltungsfehler bereits bei einer geringen Zahl von Testrechnungen sichtbar werden. Bei LUT-basierten Implementierungen ist die Verifizierung weit schwieriger (prominentes Beispiel: Pentium-Bug).

Durch die Anwendung von Spezialoperatoren konnten die Ressourcenanforderungen der direkten Implementierung auf ein Niveau vergleichbar mit zwei Multiplikationen und zwei Additionen

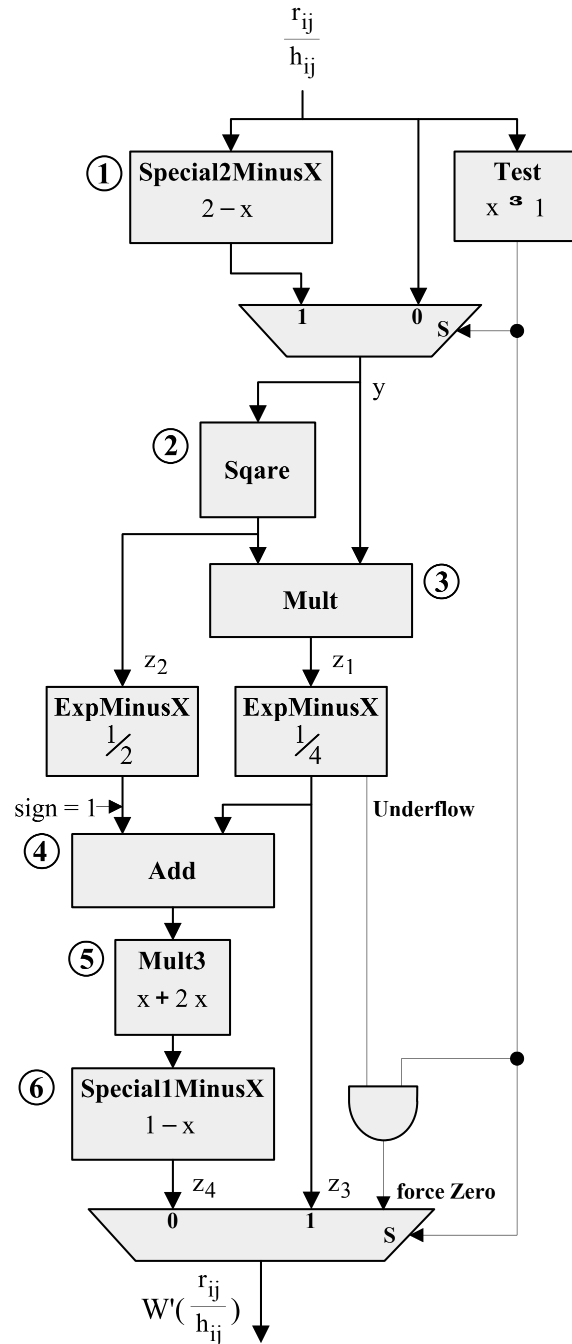


Abbildung 7.2: Schaltung zur Berechnung der Spline-Kernfunktion nach Gleichung 7.2.

geführt werden, was ungefähr dem Aufwand für die Interpolationseinheit einer LUT-basierten Näherung zweiten Grades entspricht.

Die Implementierung des Rechenwerks für den Spline-Kernel zeigt Abbildung 7.2. Es wird vorausgesetzt, dass das Teilchen mit Index j ein echter SPH-Nachbar von Teilchen i ist, was bedeutet, dass $x = r_{ij}/h_{ij} < 2$ gilt[1]. Die Fallunterscheidung, ob $x < 1$ oder $x \geq 1$ gilt, wird im Modul *Test* durchgeführt. Darin ist lediglich zu testen, ob der Exponent von x identisch mit dem Exponenten-Bias der Gleitkommadarstellung ist. Im Fall $x < 1$ kann die Berechnung von W' nach Gleichung 7.2 in folgende Operationen zergliedert werden, wobei die eingekreisten Nummern auf die in Abbildung 7.2 gleichermaßen gekennzeichneten Operatoren verweisen.

$$\begin{aligned} y &= x & \\ z_2 &= y^2 & ② \\ z_1 &= z_2\, y & ③ \\ z_2' &= z_2 \cdot \tfrac{1}{2} & \\ z_1' &= z_1 \cdot \tfrac{1}{4} & \\ z_{12} &= z_1' - z_2' & ④ \\ z_{12}' &= 2 \cdot |z_{12}| + |z_{12}| & ⑤ \\ z_4 &= 1 - z_{12}' & ⑥ \\ W(x) &= z_4. & \end{aligned} \tag{7.3}$$

Für $1 \leq x < 2$ ergibt sich dagegen folgender Ablauf:

$$\begin{aligned} y &= 2 - x & ① \\ z_2 &= y^2 & ② \\ z_1 &= z_2\, y & ③ \\ z_3 &= z_1 \cdot \tfrac{1}{4} & \\ W(x) &= z_3. & \end{aligned} \tag{7.4}$$

Für Operator ① (*Special2MinusX*) wurde eine spezielle Variante eines Addierers implementiert. Dieser Operator wird nur benötigt, wenn $1 \leq x < 2$ gilt. Die Zahl x entspricht dann der Mantisse. Somit kann ein Festkomma-Subtrahierer angewandt werden, der sich aufgrund des zweiten Arguments, welches identisch 2 ist, auf eine 2er-Komplement-Logik reduziert. Abschließend ist die Differenz wie bei der Gleitkommaaddition zu normalisieren, wobei der Fall der Rechtsverschiebung nicht auftreten kann.

Bei den Operatoren ②, ③ und ④ handelt es sich um normale Gleitkommaoperationen, die gegen Underflow-Ausnahmen abgesichert sein müssen.

Der Operator ⑤ (*Mult3*) ist eine spezielle Version eines Addierers für positive Zahlen. Die Auswertung der Exponenten und Verschiebung der Mantissen in der *Preparation*-Stufe kann entfallen. Nach der Addition der Mantisse zur um eine Stelle linksverschobenen Mantisse geschieht die Normalisierung wie im Fall des Addierers für positive Zahlen aus Abschnitt 6.2.2.

[1] Falls eine Routine zur Bereitstellung der Nachbarindizes dies nicht garantieren kann, ist es erforderlich, den Fall $x \geq 2$ zu erkennen und daraufhin die Ausgabe von $W'(x) = 0$ zu erzeugen.

Für den Operator ⑥ kann ebenfalls eine Ressourcen sparende spezielle Addierervariante implementiert werden. Denn es gilt die Beziehung $z'_{12} <= 3/4$. Damit vereinfacht sich die Normalisierungsstufe erheblich, denn es kann höchstens eine Verschiebung um zwei Stellen nach links erforderlich werden.

Die Divisionen durch 4 und 2, welche auf z_1 und z_2 anzuwenden sind, werden durch eine einfache Subtraktionsoperation auf den Exponenten dieser Zahlen implementiert, wofür nur sehr wenig Logikressourcen benötigt werden. Bei der Division $z_1/4$ muss ein auftretender Exponent-Underflow festgestellt werden, um für $x \geq 1$ gegebenenfalls das Ergebnis W' auf Null zu setzten. Bis auf diesen Fall braucht keine weitere spezielle Aktion bei auftretenden Underflow-Ausnahmen implementiert zu werden. Falls im Logikpfad zur Bestimmung von z_4 an irgendeiner Stelle ein Underflow auftritt, ist der in den Operator ⑥ eingehende Exponent so klein, dass automatisch $z_4 = 1$ ausgegeben wird. Auf der anderen Seite wird jeder auftretende Underflow im Logikpfad für z_3 zu einem erneuten Underflow in *ExpMinusX* führen, welcher wie beschrieben behandelt wird.

Berechnung des Gradienten $\nabla \mathbf{W}(\mathbf{r},\mathbf{h})$

Der Gradient des Spline-Kernes aus Gleichung 7.1, der insbesondere für die Kraftberechnung der SPH-Methode benötigt wird, berechnet sich wie folgt ($x = |\vec{r}_i - \vec{r}_j|/h$):

$$\begin{aligned} \nabla W(|\vec{r}_i - \vec{r}_j|, h) &= \frac{1}{\pi\, h^5}\, \vec{r}_{ij} \begin{cases} -3 + \frac{9}{4}\,x & : \; 0 \leq x < 1 \\ -\frac{3}{4}\,x + 3 - \frac{3}{x} & : \; 1 \leq x < 2 \\ 0 & : \; sonst \end{cases} \\ &= \frac{3}{\pi\, h^5}\, \vec{r}_{ij}\, \Omega(x). \end{aligned} \tag{7.5}$$

Der Kern dieser Funktion ist die skalare Funktion $\Omega(x)$, die hier noch einmal separat aufgestellt wird:

$$\Omega(x) = \begin{cases} -1 + \frac{3}{4}\,x & : \; 0 \leq x < 1 \\ -\frac{1}{4}\,x + 1 - \frac{1}{x} & : \; 1 \leq x < 2 \\ 0 & : \; sonst. \end{cases} \tag{7.6}$$

Abbildung 7.3 zeigt den Verlauf dieser Funktion. Sie wurde nicht aus Modulen für die Standardoperationen zusammengesetzt, sondern es wurde dafür ein spezieller Gleitkomma-Operator aufgebaut, um optimal von den Eigenschaften dieser speziellen Funktion zu profitieren. Abbildung 7.4 zeigt die Umsetzung der Schaltung. Es wurden zwei getrennte Rechenwerke zur Berechnung von $1 - 3/4x$ (links) und $1/4\,x + 1/x - 1$ (rechts) implementiert. Die Unterscheidung, welche Formel angewandt wird, kann allein aufgrund des Exponenten getroffen werden. Liegt die Zahl im Intervall $[1,2)$, ist der Exponent der Zahl identisch mit dem Bias der Exponentendarstellung. In diesem Fall wird durch einen Multiplexer das Ergebnis der rechten Schaltung ausgewählt, ansonsten das der linken Schaltung. Kann eine Eingangszahl größer oder gleich 2 auftreten, ist dies durch Test auf $e > bias$ festzustellen und in diesem Fall die Ausgabe auf Null zu setzen (in der Schaltung nicht gezeigt).

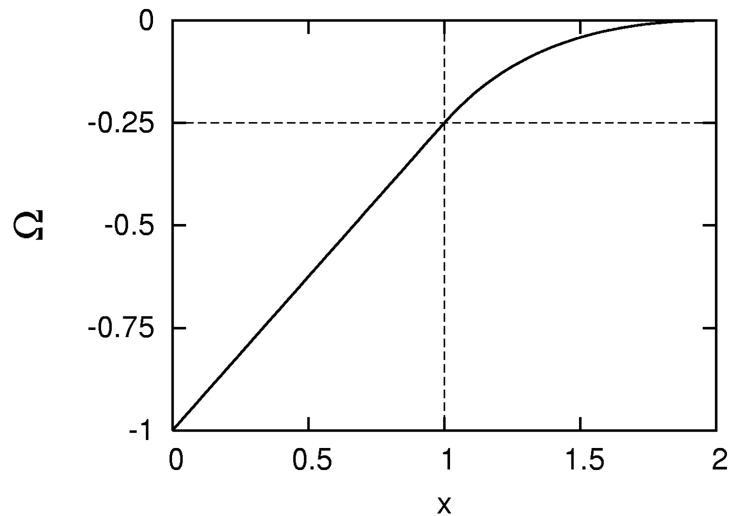


Abbildung 7.3: Kernfunktion Ω für SPH entsprechend Formel 7.6.

Linkes Rechenwerk Dieses Rechenwerk ist eine Abwandlung des Operators *Special1MinusX* aus Abbildung 7.2 (Operator ⑥), denn es wird ebenfalls eine Zahl $\leq 3/4$ von 1 subtrahiert. Hier wird jedoch zusätzlich die Berechnung des Produktes $3/4\,x$ in den Operator integriert, da diese Operation dann über eine einfache Festkommazahlberechnung auf der eingehenden Mantisse s ausgeführt werden kann. Die Multiplikation von x mit drei erfolgt durch die Addition der um eine Stelle linksverschobenen Mantisse s zu s. Die resultierende Zahl (mit MSB der Wertigkeit 2^2) wird über das Modul *ShiftN* rechtsverschoben, genauso wie es in der *Preparation*-Stufe bei allen Gleitkomma-Addierern durchgeführt wird. Die Verschiebeweite ergibt sich hier aus der Differenz des Exponenten-Bias mit dem eingehenden Exponenten e. Die Division durch 4 erfolgt durch die Interpretation der Shifter-Ausgabe als Zahl mit einem MSB der Wertigkeit 2^0, wobei die beiden LSBs dann die Stelle eines Guard- und Round-Bits annehmen. Die Differenz $1 - 3/4x$ ergibt sich dann durch Bildung des Zweierkomplements und Invertierung des MSBs. Das Ergebnis kann maximal zwei führende Nullen aufweisen, welche in der Normalisierungseinheit eine entsprechende Linksverschiebung induzieren. Entsprechend den Werten der beiden MSBs kann der Exponent des Resultats über eine Look-Up-Table mit drei Einträgen in *LookupExponent* gefunden werden. Um Ressourcen zu sparen, wurde keine Rundungsoperation implementiert.

