Diplomarbeit

Implementierung eines Koprozessors für geometrische Algebra auf einem FPGA

Christian Gebken
chg@ks.informatik.uni-kiel.de

Institut für Informatik
Lehrstuhl Kognitive Systeme
Christian-Albrechts-Universität zu Kiel

Vorsitz:
Prof. Dr. G. Sommer

Betreuung:
Dr. Christian Perwass

Kiel, 22. Februar 2003

Vorwort

Mit der Entwicklung immer leistungsfähigerer Computer in den letzten Jahren wuchsen die Möglichkeiten bezüglich der Auswertung von Bildinformationen. Ein umfassendes Konzept hierfür repräsentiert der Einsatz geometrischer Algebra **GA**, auch bekannt als Cliffordalgebra. Sie findet neben der Bildverarbeitung gleichermaßen in vielen anderen Bereichen der Naturwissenschaft deutlichen Zuspruch. **GA** kann als Erweiterung der Standard-Vektoralgebra angesehen werden und verkörpert Arbeits- und Beschreibungsmittel in einem.

Diese Arbeit behandelt eine Implementierung der geometrischen Algebra auf einem reprogrammierbaren Logikchip (FPGA). Das Ziel ist die Realisierung einer Rechnerarchitektur, die parallel zu einer PC-CPU als Koprozessor für Operationen der **GA** betrieben werden kann.

Den Mittelpunkt der **GA** bildet das äußerst berechnungsintensive geometrische Produkt. Zusammen mit der Tatsache, daß bereits zahlreiche Beispiele für spezialisierte Hardware existieren, entstand die Motivation zu untersuchen, ob und bis zu welchem funktionellen Umfang ein **GA**-Koprozessor auf einem FPGA verwirklicht werden kann. Zusätzlich stellte sich die Frage, ob durch dessen Verwendung schließlich eine Leistungssteigerung erzielt werden kann. Mögliche Gründe hierfür wären eine Entlastung der PC-CPU und eine effizientere Verarbeitung durch ein speziell entwickeltes Rechenwerk.

Der in dieser Arbeit entwickelte Prozessor kann jederzeit dynamisch auf dem FPGA erzeugt werden, um das geometrische Produkt von Multivektoren für **GA**-Dimensionen bis 2^8 zu berechnen. Hierfür wird er mit Instruktionen geladen, die anschließend fließbandartig verarbeitet werden. Alle Vorgänge laufen unter Einbeziehung eines eigenen Speichers ab, in dem sich Eingabedaten und Ergebnisse befinden. Während der Berechnungen treten häufig Datenabhängigkeiten auf, deren Behandlung einen wichtigen Aspekt des Prozessorentwurfs darstellt. Nach Abarbeitung der geladenen Instruktionen wird der PC-CPU die Bereitschaft zum Übertragen der Resultate signalisiert.

Nachstehend wird eine inhaltliche Kurzübersicht aller zehn folgenden Kapitel angegeben.

+ Die Arbeit beginnt mit einer kurzen Einführung (Kapitel Eins).
+ Das zweite Kapitel befaßt sich mit den theoretischen Grundlagen der geometrischen Algebra (Cliffordalgebra).
+ In Kapitel Drei werden die in den Entwicklungsprozeß involvierten Hard- und Softwarekomponenten beschrieben, die den Ausgangspunkt gebildet haben.
+ Den zentralen Teil bildet Kapitel Vier, in dem die Struktur und der Betrieb des **GA**-Koprozessors erläutert werden.
+ Kapitel Fünf setzt sich mit einem inhärenten Problem von Prozessorstrukturen - Datenabhängigkeiten - auseinander. Es werden Lösungsansätze und andere Erweiterungen angegeben.
+ Als Übersicht dient Kapitel Sechs, in dem die entwickelten Teilschaltungen illustriert werden, die als Bauteile für den **GA**-Koprozessor fungieren.
+ In Kapitel Sieben wird das Ergebnis und das Verfahren einer Leistungsbewertung präsentiert. Die Resultate werden mit anderen bestehenden Ansätzen verglichen.
+ Den Nachspann leitet Kapitel Acht ein. Dort wird eine Zusammenfassung, eine abschließende Beurteilung und ein Ausblick gegeben.
+ Das neunte Kapitel zeigt die Bibliographie dieser Arbeit.
+ Kapitel Zehn beinhaltet ein Glossar.

Für das bessere Verständnis dieser Abhandlung werden Kenntnisse im Bereich der Rechnerarchitekturen, der digitalen Logikschaltungen bzw. der Logik vorausgesetzt. Eine prinzipielle Vertrautheit mit algebraischen Strukturen wird angenommen. Erfahrungen mit maschinennaher Programmierung wären zudem hilfreich.

Die *kursiv gedruckten Wörter* dieser Arbeit besitzen einen Eintrag im Glossar ab Seite 99, wo sich die Erläuterungen wichtiger Begriffe oder Abkürzungen befinden. Bezeichner von Signalen, Bauteilen und anderen schaltungsspezifischen Elementen sind **fettgedruckt hervorgehoben**, um das Referenzieren auf den Seiten zu erleichtern.

Für die Geduld und Unterstützung während meines Studiums möchte ich mich an dieser Stelle herzlich bei meinen Eltern Hannelies und Wilhelm Gebken sowie bei meiner Verlobten Stephanie Eilers bedanken. Ich hab euch lieb.

Kiel, Februar 2003 Christian Gebken

Inhaltsverzeichnis

Kapitel 1

Einleitung

Geometrische *Algebra* **GA**, als Erweiterung der Standard-Vektoralgebra [32, 33], eignet sich sehr gut zur Analyse, Beschreibung und Handhabung von geometrischen Objekten. Die Repräsentation von Entitäten wie Geraden oder Flächen ist anschaulich und unkompliziert. Geometrische *Algebra* hat sich auch für die projektive Geometrie [34, 39] als zweckmäßig erwiesen, wo sie als profundes Arbeitsinstrument und ausgezeichnetes Beschreibungsmittel zugleich eingesetzt wird [16, 18]. Diese Kombination erlaubt beispielsweise eine einheitliche Behandlung von normalen Punkten und sogenannten 'Punkten im Unendlichen', die in der konventionellen Vektoralgebra oftmals für Irregularitäten sorgen [34, 39]. Projektive Geometrie und daher auch **GA** kann in der Bildverarbeitung zur 3D-Rekonstruktion verwendet werden, etwa für die Berechnung von Fundamentalmatrizen [48].

Im Rahmen eines echtzeitfähigen 3D-Rekonstruktionsverfahrens besteht das Interesse, komplexe und somit rechenaufwendige Operationen der **GA** möglichst schnell auszuführen. Der Einsatz spezieller externer Hardware in Form eines Koprozessors für geometrische *Algebra* könnte eine solche Beschleunigung herbeiführen. Allein die Parallelisierung von Berechnungen auf einem **GA**-Koprozessor und anderen Aufgaben, die zwischenzeitlich von der PC-CPU abgearbeitet werden, würde eine Leistungssteigerung erzielen.

Die Entwicklung der Hardware-Realisierung eines **GA**-Koprozessors, in den Folgekapiteln auch als '*Design*' bezeichnet, und dessen Vergleich mit einer Softwarelösung ist Gegenstand dieser Arbeit. An den Prozessor wurde die Anforderung gestellt, das geometrische Produkt **GP** berechnen zu können. Er sollte ferner durch andere Operationen im nachhinein erweitert werden können.

Es handelt sich bei dem **GP** um den zentralen Operationstyp der geometrischen *Algebra*, der zwei geometrische Objekte, die als *Multivektoren* bezeichnet werden, miteinander verknüpft [17]. Ein *Multivektor* ist mit einem Vektor vergleichbar, der in seiner Summendarstellung neben den Basiseinheitsvektoren in gleicher Weise zusätzlich multidimensionale skalarbehaftete Basiskomponenten enthält. Diese geometrischen Basiselemente setzen sich wiederum multiplikativ aus Kombinationen von Basiseinheitsvektoren zusammen und werden *Basisblades* oder kurz *Blades* genannt. Gegliedert in geometrischen und skalaren Anteil, repräsentieren sie die Elemente der *Multivektoren*. Das geometrische Produkt erfordert, daß jedes *Blade* des einen *Multivektors* mit jedem des anderen multipliziert wird. Diese Operation spaltet sich in zwei Teilaufgaben auf - die Multiplikation der beteiligten Skalaranteile und die der geometrischen Anteile.

Die Grundlage der Hardware-Entwicklung ist ein auf einer *PCI*-Einsteckkarte befindlicher *FPGA*-Chip mit eigenem separaten RAM. Auf dem Chip können relativ umfangreiche *Schaltwerke* reversibel generiert werden. Als ein solches wird der **GA**-Koprozessor aus den Ressourcen des *FPGAs* erzeugt.

Nachstehend soll durch eine kurze Vorschau ein erster Eindruck von den Abläufen in dem *Design* gewonnen werden:

Bevor der Prozessor nach seiner Generierung den Betrieb aufnehmen kann, müssen auszuführende *Instruktionen* in das angegliederte RAM geladen werden. Einer *Instruktion* entspricht die Verknüpfung von je zwei *Multivektoren*.
Während der Operationsphase werden die *Blades* der beiden *Multivektoren* gemäß einer durch das geometrische Produkt bedingten Reihenfolge ausgelesen, um paarweise miteinander multipliziert zu werden. Die Ergebnisse dieser *Blade*-Einzelmultiplikationen einer einzelnen *Instruktion* können zum Teil identische geometrische Anteile aufweisen. Die Skalaranteile solch korrespondierender Resultate müssen aufaddiert werden. Daher werden die Skalaranteile aller Ergebnisblades auf den Inhalt von Speicherstellen im RAM aufaddiert, die anfänglich zu Null initialisiert wurden. Die betreffenden Zieladressen werden durch den jeweiligen geometrischen Anteil und durch die aktuelle *Instruktion* eindeutig festgelegt. Die entstehenden Speicherblöcke stellen die Ergebnismultivektoren dar.
Nach der Abarbeitung aller *Instruktionen* werden die erzeugten Resultatmultivektoren auf den PC heruntergeladen.

Kapitel 2

Einführung in die geometrische Algebra

Eine allgemeine Fragestellung des 19. Jahrhunderts war, wie Rotationen in 3D am besten repräsentiert werden könnten. Gerade passend ersann William Rowan Hamilton (*1805-1865*) im Alter von 38 die *Quaternionen*. Deren *Algebra* besteht aus den vier Elementen $\{1, \mathbf{i}, \mathbf{j}, \mathbf{k}\}$, die im wesentlichen Ebenen mit assoziiertem Rotationswinkel darstellen.
Fast parallel erscheint Hermann Gunther Graßmanns (*1809-1877*) Werk 'Lineare Ausdehnungslehre', das die Fundamente der heutigen (multi-) linearen *Algebra* beinhaltet. In der von dem deutschen Schullehrer begründeten äußeren *Algebra* steht das äußere Produkt im Mittelpunkt. Das Tensorprodukt geht ebenfalls auf ihn zurück.

Den entscheidenden Schritt machte 1878 William Kingdon Clifford (*1845-1879*). Sein Ziel war es, die Algebren von Graßmann und Hamilton in eine einzelne allgemeinere Struktur zu vereinigen, weil er Überschneidungen beider in grundlegenden Konzepten erkannte. Er entwickelte eine generellere *Algebra* (Cliffordalgebra), in deren 'geometrischem' Produkt ein inneres Produkt (aus der Gibbs Vektor *Algebra*) und ein äußeres Produkt enthalten sind. Sein geometrisches Produkt ist assoziativ, distributiv und überdies invertierbar [14, 12, 15].

In der Mathematik der Physik existieren bis heute viele verschiedene algebraische Systeme, die alle bestimmte Gemeinsamkeiten sowie individuelle Vor- und Nachteile haben. Cliffordalgebren sind vergleichbar umfassender und allgemeiner. Sie werden in vielen Bereichen der Forschung eingesetzt [13, 9, 11, 10]. Hierzu zählt u.a. die Mechanik (Rotation des starren Körpers) [50, 51], die Elektrodynamik (Maxwellgleichungen) [52, 54], die relativistische Physik (Raum-Zeit-*Algebra*), die Gravitationstheorie, die Quantenmechanik (Quantenspin) [53, 55] und die projektive Geometrie (Computer Vision) [1]. Insbesondere bei der Beschreibung von Fermionen (Elementarteilchen mit halbzahligem Spin) lieferte die Cliffordalgebra das entscheidende mathematische Hilfsmittel [47].

Der in dieser Arbeit verwendete Terminus 'Geometrische Algebra' bezeichnet eine endlichdimensionale, nicht-*degenerierte universelle* Cliffordalgebra $\mathcal{C}\ell_n$ über den n-dimensionalen linearen Raum $\mathbb{R}^n$ [48, 47].

[1] Unter '`http://www.mrao.cam.ac.uk/~clifford/publications/index.html`' finden sich eine Reihe von Abhandlungen über Anwendungen der Cliffordalgebra in den genannten Bereichen.

Die Bezeichnung 'Geometrische Algebra' steht allgemein für eine Cliffordalgebra die um eine durchgehende geometrische Interpretation erweitert wurde [9, 11]. Hier sollen beide Begriffe als äquivalent angesehen werden. Nachstehend wird analog das Symbol $\mathcal{G}_n$ statt $\mathcal{C}\ell_n$ verwendet.

Eine Konstruktion der geometrischen Algebra

In den folgenden Passagen sollen die wichtigsten Grundlagen zum ersten Verständnis der geometrischen *Algebra* vermittelt werden. Siehe auch [11, 14, 17, 15, 48]. Sei

$$\mathbb{B} = \{e_1, e_2, \dots, e_n\}$$

die kanonische Basis von $\mathbb{R}^n$. Gemäß einer Definition bilden alle *geordneten Produkte*

$$\{1,\ e_1, e_2, \dots, e_n\,,\quad e_{i_1} e_{i_2} \dots e_{i_r}\}, \qquad r = 2, 3, \dots, n$$

die Basis $\mathbb{B}_n$ von $\mathcal{G}_n$, falls $i_1 < i_2 < \dots < i_r$ und $i_k = 1, 2, \dots, n$ für alle k ist. Es gilt $\dim \mathcal{G}_n = |\mathbb{B}_n| = |2^{\mathbb{B}}| = 2^n$. Zum besseren Rechnen wird ab hier häufig $\mathcal{G}_3 = \mathcal{C}\ell_3$ verwendet. Dann ist

$$\mathbb{B}_3 = \{1,\ e_1, e_2, e_3,\ e_1e_2, e_1e_3, e_2e_3,\ e_1e_2e_3\}\ .$$

Alle Elemente von $\mathbb{B}_n$ werden geordnete *Basisblades* b genannt. Sie entstehen durch die Anwendung des geometrischen Produkts von Clifford auf alle möglichen Kombinationen der Basiseinheitsvektoren e_i aus $\mathbb{B}$. Hierzu gelte zunächst folgende lineare Abhängigkeit: $e_i\, e_j \propto e_j\, e_i$, $i \neq j$ [2].

Falls nicht explizit anders angegeben, sind lateinische Indices (i,j, ect.) in diesem Kaptiel immer aus der Menge $\mathcal{N} := \{1, 2, \dots, n\}$ mit $n := \dim \mathbb{B}$. *Basisblades* werden über griechischen Buchstaben (μ, ν, σ, …) im Index ausgezeichnet, die beliebige Teilmengen von $\mathcal{N}$ repräsentieren, also $\mu \in 2^{\{1,2,\dots,n\}}$. Die *geordneten Basisblades* aus $\mathbb{B}_n$ werden durch b_μ abgekürzt. In den folgenden Ausführungen wird meist die Rede von *allgemeinen Basisblades* sein, die durch e_μ , e_ν etc. symbolisiert werden. Es ist möglich, daß $\mu = \nu = \{1, 2, 4\}$ ist, aber $e_\mu \neq e_\nu$, mit beispielsweise $e_\mu = e_1e_2e_4$ und $e_\nu = e_1e_4e_2$.

Die Notation $|\mu|$ bezeichnet den *Grad* von e_μ (bzw. b_μ) und gibt die Anzahl der Basiseinheitsvektoren e_i an, die in e_μ enthalten sind ($e_\mu = e_1e_3 \Rightarrow |\mu| = 2$). Bezüglich e_μ und e_ν seien auf μ und ν alle mengentheoretischen Operationen möglich. Es entstehe etwa e_σ über $\sigma = \mu \cap \nu$ als Teilmenge von $\{1, 2, \dots, n\}$, so daß e_σ nur Basiseinheitsvektoren enthält, die in e_μ und e_ν vorkommen.

Griechische Buchstaben (α, β, …), die nicht als Index dienen, seien aus $\mathbb{R}$, falls dies nicht anders festgelegt wurde.

Die Elemente von $\mathcal{G}_n$ sind *Multivektoren*. Diese Objekte sind lineare Kombinationen von skalarbehafteten *Basisblades*. Sie werden durch Großbuchstaben gekennzeichnet:

$$A = \sum_\mu \alpha_\mu\, e_\mu = \sum_\mu \alpha'_\mu\, b_\mu$$

[2] Um auszuschließen daß in $\mathbb{B}_3$ neben e_1e_2 etwa e_2e_1 auftaucht. Später wird $e_1e_2 = -e_2e_1$.

Jeder Multivektor kann mit Hilfe von allgemeinen oder *geordneten Basisblades* (e_μ bzw. b_μ) ausgedrückt werden.
Speziellere Elemente sind k-Vektoren. Sie sind Summen von *Basisblades* vom Grad k. Ein k-Vektor $A_{\langle k\rangle}$ kann durch

$$A_{\langle k\rangle} = \sum_{i\in I} \alpha_i \, e_{\mu_i} \,, \qquad |\mu_i| = k \,, \quad I = \left\{1, 2, \ldots, \binom{n}{k}\right\}$$

dargestellt werden. Zur Bezeichnung eines k-Vektors unbestimmten Grades wird im nachfolgenden der Ausdruck κ -Vektor verwendet. Für k-Vektoren mit $k \leq 3$ gibt es besondere Namen - Skalar, Vektor, Bivektor und Trivektor. Beispielsweise ist

$$\mathbb{B}_3 = \{ \underbrace{1}_{Skalar}, \underbrace{e_1, e_2, e_3}_{Vektor}, \underbrace{e_1e_2, e_1e_3, e_2e_3}_{Bivektor}, \underbrace{e_1e_2e_3}_{Trivektor} \} \,.$$

Das *Basisblade* mit dem höchsten Grad n wird als Pseudoskalar I_n bezeichnet. In $\mathcal{G}_3$ ist $I_3 = e_1e_2e_3$. Die übliche Dar- und Vorstellung eines Vektors bleibt in $\mathcal{G}_n$ erhalten. Sie werden durch fette Kleinbuchstaben angegeben

$$\mathbf{a} = \sum_{i=1}^{n} \alpha_i \, e_i \,.$$

Es sei bemerkt, daß jeder Basiseinheitsvektor auch ein *Basisblade* ist und jedes *Basisblade* auch ein κ -Vektor ist und jeder κ -Vektor auch ein Multivektor ist.

Operationen in $\mathcal{G}_n$

Im folgenden wird eine Zusammenstellung der wichtigsten Eigenschaften und Rechenregeln von Operationen in $\mathcal{G}_n$ gezeigt. Eine Beleuchtung ihrer geometrischen Aspekte wird hier noch außer Acht gelassen, ist aber Gegenstand von Abschnitt 'Interpretation der GA' auf Seite 13. Zu den wesentlichen Operationen gehört das geometrische Produkt, das äußere Produkt sowie das innere Produkt. Nach einer separaten Betrachtung werden anschließend Rechenregeln für gemischte Terme der drei Produkte behandelt. Später wird kurz auf erweiterte Konzepte wie das Reverse, das Inverse oder das Duale eingegangen.
Zunächst werden die distributiven, nicht assoziativen Operatoren $\veebar$ (Kommutator) und $\barwedge$ (Anti-Kommutator) eingeführt. Sie sind wie folgt definiert

$$\begin{array}{lrcl} \text{Kommutator:} & A\veebar B & = & \frac{1}{2}(AB - BA) \\ \text{Anti-Kommutator:} & A\barwedge B & = & \frac{1}{2}(AB + BA) \end{array}$$

Folgende Formeln geben einige nützliche Rechenregeln an

1. $A\veebar B = -\,B\veebar A$
2. $A\barwedge B = B\barwedge A$
3. $A\veebar(B\veebar C) + B\veebar(C\veebar A) + C\veebar(A\veebar B) = 0$ (Jakobi Identität)
4. $(AB)\veebar C = A(B\veebar C) + (A\veebar C)B$

Das geometrische Produkt

Das geometrische Produkt (von Clifford) läßt sich als Summe von Kommutator $A \underline{\times} B$ und Anti-Kommutator $A \overline{\times} B$ schreiben

$$AB = A \underline{\times} B + A \overline{\times} B .$$

Es ist abgeschlossen in $\mathcal{G}_n$, nicht kommutativ, distributiv bzgl. der Addition von Elementen aus $\mathcal{G}_n$ und assoziativ bzw. multilinear

$$\alpha(ABC) = \alpha(AB)C = \alpha A(BC) = (\alpha A)BC = A(\alpha B)C = AB(\alpha C) \ \alpha \in \mathbb{R} .$$

Für die Definition des geometrischen Produkts muß zuerst der Begriff der Signatur erläutert werden. Sie legt das Ergebnis $e_i e_i{=}(e_i)^2$ des geometrischen Produkts zweier identischer Basiseinheitsvektoren e_i fest. Die Basiseinheitsvektoren können zu -1, $+1$ oder zu 0 quadrieren. Letzteres wird in dieser Arbeit jedoch ausgeschlossen, da nur nicht-entartete Algebren betrachtet werden. Die Signatur wird durch die Menge $\Lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ angegeben. Es gilt dann

$$e_i e_i = \lambda_i , \quad 1 \le i \le n .$$

Eine Cliffordalgebra, in der beispielsweise p Basiseinheitsvektoren zu $+1$ und q Basiseinheitsvektoren zu -1 quadrieren, wird mit $\mathcal{C}\ell_{p,q}$ resp. $\mathcal{G}_{p,q}$ angegeben.

Das geometrische Produkt zweier Basiseinheitsvektoren e_i, e_j ist definiert durch

$$e_i e_j = \begin{cases} -\, e_j e_i & \text{für} \quad i \ne j \\ \lambda_i & \text{für} \quad i = j \end{cases}$$

Die Assoziativität erlaubt

$$\begin{aligned} &e_i e_j e_k = (e_i e_j) e_k = (-\, e_j e_i) e_k = -\, e_j (e_i e_k) = -\, e_j (-\, e_k e_i) = e_j e_k e_i \quad \text{oder} \\ &e_j\, (e_i e_j e_k) = -\, e_i e_j e_j e_k = e_i e_j e_k e_j = (e_i e_j e_k)\, e_j . \end{aligned}$$

Für *allgemeine Basisblades* e_μ und e_ν läßt sich ableiten, daß

$$\begin{aligned} e_\mu\, e_\nu &= (-1)^{|\mu||\nu| - |\mu \cap \nu|}\, e_\nu\, e_\mu \\ e_\mu\, e_\nu &= \gamma\, e_\sigma , \quad \gamma \in \{+1, -1\} \qquad \sigma = (\mu \cup \nu) \setminus (\mu \cap \nu) . \end{aligned}$$

Der Wert von Faktor γ hängt von den zu λ_i quadrierenden Basiseinheitsvektoren e_i, $i \in \mu \cap \nu$ und von der Anordnung der e_i in e_σ ab.

Für Vektoren hat die Definition des geometrischen Produkts bedeutende Folgen - sie quadrieren zu Skalaren:

$$\mathbf{a}^2 = \mathbf{aa} = \sum_{i,j} \alpha_i \alpha_j\, e_i e_j = \sum_i \alpha_i^2\, e_i^2 = \sum_i \alpha_i^2\, \lambda_i \in \mathbb{R}$$

Die nächste Formel gibt an, wie das Produkt aus zwei *Multivektoren* $A = \sum_{x=1}^{2^n} \alpha_x b_x$ und $B = \sum_{y=1}^{2^n} \beta_y b_y$ aussieht. Hier stellen b_x, b_y und b_z das $x$te, $y$te bzw. $z$te *Basisblade* in $\mathbb{B}_n$ dar.

$$A\,B \;=\; \left(\sum_{x=1}^{2^n} \alpha_x b_x\right)\left(\sum_{y=1}^{2^n} \beta_y b_y\right) \;=\; \sum_{x=1}^{2^n}\sum_{y=1}^{2^n} (\alpha_x\beta_y)\,(b_x b_y) \;=\; \sum_{z=1}^{2^n} \gamma_z b_z$$

Die Abgeschlossenheit von $\mathcal{G}_n$ bzgl. des geometrischen Produkts erzwingt, daß die $2 \cdot 2^n$ Summanden wieder auf 2^n Terme abbilden. Der Skalar γ_z von b_z setzt sich im Extremfall aus 2^n Summanden $\alpha_x\beta_y$ zusammen. Für deren *Basisblades* b_x und b_y gilt

$$b_x b_y \;=\; b_z \Longrightarrow \big(\nexists\, \check{x} \neq x : b_{\check{x}} b_y \;=\; b_z\big) \text{ und } \big(\nexists\, \check{y} \neq y : b_x b_{\check{y}} \;=\; b_z\big)\;,$$

mit $x, \check{x}, y, \check{y}, z \in \{1, 2, \dots\,, 2^n\}$. Das geometrische Produkt ist eine injektive Abbildung, falls ein Operand konstant gehalten wird.

Das äußere Produkt

Das äußere Produkt wird in der geometrischen *Algebra* durch das '$\wedge$' (engl.: wedge) symbolisiert. Die wichtigsten Eigenschaften im Hinblick auf Basiseinheitsvektoren sind

$$e_i \wedge e_j = \begin{cases} -\,e_j \wedge e_i & \text{für} \quad i \neq j \\ \quad 0 & \text{für} \quad i = j\,. \end{cases}$$

Das äußere Produkt ist distributiv und assoziativ

$$\begin{aligned} e_i \wedge (e_j + e_k) \;&=\; e_i \wedge e_j + e_i \wedge e_k \\ e_i \wedge (e_j \wedge e_k) \;&=\; (e_i \wedge e_j) \wedge e_k\;. \end{aligned}$$

Die Distributivität erlaubt eine Ausdehnung dieser Eigenschaften auf *Basisblades* und *Multivektoren* $A, B, C \in \mathcal{G}_n$. Beispiel:

$$A \wedge (B + C) \;=\; A \wedge B + A \wedge C$$

Das äußere Produkt ist über das geometrische Produkt definiert

$$e_\mu \wedge e_\nu \;=\; \begin{cases} e_\mu\, e_\nu & \text{für } \mu \cap \nu = \emptyset \\ \quad 0 & \text{sonst}\,. \end{cases}$$

Für *Basisblades* e_μ mit $\mu_i \in \mu$, $1 \leq i \leq r \;:=\; |\mu|$ gilt die Äquivalenz

$$e_\mu \;=\; e_{\mu_1} e_{\mu_2} e_{\mu_3} \dots e_{\mu_r} \equiv e_{\mu_1} \wedge e_{\mu_2} \wedge e_{\mu_3} \wedge \,\dots\, \wedge e_{\mu_r}\;.$$

Eine wichtige Rechenregel für *Basisblades* lautet

$$e_\mu \wedge e_\nu \;=\; (-1)^{|\mu||\nu|}\; e_\nu \wedge e_\mu\;.$$

Sie läßt sich über die Distributivität auf κ -Vektoren erweitern

$$A_{\langle k \rangle} \wedge B_{\langle l \rangle} \;=\; (-1)^{kl}\; B_{\langle l \rangle} \wedge A_{\langle k \rangle}\;.$$

Aus der Definition folgt die Multilinearität des äußeren Produkts

$$\alpha \wedge e_\mu \;=\; (\alpha e_\emptyset)\wedge e_\mu \;=\; \alpha(e_\emptyset \wedge e_\mu) \;=\; \alpha e_\mu \qquad \text{und}$$
$$\alpha(e_\mu \wedge e_\nu \wedge e_\sigma) \;=\; (\alpha e_\mu)\wedge e_\nu \wedge e_\sigma \;=\; e_\mu \wedge e_\nu \wedge (\alpha e_\sigma)\,.$$

Das äußere Produkt eines Basiseinheitsvektors e_i mit einem *Basisblade* e_μ läßt sich in der Kommutator-Schreibweise darstellen

$$e_i \wedge e_\mu = \begin{cases} e_i \overline{\times} e_\mu & \text{falls } |\mu| \text{ gerade oder Null ist,} \\ e_i \underline{\times} e_\mu & \text{falls } |\mu| \text{ ungerade ist.} \end{cases}$$

BLADES

Ein *Blade* ist definiert als äußeres Produkt von Vektoren. *Blades* besitzen in der geometrischen *Algebra* eine herausgestellte Bedeutung, da ihr praktischer Wert, auf den erst später eingegangen wird, von besonderem Interesse ist. Der Ausdruck

$$\mathbf{a}_1 \wedge \mathbf{a}_2 \wedge \ldots \wedge \mathbf{a}_k$$

wird als *Blade* vom Grad k bezeichnet. Dessen Ausmultiplizieren im Rahmen der Distributivität erzeugt eine Linearkombination von *Basisblades* vom Grad k - einen k-Vektor. Entsprechend werden *Blades* vom Grad k ebenfalls durch $A_{\langle k \rangle}$ symbolisiert

$$A_{\langle k \rangle} \;=\; \mathbf{a}_1 \wedge \mathbf{a}_2 \wedge \ldots \wedge \mathbf{a}_k\,.$$

Die umgekehrte Richtung der Implikation gilt nicht, denn nicht jeder κ -Vektor kann als κ -*Blade* ausgedrückt werden:

$$(e_1 + e_2)\wedge(e_2 + e_3) \iff e_1 e_2 + e_1 e_3 + e_2 e_3 \qquad \text{aber}$$
$$e_1 e_2 + e_1 e_3 + e_2 \mathbf{e_4} \implies \quad ??$$

Für einen Vektor $\mathbf{a}$ ist immer

$$\begin{aligned} \mathbf{a} \wedge \mathbf{a} &= \sum_i \alpha_i e_i \wedge \sum_j \alpha_j e_j = \sum_{\substack{i,j \\ i \neq j}} \alpha_i \alpha_j\, e_i\, e_j \\ &= \sum_{\substack{i,j \\ i > j}} \alpha_i \alpha_j\, e_i\, e_j + \sum_{\substack{k,l \\ k<l}} \alpha_k \alpha_l\, e_k\, e_l = \sum_{\substack{i=l,\, j=k \\ i>j,\, k<l}} \Big(\alpha_i \alpha_j\, e_i\, e_j - \alpha_k \alpha_l\, e_l\, e_k\Big) \\ &= \sum_{\substack{i,j \\ i>j}} \Big(\alpha_i \alpha_j - \alpha_j \alpha_i\Big)\, e_i\, e_j = \mathbf{0}\,. \end{aligned}$$

Das innere Produkt

Das innere Produkt wird durch den '$\cdot$' (engl.: dot) repräsentiert. Die wichtigsten Eigenschaften im Hinblick auf Basiseinheitsvektoren sind

$$e_i \cdot e_j = \begin{cases} \lambda_i & \text{für} \quad i = j\,, \quad \lambda_i \in \Lambda \\ 0 & \text{für} \quad i \neq j\,, \end{cases}$$

wo $\Lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ mit $\lambda_i \in \{+1, -1\}$ die gewählte Signatur von $\mathcal{G}_n$ ist. Das innere Produkt ist distributiv, aber nicht assoziativ und kommutativ

$$e_i \cdot (e_j + e_k) = e_i \cdot e_j + e_i \cdot e_k$$

Wie bei dem äußeren Produkt gilt die Distributivität auch für *Basisblades* und *Multivektoren* $A, B, C \in \mathcal{G}_n$

$$A \cdot (B + C) = A \cdot B + A \cdot C \,.$$

Das innere Produkt ist ebenfalls über das geometrische Produkt definiert

$$e_\mu \cdot e_\nu = \begin{cases} e_\mu \, e_\nu & \text{falls } \mu \subseteq \nu \text{ oder } \nu \subseteq \mu \\ 0 & \text{sonst}\,. \end{cases}$$

Es sei $k := |\mu|$ und $l := |\nu|$. Dann gilt für *Basisblades*

$$e_\mu \cdot e_\nu = (-1)^{k+kl} \, e_\nu \cdot e_\mu \,, \qquad k \leq l \,.$$

Analog zum äußeren Produkt gilt für κ -Vektoren

$$A_{\langle k \rangle} \cdot B_{\langle l \rangle} = (-1)^{k+kl} \, B_{\langle l \rangle} \cdot A_{\langle k \rangle} \,, \qquad k \leq l \,.$$

Aus der Definition lassen sich folgende Eigenschaften ableiten

$$\begin{aligned} \alpha \cdot e_\mu &= (\alpha e_\emptyset) \cdot e_\mu = \alpha (e_\emptyset \cdot e_\mu) \stackrel{!}{=} 0 \qquad\qquad \text{und} \\ \alpha (e_\mu \cdot e_\nu) &= (\alpha e_\mu) \cdot e_\nu = e_\mu \cdot (\alpha e_\nu) \,. \end{aligned}$$

Das innere Produkt eines Basiseinheitsvektors e_i mit einem *Basisblade* e_μ läßt sich in der Kommutator-Schreibweise darstellen, die gerade umgekehrt zu der des äußeren Produkts ist

$$e_i \cdot e_\mu = \begin{cases} e_i \underline{\times} e_\mu & \text{falls } |\mu| \text{ gerade ist,} \\ e_i \overline{\times} e_\mu & \text{falls } |\mu| \text{ ungerade ist.} \end{cases}$$

Für das innere Produkt zweier κ -Vektoren $A_{\langle k \rangle}$ und $B_{\langle l \rangle}$ ergibt sich

$$A_{\langle k \rangle} \cdot B_{\langle l \rangle} = C_{\langle |k-l| \rangle} \,.$$

Daher ist das innere Produkt von $A_{\langle k \rangle}$ und $B_{\langle k \rangle}$ stets ein Skalar und im Gegensatz zum äußeren Produkt verschwindet i.a. weder $\mathbf{a} \cdot \mathbf{a}$ noch $\mathbf{a} \cdot \mathbf{b}$

$$\begin{aligned} \mathbf{a} \cdot \mathbf{b} &= \sum_i \alpha_i e_i \cdot \sum_j \beta_j e_j = \sum_{\substack{i,j \\ i=j}} \alpha_i \beta_j \, e_i \, e_j \\ &= \sum_i \alpha_i \beta_i \, e_i \, e_i = \sum_i \alpha_i \beta_i \, \lambda_i \in \mathbb{R} \,, \quad \lambda_i \in \Lambda \,. \end{aligned}$$

Synopsis der Produkte

In diesem Abschnitt werden die Produkte nicht mehr getrennt betrachtet. Es sollen einige nützliche Formeln angegeben werden, die das Rechnen in der geometrischen *Algebra* erleichtern.

Besonders wichtig ist die Priorität der genannten Produkte. Solange durch Klammerung nicht ausdrücklich festgelegt[3], gilt die Reihenfolge

$$\wedge \quad \text{vor} \quad \cdot \quad \text{vor} \quad GP\,.$$

Im folgenden sei etwa $A_{\langle k\rangle} = \mathbf{a}_1 \wedge \mathbf{a}_2 \wedge \ldots \wedge \mathbf{a}_k$ und $B_{\langle l\rangle} = \mathbf{b}_1 \wedge \mathbf{b}_2 \wedge \ldots \wedge \mathbf{b}_l$. Es sei weiterhin mit $1 \le j \le k$

$$[A_{\langle k\rangle} \setminus \mathbf{a}_j] = \mathbf{a}_1 \wedge \mathbf{a}_2 \wedge \ldots \wedge \mathbf{a}_{j-1} \wedge \mathbf{a}_{j+1} \wedge \ldots \wedge \mathbf{a}_k = A_{\langle k-1\rangle}\ .$$

Die Reihenfolge der übrigen Vektoren $\mathbf{a}_i$, $i \in \{1, 2, \ldots, k\} \setminus \{j\}$ bleibt in $A_{\langle k-1\rangle}$ erhalten.

Die beiden nachstehenden Formeln stellen ein gutes Hilfsmittel dar, um bestehende Terme umzuwandeln und sie weiter zu expandieren. Erstere kann als eine Quasi-Assoziativität für das innere Produkt verwendet werden. Rechenregeln, die für κ-Vektoren gültig sind, können auch auf *Basisblades* resp. Basiseinheitsvektoren angewandt werden.

$$\begin{aligned}
\left(A_{\langle r\rangle} \cdot B_{\langle s\rangle}\right) \cdot C_{\langle t\rangle} &= A_{\langle r\rangle} \cdot \left(B_{\langle s\rangle} \cdot C_{\langle t\rangle}\right) \qquad & s \ge r + t \\
\left(A_{\langle r\rangle} \wedge B_{\langle s\rangle}\right) \cdot C_{\langle t\rangle} &= A_{\langle r\rangle} \cdot \left(B_{\langle s\rangle} \cdot C_{\langle t\rangle}\right) \qquad & t \ge r + s
\end{aligned}$$

Hieraus läßt sich unter der Bedingung $k \le l$ zuerst

$$\begin{aligned}
A_{\langle k\rangle} \cdot B_{\langle l\rangle} &= \left([A_{\langle k\rangle} \setminus \mathbf{a}_k] \wedge \mathbf{a}_k\right) \cdot B_{\langle l\rangle} \qquad & k \le l \\
&= [A_{\langle k\rangle} \setminus \mathbf{a}_k] \cdot (\mathbf{a}_k \cdot B_{\langle l\rangle})
\end{aligned}$$

ableiten.
Es sei $\mathcal{L} := \{1, 2, \ldots, l\}$ und es gelte stets $\{j_1, j_2, \ldots, j_l\} = \mathcal{L}$. Dann erbringt eine andere Anwendung als Ableitung

$$A_{\langle k\rangle} \cdot B_{\langle l\rangle} = \sum_{\substack{\{j_1, j_2, \ldots, j_k\} \\ \subset \mathcal{L}}} \epsilon_{j_1 j_2 \cdots j_l} \left[A_{\langle k\rangle} \cdot (\mathbf{b}_{j_1} \wedge \mathbf{b}_{j_2} \wedge \ldots \wedge \mathbf{b}_{j_k})\right] \mathbf{b}_{j_{k+1}} \wedge \ldots \wedge \mathbf{b}_{j_l}\ , \quad k \le l\ ,$$

wo die Summe über alle $\binom{l}{k}$ unterschiedlichen Belegungen (k-Kombinationen) der $j_1, j_2, \cdots, j_k$ mit Indices aus $\{1, 2, \ldots, l\} = \mathcal{L}$ laufen soll. Die Permutation $(j_1\, j_2\, \cdots\, j_l)$ gehe durch eine Anzahl π von paarweisen Elementvertauschungen aus $(1\,2\,\cdots\,l)$ hervor. Dann ist

$$\epsilon_{j_1 j_2 \cdots j_l} = \begin{cases} +1 & \text{falls } \pi \text{ gerade ist,} \\ -1 & \text{falls } \pi \text{ ungerade ist.} \end{cases}$$

[3] Stellenweise gemachte redundante Klammerungen sollen betreffende Formeln eingängiger darstellen.

Als Beispiel sei $A_{\langle 2\rangle} = \mathbf{a}_1\wedge\mathbf{a}_2$ und $B_{\langle 4\rangle} = \mathbf{b}_1\wedge\mathbf{b}_2\wedge\mathbf{b}_3\wedge\mathbf{b}_4$.

$$\begin{aligned} A_{\langle 2\rangle}\cdot B_{\langle 4\rangle} \;=\; &+\;\big[A_{\langle 2\rangle}\cdot(\mathbf{b}_1\wedge\mathbf{b}_2)\big]\,(\mathbf{b}_3\wedge\mathbf{b}_4) \;-\; \big[A_{\langle 2\rangle}\cdot(\mathbf{b}_1\wedge\mathbf{b}_3)\big]\,(\mathbf{b}_2\wedge\mathbf{b}_4)\\ &+\;\big[A_{\langle 2\rangle}\cdot(\mathbf{b}_1\wedge\mathbf{b}_4)\big]\,(\mathbf{b}_2\wedge\mathbf{b}_3) \;+\; \big[A_{\langle 2\rangle}\cdot(\mathbf{b}_2\wedge\mathbf{b}_3)\big]\,(\mathbf{b}_1\wedge\mathbf{b}_4)\\ &-\;\big[A_{\langle 2\rangle}\cdot(\mathbf{b}_2\wedge\mathbf{b}_4)\big]\,(\mathbf{b}_1\wedge\mathbf{b}_3) \;+\; \big[A_{\langle 2\rangle}\cdot(\mathbf{b}_3\wedge\mathbf{b}_4)\big]\,(\mathbf{b}_1\wedge\mathbf{b}_2) \end{aligned}$$

Im nachfolgenden Teil werden ausschließlich Formeln angegeben, die Verknüpfungen mit Vektoren betreffen. Es gilt allgemein

$$\begin{aligned} \mathbf{a}\cdot B_{\langle l\rangle} &= \textstyle\sum_{i=1}^{l}(-1)^{i-1}\,\big(\mathbf{a}\cdot\mathbf{b}_i\big)\big[B_{\langle l\rangle}\setminus\mathbf{b}_i\big] && \text{bzw.}\\ B_{\langle l\rangle}\cdot\mathbf{a} &= \textstyle\sum_{i=0}^{l-1}(-1)^{i}\,\big[B_{\langle l\rangle}\setminus\mathbf{b}_{l-i}\big]\big(\mathbf{b}_{l-i}\cdot\mathbf{a}\big) && \text{und}\\ \big(A_{\langle k\rangle}\wedge\mathbf{a}\big)\cdot\mathbf{b} &= A_{\langle k\rangle}\,(\mathbf{a}\cdot\mathbf{b}) - \big(A_{\langle k\rangle}\cdot\mathbf{b}\big)\wedge\mathbf{a}\,. \end{aligned}$$

Es sei daran erinnert, daß Terme aus Vektoren wie $\big(\mathbf{a}_i\cdot\mathbf{b}_j\big)$ Skalare darstellen, die zudem in sich kommutieren.
Über Distributivität lassen sich folgende Erweiterungen zu vorigen Abschnitten finden

k gerade	k ungerade
$A_{\langle k\rangle}\,\underline{\times}\,\mathbf{a} = A_{\langle k\rangle}\cdot\mathbf{a}$	$A_{\langle k\rangle}\,\overline{\times}\,\mathbf{a} = A_{\langle k\rangle}\cdot\mathbf{a}$
$A_{\langle k\rangle}\,\overline{\times}\,\mathbf{a} = A_{\langle k\rangle}\wedge\mathbf{a}$	$A_{\langle k\rangle}\,\underline{\times}\,\mathbf{a} = A_{\langle k\rangle}\wedge\mathbf{a}$

Dies läßt sich auch anders ausdrücken:

$$\begin{aligned} \mathbf{a}\cdot B_{\langle l\rangle} &= \frac{1}{2}\Big(\mathbf{a}B_{\langle l\rangle} - (-1)^l\,B_{\langle l\rangle}\,\mathbf{a}\Big) && \text{und}\\ \mathbf{a}\wedge B_{\langle l\rangle} &= \frac{1}{2}\Big(\mathbf{a}B_{\langle l\rangle} + (-1)^l\,B_{\langle l\rangle}\,\mathbf{a}\Big)\,. \end{aligned}$$

Die Summe beider Formeln ermöglicht zusammengenommen

$$\mathbf{a}\,B_{\langle l\rangle} = \mathbf{a}\wedge B_{\langle l\rangle} + \mathbf{a}\cdot B_{\langle l\rangle}\,.$$

Speziell für Vektoren gilt $\mathbf{a}\wedge\mathbf{b} = \mathbf{a}\underline{\times}\mathbf{b}$ und $\mathbf{a}\cdot\mathbf{b} = \mathbf{a}\overline{\times}\mathbf{b}$. Aus

$$\begin{aligned} \mathbf{ab} &= \mathbf{a}\underline{\times}\mathbf{b} + \mathbf{a}\overline{\times}\mathbf{b} && \text{folgt}\\ \mathbf{ab} &= \mathbf{a}\wedge\mathbf{b} + \mathbf{a}\cdot\mathbf{b}\,. \end{aligned}$$

Das Reverse

Der Operator $\tilde{\ }$ bewirkt eine Verkehrung aller beteiligten Elemente des Objekts, auf das er angewandt wird.
Für ein *Basisblade* e_μ mit $\mu_i \in \mu$, $1 \le i \le m := |\mu|$, einem *Blade* $A_{\langle k\rangle}$ und einen Multivektor A gelten

$$\begin{aligned} \tilde{e}_\mu &= \big(e_{\mu_1}e_{\mu_2}\dots e_{\mu_m}\big)^\sim &&= e_{\mu_m}e_{\mu_{m-1}}\dots e_{\mu_1}\\ \tilde{A}_{\langle k\rangle} &= \big(\mathbf{a}_1\wedge\mathbf{a}_2\wedge\dots\wedge\mathbf{a}_k\big)^\sim &&= \mathbf{a}_k\wedge\mathbf{a}_{k-1}\wedge\dots\wedge\mathbf{a}_1\\ \tilde{A} &= \Big(\sum_\mu \alpha_\mu\,e_\mu\Big)^\sim &&= \sum_\mu \alpha_\mu\,\tilde{e}_\mu\,. \end{aligned}$$

Hier wurde vorausgesetzt, daß

$$(\alpha\, e_\mu)^\sim \equiv \alpha\, \tilde{e}_\mu\ .$$

Folgende Formeln sind auch für *Basisblades* gültig

$$\begin{aligned}
\tilde{A}_{\langle k\rangle} &= (-1)^{k(k-1)/2}\, A_{\langle k\rangle} \\
\big(A_{\langle k\rangle}\ B_{\langle l\rangle}\big)^\sim &= \tilde{B}_{\langle l\rangle}\, \tilde{A}_{\langle k\rangle} \\
\big(A_{\langle k\rangle} \wedge B_{\langle l\rangle}\big)^\sim &= \tilde{B}_{\langle l\rangle} \wedge \tilde{A}_{\langle k\rangle} \\
\big(A_{\langle k\rangle} \cdot B_{\langle l\rangle}\big)^\sim &= \tilde{B}_{\langle l\rangle} \cdot \tilde{A}_{\langle k\rangle}\ .
\end{aligned}$$

Das Inverse

Wie allgemein üblich wird das Inverse eines Objekts $\mathcal{O}$ durch $\mathcal{O}^{-1}$ symbolisiert. Das Inverse muß stets $\mathcal{O}\,\mathcal{O}^{-1} = 1$ oder $\mathcal{O}^{-1}\,\mathcal{O} = 1$ erfüllen.

