Modellierung und Analyse arithmetikorientierter eFPGA-Architekturen

Von der Fakultät für Elektrotechnik und Informationstechnik der
Rheinisch-Westfälischen Technischen Hochschule Aachen
zur Erlangung des akademischen Grades eines
Doktors der Ingenieurswissenschaften
genehmigte Dissertation

vorgelegt von
Master of Science

Bernd Neumann

aus Kerpen

Berichter: Universitätsprofessor Dr.-Ing. Tobias G. Noll
Universitätsprofessor Dr. rer. nat. Rainer Leupers

Tag der mündlichen Prüfung: 20.07.2010

Inhaltsverzeichnis

Abstract 1

Kurzfassung 3

1 Einleitung 5

2 FPGA-Architekturen 9

2.1 Allgemeines 9

2.2 Aufbau und Architekturmerkmale 10

2.2.1 Konfigurationsspeicher 10

2.2.2 Grundlegende FPGA-Architektur 10

2.2.3 Logikelement 12

2.2.4 Routing-Switch 14

2.2.5 Connection-Box 16

2.3 Implementierung von Anwendungen auf FPGAs 17

3 Anwendungsklassenspezifische eFPGA-Architekturen 19

3.1 Motivation 19

3.2 Prozessor-eFPGA-Kopplung 21

3.3 Anwendungsklassenspezifische Optimierung 22

3.3.1 Gesamtarchitektur Prozessor-eFPGA 22

3.3.2 Granularität der Accelerator-Architektur 22

3.3.3 Anforderungen an die Verbindungsarchitektur 23

3.3.4 Kommerzielle FPGAs 26

3.4 Beispielarchitekturen anwendungsklassenspezifischer FPGAs 27

3.4.1 Template-based embedded reconfigurable unit (Univ. Eindhoven) 27

3.4.2 Chimaera (Univ. Washington) 29

3.4.3 PiCoGA (Univ. Bologna) 30

3.4.4 SoC mit Tensilica XTensa und M2000 FlexEOS eFPGA (STM) 31

3.4.5 Bewertung 33

3.5 Entwurfswerkzeuge 34

3.5.1 VPR-konforme Architektur 34

3.5.2 Weitere Programme für das Mapping ... 36
3.5.3 VLSI-Entwurf ... 37
3.6 Optimierungspotenzial ... 38
3.7 Zusammenfassung ... 42
4 Parametrisierbare arithmetikorientierte eFPGA-Architektur ... 45
4.1 Übersicht ... 45
4.2 Arithmetikorientiertes Architektur-Template ... 46
4.2.1 Übersicht ... 46
4.2.2 Cluster ... 47
4.2.3 Logikelement ... 50
4.2.4 Routing-Switch ... 55
4.2.5 Connection-Box ... 57
4.2.6 Konfigurationsspeicher ... 59
4.2.7 Sonstiges ... 65
4.3 Automatisierte eFPGA-Implementierung ... 66
4.4 Konfiguration und Simulation ... 67
4.5 Zusammenfassung ... 70
5 Implementierung und Modellierung ... 73
5.1 Übersicht ... 73
5.2 Schaltungstechnische Realisierung ... 73
5.3 Implementierte Architekturen ... 74
5.4 Architektur 1: Parametrisierbare Standardarchitektur ... 75
5.4.1 Architektur ... 75
5.4.2 VLSI-Implementierung ... 77
5.4.3 Einfluss der Flexibilität der Connection-Box auf die Effizienz ... 80
5.5 Architektur 2: Arithmetikorientierte Architektur mit shared-SRAMs ... 82
5.5.1 Architektur ... 82
5.5.2 VLSI-Implementierung ... 84
5.6 Architektur 3: Parametrisierbare arithmetikorientierte Architektur ... 87
5.6.1 Architektur ... 87

5.6.2 VLSI-Implementierung 89
5.6.3 Bewertung der physikalischen Implementierungskosten 90
5.6.4 Konfiguration des eFPGA-Makros 93
5.6.5 Verwendung des eFPGAs als ASIP-Accelerator 94
5.7 Vergleich der implementierten Architekturen 96
5.8 Modellbasierte Optimierung 98
5.8.1 Implementierung und Charakterisierung der Basiszellen 98
5.8.2 Kernlogik 98
5.8.3 Dedizierter Routing-Block 100
5.8.4 SRAM-Zellen 101
5.8.5 Registerstufe 103
5.8.6 Switch-Points des Routing-Switch 103
5.8.7 Tristate-Treiber der Connection-Box 105
5.8.8 Taktnetz 105
5.8.9 ATE-Modell des eFPGAs 106
5.8.10 Ergebnisse 107
Zusammenfassung 119
Anhang 121
Literatur 133
Lebenslauf 139

Abstract

Digital signal processing systems typically consist of a data flow oriented and a control oriented part. Processors with reconfigurable accelerators are an attractive platform for implementing complex algorithms flexibly and at low physical implementation costs. The architecture of the accelerator has significant influence on the efficiency of the algorithms' implementation. Several commercial and academic projects suggest that fine-grain reconfigurable FPGAs, so-called eFPGAs, are a well-suited architecture for this purpose. However, these projects typically focused on conventional FPGA architectures.

In this thesis, an eFPGA architecture featuring a power- and area-efficiency significantly higher than conventional FPGAs is presented. The architecture is designed with respect to the typical requirements of arithmetic data paths, like high locality of the signal flow. To allow for a systematic analysis, the architecture is described as a general, parametrisable template. By carefully choosing the parameters, the efficiency of the eFPGA can be optimised for given arithmetic-oriented applications.

Based on a generic description, three exemplary architecture alternatives implemented as physically optimised VLSI macros in 180nm- and 130nm-technologies are presented. The design methodology allows for passing certain architectural parameters to the hardware description of the macro. By means of an automatic design flow it is possible to analyse the influence of architecture parameters on the VLSI implementation with moderate effort. The implemented macros are used to analyse the potential for optimisation for arithmetic oriented eFPGAs quantitatively. For this, exemplary operators are implemented both on the eFPGAs and on commercially available FPGAs.

Finally, a model based analysis of the conceived eFPGA architecture is presented in which the physical implementation costs of an eFPGA based on the aforementioned template are modelled as mathematic equations. This is achieved by implementing and characterising all key components like logic elements and configurable switches in a 90nm technology.

By means of exemplary operations that are mapped to the eFPGA, the results are discussed regarding their subsumption in the design space, and the efficiency of the different architecture variants is evaluated.

Based on the implementations and the model based results, a potential for optimisation with respect to power and area efficiency in mW/MOPS and MOPS/mm^2 of up to one order of magnitude compared to commercial standard FPGAs is shown up.

Kurzfassung

Systeme der digitalen Signalverarbeitung bestehen typischerweise aus einem datenfluss-orientierten und einem kontrollflussorientierten Anteil. Um anspruchsvolle Algorithmen flexibel und mit geringen physikalischen Kosten implementieren zu können, stellen Prozessoren mit rekonfigurierbarem Accelerator eine attraktive Plattform dar. Die Architektur des Accelerators hat einen wesentlichen Einfluss auf die Effizienz, mit der die Algorithmen implementiert werden können. In verschiedenen kommerziellen und akademischen Projekten wurden eingebettete, feingranulare rekonfigurierbare FPGAs, so genannte eFPGAs, als geeignete Architektur identifiziert. In der Regel wurden dabei herkömmliche FPGA-Architekturen betrachtet.

Im Rahmen dieser Arbeit wird eine eFPGA-Architektur vorgestellt, die im Bereich arithmetikorientierter Anwendungen eine deutlich höhere Verlustleistungs- und Flächen-effizienz erreicht als herkömmliche FPGAs. Die Architektur ist unter Berücksichtigung der Anforderungen von Arithmetikdatenpfaden, wie z.B. der in diesem Bereich typischen hohen Lokalität der Verbindungssignale, konzipiert. Um eine methodische Untersuchung zu ermöglichen, wird die Architektur als allgemeines, parametrisierbares Template beschrieben. Durch geeignete Wahl der Parameter kann die Effizienz des eFPGAs für entsprechende Arithmetikanwendungen optimiert werden.

Auf Basis der generischen Architekturbeschreibung werden exemplarisch drei Architektur-varianten vorgestellt, die als physikalisch optimierte VLSI-Makros in 180nm- bzw. 130nm-Halbleitertechnologien implementiert wurden. Die verwendete Entwurfsmethodik ermöglicht es dabei, bestimmte Architekturparameter als Freiheitsgrad in der Hardware-Beschreibung eines Makros anzugeben. Durch die Verwendung einer automatisierten Entwurfsumgebung kann mit moderatem Aufwand der Einfluss von Architekturparametern auf die VLSI-Implementierung untersucht werden. Anhand der implementierten Makros wird das Optimierungspotenzial arithmetikorientierter eFPGAs quantitativ analysiert und diskutiert. Dazu werden exemplarische Operatoren sowohl auf den entworfenen eFPGAs als auch auf kommerziell verfügbaren FPGAs implementiert.

Abschließend erfolgt eine modellbasierte Betrachtung des konzipierten eFPGA-Architektur-Templates. Es wird dargestellt, wie sich die physikalischen Implementierungskosten eines eFPGAs auf Basis des Templates als mathematische Modellgleichungen formulieren lassen. Dazu wurden alle wesentlichen Kernkomponenten wie Logikelemente und konfigurierbare Schalter in einer 90nm-Technologie implementiert und charakterisiert.

Anhand exemplarisch abgebildeter Operationen erfolgt eine Einordnung der Ergebnisse in den Entwurfsraum und eine Bewertung der Effizienz für die verschiedenen Architektur-varianten. Auf Basis der implementierten und modellierten eFPGA-Architekturen wird in dieser Arbeit ein Optimierungspotenzial bezüglich der Verlustleistungs- und Flächeneffizienz

in mW/MOPS bzw. MOPS/mm^2 von jeweils bis zu einer Größenordnung gegenüber einem kommerziellen Standard-FPGA aufgezeigt.

1 Einleitung

Viele Anwendungen der digitalen Signalverarbeitung haben einen zunehmenden Bedarf an hoher Rechenleistung, um immer komplexer werdende Algorithmen implementieren zu können. Gleichzeitig führt der Trend zu mobilen Geräten zu der Notwendigkeit, den Energiebedarf möglichst stark zu reduzieren. Neuartige Geräte wie so genannte Smartphones, welche die Funktionalität von Mobiltelefon, Videoabspielgerät, mobilem PC mit Internetzugang und weiteren Merkmalen verbinden, stellen Anforderungen an die Rechenleistung, Energieeffizienz und Flexibilität der Hardware, die von diskreten Prozessoren nicht mehr erfüllt werden können. Ein wesentlicher Trend der letzten Jahre ist daher die Integration verschiedener Hardware-Blöcke in einem so genannten SoC (System on Chip). Ein heterogenes SoC kann z.B. Prozessorkerne, dedizierte Hard-Makros, Speicher und konfigurierbare Hardware auf einem einzelnen Chip kombinieren. Durch geeignete Partitionierung des Systems und Abbilden der Systemblöcke auf entsprechende Hardware-Blöcke kann gegenüber monolithischen Hardware-Architekturen eine wesentlich effizientere Implementierung von Anwendungen der digitalen Signalverarbeitung erreicht werden. Aus diesem Grund kommt einer effizienten Kombination unterschiedlicher Architekturblöcke eine immer größere Bedeutung zu. Diese Aufgabenstellung bezeichnet man als Entwurfsraumexploration (englisch Design-Space-Exploration) [12]. Das Ziel der Entwurfsraumexploration ist die Abbildung von Systemblöcken auf diejenigen Hardware-Blöcke, die bezüglich ihrer Flexibilität und Performance am besten geeignet sind. Eine typischerweise gewählte Partitionierung ist dabei die Abbildung von kontrollflussorientierten Anteilen auf einen Prozessorkern, während datenflussorientierte Anteile auf dedizierte Acceleratoren ausgelagert werden. Änderungen in den Systemspezifikationen können dazu führen, dass sich die Anforderungen an die Hardware-Blöcke ändern. Während Prozessorkerne in der Regel flexibel genug sind, solche Änderungen zu adaptieren, kann die Verwendung dedizierter Beschleuniger-Makros bereits durch geringfügige Abweichungen in den Spezifikationen beeinträchtigt sein. Abhilfe können hier rekonfigurierbare Acceleratoren schaffen, die zur Laufzeit des Systems in ihrer Funktionalität geändert werden können. Bei einem eingebetteten feingranularen Accelerator, der auf Bitebene konfiguriert wird, spricht man von einem embedded FPGA (eFPGA).

Kommerziell erhältliche und in der Forschung betrachtete eFPGAs basieren häufig auf den Architekturen von diskreten Standard-FPGAs (Field-Programmable-Gate-Arrays). Diese sind für eine große Vielfalt von Anwendungen sowohl kontrollflussorientierter als auch datenflussorientierter Art optimiert. Für das oben beschriebene Anwendungsszenario wird das eFPGA jedoch speziell für die Abbildung verschiedener Arithmetikanwendungen (eine so genannte arithmetikorientierte Anwendungsklasse) verwendet. Eine Optimierung für möglichst allgemeine Verwendbarkeit führt dazu, dass die Effizienz solcher Architekturen für eine arithmetikorientierte Anwendungsklasse deutlich geringer ausfällt, als dies theoretisch

möglich wäre. Verschiedene Forschungsarbeiten hatten in der Vergangenheit die Optimierung von Standard-FPGA-Architekturen für Arithmetikanwendungen zum Gegenstand. Durch die grundsätzlichen Nachteile solcher Standardarchitekturen sind jedoch nur moderate Steigerungen der Effizienz erreichbar.

Ein weiterer wichtiger Aspekt der Optimierung von eFPGA-Architekturen ist die Möglichkeit, effiziente VLSI-Implementierungen von eFPGA-Makros durchführen zu können. Je nach Anwendungsklasse können verschiedene Optimierungskriterien zum Einsatz kommen. Dies führt dazu, dass der Aufbau eines eFPGA-Makros stark von den gewählten Architekturparametern abhängt. Für eine effiziente Implementierung des Makros ist es unabdingbar, einen physikalisch optimierten Entwurfsstil zu wählen. Im Bereich der Forschung werden u.a. Implementierungen basierend auf Standardzell-Synthese vorgeschlagen, um Änderungen der Architekturparameter adaptieren zu können [29][71]. Die gegenüber physikalisch optimierten Implementierungen geringere Effizienz dieses Entwurfsstils führt jedoch dazu, dass die Effizienz selbst bei starker Optimierung der Architekturparameter für eine Anwendungsklasse bestenfalls die Effizienz eines Standard-FPGAs erreicht. Andererseits ist ein vollständig manueller Entwurf angepasster eFPGA-Makros äußerst zeitaufwändig. Abhilfe können hier nur spezielle Entwurfstechniken schaffen, die den Aufbau der Makros teilweise automatisieren, ohne dabei die Nachteile einer Standardzell-Implementierung zu haben.

In der vorliegenden Arbeit wird eine neuartige eFPGA-Architektur vorgeschlagen, die auf einem parametrisierbaren Architektur-Template basiert. Das Template ist für die Verwendung in einer arithmetikorientierten Anwendungsklasse konzipiert. Ein wesentlicher Aspekt ist dabei die Verringerung der Komplexität des Verbindungsnetzes, das bei Standard-FPGAs einen Großteil der Flächen- und Verlustleistungsbilanz bestimmt. Durch gezieltes Anpassen der Logikelement- und Verbindungsarchitektur für die Anforderungen arithmetikorientierter Anwendungen kann eine gegenüber Standard-FPGAs deutlich gesteigerte Flächen- und Verlustleistungseffizienz erreicht werden. Eine effiziente VLSI-Implementierung der so definierten eFPGA-Makros wird durch einen automatisierten Entwurfsablauf erreicht, der die hohe Regularität der eFPGA-Architektur ausnutzt, um bei moderater Entwurfskomplexität besonders günstige physikalische Implementierungskosten zu erreichen. Dazu werden die Grundbausteine des eFPGAs als handoptimierte Layouts aufgebaut, während das Zusammensetzen des gesamten Makros durch ein entsprechendes Entwurfswerkzeug erfolgt. An dieser Stelle sei zur Begriffsbildung darauf hingewiesen, dass die häufig anzutreffende Bezeichnung „FPGA-Entwurfstechnik“ zweideutig ist. Häufig bezeichnet man damit das Abbilden einer Anwendung auf ein FPGA. Dieses ist nicht zu verwechseln mit der VLSI-Implementierung des FPGA selbst, wie sie in der vorliegenden Arbeit beschrieben wird.

Auf Basis der aus den Layouts gewonnenen Erkenntnisse wird ein mathematisches Modell der parametrisierbaren eFPGA-Architektur vorgestellt, das die physikalischen Implemen-

tierungskosten in Abhängigkeit von den Architekturparametern und der abgebildeten Anwendung beschreibt. Anhand dieses Modells ist es möglich, den Einfluss von Parametervariationen auf die Verlustleistungs- und Flächeneffizienz einer VLSI-Implementierung bereits in einem frühen Entwurfsstadium abzuschätzen. Anhand exemplarischer VLSI-Implementierungen und modellbasierter Betrachtungen kann gezeigt werden, dass die Verlustleistungs- und Flächeneffizienz von arithmetikorientierten eFPGA um jeweils bis zu eine Größenordnung gesteigert werden kann.

Die vorliegende Arbeit gliedert sich wie folgt:

In Kapitel 2 wird eine allgemeine Einführung zum Aufbau von FPGA-Architekturen gegeben. Neben dem grundlegenden Aufbau von FPGAs wird hier auch ein kurzer Überblick über den Ablauf der Implementierung von Anwendungen auf FPGAs gegeben.

Kapitel 3 beschreibt die Verwendung von anwendungsklassenspezifischen eingebetteten FPGAs als Acceleratoren für Prozessoren. Es werden vier exemplarische Prozessor-eFPGA-Architekturen aus dem akademischen Bereich beschrieben. Weiterhin werden existierende Entwurfswerkzeuge zur Untersuchung generischer FPGA-Architekturen vorgestellt.

In Kapitel 4 wird das im Rahmen dieser Arbeit konzipierte parametrisierbare eFPGA-Architektur-Template vorgestellt. Neben der detaillierten Diskussion des Templates mit allen Elementen und Parametern wird in diesem Abschnitt auch die automatisierte VLSI-Implementierung und Simulation der erstellten Makros beschrieben.

In Kapitel 5 wird anhand von drei im Rahmen dieser Arbeit exemplarisch implementierten eFPGA-Makros das Optimierungspotenzial gegenüber existierenden FPGA-Architekturen aufgezeigt. Weiterhin wird die modellbasierte Beschreibung der parametrisierbaren eFPGA-Architektur vorgestellt, die auf Basis des Architektur-Templates und exemplarischer VLSI-Implementierungen erarbeitet wurde. Anhand des Mappings verschiedener Operatoren auf die eFPGAs wird die Verlustleistungs- und Flächeneffizienz der konzipierten Architektur quantitativ bewertet.

2 FPGA-Architekturen

2.1 Allgemeines

FPGAs (Field-Programmable-Gate-Arrays) sind konfigurierbare Logikbausteine, bei denen die eigentliche Funktionalität erst nach der Fabrikation festgelegt wird. Anders als bei Prozessoren handelt es sich jedoch nicht um programmierbare, seriell arbeitende Rechnerarchitekturen mit einem festgelegten Befehlssatz, sondern um parallel rechnende Hardware, die auf Bitebene konfigurierbar ist.

FPGAs sind eine Weiterentwicklung der PALs (Programmable-Array-Logic), die erstmals 1977 kommerziell verfügbar waren. Der Idee zum Aufbau dieser Bausteine basiert auf der Tatsache, dass beliebige boolesche Funktionen in einer disjunktiven bzw. konjunktiven Normalform (DNF bzw. KNF) als zweistufige Verknüpfung mit UND- sowie ODER-Operationen beschrieben werden können. Entsprechend bestehen PALs aus zwei Ebenen von UND- bzw. ODER-Gattern, die über programmierbare Verbindungen verknüpft werden können. Um die Beschränkungen von PALs hinsichtlich der Realisierbarkeit komplexerer Schaltungen zu überwinden, wurden bei den CPLDs (Complex-Programmable-Logic-Devices) mehrere Instanzen von programmierbaren Feldern zu so genannten Makrozellen zusammengefasst, die untereinander durch ein programmierbares Verbindungsnetz kommunizieren. Aus diesen entwickelten sich schließlich die FPGAs, bei denen Basiszellen geringerer Komplexität mit der flexiblen Verbindungsarchitektur von CPLDs kombiniert werden.

Diskrete FPGA-Bausteine werden heute z.B. im Bereich der Prototypenentwicklung oder für Anwendungen mit kleinen bis mittleren Stückzahlen verwendet. Da FPGAs ohne weitere Fertigungsschritte verwendet werden können, sind die Entwicklungszyklen deutlich kürzer als bei ASICs (Application-Specific-Integrated-Circuits). Der Markt wird weitestgehend von den Firmen Altera [4] und Xilinx [72] dominiert. FPGAs werden typischerweise in den modernsten verfügbaren Technologien gefertigt. Durch die hohe Regularität des physikalischen Layouts sind sie besonders geeignet für die schnelle Adaption von neuen Halbleitertechnologien, so dass High-End-FPGAs häufig zu den ersten Chips gehören, die in einer neuen Technologie gefertigt werden. Aktuelle FPGAs werden typischerweise in 65nm- oder 40nm-Technologien gefertigt.

Im folgenden Kapitel wird der Aufbau typischer FPGA-Architekturen beschrieben. Weiterführende Informationen zu aktuellen FPGAs können z.B. [25] entnommen werden. Als Beispiel für ein relativ modernes kommerzielles FPGA wird in Anhang A kurz die Architektur eines Altera Stratix III vorgestellt.

2.2 Aufbau und Architekturmerkmale

2.2.1 Konfigurationsspeicher

Bei der Konfiguration eines FPGAs werden die entsprechenden Konfigurationsinformationen auf dem Baustein gespeichert. FPGAs lassen sich bezüglich der verwendeten Konfigurationsspeicher klassifizieren. Es wird dabei unterschieden zwischen Flüchtigkeit und Reversibilität der Programmierung. Eine flüchtige Programmierung der Speicher liegt vor, wenn der Speicherinhalt mit dem Abschalten der Spannungsversorgung verloren geht. Bei reversibler Programmierung kann die Konfigurationsinformation mehrfach verändert werden, während bei nichtreversibler Programmierung die einmal eingespeicherten Konfigurationsdaten nicht mehr verändert werden können. Die am häufigsten verwendete Form bei aktuellen FPGA-Bausteinen ist die flüchtige, reversible Programmierung. Dabei werden die Konfigurationsdaten auf dem FPGA in SRAMs gespeichert. Das Beschreiben der Konfigurationsspeicher muss daher bei jedem Einschaltvorgang erneut erfolgen. Dazu werden die Konfigurationsdaten typischerweise in einem externen Flash-Baustein gespeichert. Es existieren auch FPGAs, bei denen die Konfigurationsdaten direkt in Flash-Speichern im Baustein abgelegt werden (nichtflüchtig, reversibel). Weiterhin gibt es Bausteine, die durch Verwendung von so genannten Antifuse-Technologien nichtflüchtig und nichtreversibel programmierbar sind. Diese Speicherform hat insbesondere bei Anwendung in Bereichen mit hoher Strahlenbelastung (z.B. bei Satelliten) Vorteile, da die Strahlungstoleranz wesentlich größer ist als bei SRAM-basierten FPGAs.

Typischerweise wird die Konfiguration eines reversibel programmierbaren FPGAs direkt nach dem Einschalten in den Baustein geladen und dann nicht mehr verändert. Es gibt jedoch spezielle Anwendungen, bei denen die Konfiguration des Bausteins zur Laufzeit geändert wird. In diesem Fall spricht man von dynamischer Rekonfiguration. Auf diese Weise können z.B. mehrere Algorithmen im Zeitmultiplex auf dem FPGA ausgeführt werden. Jeder einzelne Algorithmus kann dabei das komplette FPGA belegen. Eine Sonderform der dynamischen Rekonfiguration ist die partielle dynamische Rekonfiguration, bei der nicht das komplette FPGA, sondern nur bestimmte Bereiche im laufenden Betrieb umkonfiguriert werden.

2.2.2 Grundlegende FPGA-Architektur

FPGAs bestehen aus konfigurierbaren, feingranularen Logikelementen (LE), die über ein konfigurierbares Verbindungsnetz miteinander vernetzt sind. Connection-Boxes (CB) verbinden die Logikelemente mit dem globalen Verbindungsnetz. In diesem werden Verbindungen zwischen den horizontal und vertikal laufenden Verbindungskanälen durch Routing-Switches (RS) hergestellt. Jeder Routing-Switch besteht dabei aus einer Vielzahl von konfigurierbaren Verbindungspunkten (Switch-Points). Die Logikelemente bestehen in der Regel aus Lookup-Tables (LUT) mit vier bis sechs Eingängen und Flip-Flops. Lookup-Tables

unterschiedlicher Komplexität können zur Abbildung beliebiger boolescher Funktionen genutzt werden. Oftmals wird die konfigurierbare Logik durch dedizierte Logik (wie Volladdierer oder spezielle Gatter) ergänzt, um die Berechnung von bestimmten arithmetischen Funktionen zu beschleunigen. Die konfigurierbaren Verbindungsressourcen ermöglichen die Verbindung aller Logikblöcke untereinander. Abbildung 2.1 zeigt den grundlegenden Aufbau einer FPGA-Architektur basierend auf dem so genannten Island-Style. Dabei wird ein Logikelement mit zwei Connection-Boxes und einem Routing-Switch zu einer regulären „Insel“ zusammengefasst, die durch reguläres zeilen- und spaltenweises Wiederholen zum Aufbau des gesamten FPGA verwendet wird.

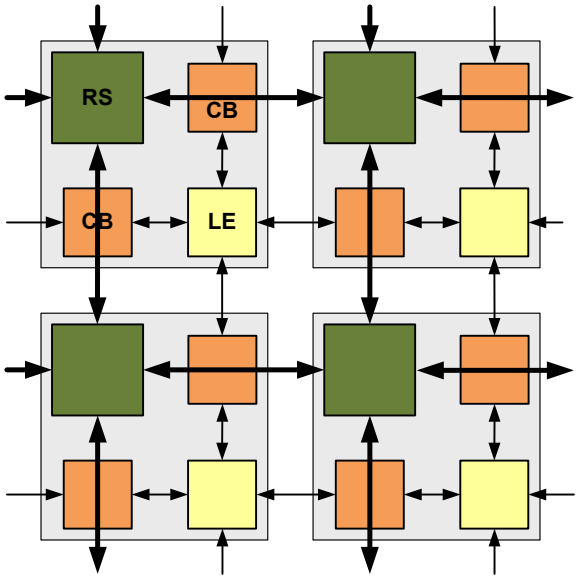


Abbildung 2.1: Island-Style-FPGA-Architektur

Um die Anforderungen an das globale Verbindungsnetz zu reduzieren, werden mehrere (typischerweise vier bis zehn) Logikelemente zu so genannten Clustern gruppiert, die über eine gemeinsame Connection-Box mit dem Verbindungsnetz verbunden werden. Abbildung 2.2 zeigt ein exemplarisches Cluster mit vier Logikelementen.

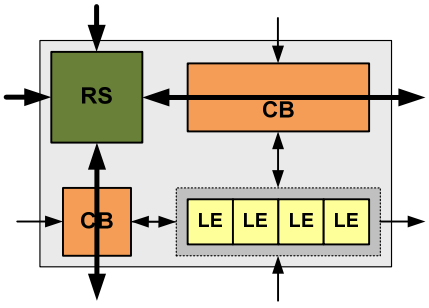


Abbildung 2.2: Cluster mit Logikelementen

Ein Nachteil der beschriebenen Island-Style-Architektur liegt in den Verbindungslängen zwischen den Logikelementen. Für kurze Verbindungen (z.B. benachbarte Logikelemente) fällt der Verbindungsweg über Connection-Boxes und Routing-Switch relativ kurz aus.

Werden Verbindungen zwischen weiter entfernten Logikelementen konfiguriert, so liegt gegebenenfalls eine Vielzahl von Routing-Switches im zeitkritischen Signalpfad. Um diesen Nachteil zu überwinden, verfügen moderne FPGA-Architekturen über so genannte segmentierte Verbindungsarchitekturen. Dabei werden neben den direkten Verbindungen zwischen zwei Routing-Switches (Manhattan-Distanz *L=1*) auch Verbindungen über mehrere Routing-Switches hinweg (Manhattan-Distanz *L>1*) vorgesehen. Beim Mapping von Anwendungen auf das FPGA identifiziert ein Routing-Algorithmus kritische Verbindungen zwischen weiter entfernten Logikelementen und weist diese den längeren Leitungen zu. Abbildung 2.3 illustriert das Konzept eines segmentierten Verbindungsnetzes anhand eines horizontalen Routing-Kanals. Im gezeigten Beispiel können Verbindungen über einen Distanz von einem oder drei Logikelementen hergestellt werden.

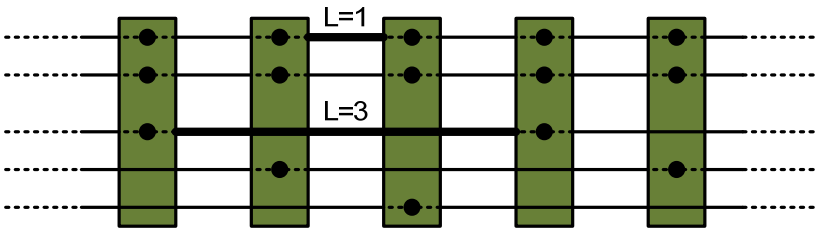


Abbildung 2.3: Segmentiertes Verbindungsnetz

Um Verschnitteffekte beim Mapping auf das FPGA zu vermeiden, werden die Segmente mit *L>1* gestaffelt („staggered“). Dadurch hat jeder Routing-Switch die gleiche Anzahl angeschlossener Segmente. Typischerweise werden verschiedene Verbindungslängen realisiert, um ein möglichst flexibles Routing zu ermöglichen.

Neben den Basiselementen (Logikelemente, Connection-Boxes, Routing-Switches) verfügen moderne FPGAs in der Regel über weitere dedizierte Architekturblöcke wie z.B. eingebettete Speicher, schwach konfigurierbare Multiplizierer oder kleine Prozessorkerne. Dadurch ist es möglich, auch komplexe Anwendungen komplett auf einem FPGA zu implementieren.

2.2.3 Logikelement

Das Kernstück der meisten modernen FPGA-Logikelemente ist eine speicherbasierte Tabelle, die so genannte Lookup-Table (LUT). Abbildung 2.4 zeigt eine exemplarische LUT mit drei Eingängen (x_0, x_1, x_2) und einem Ausgang (y).

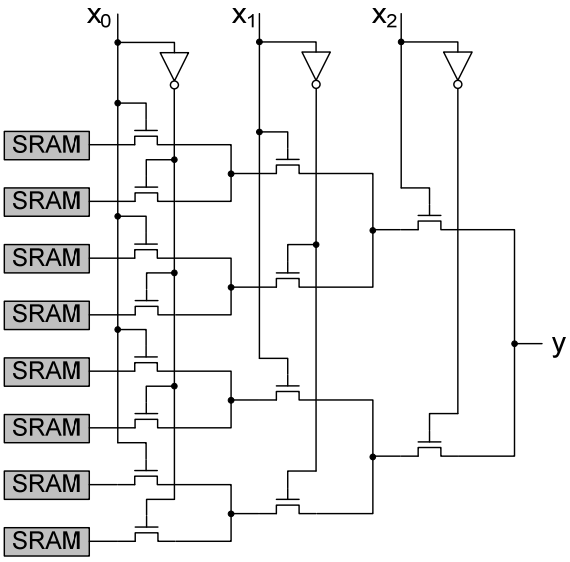


Abbildung 2.4: Lookup-Table mit drei Eingängen (LUT3)

Durch ein zusätzliches Register ist es möglich, auch Schaltwerke mit dem Logikelement zu realisieren. Der grundlegende Aufbau eines entsprechenden Logikelements wurde 1987 durch die Firma Xilinx in einem entsprechenden Patent beschrieben [16]. Abbildung 2.5 zeigt das dort beschriebene Schaltbild eines Logikelements bestehend aus einer kombinatorischen Logik (100) und einem Register (120) sowie einem schaltbaren Bypass für das Register (140). Das Patent wurde 2001 durch ein Gericht für ungültig erklärt.

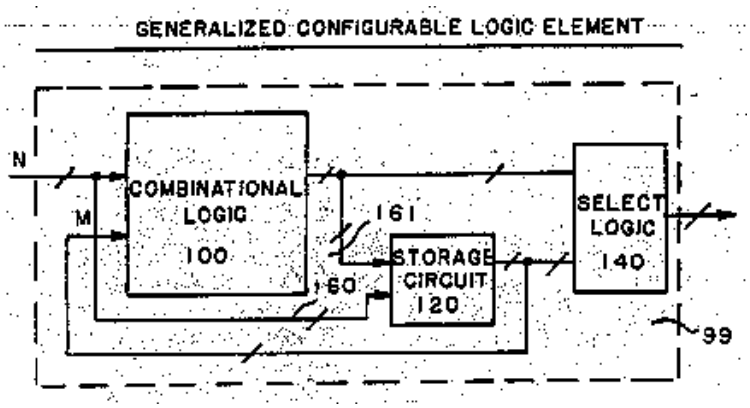


Abbildung 2.5: Logikelement gemäß Patent der Firma Xilinx von 1987 [16]

Der Aufbau des Logikelements ermöglicht es, beliebige boolesche Funktionen auf diesem zu realisieren. Die Anzahl der LUT-Eingänge k spielt eine entscheidende Rolle für die Effizienz beim Mapping von Funktionen auf das FPGA. Die resultierende Fläche der LUT bei einer physikalisch optimierten Implementierung berechnet sich auf Basis der Fläche eines einzelnen programmierbaren Schalters (z.B. Pass-Transistor) $A_{Schalter}$, der Fläche einer SRAM-Zelle A_{SRAM} und der Fläche eines Inverters $A_{Inverter}$ näherungsweise wie folgt:

$$A_{LUT,k} = 2 \cdot \left(2^k - 1\right) \cdot A_{Schalter} + 2^k \cdot A_{SRAM} + k \cdot A_{Inverter} \qquad (2.1)$$

Wird die Speichertabelle zu groß gewählt, steigen die physikalischen Implementierungskosten rapide an, da die Fläche der LUT gemäß der angegeben Formel exponentiell mit *k* wächst und zudem der zeitkritische Signalpfad länger wird. Wird die LUT hingegen zu klein gewählt, müssen bei der Abbildung von Operationen mit großer logischer Tiefe mehrere LUTs kaskadiert werden, so dass ebenfalls die effektiven Implementierungskosten steigen. Die Optimierung der LUT-Größe ist daher entscheidend für die Effizienz von Standard-FPGA-Architekturen. Ausführliche Untersuchungen z.B. in [1] haben ergeben, dass LUTs mit vier bis sechs Eingängen optimal im Hinblick auf das AT-Produkt (Fläche · Laufzeit) sind.

Neben einer Lookup-Table und einem Register enthalten Logikelemente moderner FPGA-Architekturen häufig zusätzliche Strukturelemente zur effizienteren Implementierung häufig verwendeter Operationen. Ein typisches Beispiel dafür sind dedizierte Carry-Chains, die eine schnelle Übertragsweiterleitung bei Carry-Propagate-Addierern (CPA) erlauben. Da die Laufzeiten von FPGAs durch das Verbindungsnetz dominiert werden, lassen sich häufig verwendete Additionsoperationen mit deutlich kürzeren Laufzeiten durch die Carry-Chains realisieren. Abbildung 2.6 zeigt schematisch die Verknüpfung mehrerer Logikelemente durch dedizierte Blöcke zur Übertragsberechnung (in der Abbildung mit cy gekennzeichnet).

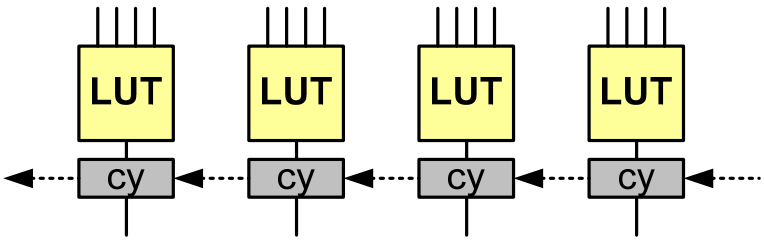


Abbildung 2.6: Dedizierte Carry-Chain

2.2.4 Routing-Switch

Routing-Switches sind eine zentrale Komponente von FPGA-Architekturen. Sie ermöglichen es, Signale auf den sich kreuzenden horizontalen und vertikalen Leitungen des globalen Verbindungsnetzes frei zu verschalten. Abbildung 2.7 zeigt den grundsätzlichen Aufbau eines Routing-Switch am Beispiel einer so genannten Disjoint-Architektur. Bei dieser Anordnung werden zwischen den sich kreuzenden horizontalen und vertikalen Leitungen nur auf der Haupt- bzw. Nebendiagonale konfigurierbare Schalter, so genannte Switch-Points, vorgesehen. Untersuchungen haben gezeigt, dass diese Belegung für viele Anwendungen bereits ausreicht, um alle Signale innerhalb des Designs zu verbinden [35].

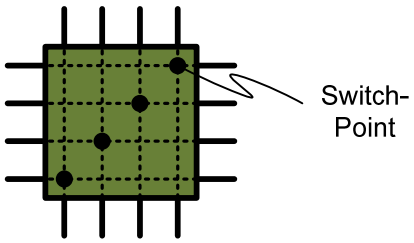


Abbildung 2.7: Disjoint-Routing-Switch

Die Flexibilität eines Routing-Switch lässt sich durch verschiedene Parameter beschreiben. Nach [10] kann die Flexibilität durch das Maß F_S ausgedrückt werden. F_S gibt dabei die Anzahl der möglichen Ausgänge an, zu denen ein beliebiges Eingangssignal weitergeleitet werden kann. Für den Disjoint-Routing-Switch aus Abbildung 2.7 gilt $F_S=3$. Eine weitere häufig angegebene Größe ist die Anzahl der zur Verfügung stehenden Verbindungsleitungen in horizontaler und vertikaler Richtung W.

Jeder Switch-Point besteht aus einer Anzahl konfigurierbarer Schalter, die Verbindungen zwischen den Ein-/Ausgängen des Switch-Points verschalten können. Typischerweise hat ein Switch-Point vier Ports, die entsprechend den Himmelsrichtungen mit Nord, Süd, Ost und West bezeichnet werden. Abbildung 2.8 zeigt einen exemplarischen Switch-Point, bei dem alle möglichen Verbindungen zwischen den vier Ein-/Ausgängen geschaltet werden können.

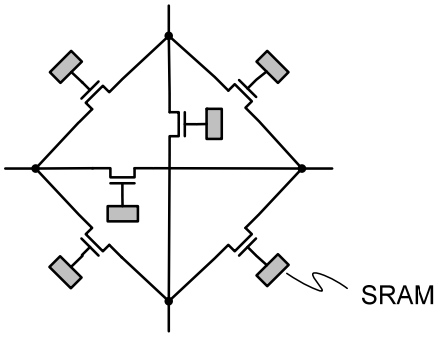


Abbildung 2.8: Exemplarischer Switch-Point

Neben den Architektureigenschaften eines Routing-Switch haben auch die schaltungstechnischen Implementierungsdetails einen großen Einfluss auf die Effizienz des Verbindungsnetzes von FPGAs. Zwei wichtige Einflussgrößen sind die Art der konfigurierbaren Schalter in den Switch-Points (z.B. Pass-Transistoren, Transmission-Gates) und die Verwendung aktiver Buffer-Elemente zur Signalauffrischung. Moderne FPGAs, die in 90nm- oder 65nm-Technologien gefertigt werden, verwenden typischerweise rein aktive Verbindungsnetze (auch als „fully-buffered“ bezeichnet). Das heißt, dass in jedem Switch-Point die ausgehenden Signale durch einen Buffer aufgefrischt werden. Abbildung 2.9 zeigt einen entsprechenden Switch-Point eines fully-buffered Interconnects.

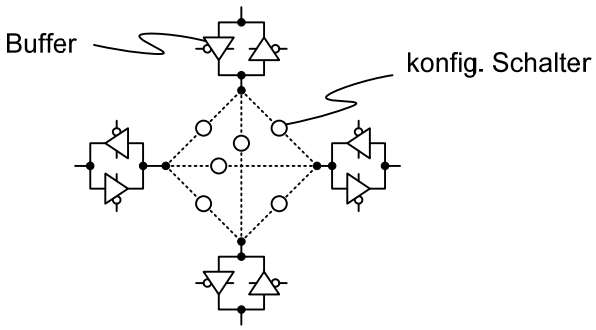


Abbildung 2.9: Fully-buffered Switch-Point

Das hohe Maß an Flexibilität, das Routing-Switches bieten, ist mit einem erheblichen Mehraufwand bezüglich der Siliziumfläche und Verlustleistung verbunden. Untersuchungen haben gezeigt, dass der konfigurierbare Interconnect (Routing-Switches und Connection-Boxes) weit mehr als die Hälfte der Fläche eines Island-Style-FPGAs ausmachen [23]. Zahlreiche Arbeiten [7][8][43] haben daher die Optimierungsmöglichkeiten von Routing-Switches sowohl auf Architektur- als auch auf Schaltungsebene untersucht.

2.2.5 Connection-Box

Ähnlich wie Routing-Switches bestehen Connection-Boxes aus einer Vielzahl von programmierbaren Schaltern, die Signale vom globalen Verbindungsnetz in die Logikelemente schalten können. Nach [10] kann als Maß für die Flexibilität einer Connection-Box die Anzahl der konnektierbaren Verbindungsleitungen pro LE-Eingang F_C verwendet werden. Dabei wird F_C häufig relativ in Bezug auf die Anzahl insgesamt vorhandener Verbindungsleitungen W angegeben. Abbildung 2.10 zeigt eine exemplarische Connection-Box mit $F_C=0,5W$ bei einem Kanal mit $W=6$ Leitungen.

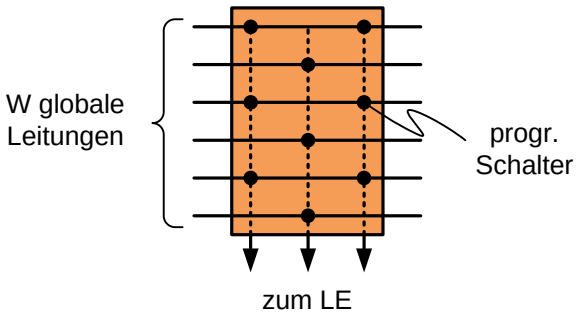


Abbildung 2.10: Connection-Box mit $F_C=0,5W$

Die programmierbaren Schalter realisieren einen Multiplexer, durch den nur jeweils eine aktive Leitung zum Logikelement geschaltet wird. Dabei können grundsätzlich passive Schalter (Pass-Transistoren, Transmission-Gates) oder aktive Buffer-Elemente verwendet werden. Abbildung 2.11 zeigt einen Ausschnitt einer Connection-Box für einen Logikelementeingang und $F_C=1W$.

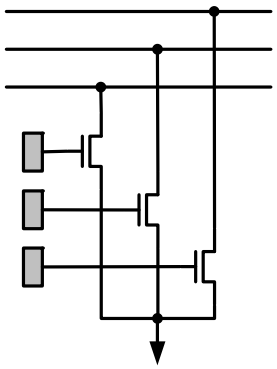


Abbildung 2.11: Programmierbare Schalter einer Connection-Box mit F_C=W

Wie die Routing-Switches stellen die Connection-Boxes bezüglich ihrer physikalischen Implementierungskosten einen erheblichen Aufwand in FPGAs dar. Durch geeignete Wahl von F_C kann dieser Implementierungsaufwand optimiert werden, indem man nur die tatsächlich benötigte Konnektivität vorsieht.

2.3 Implementierung von Anwendungen auf FPGAs

Um eine Anwendung auf ein FPGA abzubilden, sind zahlreiche komplexe Arbeitsschritte nötig. Dieser Vorgang wird häufig als FPGA-Entwurfstechnik bezeichnet und ist nicht zu verwechseln mit der in dieser Arbeit beschriebenen VLSI-Implementierung eines FPGAs. Kommerzielle Entwurfsumgebungen, die heute in der Regel von den FPGA-Herstellern selbst verfügbar sind, erreichen ein hohes Maß an Automatisierung für die meisten dieser Schritte, so dass der Entwurf von Schaltungen für FPGAs heute ein Prozess von moderatem Aufwand ist, der viele Ähnlichkeiten mit der Programmierung von Software in einer Hochsprache aufweist. Abbildung 2.12 stellt schematisch den Entwurfsablauf von der Schaltungsbeschreibung in einer Hardware-Beschreibungssprache (Hardware-Description-Language, HDL) bis zur Generierung der Programmierbits für den FPGA-Baustein dar. Die Schaltungsbeschreibung liegt typischerweise in VHDL vor und ist in der Regel technologieunabhängig.

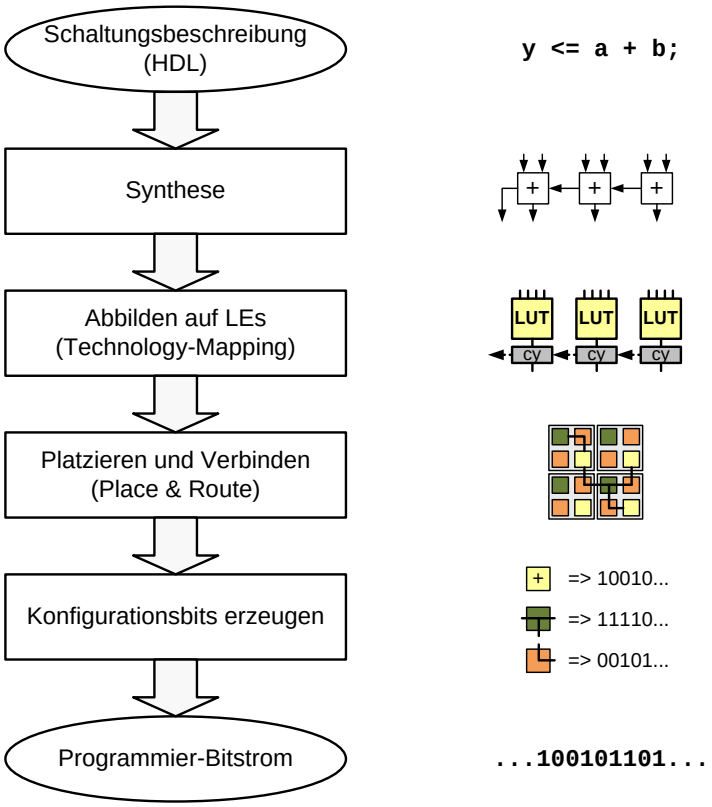


Abbildung 2.12: Implementierung von Anwendungen auf FPGAs

Zunächst wird bei der Synthese die beschriebene Schaltung auf logische Standardgatter abgebildet. Beim Technology-Mapping erfolgt das Abbilden auf die Funktionalität der Logikelemente. Dabei werden auch dedizierte Einheiten wie z.B. Carry-Chains berücksichtigt. Beim Place&Route werden alle allokierten Logikelemente so auf dem FPGA platziert, dass sich für das Verbinden untereinander eine möglichst vorteilhafte Anordnung ergibt. Beim eigentlichen Routing werden die zur Verfügung stehenden Verbindungskanäle allokiert. Im letzten Schritt erfolgt die Erzeugung der Programmierbits, mit denen der FPGA-Baustein schließlich konfiguriert wird.

Die von den Herstellern angebotenen Entwicklungsumgebungen (z.B. Altera Quartus II, Xilinx ISE) unterstützen dabei in der Regel alle Schritte von der Schaltungsbeschreibung (häufig mit zusätzlichem grafischen Frontend) bis zur Programmierung der Bausteine und stellen neben der eigentlichen Hardware eine Schlüsselkomponente zum kommerziellen Erfolg der angebotenen FPGAs dar.

3 Anwendungsklassenspezifische eFPGA-Architekturen

3.1 Motivation

Verschiedene Untersuchungen [26][44][68] haben ergeben, dass Prozessoren mit rekonfigurierbarem Accelerator eine attraktive Plattform für Anwendungen der digitalen Signalverarbeitung darstellen, da sie die Flexibilität einer Software-Implementierung mit der hohen Rechenleistung und dem geringen Energieumsatz paralleler Hardware verbinden. Anwendungen der digitalen Signalverarbeitung lassen sich typischerweise in einen kontrollflussorientierten und einen datenflussorientierten Anteil partitionieren. Der kontrollflussorientierte Anteil lässt sich effizient auf einen Prozessorkern mit geringer Komplexität abbilden. In der Regel sind die Anforderungen an die Rechenleistung gering, durch die Irregularität von Kontrollflussoperationen ist die sequentielle Abarbeitung im Prozessorkern jedoch einfach und flexibel implementierbar. Der datenflussorientierte Anteil hat in der Regel erheblich höhere Anforderungen an den Durchsatz und wird daher auf parallel organisierter Hardware verarbeitet. Eine Möglichkeit sind hier dedizierte Acceleratoren, wie sie z.B. in DSPs (Digitalen Signalprozessoren) zum Einsatz kommen.

Ein Vorteil der beschriebenen Partitionierung ist der gute Kompromiss aus Flexibilität und Performance bzw. Energieeffizienz. Durch die Verwendung eines Prozessorkerns können die implementierten Anwendungen als Software in einer Hochsprache entwickelt werden. Die bezüglich der Performance kritischen datenflussorientierten Anteile können bei geeigneter Kopplung zwischen Prozessorkern und Accelerator als so genannte Custom-Instructions in den Programmcode eingebunden werden.

Ein wesentlicher Nachteil bei der Verwendung von dedizierten Accelerator-Makros ist die geringe Flexibilität. Bei Änderungen in Standards oder Produkt-Updates sind gegebenenfalls vorhandene Acceleratoren nicht mehr geeignet, da sie die Anforderungen nicht erfüllen. In diesem Fall ist ein Redesign der Hardware nötig. Durch die exponentiell steigenden Entwicklungs- und Fertigungskosten bei Halbleiterschaltungen ist es jedoch erstrebenswert, dass bestehende Hardware einen möglichst langen Produktzyklus aufweist und im Idealfall als Plattform für eine ganze Gruppe von Anwendungen verwendet wird, so dass die Entwicklungskosten sich besser amortisieren. Eine attraktive Alternative zu dedizierten Accelerator-Makros sind daher rekonfigurierbare Acceleratoren, die nach der Fertigung durch Konfiguration in ihrer Funktionalität modifiziert werden können. Dabei kann es sich sowohl um grobgranulare Strukturen handeln, die z.B. auf Wortebene konfiguriert werden, als auch um feingranulare eFPGAs, die auf Bitebene konfiguriert werden. Die Firma Stretch [64] bietet z.B. mit dem S6000 eine Kombination aus einem VLIW-Prozessorkern und einer grobgranularen rekonfigurierbaren Einheit bestehend aus 4096 ALUs an. Embedded FPGAs werden z.B. kommerziell von den Firmen M2000 [40] und MENTA [42] angeboten.

Typischerweise basieren existierende eFPGA-Architekturen auf den Architekturen diskreter Standard-FPGAs.

Abbildung 3.1 zeigt eine exemplarische Prozessor-eFPGA-Architektur. Auf der linken Seite ist der Prozessorkern mit zugehörigem Befehlsspeicher dargestellt. Rechts im Bild ist das eFPGA gezeigt. Es verfügt über einen Konfigurationspeicher, in dem die Konfigurationsdaten für verschiedene Custom-Instructions (CI) abgelegt sind. Das eFPGA kann im dargestellten Beispiel sowohl direkt über die Prozessorregister als auch über den Prozessor- oder Peripheriebus adressiert werden. Der Befehlssatz des Prozessors wird durch die Custom-Instructions erweitert, die auf das eFPGA abgebildet werden. Ein Controller steuert die Kommunikation zwischen den beiden Architekturblöcken.