Rechtes Rechenwerk Da das rechte Rechenwerk nur für $x \in [1,2)$ zum Zuge kommt, entfällt der *Preparation*-Schritt. Die aufwändigste Operation dieses Rechenwerkes ist die Kehrwertberechnung der Mantisse. Diese wird mit einer Genauigkeit von $M+1$ Stellen durchgeführt. Die Addition von $1/4\,x$ erfolgt über einen einfachen Ripple-Carry-Addierer, die nachfolgende Subtraktion von 1 durch die Invertierung des MSB des Additionsresultats. Die resultierende Festkommazahl liegt im Intervall $[0.25,0]$. Es kann also eine Linksverschiebung um $M+1$ Stellen notwendig werden, mindestens jedoch um zwei Stellen. Letzteres impliziert, dass bereits der Addierer um zwei führende Stellen gekürzt werden kann und auch die Operation *InvertMSB* ent-

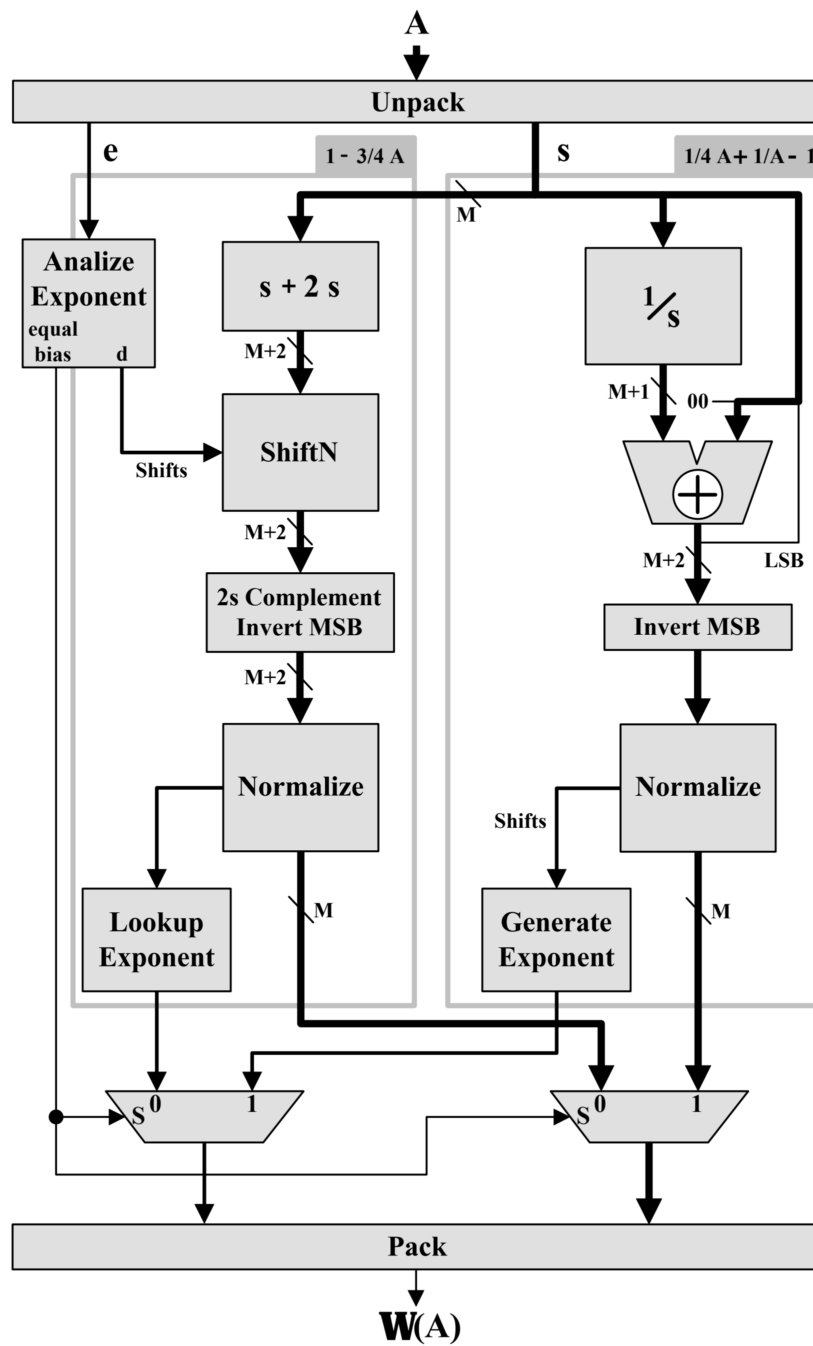


Abbildung 7.4: Spezialoperator zur Berechnung der Kernfunktion Ω für SPH entsprechend Formel 7.6.

behrlich ist (aus Gründen der Verständlichkeit wurde diese Vereinfachung nicht in Abbildung 7.4 gezeigt). Der Ergebnisexponent wird alleine durch die Verschiebeweite bestimmt. Er wird im Element *GenerateExponent* gebildet. Eine Rundung des Ergebnisses erfolgt wie im linken Rechenwerk nicht.

7.1.2 Designmethode für die Rechenwerke

Alle FPGA-Designs wurden in VHDL entwickelt. Um die aus insgesamt über 100 Gleitkomma-operatoren bestehenden Designs sowohl für die Erzeugung der Codes als auch für die Simulation der Schaltungen beherrschbar zu machen, wurden folgende Methoden angewandt:

- Die Schaltungen wurden hierarchisch aufgebaut. Es wurden nur vier Hierarchiestufen verwendet: die elementaren Operatoren (z.B. Addition, Multiplikation), die zusammengesetzten Operatoren (z.B. Skalarprodukt), Formelteile (z.B. Berechung der Glättungskerne) und das Gesamtrechenwerk. Dadurch wurde ein Kompromiss zwischen der Codelänge für die Module und der Übersichtlichkeit der Signale bei der Simulation erreicht.

- Für die Ebene der Formelteile und des Gesamtrechenwerks wurde ein Design-Schema angewandt, welches durch Abstraktion der Gleitkommazahlsignale und Vereinheitlichung des Codes für das Verbinden von Operatoren eine übersichtliche und leicht zu verifizierende Design-Erstellung erlaubt.

Der zweite Punkt soll nun noch etwas weiter ausgeführt werden. Die Simulation eines komplexen Rechenwerks ist extrem schwierig, wenn die Zwischenergebnisse nur als Bitvektoren vorliegen. Deshalb wurde eine abstrahierende Datenstruktur für die Zwischenergebnisse implementiert, die für die Syntheseversion die Bitvektoren für Mantisse, Exponent und Vorzeichen enthält. Für die Simulation ist dagegen zusätzlich ein Gleitkommafeld vorhanden, welches zu jeder Zeit die durch die Bitvektoren dargestellte Zahl anzeigt.

Einzelne Operatoren werden verbunden, indem über einheitliche VHDL-Prozeduren eine Verknüpfung der abstrahierten Gleitkommasignale zu den Ein- und Ausganssignalen der Operatoren erzeugt wird. Diese Signale bleiben vorerst ebenfalls in der abstrahierten Darstellung, und werden erst bei der Erzeugung der Operatoren aufgelöst. Auf diese Weise lässt sich sehr intuitiv die Verbindungsstruktur des Rechenwerks programmieren und der große Overhead an Code zur Erzeugung der Operatoren wird in einen separaten Codeabschnitt verschoben.

In diesem von der strukturellen Beschreibung abgesonderten VHDL-Code werden die Instanzen der Operatoren durch ein festes Schema implementiert. Die Ein- und Ausgangssignale haben für alle Operatoren die gleichen Bezeichnungen und unterscheiden sich nur in der Operatornummer. Die noch abstrahierten Eingangssignale werden in Bitvektoren aufgelöst und mit dem betreffenden Operator verbunden. Die Ausgangssignale werden sofort wieder in eine abstrahierte Darstellung überführt. Auf diese Weise kann die Operatorinstanziierung sehr schnell anhand von Vorlagen geschehen.

Die Operatoren können für verschiedene Mantissenbreiten verschiedene Latenzzeiten aufweisen. Deshalb wurde auch die Erzeugung von Ausgleichsgliedern zwischen den Operatoren

schematisiert. Zu jedem Operator gibt es eine Funktion, welche in Abhängigkeit von den Operatorparametern die Latenzzeit zurückgibt. Anhand dieser Funktionen wird jedem Zwischenergebnis eine Latenzzeit zugeordnet. Basierend auf diesen Zahlen wird das Einfügen von Ausgleichsgliedern durch ein ähnliches Abstraktionsschema umgesetzt wie bei der Erzeugung der Operatoren.

Das Verfahren kann automatisiert werden, indem beispielsweise anhand der abstrahierten Signale nur die Verbindungsstruktur programmiert wird und der Rest durch einen Codegenerator erzeugt wird. Diese Automatisierung wurde noch nicht implementiert.

7.1.3 Rechenwerk für die SPH-Dichteberechnung

Struktur

Wie in der Formulierung nach Abschnitt 2.3.2 sind im ersten Schritt des SPH-Formalismus folgende Formeln zu berechnen:

$$\rho_i = \sum_j m_j \, W(|\vec{r_{ij}}|/h_{ij}) \tag{7.7}$$

$$\rho_i \, (\nabla \cdot \vec{v})_i = \sum_j m_j \, (\vec{v_j} - \vec{v_i}) \, \nabla_i W(|\vec{r_{ij}}|/h_{ij}) \tag{7.8}$$

$$\rho_i \, (\nabla \times \vec{v})_i = \sum_j m_j \, (\vec{v_i} - \vec{v_j}) \times \nabla_i W(|\vec{r_{ij}}|/h_{ij}). \tag{7.9}$$

Zur Implementierung der Summationsvorschrift nach Gleichung 7.7 ist anzumerken, dass die Kernfunktion $W(|\vec{r_{ij}}|/h_{ij})$ über die Gleichung $W(|\vec{r_{ij}}|/h_{ij}) = 1/\pi \cdot W'(|\vec{r_{ij}}|/h_{ij})/h_{ij}^3$ implementiert wurde (siehe Beschreibung zu Gleichung 7.2). Der Vorfaktor $1/\pi$ wurde weggelassen, was leicht im Steuerungsrechner korrigiert werden kann. Der Gradient von W wird nach der in Gleichung 7.5 gegebenen Formel berechnet. Auch hier wird die Multiplikation des Vorfaktors $3/\pi$ auf eine Korrekturmultiplikation im Steuerungsrechner verschoben. Es ergibt sich damit folgendes Rechenschema für die Gleichungen 7.7 bis 7.9:

$$\pi \, \rho_i = \sum_j \frac{m_j}{h_{ij}^3} \, W'(|\vec{r_{ij}}|/h_{ij}) \tag{7.10}$$

$$-\frac{\pi}{3} \rho_i \, (\nabla \cdot \vec{v})_i = \sum_j \frac{m_j}{h_{ij}^5} \, (\vec{v_{ij}} \cdot \vec{r_{ij}}) \, \Omega(|\vec{r_{ij}}|/h_{ij}) \tag{7.11}$$

$$\frac{\pi}{3} \rho_i \, (\nabla \times \vec{v})_i = \sum_j \frac{m_j}{h_{ij}^5} \, (\vec{v_{ij}} \times \vec{r_{ij}}) \, \Omega(|\vec{r_{ij}}|/h_{ij}). \tag{7.12}$$

Ausschlag gebend für die Architektur der Implementierung des Rechenwerkes für diese Formeln war die Maßgabe, möglichst wenig Ressourcen zu verwenden und daher die Anzahl an Operatoren zu minimieren. Gemäß dieser Regel bleibt nur eine geringe Gestaltungsfreiheit, die sich im Wesentlichen auf die Anordnung nacheinander folgender kommutierender Operationen begrenzt. In Abbildung 7.5 ist der Aufbau der resultierenden Implementierung zu sehen.

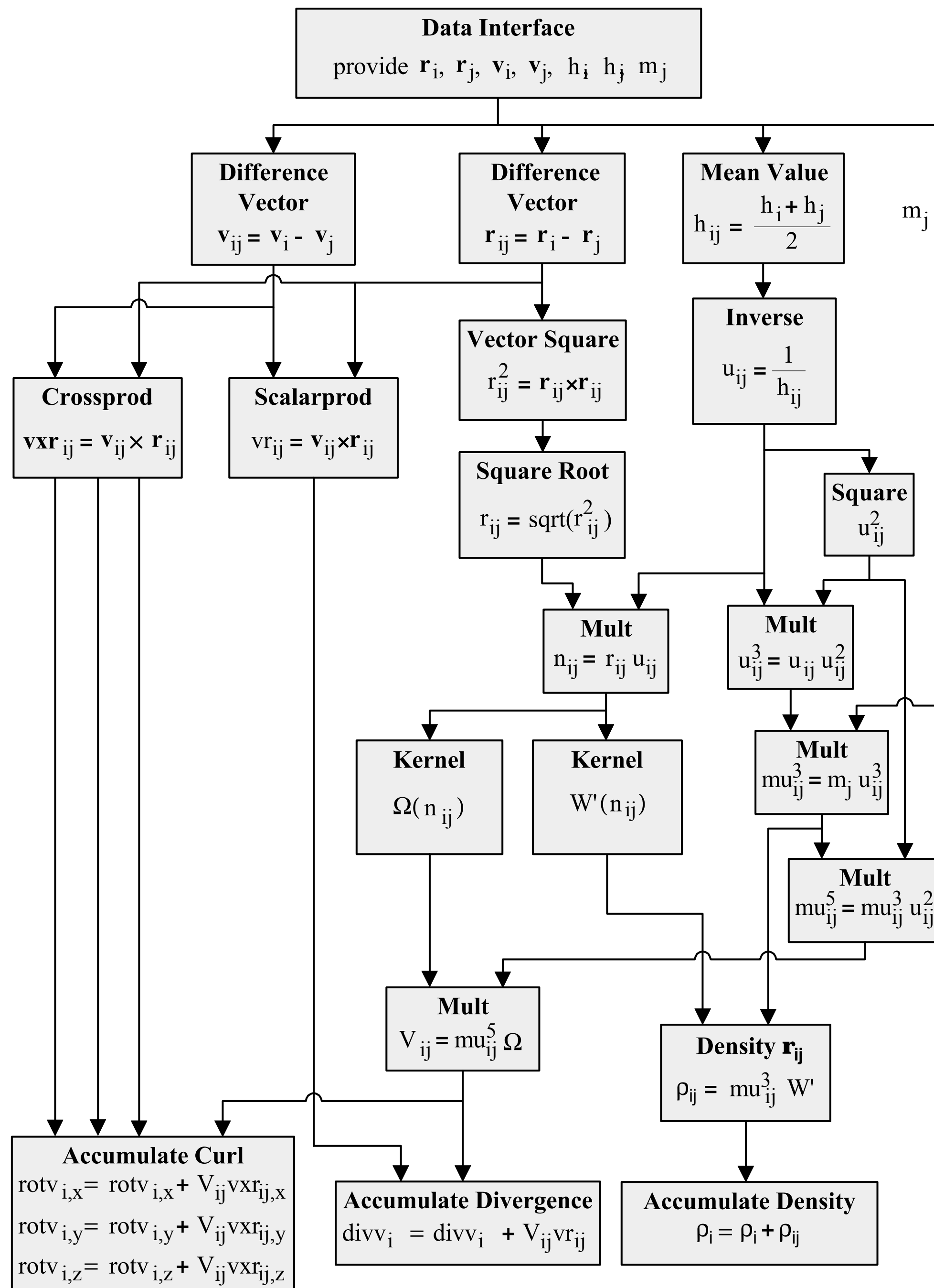


Abbildung 7.5: Blockschaltbild zur SPH-Dichteberechnung.