Das Inverse eines *Basisblades* ist wohldefiniert als

$$e_\mu^{-1} = \frac{\tilde{e}_\mu}{e_\mu\, \tilde{e}_\mu}$$

Da für *Basisblades* e_μ und e_ν mit $e_\mu \neq e_\nu$ aber $\mu = \nu$

$$e_\mu\, e_\nu = e_\nu\, e_\mu$$

gilt, ist auch $e_\mu\, \tilde{e}_\mu = \tilde{e}_\mu e_\mu$. Daher kommutieren e_μ und e_μ^{-1}

$$e_\mu\, e_\mu^{-1} = e_\mu^{-1} e_\mu = 1\ .$$

Daß das geometrische Produkt $A_{\langle k\rangle}\, A_{\langle k\rangle}$ eines *Blades* $A_{\langle k\rangle}$ mit sich selbst gleich dem inneren Produkt $A_{\langle k\rangle} \cdot A_{\langle k\rangle}$ entspricht, erlaubt die Erweiterung des Konzepts des Inversen auf *Blades*[4]

$$A_{\langle k\rangle}{}^{-1} = \frac{\tilde{A}_{\langle k\rangle}}{A_{\langle k\rangle}\ \tilde{A}_{\langle k\rangle}}\ ,$$

da der Term im Nenner auf einen Skalar abbildet. Für *Multivektoren* läßt sich im Allgemeinen kein Inverses finden.

Über das Inverse kann eine Division definiert werden:

$$\begin{aligned}
\tfrac{e_\mu}{e_\nu} &= e_\mu\, e_\nu^{-1} &= \tfrac{e_\mu\ \tilde{e}_\nu}{e_\nu\ \tilde{e}_\nu} \\
\frac{A_{\langle k\rangle}}{B_{\langle l\rangle}} &= A_{\langle k\rangle}\, B_{\langle l\rangle}{}^{-1} &= \tfrac{A_{\langle k\rangle}\ \tilde{B}_{\langle l\rangle}}{B_{\langle l\rangle}\ \tilde{B}_{\langle l\rangle}}
\end{aligned}$$

Da das geometrische Produkt nicht kommutativ ist, muß

$$\begin{aligned}
e_\mu\, e_\nu^{-1} &\neq e_\nu^{-1} e_\mu \\
A_{\langle k\rangle}\, B_{\langle l\rangle}{}^{-1} &\neq B_{\langle l\rangle}{}^{-1} A_{\langle k\rangle}
\end{aligned}$$

beachtet werden.

[4]Dies gilt nur für *Blades* und daher nicht für alle $\mathcal{K}$ -Vektoren.

Das Duale

Das Duale eines *Basisblades* e_μ , κ -Vektors $A_{\langle k\rangle}$ oder *Multivektors* A ergibt sich aus dem geometrischen Produkt mit dem Inversen des Pseudoskalars $I := I_n$. Da I bzw. I^{-1} das Element höchsten Grades darstellt, reduziert sich das geometrische Produkt auf das innere Produkt. Das Duale wird durch ein Sternchen $*$ gekennzeichnet

$$\begin{aligned} e_\mu^* &= e_\mu I^{-1} &&= e_\mu \cdot I^{-1} \\ A_{\langle k\rangle}^* &= A_{\langle k\rangle} I^{-1} &&= A_{\langle k\rangle} \cdot I^{-1} \\ A^* &= AI^{-1} &&= A \cdot I^{-1}\,. \end{aligned}$$

Eine weitere Notation für das Duale ist die Dual-Klammer $[\![\ ]\!]$. Es ist

$$[\![\mathbf{a}_1\mathbf{a}_2\cdots\mathbf{a}_i]\!] \equiv \big(\mathbf{a}_1\wedge\mathbf{a}_2\wedge\ldots\wedge\mathbf{a}_i\big)\cdot I^{-1}$$

Über

$$\big(A_{\langle r\rangle}\wedge B_{\langle s\rangle}\big)\cdot C_{\langle t\rangle} = A_{\langle r\rangle}\cdot\big(B_{\langle s\rangle}\cdot C_{\langle t\rangle}\big) \qquad t \geq r+s$$

kann gezeigt werden, daß die Elemente einfach aus der Klammer herausgezogen werden können

$$\begin{aligned} [\![\mathbf{a}_1\mathbf{a}_2\mathbf{a}_3\mathbf{a}_4\cdots\mathbf{a}_i]\!] &= \mathbf{a}_1\cdot\big((\mathbf{a}_2\wedge\mathbf{a}_3\wedge\mathbf{a}_4\wedge\ldots\wedge\mathbf{a}_i)\cdot I^{-1}\big) \\ &= \mathbf{a}_1\cdot[\![\mathbf{a}_2\mathbf{a}_3\mathbf{a}_4\cdots\mathbf{a}_i]\!] \\ &= \big(\mathbf{a}_1\wedge\mathbf{a}_2\big)\cdot[\![\mathbf{a}_3\mathbf{a}_4\cdots\mathbf{a}_i]\!] \\ &= \big(\mathbf{a}_1\wedge\mathbf{a}_2\wedge\mathbf{a}_3\big)\cdot[\![\mathbf{a}_4\cdots\mathbf{a}_i]\!] \\ &= \vdots \\ &= \big(\mathbf{a}_1\wedge\mathbf{a}_2\wedge\mathbf{a}_3\wedge\mathbf{a}_4\wedge\ldots\wedge\mathbf{a}_i\big)\cdot[\![1]\!]\,. \end{aligned}$$

Das innere Produkt eines κ -Vektors mit seinem Dualen ist stets Null:

$$A_{\langle k\rangle}\cdot[\![A_{\langle k\rangle}]\!] = A_{\langle k\rangle}\cdot\big(A_{\langle k\rangle}\cdot I^{-1}\big) = \big(A_{\langle k\rangle}\wedge A_{\langle k\rangle}\big)\cdot I^{-1} = 0$$

Das Duale wird anläßlich dieser Orthogonalitätseigenschaft auch das Normale genannt.

Geometrische Interpretation der Algebra

In diesem Abschnitt wird ein Schritt von der als bekannt vorausgesetzten Standard-Vektoralgebra (SVA) hin zur geometrischen *Algebra* gemacht. Es sei das Ziel, einige Zusammenhänge bzw. praktische geometrische Bedeutungen herauszustellen. Hierzu wird mit den bereits eingeführten Produkten begonnen. Gegen Ende des Kapitels werden in Kürze einzelne Anwendungen der zugrunde liegenden **GA**-Konzepte im Mittelpunkt der Ausführungen stehen [18]. Zuletzt wird die Verwendung von **GA** in der projektiven Geometrie Anklang finden [48].

Die Standard-Vektoralgebra [32, 31] ist Teil der **GA**. Das Skalarprodukt der SVA entspricht dem inneren Produkt von Vektoren. Das ausschließlich in $\mathbb{R}^3$ definierte Kreuzprodukt $\mathbf{a}\times\mathbf{b}$ kann in der **GA** über das Duale ausgedrückt werden

$$\mathbf{a}\times\mathbf{b} = \big(\mathbf{a}\wedge\mathbf{b}\big)^*\,.$$

Um dies zu zeigen sei $\mathbf{a} = \alpha e_1$ parallel zu e_1 und $\mathbf{b} = \beta \cos\phi\, e_1 + \beta \sin\phi\, e_2$ ein Vektor in der (e_1, e_2)-Ebene mit einem Winkel ϕ zu $\mathbf{a}$. Dann ist in $\mathcal{G}_3$

$$\begin{aligned} \mathbf{a} \wedge \mathbf{b} &= \alpha e_1 \wedge (\beta \cos\phi\, e_1 + \beta \sin\phi\, e_2) \\ &= \alpha\beta \sin\phi\, e_1 \wedge e_2 \,, \end{aligned}$$

so daß

$$\begin{aligned} \left(\alpha\beta \sin\phi\, e_1 \wedge e_2\right)^* &= \left(\alpha\beta \sin\phi\, e_1 e_2\right) \overbrace{e_3 e_2 e_1}^{I_3^{-1}} \\ &= \alpha\beta \sin\phi\, e_1 e_2\, e_3 e_2 e_1 \\ &= \alpha\beta \sin\phi\, e_3 \;=\; \mathbf{a} \times \mathbf{b} \end{aligned}$$

einem Vektor senkrecht auf e_1 und e_2 entspricht, dessen Betrag dem Flächeninhalt des von $\mathbf{a}$ und $\mathbf{b}$ aufgespannten Parallelogramms gleicht.

Während bei dem Skalarprodukt eine rein metrische Funktion im Vordergrund steht, lassen sich durch das innere Produkt zusätzlich komplexere geometrische Operationen darstellen, indem es auf höherdimensionale Objekte angewandt wird. Mit den ab Seite 6 eingeführten Produkten auf $\mathcal{G}_n$ läßt sich zudem rechnen, ohne auf eine Koordinatenrepräsentation zurückgreifen zu müssen.

Das äußere Produkt als Teil des geometrischen Produkts ermöglicht die Bildung von *Blades* und somit eine Ausdehnung des Raums $\mathbb{R}^n$ auf $\mathcal{G}_n$ mit 2^n Dimensionen. Die Elemente der Berechnungen dieses Raums sind Unterräume [17, 18]. Beispielsweise formen Bivektoren den sog. 'Raum der planen Elemente' oder der 'orientierten Flächenelemente', während im Zusammenhang mit Trivektoren auch von 'orientierten Volumenelementen' gesprochen wird.
Allgemein läßt sich sagen, daß das äußere Produkt den Grad eines Objekts erhöht, während das innere Produkt den Grad verringert

$$A_{\langle k\rangle} \wedge B_{\langle l\rangle} = C_{\langle k+l\rangle} \quad \text{bzw.} \quad A_{\langle k\rangle} \cdot B_{\langle l\rangle} = C_{\langle |k-l|\rangle} \,.$$

Um die Auswirkungen des inneren Produkts zu verdeutlichen sei

$$\begin{aligned} \mathbf{c} &= \mathbf{a} \cdot (\mathbf{b}_1 \wedge \mathbf{b}_2) \\ &= (\mathbf{a} \cdot \mathbf{b}_1)\mathbf{b}_2 - (\mathbf{a} \cdot \mathbf{b}_2)\mathbf{b}_1 \end{aligned}$$

ein Vektor in der durch $\mathbf{b}_1$ und $\mathbf{b}_2$ aufgespannten Ebene $\mathbf{b}_1 \wedge \mathbf{b}_2$. Die Auswertung von

$$\begin{aligned} \mathbf{a} \cdot \mathbf{c} &= \mathbf{a} \cdot [\mathbf{a} \cdot (\mathbf{b}_1 \wedge \mathbf{b}_2)] \\ &= (\mathbf{a} \wedge \mathbf{a}) \cdot (\mathbf{b}_1 \wedge \mathbf{b}_2) \\ &= 0 \;\Rightarrow\; \mathbf{a} \perp \mathbf{c} \end{aligned}$$

ergibt, daß $\mathbf{a}$ senkrecht auf $\mathbf{c}$ steht ($\mathbf{a} \perp \mathbf{c}$). Die Anwendung des inneren Produkts in $\mathbf{c}$ bewirkt eine Auflösung der linearen Abhängigkeiten in Bezug auf Vektor $\mathbf{a}$.
Die geometrische Bedeutung des inneren Produkts zweier κ -Vektoren $A_{\langle k\rangle}$ und $B_{\langle l\rangle}$ kann wie folgt beschrieben werden [17]: *"$A_{\langle k\rangle} \cdot B_{\langle l\rangle}$ repräsentiert einen κ -Vektor innerhalb des*

Unterraums $B_{\langle l\rangle}$, der dem Komplement der Orthogonalprojektion von $A_{\langle k\rangle}$ auf $B_{\langle l\rangle}$ entspricht."

Das geometrische Produkt zweier κ-Vektoren $A_{\langle k\rangle}$ und $B_{\langle l\rangle}$ kann als Summe einer Reihe neuer κ-Vektoren $C_{\langle t\rangle}$ dargestellt werden, wo der jeweilige Grad t aus $(|k-l|), (|k-l|+2), (|k-l|+4), \ldots, (k+l)$ ist. Diese Terme enthalten geometrische Informationen über die Beziehungen von $A_{\langle k\rangle}$ zu $B_{\langle l\rangle}$, wie z.B. der aufgespannte Raum, der Schnitt, die relative Orientierung etc. Der κ-Vektor mit dem niedrigsten Grad $(|k-l|)$ resp. dem höchsten Grad $(k+l)$ entspricht dem Ergebnis des inneren Produkts resp. dem des äußeren Produkts.

Hier soll ein Beispiel gegeben werden, das zeigen soll wie die senkrechte Komponente $\mathbf{x}_\perp$ eines Vektors $\mathbf{x}$ bezüglich eines anderen Vektors $\mathbf{a}$ ermittelt werden kann. Nachstehende hilfreiche Identitäten lassen sich aus den vorherigen Abschnitten ableiten. So gilt für einen Vektor $\mathbf{a}$ und einen κ-Vektor $B_{\langle l\rangle}$, daß

$$\begin{aligned}
\mathbf{a}\cdot B_{\langle l\rangle} &= (\mathbf{a}_\perp + \mathbf{a}_\parallel)\cdot B_{\langle l\rangle} &&= \overbrace{\mathbf{a}_\perp \cdot B_{\langle l\rangle}}^{=0} + \mathbf{a}_\parallel \cdot B_{\langle l\rangle} &&= \mathbf{a}_\parallel \cdot B_{\langle l\rangle} \\
\mathbf{a}\wedge B_{\langle l\rangle} &= (\mathbf{a}_\perp + \mathbf{a}_\parallel)\wedge B_{\langle l\rangle} &&= \mathbf{a}_\perp \wedge B_{\langle l\rangle} + \underbrace{\mathbf{a}_\parallel \wedge B_{\langle l\rangle}}_{=0} &&= \mathbf{a}_\perp \wedge B_{\langle l\rangle}\ ,
\end{aligned}$$

wo $\mathbf{a}_\perp$ die Komponente des Vektors $\mathbf{a}$ darstellt, die nicht im Unterraum $B_{\langle l\rangle}$ enthalten ist bzw. senkrecht auf diesem steht. Der zweite Teil $\mathbf{a}_\parallel$ von $\mathbf{a}$ ist in $B_{\langle l\rangle}$ enthalten. Es ergibt sich zusammen

$$\mathbf{a}\,B_{\langle l\rangle} = \mathbf{a}_\perp \wedge B_{\langle l\rangle} + \mathbf{a}_\parallel \cdot B_{\langle l\rangle}\ .$$

Die Randbedingungen des Problems $\mathbf{x}_\perp \cdot \mathbf{a} = 0$ und $\mathbf{x}_\perp \wedge \mathbf{a} = \mathbf{x}\wedge\mathbf{a}$ können zum geometrischen Produkt ergänzt werden

$$\mathbf{x}_\perp \wedge \mathbf{a} + \mathbf{x}_\perp \cdot \mathbf{a} = \mathbf{x}_\perp\,\mathbf{a} = \mathbf{x}\wedge\mathbf{a}\,.$$

Es folgt die Lösung per Division

$$\mathbf{x}_\perp = \mathbf{x}_\perp\,\mathbf{a}/\mathbf{a} = (\mathbf{x}\wedge\mathbf{a})\,\mathbf{a}^{-1}\ .$$

GA in der projektiven Geometrie

Dieser letzte Teil des Kapitels soll einen abrißartigen Überblick des Arbeitens mit geometrischer *Algebra* in projektiver Geometrie vermitteln [48, 39, 34, 49].
Gegenstand einer Geometrie ist das Studium geometrischer Objekte im Raum, z.B. Kurven oder Oberflächen. Zudem behandelt sie die Beziehungen dieser Objekte zueinander, wie Schnitt, Abstand oder eingeschlossener Winkel. Eine Geometrie kann über die Invarianz gegenüber Gruppen geometrischer Transformationen charakterisiert werden. Der klassischste Vertreter ist die Euklidische Geometrie, in der ausschließlich rigide (starre) Transformationen erlaubt sind. Grundsätzlich dürfen alle Geometrien als eingebettet in eine universelle,

'projektive Geometrie' angesehen werden. Diese umfaßt die Gruppe der projektiven Transformationen (Kollineationen; invertierbare Abbildungen) und gründet auf der Vorstellung von Zentralprojektionen. Zu einer Geometrie gehört stets auch ein algebraischer Raum, auf dessen Elemente die Transformationen angewendet werden. Der dazugehörende $(n+1)$-dimensionale projektive Raum $\mathbb{PR}^n$ wird durch Basiserweiterung von $\mathbb{R}^n$ unter Hinzunahme des Basiseinheitsvektors e_{n+1} induziert. Die Vektoren $\mathbf{x} = \sum_i \alpha_i e_i$ aus $\mathbb{R}^n$ der SVA werden über die Einbettung

$$\mathcal{M} : \mathbf{x} \xrightarrow{\mathcal{M}} \mathbf{X} = \sum_i \alpha_i e_i + e_{n+1}$$

zu Elementen von $\mathbb{PR}^n$ erweitert. Die hinzugekommene sogenannte 'projektive Dimension' ermöglicht die Kumulation aller Skalierungen, die ein Element erfährt. Aufgrund dieser Konstruktion wäre es lediglich grob formuliert, projektive Geometrie als 'lineare Algebra - erweitert um einen Skalar' [39] zu bezeichnen.
Die Vektoren aus $\mathbb{PR}^n$ werden 'homogene Vektoren' genannt und durch fette große Buchstaben $\mathbf{X}, \mathbf{Y}$ usw. symbolisiert. Die Koordinaten heißen entsprechend 'homogene Koordinaten'.
Der zur Einbettung umgekehrte Vorgang

$$x = \sum_i \frac{\mathbf{X} \cdot e_i}{\mathbf{X} \cdot e_{n+1}} e_i$$

kann als Deskalierung oder Rückprojektion bezeichnet werden. Er läßt den homogenen Vektor $\mathbf{X}$ wieder auf $\mathbf{x}$ runterprojizieren. Abbildung 2.1 visualisiert dies anhand der beschaulichen Verhältnisse in $\mathbb{PR}^2$.

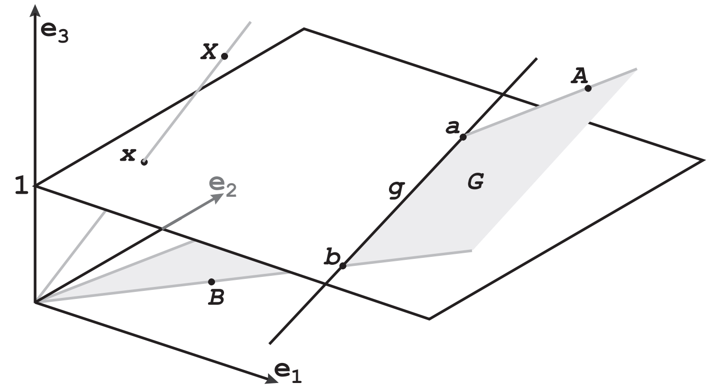

Abb. 2.1: Projektion von $\mathbf{A}, \mathbf{B}, \mathbf{X}, \mathbf{G}$ aus $\mathbb{PR}^2$ auf die einbettende Ebene $\mathbb{R}^2$.

Die Hyperebene $H_\infty \subset \mathbb{PR}^n$ der Elemente mit $x_{n+1} = 0$ besteht aus den sogenannten 'Punkten im Unendlichen', die unter Rückprojektion dem Sinne nach in das Unendliche projezieren. Sie werden als Richtungen interpretiert und stellen somit eine Trennung von realem Punkt und Richtung dar, die im Euklidischen fehlt. Dort wird zwischen einem 0D-Punkt und einer 1D-Gerade nicht unterschieden, obwohl beide verschiedene Räume repräsentieren. Die Trennung ermöglicht eine einheitliche rechnerische Behandlung aller Elemente der projektiven Geometrie, ohne Spezialfälle (z.B. Ausnahmebehandlungen in Computer-Algorithmen) berücksichten zu müssen.

Jeder Vektor mit $x_{n+1} \neq 0$ definiert unter Homogenisierung[5] seine eigene Äquivalenzklasse. So fallen alle homogenen Vektoren $\alpha\mathbf{X}$, $\alpha \in \mathbb{R} \setminus \{0\}$ in die Äquivalenzklasse des Vektors $\bar{\mathbf{X}} = \bar{\alpha}\mathbf{X}$ mit $\bar{x}_{n+1} = 1$. Sie beschreibt über die Menge ihrer Elemente eine projektive Gerade in $\mathbb{PR}^n$, die im einbettenden Raum mit einem Punkt korrespondiert. Nach dem gleichen Schema repräsentiert eine projektive Ebene eine Gerade. Dies zeigt das Beispiel aus Abbildung 2.1, wo die projektive Ebene G eindeutig die Lage der Geraden g festlegt. Analog stellt eine projektive Hyperebene aus $\mathbb{PR}^n$ eine Ebene dar. In der projektiven Geometrie und im folgenden werden die Bezeichnungen gemäß den vorangegangenen Schilderungen gewechselt, so daß z.B. der Ausdruck 'Gerade' statt '(projektive) Ebene' verwendet wird.
Die 'Extradimension' der projektiven Geometrie macht ein Umdenken seitens der Benennung ihrer geometrischen Objekte erforderlich. Jetzt sollen beispielhaft einige praktische Anwendungen der **GA** in der projektiven Geometrie erläutert werden.

Zwei Punkte $\mathbf{X}, \mathbf{A}$ aus $\mathbb{PR}^n$ sind identisch, wenn sie auf der gleichen projektiven Geraden liegen. Dann gilt wegen der linearen Abhängigkeit

$$\mathbf{X} \wedge \mathbf{A} = 0\,.$$

Hieraus ergibt sich die äquivalente Bedingung $[\![\mathbf{X}\mathbf{A}]\!] = 0$ bzw. $\mathbf{X} \cdot [\![\mathbf{A}]\!] = 0$. Wird $\mathbf{N}_a \equiv [\![\mathbf{A}]\!]$ als das Normale von $\mathbf{A}$ bezeichnet, so ist

$$\mathbf{X} \wedge \mathbf{A} = 0 \quad \Longleftrightarrow \quad \mathbf{X} \cdot \mathbf{N}_a = 0\,.$$

Für den Test, ob ein Punkt $\mathbf{X}$ auf einer Geraden (proj. Ebene) $\mathbf{L}_{ab} := \mathbf{A} \wedge \mathbf{B}$ liegt, kann ebenso verfahren werden:

$$\mathbf{X} \wedge \mathbf{L}_{ab} = 0 \quad \Longleftrightarrow \quad \mathbf{X} \cdot \mathbf{N}_{ab} = 0,$$

wo $\mathbf{N}_{ab}$ das Duale resp. Normale von $\mathbf{L}_{ab}$ sei. Punkt $\mathbf{X}$ liegt auf der Geraden, falls er eine Darstellung gemäß $\mathbf{X} = \alpha_1\mathbf{A} + \alpha_2\mathbf{B} \neq 0$ mit $\alpha_1, \alpha_2 \in \mathbb{R}$ besitzt.

Es seien $\mathbf{L}_{ab} = \mathbf{A} \wedge \mathbf{B}$ und $\mathbf{L}_{cd} = \mathbf{C} \wedge \mathbf{D}$ zwei Geraden aus $\mathbb{PR}^2$. Ihr Schnittpunkt $\mathbf{X}$ kann mit Hilfe ihrer dualen Repräsentationen ermittelt werden. Es muß das innere Produkt des einen Objekts mit dem Dualen des anderen gebildet werden

$$\mathbf{X} = \mathbf{N}_{ab} \cdot \mathbf{L}_{cd} = \mathbf{L}_{ab} \cdot \mathbf{N}_{cd}\,.$$

Der Schnittpunkt zweier paralleler Geraden liegt im Unendlichen und ergibt daher eine Richtung in Form eines Punktes im Unendlichen. Das Verfahren zur Schnittberechnung kann analog zum vorigen Absatz auf höherdimensionale Objekte erweitert werden.

[5]Reskalierung eines Vektors, so daß $e_{n+1} = 1$ gilt; Der Vektor befindet sich wieder in der einbettenden Hyperebene.

Kapitel 3

Arbeitsplattform

In den folgenden Abschnitten wird die dem GA-Prozessor zugrunde liegende Soft- und Hardware dieser Arbeit erläutert. Die Software [19] besteht aus den zum Chip gehörenden Komponenten und den selbst entwickelten `C++`-Modulen zum Programmieren und Steuern des *FPGAs*. Die Hardware ist eine *PCI*-Karte, auf der sich unter anderem der *FPGA*-Chip befindet [20]. Benötigte Grundlagen können z.B. [29, 38, 24, 4, 36] entnommen werden.

3.1 FPGA-Mutterplatine

Der *FPGA*-Chip bildet die zentrale Recheneinheit der *PCI*-Erweiterungskarte (Abbildung 3.1). Ebenfalls auf der Platine befindet sich ein asynchrones *SRAM*, Verbindungsbuchsen (Connector) zum Anschluß von externer Hardware, ein *PCI-Interface* (*PCI* bridge chip) sowie ein *Clock*&Support-System.

Das *SRAM* wird per 64 Bit Datenleitung exklusiv vom *FPGA* genutzt. Die maximale Frequenz liegt bei etwa 40 MHz und wird im wesentlichen durch die Asynchronität begrenzt. Von den 64 Bit sind 36 Bit Rohdaten, 21 Bit Adressdaten[1] und 7 Bit Steuerdaten. Es gibt zwei Verbindungsbuchsen der Breite 64 Bit, die der *S-Link*-Spezifikation und die der *CMC*-Spezifikation. Jede Buchse besitzt einen eigenen Taktgenerator und kann mit bis zu 120 MHz getaktet werden. Das *PCI-Interface* der Karte zum Datenaustausch mit dem Computer ist über einen lokalen *Bus* mit dem *FPGA* verbunden. Es kann mit 10 MHz bis 40 MHz getaktet werden und hat *Fifo*-Charakteristika, da der computerseitige Teil der Schnittstelle mit 33 MHz arbeitet. Es ist 32 Bit breit[2] und zeitlich gemultiplext, d.h. der *Bus* ist abwechselnd mit Adresse und zugehörigem Datum belegt. Weiterhin gibt es den *Burst*-Zyklus, wo anfangs nur eine Startadresse übertragen wird, gefolgt von den Daten entsprechend den aufsteigenden Adressen. Die maximale theoretische Bandbreite des *PCI*-Busses liegt bei 132 MByte/s. Sie wird in der Praxis (ohne *Burst*) bei weitem nicht erreicht. In dem *Clock* & Support-System befinden sich alle Taktgeneratoren. Neben den bereits genannten gibt es noch zwei weitere (Clock1, Clock2), die den *FPGA* betakten. Clock1 ist phasensynchron zu allen anderen Taktsignalen (bis auf die des local bus) und kann mit bis zu 120MHz laufen. Die langsamere

[1] Nur die untersten 19 Bit werden verwendet, da das vorhandene *SRAM* 2 MB (genau: $2^{19} \cdot 36$ Bit) groß ist. Alle Elemente im *Design* sind dennoch bereits auf 21 Bit ausgelegt.

[2] Daher verwendet das *Design* nur die untersten 32 der 36 Bit des *SRAMs*.

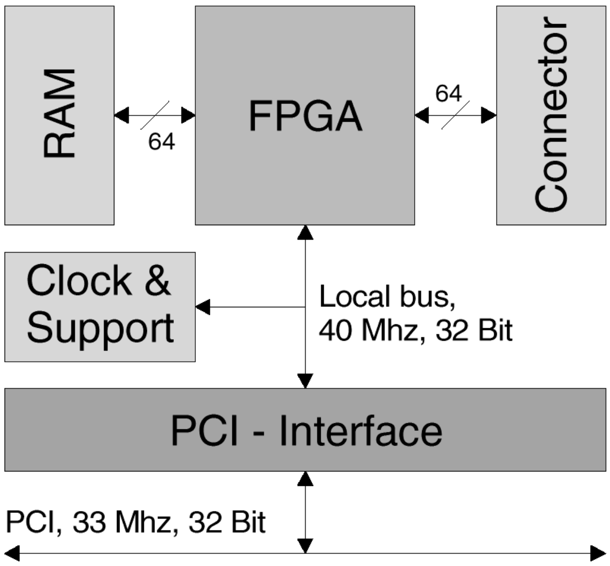

Abb. 3.1: Schematische Darstellung der Einbettung des *FPGAs* auf der verwendeten *PCI*-Karte.

Clock2 ist wählbar im Verhältnis 2 oder 4 zu Clock1 untersetzt. Eine der Clocks kann zum Betreiben des *SRAMs* benutzt werden. Überdies befindet sich im *Clock* & Support-System eine Steuerlogik zur Konfiguration des *FPGAs* und eine Funktion zum Auslesen von dessen internen Zuständen. Dies und ein *Einzelschrittmodus* erlaubt den Betrieb eines Software-Online-Hardware-*Debuggers* zum Überwachen aus dem laufenden Betrieb.

3.2 Field programmable gate array (FPGA)

FPGAs gehören zu der Gruppe der programmierbaren Logikschaltungen (*PLDs*), zu der sich auch *PROMs*, *PALs* und *PLAs* zählen [38]. Im Gegensatz zu diesen besteht ein *FPGA* nicht aus UND- und ODER-Matrizen mit programmierbaren Verbindungen, sondern aus reihenweise angeordneten Logikblöcken (siehe Abbildung 3.2). Diese sind, wie auch die Verdrahtung zwischen ihnen, beliebig mehrfach und frei programmierbar. Die $56 \times 56 = 3136$ Logikblöcke (*CLBs*, siehe Abb. 3.3 auf Seite 22) des verwendeten *FPGAs* [23] bestehen im wesentlichen aus zwei *LUTs* (LookUpTable), zwei *FlipFlops* (1 Bit Speicher) sowie einer speziellen *Carry*-Logik zur beschleunigten Verarbeitung von Überträgen, z.B. für Additionen. Innerhalb der durch die Komplexität gesetzten Grenzen können sie dazu verwendet werden, jeden beliebigen endlichen Automaten umzusetzen. Da die *LUTs* als RAM-Zellen realisiert sind und nicht mehr von einem *Gate*-Array gesprochen werden kann, gehört der *FPGA* zur Unterklasse der *LCAs* (logic cell array).
Die RAM-*LUTs* besitzen jeweils vier Adressleitungen, so daß jeder einzelne eine Schaltfunktion mit vier Eingängen abdecken kann. Durch eine zusätzliche Logik in jedem *CLB* können sie während der *FPGA*-Laufzeit als 16 Bit tiefes, 1 Bit breites, sehr schnelles RAM angesprochen werden. Der in Abbildung 3.3 dargestellte *CLB* besitzt eine große Anzahl von Steuereinheiten (Multiplexern), wodurch Flexibilität in der Programmierbarkeit entsteht. Dies zeigt

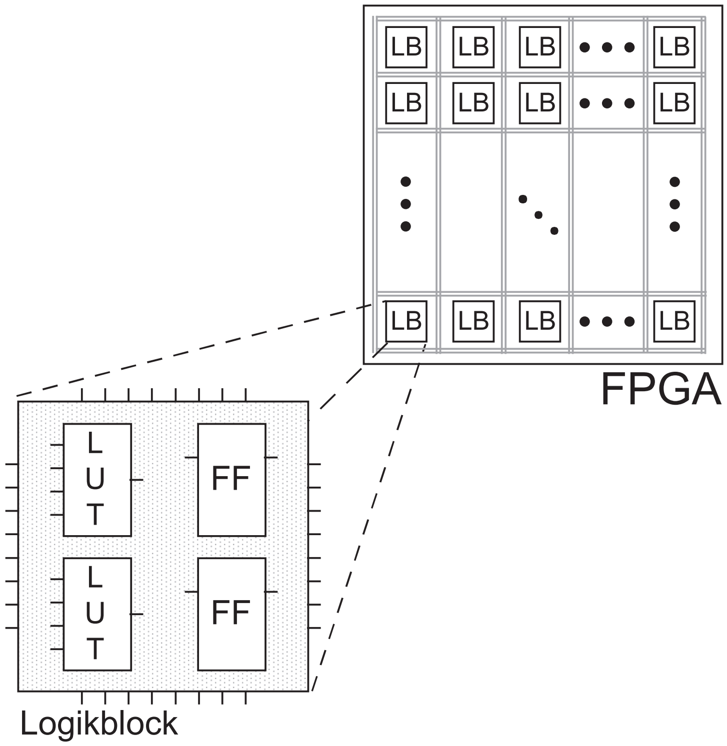

Abb. 3.2: Prizipieller Aufbau eines *FPGAs* aus programmierbaren Logikblöcken und Verdrahtungen.

sich z.B. an der Menge der zu einem *FlipFlop* durchschaltbaren Signale oder der Kombination der Ausgänge der *LUTs*. Zusammen mit dem nachgestellten kleineren 3er-Funktionsblock, dem sog. H-Block, kann u.a. eine maximal neunstellige Funktion einfacher Art (z.B. *XOR*, AND, OR) konstruiert werden.

3.3 Softwarekomponenten

In den sich anschließenden Unterabschnitten werden alle wichtigen Softwarekomponenten, deren Zusammenspiel und Abfolge dargelegt.

3.3.1 `C++` hardware definition language (CHDL)

Die Hardwarebeschreibungssprache *CHDL* ähnelt *VHDL*, basiert jedoch auf `C++` und ist daher leicht zugänglich [19, 40]. *CHDL* ist eine `C++`-Klassenbibliothek und erlaubt eine objektorientierte Entwicklung von *FPGA-Designs*. Die an der Universität Mannheim entwickelte Software ermöglicht es, solche *Designs* in einem Simulatorprogramm zu simulieren und zu testen.

Die umfangreiche Klassenbibliothek enthält vordefinierte Bauteile aus allen Bereichen. Dies sind arithmetische, logische, propagierende, verbindende, verstärkende, speichernde oder undifferenzierte Elemente sowie solche, die die Ein- und Ausgabe betreffen. Beispiele hierfür sind der Reihe nach Addierer, UND-Gatter, Multiplexer, Nodes, Buffer, *FlipFlops/Register*, RAM- sowie PinIn/PinOut-Elemente [27, 28, 36]. Die Bauteile können leicht erstellt, initialisiert und verbunden werden:

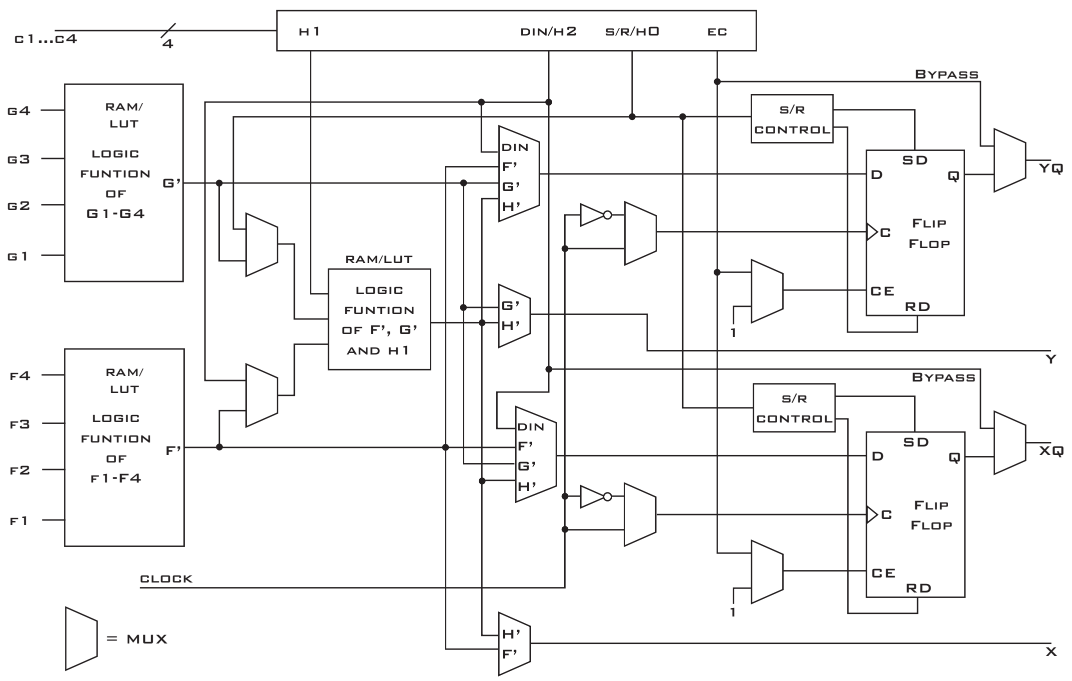

Abb. 3.3: Genauer Aufbau eines Logikblocks (ohne *Carry*-Logik).

```
0  Node InputNode;                   // Dateneingang
1  Node ClkIn;                       // Taktsignal
2  Node OutputNode;                  // Datenausgang
3
4  DFF MemDFF("MemDFF");             // erstelle D-FlipFlop
5  MemDFF.Init(0);                   // initialisiere zu 0
6  MemDFF.C = ClkIn;                 // verbinde Takt
7  MemDFF.CE = High;                 // setze ClockEnable
8  MemDFF.D = MemDFF.Q ^ InputNode;  // setze DFF-Eingang
9
10 OutputNode = MemDFF.Q;            // DFF-Ausgang
```

Bei der Verbindung von Bauteilen mit mehreren Ein- oder Ausgängen wird die Notation

`Bauteilname.Eingangsname = Bauteilname.Ausgangsname`

verwendet. Ist implizit klar was verbunden werden muß, kann der Qualifizierer entfallen:

`DFF1 = DFF2` statt **`DFF1.D = DFF2.Q`**.

In den ersten drei Zeilen des obigen C++-Codes werden die Ein- und Ausgänge definiert. In Zeile 4 wird ein sog. *D-FlipFlop* erstellt. Das Element bekommt den gleichen symbolischen Namen, den auch das Objekt trägt. Zeile 5 ruft die Memberfunktion 'Init()' auf, die das *FlipFlop* zu Null initialisiert. Zeile 6 verbindet den *Clock*-Eingang **`C`** des *DFF*'s mit dem Taktsignal. Zeile 7 setzt ClockEnable **`CE`** auf *High* [3], so daß der am *DFF*-Eingang aktuell anliegende Wert (0 oder 1) zu jeder steigenden *Flanke* im *FlipFlop* gespeichert wird. In Zeile 8 wird der *DFF*-Eingang verbunden - diesem wird ein *XOR*-Element vorgeschaltet, erkennbar am Dach. Der *DFF*-Eingang ist genau dann 1, wenn entweder der *DFF*-Ausgang 0 und der Eingang 'InputNode' 1 ist, oder falls dieser 0 und der *DFF*-Ausgang 1 ist. In Zeile 10 wird der Ausgang gesetzt.

[3] *Low* und *High* symbolisieren die logischen Zustände Null und Eins.

Das oben definierte Bauteil besteht aus einem *D-FlipFlop* und einem *XOR-Gatter* mit 2 Eingängen (Abb. 3.4). Durch Aufruf einer zu *CHDL* gehörenden Funktion kann ein *Design*

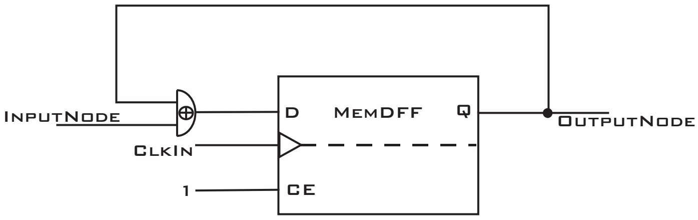

Abb. 3.4: Schaltbild des obigen Beispiels.

synthetisiert werden. Das bedeutet, daß eine Netzliste des *Designs* erstellt wird, in der sich die Verdrahtung der verwendeten Bauteile wiederfindet. Dies entspricht einer kryptischeren Darstellung der gleichen Information, die der `C++`-Code enthält, nur daß die Netzliste in einem standardisierten Format als Eingabe für Place & Route Programme vorliegen muss. Vor dem Generieren der Netzliste wird ein erster Check des *Designs* durchgeführt, in dem u.a. geprüft wird, ob wichtige Ein- oder Ausgänge nicht gesetzt sind, ob es bei den Elementen Namenskollisionen gibt oder ob eine Leitung mit mehreren Quellen verbunden ist.

Simulation

Neben der Funktion zum Synthetisieren gibt es in *CHDL* eine Funktion zum Simulieren eines *Designs*. Hierzu wird die *FPGA*-Hardware nicht benötigt. Bei deren Simulation werden Verzögerungszeiten des Designs berücksichtigt [27], so daß eine laufende Simulation fast eine Garantie für die Lauffähigkeit auf dem *FPGA* selbst ist. Nur wenn die Grenzen der Bauteile stark ausgereizt werden, kann die Praxis von der Theorie abweichen.

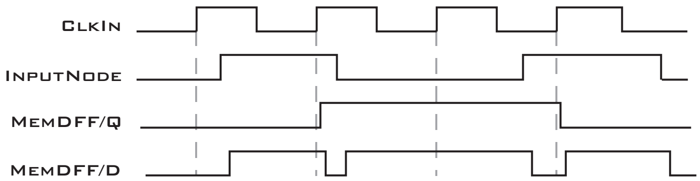

Abb. 3.5: Schaltverhalten des obigen Beispiels. InputNode ändert sich asynchron. Die Signalflanken sind idealisiert als senkrechte Übergänge dargestellt.

Die Simulation hat die Vorteile, daß die Hardware nicht benötigt wird, daß jedes Signal zu jedem Zeitpunkt an jedem Ort beobachtet werden kann und daß die volle Funktionalität des *Designs* verfügbar ist. Dies geht zu Lasten der Geschwindigkeit, dem Nachteil der Simulation, kann doch die Hardware über 1000 mal schneller sein. Das Zeitdiagramm in Abbildung 3.5 stellt dar, auf welche Weise Signale verfolgt werden können. Die symbolischen Namen der Bauteile sind die für die Simulation wichtigen Bezeichner der Signale.

3.3.2 Bitstream-Generierung

Die Bitstream-Generierung ist das Ziel des nächsten Schritts, nachdem das *Design* als Netzliste (siehe Abschnitt 3.3.1) vorliegt. Hierbei wird das *Design* noch mal gründlich, z.B. auf Timingprobleme, überprüft. Dann wird nach Optimierungsmöglichkeiten gesucht, wobei z.B.

Konstrukte wie

$$H \dot{\vee} ((F \dot{\vee} H) \wedge ((F \dot{\vee} H) \vee G))$$

durch F ersetzt/gekürzt werden [35].

Anschließend wird der Place & Route Prozess gestartet, bei dem durch ein iteratives Verfahren die besten Positionen sowie die kürzesten Leitungswege auf dem *FPGA*-Chip für alle zu realisierenden Bauteile gesucht werden [24]. Bei diesem Optimierungsprozess kann zusätzlich gewichtet werden, ob der Platz oder eher die Signalwege und damit die Signallaufzeiten minimiert werden sollen. Am Ende des ganzen Vorgangs wird ein Bitstream-File erzeugt, welches auf den *FPGA* geladen werden muß, um ihn zu konfigurieren resp. zu initialisieren.

3.3.3 Treiber

Vom Kartenhersteller wird eine Funktionsbibliothek für die Programmierung der Karte mitgeliefert. Diese enthält Funktionen zum Initialisieren, zum Laden und Entladen von *Designs*, zum Setzen von Taktfrequenzen, zum Datenaustausch während der Laufzeit, zur Fehlerdiagnostik sowie weitere Supportfunktionen [20].

Der gewöhnliche Ablauf bei der Benutzung des *FPGAs* ist zuerst die Karte zu initialisieren. Anschließend werden die Taktfrequenzen auf der Karte gesetzt, mit denen das *Design* später laufen soll. Jetzt wird der im vorigen Abschnitt 3.3.2 beschriebene Bitstream auf die Karte geladen. Der *FPGA* ist betriebsbereit und erhält per Datenaustausch mit dem PC seine Eingabedaten sowie das Signal, die Berechnugen zu starten. Ist alles ausgeführt, signalisiert dies das *Design* auf dem *FPGA* und der Datenaustausch in umgekehrter Richtung wird eingeleitet, d.h. die Ergebnisse werden ausgelesen.

Kapitel 4

Realisierung des Prozessors für geometrische Algebra

In den nachstehenden Ausführungen dieses Kapitels werden Funktionsweise, Aufbau und Mechanismen des Prozessors für geometrische *Algebra* erläutert. Es wird auf die Hauptkomponenten sowie auf alle verwendeten Grundbausteine und deren Eigenschaften in Bezug auf Antwortverhalten, Timing etc. eingegangen. Es wird die zur Anwendung gebrachte Speicherorganisation, die Formate der Steuer- und Datenworte, sowie die Kommunikation zwischen PC und *FPGA*-Design beschrieben. Für das Arbeiten mit Rechnerarchitekturen sei auf [44, 46, 25, 36] verwiesen.

4.1 Struktur, Prozesse und Datenpfade

Das *Design* (Abb. 4.1) besteht im wesentlichen aus drei großen Bausteinen; der Kontrolleinheit, der Berechnungseinheit sowie der Adreßeinheit. Weiterhin existiert ein kleineres Bauteil, welches, parallel zur Berechnungseinheit geschaltet, Datenabhängigkeiten innerhalb der *Pipeline* erkennt. Näheres hierzu folgt erst in Abschnitt 5.2 ab Seite 48.