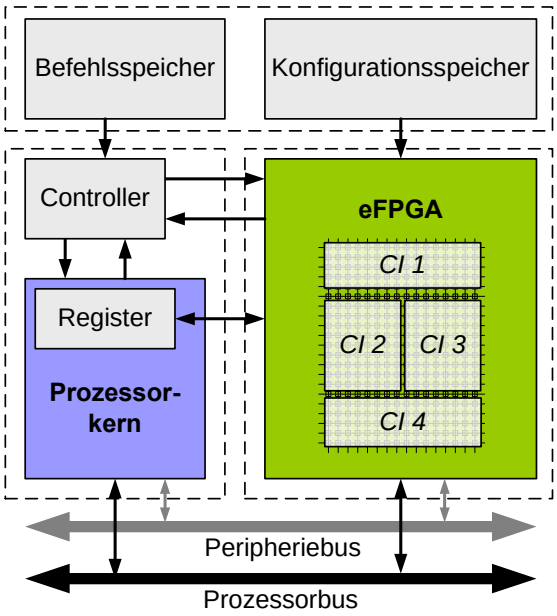


Abbildung 3.1: Exemplarische Prozessor-eFPGA-Architektur

Anders als bei diskreten FPGAs bietet diese Art der Architektur die Möglichkeit, die eFPGA-Architektur gezielt für die entsprechende Anwendungsklasse zu optimieren, da eine geeignete Partitionierung des Systems bereits vorausgesetzt werden kann. Dadurch ist es möglich, bislang bezüglich der physikalischen Implementierungskosten aufwändige rekonfigurierbare Einheiten deutlich effizienter zu gestalten.

Die Kopplung zwischen Prozessor und eFPGA ist Gegenstand der parallel zu der vorliegenden Arbeit entstandenen Co-Dissertation [67]. Die im folgenden Kapitel beschriebenen Aspekte werden dort ausführlich behandelt.

3.2 Prozessor-eFPGA-Kopplung

Gemäß der Klassifikation in [21] lassen sich Prozessor-eFPGA-Architekturen bezüglich ihrer Kopplung in drei Klassen unterteilen. Die drei Klassen unterscheiden sich bezüglich des Kopplungsgrades.

Die so genannte ARPU (Attached-Reconfigurable-Processing-Unit) stellt die loseste Anbindungsform dar. Dabei wird das eFPGA über den verhältnismäßig langsamen Peripheriebus zusammen mit Komponenten wie dem externen Speicher angebunden. Dadurch sind die Kommunikationskosten zwischen Prozessor und eFPGA relativ groß. Diese Form der Kopplung bietet sich daher an, wenn komplexere Operationen mit geringem Kommunikationsbedarf ausgelagert werden (z.B. kompletter Verschlüsselungsalgorithmus). Der Befehlssatz des Prozessorkerns muss dazu nicht verändert werden. Die Synchronisation zwischen Prozessorkern und eFPGA erfolgt vollständig in Software.

Die Anbindung als so genannter RC (Reconfigurable-Coprocessor) erfolgt über den schnelleren Prozessorbus. Das eFPGA ist enger an den Prozessorkern gekoppelt, so dass die Kommunikationskosten für das Auslagern von Operationen geringer sind. Für diese Form der Anbindung ist es erforderlich, den Instruktionssatz des Prozessorkerns so zu ändern, dass auch auf die Custom-Instructions auf dem RC zugegriffen werden kann.

Die engste Form der Anbindung liegt bei der so genannten RFU (Reconfigurable-Functional-Unit) vor. Dabei wird das eFPGA in die Pipeline des Prozessorkerns integriert. Die Kommunikation erfolgt direkt über den Registersatz des Prozessorkerns. Die enge Ankopplung erlaubt es, einzelne Operationen geringer Komplexität (z.B. Multiplikation mit angepassten Wortbreiten) auf das eFPGA auszulagern. Die Custom-Instructions können im Programm-Code einfach eingebunden werden, da sie eine direkte Erweiterung des Basisinstruktionssatzes darstellen. Synchronisationsmaßnahmen müssen in Hardware vorgesehen werden.

Abbildung 3.2 illustriert die verschiedenen Kopplungsmechanismen. Das eFPGA wird dabei jeweils als RFU, RC und ARPU integriert.

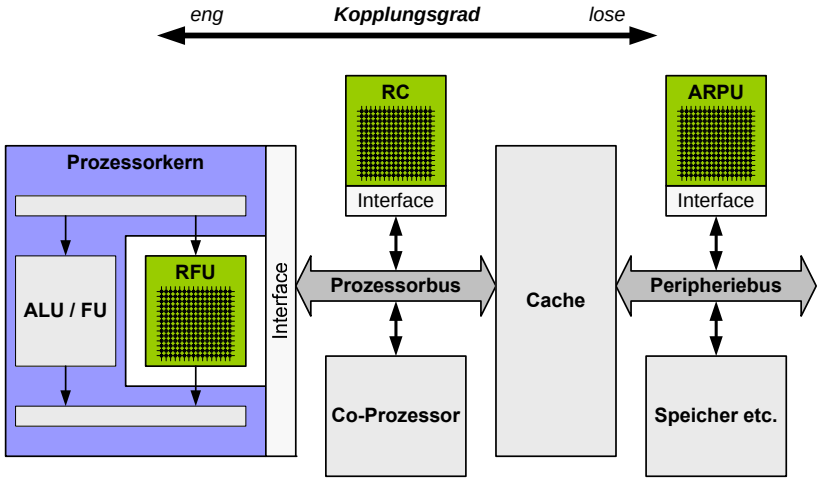


Abbildung 3.2: Kopplungsklassen Prozessor-eFPGA

3.3 Anwendungsklassenspezifische Optimierung

3.3.1 Gesamtarchitektur Prozessor-eFPGA

Die Partitionierung von Systemen der digitalen Signalverarbeitung in kontrollflussorientierten Anteil und datenflussorientierten Anteil erlaubt es, die jeweiligen Architekturblöcke, auf welche die Systemkomponenten abgebildet werden, hinsichtlich einer Anwendungsklasse zu optimieren [50]. Indem der Prozessorkern als ASIP (Application-Specific-Instruction-Set-Processor) konzipiert wird, kann er bezüglich seiner Komplexität für die Anforderungen von Kontrollanwendungen optimiert werden. Durch Anpassung des Befehlssatzes, der verfügbaren Recheneinheiten und der Wortbreiten der Datenpfade ergibt sich gegenüber einem General-Purpose-RISC-Prozessor eine deutliche Steigerung der Verlustleistungs- und Flächeneffizienz. Entsprechend kann die eFPGA-Architektur für die Abbildung von Datenflussoperationen optimiert werden. Typischerweise handelt es sich dabei um Arithmetikanwendungen, die sich bezüglich ihrer Anforderungen an die eFPGA-Architektur erheblich von Kontrollflussoperationen unterscheiden.

3.3.2 Granularität der Accelerator-Architektur

Bei den rekonfigurierbaren Acceleratoren unterscheidet man typischerweise zwischen grobgranularen (coarse-grain) und feingranularen (fine-grain) Architekturen. Grobgranulare Architekturen arbeiten auf Wortebene und verfügen in der Regel über einen eingeschränkten Satz von ausführbaren Elementaroperationen. Beispiele für grobgranulare rekonfigurierbare Einheiten sind der kommerziell verfügbare PACT XPP-III [55] oder die in [18] beschriebene RaPiD-Architektur (Reconfigurable Pipelined Datapath). Feingranulare Acceleratoren sind hingegen auf Bitebene konfigurierbar und verfügen anstelle von weniger flexiblen Prozessor-

elementen über frei konfigurierbare Logikelemente z.B. in Form von programmierbaren Speichertabellen (LUTs) und Registern. Feingranulare Architekturen wie die in dieser Arbeit konzipierte eFPGA-Architektur sind wesentlich flexibler, da sie nicht den befehlssatzbedingten Einschränkungen grobgranularer Elemente unterliegen. Ein wesentlicher Vorteil grobgranularer Architekturen liegt in der potenziell größeren Effizienz beim Mapping von Datenpfaden, die bezüglich ihrer Wortbreiten und Anforderungen an die Elementaroperationen gut auf die Architektur abgestimmt sind. In diesem Fall ist der geringere Mehraufwand für Verbindungsnetz und Prozessorelemente ein erheblicher Vorteil. Allerdings lassen sich Datenpfade häufig so optimieren, dass insbesondere intern nicht mehr zwangsläufig mit typischen Wortbreiten wie 4 Bit, 8 Bit oder 16 Bit gerechnet werden muss. In solchen Fällen können feingranulare Architekturen, bei denen nur die jeweils tatsächlich benötigte Anzahl an Logikelementen und Verbindungsleitungen allokiert wird, die effizientere Implementierungsform bieten. Bei der hier vorgestellten eFPGA-Architektur besteht zudem die Möglichkeit, als Kompromiss zwischen den beiden Ansätzen feingranulare Elemente zu verwenden, aber die bezüglich der Siliziumfläche teuren Konfigurationsspeicher mehrfach zu nutzen, so dass mehrere nebeneinander liegende Elemente identisch konfiguriert sind. Im Folgenden sollen nur die Optimierungsmaßnahmen für feingranulare Architekturen betrachtet werden.

3.3.3 Anforderungen an die Verbindungsarchitektur

Das konfigurierbare Verbindungsnetz ist bzgl. der Effizienz der kritische Punkt einer FPGA-Architektur. In [35] wird sein Anteil an der Gesamtfläche mit bis zu 90% angegeben. Ebenso dominiert es die Verlustleistungsbilanz. Abbildung 3.3 zeigt Verlustleistungsanalysen verschiedener FPGAs, die der Literatur der letzten Jahre entnommen wurden [6][20][23][62].

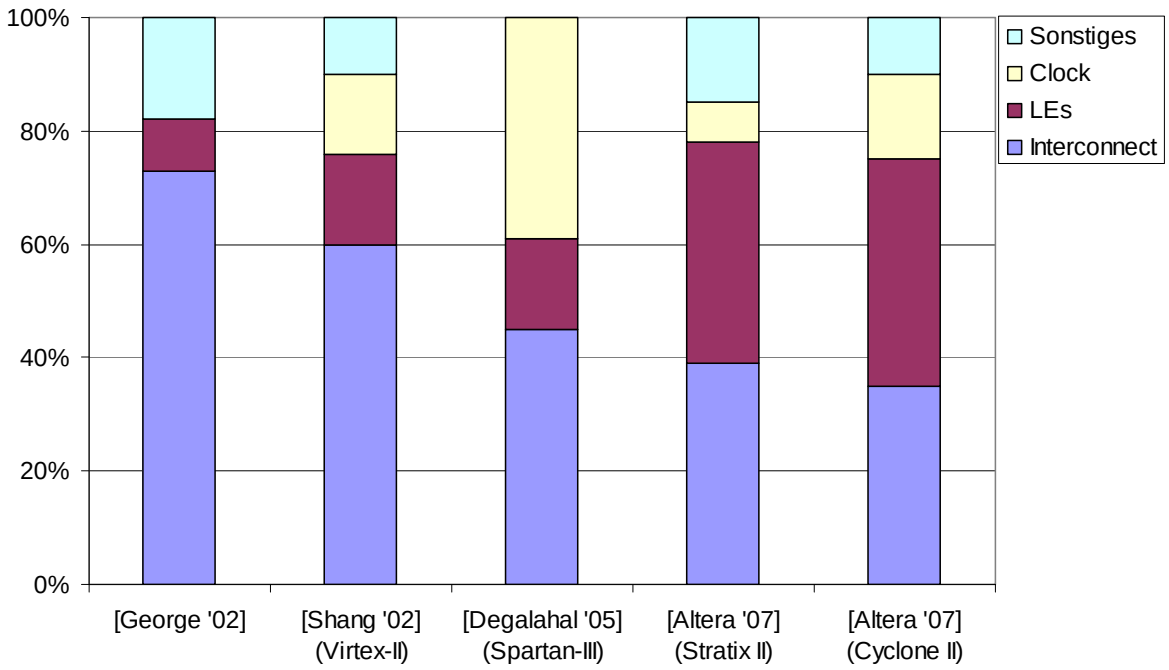


Abbildung 3.3: Verlustleistungsbilanz von FPGAs gemäß Literatur

Die Darstellung zeigt, dass der Anteil des Verbindungsnetzes an der Verlustleistungsbilanz erheblich ist, wenngleich er im Verlauf der Jahre gesunken ist. Ein wesentlicher Grund dafür ist der Trend der Hersteller, die Logikelemente immer komplexer zu machen (z.B. von LUTs mit vier Eingängen im Stratix I zu LUTs mit sechs Eingängen im Stratix II). Dadurch ist es möglich, beim Technology-Mapping kombinatorische Blöcke mit größerer logischer Tiefe auf die Logikelemente abzubilden und so weniger LEs über das Verbindungsnetz kaskadieren zu müssen. Die hohe Komplexität des Interconnects resultiert aus den a priori unbekannten Anforderungen der auf das FPGA abzubildenden Anwendungen. Da diskrete Standard-FPGAs zur Implementierung quasi beliebiger Digitalschaltungen verwendet werden können, wird für die Konnektivität ein sehr hohes Maß an Flexibilität vorgehalten. Schränkt man die Anwendungsklasse auf Arithmetikanwendungen ein, ergibt sich allerdings ein erhebliches Optimierungspotenzial. Grund dafür ist die Art des Signalflusses zwischen den Basiselementen eines auf das FPGA abgebildeten Signalflussgraphen. Gemäß Rent's Rule [33] besteht empirisch betrachtet ein fester exponentieller Zusammenhang zwischen der Anzahl der Gatter (bzw. Basiselemente) innerhalb einer Region und der Anzahl der Pins an deren Peripherie. Dieser Zusammenhang gilt sowohl für Random-Logik mit geringer Regularität als auch für Datenpfade mit hoher Regularität. Ein wesentlicher Unterschied besteht jedoch in der Verteilung der Verbindungslängen innerhalb der betrachteten Region. Während irreguläre Logik typischerweise eine Vielzahl unterschiedlicher Verbindungslängen zwischen den Basiselementen erfordert, werden bei regulären Datenpfaden hauptsächlich lokale Nachbarschaftsverbindungen sowie einige lange Leitungen für das Broadcasting von Operanden benötigt.

Die Verbindungsarchitektur ist also ein wesentlicher Angriffspunkt bei der Optimierung von FPGAs. Ein entsprechender Vergleich wurde z.B. in [30] für einen GPS-Korrelator durchgeführt. Die entsprechenden Histogramme der Verbindungslängen sind in Abbildung 3.4 qualitativ (links) und exemplarisch für den Korrelator quantitativ (rechts) dargestellt.

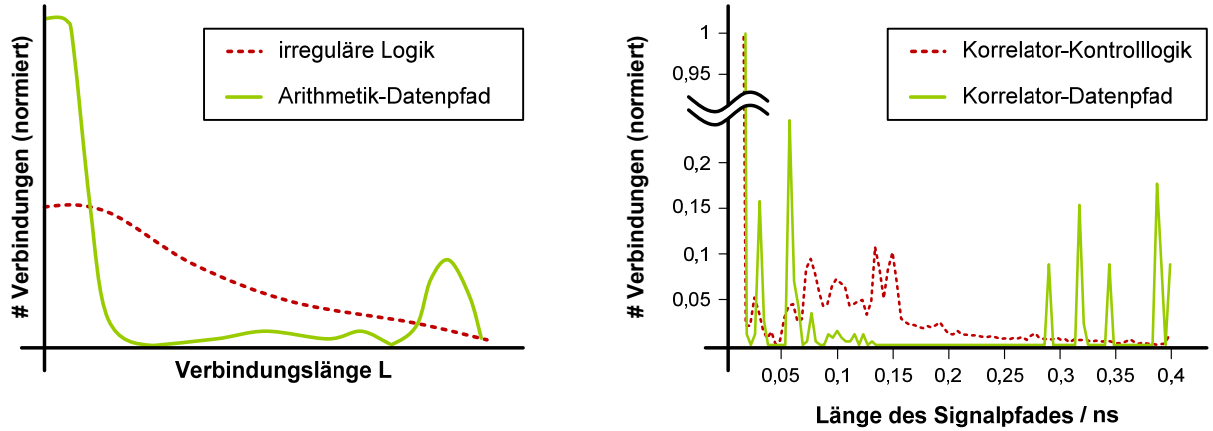


Abbildung 3.4: Qualitatives und quantitatives Histogramm der Verbindungslängen für irreguläre Logik und Arithmetikdatenpfade [30]

Da Standard-FPGAs für die Abbildung beliebiger Logik geeignet sein sollen, muss die Verbindungsarchitektur so flexibel sein, dass sich beliebige Verbindungslängen und Signalflussrichtungen realisieren lassen. Das führt dazu, dass hierfür große, komplexe Routing-Switches vorgehalten werden müssen.

Begrenzt man hingegen die Anforderungen auf Arithmetikanwendungen, so kann die Komplexität des Verbindungsnetzes deutlich reduziert werden. Obwohl die Anzahl der zu konfigurierenden Verbindungswege grundsätzlich in der gleichen Größenordnung wie bei Random-Logik liegt, handelt es sich typischerweise um direkte Nachbarschaftsverbindungen, die zudem häufig eine Vorzugsrichtung aufweisen. Mit diesem a priori bekannten Wissen über die Art der abzubildenden Anwendungen ist es möglich, einen Großteil der Verbindungen nicht über hochflexible Routing-Switches, sondern über dedizierte Routing-Ressourcen mit deutlich geringeren Implementierungskosten zu konfigurieren. Im Rahmen dieser Arbeit führt diese Idee zum Konzept der dedizierten Routing-Blöcke (DRB, siehe Kapitel 4.2.3). Voraussetzung für die effiziente Verwendung solcher dedizierter Routing-Ressourcen ist dabei eine Logikelement-Architektur, die sich gut für die Abbildung der regulären Basiselemente typischer Arithmetikdatenpfade eignet. Ist dies nicht der Fall, führt das erforderliche Kaskadieren von Logikelementen zum Durchbrechen des regulären Signalflusses. Daher ist es trotz des vergleichsweise geringen Anteils der Logikelemente an der Flächen- und Verlustleistungsbilanz entscheidend für die Effizienz der Gesamtarchitektur, auch die LE-Architektur entsprechend zu optimieren.

3.3.4 Kommerzielle FPGAs

Bei den aktuellen Generationen von FPGAs sind die großen Hersteller Altera und Xilinx inzwischen dazu übergegangen, die Bausteine in verschiedenen Ausprägungen anzubieten. Dazu ist die Architektur je nach Anwendungsbereich für allgemeine Logikanwendungen oder für Anwendungen der digitalen Signalverarbeitung optimiert. Effektiv beschränken sich die Anpassungen der Architektur auf einen geänderten Mix zwischen Logikelementen und verschiedenen dedizierten Hardware-Blöcken wie Speicher oder DSP-Blöcken mit dedizierten Multiplizierern. Tabelle 3.1 und Tabelle 3.2 zeigen eine Übersicht der verfügbaren Ressourcen bei aktuellen FPGA-Bausteinen von Altera und Xilinx. Die Angaben wurden den Datenblättern der Hersteller entnommen. Da der Aufbau der Logikelemente und DSP-Blöcke bei den beiden FPGA-Architekturen geringfügige Unterschiede aufweist, sind sie nicht für einen direkten Vergleich geeignet. Ein Vergleich der jeweiligen Ausprägungen innerhalb der FPGA-Familie zeigt jedoch deutlich, wie durch die Wahl der zur Verfügung stehenden Ressourcen die Bausteine für gewisse Anwendungsklassen optimiert wurden.

FPGA	Anwendungs-klasse	Logikelemente (ALMs)	ded. Speicher-blöcke (kBit)	DSP-Blöcke	Sonstiges
Stratix III L	allg. Logik	19.000 – 135.200	1.836 – 16.272	216 - 576	
Stratix III GX	*Kommunikation*	*noch nicht verfügbar*	*noch nicht verfügbar*	*noch nicht verfügbar*	*Multi-Gigabit-Transceiver*
Stratix III E	Signalverarbeitung	19.000 – 101.760	5.328 – 14.688	384 - 768	

Tabelle 3.1: Varianten des Altera Stratix III für verschiedene Anwendungsklassen

FPGA	Anwendungs-klasse	Logikelemente (Virtex-5-Slices)	ded. Speicher-blöcke (kBit)	DSP-Blöcke	Sonstiges
Virtex-5 LX	allg. Logik	4.800 - 51.840	1.152 – 10.368	32 – 192	
Virtex-5 LXT	Kommunikation	4.800 - 51.840	1.296 – 11.664	32 – 192	PCIe, Ethernet, Multi-Gigabit-Transceiver
Virtex-5 SXT	Signalverarbeitung	5.440 - 14.720	3.024 – 8.784	192 - 640	PCIe, Ethernet, Multi-Gigabit-Transceiver

Tabelle 3.2: Varianten des Xilinx Virtex-5 für verschiedene Anwendungsklassen

Abbildung 3.5 zeigt für alle verfügbaren FPGA-Bausteine der Serien Stratix III und Virtex-5 die Ausprägungen für allgemeine Logik bzw. Signalverarbeitung im Hinblick auf die Anzahl von Logikelementen, DSP-Blöcken und dedizierten Speicherblöcken. Die Positionierung in dem dreidimensionalen Raum entspricht der Zuordnung zu verschiedenen Anwendungs-klassen.

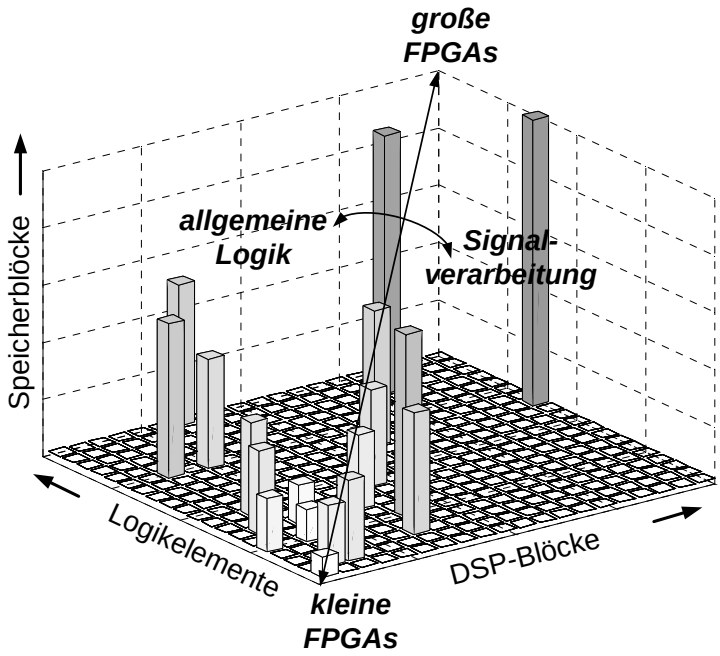


Abbildung 3.5: Ausprägungen der Altera Stratix III- und Xilinx Virtex-5-Bausteine für verschiedene Anwendungsklassen

3.4 Beispielarchitekturen anwendungsklassenspezifischer FPGAs

Im Folgenden werden einige Forschungsprojekte vorgestellt, welche den Entwurf einer anwendungsklassenspezifischen eFPGA-Architektur zum Gegenstand hatten. Dabei werden auch Arbeiten berücksichtigt, bei denen das FPGA nicht explizit für die Integration mit einem Prozessor entworfen wurde, sondern allgemein bezüglich einer Anwendungsklasse optimiert wurde.

3.4.1 Template-based embedded reconfigurable unit (Univ. Eindhoven)

In [34] wird eine an der Universität Eindhoven entwickelte FPGA-Architektur vorgestellt, die auf anwendungsklassenspezifisch optimierten Logikelementen beruht. Die LEs sind dabei speziell für Arithmetik, Random-Logik oder andere Anwendungsbereiche konzipiert worden. Abbildung 3.6 zeigt den verwendeten Logikblock für Arithmetikanwendungen. Es handelt sich dabei um ein Cluster mit vier Logikelementen, wobei jedes Logikelement aus einer LUT2 sowie Erweiterungen für konfigurierbare Invertierung und dedizierte Übertragsweiterleitung besteht.

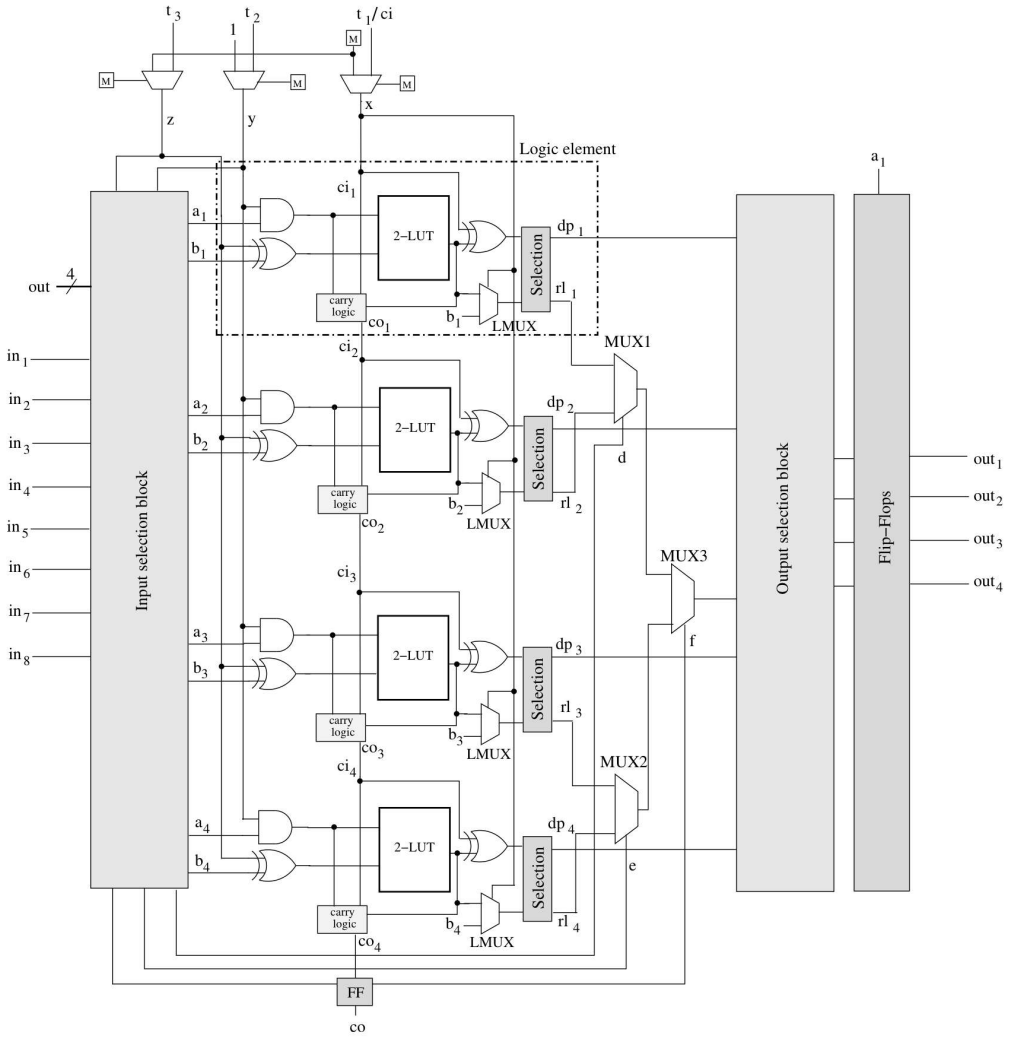


Abbildung 3.6: Arithmetikorientierter Logikblock [34]

Die Logikelemente sind über ein Verbindungsnetz im regulären Island-Style miteinander verbunden. Die Connection-Boxes verfügen über verschiedene Abgriffe für datenpfad-orientierte Busse und General-Purpose-Leitungen. Für die datenpfadorientierten Busse beträgt die Flexibilität $F_C=0,25W$. Dabei werden nebeneinander liegende LEs auch an benachbarte Verbindungsleitungen angeschlossen, so dass der Zugriff ähnlich wie bei einem Bus einem Wertigkeitsschema der Leitungen folgt. Die General-Purpose-Leitungen werden eingangsseitig mit $F_C=0,5W$ und ausgangsseitig mit $F_C=1W$ angeschlossen. Insgesamt stehen im Verbindungsnetz 46 Leitungen zur Verfügung. Die Komplexität des Verbindungsnetzes entspricht damit etwa dem kommerzieller Standard-FPGAs.

Für ein speicherorientiertes eFPGA wurde ein Testchip in einer 130nm-Standardzell-Technologie gefertigt. Die Gesamtfläche ist bei Normierung der vorhandenen Logikelemente auf LUT4-Äquivalente um 41% größer als die eines vergleichbaren Altera APEX 20KE. Dabei wird der Mehraufwand der Fläche bei der Standardzell-Synthese gegenüber einer physikalisch optimierten Implementierung gemäß Untersuchungen in [58] auf 42% abgeschätzt.

Die eFPGA-Architektur wurde anhand von Modellen evaluiert und mit kommerziellen Standard-FPGAs verglichen. Für das Mapping von Beispielanwendungen wurde ein modifizierter VPR-Designflow [9] verwendet (siehe auch Kapitel 3.5.1). Für Benchmarks mit Schwerpunkt auf Anwendungen der digitalen Signalverarbeitung konnte bei der arithmetik-orientierten Architektur eine Flächenreduktion des gesamten FPGA von 37% gegenüber einem Altera APEX 20KE erreicht werden.

3.4.2 Chimaera (Univ. Washington)

In [26] wurde die an der Universität Washington entwickelte Prozessor-eFPGA-Architektur Chimaera vorgestellt. Der rekonfigurierbare Accelerator besteht aus zeilenweise angeordneten feingranularen Logikelementen. Jedes Logikelement ist aus zwei LUT3 und einer dedizierten Übertragslogik aufgebaut. Der Accelerator ist als RFU (siehe Kapitel 3.2) an den Prozessor angekoppelt. Der Datenfluss innerhalb der RFU verläuft dabei von oben nach unten. Jede Zeile besteht aus 32 Logikelementen, die als so genannte Function-Slice betrachtet werden können, in der eine konfigurierbare Basisoperation durchgeführt wird. Mehrere aufeinander folgende Function-Slices können somit eine Custom-Instruction bestehend aus mehreren Basisoperationen realisieren. Abbildung 3.7 zeigt den Aufbau des Logikelements und der zeilenweise organisierten Verbindungsarchitektur. Das Verbindungsnetz ist segmentiert und verfügt über so genannte Longlines, die über eine gesamte Zeile von Logikelementen verlaufen.

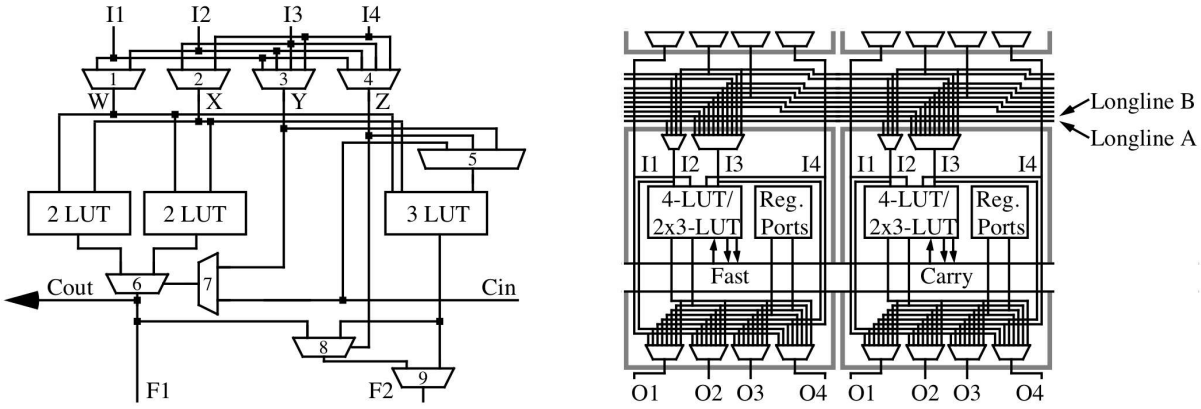


Abbildung 3.7: Logikelement und Verbindungsarchitektur der Chimaera-RFU [26]

Eine besondere Eigenschaft der Chimaera-RFU ist die Möglichkeit zur partiellen dynamischen Rekonfiguration. Ein spezieller Konfigurations-Cache erlaubt es, die RFU mit geringer Latenz komplett oder zeilenweise neu zu konfigurieren. Ein Testchip der RFU wurde in einer 0,5µm-CMOS-Technologie gefertigt. Aus den in Veröffentlichungen angegebenen Flächenwerten lässt sich eine LE-Dichte von ungefähr 159 Logikelementen pro mm^2 bei Skalierung auf eine 180nm-Technologie gemäß Anhang E ermitteln. Die LE-Dichte eines

Altera APEX 20K, der ebenfalls in einer 180nm-Technologie gefertigt wurde, beträgt ungefähr 102 LEs/mm^2.

Für die gesamte Prozessor-eFPGA-Architektur wurde anhand ausgewählter Benchmarks ein Vergleich mit einem Standard-MIPS-Prozessor durchgeführt. Dabei wurden je nach Anwendung Beschleunigungen um Faktor sieben bis 160 gegenüber der reinen Software-Lösung auf dem MIPS ermittelt. Eine detailliertere Gegenüberstellung zwischen der RFU und einer Standard-FPGA-Architektur ist in der Literatur nicht verfügbar.

3.4.3 PiCoGA (Univ. Bologna)

Das Pipelined Configurable Gate Array (PiCoGA) [39] wurde an der Universität Bologna entwickelt. Es ist ein rekonfigurierbares Array von Logikelementen bestehend aus bis zu 48 Reihen mit je 16 LEs. Die Logikelemente haben eine Granularität von 2 Bit. Ähnlich wie bei Chimaera sind die Zeilen des Arrays als Function-Slices organisiert, die jeweils eine elementare Basisoperation ausführen können. Durch Pipeline-Register zwischen den Zeilen ist es möglich, eine Sequenz von PiCoGA-Instruktionen parallel in einer Pipeline auszuführen. Jedes Logikelement besteht aus zwei LUT4, wobei die LUTs jeweils eine Granularität von 2 Bit haben, d.h. vier Eingänge und zwei Ausgänge. Durch Zusammenschalten lässt sich auch eine LUT6-Funktionalität abbilden. Zusätzlich ist eine dedizierte Übertragslogik vorhanden. Abbildung 3.8 zeigt ein Blockschaltbild der PiCoGa-Architektur bzw. der entsprechenden Basiszelle.

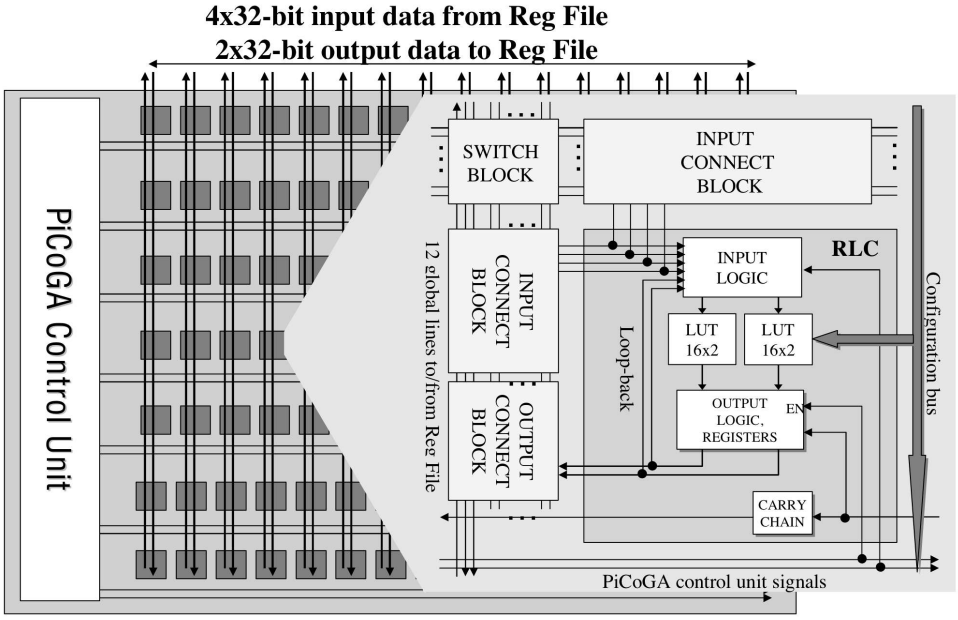


Abbildung 3.8: Logikblock und Architektur des PiCoGA [39]

Das Verbindungsnetz basiert auf einer Island-Style-Architektur. In der Connection-Box wird ein spezielles Decoder-Schema angewendet, um den Konfigurationsaufwand zu verringern.

Dazu werden die Konfigurationsbits für die einzelnen Schalter der Connection-Box in einem One-Hot-Decoder erzeugt. Dabei ist nur jeweils eine Leitung des Interconnects mit einem Ein- bzw. Ausgang des Logikelements verbunden. Die Anzahl der benötigten SRAMs kann durch diese Maßnahme von $O(n)$ auf $O(\log_2 n)$ gesenkt werden. Der Nachteil dieser Methode ist eine Verringerung der Flexibilität des Verbindungsnetzes. Insbesondere können keine Broadcast-Verbindungen von einer Quelle (LE-Ausgang) mit mehreren Senken (Verbindungsleitungen, die zu weiteren Logikelementen führen) realisiert werden.

Die PiCoGA-Architektur ist besonders für die dynamische Rekonfiguration optimiert. Bei Standard-FPGA-Architekturen dauert es üblicherweise einige hundert oder tausend Taktzyklen, die Konfigurationsdaten zu ändern. Um eine deutlich schnellere Rekonfiguration zu ermöglichen, speichert PiCoGA die Konfigurationsbits in Multi-Kontext-SRAMs. Dadurch ist es möglich, innerhalb von einem Taktzyklus der PiCoGA-Pipeline die Konfiguration des Arrays zu ändern. Insgesamt gibt es vier Konfigurations-Kontexte, die dynamisch gewählt werden können. Dabei wird auch partielle Rekonfiguration unterstützt. Das Nachladen eines kompletten Konfigurationszyklus in die SRAMs geschieht über einen dedizierten 192-Bit breiten Bus. Damit ist eine komplette Rekonfiguration einer Reihe in maximal 16 Zyklen möglich.

Die in [39] vorgestellte Gesamtarchitektur mit der Bezeichnung XiRisc (eXtended Instruction Set RISC) umfasst neben dem eFPGA einen VLIW-Prozessorkern. Die Anbindung des eFPGAs an den Prozessorkern erfolgt dabei als RFU (siehe Kapitel 3.2). Ein Testchip mit dem Prozessorkern und einem PiCoGA mit acht Zeilen wurde in einer 180nm-CMOS-Technologie gefertigt. Anhand der angegebenen Flächenwerte berechnet sich die Logikelementdichte zu ungefähr 53 LEs/mm^2 bei Normierung auf LUT4-Logikelemente. Der hohe Flächenbedarf entsteht vermutlich insbesondere durch die Multi-Kontext-Konfigurierbarkeit. Typischerweise wird bei Standard-FPGAs mit einem Konfigurationskontext ungefähr die Hälfte der Gesamtfläche durch die SRAMs belegt. Für exemplarische Anwendungen der digitalen Signalverarbeitung konnte mit XiRisc eine Verringerung des Energieumsatzes um den Faktor fünf bis 17 erreicht werden.

3.4.4 SoC mit Tensilica XTensa und M2000 FlexEOS eFPGA (STM)

Das in [13] von der Firma STMicroelectronics vorgestellte SoC kombiniert einen Tensilica XTensa [65][24] als Prozessorkern mit einem eFPGA von M2000 [40] sowie einigen Peripheriekomponenten. Bei allen verwendeten Komponenten handelt es sich um kommerziell erhältliche Produkte. Das Logikelement des verwendeten M2000 FlexEOS besteht aus einer LUT4 und einem Register. Es entspricht damit exakt dem in Kapitel 2.2.3 beschriebenen Aufbau. Da von M2000 keine weiteren Informationen über die Architektur des

FlexEOS zur Verfügung stehen, werden hier die Angaben aus [13] übernommen. Der Aufbau des eFPGA ist schematisch in Abbildung 3.9 dargestellt.

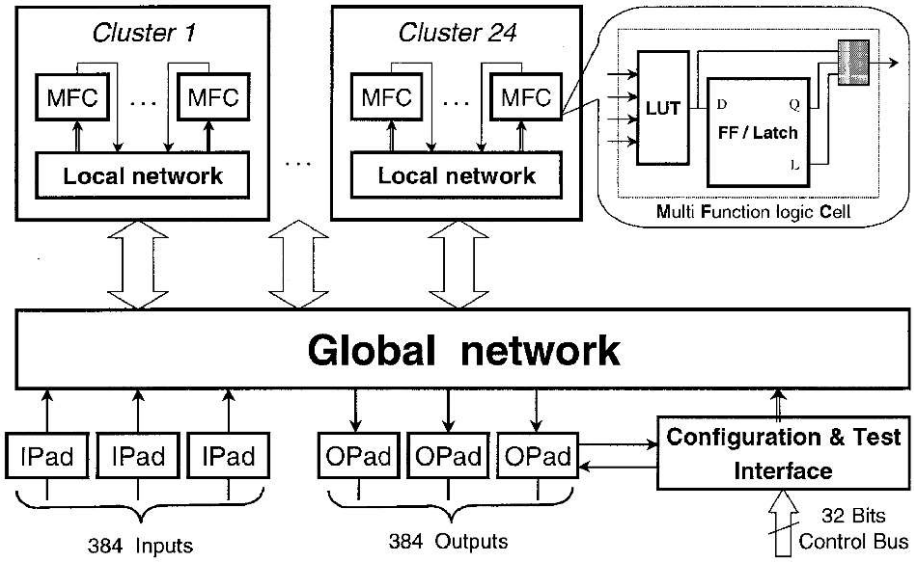


Abbildung 3.9: Logikelement und Architektur des M2000 FlexEOS nach [13]

Bei der in [13] beschriebenen Prozessor-eFPGA-Architektur hat das eFPGA 3000 Logikelemente, die in 24 Clustern organisiert sind. Dies entspricht 125 Logikelementen pro Cluster. Aus den zur Verfügung stehenden Informationen ist kein Rückschluss auf die Verbindungsarchitektur zwischen den Logikelementen möglich. Es kann lediglich festgestellt werden, dass es sich um eine hierarchische Struktur handelt, bei der innerhalb der Cluster ein lokales Verbindungsnetz existiert, während die Cluster über ein globales Netz kommunizieren. Die Ankopplung des eFPGA an den Prozessorkern erfolgt auf mehrere Weisen. Die rekonfigurierbare Einheit kann als RFU (siehe Kapitel 3.2) direkt in die Pipeline des Prozessors eingebunden werden, um dessen Befehlssatz zu erweitern. Weiterhin ist es möglich, das eFPGA als rekonfigurierbaren Coprozessor (RC) über den Prozessorbus einzubinden. Schließlich kann über ein entsprechendes Interface das eFPGA als Schnittstelle zu externen Peripheriegeräten wie Sensoren etc. verwendet werden und die erforderlichen Kommunikationsprotokolle zwischen Prozessor und Peripherie implementieren.

Das SoC wurde als Testchip in einer 180nm-CMOS-Technologie implementiert. Das eFPGA mit 3000 Logikelementen entspricht den Publikationen zufolge für die betrachteten Anwendungen 100.000 ASIC-Gatteräquivalenten. Daraus ergibt sich bezüglich der Siliziumfläche ein Mehraufwand von 42% gegenüber einer physikalisch optimierten Implementierung, die sämtliche abgebildeten Funktionen realisieren kann.

Auf Basis der in [13] betrachteten Anwendungen aus dem Bereich der Gesichts- und Spracherkennung wurde eine Verbesserung der Performance um einen Faktor zwei bis zehn erreicht. Die dynamische Rekonfiguration des eFPGAs dauert 500µs und ist damit gegenüber der gesamten Ausführungszeit der betrachteten Anwendungen vernachlässigbar.

3.4.5 Bewertung

Alle zuvor beschriebenen Ansätze haben die Optimierung von FPGA-Architekturen zur Verbesserung der Flächen- und Verlustleistungseffizienz zum Ziel. Dabei werden alle wesentlichen Architekturmerkmale von FPGAs betrachtet. Die „Template-based embedded reconfigurable unit“ benutzt beispielsweise ein arithmetikorientiertes Logikelement. Im Gegensatz zu den LEs kommerzieller diskreter FPGAs mit größeren LUTs (vier bis sechs Eingänge) werden hier kleinere LUTs und zusätzliche dedizierte Logik verwendet, um das Logikelement möglichst klein zu halten. Dadurch ist auf Bitebene eine effiziente Abbildung von Arithmetikanwendungen möglich. Durch die Verwendung eines herkömmlichen Island-Style-Interconnects ist jedoch kein effizientes Routing der mehrheitlich lokalen Signale solcher Anwendungen möglich.

Bei „Chimaera“ wurde das Verbindungsnetz für die zeilenbasierte Abbildung von Operationen optimiert. Dadurch ist es möglich, Custom-Instructions, die mit einer LE-Zeile abgebildet werden können, effizient zu implementieren. Die lokalen Signale eines Arithmetik-datenpfades laufen jedoch nicht nur in horizontaler, sondern auch in vertikaler Richtung. Zudem ist das LUT-basierte Logikelement nur bedingt für die Abbildung von Arithmetik-anwendungen geeignet. Weiterhin erfordert die Möglichkeit zur partiellen Rekonfiguration einen sehr hohen Flächenmehraufwand.

„PiCoGa“ bietet ebenfalls die Möglichkeit der dynamischen Rekonfiguration. Sowohl die Logikelemente als auch das Verbindungsnetz sind jedoch Standard-FPGA-Architekturen entlehnt. Lediglich die reduzierte Flexibilität des verwendeten Island-Style-Interconnect lassen für entsprechende Anwendungen einen Effizienzvorteil erwarten.

Bei dem „M2000 FlexEOS“ wird ein herkömmliches LUT4-basiertes Logikelement mit einem nicht weiter erläuterten hierarchischen Interconnect kombiniert. Die Beschreibung legt jedoch nah, dass auch hier keine nennenswerte Optimierung der Architektur auf den typischen Signalfluss in Arithmetikanwendungen durchgeführt wurde.

Alle gezeigten Beispiele weisen somit Optimierungen gegenüber Standard-FPGAs auf, sind jedoch stark an diese angelehnt. Bei keiner der Architekturen wird hingegen das a priori vorhandene Wissen über den typischen Signalfluss in Arithmetikanwendungen benutzt, um den Interconnect als bezüglich der Effizienz kritischstes Element im Zusammenspiel mit entsprechenden Logikelementen wesentlich effizienter zu gestalten (vgl. 3.3.3). Stattdessen wird vermutlich mit Hinblick auf die Verfügbarkeit von FPGA-Place&Route-Tools entweder ein herkömmlicher Island-Style-Interconnect oder ein bezüglich des Mappings einfaches Zeilenschema verwendet. Letztendlich wurde bei den dargestellten Beispielen stets eine FPGA-Architektur relativ losgelöst von den später darauf abzubildenden Anwendungen anhand einzelner Architekturmerkmale optimiert. Bei keinem der Ansätze wurde konsequent der umgekehrte Weg von der Arithmetikanwendung zur Architektur gegangen, um insbesondere die Implementierungskosten des Verbindungsnetzes zu verringern.

3.5 Entwurfswerkzeuge

Im Folgenden werden verschiedene Entwurfswerkzeuge aus dem akademischen Bereich aufgeführt, die für Entwurf, Optimierung und Implementierung von generischen FPGA-Architekturen verwendet werden können. Als häufig in der Forschung verwendete Referenzarchitektur wird die sogenannte VPR-konforme Architektur mit den entsprechenden Werkzeugen für Technology-Mapping sowie Place&Route vorgestellt. Weiterhin werden verschiedene Werkzeuge für den VLSI-Entwurf betrachtet.

3.5.1 VPR-konforme Architektur

Die bekannteste frei verfügbare Entwurfsumgebung für parametrisierbare FPGAs ist der VPR-Designflow [9], der an der Universität Toronto entwickelt wurde. VPR (Versatile Place&Route) ist ein Programm zum Platzieren und Verdrahten für generische Island-Style-FPGAs. Abbildung 3.10 zeigt den allgemeinen Aufbau der VPR-konformen Architektur. Es handelt sich dabei um einen Island-Style mit Clustern aus LUT-basierten Logikelementen. Die Anzahl der Verbindungsleitungen in den Routing-Kanälen kann als Parameter frei gewählt werden. Das Verbindungsnetz ist segmentiert, wobei die Anzahl und Länge der jeweiligen Segmente anteilmäßig in Bezug auf die Gesamtzahl der Verbindungsleitungen festgelegt werden kann.

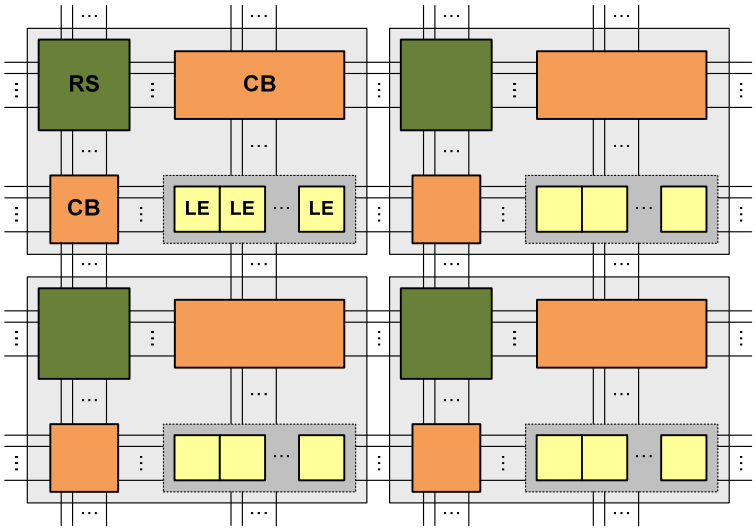


Abbildung 3.10: VPR-konforme Architektur (Island-Style mit Clustern)

Bei den Routing-Switches stehen drei verschiedene Topologien (Disjoint, Wilton, Universal) zur Auswahl. In Abbildung 3.11 sind die drei Topologien mit den jeweils schaltbaren Verbindungswegen dargestellt. Eine beliebige Definition der Routing-Switches z.B. durch Angabe der Konnektivität zwischen allen sich kreuzenden Leitungen ist nicht möglich.

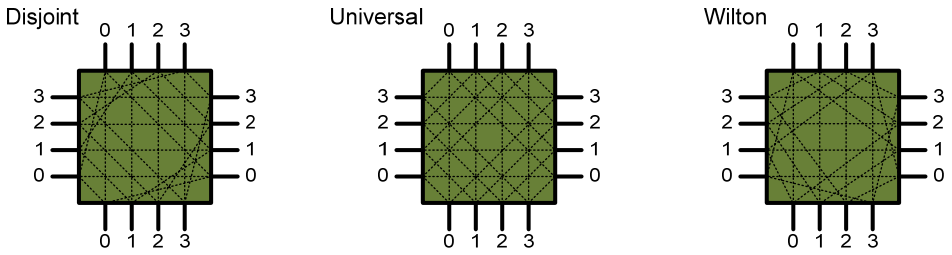


Abbildung 3.11: Routing-Switch-Topologien in VPR

Die Konnektivität der Connection-Boxes kann durch die Angabe des Anteils der angeschlossenen Verbindungsleitungen im Verhältnis zur insgesamt verfügbaren Anzahl an Leitungen variiert werden. Eine genaue Festlegung der Schalterpositionen ist nicht möglich, so dass keine explizite Zuordnung von bestimmten Logikelementen eines Clusters zu einer bestimmten Verbindungsleitung im Routing-Kanal möglich ist. Die Cluster bestehen aus einer durch einen entsprechenden Parameter einstellbaren Anzahl von Logikelemente. Jedes Logikelement ist aus einer LUT mit ebenfalls variabler Anzahl von Eingängen (zwischen zwei und sieben) und einem nachfolgenden Register aufgebaut. Abbildung 3.12 zeigt den Aufbau des Logikelements und eines Clusters. Die Definition dedizierter Verbindungen oder andersartiger Logikelemente ist nicht vorgesehen, so dass sich z.B. keine Carry-Chains realisieren lassen. Neben den zuvor beschriebenen Architekturparametern können noch detaillierte Angaben zu der Position der Ein- und Ausgänge der Cluster getroffen werden.