Bewertet man die Berechnung von Ω nach Gleichung 7.6 mit vier Operationen, sind für das gesamte Rechenwerk 54 elementare Gleitkommaoperatoren zu implementieren. Darunter sind 20 Multiplizierer, 5 Quadrierer, 15 Addierer, 5 Akkumulatoren, jeweils ein Dividierer und ein Quadratwurzel-Operator und diverse spezielle Operatoren für die Kernfunktionen.

Für alle Operationen wurde die gleiche Mantissenbreite implementiert. Bis auf die Berechnung von Ω wurde für alle Operationen der Rundungsmodus *Round To Nearest Integer* verwendet, da die Simulation bei Verwendung dieses Modus gegenüber dem Standardmodus keinerlei Nachteile zeigte, sich jedoch etwa 10 % der FPGA-Ressourcen einsparen lassen. Sollten sich für spätere astrophysikalische Simulationen bezüglich des Rundungsmodus Probleme ergeben kann auch der Modus *Round To Nearest Even* verwendet werden, da wie im letzten Kapitel beschrieben auch dieser Modus von den implementierten Gleitkommaoperationen unterstützt wird.

Implementierungsaufwand

Die Implementierung des Rechenwerks für den ersten Schritt des SPH-Algorithmus führte zu folgendem Ressourcenverbrauch, wobei in Klammern die Anteile an den insgesamt zur Verfügung stehenden Ressourcen eines Virtex-II-FPGAs vom Typ XC2V3000 notiert sind:

10618	LUTs	(37 %)
8196	1-Bit-Register	(29 %)
25	18×18-Bit-Blockmultiplizierer	(26 %)
6924	Slices	(48 %)

Für das Rechenwerk wird also etwas weniger als die Hälfte eines FPGAs mittlerer Größe aufgewendet. Damit gibt es also noch Erweiterungsspielraum für zusätzliche Gleichungen, beispielsweise für die Berücksichtigung zusätzlicher Physik. Laut Synthese-Werkzeug kann das Rechenwerk mit einer Taktfrequenz von 74 MHz betrieben werden. Die unter Verwendung dieses Rechenwerks erzielte Rechenleistung und Rechengenauigkeit wird in Abschnitt 7.4 diskutiert werden.

7.1.4 Rechenwerk für die SPH-Kraftberechnung

Struktur

Für den zweiten Schritt des SPH-Algorithmus wurden die Formeln für die Druckkraft des Gases einschließlich Berücksichtigung der künstlichen Viskosität implementiert. Entsprechend den

Gleichungen 2.32 und 2.35 sind dazu folgende Berechnungen durchzuführen:

$$\frac{d\vec{v}_i}{dt} = -\sum_j m_j \left(\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2} + \Pi_{ij}\right) \vec{\nabla}_i W\left(\left|\vec{r}_{ij}\right| / h_{ij}\right) \tag{7.13}$$

$$\Pi_{ij} = \begin{cases} \dfrac{\left(-\alpha c_{ij}\mu_{ij} + \beta\mu_{ij}^2\right)}{\rho_{ij}} & : \quad \vec{v}_{ij}\vec{r}_{ij} \leq 0 \\ 0 & : \quad \vec{v}_{ij}\vec{r}_{ij} > 0 \end{cases} \tag{7.14}$$

$$\mu_{ij} = \frac{h_{ij}(\vec{v}_{ij} \cdot \vec{r}_{ij})}{\vec{r}_{ij}^2 + \eta^2 h_{ij}^2} f_{ij}. \tag{7.15}$$

Die Größen f_{ij}, c_{ij} und ρ_{ij} sind Mittelwerte aus den Balsara-Faktoren, Schallgeschwindigkeiten und Drücken für Teilchen i und j, genau wie bei h_{ij}. Die Berechnung des Entropieterms aufgrund der künstlichen Viskosität nach Gleichung 2.41 wurde vorerst nicht implementiert. Wird eine isotherme oder isentrope Zustandsgleichung angenommen, ist dieser Term entbehrlich. Soll stattdessen auf Grundlage einer adiabatischen Zustandsänderung simuliert werden, lässt sich dieser Term leicht einbauen. Es werden lediglich zwei weitere Multiplizierer und ein Akkumulator dazu benötigt. Der erste Multiplizierer bildet dann das Produkt der für die Viskosität berechneten Terme $\Pi_{ij} \cdot (\vec{v_{ij}} \cdot \vec{r_{ij}})$, welches danach mit dem ebenfalls bereits für die Druckkraftberechnung bestimmten Faktor $m_{ij}/h_{ij}^5\,\Omega$ zu multiplizieren ist. Das Schema entspricht genau der Vorgehensweise zur Berechnung von $\rho_i(\vec{\nabla} \cdot \vec{v})_i$ im letzten Abschnitt (siehe Gleichung 7.11).

Für Gleichung 7.13 kann folgendes Rechenschema angewendet werden:

$$\frac{\pi}{3}\frac{d\vec{v}_i}{dt} = -\sum_j \frac{m_j}{h_{ij}^5} \left(\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2} + \Pi_{ij}\right) \Omega\left(|\vec{r_{ij}}|/h_{ij}\right). \tag{7.16}$$

Für diese Berechnung müssen 52 Operatoren implementiert werden, wobei nur die "großen" Operatoren gezählt wurden. Rechnet man entsprechend der Operatorzahl einer Softwareimplementierung, wo auch Divisionen durch Potenzen von 2 eine volle Gleitkommaoperation bedeuten, kommt man auf 60 Operationen. Von den elementaren Operatoren werden drei Akkumulatoren, 9 allgemeine Addierer, 8 Addierer für positive Zahlen, 14 Multiplizierer, 10 Quadrierer, drei Dividierer und ein Quadratwurzel-Operator benötigt. Hinzu kommt der Ω-Operator aus Abschnitt 7.1.1. In Abbildung 7.6 ist die resultierende Schaltung skizziert. Wie für den ersten Schritt des SPH-Algorithmus wurde auch hier für alle Operatoren die gleiche Mantissenbreite verwendet. Es wurde ebenfalls wieder, mit Ausnahme der Berechnung von Ω, ausschließlich der Rundungsmodus *Round To Nearest Integer* verwendet.

Implementierungsaufwand

Die Implementierung des Rechenwerks für den zweiten Schritt des SPH-Algorithmus führte zu folgendem Ressourcenverbrauch (In Klammern der Anteil an den Gesamtressourcen des Virtex-II XC2V3000):

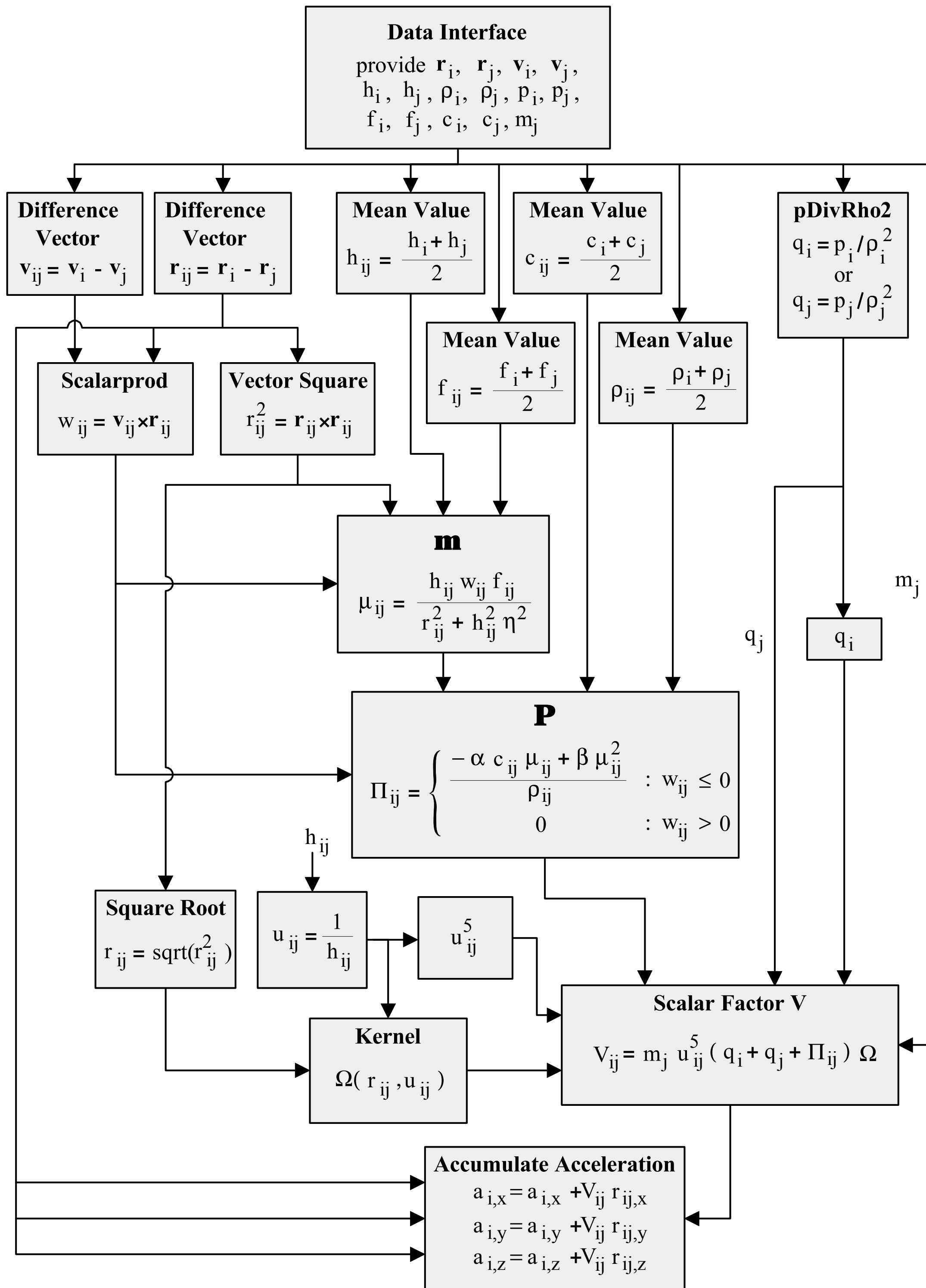


Abbildung 7.6: Blockschaltbild zur Berechnung der Beschleunigung durch den Druckgradienten und künstliche Viskosität.

10900	LUTs	(38 %)
8433	1-Bit-Register	(29 %)
25	18×18-Bit-Blockmultiplizierer	(26 %)
7097	Slices	(49 %)

Für dieses Rechenwerk wird also leicht mehr an FPGA-Ressourcen aufgewendet als für die Schaltung zu Schritt 1 des SPH-Formalismus. Es gibt auch hier noch Erweiterungsspielraum für zusätzliche Berechnungen. Laut Synthese-Werkzeug kann das Rechenwerk ebenfalls mit einer Taktfrequenz von 74 MHz betrieben werden. Zur erzielten Rechenleistung und Rechengenauigkeit sei auf Abschnitt 7.4 verwiesen.

7.2 Einbettung in den FPGA-Prozessor

Es wurden getrennte FPGA-Designs für den ersten und zweiten Schritt des SPH-Algorithmus implementiert. Ein gemeinsames FPGA-Design für beide Schritte wurde für die Prototypimplementierung aus folgenden Gründen nicht angestrebt:

- Die verfügbaren Logikressourcen sind für zwei eigenständige Rechenwerke für Schritt 1 und Schritt 2 nicht ausreichend. Es müssten unter hohem Entwicklungsaufwand Designs erstellt werden, die beide Rechenschritte mit überlappenden Teilrechenwerken umsetzen, was die Effizienz der Implementierung deutlich senken würde.

- Schon aufrund der begrenzten Speicherbandbreite können die beiden Schritte nicht simultan gerechnet werden. Die gleichzeitige Berechnung erscheint aufgrund der Struktur des Simulationscodes auch wenig sinnvoll.

- Die Konfigurationszeit des FPGAs beträgt etwa 40 Millisekunden, wenn sich der Konfigurationsbitstrom bereits im Speicher befindet. Demgegenüber liegt die Rechenzeit pro SPH-Schleife bei typischerweise über 10 Sekunden. Für die Prototypimplementierung bedeutet die Rekonfigurierung deshalb nur eine unwesentliche Verschlechterung der Rechenleistung.

Das FPGA-Design der Rechenbeschleunigerarchitektur wurde unter dem Gesichtspunkt der leichten Portierbarkeit auf andere Systeme aufgebaut. Es wurde ein modulares Design gewählt, bei dem die Rechenwerke logisch von der Peripherie getrennt wurden. Die Rechenpipeline kommuniziert über eine architekturunabhängige Schnittstelle mit einem Interface-Modul (*SPH-Interface*). Dieses implementiert alle plattformspezifischen Schnittstellen zur Umgebung. Dies sind insbesondere die Anbindung an Speicherressourcen und die Verbindung mit dem *FPGA-Interface*, welches mit der Außenwelt kommuniziert. Abbildung 7.7 zeigt die Struktur bei Verwendung des MPRACE-Boards aus Abschnitt 3.3. Es wurden auch Portierungen auf andere Systeme implementiert, insbesondere auf das japanische PROGRAPE-System aus Abschnitt 2.4.2.