Zur Kommunikation per *PCI* muß zusätzliche Logik in Form des sog. `PCI_SLAVE` Moduls auf dem *FPGA* untergebracht werden [29]. Hierdurch werden u.a. bestimmte Protokolle sowie Timingbedingungen des *PCI* bridge chips (*PLX* 9080) der Karte eingehalten.

Im anschließenden Unterabschnitt wird ein Querschnitt durch den ersten Teil dieses Kapitels gemacht. Es soll die wichtigsten Dinge verdeutlichen, um einen ersten Überblick zu gewinnen.

4.1.1 Generelle Einführung

Folgender Text skizziert im groben Rahmen die Situation im normalen Betrieb mit Betonung auf den nächsten Unterabschnitt.

Nach Abschluß des Datentransfers zwischen PC und *FPGA*-Board befinden sich die vom **GA**-Koprozessor auszuführenden *Instruktionen* im *SRAM*. Jede besitzt zwei Operanden - die

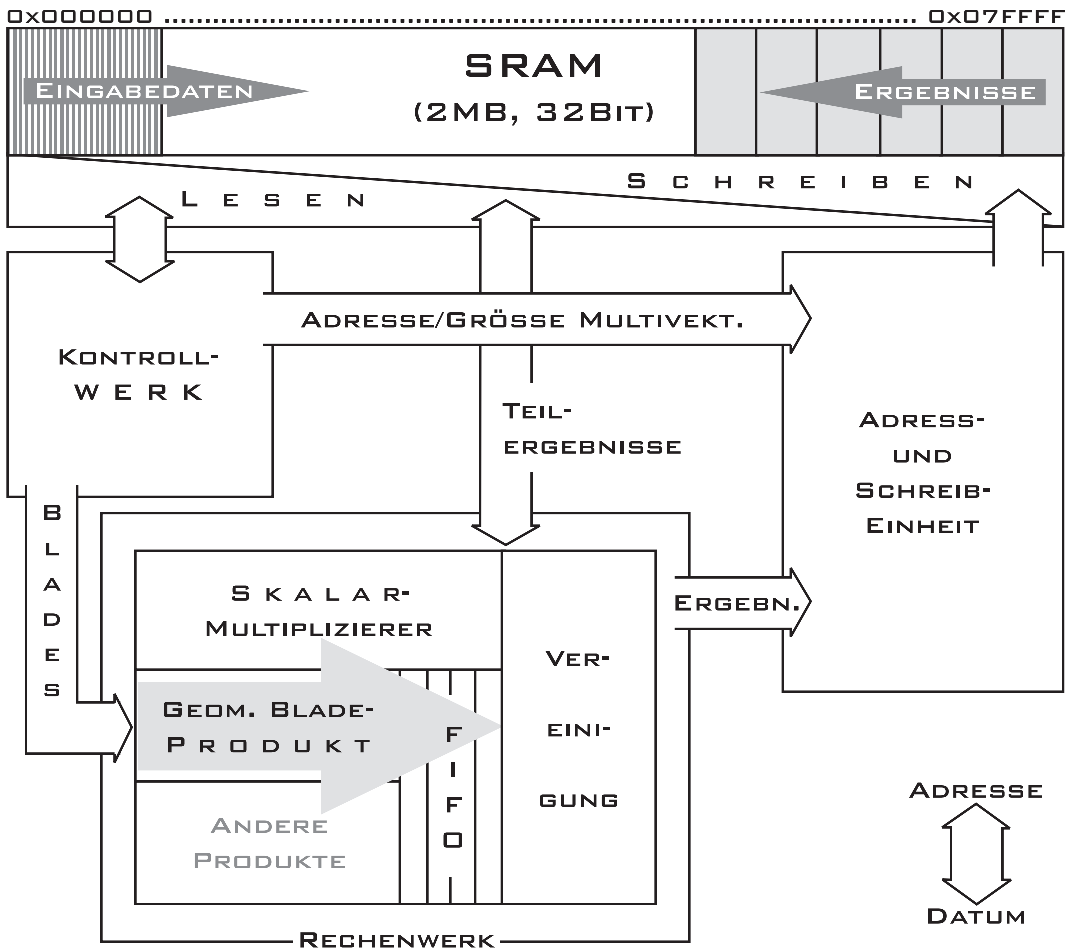

Abb. 4.1: Hauptdatenpfade. Im *SRAM* wachsen die Ergebnisse in Richtung der Eingabedaten, d.h. gen kleinerer Adressen (vgl. einem *Stack*-Speicher).

Multivektoren MVA und *MVB* - aus denen das geometrische Produkt gebildet werden soll. Zu diesem Zweck werden die einzelnen Elemente (*Blades*) von *MVB* distributiv mit Multivektor A verknüpft[1]. Detaillierter formuliert muß sukzessive je ein B-*Blade* aus *MVB* mit sämtlichen A-*Blades* von *MVA* multipliziert werden, d.h. das geometrische Produkt gebildet werden. Das Auslesen der *Blades* in der richtigen Reihenfolge erfolgt in schleifenartigen Durchläufen, die vom Adreßwerk koordiniert werden. Die Daten eines *Durchgangs* einer *Instruktion*, je ein B-*Blade* gefolgt von allen A-Baldes, gelangen über das Kontrollwerk in die Berechnungseinheit. Aus m B-*Blades* und n A-*Blades* werden hier $m \times n$ Ergebnisblades gebildet, die i.A. auf weniger *Basisblades* abbilden (siehe Kapitel 2), so daß *Blades* mit identischem geometrischen Anteil aufaddiert werden müssen. Hierbei entstehen Zwischenresultate, die wegen der Datenmenge ins *SRAM* ausgelagert werden. Wenn diese vorläufigen Werte wieder benötigt werden, fordert sie das Rechenwerk dynamisch an, in Abb. 4.1 durch den Datenpfeil 'Teilergebnisse' symbolisiert. Die ggf. wieder nicht endgültigen Resultate werden mittels der Adreßeinheit im *SRAM* gespeichert.
Nachdem alle Instrukionen der Reihe nach abgearbeitet wurden, werden die Ergebnisse auf den PC heruntergeladen.

[1]Eine umgekehrt definierte Reihenfolge (*MVA* auf *MVB*) wäre ebenso möglich gewesen.

4.1.2 Berechnungseinheit

Die Berechnungseinheit (Abbildung 4.2 auf Seite 28) bildet das geometrische Produkt zweier *Blades*. Ihre Einbettung im *Design* zeigt Abbildung 4.1 auf Seite 26.

Ausgelesene *Blades*, die wesentlichsten Eingabedaten des Rechenwerks, passieren das Kontrollwerk (Abb. 4.4, S. 32), wo sie zuvor in einen skalaren sowie in einen geometrischen Anteil aufgespalten werden. Diese zwei Komponenten sind mit den Eingangsleitungen **`ScalarIn`** bzw. **`BladeIn`** der Berechnungseinheit verbunden. Jede der Leitungen mündet zugleich in einem A- und B-Operanden-*Register*. Die **`SelA`** und **`SelB`**-Signale des Kontrollwerks geben an, ob es sich bei den anliegenden Daten um A- oder B-Operanden handelt und sind folglich niemals simultan auf *High*. Sie wählen genau eines der beiden *Register* als Ziel aus, indem nur dessen **`CE`**-Eingang auf einen hohen *Pegel* gelegt wird (in der Abbildung durch seitliche Anschlüsse symbolisiert).
Die **`Signature`**-Leitung kommt vom Kontrollwerk. Dort wird die für die algebraischen Berechnungen benötigte Signaturinformation, die sich im Laufe der Designzeit i.A. nicht ändert, gespeichert.
Der Eingang **`ValidSig`** wird einige Takte später zu dem Ausgangssignal **`ValidSigOut`** und bestimmt dann, ob ein *Blade* von der Adreßeinheit gespeichert werden soll. Dies wäre der Fall, wenn die Daten an **`ScalarIn`** und **`BladeIn`** nach Durchlaufen des Rechenwerks ein *gültiges* Resultat (*Blade*) hervorbringen. **`ValidSig`** wird vom Kontrollwerk generiert und ist gesetzt, falls folgende Punkte erfüllt sind.

1. An **`ScalarIn`** und **`BladeIn`** liegt ein Operand an.

2. Das *Blade* ist Element von Multivektor *MVA*.

3. In einem Operanden-*Register* befindet sich das zu dem gleichen *Durchlauf* gehörige B-*Blade* aus *MVB*.

4. Alle betreffenden Skalaranteile sind ungleich Null.

Anders ausgedrückt ist **`ValidSig`** *High*, falls sich durch das momentan anliegende *Blade* ein *gültiges* Paar ergibt, dessen geometrisches Produkt einen von Null verschiedenen Skalaranteil haben wird.
`ValidSig` verläuft über eine Kette von *FlipFlops* datenbegleitend durch die Register des Rechenwerks und unterscheidet auf diese Weise gültige von zu verwerfenden Daten. Dadurch ist es möglich, daß die *Pipe* der Berechnungseinheit während des Betriebs nie still stehen muß, sondern die nicht als gültig markierten Daten ignorieren kann.

Die an die **`BladeIn`**-Leitungen angeschlossenen Eingangsregister sind mit allen algebraischen Bauteilen[2] über acht Bit breite Leitungen verbunden. Alle Elemente liefern je ein vorzeichenbehaftetes Resultatblade. Der größte Aufwand liegt hierbei in der Berechnung des Vorzeichens (Sign). Das resultierende *Blade* besteht dagegen, am Beispiel des geometrischen Produkts, aus einer einfachen paarweisen *XOR*-Verknüpfung der Bits der Bladeoperanden und steht fast instantan zu Beginn eines *Taktzyklusses* fest.

[2]Zur Zeit nur geometrisches Produkt.

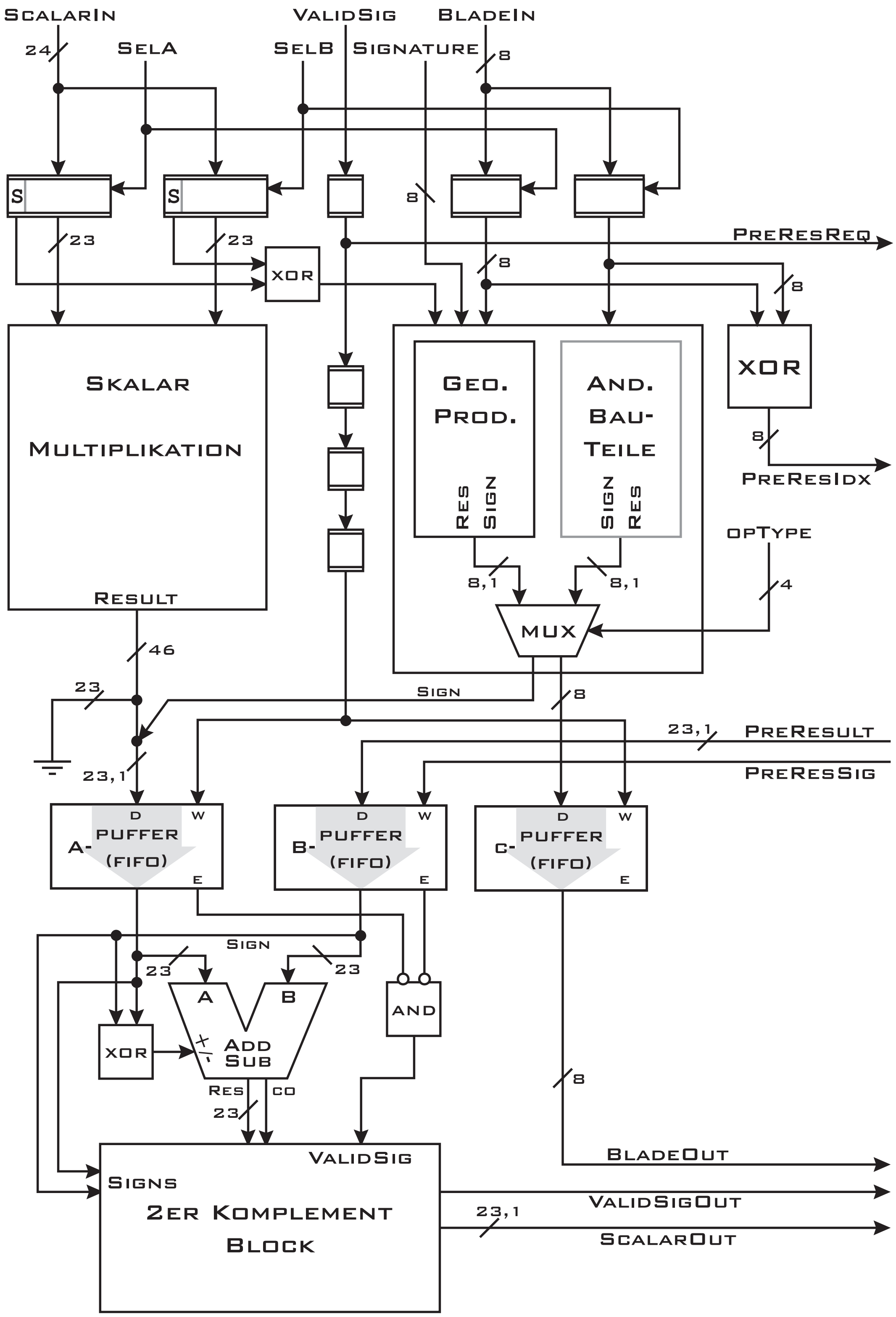

Abb. 4.2: Rechenwerk (Berechnungseinheit).

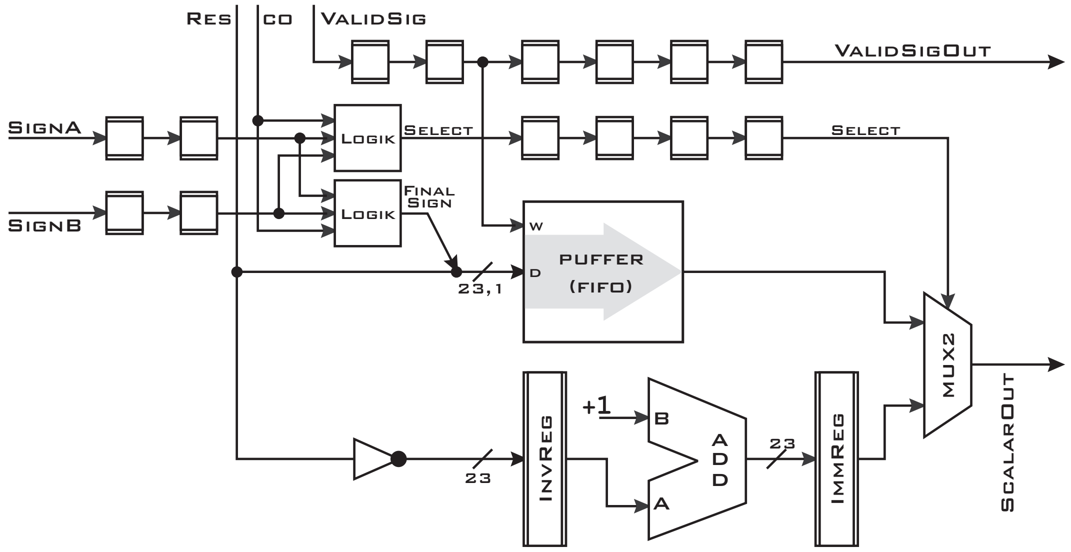

Abb. 4.3: Aufbau des `2er Komplement Block` des Rechenwerks. Die Konvertierung erfolgt nach dem Schema: Invertieren und Eins addieren.

Die Vorzeichen der Skalaranteile werden *XOR*-verknüpft, um zunächst das Sign der Multiplikation zu erhalten. Letzteres wird über weitere *XOR*-Operationen mit den jeweiligen Signs aller algebraischen Bauteile kombiniert, sobald diese vorliegen (nicht eingezeichnet). Ein *Multiplexer* wählt das **`Res`**/**`Sign`** - Ergebnis desjenigen Bauteils aus, das dem von der aktuellen *Instruktion* diktierten Operationstyp entspricht. Die Steuerung des **`MUX`** erfolgt über die vier Bit breite Leitung **`opType`** vom Kontrollwerk. Die Ergebnisse aller Elemente müssen ggf. durch ein oder mehrere *Register* zwischengespeichert werden (*Fifo*, siehe Abb. 4.1), damit sie die gleiche Anzahl von Taktzyklen besitzen, die für die skalare Multiplikation benötigt werden.

Die in der Grafik gezeigte 8 Bit *XOR*-Verknüpfung der Eingabeblades wurde zur Veranschaulichung aus dem Teil `Geo.Prod.` herausgelöst[3]. Der Ausgang **`PreResIdx`** des *XORs* dient als Index zur Anforderung von Teilergebnissen aus dem *SRAM*, die zu dem einstweiligen Ergebnis der aktuellen Operation aufaddiert werden müssen, um anschließend unter dem selben Index abgespeichert zu werden. Die entsprechende Summation wurde bereits als Teil 'Vereinigung' der Berechnungseinheit in Abbildung 4.1 schematisch dargestellt. Das RAM ist zum Designstart initialisiert, so daß alle Teilsummen anfangs Null sind. Das Signal **`PreResReq`** signalisiert der Adreßeinheit, daß eine Anforderung per **`PreResIdx`** vorliegt. Es stellt das um einen Takt verzögerte **`ValidSig`** dar. Das Eintreffen der gewünschten Daten via **`PreResult`**-Leitung wird durch **`PreResSig`** angezeigt.

Die vorzeichenlosen 23 Bit breiten Skalare werden multipliziert. Der verwendete Multiplizierer benötigt hierfür 4 Taktzyklen, was in der Abbildung durch die drei nebenstehenden *Flip-Flops* des **`ValidSig`** angedeutet wird. Da das *Design* zur Zeit nur mit ganzzahligen Werten rechnen kann, ist es notwendig, die obere Hälfte des 46 bittigen Ergebnisses zu verwerfen.

Das Produkt des Multiplizierers wird zusammen mit dem vorläufigen Sign in den (A-)Puffer geschrieben. Die Bladeinformation kommt in den (C-)Puffer. Als Schreibsignal an den **`W`**-Pin

[3] Aus Platzgründen ist ein *Multiplexer*, der das richtige Resultatblade aller algebraischen Bauteile durchschaltet, nicht eingezeichnet.

(für 'write') beider *Fifos* fungiert **ValidSig**, dessen Kette aus *FlipFlops* hier vorerst endet. Wenn das angeforderte Teilergebnis **PreResult** angelangt ist, kommt es in den (B-)Puffer. Da sich die Reihenfolge, in der die Teilsummen angefordert werden, nicht von der unterscheidet, in der sie ankommen, befinden sich in den *Fifos* A und B an gleichen Positionen zusammengehörige Paare, die addiert werden sollen. Die Puffer bezwecken, daß korrespondierende Daten zu verschiedenen Zeiten (Takten) ankommen dürfen. Die **E**-Pins (für 'empty') beider Puffer sind über ein AND in der Art verschaltet, daß dessen Ausgang auf *High* geht, sobald beide Puffer nicht leer sind. Dieses Signal dient als Eingabe für eine neue **ValidSig**-Kette, der Fortsetzung der alten. Es veranlaßt den A- und B-Puffer zum Löschen des ersten Datenpaares (nicht eingezeichnet), worauf deren Ausgänge die in den Puffern nachfolgenden Werte übernehmen. Im Addierer/Subtrahierer wird Skalar B von Skalar A abgezogen resp. addiert, falls genau ein Wert negativ ist bzw. beide das gleiche Vorzeichen haben. Die Steuerung wird durch ein *XOR-Gatter* realisiert.

Es wird geprüft, ob es im **ADD/SUB**-Element eine Differenzbildung gab und ob diese ein negatives Ergebnis entstehen ließ. In solchen Fällen hat der Addierer/Subtrahierer das **CO** (für 'carryout')[4] gelöscht und das zugehörige Resultat im 2er-Komplement ausgegeben. Dies muß rückgängig gemacht werden, da das *Design* eine hiervon abweichende Darstellung für negative Zahlen verwendet. Die Konvertierung findet im `2er Komplement Block` (Abbildung 4.3 auf Seite 29) statt, wo von jedem Resultat R pauschal das 2er-Komplement R' gebildet wird. Der ursprüngliche Wert R durchläuft parallel einen Puffer, um weiterhin zur Verfügung zu stehen. Je nachdem, ob das Ergebnis im 2er-Komplement ausgegeben wurde, legt ein *MUX* entweder R' oder R an **ScalarOut** an.
Das ermittelte endgültige Vorzeichen **FinalSign** hat ebenfalls den Puffer durchlaufen und stellt stets das höchstwertigste Bit 23 von **ScalarOut** dar.

4.1.3 Kontrolleinheit

Die Kontrolleinheit ist dem Rechenwerk vorausgeschaltet und interpretiert Eingaben seitens des PC. Dazu gehören das Bestimmen der *Design*-Modi (RAM schreiben, Operation, RAM lesen), das Setzen des Lese-/Schreibzählers, das Festlegen der Schreibbasisadresse, das globale Initiieren des Designstarts und das Setzen der Signatur sowie von *Flags* (Kontroll-Bits). Es ist zudem dafür zuständig, daß vom *SRAM* stammende Daten erkannt und entsprechend weitergeleitet werden. Die *Blades* der *Multivektoren*, Informationen über deren Art oder zu deren Weiterverarbeitung werden an die Berechnungseinheit gesandt. Die Weiterverarbeitung bestimmt den Typ der gewählten Operation, d.h. inneres, äußeres, geometrisches Produkt oder eine andere Verknüpfung, während die Art angibt, ob es sich um einen A- oder B-Operanden handelt. Zur Wahrnehmung aller genannten Funktionen ist das Kontrollwerk neben allen anderen Einheiten mit dem `PCI_SLAVE` und indirekt mit dem *SRAM*-Ausgang verbunden.

Das Kontrollwerk hat zusammenfassend die Aufgabe, die betreffenden Daten der nachstehend aufgelisteten Ereignisse zu verarbeiten und hieraus den Zustand des Systems abzuleiten, um die weiteren Vorgänge im *Design* maßgeblich zu steuern.

[4]Ein Übertrag an **CO** bei Additionen wird aus den gleichen Gründen wie bei der Multiplikation verworfen.

- PC-KOMMUNIKATION. Jeder Einsatz des *Designs* beginnt und endet mit einem Informationsaustausch zwischen *FPGA*-Board und der Software des PC's. Die Kontrolleinheit erhält über den *PCI-Bus* Befehle, die das Verhalten des *Designs* lenken.

- INSTRUKTIONSWECHSEL. Eine *Instruktion* bezeichnet die Verknüpfung zweier *Multivektoren*. Sie beginnt mit dem Auslesen dreier 32 Bit Instruktionsworte aus dem *SRAM*, die von dem Kontrollwerk empfangen werden. Die enthaltenen Daten über Adresse der nächsten *Instruktion*, Adressen der beiden Operandenmultivektoren sowie deren jeweilige Bladeanzahl werden an die Adreßeinheit übermittelt. Instruktionsspezifische *Flags* und der Operationstyp werden gesichert.
  Ähnlich wie bei *CISC*-Strukturen haben die *Instruktionen* keine fixe Länge, sondern variieren zwischen drei und 515 Worten, abhängig von den Operanden [38, 29].

- OPERANDEN-ANKUNFT. Das Eintreffen von Speicherwörtern aus dem *SRAM* wird durch mehrere Signale begleitet, die das korrespondierende Datum eindeutig, z.B. als B-*Blade*, typisieren.

Die Summe aller so eingehenden Informationen definiert zusammen mit dem momentanen Zustand den Folgezustand des *Designs*. Er wird zu Beginn eines jeden Taktes in die *Register* der Kontrolleinheit übernommen, aktualisiert auf diesem Weg den alten Zustand und bestimmt die *Pegel* der Steuersignale.

Das in Abbildung 4.4 dargestellte Kontrollwerk arbeitet mit vier Hauptregistern, die den Zustand des Systems wiedergeben bzw. speichern. Den meisten *Register*n ist eine Eingabelogik vorangestellt, die für jedes Bit individuell bestimmt, wann und welchen Wert es übernehmen soll. Die Logik eines Bits kann einfach ausfallen, hängt aber mitunter von bis zu neun Eingängen ab. Die Ausgabelogik des `ControlReg` soll andeuten, daß die Registerwerte nicht direkt als Ausgabe des Kontrollwerks dienen, sondern vorher noch verknüpft werden.

Das `DsgnDataReg` dient dazu, die Daten aus dem `DPSRAM` (*SRAM*-Aufsatz, siehe Abb. 4.4) unmittelbar zu sichern, d.h., um Timingprobleme bei der Datenübernahme zu vermeiden. Diese wären bei einer zwischengeschalteten, zusätzlich verzögernden Logik nicht auszuschließen, da der interne Aufbau des in *CHDL* integrierten **`DPSRAM`**s unbekannt ist. Zudem erbt letzteres die Asynchronität des *SRAMs*. Das `DsgnDataReg` ist 36 Bit breit von denen z.Zt. nur 32 Bit verwendet werden. Über Leitung [2] (kurz: *Ltg*[2]) gehen die im vorigen Absatz genannten Informationen der Instruktionsworte an die Adreßeinheit. Nummern sind neun Bit breit, weil die Anzahl der *Blades* bis zu 256 betragen kann. Adressen sind 21 Bit breit. Die Berechnungseinheit erhält den Skalaranteil (24 Bit) und den achtbittigen Bladeanteil der Multivektorelemente via *Ltg*[9].

Das **`GlobalReg`** speichert die Signatur, die sogn. **`WaitAlwaysNode`**, sowie die Betriebszustände **`RdThru`**, **`WrThru`** und **`opEnG`**. Letzterer steht dafür, alle Berechnungen global zu erlauben. Wird dieser Zustand auf hohen *Level* gezogen, versucht das *Design* zu starten, d.h. beginnt die ersten drei Werte ab Adresse Null des *SRAMs* zu lesen und interpretiert sie als Instruktionsworte. **`RdThru`** und **`WrThru`** stehen für das Auslesen, bzw. das Beschreiben des *SRAMs*. Sie haben die höchste Priorität im *Design*, überstimmen sogar **`opEnG`** und setzen die Adreßeinheit (und damit auch das ganze System) per *Ltg*[6] in den entsprechenden Lese-

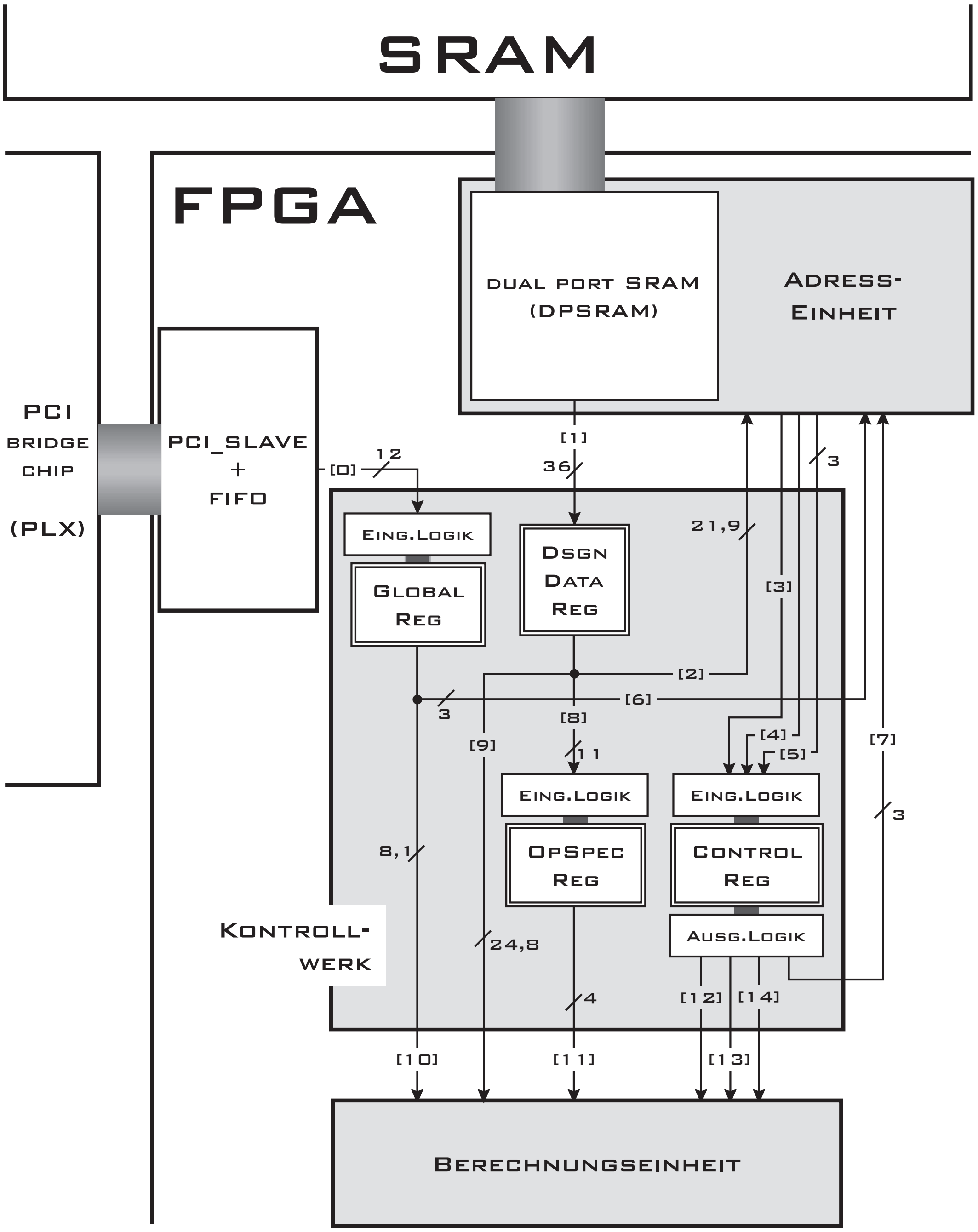

Abb. 4.4: Kontrollwerk mit dessen Hauptbestandteilen und wichtigsten Leitungen. Die umgebenden Bauteile sind lediglich symbolisch dargestellt.

oder Schreib-Modus. Das **`GlobalReg`** wird per *Ltg*[0] ausschließlich über den Benutzer gesetzt. Die Eingabelogik dient zur Detektion solcher Eingaben. Signatur (8 Bit) und **`opEnG`** gehen via *Ltg*[10] an die Berechnungseinheit.

Das **`OpSpecReg`** sichert Daten, die die aktuelle Operation betreffen. Das sind *Flags* und Operationstyp. Jene geben u.a. an, ob die Operanden bei der Berechnung vertauscht werden sollen ($\mathcal{B} \otimes \mathcal{A}$ statt $\mathcal{A} \otimes \mathcal{B}$) oder ob ein Operand das zuvor berechnete Ergebnis ist. Dies ist im Zusammenhang mit Datenabhängigkeiten wichtig (Abschnitt 5.2). Das *Register* erhält seine

Daten zu unterschiedlichen Zeitpunkten, welche durch die Eingabelogik erkannt werden. Der von der Berechnungseinheit benötigte Operationstyp gelangt über *Ltg*[11] zum Ziel.

Im **ControlReg** wird verwaltet, welche Art von Daten sich gerade im **DsgnDataReg** befinden, mit anderen Worten was vor einigen Takten als Leseadresse am *SRAM* anlag. Die Adreßeinheit liefert dies durch die Leitungen [3,4,5]. Ist *Ltg*[3] gesetzt, befand sich die Adreßeinheit im Pausezustand, so daß das aktuell anliegende Datum kein gültiges *Blade* für die Berechnungseinheit darstellt. Ist sie nicht gesetzt, entscheidet *Ltg*[4] über die Herkunft des Operanden, d.h. von Multivektor A oder B. Wenn die Adressen von Instruktionswörtern anlagen, dann signalisiert die Adreßeinheit per 3 Bit-*Ltg*[5] **FirstInstSig**, **SecondInstSig** oder **ThirdInstSig**, je nachdem, ob der erste, zweite oder dritte Teil der *Instruktion* gerade am **DsgnDataReg** anliegt. Alle Eingangssignale werden nach weiteren Verknüpfungen in der vorgeschalteten Logik im **ControlReg** gespeichert. Leitungen [12,13,14] finden ihre Entsprechung in den Leitungen [3,4]: *Ltg*[12] zeigt der Berechnungseinheit an, ob die via *Ltg*[9] anliegenden Daten gültig sind und ob sich die Adreßeinheit nicht im Pausezustand befand. Ist dies der Fall, dann ist in Abhängigkeit von *Ltg*[4] entweder *Ltg*[13] (A-Operand) oder *Ltg*[14] (B-Operand) gesetzt. *Ltg*[7] ist primär eine Rückführung von *Ltg*[5], dessen Signale in der Adreßeinheit durch Laden von *Registern* oder Zählern, die Ausfürung einer neuen *Instruktion* vorbereiten.

Es sei an dieser Stelle darauf hingewiesen, daß alle am Ausgang des *SRAMs* anliegenden Daten (Baldes, Instruktionsworte und Teilergebnisse **PreResult**) die *Register* des *Designs* durchlaufen: Für jedes ausgelesene Datum wird im Rechenwerk ein geometrisches Produkt gebildet, usw. Der resultierende Wert wird am Ende nur dann nicht verworfen, falls das betreffende Datum *gültig* war (A-Operand ungleich Null).

PCI_SLAVE - Element

Zur Verbindung mit dem *PCI-Bus* via dem *PCI* bridge chip (*PLX* 9080) gemäß Abbildung 4.5 wird seitens des *FPGAs* das `PCI_Slave` Bauteil benötigt [19, 20, 29]. Es dient zur Kommunikation mit dem *PLX* (siehe Anfang Sektion 4.1 auf Seite 25) und stellt zusätzliche Funktionalitäten bereit. Dazu gehöhren die Definition von bis zu vier Adreßbereichen im PC-RAM,

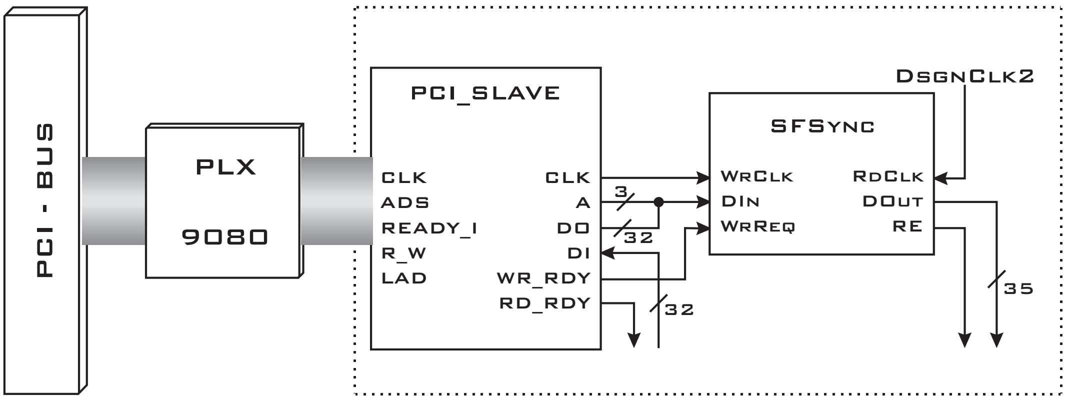

Abb. 4.5: Datenpfad vom *PCI-Bus* zum *Design*. Das gepunktet umrandete Schaltbild entspricht dem Bauteil '`PCI_SLAVE` + FIFO' in Abb. 4.4 auf Seite 32.

die explizite Spezifikation von Wartezyklen, die Verfügbarkeit von Kontrollsignalen (**WR_RDY** und **RD_RDY**) und spezielle Modi wie pipelined burst[5] [29, 38].
Das Auslesen und Beschreiben des *PCI*-Busses entspricht unter C++ dem Zugriff über einen Zeiger auf ein 64, 128 oder 256 KByte großes Integer-Feld [40]. Es wird durch den bei der Erstellung des **PCI_Slave** Elements definierten Adreßbereich festgelegt. Der **PCI_Slave** wird mit der local bus clock betrieben, mit der auch der *PLX* chip läuft. Es können Taktfrequenzen von 10 MHz bis 40 MHz eingestellt werden. Das clock-Signal kann auch im *Design* unter dem Namen **PCI.CLK** weiterverwendet werden, ist jedoch zu allen anderen Taktsignalen asynchron.
Die **LAD**-Leitung fungiert als *Bus* zwischen *PLX* und **PCI_Slave**. Sie ist 32 Bit breit, bidirektional und zeitlich gemultiplext (Adresse und korrespondierende Daten werden abwechselnd sukzessive übertragen). Der **ADS**-PinIn (Eingang) ist genau dann *Low*, wenn eine Adresse übertragen wird. PinIn **R_W** zeigt den aktuellen Modus an (*Low*: Lesen, *High*: Schreiben) und ändert sich synchron mit **ADS**. *Tristate*-Ausgang **READY_I** signalisiert Wartezyklen durch hohen *Level*, geht auf *Low*, falls sich Daten auf dem *Bus* (**LAD**) befinden, und ist hochohmig, falls keine eigene Aktivität am *Bus* herrscht.
Im **PCI_Slave** Element werden Adressen und Daten parallelisiert. Der Zugriff erfolgt über die Pins **A**, **DO** und **DI**. Signal **WR_RDY** zeigt das Vorhandensein eines neuen Wortes an, das noch im gleichem Takt verarbeitet werden muß, damit es nicht verloren geht. Im Gegensatz dazu berechtigt **RD_RDY** zum Schreiben eines Datums auf den *Bus*.
Das **SFSync**-Bauteil (Slow-Fast-Synchronizer) aus aus Abbildung 4.5 übersetzt die Daten auf die größere Frequenz des *Designs*, um Timingfehler zu vermeiden. Das Element ist wie ein *Fifo* organisiert. Voraussetzung ist, daß die Frequenz, mit der die Daten empfangen werden, unter der der Ausgabeseite liegt. Siehe hierzu auch Abschnitt 6.1.1 auf Seite 57.

Instruktions- und Datenworte des Prozessors

Eine *Instruktion* besteht aus drei Instruktionswörtern und zwei *Multivektoren* MVA und MVB. Die erste *Instruktion* steht an Adresse Null. Die folgenden ergeben sich aus einer einfach verketteten linearen Liste (Abb. 4.7), dessen Zeiger **NxtCntlWrdAddres** sich im jeweils ersten Instruktionswort IW0 befindet.

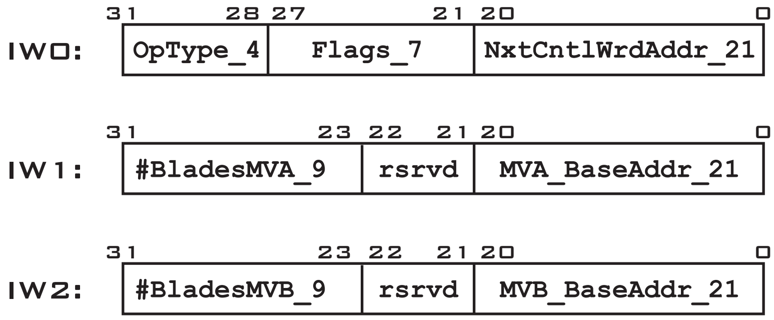

Abb. 4.6: Instruktionswörter. Sie enthalten neben aktuellen Befehlsinformationen die Adresse der jeweils nächsten *Instruktion*. Die Endung '_T' gibt die Bit-Länge T der Einträge an, '#' steht für 'Anzahl'.

[5] Zum Zeitpunkt dieser Arbeit nicht verfügbar

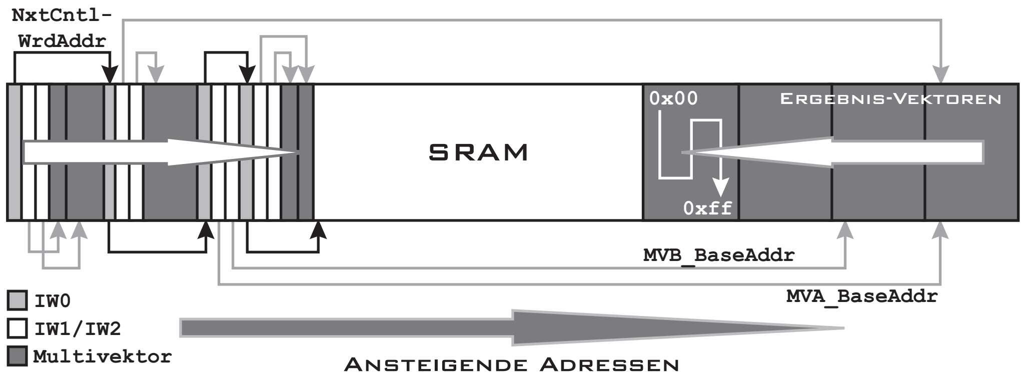

Abb. 4.7: Speichermanagement. Zwei der vier *Instruktionen* beziehen sich auf Ergebnisse voriger Berechnungen. Es ist $\texttt{VecDim} = 8$.

Die Felder **`OpType`** und **`Flags`** bestimmen den Operationstyp und definieren Kontrollbits. Diese Informationen finden sich später im **`OpSpecReg`** wieder (Seite 32). Die obersten neun Bit von `IW1` und `IW2` enthalten die Anzahl `#BladesMVX` der *Blades* der Operanden. Die Angabe absoluter Positionen (`MVX_BaseAddr`) über die Bits 0 bis 20 erlaubt, daß Operanden an beliebiger Stelle im Speicher stehen können. Auf diesem Weg ist es möglich, daß Ergebnismultivektoren die Eingabe(n) nachfolgender *Instruktionen* sind. Die Speicherung der resultierenden *Multivektoren* erfolgt in Richtung abnehmender Adressen und beginnt am Ende des Speichers, an der Basisadresse `0x07ff00`[6]. Die *Multivektoren* belegen dort stets Blöcke fester Größe von $2^{\texttt{VecDim}}$ Wörtern. Die enthaltenen *Blades* werden als ihre eigenen Indices verwendet, so daß eine sortierte Reihenfolge entsteht: Das *Blade* $e_7e_6e_3e_2e_1$ korrespondiert beispielsweise mit der Darstellung `01100111b`, dem hexadezimalen Index `0x67`. Es würde an Adresse `0x07fb67` geschrieben, falls als Basisadresse gerade `0x07fb00` verwendet wird. Die fixe Größe der Speicherblöcke impliziert, daß bei *Instruktionen*, deren Operandenmultivektoren wenige *Blades* besitzen, im Ergebnisvektor vermehrt Null-*Blades* zu finden sind. Sie stellen 'leere' Speicherstellen dar, auf die nicht zugegriffen wurde, weil sich die betreffenden, mit ihnen assoziierten geometrischen Anteile bei den Berechnungen nicht ergeben haben.

Die Datenwörter, die die *Blades* eines *Multivektors* repräsentieren, haben einen zweigeteilten Aufbau, den Abbildung 4.8 zeigt. Das höchstwertigste Bit des Skalaranteils gibt das Vorzeichen (Sign) des ganzzahligen Wertes an, der durch die restlichen Bits dargestellt wird. Weil sich Zahl und Komplement nur im Sign unterscheiden, existiert eine negative Null -0, die stets als Null behandelt wird.
Der zweite Part - der geometrische Anteil - speichert das *Basisblade*. Jedes Bit steht für ein festes Element der orthonormalen Basis $E = \{e_1, e_2, \dots, e_n\}$, $n \in \mathbb{N}$ des n-dimensionalen Vektorraums $\mathbb{R}^n$. Die im vorherigen Absatz implizit verwendete Zuordung, bei der Bit Null mit e_1, Bit Eins mit e_2 usw. korrespondiert, gilt durchgängig in dieser Arbeit. Bei der Benutzung des Designs können gleichfalls andere Belegungen zugrunde gelegt werden, da dies nur auf Umbenennungen von Variablennamen hinausläuft, z.B. $e_2e_3e_6e_7e_8$ für `01100111b`. Die Angabe des geometrischen Anteils ermöglicht die Identifikation der *Blades* beim Laden der *Instruktionen* von dem PC. Die Reihenfolge der *Blades* darf zufällig sein. Die Alternative wäre eine Übertragung der *Multivektoren* in Blöcken zu $2^{\text{VecDim}} \cdot 32$ Bit, so daß eine Iden-

[6] Aus technischen Gründen tatsächlich `0x07fe00`.

tifikation anhand der Blockposition durchgeführt werden könnte. Voraussetzung wäre eine feste Reihenfolge der *Blades* innerhalb der Blöcke. Es gäbe viele nicht belegte Einträge, die zusätzlich als Null-*Blades* verarbeitet werden müßten. Ein weiterer Nachteil wäre das erheblich gesteigerte benötigte Datenvolumen.
Da die Speicherung der Ergebnisvektoren von einem solchen Blockformat Gebrauch macht, könnte der geometrische Anteil bei der Bladerepräsentation eingespart werden, um etwa die Rechengenauigkeit zu verbessern. Es wäre andererseits ein Extra an Logik erforderlich.

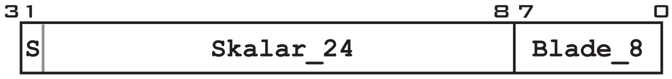

Abb. 4.8: Bladerepräsentation. Im 24 Bit breiten Skalaranteil ist das Vorzeichen an oberster Bitposition enthalten.

4.1.4 Adreßeinheit

Die Adreßeinheit (Abb. 4.11, Seite 39) koordiniert alle Zugriffe des *Designs* auf das **DPSRAM**. In ihr befindet sich das Bauteil **PRLink** (für 'PC-RAM-Link'), mit dem das RAM vom PC aus beschrieben und ausgelesen werden kann. Neben dem Zugang für PRLink zum RAM werden vom Adreßwerk auch die Lese- und Schreibanforderungen des Berechnungswerkes verwaltet. Erstere erbitten Teilsummen, während letztere Forderungen das Speichern des Ergebnisses veranlassen. Die dritte Aufgabe besteht in dem geeigneten Auslesen der *Blades* der *Multivektoren*.