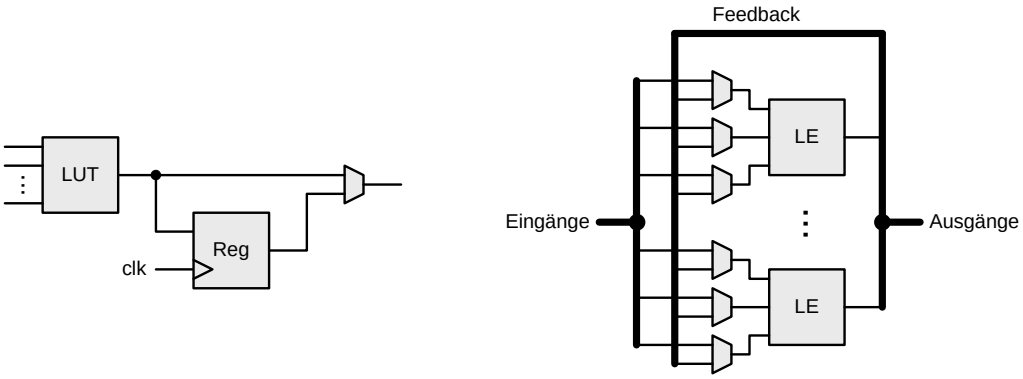


Abbildung 3.12: Logikelement und Cluster der VPR-konformen Architektur

Das Programm VPR ist im Wesentlichen ein Place&Route-Programm auf Basis eines Thermal-Annealing-Algorithmus für die Platzierung und eines modifizierten PathFinder-Algorithmus [41] für das Routing. Das Technology-Mapping für VPR-konforme FPGA-Architekturen erfolgt mit T-VPack [9]. T-VPack bildet Netzlisten, die im so genannten BLIF-Format (Berkeley Logic Interchange Format) beschrieben sind, auf LUT-basierte Logikelemente ab. Dabei werden die Größe der Cluster und die Anzahl der LUT-Eingänge entsprechend berücksichtigt.

VPR ermittelt anhand der Platzierung und dem Routing modellbasiert die Fläche und Länge des zeitkritischen Signalpfades für die abgebildete Schaltung. VPR und T-VPack sind als

Quellcode frei verfügbar. In der Forschung werden daher häufig modifizierte Versionen von VPR eingesetzt. Zwischen 2000 und 2009 war die Entwicklung von VPR eingestellt. Seit 2009 ist eine neue Version verfügbar, in welcher das Architekturmodell um heterogene Blöcke (z.B. zur Verwendung dedizierter Multiplizierer oder Speicher) und unidirektionale Verbindungsleitungen erweitert wurde. An der University of British Columbia wurde zudem für VPR die Erweiterung ACE (Activity-Estimation) entwickelt, mit der zusätzlich zur Flächen- und Laufzeitbestimmung auch eine probabilistische Abschätzung der Verlustleistungsaufnahme der von VPR unterstützten FPGA-Architektur möglich ist [59]. ACE ist ebenfalls frei als Quellcode verfügbar. Neben diesen Erweiterungen existiert eine Vielzahl weiterer modifizierter Varianten von VPR und T-VPack, die im akademischen Umfeld zur Untersuchung von FPGA-Architekturen verwendet werden.

3.5.2 Weitere Programme für das Mapping

Einen ähnlichen Ansatz wie VPR verfolgt das im Philips Research Lab Eindhoven entwickelte Place&Route-Programm PYTHAGOR mit dem dazugehörigen grafischen Architektureditor ARCHIMED (Architecture Micro Editor) [19]. Wie bei VPR/T-VPack werden nur Island-Style-Architekturen mit LUT-basierten Logikelementen unterstützt. Allerdings besteht bei ARCHIMED/PYTHAGOR größere Flexibilität bei der Definition der Routing-Architektur. So ist es z.B. möglich, direkte LUT-LUT-Verbindungen zwischen Logikelementen unter Umgehung des globalen Verbindungsnetzes zu verwenden. ARCHIMED/PYTHAGOR ist derzeit weder frei noch kommerziell verfügbar.

Von der Firma Altera existiert mit QUIP (Quartus University Interface Program) [5] ein frei verfügbares Tool, das neben definierbaren Architekturbeschreibungen von FPGAs auch die Anbindung an die kommerziellen Design-Tools des Herstellers erlaubt, so dass ebenfalls auf Mapper, Place&Route-Tools etc. zugegriffen werden kann. Die Beschreibung der FPGA-Architektur ist extrem komplex und unterliegt ähnlich wie bei VPR einer Vielzahl von Einschränkungen.

In [37] wird der Technology-Mapper SATMAP beschrieben, der für beliebige Logikelemente funktioniert. Die Funktionalität der konfigurierbaren Logikelemente wird durch eine entsprechende logische Gleichung definiert. Damit ist es möglich, auch Logikelemente, die nicht LUT-basiert sind, zu verwenden. SATMAP ist eine gemeinsame Entwicklung von Altera und der Universität Toronto. Das Programm ist derzeit weder frei noch kommerziell verfügbar.

Ein häufig verwendetes Programm zur Synthese und Optimierung von Schaltungen vor dem eigentlichen Technology-Mapping ist das an der Universität Berkeley entwickelte SIS (Sequential Interactive Synthesis) [61]. SIS fasst zahlreiche Algorithmen zur Minimierung und Optimierung logischer Gleichungen in einer einheitlichen Umgebung zusammen.

Eine detaillierte Übersichtsgrafik verschiedener Entwurfswerkzeuge aus dem akademischen Bereich sowie deren Zuordnung zu den verschiedenen Phasen des FPGA-Entwurfsablaufs ist in Anhang B dargestellt.

3.5.3 VLSI-Entwurf

Neben Werkzeugen zum Mapping von Schaltungen auf generische FPGA-Architekturen, um deren Effizienz zu untersuchen, spielt der eigentliche VLSI-Entwurf solcher FPGAs ebenfalls eine wichtige Rolle. Entsprechende Methoden wurden in verschiedenen Forschungsprojekten untersucht. In [29] werden eFPGA-Architekturen beschrieben, die durch Standardzell-Synthese als Bestandteil eines SoC implementiert werden. Der Schwerpunkt liegt dabei auf der Implementierung eine Vielzahl kleiner eFPGA-Makros auf dem Chip mit jeweils angepassten Architekturparametern. Die Architektur entspricht im Wesentlichen einem klassischen Island-Style mit LUT3-basierten Logikelementen. Die gegenüber gängigen Standard-FPGAs geringere Größe der LUT wird durch das Flächenverhältnis von Logik-elementen zu konfigurierbaren Verbindungsressourcen begründet. Den Untersuchungen in [29] zufolge ist bei synthetisierten eFPGAs der Flächenmehraufwand durch die Verwendung von Standardzellen für die Logikelemente im Verhältnis größer als für die Verbindungsressourcen. Es werden zwei Architekturvarianten beschrieben, von denen eine speziell für die beschriebenen Rahmenbedingungen (kleine Makros, generische Architektur-beschreibung) angepasst ist. Insbesondere wird bei dieser Variante kein homogener Aufbau des Makros gewählt, sondern das Verbindungsnetz abhängig von der Position im Makro unterschiedlich flexibel gewählt. Als Entwurfsumgebung wurde VPR verwendet. Anhand von Synthese-Ergebnissen wurde ermittelt, dass der Flächenbedarf der implementierten eFPGA-Makros im Vergleich zu physikalisch orientierten Layouts um den Faktor 6,4 größer ist. Damit eignen sich die Methodik und Architektur nur für kleine eFPGA-Makros geringer Komplexität, bei denen der Mehraufwand im Vergleich zu den restlichen Komponenten des SoC vernachlässigbar ist.

In [71] wird eine datenpfadorientierte eFPGA-Architektur vorgestellt, die durch Standardzell-Synthese implementiert wird. Durch Verwendung von shared-SRAMs (siehe Kapitel 4.2.6) für Logikelemente und Verbindungsnetz wird die Flächeneffizienz der Architektur für Arithmetikdatenpfade erheblich gesteigert. Zusätzlich verfügt das eFGPA über eine einstellbare Anzahl von dedizierten Multiplizierern. Das Verbindungsnetz hat eine gegenüber Standard-FPGA-Architekturen deutlich verringerte Flexibilität. Insbesondere haben die konfigurierbaren Verbindungen vorgegebene Signalrichtungen, so dass keine beliebigen Verbindungswege geschaltet werden können. Die Flexibilität der Gesamtarchitektur ist damit stark eingeschränkt und bedeutet eine erhebliche Einschränkung der Anwendungsklasse. Durch systematische Analyse der Architekturparameter anhand von Benchmark-Schaltungen

wird eine Flächeneffizienz erreicht, die den Angaben zufolge Standard-FPGAs entspricht, welche als handoptimierte Layouts implementiert werden.

In [32][56] wird das Entwurfswerkzeug GILES (Good Instant Layout of Erasable Semiconductors) vorgestellt, das auf Basis handoptimierter Layouts der wesentlichen Basiselemente eines FPGAs automatisch Layouts generiert. Die Architekturbeschreibung des FPGAs ist dabei VPR-konform. Dadurch ist es möglich, in VPR eine modellbasierte Optimierung der Architekturparameter durchzuführen und die spezifizierten FPGAs automatisiert als VLSI-Layouts erzeugen zu lassen. GILES umfasst dazu geeignete Platzierungs- und Routing-Algorithmen. Zur Generierung VPR-konformer FPGAs sind ungefähr 15 handoptimierte Basiszell-Layouts erforderlich. Dabei handelt es sich um Zellen sehr geringer Komplexität wie z.B. Buffer, Multiplexer, LUTs und SRAM-Zellen. Die Granularität beim automatischen Platzieren des FPGA-Layouts ist somit sehr fein. Sie reicht von Invertern mit zwei Transistoren bis zu Multiplexern mit 20 Eingängen und 38 Transistoren. Außerdem wird ein SRAM-Feld mit 4×4 Zellen und 80 Transistoren als größtes Basiselement verwendet. Anhand von Beispielimplementierungen wird ein Flächenmehraufwand zwischen 36% gegenüber einem Xilinx VirtexE [32] und 97% gegenüber einem Altera Apex 20K400E [56] angegeben. Ein entsprechender Testchip wurde in einer 180nm-Technologie gefertigt.

3.6 Optimierungspotenzial

In verschiedenen Arbeiten wurde anhand des in Kapitel 3.5.1 beschriebenen Entwurfsablaufs mit VPR/T-VPack der Einfluss verschiedener Architekturparameter wie LUT-Größe, Cluster-Größe und Flexibilität des Verbindungsnetzes auf Standard-FPGA-Architekturen untersucht. So wurde z.B. in [1] festgestellt, dass die optimale LUT-Größe zwischen vier und sechs Eingängen beträgt, während die Cluster-Größe im Bereich drei bis zehn liegen sollte. Beide Ergebnisse wurden im Hinblick auf ein optimales AT-Produkt bestimmt.

In [73] wird eine umfassende Studie zur Verwendung von Bus-basierten Verbindungskanälen in FPGAs beschrieben. Dabei werden mehrere Switch-Points eines Routing-Switch mit einem gemeinsamen Satz von Konfigurationsbits konfiguriert, um den Flächenmehraufwand für die SRAMs zu reduzieren (shared-SRAMs). Da bei datenflussorientierten Anwendungen häufig ganze Signalworte statt einzelner Bits innerhalb des FPGAs verbunden werden müssen, lässt sich für derartige Anwendungen ein Flächenvorteil erwarten. Für die Untersuchungen wurde eine modifizierte Version des VPR-Designflows verwendet. Es wurde gezeigt, dass die Gesamtfläche eines FPGAs durch Verwendung Bus-basierter Verbindungskanäle um bis zu 10% gesenkt werden kann. Weitere ausführliche Betrachtungen zur Optimierung der Flexibilität von Routing-Switch (F_S) und Connection-Box (F_C) mit VPR sind z.B. [10] zu entnehmen.

In [35] wurde eine detaillierte Untersuchung von Routing-Switches durchgeführt. Neben Architekturaspekten wurden dabei auch schaltungstechnische Implementierungsdetails der Switch-Points betrachtet. Insbesondere wird dort ein so genannter Sparse-Crossbar, also ein Crossbar mit reduzierter Population, vorgestellt. Zusammen mit LUT6-basierten Logikelementen konnte so eine Verbesserung der AT-Effizienz um 22% demonstriert werden. Weiterhin wurde untersucht, wie durch die Verwendung von Pass-Transistoren statt programmierbaren Treibern die AT-Effizienz verbessert werden kann. Den Ergebnissen zufolge kann die AT-Effizienz um bis zu 14% verbessert werden, wenn statt eines vollständig treiberbasierten Verbindungsnetzes (fully-buffered), wie es in modernen FPGAs üblich ist, teilweise Pass-Transistoren verwendet werden.

Im Rahmen der vorliegenden Arbeit wurden erste Abschätzungen zum Optimierungspotenzial von FPGA-Architekturen mit dem in Kapitel 3.5.1 beschriebenen Entwurfsablauf basierend auf VPR, T-VPack und ACE durchgeführt. Da die Untersuchung sich in erster Linie auf die Optimierung der Logikelemente konzentrierte, wurden als veränderliche Architekturparameter die LUT-Größe, die Cluster-Größe und die Anzahl der Eingänge pro Cluster variiert. Die Anzahl der vorhandenen Leitungen im Verbindungsnetz wurde stets so gewählt, dass sie 20% über der mindestens benötigten lag („Low-Stress-Routing“). Mit Hilfe eines automatisierten Entwurfsablaufs wurden systematisch sinnvolle Parameterkombinationen gewählt, eine entsprechende Architekturbeschreibung für VPR generiert und alle Schritte von der Synthese bis zum Place&Route mit VPR durchgeführt. Die ATE-Kosten wurden anhand der Angaben von VPR (kritischer Signalpfad, Fläche des Verbindungsnetzes), ACE (Energieumsatz) und einer eigenen Modellfunktion für den Flächenbedarf der Logikelemente bzw. Cluster ermittelt. Als exemplarische Anwendungen wurden verschiedene Varianten eines ACS-Elements (Add-Compare-Select-Element) implementiert. ACS-Elemente sind der Grundbaustein einer ACSU (Add-Compare-Select-Unit) eines Viterbi-Decoders. In der ACSU wird auf Basis eines Trellis-Diagramms der Pfad mit der geringsten Fehlerwahrscheinlichkeit gesucht. Dazu werden die Fehlerwahrscheinlichkeiten aus der vorherigen Pfadmetrik und der aktuellen Zweigmetrik ermittelt. Hierzu werden mehrere ACS-Elemente entsprechend dem Aufbau des Trellis-Diagramms kaskadiert. Abbildung 3.13 zeigt ein einzelnes ACS-Element. Es wird durch die folgende Gleichung beschrieben:

$$\Gamma_t = \min\left\{\Gamma_{i,t-1} + \lambda_i\right\} \tag{3.1}$$

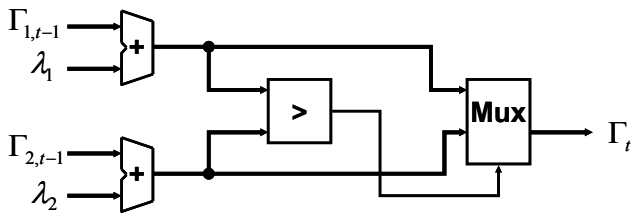


Abbildung 3.13: Add-Compare-Select-Element

Dabei ist $\Gamma_{i,t-1}$ die vorherige Pfadmetrik und λ_i die aktuelle Zweigmetrik. Das ACS-Element addiert die jeweiligen Metriken zweier Zweige (Add), vergleicht sie (Compare) und wählt je nach gewählter Darstellungsform die kleinere bzw. größere der Summen aus (Select). Insgesamt wurden sechs verschiedene Implementierungsformen eines ACS-Elements für 8 Bit breite Eingangsworte untersucht. Dabei wurden sowohl bitserielle als auch parallele Implementierungen sowie unterschiedliche Radices (der Radix bestimmt die Anzahl der Zweige/Pfade, die pro ACS-Element verglichen werden können) und Zahlendarstellungen (Carry-Ripple, Carry-Save) untersucht. Bei den Varianten mit Radix-4 wurden zudem zwei Varianten des Komparators betrachtet (baumförmig mit zwei Radix-2-Stufen und einstufig). Abbildung 3.14 zeigt exemplarisch die ATE-Effizienz (Kehrwert des Produkts aus Fläche, Laufzeit und Energie pro Sample) zweier Implementierungsalternativen in Abhängigkeit von der LUT-Größe und der Cluster-Größe. Größere Werte entsprechen dabei effizienteren Implementierungen.

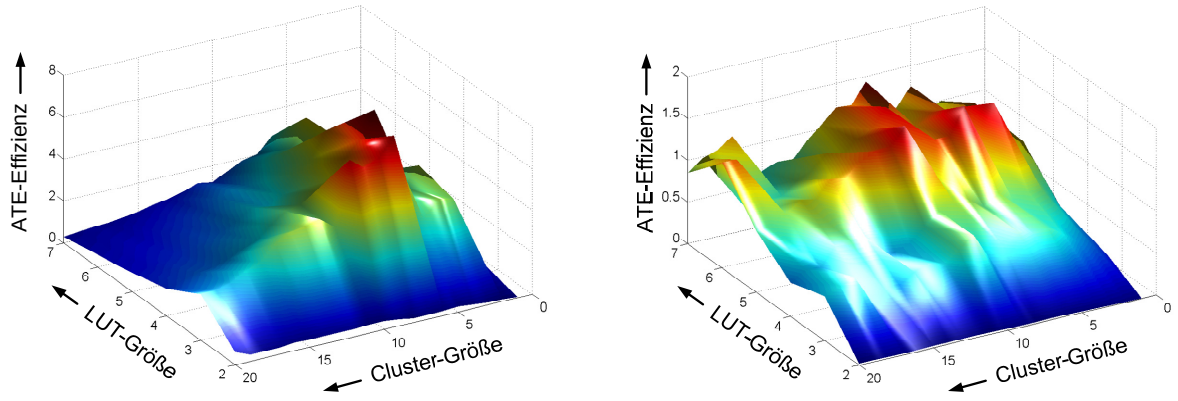


Abbildung 3.14: ATE-Effizienz eines bitseriellen Radix-2-ACS-Elements (links) und eines parallelen Radix-4-ACS-Elements mit sechs Komparatoren (rechts)

Die Diagramme zeigen, dass die Wahl der Architekturparameter einen erheblichen Einfluss auf die ATE-Effizienz der implementierten Schaltungen hat. Abhängig von der Art der Anwendung können zudem die Profile der ATE-Effizienz sehr unterschiedlich aussehen. Die Architekturparameter müssen daher so gewählt werden, dass sie einen möglichst günstigen Kompromiss zwischen den verschiedenen Anforderungen der einzelnen Anwendungen der gemeinsamen Anwendungsklasse darstellen. Zusätzlich zur Optimierung der FPGA-Architektur ist es entscheidend, eine architekturgerechte Implementierungsform der jeweiligen Anwendung zu finden. In Abbildung 3.15 ist zur Illustration dieser Problematik der Entwurfsraum für die sechs betrachteten ACS-Varianten bezüglich des Flächenbedarfs und der Symbolperiode dargestellt. Punkte gleicher Form bzw. Farbe entsprechen identischen Implementierungsformen. Die Streuung der Punkte innerhalb des AT-Raums entsteht durch die Variation der drei Architekturparameter des Logikelements. Die beiden Geraden entsprechen konstanten AT-Produkten. Das Diagramm verdeutlicht die große Dynamik des Entwurfsraums einerseits durch die Implementierungsalternativen und andererseits durch die

Anpassung der FPGA-Architektur. Insgesamt variiert das AT-Produkt wie im Diagramm dargestellt ungefähr um bis zu Faktor 50. Die parallelen ACS-Elemente erreichen dabei einen hohen Durchsatz (entsprechend einer kleinen Symbolperiode) bei hohem Flächenbedarf, während die seriellen Varianten einen geringeren Durchsatz bei gleichzeitig geringerem Flächenbedarf erzielen. Neben den mittels des VPR-Designflows ermittelten AT-Produkten verschiedener ACS-Elemente ist in Abbildung 3.15 zusätzlich eine parallele Radix-2-Implementierung auf einem kommerziellen FPGA (Altera APEX 20KE) dargestellt. Erwartungsgemäß liegt der Punkt für die Implementierung auf dem kommerziellen FPGA in der Mitte der Punktmenge, die mit den VPR-basierten Ergebnissen erzeugt wurde. Durch geeignete Wahl der Parameter können erheblich effizientere Varianten gefunden werden. Insgesamt ergibt sich so eine Pareto-Front [57], welche die effizientesten Implementierungen umfasst. Dabei sind sowohl die verschiedenen Implementierungsformen als auch die Architekturvariationen maßgebend.

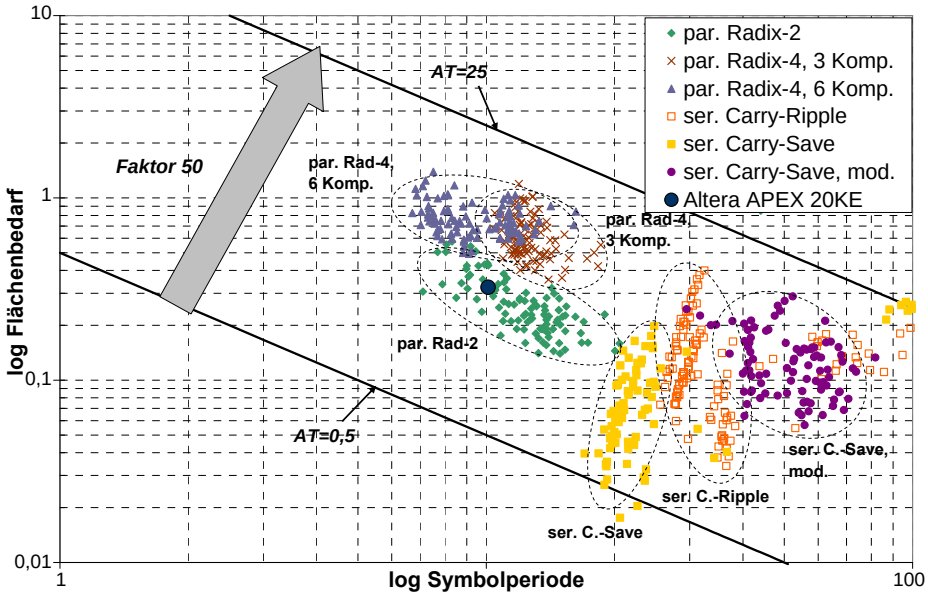


Abbildung 3.15: Entwurfsraum verschiedener ACS-Implementierungen

Bei der Wahl der geeigneten Implementierungsalternative (seriell, parallel, etc.) müssen daher zunächst die Anforderungen der jeweiligen Anwendung z.B. bezüglich der Symbolperiode definiert werden. Je nach gewählter Implementierungsalternative ergeben sich dann unterschiedliche Architekturparameter, um eine optimale Effizienz zu erhalten. Das Diagramm zeigt auch, dass der Einfluss von Parametervariationen bei verschiedenen Implementierungsalternativen sehr unterschiedlich ist. So ist z.B. bei den parallelen ACS-Elementen für jede Implementierungsform ein relativ gleichmäßiger Abtausch zwischen Symbolperiode und Flächenbedarf möglich. Im Diagramm zeigt sich dies durch eine Verteilung der Punkte in einem schmalen Band ähnlicher AT-Produkte. Bei den bitseriellen Implementierungen ist das Band der AT-Produkte breiter. Das bedeutet, dass Architektur-variationen bei ähnlicher Symbolperiode zu deutlichen Unterschieden im Flächenbedarf

führen können. Ursache dafür ist der unregelmäßigere Aufbau und die kleine Anzahl an benötigten Logikelementen. Änderungen der LUT- und Cluster-Größe lassen sich dadurch weniger gut als bei den parallelen ACS-Elementen mit vielen Logikelementen kompensieren.

Die Untersuchungen mit dem VPR-basierten Entwurfsablauf haben gezeigt, dass ein erhebliches Optimierungspotenzial bei FPGA-Architekturen besteht. Durch gezielte Anpassung der Architekturparameter einer Standard-FPGA-Architektur konnte die Effizienz bezüglich Laufzeit, Energieumsatz und Fläche erheblich verbessert werden. Eine weitere Verbesserung lässt sich durch die Anpassung der gesamten Architektur an eine arithmetik-orientierte Anwendungsklasse erwarten. Eine ausführlichere Diskussion der durchgeführten Arbeiten wird in [47] präsentiert. Weitere Informationen zu dem verwendeten Designflow und den verschiedenen ACS-Implementierungen können [46] entnommen werden.

3.7 Zusammenfassung

Die in Kapitel 3.4 vorgestellten Beispielarchitekturen zeigen, dass es durch die Verwendung anwendungsklassenspezifisch optimierter FPGAs als Accelerator möglich ist, effizientere Architekturen für Anwendungen der digitalen Signalverarbeitung zu konzipieren. Die verwendeten eFPGAs sind dabei jedoch in der Regel stark an herkömmliche FPGAs angelehnt und nicht für die Abbildung datenflussorientierter Algorithmen optimiert. Die Verwendung von Logikelementen mit großen Lookup-Tables (bis zu sechs Eingänge) und komplexen Verbindungsarchitekturen mit entsprechend großer Transistorzahl führt dazu, dass nur moderate Effizienzsteigerungen gegenüber Standard-FPGAs erreicht werden. Zudem erfordert eine systematische Optimierung des eFPGAs für eine Anwendungsklasse eine flexible Beschreibungsform der Architektur, so dass der Einfluss verschiedener Architektur-parameter (z.B. Flexibilität des Verbindungsnetzes) auf die Effizienz untersucht werden kann. Dabei müssen die Anforderungen von Arithmetikdatenpfaden wie die hohe Lokalität der Signale explizit berücksichtigt werden. Demgegenüber verwenden z.B. die in Kapitel 3.4.1 und Kapitel 3.4.3 beschriebenen Architekturen einen reinen Island-Style-Interconnect, der für die Abbildung von Datenpfaden mit hoher Lokalität nur bedingt geeignet ist. Vergleichbare Einschränkungen gelten auch für VPR-konforme FPGAs, die bei Implementierung mit GILES nur die Effizienz eines Standard-FPGAs erreichen.

Die Verwendung Standardzell-basierter Entwurfsverfahren führt zudem dazu, dass die auf Architekturebene erreichten Effizienzverbesserungen bei der Implementierung wieder verloren gehen. Insgesamt bleibt so das Optimierungspotenzial bestehender eFPGAs deutlich hinter der theoretisch möglichen Steigerung zurück, die bei einer konsequenten Optimierung der Architekturen auf die Arithmetikanwendungen zu erwarten ist.

Die im Rahmen dieser Arbeit konzipierte eFPGA-Architektur, die im Folgenden beschrieben wird, ist daher sowohl bezüglich der Logikelemente als auch des Verbindungsnetzes für die Anforderungen von Arithmetikdatenpfaden optimiert worden. Ähnlich wie bei dem in

Kapitel 3.4.1 beschriebenen eFPGA ist zudem das hier vorgestellte Architekturmodell parametrisierbar, so dass eine systematische Optimierung der Architekturparameter für die entsprechende Anwendungsklasse möglich ist. Ein besonders effizientes Implementierungsverfahren führt zudem zu hoher Transistordichte und somit geringen Implementierungskosten.

4 Parametrisierbare arithmetikorientierte eFPGA-Architektur

4.1 Übersicht

Um die Nachteile existierender eFPGA-Architekturen zu überwinden, wurde in dieser Arbeit eine Architektur konzipiert, die stark an den Architekturen von Arithmetikdatenpfaden orientiert ist. Ein wesentliches Ziel dabei war es, bei Verwendung für Arithmetikanwendungen eine besonders hohe Effizienz zu erreichen, ohne die allgemeine Verwendbarkeit z.B. für Kontrollstrukturen grundsätzlich einzuschränken. Statt einer Optimierung für eine sehr breite Vielfalt von Anwendungen, wie sie bei Standard-FPGAs und den im vorhergehenden Kapitel beschriebenen daraus abgeleiteten Architekturen betrieben wird, wurde hier deutlich stärker auf die Anforderungen von Arithmetikdatenpfaden eingegangen. Dies führt dazu, dass beim Mapping kontrollflussdominierter Datenpfade die Effizienz deutlich hinter der eines Standard-FPGAs zurückbleiben kann. Im Gegenzug wird jedoch für Arithmetikdatenpfade eine besonders hohe Effizienz erreicht wird.

Anders als bei den in Kapitel 3.4 beschriebenen Beispielen anwendungsklassenspezifischer FPGAs wurde in dieser Arbeit die gesamte Architektur des eFPGAs von den Logikelementen bis zum Verbindungsnetz auf Basis der Anforderungen typischer Arithmetikdatenpfade konzipiert. Anstatt also eine Optimierung von Parametern auf Basis existierender FPGA-Architekturen (z.B. LUT-Größe und Konnektivität des Island-Style-Interconnects) durchzuführen, wurden die wesentlichen Architekturelemente von Anfang an so gestaltet, dass sie sich besonders gut für die Abbildung elementarer Arithmetikoperationen und hochgradig lokaler Kommunikation eignen.

Ein wesentlicher Nachteil herkömmlicher FPGAs ist die bereits festgelegte Architektur und VLSI-Implementierung. Die systematische Optimierung einer eFPGA-Architektur für eine Anwendungsklasse muss jedoch unter der Berücksichtigung einer Vielzahl von Architekturparametern erfolgen. Kommerzielle FPGA-Architekturen und die dazu vorhandenen Entwurfswerkzeuge sind für derartige Untersuchungen ungeeignet. Die Entwurfswerkzeuge unterstützen nur die tatsächlich verfügbaren FPGA-Bausteine, so dass vom Anwender kein Einfluss auf die FPGA-Architektur besteht. Es ist daher erforderlich, entweder auf Tools aus dem akademischen Bereich zurückzugreifen, die für den Zweck der Architekturoptimierung entwickelt wurden, oder die konzipierte FPGA-Architektur mit ihren Parametern auf geeignete Weise zu modellieren. Zur effizienten VLSI-Implementierung einer solchen flexiblen Architektur ist zudem ein automatisierter Entwurf auf Basis eines parametrisierbaren Templates unabdingbar.

Obwohl die Optimierung von FPGA-Architekturen Gegenstand zahlreicher Forschungsprojekte ist, gibt es nur wenige frei verfügbare Werkzeuge, die eine systematische Untersuchung des Einflusses von Architekturparametern auf die Effizienz einer FPGA-Architektur sowie eine entsprechende VLSI-Implementierung ermöglichen. Ein erhebliches

Problem stellt insbesondere die Komplexität des Mappings von Anwendungen auf eine generische Architektur dar. Der in Kapitel 2.3 beschriebene Ablauf zur Implementierung von Anwendungen auf einem FPGA muss dazu so angepasst werden, dass statt einer definierten Architektur mit bekannten Eigenschaften auch generische, parametrisierbare Architekturen unterstützt werden. Dies führt jedoch zu einem drastischen Anstieg der Entwurfskomplexität eines geeigneten Technology-Mappers. Entsprechend sind Technology-Mapper für generische FPGAs typischerweise lediglich in der Lage, LUT-basierte Logikelemente mit verschiedenen LUT-Größen bzw. Cluster-Größen zu unterstützen. Für das Mapping der in dieser Arbeit betrachteten Arithmetikdatenpfade wird daher kein vollautomatisches Werkzeug benutzt, sondern der in Kapitel 4.4 beschriebene manuelle Ansatz.

Eine Schlüsselrolle des hier vorgestellten Architektur-Templates für Arithmetikanwendungen spielt die effiziente Kommunikationsstruktur, welche stark an den Anforderungen regulärer Arithmetikdatenpfade orientiert ist. Diese steht in starker Wechselwirkung zum verwendeten Logikelement, da nur bei effizientem Mapping der Rechenelemente eines Datenpfades auf die Logikelemente eine optimale Ausnutzung des Verbindungsnetzes mit möglichst geringem Mehraufwand für die Kaskadierung mehrerer Logikelemente möglich ist.

4.2 Arithmetikorientiertes Architektur-Template

4.2.1 Übersicht

Ein Kernbestandteil der vorliegenden Arbeit ist die Konzeption eines flexiblen, parametrisierbaren Architektur-Templates für arithmetikorientierte eFPGAs [51]. Es handelt sich dabei um eine formalisierte Beschreibung aller wesentlichen Architektureigenschaften und zugehörigen Parameter eines eFPGAs. Das Template dient dabei sowohl als Hardwarebeschreibung bei der automatisierten VLSI-Implementierung von eFPGA-Makros als auch der Modellierung der physikalischen Implementierungskosten von Datenpfaden auf dem eFPGA. Da es sich bei dem Template nicht um eine völlig generische Beschreibung z.B. auf Gatterebene handelt, ist die Komplexität der Architekturbeschreibung moderat, so dass eine Modellierung der physikalischen Implementierungskosten eines eFPGA auf Basis des Templates durch mathematische Gleichungen möglich ist.

Abbildung 4.1 zeigt eine Übersicht des Architektur-Templates mit den wesentlichen Architekturelementen. Alle Elemente und die zugehörigen Parameter werden in den folgenden Kapiteln detailliert erläutert. Eine detaillierte Darstellung sowie eine Übersichtstabelle aller Parameter ist Anhang C zu entnehmen. Die Gesamtarchitektur basiert auf zweidimensionalen Clustern von Logikelementen mit dedizierten Routing-Blöcken und einer zentralen Registerstufe. Die Cluster werden über speziell zugeschnittene Connection-Boxes und Routing-Switches verbunden. Durch spezielle Broadcast-Leitungen können Operanden zeilen- und spaltenweise an die Logikelemente geschaltet werden.

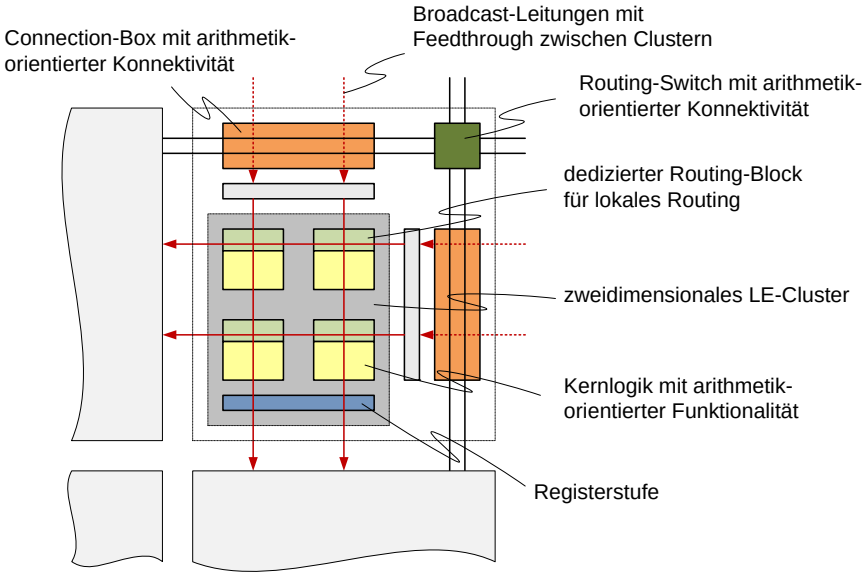


Abbildung 4.1: Übersicht des eFPGA-Architektur-Templates

4.2.2 Cluster

Eine wesentliche Besonderheit des arithmetikorientierten Architektur-Templates ist die Verwendung neuartiger zweidimensionaler Logikelement-Cluster. In herkömmlichen FPGA-Architekturen wie auch bei der VPR-konformen Architektur (Kapitel 3.5.1) und anderen aus der Forschung bekannten Architekturen werden alle Signale vom Verbindungsnetz über die zentrale Connection-Box an eine eindimensionale Anordnung von Logikelementen verschaltet. Dabei kann jedes Logikelement auf eine durch die Konnektivität F_C festgelegte Anzahl von Leitungen zugreifen. Arithmetikdatenpfade sind typischerweise in so genannten Function-Slices und Bit-Slices organisiert. Eine Function-Slice ist eine Elementarfunktion auf Wortebene, so dass die Elementaroperationen auf Bitebene für jede Wertigkeit der Daten ausgeführt werden. Eine Bit-Slice fasst alle Funktionselemente innerhalb der gleichen Bitwertigkeit zusammen. Die Daten werden häufig durch Broadcasting über mehrere Function-Slices hinweg an die Basiselemente verschaltet. Abbildung 4.2 zeigt die Organisation eines Arithmetikdatenpfades am Beispiel eines Signalflussgraphen für das in Abbildung 3.13 dargestellte ACS-Element.

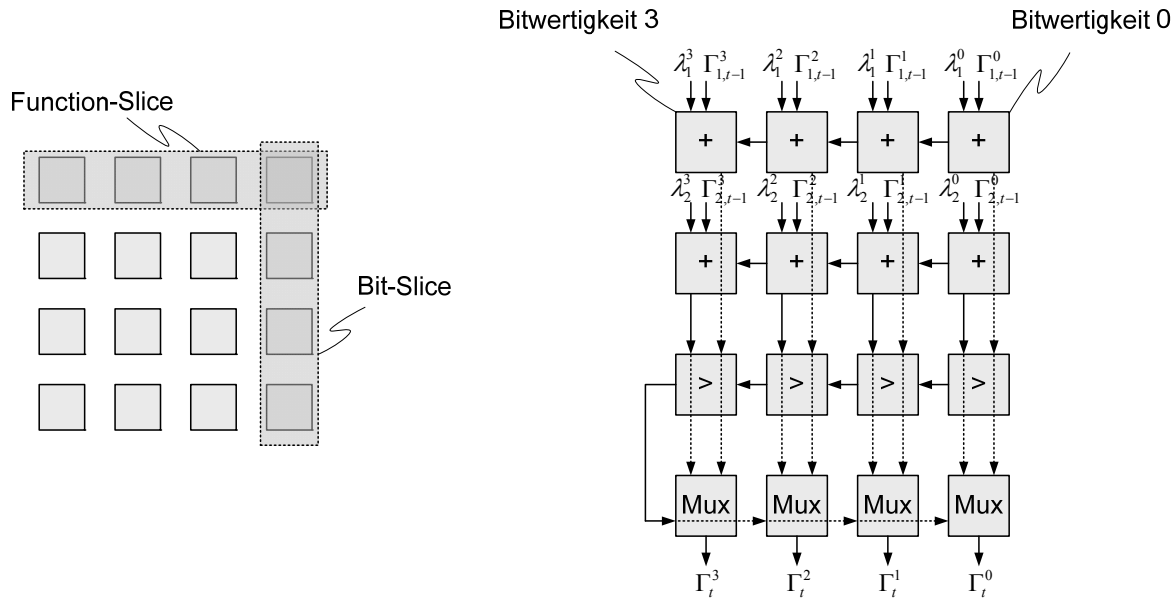


Abbildung 4.2: Organisation von Arithmetikdatenpfaden (links) und Signalflussgraph eines ACS-Elements (rechts)

Die Organisation in Bit-Slices und Function-Slices wird auch bei dem hier beschriebenen eFPGA-Architektur-Template verwendet. Dabei werden erstmals in einer eFPGA-Architektur innerhalb eines zweidimensionalen Clusters die Signale von der Connection-Box nach einem zeilen- und spaltenweisen Broadcast-Schema auf die Logikelemente verteilt. In bisherigen Ansätzen wurden stets die Logikelemente einzeln ohne ein solches Broadcast-Schema an die Connection-Box angeschlossen. Durch diese Organisation kann die Anzahl der benötigten Schaltpunkte innerhalb der Connection-Box klein gehalten werden, während für Arithmetikdatenpfade trotzdem eine hinreichend gute Konnektivität erreicht wird. Die Größe des Clusters in horizontaler und vertikaler Richtung wird durch die Parameter C_H und C_V definiert. Abbildung 4.3 zeigt den Aufbau des Clusters sowie das entsprechende Schema der Bit-Slices und Function-Slices.

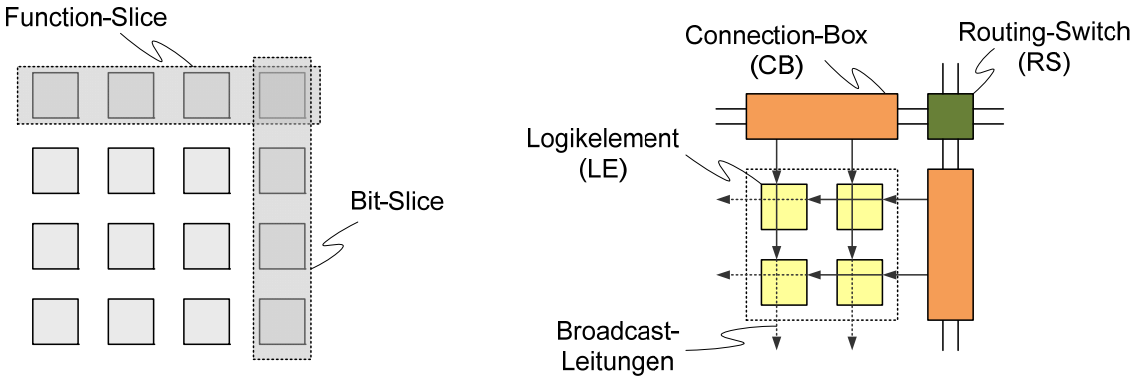


Abbildung 4.3: Organisation von Arithmetikdatenpfaden (links) und schematischer Aufbau des zweidimensionalen Clusters (rechts)

Die Cluster werden nach einem Island-Style-Schema angeordnet. Die globalen Eingangssignale, die auf die Broadcast-Leitungen geschaltet werden, können statt von der Connection-Box auch von dem benachbarten Cluster durch eine so genannte Feedthrough-Stufe zugeschaltet werden. Dadurch ist es möglich, virtuelle Cluster mit größerer Ausdehnung zu erzeugen, bei denen identische Broadcast-Signale effektiv mehrere Zeilen bzw. Spalten von Clustern beschalten. Zwischen den Logikelementen bestehen lokale Verbindungen, die über dedizierte Routing-Blöcke (DRB, siehe Kapitel 4.2.3) innerhalb der Logikelemente verschaltet werden. Diese lokalen Verbindungen bestehen auch über Cluster-Grenzen hinweg, so dass auf unterster Hierarchieebene alle Logikelemente über ein reguläres, lokales Verbindungsnetz kommunizieren können. Im Extremfall ist es sogar denkbar, dass ein besonders regelmäßiger Datenpfad nahezu ohne globale Verbindungsleitungen auf ein solches Feld von Logikelementen abgebildet werden kann. Dies ist eine Besonderheit gegenüber existierenden FPGA-Architekturen, die über keine solche vollständige Vernetzung direkter LE-Ebene verfügen. Abbildung 4.4 illustriert den Aufbau der Verbindungsarchitektur.

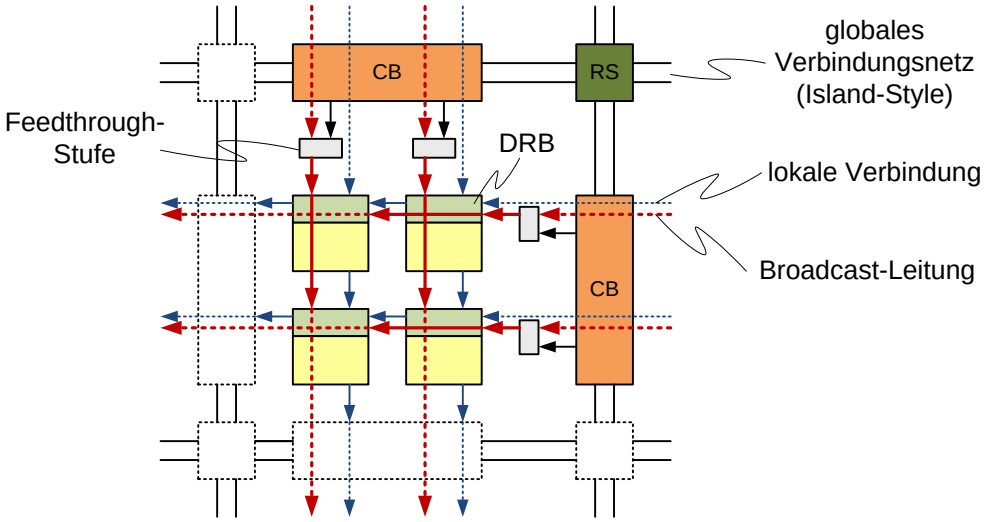


Abbildung 4.4: Lokale und globale Verbindungen im Cluster

Die Anzahl und Richtungen der Broadcast-Leitungen sind bei dem Architektur-Template frei definierbar. Durch den Parameter I_x mit $x \in \{N,W,S,E\}$ wird für jede der vier Himmelsrichtungen Nord (N), West (W), Süd (S) und Ost (E) die Anzahl der Broadcast-Leitungen pro Zeile bzw. Spalte von Logikelementen festgelegt, die von der zugehörigen Connection-Box zur Verfügung gestellt werden.

Als Ausgänge eines Clusters dienen die Ausgangssignale der Logikelemente an den jeweiligen Kanten des rechteckigen Feldes von Logikelementen. Der Parameter D_{CL} bestimmt die Richtungen der Ausgangssignale aus dem Cluster durch eine Kombination aus den vier möglichen Himmelsrichtungen $\{N,W,S,E\}$. Durch Angabe von $D_{CL}=\{S\}$ werden z.B. die Logikelemente in der untersten Reihe des Clusters an die darunter liegende Connection-Box angeschlossen. Durch geeignete Wahl von I_x und D_{CL} kann die eFPGA-Architektur so ausgelegt werden, dass der Datenfluss einer bestimmten Vorzugsrichtung folgt. Bei dem in

Abbildung 4.4 gezeigten Beispiel laufen die Signale von oben nach unten und von rechts nach links. Nicht benötigte Verbindungsrichtungen können zur Reduktion der physikalischen Implementierungskosten ausgelassen werden. Tabelle 4.1 fasst die Parameter eines Cluster zusammen.

Parameter	**Bereich**	**Beschreibung**
C_H	[*1..64*]	Anzahl LEs pro Cluster in horizontaler Richtung
C_V	[*1..64*]	Anzahl LEs pro Cluster in vertikaler Richtung
I_N	[*0..4*]	Anzahl Broadcast-Leitungen von Norden
I_W	[*0..4*]	Anzahl Broadcast-Leitungen von Westen
I_S	[*0..4*]	Anzahl Broadcast-Leitungen von Süden
I_E	[*0..4*]	Anzahl Broadcast-Leitungen von Osten
D_{CL}	$\in \{N,W,S,E\}$	Richtungen der Ausgangssignale

Tabelle 4.1: Parameter des Clusters

4.2.3 Logikelement

Obwohl das Logikelement gemäß der Darstellung in Kapitel 3.3 im Hinblick auf Fläche und Verlustleistung nur eine sekundäre Rolle gegenüber der Kommunikationsstruktur hat, ist sein innerer Aufbau von entscheidender Bedeutung für die Effizienz der eFPGA-Architektur. Durch geeignete Wahl des inneren Aufbaus ist es möglich, das globale Verbindungsnetz erheblich zu entlasten (z.B. durch Verwendung von dedizierten Carry-Chains). Bei entsprechender Auslegung der Funktionalität kann so eine möglichst direkte Abbildung der Funktionalität von Elementen des Datenpfades auf das Logikelement erfolgen, so dass keine Kaskadierung von Logikelementen mit entsprechenden Anforderungen an das Verbindungsnetz nötig ist.

Das konzipierte Logikelement unterscheidet sich erheblich von bisher bekannten Logikelementen in Standard-Architekturen wie z.B. der im VPR-Designflow verwendeten LUT-basierten Zelle. Es besteht im Wesentlichen aus einer Kernlogik, in welche die eigentliche Abbildung einer booleschen Gleichung erfolgt. Die Kernlogik entspricht damit bezüglich seiner Funktion innerhalb der Architektur dem Logikelement einer Standardarchitektur (siehe Kapitel 2.2.3). Zusätzlich steht jeder Kernlogik ein neuartiger dedizierter Routing-Block (DRB) zur Verfügung, durch welchen die lokalen Verbindungen und die Broadcast-Leitungen verschaltet werden. In bisherigen eFPGA-Architekturen erfolgt die Konnektivität ausschließlich über Connection-Boxes und gegebenenfalls über vereinzelte lokale Verbindungen, die nur innerhalb eines Clusters existieren.

Ein DRB ist bei einer Kernlogik mit I_{LE} Eingängen aus I_{LE} Multiplexern zusammengesetzt. Die Quellen, mit denen die Multiplexer-Eingänge beschaltet sind, werden als

Architekturparameter festgelegt. Als Quelle kann jeweils eine beliebige Broadcast-Leitung innerhalb der aktuellen Zeile/Spalte des Logikelements oder ein beliebiger Ausgang eines Logikelementes dienen. Bei lokalen Verbindungen werden die relative Position des Quell-Logikelements und der entsprechende Ausgang der Kernlogik (falls mehrere vorhanden sind) angegeben. Die Konnektivität der Logikelemente wird durch den komplexen Parameter *DRB* definiert. Dieser besteht aus *i* Elementen entsprechend *i* Quellen, zu denen eine Verbindung besteht. Es gilt also $DRB_i=\{t,x,y,q\}$, wobei *t* den Typ der Leitung (lokal oder Broadcast) beschreibt, während *x*, *y* und *q* die relative Position der Quelle und den Ausgang der Quelle beschreiben. Diese Beschreibungsform erlaubt es, sehr flexibel die Konnektivität zwischen den Logikelementen zu definieren. Neben Verbindungen über den dedizierten Routing-Block ist es auch möglich, dedizierte Signale zu einer benachbarten Kernlogik zu spezifizieren. Abbildung 4.5 verdeutlicht den Aufbau und die Verbindung von Logikelementen. Diese Verbindungen sind inhärenter Bestandteil der Funktionalität der Kernlogik und müssen daher nicht in der Konnektivität der Logikelemente beschrieben werden.

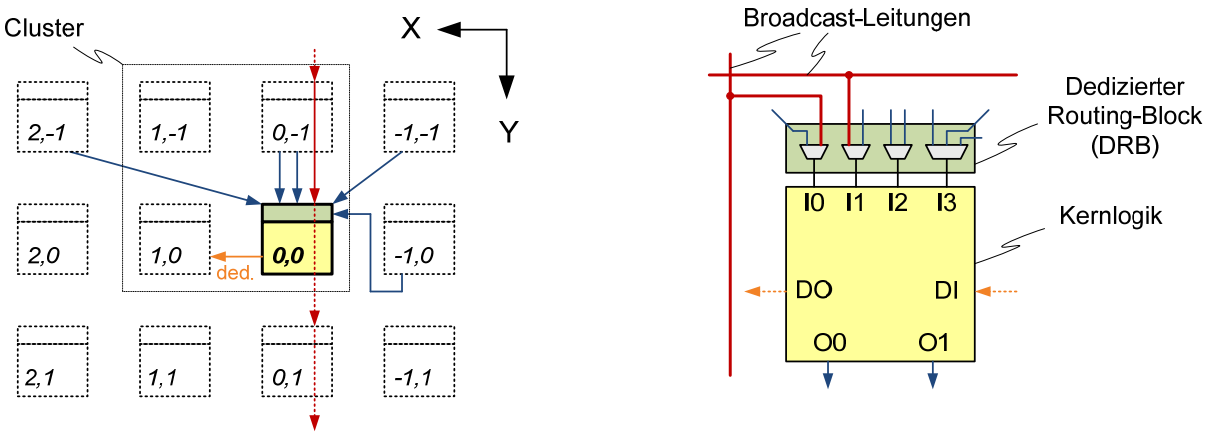


Abbildung 4.5: Lokale Verbindungen zwischen Logikelementen (links), Logikelement mit Kernlogik und dediziertem Routing-Block (rechts)

Die Funktionalität der Kernlogik wird im Parameter F_{LE} durch eine Liste von Operationen definiert, die in dem Logikelement konfiguriert werden können (z.B. LUT3, Volladdition, XOR-Verknüpfung). Dadurch ist es nicht nötig, den genauen Aufbau der Kernlogik in Form von komplexen logischen Gleichungen anzugeben. Die Anzahl der regulären Ein- bzw. Ausgänge der Kernlogik wird durch den Parameter I_{LE} bzw. O_{LE} beschrieben. Dedizierte Ein-/Ausgänge werden dabei nicht berücksichtigt. Für die in Kapitel 5.7 beschriebene Modellierung der parametrisierbaren eFPGA-Architektur wurde exemplarisch eine Kernlogik bestehend aus drei LUT2, einer dedizierten Summenlogik und einer dedizierten Carry-Chain verwendet. Abbildung 4.6 zeigt den internen Aufbau der Kernlogik.