Das Modul *SPH Pipeline* nimmt die Rechenwerke für Schritt 1 oder 2, die in den letzten Abschnitten beschrieben wurden, auf. Es verfügt über getrennte Eingänge für die Daten von Teilchen mit Index i und j (Nomenklatur wie bei der Formulierung der SPH-Gleichungen). Als

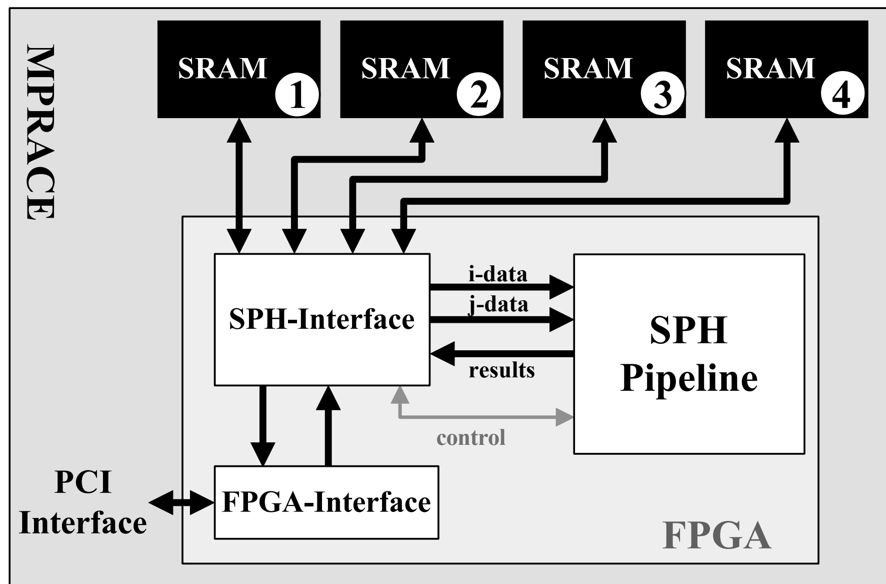


Abbildung 7.7: Modulares Design des Rechenbeschleunigers zur Abstraktion der Hardwareumgebung.

Flusskontrolle wird signalisiert, wann die Daten für eine Berechnung übernommen werden sollen, ob eine neue Summation begonnen wird und wann die letzten Daten eingehen. In Abbildung 7.7 sind diese Signale durch die Bezeichnung *control* angedeutet. Registriert das Modul *SPH Pipeline* das Eingehen eines abschließenden Datensatzes, wird die aktuelle Akkumulation abgeschlossen und das Resultat an *SPH-Interface* übertragen. Die Pufferung der Resultate muss von dieser Schnittstelle übernommen werden. Denn die Pipeline ist dazu ausgelegt, kontinuierlich Berechnungen durchzuführen, weshalb die Akkumulatorinhalte sofort nach Abschluß einer Akkumulation gelöscht werden.

Das MPRACE-Board verfügt über vier unabhängige SRAM-Bänke mit einer Breite von jeweils 36 Bit. Darin werden die Teilchendaten für die SPH-Berechnung in einer Form abgelegt, sodass auf sie parallel zugegriffen werden kann. Für das in der Prototypimplementierung verwendete Gleitkommaformat mit 16-Bit-Mantissen und 8-Bit-Exponenten werden pro Zahl 24 Bit benötigt. Damit kann also maximal auf sechs Werte gleichzeitig zugegriffen werden. Für die Kraftberechnung werden jedoch 12 Teilchenvariablen benötigt. Deshalb wird das Speicherinterface mit doppelter Taktfrequenz relativ zur Pipeline betrieben. Damit können in jeder Taktperiode die benötigten Teilchendaten parallel ausgelesen werden und die Pipeline kann mit voller Auslastung arbeiten. Auf dem MPRACE-Board können die Daten von maximal 65536 Teilchen gespeichert werden.

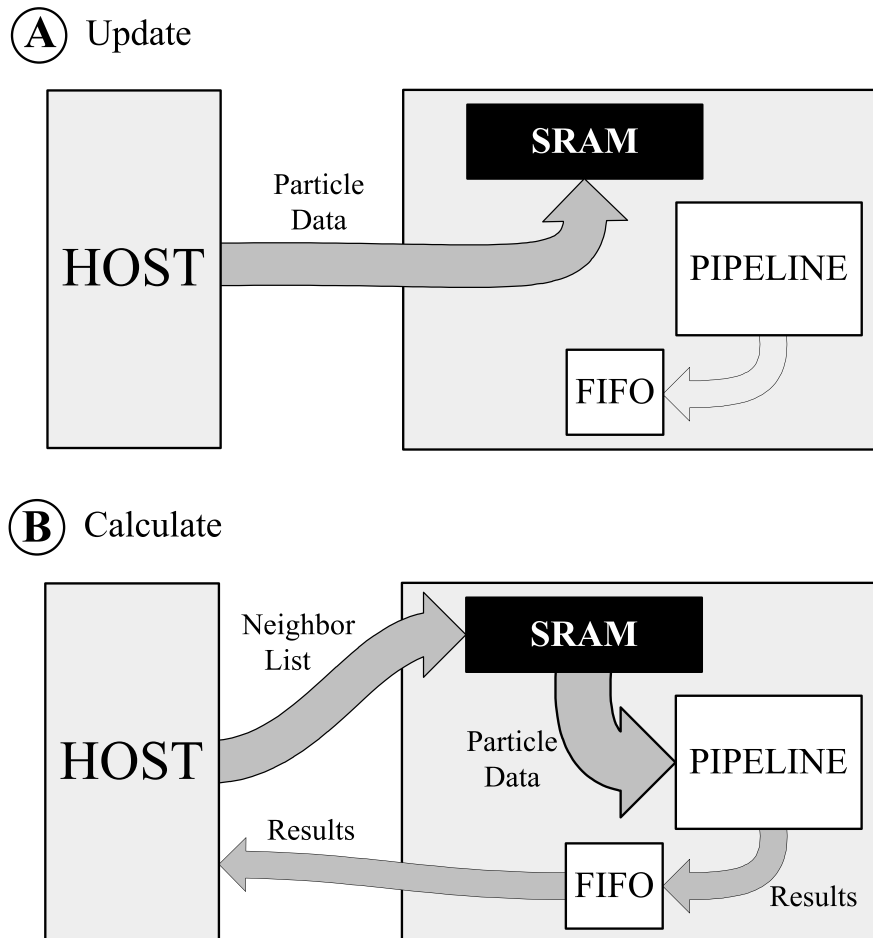


Abbildung 7.8: Datenfluss zwischen Host und Rechenbeschleuniger für die Prototypimplementierung. Vor der Berechnung werden die Teilchendaten aktualisiert (A). Die Berechnung geschieht synchron mit der Übertragung der Nachbarschaftslisten (B).

7.3 Datenfluss und Steuerung der Prototypimplementierung

Aus dem derzeitigen Design des Rechenbeschleunigers ergibt sich ein Kommunikationsschema in zwei Phasen, wie es in Abbildung 7.8 dargestellt ist. In der ersten Phase werden die benötigten Teilchendaten über einen DMA-Zugriff in das SRAM des MPRACE geschrieben. Während dieser Phase ruhen die Rechenwerke. In der zweiten Phase werden die Nachbarschaftslisten der Teilchen transferiert, für welche die SPH-Formeln berechnet werden sollen. Diese Listen bestehen aus den Speicherindices für die Daten der Nachbarteilchen. Diese Indices werden direkt zur Addressierung der Teilchendaten im Speicher verwendet und die so ausgelesenen Daten werden für den nächsten Berechnungsschritt in die SPH-Pipeline geführt. Die Pipeline rechnet synchron mit den Zugriffen auf die Teilchendaten und pro Takt kann eine SPH-Summation durchgeführt werden.

Die Resultate der Rechenpipeline müssen gepuffert werden. Um eine hohe Übertragungs-

	Design mit Pipeline für SPH-Schritt 1	Design mit Pipeline für SPH-Schritt 2
LUTs	11049 (38.5 %)	11139 (38.8 %)
Regs	9028 (31.5 %)	9597 (33.5 %)
Slices	7625 (53.2 %)	7975 (55.6 %)
18×18-Block-Multipizierer	25	25
Design Freq. (MHz)	67	69

Tabelle 7.1: Ressourcenverbrauch und Geschwindigkeit der FPGA-Designs für die Prototypimplementierung auf dem Virtex-II-FPGA XC2V3000-4.

bandbreite zwischen Host und MPRACE zu gewährleisten, wurde von einer DMA-Blockgröße von 8 KByte ausgegangen, wofür sowohl für Schreib- als auch Lesezugriffe etwa 200 MBytes/sec erreicht werden (siehe Abbildung 5.8 aus Kapitel 5). Dann müssen 2000 Resultate zwischengespeichert werden. Da in der Prototypimplementierung die Resultate eine Breite von 24-Bit haben, genügen dafür bereits drei Block-RAM-Elemente des Virtex-II-FPGAs, die als FIFO betrieben werden können. Mit neun Rechenergebnissen pro Nachbarschaftsliste bei der Pipeline zur Dichteberechnung (inklusive Divergenz und Rotation der Geschwindigkeit; positive und negative Ergebnisse werden getrennt akkumuliert) können die Ergebnisse von 227 Berechnungen gepuffert werden. Die zugehörigen Nachbarschaftslisten haben bei durchschnittlich 50 Nachbarn pro Teilchen dann eine Größe von etwa 40 KByte.

7.4 Ergebnisse für den Rechenbeschleuniger

In Tabelle 7.1 sind die Resultate für den Ressourcenverbrauch und die Geschwindigkeit der in Abschnitt 7.2 beschriebenen Prototypimplementierung gezeigt. Die Pipelines zu Schritt 1 und Schritt 2 des SPH-Algorithmus entsprechen genau den in den Abschnitten 7.1.3 und 7.1.4 beschriebenen Schaltungen.

Beide FPGA-Designs können mit der Taktfrequenz des PCI-Bus rechnen. Wird der erste Schritt des SPH-Algorithmus mit 54 Operationen bewertet, resultiert für das erste Design eine Peak-Rechenleistung von 3.5 GFlops. Für das zweite Design ergibt sich bei 60 Operationen eine Spitzenleistung von 3.9 GFlops. Solange Nachbarschaftslisten mit 66 MHz bereitgestellt werden, kann diese Rechenleistung aufrecht erhalten werden. Etwa 20 Prozent der Rechenleistung für Schritt 1 und 15 Prozent für Schritt 2 gehen dadurch verloren, dass die Ergebnisse zurück zum Host transferiert werden müssen. Berücksichtigt man noch den Transfer der Teilchendaten vor der Berechnung (Etwa 20 % des Volumens der Nachbarschaftslisten), findet man, dass die Prototyparchitektur nur mit etwa 50 % Auslastung rechnen kann (ca. 1.75 GFlops für Schitt 1 und 1.95 GFlops für Schritt 2).

Mit der aktuellen Prototyparchitektur sind die Pipelines zwar in der Lage, mit einer Geschwindigkeit von 66 MHz zu arbeiten, jedoch hat sich gezeigt, dass das mit doppelter Taktrate betriebene SRAM des MPRACE nicht bis 133 MHz betrieben werden kann. Es wurde eine Re-

Größe	mittlerer relativer Fehler für FPGA-Implementierung	mittlerer relativer Fehler nach Simulation
ρ	$(2.85 \pm 1.53) \cdot 10^{-5}$	$(2.24 \pm 1.04) \cdot 10^{-5}$
$\vec{\nabla} \cdot \vec{v}$	$(0.84 \pm 0.38) \cdot 10^{-3}$	$(0.88 \pm 0.26) \cdot 10^{-3}$
$\vec{\nabla} \times \vec{v}$	$(0.88 \pm 0.23) \cdot 10^{-3}$	$(0.86 \pm 0.19) \cdot 10^{-3}$
$d\vec{v}/dt$	$(1.3 \pm 0.8) \cdot 10^{-3}$	$(1.2 \pm 1.1) \cdot 10^{-3}$

Tabelle 7.2: Rechengenauigkeit der FPGA-Implementierung für Schritt 1 und Schritt 2 des SPH-Algorithmus im Vergleich mit der Simulation aus Abschnitt 5.3.2. Es wurde mit den gleichen Teilchenverteilungen gerechnet, die für die Simulation in Kapitel 5 verwendet wurden.

duktion der Taktfrequenz auf 50 MHz erforderlich, wodurch sich die Spitzenleistung auf 2.7 GFlops für den ersten SPH-Schritt und 3 GFlops für den zweiten reduziert. Um damit eine Dauerrechenleistung von über 2 GFlops zu erreichen, wie in Abschnitt 5.4 gefordert wurde, muss ein Re-Design des SPH-Algorithmus und der FPGA-Architektur erfolgen, sodass mit komprimierten Nachbarschaftslisten gearbeitet werden kann. Nur so kann das derzeit die Rechenleistung limitierende Speicher- und Kommunikationsbandbreitenproblem gelöst werden. Ein möglicher Ansatz besteht darin, auszunutzen, dass sich die Nachbarschaftslisten benachbarter SPH-Teilchen stark überlappen, also viele der Listeneinträge übereinstimmen. Berücksichtigt man zudem, dass die Akkumulatoren mit geringem Aufwand mit der Fähigkeit ausgestattet werden können, mehrere unabhängige SPH-Summen überlappend auszuführen, sollte sich das Bandbreitenproblem lösen lassen. Diese Strategie wird Bestandteil der weiteren Entwicklungen in diesem Projekt sein.

In Tabelle 7.2 sind die erzielten Rechengenauigkeiten der FPGA-Implementierungen gezeigt. Zur deren Messung wurden die gleichen Bedingungen angesetzt, die schon in Kapitel 5 in Abschnitt 5.3.2 im Zusammenhang mit der Simulation der Rechenwerke vorgestellt wurden. Zum Vergleich wurden in Tabelle 7.2 auch die Simulationsergebnisse aufgeführt. Für die Dichteberechnung ergeben sich leicht erhöhte Rechenfehler, die jedoch gegenüber der Schwankung der Fehler für die verschiedenen Teilchenverteilungen nicht signifikant sind. Die Abweichungen gegenüber der Simulation lassen sich aus dem speziellen Ressourcen sparenden Aufbau des Rechenwerks für den Spline-Kern erklären. Hierfür wurden Spezialoperatoren für $(2-x)$ und $(1-x)$ verwendet (siehe Abschnitt 7.1.1), die in dieser Art nicht simuliert wurden. Für alle übrigen Fehlergrößen aus Tabelle 7.2 lassen sich keine signifikanten Abweichungen feststellen. Die spezielle Architektur zur Berechnung von Ω nach Abschnitt 7.1.1 führt also nicht zu einer Verschlechterung der Genauigkeit.