PC-RAM-Link (PRLink)

Die Komponente **PRLink** ist Instanz der Klasse **RamIO**, deren Aufbau in Abschnitt 6.1.4 auf Seite 60 ausführlich erläutert wird.
Es dient als Direktverbindung zwischen Außenwelt (PC) und *SRAM*. Das Element besteht aus einem ladbaren Adreßzähler für Lese- und Schreiboperationen und einem *Fifo* zur Zwischenspeicherung von Daten aus dem *SRAM*. Die Steuerung übernimmt das Kontrollwerk über die Leitungen **RdMode**, **WrMode**, **BusWrReq**, **BusRdRdy** und **SetCntrPulse**. Sie bedeuten „Lesemodus"(auch: **RdThru**), „Schreibmodus"(auch: **WrThru**), „RAM-Schreibanforderung seitens des PC(I)", „*PCI-Bus* kann ein Datum aufnehmen"sowie „Signal zum Setzen des Zählers".

Dual port SRAM (DPSRAM)

Das **DPSRAM** (Abb. 4.9, Seite 37) ist Bestandteil der *CHDL*-Klassenbibliothek [19]. Der innere Aufbau ist nicht bekannt. Die Bezeichnung 'dual port' wird verwendet, weil es im Gegensatz zum gewöhnlichen *SRAM* einen weiteren Adreß- bzw. Datenport (für 'Anschluß') gibt, der parallel verwendet werden kann. Diese Funktionalität wird simuliert, indem das asynchrone *SRAM* über eine im Vergleich zum *Design* doppelt schnelle Taktfrequenz angesprochen wird.[7] Der einzelne *Port* des *SRAMs* wird per zeitlichem Multiplexing in der *High*-Phase des

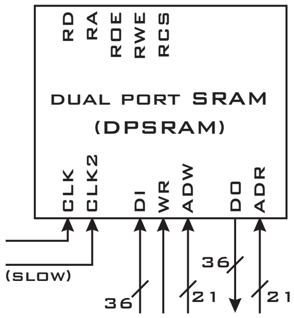

Abb. 4.9: **DPSRAM** aus *CHDL*. Dank doppeltem internen Takt kann gleichzeitig gelesen und geschrieben werden.

Designtakts zum Schreiben und in der *Low*-Phase zum Lesen verwendet. Die Ansschlüsse **RD**, **RA**, **ROE**, **RWE** und **RCS** zur Verbindung des **DPSRAM**s mit dem *SRAM* entsprechen einem bidirektionalen 36 Bit Datenbus, der 21 Bit breiten Adresse, dem 'OutputEnable'-Pin, dem 'WriteEnable'-Pin und dem 'ChipSelect'-Pin. Das Zeitdiagramm in Abbildung 4.10 zeigt exemplarisch die Funktionsweise des **DPSRAM**s.
Die an den vom *Design* anzusteuernden Adreßanschlüssen **ADW** (für 'AddresWrite') und **ADR** (für 'AddresRead') anliegenden Werte werden parallel verarbeitet. Die Daten an **DO** (für 'DataOut') haben eine Verzögerung von zwei Takten auf **ADR**, während es nur einen Takt dauert, bis der an **DI** (für 'DataIn') anliegende Wert an Adresse **ADW** geschrieben ist (→ Zeitdiagramm). Damit ein Datum gespeichert wird, muß PinIn **WR** einen hohen *Pegel* haben. Sind **ADW** und **ADR** identisch, ist der ausgelesene Wert undefiniert.

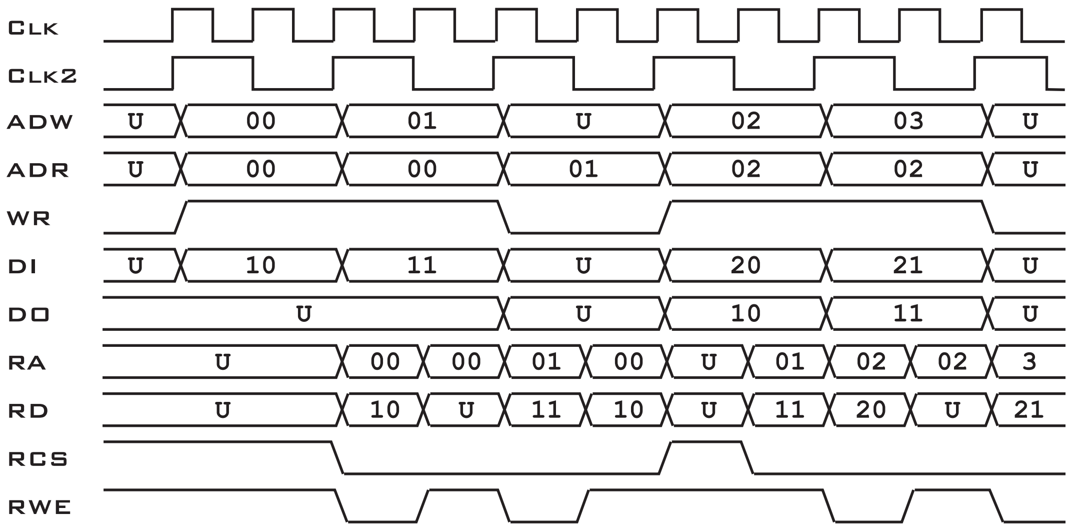

Abb. 4.10: **DPSRAM** Zeitdiagramm. Der Takt an **CLK2** darf nur halb so schnell sein, wie der an **CLK**.

Interna des Adreßwerkes

Die meisten der in diesem Unterabschnitt erwähnten Steuersignale sind in Abbildung 4.11 aus Komplexitätsgründen nicht eingezeichnet. Dies betrifft im Inneren des Adreßwerkes ver-

[7] Da das *SRAM* z.Zt. nur mit max. 40 MHz betrieben werden kann, begrenzt der Einsatz des **DPSRAM**s die Designfrequenz auf max. 20 MHz.

laufende sowie extern zugeführte Leitungen.
Das Signal **WrThru** der Kontrolleinheit bestimmt, daß **PRLink** zur Schreibadresse **ADW** des **DPSRAM**s durchgeschaltet wird. Ist **WrThru** *Low*, läßt der Multipliexer **MUX2** die Adresse der Berechnungseinheit durch. Sie setzt sich aus dem acht Bit Bladeindex **BladeIn** und der aktuellen Schreibbasisadresse, die vom Dekrementer **WrBaseDec** ausgegeben wird, zusammen.[8]
Im **WrThru**-Modus muß das **DiReg** mit den **DataToRam**-Pins des **PRLink** verbunden werden. Im normalen Modus schaltet der **MUX2** das Ergebnisblade (**ScalarIn** und **BladeIn**) aus dem Berechnungswerk durch. Das **WrReg** wird von der Berechnungseinheit und von **PRLink** gesteuert und signalisiert, ob das aktuell anliegende Datum gespeichert werden soll. Die Steuerung der Leseadresse **ADR** übernimmt der vor **RdAddrReg** geschaltete **MUX4**.
Im normalen Betrieb wird entweder die Adresse des A-, oder die des B-Operanden weitergeleitet. Diese befinden sich in den ladbaren Adreßzählern **AddrCntA** und **AddrCntB**. Falls es eine Teilsummenanforderung des Berechnungswerkes über die Leitungen **PreResReq** und **PreResIdx** als acht Bit Bladeindex gibt, wird letzterer zusammen mit dem Inhalt des **WrBaseDec** als obere 13 Bit durchgeschaltet. Der Rest des Adreßwerks pausiert währendessen.
Im **RdThru**-Modus bekommt, analog zu **WrThru**, das **PRLink**-Element exklusiven Zugriff auf das **RdAddrReg**. Die in Abbildung 4.4 (Kontrollwerk) auf Seite 32 eingezeichnete drei Bit breite *Ltg*[7] initiiert drei Phasen nach Beginn einer neuen *Instruktion*: **Phase_I**, **Phase_A** und **Phase_B**. Sie signalisieren jeweils, daß sich ein Instruktionswort im **DsgnDataReg** des Kontrollwerks befindet. Dessen Befehlsinformationen liegen auf bestimmten Leitungen an und sollen in entsprechende Zielregister der Adreßeinheit geladen werden. **Phase_I** korrespondiert mit Instruktionswort `IW0` und zieht das **CE** von **CntlWrdAddrReg** auf einen hohen *Pegel*. Dadurch wird die Adresse des nächsten Instruktionstripels **NxtCntlWrdAddr** gespeichert. In den Phasen A und B müssen alle Daten, die die *Multivektoren* A bzw. B betreffen, geladen werden. Leitung **Words** gibt jeweils die Anzahl der *Blades* an. Über **OpBaseAddr** wird die Basisadresse des zugehörigen *Multivektors* übermittelt. **AddrCntA** und **AddrCntB** repräsentieren einen inneren bzw. einen äußeren Schleifenzähler. Die einzelnen *Blades* des *Multivektors* B werden sukzessive mit jeweils allen A-*Blades* verknüpft. Da der A--Zähler nach jedem Schleifendurchlauf neu geladen werden muß, werden die Informationen von `IW1` in den *Registern* **WrdsReg** und **BaseAddrReg** gesichert. In **Phase_B** werden alle Daten direkt in Dekrementer **DecB** bzw. in **AddrCntB** geladen.
Während die Adreßzähler nur hochzuzählen brauchen, dienen die neun Bit Dekrementer **DecA**, resp. **DecB** zum Anzeigen des jeweiligen Schleifenendes. Das Signal dazu heißt in Abbildung 4.11 **UF** (für 'UnderFlow') und steuert einen grossen Teil des Adreßwerks. Ein Unterlauf von **DecA** bewirkt, daß er selbst und auch **AddrCntA** neu geladen wird sowie daß **DecB** und **AddrCntB** einen weiteren Zählschritt machen. Wenn **DecB UF** signalisiert, ist das Ende der aktuellen *Instruktion* erreicht. Das Adreßwerk wird in den Pausezustand versetzt. Für den Ergebnismultivektor der nachfolgenden *Instruktion* muß **WrBaseDec**[9] erniedrigt werden. Der **MUX2**, der den Dateninput **D** des A-Zählers auswählt, schaltet zum Laden die Adresse des nächsten Instruktionstripels (**CntlWrdAddrReg**) durch. Durch eine nicht eingezeichnete Logik macht **AddrCntA** zwei Zählschritte, so daß die drei aufeinanderfol-

[8] Ist der aktuelle Inhalt des **WrBaseDec** beispielsweise `0x07fa`, dann befinden sich die Ergebnisblades, auf denen momentan gearbeitet wird, im Adreßbereich `0x07fa00` bis `0x07faff`.

[9] Die Existenz von noch nicht geschriebenen *Blades* in der *Pipe* verbietet das sofortige Herunterzählen. Aus diesem Grund ist die **CE**-Leitung des **WrBaseDec** in der Abbildung gepunktet dargestellt.

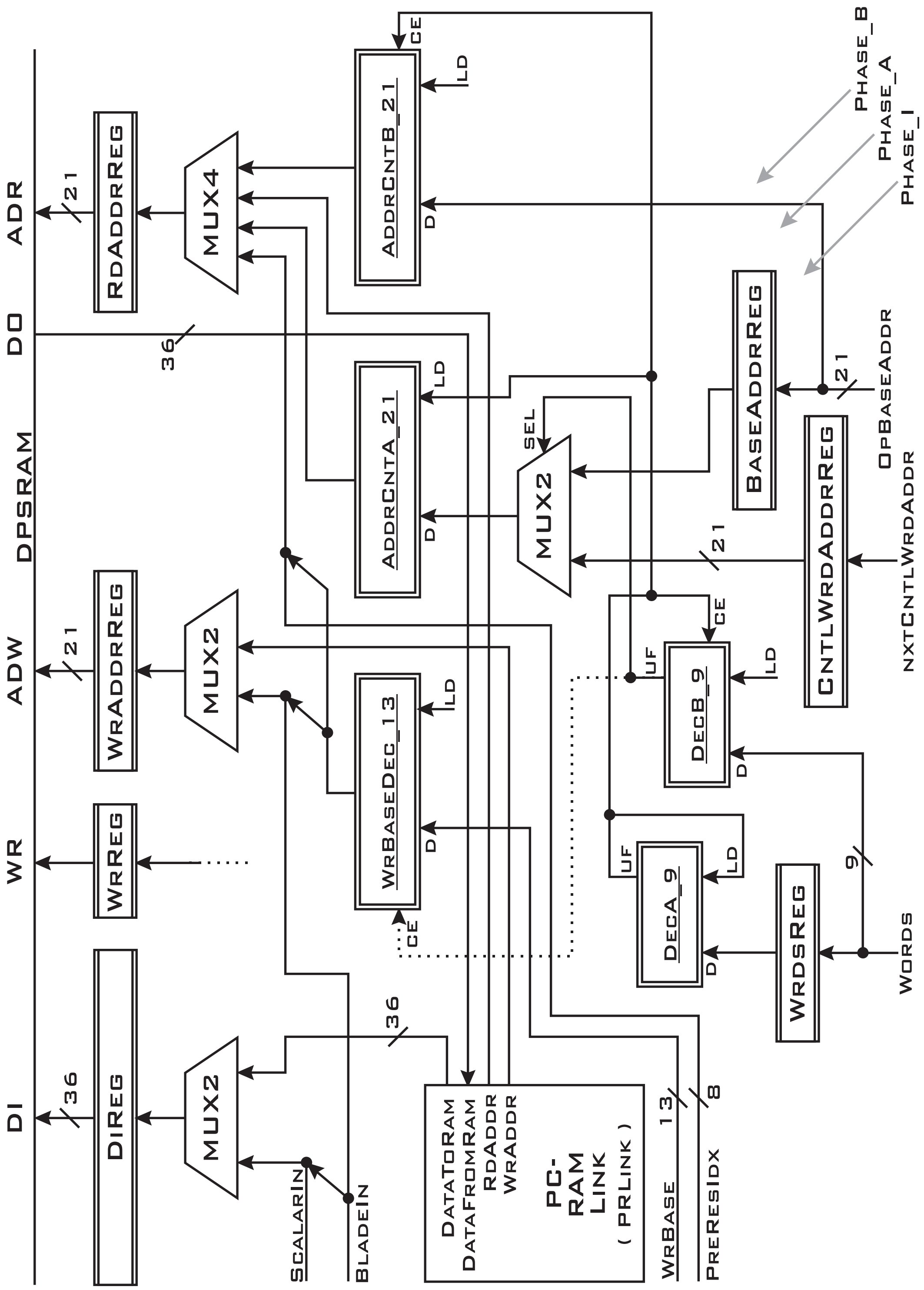

Abb. 4.11: Adreßeinheit. Ein Großteil der von **Phase_{I,A,B}** beeinflußten Steuerungslogik mit deren Signalen ist nicht dargestellt. **D**,**CE**,**LD** und **SEL** steht für Data, *Clock*Enable, Load und Select.

genden Instruktionsworte `IW0`, `IW1` und `IW2` ausgelesen werden. Das Kontrollwerk erhält via *Ltg*[5] begleitend die Signale **`FirstInstSig`**, **`SecondInstSig`** und **`ThirdInstSig`**, die über *Ltg*[7] einen neuen Zyklus auslösen.

4.2 Ablauf eines FPGA-Einsatzes

In den folgenden Textpassagen soll der Ablauf eines normalen *FPGA*-Einsatzes geschildert werden. Der Vorgang gliedert sich in die drei Hauptteile

- *Instruktionen* laden und *Design* vorbereiten,
- Datenverarbeitung und
- Ergebnisse auslesen.

Zuvor wird kurz beschrieben, wie ein Benutzer mit dem *FPGA* interagieren kann.

4.2.1 Kommunikation mit dem PC

Nachdem Board und *FPGA* initialisiert bzw. konfiguriert wurden, liefert eine der Treiberfunktionen den Zugriffszeiger `m_pBus` vom Typ `unsigned int` zurück(→ Abschnitt 4.1.3, Seite 34). Die Anweisung

`m_pBus[2] = 0x01f`

wird intern zur Ansteuerung des *PCI*-Busses umgesetzt. Zuerst wird der Wert Zwei als Adresse auf den *PCI-Bus* gelegt. Ihm folgt der Wert `0x01f` als Datum. Obige Abbildung

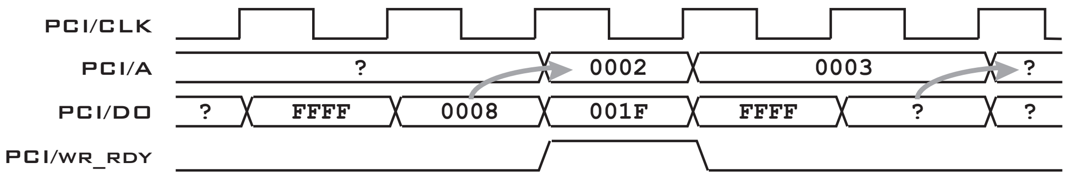

Abb. 4.12: Zeitdiagramm einiger **`PCI_Slave`**-Pins

4.12 veranschaulicht das Beispiel. Über **`PCI.DO`** werden alle Daten übertragen - jeweils ein Pufferwort `0x0ffff`, ein Adreßwort (um zwei Bits nach links verschoben) und das Datenwort. Der *PLX* inkrementiert **`PCI.A`** automatisch, da im Falle einer Burstübertragung nur eine Anfangsadresse übermittelt werden würde. Das *Design* erkennt gültige Daten am **`wr_rdy`**-Signal und identifiziert sie anhand der Adresse. Folgender Ausschnitt aus einem fiktiven *Design* zeigt, wie die Daten in ein *Register* übernommen werden könnten:

```
DRegister SaveReg( 32, "SaveReg" );
SaveReg.D = PCI.DO;
SaveReg.C = PCI.CLK;
SaveReg.CE = PCI.wr_rdy & ( PCI.A == 0x02 );
```

In Zeile 0 wird das 32 Bit breite *Register* **SaveReg** erstellt. Der Dateneingang **D** wird mit dem Ausgang **PCI.DO** des **PCI_Slave**-Elements verbunden. Damit jeder Wert sicher übernommen werden kann, wird **SaveReg** mit der *Clock* des *PCI*-Interfaces betaktet. In Zeile 3 wird das **CE** (für 'ClockEnable') gesetzt. Es bestimmt, daß anliegende Daten nur dann im *Register* gespeichert werden sollen, wenn **wr_rdy** *High* ist und die Adresse gleich Zwei ist.

Der Austausch in umgekehrter Richtung wird seitens des PCs durch sog. *Polling* realisiert. Der *Bus* wird über eine Schleife permanent abgefragt:

```
0   unsigned int uiValue;
    ...
1   do {
2        uiValue = m_pBus[ 0 ];
3   while { uiValue != 2 };
```

Der **PCI_Slave** reagiert auf Zeile 2, indem das Signal **PCI.rd_rdy** auf einen hohen *Level* gezogen wird. **rd_rdy** zeigt dem *Design* an, daß der *Bus* ein Datum übernehmen kann, das `uiValue` zugewiesen wird.

4.2.2 Vorbereitungsphase

Zu dieser Phase gehört alles, was bis zum Signal für den Start der Berechnungen passiert. Als erstes wird die Schreibbasisadresse per '`m_pBus[1] = 0x007ff`' gesetzt. Der Wert wird im **WrBaseDec** Dekrementer des Adreßwerks gespeichert. Um die gesamten *Instruktionen* in das *SRAM* zu transferieren, muß zunächst über die Anweisung '`m_pBus[2] = 0`' die Schreibstartadresse im Zähler des **PRLink**-Elements auf Null gesetzt werden. Anschließend wird durch einen weiteren Befehl der **WrThru**-Modus eingestellt. Alle folgenden Schreibzugriffe mit dem Index Null (`m_pBus[0] = X`) bewirken das Abspeichern der *Instruktion* `X` und das Inkrementieren der *SRAM*-Zieladresse. Wenn alle Daten übertragen sind, wird die Schreibstartadresse erneut gesetzt. Dafür wird **WrThru** zwischenzeitlich kurz gelöscht[10]:

```
m_pBus[3] = 0000b;      // WrThru loeschen
m_pBus[2] = newAddres;
m_pBus[3] = 0100b;      // WrThru setzen
```

Die neue Adresse ist so gewählt, daß im folgenden Speicherbereich alle erwarteten Resultate hineinpassen. Dieser Restspeicher muß dann mit Nullen gefüllt werden, um zu verhindern, daß bei erstmaligen Lesezugriffen während des Betriebs ein Wert ungleich Null als aufzuaddierendes Teilergebnis verwendet wird. Über '`m_pBus[3] = 0001b`' wird **opEnG** (für 'operateEnableGlobal') gesetzt, der **WrThru**-Modus aufgehoben und der Designstart initiiert.

4.2.3 Operationsmodus

In dieser Betriebsphase werden die Berechnungen auf den *Multivektoren* im *SRAM* ausgeführt.

[10] Die nicht in C++ enthaltene binäre Darstellung $x_3x_2x_1x_0$b mit $x_i \in \{0,1\}$, $0 \leq i \leq 3$ entspreche im Folgenden der Zahl $x_3 \cdot 2^3 + x_2 \cdot 2^2 + x_1 \cdot 2 + x_0$.

Der Zustand des Adreßwerks ist so initialisiert, daß die kürzliche Beendigung einer *Instruktion* vorgetäuscht wird. Wenn **opEnG** auf *High* geht, versucht das *Design* den logischen Folgezustand zu erreichen und startet die „nächste“ *Instruktion*, die tatsächlich die erste ist. Der im folgenden geschilderte Ablauf ist exemplarisch für die Abarbeitung einer Instruktion.
Gemäß Abbildung 4.11 auf Seite 39 wird der anfängliche Wert Null des **CntlWrdAddrReg** in den Zähler **AddrCntA** geladen. Dessen Inhalt wird im normalen Betrieb immer dann vom **RdAddrReg** übernommen, falls keine Teilergebnisanforderung via **PreResReq** vorliegt. Der Vorgang bewirkt, daß Instruktionswort `IW0` ausgelesen wird. Eine Kette von drei *FlipFlops* steuert weiterhin **AddrCntA**, der auf diese Weise zweimal hochzählt und die Adressen Eins und Zwei an das *SRAM* anlegt. Das Eintreffen von `IW0`, `IW1` und `IW2` wird im Kontrollwerk durch die Signale **FirstInstSig**, **SecondInstSig** und **ThirdInstSig** registriert. Das **opSpecReg** der Kontrolleinheit speichert die spezifischen *Flags*. Die weiteren entnommenen Informationen werden über die Leitungen **NxtCntlWrdAddr**, **Words** und **OpBase-Addr** zum Adreßwerk zurückgesandt. **Phase_I**, **Phase_A** und **Phase_B** bestimmen, wie in Abschnitt 4.1.4 auf Seite 38 beschrieben, welche Register welche Daten erhalten. Nach letzterer Phase wird die Adresse des ersten B-*Blades* ($MVB[0]$) an das *SRAM* angelegt. Für die folgenden Takte wird der Inhalt des A-Zählers zum **RdAddrReg** durchgeleitet. Währenddessen empfängt die Kontrolleinheit die ankommenden Operanden und identifiziert den ersten anhand der nicht gesetzten, sog. **ASig**-Leitung ($Ltg[4]$ in Abbildung 4.4 auf Seite 32) als B-*Blade*. Ist beispielsweise das **SWAP**-*Flag* im **opSpecReg** gesetzt, das eine Vertauschung der B- mit den A-Operanden bewerkstelligen soll, wird dem Rechenwerk das Vorhandensein eines A-*Blades* per **SelA**-Leitung signalisiert. Die drauffolgenden Elemente des Multvektors A ($MVA[0]$, $MVA[1]$, $MVA[2]$, $\ldots$), werden zu B-Operanden erklärt. Dessen vorderstes *Blade* ermöglicht die erste Bildung eines neuen Berechnungspaares ($MVB[0] \otimes MVA[0]$), so daß **ValidSig** (auch: $Ltg[12]$) auf *High* geht. In Konsequenz wird über **PreResReq** und **PreResIdx** das entsprechende Teilergebnis angefordert, das noch Null ist. Bis zu diesem Zeitpunkt wurden ein B- und fünf A-Operanden ausgelesen, die jetzt fünf korrespondierende Teilresultate anfordern. Die Adreßeinheit geht währenddessen in den Pausezustand, d.h. stoppt den Schleifendurchlauf. Die Daten der skalaren Multiplikation und des geometrischen Produkts wandern, jeweils begleitet durch **ValidSig**, *Stufe* für *Stufe* durch die *Registerebenen* des Rechenwerks. Drei Takte nach **PreResReq** zeigt **PreResSig** an, daß die erste angeforderte Teilsumme am Ausgang des *SRAMs* anliegt. Über Leitung **PreResult** kommt sie, wie auch die vier folgenden, direkt in den B-Puffer. Die Adreßeinheit hat mittlerweile erneut den Betrieb aufgenommen und schaltet wieder den Inhalt des **AddrCntA** zum **RdAddrReg** durch. Parallel ist die Multiplikation der Skalaranteile beendet, gekennzeichnet durch den *High*-Zustand des letzten *FlipFlops* der **ValidSig**-Kette. Dessen Ausgang wird als Schreibsignal für den A- und C-Puffer verwendet, wo der skalare resp. algebraische Teil des Berechneten gespeichert wird. Der restliche Weg der Daten durch das Rechenwerk entspricht den Schilderungen in Abschnitt 4.1.2 auf Seite 29.
Die Leitungen **ScalarOut** und **BladeOut** werden mit den Eingängen **ScalarIn**, bzw. **BladeIn** des Adreßwerks verbunden. Die Ergebnisse gelangen in das **DiReg** und werden gespeichert, da gleichzeitig **ValidSigOut** gesetzt ist, d.h. eine Schreibanforderung stellt.

Sind alle A-*Blades* mit dem ersten B-*Blade* verknüpft worden, wird der A-Zähler auf die Adresse von $MVA[0]$ zurückgesetzt. Der B-Zähler inkrementiert und die Adreßeinheit liest

MVB[1] aus, der wiederum mit allen A-*Blades* kombiniert wird. Dieser Vorgang wiederholt sich, bis alle A- mit allen B-Operanden verknüpft wurden.
Dann wird mit dem Auslesen der Folgeinstruktion begonnen. Ist in `IW0` ein bestimmtes Kontrollbit nicht gesetzt, wird dies als Ende aller Berechnungen interpretiert und Leitung **`End`** des Kontrollwerks auf einen hohen *Pegel* gezogen.

4.2.4 Auslesen der Ergebnisse

Das im vorigen Abschnitt beschriebene *Polling* erkennt das Signal **`End`** und löscht **`opEnG`**, um alle Aktivitäten zu beenden:

```
m_pBus[3] = 0000b; // opEng := false
m_pBus[2] = 0x0;   // Leseadresse := 0
m_pBus[3] = 0010b; // RdThru := true
```

In den **`PRLink`**-Zähler wird die Adresse Null geladen. **`RdThru`** wird aktiviert, um die Instruktionsdaten zur Kontrolle zurückzulesen. Es soll nach Anzeichen von Übertragungsfehlern gesucht werden. Um danach die Resultate herunterzuladen, muß der **`PRLink`**-Zähler auf die entsprechende Startadresse eingestellt werden. **`RdThru`** wird hierfür kurzzeitig deaktiviert. Das *SRAM* wird anschließend bis zur obersten Speicherstelle ausgelesen. Die Ergebnisse werden anhand der *Instruktionen* aus dem *SRAM* nachgerechnet, um Fehler möglichst nur auf die Operationsphase des *Designs* zurückzuführen.
Der zuvor beschriebene Kontrollmechanismus kann softwaremäßig ausgeschaltet werden, so daß nur die Resultate ausgelesen werden.

Kapitel 5

Erweiterungen

Dieses Kapitel soll einen tieferen Einblick in den Designablauf ermöglichen. Es wird gezeigt, welche Probleme das *Pipeline*-Konzept mit sich bringt und wie sie einfach gehandhabt werden [44, 25, 46, 43, 41]. Eine komplexe Erweiterung hierzu präsentiert Folgeabschnitt 5.3. Zuvor wird eine einfache durchsatzsteigernde Maßnahme vorgestellt.

5.1 Skip-Kommando

Das Skip-Kommando bewirkt einen Abbruch des aktuellen *Durchlaufs*, falls dessen B-*Blade* nullwertig ist. Die Analogie zur `C++`-Anweisung `continue` vermittelt folgendes, erfundenes Programmfragment [40].

```
// outer loop
for ( int b=0; b < size( MVB ); b++ )      // Get one B-Blade
{
  if ( scalar( MVB[b] ) == 0 )             // skip this run ??
    continue;                              // SKIP !!

  // inner loop
  for ( int a=0; a < size( MVA ); a++ )    // Get all A-Blades
  {
    GP( MVA[a], MVB[b] );                  // geometric product
  }
}
```

Ein **`Skip`** ist von Vorteil, falls Multivektor B Ergebnis einer vorherigen *Instruktion* ist, in dem sich eine größere Anzahl von Null-Blades befinden. Sie repräsentieren Speicherstellen, auf die nach ihrer Initialisierung zu Null nicht wieder zugegriffen wurde:
Die Wahrscheinlichkeit, daß ein B-*Blade* eines Ergebnisvektors Null ist, ist um so größer, je kleiner die Gesamtanzahl aller Operanden derjenigen *Instruktion* war, die das Ergebnis erzeugt hat. Ist beispielsweise Operand A zehn Elemente und Operand B sieben Elemente mächtig, kann das Ergebnis der *Instruktion* höchstens 70 von maximal 256 Einträgen besitzen. Ist dieser nur zu 27% belegte Resultatvektor ein B-Operand, müssten $256 - 70 = 186$ Null-Blades mit jeweils 256 A-*Blades* (im ungünstigsten Fall) verknüpft werden. Ein Skipping

initiiert statt dessen einen neuen B-Zyklus (*Durchlauf*). Es beinhaltet das Inkrementieren des **`AddrCntB`** und das Zurücksetzen aller zum A-Zähler gehörigen Bauteile des Adreßwerks. Bis das nächste B-*Blade* im Kontrollwerk ist, können maximal fünf A-*Blades* eintreffen. Sie sind die ersten Elemente des *Multivektors* A, die bereits von der Adreßeinheit bis zu der Reaktion auf den **`Skip`** geliefert wurden. Dieser Nachlauf wird als nicht *gültig* markiert und durchläuft die *Pipe*, ohne ein Ergebnis hervorzubringen. Es gibt drei Voraussetzungen für einen Schleifenabbruch durch ein Skip-Signal:

- Die Anzahl der A-Operanden darf nicht kleiner als sechs sein.
- Die Anzahl der verbleibenden B-Operanden einer *Instruktion* darf nicht kleiner als zwei sein.
- Das Skipping muß durch ein Kontrollflag im `IW0` der momentanen *Instruktion* erlaubt sein.

Das *Flag* aus der letzten Bedingung wird zu Debug-Zwecken benötigt und sollte immer gesetzt sein.
Die erste Voraussetzung soll verhindern, daß der falsche *Durchlauf* abgebrochen wird. Bei weniger als sechs A-*Blades* ist es aufgrund des oben genannten Nachlaufs möglich, daß der aktuelle *Durchgang* bereits beendet war, als das Skip-Signal gegeben wurde. Es würde eine Runde übersprungen werden.
Der zweite Punkt dient dazu, daß ein **`Skip`** nicht den letzten *Durchlauf* einer *Instruktion* abbricht, um automatisch einen neuen zu starten. Das würde eine Art Bereichsüberschreitung verursachen, vgl. dem Zugriff auf dasjenige Element hinter der letzten Speicherstelle eines `C++`-Feldes. Über das 256ste *Blade* hinauszulesen ist möglich, weil die Dekrementer **`DecA`** und **`DecB`** neun Bit breit sein müssen. Das *Design* würde in einen nicht definierten Zustand versetzen werden. Die zweite Bedingung unterbindet weiterhin Störungen, die durch das komplexe Zusammenspiel aller Signale während des Instruktionswechsels auftreten könnten.
Als Vorsichtsmaßnahme, und weil es technisch einfacher ist, werden statt der erwähnten Grenzwerte sechs und zwei, die nächstgrößeren Zweierpotenzen acht und vier verwendet. Für die Leistung des *Designs* ergibt sich nur ein marginaler Rückgang.

Diese Art der Durchsatzsteigerung funktioniert nicht für A-*Blades*. Wird jedoch eines als nullwertig identifiziert, geht **`ValidSig`** nicht auf *High*, und es gibt kein **`PreResReq`**. Die Daten in der *Pipe* gelten als nicht *gültig*. Die einzige Einbuße besteht in dem einen für den Ausleseprozeß in Anspruch genommenen *Taktzyklus*.

5.1.1 Beispiel eines Skip-Fehlers

Das Zeitdiagramm in Abbildung 5.2 auf Seite 47 zeigt die Auswirkungen eines Skip-Kommandos bei einer Mißachtung der ersten Regel (weniger als sechs A-*Blades*).
Die dargestellte Situation entspricht derjenigen unmittelbar nach einem Designstart. Die Instruktionsworte der Adressen `0x00`,`0x01` und `0x02` wurden gerade gelesen und interpretiert. Die folgende Abbildung verschafft einen Überblick des *SRAM*-Inhalts. Es gibt drei

`000`	`IW0`
`001`	`IW1`
`002`	`IW2`
`003`	`MVA[0]=01Eh`
`004`	`MVA[1]=331h`
`005`	`MVA[2]=023h`
`006`	`MVB[0]=000h`
`007`	`MVB[1]=0FEh`
`008`	`MVB[2]=002h`
`009`	`MVB[3]=009h`
`00A`	`IW0’`
`00B`	`...`

Abb. 5.1: Speicherauszug für das Beispiel.

A-*Blades* an den Speicherstellen `0x03`, `0x04` und `0x05`, die mit den vier drauffolgenden B-*Blades* verknüpft werden sollen. Das nicht aufgeführte Signal **`Phase_B`** des Kontrollwerks beendet anfangs den Pausezustand (→ **`Pause`**) des Adreßwerks. Dessen interne Steuerleitung **`BCycle`** leitet jeweils den aktuellen Inhalt von **`AddrCntB`** an das **`RdAddrReg`** durch und initiiert neue B-Schleifendurchläufe, d.h. lädt **`AddrCntA`**, bzw. **`ADec`**[1]. Das **`DsgnDataReg`** empfängt in Takt Fünf den nullwertigen B-Operanden der Adresse `0x06`. Als Konsequenz wird ein Skip-Signal gegeben, so daß in Takt Sieben ein neuer B-Zyklus gestartet wird. Zu diesem Zeitpunkt war der abzubrechende *Durchgang* schon seit zwei Taktzyklen beendet. Der angefangene neue *Durchlauf* wird irrtümlich um zwei Berechnungen verkürzt. Der dritte **`BCycle`** hätte erst in Takt Neun auftreten dürfen. Die noch ausgelesenen *Blades* `0x0FE` und `0x01E`

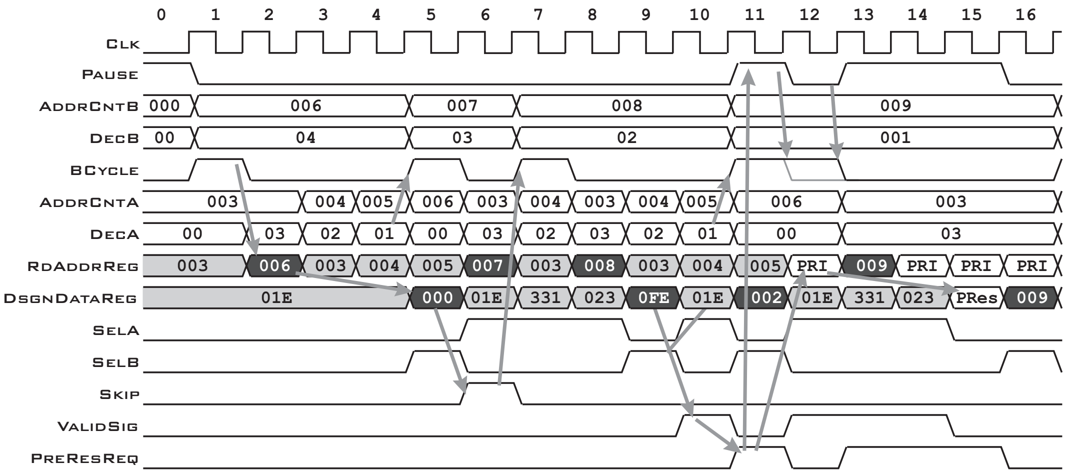

Abb. 5.2: Fehler und weiterer Verlauf nach einer Verletzung der ersten Skip-Bedingung. Zum A- und B-Vektor gehörende Daten sind hell bzw. dunkel unterlegt.

ergeben ein *gültiges* Paar, so daß **`ValidSig`** auf *High* geht und in Takt 11 per **`PreResReq`** eine Teilsummenanfoderung stellt. Der Ausdruck **`PRI`** im **`RdAddrReg`** steht für **`PreResIdx`**

[1]**`AddrCntA`** hat zum Auslesen der Instruktionsworte drei mal hochgezählt und steht korrekterweise bereits auf Adresse `0x03`.

und symbolisiert die Adresse des gewünschten Ergebnisblades. Deren entsprechendes Speicherwort wird durch **`PRes`**, für **`PreResult`**, repräsentiert. Jedes **`PreResReq`**-Signal versetzt das Adreßwerk in den Pausezustand, in dem die Zähler keine Übergänge machen (Takt 11, 13, 14, 15).

5.2 Datenabhängigkeiten

Datenabhängigkeiten spielen bei fast allen heutigen Prozessoren eine immense Rolle [44, 25]. Sie treten auf, wenn zwei Entitäten (*Instruktionen* oder Bauteile) gleichzeitig exklusiven Zugriff auf eine Ressource (Daten im *Cache*, RAM, *Bus*,...) benötigen - es sei denn, es handelt sich ausschließlich um Leseoperationen. Es gibt drei Arten von sog. Data Hazards: RAW, WAR, WAW. Ein W steht für 'write', ein R für 'read' und ein A für 'after'. Letzteres bezieht sich auf die eigentliche zeitliche und semantische Reihenfolge der Einzelaktionen R resp. W, d.h. wie sie sein sollte. Bei einem *Hazard* ist dieser Ablauf gerade umgekehrt.

- RAW: Diese Abhängigkeit besteht, wenn ein lesender Zugriff (→R) einer Folgeinstruktion (→A) dann stattfindet, falls das zu schreibende Datum (→W) der Vorgängerinstruktion noch keine Gültigkeit erlangt hat.
- WAR: Das ursprünglich später eintretende Write verändert das Datum, bevor es ausgelesen werden konnte. Dies kann sich z.B. ereignen, wenn eine *Instruktion* am Ende ihrer Berechnungen Daten zur Adressierung aus dem Instruktionsregister benötigt, dieses aber bereits durch nachfolgende Befehle überschrieben wurde.
- WAW: Das Ergebnis einer *Instruktion* wird im Nachhinein durch das einer vorangehenden, länger andauernden Operation überschrieben. Beispiel: Ein Produkt soll als Feldindex verwendet und zuvor in einem *Register* gespeichert werden. Die unteren Bits dieses Wertes sollen durch eine nachfolgende Store-Funktion explizit überschrieben werden. Da die Multiplikation wesentlich mehr Taktzyklen benötigt als der Store-Befehl, setzt letzterer die unteren Offsetbits bevor das Produkt berechnet ist. Sie werden anschließend von dem Multiplikationsergebnis überschrieben und gehen verloren.

Alle drei genannten Typen gab es im *Design*. Sie entstehen nur durch die Verwendung des *Fließbandprinzips*. WAR- und WAW-Hazards konnten durch einfache Maßnahmen unterbunden werden. Der 'read after write'-*Hazard* tritt an zwei Stellen auf. Der offensichtlichere Fall, für den es nur eine Lösung gibt, wird nachstehend beschrieben.

5.2.1 Hazards beim Arbeiten mit Ergebnisvektoren

Die Hazards entstehen, wenn auf Ergebnisblades im *SRAM* zugegriffen wird, die nicht aktuell sind, weil sich die neuen Daten noch in der *Pipe* befinden. Diese Situation kann nach einem Instruktionswechsel I→I' auftreten, falls mindestens einer der Operanden *MVA'* und *MVB'* der neuen *Instruktion* I' dem letzten Ergebnisvektor EV entspricht. Nachdem die

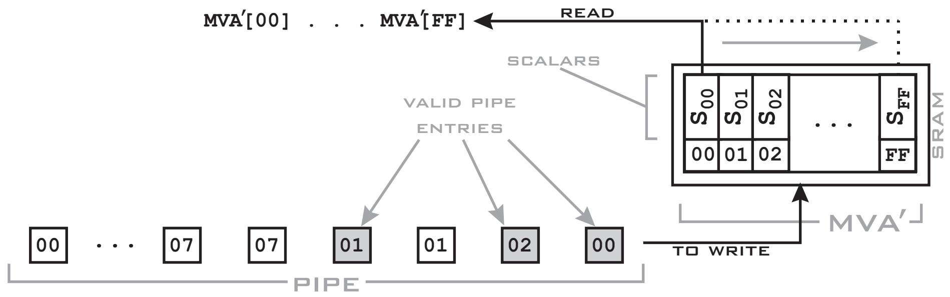

Abb. 5.3: RAW-*Hazard*. Der Zustand von *MVA'* im *SRAM* ist inkonsistent, da die *Blades* `01`, `02` und `00` der vorigen *Instruktion* noch nicht geschrieben wurden. Deren Auslesen führt zum Konflikt. Auf *Blade* `01` dürfte erst in fünf Takten zugegriffen werden.

Wörter `IW0`, `IW1` und `IW2` von I' gelesen wurden, werden die Operanden in der Reihenfolge *MVB'*[0]→ *MVA'*[0]→ *MVA'*[1]→ *MVA'*[2]→ etc., angefordert. Ist etwa *MVA'* == EV und EV[1] noch ungespeichert, führt der Zugriff auf *MVA'*[1] zu einem RAW-Hazard.

Die einfachste Lösung besteht darin, vorher zu warten, bis alle ausstehenden Elemente der *Pipeline* gespeichert sind. Dies wird durch einen Pausezustand des Adreßwerks während des ersten B-Zyklusses von I' realisiert. Bit 21 des `IW0` von I' und einige Takte später, Bit Vier des **`OpSpecReg`**, enthalten das **`LASTResult`**-*Flag*, das anzeigt, ob die Möglichkeit eines Konfliktes durch den gerade laufenden Instruktionswechsel besteht. Ist es gesetzt, gibt es an, daß mindestens ein Operand der *Instruktion* I' mit dem vorigen Ergebnisvektor EV identisch ist[2]. Es wird nur gewartet, falls das *Flag* gesetzt ist. Der Pausezustand wird so lange aufrecht erhalten, bis die *Pipe* leer ist. Hierfür gibt es das **`Empty`**-Signal. Es zeigt an, ob sich im *Design* noch *gültige*, einmal zu speichernde *Blades* befinden. Zur Auswertung wird der Teil der *Pipeline*, ab einschließlich der Operandenregister des Rechenwerks herangezogen. Ein A-*Blade* ungleich Null im **`DsgnDataReg`** wird von **`Empty`** beispielsweise nicht mehr berücksichtigt.

Abbildung 5.4 zeigt den Ablauf eines Instruktionswechsels, bei dem ein möglicher Konflikt verhindert worden wäre. Während die letzten drei A-*Blades* `0x000`, `0x000` und `0x007`, der Adressen `AA0`, `AA1` und `AA2` ausgelesen werden, leitet das interne Signal **`AExclusive`** den Übergang zu I' ein. Es steuert das Laden und das zweimalige Hochzählen des **`AddrCntA`**. Die Speicherstellen `AI0`, `AI1` und `AI2` beinhalten die Instruktionsworte `IW0`, `IW1` und `IW2` von I'. Parallel geht **`Pause`** auf *High* und deklariert alle fortan ausgelesenen Werte als Nicht-*Blades*. Da die *Pipe* ab Takt Drei unbelegt ist, hätte I' problemlos anlaufen können. Der letzte A-Operand ist indes `0x007` und bewirkt eine Teilergebnisanforderung per **`PreResReq`**, so daß **`Empty`** wieder auf *Low* geht. Da das hier behandelte *Design* ab dem **`PreResReq`** 13 Takte benötigt, bis die korrespondierenden Daten die *Pipe* verlassen haben, wird der **`BCycle`** um insgesamt 11 Takte verlängert (nicht eingezeichnet).
Wäre der `AA2`-Wert auch nullwertig gewesen, wäre **`Empty`** auf hohem *Pegel* geblieben und **`Pause`** wäre zusammen mit **`PhB`** (**`Phase_B`**) auf *Low* gegangen. **`BCycle`** wäre ferner nicht über Takt 10 hinausgezögert worden.

Das in den letzten Passagen beschriebene Datenabhängigkeitsproblem ließe sich durch *Forwarding* nicht angemessen lösen, da der Aufwand in keinem Verhältnis zum Nutzen steht.

[2]Bei einer langen *Pipe* kann es noch Konflikte mit der vorletzten *Instruktion* geben (I mit I''), wenn die letzte sehr kurz war, d.h. ein einziges B-*Blade* und wenige A-*Blades* enthielt. **`LASTResult`** ist dann ebenfalls gesetzt.

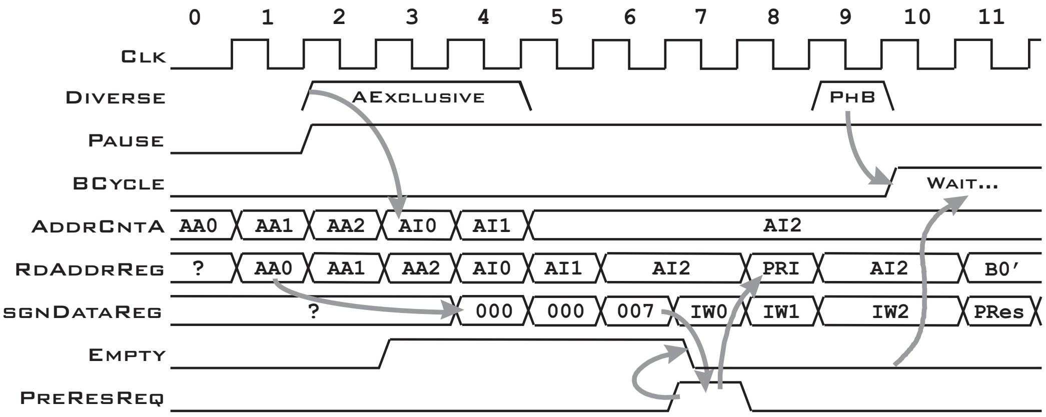

Abb. 5.4: Warten beim Instruktionswechsel. Weil die *Pipe* in Takt Zehn nicht leer ist, wird **`BCycle`** verlängert, bis **`Empty`** wieder auf *High* geht.