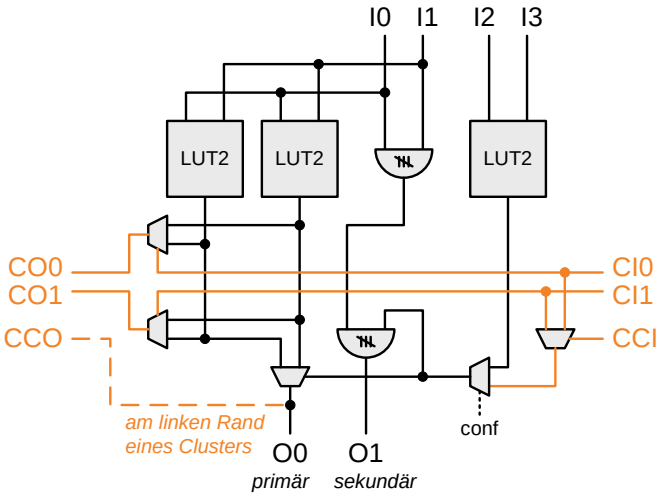


Abbildung 4.6: Aufbau der Kernlogik

Sie kann u.a. einen Volladdierer mit vorgeschaltetem Partialproduktgatter (Gated-Full-Addition), eine Basiszelle eines Carry-Select-Addierers und eine LUT3 implementieren. Dadurch kann eine Vielzahl arithmetikorientierter Basisoperationen effizient abgebildet werden. Die dedizierte Carry-Chain mit den Eingängen CI0/CI1 und den Ausgängen CO0/CO1 kann zudem benutzt werden, um mehrere LUT3 in horizontaler Richtung zu kaskadieren. Auf diese Weise können auch boolesche Funktionen mit großem Fan-In implementiert werden. Die dedizierten Eingänge können jeweils am rechten Cluster-Rand durch SRAM-Zellen fest auf die logischen Pegel '0' oder '1' programmiert werden. Pro Cluster gibt es zudem ein Block-Carry-Signal (CCI, CCO), welches zur Realisierung von Carry-Select-Addieren verwendet wird. Die Verknüpfung mehrerer Logikelemente zu einer Carry-Chain ist schematisch in Abbildung 4.7 am Beispiel eines Clusters mit $C_H=4$ dargestellt. Je größer die Cluster des eFPGAs in horizontaler Richtung sind, desto schneller wird die Carry-Select-Chain im Vergleich zu einer Carry-Ripple-basierten Lösung.

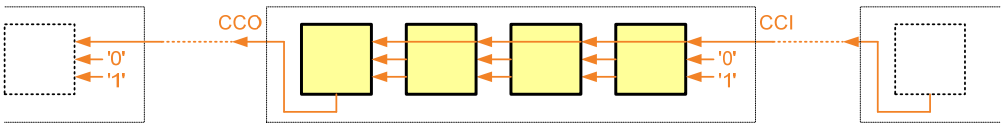


Abbildung 4.7: Carry-Chain

Die Kernlogik enthält nur kombinatorische Logik. Für die Implementierung sequentieller Schaltungen können zusätzlich Register spezifiziert werden. Die Anzahl der benötigten Register hängt in der Regel stark von der betrachteten Anwendungsklasse ab. Daher kann bei dem hier beschriebenen Architektur-Template festgelegt werden, welche Ausgangssignale der Kernlogik durch ein Register laufen. Da bei Arithmetikdatenpfaden häufig Pipeline-Schnitte in horizontaler Richtung auftreten, können zusätzliche Register für die vertikalen Broadcast-Leitungen (N, S) eingefügt werden.

Um die Effizienz für Anwendungsklassen mit moderatem bis geringen Pipelining-Grad zu steigern, kann festgelegt werden, dass in einem Cluster nur alle N_{Reg} LE-Zeilen ein Register eingefügt wird. Jedes Register verfügt über einen Bypass, so dass auch rein kombinatorische Schaltungen ohne Register implementiert werden können. Tabelle 4.2 fasst die Parameter des Logikelements zusammen.

Parameter	**Bereich**	**Beschreibung**
I_{LE}	$[2..8]$	Anzahl Eingänge der Kernlogik
O_{LE}	$[1..4]$	Anzahl Ausgänge der Kernlogik
F_{LE}		Liste mit Funktionen
DRB_i	$DRB_i=\{t,x,y,q\}$ *mit* $t \in \{lo,bc\}$ $x,y=[0..32]$ $q=[1..O_{LE}]$	Liste aller Quellen des DRB
N_{Reg}	$[0..C_V]$	Anzahl der LEs pro Register (spaltenweise), =0 für Architektur ohne Register
O_{Reg}	$\{1..O_{LE} / 1..I_{N/S}\}$	Ausgänge mit Register (lokal und Broadcast)

Tabelle 4.2: Parameter des Logikelements

Im Folgenden werden einige exemplarische Konfigurationen der Kernlogik diskutiert, die im Rahmen der durchgeführten Analysen verwendet wurden. Bei der in Abbildung 4.8 dargestellten Konfiguration zweier benachbarter Logikelemente als Carry-Select-Addierer werden die LUTs für die Vorabberechnung zweier Terme benutzt, die durch Verknüpfung über die dedizierten Carry-Select-Multiplexer die beiden Carry-Signale bilden. Die entsprechende logische Gleichung ist in der Abbildung angegeben. Das Block-Carry-Signal steht jeweils für ein gesamtes Cluster identisch zur Verfügung und wird am linken Clusterrand am primären LE-Ausgang generiert.

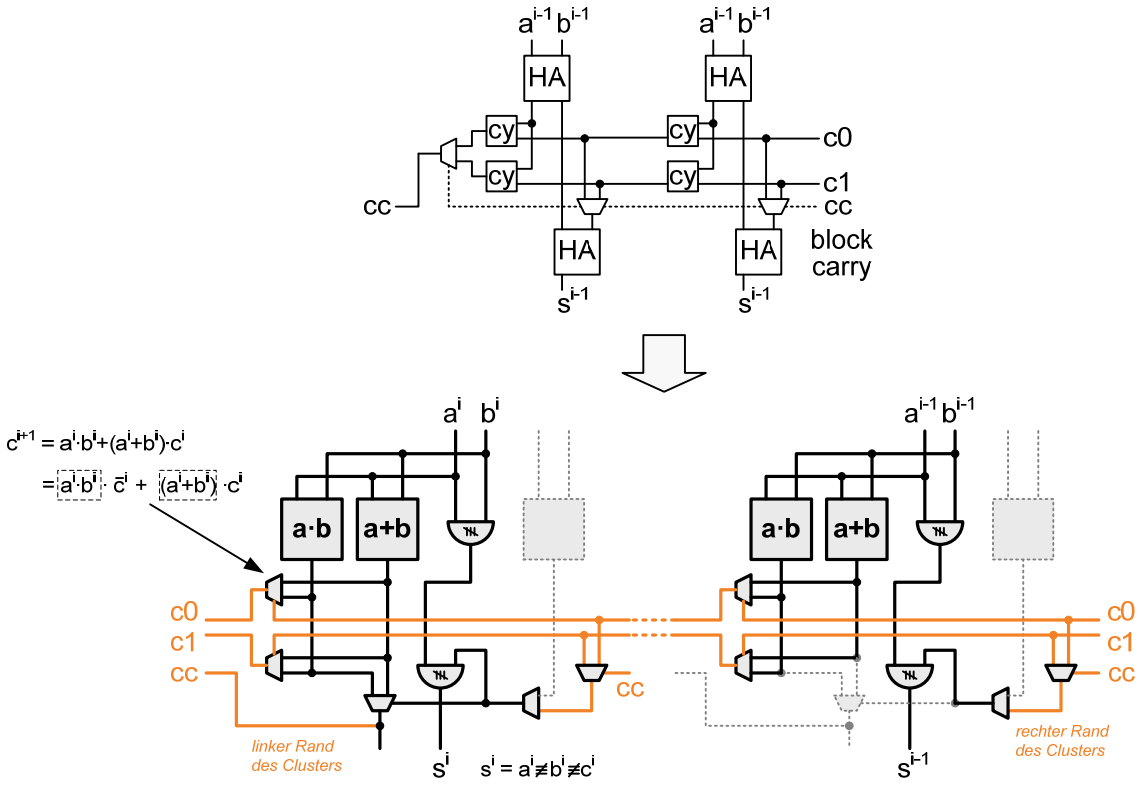


Abbildung 4.8: Konfiguration als Carry-Select-Addierer

Die Abbildung verdeutlicht durch farbliche Unterlegung der tatsächlich allokierten Elemente die effiziente Auslastung der zur Verfügung stehenden Ressourcen. Der zeitkritische Pfad des so implementierten Carry-Select-Addierers wird nur noch durch dedizierten Multiplexer gebildet.

Eine ebenfalls häufig verwendete Konfiguration ist die Gated-Full-Addition, die z.B. als Kernelement von Array-Multiplizierern benötigt wird. Abbildung 4.9 zeigt die entsprechende Konfiguration der Kernlogik. Die dritte LUT wird dabei als Partialproduktgatter konfiguriert. Der Carry-Pfad wird über den primären Ausgang geschaltet, da die Verwendung eines Carry-Select-Schemas bei Array-Multiplizierern nachteilig bezüglich des zeitkritischen Pfades ist.

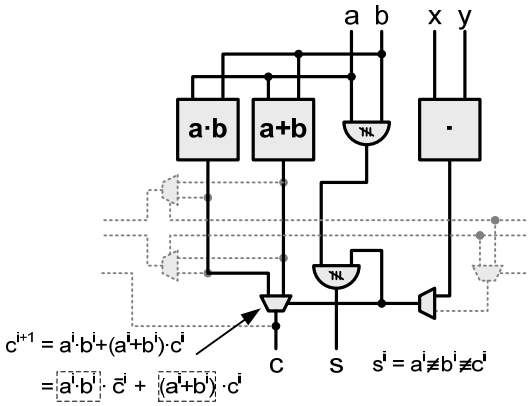


Abbildung 4.9: Konfiguration als Gated-Full-Adder

Weiterhin wird in Arithmetikdatenpfaden häufig eine boolesche Verknüpfung zweier Operanden mit mehreren Bits durchgeführt. Derartige boolesche Funktionen mit großem Fan-In werden z.B. bei der Korrelationsrechnung verwendet. Abbildung 4.10 verdeutlicht die entsprechende Konfiguration der Kernlogik in diesem Modus. Durch Zusammenschalten von zwei LUT2 wird eine LUT3 gebildet, deren dritter Eingang mit einem der dedizierten Carry-Eingänge beschaltet wird. So werden mehrere Logikelemente in einer Reihe effizient kaskadiert.

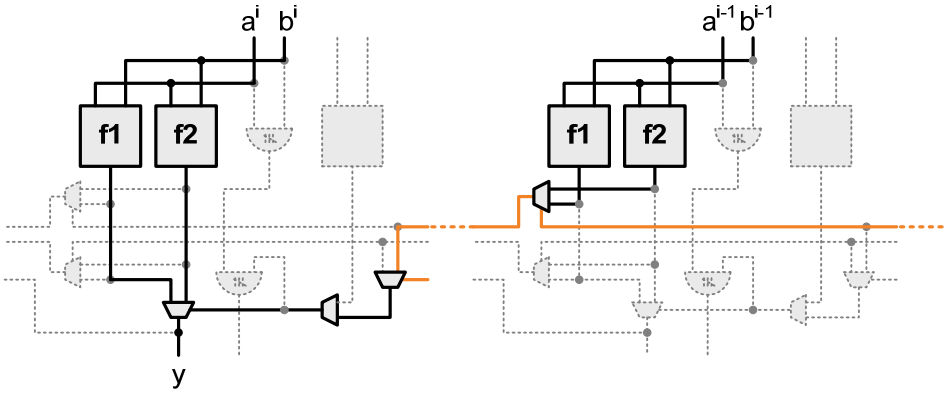


Abbildung 4.10: Konfiguration als boolesche Funktion mit großem Fan-In

4.2.4 Routing-Switch

Das Verbindungsnetz der arithmetikorientierten eFPGA-Architektur basiert auf dem bekannten Island-Style. Die Anzahl der Leitungen in horizontaler und vertikaler Richtung kann getrennt durch die Parameter W_H und W_V festgelegt werden. Statt nur eine Auswahl festgelegter Konnektivitäts-Schemata zuzulassen (wie bei VPR), werden die Positionen $P_{RS,k}$ aller Switch-Points in der $W_H \times W_V$-Matrix als Koordinaten angegeben. Abbildung 4.11 zeigt den schaltungstechnischen Aufbau eines exemplarisch hier betrachteten Switch-Points mit reduzierter Flexibilität.

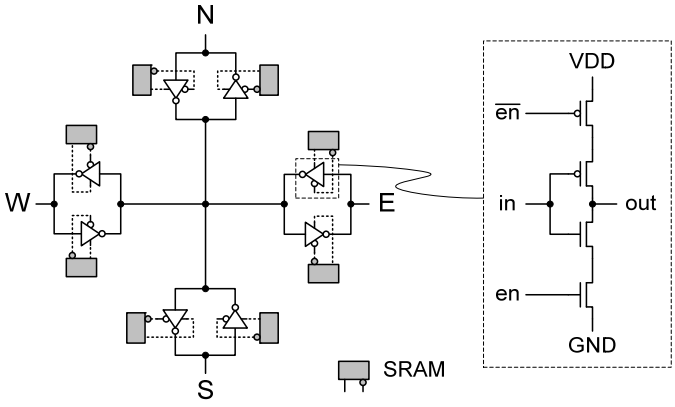


Abbildung 4.11: Schaltbild Switch-Point

Statt Tristate-Treibern werden hier Tristate-Inverter verwendet. Da jedes am Eingang invertierte Signal ausgangsseitig ebenfalls invertiert wird, erfolgt im Switch-Point keine logische Invertierung. Für jede angeschlossene horizontale und vertikale Leitung kann die Segmentlänge $L_{V,i}$ bzw. $L_{H,j}$ frei definiert werden. Um eine einheitliche Architektur für alle Routing-Switches des eFPGAs zu erhalten, werden Segmente mit L_i>1 gestaffelt angeordnet (siehe Kapitel 2.2.2). Abbildung 4.12 illustriert die Parameter des Routing-Switch.

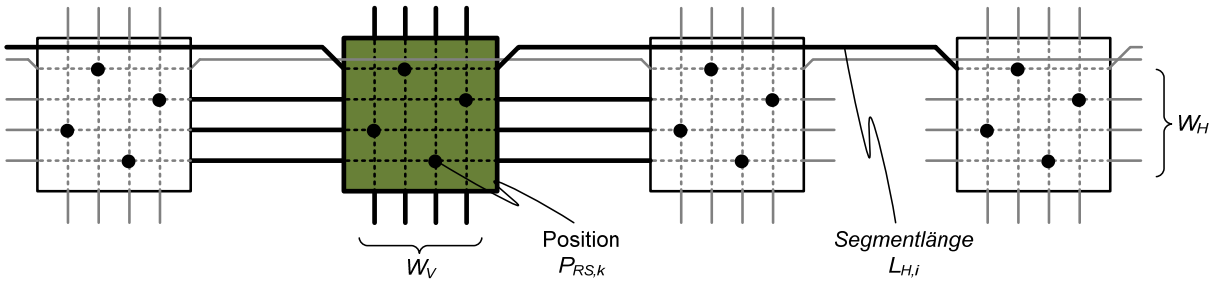


Abbildung 4.12: Parametrisierbarer Routing-Switch

Neben der Gesamtarchitektur des Routing-Switch kann für jeden definierten Switch-Point detailliert festgelegt werden, welche Verbindungswege konfiguriert werden können. Der Parameter SP_k definiert für jeden Switch-Point k eine Anzahl möglicher Verbindungswege zwischen den vier Ports Nord, West, Süd und Ost. Es können neben einzelnen Verbindungen (z.B. $N \rightarrow S$) auch Varianten definiert werden, bei denen zwei unabhängige Verbindungen gleichzeitig konfiguriert werden (z.B. $N \rightarrow S, E \rightarrow W$) oder ein Eingang zu zwei oder drei Ausgängen geschaltet wird (z.B. $N \rightarrow S+E$). Die Anzahl der schaltbaren Verbindungswege und damit die Flexibilität des Switch-Points hat erheblichen Einfluss auf die Komplexität seiner physikalischen Implementierung. So kann z.B. durch gezieltes Wählen von Verbindungen, die nur gewisse Vorzugsrichtungen schalten können, die Anzahl der benötigten Schalter im Switch-Point deutlich reduziert werden. Abbildung 4.13 zeigt exemplarisch einige konfigurierbare Verbindungswege und die zugehörige Nomenklatur.

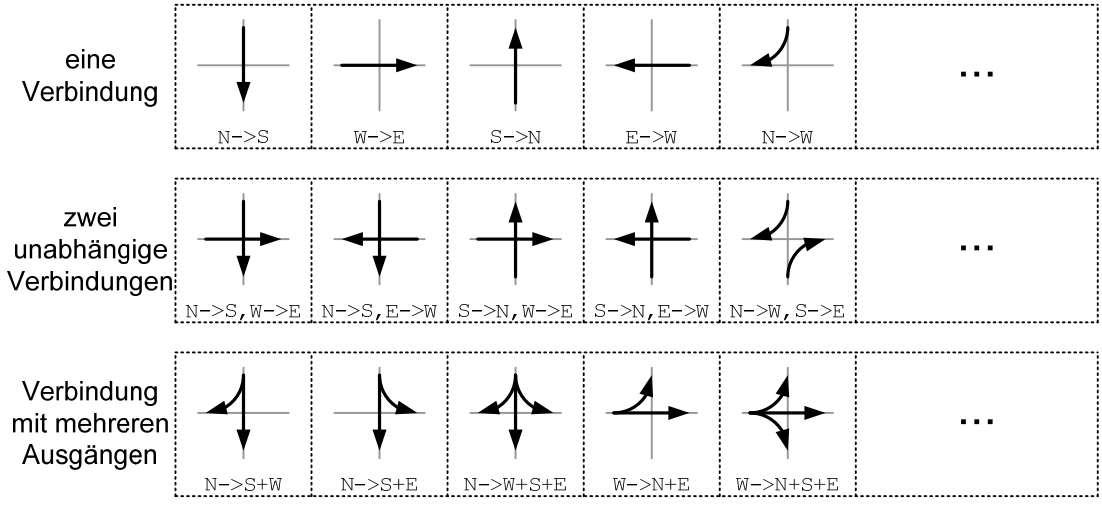


Abbildung 4.13: Exemplarische konfigurierbare Verbindungswege im Switch-Point

Die Zusammenfassung aller definierbaren Parameter des Routing-Switch ist in Tabelle 4.3 dargestellt.

Parameter	**Bereich**	**Beschreibung**
W_H	[*1..256*]	Anzahl horizontaler Verbindungsleitungen
W_V	[*1..256*]	Anzahl vertikaler Verbindungsleitungen
$L_{H,i}$	[*1..256*]	Segmentlängen der horizontalen Leitungen
$L_{V,j}$	[*1..256*]	Segmentlängen der vertikalen Leitungen
$P_{RS,k}$	$P_{RS,k}=\{x,y\}$ *mit* $x=[1..W_H]$ $y=[1..W_V]$	Positionen der Switch-Points
SP_k	$SP_k=\{q_1 \rightarrow s_{1,i} / q_2 \rightarrow s_{2,j}\}$ $q_i,s_i \in \{N,W,S,E\}$ *mit* $i,j=[1..3]$	konfigurierbare Verbindungswege pro Switch-Point {Quelle → Senken}

Tabelle 4.3: Parameter des Routing-Switch

4.2.5 Connection-Box

Neben den Routing-Switches haben die Connection-Boxes einen wesentlichen Anteil am hohen Flächenbedarf und Energieumsatz von FPGAs. Es ist daher wichtig, nur soviel Flexibilität vorzusehen, wie tatsächlich für eine Anwendungsklasse benötigt wird. Bei dem hier beschriebenen eFPGA-Architektur-Template können daher drei Klassen von Verbindungskanälen mit unterschiedlichen Implementierungskosten definiert werden: nicht angeschlossen (NC), volle Konnektivität (FC) und periodische Konnektivität (PC). Im ersten Fall werden Leitungen in der Connection-Box nicht verbunden, sie dienen dann lediglich als Verbindungen zwischen Routing-Switches. Gemäß Kapitel 2.2.5 gilt für diese Kanäle $F_C=0$. Es muss bei der Definition des Routing-Switch beachtet werden, dass solche Leitungen an anderer Stelle wieder mit der Connection-Box verbunden sind. Der Vorteil nicht verbundener Leitungen liegt in der geringen kapazitiven Last, da keine Tristate-Treiber als Senken angeschlossen sind. Diese Leitungen können also für besonders schnelle und verlustleistungsarme Verbindungen verwendet werden. Für Signale, die eine hohe Flexibilität der Connection-Box erfordern, können Kanäle mit voller Konnektivität definiert werden. Bei diesen gilt $F_C=1W$. Aufgrund der hohen Implementierungskosten für diese Kanäle sollten nur wenige Leitungen z.B. für Steuersignale bei Datenpfaden vorgesehen werden. Eine wesentliche Neuerung gegenüber existierenden Connection-Box-Architekturen sind spezielle Kanäle mit periodischem Zugriffsschema, die sich besonders für die Abbildung von Arithmetikdatenpfaden eignen. Bei diesen Kanälen ändert sich die Position der programmierbaren Schalter entlang einer Leitung nach einem einstellbaren Wertigkeitsschema. Dadurch können z.B. benachbarte Logikelemente auch auf benachbarte

Verbindungsleitungen zugreifen, ohne den Implementierungsaufwand eines Kanals mit voller Konnektivität in Kauf nehmen zu müssen. Abbildung 4.14 verdeutlicht das Prinzip einer Connection-Box mit Wertigkeitsschema. Die vier dargestellten Logikelemente werden mit den Eingangssignalen x^0 bis x^3 entsprechend ihrer Wertigkeiten verschaltet. Bei der links im Bild dargestellten Connection-Box werden nur die dafür benötigten Verbindungen vorgehalten, während bei der rechts dargestellten konventionellen Connection-Box durch einen Multiplexer die Eingangsleitung frei gewählt werden kann.

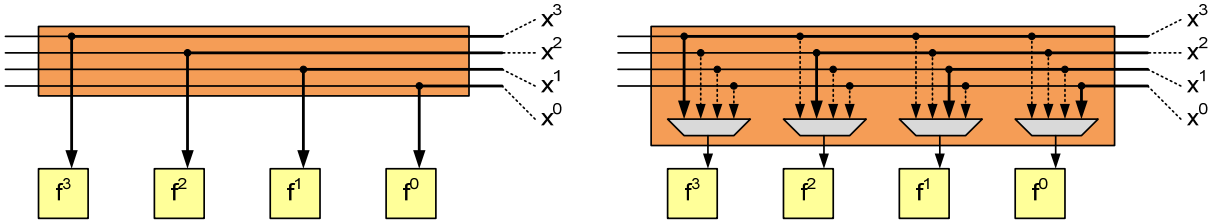


Abbildung 4.14: Connection-Box mit (links) und ohne (rechts) Wertigkeitsschema

Periodische Verbindungen werden durch einen Satz von Leitungen des Kanals, das so genannte Fenster, w_{PC} definiert. Die Verschiebung dieses Fensters entlang des Kanals wird durch die Geschwindigkeit v_{PC} (entsprechend der Anzahl an Leitungen, um die das Fenster pro Schritt weitergeschoben wird) und die Phasenlage p_{PC} (Anzahl der Schritte, nach denen das Fenster durch die Periodizität wieder in der Ausgangsposition ist) definiert.

Die Definition der Connection-Box erfolgt unabhängig voneinander für die horizontalen (H) und vertikalen (V) Verbindungsleitungen. Die Anzahl der Leitungen pro Verbindungskanal wird für die drei Klassen jeweils durch den Parameter r angegeben, so dass gilt:

$$r_{FC,H} + r_{NC,H} + \sum_i r_{PC,H,i} = W_H \tag{4.2}$$

$$r_{FC,V} + r_{NC,V} + \sum_j r_{PC,V,j} = W_V \tag{4.3}$$

Abbildung 4.15 verdeutlicht das Konzept der parametrisierbaren Connection-Box anhand eines Beispiels. In der Abbildung sind nur die Verbindungen von der Connection-Box zu den darunter liegenden Eingängen der Logikelemente dargestellt. Die Verbindungen von den Ausgängen der Logikelemente zur Connection-Box hin werden in gleicher Weise behandelt.

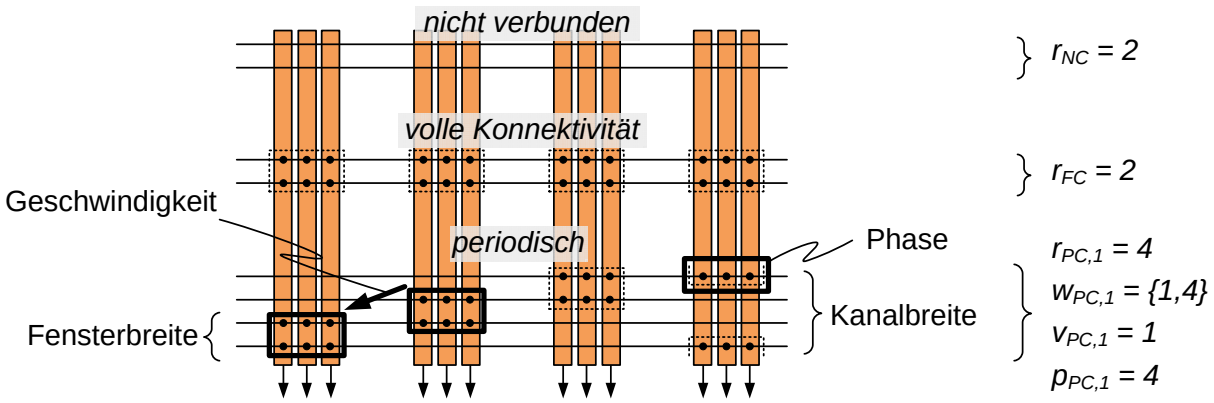


Abbildung 4.15: Parametrisierbare Connection-Box

In Tabelle 4.4 sind alle Parameter der Connection-Box aufgeführt. Der Index *H* bzw. *V* kennzeichnet dabei jeweils, ob es sich um die horizontale oder vertikale Connection-Box handelt.

Parameter	**Bereich**	**Beschreibung**
$r_{NC,H/V}$	$[0..256]$	Anzahl Leitungen ohne Konnektivität ($F_C=0$)
$r_{FC,H/V}$	$[0..256]$	Anzahl Leitungen mit voller Konnektivität ($F_C=1$)
$r_{PC,H/V,i}$	$[0..256]$	Anzahl Leitungen mit periodischer Konnektivität
$w_{PC,H/V,i}$	$\in \{1..r_{PC,H/V,i}\}$	Fensterbreiten der periodischen Kanäle
$v_{PC,H/V,i}$	$[1..r_{PC,H/V,i}-1]$	Geschwindigkeit der Fenster
$p_{PC,H/V,i}$	$[1..r_{PC,H/V,i}]$	Periodizität der Fenster

Tabelle 4.4: Parameter der Connection-Box

4.2.6 Konfigurationsspeicher

Auf Basis der im Rahmen dieser Arbeit implementierten eFPGA-Makros wurde der Flächenanteil der SRAMs an der Gesamtfläche eines eFPGA zu etwa 50% abgeschätzt. Gerade bei modernen Halbleitertechnologien bedeutet das neben hoher Kosten durch Siliziumfläche auch eine erhebliche Verlustleistungsaufnahme durch Sub-Threshold-Leakage. Da Arithmetikdatenpfade häufig mit Wortbreiten $w>1$ arbeiten, sind innerhalb einer Function-Slice häufig mehrere identische Basisoperationen vorhanden (siehe Kapitel 4.2.2). In einer FPGA-Architektur entspricht das identisch konfigurierten Logikelementen. Durch die Verwendung von so genannten shared-SRAMs ist es möglich, den Anteil der SRAMs in einem eFPGA erheblich zu verringern, indem mehrere nebeneinander liegende Logikelemente mit identischen SRAM-Zellen konfiguriert werden. Abbildung 4.16 verdeutlicht das Konzept.

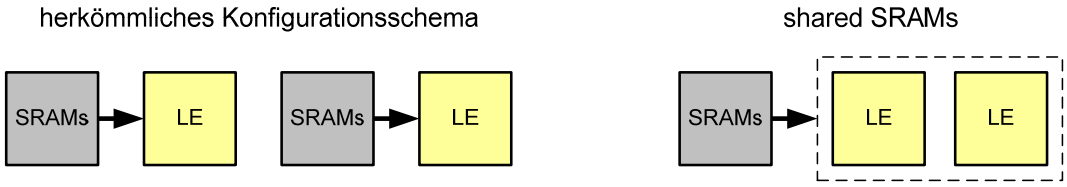


Abbildung 4.16: Shared-SRAMs für Logikelemente

Beim Abbilden von Datenpfaden auf die Logikelemente wird im Folgenden die Anzahl der gemeinsam konfigurierten Logikelemente als Granularität *M* bezeichnet, während die Anzahl benachbarter Elemente mit identischer Funktionalität innerhalb einer Function-Slice des Datenpfades als Homogenität *H* bezeichnet wird. Abbildung 4.17 verdeutlicht die Definition von *M* und *H*.

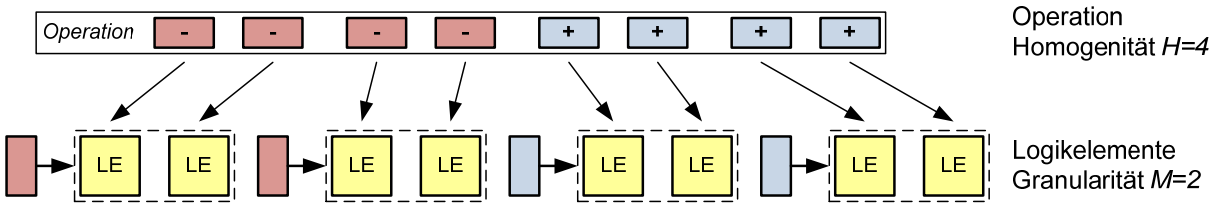


Abbildung 4.17: Homogenität H und Granularität M

Das gleiche Konzept lässt sich auf die Connection-Boxes und Routing-Switches übertragen, in denen die Daten ebenfalls auf Wortebene verschaltet werden. Die Wahl der Konfigurationsgranularität hat erheblichen Einfluss auf die physikalischen Implementierungskosten eines eFPGA, allerdings kann es durch Wahl einer zu groben Granularität zu Verschnitteffekten kommen. In diesem Fall können gegebenenfalls mehrere Elemente nicht verwendet werden, weil die Konfiguration nur eine einzelne Bit-Slice betrifft. Dies kann z.B. in Datenpfaden beim MSB auftreten, das häufig anders behandelt wird als die übrigen Wertigkeiten. Abbildung 4.18 illustriert diesen Effekt an einem einfachen Beispiel für eine Operation mit vier Bit Wortbreite.

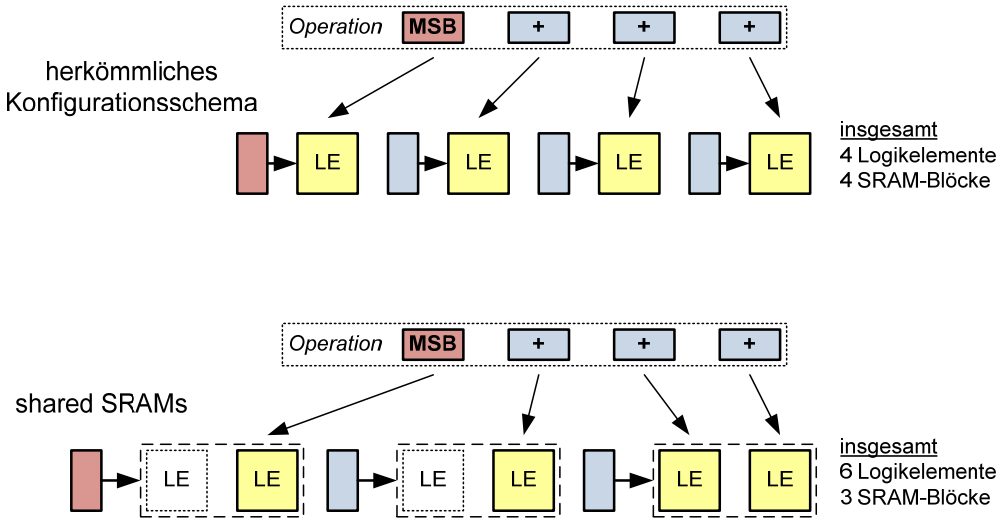


Abbildung 4.18: Verschnitteffekte bei Verwendung von shared-SRAMs

Bei dem oben gezeigten Konfigurationsschema ohne shared-SRAMs wird jede Bit-Slice des Datenpfades mit einem eigenen SRAM-Block konfiguriert. Neben den vier eigentlichen Logikelementen werden daher vier SRAM-Blöcke benötigt. Bei dem unten dargestellten Beispiel werden jeweils zwei Logikelemente von einem identischen SRAM-Block konfiguriert. Die beiden niederwertigsten Bits können dabei ohne Verschnitt auf zwei benachbarte Logikelemente abgebildet werden, da sie eine identische Konfiguration erfordern. Die verbleibenden Bits erfordern unterschiedlich konfigurierte Logikelemente, so dass sie auf zwei Logikelemente mit verschiedenen Konfigurationsspeichern abgebildet werden müssen. Dadurch bleiben zwei Logikelemente ungenutzt. Insgesamt werden dadurch bei dieser Anordnung sechs Logikelemente und drei SRAM-Blöcke statt vier Logikelementen und vier SRAM-Blöcken benötigt. Da der Flächenaufwand für die SRAM-Blöcke etwa mit dem für die Logikelemente vergleichbar ist, wird bei Verwendung von shared-SRAMs in dem Beispiel aus Abbildung 4.18 effektiv mehr Chipfläche benötigt als bei einem FPGA ohne shared-SRAMs. Die Verwendung von shared-SRAMs ist daher nur sinnvoll, wenn die abzubildenden Datenpfade ein hinreichend großes Maß an Homogenität in den Konfigurationsdaten aufweisen. Dieser Zusammenhang wird im Folgenden analysiert.

Bildet man einen Operanden mit der Homogenität H auf Logikelemente mit der Granularität M ab, so werden dafür n_{LE} Logikelemente und n_{cf} Konfigurationsspeicher benötigt. Es gilt:

$$n_{LE} = M \cdot \left\lceil \frac{H}{M} \right\rceil \quad und \quad n_{cf} = \left\lceil \frac{H}{M} \right\rceil \tag{4.1}$$

Die insgesamt benötigte Fläche zur Implementierung des Operanden ist die Summe aus den Teilflächen von Logikelementen und den zugehörigen SRAM-Zellen:

$$A = \left(A_{LE} \cdot n_{LE} + A_{cf} \cdot n_{cf} \right) \tag{4.2}$$

Da das Verhältnis des Flächenbedarfs von Logikelementen und zugehörigen Konfigurations-SRAMs ungefähr 1:1 ist, ergibt sich aus den Gleichungen (*4.1*) und (*4.2*) der gesamte Flächenbedarf bei Normierung auf *M=1* und *H=1* wie folgt:

$$A = \frac{1}{H}\left(0{,}5 \cdot M \cdot \left\lceil \frac{H}{M} \right\rceil + 0{,}5 \cdot \left\lceil \frac{H}{M} \right\rceil \right) \tag{4.3}$$

Bei der implementierten Beispielarchitektur mit *M=4* ergibt sich im günstigsten Fall (Operand mit *H=4*) der relative Flächenbedarf wie folgt:

$$A_{opt} = \min\left(\frac{1}{H}\left(0{,}5 \cdot M \cdot \left\lceil \frac{H}{M} \right\rceil + 0{,}5 \cdot \left\lceil \frac{H}{M} \right\rceil \right) \right)\Bigg|_{M=4} = 62{,}5\% \tag{4.4}$$

Abbildung 4.19 stellt die Abhängigkeit des Flächenbedarfs von der Homogenität des abgebildeten Operanden und der shared-SRAMs am Beispiel eines 16-Bit-Operanden grafisch

dar. Das Diagramm verdeutlicht, wie durch ungünstig gewählte eFPGA-Architekturparameter der Flächenbedarf eines abgebildeten Operanden durch Verschnitteffekte bzw. Wahl eines zu großen Wertes für *M* stark ansteigen kann.

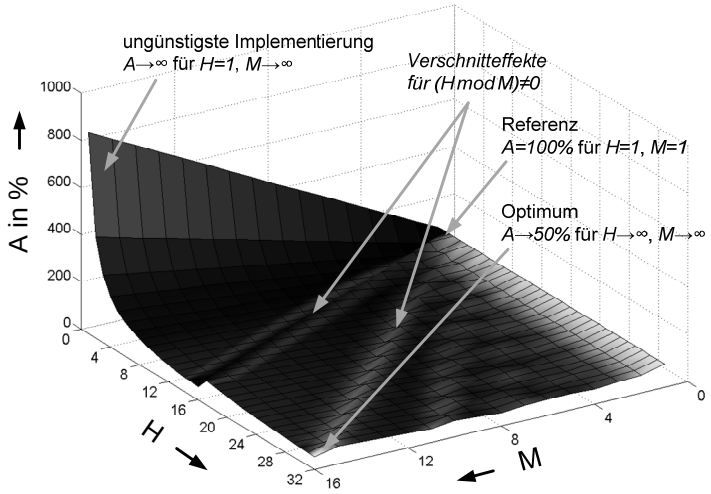


Abbildung 4.19: Flächenbedarf bei Mapping eines Operanden mit gegebener Homogenität auf Logikelemente mit shared-SRAMs

Um die Homogenität der Konfigurationsinformationen an komplexeren Beispielen untersuchen zu können, ohne diese auf die implementierte Beispielarchitektur abbilden zu müssen, wurde ein Analysewerkzeug entwickelt, dass sich in die Entwicklungsumgebung Quartus II für FPGAs der Firma Altera einbinden lässt. Dieser Konfigurationsanalysator liest die von Quartus generierte Netzliste nach dem Place&Route ein und analysiert die auf die Logikelemente abgebildeten logischen Gleichungen. Dazu werden die entsprechenden Gleichungen aus der Netzliste zunächst mit SIS (siehe Kapitel 3.5.2) in die Disjunktive Normalform (DNF) umgeformt und alle Signalnamen gegen allgemeine Platzhalter ausgetauscht. Weiterhin wird der allgemeine Operationsmodus der Logikelemente (arithmetisch, normal) aus der Netzliste ausgelesen. Dieser beschreibt z.B., ob die Carry-Chain verwendet wird. Anhand der so verbleibenden Struktur der logischen Gleichung kann die Konfiguration des Logikelementes bestimmt werden. Durch Vergleich der abgebildeten Gleichungen für alle Logikelemente innerhalb eines LABs (entspricht einem Cluster mit zehn Logikelementen, siehe Anhang A) kann die Homogenität der Konfigurationsinformation auf Cluster-Ebene für beliebige mit Quartus synthetisierte Datenpfade ermittelt werden. Im Rahmen der durchgeführten Untersuchungen wurden u.a. ein 32-Bit-Addierer und ein 32-Bit-Multiplizierer aus der Megafunction-Bibliothek von Quartus implementiert, die im Folgenden als *Add32 (MF)* und *Mul32 (MF)* bezeichnet werden. Megafunctions sind vorgefertigte IP-Blöcke, die von Altera mitgeliefert werden und die besonders für die entsprechenden FPGA-Architekturen optimiert sind. So wird z.B. sichergestellt, dass der Addierer die dedizierte Carry-Chain verwendet. Neben den beiden Megafunctions wurde der Multiplizierer zusätzlich als handoptimierter Array-Multiplizierer, im Folgenden als *Mul32 (Array)*

bezeichnet, implementiert, der einen hochregulären Aufbau hat. Schließlich wurde als komplexeres Beispiel ein Reed-Solomon-Decoder gemäß [22] implementiert. Die Datenpfade wurden exemplarisch für einen APEX 20KE und für einen Cyclone I synthetisiert. Die Logikelemente des APEX haben eine geringere Komplexität, insbesondere ist die Carry-Chain einfacher aufgebaut. Das Logikelement des Cyclone hat bei Verwendung der Carry-Chain verschiedene Konfigurationen, die innerhalb eines Clusters alternierend verwendet werden. Dies führt dazu, dass für diesen Baustein die Homogenität des Datenpfades bei der Abbildung auf die Logikelemente teilweise verloren geht. Abbildung 4.20 und Abbildung 4.21 zeigen die Histogramme der Konfigurationshomogenität innerhalb eines LABs für die beiden FPGAs. Zu diesem Zweck ist die Häufigkeit von LABs über der Anzahl verschiedener LE-Konfigurationen innerhalb dieser LABs aufgetragen.

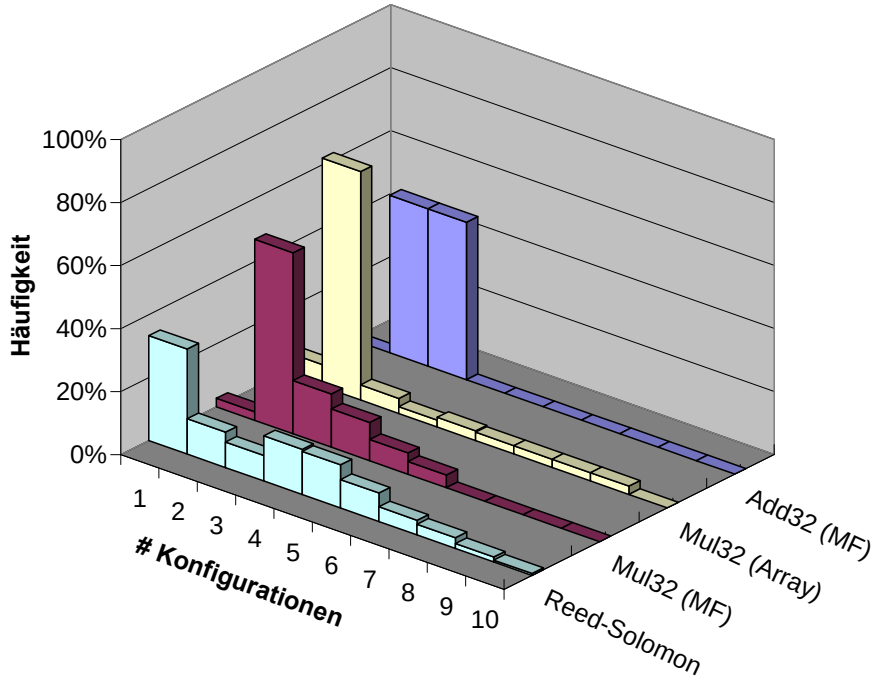


Abbildung 4.20: Histogramm der Konfigurationshomogenität (Altera APEX 20KE)

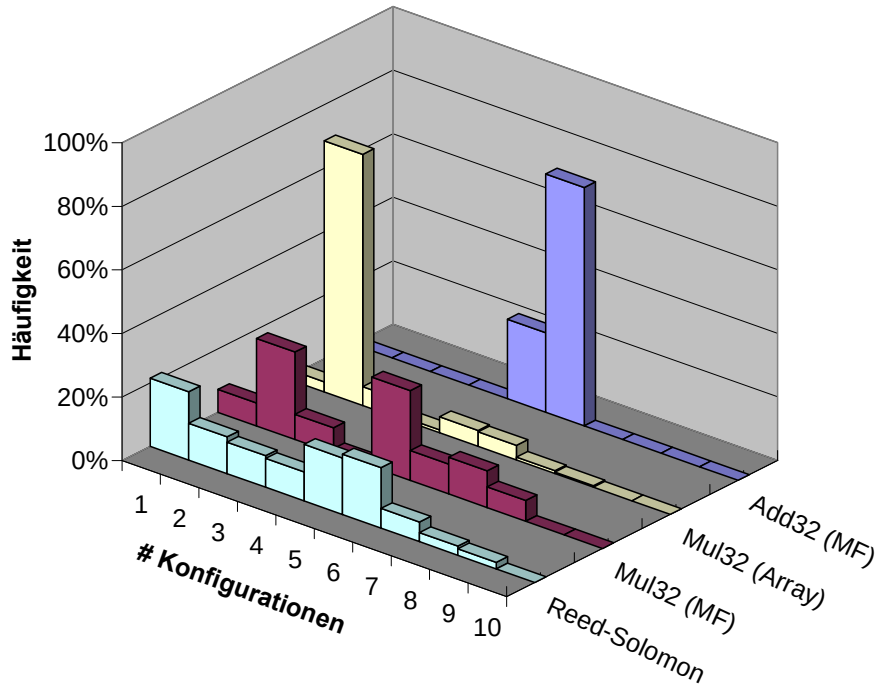


Abbildung 4.21: Histogramm der Konfigurationshomogenität (Altera Cyclone I)

Die Diagramme zeigen, dass gerade für Basisoperationen wie *Mul32* und *Add32* eine Vielzahl von Clustern nur zwei verschiedene LE-Konfigurationen verwendet. Die Untersuchung ergibt, dass in den Konfigurationsdaten von Datenpfaden ein erhebliches Maß an Homogenität existiert, welches durch die Verwendung von shared-SRAMs potenziell zur Verbesserung der Flächeneffizienz genutzt werden kann. Dabei spielt neben der Häufigkeit einer Konfiguration innerhalb eines Cluster auch die genaue Anordnung der verschiedenen Konfigurationen eine Rolle. Für die Logikelemente des APEX bzw. Cyclone ergibt sich durch den internen Aufbau eine ungünstige Verteilung mit einem alternierenden Konfigurations-schema. Die im Rahmen dieser Arbeit implementierten eFPGA-Architekturen sind dagegen so aufgebaut, dass bei der Verwendung der Carry-Chain benachbarte Logikelemente identisch konfiguriert sind. Dadurch kann die Homogenität der Konfigurationsinformationen besser ausgenutzt werden.

Das arithmetikorientierte Architektur-Template sieht die Verwendung von shared-SRAMs für Logikelemente, Routing-Switches und Connection-Boxes mit einstellbarer Granularität vor. Durch die Parameter M_{LE}, M_{RS} bzw. M_{CB} wird festgelegt, wie viele Basiselemente identische Konfigurationsspeicher verwenden. Für alle drei Parameter gilt, dass nur direkt nebeneinander liegende Elemente gruppiert werden können. In Tabelle 4.5 sind die Parameter der shared-SRAMs mit ihren Definitionsbereichen zusammengefasst.

Parameter	Bereich	Beschreibung
M_{LE}	[$1..C_H$]	Anzahl Logikelemente mit identischen SRAMs
M_{RS}	[$1..64$]	Anzahl Switch-Points mit identischen SRAMs
M_{CB}	[$1..64$]	Anzahl progr. Schalter der Connection-Box mit identischen SRAMs

Tabelle 4.5: Parameter der shared-SRAMs

4.2.7 Sonstiges

Ein eFPGA-Makro gemäß der Beschreibung durch das Architektur-Template wird durch reguläres Aneinandersetzen („Kacheln“) der einzelnen Cluster erzeugt. Die Größe des gesamten eFPGA wird dabei durch die Anzahl der Logikelemente in horizontaler und vertikaler Richtung mit den Parametern S_H und S_V festgelegt. Alle Logikelemente, Routing-Switches und Connection-Boxes des eFPGAs sind dabei identisch aufgebaut. Es ist also z.B. nicht möglich, verschiedene Logikelemente innerhalb eines eFPGAs zu verwenden. In Tabelle 4.6 sind die Parameter zur Definition der Makrogröße sowie die zugehörigen Definitionsbereiche angegeben.

Parameter	Bereich	Beschreibung
S_H	[$C_H..256 \cdot C_H$]	Anzahl LEs (gesamt) in horizontaler Richtung
S_V	[$C_V..256 \cdot C_V$]	Anzahl LEs (gesamt) in vertikaler Richtung

Tabelle 4.6: Sonstige Parameter

An den Rändern des Makros enden sämtliche Leitungen des globalen und lokalen Verbindungsnetzes unabhängig von ihrer Segmentlänge. Die Routing-Switches entsprechen damit auch an den Rändern vollständig der Architekturspezifikation. Das kann zu einem geringfügigen Mehraufwand für nicht sinnvoll benutzbare Verbindungsressourcen führen, allerdings vereinfacht es die VLSI-Implementierung und Architekturbeschreibung erheblich. Grundsätzlich können alle an den Rändern unbenutzten Leitungen für die Kommunikation des eFPGAs mit anderen Komponenten auf einem Chip vorgesehen werden, tatsächlich wird nur eine kleine Anzahl der vorhandenen Anschlüsse benutzt werden. Je nach Verwendung des eFPGAs ist es z.B. sinnvoll, nur Leitungen des globalen Verbindungsnetzes nach außen zu verbinden. Die an den Rändern nicht benutzten Anschlüsse (z.B. lokale Verbindungen) werden auf einen definierten Signalpegel (z.B. GND) geschaltet.

4.3 Automatisierte eFPGA-Implementierung

Im Rahmen der vorliegenden Arbeit wurde eine Entwurfsumgebung erarbeitet, mit der auf Basis einer generischen Architekturbeschreibung und einigen handoptimierten Basiszellen eFPGA-Makros implementiert werden können. Dieser Layout-Generator basiert im Wesentlichen auf dem in [70] vorgestellten so genannten Datenpfadgenerator (DPG). Dieses am Lehrstuhl für Allgemeine Elektrotechnik und Datenverarbeitungssysteme der RWTH Aachen entwickelte Werkzeug wird ursprünglich für den Aufbau hochoptimierter Datenpfade als VLSI-Makros verwendet. Dazu wird der Signalflussgraph des Datenpfades als parametrisierbarer, struktureller VHDL-Code beschrieben. Dadurch können z.B. Wortbreiten oder die Anzahl der Taps bei einem Filter-Makro durch einen Parameter definiert werden. Der Datenpfadgenerator verwendet handoptimierte Layouts häufig wiederkehrender Basiszellen (so genannte Leaf-Zellen) und platziert diese gemäß den Vorgaben im Signalflussgraphen. Anschließend wird ein modifizierter Lee-Routing-Algorithmus benutzt, um die entsprechenden Netze zu verbinden. Der Routing-Algorithmus ist besonders für lokale Verbindungen zwischen benachbarten Zellen optimiert, die in Datenpfaden häufig vorkommen. Die Vorgabe der Platzierung durch den Anwender führt in der Regel zu kompakteren Layouts als ein automatischer Platzierungs-Algorithmus, da bei Datenpfaden die Anordnung der Leaf-Zellen einem regulären Schema von Function-Slices und Bit-Slices folgt (siehe Abbildung 4.3).

Dieser Ansatz ist auch bei der Implementierung von eFPGA-Makros besonders geeignet, da diese ebenfalls inhärent über eine große Regularität verfügen. Anders als bei GILES wird bei dem hier beschriebenen Ansatz beim Generieren der Layouts eine gröbere Granularität der handoptimierten Basiszellen verwendet. Insbesondere werden die Logikelemente als eigene Leaf-Zellen bereitgestellt. Die Entwurfskomplexität ist dadurch geringfügig höher als bei GILES, da Leaf-Zellen mit einer Komplexität von über hundert Transistoren von Hand erstellt werden müssen. Der Vorteil dieser Methode liegt in der höheren Layout-Dichte, da keine Verschnittflächen durch suboptimale Platzierungsalgorithmen auftreten können. Dadurch ist es möglich, eFPGAs auf Basis einer generischen Architektur mit bislang nicht erreichter Packungsdichte zu implementieren. Zudem ist es durch den hier verwendeten Ansatz erstmals möglich, auch Routing-Switches mit frei definierter Konnektivität aufzubauen, während existierende Ansätze bestenfalls die Auswahl zwischen wenigen Varianten ermöglichen.

Neben den Logikelementen werden verschiedene Switch-Points zum Aufbau des Routing-Switch, Multiplexer für die dedizierten Routing-Blöcke, Tristate-Buffer für die Connection-Box und SRAM-Zellen benötigt. Abbildung 4.22 illustriert die Funktionsweise des Layout-Generators beim Aufbau eines eFPGA-Makros für ein einzelnes Cluster.