Die erzielten Ergebnisse bestätigen den Grundgedanken der Arbeit, dass basierend auf der FPGA-Technologie ein Rechenbeschleuniger für SPH-Algorithmen aufgebaut werden kann. Dabei kann die Rechengenauigkeit auf solch hohem Niveau gehalten werden, dass keinerlei Anomalien im dynamischen Verhalten der simulierten Systeme zu befürchten sind. Die erreichbare Rechenleistung genügt für einen deutlichen Speedup der Gesamtanwendung, jedoch bedarf es zur Ausschöpfung der Beschleunigung noch weiterer Entwicklungen an den Simulationsalgorithmen, um die derzeitigen Einschränkungen durch die Datenkommunikation zu überwinden.

Kapitel 8

Zusammenfassung und Diskussion

In dieser Arbeit wurde untersucht, wie das Verfahren *Smoothed Particle Hydrodynamics* zur Simulation der Hydrodynamik astrophysikalischer Systeme durch den Einsatz eines rekonfigurierbaren Koprozessors beschleunigt werden kann. Damit wurde ein wichtiges Teilproblem astrophysikalischer Simulationen unter dem Aspekt der Anwendbarkeit neuer Rechentechniken behandelt. Es konnte ein System aufgebaut werden, welches für die zeitkritischen Berechnungsschritte des SPH-Formalismus eine Rechenleistung von über 3 GFlops erreicht. Daraus ergibt sich für astrophysikalische Simulationen ein Beschleunigungspotential um den Faktor 10. Dieses Potential kann jedoch erst ausgeschöpft werden, wenn die Simulationscodes tiefgreifend verändert werden. Dies wird Gegenstand der weiteren Forschung sein.

Motivation und Strategie

Der Ansatz, Spezialarchitekturen zu verwenden, motiviert sich aus dem grundsätzlichen Problem, dass die Entwicklung von allgemein verwendbaren Rechnersystemen die Untersuchung dringender astrophysikalischer Fragestellungen auf absehbare Zeit nicht ermöglichen wird, obwohl die Entwicklung der Rechenleistung durchaus beeindruckend ist. Mit dem Gedanken, durch Spezialarchitekturen die Rechenleistung linear sowohl an die Steigerung der Transistorzahl als auch der Taktfrequenz zu koppeln, erhofft man sich, die Mainstream-Entwicklung der Rechenleistung weit übertreffen zu können. Dies würde einen Durchbruch in den Möglichkeiten astrophysikalischer Simulationen bedeuten. Eine solche Skalierung mit der Chipentwicklung ist möglich, wenn ein Algorithmus auf der Ebene der Rechenoperationen durch Spezialrechenwerke parallel implementiert werden kann, so dass die Schaltung dauerhaft ausgelastet wird. Für die Berechnung der Gravitationswechselwirkung hat sich dieser Ansatz bereits als sehr erfolgreich erwiesen. So erreichen die GRAPE-Spezialrechner (beschrieben in Abschnitt 2.4.2) mit einem einzigen System in der Größe einer Workstation eine Rechenleistung eines Supercomputers (1 TFlop). Kann SPH durch einen ähnlichen Ansatz ebenfalls beschleunigt werden, lässt sich für viele Anwendungen eine Vervielfachung der Simulationsleistung erreichen.

Diese Arbeit wurde interdisziplinär durchgeführt. Die Grundlagen von Seiten der Astrophysik wurden in Kapitel 2 eingeführt, in den Kapiteln 3 und 4 dagegen wurden die notwendigen Grundlagen der Technik und Computerarithmetik ausgeführt. In Kapitel 5 wurden anhand

von Messungen an den astrophysikalischen Codes und Abschätzungen des Speedups einer hybriden Rechnerarchitektur aus Standardrechner und Rechenbeschleuniger die Voraussetzungen aus Physik und Informatik zusammengeführt. Daraus ergaben sich die Kriterien nach denen die Beschleunigerarchitektur entwickelt wurde. Der Ansatz, einen FPGA-Koprozessor zu verwenden und Gleitkommaarithmetik mit beschränkter Genauigkeit anzuwenden, konnte als geeignete Strategie herausgestellt werden. Für die Architektur innerhalb des FPGAs wurde die vollständige Parallelisierung der SPH-Berechnungen in Form von Rechenpipelines gewählt. Demzufolge wurden die Rechenelemente für ein FPGA-basiertes System entwickelt (Kapitel 6), und aufbauend darauf ein Prototyp für einen Rechenbeschleuniger synthetisiert (Kapitel 7). Es soll nun auf diese Schritte im einzelnen eingegangen werden.

Voraussetzungen von Seiten der astrophysikalischen Simulationsalgorithmen

In Kapitel 2 wurden die Standardmethoden der numerischen Simulation astrophysikalischer Systeme erläutert. Es wurde gezeigt, dass für stoßfreie Systeme wie Galaxien effiziente Simulationsverfahren existieren, die im Rechenaufwand mit $N \log N$ skalieren (Teilchenzahl N). Die für eine Vielzahl von astrophysikalischen Systemen zu berücksichtigende Hydrodynamik kann durch das SPH-Verfahren durch eine Teilchenmethode (wie die Gravitation) berücksichtigt werden. In SPH wird über eine Glättungskernmethode mit lokal begrenzten Wechselwirkungen zwischen den Teilchen gerechnet. Es ergibt sich ein Rechenaufwand von $N \cdot N_{neigh}$, wenn N_{neigh} die durchschnittliche Zahl von Nachbarteilchen ist, die einen Wechselwirkungsbeitrag leisten. Der SPH-Algorithmus zerteilt sich in zwei Berechnungsschritte. Im ersten Schritt wird für alle Teilchen die Dichte an deren Positionen im Raum berechnet, denn diese Werte werden für alle weiteren Berechnungen benötigt. Im zweiten Schritt werden die Beschleunigungen, Energien und eventuell weitere Größen berechnet.

In typischen astrophysikalischen Codes liegt der Rechenaufwand für die Gravitation etwa um einen Faktor 2–3 über dem der Hydrodynamik. Wird für die Gravitation ein GRAPE-Rechenbeschleuniger eingesetzt, dominiert die SPH-Berechnung die Simulationszeit. Dies wurde in Abschnitt 5.1 anhand der Analyse eines aktuellen Simulationscodes demonstriert.

Voraussetzungen von Seiten der Informatik

Die Funktionsweise von rekonfigurierbaren Koprozessoren basierend auf FPGAs wurde in Kapitel 3 beschrieben. Im Wesentlichen stellen FPGAs einen Pool von programmierbaren Logikressourcen bereit, die zu beliebigen digitalen Schaltkreisen zusammengesetzt werden können, limitiert nur durch die Anzahl an verfügbaren Logikelementen. In Kapitel 4 wurden verschiedene Strategien diskutiert, für unterschiedliche Zahlensysteme arithmetische Operatoren mit FPGAs zu implementieren. Es wurde dabei kein spezieller FPGA-Typ vorausgesetzt, da die Ergebnisse der Arbeit auf andere Architekturen übertragbar sein sollen. So wurden für alle Operationen Berechnungsmethoden formuliert, die ohne spezielle FPGA-Ressourcen wie Block-Muliplizierer auskommen. Es wurde zusätzlich auf die Optimierungsmöglichkeiten bei modernen FPGAs, insbesondere für den verwendeten Virtex-II-FPGA eingegangen. Als Kern der Operationen in anderen Zahlensystemen wurden Ganzzahloperationen besonders eingehend analysiert. Für Addierer

wurden die wichtigsten Standardmethoden wiedergegeben. In Abschnitt 6.1.1 erwies sich, dass in FPGAs die einfachen Ripple-Carry-Addierer die beste Performance erreichen. Für Multiplizierer ergab sich die Architektur eines Array-Multiplizierers als besonders vielversprechend für eine FPGA-Implementierung. In Abschnitt 6.1.2 wurde dagegen die Kombination aus kleinen Array-Multiplizierern mit einem nachfolgenden Addiererbaum als bessere Implementierungsstrategie begründet. Sind im FPGA Block-Multiplizierer vorhanden, können natürlich diese verwendet werden. Für die Dividierer und Quadratwurzeloperatoren konnten Digit-Recurrence-Methoden gefunden werden, die sich optimal auf die FPGA-Ressourcen abbilden lassen. Es wurden alternative Verfahren zur Berechnung elementarer Funktionen wie $1/x$ und $\sqrt{x}$ diskutiert (in Abschnitt 4.5), wie sie auch in Mikroprozessoren eingesetzt werden. Diese Alternativen sind teilweise auch für eine FPGA-Implementierung interessant, für diese Arbeit überwogen jedoch deren Nachteile. Die einfachen Digit-Recurrence-Methoden stellten sich als beste Implementierungsvariante heraus.

In den Abschnitten 4.3 und 4.4 wurden die für die Implementierung von SPH-Rechenwerken interessantesten Systeme der Gleitkommazahlen, der logarithmischen und der semilogarithmischen Zahlen diskutiert.

Kriterien für die Beschleunigerarchitektur

Ausgehend von der Analyse eines aktuellen astrophysikalischen Simulationscodes wurden in Kapitel 5.1 die Anforderungen an einen Rechenbeschleuniger abgeleitet. Für ein System bestehend aus einer Workstation und einem GRAPE-5-System ergab sich ein Anteil von etwa 14 % für die Gravitationskraftberechnung, wobei die Rechenzeit von der Kommunikation mit dem GRAPE dominiert wird. Bis auf wenige Prozent entfällt der Rest der Rechenzeit auf den SPH-Algorithmus. Für die neuen Bus-basierten GRAPE-6-Systeme kann erwartet werden, dass dann nur deutlich unter 10 % der Zeit für die Gravitationsberechnung aufgewendet wird. Es ergibt sich ein Beschleunigungspotential der Gesamtanwendung von 10, wenn für SPH ein Speedup auf eine Dauerrechenleistung von mehr als 2 GFlops erreicht wird. Dazu kann ein Rechenbeschleuniger eingesetzt werden, der eine Kommunikationsbandbreite mit dem Host-System von etwa 200 MByte/sec aufweisen muss. Voraussetzung ist auch eine Bereitstellung der Nachbarschaftslisten mit dieser Übertragungsrate. Bei der aktuellen Struktur des Simulationscodes wird noch etwa 30 % der Rechenzeit für die Generierung dieser Listen aufgewendet. Es wurde davon ausgegangen, dass dieser Anteil durch eine Änderung des Simulationscodes ausreichend gesenkt werden kann, wobei die erforderlichen Optimierungsstrategien skizziert wurden.

Ein zentraler Aspekt für eine Implementierung auf einem rekonfigurierbaren Rechensystem ist die Rechengenauigkeit, da davon der Ressourcenverbrauch abhängt. Der Ressourcenaufwand entscheidet wiederum über daß Maß an Parallelisierung und damit über die Rechengeschwindigkeit. Die erforderliche Rechengenauigkeit wurde eingegrenzt, indem Testsimulationen mit künstlich eingeschränkter Genauigkeit durchgeführt wurden und signifikante Abweichungen im Zeitverhalten der Energien als Indikator für eine zu geringe Rechengenauigkeit angenommen wurden. Bereits bei einer Mantissenbreite von 12 Bit ergab sich eine sehr hohe Qualität der Simulation. Zur detaillierteren Analyse wurde eine Simulationsumgebung für Rechenwerke mit beschränkter Genauigkeit erstellt. Es konnte gezeigt werden, dass die Testsimulationen das Verhal-

ten von Rechenwerken für Gleitkommazahlen mit reduzierter Mantissenbreite bis auf einen Unsicherheitsfaktor von 1–2 Bits korrekt wiedergeben. Für die Dichten ergab sich eine Rechengenauigkeit im Bereich der Darstellungsgenauigkeit. Für eine Mantissenbreite von 16 Bits übertreffen bereits sämtliche lokale SPH-Größen die Genauigkeitsanforderungen von $O(0.1)$ % an die makroskopischen Größen (z.B. Energien). Mit großer Sicherheit genügt also ein den Berechnungen zugrunde liegendes Gleitkommaformat mit 16 Mantissenbits, mit hoher Wahrscheinlichkeit sind bereits 13–14 Bits ausreichend, was Spielraum für Optimierungen eröffnet. Die verbleibende Unsicherheit bezüglich der optimalen Rechengenauigkeit führt zum Design-Paradigma, dass alle Rechenwerke in der Genauigkeit skalierbar sein müssen. Die Strategie, die Rechenwerke mit Gleitkommaarithmetik zu implementieren, konnte gegenüber der Verwendung logarithmischer oder semilogarithmischer Zahlensysteme am meisten überzeugen.

Implementierung der Rechenoperatoren für den Beschleuniger

In Kapitel 6 wurde detailliert auf die Implementierung der Gleitkommaarithmetik auf FPGAs eingegangen. Es wurde eine leistungsfähige Bibliothek von Gleitkommaoperatoren aufgebaut, welche für verschiedene FPGA-Typen anwendbar ist. Alle Operatoren sind in einem weiten Bereich in der Rechengenauigkeit skalierbar. Der Leitgedanke bei der Implementierung war, möglichst wenig Logikressourcen für die Operatoren aufwenden zu müssen und dennoch eine akzeptable Geschwindigkeit zu erreichen. Es wurden die Operationen Additon, Multiplikation, Division und Quadratwurzel implementiert, da diese für die Berechnung der SPH-Formeln benötigt werden. Besonders wichtig war auch die effiziente Implementierung von Gleitkommaakkumulatoren, da alle SPH-Berechnungen über Summationsvorschriften durchgeführt werden. Mit großer Ressourcenersparnis konnten spezialisierte Operatoren entwickelt werden, welche von Randbedingungen an die Numerik profitieren. So wurde bei der Addition und Akkumulation unterschieden, ob die Operanden über ein Vorzeichen verfügen oder nur positive Beträge addiert werden. Im letzteren Fall können 30–50 % an Logikressourcen eingespart werden. Es wurden getrennte Designs für die Quadratur und Multiplikation erstellt, mit einer Ressourcenersparnis von etwa 35 % gegenüber einem Multiplizierer, wenn nur quadriert werden muss. Die Optionen, Block-Multiplizierer zu verwenden (der Xilinx Virtex-II-FPGA XC2V3000 hat z.B. 96 dieser Elemente), oder nur mit allgemeinen Logikressourcen zu arbeiten, wurden beide umgesetzt. Werden Block-Multiplizierer verwendet, können über 80 % an Logikressourcen eingespart werden. Es wurden darüber hinaus verschiedene Rundungsmodi implementiert, um eine Reduktion des Logikbedarfs um bis zu 15 % gegenüber dem Standardrundungsmodus zu ermöglichen. Die Quadratwurzel- und Divisions-Operatoren wurden über Digit-Recurrence-Methoden implementiert, mit der Freiheit die Pipelinetiefe zu variieren. Damit wurde eine direkte Einflussmöglichkeit auf den Ressourcenverbrauch und die Geschwindigkeit dieser Operatoren bereitgestellt. Alle Operatoren wurden über einen Mantissenbreitenbereich von 12 bis 24 Bit synthetisiert. Die Ergebnisse erlauben die schnelle Abschätzung des Implementierungsaufwands zukünftiger Rechenwerke für verschiedene Rechengenauigkeiten.