Forwarding ist mit einem Bypass zu vergleichen: Soll ein Ergebnisblade aus dem *SRAM* gelesen werden, wird anhand des Index zuerst geprüft, ob und wo es sich in der *Pipe* befindet. Ist es verfügbar, wird es direkt an die anfordernde Stelle im *Design* geleitet, statt den veralteten Wert aus dem *SRAM* zu nehmen. Dieser wird derweil auf den neuesten Stand gebracht.

5.2.2 Hazards beim Wechsel des B-Operanden

Hier ist die sehr viel häufiger auftretende zweite Stelle, an der es RAW-Hazards gibt. Wie im vorigen Abschnitt führt das Auslesen veralteter Bladeinformationen, während sich die eigentlichen, meist unfertigen Daten noch in der *Pipe* befinden, zu falschen Resultaten.
Die Auslöser hierfür sind die jeweils ersten Teilergebnisanforderungen eines neuen *Durchgangs* R+, der stets durch das **`BCycle`**-Signal eingeleitet wird. Simultan wird das letzte, zu dem vorherigen *Durchlauf* R gehörige A-*Blade* ausgelesen. Zusammen mit dessen unmittelbar vorausgegangenen A-Operanden von R, können sich zu diesem Zeitpunkt bis zu zehn *gültige* Einträge in der *Pipeline* befinden, deren betreffende Resultate baldig zur Speicherung im *SRAM* anstehen. Bis der erste **`PreResReq`** als der frühestmögliche Konfliktauslöser von Folgedurchgang R+ stattfindet, halten sich schließlich noch maximal fünf designierte Ergebnisse von *Durchlauf* R in der *Pipe* auf. Sie könnten eine Daten-Kollision verursachen, falls die Teilergebnisanforderung auf eines von ihnen abzielt.
Abbildung 5.5 veranschaulicht, wie es zu Konfliktsituationen kommen kann. Es wird anhand einer beliebigen *Instruktion* die Belegung der *Pipeline* während des Wechsels von *Durchgang* R auf *Durchgang* R+ gezeigt. Rechts der Grafik ist angegeben, daß in Takt Null das **`BCycle`**-Signal *High* ist, worauf sich in der folgenden Zeile Eins konsequent die Adresse des entsprechend auszulesenden B-*Blades* von *Durchgang* R+ im **`RdAddrReg`** befindet.

Die Spalten A bis Q entsprechen den 17 *Registerebenen* des *Designs*. Einige Spalten können durch besonders repräsentative *Register* vertreten werden. Die Elemente wandern in jedem Takt um eine Ebene von links nach rechts durch die *Pipeline*. Die Zeilen zeigen die korrespondierenden Belegungen der dargestellten acht aufeinanderfolgenden Taktzyklen.
Als Beispiel sei Position P7 angeführt, an der eine Fünf steht. Es bedeutet laut Legende, daß die Adresse des Ergebnisses der Verknüpfung von A-*Blade*[3] Nr.5 mit dem betreffenden B-

[3]Nr.X ist ein prak. Bezeichner und steht nicht für *MVA*[X]. Es ist z.B. auch M0 $\not\sim$ A2.

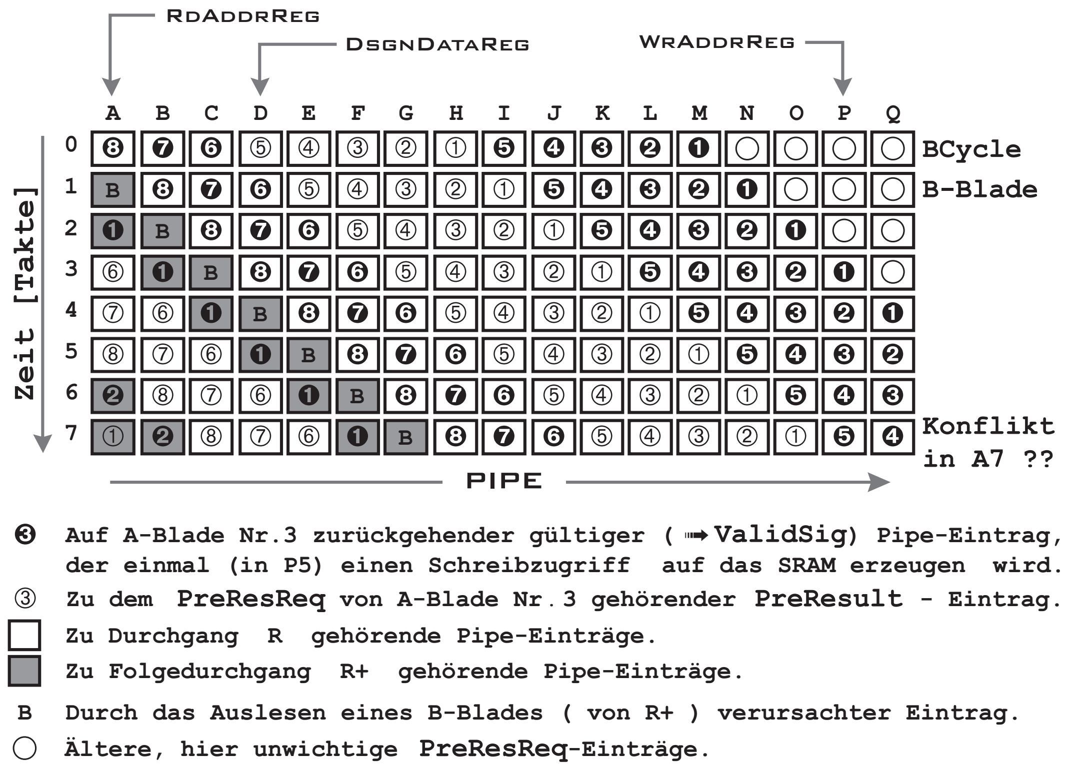

Abb. 5.5: *Pipe*-Belegung über acht Taktzyklen. Die Spalten A bis Q repräsentieren die 17 *Register (-ebenen)* des *Designs*. Element A7 steht im potentiellen Konflikt mit H7,I7,J7,P7 und Q7. Erklärung siehe Text.

Blade von *Durchgang* R in das **`WrAddrReg`** geladen wurde. Gültige Elemente, die Spalte Q verlassen haben, wie die Drei aus Position P5 (in Takt Sieben), vermögen keinen *Hazard* zu erzeugen, da ihre Speicherung im *SRAM* konsistent geworden ist. Nicht *gültige* Einträge, die durch das Auslesen von B-*Blades* oder **`PreResult`**s entstehen, werden durch ein `B` resp. eine helle 'Billardkugel' symbolisiert. Sie durchlaufen die *Pipe* vollständig, werden dabei wie A-Blades behandelt, d.h., sie werden allen anfallenden Operationen unterzogen, um schließlich dennoch verworfen zu werden.

Die Konfliktsituation aus Abbildung 5.5 ergibt sich in Zeile Sieben: In E6 wird zunächst der **`PreResReq`** für das erste Operandenpaar von *Durchgang* R+ erzeugt, das aus den beiden in A1 bzw. A2 ausgelesenen Elementen, B-*Blade* und A-*Blade*[4] Nr.1 besteht. Der resultierende Lesezugriff in A7 könnte mit einem der fünf *Pipe*-Einträge H7, I7, J7, P7 und Q7 kollidieren. Ihre Schreibzugriffe erfolgen auf den gleichen Ergebnismultivektor, aus dem das in A7 angeforderte Teilergebnis **`PreResult`** kommt, da R und R+ einer *Instruktion* angehören. Weil alle fünf ausstehenden Einträge erst innerhalb der nächsten zehn Takte gespeichert sein werden, d.h. konsistent im *SRAM* vorliegen, könnte der **`PreResReq`** eines der noch nicht aktualisierten Teilergebnisse auslesen.
Angenommen, der (zu F7 gehörende) Eintrag in A7 würde mit P7 kollidieren, so daß das erhaltene **`PreResult`** falsch wäre. Die Speicherung von P7 wäre in Takt Neun abgeschlossen und würde in Takt 19 von dem folgefalschen Ergebnis der Berechnungen von Eintrag F7

[4] Hier ist 'A-Blade Nr.1' gleich *MVA*[0].

überschrieben werden.
Die zu der '2' an Position B7 gehörige Teilergebnisanforderung findet in Takt 11 statt. Zu diesem Zeitpunkt haben die Elemente aus P7 und P8 die *Pipeline* bereits verlassen, während Einträge aus H7, I7 und J7 noch immer einen Konflikt erzeugen könnten.

Es sei darauf hingewiesen, daß Abbildung 5.5 gut wiedergibt, wie zukünftige Ergebnisblades (*gültige* Elemente: dunkle 'Billardkugeln') stets im Abstand von fünf Ebenen/Takten von ihren korrespondierenden **`PreResult`**-Einträgen gefolgt werden. Dieser maximale Abstand ergibt sich, da die **`PreResReq`**s, in Ebene[5] E erzeugt werden, wie etwa die Sechs an Position E2. Aus diesem Grund entsteht in der *Pipeline* häufig eine regelmäßige 5er-Block-Bildung.

> Ein andere Darstellung zur Erklärung des 5er-Abstands findet sich in dem Zeitdiagramm auf Seite 47. Dort wird in Takt Sieben auf Speicherstelle `0x003` zugegriffen. Der ausgelesene Wert `0x01E` des A-Operanden hat einen *gültigen* Eintrag zur Folge, so daß sich fünf Zyklen später, in Takt 12, Adresse **`PRI`** (für **`PreResIdx`**) im **`RdAddrReg`** befindet.

Es werden zwei weitere Beispiele angegeben, wie der Wechsel zwischen zwei Durchläufen aussehen kann:

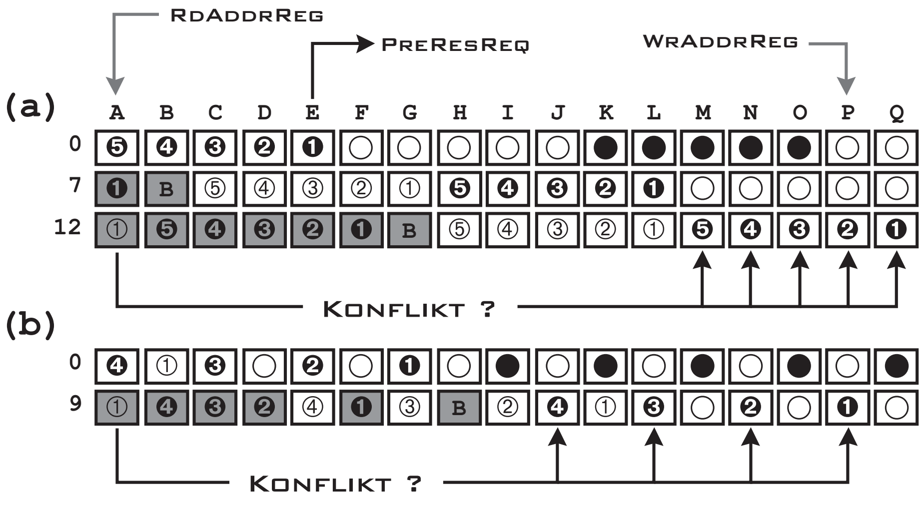

Abb. 5.6: *Pipe*-Belegungen. **`BCycle`** ist in Takt Null jeweils *High*.

In beiden Konstellationen aus Abbildung 5.6 ist Signal **`BCycle`** im Takt Null auf *High* gegangen, um den nächsten *Durchgang* R+ einzuleiten bzw. dessen B-*Blade* auszulesen.

(a) In dieser Ausgangssituation bildet der Eintrag in E0 nur das erste Element des 5er-Blocks E0 bis A0, das einen **`PreResReq`** erzeugt. Hierdurch wird **`BCycle`** fünf Takte in Folge ignoriert (G7 bis C7), bis der neue *Durchgang* R+ gestartet werden kann. In Takt 12 sind fünf designierte Resultatblades ungespeichert, die mit der Teilergebnisanforderung an Position A12 im Konflikt stehen.

[5] Spalte E kann mit den Operandenregistern des Rechenwerks (s.S. 28) assoziiert werden.

(b) Diese alternierende Belegung kann sich einpendeln, falls einige A-*Blades* von R zuvor nullwertig waren. Der einsetzende *Durchgang* R+ wird anfangs zerstückelt, da die **`PreResReq`**s (von R) im Operationsmodus die höchste Priorität besitzen. In Takt Neun existieren vier mögliche Konfliktpartner.

Im folgenden wird aus Sicht der algebraischen Seite die Entstehung von Konflikten erläutert. Zur Bestimmung des Enthaltenseins der Basiselemente e_1 bis e_8 im künftigen Ergebnisblade im Rahmen des geometrischen Produkts (Abschnitt 2, Seite 6) wird eine *XOR*-Verknüpfung durchgeführt. Der Schleifenablauf im Adreßwerk impliziert, daß die A-Operanden bei festem B-Operanden variiert werden. Da erstere alle disjunkt sind, ist die Bildung der Ergebnisblades durch *XOR* innerhalb eines *Durchgangs* eine injektive Abbildung. Einen Konflikt während dieser Periode kann es nicht geben.
Ein *Hazard* entsteht immer dann, wenn unter den letzten *Blades* des vorangegangen *Durchlaufs* R und unter den ersten des direkt nachfolgenden R+ mindestens ein gemeinsames Ergebnisblade exisitiert. In dieser Situation mit zwei variablen Operanden ist *XOR* nicht mehr injektiv, was durch ein Beispiel in Abbildung 5.7 demonstriert wird. Die Indices `001b` und

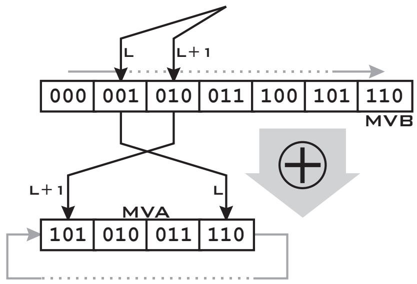

Abb. 5.7: Konflikt: Das *XOR* der binär dargestellten, drei Bit breiten Bladeindices `001b` und `110b` in Schritt `L` bildet auf das gleiche Resultat (`111b`) ab, wie das des unmittelbaren Folgeschritts `L+1`.

`110b` sowie `010b` und `101b` im neuen Zyklus bilden beide auf `111b` ab, so daß sich ein RAW-*Hazard* bildet. Wenn die zweite Operation *Blade* `111b` aus dem *SRAM* anfordert, wird sie falsche Daten bekommen, da die erste noch nicht geschrieben hat. Wenn der aktuelle B-Index `010b` auf A-Index `011b` trifft, kommt es im Folgedurchgang zu einem weiteren Konflikt mit dem Paar aus B-Index `011b` und A-Index `010b`.

Als angemessene Gegenmaßnahme kommt erneut Warten in Betracht. Die Mechanismen und Auswirkungen sind die gleichen wie im vorigen Abschnitt. Im Gegensatz dazu wird kein spezieller Auslöser wie das **`LASTResult`**-*Flag* benötigt, da jeder **`BCycle`** verlängert werden muß, bis das **`Empty`**-Signal gegeben wird. Daß noch zur steigenden *Flanke* von **`BCycle`** eine (letzte) A-*Blade*-Adresse in das **`RdAddrReg`** übernommen wird, ist ein Erschwernis, da die *Pipe* zu diesem Zeitpunkt unbelegt sein könnte. **`Empty`** würde fälschlicherweise das Ende des Wartens einleiten. Um dies zu verhindern, existiert im Adreßwerk der Zähler **`WaitCntr`**, der am Ende eines jeden *Durchgangs* mindestens sechs Zählschritte macht, so daß **`BCycle`** um insgesamt sieben Takte verlängert wird. Zum Ende der Zählperiode ist selbst das letzte A-*Blade* ausgelesen und entschieden, ob sich ein *gültiger Pipe*-Eintrag daraus ergibt. Ab diesem Moment kann **`Empty`** zuverlässig bestimmen, wie lange der Warte-, resp. Pausezustand weiter aufrecht erhalten wird.

Das zugrunde liegende *Design* hat 13 *Stufen*, zu denen noch drei hinzugenommen werden müssen - zwei für die *Latenz* beim Auslesen und eine für die beim Beschreiben des *SRAMs*. Die Wahrscheinlichkeit eines Konflikts ist um so größer, je länger die *Pipe* ist und desto kleiner die *Algebra* ist. Bei Letzterer sinkt mit der Zahl der Basiselemente die Menge der möglichen Kombinationen zu *Blades* und somit die statistisch mittlere Zeitdauer zwischen dem Erscheinen zweier gleicher Indices. Die Länge der *Pipeline* hängt von der verwendeten Breite der Eingabedaten und der Stufigkeit ab.

5.3 Dynamisches Warten

Bei dem dynamischen Warten handelt es sich um eine Alternative gegenüber dem zuvor beschriebenen unbedingten Warten. Das *Design* kann vor seinem Berechnungsstart über die in Abschnitt 4.1.3 auf Seite 31 erwähnte **`WaitAlwaysNode`** in einen der beiden Modi versetzt werden. Die Node fungiert als Kontrollflag und wird im **`GlobalReg`** des Kontrollwerks gespeichert. Wurde sie gesetzt, wird die *Pipe* nach jedem Zyklus vollständig geleert. Ist sie *Low*, werden *Blades*, die einen Konflikt verursacht haben, direkt hinter den letzten regulären A-Operanden des *Multivektors* wieder eingereiht und durchlaufen das *Design* anschließend ein zweites Mal, ohne eine Kollision hervorzurufen. Das auf falschen Daten beruhende Ergebnis des ursprünglichen Laufs wird verworfen.

5.3.1 Konflikte beheben

Die nachstehenden Ausführungen beschreiben den 'Wiedervorlage'-Vorgang, der bei einem Konflikt in Kraft tritt. Eine erste Notwendigkeit besteht im rechtzeitigen Aufspüren eines Konflikts. Die in keiner *Design*-Hauptkomponente integrierte Testeinheit **`CheckUnit`** erfüllt diese Funktion. Eine detaillierte Umschreibung ist Gegenstand von Abschnitt 6.1.5. Das Bauteil erhält parallel zur Adreßeinheit ebenfalls die Teilergebnisanforderungen über **`PreResReq`** und **`PreResIdx`**. Die beiden Leitungen werden mit den Eingängen **`ValidSig`** bzw. **`Index`** verbunden (Abb. 6.5, S. 63).
Das **`PreResReq`**-Signal wird direkt hinter den Eingaberegistern des Rechenwerks erzeugt. Die resultierende **`PreResult`**-Anforderung kann mit einem von bis zu sechs designierten, nicht gespeicherten Ergebnisblades kollidieren, die im restlichen Teil des *Designs* enthalten sein können. Die **`CheckUnit`** vergleicht den **`PreResIdx`** mit den Indices dieser *Blades* und läßt den Ausgang **`Conflict`** bei einer Übereinstimmung auf *High* gehen. Zur Vorsicht und aufgrund der Konsistenzbedingungen des **`DPSRAM`**s darf sich eine Adresse nicht im nächsten Takt, nachdem sie im **`WrAddrReg`** war, im **`RdAddrReg`** befinden. Deshalb muß die Testeinheit den Index der Schreibadresse, die sich während der Anforderung im **`WrAddrReg`** befand, als siebtes Vergleichselement miteinbeziehen. Bei *Designs* mit einer sehr kleinen Stufigkeit kann **`CheckUnit`** eine Tiefe von über 16 haben. In ihr wird als Nebenprodukt das oftmals verwendete **`Empty`**-Signal erzeugt.

fail-save Mechanismus

Der fail-save Mechanismus des Kontrollwerks reagiert in Konfliktsituationen. Er steuert die sehr komplizierte Speicherung und Wiederfreisetzung von *Blades*, die einen *Hazard* erzeugt

haben.
Der Inhalt des **DsgnDataReg**s wird über zwei Zwischenregister **ImmRegA** und **ImmRegB** gesichert (Abb. 5.8). Wenn **Conflict** auf *High* geht, wird das Verursacherblade des *Multivektors* A (!) in einen *Fifo*/Puffer übernommen. Solange dieser nicht leer ist, wird über Ausgangsleitung **ReleaseReq** die Bereitschaft angezeigt, dessen Inhalt erneut in die *Pipe* zu geben. In der TTL[6] werden die Signale **WaitReq** und **ReleaseReq** über eine ODER-

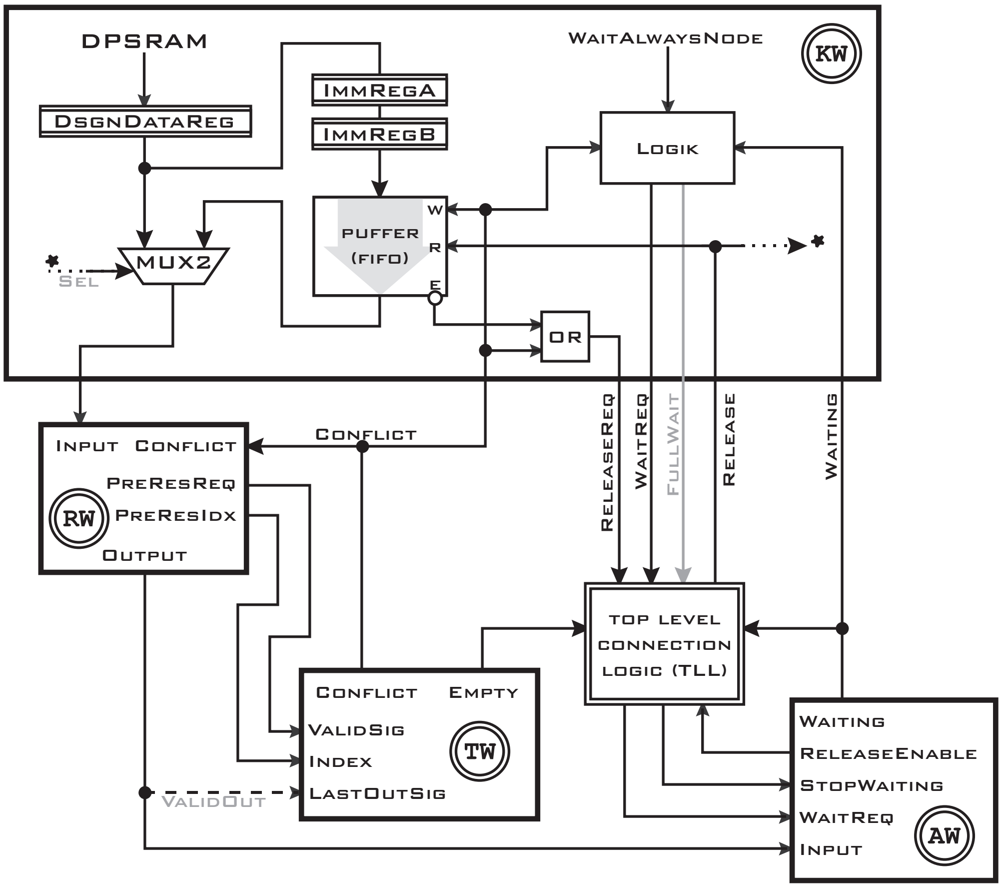

Abb. 5.8: Konzept zur Konflikthandhabung. Die dargestellten Leitungen des Testwerks TW, Kontrollwerks KW, Rechenwerks RW und der Adreßeinheit AW, werden zur Behandlung von Konfliktsituationen benötigt. Zwischen den Bauteilen existiert eine separate Verbindungslogik TLL.

Verknüpfung zu einem allgemeinen **WaitReq** für das Adreßwerk zusammengefaßt. Im Falle eines Hazards muß vor dem nächsten *B-Zyklus* gewartet werden, bis die zuvor gesicherten *Blades* wieder in die *Pipe* eingegliedert werden können. Hierzu wird der in Abschnitt 5.2.2 (Seite 53) erwähnte **WaitCntr** verwendet, der von Null an hochzählt, sobald **BCycle** auf *High* gegangen ist. Steht der Zähler auf Vier, hat das letzte A-*Blade* des aktuellen *Durchgangs* (das ggf. noch einen Konflikt erzeugen könnte) gerade das **DsgnDataReg** verlassen und das Adreßwerk gibt das Signal **ReleaseEnable**. Es wird unabhängig davon gebildet, ob ein **ReleaseReq** vorliegt, zeigt hingegen den Zeitpunkt an, ab dem *Blades* aus dem Puffer freigesetzt werden könnten, um sich möglichst nahtlos an den Datenfluß der *Pipe* anzuschließen.

[6] Diese Logik hätte genausogut in die Kontrolleinheit integriert werden können.

In der Zählerstellung Vier ist überdies **`Empty`** korrekt ausgewertet. Eine leere *Pipe* in dieser Situation gestattet es der Logik TTL, das **`Release`**-Signal zu geben. Es löst die Entleerung des *Fifos* aus. Die Nachzügler-*Blades* gelangen über einen *Multiplexer* ein zweites Mal in die Recheneinheit.
Die primäre Bedeutung des **`WaitCntr`**s ergibt sich einen Takt später, wenn definitiv feststeht, ob das letzte A-*Blade* des derzeitigen *Durchlaufs* einen Konflikt hervorgerufen hat. Bis hierhin muß gewartet werden, damit der fail-save Mechanismus rechtzeitig reagieren kann. Für mehr Spielraum zählt der **`WaitCntr`** einen weiteren Takt hoch, bis zur Sechs. Das theoretisch nicht mögliche Eintreten von **`Conflict`** nach dieser Zählperiode, würde zu einem Fehler führen, da die Beendigung des **`BCycle`**s den neuen *Durchgang* eingeleitet hätte. Wenn **`WaitCntr`** auf Sechs steht und kein **`ReleaseReq`** vorliegt, erzeugt die TTL das **`StopWaiting`**-Signal, um den Pausezustand aufzuheben. Die Leitung **`Waiting`** der Adreßeinheit teilt stets mit, ob gegenwärtig gewartet wird.
Eine leere *Pipe* ist ein sehr sicherer Anlaß, **`Release`** zu geben; dieses kommt allerdings selten vor. Zur Abhilfe existiert innerhalb der TTL der sog. **`ReleaseCntr`**. Mit dem ersten eintreffenden A-*Blade* eines *Durchlaufs*, beginnt er herunterzuzählen. Sein anfänglicher Inhalt entspricht der Länge der *Pipe* - soviele Taktzyklen müssen höchstens vergehen, bis alle *Blades* des vorherigen *Durchgangs* das *Design* verlassen haben. Dann können keine Konflikte mehr entstehen und bis dahin gepufferte A-Operanden dürfen per **`Release`** freigesetzt werden, sobald **`ReleaseEnable`** *High* ist.

In der Berechnungseinheit führt das Auslösen eines Konflikts zu einer *Ungültigkeitsdeklaration* des betreffenden A-*Blades*. Es passiert die *Pipe* in gewohnter Weise, aber das Ergebnis wird am Ende verworfen.

Das Kontrollwerk kann über die grau unterlegte Leitung **`FullWait`** signalisieren, daß ein über **`WaitReq`** angeforderter Wartezustand erst beendet wird, wenn **`Empty`** gilt. Diese vollständige Leerung der *Pipe* wird durchgeführt, wenn

- die **`WaitAlwaysNode`** gesetzt ist,
- ein Operand der Folgeinstruktion dem Ergebnisvektor entspricht oder
- ein Konflikt im letzten *Durchgang* vor einer neuen *Instruktion* stattfand.

Die TTL erzeugt einen **`WaitReq`**, falls

- die Kontrolleinheit einen **`WaitReq`** gestellt hat,
- ein **`ReleaseReq`** vorliegt oder
- solange der **`ReleaseCntr`** ungleich Null ist. Dies würde bedeuten, daß sich eventuell noch ein *Blade* des vorherigen *Durchlaufs* in der *Pipe* befindet, das zu einem Konflikt führen könnte.

Folgende Bedingungen müssen erfüllt sein, damit das **`Release`**-Signal gegeben wird:

- Es muß ein **`ReleaseReq`** existieren.
- Das Adreßwerk muß **`ReleaseEnable`** anzeigen und
- der **`ReleaseCntr`** hat heruntergezählt oder die *Pipe* ist leer.

Kapitel 6

Entwickelte Komponenten

In diesem Abschnitt sollen die entwickelten und verwendeten *CHDL*-Elemente dieser Arbeit beschrieben werden [19]. Die für eine eigene Integration benötigten Informationen, z.B. zu den Bedeutungen der Anschlußleitungen der einzelnen Komponenten, gehen aus den nachstehenden Schilderungen implizit hervor oder können den zugrunde liegenden Dateien entnommen werden.
Im Laufe des Designentwurfs ergab sich eine praktikable Aufteilung in zwei Elementkategorien. Es gibt die Klasse der arithmetischen Bauteile und eine Klasse, die sich mit der Organisation von Daten befaßt. Letztere Gruppe soll im folgenden als erstes geschildert werden.

Theoretische Grundlagen können beispielsweise [44, 27, 43] entnommen werden. Der Entwurf der Bauteile berücksichtigt stets [22, 23].

6.1 Bauteile zum Datenmanagement

Dieser Abschnitt erläutert jene (`C++` -) Klassen, die in die Verwaltung von Daten involviert sind. Vor jeder Element-Beschreibung wird der korrespondierende `C++`-Konstruktor angegeben. Der gesamte Programmcode ist in der Datei `DataManagement.cpp` enthalten.

6.1.1 Klasse `SFSync`

```
CSFSync( const char *pcName, int iWidth = 32 );
```

Der Name leitet sich von 'slow to fast synchronizer' ab. Das Element fungiert als eine Art Brücke, wenn Daten der Breite `iWidth` von einem Bereich `I` eines Systems in einen anderen Bereich `II` transferiert werden sollen, in dem nicht die gleiche Taktfrequenz herrscht. Voraussetzung für **`SFSync`** ist, daß die Frequez am Zielort größer oder gleich der in Bereich `I` ist, so daß Daten schneller entnommen werden können, als sie eintreffen.
Als Spezialversion eines *Fifos* benötigt **`SFSync`** weniger als die Hälfte der Ressourcen von **`AsynFifo`** aus *CHDL*. Die Einsparungen gehen auf Kosten der Funktionalität.
Das Bauteil dient als Verbindungsstück zwischen dem **`PCI_SLAVE`**-Element (siehe Abschnitt

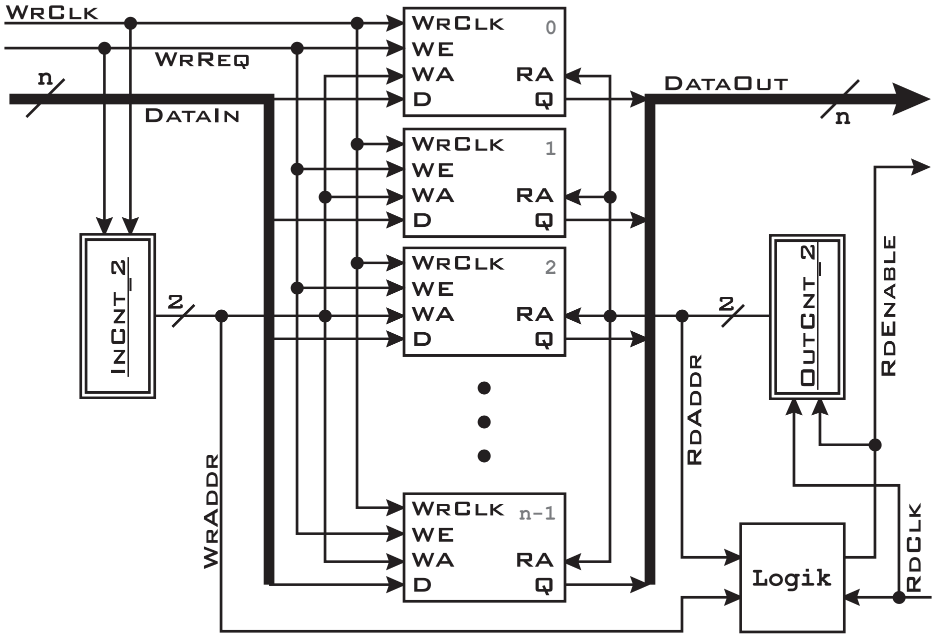

Abb. 6.1: Element **SFSync**. Ein Datum kann gelesen werden, falls die Stände des Eingangs- und des Ausgangszählers unterschiedlich sind. Dies bestimmt **Logik**. **RdClk** muß mindestens die gleiche Frequenz haben wie **WrClk**.

4.1.3 auf Seite 33) und dem *Design*. Zwar können beide Seiten auf eine gemeinsame Frequenz eingestellt werden, sind aber nicht notwendigerweise gleichphasig[1], so daß das Einhalten der *Setup-Zeiten* von ggf. nachgeschalteten *FlipFlops* nicht gewährleistet werden könnte.
Eine Grafik zu **SFSync** zeigt Abbildung 6.1 auf Seite 58. Ein hoher *Pegel* auf Leitung **WrReq** (Schreibanforderung) zeigt das Anliegen eines *gültigen*, n Bit breiten Datums auf **DataIn** an. Es kann in den n *LUTs* gespeichert werden, die je als 16 Bit tiefes RAM-Element verwendet werden. Während das Schreiben synchron zu **WrClk** stattfindet, ist das Lesen asynchron möglich. Die 2 Bit Zähler **InCnt** und **OutCnt** stehen initial auf Null. Sie bestimmen die Schreibadresse **WA** resp. die Leseadresse **RA** der RAM-*LUTs*. Bei jedem **WrReq** zur steigenden *Flanke* von **WrClk** ändert sich der Wert von **WA** / **WrAddr** des Eingangszählers $(0, 1, 2, 3, 0, 1, \ldots)$ und das aktuell anliegende Datum wird gespeichert. Die Teilschaltung **Logik** erkennt anhand von unterschiedlichen Zählerständen, ob ein Datum gelesen werden kann. **Logik** gibt dann in Abhängigkeit von **RdClk** das **RdEnable**-Signal, so daß dieses stabil zur steigenden *Flanke* von **RdClk** einen ganzen Takt gesetzt ist. Mit **RdEnable** macht auch **OutCnt** einen Zählschritt und geht zu der nachfolgenden Leseadresse über.

6.1.2 Klasse `EasyFifo`

```
CEasyFifo( const char *pcName, int iWidth = 32 );
```

Das Element **EasyFifo** realisiert einen einfachen *Fifo*(-Speicher), der maximal 15 Elemente der Bitbreite `iWidth` aufnehmen kann. Ein geschriebener Wert liegt ab dem nächsten *Taktzyklus* am Ausgang **DataOut** an. Im Gegensatz zum asynchronen **AsynFifo** von *CHDL*,

[1] In der Praxis genau gegenphasig.

arbeitet **EasyFifo** mit einem einzelnen Taktsignal **Clk** für Lese- und Schreiboperationen. Zum Einsparen von FlipFlops beispielsweise, ersetzt **EasyFifo** Registerketten bis zu einer Tiefe von 15. Sollte ein 32 Bit-Wert für 12 Takte zwischengespeichert werden, würden

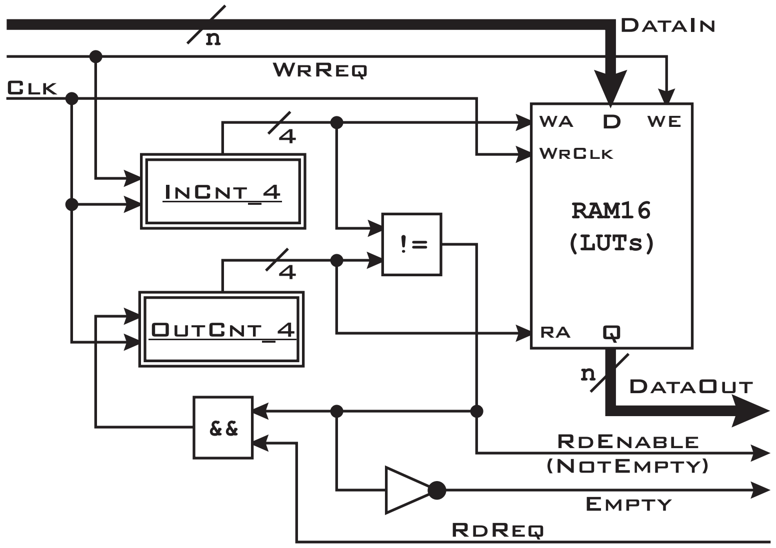

Abb. 6.2: Element **EasyFifo**. Aufbau und Funktionsweise entsprechen im wesentlichen denen von **SFSync**.

12x32=384 *FlipFlops* benötigt. Ein entsprechend dimensionierter **EasyFifo** besteht grob gerechnet aus 44 *LUTs* und acht *FlipFlops*. Für die zusätzlich notwendige Bestimmung der Zeitverzögerung (12 Takte), ab der ein geschriebener Wert wieder ausgelesen werden muß, eignet sich am besten eine parallel geschaltete Kette von 12 *FlipFlops*. In dem dieser Arbeit zugrunde liegenden *Design* gibt es im Berechnungswerk (Abschnitt 4.1.2) die **ValidSig**-Kette, die obenhin diese Funktion übernimmt.
Der Aufbau von **EasyFifo** ist in Abbildung 6.2 skizziert und ist fast mit dem von Komponente **SFSync** identisch. Die einzelnen *LUTs* aus Abb. 6.1 sind hier als **RAM16** zusammengefaßt dargestellt, dessen Adreßbereich zwischen Null und 15 durch die 4 Bit-Zähler **InCnt** und **OutCnt** jetzt vollständig genutzt wird. Der Teil **Logik** von Bauteil **SFSync** hat sich bei **EasyFifo** zu einem Test auf Ungleichheit vereinfacht, weil Lesen und Schreiben synchron stattfinden. Der Zähler **OutCnt** wechselt zur nächsten Adresse, falls **EasyFifo** nicht leer (**NotEmpty**) ist und eine Leseanforderung über **RdReq** vorliegt. Ein möglicher Überlauf kann nicht festgestellt werden.

6.1.3 Klasse `EasyChain`

```
CEasyChain( const char *pcName, int iDepth, int iWidth );
```

Das Bauteil **EasyChain** erstellt variable Registerketten. Die Breite der *Register* sowie die Länge der Kette ist frei definierbar. Auf einen beliebigen Ausgang Q_i einer *Stufe* i kann durch dynamisch erzeugte `PinOutArrays` zugegriffen werden. Es ist möglich, das ClockEnable-Signal **CE** für jede einzelne *Stufe* der Kette festzulegen, um das Propagieren von Daten zu

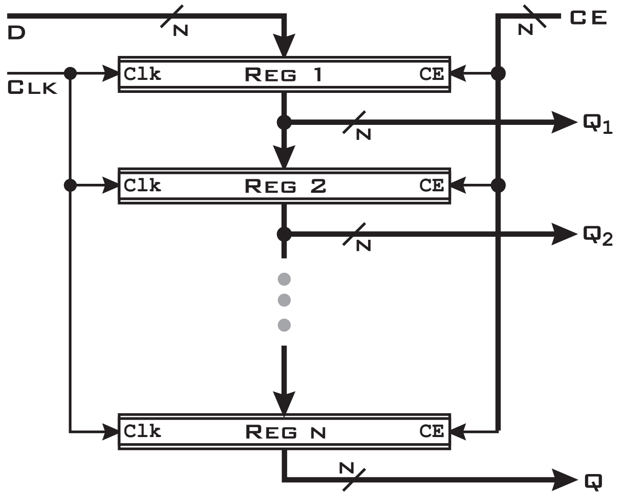

Abb. 6.3: Element **EasyChain** erstellt *Registerstufen* beliebiger Breite m und Tiefe n. Auf den **CE**-Eingang und den Ausgang **Q** jeder *Stufe* der Kette kann individuell zugegriffen werden.

beeinflussen. Über eine Init-Funktion können alle *Register* individuell mit Startwerten initialisiert werden. **EasyChain** wird eingesetzt, um Ketten von *FlipFlops* (*Register* der Breite Eins) ohne viel Schreibaufwand zu generieren und ohne die Flexibilität bzgl. der Programmierung einzelner Kettenglieder zu verlieren. Das Bauteil stellt eine Erweiterung des SWPRM-Elements (Serial-Write-Parallel-Read-Memory) von Torge Rabsch dar.
Nachfolgend soll die Funktionsweise von **EasyChain** anhand von Abbildung 6.3 kurz erläutert werden. Eine *Stufe* übernimmt zur steigenden *Flanke* von **Clk** den Wert an ihrem Dateneingang, falls ihr **CE** gesetzt, d.h. *High* ist. Das Datum eines *Registers* kann über die Q_i einzeln abgegriffen werden.

6.1.4 Klasse `RamIO`

```
CRamIO( const char *pcName, int iRdWaitStates = 3 );
```

Das Element **RamIO** (Abb. 6.4) regelt den direkten Datentransfer zwischen PC(I) und dem RAM auf dem *FPGA*. Es ist zwischen **PCI_SLAVE** und **DPSRAM** geschaltet und kennt zwei Betriebsarten - Lese- und Schreibmodus. Weil beim Schreiben die von dem *PCI-Bus* kommenden Daten durch das **SFSync**-Element (Abschnitt 6.1.1) auf den Designtakt synchronisiert sind, werden diese von **RamIO** lediglich durchgeleitet. Der Zähler **AddrCntr** gibt die jeweilige Zieladresse an.
Im Lesemodus müssen mehrere Dinge beachtet werden:

- Die Daten müssen korrekt und in der richtigen Reihenfolge ausgelesen werden: Es darf kein Wort übersprungen oder doppelt gelesen werden.
- Der *Bus* übernimmt genau dann ein Datum, falls **BusRdRdy** *High* ist. Bleibt das Signal aus, muß das nichtgelesene Wort gepuffert werden.
- Ist der betreffende Zwischenspeicher (*Fifo*) voll, muß der gesamte Lesevorgang seitens des *FPGA* angehalten werden.

- Die Software auf seiten des PCs muß annehmen, daß es sich bei jedem eingehenden Datum um ein Speicherwort handelt, da die Daten nicht typisiert werden können.
- Das verwendte **DPSRAM** hat zusammen mit seiner Registervorstufe eine Verzögerung von drei Takten auf das Anlegen einer Leseadresse (Abschnitt 4.1.4 auf Seite 37).
- Bustakt und Designtakt sind nicht synchron.

Als Lesepuffer wird der **AsynFifo** aus *CHDL* verwendet, der maximal 16 Wörter speichern kann. Nach dem Laden des **AddrCntr** mit der Startadresse wird das Auslesen durch Setzen von Signal **RdMode** gestartet. Drei Takte später liegt das erste Datum aus dem *SRAM* via Leitung **DataFromRam** am *Fifo* an. Parallel geht der Eingang **WE** (für 'write enable') durch die vorgeschaltete Logik mit den drei *FlipFlops* auf *High*. Deren Anzahl wird durch den Parameter `iRdWaitStates` im Konstruktor von **RamIO** gewählt und kann so der erwähnten *Latenz* des RAMs angepaßt werden[2].
Wenn der *Fifo* weniger als fünf weitere Wörter aufnehmen kann, geht der Ausgang **AF** (für 'almost full') auf *High*. Der Zähler **AddrCntr** stoppt das Inkrementieren der Leseadresse.

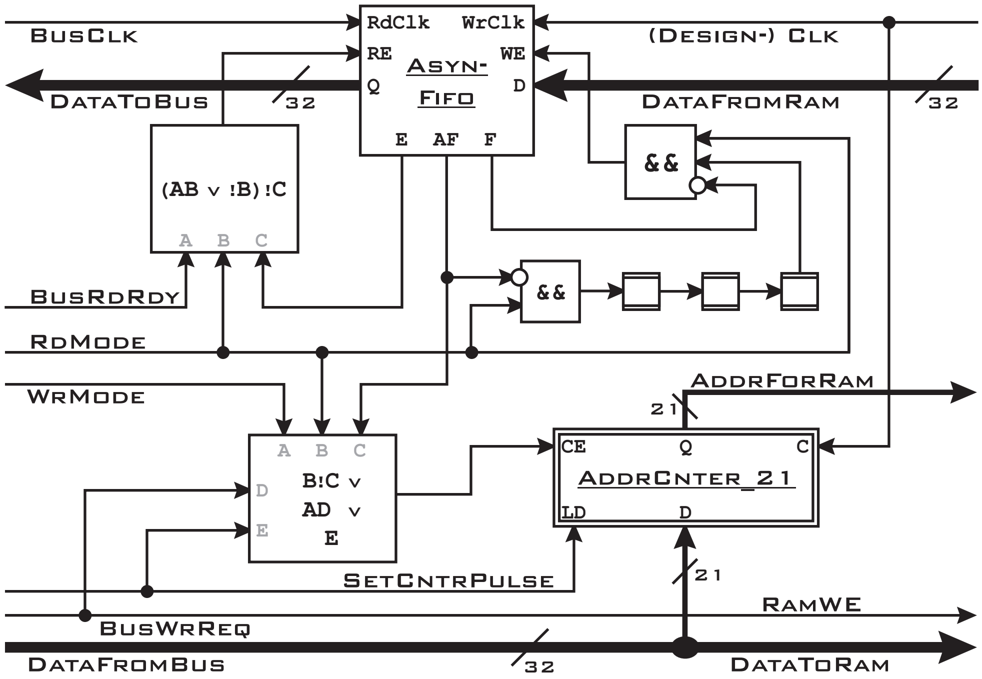

Abb. 6.4: Element **RamIO** steuert den direkten Datenaustausch zwischen *PLX* und *SRAM*.