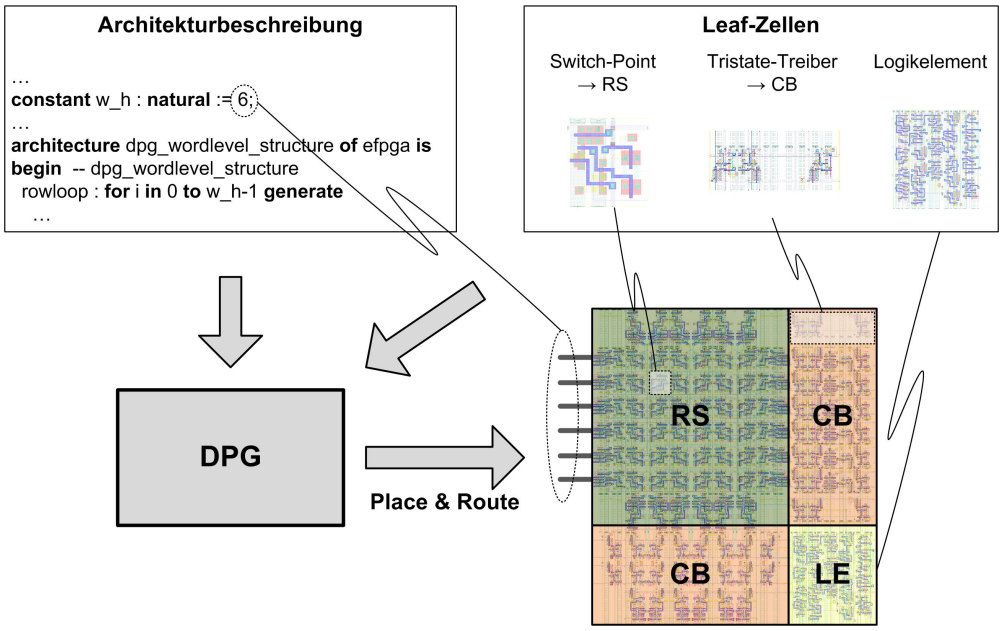


Abbildung 4.22: Aufbau eines eFPGA-Clusters mit dem Layout-Generator

Im dargestellten Beispiel ist die Anzahl der verfügbaren Verbindungsleitungen als Parameter dargestellt. Das gesamte eFPGA-Makro lässt sich durch regelmäßiges Aneinanderfügen des Clusters in Zeilen und Spalten erzeugen. Die Portierung eines so implementierten Makros in eine neue Halbleitertechnologie ist mit geringem Aufwand möglich, da lediglich die einzelnen Leaf-Zellen neu entworfen werden müssen.

4.4 Konfiguration und Simulation

Aus den mit dem Layout-Generator erzeugten VLSI-Layouts werden Netzlisten auf Transistorebene extrahiert. Auf Basis dieser Netzlisten können Simulationen durchgeführt werden, um das Laufzeitverhalten und die Verlustleistungsaufnahme des eFPGA-Makros zu ermitteln. Dazu müssen zunächst die Konfigurationsinformationen in die SRAM-Zellen geschrieben werden. Wie in Kapitel 4.1 dargestellt wurde, gibt es zurzeit keine Werkzeuge, die eine generische Architektur wie die hier vorgestellte unterstützen. Die Entwicklung eines vollautomatischen Mappers und Konfigurators ist jedoch zu aufwändig, um sie im Rahmen der hier vorgestellten Arbeit durchführen zu können. Daher wird die Allokation der Logikelemente und Verbindungsressourcen für die betrachteten Datenpfade manuell durchgeführt. Aufgrund der hohen Regularität der Datenpfade ist die Komplexität dieses Vorgangs deutlich geringer als bei kontrollorientierten Anwendungen. Auf Basis der manuellen Allokation müssen dann die Konfigurationsdaten für das gesamte Makro erzeugt werden. Je nach Größe und Komplexität des eFPGAs sind dafür weit über zehntausend Bits nötig. Um die Erzeugung dieser Konfigurationsbits auch für komplexere Datenpfade zu ermöglichen, wurde ein Konfigurationswerkzeug entwickelt, das auf Basis des in Kapitel 4.2 beschriebenen Architektur-Templates Konfigurationsdatenströme für vorgegebene Netzlisten

erzeugt. Insbesondere wurde für die Routing-Switches eine geschlossene Entwurfsumgebung erarbeitet, welche neben der automatischen Layout-Erzeugung auch das Erstellen einer VHDL-Verhaltensbeschreibung und die automatische Konfiguration erlaubt. Dazu werden die Konfigurationsdaten der Switch-Points für alle möglichen Fälle in einer Tabelle gespeichert und anhand einer vereinfachten funktionalen Netzliste mit den zu konfigurierenden Verbindungswegen für den gesamten Routing-Switch ausgelesen und zusammengesetzt. Auf Basis der vom Layout-Generator verfügbaren Platzierungsinformationen für die Switch-Points und der zugehörigen SRAM-Zellen können dann die Konfigurationsbits automatisch in die entsprechenden Speicher geschrieben werden. Abbildung 4.23 zeigt einen Überblick über den gesamten Entwurfsablauf für die generischen Routing-Switches gemäß dem in Kapitel 4.2 diskutierten Architektur-Template.

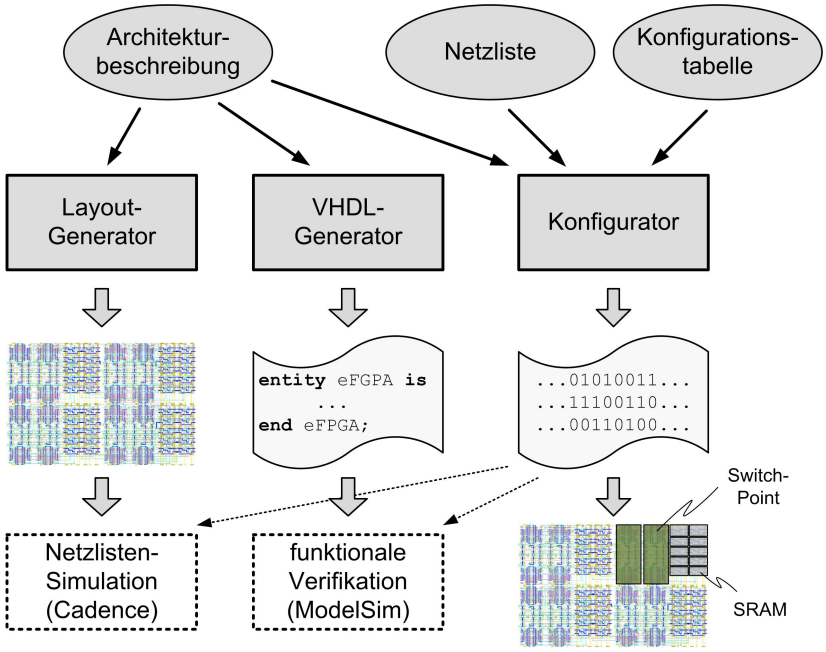


Abbildung 4.23: Geschlossener Entwurfsablauf für den generischen Routing-Switch

Das Konfigurationswerkzeug für den Routing-Switch unterstützt alle wesentlichen Parameter wie verschiedene Typen von Switch-Points, shared-SRAMs und beliebige Positionierung der Switch-Points im Routing-Switch. Es wurde insbesondere im Rahmen der in Kapitel 5.7 vorgestellten Modellierung verwendet, um systematisch verschiedene Routing-Switches aufzubauen und für vorgegebene Verbindungswege zu konfigurieren. Als Grundlage für den Entwurfsablauf dient dabei die MATLAB-basierte Architekturbeschreibung des Templates, so dass bei Änderungen an der Architektur sämtliche Schritte vom Erzeugen des Layouts bis zur Konfiguration geschlossen durchgeführt werden können. Im Folgenden ist ein entsprechender Auszug aus der Architekturbeschreibung eines Routing-Switch dargestellt:

```
% Anzahl der Verbindungsleitungen
rs.trk_h = 32; % horizontal
rs.trk_v = 32; % vertikal

% Switch-Point 1: Position und Verbindungswege
rs.sp(1).x = 1; % horizontale Position
rs.sp(1).y = 1; % vertikale Position
rs.sp(1).dir = {'n-s', 'e-w', ...};

% Switch-Point 2 Position und Verbindungswege
rs.sp(2).x = 2; % horizontale Position
rs.sp(2).y = 2; % vertikale Position
rs.sp(2).dir = {'n-s', 'e-w', ...};

... % etc.
```

Auf Basis der Angaben zu den konfigurierbaren Verbindungswegen wird in einer Zwischendarstellung der Architekturbeschreibung eine Zuordnung der Switch-Points zu den verfügbaren Leaf-Zellen durchgeführt. Über den entsprechenden Bezeichner der Leaf-Zelle wird der Switch-Point dann in den Konfigurationstabellen identifiziert. Die Netzliste besteht für den Routing-Switch aus einer textuellen Aufzählung der Switch-Points und den zu konfigurierenden Verbindungswegen nach folgendem Schema:

```
sp1: n-s      % Verbindung von Norden nach Süden
sp2: n-s,w-e  % Verbindung von Norden nach Süden und von Westen nach Osten
...           % etc.
```

Ein Parser liest die Netzliste und die Zwischenform der Architekturbeschreibung ein und setzt auf Basis der Layout-Informationen den Bitstrom gemäß der Anordnung der SRAM-Zellen im Layout automatisch zusammen. Abbildung 4.24 verdeutlicht die Arbeitsweise des Konfigurators. Der dargestellte Auszug der Architekturbeschreibung definiert einen Routing-Switch mit verschiedenen Typen von Switch-Points (als lc_s4 und lc_s6 bezeichnet). Eine detaillierte Beschreibung des Entwurfsablaufs für den generischen Routing-Switch wird in [74] gegeben.

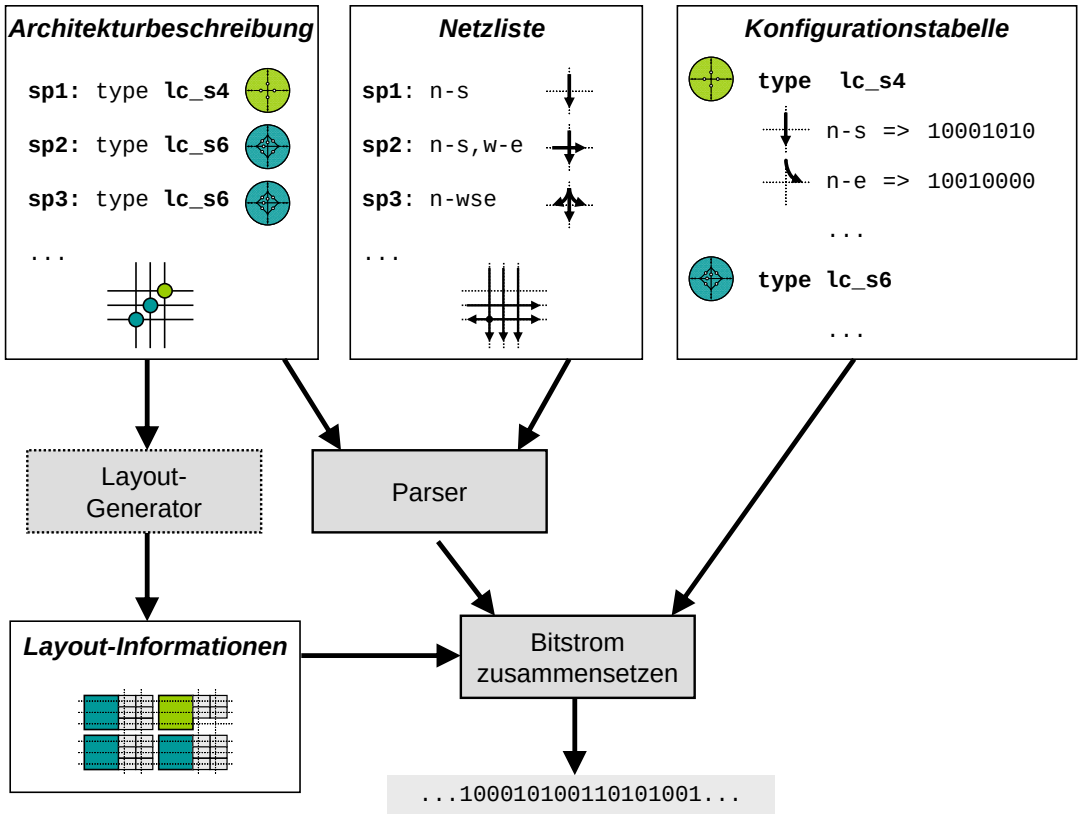


Abbildung 4.24: Zusammensetzen des Konfigurations-Bitstroms

Für ein Teilproblem des Mappings, das Routing von Signalen im globalen Island-Style-Interconnect, existiert inzwischen eine Methode, die auf Basis kommerzieller VLSI-Routing-Tools arbeitet und für das hier beschriebene Architektur-Template entwickelt wurde [17][54].

4.5 Zusammenfassung

Das hier vorgestellte Architektur-Template bildet eine flexible Grundlage für die Modellierung und Implementierung arithmetikorientierter eFPGA-Makros. Es berücksichtigt dabei im Gegensatz zu existierenden Ansätzen konsequent die Anforderungen von Arithmetikdatenpfaden. Das gilt insbesondere für das Verbindungsnetz, welches auf das Verbindungsprofil dieser Datenpfade, also hohe Lokalität zwischen benachbarten Logikelementen sowie die Verwendung von Broadcast-Leitungen, abgestimmt ist. Zu den wesentlichen Neuerungen gegenüber existierenden eFPGA-Architekturen gehören die Verwendung zweidimensionaler Cluster, die Broadcast-Leitungen zur effizienten Zuführung von Operanden und die dedizierten Routing-Blöcke. Die Logikelemente selbst sind dabei so aufgebaut, dass sie ein möglichst effizientes Mapping der Bit-Operationen auf die Kernlogik ermöglichen und so die Anforderungen an das Verbindungsnetz reduzieren. Durch geeignete Wahl der Architekturparameter lässt sich das eFPGA systematisch für die Anforderungen einer Anwendungsklasse optimieren. Das Template liegt als MATLAB-basierte Beschreibung vor, in der alle zuvor beschriebenen Parameter definiert sind. Eine automatische Konsistenz-

prüfung stellt sicher, dass die gewählten Parameter plausibel sind. Auf Basis dieser Beschreibung erfolgt eine automatisierte VLSI-Implementierung von Kernkomponenten des eFPGAs. Exemplarische Implementierungen werden in Kapitel 5 beschrieben. Dieselbe Architekturbeschreibung wird benutzt, um die mathematischen Modellgleichungen zur Beschreibung der Implementierungskosten des eFPGAs auszuwerten. Die Modellbildung wird in Kapitel 5.7 diskutiert.

Mit der beschriebenen Methodik ist es erstmals möglich, auf Basis einer sehr frei parametrisierbaren eFPGA-Architektur effiziente VLSI-Implementierungen, elementare Konfigurationswerkzeuge und modellbasierte Kostendaten zu erzeugen, ohne erhebliche Einschränkungen bezüglich der Architektur (z.B. bei VLSI-Implementierung auf Basis einer VPR-konformen Architektur) oder bei den Implementierungskosten (z.B. bei Implementierungen mit Standardzell-Synthese) hinnehmen zu müssen.

5 Implementierung und Modellierung

5.1 Übersicht

Neben der Modellierung von eFPGA-Architekturen stellt die physikalische Implementierung entsprechender eFPGA-Makros in modernen Halbleitertechnologien eine erhebliche Herausforderung dar. Aufgrund der hohen Mehrkosten durch die Konfigurierbarkeit ist es entscheidend, einerseits eine effiziente Architektur zu verwenden und andererseits eine effiziente VLSI-Implementierung unter Berücksichtigung der gewählten Architekturparameter durchzuführen. Grundsätzlich können als Entwurfsverfahren eine automatische Standardzell-Synthese oder physikalisch orientierte Layout-Techniken verwendet werden. Standardzell-Implementierungen erlauben es, auf einfache Weise anhand einer HDL-basierten Beschreibung der eFPGA-Architektur ein VLSI-Layout zu generieren. Allerdings ist die Effizienz solcher Implementierungen gering. Diskrete FPGA-Bausteine, die als Massenprodukt mit hohen Stückzahlen gefertigt werden, werden daher mit physikalisch orientierten Entwurfsmethoden implementiert. Allerdings ist bei diesem Entwurfsstil die Anpassung der Architekturparameter sehr aufwändig, da jeweils ein Redesign des Makros nötig ist.

5.2 Schaltungstechnische Realisierung

Für die schaltungstechnische Realisierung der eFPGA-Makros ist insbesondere der Aufbau der programmierbaren Schalter von entscheidender Bedeutung, da diese eine häufig verwendete Komponente sowohl im Verbindungsnetz als auch in den Logikelementen darstellen. Um die optimale Implementierungsform zu ermitteln, wurde am Beispiel des Decoder-Baums einer Lookup-Table der Einfluss der verwendeten Schaltungstechnik auf die Fläche, Laufzeit und Verlustleistung untersucht. Der Decoder-Baum wurde mit Tiefen von zwei, drei und vier Stufen aufgebaut. Bei den Varianten mit Transmission-Gates wurde zudem untersucht, welchen Einfluss die Verwendung von zusätzlichen Treibern zwischen den passiven Schaltelementen hat. Die Decoder-Bäume wurden in einer 130nm-Technologie implementiert. Abbildung 5.1 zeigt die Ergebnisse der Untersuchung bezüglich der Fläche (A), Laufzeit (T) und Verlustleistung (P) sowie das ATE-Produkt aus Fläche, Laufzeit und Energie. Als Schaltungstechniken wurden Pass-Transistoren (PT), Transmission-Gates (TMG) und C^2MOS-Gatter verwendet. Die Varianten mit zusätzlichen Treibern zwischen den Stufen haben entweder nach jeder (buf 100%) oder nach jeder zweiten (buf 50%) Stufe einen aktiven Treiber. Alle Decoder haben einen Treiber nach der letzten Stufe des Baums bzw. einen zusätzlichen Pull-Up-Transistor zur Wiederherstellung des Pegels bei der Variante mit Pass-Transistoren.

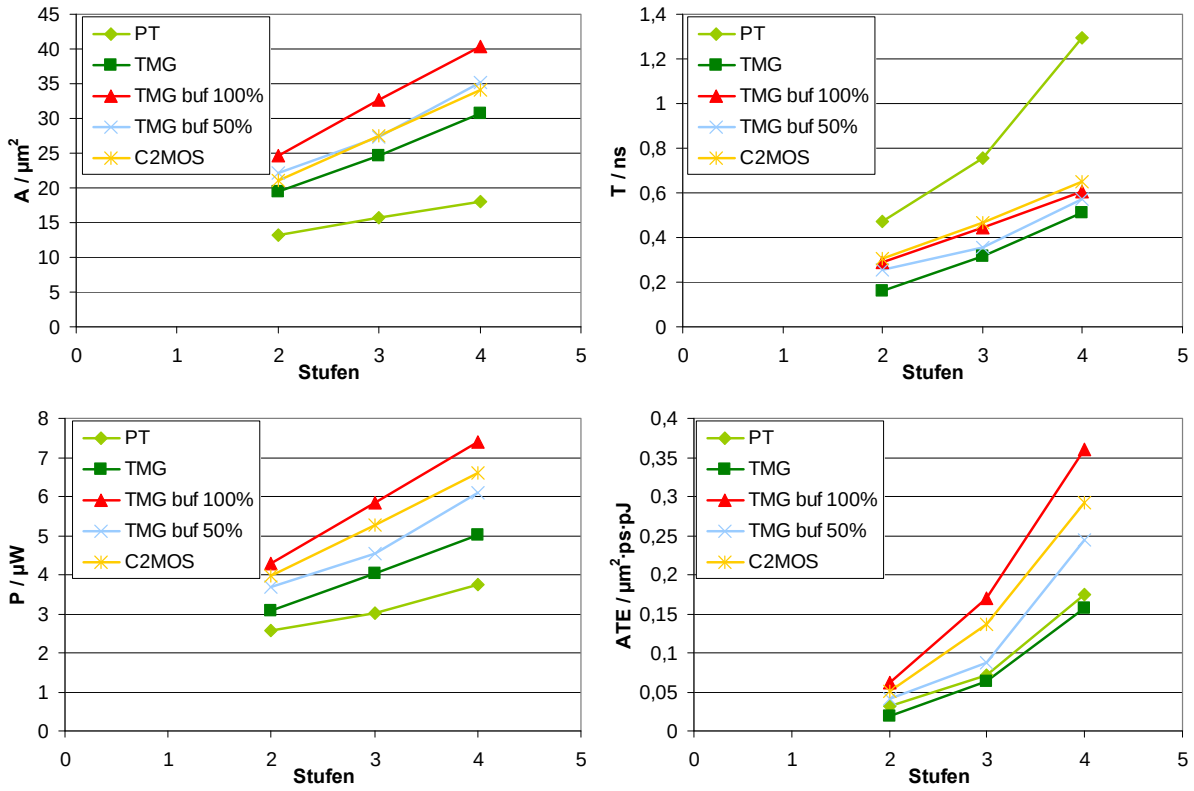


Abbildung 5.1: Implementierungskosten eines LUT-Decoders in einer 130nm-Technologie

Die Ergebnisse zeigen, dass für die verwendete 130nm-Technologie die Verwendung von Transmission-Gates ohne zusätzliche Treiber für alle Baumtiefen die günstigsten Ergebnisse bezüglich des ATE-Produkts liefert. Dabei ist zu berücksichtigen, dass bei Verwendung des ATE-Produkts als Gütekriterium ein Abtausch zwischen den drei verwendeten Größen stattfinden kann. Neben der Betrachtung dieses Wertes müssen daher auch die einzelnen eingehenden Größen genau bewertet werden, um eine einseitige Optimierung zu verhindern (z.B. übermäßige Verkleinerung der Fläche auf Kosten der Laufzeit, so dass die bezüglich des ATE-Produkts günstigste Schaltung schließlich ein für die Zielanwendung ungeeignetes Zeitverhalten aufweist). Die hier untersuchten Varianten des Decoders mit Pass-Transistoren sind z.B. bezüglich der Fläche und Verlustleistung besonders günstig, haben aber deutlich größere Laufzeiten als die anderen Implementierungsalternativen. Für die exemplarisch implementierten eFPGA-Makros wurden in der Regel Transmission-Gates gewählt, trotzdem wurde eine Variante zur Demonstration des Flächenpotenzials teilweise mit Pass-Transistoren implementiert.

5.3 Implementierte Architekturen

Im Folgenden werden drei Beispielmakros, die auf Basis des Architektur-Templates implementiert wurden, sowie eine modellbasierte Untersuchung auf Grundlage implementierter Leaf-Zellen beschrieben. Die Makros sind chronologisch in der

beschriebenen Reihenfolge entstanden und wurden in einer jeweils zu dieser Zeit aktuellen Halbleitertechnologie implementiert. Neben grundsätzlichen Entwurfsentscheidungen (z.B. besonders flächensparende Implementierung bei Beispielarchitektur 2) wurden die Erkenntnisse aus den Auswertungen der vorher implementierten Makros benutzt, um eine sukzessive Verbesserung zu erreichen. Ausgangspunkt ist eine Architektur, die sich bezüglich des Verbindungsnetzes stark an Standard-FPGA-Architekturen anlehnt und die insbesondere bezüglich der Entwurfsflexibilität (Layout-Generierung) und Verwendung neuartiger, arithmetikorientierter Logikelemente optimiert ist. Die Implementierung erfolgte in einer 180nm-Technologie. Die Kombination eines hochgradig für Arithmetikanwendungen optimierten Logikelements mit einem herkömmlichen Verbindungsnetz hat sich dabei als nicht optimal herausgestellt, da die erhöhte Anzahl von LE-Eingängen entsprechende Anforderungen an die lokale Kommunikation stellt. Bei der zweiten Architektur, die in einer 130nm-Technologie implementiert wurde, wurde das Verbindungsnetz daher konsequent an die Anforderungen von Arithmetikdatenpfaden angepasst, während das Logikelement gegenüber der ersten Architektur flexibler gewählt wurde. Zusätzlich wird durch Verwendung von shared-SRAMs und Pass-Transistoren die Flächeneffizienz zum Teil auf Kosten der Signallaufzeiten erheblich gesteigert. Die dritte hier beschriebene Beispielarchitektur ist ein ausgewogener Kompromiss aus den beiden Extremen, der eine hohe Flächen- und Verlustleistungseffizienz erreicht und zudem einen flexiblen Entwurf mit dem Layout-Generator ermöglicht. Bei der abschließenden modellbasierten Betrachtung wurden sämtliche Leaf-Zellen in einer 90nm-Technologie implementiert und das Template mit allen verfügbaren Parametern modelliert.

Sämtliche Simulationen von Laufzeiten und Verlustleistung basieren auf Layout-extrahierten Netzlisten und wurden, sofern nicht anders beschrieben, unter Worst-Case-Bedingungen durchgeführt.

5.4 Architektur 1: Parametrisierbare Standardarchitektur

5.4.1 Architektur

Als Referenz wurde eine Standardarchitektur mit hierarchischem Verbindungsnetz gewählt. Statt eines in herkömmlichen FPGAs meist verwendeten LUT-basierten Logikelements wird bei der Beispielarchitektur 1 ein spezielles gatterbasiertes LE mit verringerter Flexibilität verwendet. Dieses LE ist so entworfen, dass es sich für eine möglichst effiziente Abbildung von Addierern und Multiplizierern eignet. Dabei wird auch Carry-Save-Arithmetik unterstützt. Neben den drei Konfigurationsbits in der Kernlogik werden auch die Eingänge des Logikelements zur Programmierung der gewünschten Funktionalität benutzt. Dabei wird ausgenutzt, dass sich Signale innerhalb der Connection-Box auf definierte Pegel (VDD, GND) schalten lassen. Abbildung 5.2 zeigt den Aufbau der Kernlogik.

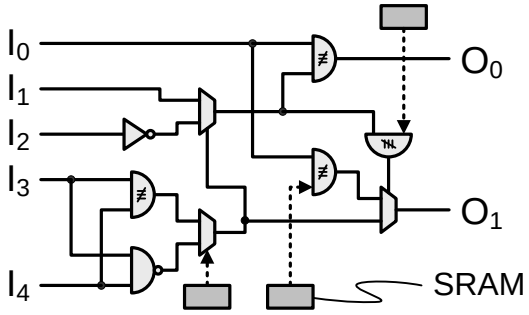


Abbildung 5.2: Logikelement Beispielarchitektur 1 (Kernlogik)

Die Kernlogik hat $I_{LE}=5$ Eingänge und $O_{LE}=2$ Ausgänge. Die Liste der implementierbaren Funktionen F_{LE} umfasst neben den gängigen booleschen Verknüpfungen von zwei Eingängen (OR, NOR, AND, NAND, XOR, XNOR) auch einen Multiplexer mit zwei Eingängen und eine Gated-Full-Addition für den Aufbau von Array-Multiplizierern. Jedes Logikelement verfügt über ein Register pro Ausgang, so dass $N_{Reg}=1$ und $O_{Reg}=\{O_0,O_1\}$ gilt.

Die Logikelemente werden zunächst zu Subclustern mit vier Elementen zusammengefasst. Die Architektur hat keine Broadcast-Leitungen, stattdessen haben alle Logikelemente eines Clusters direkten Zugriff auf die Connection-Box. Innerhalb des Clusters ist eine Carry-Chain implementiert. Die so zusammengefassten Logikelemente werden dann innerhalb einer eigenen Hierarchiestufe zu horizontalen Clustern ($C_V=1$) mit einstellbarer Größe C_H zusammengefasst. Innerhalb dieser Hierarchiestufe entspricht also die Connection-Box, welche vier Logikelemente zu einem Subcluster zusammenfasst, dem dedizierten Routing-Block. Abbildung 5.3 zeigt den Aufbau der Architektur.

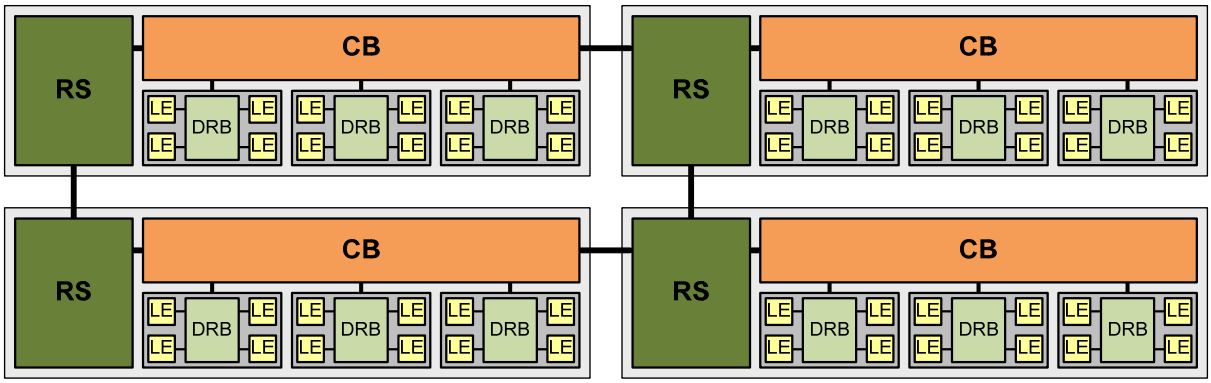


Abbildung 5.3: Architektur 1 (Standardarchitektur mit hierarchischem Verbindungsnetz)

Der DRB verbindet zwölf Leitungen aus der Connection-Box mit den jeweils vier Logikelementen eines Subclusters. Die acht Ausgangssignale der Logikelemente sind wieder als Eingangssignale für das Subcluster verwendbar. Die Connection-Box auf höchster Hierarchieebene hat stets die maximale Konnektivität, es gilt also $r_{FC,H}=W_H$. Es existieren keine periodisch angeschlossenen oder nicht angeschlossenen Verbindungskanäle.

Die Anzahl der zur Verfügung stehenden Leitungen im Verbindungsnetz W_H und W_V kann frei gewählt werden, ebenso kann die Segmentierung $L_{H,i}$ und $L_{V,j}$ als Parameter eingestellt

werden. Es werden zwei verschiedene Switch-Points verwendet. Für Verbindungen innerhalb einer Hierarchieebene wird ein Switch-Point mit $F_S=3$ verwendet, wie er auch in Abbildung 2.8 gezeigt ist. Zusätzlich werden so genannte interhierarchische Switch-Points mit $F_S=5$ verwendet. Bei diesen werden die Anschlüsse der Connection-Box jeweils an die beiden zusätzlichen Ports des Switch-Points angeschlossen.

5.4.2 VLSI-Implementierung

Die VLSI-Implementierung des eFPGAs erfolgt nach dem in Kapitel 4.3 dargestellten Verfahren in einer 180nm-Technologie mit insgesamt fünf verschiedenen Leaf-Zellen. Die Simulationen erfolgten bei dieser Implementierung unter Nominalbedingungen (Typical-Corner).

Als komplexeste Leaf-Zelle wird das Subcluster mit vier Logikelementen und DRB verwendet. Der Aufbau des Logikelements ist in Abbildung 5.2 dargestellt, der dedizierte Routing-Block besteht aus mehreren konfigurierbaren Schaltern, die als Transmission-Gates aufgebaut sind. Abbildung 5.4 verdeutlicht den Aufbau des DRB und zeigt die entsprechende Leaf-Zelle.

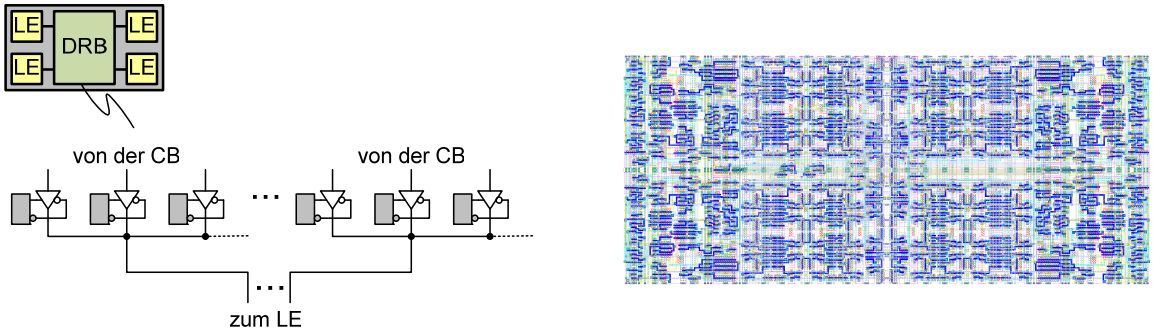


Abbildung 5.4: Subcluster mit vier Logikelementen und DRB (Leaf-Zelle)

Für den Aufbau des Verbindungsnetzes werden ein regulärer und ein interhierarchischer Switch-Point benötigt. Abbildung 5.5 zeigt den Switch-Point mit $F_S=3$. Er besteht aus sechs konfigurierbaren Transmission-Gates. An den vier Ein-/Ausgängen sind bidirektionale Treiber angebracht, so dass alle Signale im globalen Verbindungsnetz vollständig aktiv übertragen werden.

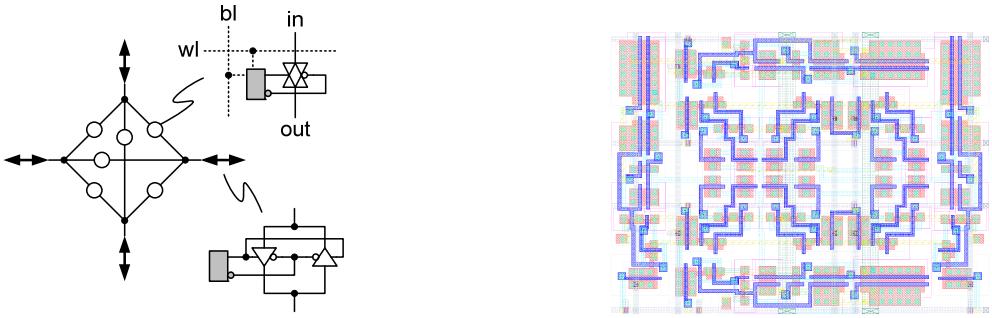


Abbildung 5.5: Switch-Point mit F_S=3 (Leaf-Zelle)

In Abbildung 5.6 ist der Aufbau und die Leaf-Zelle des interhierarchischen Switch-Points mit F_S=5 dargestellt. Der Switch-Point ist ebenfalls vollständig aktiv aufgebaut. Durch die deutlich größere Anzahl an konfigurierbaren Schaltern ist die Zelle ungefähr 90% größer als der einfachere Switch-Point mit F_S=3.

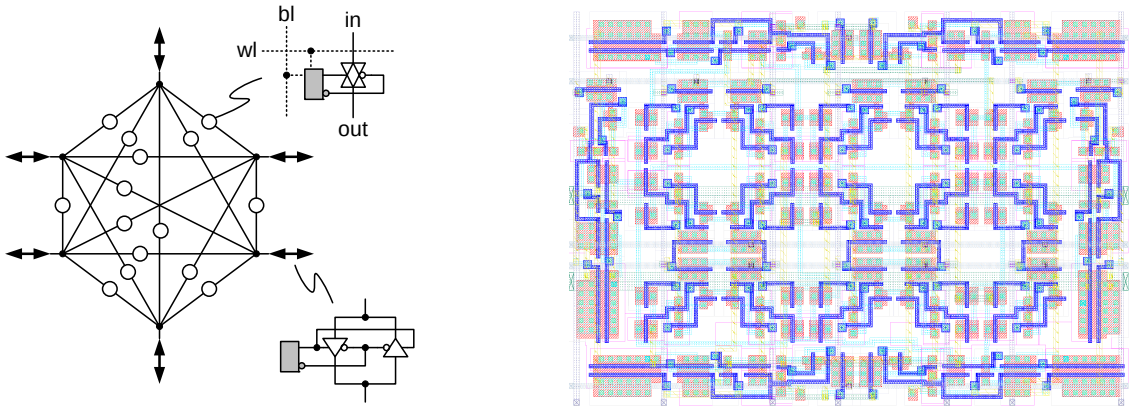


Abbildung 5.6: Interhierarchischer Switch-Point mit F_S=5 (Leaf-Zelle)

Für den Aufbau der Connection-Box werden zwei weitere Leaf-Zellen benötigt. Die Ausgangsstufe besteht aus acht konfigurierbaren Tristate-Treibern, die Signale aus dem Cluster auf das Verbindungsnetz schreiben. Abbildung 5.7 zeigt den entsprechenden Aufbau der Leaf-Zelle.

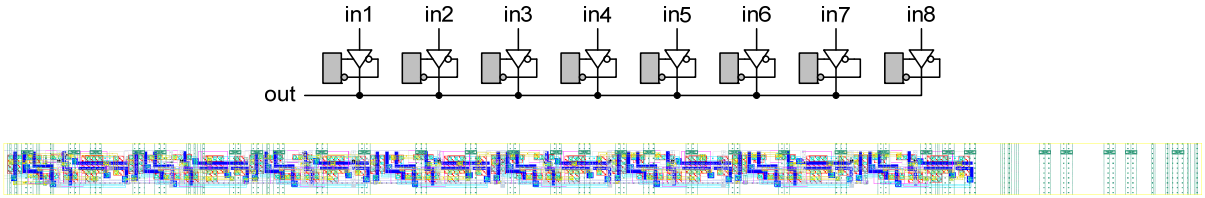


Abbildung 5.7: Ausgangsstufe der Connection-Box (Leaf-Zelle)

Für die Eingänge der Connection-Box werden acht passive konfigurierbare Schalter in einer Leaf-Zelle zusammengefasst. Der Aufbau ist in Abbildung 5.8 dargestellt.

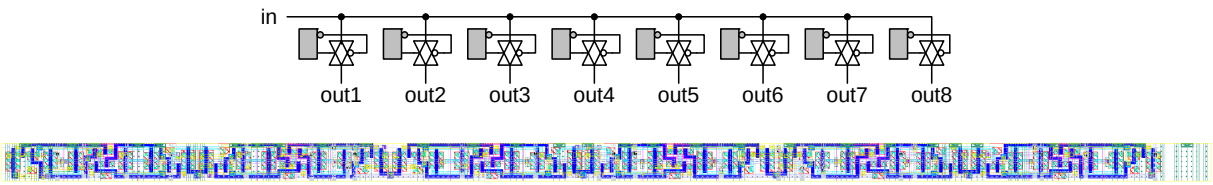
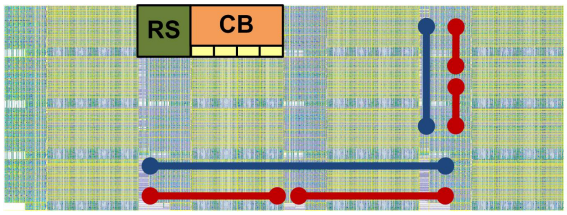
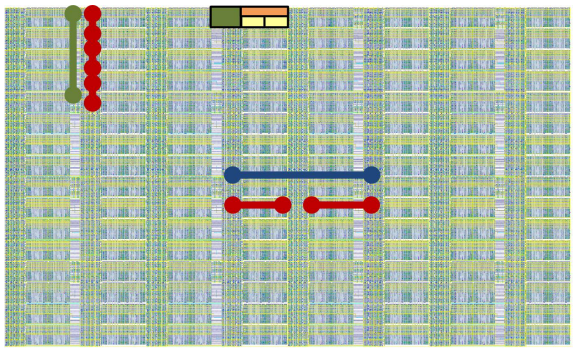


Abbildung 5.8: Eingangsstufe der Connection-Box (Leaf-Zelle)

Auf Basis der fünf beschriebenen Leaf-Zellen wird das komplette Makro des eFPGAs aufgebaut. Abbildung 5.9 zeigt zwei exemplarische Implementierungen mit unterschiedlichen Architekturparametern. Diese sind durch die entsprechende Kennzeichnung im Layout verdeutlicht. Insbesondere wurden verschiedene Cluster-Größen sowie verschiedene Segmentierungsschemata verwendet. Die Länge der Segmente wird dabei durch die im Layout eingezeichneten Balken illustriert.





Variante 1
$S_H=64$, $S_V=4$ (256 Logikelemente)
$C_H=4$ (4×4 LEs pro Cluster)
$W_H=W_V=66$ (Anzahl Verbindungsleitungen)
$L_{H,1..52}=L_{V,1..52}=1$ (52 Leitungen $L=1$)
$L_{H,53..66}=L_{V,53..66}=2$ (14 Leitungen $L=2$)

Variante 2
$S_H=64$, $S_V=16$ (1024 Logikelemente)
$C_H=2$ (2×4 LEs pro Cluster)
$W_H=W_V=18$ (Anzahl Verbindungsleitungen)
$L_{H,1..12}=L_{V,1..12}=1$ (12 Leitungen $L=1$)
$L_{H,13..18}=2$ (6 Leitungen $L=2$ horizontal)
$L_{V,13..18}=4$ (6 Leitungen $L=4$ vertikal)

Abbildung 5.9: Exemplarische VLSI-Implementierungen Architektur 1

Ein wesentliches Kriterium zur Bewertung der Flächeneffizienz eines eFPGAs ist die LE-Dichte, also die Anzahl der Logikelemente pro mm^2. Sofern beim Mapping eines Datenpfades auf verschiedene eFPGAs die Anzahl der benötigten Logikelemente vergleichbar ist, kann die LE-Dichte direkt als Maß der Flächeneffizienz verwendet werden. Das setzt in der Regel voraus, dass die Logikelemente in den betrachteten Architekturen von vergleichbarer Komplexität sind. Die LE-Dichte eines Altera APEX 20KE, der in dieser Arbeit stellenweise als Referenz verwendet wird, beträgt ungefähr 102 LEs/mm^2. Die beiden implementierten Varianten des eFPGA mit hierarchischem Verbindungsnetz haben LE-Dichten von 127 LEs/mm^2 und 280 LEs/mm^2. Die größere Dichte bei der zweiten Implementierungsvariante resultiert aus der deutlich verringerten Anzahl an Verbindungsleitungen, so dass die Routing-Switches und Connection-Boxes im Verhältnis kleiner ausfallen. Abbildung 5.10 zeigt einen Vergleich der LE-Dichten verschiedener FPGAs bzw. eFPGAs. Alle Werte sind

gemäß Anhang E auf eine 180nm-Technologie skaliert. Als Vergleichswerte sind zusätzlich die LE-Dichten für den M2000 FlexEOS (siehe Kapitel 3.4.4) und Chimaera (siehe Kapitel 3.4.2) angegeben. Das Diagramm verdeutlicht, wie durch Wahl der Architekturparameter die Flächeneffizienz beeinflusst werden kann. Die Anzahl der zur Verfügung stehenden Verbindungsleitungen sollte dabei stets so gewählt werden, dass sie den Anforderungen der Anwendungsklasse genügt. Die hier vorgestellten Implementierungen erreichen eine moderate Flächeneffizienz. Eine besonders hohe Flächendichte wird bei dem M2000 FlexEOS erreicht. Aufgrund fehlender Informationen seitens des Herstellers ist eine Bewertung der hohen Flächendichte nur schwer möglich, vermutlich verfügt jedoch das Verbindungsnetz nur über relativ wenige Verbindungsleitungen.

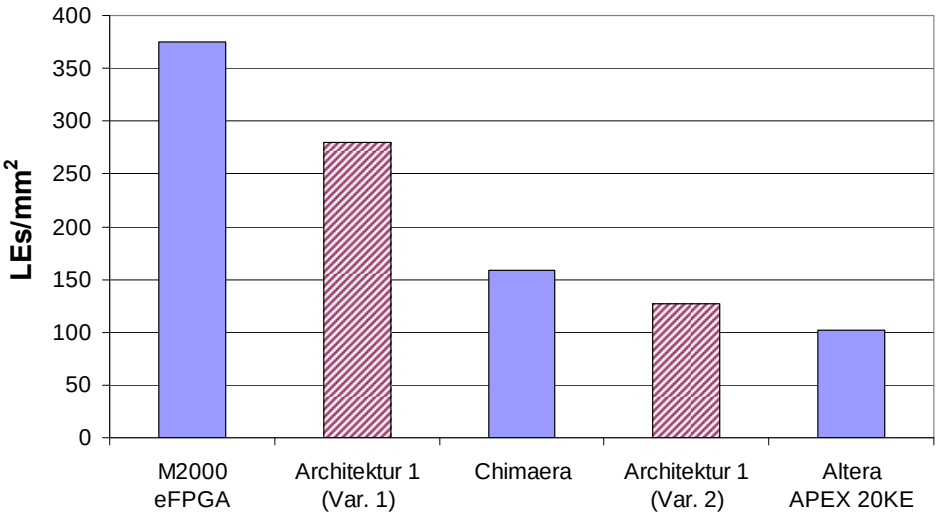


Abbildung 5.10: LE-Dichten verschiedener (e)FPGAs

5.4.3 Einfluss der Flexibilität der Connection-Box auf die Effizienz

Am Beispiel der Connection-Box wurde für die beiden implementierten eFPGA-Makros untersucht, welchen Einfluss die Flexibilität des Verbindungsnetzes auf die LE-Dichte hat. Dazu wurde ausgehend von einer Connection-Box mit voller Flexibilität ($F_C=W$) schrittweise die Anzahl der programmierbaren Schalter reduziert, bis die Connection-Box nur noch zur Abbildung von Multiplizierer- und Addierer-Basiszellen auf das Logikelement geeignet ist. Dieses Experiment verdeutlicht die Dynamik, die alleine durch Optimierung der Connection-Box besteht. Abbildung 5.11 zeigt die entsprechenden Ergebnisse. Für die beiden gewählten Architekturvarianten beträgt die Dynamik der LE-Dichte 71% bzw. 30% zwischen geringster und höchster Flexibilität.

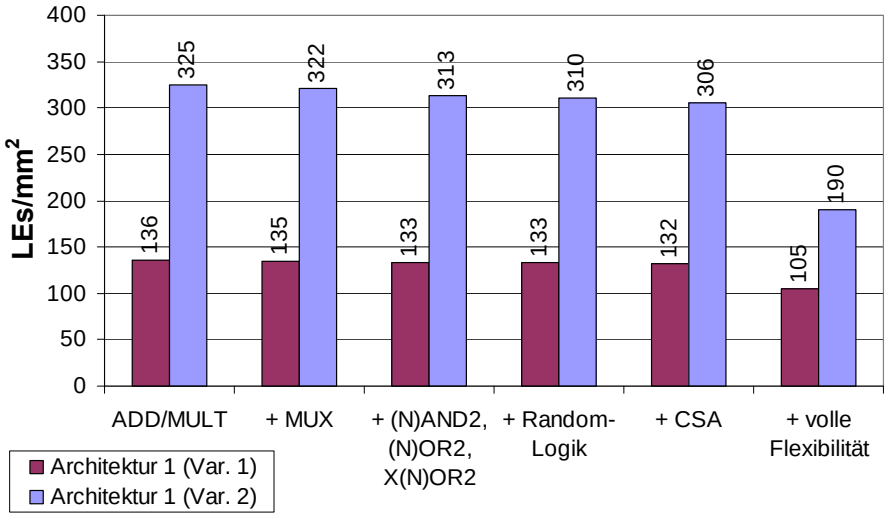


Abbildung 5.11: LE-Dichte in Abhängigkeit von der Flexibilität der Connection-Box

Um die Effizienz der implementierten Makros im Entwurfsraum bewerten zu können, wurde ein paralleles 8-Bit-Radix-2-ACS-Element gemäß Kapitel 3.6 auf das eFPGA abgebildet. Es wurde dabei die Connection-Box mit voller Flexibilität verwendet. Abbildung 5.12 zeigt den AT-Raum unter Berücksichtigung der Werte für die beiden hier diskutierten Makros bei Skalierung aller Werte auf eine 130nm-Technologie. Die Laufzeiten sind durch das hierarchische Verbindungsnetz und die kurzen Carry-Chains länger als bei dem kommerziellen APEX-Baustein. Der Flächenbedarf ist für beide Varianten geringer als beim APEX. Insgesamt ergibt sich für beide Architekturvarianten eine AT-Effizienz, die in einem mit der kommerziellen Architektur vergleichbaren Bereich liegt. Durch Variation der Architekturparameter lässt sich jedoch für die dargestellten Beispiele ein Abtausch zwischen Fläche und Laufzeit von ca. jeweils einem Faktor 2 erreichen. Der Vergleich zeigt, dass die flexible Architekturbeschreibung durch geeignete Wahl der Parameter einen Abtausch zwischen Flächenbedarf und Laufzeit ermöglicht.

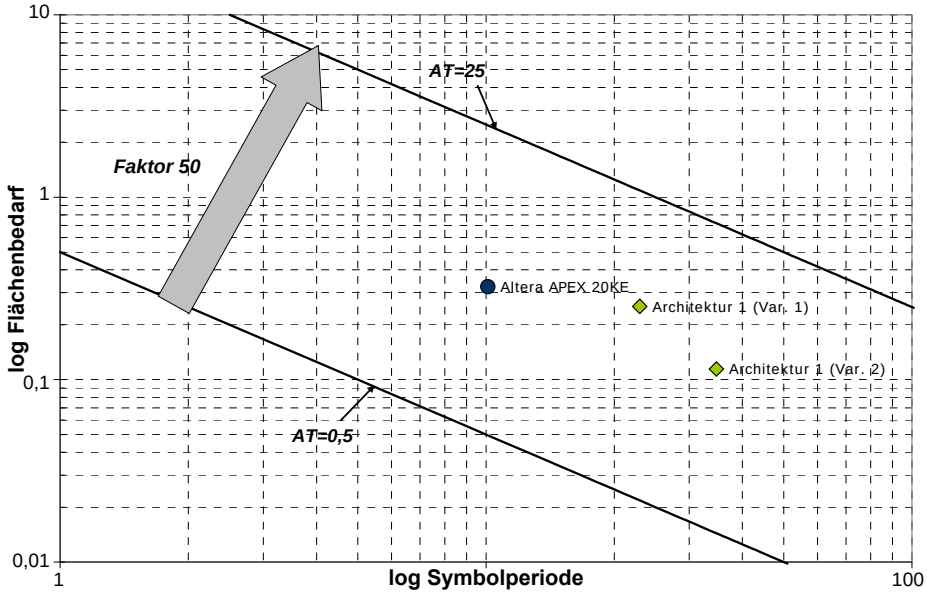


Abbildung 5.12: AT-Kosten ACS-Element für Architektur 1

Ein grundsätzlicher Nachteil der hier beschriebenen Architektur ist der hohe Anteil des Verbindungsnetzes an der Gesamtfläche. Obwohl durch Wahl der Parameter der Flächenanteil der Logikelemente beträchtlich erhöht werden kann, ist das Verbindungsnetz bei der Abbildung von Arithmetikdatenpfaden suboptimal. Die hierarchische Architektur erlaubt es zwar, längere Verbindungen effizient abzubilden, allerdings sind abgesehen von der dedizierten Carry-Chain innerhalb der Cluster lokale Verbindungen zwischen benachbarten Logikelementen über Cluster-Grenzen hinweg teuer. Zudem ist der Flächenanteil der SRAMs bei dieser Architektur relativ groß, da keine shared-SRAMs benutzt werden. Damit ergibt sich bezüglich der AT-Kosten keine Verbesserung. Bei der im folgenden Kapitel beschriebenen Architektur wurden daher entsprechende Architekturoptimierungen durchgeführt, die zu einer höheren Effizienz bzgl. der AT-Kosten führen. Die hier vorgestellten Ergebnisse wurden in [49] veröffentlicht.

5.5 Architektur 2: Arithmetikorientierte Architektur mit shared-SRAMs

5.5.1 Architektur

Bei der zweiten implementierten Beispielarchitektur wird das Konzept der Broadcast-Leitungen, zweidimensionalen Cluster und shared-SRAMs angewendet, um das damit verbundene Optimierungspotenzial für Arithmetikdatenpfade zu analysieren. Die Komplexität des Logikelements wurde gegenüber der in Kapitel 5.4 dargestellten Architektur vergrößert, gleichzeitig jedoch die Anzahl der Eingänge verringert. Um dieses zu erreichen, wurde die Kernlogik im Wesentlichen als eine LUT mit einigen arithmetikspezifischen Erweiterungen konzipiert. Ziel dieser Maßnahme ist es, die Anforderungen des Logikelements an das Verbindungsnetz zu reduzieren. Abbildung 5.13 zeigt den Aufbau der Kernlogik. Diese

besteht im Wesentlichen aus zwei Lookup-Tables mit je zwei Eingängen. Die links im Bild dargestellte LUT hat dabei einen aufgedoppelten Decoder-Baum, so dass eine zusätzliche identisch konfigurierte LUT2 entsteht. Durch Zusammenschalten der LUTs lässt sich über einen entsprechenden Multiplexer eine LUT3 realisieren. Zusätzlich verfügt die Kernlogik über eine dedizierte Carry-Logik und die Möglichkeit, eine Gated-Full-Addition abzubilden.