Prototypimplementierung des Rechenbeschleunigers

Die Plattform für die Implementierung war das an der Universität Mannheim entwickeltes Koprozessorboard MPRACE, basierend auf einem Virtex-II-FPGA von Xilinx (XC2V3000-4). Es wurden zwei getrennte Schaltungsdesigns für den ersten und zweiten Schritt des SPH-Algorithmus implementiert. Die Schaltungsarchitektur fußt auf dem Ansatz, alle Rechenoperationen für die innersten Schleifen des SPH-Algorithmus vollständig parallel als Pipeline zu implementieren. Für den ersten Schritt mussten dazu 54 Gleitkommaoperationen zusammengesetzt werden. Es gelang eine Implementierung unter Aufwendung von 48 % der Logikressourcen für das Rechenwerk. Es wurde ein Gleitkommaformat mit 16 Mantissenbits und 8 Exponentenbits verwendet und bis auf wenige Ausnahmen mit dem Rundungsmodus *Round To Nearest Integer* gerechnet. Für die Interpolationskerne der SPH-Methode wurden Spezialoperatoren implementiert, um von der speziellen Form dieser Funktionen eine besonders ressourceneffiziente Implementierung ableiten zu können. Für den zweiten Schritt wurde eine ähnliche Pipeline wie für den ersten Schritt aufgebaut. Hier mussten 60 Operationen parallel implementiert werden und es folgte ein Aufwand von 49 % der Logikressourcen des FPGAs. Um die Rechenwerke in die Koprozessorplattform einzubetten, wurde eine modulare Interfacestruktur implementiert, die es gestattet, sehr einfach Portierungen auf andere FPGA-Plattformen durchzuführen. So wurde zu Evaluierungszwecken innerhalb von vier Wochen auch eine Portierung auf das japanische PROGRAPE-System durchgeführt, (es wurde in der Arbeit nicht weiter darauf eingegangen, da sich diese Plattform als zu klein für Pipelines der gewünschten Rechengenauigkeit erwies).

Für die Gesamtschaltung auf dem MPRACE-Board ergab sich ein Ressourcenbedarf von etwa 53 % für Schritt 1 und 56 % für Schritt 2. Die Pipelines konnten mit der PCI-Bus-Geschwindigkeit von 66 MHz betrieben werden – mit einer Spitzenrechenleistung von 3.5 und 3.9 GFlops. Allerdings erzwang die Anbindung an den mit doppelter Taktfrequenz arbeitenden Speicher des MPRACE-Boards eine Reduzierung der Frequenz auf 50 MHz. Der Datenfluss der Prototypimplementierung führt zu einer weiteren Reduktion der Verarbeitungsgeschwindigkeit, so dass die erzielbare Dauerleistung etwa 1.5 GFlops beträgt.

Diskussion der Perspektiven

Durch diese Arbeit konnte gezeigt werden, dass die rechenzeitkritischen Formeln des SPH-Algorithmus auf FPGAs wesentlich schneller berechnet werden können als auf Standardprozessoren. Mit 3.9 GFlops ist die FPGA-Implementierung für die SPH-Kraftberechnung etwa um einen Faktor 20 schneller als eine moderne CPU. Es ergab sich jedoch auch, dass dieses Beschleunigungspotential derzeit nicht ausgeschöpft werden kann. Die limitierte Kommunikationsbandbreite zwischen Host und Koprozessor sowie zwischen FPGA und dem lokalen Speicher reduzieren die Rechenleistung um etwa 50 %. Auch sind die aktuellen Simulationscodes noch nicht in der Lage, den Geschwindigkeitsvorteil effizient auszunutzen. Es muss vor allem die Erzeugung der Nachbarschaftslisten optimiert werden, um sowohl die Rechenzeit dafür von derzeit etwa 30 % auf deutlich unter 10 % der Rechenzeit für SPH zu reduzieren und gleichzeitig das zum Koprozessor zu transferierende Datenvolumen zu verringern. Mit dem Ansatz, für mehrere SPH-Teilchen gemeinsame Nachbarschaftslisten zu verwenden, können diese Ziele voraussicht-

lich erreicht werden. Die verwendete FPGA-Plattform bietet noch einige bisher brach liegende Logikressourcen, so dass Erweiterungen der FPGA-Designs zur Unterstützung neuer Verfahren mit komprimierten Nachbarschaftslisten möglich sind. Eine genauere Analyse dieser Modifikation steht noch aus. Es müssen in jedem Fall tiefe Eingriffe in den Simulationscode erfolgen um eine neue Methode der Nachbarschaftslistengenerierung zu implementieren und das Timing verschiedener Codeteile aufeinander abzustimmen. Das einfache Anwendungsprinzip der GRAPE-Systeme, nur die Teilchendaten zum Beschleuniger zu schicken und alles übrige autonom auf der Beschleunigerplattform durchzuführen, lässt sich nicht auf SPH übertragen. Der Grund dafür ist die Lokalität der Wechselwirkung, was dazu führt, dass für jedes Teilchen eine individuelle Interaktionsliste abgearbeitet werden muss.

Dennoch stehen die Chancen gut, durch die Koprozessorarchitektur eine Beschleunigung um den Faktor 7–10 zu erreichen. Mit diesem Speedup kann ein relativ kleines System aus Host, GRAPE und einem rekonfigurierbaren Rechensystem mit der Rechenleistung eines Supercomputers konkurrieren – bei äußerst hoher Preis-Performance. Wird diese hybride Plattform parallelisiert, wird es dann möglich sein, bereits mit etwa 32 Knoten astrophysikalische Simulationen unter Berücksichtigung der Hydrodynamik mit einer Dauerrechenleistung weit über einem TFlop durchzuführen.

Tabellenverzeichnis

2.1 Entwicklung der GRAPE-Systeme 24

4.1 ANSI/IEEE-754-Standard für Gleitkommazahlen einfacher und doppelter Genauigkeit 69

6.1 Vergleich von Addierertechniken für Virtex-II-FPGA 112
6.2 Ressourcenverbrauch und Geschwindigkeit von Array-Multiplizierern 116
6.3 Implementierungsergebnisse für Integer-Multiplizierer 116
6.4 Implementierungsergebnisse für Integer-Quadrierern 120
6.5 Ressourcenverbrauch und Geschwindigkeit von Festkomma-Dividierern 123
6.6 Implementierungsergebnisse für Quadratwurzel-Operatoren 126
6.7 Implementierungsergebnisse zu Shift-Bausteinen 130
6.8 Zählen führender Nullen oder selektive Zählung von Nullen und Einsen 131
6.9 Implementierungsergebnisse für Gleitkomma-Addierer für positive Zahlen 137
6.10 Implementierungsergebnisse für Gleitkomma-Addierer 143
6.11 Implementierungsergebnisse für Akkumulatoren 148
6.12 Implementierungsergebnisse für Gleitkomma-Multiplizierer 154
6.13 Implementierungsergebnisse für Gleitkomma-Multiplizierer (ohne Block-Mult) . 154
6.14 Implementierungsergebnisse für Gleitkomma-Quadrierer 156
6.15 Implementierungsergebnisse für Gleitkomma-Dividierer (ohne Runden) 160
6.16 Implementierungsergebnisse für Gleitkomma-Dividierer 161
6.17 Implementierungsergebnisse für Gleitkomma-Quadratwurzel (ohne Runden) . . . 164
6.18 Implementierungsergebnisse für Gleitkomma-Quadratwurzel 165
6.19 Vergleich mit kommerziellen Produkten 166

7.1 Ergebnisse für die Prototypimplementierung 185
7.2 Rechengenauigkeit der FPGA-Implementierung 186

Abbildungsverzeichnis

2.1 Veranschaulichung des SPH-Verfahrens 11
2.2 Kubische Splinefunktion des Standard-Glättungskerns 13
2.3 Simulationssystem mit GRAPE-6-Rechenbeschleuniger 21
2.4 Architektur des GRAPE-6-Spezialrechners aus Anwendersicht 22
2.5 Pipeline der GRAPE-6-Chips 23

3.1 Prinzipieller Aufbau eines FPGAs 28
3.2 Struktureller Aufbau eines Virtex-II-FPGAs von Xilinx 29
3.3 Struktur einer Slice-Einheit des Virtex-II-FPGAs 30
3.4 Halbaddierer und Volladdierer 32
3.5 Volladdierer und Addier/Subtrahier-Elemente mit Virtex-II-FPGA 33
3.6 Volladdierer mit Selektion eines Eingangs mit Virtex-II-FPGA 33
3.7 Halbe Slice-Einheit eines Virtex-II-FPGAs 34
3.8 Carry-Signale innerhalb einer CLB des Virtex-II-FPGAs 35
3.9 Rechensystem aus Steuerrechner und rekonfigurierbaren Koprozessoren 37
3.10 Grundlegende Architektur eines rekonfigurierbaren Koprozessors 39
3.11 Der rekonfigurierbare Koprozessor MPRACE 39
3.12 Schematischer Aufbau des MPRACE Koprozessors 40

4.1 Addier/Subtrahier-Operator für 1er-Komplement-Zahlen 44
4.2 Addier/Subtrahier-Operator für Zahlen in der Vorzeichen-Betrag-Darstellung . . 46
4.3 Addierer für 4-Bit-Ganzzahlen als Ripple-Carry-Schaltung 47
4.4 Lookahead-Carry-Generator 48
4.5 Carry-Lookahead-Addierer für 16-Bit-Ganzzahlen 49
4.6 Addierer für Ganzzahlen als Carry-Save-Schaltung 49
4.7 Addierer für Ganzzahlen als einstufige Carry-Select-Schaltung 50
4.8 Zweistufiger Carry-Select-Addierer 50
4.9 Hardwareumsetzung der Multiplikation nach der Shift/Add-Methode 52
4.10 Hardwareumsetzung der Multiplikation nach der Tree-Methode 52
4.11 Hardwareumsetzung der Multiplikation als Array-Multiplizierer 53
4.12 Detailschaltbild eines Array-Multiplizierers für 4-Bit-Zahlen 54
4.13 Serieller Dividierer nach der Non-Performing-Methode 57
4.14 Detailschaltbild eines Non-Restoring Array-Dividierers 58

4.15 Blockschaltbild eines einfachen seriellen Dividierers nach der SRT-Methode . . . 60
4.16 Berechnungsbeispiel zur Quadratwurzel nach der Papier-und-Bleistift-Methode . 63
4.17 Detailschaltbild eines Quadratwurzel-Operators mit Non-Restoring-Verfahren . . 65
4.18 Rundungsmodi des IEEE-754-Standards für Gleitkommazahlen 68
4.19 Maximaler relativer Fehler von Gleitkommaoperationen 70
4.20 Allgemeine Struktur eines Gleitkommaoperators 71
4.21 Funktioneller Aufbau eines Addierers für Gleitkommazahlen 73
4.22 Funktioneller Aufbau eines Multiplizierers für Gleitkommazahlen 75
4.23 Funktioneller Aufbau eines Dividierers für Gleitkommazahlen 77
4.24 Funktioneller Aufbau eines Quadratwurzeloperators für Gleitkommazahlen . . . 78
4.25 Lineare Approximation nach der Bipartite-Table-Methode 84

5.1 Energievergleich zur Analyse der erforderlichen Rechengenauigkeit 98
5.2 Relative Fehler der Energien für 8-Bit- und 12-Bit-Mantissen 99
5.3 Relative Fehler der Energien für 20-Bit-Mantissen 100
5.4 Teilchenverteilungen zur Analyse der Rechenwerke 102
5.5 Abhängigkeit des mittleren relativen Fehlers von der Mantissenbreite – 1 103
5.6 Abhängigkeit des mittleren relativen Fehlers von der Mantissenbreite – 2 103
5.7 Grundlegende Hardwarearchitektur für SPH 108
5.8 Datentransferleistung für das MPRACE-Board 108

6.1 Multipliziererschaltung für 4-Bit-Integer-Zahlen 114
6.2 Multipliziererschaltung für 16-Bit-Integer-Zahlen 117
6.3 Quadrierung einer 4-Bit-Zahl 119
6.4 Quadriererschaltung für 16-Bit-Integer-Zahlen 119
6.5 Dividiererschaltung für 4-Bit-Festkommazahlen 122
6.6 Quadratwurzel für 4-Bit-Festkommazahlen 125
6.7 Verschiebeschaltung für 32-Bit-Vektor 130
6.8 Verarbeitung der Exponenten bei Addierer für pos. Gleitkommazahlen 133
6.9 Gleitkommazahl-Addierer für positive Zahlen 134
6.10 Rundung und Normalisierung bei Addition positiver Gleitkommazahlen 136
6.11 Ressourcen für Gleitkomma-Addierern für positive Zahlen 138
6.12 Addierer für beliebige Gleitkommazahlen 140
6.13 Normalisierung beim Gleitkommazahl-Addierer 142
6.14 2er-Komplement-Bildung und Rundung beim Addierer für beliebige Gleitkommazahlen 142
6.15 Ressourcen für Gleitkomma-Addierer 144
6.16 Vorverarbeitung der Exponenten im Akkumulatorbaustein 146
6.17 Akkumulator für positive Gleitkommazahlen 147
6.18 Ressourcenverbrauchs für die Gleitkomma-Akkumulatoren 149
6.19 Akkumulator für Gleitkommazahlen 151
6.20 Multiplizierer für Gleitkommazahlen 153
6.21 Ressourcen für Gleitkomma-Multiplizierer (ohne Block-Mult-Elemente) 155

6.22 Ressourcen für Gleitkomma-Multiplizierer . . . 155
6.23 Implementierungsergebnisse für Gleitkomma-Quadrierer . . . 157
6.24 Dividierer für Gleitkommazahlen . . . 159
6.25 Ressourcenverbrauch für Gleitkomma-Dividierer . . . 161
6.26 Quadratwurzel-Operator für Gleitkommazahlen . . . 163
6.27 Implementierungsergebnisse für Gleitkomma-Quadratwurzel . . . 165
6.28 Vergleich des Ressourcenaufwands für verschiedene Gleitkommaoperatoren . . . 167

7.1 Spline-Kernfunktion für SPH . . . 170
7.2 Schaltung zur Berechnung der Spline-Kernfunktion . . . 171
7.3 Kernfunktion Ω für SPH . . . 174
7.4 Spezialoperator zur Berechnung der Kernfunktion Ω . . . 175
7.5 Blockschaltbild zur SPH-Dichteberechnung . . . 178
7.6 Blockschaltbild zur Berechnung der Beschleunigung . . . 181
7.7 Modulares Design des Rechenbeschleunigers . . . 183
7.8 Datenfluss zwischen Host und Rechenbeschleuniger . . . 184

Literaturverzeichnis

[1] S. J. Aarseth, M. Henon, and R. Wielen. A comparison of numerical methods for the study of star cluster dynamics. *Astron. Astrophys.*, 37:183–187, 1974.