Aufgrund der *Latenz* von drei Takten werden anschließend noch drei Speicherworte ausgelesen, die der *Fifo* noch aufnehmen kann. Der Leseprozeß ist solange angehalten bis **AF** wieder auf *Low* geht. Ein gesetztes **RE**-Signal (für 'read enable') zeigt **AsynFifo** an, daß das an seinem Ausgang **Q** anliegende Datum ausgelesen wurde.
Die Steuerung von Bauteil **RamIO** erfolgt durch PC-Software und läuft indirekt über das Kontrollwerk ab. Neben **RdMode** und **WrMode** gibt es eine dritte Steuerleitung namens **Set-CntrPulse**, die das Laden des **AddrCntr** mit den unteren 21 Bit der Leitung **DataFromBus**

[2] Falls das Element **RamIO** auf anderen *FPGAs* verwendet werden soll.

veranlaßt.
Die Leitungen **DataFromRam** und **DataToRam** sind im Gegensatz zu den Angaben in der Grafik 6.4 36 Bit breit. Die oberen vier Stellen werden beim Lesen und Schreiben ignoriert resp. zu Null ergänzt.

6.1.5 Klasse `CheckHaz`

```
CCheckHaz( const char *pcName, int iWidth, int iDepth );
```

Das Bauteil **CheckHaz** prüft, ob eine Zahl in einer Menge von Vorgängerzahlen enthalten ist. Es wird im *Design* als Testwerk eingesetzt, um RAW-Hazards (Abschnitt 5.2) rechtzeitig zu erkennen.
CheckHaz verwaltet je eine Kopie von jedem Index aller *gültigen*, gerade in der *Pipeline* befindlichen *Blades*. Die Leitungen **PreResReq** und **PreResIdx** des Rechenwerks sind mit den Eingängen **ValidSig** bzw. **Index** verbunden (Abb. 5.8, S. 55), so daß **CheckHaz** den geometrischen Anteil eines *Blades* erhält, sobald dieser feststeht. Er wird anschließend mit den gepeicherten Indices verglichen, was nur Auswirkungen hat, falls der getestete Index zu einem *gültigen Blade* gehört. Bei einem positiven Ergebnis besteht die Gefahr einer *Datenabhängigkeit* und der Ausgang **Conflict** geht auf *High*. Ist der Index nicht in der Liste seiner Vorgänger enthalten, wird er von **CheckHaz** als jüngster Eintrag in die *Pipe*-Historie aufgenommen.
Der interne Aufbau des Elements ist in Abbildung 6.5 dargestellt und besteht hauptsächlich aus einer Registerkette der Länge `iDepth`, in der die Bladeanteile aller *gültigen* Einträge der *Pipeline* gespeichert sind. Die *Register* enthalten neben den `iWidth` Bit breiten Bladeindex ein zusätzliches Bit I, das zu Eins gesetzt ist, falls das korrespondierende *Blade ungültig* ist. Ein *Register* R_i, $i \in \{0, 1, \ldots, \texttt{iDepth} - 1\}$ übernimmt genau dann ein anliegendes Datum, wenn es ein $j \in \{i, i+1, \ldots, \texttt{iDepth} - 1\}$ gibt, so daß das Bit I_j von R_j gesetzt ist, oder falls Eingang **LastOutSig** *High* ist. Letzterer ist mit dem um zwei Takte verzögerten Signal von Ausgang **ValidSigOut** des Rechenwerks verbunden und ist auf hohem *Pegel*, wenn ein *Blade* die *Pipe* verlassen hat und konsistent im *SRAM* gespeichert wurde. Mit anderen Worten: Das **CE** eines *Registers* ist gesetzt, falls dessen Inhalt oder der Inhalt eines der Nachfolgerregister *ungültig* ist, oder falls ein *Blade* die *Pipe* verlassen hat und keinen *Hazard* mehr erzeugen kann. Die entsprechende Logik repräsentiert die Reihe aus OR-Gattern im unteren Teil von Abbildung 6.5.

Durch das soeben beschriebene Aufrück-Verhalten wird erreicht, daß das Testwerk nur so tief zu sein braucht, wie es maximal *gültige* Einträge in der *Pipe* geben kann. Der Vorteil ist, daß etwa die Hälfte der Elemente, die die Gleichheit von Indices ermitteln, gespart werden kann[3]. Dies reduziert die Anzahl der zu treibenden Leitungen/*Gatter* und verkürzt die Signallaufzeiten. Aus dem gleichen Grund wird der Index von **Index** in zwei *Registern* (**Reg1** und **Reg2**) gleichzeitig gehalten. Ohne diese Maßnahmen wäre der Einsatz dieses Elements u.U. nicht praktikabel gewesen.

Falls alle *Register ungültige* Indices halten und der Index in **Reg1** auch *ungültig* ist, kann die *Pipe* als unbelegt angesehen werden und der Ausgang **Empty** geht auf *High*.

[3]Jedes der Elemente mit dem **==**-Symbol hat $2 \cdot \texttt{iWidth}$ Eingänge!

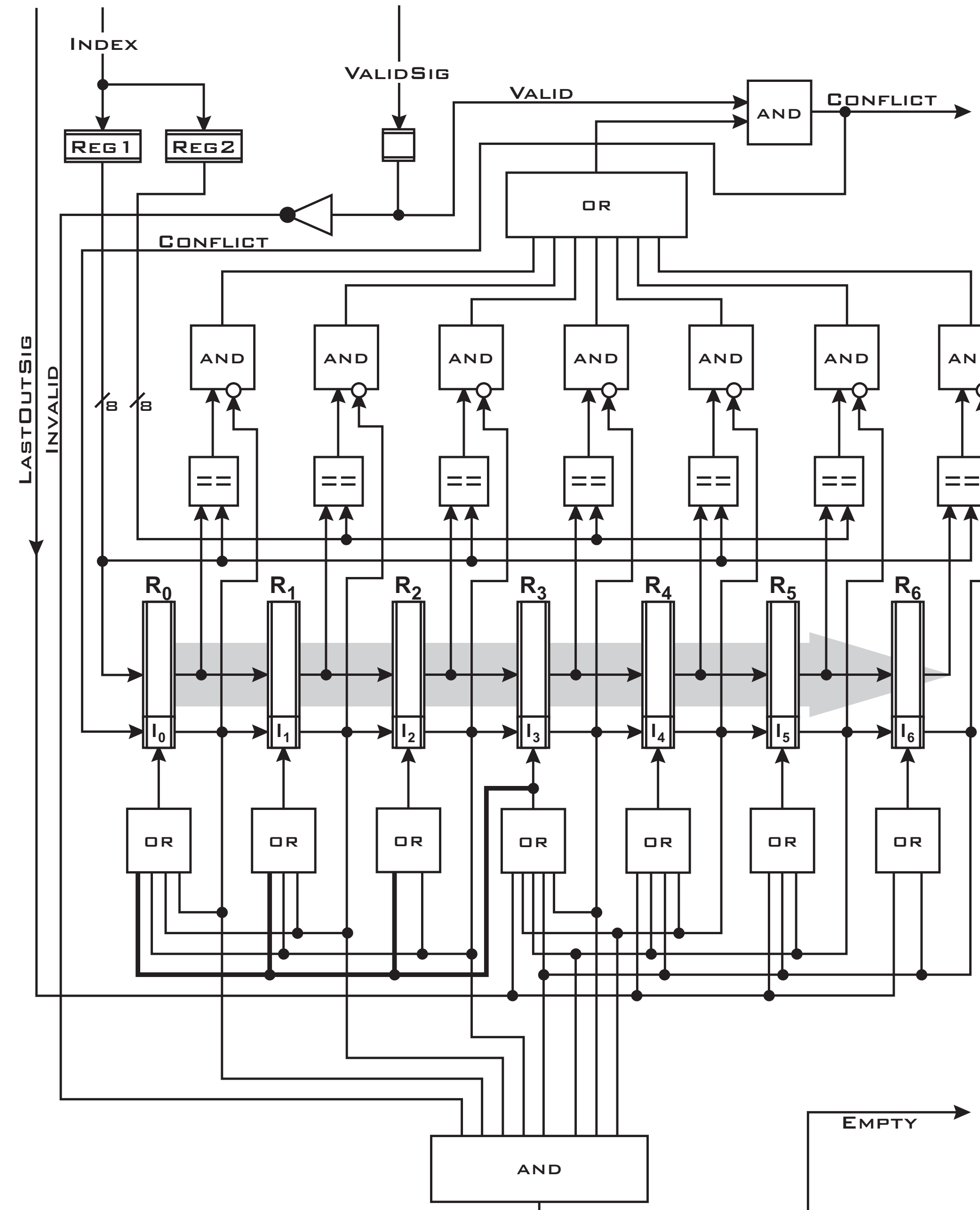

Abb. 6.5: Element **`CheckHaz`**. Es können bis zu sieben (*Blade*-)Indices verwaltet werden. Einige Pfeilspitzen an den Leitungsenden wurden aus Gründen der Übersichtlichkeit weggelassen.

6.2 Arithmetische Elemente

Die folgen `C++`-Klassen implementieren arithmetische Standardbauteile zur Addition, Multiplikation etc. Sie wurden alle für das *Pipeline*-Konzept entworfen und sind daher in ihrem Aufbau fast beliebig variabel [44, 27, 35, 43, 36].

Der Parameter `iStep` im Konstruktor der Elemente bestimmt die Stufigkeit, das ist die maximale Länge einer Kette aus hintereinandergeschalteten *LUTs* zwischen zwei *DFFs* (auch:

Anzahl der *logischen Level*); jedes Bauteil wird im Abstand von `iStep` *LUTs* von Registerketten durchzogen. Wird der Wert immer größer gewählt, ist das generierte Element irgendwann nicht mehr pipelined.

Ein anderer Parameter `iStartLevel` legt die Anzahl der *logischen Level* fest, nach der die erste *Registerstufe* eingefügt werden soll. Dieser Wert muß kleiner als `iStep` sein. `iStartLevel` dient zur Konkatenation von arithmetischen Elementen. Es soll beispielsweise Addierer `Add2` hinter Addierer `Add1` geschaltet werden. Die Stufigkeit des *Designs* sei sechs. Wenn `Add1` nach seiner letzen *Registerstufe* lediglich drei weitere *logische Level* für die Berechnung des Resultats benötigt, kann `Add2` seinen Startlevel auf drei setzen, um eine *Registerstufe* direkt zwischen den Addierern auszulassen.

Alle Komponenten sind zudem in der Breite `iWidth` der Eingangsdaten variabel erstellbar.

Der Programmcode ist in der Datei `ArithmeticLib.cpp` enthalten. Die Datei `GALib.cpp` enthält indes Bauteil **`GeoProd`** aus Abschnitt 6.2.6.

6.2.1 Klasse `Carry look ahead tree adder`

```
CCLATAdder( const char *pcName, int iWidth,
            int iStep, int iStartLevel );
```

Der in diesem Abschnitt beschriebene Addierer entstand aus der Zielsetzung, die Berechnungsgeschwindigkeit großer Summen zu maximieren, ohne die Kosten außer acht zu lassen. Bei dem **`carry look ahead tree adder`** wurden die Mechanismen eines *carry look ahead*-Addierers auf eine Baumstruktur übertragen. Abbildung 6.6 illustriert einen Beispielaufbau. Im folgenden wird, auf vorausgesetzten Kenntnissen bzgl. des *carry look ahead*-Addierers aufbauend, das Prinzip des 'Baumaddierers' erläutert.

Die Kernidee des *carry look ahead*-Addierers besteht in der parallelen Berechnung der Überträge durch die Hilfsvariablen `G` und `P` (für: 'generate carry' und 'propagate carry'). Dadurch kann die Zeit für die Bestimmung des Übertrags an einer beliebigen Stelle, im Gegensatz zu einem seriellen Addierer, stark reduziert werden.
Es seien a_i, b_i, $i \in \{0, 1, \ldots, n-1\}$ die Eingangsvariablen eines n Bit breiten Addierers. Das G_i, P_i, das *Carry* C_i sowie die Summe S_i an Stelle i waren wie folgt definiert:

$$\begin{aligned}
G_i &= a_i b_i \\
P_i &= a_i \oplus b_i \\
C_i &= G_{i-1} \vee P_{i-1}G_{i-2} \vee P_{i-1}P_{i-2}G_{i-3} \vee \ldots \vee P_{i-1}P_{i-2}\ldots P_1 G_0 \\
S_i &= C_i \oplus a_i \oplus b_i \;=\; C_i \oplus P_i
\end{aligned}$$

Bei C_i fällt der letzte Klammerterm $(\vee P_{i-1}P_{i-2}\ldots P_1P_0C_0)$ für diesen Addierer weg, da das C_0 direkt im G_0 mitverarbeitet wird.
Eine direkte Implementierung obiger Formeln hätte zwei Nachteile:

- Starke Belastung der `G`/`P`-*Gatter* durch hohen *Fan-Out*: Ein *Gatter* P_{x-1}, $x \in \{1, 2, \ldots, n\}$ eines n Bit Addierers muß $(n-x)(x-1)+1$ andere *Gatter* treiben. Der Ausgang von $P_{x-1} \mathrel{\widehat{=}} P_{15}$ mit $x = \left\lfloor \frac{n+1}{2} \right\rfloor$ und n=32 müßte beispielsweise mit 241 weiteren Elementen verbunden sein.

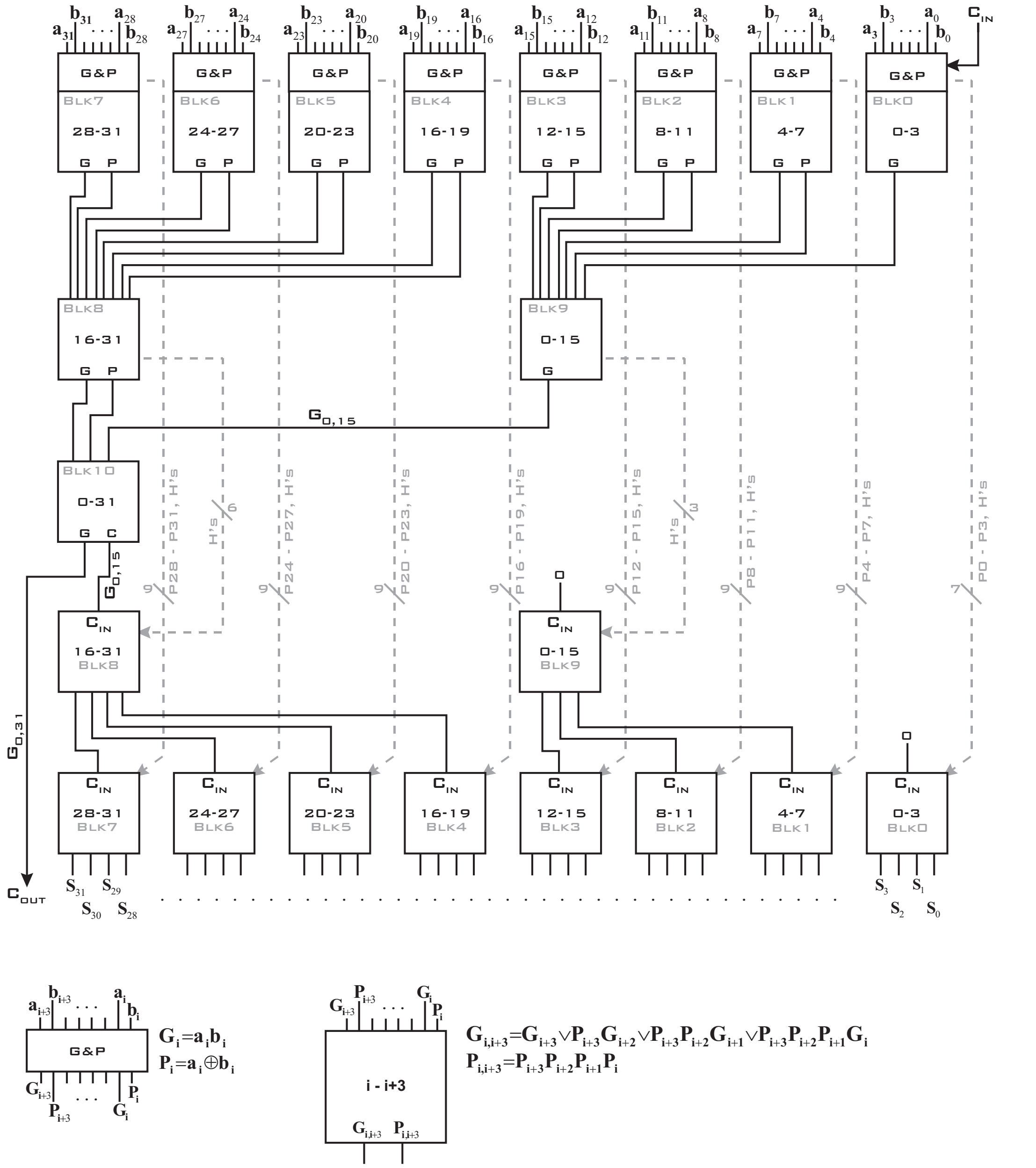

Abb. 6.6: **`Carry look ahead tree adder`**-Prinzip anhand eines 32 Bit breiten Beispielaufbaus. Das Datenvolumen ändert sich mit jeder Ebene um Faktor Vier. Die Daten fließen aus den Blättern (`Blk0` - `Blk7`) in die Wurzel des Baums (`Blk10`) und in die Blätter zurück.

- Irreguläre Strukturen

Das Konzept zur Behebung dieser Probleme ist eine schrittweise Berechnung der `G`'s und `P`'s. Es lassen sich folgende allgemein geltende Rekursionen ableiten:

$$
\begin{aligned}
C_{k+1} &= G_{i,k} \vee P_{i,k} C_i \\
G_{i,k} &= G_{j+1,k} \vee P_{j+1,k} G_{i,j} \\
P_{i,k} &= P_{j+1,k} P_{i,j}
\end{aligned}
$$

mit $i, j, k \in \mathbb{N}, i \leq j \leq k$ und $G_{i,i} = G_i$ bzw. $P_{i,i} = P_i$.

Beispielsweise gilt:

$$\begin{aligned} C_4 &= G_{1,3} \vee P_{1,3}C_1 \\ &= (G_{2,3} \vee P_{2,3}G_{1,1}) \vee P_{1,1}P_{2,3}G_0 \\ &= (G_{3,3} \vee P_{3,3}G_{2,2}) \vee P_{2,2}P_{3,3}G_1 \vee P_1P_{2,2}P_{3,3}G_0 \\ &= G_3 \vee P_3G_2 \vee P_3P_2G_1 \vee P_3P_2P_1G_0 \end{aligned}$$

Auf diese Art und Weise ist es möglich Teilergebnisse zu bilden, die an vielen Stellen wiederverwertet werden können. Bei dem Addierer aus Abbildung 6.6 kann der Wert von $G_{0,15}$ beispielsweise für alle oberen 16 Bit der Summe benutzt werden:

$$\begin{aligned} S_{22} &= P_{22} \oplus \big(G_{21} \vee P_{21}G_{20} \vee P_{21}P_{20}G_{16,19} \vee P_{21}P_{20}P_{16,19}G_{0,15}\big) \\ &= P_{22} \oplus \big(G_{21} \vee P_{21}G_{20} \vee P_{21}P_{20}C_{20}\big) \;=\; P_{22} \oplus C_{22} \end{aligned}$$

wo folgender Term ebenfalls ein Zwischenergebnis repräsentiert:

$$C_{20} = G_{16,19} \vee P_{16,19}G_{0,15} \;=\; G_{16,19} \vee P_{16,19}C_{16}$$

In Abbildung 6.6 werden in den obersten `G&P`-Blöcken zunächst die `G`'s und `P`'s aller Eingangsdatenpaare a_i, b_i, $i \in \{0, 1, \ldots, 31\}$ des dargestellten 32 Bit Addierers berechnet. Der restliche Aufbau besteht aus den Blöcken `Blk0` bis `Blk10`. Die Abbildung 6.7 zeigt einen vollständigen (Basis-)Block[4], aus dem sich alle anderen Blöcke, meist durch Weglassen von Teilstrukturen, erzeugen lassen. Diese Regularität erleichert die Realisierung beliebig dimensionierter Addierer, schafft eine regelmäßige Baumstruktur und vereinfacht die Umsetzung des *Pipeline*-Konzepts. Die Blöcke aus Abbildung 6.6 sind jeweils in zwei Teile aufgetrennt dargestellt, um zu zeigen, daß der Datenfluß jeden Block zweimal passiert. Ein Block erzeugt zunächst aus jeweils (bis zu) vier Generate- und Propagate-Signalen zwei neue Signale `G` bzw. `P` gemäß den o.g. Rekursionen:

$$\begin{aligned} G_{i,m} &= G_{l+1,m} \vee P_{l+1,m}G_{k+1,l} \vee P_{l+1,m}P_{k+1,l}G_{j+1,k} \vee P_{l+1,m}P_{k+1,l}P_{j+1,k}G_{i,j} \\ P_{i,m} &= P_{l+1,m}\, P_{k+1,l}\, P_{j+1,k}\, P_{i,j} \\ \text{mit} \quad & i,j,k,l,m \in \mathbb{N},\ i \le j \le k \le l \le m \end{aligned}$$

Sie dienen erneut als Eingänge nachfolgender Blöcke höherer Ebenen. Der zweite Teil ist die Berechnung der Überträge (Summen bei Basisblöcken), für die das CarryIn $C_{\texttt{in}}$ benötigt wird. Dessen Eintreffen hängt von Blöcken höherer Ebene ab und kann, je nach Stufigkeit der *Pipeline* und der Größe des Addierers, einige Takte dauern. Andere benötigte Signale müssen zur Überbrückung dieser Zeitspanne ggf. in FlipFlop-Ketten zwischengespeichert werden. Sie sind in Abb. 6.6 als grau gestrichelte Leitungen dargestellt.

Die Daten fließen von den Blättern des Baums (`Blk0` bis `Blk7`) zur Wurzel (`Blk10`) und bilden hierbei immer umfassenderere Teilausdrücke. Den größten repräsentiert $G_{0,31}$ als CarryOut. Ab der Wurzel wird die Kaskade in entgegengesetzter Richtung durchlaufen und endet wieder in den Basisblöcken `Blk0` bis `Blk7`, wo die Summen S_0 bis S_{31} erzeugt werden.

[4]Während Basisblöcke Endsummen ausgeben, bestimmen normale Blöcke Überträge.

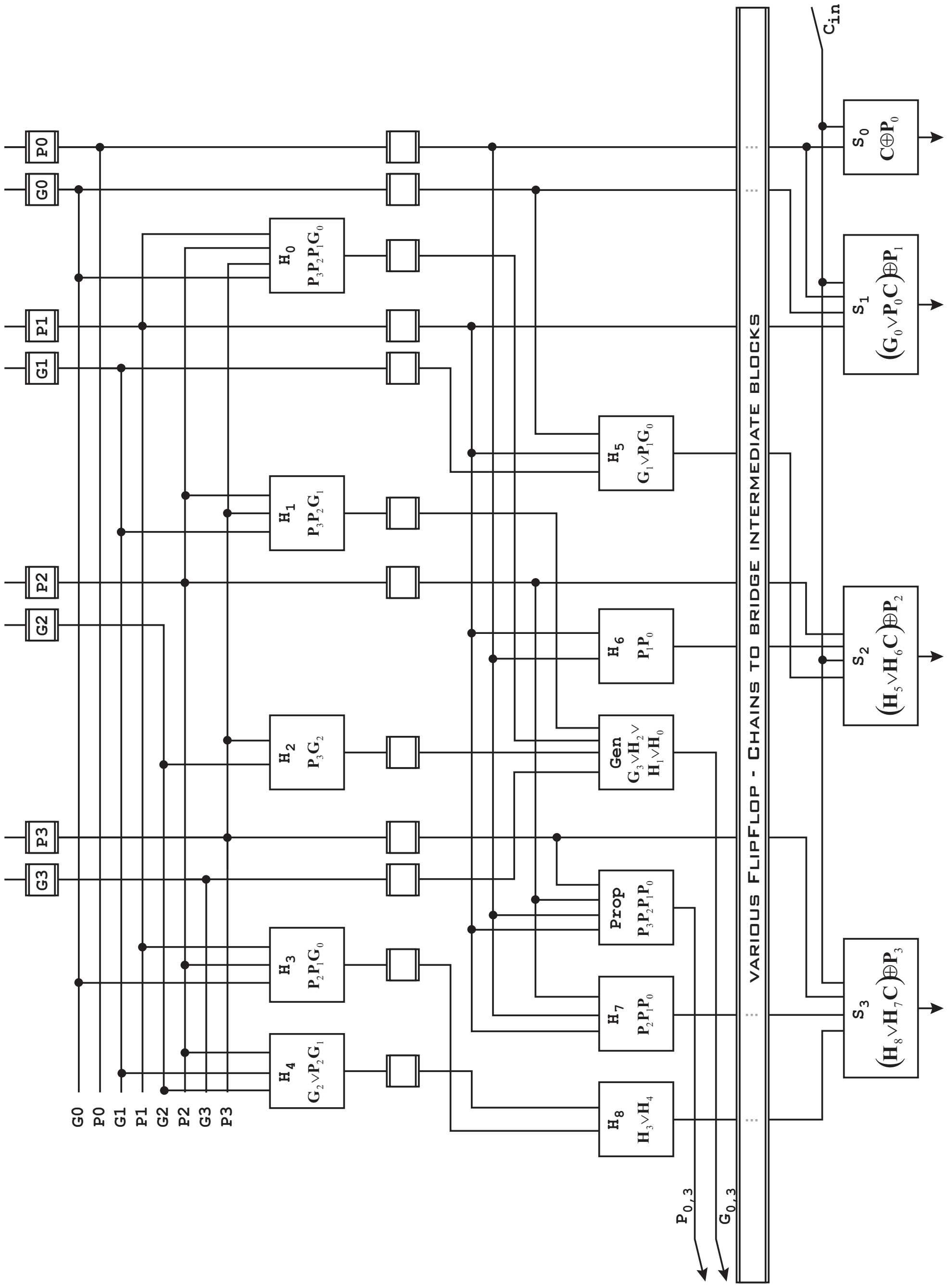

Abb. 6.7: **`Carry look ahead tree adder`**. Aufbau eines 4er-Basisblocks mit `iStep` gleich Eins. Die Hilfsvariablen H_5 bis H_8, G_0 sowie P_0 bis P_3 müssen in *DFF*-Ketten gesichert werden, bis $C_{\mathtt{in}}$ feststeht.

Der **`carry look ahead tree adder`** benötigt pro Bit Breite etwa sieben *LUTs*. Die Anzahl der *logischen Level* beträgt bei einer Breite von n Bit etwa

$$\lceil 3 \cdot \log_4 n \rceil \; .$$

Dies bedeutet, daß ein 256 Bit Addierer gerade halb so schnell ist wie ein 16 Bit Addierer, während ein serieller 256 Bit Addierer 16 mal so langsam wäre wie ein serielles 16 Bit Exemplar.

Der im *Design* verwendete 38 Bit breite Addierer hat acht *logische Level* und 248 *LUTs*. Bei `iStep` = 6 und `iStartLevel` = 4 hat er eine *Registerstufe*, die 108 *DFFs* in Anspruch nimmt.

Typ	Zeit	Ressourcen
Ripple	$O(n)$	$O(n)$
Carry look ahead	$O(\log n)$	$O(n \log n)$
Carry look ahead tree	$O(\log n)$	$O(n)$
Carry skip	$O(\sqrt{n})$	$O(n)$
Carry select	$O(\sqrt{n})$	$O(n)$

Asymptotische Zeit- und Ressourcenkomplexität für unterschiedliche Typen von Addierern. Eine mäßige Investition in Ressourcen führt beim `CLAT`-adder zu einer großen Geschwindigkeitssteigerung.

Da die *CLBs* über eine separate *Carry*-Logik (Abschnitt 3.2) verfügen, können über *CHDL* leicht sehr schnelle, kostengünstige Ripple-adder erzeugt werden. Ein empirischer Geschwindigkeitsvorteil des **`carry look ahead tree adder`**s ergibt sich ab einer Datenbreite von 20 Bit.

6.2.2 Klasse `RAdder`

```
CRAdder( const char *pcName, int iWidth,
         int iStep, int iStartLevel );
```

Der seriell arbeitende **`RAdder`** ist ein Addierer-/Subtrahierer-Bauteil. Einen Beispielaufbau zeigt Abbildung 6.8. Das Element besteht hauptsächlich aus den 2 Bit Volladdierern **`XCADDSUB2`**[5] aus *CHDL*. Diese Komponenten nutzen die schnelle *Carry*-Logik (Abschnitt 3.2, Seite 20), die in vielen *CLBs* enthalten ist, um ripple-adder eine schnelle Übertragsberechnung zu ermöglichen. Für die Logik existieren spezielle, besonders kurze Leitungen auf dem *FPGA*, die die Signallaufzeiten minimieren. Ungerade Datenbreiten müssen durch ein abschließendes **`XCADDSUB1`** Element realisiert werden. **`RAdder`** ist voll pipeline-fähig. Die Schrittweite `iStep=1` ist so festgelegt, daß sich zwischen zwei *Registerstufen* genau ein **`XCADDSUB2`**-Element befindet.
Alle *Carry*-Leitungen, die nicht direkt zwei **`XCADDSUB`**-Komponenten verbinden, erfordern das sog. Importieren oder Exportieren des *Carry*-Signals durch besondere *CHDL*-Elemente. Diese sind in Abbildung 6.8 nicht eingezeichnet.

Der Vorteil des mehrfach im *Design* eingesetzten Bauteils besteht in der geringen Menge an benötigten Ressourcen.

[5] In der Grafik durch **`XCADD2`** abgekürzt.

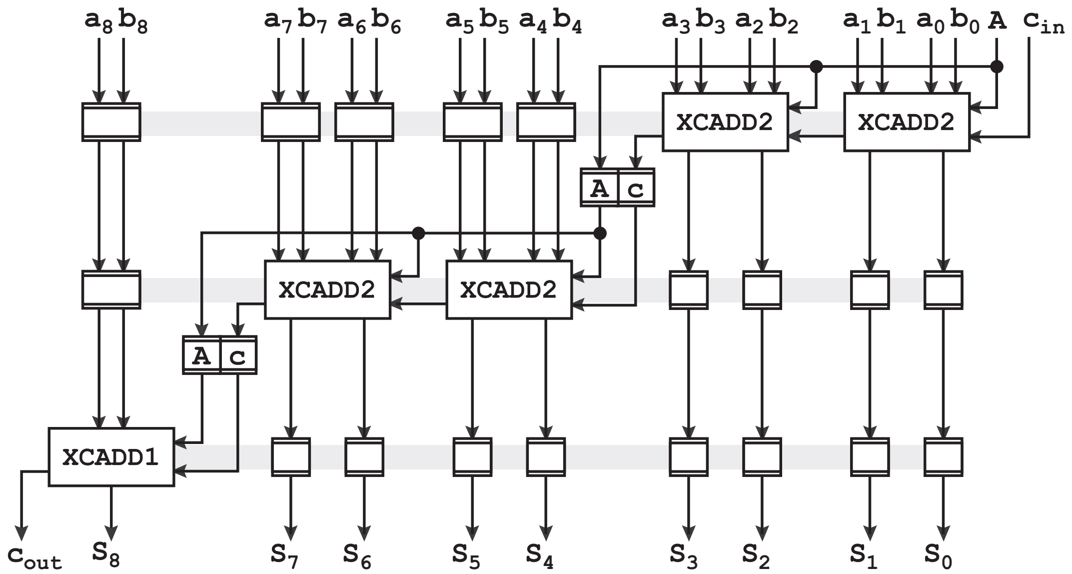

Abb. 6.8: Element **RAdder**. Es ist `iWidth`=9 und `iStep`=2. Am Eingang **A** auf der rechten Seite liegt an, ob addiert oder subtrahiert werden soll.

6.2.3 Klasse `RInc`

```
CRInc( const char *pcName, int iWidth,
       int iStep, int iStartLevel );
```

Die Klasse **RInc** realisiert einen seriellen Inkrementierer/Dekrementierer. Die Struktur des Aufbaus entspricht der von **RAdder**. **RInc** besteht im Gegensatz zu letzterem aus **XCINC-DEC**-Komponenten aus *CHDL*, die ebenfalls auf die schnelle *Carry*-Logik aufbauen. Analog zu **RAdder** gibt es einen Eingang, der festlegt, ob inkrementiert oder dekrementiert werden soll. Element **RInc** eignet sich im *Pipeline*-Betrieb nicht als ladbarer Zähler, weil eine Aktualisierung des Ausgangs nach einem Ladevorgang erst nach *Durchlauf* aller enthaltenen *Registerstufen* eintreten kann.

6.2.4 Klasse `WallaceTreeStruct`

```
CWallaceTreeStruct( const char *pcName,
                    int iWidth,
                    int iStep,
                    int iStartLevel,
                    int iMaxGates );
```

Das Element **WallaceTreeStruct** stellt eine Struktur dar, die schnelles, paralleles Addieren ermöglicht. Sie wird im *Design* als erste von zwei Teilkomponenten eines Multiplizierers (s. Abschnitt 6.2.5) verwendet.

1	0	0	1	x	1	0	1	1
					1	0	1	1
				0	0	0	0	
			0	0	0	0		
		1	0	1	1			
		1	1	0	0	0	1	1

Die Binärzahlen einer Spalte müssen wie bei der schriftlichen Multiplikation von Hand aufaddiert werden. Die Summe der vierten Spalte ergibt hier Null und erzeugt einen Übertrag.

Obwohl 'Wallace tree' eine baumartige Struktur (Abb. 6.10) bezeichnet, die dazu dient, Ziffern aus einer Spalte von n möglicherweise verschiedenen, untereinanderstehenden Zahlen zu addieren, erwartet **`WallaceTreeStruct`** lediglich die zwei n Bit breiten Zahlen L und M als Eingabe. Dies erklärt sich aus der expliziten Verwendung als Vorstufe des Multiplizierwerks, wo $L \cdot M$ berechnet werden soll.
Der erste Teil von **`WallaceTreeStruct`** ist wie obiges Rechenbeispiel vergleichbar gegliedert und soll in gleicher Weise funktionieren. Das korrespondierende, in Abbildung 6.9 dargestellte Gitter verknüpft jedes Bit von L mit jedem Bit von M durch ein UND-*Gatter*.

Der zweite Teil von **`WallaceTreeStruct`** besteht aus der erwähnten Baumstruktur. Die Bits $v_{i,i+j}$, $i, j \in \{0, 1, \ldots n-1\}$ einer Spalte, gemäß Abbildung 6.9, werden zu Gruppen von möglichst drei Elementen zusammengefaßt. *CHDL*-Volladdierer **`VA`** erzeugen aus jeder Dreiergruppe ein Summenbit und ein *Carry*. Zweiergruppen und übrige, einzelne Bits werden durch *CHDL*-Halbaddierer **`HA`** verarbeitet bzw. in eine nächste Ebene aufgeschoben.
In Abb. 6.10 wird am Beispiel von Spalte k=8, $k \in \{0, 1, \ldots, 2n-1\}$ aus Abb. 6.9 die Addition von neun Bits veranschaulicht. Diese generieren in der ersten Ebene beispielsweise drei Summenbits und die drei *Carries* $c_{k,0}$, $c_{k,1}$ und $c_{k,2}$. Für die Verarbeitung in den folgenden Ebenen gilt:

- Jede Ebene erhält ihre Eingadaten aus der vorherigen.
- Überträge werden stets zur nächsthöheren Spalte geleitet.
- In jeder Spalte k werden die Summenbits aus k und die Überträge aus Spalte $k-1$ aufaddiert.

Schließlich erzeugt jede[6] Spalte k zwei Bits - einen Übertrag a_{k+1} und eine Summe b_k.
Ab diesem Punkt ist eine Verwendung von **`WallaceTreeStruct`** zur Bildung der endgültigen Summe $(a_k + b_k)$ nicht mehr sinnvoll, weil dies auf das Prinzip der seriellen Addition hinausläuft.

[6]Bei genauer Betrachtung ergibt sich, daß die Spalten $0, 1, \ldots E-1$ sowie Spalte $2n-1$ kein solches Paar erstellen, wo E die Anzahl der benötigten Ebenen ist.

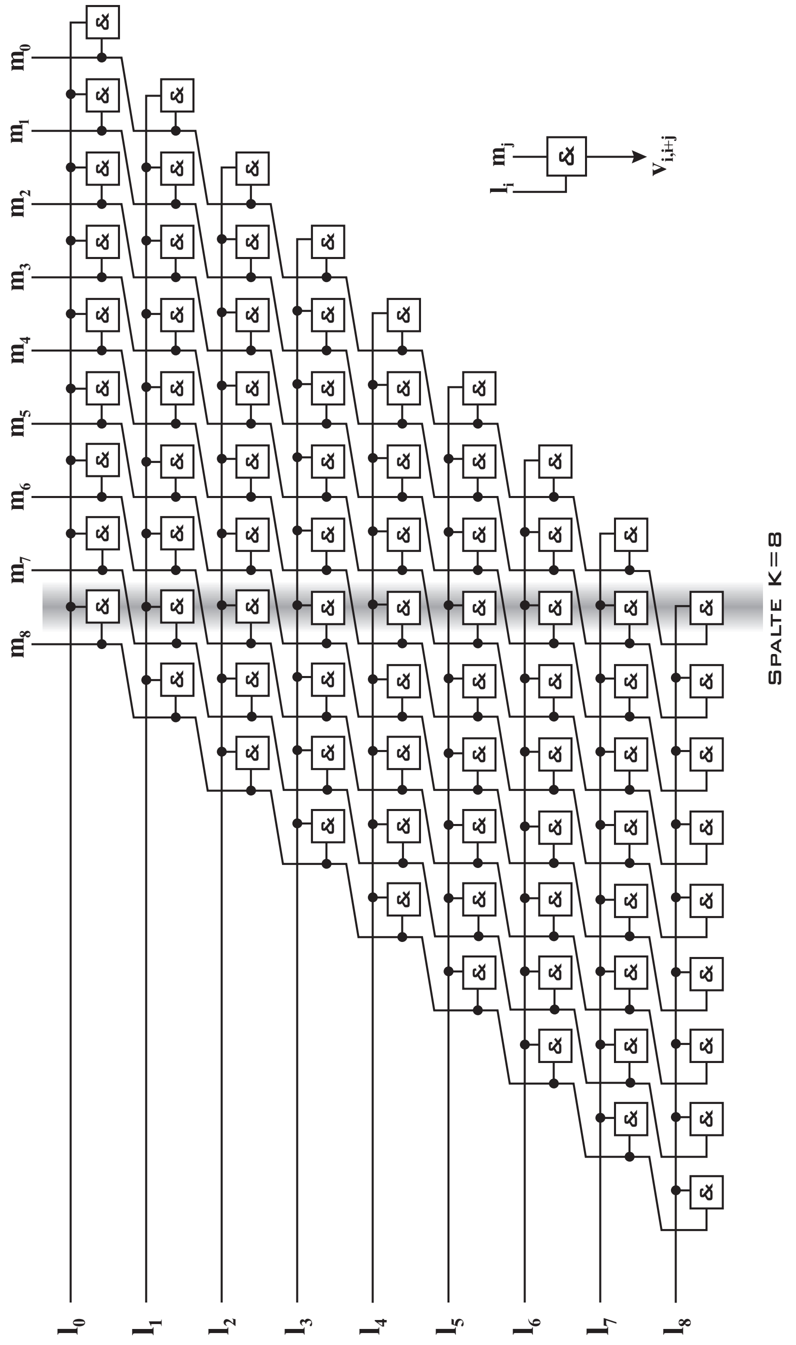

Abb. 6.9: Element **`WallaceTreeStruct`** soll analog zu dem Schema der schriftlichen Multiplikation die (Binär-)Ziffern jeder Spalte aufaddieren.

Laut Abb. 6.9 treibt jedes Bit von L und M jeweils n UND-*Gatter*, was bei großen Breiten n die Leistung mindert. Zur Vermeidung dieses fan-out-Problems kann über den Konstruktor-Parameter `iMaxGates` ein oberer Schwellwert für die Anzahl der zu treibenden *Gatter* festgelegt werden. Die Auswirkungen sollen an dem Beispiel verdeutlicht werden, daß `iMax-Gates=10` ist und n gleich 23 ist. **`WallaceTreeStruct`** setzt anhand dieser Parameter voraus, daß jedes Eingangsbit von L bzw. M dreifach ($\lceil 23 \div 3 \rceil = 3$) vorliegt und verteilt die 23 UND-*Gatter* nach dem Schema 8 - 8 - 7, d.h. möglichst gleichmäßig. Das Vervielfältigen der

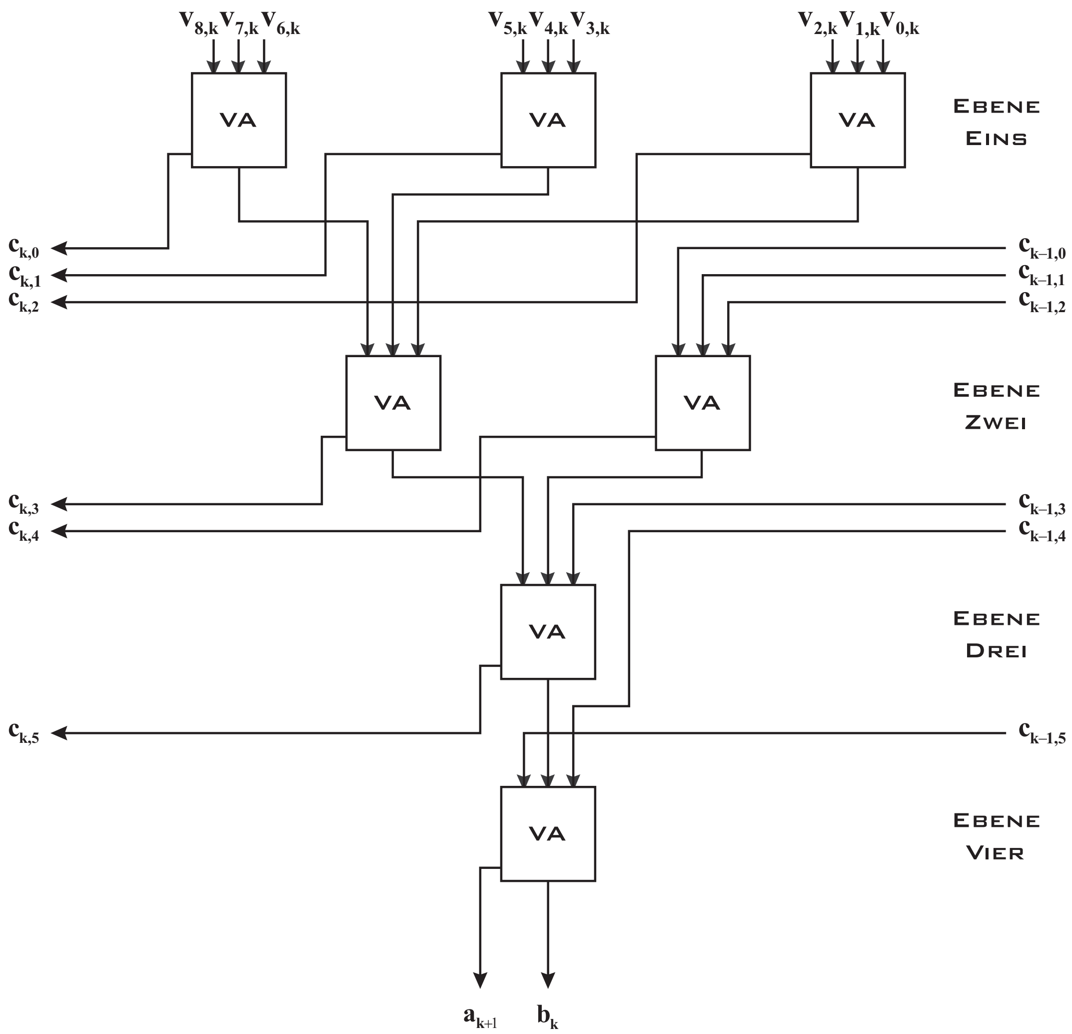

Abb. 6.10: Element **`WallaceTreeStruct`**. Baumstruktur zu Spalte Acht aus Abb.6.9, S.71. Das Problem der Summierung von neun Binärzahlen kann innerhalb von vier Ebenen auf eine einfache Addition heruntergebrochen werden.

Signale muß außerhalb von **`WallaceTreeStruct`** durchgeführt worden sein.

Da die Datenmenge mit jeder Ebene i.A. um ein Drittel abnimmt, gilt folgende grobe Eingrenzung für die Anzahl E der von **`WallaceTreeStruct`** benötigten Ebenen:

$$\mathrm{round}\big(\log_{1,55} n\big) \;\leq\; E \;\leq\; \mathrm{round}\big(\log_{1,45} n\big)$$

Die Anzahl R der in Anspruch genommenen *LUTs* hängt quadratisch von der Breite n der Eingangsdaten ab:

$$R \;\approx\; 2,4 \cdot n^{2,1}$$

Der im *Design* integrierte Wallace tree arbeitet mit einer Datenbreite von 23 Bit. Die Daten passieren acht *Level* bis zum Ausgang. Es werden 1747 *LUTs* benötigt, was 57% der vom gesamten *Design* an *LUTs* verbrauchten Ressourcen ausmacht.