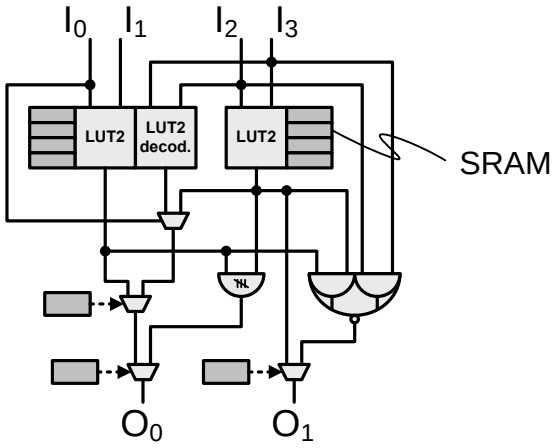


Abbildung 5.13: Logikelement Beispielarchitektur 2 (Kernlogik)

Die Kernlogik hat $I_{LE}=4$ Eingänge und $O_{LE}=2$ Ausgänge. In jeder Zeile von Logikelementen liegt an beiden Ausgängen der Kernlogik ein Register. Es gilt also $N_{Reg}=1$ und $O_{Reg}=\{O_0,O_1\}$. Die Logikelemente sind in Clustern der Größe 4×4 organisiert ($C_H=C_V=4$). Dabei wird jeweils eine komplette Zeile von Logikelementen innerhalb eines Clusters mit identischen SRAMs konfiguriert ($M_{LE}=4$). Abbildung 5.14 verdeutlicht den Aufbau der Gesamtarchitektur.

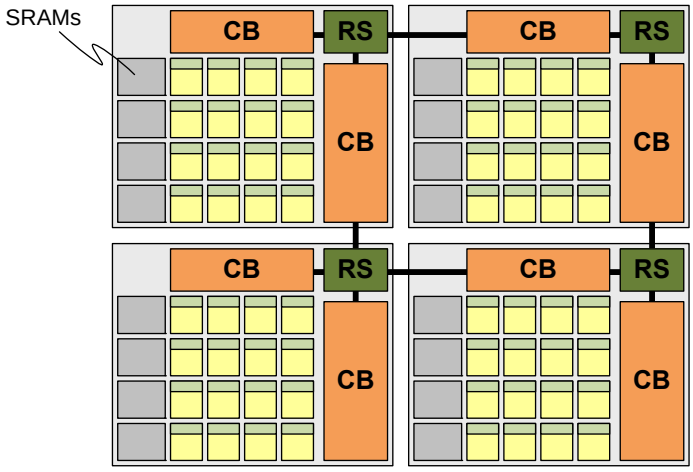


Abbildung 5.14: Architektur 2 (arithmetikorientiert mit shared-SRAMs)

Das globale Verbindungsnetz ist als Island-Style aufgebaut. Es stehen 32 Leitungen in horizontaler und vertikaler Richtung zur Verfügung, die jeweils in Busse mit 4 Bit unterteilt werden ($M_{RS}=M_{CB}=4$). Der Routing-Switch hat eine Disjoint-Architektur, so dass $P_{RS,k}=\{k,k\}|_{0\le k\le 32}$ gilt. Der Aufbau der Switch-Points entspricht dabei dem Switch-Point mit

$F_S=3$ gemäß Abbildung 2.8. Alle Leitungen haben eine Segmentlänge von $L_{H,i}=L_{V,j}=1$. Die Connection-Box verwendet ein periodisches Anschlussschema, bei dem jeweils acht Eingänge und vier Ausgänge an die Routing-Kanäle angeschlossen sind.

Die dedizierten Routing-Blöcke sind so aufgebaut, dass die globalen Broadcast-Leitungen an die beiden Eingänge I_0 und I_1 der Kernlogik geschaltet werden können. Diese Eingänge werden in der Kernlogik verwendet, um bei der Konfiguration als Multiplizierer oder Filter das Partialprodukt in der linken LUT2 zu berechnen. Insgesamt stehen $I_N=2$ Broadcast-Leitungen in vertikaler Richtung und $I_E=1$ Broadcast-Leitung in horizontaler Richtung zur Verfügung. Die lokalen Verbindungen folgen einer Vorzugsrichtung von oben nach unten. Außerdem können sie diagonal nach unten links und unten rechts verschaltet werden. Die Konnektivität der Eingangsmultiplexer ist dabei so klein wie möglich gewählt, jedoch groß genug, um damit u.a. Array-Multiplizierer und Modified-Bitplane-Filter [53] realisieren zu können. Insbesondere besteht eine feste Zuordnung der beiden Ausgänge zu den wählbaren Signalrichtungen, so dass nur die bei den genannten Anwendungen tatsächlich benötigten Verbindungen geschaltet werden können. Zusätzlich zu den regulären lokalen Verbindungen besteht die Möglichkeit, die Broadcast-Leitungen in einem Logikelement zu unterbrechen und mit den Ausgängen der Kernlogik zu beschalten. Dadurch können die Broadcast-Leitungen benutzt werden, um z.B. Signale effizient durch die Kernlogik durchzuschleifen. Außerdem ist es auf diese Weise möglich, die Ausgangsregister der Kernlogik als Verzögerungselemente für die Broadcast-Leitungen zu benutzen. Abbildung 5.15 verdeutlicht den Aufbau des entsprechenden dedizierten Routing-Blocks.

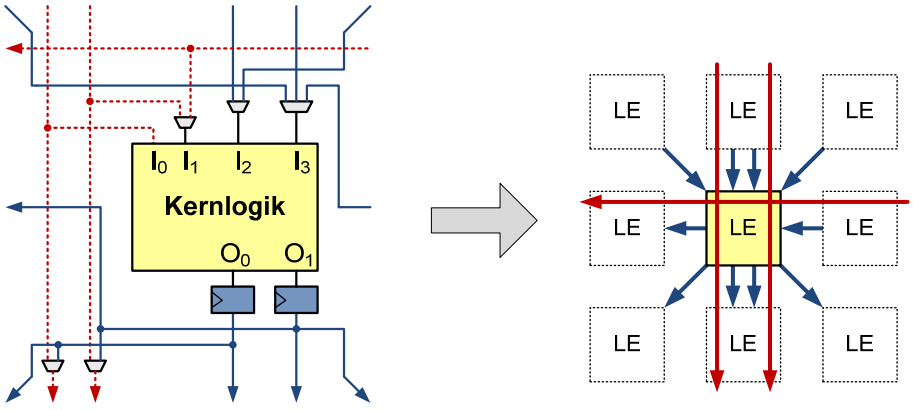
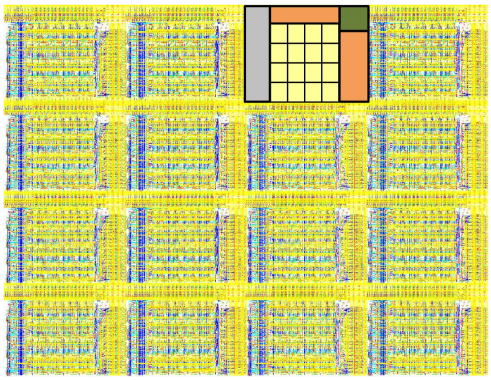


Abbildung 5.15: Lokale Verbindungen über dedizierten Routing-Block

5.5.2 VLSI-Implementierung

Die VLSI-Implementierung des Makros erfolgte in einer 130nm-Technologie durch Kacheln eines kompletten Clusters in horizontaler und vertikaler Richtung. Ziel bei der Implementierung dieser Architektur war es insbesondere, das maximal erreichbare Optimierungspotenzial des Flächenbedarfs durch das Anpassen der kompletten Architektur an eine

arithmetikorientierte Anwendungsklasse zu evaluieren. Neben den Architekturmaßnahmen wurde auch bei der schaltungstechnischen Realisierung besonders auf eine flächeneffiziente Implementierung geachtet. Insbesondere wurden die programmierbaren Verbindungen mit Pass-Transistoren aufgebaut, wenngleich dies zu höheren Signallaufzeiten führt. Außerdem wurden die Connection-Boxes zur Verringerung der Anzahl an Konfigurationsspeichern als baumförmige Multiplexer realisiert. Abbildung 5.16 zeigt das Layout des implementierten eFPGA-Makros mit den wesentlichen Architekturparametern.



Architekturparameter
$S_H=16$, $S_V=16$ (256 Logikelemente)
$C_H=C_V=4$ (4×4 LEs pro Cluster)
$I_N=2$ (vertikale Broadcast-Leitungen)
$I_E=1$ (horizontale Broadcast-Leitung)
$W_H=W_V=32$ (Anzahl Verbindungsleitungen)
$L_{H,1..32}=L_{V,1..32}=1$ (Segmentlänge $L=1$)
$P_{RS,k}=\{k,k\}|_{0\le k\le 32}$ (Disjoint-Routing-Switch)
$M_{LE}=M_{RS}=M_{CB}=4$ (shared-SRAMs)

Abbildung 5.16: VLSI-Implementierung Architektur 2

Die mit dieser Architektur erreichte LE-Dichte beträgt 2712 LEs/mm^2 bzw. bei Skalierung auf eine 180nm-Technologie gemäß Anhang E 1415 LEs/mm^2 und liegt um eine Größenordnung über der von herkömmlichen FPGAs. Berücksichtigt man die geringere Komplexität des Logikelements (LUT3 statt LUT4), so ergibt sich immer noch eine LE-Dichte von 708 LEs/mm^2. Durch die intensive Nutzung von shared-SRAMs und dedizierten Routing-Blöcken statt einer aufwändigen globalen Verbindungsarchitektur konnte der Anteil der Logikelemente gegenüber Standardarchitekturen von typischerweise bis zu 10% auf 40% gesteigert werden.

Zur Bewertung der Architektur wurde erneut das in Kapitel 3.6 beschriebene parallele 8-Bit-Radix-2-ACS-Element auf das eFPGA abgebildet und das Ergebnis im AT-Entwurfsraum dargestellt. Abbildung 5.17 zeigt das AT-Diagramm unter Einbeziehung der hier beschriebenen Architektur mit shared-SRAMs bei Skalierung auf eine 180nm-Technologie. In dem Diagramm ist zu erkennen, dass die AT-Effizienz gegenüber der kommerziellen Standardarchitektur insgesamt deutlich verbessert wurde. Die Symbolperiode ist geringer als bei dem APEX-Baustein, allerdings ist der Flächenbedarf durch die Verwendung der gemeinsam genutzten SRAMs und das für reguläre Datenpfade optimierte Verbindungsnetz gegenüber diesem um nahezu Faktor 10 reduziert.

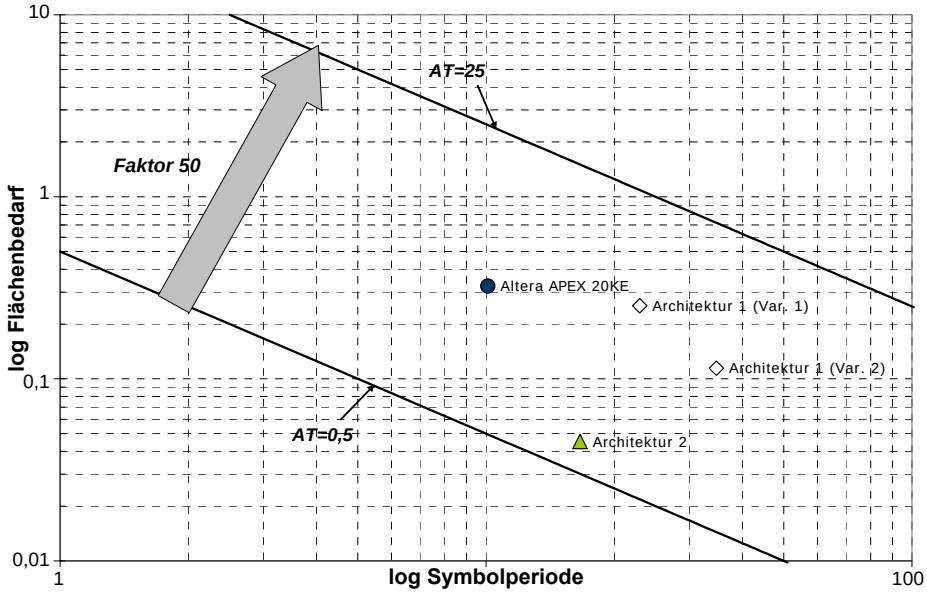


Abbildung 5.17: AT-Kosten ACS-Element für Architektur 2

Neben dem ACS-Element wurde auch ein FIR-Filter mit vier Taps als Modified-Bitplane-Filter auf das eFPGA abgebildet. Die Eingangswortbreite beträgt 8 Bit (vorzeichenlos), die Koeffizienten haben eine Wortbreite von 6 Bit. Für das Filter werden insgesamt 480 Logikelemente benötigt. Eine vergleichbare Implementierung auf einem Altera Cyclone I benötigt 133 LUT4-basierte Logikelemente. Unter Berücksichtigung der LE-Dichten ergibt sich also für das arithmetikorientierte eFPGA ein deutlich geringerer Flächenbedarf. Der Durchsatz des Filters auf dem eFPGA ist gegenüber der Implementierung auf dem Altera Cyclone nur etwa halb so groß. Bezüglich des AT-Produkts ergibt sich damit für das eFPGA eine Verbesserung um 130% gegenüber der kommerziellen Implementierung. Tabelle 5.1 zeigt den Vergleich der Implementierungskosten des Filters für beide Architekturen.

	Cyclone I EP1C20F400C7	**Architektur 2**
Technologie	130nm	130nm
LEs	133	480
LEs/mm^2	196 (skaliert von APEX)	2712
Fläche	0,679mm^2	0,177mm^2
max. Taktfrequenz	104MHz	63MHz
1/AT (MHz/µm^2)	153	356

Tabelle 5.1: AT-Werte für die Implementierung eines 4-Tap-1D-FIR-Filter

Für Datenpfade mit hinreichend hohem Maß an Homogenität ist die hier vorgestellte Architektur wesentlich effizienter als die in Kapitel 5.4 beschriebene Architektur mit hierarchischem Verbindungsnetz und ohne shared-SRAMs. Die Verwendung von dedizierten lokalen Verbindungen und die gleichzeitige Reduktion der Anzahl an globalen Verbindungen resultiert in einer besonders flächeneffizienten Architektur. Bei der Abbildung von

Datenpfaden mit geringer Homogenität z.B. durch gesonderte Vorzeichenbehandlung oder für Anwendungen, die Anteile von Kontrolllogik enthalten, ist durch das relativ unflexible globale Verbindungsnetz und starke Verschnitteffekte eine gegenüber Standardarchitekturen deutlich verringerte Effizienz zu erwarten. Die hier vorgestellten Ergebnisse wurden in [49] veröffentlicht.

5.6 Architektur 3: Parametrisierbare arithmetikorientierte Architektur

5.6.1 Architektur

Die dritte betrachtete Architekturvariante basiert wie die in Kapitel 5.5 diskutierte arithmetikorientierte Architektur mit shared-SRAMs auf zweidimensionalen Clustern, bei denen die Logikelemente über Broadcast-Leitungen an das globale Verbindungsnetz angeschlossen werden. Um auch für Anwendungen mit mäßiger Regularität hinreichend flexibel zu sein, werden keine shared-SRAMs verwendet und die Konnektivität der dedizierten Routing-Blöcke größer gewählt. Das Logikelement ist ein Kompromiss aus den beiden zuvor beschriebenen LE-Architekturen. Es kombiniert zwei LE-Hälften relativ geringer Komplexität zu einem gemeinsamen Block. Dadurch kann die Granularität beim Mapping von Datenpfaden (2 Bit pro Logikelement, z.B. zwei Volladditionen) und Random-Logik mit drei Eingängen (1 Bit pro Logikelement, z.B. ein AND3) so angepasst werden, dass für Datenpfade eine besonders hohe Effizienz erreicht wird, Random-Logik aber trotzdem ohne aufwändiges Kaskadieren von Logikelementen über das Verbindungsnetz möglich ist. Abbildung 5.18 zeigt ein vereinfachtes Blockschaltbild der Kernlogik.

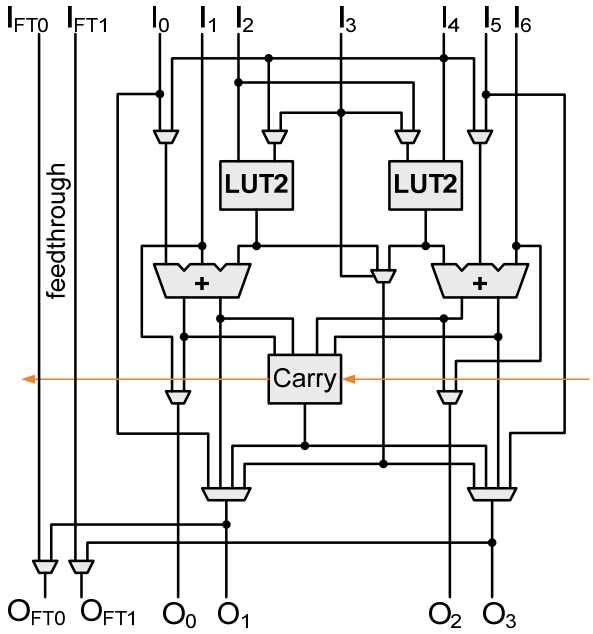


Abbildung 5.18: Logikelement Beispielarchitektur 3 (Kernlogik)

Jede Hälfte der Kernlogik setzt sich aus einem dedizierten Volladdierer mit einer vorgeschalteten LUT2 zusammen. Dadurch können zwei Bit eines Array-Multiplizierers auf das Logikelement abgebildet werden. Durch Kombination der beiden LUT2 über den entsprechenden Multiplexer und den gemeinsamen dritten LUT-Eingang kann eine LUT3 realisiert werden. Durch Verwendung des gemeinsamen Eingangs werden die Anforderungen an das Verbindungsnetz reduziert, da statt acht Eingängen (vier pro Hälfte der Kernlogik) nur sieben (drei pro Hälfte und ein gemeinsamer) angeschlossen werden müssen. Eine dedizierte Carry-Logik erlaubt es, die beiden Volladdierer parallel rechnen zu lassen und so einen Carry-Select-Addierer aufzubauen. Zusätzlich zu den regulären Eingängen der Kernlogik stehen zwei Feedthrough-Leitungen zur Verfügung. Diese können neben den Broadcast-Leitungen benutzt werden, um Signale schnell durch die Kernlogik zu leiten. Insgesamt hat die Kernlogik $I_{LE}=7+2$ Eingänge und $O_{LE}=4+2$ Ausgänge.

Die Logikelemente sind als zweidimensionale Cluster angeordnet, die an ein Island-Style-Verbindungsnetz angeschlossen sind. Anders als bei der in Kapitel 5.5 diskutierten Architektur werden hier auch Feedthrough-Stufen (siehe Kapitel 4.2.2) zum Erzeugen virtueller Cluster verwendet. Abbildung 5.19 zeigt den Aufbau der gesamten Architektur. Die beiden Hälften der Kernlogik werden dabei als einzelne Blöcke angedeutet, die an einen gemeinsamen dedizierten Routing-Block angeschlossen sind.

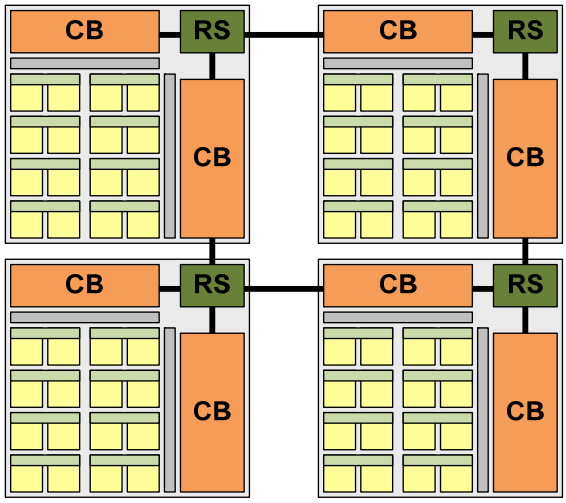


Abbildung 5.19: Architektur 3 (arithmetikorientiert, parametrisierbar)

Die Broadcast-Leitungen laufen von oben nach unten und von rechts nach links über die Logikelemente. Es stehen insgesamt $I_N=2$ vertikale und $I_E=1$ horizontale Leitungen zur Verfügung, so dass sich z.B. Array-Multiplizierer und Bitplane-Filter besonders günstig abbilden lassen. In jeder Zeile von Logikelementen ist an den vier primären Ausgängen der Kernlogik (also ohne die Feedthrough-Leitungen) sowie an den Broadcast-Leitungen ein Register angeschlossen, so dass $N_{Reg}=1$ und $O_{Reg}=\{O_0,O_1,O_2,O_3/BC_0,BC_1\}$. Die Logikelemente sind in Clustern wählbarer Größe $C_H \times C_V$ organisiert. Das globale Verbindungsnetz ist unter Berücksichtigung der hohen Konnektivität auf lokaler Ebene mit $W_H=W_V=4$ sehr

einfach ausgelegt. Wie bei der arithmetikorientierten Architektur aus Kapitel 5.5 wurden alle Segmentlängen der Leitungen zu $L_{H,i}=L_{V,j}=1$ gewählt, um die Komplexität des Verbindungsnetzes klein zu halten. Innerhalb der Connection-Box sind alle Eingänge zum Cluster mit $r_{FC,H/V}=4$ angeschlossen, um trotz der geringen Anzahl an Leitungen eine hinreichend große Flexibilität zu erreichen. Die Ausgänge O_1 und O_3 werden am östlichen, südlichen und westlichen Rand des Clusters an die Connection-Box angeschlossen. Das gewählte Anschlussschema ist besonders günstig für die Abbildung von Datenpfaden mit einem von oben nach unten und von rechts nach links gerichteten Signalfluss. Zur Reduktion der Implementierungskosten für die Connection-Boxes sind die eingangsseitigen Multiplexer baumförmig aufgebaut. Die Anordnung der Switch-Points im Routing-Switch erfolgte wie in [60] beschrieben nach dem Kriterium der minimalen euklidischen Distanz. Durch schaltungstechnische Maßnahmen in den Switch-Points wird bei dem hier verwendeten Routing-Switch eine besonders geringe Verlustleistungsaufnahme erzielt. Dazu können Switch-Points in einen passiven Modus geschaltet werden, in dem statt der aktiven Tristate-Treiber nur passive Transmission-Gates das Signal weiterleiten. Der Aufbau des Routing-Switch und des zugehörigen Switch-Points sind in Abbildung 5.20 dargestellt.

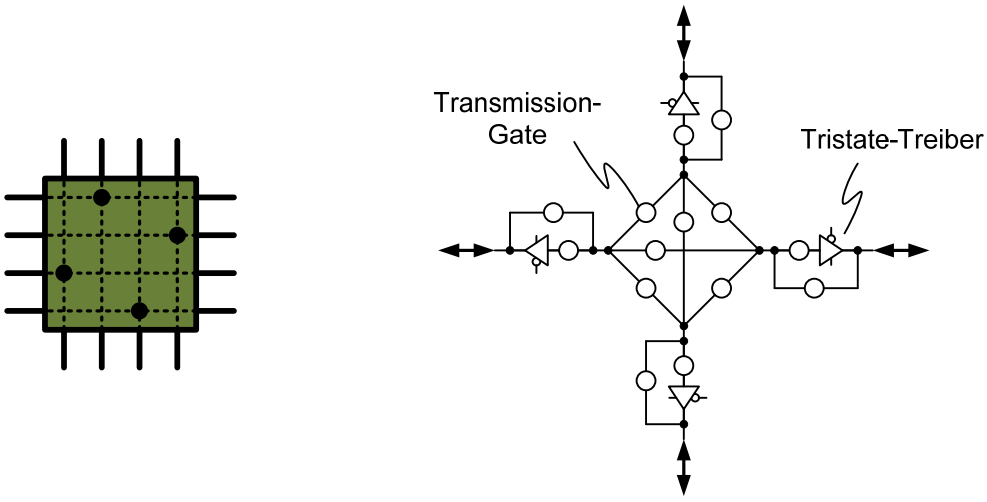


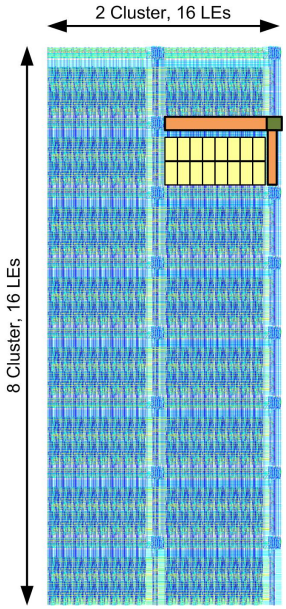
Abbildung 5.20: Routing-Switch und zugehöriger Switch-Point

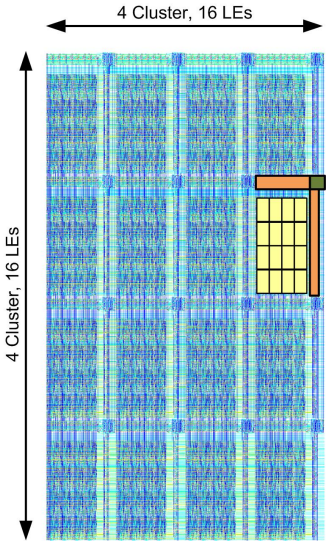
Die dedizierten Routing-Blöcke bestehen aus sieben Multiplexern mit einem komplexen lokalen Verbindungsschema, das für die Anforderungen von Array-Multiplizierern mit und ohne Vorzeichenbearbeitung sowie Bitplane-Filter optimiert ist. Der Aufbau des Logikelementes inklusive des dedizierten Routing-Blocks ist in Anhang D detailliert dargestellt.

5.6.2 VLSI-Implementierung

Die VLSI-Implementierung des eFPGAs erfolgt mit dem Layout-Generator nach dem in Kapitel 4.3 dargestellten Verfahren in einer 130nm-Technologie. Sämtliche Simulationen wurden im Worst-Case durchgeführt. Insgesamt werden sechs Leaf-Zellen benötigt, so dass

die Portierung auf neue Halbleitertechnologien mit geringem Aufwand möglich ist. Der Routing-Switch liegt aufgrund seiner geringen Komplexität als monolithische Zelle vor. Zum Aufbau der Connection-Boxes werden vier verschiedene Leaf-Zellen verwendet. Eine weitere Leaf-Zelle ist für das Logikelemente erforderlich. Neben der Gesamtgröße des Makros kann in der Architekturbeschreibung auch die Größe der Cluster frei eingestellt werden. Abbildung 5.21 zeigt zwei exemplarische Implementierungen des eFPGAs mit verschiedenen Cluster-Größen.





Variante 1
$S_H=16$, $S_V=16$ (256 Logikelemente)
$C_H=8$, $C_V=2$ (8×2 LEs pro Cluster)
$W_H=W_V=4$ (Anzahl Verbindungsleitungen)
$I_N=2$ (vertikale Broadcast-Leitungen)
$I_E=1$ (horizontale Broadcast-Leitung)
$L_{H,1..4}=L_{V,1..4}=1$ (Segmentlänge $L=1$)

Variante 2
$S_H=16$, $S_V=16$ (256 Logikelemente)
$C_H=C_V=4$ (4×4 LEs pro Cluster)
$W_H=W_V=4$ (Anzahl Verbindungsleitungen)
$I_N=2$ (vertikale Broadcast-Leitungen)
$I_E=1$ (horizontale Broadcast-Leitung)
$L_{H,1..4}=L_{V,1..4}=1$ (Segmentlänge $L=1$)

Abbildung 5.21: Exemplarische VLSI-Implementierungen Architektur 3

Die LE-Dichte bei Skalierung auf eine 180nm-Technologie gemäß Anhang E beträgt für die gezeigten Beispielimplementierungen 224 LEs/mm^2 bzw. 228 LEs/mm^2. Berücksichtigt man, dass für Arithmetikoperationen ein Logikelement zwei Bitwertigkeiten repräsentiert, ergeben sich entsprechend 448 LEs/mm^2 bzw. 456 LEs/mm^2. Das eFPGA stellt damit bezüglich der Flächeneffizienz einen Kompromiss zwischen den beiden zuvor diskutierten Architekturen dar.

5.6.3 Bewertung der physikalischen Implementierungskosten

Zur Bewertung der physikalischen Implementierungskosten wurde wie für die Architekturen 1 und 2 ein ACS-Element auf das eFPGA abgebildet und bezüglich der Fläche

und Symbolperiode charakterisiert. Die AT-Kosten sind für die beiden implementierten Varianten annähernd gleich. Abbildung 5.22 zeigt einen Vergleich aller drei implementierten Beispielarchitekturen bei Skalierung auf eine 180nm-Technologie. Während Architektur 1 keine Effizienzsteigerung gegenüber der kommerziellen Architektur gezeigt hat, wurde bei Architektur 2 eine erhebliche Verbesserung der Flächeneffizienz erreicht, allerdings sehr stark zu Lasten des Flächenbedarfs. Architektur 3 ist diesbezüglich ausgewogen, insbesondere wird gegenüber dem APEX 20KE eine höhere Symbolperiode bei gleichzeitig geringfügig kleinerer Fläche erreicht.

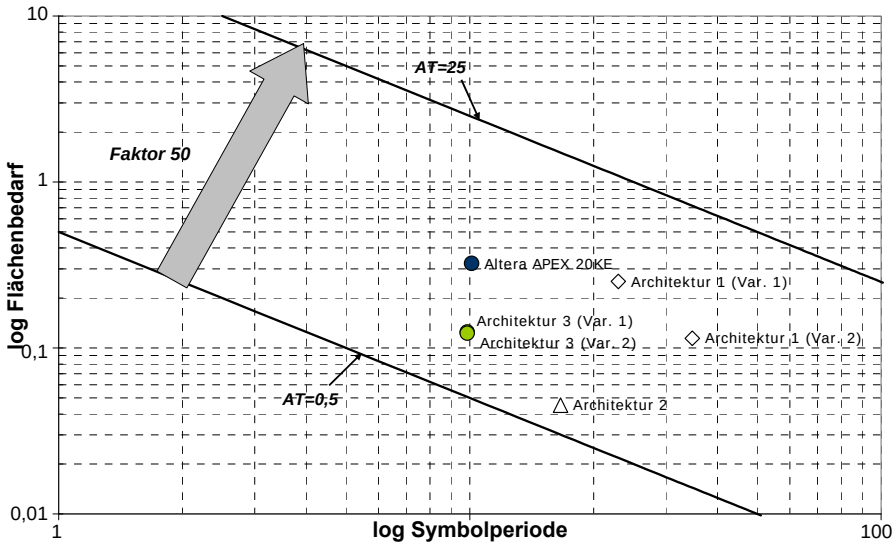


Abbildung 5.22 AT-Kosten ACS-Element für Architektur 3

Neben der Bestimmung der AT-Kosten wurden für Architektur 3 auch eine ausführliche Untersuchung der Energieeffizienz durchgeführt, die in modernen FPGAs eine immer höhere Bedeutung gewinnt. In diesen werden daher verschiedene Maßnahmen auf technologischer und schaltungstechnischer Ebene getroffen, um die statische Verlustleistung durch Leakage zu reduzieren [38]. Insbesondere können in aktuellen Halbleitertechnologien Transistoren mit verschiedenen Schwellenspannungen implementiert werden [45]. Dadurch ist es möglich, Transistoren im zeitkritischen Pfad einer Schaltung mit verringerter Schwellenspannung (low-VT) auf Kosten erhöhter Leckströme schneller schalten zu lassen, während Transistoren mit geringeren Anforderungen an die Schaltgeschwindigkeit mit erhöhter Schwellenspannung (high-VT) realisiert werden. Ebenfalls zum Einsatz kommt Back-Biasing. In modernen Bausteinen werden dabei programmierbare Schwellenspannungen verwendet, so dass beim Mapping Schaltungsteile außerhalb des kritischen Pfades durch Reduktion der negativen Back-Bias-Spannung langsamer aber verlustleistungsärmer betrieben werden.

Um die Schaltgeschwindigkeit und die dynamische Verlustleistung zu verbessern, werden auf technologischer Ebene z.B. Dielektrika mit besonders geringer Permittivität zwischen den Metallisierungsebenen verwendet, um die Kopplungskapazitäten zu verringern. Weiterhin kann durch Verwendung von so genanntem strained-Silicon die Ladungsträgerbeweglichkeit

und damit die Schaltgeschwindigkeit der Transistoren erhöht werden, indem die Gittereigenschaften des Siliziums durch mechanische Beeinflussung verändert werden.

Zur Bewertung der Verlustleistungs- und Flächeneffizienz der eFPGA-Architektur wurden exemplarisch einige Basisoperatoren sowie ein 4-Tap-FIR-Filter mit den in Kapitel 5.8.10 angegebenen Spezifikationen auf ein Makro mit $C_H=2$ und $C_V=4$ abgebildet. Diese Konfiguration des Clusters entspricht bei Normierung auf die verarbeiteten Wortbreiten Clustern mit 4×4 Elementen wie bei der in Kapitel 5.5 diskutierten Beispielarchitektur 2 mit shared-SRAMs. Für das Mapping der Operatoren wurde das eFPGA-Makro vollständig konfiguriert (siehe auch Kapitel 5.6.4) und Simulationen auf Basis Layout-extrahierter Netzlisten durchgeführt. Als Referenzwerte wurden die gleichen Operatoren für einen Altera Cyclone I synthetisiert. Tabelle 5.2 zeigt die Ergebnisse der Flächen-, Laufzeit- und Verlustleistungsanalyse für das FIR-Filter.

	Cyclone I EP1C20F400C7	**Architektur 3 ($C_H=2$, $C_V=4$)**
LEs	133	224
LEs/mm^2	196 (skaliert von APEX)	343
Kernspannung	1,5V	1,32V
max. Taktfrequenz	104MHz	238MHz
Operationen pro Takt	7 (4 Mul + 3 Add)	7 (4 Mul + 3 Add)
Verlustleistung bei f_{max}	37,2mW (skaliert auf 1,32V)	13,3mW (bei 1,32V)
Fläche A	0,68mm^2	0,65mm^2
Mio. Ops pro Sekunde	728	2550
Energie pro Operation	51pJ (skaliert auf 1,32V)	8pJ

Tabelle 5.2: ATE-Werte für 4-Tap-1D-FIR-Filter (skaliert auf 130nm-Technologie)

Für das Mapping auf das eFPGA wurde dabei eine Bitplane-Filterarchitektur verwendet, die einen relativ hohen Flächenbedarf zu Gunsten eines hohen Durchsatzes hat. Bei beiden Architekturen wurden nur tatsächlich allokierte Logikelemente in der Flächen- und Verlustleistungsbilanz berücksichtigt. Um identische Vergleichsbedingungen zu erreichen, wurden für die Implementierung auf dem Cyclone die dedizierten DSP-Blöcke abgeschaltet und nur Logikelemente verwendet. Für eine vergleichende Darstellung wurde eine vergleichende Darstellungsform der Verlustleistungs- und Flächeneffizienz gemäß [52] verwendet. Dabei werden unterschiedliche Algorithmen, die auf verschiedenen Hardware-Architekturen implementiert sind, auf einheitliche Bewertungsmaße (mW/MOPS und MOPS/mm^2) normiert und in einem Diagramm aufgetragen. Durch die Umrechnung der maximalen Taktfrequenz auf MOPS (Millionen Operationen pro Sekunde) sind auch verschiedene Algorithmen (Filter, Viterbi-Decoder, etc.) direkt bezüglich ihrer Verlustleistungs- und Flächeneffizienz vergleichbar. Abbildung 5.23 zeigt das Ergebnis der Einordnung in den

Entwurfsraum. Die vier hier betrachteten Operationen sind entsprechend hervorgehoben und in der Legende angegeben.

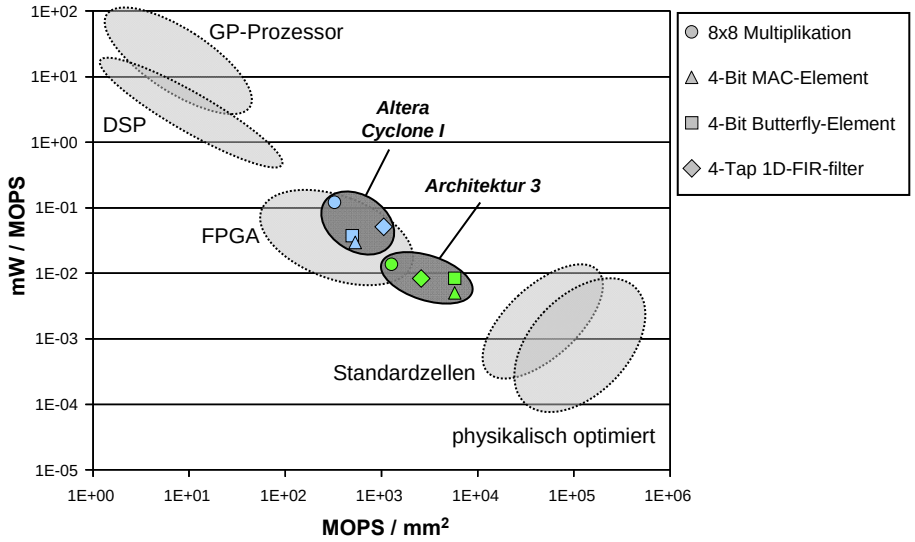


Abbildung 5.23: Einordnung der eFPGA-Architektur im Entwurfsraum

Für den Altera Cyclone I ergibt sich die zu erwartende Positionierung bezüglich mW/MOPS und MOPS/mm^2 im Bereich herkömmlicher FPGA-Architekturen. Die hier diskutierte arithmetikorientierte eFPGA-Architektur erreicht eine Positionierung zwischen FPGAs und Standardzell-Implementierungen. Gegenüber dem Cyclone wird eine Verbesserung der Verlustleistungs- und Flächeneffizienz um jeweils nahezu eine Größenordnung erreicht. Die Ergebnisse verdeutlichen das große Potenzial arithmetikorientierter eFPGA-Architekturen bei der Abbildung von Arithmetikdatenpfaden im Vergleich zu existierenden Architekturen, die stets einen Kompromiss für verschiedene Anwendungsklassen darstellen.

5.6.4 Konfiguration des eFPGA-Makros

Um eine vollständige Netzlistensimulationen der erzeugten eFPGA-Makros durchführen zu können, wurde für die hier beschriebene Architektur 3 ein elementares Konfigurationswerkzeug für alle Basiskomponenten des eFPGAs entwickelt. In Erweiterung zu dem in Kapitel 4.4 vorgestellten Verfahren steht dazu eine grafische Benutzeroberfläche zur Verfügung, welche eine visuelle Darstellung der Basiselemente der Architektur (Logikelement, Routing-Switch, Connection-Boxes) beinhaltet. In dieser erfolgt manuell die Auswahl der abzubildenden Funktionalität bzw. der zu schaltenden Verbindungswege. Für einen Anwender mit hinreichender Kenntnis über die eFPGA-Architektur ist es so möglich, reguläre Datenpfade mit moderatem Aufwand auf das Makro abzubilden, indem die Basiskomponenten entsprechend konfiguriert werden. Abbildung 5.24 zeigt die Benutzeroberfläche zur Konfiguration eines einzelnen Routing-Switch und des gesamten Clusters. Bei der Konfiguration des Routing-Switch können die Schalter innerhalb der Switch-Points gemäß

dem gewünschten Signalweg gesetzt werden. Außerdem lässt sich der Zustand der Ausgangstreiber einstellen, um eine aktive (auf eine Leitung schreibende) oder passive (von einer Leitung lesende) Verbindung herzustellen. Die Konfiguration des Routing-Switch wird dann in einer Datei abgespeichert. Ebenso werden die Logikelemente und Connection-Boxes konfiguriert. Die Konfiguration des gesamten Makros erfolgt durch Einlesen der gewünschten Konfiguration für die Einzelelemente auf Makroebene. In Abbildung 5.24 ist die entsprechende Benutzeroberfläche in der unteren Hälfte am Beispiel eines Makros mit $S_H=S_V=4$ und $C_H=2$ sowie $C_V=4$ dargestellt. Die Ausgabe der Konfigurationsbits kann direkt als Verilog-Code erfolgen, der von den Netzlistensimulatoren gelesen werden kann.

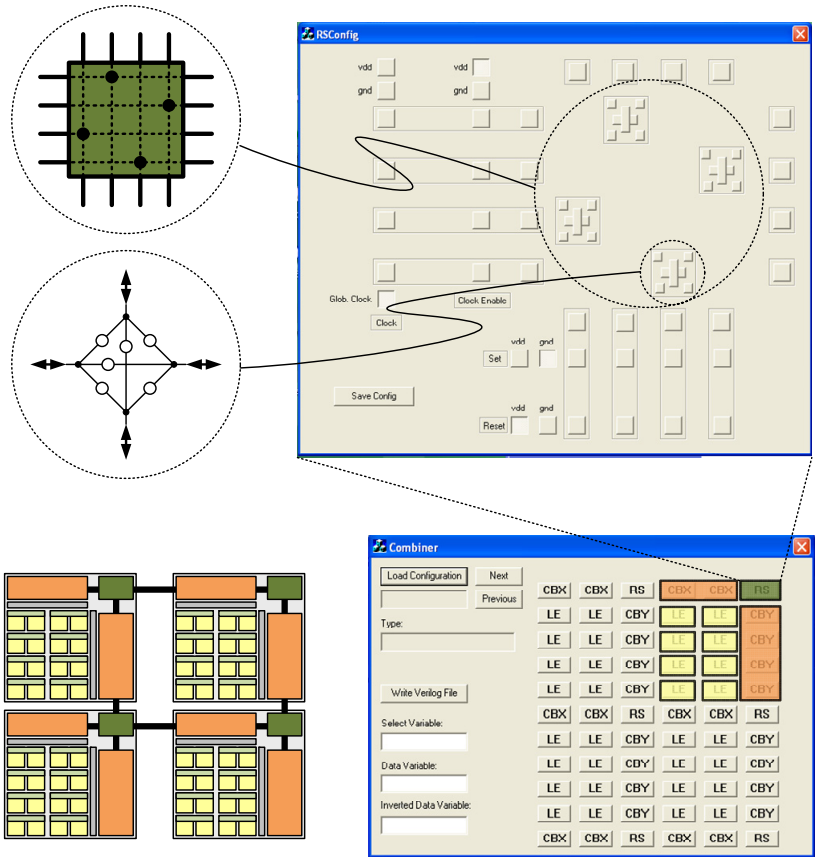


Abbildung 5.24: Benutzeroberfläche zur Konfiguration von Routing-Switch und Cluster

Die hier vorgestellten Ergebnisse wurden in [31][48][69] veröffentlicht.

5.6.5 Verwendung des eFPGAs als ASIP-Accelerator

Auf Basis des hier beschriebenen arithmetikorientierten eFPGAs wurde in der parallel zu dieser Arbeit entstandenen Co-Dissertation [67] ein Prozessor mit rekonfigurierbarem Accelerator konzipiert. Das eFPGA wurde dabei als RFU eng an den Prozessorkern gekoppelt (siehe Kapitel 3.2). Als Prozessorkern wurden ein MIPS-IV und ein ARM940T untersucht. Die auf den Accelerator ausgelagerten Operationen wurden auf Basis der hier vorgestellten

Ergebnisse modelliert. Als Entwicklungsumgebung für den MIPS wurden die frei verfügbaren Werkzeuge SimpleScalar [63] und Wattch [15] für die zyklengenaue Prozessorsimulation sowie Verlustleistungsmodellierung verwendet. Für den ARM wurden die ARM-Entwicklungsumgebung sowie ein hybrides FLPA-ILPA-basiertes Verlustleistungsmodell [11] verwendet. Als exemplarische Anwendungen wurden eine DES-Verschlüsselung und ein Median-Filter untersucht. Beide Anwendungen wurden sowohl als reine Software-Lösung auf dem MIPS bzw. ARM als auch auf der Prozessor-eFPGA-Architektur implementiert. Die Ergebnisse sind in Abbildung 5.25 wie in Kapitel 5.6.3 beschrieben im Entwurfsraum bezüglich ihrer Verlustleistungs- und Flächeneffizienz eingeordnet.

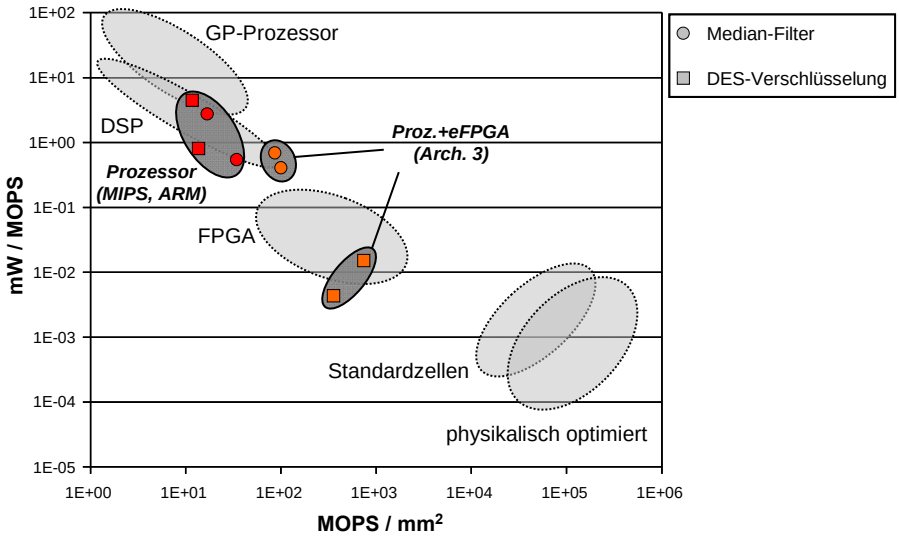


Abbildung 5.25: Einordnung der Prozessor-eFPGA-Architekturen im Entwurfsraum

Für das Median-Filter ist der Effizienzgewinn relativ gering, da der hohe Kommunikationsaufwand zwischen Prozessorkern und eFPGA aufgrund der Art der gewählten Kopplungsmechanismen zu einem häufigen Anhalten der Prozessor-Pipeline führt. Bei der DES-Verschlüsselung sind die Anforderungen an die Kommunikationsschnittstelle deutlich geringer, so dass hier eine Steigerung der Verlustleistungs- und Flächeneffizienz von jeweils knapp zwei Größenordnungen erreicht wird. Die Einordnung im Entwurfsraum ergibt eine Positionierung im unteren rechten Bereich der FPGA-basierten Implementierungen. Diese selbst für FPGAs besonders günstigen Werte unterstreichen die Synergieeffekte bei Verwendung eines arithmetikorientierten Accelerators für einen rekonfigurierbaren Prozessor. Obwohl die Prozessor-eFPGA-Architektur bezüglich ihrer Flexibilität mit einem herkömmlichen General-Purpose-Prozessor vergleichbar ist, liegt die erreichbare Effizienz im Bereich besonders effizienter FPGA-Implementierungen. Eine detaillierte Beschreibung der ASIP-eFPGA-Architektur ist in [67][68] gegeben.

5.7 Vergleich der implementierten Architekturen

Wie bereits zuvor beschrieben wurde, war der Entstehungsprozess der drei implementierten Architekturen chronologisch und durch sukzessive Verbesserungsmaßnahmen getrieben. Neben den architekturellen Verfeinerungen wurden dabei auch die Architekturparameter variiert. Tabelle 5.3 zeigt eine entsprechende Übersicht aller implementierten Varianten. Zur besseren Vergleichbarkeit sind die angegebenen Implementierungskosten jeweils auch skaliert auf eine 130nm-Technologie angegeben.

	Architektur 1		**Architektur 2**	**Architektur 3**	
	Var. 1	**Var. 2**		**Var. 1**	**Var. 2**
Technologie	180nm *		130nm	130nm	
# LEs	256	1024	256	256	256
Fläche	2,017mm^2 (1,052mm^2)	3,657mm^2 (1,908mm^2)	0,0943mm^2	0,596mm^2	0,586mm^2
LE	1 Bit, Arithmetik, wesentliche Verknüpfungen mit zwei Eingängen		1 Bit, LUT-3, ded. Carry, gated-FA	2 Bit, LUT-4, ded. Carry-Select, gated-FA	
$S_H \times S_V$	64×4	64×16	16×16	64×4	64×4
$C_H \times C_V$	4×4	2×4	4×4	8×2	4×4
$W_H \times W_V$	66×66	18×18	32×32	4×4	4×4
L_H, L_V (Segmentierung)	52 × $L_{H,V}$=1 14 × $L_{H,V}$=2	12 × $L_{H,V}$=1 6 × L_H=2 6 × L_V=4	1	1	1
I_N	0	0	2	2	2
I_E	0	0	1	1	1
$M_{LE,RS,CB}$	1	1	4	1	1
LEs/mm^2	127 (243)	280 (573)	2712	430	437
Implem.-kosten A	0,483mm^2 (0,251mm^2)	0,219mm^2 (0,114mm^2)	0,045mm^2	0,125mm^2	0,123mm^2
für ACS-Element T	44ns (22,95ns)	66,6ns (34,75ns)	12ns	9,85ns	9,85ns

Tabelle 5.3: Übersicht der implementierten Architekturvarianten
** Werte in Klammern skaliert auf 130nm*

Architektur 1 hat die größte Flexibilität im Verbindungsnetz und kommt damit herkömmlichen FPGA-Architekturen bezüglich der Eignung für kontrollflussorientierte Anwendungen am nächsten. Die große Anzahl von Verbindungsleitungen im globalen Interconnect erlaubt es, auch relativ irreguläre Logik ohne Ressourcenengpässe abzubilden. Da in dieser Architektur die Carry-Chain nicht über die Cluster-Grenzen hinausgeht, müssen

bei Operationen mit größerer Wortbreite zunehmend Geschwindigkeitseinbußen hingenommen werden. Am besten geeignet ist Architektur 1 für Anwendungen mit relativ hohem Kontrollflussanteil. Bei der Wahl der Architekturparameter ist es besonders wichtig, die Cluster-Breite gut auf die Wortbreiten der Basisoperationen der Zielanwendungen abzustimmen. Bei Anwendungen mit sehr hoher Regularität ist Architektur 2 deutlich effizienter. Die Verwendung der shared-SRAMs kann zu erheblichen Flächeneinsparungen führen, wobei die gewählte Granularität M=4 für die meisten Anwendungen einen sinnvollen Kompromiss darstellen dürfte, sofern die typischen Wortbreiten der Datenpfade ganzzahlige Vielfache dieses Wertes sind. Die Wahl des Logikelements und die Auswahl der möglichen Verbindungen durch die dedizierten Routing-Blöcke ist stark auf zweidimensionale Felder von Carry-Save-basierten Arrays zugeschnitten. Sofern die typischen Signalflussrichtungen zu diesem Schema passen, kann man ein relativ effizientes Mapping erreichen. Architektur 3 ist zu bevorzugen, wenn die Regularität innerhalb der Function-Slices keine sinnvolle Nutzung von shared-SRAMs erlaubt (z.B. weil die Funktionalität und damit die Konfiguration nebeneinander liegender Logikelemente alternierend ist). Die größeren LUTs und die flexibleren DRBs erlauben es im Gegensatz zu Architektur 2, auch Bereiche mit Random-Logik ohne übermäßige Effizienzeinbußen abzubilden. Ebenfalls zu bevorzugen ist diese Architektur für besonders große Arrays mit hohen Anforderungen an den Durchsatz, da die Feedthrough-Stufen und die vollständige Vernetzung über Cluster-Grenzen hinweg hier eine sehr gute Skalierung auch zu großen Wortbreiten hin zulässt.