[2] P. Alfke. Evolution, revolution, and convolution - recent progress in field programmable logic. In *Proc. of the 7th Workshop on Electronics for LHC Experiments*, pages 25–31, Stockholm, Sweden, 2001.

[3] H. Amano, Y. Shibata, and M. Uno. Reconfigurable systems: New activities in asia. In *Proc. International Conference on Field Programmable Logic and Applications*, pages 585–594, 2000.

[4] D. S. Balsara. Phd thesis, University of Illinois, 1990.

[5] J. Barnes and P. Hut. A Hierarchical O(N log N) Force-Calculation Algorithm. *Nature*, 324:446–449, 1986.

[6] R. Bate, Matthew and A. Burkert. Resolution requirements for smoothed particle hydrodynamics calculations with self-gravity. *Mon. Not. R. astr. Soc.*, 288:1060–1072, 1995.

[7] J. Becker, A. Kirschbaum, F.-M. Renner, and M. Glesner. Perspectives of reconfigurable computing in research, industry and education. In *Proc. International Conference on Field Programmable Logic and Applications*, pages 39–48, 1998.

[8] P. Belanovic and M. Leeser. A library of parameterized floating-point modules and their use. In *Proc. International Conference on Field Programmable Logic and Applications*, pages 657–666, Springer-Verlag Berlin, Heidelberg, 2002.

[9] G. Bell and J. Gray. High performance computing: Crays, clusters and centers. what next? Technical Report MSR-TR-2001-76, ACM, 2001.

[10] D. Benitez. A quantitative understanding of the performance of reconfigurable coprocessors. In *Proc. International Conference on Field Programmable Logic and Applications*, pages 976–986, 2002.

[11] W. Benz. Smooth particle hydrodynamics: A review. *The Numerical Modeling of Nonlinear Stellar Pulsations*, pages 269–288, 1990.

[12] E. Bertschinger. Simulations of structure formation in the universe. *Astron. Astrophys.*, 36:599–654, 1998.

[13] J.-L. Beuchat and A. Tisserand. Small Multiplier-Based Multiplication and Division Operators for Virtex-II Devices. In M. Glesner, P. Zipf, and M. Renovell, editors, *Proc. International Conference on Field Programmable Logic and Applications*, pages 513–522, Springer Verlag Berlin, Heidelberg, 2002.

[14] J. Binney and S. Tremaine. *Galactic Dynamics*. Princeton University Press, 1987.

[15] T. Bubeck, M. Hipp, S. Hüttemann, S. Kunze, M. Ritt, W. Rosenstiel, H. Ruder, and R. Speith. Parallel SPH on Cray T3E and NEC SX-4 using DTS. In *High Performance Computing in Science and Engineering*. Springer Verlag, 1998.

[16] J. N. Coleman and E. I. Chester. A 32-bit logarithmic arithmetic unit and its performance compared to floating-point. In Koren and Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (Adelaide, Australia)*, pages 142–151, Los Alamitos, CA, Apr. 1999. IEEE Computer Society Press.

[17] K. Compton and S. Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, June 2002.

[18] T. A. Cook, H.-R. Kim, and L. Louca. Hardware acceleration of n-body simulations for galactic dynamics. In J. Schewel, editor, *Proc. Field Programmable Gate Arrays for Fast Board Development and Reconfigurable Computing (SPIE 2607)*, pages 115–126, Bellingham, WA, Oct. 1995. SPIE The International Society for Optical Engineering.

[19] H. M. P. Couchman. Efficient algorithms for the cosmological n-body problem. *Speedup, HPC Applications in Natural Science and Engineering*, 12(2):22–26, Sept. 1999.

[20] H. M. P. Couchman, P. A. Thomas, and F. R. Pearce. Hydra: An Adaptive-Mesh Implementation of P^3M-SPH. *Astrophys. J.*, 452:797, 1995.

[21] D. Das Sarma and D. W. Matula. Faithful Bipartite ROM Reciprocal Tables. In *Proc 12th Symposium on Computer Arithmetic*, pages 17–28, 1995.

[22] D. Das Sarma and D. W. Matula. Faithful Interpolation in Reciprocal Tables. In *Proc. 13th Symposium on Computer Arithmetic (ARITH13)*, pages 82–91, 1997.

[23] F. de Dinechin and A. Tisserand. Some improvements on multipartite table methods. Technical Report 4059, Unité de recherche INRIA Rhône-Alpes, Montbonnot-St-Martin, France, 2000.

[24] J. Dubinski. A parallel tree code. *New Astronomy*, 1:133–147, 1996.

[25] J. Dubinski, R. Humble, C. Loken, U.-L. Pen, and P. Martin. McKenzie: A Teraflops Linux Beowulf Cluster for Computational Astrophysics. In *Proc. 17th Ann. Intl. Symp. on High Performance Computing Systems and Applications (HPCS2003)*, May 2003.

[26] T. Ebisuzaki, J. Makino, T. Fukushige, M. Taiji, D. Sugimoto, T. Ito, and S. K. Okumura. GRAPE Project: an Overview. *Publ. Astron. Soc. Japan*, 45:269–278, June 1993.

[27] S. Elzinga, J. Lin, and V. Singhal. Design Tips for HDL Implementation of Arithmetic Functions. XAPP215(v1.0), http://www.xilinx.com, June 2000.

[28] M. D. Ercegovac, L. Imbert, D. W. Matula, J. M. Muller, and G. Wei. Improving goldschmidt division, square root, and square root reciprocal. *IEEE Transactions on Comput.*, 49(7):759–763, July 2000.

[29] G. Even and W. J. Paul. On the design of IEEE compliant floating-point units. *IEEE Transactions on Comput.*, 49(5):398–413, May 2000.

[30] B. Fagin and C. Renard. Field programmable gate arrays and floating point arithmetic. *IEEE Transactions on VLSI*, 2(3):365–367, 1994.

[31] C. S. Frenk, S. D. White, P. Bode, J. R. Bond, G. L. Bryan, R. Cen, H. M. P. Couchman, A. E. Evrard, N. Gnedin, A. Jenkins, A. M. Khokhlov, A. Klypin, J. F. Navarro, M. L. Norman, J. P. Ostriker, J. M. Owen, F. R. Pearce, U.-L. Pen, M. Steinmetz, P. A. Thomas, J. V. Villumsen, J. W. Wadsley, M. S. Warren, G. Xu, and G. Yepes. The santa barbara cluster comparison project: A comparison of cosmological hydrodynamics solutions. *Astrophys. J.*, 525:554, 1999.

[32] A. A. Gaffar, W. Luk, P. Y. K. Cheung, N. Shirazi, and J. Hwang. Automatic customization of floating-point designs. In *Proc. International Conference on Field Programmable Logic and Applications*, pages 523–533, 2002.

[33] A. A. Gaffar, O. Mencer, W. Luk, P. Y. K. Cheung, and N. Shirazi. Floating-point bitwidth analysis via automatic differentiation. In *Proc. IEEE International Conference on Field-Programmable Technology*, pages 158–165, 2002.

[34] S. Gelato, D. F. Chernoff, and I. Wasserman. An adaptive hierarchical particle-mesh code with isolated boundary conditions. *Astron. J*, 480:115, 1997.

[35] L. Geppert. Champagne supercomputer on a beer budget. *IEEE Spectrum*, 36(5):19–22, 1999.

[36] R. A. Gingold and J. J. Monaghan. Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Mon. Not. R. astr. Soc.*, 181:375–389, 1977.

[37] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–47, Mar. 1991.

[38] R. E. Goldschmidt. Applications of division by convergence, 1964. master thesis.

[39] Z. Guo, W. Najjar, F. Vahid, and K. Vissers. A Quantitative Analysis of the Speedup Factors of FPGAs Over Processors. In *Proc. of FPGA'04*, pages 171–180, Monterey, California, USA, Feb. 2004.

[40] T. Hamada, T. Fukushige, A. Kawai, and J. Makino. PROGRAPE-1: A Programmable, Multi-Purpose Computer for Many-Body Simulations. *Publ. Astron. Soc. Japan*, 52:943–954, Oct. 2000.

[41] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *Proc. of the Conference on Design, Automation, and Test in Europe (DATE'01)*, pages 642–649, 2001.

[42] D. C. Heggie and P. Hut. *The Gravitational Million-Body Problem - A multidisciplinary approach to Star Cluster Dynamics*. Cambridge Univ. Press, 2003.

[43] L. Hernquist and N. Katz. Treesph: A Unification of SPH with the Hierarchical Tree Method. *Astron. Astrophys. Suppl. Series*, 70:419–446, June 1989.

[44] M. A. Hitz and E. Kaltofen. Integer division in residue number systems. *IEEE Transactions on Comput.*, 44(8):983–989, 1995.

[45] C. H. Ho, M. P. Leong, P. H. W. Leong, J. Becker, and M. Glesner. Rapid Prototyping of FPGA based Floating Point DSP Systems. In *Proc. 13th IEEE International Workshop on Rapid System Prototyping*, pages 19–24, 2002.

[46] C. H. Ho, K. H. Tsoi, H. C. Yeung, Y. M. Lam, K. H. Lee, P. H. W. Leong, R. Ludewig, P. Zipf, A. G. Ortiz, and M. Glesner. Arbitrary Function Approximation in HDLs with Application to the N-body Problem. In *Proc. IEEE International Conference on Field-Programmable Technology*, 2003. (to appear).

[47] R. Hockney and J. Eastwood. *Computer Simulation Using Particles*. Adam Hilger, Bristol, 1989.

[48] *IEEE Standard for Binary Floating-Point Arithmetic*. IEEE Press, New York, 1985. (ANSI/IEEE Std 754-1985).

[49] A. Jaenicke and W. Luk. Parameterised Floating-Point Arithmetic on FPGAs. In *Proc. IEEE ICASSP*, volume 2, pages 897–900, 2001.

[50] A. Kawai, T. Fukushige, J. Makino, and M. Taiji. GRAPE-5: A Special-Purpose Computer for N-Body Simulations. *Publ. Astron. Soc. Japan*, 52:659–676, Aug. 2000.

[51] A. Kawai, J. Makino, and T. Ebisuzaki. Performance analysis of high-accuracy tree code based on the pseudoparticle multipole method. *Astrophys. J. Suppl. Series*, 151:13, 2004.

[52] O. Kessel-Deynet. *Berücksichtigung ionisierender Strahlung im Smoothed-Particle-Hydrodynamics-Verfahren und Anwendung auf die Dynamik von Wolkenkernen im Strahlungsfeld massiver Sterne*. Dissertation, Ruprecht-Karls-Universität Heidelberg, 1999.

[53] I. Koren. *Computer arithmetic algorithms*. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[54] T. Kuberka et al. AHA-GRAPE: Adaptive Hydrodynamic Architecture - GRAvityPipE. In *Procs. of The 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 99), Las Vegas, USA*, 1999.

[55] A. Kugel. MPRACE - A PCI-64 based High Performance FPGA Co-Processor, 2003. URL: http://www-li5.ti.uni-mannheim.de/fpga/?race/.

[56] U. W. Kulisch and W. L. Miranker. *Computer arithmetic in theory and practice*. Academic Press, New York, 1981.

[57] Y. Li and W. Chu. Implementation of Single Precision Floating Point Square Root on FPGAs. In K. L. Pocek and J. Arnold, editors, *Proc. of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 226–232, Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[58] Y. Li and W. Chu. Parallel-array implementation of a non-restoring square root algorithm. In *Proc. International Conference on Computer Design (ICCD´97)*, pages 690–695, Austin, Texas, USA, Oct. 1997.

[59] J. Liang and R. Tessier. Floating Point Unit Generation and Evaluation for FPGAs. In *Proc. of IEEE Symposium on FPGAs for Custom Computing Machines*, Napa, CA., 2003.

[60] A. A. Liddicoat. *High-Performance Arithmetic for Divison and The Elementary Functions*. PhD thesis, Stanford University, 2002.

[61] G. Lienhart. Beschleunigung Hydrodynamischer N-Körper-Simulationen mit Rekonfigurierbaren Rechensystemen. In *Mitteilungen – Gesellschaft für Informatik e.V. Parallel-Algorithmen, -Rechnerstrukturen und -Systemsoftware (PARS)*, pages 9–16, 2003. ISSN 0177-0454.

[62] G. Lienhart. Implementing Hydrodynamic N-Body Codes on Reconfigurable Computing Platforms. In *Proc. of the International Conference on High Performance Scientific Computing*, Hanoi, Vietnam, 2003. In press.

[63] G. Lienhart, A. Kugel, and R. Männer. Simulation mit konfigurierbarer Hardware. In *Tagungsband zum Symposium auf der 66. Physikertagung in Leipzig, Simulation in Physik, Informatik und Informationstechnik*, pages 18–21, Leipzig, Germany, 2002. ISSN 0944-7121.

[64] G. Lienhart, A. Kugel, and R. Männer. Using Floating Point Arithmetic on FPGAs for Accelerating Scientific N-Body Simulations. In *Proc. of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 182–191, Napa Valley, CA, USA, 2002.