6.2.5 Klasse `Multiplier`

```
CMultiplier( const char *pcName,
             int iWidth,
             int iStep,
             int iStartLevel,
             int iMaxGates,
             bool bRipple );
```

Das Bauteil **`Multiplier`** erzeugt einen auf Geschwindigkeit optimierten Multiplizierer, der das Produkt zweier ganzer, vorzeichenloser, je n Bit breiter Zahlen berechnet.
`Multiplier` besteht im wesentlichen aus Komponente **`WallaceTreeStruct`** und einem

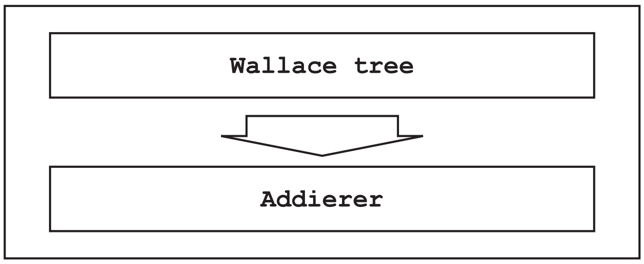

Abb. 6.11: Element **`Multiplier`** ist nur ein Rahmen für die Hintereinanderschaltung von Komponente **`WallaceTreeStruct`** und einem Addierer.

nachgeschalteten Addierer (Abb. 6.11). Zwischen diesen beiden kann sich, abhängig von den *Pipelining*-Parametern, eine *Registerstufe* befinden. Das Element ist ebenfalls für den Aufbau eines Baums zur Vervielfältigung der Eingangssignale zuständig, wie es der Wallace tree u.U. voraussetzt (siehe S.71). Der Baum besteht aus kaskadierten **`BUF`**-Elementen von *CHDL*,

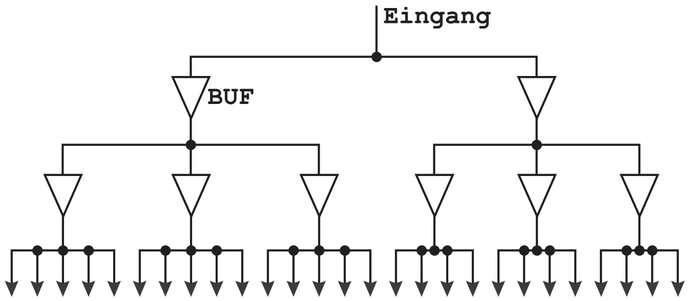

Abb. 6.12: Element **`Multiplier`**. Ein Signal wird durch eine Kaskade von **`BUF`**-Elementen versechsfacht, so daß gemäß Parameter `iMaxGates`=5 bis zu 30 *Gatter* getrieben werden könnten. Diese Baumstruktur kann von Komponente Wallace tree vorausgesetzt werden.

die zur Signalverstärkung dienen. Den Umfang der Vervielfältigung steuert analog zu **`WallaceTreeStruct`** der Parameter `iMaxGates`. Der Wert gibt an, mit wievielen Gattern ein (Eingangs-)Signal verbunden werden darf. Ist `iMaxGates`=5 bei einer Datenbreite von n=27, muß **`Multiplier`** einen zweistufigen Baum (Abb. 6.12) erstellen.

Über den Parameter `bRipple` im Konstruktor kann optional der serielle *Pipeline*-Addierer **`RAdder`** statt dem **`carry look ahead tree adder`** aus Abschnitt 6.2.1 verwendet werden. Ersterer ist für kleine n schneller und benötigt stets weniger Ressourcen. Letzterer eignet sich ab einer Datenbreite von n=20 Bit und wird im *Design* verwendet. Er braucht dort etwa 14% der *LUTs*, die der Wallace tree in Anspruch nimmt.

Das im *Design* verwendete Element **`Multiplier`** benötigt unter den Bedingungen `iStep` = 6 und `iStartLevel` = 0 insgesamt 1995 *LUTs* und 227 *DFFs*. Es werden 16 *logische Level* durch zwei *Registerstufen* getrennt, so daß für eine Multiplikation zweier 23 Bit Zahlen $2,66$ Taktzyklen benötigt werden.

6.2.6 Klasse `GeoProd`

```
CGAGeoProd(const char *pcName, int iVDim, int iStep );
```

Das Bauteil **`GeoProd`** berechnet das geometrische Produkt R zweier *Basisblades* e_μ und e_ν . Die theoretischen Grundlagen und die hier verwendeten Konventionen können Kapitel 2 entnommen werden. Es sei dennoch daran erinnert, daß z.B. $\mathcal{N} := \{1, 2, \ldots, n\}$ mit $n = \dim \mathbb{R}^n$ galt.
Der Programmcode zu Element **`GeoProd`** steht in Datei `GALib.cpp`. Die Dimension n der zugrunde liegenden *Algebra* kann über den Parameter `iVDim` frei gewählt werden. In allen Abbildungen dieses Abschnitts gilt `iStep=2` und `iVDim=5`, also $n = 5$.
Die Bildung des geometrischen Produkts R zweier *Basisblades* $e_\mu = e_4e_1$ und $e_\nu = e_4e_3$ entspricht der Auswertung des Terms $e_\mu\, e_\nu = e_4e_1\, e_4e_3$ zu $R = e_1e_3$, falls $e_4e_4 = -1$ ist.
Für eine einfache technische Repräsentation der *Blades* muß jeder Leitung/jedes Bit einem festen Basiseinheitsvektor zugeordnet werden (siehe Abschnitt 4.1.3, Seite 35), etwa daß `0101b` mit e_3e_1 korrespondiert (Bit 0 mit e_1, Bit 1 mit e_2 etc.). Über die interne Darstellung wird gleichzeitig eine Reihenfolge der Basiseinheitsvektoren impliziert, die e_3e_1 von $R = e_1e_3$ unterscheidet. **`GeoProd`** multipliziert hiervon abweichende *Blades* ggf. mit -1, die sich aus einer ungeraden Anzahl von Vertauschungen der Basiseinheitsvektoren eines Ergebnisblades R ergeben, um diese in die festgelegte Reihenfolge bringen zu können. Das geometrische Produkt $R = e_1e_3$ wird zu `-0101b`, d.h. als $-e_3e_1$ behandelt. Im folgenden sei $e(a_\tau),\ \tau \in \{0, 1, \ldots, n-1\}$ der mit Leitung/Bit $\mathtt{a}_\tau$ assoziierte Basiseinheitsvektor. Weiterhin werden die über **`GeoProd`** verknüpften A- und B-*Blades* durch A bzw. B statt e_μ oder e_ν abgekürzt.
Die Bestimmung der in R enthaltenen Basiseinheitsvektoren ist die vordergründigste Aufgabe. Wegen

$$e_ie_j = \pm 1, \text{ für } i = j \quad i, j \in \mathcal{N}$$

kürzen sich gleiche e_i aus der Darstellung raus, so daß R einer bitweisen *XOR*-Verknüpfung von A und B entspricht. Abbildung 6.13 zeigt die betreffende Schaltung. Die zwei *Registerstufen* dienen als Zwischenspeicher, weil die anderen am geometrischen Produkt beteiligten Komponenten mehr Zeit benötigten und das Ergebnis synchron ausgegeben werden soll.

Eine weitere Aufgabe besteht in der Bestimmung des Vorzeichens von R. Sie gliedert sich in zwei Teile. Zunächst soll der Beitrag der Signatur S (Λ in Kap.2) der *Algebra* betrachtet werden. Abbildung 6.14 zeigt die Umsetzung dieses Parts. Ein gesetztes Bit s_t, $t \in \{0, 1, \ldots, n-1\}$ der Signatur definiert, daß der mit Leitung/Eingang $\mathtt{a_t}$ assoziierte

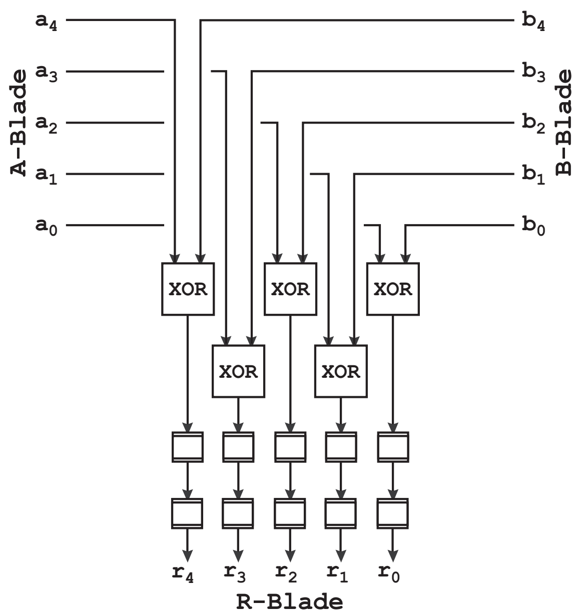

Abb. 6.13: Element **GeoProd**. Diese Schaltung bestimmt die in dem resultierenden *Blade* enthaltenen Basiseinheitsvektoren durch paarweise *XOR*-Verknüpfungen.

Basiseinheitsvektor $e(a_t)$ zu -1 quadriert. Andernfalls ist das Ergebnis $+1$. Tritt bei der Bildung des geometrischen Produkts von A und B eine ungerade Anzahl m von Paaren $e(a_t)e(b_t) \;\equiv\; e_ie_i,\; i \in \mathcal{N}$ auf, die durch $s_t = \mathit{High}$ zu -1 quadrieren, ergibt sich wegen $(-1)^m = -1$ mit $m = 2p+1, p \in \mathbb{N}$ insgesamt ein negatives Vorzeichen.

Damit ein Ergebnisblade die korrekte Form bzgl. der festgelegten technischen *Blade*-Repräsentation (s.o.) hat, und damit

$$e_ie_j = \pm 1,\ \text{für}\, i = j \quad i,j \in \mathcal{N}$$

angewendet werden kann[7], muß zur Bildung des geometrischen Produkts R der jeweilige Anfangsterm AB umgeformt werden. Dies erfolgt ausschließlich über

$$e_ie_j = -\,e_je_i\ \text{für}\, i \neq j \quad i,j \in \mathcal{N}\ .$$

Die sich ergebenden Vorzeichenänderungen sind der zweite Beitrag zum endgültigen Vorzeichen. Dieser Part macht den Großteil von **GeoProd** aus und bestimmt dessen Ebenenanzahl. Abbildung 6.15 zeigt den Aufbau des letzten Teils. Es wird die Anordnung des Ausgangsterms AB simuliert. Die Basiseinheitsvektoren $e(a_t)$ von A werden durch Vertauschungen innerhalb von AB verschoben, so daß folgende Ordnung respektiert wird

$$e(a_{t+1}),\ e(b_{t+1}),\ \mathbf{e(a_t)},\ e(b_t),\ e(a_{t-1})\ .$$

Der Term $AB = e_4e_1\, e_4e_3$ wird z.B. mittels zwei Tauschungen von e_1 zu $e_4e_4e_3e_1$. Das rekursive Verfahren soll anhand von Abb. 6.15 erläutert werden. Die Auswertung der Basiseinheitsvektoren von A erfolgt von links nach rechts, beginnt also mit $e(a_4)$. Für jedes Element $e(a_t)$ des A-*Blades* wird stets vorausgesetzt, daß alle vorherigen Basiseinheitsvektoren

[7]Identische $e_i \in \mathbb{B}$ aus AB müssen über Vertauschungen zueinander gebracht werden.

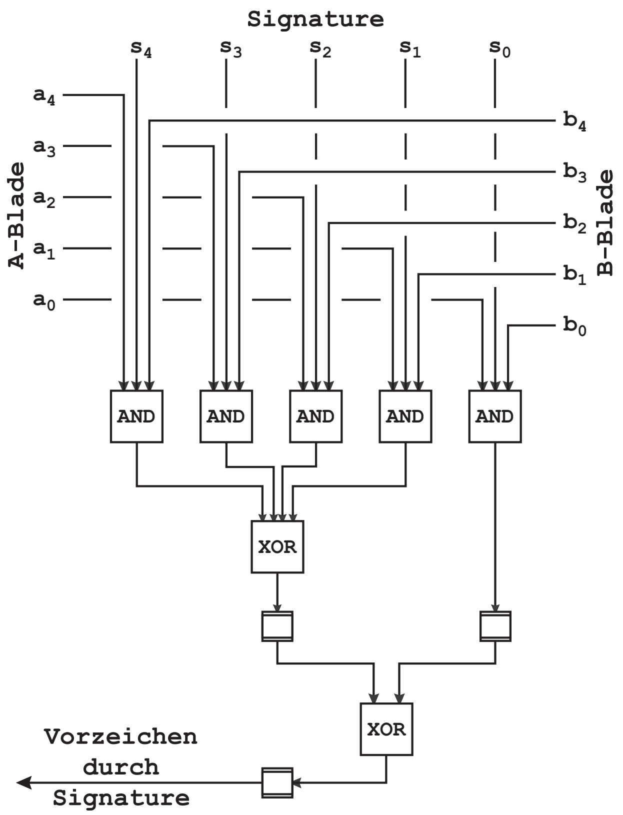

Abb. 6.14: Element **`GeoProd`**. Es wird geprüft, ob die Anzahl der Basiseinheitsvektoren $e_i, i \in \mathcal{N}$ die in beiden *Blades* vorkommen und zu -1 quadrieren, ungerade ist, so daß ein negatives Vorzeichen entsteht.

$e(a_{t-1}), e(a_{t-2}), \ldots, e(a_0)$ bereits einsortiert sind. Zur Einordung von $e(a_t)$ müssen folglich die noch maximal $(n-1)-t$ Basiseinheitsvektoren $e(b_{n-1}), e(b_{n-2}), \ldots, e(b_{t+1})$ übersprungen werden. Der Basiseinheitsvektor $e(a_2)$ müßte mit $e(b_4)$ und $e(b_3)$ getauscht werden, sofern all diese existieren.

Ein mit $\mathtt{a_t}$ verbundenes UND-*Gatter* geht auf *High*, falls $e(a_t)$ in A enthalten ist UND falls die Anzahl der zu überspringenden, existierenden Basiseinheitsvektoren aus B ungerade ist. Dies wird durch eine Kaskade aus *XOR*-Gattern ermittelt, die sich im rechten Teil von Abb. 6.15 befindet. Jeder gesetzte UND-Ausgang stellt eine Multiplikation von $R = AB$ mit -1 dar. Die beiden *XOR-Gatter* im unteren Bereich der Abbildung bestimmen, ob die Anzahl dieser -1-Faktoren ungerade ist. Hierbei wird ein mögliches Vorzeichen durch Quadrieren von Basiseinheitsvektoren zu -1 mit berücksichtigt. Schließlich geht das letzte *XOR* auf *High*, falls bei der Auswertung von AB innerhalb **`GeoProd`** das Vorzeichen -1 entsteht. Es muß mit dem Vorzeichen der skalaren Multiplikation weiterverknüpft werden.

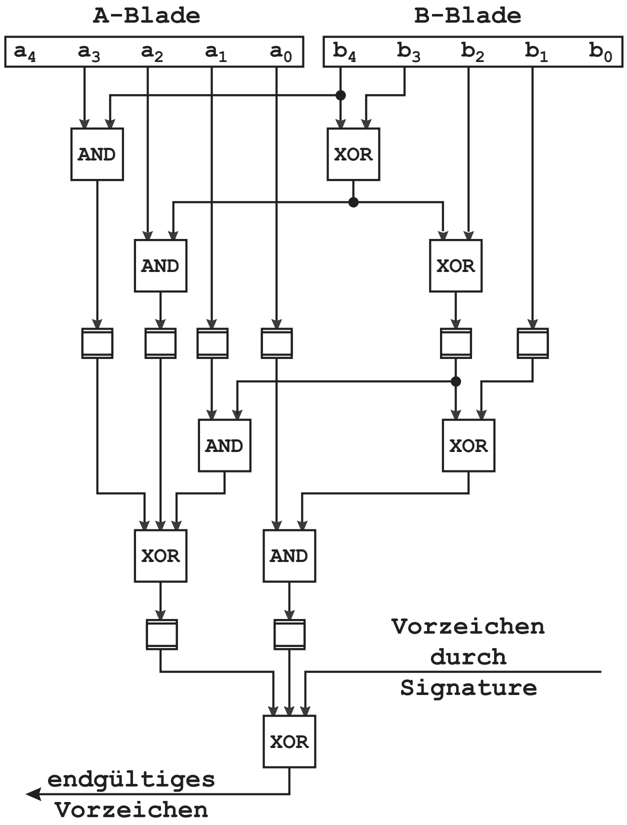

Abb. 6.15: Element **GeoProd**. Bestimmung des endgültigen Vorzeichens aus **GeoProd**. Während der Auswertung des geometrischen Produkts $R = AB$ können skalare Faktoren von -1 erzeugt werden.

Kapitel 7

Ergebnisse

In diesem Kapitel soll eine Bilanz gezogen werden. Zunächst werden die im Rahmen einer *Design*-Durchsatzermittlung gefundenen Testwerte präsentiert, die sich ausschließlich auf die Berechnung des geometrischen Produkts beziehen. Anschließend werden die Ergebnisse mit Daten anderer bestehender Ansätze verglichen, um das Zukunftspotential des entwickelten Prozessors besser einschätzen zu können.

7.1 Leistungsanalyse des Designs

Das Ziel dieses Abschnitts ist die Bestimmung der Designleistung für alle sieben möglichen **GA**-Dimensionen (2D bis 8D). Die Leistungen werden danach mit bestehenden Ansätzen (Softwarelösungen) verglichen.
Im nachstehenden Abschnitt wird zuerst das Testverfahren erläutert. Hierzu zählt die Art und der Umfang der Messungen sowie die Rahmenbedingungen.

7.1.1 Verfahrensweise

Der Begriff 'Leistung' bezieht sich auch den Durchsatz des *Designs*. Dieser sei definiert als die Anzahl abgeschlossener Berechnungen pro Zeiteinheit bei fester Taktfrequenz. Alle Ergebnisse dieses Kapitels beziehen sich auf die Taktfrequenz von 20 MHz. Die Leistung wird jeweils bestimmt für die Berechnungen

1. *Blade*-Einzelmultiplikationen pro Sekunde und
2. *Instruktionen* pro Sekunde
   ('GOPS' - geometrische Operationen pro Sekunde).

Um genaue Resultate für den Durchsatz zu erhalten, wird die Zeit über möglichst viele Berechnungen gemessen. Die Anzahl der *Instruktionen* ist so gewählt, daß die Größe des *SRAMs* (ca. 2 MByte) optimal ausgenutzt wird. Alle Operanden sind zufallsgeneriert. Der Aufbau der benutzten *Instruktionen* soll an einem Beispiel verdeutlicht werden:
Es sollen die Leistungsdaten bzgl. einer siebendimensionalen **GA** auf einem *Design* ermittelt

werden, das für eine Höchstdimension von Acht generiert wurde[1]. Die betreffenden verwendeten Eingabemultivektoren besitzen stets 128 *Basisblades*. Der Test soll über r *Instruktionen* laufen. Eine erste *Instruktion* I_0 verknüpft zwei *Multivektoren*. Es folgen $\lfloor (r-1)/2 \rfloor$ *Instruktionen*, deren A-Operand das Ergebnis von I_0 ist und deren B-Operand vom PC geladen ist. Bei weiteren $\lceil (r-1)/2 \rceil$ ist es umgekehrt, so daß *Multivektor* B dem Ergebnis von I_0 entspricht und *MVA* aus den Eingabedaten seitens des PC stammt.
Es gehen r *Instruktionen*, die r Ergebnismultivektoren à 256 Wörter erzeugen, einher mit $r+1$ Eingabemultivektoren à 128 Speicherwörtern. Das *SRAM* stellt grob 500.000 Speicherstellen à 4 Bytes zur Verfügung, so daß r etwa zu 1200 gesetzt werden könnte, um den Speicher des *SRAMs* gut zu nutzen. Insgesamt würden $1200 \cdot 128^2 = 1200 \cdot 16384$ Blade-Einzelmultiplikationen ausgeführt werden.

Jedes Meßergebnis in den nachstehenden Abschnitten ist über neun Meßwerte gemittelt. Es wurden jeweils drei Meßreihen à drei Messungen durchgeführt: Die Einzelwarscheinlichkeit, daß ein *Basisblade* eines *Multivektors* einen nullwertigen Skalaranteil besitzt, wurde für die Dreiergruppen zu $1/8$, $1/32$ und $1/256$ festgelegt, um verschiedenen Zusammensetzungen der Eingabedaten gerecht zu werden. Im Schnitt sind ca. 5 % aller Elemente der *Multivektoren* nullwertig.

Es wurden für jede Dimension die vier Berechnungsmodi 'Standard', 'Skip', 'FS' und 'FS & Skip' getestet. 'Skip' steht für den in Abschnitt 5.1 erklärten Mechnismus, bei dem ein *Durchlauf* abgekürzt werden kann, wenn sein betreffendes B-*Blade* nullwertig ist. 'FS' ist das Kürzel für 'fail-save' und steht für das im Falle eines aufgetretenen Konfliktes angewandte Verfahren, das in Abschnitt 5.3 beschrieben wurde. Die Kombination 'FS & Skip' repräsentiert den Betrieb unter Anwendung aller Optimierungen, während 'Standard' garantiert, daß kein Konflikt auftritt, weil nach jedem *Durchlauf* gewartet wird, bis die *Pipe* leer ist. Eine leistungssteigernde Maßnahme existiert auch im Standard-Modus und kann nicht deaktiviert werden - das Äquivalent von Skip für A-*Blades*: Ist der Skalaranteil eines A-*Blades* nullwertig, wird der korrespondierende *Pipe*-Eintrag als ungültig erklärt und es wird kein **`PreResReq`** gestellt, so daß ein *Taktzyklus* eingespart wird (Seite 46).

Für die Messungen wurden zwei *Designs* herangezogen. Die Dimensionen 7D und 8D wurden auf einem 8D-*Design* getestet, während die Dimensionen 2D bis 6D auf einem 6D-*Design* getestet wurden. Da Letzteres nur $2^6 = 64$ Speicherwörter für jeden Ergebnismultivektor reserviert, statt 256 wie das 8D-*Design*, passen mehr *Instruktionen* in das *SRAM*, was genauere Messungen ermöglicht. Hochrechnungen vom 8D- auf das 6D-*Design* belegen, daß die Resultate vollkommen identisch wären.

Die Zeitmessung erfolgt mit der `C++`-Funktion `ftime()`, die die aktuelle Zeit unter Angabe von Millisekunden liefert. Da der *FPGA* unabhängig arbeitet, sind Einflüsse des PC-Systems[2] auf die Meßergebnisse weitestgehend auszuschließen.

[1] Da das *Design* keine Kenntnis von der Dimension der geladenen Eingabedaten besitzt, reserviert es implizit $2^8 = 256$ Speicherwörter für jeden Ergebnismultivektor, von denen in diesem Beispiel nur die unteren 128 genutzt werden (siehe Seite 35).

[2] AMD Athlon 900 MHz, 256 MB RAM, SUSE Linux, Kernel 2.2.16

7.1.2 Resultate der Leistungsmessung

Die folgende Tabelle zeigt die Ergebnisse der Messungen.

Dimension	2D	3D	4D	5D	6D	7D	8D
Ausgeführte Instruktionen	7000	6000	5000	5000	3000	1200	1000
Multi./Instr.	16	64	256	1024	4096	16384	65536
Standard	30 ms	71 ms	187 ms	619 ms	1323 ms	2020 ms	6594 ms
nur SKIP	30 ms	71 ms	186 ms	615 ms	1307 ms	1995 ms	6479 ms
nur FS	24 ms	53 ms	136 ms	516 ms	1209 ms	1922 ms	6392 ms
FS & SKIP	24 ms	53 ms	136 ms	511 ms	1195 ms	1897 ms	6308 ms
Relativer Durchsatz	124 %	134 %	137 %	121 %	111 %	107 %	105 %

Tab. 7.1.: Ergebnisse der Leistungsanalyse. 'FS' steht für den fail-save-Mechanismus (Seite 54), während 'SKIP' den Einsatz des in Abschnitt 5.1 beschriebenen Verfahrens bezeichnet.

Die Zeile 'Relativer Durchsatz' obiger Tabelle 7.1. setzt das Standard-Verfahren in Relation zu 'FS & Skip' und spiegelt die gewonnene Verbesserung wieder, indem die Standardwerte zu 100% gesetzt werden. Die Diagramme in den Abbildungen 7.1 (*Blade*-Einzelmultiplikationen) und 7.2 (GOPS) stellen ebenfalls die Beziehungen zwischen diesen Modi heraus. Nachstehend soll zunächst analysiert werden, welche Tragweite dem Skip beigemessen werden muß.

Tabelle 7.1. zeigt, daß die Anwendung von Skip mit steigender Dimension zwar an Bedeutung gewinnt, aber dennoch keinen signifikanten Einfluß auf die Leistung hat. Der Grund hierfür ist, daß mit vollbesetzten *Multivektoren* gearbeitet wird, deren Basisblades fast durchweg einen Skalaranteil ungleich Null besitzen. Bei dem Teil des Testvorgangs, bei dem *Multivektor* B das Ergebnis von *Instruktion* I_0 ist, reduziert sich zusätzlich die Warscheinlichkeit auf ein nullwertiges B-*Blade* zu treffen, weil der I_0-Ergebnisvektor als Produkt zweier *Multivektoren* verhältnismäßig weniger Null-*Blades* enthält.
Die schwache Verbesserung mit steigender Dimension ist darauf zurückzuführen, daß der absolute Zeitgewinn im Falle eines Skips größer wird. Ein Skip dauert fünf Takte, so daß bei 4D höchstens 10 von 16 Takten gespart werden, während bei 8D maximal 250 von 255 Takten gewonnen werden. Bei 3D und 2D wird Skip niemals eingesetzt, da die erste Bedingung für einen Skip von Seite 46 kategorisch nicht erfüllt ist.
Der positive Einfluß von Skip tritt in Abbildung 7.1 deutlicher hervor: Das *Design* arbeitet mit 20 MHz und kann alle zwei Takte ein Ergebnis produzieren, da für jedes Resultat

mindestens zwei obligatorische Lesezugriffe (A-*Blade* und `PreResult`) durchgeführt werden müssen. Die theoretische Obergrenze für den Durchsatz liegt daher bei 10.000.000 *Blade*-Einzelmultiplikationen pro Sekunde. Sie kann nur mittels Skip durchbrochen werden, da es den Durchsatz steigert.

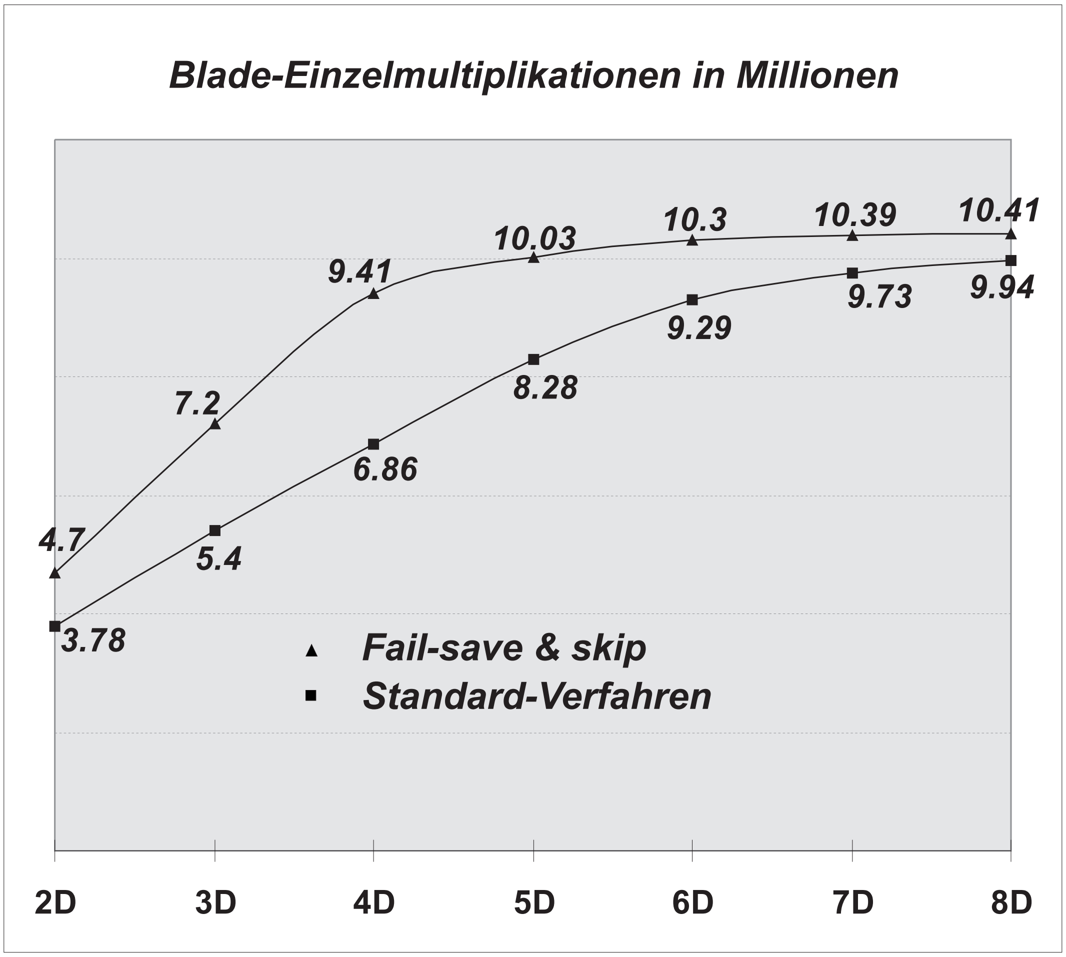

Abb. 7.1: *Blade*-Einzelmultiplikationen pro Sekunde in Millionen bei 20 MHz.

Der relative Durchsatz ist Gegenstand der Diagramme in den Abbildungen 7.1 und 7.2, in denen jeweils die Durchsatzleistungen des Standard- und des 'FS & Skip'-Modus aufgetragen sind.
Der relative Durchsatz erreicht gemäß Tab. 7.1. bei 4D ein Maximum von 137% der Standardleistung. Abbildung 7.1 visualisiert dies durch die Anzahl der *Blade*-Einzelmultiplikationen pro Sekunde. Mit zunehmender Dimension wächst die Anzahl der *Basisblades* der *Multivektoren* (. . . , 64, 128, 256), so daß ein Wechsel des *Durchgangs* den Durchsatz, der sich bereits im Standardmodus stetig der 10.000.000-Marke nähert, immer weniger beeinflußt. Weiterhin resultiert ein Absinken der Wahrscheinlichkeit für das Auftreten eines Konfliktes während eines Durchgangswechsels. Insgesamt reduziert sich mit zunehmender Dimension das Steigerungsvermögen des Durchsatzes mittels 'FS & Skip'.
Mit abnehmender Dimension verkehren sich die Bedingungen in das Gegenteil - das *Design* ist stark vermehrt mit Durchgangswechseln beschäftigt und die statistische Warscheinlichkeit eines Konflikts nimmt zu. Die Zeit für einen Durchgangswechsel könnte auf Null

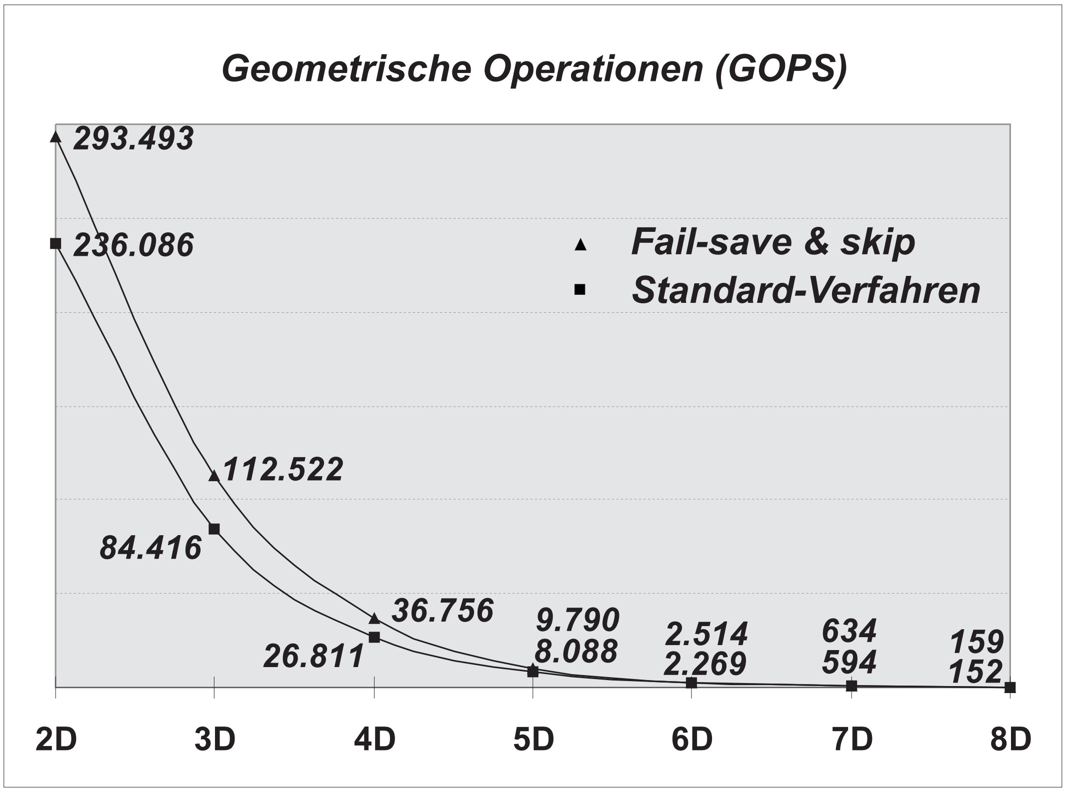

Abb. 7.2: Geometrische Operationen (*Instruktionen*) pro Sekunde bei 20 MHz.

reduziert werden, falls das Problem der *Datenabhängigkeiten* nicht existieren würde. Ein entsprechend (einfacher) konzipiertes *Design* würde für alle Dimensionen mit 10.000.000 Multiplikationen pro Sekunde arbeiten. Dies zeigt anhand von Abbildung 7.1, daß die häufigen Durchgangswechsel bei 2D trotz des betriebenen Aufwands mehr als 50 % der Rechenzeit in Anspruch nehmen.

Weitere Resultate

In diesem Abschnitt soll auf die benötigten Zeiten für Speichertransfers PC ⇔ *SRAM* eingegangen werden. Im Laufe einer Anwendung des *FPGAs* finden gemäß Abschnitt 4.2 auf Seite 40*ff* der Reihe nach drei wesentliche Übertragungen statt:

1. Schreiben der *Instruktionen* in das *SRAM*,
2. Schreiben von Nullen[3] zur Initialisierung des Teils des *SRAMs*, in dem voraussichtlich Ergebnismultivektoren gespeichert werden und
3. Auslesen der Ergebnismultivektoren.

Punkt Eins und Zwei stellen die Eingabedaten dar, von deren Datenvolumen die Nullen aus Punkt Zwei einen Anteil zwischen 30 % und 95 % einnehmen können. Der Wert hängt

[3]Es hat sich gezeigt, daß die *PCI*-Übertragung von ausschließlich gleichen Werten, z.B. Nullen, im Vergleich zu willkürlichen Daten mit der doppelten Transferrate abläuft.

von den *Instruktionen* ab, die die Belegung des *SRAMs* bestimmen. Er betrug während der Leistungsanalyse im Schnitt 69 % und machte etwa 26 % der gesamten Transferzeit aus. Die Datenmenge von Punkt Zwei und Drei ist identisch.

Dimension	2D	3D	4D	5D	6D	7D	8D
Transfers	634 ms	560 ms	497 ms	563 ms	429 ms	548 ms	625 ms
Operation	24 ms	53 ms	136 ms	511 ms	1195 ms	1897 ms	6308 ms
Prozent an Gesamt	3,7 %	8,6 %	21,5 %	47,6 %	73,6 %	77,6 %	91,0 %

Tab. 7.2.: Vergleich der benötigten Zeiten für Rechenoperationen und Speichertransfers. Die reine Rechenzeit macht erst ab 5D den Großteil der Gesamtzeit aus.

Um den Beitrag der Datenübertragungsdauer an der Gesamtleistung bemessen zu können, soll Tabelle 7.2. herangezogen werden. Sie gibt alle erfaßten Zeiten wieder und nennt jeweils den Anteil der Operationszeit an der Gesamtzeit. Intuitiv betrachtet nimmt dieser frühestens ab 6D einen akzeptabelen Umfang an.

Die ermittelte Transferrate liegt etwa bei 1.100.000 Wörter à 32 Bit pro Sekunde. Wenn die Möglichkeit bestünde, die z.Zt. verwendeten *PCI*-Einzeltransfers durch effektive *Burst*- oder DMA-Übertragungen zu ersetzen, könnte der Durchsatz mindestens um den Faktor 20 gesteigert werden. Für Faktor 30 müßten unter Umständen kleine Modifikationen am *Design* vorgenommen werden, da das eingesetzte `DPSRAM` nur bis 20 MHz zuverlässig arbeitet (s.S. 36). Der oben genannte Punkt Zwei könnte komplett entfallen, falls das *Design* so erweitert werden würde, daß ein einzelner Befehl seitens des PCs die Initialisierung des *SRAMs* zu Null veranlassen würde. Der Befehl könnte zu beliebiger Zeit, jedoch vor Aufnahme des Rechenbetriebs, gegeben werden. Zusammengenommen wäre es möglich, die Zeiten für die Datenübertragung von etwa einer halben Sekunde auf bestenfalls 20 ms zu verkürzen.

7.1.3 Vergleich mit anderen Ansätzen

In dem nachstehenden Teil dieses Kapitels soll der Versuch unternommen werden, die Leistung des vorgestellten **GA**-Koprozessors in Relation zu anderen vorhandenen Lösungen zu setzen. Bei den zwei Vergleichskandiaten `CLU` und `Gaigen` handelt es sich um reine Softwarerealisierungen, da eine Hardwareentsprechung nicht verfügbar war. Es lagen nur Referenzdaten für die Dimensionen 3D, 4D und 5D vor.
Das Softwarepaket `CLU` ('Clifford algebra Library and Utilities') von Dr. C. Perwass gründet auf einer entwickelten `C++`-Bibliothek, die geometrische Algebra implementiert. Das enthaltene Programm `CLUCalc` dient zur Berechnung / Visualisierung von geometrischen Operationen wie z.B. das geometrische Produkt.
Das Programm `Gaigen` ('geometric algebra implementation generator') von D. Fontijne generiert `C++`-Softwarebibliotheken, die jeweils eine einzelne **GA** bestimmter Dimension und Signatur implementieren. Das Entwicklungsziel war es, den erzeugten Code möglichst

schnell und auch speichereffizient zu machen. Wie `CLU`, stellt auch `Gaigen` eine umfangreiche Funktionalität zur Verfügung, zu der ebenfalls das geometrische Produkt zählt.

Dimension	3D	4D	5D
FPGA	112.522	36.756	9.790
CLU	2.380	1.120	350
Gaigen	75.457	26.277	6.104

Tab. 7.3.: Vergleich der Durchsatzleistungen in GOPS. Das *Design* wird den auf 20 MHz runterskalierten Programmen `CLU` und `Gaigen` gegenübergestellt.

Bevor die Ergebnisse aus Tabelle 7.3. verinnerlicht werden, muß auf zwei Dinge aufmerksam gemacht werden.
Zum einen werden die reinen Operationszeiten verglichen, um die großen Unsicherheiten bezüglich der (maximal machbaren) Übertragungszeiten PC ⇔ *SRAM* auszuklammern. Eine abwechselnde Verwendung dreier *FPGAs* könnte den folgeschweren Einfluß der Speichertransfers möglicherweise eleminieren (siehe Fazit, S.91).
Während die Softwarefraktion auf 1,5 GHz Maschinen lief, betrug die Taktfrequenz des **GA**-Koprozessors lediglich 20 MHz. Um eine grundsätzliche Aussage über die Durchsatzleistung machen zu können, war es notwendig die Daten von `CLU` und `Gaigen` auf 20 MHz runterzurechnen.

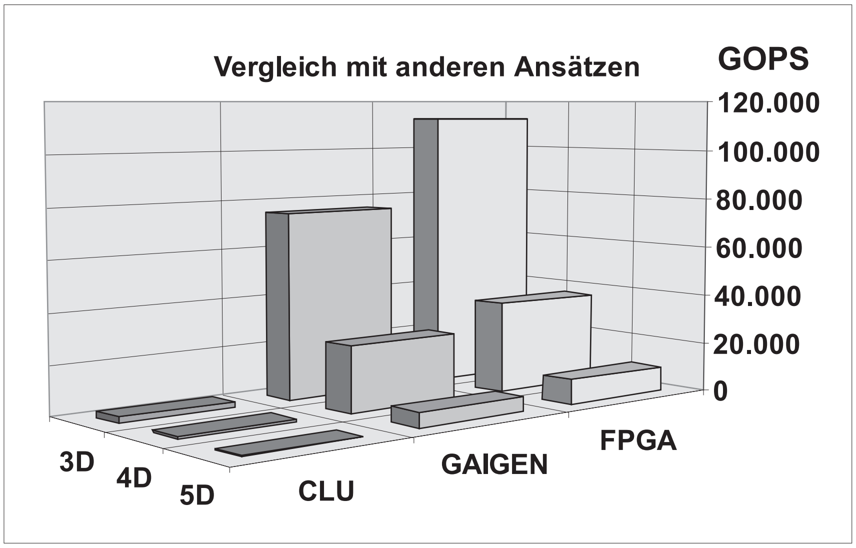

Abb. 7.3: Vergleich der Durchsatzleistungen als Visualisierung von Tabelle 7.3.

Absolut gesehen, d.h., ohne die gerade genannten ’Schönheitskorrekturen’, sind die Softwarerealisierungen wesentlich schneller als die Hardware. Ein Vergleich bei 20 MHz ergibt indes, daß das *Design* etwa $1,5$ mal schneller wäre als `Gaigen`. Es arbeitet ferner durchschnittlich um den Faktor 35 effizienter als `CLU`.

Kapitel 8

Zusammenfassung, Fazit und Ausblick

In dem nachfolgenden Abschnitt wird eine Zusammenfassung aller inhaltlichen Aspekte dieser Arbeit gegeben. Im Anschluß soll ein Fazit 'Für' und 'Wider' abwiegen, um eine abschließende Beurteilung zu erbringen.

Im letzten Teil dieses Kapitels wird der Abschnitt Ausblick weitere mögliche Entwicklungen auf dem Gebiet der *FPGA*-Programmierung bzw. der Nutzung spezialisierter Hardwarekomponenten aufzeigen.

8.1 Zusammenfassung

Das Thema dieser Arbeit war es zu erforschen, ob und bis zu welcher funktionellen Komplexität es möglich ist, einen Koprozessor für geometrische *Algebra* **GA** auf einem *FPGA* zu erzeugen. Es sollte zudem festgestellt werden, inwiefern eine solche Hardwarerealisierung Geschwindigkeitsvorteile gegenüber konventionellen Softwarelösungen bringt.

Anforderung an den Prozessor war es, das geometrische Produkt **GP** von *Multivektoren* berechnen zu können. Um eine nachträgliche Hinzunahme von Operationen wie das innere und äußere Produkt zu gewährleisten, sollten zudem die benötigten Grundstrukturen in dem Koprozessor fest integriert sein.
Die maximale Dimension der zu implementierenden **GA** wurde auf 2^8 beschränkt. Die Basiseinheitsvektoren durften, individuell durch die Signatur bestimmt, entweder zu $+1$ oder zu -1 quadrieren. Die Signatur sollte nicht fest verdrahtet sein, sondern per Software laufzeitvariabel geändert werden können.

Für einen typischen Einsatz des *FPGAs* war folgendes Schema angedacht:
Vor einer Anwendung sollte eine größere Menge von *Instruktionen* angesammelt werden, die dann in einem Block, möglichst ohne Datenvorverarbeitung, an den Prozessor geschickt wird. Während letzterer die Bearbeitung aufnimmt, kann die PC-CPU anderen Aufgaben nachgehen. Nach Auswertung aller geladen Befehle zeigt der Koprozessor die Beendigung durch ein spezielles Signal an. Die Ergebnismultivektoren können auf den PC heruntergeladen werden.

Den Ausgangspunkt der Arbeit bildeten im wesentlichen die drei Komponenten

- *PCI*-Erweiterungskarte 'microEnable `II`' mit *FPGA*
  - + Kartenhersteller ist Firma 'Silicon Software GmbH' aus D-68159 Mannheim. Informationen unter 'www.silicon-software.de'.
  - + Der *FPGA* ist vom Typ 'XC4085XLA HQ240 900'. Er wird von der Firma 'Xilinx' aus den USA, San Jose, CA 95124 produziert. Informationen unter 'www.xilinx.com'.
- `C++` - Hardwarebeschreibungssprache (*CHDL*)
  - + *CHDL*-Version: 1.97
  - + *CHDL* wurde an der Universität Mannheim am Lehrstuhl Informatik `V` entwickelt. Informationen unter 'http://www-li5.ti.uni-mannheim.de/fpga/index.shtml'.
- Entwicklungsumgebung
  - + Rechner: AMD Athlon 900 MHz, 256 MB RAM
  - + Betriebssystem: SUSE Linux, Kernel 2.2.16
  - + `C++` - Compiler: gcc version 2.95.2

Die Entwicklung des **GA**-Koprozessors erfolgte in der auf `C++` basierenden Hardwarebeschreibungssprache *CHDL*. Diese stellt strenggenommen eine vom Benutzer erweiterbare `C++` - Klassenbibliothek für digitale Standardschaltungen wie beispielsweise Addierer dar. Mit ihr können *Schaltwerke* zusammengesetzt werden, die zunächst nur ideell existieren, d.h. als Datenstruktur in Form einer Netzliste vorliegen. Über einen weiteren Schritt, Synthetisierung genannt, wird hieraus ein binärer Konfigurationsdatenstrom generiert, der den *FPGA* in weniger als 20 Millisekunden vollständig programmiert. Dem *Schaltwerk* wird gewissermaßen Substanz verliehen.

Der *FPGA* ist mit einem eigenen *SRAM* verbunden, das eine Speicherkapazität von zwei MByte zur Verfügung stellt. Dessen unabdingbare Nutzung machte die Konzipierung eines Protokolls zur Datenübertragung PC $\Leftrightarrow$ *SRAM* erforderlich. In die Konzeption wurde zudem der Informationsaustausch PC $\Leftrightarrow$ *Design* integriert, um dem Prozessor z.B. die Signatur übermitteln zu können oder ihn in bestimmte Betriebsmodi (Eingabedaten schreiben, Berechnungen durchführen, Ergebnisse auslesen, Warten) zu versetzten. Das Protokoll umfaßt auf Seiten des *FPGAs* mehrere Elemente, die Teil des **GA**-Koprozessors sind.