Die implementierten Architekturen stellen einen sehr kleinen Ausschnitt des großen Parameterraums dar, der sich durch die Freiheitsgrade des Templates nach Kapitel 4 ergibt. Da es kein automatisiertes Verfahren zur Auswahl der geeigneten Parameter für eine vorgegebene Klasse von Anwendungen gibt, ist die Wahl bzw. Optimierung dieser Parameter ein komplexer Prozess, der sehr gute Kenntnis über die abzubildenden Anwendungen auf tiefer Implementierungsebene erfordert. Ein denkbarer Startpunkt wäre die Identifikation eines Satzes von Anwendungen, die auf das eFPGA abgebildet werden sollen. Anschließend kann man zu jedem der damit zur Verfügung stehenden Signalflussgraphen zunächst eine bitweise Abbildung auf Logikelemente durchführen. Dadurch ergeben sich die Anforderungen an die zu verwendende Kernlogik des eFPGAs. Anschließend muss man den lokalen Signalfluss zwischen benachbarten LEs für alle Datenpfade aufzeichnen und aus den sich ergebenden Alternativen die entsprechende Belegung der Multiplexer innerhalb der DRBs festlegen. Schließlich sind noch die globalen Signale über den Island-Style-Interconnect zu verdrahten, wobei wiederum aus der Summe der Anwendungen die maximal erforderliche Anzahl von Verbindungen ermittelt werden kann. Bei der Wahl der Granularität (shared-SRAMs) muss überprüft werden, inwieweit mögliche Einsparungen bei Datenpfaden mit entsprechender Homogenität bei anderen Datenpfaden den gegenteiligen Effekt haben (mangels entsprechender Homogenität) und einen sinnvollen Kompromiss wählen.

5.8 Modellbasierte Optimierung

Das Mapping von Datenpfaden auf eine parametrisierbare eFPGA-Architektur ist aufgrund der Komplexität der zu konfigurierenden Schaltung ein sehr aufwändiger Prozess, der ohne Automatisierung nur schwierig durchzuführen ist. Für eine allgemeine Evaluierung einer eFPGA-Architektur hinsichtlich ihrer Effizienz abhängig von den Architekturparametern ist jedoch in der Regel mit einem modellbasierten Ansatz eine ausreichend hohe Genauigkeit erzielbar. Neben der eigentlichen Konzeption der Architektur (siehe Kapitel 4) und ihrer Implementierung ist daher eine allgemeine, mathematische Modellierung der ATE-Kosten vorgenommen worden.

5.8.1 Implementierung und Charakterisierung der Basiszellen

Die Kernkomponenten zum Aufbau eines eFPGAs wie z.B. Logikelemente oder Switch-Points wurden als physikalisch optimierte Layouts in einer 90nm-Technologie aufgebaut. Bei der Untersuchung der Konfigurationsspeicherzellen wurden im Rahmen technologischer Optimierungsmaßnahmen verschiedene Schwellenspannungen (mixed-VT) verwendet, um die statische Verlustleistungsaufnahme zu reduzieren. Die Bestimmung der Verlustleistung erfolgte bei einer mittleren Schaltaktivität von $\sigma=0,5$. Der Konfigurationsvorgang der Elemente wird bei den angegebenen Verlustleistungswerten nicht einberechnet. Dies ist eine sinnvolle Annahme für Systeme, bei denen die für Konfigurationszyklen des eFPGAs aufgewendete Zeit im Vergleich zum gesamten betrachteten Zeitraum klein ist. Sämtliche Simulationen wurden im Worst-Case durchgeführt.

5.8.2 Kernlogik

Der Aufbau der Kernlogik entspricht der exemplarischen Architektur aus Kapitel 4.2.3 bzw. Abbildung 4.6. Die Dimensionierung aller Transistoren der Kernlogik wurde unter Verlustleistungsaspekten durchgeführt. Die Größe des Layouts (Höhe·Breite) ergibt sich wie folgt:

$$A_{KL} = 8,24 \cdot 12,54 \mu m^2 \approx 103,3 \mu m^2 \qquad (5.1)$$

Zur Bestimmung des Laufzeitverhaltens wurden unterschiedliche Funktionen auf die Kernlogik abgebildet. Der Einfluss der Konfiguration auf die Laufzeiten hat sich dabei als zu vernachlässigend herausgestellt. Für die Laufzeiten der dedizierten Signale wurde eine Carry-Select-Addition mit der Kernlogik betrachtet. Ein wesentlicher Einfluss auf die Laufzeit ergibt sich durch die Anzahl der Lasten, die an die Ausgänge der Kernlogik angeschlossen sind. Daher müssen die Anzahl der angeschlossenen Multiplexer-Eingänge der dedizierten Routing-Blöcke an den Ausgängen der Kernlogik sowie die entsprechenden Leitungslängen der Metallbahnen berücksichtigt werden. Für die Realisierung größerer Wortbreiten-Shifts zwischen zwei Function-Slices ist es zweckmäßig, Leitungen zwischen Logikelementen mit

einer Entfernung von mehr als einer Manhattan-Distanz auf dem LE-Raster mit einem Tristate-Treiber zu entkoppeln. Abbildung 5.26 stellt die Lastsituation eines Logikelements exemplarisch dar. Im Beispiel sind an dem Ausgang des Logikelements zwei direkte Nachbarn (Süden und Süd-Westen) sowie ein entferntes Logikelement angeschlossen.

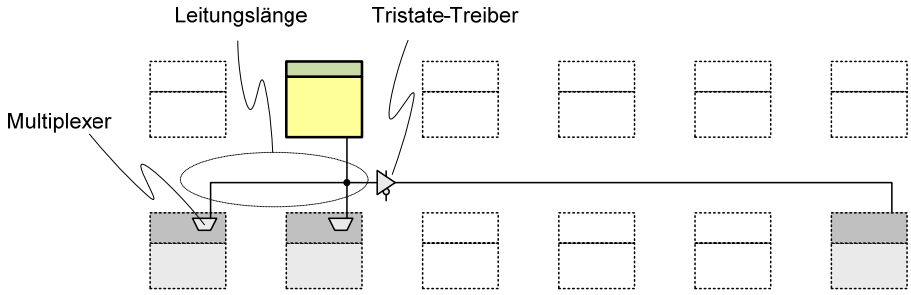


Abbildung 5.26: Lasten am Ausgang eines Logikelements

Die gesamte Laufzeit der Kernlogik bezüglich der Ausgänge O0 und O1 wurde durch lineare Regression verschiedener Simulationsreihen bestimmt. Sie ergibt sich aus einer Basislaufzeit von einem Eingang zu einem Ausgang gemäß Tabelle 5.4 und der Anzahl der offenen (no_{DRB}) und gesperrten (nc_{DRB}) DRB-Eingänge bzw. der Leitungslänge l am Ausgang wie folgt:

$$T_{KL,Ix\rightarrow O0} = T_{KL,basis,Ix\rightarrow O0} + 22{,}5ps \cdot no_{DRB} + 8{,}5ps \cdot nc_{DRB} + 1{,}3ps \cdot l/\mu m \qquad (5.2)$$

$$T_{KL,Ix\rightarrow O1} = T_{KL,basis,Ix\rightarrow O1} + 14{,}5ps \cdot no_{DRB} + 6{,}3ps \cdot nc_{DRB} + 1{,}3ps \cdot l/\mu m \qquad (5.3)$$

Für die dedizierten Signale sind dabei alle Lasten und Leitungslängen zu null zu setzen. Die Tristate-Treiber zum Entkoppeln längerer Leitungen können näherungsweise als gesperrte DRB-Eingänge gezählt werden und werden somit nc_{DRB} zugerechnet.

$T_{KL,basis}/ps$		**Eingang**						
		I0	**I1**	**I2**	**I3**	**CCI**	**CI0**	**CI1**
Ausgang	**O0**	395	380	530	490	430	390	390
	O1	240	270	580	520	420	460	460
	CO0	510	484	-	-	-	233	-
	CO1	542	514	-	-	-	-	207

Tabelle 5.4: Basislaufzeit der Kernlogik bei DRB als Ausgangslast

Für Logikelemente, die gemäß dem Architekturparameter N_{Reg} ein Register als Ausgangslast haben, ergeben sich für die Ausgänge O0 und O1 die Gesamtlaufzeiten gemäß Tabelle 5.5.

T_{KLreg}/ps		**Eingang**						
		I0	**I1**	**I2**	**I3**	**CCI**	**CI0**	**CI1**
Ausgang	**O0**	399	379	537	492	396	438	438
	O1	249	271	585	528	463	434	434

Tabelle 5.5: Laufzeit der Kernlogik bei Registerstufe als Ausgangslast

Verlustleistungssimulationen der Kernlogik für verschiedene Konfigurationen haben ergeben, dass auch hier keine Abhängigkeit von der abgebildeten Funktionalität besteht. Eine Ausnahme besteht, wenn die Kernlogik als reiner Bypass konfiguriert wird. In diesem Fall werden Signale vom Eingang I0 oder I1 über die dedizierte Summenlogik zum Ausgang O1 geschaltet. Wie bei der Laufzeit wird zudem zwischen dedizierten Routing-Blöcke oder Registerstufen als Ausgangslast differenziert. Es ergeben sich damit die folgenden Formeln zur näherungsweisen Berechnung der Verlustleistung für eine Kernlogik bei dedizierten Routing-Blöcken als Last (P_{KL}) bzw. Registern am Ausgang ($P_{KL,reg}$):

$$P_{KL} = P_{KL,basis} \cdot f/GHz + (0{,}4 + no_{DRB} + 0{,}22 \cdot nc_{DRB} + 0{,}05 \cdot l/\mu m)\mu W \cdot f/GHz \quad (5.4)$$

$$P_{KL,reg} = P_{KL,basis} \cdot f/GHz + 0{,}86\mu W \quad (5.5)$$

mit

$P_{KL,basis} = 19{,}8\mu W$ für alle Konfigurationen außer Bypass

$P_{KL,basis} = 3{,}7\mu W$ für Konfiguration als Bypass

5.8.3 Dedizierter Routing-Block

Die dedizierten Routing-Blöcke werden aus mehreren Multiplexern zusammengesetzt. Für die im Rahmen dieser Arbeit betrachteten Beispielanwendungen, die auf das eFPGA abgebildet wurden, werden in erster Linie Multiplexer mit zwei oder acht Eingängen benötigt. Es wurden exemplarisch Leaf-Zellen für beide Varianten aufgebaut. Die Breite des Mux8 entspricht dabei genau der Breite der Kernlogik, so dass sich die Multiplexer flächengünstig über der Kernlogik anordnen lassen. Der Mux2 ist so aufgebaut, dass seine Breite genau ein Viertel der Breite der Kernlogik beträgt. Neben der Verwendung im dedizierten Routing-Block ist der Mux2 auch zum Aufbau der Feedthrough-Stufe geeignet. Aus den Abmessungen der Basiszellen ergeben sich folgende Flächen für die Multiplexer:

$$A_{Mux2} = 2{,}56 \cdot 2{,}04\mu m^2 \approx 5{,}22\mu m^2 \quad (5.6)$$

$$A_{Mux8} = 8{,}24 \cdot 2{,}57\mu m^2 \approx 21{,}2\mu m^2 \quad (5.7)$$

Aufgrund des symmetrischen Aufbaus der Multiplexer ist die Laufzeit unabhängig vom gewählten Eingang. Die Last am Ausgang hängt davon ab, an welchen Eingang der Kernlogik der Multiplexer geschaltet ist. Je nach gewähltem Eingang besteht die Last aus einem LUT2-Eingang oder einer LUT2 mit parallel geschaltetem XOR-Gatter. Damit ergeben sich die in Tabelle 5.6 dargestellten Laufzeiten. Zusätzlich entstehen geringfügige Unterschiede durch die verschiedenen Leitungslängen der Metallbahnen im VLSI-Layout der Kernlogik.

T_{Mux}/ps		Multiplexer	
		Mux2	**Mux8**
Eingang Kernlogik	**I0**	157	242
	I1	160	245
	I2	129	212
	I3	130	205

Tabelle 5.6: Laufzeiten der DRB-Multiplexer abhängig vom angeschlossenen Eingang der Kernlogik

Die durch die dedizierten Routing-Blöcke verursachte Verlustleistung berechnet sich als Geradengleichung in Abhängigkeit von der Frequenz. Somit ergeben sich die folgenden Formeln für die Verlustleistung des Mux2 (P_{Mux2}) bzw. des Mux8 (P_{Mux8}) in Abhängigkeit vom Eingang *I* der folgenden Kernlogik:

$$P_{Mux2,I0} = P_{Mux2,I1} = 3{,}59\mu W \cdot f/GHz \quad (5.8)$$

$$P_{Mux2,I2} = P_{Mux2,I3} = 1{,}88\mu W \cdot f/GHz \quad (5.9)$$

$$P_{Mux8,I0} = P_{Mux8,I1} = 3{,}78\mu W \cdot f/GHz \quad (5.10)$$

$$P_{Mux8,I2} = P_{Mux8,I3} = 2{,}23\mu W \cdot f/GHz \quad (5.11)$$

5.8.4 SRAM-Zellen

Um auch bei Verwendung von shared-SRAMs mit $M>1$ flächeneffiziente Layouts erstellen zu können, werden die SRAM-Zellen komplett von den zu konfigurierenden Elementen (Logikelement, Switch-Point, etc.) getrennt und als monolithischer Block aufgebaut. Um den Flächenbedarf der SRAM-Blöcke zu reduzieren, werden Substratanschlüsse im Layout nur in der ersten Zeile eines SRAM-Blocks vorgesehen. Da die Zellen keine hohe Treiberfähigkeit benötigen, lässt sich durch diese Maßnahme auf Basis von zwei verschiedenen Leaf-Zellen der Flächenbedarf der Konfigurationsspeicher reduzieren. Es ergeben sich die folgenden Abmessungen für die Zelle ohne (SRAM) und mit (SRAM') Substratanschluss:

$$A_{SRAM} = 2{,}68 \cdot 1{,}34\mu m^2 \approx 3{,}59\mu m^2 \quad (5.12)$$

$$A_{SRAM'} = 2{,}68 \cdot 1{,}6\mu m^2 \approx 4{,}29\mu m^2 \quad (5.13)$$

Eine Laufzeitbetrachtung ist für die SRAM-Zellen nicht erforderlich, da sie an keiner Stelle der Schaltung im kritischen Pfad liegen. Bei der Untersuchung der Verlustleistung wird lediglich der statische Betrieb betrachtet, da eine Rekonfiguration des Makros im Vergleich zur Gesamtausführungszeit einer Anwendung auf dem eFPGA hier als vernachlässigbar angesehen wird. Nur bei hoher Rekonfigurationsrate, wie z.B. für die in Kapitel 3.4.3 beschriebenen Anwendungen für PiCoGA, würde dies entsprechend zu Fehlern in der

Verlustleistungsbestimmung führen. Entsprechend ist für SRAM-Zellen die Verlustleistung durch Sub-Threshold-Leakage bestimmt (siehe auch Kapitel 5.6.3).

Die statische Verlustleistung der SRAMs hängt wesentlich von der Schwellenspannung V_T der verwendeten Transistoren ab. Neben einer Implementierung der SRAM-Blöcke mit regulären Transistoren (regular-VT) wurde daher auch untersucht, wie stark sich durch Verwendung von Transistoren mit erhöhter Schwellenspannung (high-VT) die statische Verlustleistung reduzieren lässt. Die Simulation der SRAM-Zellen erfolgte sowohl im Typical-Corner als auch im Fast-Corner bei einer Betriebsspannung von 1V.

Abbildung 5.27 zeigt die Simulationsergebnisse für die Verlustleistung einer SRAM-Zelle bei verschiedenen Schwellenspannungen und verschiedenen Transistormodellen. Die Ergebnisse zeigen die Absenkung der statischen Verlustleistung bei Verwendung von high-VT statt regular-VT. Für typische Transistorparameter beträgt die Einsparung ungefähr 90%. Aufgrund des exponentiellen Anstiegs des Unterschwellenstroms in Abhängigkeit von der Schwellenspannung können prozessbedingte Fertigungsschwankungen auf einem Chip dazu führen, dass der Einfluss von V_T auf die Verlustleistung mit abnehmendem V_T erheblich größer wird. In Abbildung 5.27 ist dieser Effekt durch den großen Anstieg der Verlustleistung zwischen high-VT und regular-VT bei dem Worst-Case-Transistormodell „fast“ gekennzeichnet. Die statische Verlustleistung nimmt in diesem Fall um den Faktor zehn zu. Durch die Verwendung von high-VT-Transistoren für die SRAM-Zellen wird somit auch der Einfluss von Fertigungsstreuungen auf die Verlustleistungsaufnahme deutlich reduziert.

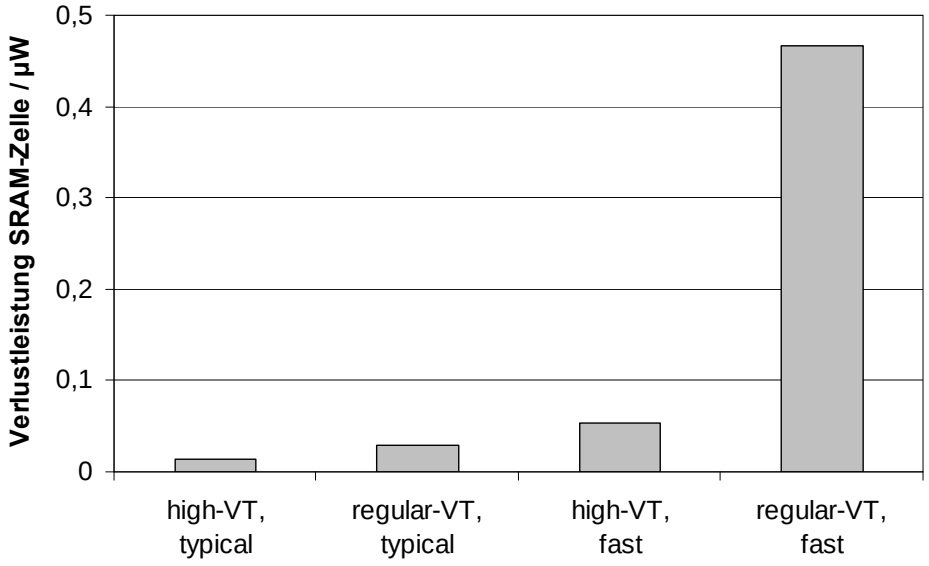


Abbildung 5.27: Verlustleistung einer SRAM-Zelle

Für das Gesamtmodell des eFPGA wurden die Verlustleistungswerte bei nominalen Transistorparametern verwendet. Damit ergeben sich je nach Verwendung von regular-VT bzw. high-VT-Transistoren die folgenden Verlustleistungswerte für eine SRAM-Zelle:

$$P_{SRAM,regVT} = 0{,}026\mu W \quad (5.14)$$

$$P_{SRAM,highVT} = 0{,}011\mu W \quad (5.15)$$

5.8.5 Registerstufe

Die Implementierung des Registers erfolgte als Master-Slave-Register mit einem optionalen Bypass vom Eingang zum Ausgang. Die Fläche des Layouts ergibt sich wie folgt:

$$A_{Reg} = 6{,}29 \cdot 6{,}97\mu m^2 \approx 39{,}3\mu m^2 \quad (5.16)$$

Bei der Bestimmung der Laufzeiten gelten die gleichen Rahmenbedingungen wie für die Kernlogik. Tristate-Treiber für lange lokale Leitungen werden auch hier näherungsweise als gesperrte DRB-Eingänge (nc_{DRB}) einberechnet. Neben der Laufzeit im normalen aktiven Betrieb des Registers ($T_{Reg,akt}$) wurde auch die Laufzeit über den Register-Bypass ($T_{Reg,byp}$) ermittelt. In Abhängigkeit von der Anzahl angeschlossener DRB-Eingänge und zu berücksichtigender Leitungslänge *l* ergeben sich die folgenden Formeln:

$$T_{Reg,akt} = 348ps + 53{,}5ps \cdot no_{DRB} + 13{,}6ps \cdot nc_{DRB} + 2{,}8ps \cdot l/\mu m \quad (5.17)$$

$$T_{Reg,byp} = 110{,}8ps + 53{,}5ps \cdot no_{DRB} + 18{,}1ps \cdot nc_{DRB} + 2{,}8ps \cdot l/\mu m \quad (5.18)$$

Für die Berechnung der Verlustleistung der Registerstufe gelten die oben genannten Bedingungen. Die Verlustleistung für ein aktives Register ($P_{Reg,akt}$) bzw. ein Register im Bypass-Betrieb ($P_{Reg,byp}$) ergibt sich damit wie folgt:

$$P_{Reg,akt} = (7{,}5 + 0{,}44 \cdot no_{DRB} + 0{,}09 \cdot nc_{DRB} + 0{,}018 \cdot l/\mu m)\mu W \cdot f/GHz \quad (5.19)$$

$$P_{Reg,byp} = (0{,}18 \cdot nc_{DRB} + 0{,}035 \cdot l/\mu m)\mu W \cdot f/GHz \quad (5.20)$$

5.8.6 Switch-Points des Routing-Switch

Für die hier vorgenommene Modellierung wurde der in Kapitel 4.2.4 (Abbildung 4.11) dargestellte Switch-Point verwendet, der gegenüber dem in Standard-FPGAs verwendeten Switch-Point gemäß Kapitel 2.2.4 (Abbildung 2.8) eine anwendungsklassenspezifisch reduzierte Flexibilität hat. Die Dimensionierung der Ausgangstreiber erfolgte mit einem Verhältnis der Gate-Weiten von P-Kanal- und N-Kanal-Transistoren im Verhältnis *2,5:1* für symmetrische Flanken und unter Berücksichtigung der anliegenden Last. Die vom Switch-Point nach außen treibenden Tristate-Inverter wurden entsprechend mit den Transistorweiten $w_p{:}w_n = 1200nm{:}480nm$ aufgebaut, während die den inneren Knoten treibenden Stufen mit $w_p{:}w_n = 600nm{:}240nm$ dimensioniert wurden. Die minimale Gate-Weite beträgt bei der verwendeten Halbleitertechnologie $w_{min} = 120nm$. Bei einer typischen Lastsituation mit Clustern der Breite $W_H=4$ und Leitungen der Segmentlänge $L=1$ ergibt sich damit ein günstiger Kompromiss zwischen Flächenbedarf und Laufzeit. Abbildung 5.28 zeigt einen

Überblick über Fläche und Laufzeit des Switch-Points für verschiedene Dimensionierungen von Eingangs- und Ausgangstreiber.

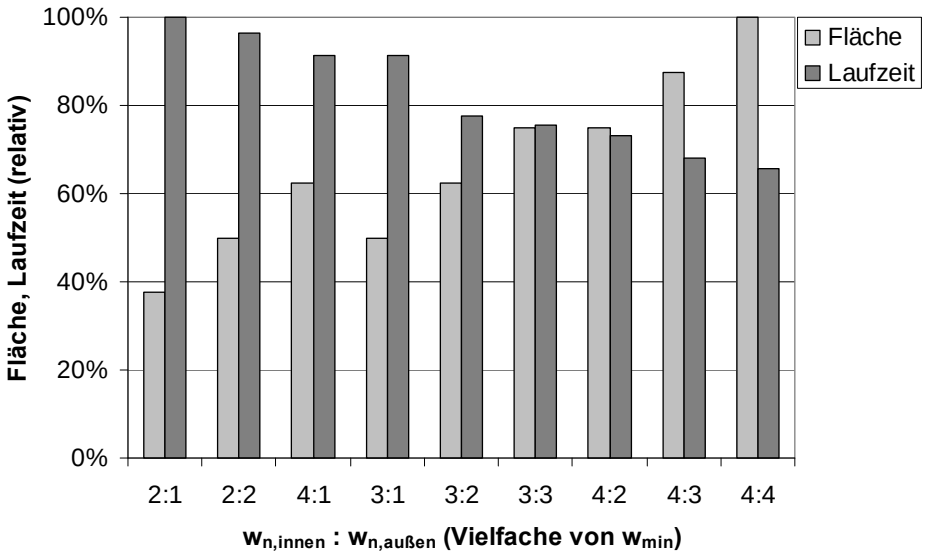


Abbildung 5.28: Fläche und Laufzeit des Switch-Points abhängig von der Dimensionierung

Gegenüber dem in Abbildung 2.8 gezeigten Aufbau mit sechs zusätzlichen Schaltern zwischen den vier Anschlüssen ergibt sich für den hier betrachteten Switch-Point bei gleicher Dimensionierung der Treiber eine Reduktion des Flächenbedarfs um 33% und eine um 75% schnellere Laufzeit. Bezüglich der Flexibilität ist die einzige Einschränkung dieses Aufbaus, dass nicht zwei Verbindungen gleichzeitig geschaltet werden können (z.B. N→S und gleichzeitig W→E). Der Flächenbedarf des Switch-Points ergibt sich wie folgt:

$$A_{SP} = 4{,}03 \cdot 5{,}88\mu m^2 \approx 23{,}7\mu m^2 \qquad (5.21)$$

Die Laufzeit hängt stark von der Anzahl zu treibender Gatter an den Ausgängen des Switch-Points bzw. der Leitungslänge l ab. Als Lasten werden ein identischer Switch-Point am nächsten angeschlossenen Routing-Switch und eine Anzahl von Tristate-Treibern in der Connection-Box angenommen. Dabei wird zwischen lesenden Verbindungen von der Verbindungsleitung zum Cluster (nr_{Tri}) und schreibenden Treibern vom Cluster zur Verbindungsleitung (nw_{Tri}) unterschieden. Der Zustand der Treiber (offen, gesperrt) wird nicht explizit berücksichtigt. Aufgrund der Symmetrie der Schaltung ist die Laufzeit zwischen zwei Anschlüssen unabhängig von deren Position (N, S, E, W). Allerdings spielt die Anzahl der aktiven Ausgänge eine Rolle. Wird eine Verbindung von einem Eingang zu einem Ausgang geschaltet (Fan-Out 1), ergibt sich die Laufzeit $T_{SP,fo1}$, während sich bei Verbindungen von einem Eingang zu zwei oder drei Ausgängen (Fan-Out ≥2) die Laufzeit $T_{SP,fo\geq 2}$ ergibt. Der Unterschied zwischen Fan-Out 2 und Fan-Out 3 ist mit weniger als 2% vernachlässigbar klein. Für die Laufzeiten ergeben sich die folgenden Formeln:

$$T_{SP,fo1} = 340ps + 5{,}2ns \cdot nr_{Tri} + 8{,}1ps \cdot nw_{Tri} + 1{,}4ps \cdot l/\mu m \quad (5.22)$$

$$T_{SP,fo\geq 2} = 354ps + 6{,}7ns \cdot nr_{Tri} + 9{,}6ps \cdot nw_{Tri} + 1{,}4ps \cdot l/\mu m \quad (5.23)$$

Analog dazu wird die Verlustleistung der Switch-Points modelliert. Die Zahl der aktiven Ausgangstreiber des Switch-Points $no_{SP} = [1..3]$ hat hier einen größeren Einfluss als bei der Laufzeit und wird als Variable in der Modellgleichung verwendet. Die Verlustleistung eines einzelnen Switch-Points wird durch folgende Formel beschrieben:

$$P_{SP} = (1{,}6 + 6{,}9 \cdot no_{SP} + 0{,}44 \cdot nr_{Tri} + 0{,}8 \cdot nw_{Tri} + 0{,}087 \cdot l/\mu m)\mu W \cdot f/GHz \quad (5.24)$$

5.8.7 Tristate-Treiber der Connection-Box

Jede Connection-Box setzt sich aus einer Anzahl von Tristate-Treibern zusammen. Die Dimensionierung der Transistoren wurde zu $w_p{:}w_n = 600nm{:}300nm$ gewählt. Für Connection-Boxes mit moderater Komplexität konnte anhand exemplarischer Implementierungen wie z.B. in Kapitel 5.6 beschrieben gezeigt werden, dass ihr Flächenbedarf für das hier betrachtete eFPGA-Architektur-Template in erster Linie vom Formfaktor des Routing-Switch abhängt. Die Größe der Connection-Box wird daher anhand der Abmessungen von Cluster ($H_{Cl} \cdot B_{Cl}$) und Routing-Switch ($H_{RS} \cdot B_{RS}$) bestimmt. Es ergibt sich damit für die horizontale ($A_{CB,H}$) bzw. vertikale ($A_{CB,V}$) Connection-Box der folgende Flächenbedarf:

$$A_{CB,H} = H_{RS} \cdot B_{Cl} \quad (5.25)$$

$$A_{CB,V} = H_{Cl} \cdot B_{RS} \quad (5.26)$$

Die Laufzeit hängt im Wesentlichen von der Leitungslänge über ein gesamtes Cluster ab. Andere Einflussfaktoren wie die Anzahl angeschlossener DRB-Eingänge werden hinreichend genau durch die Formel beschrieben, da diese mit der Leitungslänge korreliert sind. In dieser Formel ist auch die Feedthrough-Stufe bereits berücksichtigt. Es ergibt sich für die Laufzeit eines Tristate-Treibers der Connection-Box der folgende Zusammenhang:

$$T_{Tri} = 87ps + 2{,}5ps \cdot l/\mu m \quad (5.27)$$

Die Verlustleistung der Treiber hängt stark davon ab, ob der Treiber aktiv ($P_{Tri,a}$) oder abgeschaltet, also passiv ($P_{Tri,p}$) ist. Es ergeben sich die folgenden Formeln:

$$P_{Tri,a} = (1{,}4+0{,}029 \cdot l/\mu m)\mu W \cdot f/GHz \quad (5.28)$$

$$P_{Tri,p} = 0{,}057\mu W \cdot f/GHz \quad (5.29)$$

5.8.8 Taktnetz

Das Taktnetz eines FPGAs verfügt in der Regel über dedizierte Leitungen und eigene Treiberstufen für die Zuführung des Taktsignals zu den Registern. Um die Implementierung

asynchroner Schaltnetze auf dem eFPGA zu ermöglichen, können Taktsignale auch über die konfigurierbaren Verbindungsleitungen zu den Logikelementen geschaltet werden. Zur Vereinfachung der Modellierung wird das Taktnetz des hier betrachteten eFPGAs durch Switch-Points und Tristate-Treiber modelliert. Es wird dabei angenommen, dass sich das Taktsignal auf sämtlichen Routing-Switches des eFPGA über jeweils einen Switch-Point des Routing-Switch ausbreitet und in den horizontalen Connection-Boxes über einen aktiven Tristate-Treiber pro Spalte von Logikelementen in die Cluster geschaltet wird. Durch den zusätzlichen Faktor zwei wird die höhere Schaltaktivität des Taktsignals berücksichtigt.

5.8.9 ATE-Modell des eFPGAs

Die gesamten Implementierungskosten für auf das eFPGA abgebildete Operationen lassen sich durch das Auswerten der zuvor beschriebenen Modellgleichungen der Einzelelemente unter Berücksichtigung der eingestellten Architekturparameter ermitteln. Insbesondere müssen physikalische Aspekte wie Leitungslängen und die Anzahl der Lasten an Ausgängen auf Architekturparameter zurückgeführt werden. Einige Aspekte wie die Differenzierung zwischen aktiven und abgeschalteten Treibern hängen direkt vom Mapping der Anwendung auf die eFPGA-Architektur ab. Die in Kapitel 5.8.1 angegebenen mathematischen Modellgleichungen der Einzelkomponenten wurden in das MATLAB-Modell des Architektur-Templates (vgl. Kapitel 4.2 und Kapitel 4.4) integriert. Mit Hilfe der so erweiterten Beschreibung des eFPGAs können die ATE-Kosten von Datenpfaden teilweise automatisiert durchgeführt werden. Für das Mapping einer Operation auf die generische eFPGA-Architektur muss dazu eine manuelle Allokation aller Logikelemente durchgeführt werden.

Der Flächenbedarf A einer Operation beim Mapping auf ein eFPGA mit gegebenen Architekturparametern ergibt sich durch die Anzahl der benötigten Logikelemente. Die Einzelflächen der Basiselemente werden dann automatisch unter Berücksichtigung der Geometrie des physikalischen Layouts aufsummiert. Für die Bestimmung der gesamten Laufzeit T muss manuell der zeitkritische Pfad der Operation bestimmt werden. Die Angabe erfolgt dabei in Form einer Liste der Basiselemente, die im zeitkritischen Pfad liegen. Es ist nicht nötig, eine manuelle Belegung sämtlicher Verbindungsressourcen des eFPGAs durchzuführen, da lediglich dieser zeitkritische Pfad für die Laufzeit relevant ist. Der Energieumsatz E für eine Ausführung einer Operation ergibt sich durch das Produkt $E=P(f)\cdot 1/f$ der Verlustleistung bei einer gegebenen Frequenz mit dem Kehrwert der Frequenz. Für die Berechnung des Energieumsatzes muss die absolute Anzahl der sperrenden bzw. aktiven Treiber im Verbindungsnetz angegeben werden. Auf Basis dieser statistischen Angabe, die sich aus der Allokation der Logikelemente und einer Abschätzung der Anzahl von benötigten Verbindungsleitungen ergibt, werden die Modellgleichungen mit den entsprechenden Parametern für die Last ausgewertet. Die durch die Metallleitungen verursachte Last wird automatisch aus den gegebenen Architekturparametern und den

Abmessungen der Basiszellen ermittelt. Weitere Details zur Modellierung der Implementierungskosten einer parametrisierbaren eFPGA-Architektur sind in [28] angegeben.

5.8.10 Ergebnisse

Auf Basis des physikalischen Kostenmodells wurden die Implementierungskosten für drei exemplarische Datenpfade ermittelt. Neben dem auch in Kapitel 5.6 verwendeten FIR-Filter wurden zwei spezielle Operationen zur Beschleunigung eines GPS-Software-Korrelators betrachtet.

Das hier betrachtete unsymmetrische FIR-Filter in transponierter Form hat vier Taps, eine Eingangswortbreite von 8 Bit und feste Koeffizienten mit den Werten $c_0=5$, $c_1=45$, $c_2=51$ und $c_3=7$. Abbildung 5.29 zeigt den Signalflussgraph des Filters.

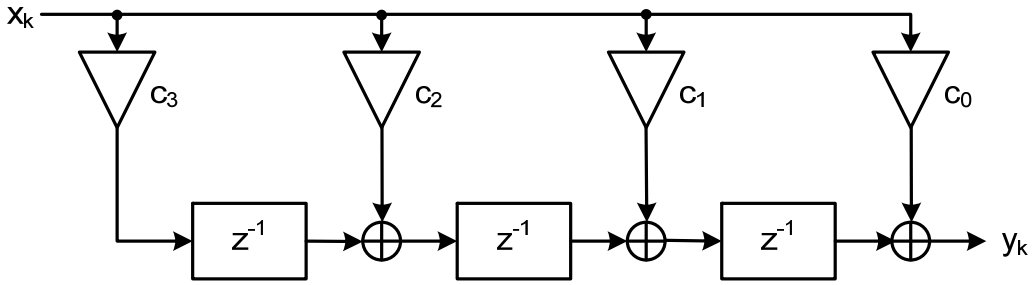


Abbildung 5.29: Signalflussgraph des FIR-Filters

Die entsprechende Gleichung für dieses Filter lautet:

$$y_k = c_0 \cdot x_k + c_1 \cdot x_{k-1} + c_2 \cdot x_{k-2} + c_3 \cdot x_{k-3} \qquad (5.30)$$

Anders als in Kapitel 5.6 wurde für das Mapping auf die modellierte Beispielarchitektur keine Bitplane-Implementierung gewählt, da diese sehr viele Register benötigt. Stattdessen erfolgt eine direkte Abbildung des Signalflussgraphen auf das eFPGA. Bei der Multiplikation mit den Koeffizienten werden alle Zero-Digits eliminiert, um den Hardware-Aufwand zu senken. Damit ergeben sich für die vier Taps bei den gegebenen Konstanten c_0 bis c_3 die folgenden Gleichungen für die Implementierung der Multiplikation des Eingangswertes mit dem konstanten Faktor:

$$x_k \cdot c_0 = x_k \cdot 2^0 + x_k \cdot 2^2 \qquad (5.31)$$

$$x_k \cdot c_1 = x_k \cdot 2^0 + x_k \cdot 2^2 + x_k \cdot 2^3 + x_k \cdot 2^5 \qquad (5.32)$$

$$x_k \cdot c_2 = x_k \cdot 2^0 + x_k \cdot 2^1 + x_k \cdot 2^4 + x_k \cdot 2^5 \qquad (5.33)$$

$$x_k \cdot c_3 = x_k \cdot 2^0 + x_k \cdot 2^1 + x_k \cdot 2^2 \qquad (5.34)$$

Die beschriebene Implementierungsform wurde mit Rücksicht auf die Komplexität des Mappings gewählt. Tatsächlich würde man für die gewählten Koeffizienten sinnvollerweise

eine CSD-Zahlendarstellung (Canonical-Signed-Digit) wählen, um die Implementierungskosten zu senken.

Abbildung 5.30 zeigt die Allokation der Logikelemente für die Abbildung des FIR-Filters. Die jeweils benötigte Konfiguration ist für jedes Logikelement angegeben. Die vier Taps des Filters werden untereinander angeordnet. In jedem Tap werden die Additionen auf Carry-Select-Addierer abgebildet. Die Multiplikation wird durch entsprechende Verschiebungen der Wertigkeiten zwischen den Function-Slices realisiert. Die für das Filter nicht benötigten Logikelemente am linken und rechten Rand sind in der Abbildung hell gekennzeichnet. Für das Mapping des FIR-Filters wurde zunächst eine Architektur gewählt, bei der die Logikelemente in Clustern der Größe 4×4 organisiert sind ($C_H=C_V=4$). Es werden vier Broadcast-Leitungen von Norden sowie eine von Osten als Eingänge verwendet ($I_N=2$, $I_E=1$).

Abbildung 5.30: Allokation der Logikelemente für das FIR-Filter

Da bei der gewählten Form des Filters der Datenfluss von oben nach unten erfolgt, werden die Ausgänge des Clusters nur nach Süden an die Connection-Box geschaltet. Es werden jeweils zwei benachbarte Logikelemente identisch konfiguriert ($M_{LE}=2$). Um den durch die Register verursachten Flächenmehraufwand möglichst stark zu reduzieren, wurde $N_{Reg}=2$ gewählt und neben den beiden Ausgängen der Kernlogik nur eine der beiden vertikalen Broadcast-Leitungen mit einem Register versehen. Im Verbindungsnetz stehen jeweils $W_H=W_V=8$

Leitungen zur Verfügung. Der Routing-Switch ist so belegt, dass ähnlich wie bei einem Disjoint-Switch jede horizontale Leitung nur einer vertikalen zugeordnet ist. Die Leitungen werden zu zwei Bussen mit $M_{RS}=4$ zusammengefasst und haben alle die Segmentlänge $L_i=1$. Die Connection-Boxes haben zwei periodisch beschaltete Kanäle mit Fensterbreiten von $w_{PC,H,i}=2$ für die horizontalen und $w_{PC,V,j}=1$ für die vertikalen Leitungen, so dass die horizontalen Verbindungen dichter belegt sind. Die Connection-Box kann mit $M_{CB}=1$ bitweise konfiguriert werden. Die dedizierten Routing-Blöcke sind so aufgebaut, dass an den Eingängen I0 und I1 der Kernlogik neben vertikalen Verbindungen auch ein diagonaler Shift um eine Wertigkeit nach links oder rechts sowie Shifts um zwei oder acht Wertigkeiten nach rechts zwischen zwei Function-Slices am Eingang I0 möglich sind. An den Eingängen I2 und I3 wird jeweils eine der Broadcast-Leitungen in horizontaler bzw. vertikaler Richtung angeschlossen. Insgesamt werden zwei Multiplexer mit je acht Eingängen sowie zwei mit je zwei Eingängen benutzt.

Das betrachtete FIR-Filter benötigt bei Mapping auf diese Architektur insgesamt 176 Logikelemente. Davon sind vier Verschnitt, also ohne Funktion für das Filter, und weitere 16 sind als reiner Bypass konfiguriert. Auf Basis der manuellen Allokation der Logikelemente gemäß Abbildung 5.30 wurde ermittelt, dass aufgrund der großen Homogenität der Konfigurationsinformationen die Anzahl der gemeinsam konfigurierten Logikelemente auf $M_{LE}=8$ bei $C_H=8$ erhöht werden kann, ohne dass die Verschnitteffekte signifikant zunehmen. Im vertikalen Verbindungsnetz wird die Anzahl der Leitungen auf $W_V=4$ reduziert, da der Datenfluss im Filter nahezu ausschließlich über die dedizierten lokalen Verbindungsleitungen erfolgen kann. M_{RS} kann dadurch von vier auf zwei verringert werden, so dass ohne Nachteile bezüglich des Flächenbedarfs die Verbindungen mit feinerer Granularität konfiguriert werden können. Als alternative Optimierungsmaßnahme wird ausgehend von der ersten Architekturvariante die Anzahl identisch konfigurierter Logikelemente auf $M_{LE}=16$ bei $C_H=16$ erhöht, ohne die Spezifikationen des globalen Verbindungsnetzes zu ändern. Da bei dieser Architekturvariante eine komplette Function-Slice auf die Breite eines Clusters abgebildet werden kann, ist eine besonders effiziente Verwendung der dedizierten Carry-Logik möglich.

In Tabelle 5.7 sind eine Übersicht der Parameter für die drei Architekturvarianten sowie die Resultate der Modellgleichungen für die Fläche, Laufzeit und Verlustleistung dargestellt. Die Ergebnisse zeigen, dass durch die geeignete Wahl von M_{LE} der Flächenbedarf des Filters um bis zu 36% verringert werden kann. Der zeitkritische Pfad ändert sich von Variante I zu Variante II nur unwesentlich, da in beiden die Carry-Chain durch zwei benachbarte Cluster läuft. Durch die Wahl einer sehr groben Granularität von $M_{LE}=16$ für Variante III steigt die maximale Taktfrequenz, da der Carry-Pfad nur innerhalb des Clusters verläuft. Die Verlustleistung ist jeweils bei der maximalen Taktfrequenz ermittelt worden. Durch die gröbere Granularität sinkt der Anteil der Verlustleistung durch die Konfigurationsspeicher von über 100µW auf ca. 35µW bei Variante III. Durch die Reduktion der Anzahl der

vertikalen Verbindungsleitungen wird bei Variante II zudem eine Verringerung der Verlustleistung in den Connection-Boxes und Routing-Switches von 110µW erreicht. Die Verlustleistung wurde für Konfigurationsspeicher sowohl mit regular-VT als auch mit high-VT ermittelt. Die dadurch erreichbare Reduktion der Verlustleistung hängt stark vom Anteil der SRAMs an der Gesamtfläche des Makros ab. Für die Architekturvariante I wurde eine Verringerung um ungefähr 4% erreicht, für die Varianten mit größerem M_{LE} ist das Optimierungspotenzial noch geringer. Ein erheblicher Vorteil ist jedoch die in Kapitel 5.8.4 beschriebene Verringerung der Abhängigkeit der Verlustleistung von den Transistorparametern.

	Arch. 3 (Modell), Variante I	**Arch. 3 (Modell), Variante II**	**Arch. 3 (Modell), Variante III**
$C_V \times C_H$	4×4	8×4	16×4
M_{LE}	2	8	16
M_{CB}	1	1	1
M_{RS}	4	2	4
$W_V \times W_H$	8×8	8×4	8×8
Fläche A	0,0793mm^2	0,0591mm^2	0,0514mm^2
max. Taktfrequenz	126MHz	122MHz	142MHz
Verlustleistung P bei f_{max}	1,74mW 1,68mW (high-VT)	1,54mW 1,50mW (high-VT)	1,67mW 1,65mW (high-VT)

Tabelle 5.7: Architekturparameter und Implementierungskosten des FIR-Filters für verschiedene Architekturvarianten

Systeme zur Satellitennavigation wie z.B. das US-amerikanische NAVSTAR-GPS oder das europäische Galileo benötigen aufwändige Korrelationsberechnungen zur Decodierung der vom Satelliten empfangenen Signale. Kommerzielle Empfängerarchitekturen verfügen daher in der Regel über dedizierte Korrelatoren. Eine attraktive Alternative ist die Verwendung von Software-Korrelatoren, um eine flexible Anpassung an die Anforderungen des jeweiligen Navigationssystems zu erreichen [14]. Typische Eingangswortbreiten eines Korrelators liegen zwischen zwei und vier Bit. Die Rechenwerke moderner Mikroprozessoren haben deutlich größere Wortbreiten (z.B. 32 Bit), so dass keine effiziente Abbildung auf die vorhandenen Mutliplizierereinheiten eines solchen Prozessors möglich ist. Bei der in [27] beschriebenen ASIP-eFPGA-Architektur wird die QuadSMul-Operation als Custom-Instruction verwendet, um die Flächen- und Verlustleistungseffizienz eines GPS-Software-Korrelators zu steigern. Die Operation führt in einem Berechnungsschritt vier Multiplikationen mit je 4 Bit breiten Eingangswörtern aus. Statt in mehreren sequentiellen Prozessorzyklen die Multiplikationsoperation des ASIPs mit 32 Bit Wortbreite zu benutzen, können durch die Verwendung der Custom-Instruction in einem Schritt vier Multiplikationen verarbeitet werden. Abbildung 5.31 zeigt den Signalflussgraph der QuadSMul-Operation.

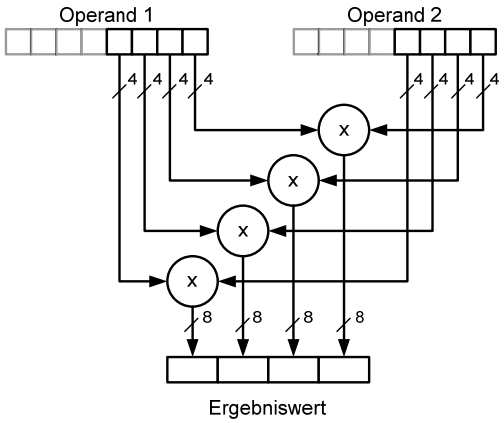


Abbildung 5.31: Signalflussgraph der QuadSMul-Operation

Abbildung 5.32 zeigt einen Ausschnitt der Allokation der Logikelemente für die QuadSMul-Operation. Die einzelnen Multiplizierer werden als Carry-Save-basierte Array-Multiplizierer realisiert. Dadurch ist eine effiziente Nutzung der Broadcast-Leitungen möglich. Die Größe der Cluster wurde zu $C_V=4$ gewählt, so dass jeweils die vier Function-Slices für die Addition der Partialprodukte innerhalb eines Clusters angeordnet sind. In horizontaler Richtung beträgt die Clustergröße $C_H=8$, um den Wortbreitenzuwachs zu berücksichtigen. Um ein moderates Pipelining zu ermöglichen, wurde $N_{Reg}=4$ gewählt, so dass nach jeweils vier Function-Slices ein Pipeline-Schnitt erfolgen kann. Für die Logikelemente und die Connection-Boxes werden keine shared-SRAMs benutzt ($M_{LE}=1$). Ansonsten gelten die gleichen Spezifikationen wie bei der Ausgangsarchitektur für das FIR-Filter.

Die QuadSMul-Operation benötigt 256 Logikelemente, wodurch sich ein Flächenbedarf von $A=0{,}11mm^2$ ergibt. Dabei sind 18 Logikelemente inaktiv und 21 als Bypass konfiguriert. Die maximale Taktfrequenz beträgt $f=129MHz$, die Verlustleistung bei dieser Frequenz beträgt $P=1{,}99mW$ für Konfigurationsspeicher mit regular-VT.

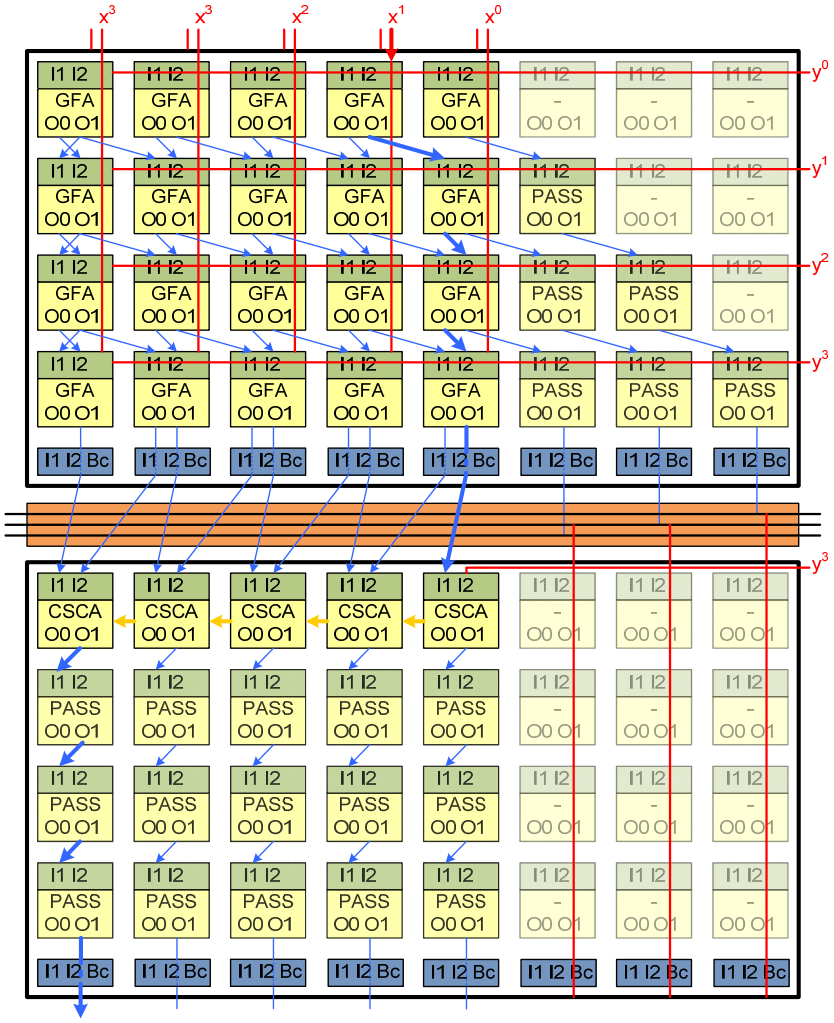


Abbildung 5.32: Allokation der Logikelemente für die QuadSMul-Operation (Auszug)

Die QuadMAC-Operation wird ebenfalls bei der in [27] beschriebenen ASIP-eFPGA-Architektur zur Beschleunigung der Korrelationsberechnung eines GPS-Empfängers verwendet. Wie bei der QuadSMul-Operation werden vier Multiplikationen und eine anschließende Akkumulation der Ergebniswerte über einen Baumaddierer zu einer einzigen Custom-Instruction zusammengefasst. Die abschließende Addition der akkumulierten Zwischenergebnisse mit einem weiteren Operanden erfolgt in einem Addierer mit 32 Bit Wortbreite. Abbildung 5.33 zeigt den Signalflussgraph der QuadMAC-Operation.

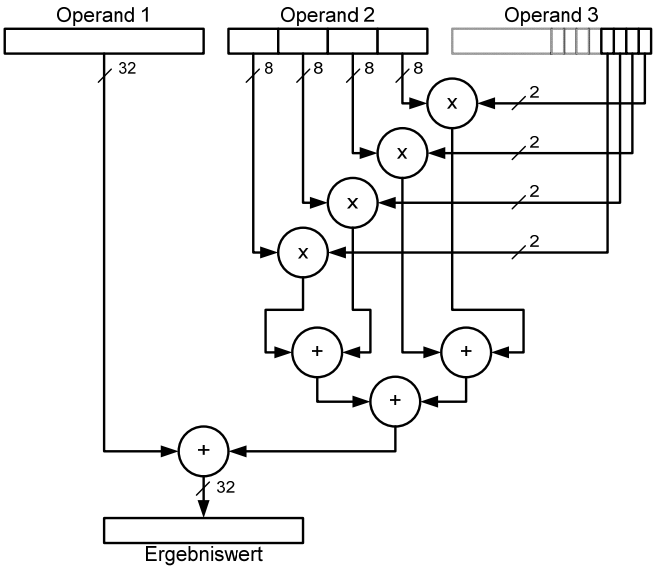


Abbildung 5.33: Signalflussgraph der QuadMAC-Operation

Bei der Abbildung der Operation auf das eFPGA wurden die gleichen Architekturparameter wie für die QuadSMul-Operation verwendet. Insgesamt werden 256 Logikelemente benötigt, von denen 32 inaktiv und 80 als Bypass konfiguriert sind. Abbildung 5.34 zeigt einen Ausschnitt der Allokation der Logikelemente. Der Flächenbedarf beträgt $A=0,11mm^2$, die maximale Taktfrequenz $f=84MHz$ und die entsprechende Verlustleistung $P=1,4mW$ bei Verwendung von Konfigurationsspeichern mit regular-VT.