[65] W. B. Ligon III, S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K. D. Underwood. A Re-evaluation of the Practicality of Floating Point Operations on FPGAs. In K. L. Pocek and J. Arnold, editors, *Proc. of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 206–215, Los Alamitos, CA, 1998. IEEE Computer Society Press.

[66] M. E. Louie and M. D. Ercegovac. Mapping division algorithms to field programmable gate arrays. In *Proceedings of the 1992 Asilomar Conference on Signals, Systems and Computers*, 1992.

[67] M. E. Louie and M. D. Ercegovac. A digit-recurrence square root implementation for field programmable gate arrays. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 178–183, Apr. 1993.

[68] M. E. Louie and M. D. Ercegovac. On digit-recurrence division implementation for field programmable gate arrays. In E. E. Swartzlander, M. J. Irwin, and J. Jullien, editors, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 202–209, Windsor, Canada, June 1993. IEEE Computer Society Press, Los Alamitos, CA.

[69] L. Lourca, T. A. Cook, and W. H. Johnson. Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs. In *Proc. of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 107–116. IEEE Computer Society Press, 1996.

[70] L. B. Lucy. A numerical approach to the testing of the fission hypothesis. *Astron. J*, 82:1013–1024, 1977.

[71] Z. Luo and M. Martonosi. Using delayed addition techniques to accelerate integer and floating-point calculations in configurable hardware. In J. Schewel, editor, *Configurable Computing: Technology and Applications, Proc. SPIE 3526*, pages 202–211, Bellingham, WA, 1998. SPIE – The International Society for Optical Engineering.

[72] J. Makino. Grape-6 user's guide. Version 0.5, 2002.

[73] J. Makino and P. Hut. Performance analysis of direct n-body calculations. *Astrophys. J. Suppl. Series*, 68:833–856, 1988.

[74] J. Makino, E. Kokubo, and T. Fukushige. Performance evaluation and tuning of GRAPE-6 – towards 40 "real" Tflops. In *Tech Papers of SC2003*, Phoenix, Arizona, USA, 2003. ACM.

[75] J. Makino, M. Taiji, T. Ebisuzaki, and D. Sugimoto. GRAPE-4: A Massively Parallel Special-Purpose Computer for Collisional N-Body Simulations. *Astrophys. J.*, 480:432–446, 1997.

[76] R. Matousek, M. Tichy, Z. Pohl, J. Kadlec, C. Softley, and N. Coleman. Logarithmic Number System and Floating-Point Arithmetics on FPGA. In M. Glesner, P. Zipf, and M. Renovell, editors, *Proc. International Conference on Field Programmable Logic and Applications*, pages 627–636, Springer Verlag Berlin, Heidelberg, 2002.

[77] J. J. Monaghan. Smoothed Particle Hydrodynamics. *Ann. Rev. Astron. Astrophys.*, 30:543–74, 1992.

[78] J. J. Monaghan and R. A. Gingold. Shock Simulation by the Particle Method SPH. *J. Comp. Phys.*, 52:374–389, 1983.

[79] J. J. Monaghan and J. C. Lattanzio. A refined particle method for astrophysical problems. *Astron. Astrophys.*, 149:135–143, 1985.

[80] J.-M. Muller, A. Scherbyna, and A. Tisserand. Semi-logarithmic number systems. *IEEE Transactions on Comput.*, 47(2):145–151, 1998.

[81] Nallatech Limited. IEEE 754 compatible floating point cores for Virtex-II FPGAs, 2002. URL: www.nallatech.com.

[82] T. Narumi, A. Kawai, and T. Koishi. An 8.61 Tflop/s Molecular Dynamics Simulation for NaCl with a Special-Purpose Computer: MDM. In *Proc. of the International Conference for High-Performance Computing 2001*, Denver, USA, 2001.

[83] T. Narumi, R. Susukita, T. Ebisuzaki, G. McNiven, and B. Elmegreen. Molecular dynamics machine: Special-purpose computer for molecular dynamics simulations. *Molecular Simulation*, 21:401–415, 1999.

[84] R. P. Nelson and J. C. B. Papaloizou. Variable smoothing lengths and energy conservation in smoothed particle hydrodynamics. *Mon. Not. R. astr. Soc.*, 270:1–20, 1994.

[85] S. F. Oberman. *Design Issues in High Performance Floating Point Arithmetic Units*. PhD thesis, Stanford University, Stanford, CA, USA, 1996.

[86] S. F. Oberman. Floating Point Division and Square Root Algorithms and Implementation in the AMD-K7 Microprocessor. In *Proc. 14th Symposium on Computer Arithmetic (ARITH14)*, pages 106–115, Apr. 1999.

[87] S. F. Oberman and M. J. Flynn. Division algorithms and implementations. *IEEE Transactions on Comput.*, 46(8):833–854, Aug. 1997.

[88] C. M. Pancake. What computational scientists/engineers should know about parallelism and performance. *Computer Applications in Engineering Education*, 4(2):145–160, 1996.

[89] B. Parhami. *Computer Arithmetic – Algorithms and Hardware Designs*. Oxford University Press, Inc., New York, 2000.

[90] F. R. Pearce and H. M. P. Couchman. Hydra: a parallel adaptive grid code. *New Astronomy*, 2:411–427, 1997.

[91] J.-A. Pineiro. *Algorithms and Architectures for Elementary Function Computation*. PhD thesis, University of Santiago de Compostela, 2003.

[92] J.-A. Pineiro and J. D. Bruguera. High-speed double-precision computation of reciprocal, division, square root and inverse square root. *IEEE Transactions on Comput.*, 52(12):1377–1388, Dec. 2002.

[93] J.-A. Pineiro, S. F. Oberman, J.-M. Muller, and J. D. Bruguera. High-speed function approximation using a minimax quadratic interpolator. Technical report, 2003. ULR: http://www.ac.usc.es/files/articulos/2003/gac2003-i01.ps.

[94] QinetiQ Limited. Quixilica Floating Point FPGA Cores Data Sheet, Dec. 2002. URL: www.QinetiQ.com/quixilica.

[95] J. Ramirez, U. Meyer-Bäse, A. Garcia, and A. Lloris. Design and Implementation of RNS-Based Adaptive Filters. In *Proc. International Conference on Field Programmable Logic and Applications*, pages 1135–1138, Springer Verlag Berlin Heidelberg, 2003.

[96] J. Ramirez, U. Meyer-Bäse, F. Taylor, A. Garcia, and A. Lloris. Design and Implementation of High-Performance RNS Wavelet Processors Using Custom IC Technlogies. In *Journal of VLSI Processing*, volume 34, pages 227–237, Kluwer Academic Publishers, The Netherlands, 2003.

[97] E. Roesler and B. Nelson. Novel Optimizations for Hardware Floating-Point Units in a Modern FPGA Architecture. In M. Glesner, P. Zipf, and M. Renovell, editors, *Proc. International Conference on Field Programmable Logic and Applications*, pages 637–646, Springer Verlag Berlin, Heidelberg, 2002.

[98] J. K. Salmon. *Parallel Hierarchical N-body Methods*. Ph.d. thesis, California Institute of Technology, 1990.

[99] J. K. Salmon and M. S. Warren. Skeletons from the treecode closet. *J. Appl. Phys.*, 111(1):136–155, 1994.

[100] M. J. Schulte and K. E. Wires. High-speed inverse square roots. In I. Koren and P. Kornerup, editors, *Proc. 14th IEEE Symposium on Computer Arithmetic*, pages 124 – 131, Apr. 1999.

[101] N. R. Scott. *Computer Systems and arithmetic*. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[102] J. A. Sellwood. The art of n-body building. *Ann. Rev. Astron. Astrophys.*, 25:151–186, 1986.

[103] N. Shirazi, A. Walters, and P. Athanas. Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines. In *Proc. of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 155–162, IEEE Computer Society Press, 1995.

[104] O. Spaniol. *Computer arithmetic: logic and design*. Wiley, New York, 1981.

[105] R. Speith. *Untersuchung von Smoothed Particle Hydrodynamics anhand astrophysikalischer Beispiele*. Dissertation, Eberhards-Karls-Universität zu Tübingen, 1998.

[106] V. Springel, N. Yoshida, and S. D. M. White. Gadged: A code for collisionless and gas-dynamical cosmological simulations. *New Astronomy*, 6:79, 2001.

[107] R. Spurzem. Direct n-body simulations. *Journal of Computational and Applied Mathematics*, 109:407–432, 1999.

[108] R. Spurzem and A. Kugel. Towards the million body problem on the computer – no news since the three-body-problem? In *Proc. of Molecular Dynamics on Parallel Computers*, pages 264–275, Singapore, 2000. World Scientific.

[109] R. Spurzem, J. Makino, T. Fukushige, G. Lienhart, A. Kugel, R. Männer, M. Wetzstein, A. Burkert, and T. Naab. Collisional stellar dynamics, gas dynamics and special-purpose computing. In *Proc. Intl. Symp. Comp. Science and Engineering 2002 (ISCSE'02)*. JSPS, 2002. astro-ph/0204326.

[110] M. Steinmetz. GRAPESPH: cosmological smoothed particle hydrodynamics simulations with the special-purpose hardware GRAPE. *Mon. Not. R. astr. Soc.*, 278:1005–1017, 1996.

[111] M. Steinmetz and E. Müller. On the capabilities and limits of smoothed particle hydrodynamics. *Astron. Astrophys.*, 268(1):391–410, 1993.

[112] J. E. Stine and M. J. Schulte. The symmetric table addition method for accurate function approximation. *Journal of VLSI Signal Processing*, 21:167–177, 1999.

[113] D. Sugimoto, Y. Chikada, J. Makino, T. Ito, T. Ebisuzaki, and M. Umemura. A special-purpose computer for gravitational many-body problems. *Nature*, 345:33, 1990.

[114] E. E. Swartzlander. *Computer Arithmetic*, volume 1–2. IEEE Computer Society Press, Los Alamitos, CA, 1990.

[115] E. E. Swartzlander and A. G. Alexpoulos. The sign-logarithm number system. *IEEE Transactions on Comput.*, 24:1238–1242, Dec. 1975.

[116] N. Takagi. Powering by a table look-up and a multiplication with operand modification. *IEEE Transactions on Comput.*, 47(11):1216–1222, Nov. 1998.

[117] R. Tessier and W. Burleson. Reconfigurable computing for digital signal processing: A survey. *Journal of VLSI Signal Processing*, 28(1):7–27, June 2001.

[118] C. Theis and R. Spurzem. On the evolution of shape in n-body simulations. *Astron. Astrophys.*, 341:361, 1999.

[119] Top 500 web site, Sept. 2003. URL: http://www.top500.org.

[120] K. Underwood. FPGAs vs. CPUs: Trends in Peak Floating-Point Performance. In *Proc. of FPGA'04*, pages 171–180, Monterey, California, USA, Feb. 2004.

[121] J. Wadsley. High performance computing in astrophysics: Parallel gasdynamics and gasoline. In *Proc. 17th Ann. Intl. Symp. on High Performance Computing Systems and Applications (HPCS2003)*, May 2003.

[122] M. Wannemacher. *Das FPGA-Kochbuch.* MITP-Verlag, 1998. ISBN 3-8266-2712-1.

[123] M. S. Warren and J. K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Proc. of the 1993 ACM/IEEE conference on Supercomputing*, pages 12–21, Portland, Oregon, Dec. 1993.

[124] M. S. Warren, E. H. Weigle, and W.-C. Feng. High-density computing: a 240-processor Beowulf in one cubic meter. In *Proc. of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–11, Baltimore, Maryland, Nov. 2002.

[125] S. Waser and M. J. Flynn. *Introduction to arithmetic for digital systems designers*. Holt, Rinehart and Winston, CBS College Publishing, New York, 1982.

[126] M. Wetzstein. Wine - a new code for astrophysical particle simulations. Diploma thesis, Ruprechts-Karls-Universität Heidelberg, 2000.

[127] Xilinx. Virtex-II Data Sheet, DS031 (V1.6) . http://www.xilinx.com, 2001.

[128] R. Zimmermann. Computer Arithmetic: Principles, Architectures, and VLSI Design, Mar. 1999.

Danksagung

Hiermit möchte ich allen danken, die mich während meiner Zeit als Doktorand unterstützt und somit das Zustandekommen dieser Arbeit ermöglicht haben.

An erster Stelle möchte ich Prof. Dr. Reinhard Männer dafür danken, dass er es mir ermöglicht hat, die Arbeit durchzuführen. Ich konnte mir seiner vollen Unterstützung zu jeder Zeit sicher sein und habe mich an seinem Lehrstuhl sehr wohl gefühlt.

Mein besonderer Dank gilt auch Priv. Doz. Rainer Spurzem und Prof. Dr. Andreas Burkert die das interdisziplinäre Projekt von der Seite der Astrophysik aufgebaut haben. Markus Wetzstein, Thorsten Naab und Olav Kessel-Deynet möchte ich danken für die hervorragende Zusammenarbeit in allem, was die Simulationsalgorithmen und Codes betraf.

Andreas Kugel und Matthias Müller möchte ich danken für die Unterstützung in allen Belangen der Hardware und Steuerungssoftware. Bedanken möchte ich mich auch bei Harald Simmler, Holger Singpiel, Erich Krause und Christian Hinkelbein für ihre spontane Hilfe bei verschiedensten technischen Problemen. Meinen Zimmerkollegen Stefan Hezel und Oliver Brosch danke ich für ihre uneingeschränkte Hilfsbereitschaft. Zudem möchte ich mich bei allen früheren und jetzigen Mitgliedern der FPGA-Arbeitsgruppe für das positive Arbeitsklima bedanken.

Klaus Lienhart und Stefan Hezel möchte ich herzlich für die vielen Stunden des Korrekturlesens danken. Danke auch an Oliver Brosch für die Durchsicht des Manuskripts.

Andrea Seeger und Christiane Glasbrenner danke ich für die vorbildliche logistische Betreuung während der gesamten Zeit. Allen anderen Kollegen danke ich für das freundliche Umfeld, in dem ich arbeiten durfte.

Bei meiner Familie und meiner Frau möchte ich mich für den wertvollen Rückhalt, den sie mir während der Arbeit gegeben haben, bedanken.

Bedanken möchte ich mich abschließend bei allen, die mich bei meiner Arbeit unterstützt haben, die ich aber an dieser Stelle nicht namentlich nennen kann.