Das Betreiben eines Prozessors bedingt das Vorhandensein geeigneter Datenstrukturen.
Die *Instruktionen* werden in Form einer durch Adreßzeiger einfach verketteten Liste in den unteren Bereich des *SRAMs* geladen. Jede *Instruktion* wird durch drei Instruktionswörter à 32 Bit charakterisiert. In diesen sind neben dem Zeiger auf die Folgeinstruktion zwei Zeiger auf die zu verknüpfenden *Multivektoren* mitenthalten. Ferner sind weitere Informationen zu den Operanden und der Operation selbst kodiert.
Multivektoren werden ausschließlich durch Sequenzen von *Basisblades* repräsentiert. Sie können aufgrund der Zeigerstruktur an beliebiger Stelle im Speicher stehen. *Basisblades* sind 32 Bit breit und untergliedern sich in den geometrischen (8 Bit) und einen skalaren Anteil.
Die Ergebnisse wachsen mit jedem Resultatmultivektor vom oberen Bereich des *SRAMs* in Richtung der Eingabedaten. Die Ergebnismultivektoren belegen hierbei stets Blöcke einer festen Größe, so daß für alle *Basisblades* der *Algebra* eine Speicherstelle reserviert ist. Diese Maßnahmen erfüllen die Notwendigkeit einer möglichst einfachen Adressierung.

Der entwickelte **GA**-Koprozessor besteht im Kern aus den drei großen Bauteilen Kontrollwerk, Rechenwerk und Adreßwerk, die von den Daten auch in dieser Abfolge durchlaufen werden.
Das Kontrollwerk implementiert die Schnittstelle zum PC. Die Hauptaufgabe besteht in der Interpretation und Weiterverteilung von Daten, die aus dem *SRAM* ausgelesen werden: Adreßdaten gehen an das Adreßwerk und Basisblades, als Elemente der Operandenmultivektoren, werden an das Rechenwerk gesandt. Das Kontrollwerk speichert und steuert ebenfalls maßgeblich den Zustand des *Designs*.
Das Rechenwerk verknüpft die empfangenen *Basisblades* gemäß der aktuellen Operation. Die entstehenden Ergebnisblades müssen dann auf individuelle Speicherstellen im *SRAM* aufaddiert werden. Das Rechenwerk stellt für das Auslesen betreffender Daten Prioritätsanforderungen an das Adreßwerk. Die aus der Addition hervorgehenden Endergebnisse werden zur Speicherung an das Adreßwerk geleitet.
Das Adreßwerk koordiniert das Auslesen der Daten aus dem *SRAM*. Dies umfaßt auch das Anfordern von *Basisblades* der Operandenmultivektoren in der korrekten Reihenfolge. Das zuvor erwähnte Speichern der Endergebnisse läuft parallel und bedarf im wesentlichen nur einer Weiterleitung von Daten an das *SRAM*.

Das entworfene *Design* erfüllt die auf Seite 87 gemachten Anforderungen.
Bauteile für das innere und äußere Produkt sollten sich relativ einfach in das bestehende Layout integrieren lassen, da diese Operationen als Spezialfälle des geometrischen Produkts anzusehen sind. Obwohl die für eine Erweiterung benötigten schaltungsspezifischen Vorkehrungen getroffen sind, ist ungewiß, ob die verbliebenen Ressourcen des *FPGAs* ausreichen, um diese zu realisieren.
Die Bestimmung des geometrischen Anteils im Rahmen des **GP** erwies sich als die geringere Herausforderung; da in *CHDL* standardmäßig kein Bauteil für eine normale Multiplikation enthalten ist, mußte für die Auswertung des skalaren Anteils ein Multiplizierwerk extra konstruiert werden. Um den Umfang dieser Arbeit nicht zu sprengen, sollten lediglich vorzeichenbehaftete ganze Zahlen die Grundlage der Berechnungen bilden.
Neben den Eingabedaten, die vom PC geladen wurden, können auch Ergebnismultivektoren als Operanden herangezogen werden. Das *Design* ermöglich ebenfalls die interne Vertauschung der Operanden, um der nicht gegebenen Kommutativität des geometrischen Produkts gerecht zu werden. Überflüssige Berechnungen, die auf eine Multiplikation mit Null hinauslaufen, werden als solche erkannt und dynamisch übersprungen.

Alle entwickelten Komponenten und somit der ganze **GA**-Koprozessor sind vollkommen *pipelined*, mit anderen Worten gemäß dem *Fließbandprinzip* konzipiert. Hierbei wird der Weg, den die Daten durch das *Design* nehmen, in (zeitlich) gleichlange Abschnitte zerlegt, in denen individuelle Teilaufgaben verrichtet werden. Sie stellen Stationen dar, über die der Prozessor von mehreren Operationen simultan fließbandartig, d.h. untereinander streng sequentiell synchron, durchlaufen wird. Die designierten Ergebnisse gehen ihrer 'Fertigstellung' mit jedem *Taktzyklus* einen Schritt entgegen. Die Sektionierung des *Designs* ist frei wählbar und bestimmt so die Stufigkeit der *Pipeline*. Eine Erhöhung derer läßt den Verbrauch der *FPGA*-Ressourcen stark anwachsen, bewirkt jedoch eine Steigerung der maximalen Taktfrequenz, mit der der Prozessor betrieben werden kann.
Das zugrunde liegende *SRAM* limitiert die Arbeitsgeschwindigkeit indes auf 20 MHz, so

daß hier umgekehrt die Stufigkeit der Taktfrequenz angepaßt werden konnte.

Die Generierung der neuentworfenen Bauteile wie Addierer, Inkrementierer etc. erlaubt die Wahl der jeweiligen Eingangsdatenbreite. Es können beispielsweise 4 Bit, 5 Bit, . . . , n Bit Multiplizierer direkt erzeugt werden.

Vor allem große Stufigkeiten der *Pipeline* bergen Probleme, da sich eine Vielzahl von unfertigen Ergebnissen im Umlauf befinden können. Es kommt zu *Datenabhängigkeiten*, falls eine chronologisch später aufgetretene Operation ein Datum benötigt, das von einer vorangehenden Operation gerade produziert wird und sich noch in der *Pipe* befindet. In erster Linie erzeugt solche sogenannten RAW-Hazards die bereits erwähnte Prioritätsanforderung des Rechenwerks beim Zugriff auf inkonsistente Daten im *SRAM*.
Das *Design* kennt Mechanismen zur Umgehung resp. Behebung von *Datenabhängigkeiten*. Ersteres wird durch einen einfachen Wartemodus erreicht, der die Fortsetzung von Berechnungen hinauszögert, bis sich etwaige Inkonsistenzen aufgelöst haben müssen. Die zweite Methode spürt entstandene *Datenabhängigkeiten* auf, verwirft die betreffenden Daten und startet die verursachende Operation zu gegebenen Zeitpunkt erneut. Die Umsetzung des letzterern Konzepts ist aufwendiger, verbessert hingegen den Durchsatz.

8.2 Fazit

Im Rahmen dieser abschließenden Beurteilung soll zunächst die Ausgangssituation betrachtet werden.
Die frei konfigurierbaren Logikblöcke (*CLBs*) des *FPGA* erwiesen sich als sehr flexibel und praktikabel. Die integrierte spezielle *Carry*-Logik zur beschleunigten Verarbeitung von Überträgen kam häufig zum Einsatz. Ein allgemeines Manko stellte das *SRAM* dar. Ein CPU-Direktzugriff seitens des PC war nicht möglich. Schmale Datenpfade, serieller Zugriff über einen Lese-*Port* und eine hohe *Latenz* zählten zu den Nachteilen des Speichers. Ohne diesen begrenzenden Einfluß hätte das *Design* mit 100 MHz getaktet werden können. Das Vorhandensein einer Treiberfunktion der *PCI*-Karte zum Initialisieren des *SRAMs* mit Nullen wäre vorteilhaft gewesen. Die Datenübertragung über den *PCI-Bus* war aufgrund fehlender DMA-Unterstützung unter Linux extrem langsam. Die Simulation eines *Designs* beschränkte sich auf die wenig eingängige Darstellung auswählbarer Signalverläufe in Form eines Zeitdiagramms und war daher ermüdend bis strapaziös. Obwohl die Anleitung zu *CHDL* einige Fehler enthielt und sich der Service durch die Betreiber als sehr knapp erwies, war das Softwarekonstrukt ein mühelos anzuwendendes und zweckmäßiges Werkzeug. In der Menge der Standardschaltungen wurde lediglich ein Multiplizierwerk für ganze Zahlen vermißt.

Eine Erweiterung des *Designs* auf Gleitkommazahlen ist schwierig, aber denkbar; eine Unterbringung des Exponenten im Datenformat geht zu Lasten der Mantisse. Da der Ressourcenverbrauch des implementierten Multiplizierers quadratisch von der folglich geringeren Breite der Eingangsdaten abhängt, wären die Einsparungen beträchtlich. Sie würden eventuell ausreichen, um den zusätzlichen Bedarf an Logik zu decken.

Es stellt sich die Frage, welche Auswirkungen ein Verzicht auf das *Pipeline*-Konzept ergeben hätte. Es gäbe keine *Datenabhängigkeiten* und es könnte knapp weniger als eine Operation pro Takt ausgeführt werden. Hochrechnungen[1] haben gezeigt, daß ein solches *Design* mit etwa 6 MHz getaktet werden könnte, so daß 6.000.000 *Blade*-Einzelmultiplikationen pro Sekunde möglich wären. Dies entspricht im Schnitt 65 % des Durchsatzes des präsentierten **GA**-Koprozessors und liegt im Falle von 2D darüber. Eine fließbandartige Verarbeitung ist dennoch vorzuziehen, da die Taktfrequenz stark angehoben werden kann. Voraussetzung wäre ein schnelleres *SRAM* und ggf. eine Angleichung der *Pipeline*-Stufigkeit.

Die Wiederverwendung der entwickelten Module, wie Multiplizierer oder Inkrementierer wäre aufgrund der umfangreichen gestalterischen Freiheiten bezüglich ihrer Erzeugung zweifellos sinnvoll. Inwiefern eine Portierbarkeit auf neuere *FPGAs*, die zudem erweiterte Funktionalitäten bereitstellen, gegeben ist, kann an dieser Stelle nicht beantwortet werden.

Es hat sich gezeigt, daß SIMD-Berechnungen ('single instruction - multiple data') wie das geometrische Produkt zweier großer *Multivektoren* für eine optimale Auslastung eines *FPGA-Designs* notwendig sind [46]. Auf diese Weise wird der Einfluß durchsatzschädigender Kontrollaufgaben wie Instruktionswechsel minimiert.
Heutige Prozessoren des Einzelhandels haben 20 bis 30 mal höhere Taktfrequenzen als ein mit 100 MHz betriebener *FPGA*. Sie verfügen zudem über sogenannte Superskalararchitekturen, um beispielsweise pro Takt eine Gleitkomma-Operation abzuschließen [30, 29]. Der Wunsch, Berechnungen auf einem *FPGA* schneller als auf einer CPU ausführen zu können, ist daher trotz der im vorherigen Kapitel präsentierten Ergebnisse schwer zu realisieren [1]. Es bleibt das Ziel, die CPU durch Parallelisierung von Aufgaben zu entlasten, um im Ganzen eine Leistungssteigerung zu erreichen. In der Praxis stellte sich der Datenaustausch PC $\Leftrightarrow$ *FPGA* als ein Engpaß heraus, der den Einsatz des **GA**-Koprozessors fast hinfällig macht. Obgleich sich der Datentransfer extrem optimieren ließe, bleibt dieser Von-Neumann-Flaschenhals doch erhalten [38, 42].
Eine kostspielige Lösung wäre ein überlappender Gebrauch von drei *FPGAs* gleichzeitig. Der Idee nach fallen Lade- und Entladevorgänge zweier Chips in die Operationszeit des jeweils dritten. Bei genügendem *Instruktions*-Aufkommen können die aus dem Datenengpaß entstandenen Defizite wettgemacht werden.

Als Fazit soll festgehalten werden, daß zwar eine relativ optimale Implementierung für das *FPGA*-Board erstellt wurde, dies zur Zeit dennoch keine echte Alternative oder Ergänzung zu einer Softwarelösung darstellt. Nur eine Verbesserung der Rahmenbedingungen könnte die Glaubwürdigkeit des Prozessors steigern. Weiterführende Betrachtungen sind Gegenstand des folgenden Abschnitts 'Ausblick'.

8.3 Ausblick

Speziell im Hinblick auf das vorgestellte *Design* gilt zweifelsfrei, daß zusätzliche Ressourcen auf dem *FPGA*, ein besseres *SRAM*, effizientere Datenübertragungen via *PCI-Bus* und eine ausgereifte Softwareintegration den **GA**-Koprozessor zu einem praktikabelen ernstzunehmenden Werkzeug machen würden. Seine Anwendung könnte künftig helfen, Echtzeitberechnungen in geometrischer *Algebra* zu verwirklichen.

[1] Mittels Auswertung von FPGA-Datenblättern [22, 23].

Die Zukunft des hier behandelten *Designs* ist stellvertretend für die vieler anderer *FPGA*-Koprozessoren und somit des *FPGAs* an sich. Der nachstehende Teil des Ausblicks beschäftigt sich mit den Perspektiven des *FPGA*-Konzepts.

FPGAs bieten reichlich Vorteile [4, 8]. Ein zentraler Punkt ist, daß ein durch Software dynamisch erzeugtes *Schaltwerk* innerhalb weniger Millisekunden 'on-demand' betriebsbereit zur Verfügung steht.
In erster Linie steht ferner die Anpassungsfähigkeit auf ein konkret gestelltes Problem. Beispielsweise ist es möglich, für Berechnungen in einer 6D-*Algebra* ein 6D-*Design* zu verwenden.
Das Chipdesign findet auf einer höheren Abstraktionsebene statt, als dies für TTL-Schaltungsentwurf der Fall ist. Lästige Optimierungsaufgaben und die Problematik der Realisierung durch geeignete logische Grundgatter tritt in den Hintergrund.
Forschungszentren und kleine Unternehmen sind in der Lage, benötigte Chips aus den Bereichen Robotik, Meßtechnik, Übertragungsprotokolle etc. durch Programmieren eines *FPGAs* selbst herzustellen - teure Spezialanschaffungen entfallen [3].
Einfache Entwurfstechniken und leichte Verifikationsmöglichkeiten verkürzen den Produkt-Lebenszyklus von Schaltungen. Schnell verfügbare Prototypen ('rapid prototyping') erlangen früher die Marktreife ('time-to-market') [3]. Fehlende Konkurrenz vergrößert den Marktanteil und sichert höhere Gewinne. Im Servicefall brauchen zur Fehlerbeseitigung keine Bauteile ausgewechselt werden, ein Update reicht.

Rasant im Wandel inbegriffene Bereiche wie Multimedia (JPEG, MPEG, Wavelet) [49] und Telekommunikation (ISDN, ATM, UMTS) [26] erfordern das rasche Adaptieren von Hardware-Funktionalitäten an neue Standards. Die Rekonfigurierbarkeit von *FPGAs* gewährleistet dies [6, 4].
Die Möglichkeit, mehrere separate Logikbauteile auf einem *FPGA* unterzubringen, unterstützt das im Trend liegende Konzept des 'Systems auf dem Chip'. Hierbei befinden sich neben dem Mikroprozessor als zentralen Kern auch die normalerweise peripheren Bauteile der Zusatzlogik auf einem Chip [8, 2]. Eine praktische Anwendung wäre das Herunterladen von Hardware-Plugins aus dem Internet [4]. Aktuelle *FPGAs* erlauben indes die sogenannte 'dynamisch partielle Rekonfiguration', bei der Teile des *FPGA*-Chips während des Betriebs neu konfiguriert werden dürfen [7, 21, 5]. Zuvor verarbeitete Daten können als Zwischenergebnisse in dem *FPGA* bleiben, während dieser einen anderen Satz von Instruktionen lädt, um die Berechnungen zum Abschluß zu bringen. Denkbar sind auch selbstmodifizierende Schaltungen, die etwa neuronale Netze realisieren könnten.

Die Klasse der programmierbaren Logikchips wie *FPGAs* profititert zwar von den Technologien Intels & Co, hängt dagegen im Herstellungsprozeß vergleichsweise weit zurück. Das kommerzielle Interesse als Triebfeder der Entwicklung könnte steigen, falls *FPGAs* die Technologielücke zwischen Mikroprozessoren und ASICs[2] ('application specific integrated circuits') schließen können. Während Mikroprozessoren vielfältige Aufgaben relativ langsam verarbeiten, stellen ASICs durchsatzstarke, aber hochspezialisierte Lösungsansätze dar. Bei *FPGAs* sind neben Algorithmen auch die Ressourcen variabel, so daß Leistungsfähigkeit nicht auf Kosten von Universalität gehen muß [8, 2].

[2]Hier sind vom Hersteller vorkonfigurierte, d.h. dann festverdrahtete ASICs gemeint.

Für *FPGAs* spricht zudem, daß die Silizium-Technologie gängiger Mikroprozessoren bald ausgereizt sein wird, so daß die *LCAs* gleichziehen und eine ernsthafte Konkurrenz darstellen könnten.
Die Integrationsdichten heutiger RAM-Bausteine und Mikroprozessoren unterscheiden sich teils um den Faktor 100. Der Grund dieser als Entwurfslücke bezeichneten Tatsache besteht in den regelmäßigen Strukturen von Speicherbausteinen. Da *FPGAs* gleichermaßen aufgebaut sind, läßt sich deren Integrationsdichte in Zukunft möglicherweise auf ein ähnliches Niveau erhöhen.

Literaturverzeichnis

[1] c't Magazin, *FPGAs mit 20 GHz*, Artikel, 18/2001/S.49

[2] c't Magazin, *Verschmelzung FPGA - ASIC*, Artikel, 20/2002/S.33, 06/2002/S.79, 14/2002/S.27

[3] c't Magazin, *Kostenfaktor FPGA*, Artikel, 10/2002/S.43, 17/2000/S.202*ff*

[4] c't Magazin, *Open Source Hardware*, Artikel, 22/2000/S.294*ff*, 21/1999/S.282*ff*, 17/2000/S.202*ff*

[5] Markus Ernst, *Real time image processing based on reconfigurable hardware acceleration [hrsoc02_image.pdf]*, Paper, Technische Universität Darmstadt,
URL: 'http://www.vlsi.informatik.tu-darmstadt.de/staff/ernst/ernst.html',
akquiriert 02/2003.

[6] Sorin A. Huss, *Rekonfigurierbare Prozessoren - Anwendungsspezifische informationstechnische Systeme*, Skriptum, Technische Universität Darmstadt, 2002/2003.

[7] Andreas Haase, *Untersuchungen zur dynamischen Rekonfigurierbarkeit von FPGA [Diplom.pdf]*, Diplomarbeit, Technische Universität Chemnitz, 2001,
URL: 'http://www-user.tu-chemnitz.de/~anhaa/diplom.htm', akquiriert 02/2003.

[8] Reiner Hartenstein, *Der Mikroprozessor im nächsten Jahrtausend - Als diskreter Baustein hat er keine Zukunft mehr [paper106b.pdf]*, Artikel aus Elektronik, 01/2000,
URL: 'http://xputers.informatik.uni-kl.de/papers/',
akquiriert 02/2003.

[9] David Hestenes, *Vectors, spinors, and complex numbers in classical and quantum physics [VeSp&ComUpdated.pdf]*, Artikel, American Journal of Physics, Vol. 39/9, Sep. 1971,
URL: 'http://modelingnts.la.asu.edu/pdf/', akquiriert 01/2003.

[10] Chris Doran, *Geometric algebra and computer vision - the auto-calibration problem [mexico.ps]*, Essay, Department of Physics, Cambridge
URL: 'http://www.mrao.cam.ac.uk/~clifford/recent/', akquiriert 10/2002.

[11] Chris Doran, *Geometric algebra and its application to mathematical physics [chris_thesis.ps]*, Dissertation, University of Cambridge, 1994
URL: 'http://www.mrao.cam.ac.uk/~clifford/publications/ps/', akquiriert 01/2003.

[12] Lasenby / Doran / Fitzgerald, *New geometric methods for computer vision: an application to structure and motion estimation [ijcv1_96.ps]*, Paper, Cambridge, URL: 'http://www.mrao.cam.ac.uk/~clifford/publications/ps/', akquiriert 10/2002.

[13] C.Doran / A.Lasenby / S.Gull / S.Somaroo / A.Challinor, *Spacetime algebra and electron physics [elec_review.ps]*, Paper, URL: 'http://www.mrao.cam.ac.uk/~clifford/publications/ps/', akquiriert 10/2002.

[14] Chris Doran *Physical applications of geometric algebra [hout01.ps, hout02.ps, ...]*, Skriptum, URL: 'http://www.mrao.cam.ac.uk/~clifford/ptIIIcourse/', akquiriert 08/2002.

[15] Leo Dorst, *Geometric algebra: the framework for geometric computations [talk200204.ps]*, Paper, University of Amsterdam, URL: 'http://carol.wins.uva.nl/~leo/clifford/', akquiriert 08/2002.

[16] Leo Dorst, *Geometric (Clifford) algebra: a practical tool for efficient geometrical representation [talknew.pdf]*, Paper, University of Amsterdam, URL: 'http://carol.wins.uva.nl/~leo/clifford/', akquiriert 08/2002.

[17] Dorst / Mann, *Geometric algebra: a computational framework for geometrical applications, part I: algebra [dorst-mann-I.pdf]*, Paper, University of Amsterdam, URL: 'http://www.science.uva.nl/~leo/clifford/', akquiriert 08/2002.

[18] Dorst / Mann, *Geometric algebra: a computational framework for geometrical applications, part II: applications [dorst-mann-II.pdf]*, Paper, University of Amsterdam, URL: 'http://www.science.uva.nl/~leo/clifford/', akquiriert 08/2002.

[19] Klaus Kornmesser, *The FPGA development system CHDL - user manual*, Universität Mannheim, Lehrstuhl für Informatik V, 2000.

[20] Silicon Software GmbH, *microEnable - Benutzerhandbuch WinNT und Linux*, Version 1.3, URL: 'http://www.silicon-software.com', 1999.

[21] Xilinx Inc., *FAQ - partial reconfiguration*, URL: 'http://www.xilinx.com/xilinxonline/partreconfaq.htm'.

[22] Xilinx Inc., *FPGA application notes [XAPP013.pdf, XAPP043.pdf, XAPP095.pdf, ...]*, URL: 'http://www.xilinx.com/apps/4000.htm', akquiriert 04/2002.

[23] Xilinx Inc., *FPGA data sheets [DS004.pdf, DS015.pdf, ...]*, URL: 'http://direct.xilinx.com/bvdocs/publications/', akquiriert 04/2002.

[24] Xilinx Inc., *global glossary [glossary.pdf]*, URL: 'http://www.xilinx.com/publications', akquiriert 04/2002.

[25] Werner Damm, *Rechnerarchitektur*, Skriptum, Universität Oldenburg, 1997.

[26] Kowalk / Burke, *Rechnernetze - Konzepte und Techniken der Datenübertragung*, Universität Oldenburg, Teubner Verlag Stuttgart, 1994.

[27] Wolfgang Nebel, *Entwurf Integrierter Schaltungen*, Skriptum, Universität Oldenburg, 1999.

[28] Wolfgang Nebel, *Rechnerstrukturen*, Skriptum, Universität Oldenburg, 1997.

[29] Hans-Peter Messmer, *PC-Hardwarebuch - Aufbau , Funktionsweise, Programmierung*, Addison-Wesley, 1996.

[30] Trutz / Eyke / Podschun, *Das Assembler-Buch - 8086 bis Pentium*, Addison-Wesley, 1995.

[31] Klaus Jänich, *Lineare Algebra*, Springer Verlag, 1996.

[32] Albrecht Beutelspacher, *Lineare Algebra - Eine Einführung*, Vieweg Verlag, 1995.

[33] Gerd Fischer, *Lineare Algebra*, Vieweg Verlag, 1995.

[34] D.Brannan / M.Esplen / J.Gray, *Geometry*, Cambridge University Press, 1999.

[35] Uwe Schöning, *Logik für Informatiker*, Spektrum Akademischer Verlag, 1995.

[36] Herwig Feichtinger, *Arbeitsbuch Mikrocomputer*, Franzis Verlag.

[37] Bernd Jähne, *Digitale Bildverarbeitung*, Springer Verlag, 1997.

[38] Rechenberg / Pomberger, *Informatik - Handbuch*, Carl Hanser Verlag, 1999.

[39] Jean Gallier, *Geometric methods and applications - for computer science and engeneering*, Springer Verlag, 2001.

[40] Bjarne Stroustrup, *Die C++ Programmiersprache*, Addison-Wesley, 1998.

[41] Arndt Bode, *RISC - Architekturen*, BI Wissenschaftsverlag, 1990.

[42] Theo Ungerer, *Innovative Rechnerarchitekturen - Bestandsaufnahme, Trends, Möglichkeiten*, McGraw-Hill, 1989.

[43] Gert Regenspurg, *Hochleistungsrechner- Architekturprinzipien*, McGraw-Hill, 1989.

[44] Hennessy & Patterson, *Computer architecture - a quantitative approach*, Morgan Kaufmann Publishers Inc., 1996.

[45] Rudolf Busch, *Elektrotechnik und Elektronik - Anwendungen für Ingenieure*, Teubner Verlag Stuttgart, 1994.

[46] Christian Märtin, *Rechnerarchitektur - Struktur, Organisation, Implementierungstechnik*, Carl Hanser Verlag, 1994.

[47] Zeidler / Hackbusch / Schwarz, *Taschenbuch der Mathematik*, Teubner Verlag Stuttgart, 1996.

[48] Christian B.U. Perwass, *Applications of geometric algebra in computer vision*, Dissertation, University of Cambridge, 2000.

[49] Foley / van Dam / Feiner / Hughes, *Computer graphics - principles and practice*, Addison-Wesley, 1999.

[50] Wolfgang Nolting, *Theoretische Physik - Klassische Mechanik*, Vieweg Verlag, 1999.

[51] Wolfgang Nolting, *Theoretische Physik - Analytische Mechanik*, Vieweg Verlag, 1999.

[52] Wolfgang Nolting, *Theoretische Physik - Elektrodynamik*, Vieweg Verlag, 1999.

[53] Wolfgang Nolting, *Theoretische Physik - Quantenmechanik*, Vieweg Verlag, 1999.

[54] Torsten Fließbach, *Elektrodynamik - Lehrbuch zur Theoretischen Physik*, Spektrum Akademischer Verlag, 2000.

[55] Torsten Fließbach, *Quantenmechanik - Lehrbuch zur Theoretischen Physik*, Spektrum Akademischer Verlag, 2000.

Glossar

Die in diesem Kapitel angegebenen Erklärungen erheben nicht den Anspruch, universell gültig zu sein, sondern beziehen sich auf diese Arbeit. Der Pfeil '→' zeigt Querverweise innerhalb des Glossars an oder nennt betreffende Textstellen.

Algebra *In einer Algebra $\mathcal{A}$ über einen linearen Raum $\mathbb{K}$ ist zusätzlich über $\mathbb{K}$ eine distributive aud assoziative Multiplikation erklärt, die einem Paar (a, b) von Elementen aus $\mathcal{A}$ eindeutig ein drittes Element ab aus $\mathcal{A}$ zuordnet.*

Algebra, degeneriert *Auch 'entartet'. Eine Algebra $\mathcal{A}$ heißt degeneriert, falls zwei Elemente $a \neq 0$, $b \neq 0$ aus $\mathcal{A}$ existieren, deren Produkt $a\,b$ Null ergibt.*

Algebra, universelle *Eine Algebra $\mathcal{A}$ wird universell genannt, falls sie durch die Art des auf ihr definierten Produkts, bis auf Isomorphien bzgl. eines Basiswechsels, eindeutig bestimmt ist.*

Basisblade, allgemein *Diese Elemente entstehen durch Permutation der in den geordneten Basisblades enthaltenen Basiseinheitsvektoren e_i ($e_1e_2 \rightarrow \{e_1e_2,\ e_2e_1\}$).*

Basisblade, geordnet *Die Elemente einer Basis von $\mathcal{C}\ell$. Bsp.: $\mathcal{C}\ell(\mathbb{R}^2) = \{1, e_1, e_2, e_1e_2\}$(→ S.4). In dieser Arbeit werden die g.B. auch die 'Elemente' der Multivektoren genannt.*

Blade *Objekt, das eine Darstellung als äußeres Produkt mehrerer (1-)Vektoren besitzt. Bsp: $e_2e_4 - e_3e_2 = e_2 \wedge (e_4 + e_3)$.*

Bus *Ein Bus bezeichnet ein Bündel von zusammengehörigen Signalleitungen, das mehrere Hardwarekomponenten zum Informationsaustausch miteinander verbindet. Auf einen Bus kann nur eine Komponente zur Zeit Daten anlegen. Die Steuerung übernimmt i.a. ein 'bus controller'.*

B-Zyklus *Entspricht → 'Durchlauf'.*

Burst, pipelined *Ein Burst ist ein Transfer eines Datenblocks über einen Bus. Statt Paare aus Adresse mit assoziiertem Datum zu übertragen, wird bei zusammenhängenden Datenblöcken lediglich am Anfang des Transfers die Startadresse des Blocks übermittelt.*

Cache *Schneller Pufferspeicher (deu: Depot, Versteck) zwischen Prozessorregistern und Hauptspeicher, der Kopien von Daten aus letzterem zur schnellen Verfügbarkeit enthält.*

Carry *Ein Carry ist Übertragsbit, das bei Additionen und Subtraktionen entstehen kann. Bei einer Addition wird ein Carry c_{x+1} an einer Stelle (Bitposition) x erzeugt, falls unter den beiden zu addierenden Bits a_x, b_x der Stelle x, zusammen mit dem Carry-Bit c_x von der Vorstelle $x - 1$, mindestens zwei Bits den Wert 1 haben.*

Carry look ahead *Verfahren, um bei Addierern und Subtrahierern → Carries möglichst schnell zu berechnen. Hierzu wird für jedes Eingangsdatenpaar separat ermittelt, ob es ein Carry selbst erzeugen (generieren) oder ein eingehendes Carry weiterleiten (propagieren) würde. Eine aufgesetzte Logik errechnet hieraus die Überträge.*

CE *Das Kürzel CE steht für 'clock-enable'. Getakteten Bauteile in Schaltwerken ändern üblicherweise in Abhängigkeit von dem Signal → 'clock' ihren Zustand. Ist CE für ein solches Element Low, kann es den Zustand nicht mehr ändern, 'clock' wird ignoriert.*

CHDL *Eine → VHDL - ähnliche Hardwarebeschreibungssprache in* `C++` *(→ S.21).*

CISC *Akronym für 'complex instruction set computer'. Prozessoren mit CISC-Architektur haben umfangreiche und komfortabele Maschinenbefehlssätze mit Befehlen verschiedener Wortlängen, die somit einen kompakten Code ermöglichen (→ RISC).*

CLB *Für 'configurable logic block'. CLB's sind auf dem → FPGA matrizenartig angeordnete Einheitsblöcke. Durch sie können → Schaltwerke realisiert werden (→ S.20).*

CMC *Kürzel von 'common mezzanine card'. IEEE Spezifikation zum mechanischen Aufbau von Tochtereinsteckkarten. Dies umfaßt auch eine Pinbelegung der Steckverbindung.*

Clock *Das erzeugte (Rechteck-) Signal eines taktgebenden Bauteils digitaler Schaltungen.*

Datenabhängigkeit *Bei einer D. steht ein Operand eines Befehls nicht unmittelbar zur Verfügung, weil der vorige Befehl diesen z.B. gerade noch produziert (→ S.48).*

Debugger *Dt: 'Entwanzer'. Ein Debugger ist ein Programm, mit dem andere Soft- oder Hardware zur Fehlersuche, i.a. zur Laufzeit, untersucht und verändert werden kann. Hierbei kann über einen → Einzelschrittmodus der Werdegang von Variablen/Signalen nachvollzogen werden.*

Design, Chip- *Bezeichnung für ein per Software generiertes → Schaltwerk, das zunächst nur als digitale Netzliste, d.h. in Form einer Datenstruktur existiert. Das Design kann auf einem programmierbaren Logikchip (z.B. FPGA) realisiert werden, um als Hardware zu funktionieren.*

D-FlipFlop, DFF *Gebräuchliches → FlipFlop, dessen Dateneingang* **`D`** *sich von 'delay' ableitet, was darauf hindeuten soll, daß das Signal auf der* **`D`***-Leitung um einen Takt verzögert (meist ab der steigenden → Flanke des nächsten Takts) am Ausgang* **`Q`** *anliegt.*

DPSRAM *Bauteil 'Dual port SRAM' aus der CHDL-Bibliothek (→ S.36), das dem SRAM als Aufsatz vorgeschaltet wird, um als funktionell erweitertes RAM zu dienen: Es täuscht über eine schnelle Wechseltechnik die Fähigkeit des gleichzeitigen Lesens und Schreibens (dual port) am SRAM vor, obwohl dort physikalisch nur ein → Port exisitiert.*

Durchlauf *Ein durch das* **`BCycle`***-Signal initiierter B-Schleifendurchlauf, bei dem stets ein B-Blade aus dem Multivektor B mit allen A-Blades des Multivektors A über eine Operation (hier geometrisches Produkt) verknüpft wird.*

Durchgang *Entspricht 'Durchlauf'.*

Einzelschrittmodus *Betriebsmodus eines → Debuggers, in dem der Ablauf eines Programms (oder einer Logikschaltung) zu dessen Verständnis in kleinsten Schritten ausgeführt werden kann. Die Schrittweite beträgt typischerweise ein oder mehrere Maschinenbefehle bzw. ein oder mehrere Takte.*

Fan-Out *Zu treibende Gatterlast (elektr. Kapazitäten), d.h. Anzahl aller angeschlossenen Gatter am Ausgang des betreffenden Bauteils.*

Fifo *Kürzel für 'first-in-first-out'. Ein F. (Puffer) ist eine auf dem Prinzip der (Warte-) Schlange basierende Struktur, die Daten zwischenspeichert. Die Elemente eines F. können nur in der Reihenfolge ihres Eintretens in den F. wieder ausgelesen werden.*

Flanke *Verlauf eines binären digitalen Signals während des kurzen Zeitraums eines elektr. Pegelwechsels von Low auf High (steigende F.) oder von High auf Low (fallende F.).*

Flag *Eine 1 Bit-wertige Variable, die einen Zustand anzeigt. Ein Flag kann gesetzt oder gelöscht sein. Zur Realisierung von Flags eignen sich → FlipFlops.*

Fließbandprinzip *Bei dem F. (→ Pipelining) werden Operationen, in Teilaufgaben zerlegt, an entsprechend viele aufeinanderfolgende Stationen verteilt, die sequentiell einzelne Schritte ausführen. Neue Operationen können initiiert werden, sobald Station Eins vakant wird. Bedingung für das F. ist das synchrone Arbeiten aller Stationen. Über das F. läßt sich der Durchsatz (Operationen pro Zeit) enorm steigern.*

FlipFlop, FF *Bistabiles Bauteil, das ein Bit speichern kann. Die Rückkopplung zweier → NOR-Gatter liefert z.B. ein einfaches sog. RS-FF. FlipFlops werden als Zustandsspeicher in Schaltwerken verwendet. Es gibt asynchrone, taktpegel- oder flankengesteuerte FFs.*

Forwarding *Dieses Prinzip soll in → Pipelinestrukturen den Durchsatz steigern, indem → Stalls durch Auflösen von → Datenabhängigkeiten vermieden werden. Hierfür werden abkürzende Datenpfade ('bypass') eingesetzt, die es dann erlauben, ein Rechenergebnis direkt als neuen Operanden zu laden, ohne Umwege über das RAM zu machen.*

FPGA *Akronym für 'field programmable gate array' (→ S.20). Ein FPGA ist ein Chip, auf dem Logikschaltungen reversibel implementiert und ausgeführt werden können.*

Gate *Entspricht 'Gatter'.*

Gatter *Grundelemente digitaler Schaltungen, die binäre Schaltvariablen verknüpfen. Die Grundgatter 'AND', 'OR' und 'NOT' erzeugen eine funktionell vollständige Schaltalgebra, d.h. sie eignen sich, um alle anderen Gattertypen (→ 'XOR',etc.) zu generieren. Gatter werden aus elektronischen Halbleiterbauelementen (z.B. Transistoren) realisiert.*

Gültigkeit *Pipe-Einträge werden als gültig bezeichnet, falls ihr Pipe-Durchlauf schließlich in einem Schreibzugriff auf den betreffenden Ergebnismultivektor (im → SRAM) resultieren wird. Gültige Einträge gehen daher stets auf A-Blades zurück bzw. dürfen im allgemeinen mit letzteren assoziiert werden.*

Hazard, Data-H. *Andere Bezeichnung für → Datenabhängigkeit (→ S.48).*

High *Bezeichnung für den hohen Pegel einer Spannung, der den Zustand '1'/'Ein' der → positiven Logik repräsentiert. Bezüglich $V_{min} = 0V$ bis $V_{max} = 5V$ entspricht High dem Spannungsbereich $2,4V \ldots 5V$.*

Instruktion *Eine Instruktion ist ein Befehl, der das Produkt zweier → Multivektoren erzeugt. In vielen → Durchläufen wird jedes → Blade von Multivektor B wird mit allen Blades von Multivektor A paarweise verknüpft (→ S.31).*

Interface *'Schnittstelle', Verbindungsstück zur Kommunikation zwischen Schaltungen, Programmcode oder Rechnern.*

Latenz *Benötigte Zugriffszeit.*

LCA *Kürzel von 'logic cell array'. LCAs sind eine Unterklasse der → FPGAs, die speziell über RAM-Zellen (wieder-)programmiert werden können. Ein LCA-Chip ist im wesentlichen ein regelmäßig strukturiertes Feld von → CLBs und deren Verdrahtung (-slogik).*

Level *Entspricht → 'Pegel'. Der Ausdruck 'Auf hohem Level' meint beispielsweise 'High'.*

Level, logische *Die l.L. dienen zur Abschätzung von Längen von Signalwegen. In dieser Arbeit ist festgelegt, das ein → LUT einem l.L. entspricht. Die Anzahl der l.L. eines → Schaltnetzes ist durch die Länge (in LUTs) des maximalen Signalpfades bestimmt.*

Logik, positiv *Gleichsetzung der abstrakten Variablenwerte 0 und 1 der Schaltalgebra mit den Pegelwerten Low und High, so daß Low $\equiv\ 0$ und High $\equiv\ 1$ ist.*

Low *Bezeichnung für den niedrigen Pegel einer Spannung, der den Zustand '0'/'Aus' der positiven Logik repräsentiert. Bei einem Spannungsintervall von $V_{min} = 0V$ bis $V_{max} = 5V$ entspricht Low dem Bereich $0V \ldots 0,7V$.*

Ltg *'Leitung'.*

LUT *'Lookuptable'. Frei programmierbare RAM-Zelle in → CLBs (→ S.20).*

Multiplexer *Bauelement, das aus einer Reihe von Eingangssignalen nur eines auf die Ausgangsleitung durchschaltet. Die Wahl des Eingangs erfolgt über Steuerleitungen.*

Multivektor *M. sind die Elemente der geometrischen Algebra (→ S.4). Sie bestehen aus skalarbehafteten → Basisblades, deren Verbund durch das '+'-Zeichen symbolisiert wird. Bsp.:* $A = 2 + 3e_1e_2e_3$.

MUX *Abk. von → 'Multiplexer'.*

MVA, MVB, ... *Kürzel für z.B. 'Multivektor A'. MVB[0] steht für Element Null von Multivektor B und entspricht so dem ersten B-Blade des Multivektors.*

NOR *Inverses der logischen Verknüpfung/Schaltfunktion OR.*

PAL *Steht für 'programmable array logic'. PLD, dessen UND-Matrix frei programmierbar ist, während die ODER-Matrix, im Ggs. zum PLA, fest ist.*

PCI *'Peripheral component interconnect'. Bei PCI handelt es sich um eine verbreitete 32 Bit breite, burstfähige Busstruktur, die ein Multiplexing-Schema verwendet. Hierbei werden die Leitungen abwechselnd für Adressen und Daten genutzt (→ S.19).*

Pegel *Ein hoher bzw. niedriger (Spannungs-)Pegel korrespondiert mit den Zuständen → High und → Low.*

Pipe, -line *'Fließband'. Der Begriff P. steht im wesentlichen für den durch die Abfolge aller → Registerebenen entstandenen Aufbau, der den Datenfluß im → Design wiederspiegelt. Die durch die Registerebenen getrennten → Stufen sind ebenfalls Teil des Ausdrucks.*

Pipelining *Dieses → 'Fließbandprinzip' beinhaltet die Aufteilung einer Rechnerstuktur durch Einfügen von → Registerebenen in kleine unabhängige Teilwerke. Diese Funktionseinheiten arbeiten fließbandartig gleichzeitig an der Ausführung der Befehle/Operationen. Am Ende jeder → Stufe werden alle Daten synchron in → Register übernommen (→ S.89).*

PLA *Für 'programmable logic array'. Ein PLA ist ein frei programmierbares PLD.*

PLD *Akronym für 'programmable logic device'. PLDs realisieren über Grundgatter (→ Gatter) die disjunktiven Formen logischer Gleichungen. Hierbei ist einer Eingangs-UND-Matrix, die die konjunktiven Terme bildet, eine ODER-Matrix nachgeschaltet, die die disjunktive Verknüpfung erzeugt. Die Verschaltung in den Matrizen erfolgt über programmierbare Verbindungen, die i.a. nicht reversibel sind.*

PLX *Hardware-Brückenelement zur Kommunikation über das → PCI-Protokoll.*

Polling *Verfahren zur Datenübertragung, bei dem die Oberservable (z.B. ein Signal) durch kontinuierliches Abfragen auf Zustandsänderungen (z.B. neue Daten) überprüft wird.*

Port *Bezeichnung für Anschluß, Steckverbindung, Schnittstelle.*

PROM *'programmable read only memory'. Im Ggs. zum PAL ist die UND-Matrix eines PROMs fest verdrahtet und die ODER-Matrix frei programmierbar.*

Quaternion *Die Menge* $\mathbb{H}$ *der 1843 von Hamilton gefundenen Q.* $\alpha + \beta\mathbf{i} + \gamma\mathbf{j} + \delta\mathbf{k}$ *mit* $\alpha, \beta, \gamma, \delta \in \mathbb{R}$ *ist ein Schiefkörper, der den Körper* $\mathbb{C}$ *der komplexen Zahlen als Unterkörper enthält. Die Algebra der Q. ist isomorph zur Cliffordalgebra* $\mathcal{C}\ell_2$.

Register *Ein n Bit breites R. besteht aus n parallelen → FlipFlops.*

Registerebene *Menge aller Register zwischen zwei benachbarten → Stufen.*

Registerstufe *Entspricht 'Registerebene'.*

RISC *Abk. von 'reduced instruction set computer'. RISC-Prozessoren haben im Ggs. zu → CISC einen eingeschränkten einheitlicheren Maschinenbefehlssatz, dessen Befehle identische Wortlängen haben. Der im Vergleich zu CISC erzeugte Code ist umfangreicherer, läßt sich aber gut auf das Pipeline-Konzept anwenden.*

Schaltnetz *Digitale kombinatorische Logikschaltungen, die keine Rückkopplungen bzw. speichernde Elemente enthalten. Die Ausgangswerte werden ausschließlich durch die anliegenden Eingangswerte bestimmt.*

Schaltwerk *Digitale Logikschaltungen mit speichernden Elementen (→ FlipFlop) werden als S. bezeichnet. Ihre Ausgangswerte können zusätzlich von vergangenen Eingangswerten abhängen. Alle endlichen Automaten lassen sich als Schaltwerke darstellen.*

Setup Zeit *Benötigte individuelle minimale Zeitdauer vor einem Pegelwechsel des Taktes, die ein Signal stabil am Eingang eines getakteten Bauteils anliegen muß, damit es fehlerfrei arbeitet.*

S-LINK *Verbreitete Spezifikation für eine Punkt-zu-Punkt Verbindung zur Kommunikation mit externer Hardware.*

SRAM *Statisches RAM, das an den → FPGA angegliedert ist (→ S.19).*

Stack *Dt: 'Stapel'. Speicherstruktur basierend auf dem Modell eines Stapels: Die Zugriffe sind nicht beliebig, sondern erfolgen nach dem lifo-Prinzip ('last-in-first-out'). Ein S. wächst i.a. in Richtung abnehmender Adressen.*

Stall *Entspricht → 'Wartezyklus'.*

Stufe *Auch 'Phase'. Bezeichnet ein Teilwerk (-schaltung) der → Pipeline, das gerade ausschließlich Daten verarbeitet, die demselben Befehl/Operation angehören. Ein Befehl muß daher zu verschiedenen Zeiten (→ Taktenzyklen) mit verschiedenen S. assoziiert werden. Eine S. ist stets in Eingangs- und Ausgangsregistern gekapselt (→ Registerebene).*

Taktzyklus *Der T. entspricht einer Schwingungsperiode der → Clock. Er bezieht sich üblicherweise auf zwei aufeinanderfolgendene steigende → Flanken.*

Tristate *Ein solcher 'Drei-Zustands'-Anschluß kann neben den → High- und → Low-Pegeln einen hochohmigen Zustand annehmen, so als ob das betreffende Bauteil nicht vorhanden wäre. Sonst bestünde z.B. bei Bussen die Möglichkeit, daß ein Anschluß mit einem niedrigen einen anderen Anschluß mit einen hohen Pegel beeinflussen könnte.*

Ungültigkeit *Gegenteil der → 'Gültigkeit'*

VHDL *Akronym von 'VHSIC hardware description language', VHSIC='very high speed integrated circuits'. VHDL basiert auf einer klassischen Programmiersprache, erweitert diese aber durch Konstrukte für die Beschreibung elektronischer Schaltungen.*

Waitstate *Entspricht 'Wartezyklus'.*

Wartezyklus *Taktzyklus, in dem keine Operation stattfindet. Bei RAM-Zugriffen oder → Datenabhängigkeiten müssen beispielsweise häufig Wartezyklen eingelegt werden.*

XOR *Ein XOR repräsentiert eine Exklusiv-Oder-Verknüpfung, die nur logisch wahr wird, wenn die Anzahl der gesetzten Eingangsvariablen ungerade ist. Symbol:* $\oplus$.