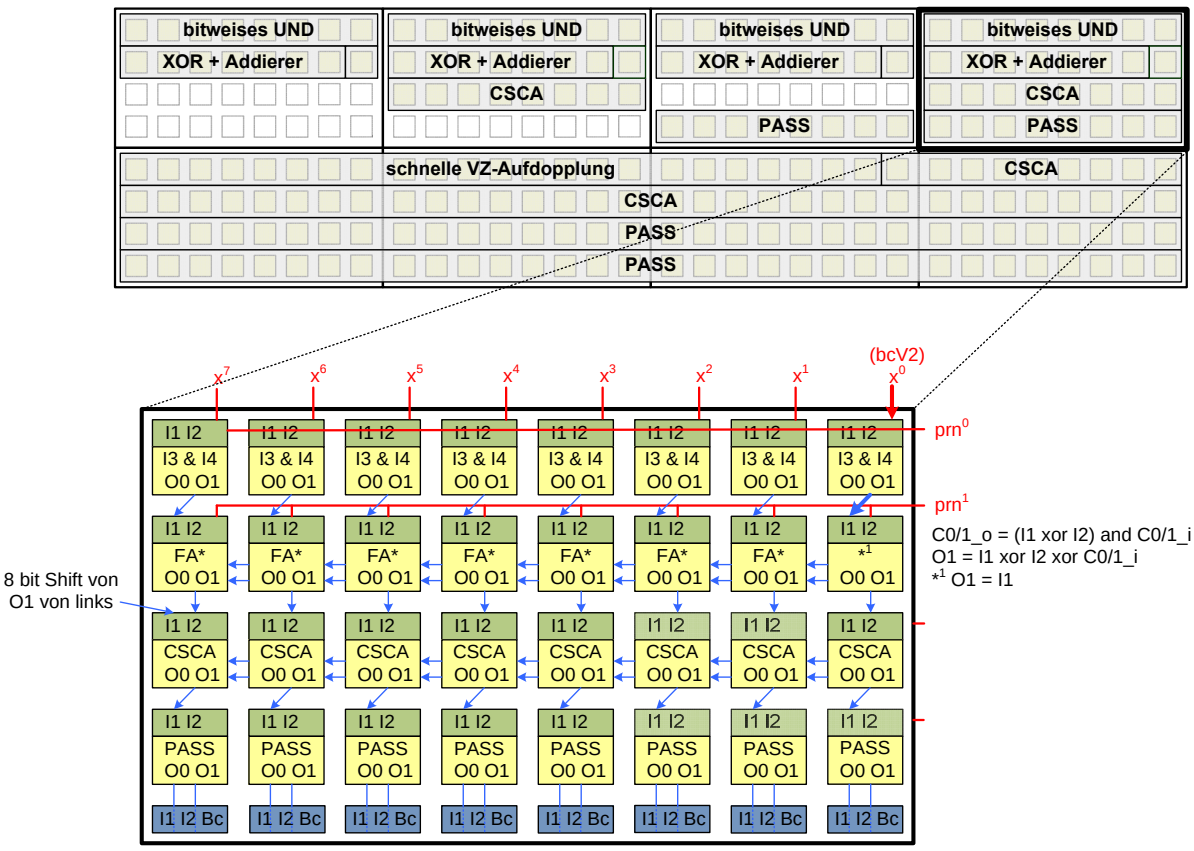


Abbildung 5.34: Allokation der Logikelemente für die QuadMAC-Operation (Auszug)

Die drei zuvor beschriebenen Operationen wurden wie in Kapitel 5.6.3 dargestellt bezüglich ihrer Verlustleistungs- und Flächeneffizienz gemessen in mW/MOPS und MOPS/mm^2 bewertet. Tabelle 5.8 listet exemplarisch für die QuadMAC-Operation die physikalischen Implementierungskosten bei Skalierung auf eine 130nm-Technologie gemäß Anhang E auf. Zusätzlich sind die Kosten für die Implementierung einer vergleichbaren Operation auf einem kommerziellen Altera Cyclone I angegeben.

	Cyclone I EP1C20F400C7	**Architektur 3 (Modell)**
LEs	291	256
LEs/mm^2	196 (skaliert von APEX)	1114
max. Taktfrequenz	104MHz	61MHz
Operationen pro Takt	7 (4 Mul + 3 Add)	7 (4 Mul + 3 Add)
Verlustleistung bei f_{max}	32,3mW (skaliert auf 1,32V)	3mW
Fläche A	1,49mm^2	0,23mm^2
Mio. Ops pro Sekunde	602	430
Energie pro Operation	51pJ (skaliert auf 1,32V)	7pJ

Tabelle 5.8: ATE-Werte für QuadMAC-Operation (skaliert auf 130nm-Technologie)

Die maximale Taktfrequenz ergibt sich aus dem Kehrwert der Laufzeit des kritischen Signalpfades, die Verlustleistung wurde bei dieser Taktfrequenz ermittelt. Die Anzahl der Operationen pro Sekunde ergibt sich aus dem Produkt von maximaler Taktfrequenz und der Anzahl der Operationen, die pro Takt durchgeführt werden. Die Anzahl der Operationen pro Takt ist ein Normierungsmaß, das die Anzahl der Elementarberechnungen eines Datenpfades beschreibt. Bei der QuadMAC-Operation nach Abbildung 5.34 sind dies z.B. vier Multiplikationen und drei Additionen. Der Energieumsatz ergibt sich als Quotient aus der Verlustleistung und den Operationen pro Sekunde.

Zur Überprüfung der Modellgenauigkeit wurde die in Kapitel 5.6 beschriebene Beispielarchitektur 3 anhand der erarbeiteten Gleichungen modelliert und die Ergebnisse der VLSI-Implementierung mit den modellbasierten Kosten verglichen. Tabelle 5.9 zeigt eine exemplarische Gegenüberstellung der Ergebnisse für die Abbildung des FIR-Filters.

	Architektur 3 (VLSI-Impl.)	**Architektur 3 (Modell)**
max. Taktfrequenz	238MHz	295MHz
Verlustleistung bei f_{max}	13,3mW	16,3mW
Fläche A	0,65mm^2	0,6mm^2

Tabelle 5.9: Modellierte Implementierungskosten des FIR-Filters

Die Ergebnisse zeigen, dass die Modellgenauigkeit bezüglich der Fläche eine relativ hohe Genauigkeit mit einem Fehler von unter 10% erreicht. Für die Taktfrequenz und

Verlustleistung ist der Fehler mit ungefähr 25% relativ groß. Dies ist im Wesentlichen auf die Unterschiede im Aufbau der Logikelemente bei der modellierten und der implementierten Architektur zurückzuführen. Bei der in Kapitel 5.6 beschriebenen Architektur ist der zeitkritische Signalpfad sowohl für die Kernlogik als auch für das Register länger. Zudem wurde bei der Implementierung noch eine weniger effiziente Carry-Logik verwendet. Dies führt für die VLSI-Implementierung zu einer niedrigeren Frequenz und einer höheren Verlustleistung. Bei einer entsprechenden Überarbeitung der implementierten Beispielarchitektur wäre somit eine höhere Modellgenauigkeit zu erwarten. Zur Abschätzung des Einflusses von Architekturveränderungen auf die ATE-Kosten (wie am Beispiel des FIR-Filters diskutiert) ist das Modell jedoch ausreichend. Auch eine erste grobe Abschätzung der Effizienz einer Architektur im Vergleich zu existierenden Implementierungen ist mit Hilfe des Modells möglich.

In Abbildung 5.35 ist die Einordnung der in diesem Kapitel diskutierten Ergebnisse bzgl. ihrer Verlustleistungs- und Flächeneffizienz dargestellt. Die drei Architekturvarianten für das FIR-Filter sind mit *I*, *II* und *III* gekennzeichnet. Zusätzlich ist die in Kapitel 5.6 beschriebene eFPGA-Architektur aufgeführt, auf die ein Filter mit identischen Spezifikationen abgebildet wurde. Dabei sind sowohl die Implementierungskosten für die VLSI-Implementierung als auch die modellbasierten Werte dargestellt.

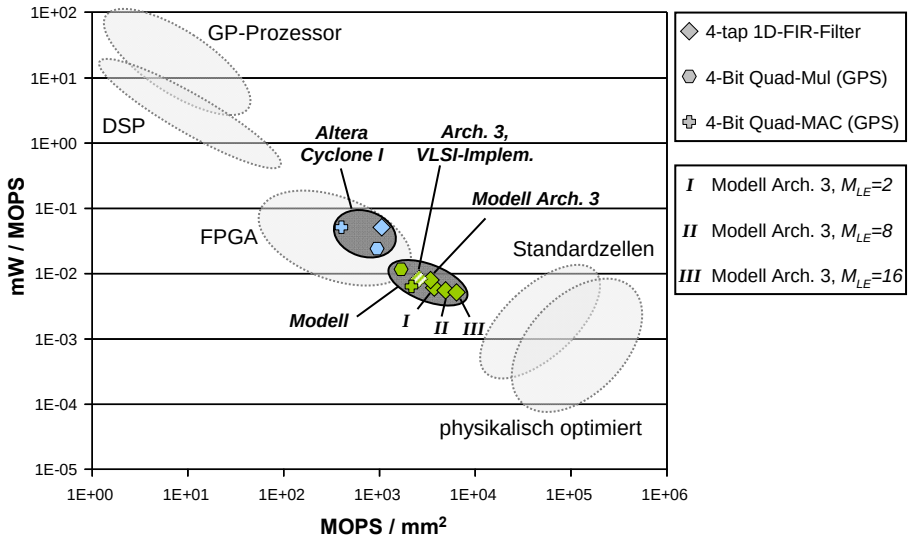


Abbildung 5.35: Einordnung der modellierten eFPGA-Architektur im Entwurfsraum

Die modellierten Ergebnisse entsprechen bezüglich ihrer Verlustleistungs- und Flächeneffizienz den in Kapitel 5.6 simulierten Wertebereichen für die eFPGA-Architektur. Durch die Optimierung der Architektur für das FIR-Filter lässt sich eine deutliche Steigerung der Effizienz feststellen. Insgesamt spiegelt auch das Modell wider, dass sich die Effizienz von feingranularen eFPGA-Architekturen für Arithmetikanwendungen gegenüber Standardarchitekturen um jeweils bis zu Faktor zehn verbessern lässt, wenn die Architektur entsprechend auf die Anwendungsklasse zugeschnitten wird. Der modellbasierte Ansatz

erlaubt es dabei, mit moderatem Aufwand eine Optimierung einzelner Architekturparameter durchzuführen.

Zusammenfassung

Bei der Implementierung von Systemen der digitalen Signalverarbeitung werden eFPGAs u.a. als rekonfigurierbare Acceleratoren für Prozessoren zukünftig eine wichtige Rolle spielen. Sie stellen bezüglich ihrer Flexibilität und der physikalischen Implementierungskosten einen attraktiven Kompromiss zwischen Software-programmierbaren Prozessorkernen und dedizierter Hardware dar. Typischerweise werden solche Acceleratoren benutzt, um datenflussorientierte Operationen vom Prozessor auszulagern und so die Verlustleistungs- und Flächeneffizienz zu steigern. Standard-FPGA-Architekturen sind jedoch für ein sehr breites Spektrum von Anwendungen optimiert, so dass ihre Effizienz für Arithmetikanwendungen begrenzt ist. Durch das gezielte Anpassen einer eFPGA-Architektur an die Anforderungen einer arithmetikorientierten Anwendungsklasse ist es möglich, die Verlustleistungs- und Flächeneffizienz einer solchen Architektur erheblich zu steigern. Um eine methodische Optimierung durchführen zu können, muss eine allgemeine Beschreibungsform der eFPGA-Architektur mit ihren Parametern vorliegen, das so genannte Architektur-Template. Im Rahmen dieser Arbeit wurde ein Template konzipiert, welches die typischen Eigenschaften von Arithmetikdatenpfaden bezüglich Konnektivität und zu implementierender Funktionalität berücksichtigt. Durch Variation der Parameter kann eine gezielte Anpassung an die Anforderungen der zu implementierenden Anwendungen erfolgen. Das Template wurde als abstrakte MATLAB-basierte Beschreibung formuliert. Anhand dieser Beschreibung wurde ein durchgehender Entwurfsablauf von der VLSI-Implementierung bis zur automatischen Erzeugung von Konfigurationsdaten am Beispiel des Routing-Switch demonstriert.

Auf Basis des Architektur-Templates wurden drei exemplarische VLSI-Makros implementiert. Der im Rahmen dieser Arbeit verwendete physikalisch orientierte Entwurfsstil unter Verwendung einer halbautomatischen Layout-Generierung erlaubt es, hochgradig flächeneffiziente Layouts von eFPGA-Makros unter Berücksichtigung der Architekturparameter mit moderatem Entwurfsaufwand anzufertigen. Anhand der Beispielarchitekturen wurde das Optimierungspotenzial von eFPGAs für Arithmetikanwendungen durch Abbilden exemplarischer Basisoperatoren quantitativ demonstriert. Um den Aufwand für das Mapping von Anwendungen auf das eFPGA zu reduzieren, wurde im Rahmen dieser Arbeit ein modellbasierter Ansatz erarbeitet. Durch Formulierung mathematischer Modellgleichungen der eFPGA-Architektur in Abhängigkeit von den Architekturparametern ist es möglich, anhand einer manuellen Allokation der Logikelemente die physikalischen Implementierungskosten der abgebildeten Operatoren zu berechnen.

Auf Basis der VLSI-Implementierungen und der modellierten Architekturvarianten wurde gezeigt, dass die Verlustleistungs- und Flächeneffizienz in mW/MOPS bzw. MOPS/mm^2 der hier vorgestellten eFPGA-Architektur für Arithmetikanwendungen bis zu zehnmal größer ist als bei einem kommerziellen Standard-FPGA.

Anhang

A Beispielarchitektur: Altera Stratix III

Als Beispiel für eine aktuelle, kommerzielle FPGA-Architektur wird hier der Altera Stratix III beschrieben. Der FPGA wird in einer 65nm-SRAM-Technologie mit Kupfermetallisierung in allen Metalllagen gefertigt. Zahlreiche weitere Maßnahmen reduzieren die Verlustleistung des Bausteins auf technologischer Ebene. Dazu gehören neben der Verwendung von Low-k-Dielektrika auch Multi-Threshold-Transistoren mit variablen Gate-Längen, Strained-Silicon und die Verwendung verschiedener Oxiddicken [3].

Das Logikelement des Stratix III wird als ALM (Adaptive Logic Module) bezeichnet. Es besteht im Wesentlichen aus zwei LUT4 und vier LUT3 sowie zwei Registern. Zusätzlich verfügt es über zwei dedizierte Volladdierer mit einer Carry-Chain. Die Lookup-Tables lassen sich über Multiplexer zu einer LUT6 verschalten. Abbildung A1 zeigt den Aufbau eines ALM gemäß [2]. Der größte verfügbare FPGA-Baustein der Stratix-Serie (EP3SL340) verfügt über 135.000 ALMs.

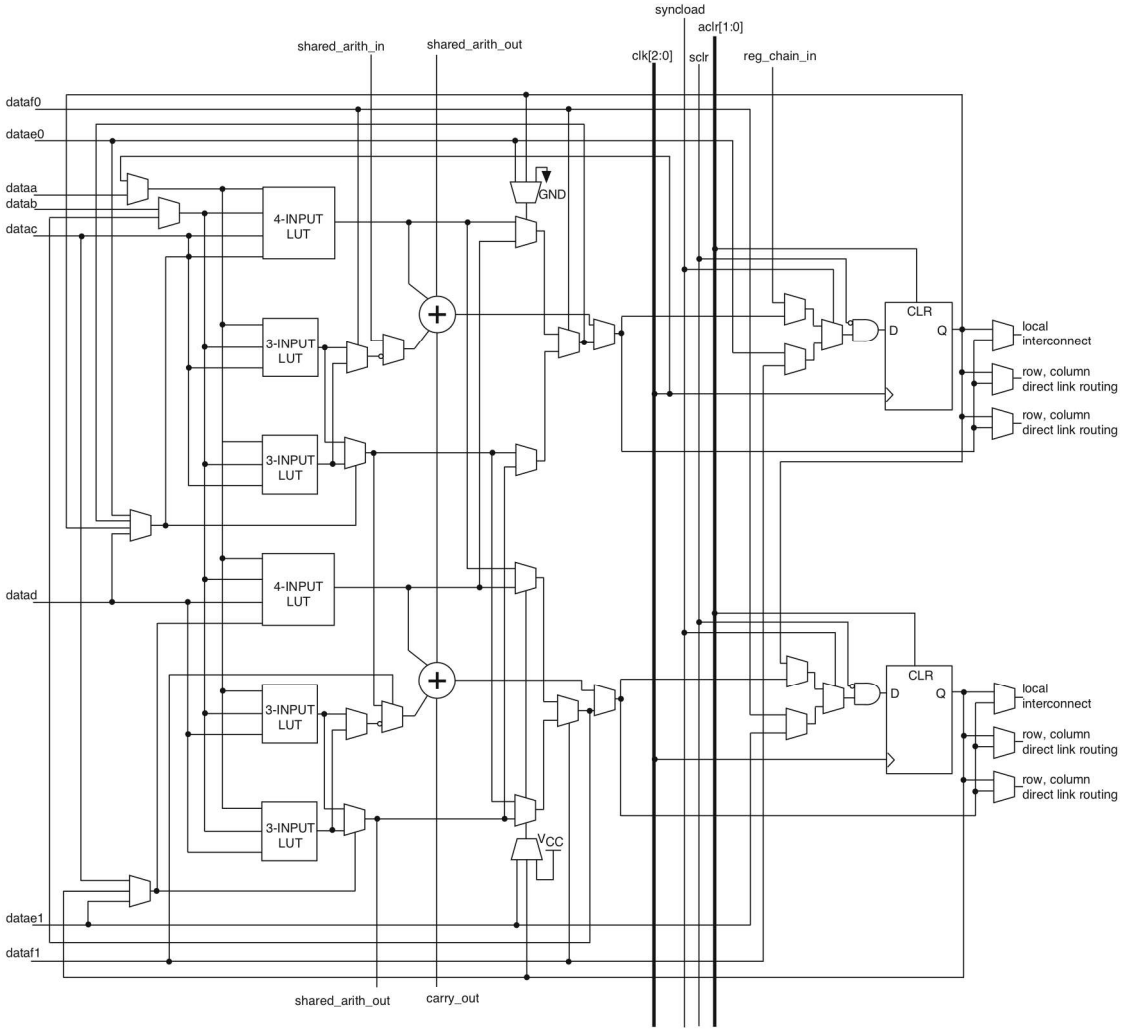


Abbildung A1: ALM des Stratix III [2]

Zehn dieser ALMs werden zu Clustern zusammengefasst, die beim Stratix III als LAB (Logic Array Block) bezeichnet werden. Die LABs sind über ein segmentiertes Verbindungsnetz miteinander verbunden. Zusätzlich existieren lokale Verbindungen, über welche die ALMs direkt verbunden sind. Abbildung A2 zeigt schematisch den Aufbau des Verbindungsnetzes gemäß [2]. Eine weiterführende Beschreibung am Beispiel des ähnlich aufgebauten Vorgängerprodukts Stratix II kann der Literatur entnommen werden [36].

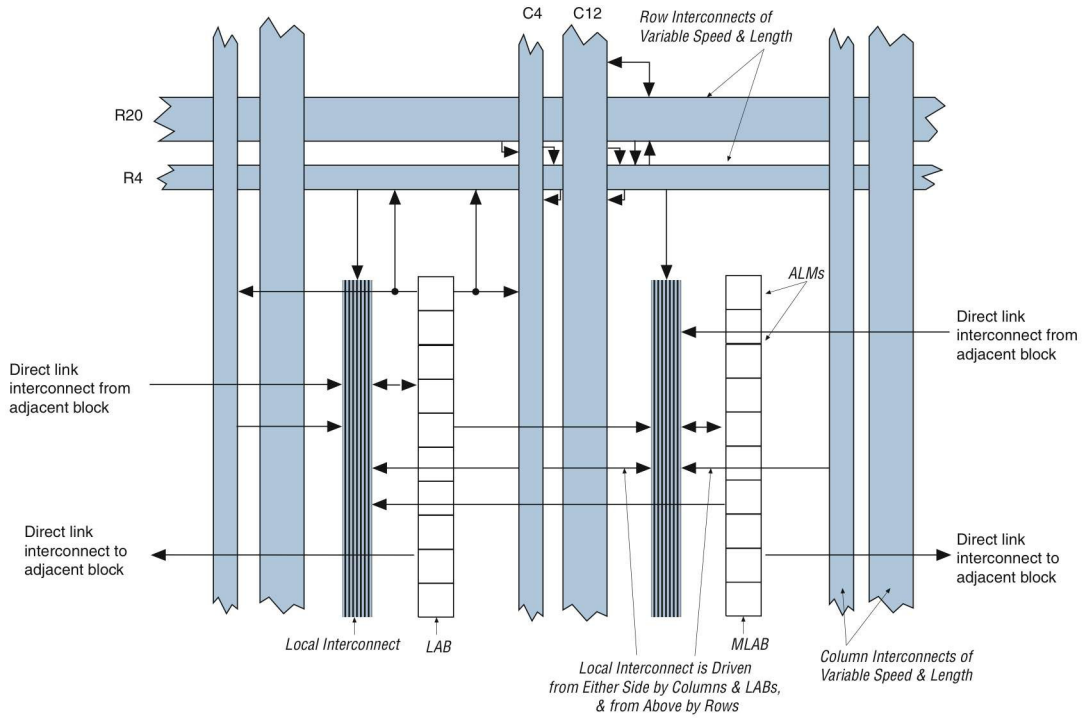


Abbildung A2: Verbindungsnetz des Stratix III [2]

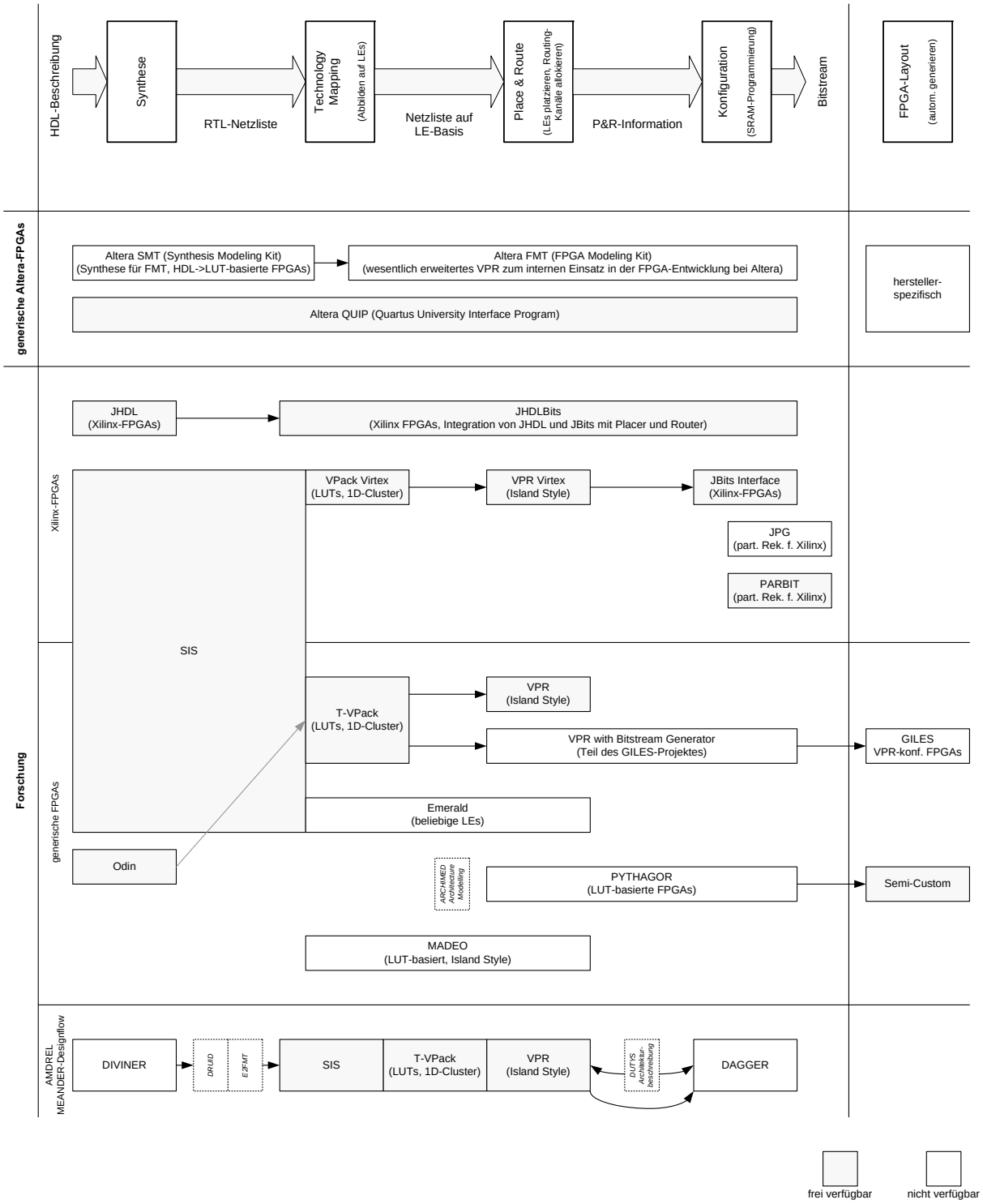
Neben den ALMs verfügt der Stratix III über eingebettete Speicherblöcke in unterschiedlichen Organisationsformen. Es stehen sowohl dedizierte Blöcke mit bis zu 144 kBit dual-port SRAMs als auch in LABs organisierte Speicher (so genannte Memory-LABs, MLABs) mit jeweils 640 Bit dual-port SRAMs zur Verfügung. Insgesamt verfügt der EP3SL340 über 20.491 kBit eingebetteten Speicher.

Zur effizienten Abbildung von Operationen der digitalen Signalverarbeitung wie z.B. Filtern verfügt der Stratix III über spezielle DSP-Blöcke mit insgesamt bis zu 896 dedizierten 18·18-Bit-Multiplizierern bei der so genannten „Stratix Enhanced Family“. Diese Bausteine sind speziell für Anwendungen der digitalen Signalverarbeitung ausgelegt und verfügen über weniger allgemeine Logikressourcen in Form von ALMs, jedoch über mehr Speicher- und Multipliziererblöcke.

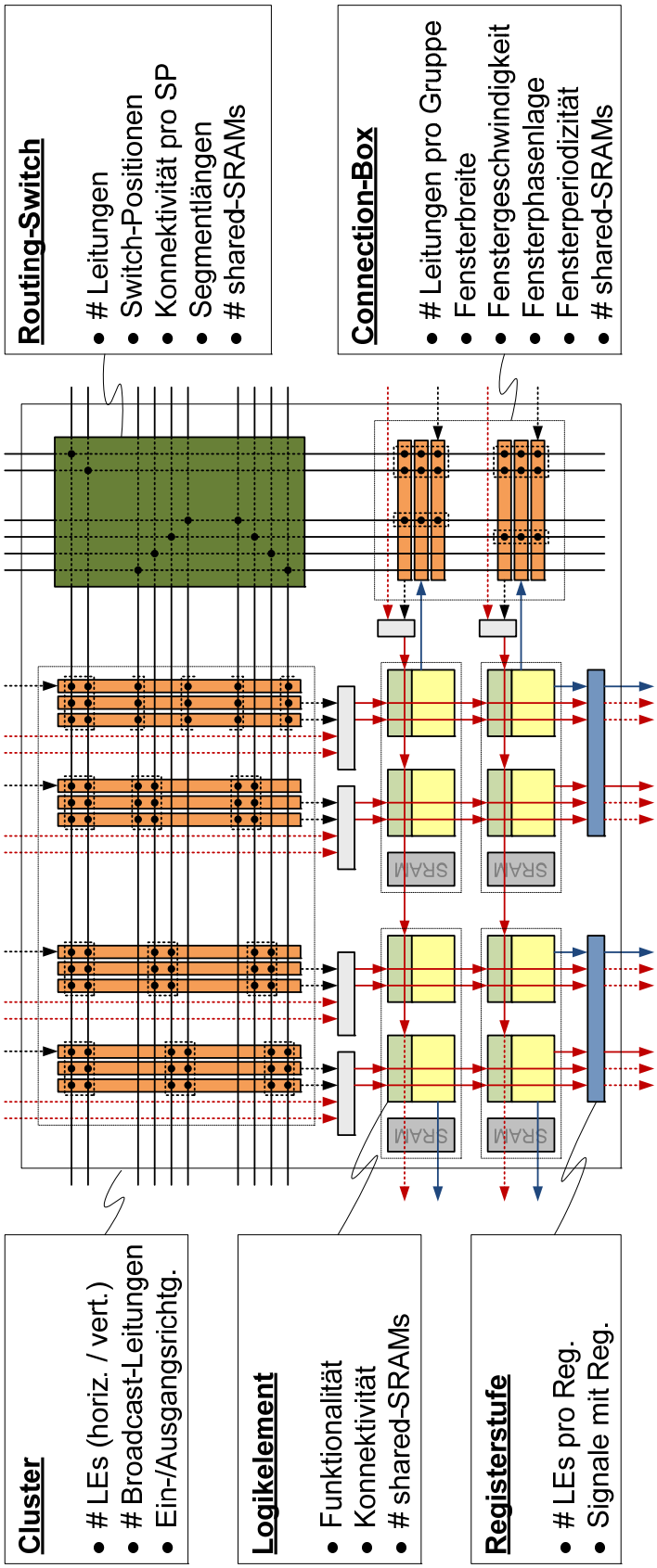
Neben den oben genannten Architekturelementen verfügt der Stratix III über einige spezielle Komponenten zur Kommunikation wie z.B. externe Speicherschnittstellen, differenzielle

I/O-Schnittstellen mit bis zu 1,25 Gbps und bis zu zwölf PLLs (Phase-Locked-Loops) für das Taktnetz.

B Übersicht FPGA-Entwurfswerkzeuge (Forschung)

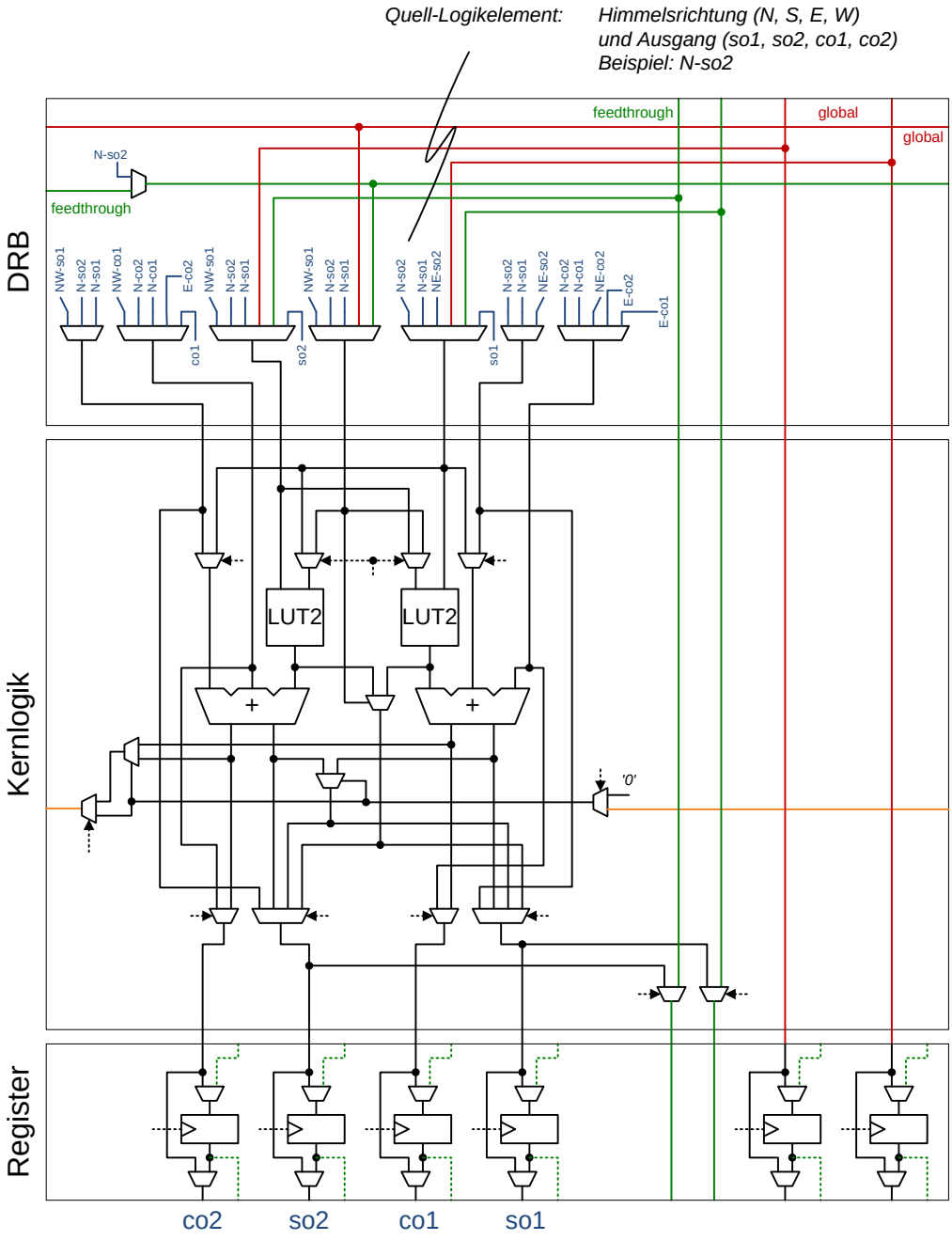


C Architekturparameter des Templates



Architektur-element	**Parameter**	**Bereich**	**Beschreibung**
Cluster (Kap. 4.2.2)	C_H	$[1..64]$	Anzahl LEs pro Cluster in horizontaler Richtung
	C_V	$[1..64]$	Anzahl LEs pro Cluster in vertikaler Richtung
	I_N	$[0..4]$	Anzahl Broadcast-Leitungen von Norden
	I_W	$[0..4]$	Anzahl Broadcast-Leitungen von Westen
	I_S	$[0..4]$	Anzahl Broadcast-Leitungen von Süden
	I_E	$[0..4]$	Anzahl Broadcast-Leitungen von Osten
Logikelement (Kap. 4.2.3)	D_{CL}	$\in \{N,W,S,E\}$	Richtungen der Ausgangssignale
	I_{LE}	$[2..8]$	Anzahl Eingänge der Kernlogik
	O_{LE}	$[1..4]$	Anzahl Ausgänge der Kernlogik
	F_{LE}		Liste mit Funktionen
	DRB_i	$DRB_i=\{t,x,y,q\}$ *mit* $t\in\{lo,bc\}$ $x,y=[0..32]$ $q=[1..O_{LE}]$	Liste aller Quellen des DRB
	N_{Reg}	$[0..C_V]$	Anzahl der LEs pro Register (spaltenweise), =0 für Architektur ohne Register
	O_{Reg}	$\{1..O_{LE} / 1..I_{N/S}\}$	Ausgänge mit Register (lokal und Broadcast)
Routing-Switch (Kap. 4.2.4)	W_H	$[1..256]$	Anzahl horizontaler Verbindungsleitungen
	W_V	$[1..256]$	Anzahl vertikaler Verbindungsleitungen
	$L_{H,i}$	$[1..256]$	Segmentlängen der horizontalen Leitungen
	$L_{V,j}$	$[1..256]$	Segmentlängen der vertikalen Leitungen
	$P_{RS,k}$	$P_{RS,k}=\{x,y\}$ *mit* $x=[1..W_H]$ $y=[1..W_V]$	Positionen der Switch-Points
	SP_k	$SP_k=\{q_1\rightarrow s_{1,i} / q_2\rightarrow s_{2,j}\}$ $q_i,s_i\in\{N,W,S,E\}$ *mit* $i,j=[1..3]$	konfigurierbare Verbindungswege pro Switch-Point {Quelle → Senken}
Connection-Box (Kap. 4.2.5)	$r_{NC,H/V}$	$[0..256]$	Anzahl Leitungen ohne Konnektivität
	$r_{FC,H/V}$	$[0..256]$	Anzahl Leitungen mit voller Konnektivität
	$r_{PC,H/V,i}$	$[0..256]$	Anzahl Leitungen mit periodischer Konnektivität
	$w_{PC,H/V,i}$	$\in\{1..r_{PC,H/V,i}\}$	Fensterbreiten der periodischen Kanäle
	$v_{PC,H/V,i}$	$[1..r_{PC,H/V,i}-1]$	Geschwindigkeit der Fenster
	$p_{PC,H/V,i}$	$[1..r_{PC,H/V,i}]$	Periodizität der Fenster
Konfigura-tionsspeicher (Kap. 4.2.6)	M_{LE}	$[1..C_H]$	Anzahl Logikelemente mit identischen SRAMs
	M_{RS}	$[1..64]$	Anzahl Switch-Points mit identischen SRAMs
	M_{CB}	$[1..64]$	Anzahl progr. Schalter der Connection-Box mit identischen SRAMs
Sonstiges (Kap. 4.2.7)	S_H	$[C_H..256\cdot C_H]$	Anzahl LEs (gesamt) in horizontaler Richtung
	S_V	$[C_V..256\cdot C_V]$	Anzahl LEs (gesamt) in vertikaler Richtung

D Aufbau und Konnektivität Logikelement Beispielarchitektur 3



E Skalierungsregeln

Skalierungsfaktor

$$s = \frac{\lambda_{ref}}{\lambda}$$

Skalierung einer Technologie mit der Strukturfeinheit λ_{ref} auf eine Technologie mit der Strukturfeinheit λ

Grundlegende Größen

Spannungen $V \sim \frac{1}{s}$

Abmessungen $W \sim \frac{1}{s}$ $L \sim \frac{1}{s}$ $t_{Ox} \sim \frac{1}{s}$

Kapazitäten $C \sim \frac{1}{s}$

Fläche A

$A \sim W \cdot L$ => $A \sim \frac{1}{s^2}$

Gatterlaufzeit T

$I_{D,sat} = v_{\max} \cdot \frac{\varepsilon_{Ox}}{t_{Ox}} \cdot W \cdot (V_{GS} - V_T)^{\alpha}$ mit $\alpha = 1...2$ => $I \sim \frac{1}{s}$ (für $\alpha = 1$)

$$I = \frac{Q}{T} = \frac{C \cdot V}{T}$$

$T = \frac{C \cdot V}{I}$ => $T \sim \frac{1}{s}$

Verlustleistung P

$P \approx \sigma \cdot f \cdot V_{DD}^2 \cdot C$ mit $f = \frac{1}{T}$, Schaltaktivität σ

$P \approx \sigma \cdot f \cdot V_{DD}^2 \cdot C$ => $P \sim \frac{1}{s^2}$ (bei gleichzeitiger Skalierung von *T*)

Skalierungsfaktor für geänderte Versorgungsspannungen

$$\tilde{s} = \frac{V_{ref}}{V}$$

Anheben oder Absenken der nominalen Referenzspannung für eine Technologie V_{ref} auf die Spannung V

Einfluss der Versorgungsspannung auf Fläche, Laufzeiten und Verlustleistung

$A \sim 1$

$T \sim 1$ (für $\alpha = 1$)

$P \sim \tilde{s}^2$

Weitere Informationen zu Skalierungsregeln können z.B. [66] entnommen werden.

Literatur

[1] Ahmed, E.; Rose, J.: „The Effect of LUT and Cluster Size on Deep-Submicron FPGA Performance and Density“, IEEE Trans. Very Large Scale Integration (VLSI) Systems, Vol. 12, Nr. 3, S. 288-298, 2004

[2] Altera Corporation: „Stratix III Device Handbook“

[3] Altera Corporation: „Stratix III Programmable Power“, Whitepaper

[4] Altera Corporation: http://www.altera.com

[5] Altera Corporation: Quartus II University Interface Program (QUIP), http://www.altera.com/education/univ/research/unv-quip.html

[6] Altera Corporation: Quartus II Development Software Handbook Version 7.1, Vol. 2 Kap. 9-2, Mai 2007

[7] Anderson, J. H.; Najm, F. N.: „A Novel Low-Power FPGA Routing Switch“, Proc. IEEE Custom Integrated Circuits Conference (CICC ’04), S. 719-722, 2004

[8] Betz, V.; Rose, J.: „FPGA Routing Architecture: Segmentation and Buffering to Optimize Speed and Density“, Proc. ACM/SIGDA 7th International Symposium on Field Programmable Gate Arrays (FPGA ’99), S. 59-68, 1999

[9] Betz, V.; Rose, J.: „VPR: A New Packing, Placement and Routing Tool for FPGA Research“, Proc. 7th International Workshop on Field-Programmable Logic and Applications (FPL ’97), S. 213-222, 1997

[10] Betz, V.; Rose, J.; Marquardt, A.: „Architecture and CAD for Deep-Submicron FPGAs”, Kluwer Academic Publishers, 1999

[11] Blume, H.; Becker, D.; Rotenberg, L.; Botteck, M.; Brakensiek, J.; Noll, T. G.: „Hybrid Functional and Instruction-Level Power Modeling for Embedded and Heterogeneous Processor Architectures“, Journal of Systems Architecture, Vol. 53, Nr. 10, S. 689-702, 2007

[12] Blume, H.; Feldkämper, H.; Noll, T. G.: „Model-based Exploration of the Design Space for Heterogeneous Systems-on-Chip“, Journal of VLSI-Signal Processing, Vol. 40, Nr. 1, S. 19-34, 2005

[13] Borgatti, M.; Lertora, F.; Forêt, B.; Calì, L.: „A Reconfigurable System featuring Dynamically Extensible Embedded Microprocessor, FPGA and Customisable I/O“, IEEE Journal of Solid-State Circuits, Vol. 38, Nr. 3, S. 521-529, 2003

[14] Borre, K.; Akos, D. M.; Bertelsen, N.; Rinder, P.; Jensen, S. H.: „A Software-Defined GPS and Galileo Receiver: A Single-Frequency Approach (Applied and Numerical Harmonic Analysis)“, Birkhäuser Boston, 2007, ISBN 0-8176-4390-7

[15] Brooks, D.; Tiwari, V.; Martonosi, M.: „Wattch: A Framework for Architectural-Level Power Analysis and Optimizations“, 27th International Symposium on Computer Architecture, S. 83-94, 2000

[16] Carter, W. S.: „Configurable Logic Element“, US Patent Nr. 4705216, 1987

[17] Coenen, T.; Schleifer, J.; Weiß, O.; Noll, T. G.: „Interconnect Routing of Embedded FPGAs Using Standard VLSI Routing Tools“, International Symposium on System-on-Chip, 2010

[18] Cronquist, D.; Franklin, P.; Fisher, C., Figueroa, M.; Ebeling, C.: „Architecture Design of Reconfigurable Pipelined Datapaths“, Proc. 20th anniversary conference on advanced research in VLSI, S. 23-40, 1999

[19] Danilin, A.; Bennebroek, M.; Sawitzki, S.: „A Novel Toolset for the Development of FPGA-like Reconfigurable Logic“, Proc. International Conference on Field Programmable Logic and Applications (FPL ’05)S. 640-43, 2005

[20] Degalahal, V.; Tuan, T.: „Methodology for High Level Estimation of FPGA Power Consumption“, Proc. Design Automation Conference Asia and South Pacific (ASP-DAC ’05), S. 657-660, 2005

[21] Enzler, R.: „Architectural Trade-offs in Dynamically Reconfigurable Processors“, Swiss Federal Institute of Technology, Promotion an der ETH Zürich, 2004

[22] Flocke, A.; Blume, H.; Noll, T. G.: „Implementation and modeling of parametrizable high-speed Reed Solomon decoders on FPGAs“, Advances in Radio Science, Vol. 3, S. 271-276, 2005

[23] George, V.; Rabaey, J. M.: Low-Energy FPGAs – Architecture and Design, Kluwer Academic Publishers, 2001

[24] Gonzalez, R. E.: „Xtensa: A configurable and extensible processor“, IEEE Micro, Vol 20, S. 60-70, 2000

[25] Hassan, Hassan; Anis, Mohab: „Low-Power Design of Nanometer FPGAs: Architecture and Eda“, Morgan Kaufman Publishers, 2009, ISBN 0123744385

[26] Hauck, S.; Fry, T. W.; Hosler, M. M.; Kao, J. P.: „The Chimaera Reconfigurable Functional Unit“, IEEE Trans. Very Large Scale Integration (VLSI) Systems, Vol. 12, S. 206-217, 2004

[27] Jestädt, B.: „Konzeption und Analyse einer anwendungsklassenspezifischen ASIP eFPGA-Architektur für multioperable GPS-Empfänger“, Diplomarbeit am Lehrstuhl für Allgemeine Elektrotechnik und Datenverarbeitungssysteme, RWTH Aachen, Juli 2007

[28] Jestädt, S.: „Modellierung und Analyse anwendungsklassenspezifischer parametrisierbarer eFPGA Architekturen“, Diplomarbeit am Lehrstuhl für Allgemeine Elektrotechnik und Datenverarbeitungssysteme, RWTH Aachen, August 2007

[29] Kafafi, N.; Bozman, K.; Wilton, S. J. E.: „Architectures and algorithms for synthesizable embedded programmable logic cores“, Proc. ACM/SIGDA 11th International Symposium on Field Programmable Gate Arrays (FPGA ’03), S. 3-11, 2003

[30] Kappen, G.; Noll, T. G.: „Application Specific Instruction Processor Based Implementation of a GNSS Receiver on an FPGA“, Proc. IEEE Design, Automation and Test in Europe Conference (DATE ’06), S. 58-63, 2006

[31] Keggenhoff, R.: „Implementierung und Analyse Arithmetik-orientierter eFPGA-Makros auf Basis einer generischen Architekturbeschreibung“, Diplomarbeit am Lehrstuhl für Allgemeine Elektrotechnik und Datenverarbeitungssysteme, Dezember 2005

[32] Kuon, I.; Egier, A.; Rose, J.: „Design, Layout and Verification of an FPGA using Automated Tools“, Proc. ACM/SIGDA 13th International Symposium on Field Programmable Gate Arrays (FPGA ’05), S. 215-226, 2005

[33] Landman, B. S.; Russo, R. L.: „On a Pin Versus Block Relationship For Partitions of Logic Graphs“, IEEE Trans. Computers, Vol. C-20, Nr. 12, S. 1469-1479, 1971

[34] Leijten-Nowak, K.: „Template-Based Embedded Reconfigurable Computing“, Promotion an der Technischen Universität Eindhoven, 2004

[35] Lemieux, G.; Lewis, D.: „Design of Interconnection Networks for Programmable Logic“, Kluwer Academic Publishers, 2004

[36] Lewis, D. et al: „The Stratix II Logic and Routing Architecture“, Proc. ACM/SIGDA 13th International Symposium on Field Programmable Gate Arrays (FPGA '05), 2005

[37] Ling, A. C.; Singh, D. P.; Brown, S. D.: „FPGA Logic Synthesis using Quantified Boolean Satisfiability“, Proc. 7th International Conference on Theory and Applications of Satisfiability Testing (SAT ’05), S. 444-450, Juni 2005

[38] Lodi, A.; Ciccarelli, L.; Guerrieri, R.: „Low Leakage Techniques for FPGAs“, IEEE Journal of Solid-State Circuits, Vol. 41, Nr. 7, S. 1662-1672, 2006

[39] Lodi, A.; Thoma, M.; Campi, F.; Cappelli, A.; Canegallo, R.; Guerrieri, R.: „A VLIW Processor with Reconfigurable Reconfigurable Instruction Set for Embedded Applications“, IEEE Journal of Solid-State Circuits, Vol. 38, Nr. 11, S. 1876-1886, 2003

[40] M2000 Corporation: http://www.m2000.fr

[41] McMurchie, L.; Ebeling, C.: „PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs“, Proc. ACM/SIGDA 3rd International Symposium on Field Programmable Gate Arrays (FPGA ’95), S. 111-117, 1995

[42] MENTA: http://www.menta.fr

[43] Mondal, S.; Memik, O.: „A Low Power FPGA Routing Architecture“, Proc. IEEE International Symposium on Circuits and Systems (ISCAS '05), S. 1222-1225, 2005

[44] Mucci, C.; Campi, F.; Gagliardi, P.; Bocchi, M.: „A Case-Study on Multimedia Applications for the XiRisc Reconfigurable Processor“, Proc. IEEE International Symposium on Circuits and Systems (ISCAS ’06), S. 4859-4862, 2006

[45] Mutoh, S.; Douseki, T.; Matsuya, Y.; Aoki, T.; Sgigematsu, S.; Yamada, J.: „1-V power supply high-speed digital circuit technology with multithreshold-voltage CMOS“, IEEE Journal of Solid-State Circuits, Vol. 30, Nr. 8, S. 847-854, 1995

[46] Neumann, B.: „Analyse und Optimierung der Architekturen von Embedded FPGAs“, Masterarbeit am Lehrstuhl für Allgemeine Elektrotechnik und Datenverarbeitungs-systeme, RWTH Aachen, September 2002

[47] Neumann, B.; Blume, H.; Feldkämper, H,; Noll, T. G.: „Embedded FPGA-Architekturen für Multimedia-Applikationen“, Proc. 10. ITG-Fachtagung "Elektronische Medien", S. 147-152, 2003

[48] Neumann, B.; von Sydow, T.; Blume, H.; Noll, T. G.: „Application Domain Specific Embedded FPGAs for Flexible ISA-Extension of ASIPs“, IEEE Journal of Signal Processing Systems, Vol. 53, S. 129-143, November 2008

[49] Neumann, B.; von Sydow, T.; Blume, H.; Noll, T. G.: „Design and quantitative analysis of parametrisable eFPGA-architectures for arithmetic“, Advances in Radio Science, Vol. 4, S. 251-259, 2006

[50] Neumann, B.; von Sydow, T.; Blume, H.; Noll, T. G.: „Design and Quantitative Analysis of ASIPs with eFPGA-based Accelerators as Flexible ISA-Extension“, Poster PhD-Forum IEEE Design, Automation and Test in Europe Conference (DATE ’07), 2007

[51] Neumann, B.; von Sydow, T.; Blume, H.; Noll, T. G.: „Design flow for embedded FPGAs based on a flexible architecture template“, Proc. IEEE Design, Automation and Test in Europe Conference (DATE ’08), S. 56-61, 2008

[52] Noll, T. G.: „Application Domain Specific Embedded FPGAs for SoC Platforms“, eingeladener Übersichtsvortrag bei der Irish Signals and Systems Conference 2004 (ISSC ’04), Juni 2004

[53] Noll, T. G.: „High Throughput Digital Filters“, VLSI Implementations for Image Communications, P. Pirsch (Editor), Elsevier Publishers, S. 171-211, 1993

[54] Noll, T. G.; von Sydow, T.; Neumann, B.; Schleifer, J.; Coenen, T.; Kappen, G.: „Reconfigurable components for application-specific processor architectures“, in „Dynamically Reconfigurable Systems - Architectures, Design Methods and Applications“, ISBN 978-90-481-3484-7, Dezember 2009

[55] PACT XPP Technologies: http://www.pactxpp.com

[56] Padalia, K.; Fung, R.; Bourgeault, M.; Egier, A.; Rose, J.: „Automatic Transistor and Physical Design of FPGA Tiles From An Architectural Specification“, Proc. ACM/SIGDA 11th International Symposium on Field Programmable Gate Arrays (FPGA ’03), S. 164-172, 2003

[57] Pareto, V.: „Cours d’Economie Politique“, Lausanne, 1896

[58] Philips, S.; Hauck, S.: „Automatic Layout of Domain-Specific Reconfigurable Subsystems for System-on-a-Chip“ Proc. ACM/SIGDA 10th International Symposium on Field Programmable Gate Arrays (FPGA ’02), S. 165-173, 2002

[59] Poon, K. K. W.; Wilton, S. J. E.; Yan, A.: „A Detailed Power Model for Field-Programmable Gate Arrays“, Trans. ACM Design Automation of Electronic Systems, Vol. 10, Nr. 2, S. 279-302, 2005

[60] Schmit, H.; Chandra, V.: „Layout Techniques for FPGA Switch Blocks“, IEEE Trans. Very Large Scale Integration (VLSI) Systems, Vol. 13, Nr. 1, S. 96-105, 2005

[61] Sentovich, E. M.; Singh, K. J.; Lavagno, L.; Moon, C.; Murgai, R.; Saldanha, A.; Savoj, H.; Stephan, P. R.; Brayton, R. K.; Sangiovanni-Vincentelli, A. L.: „SIS: A System for Sequential Circuit Synthesis“, Technical Report Nr. UCB/ERL M92/4, University of California, Berkeley, 1992

[62] Shang, L.; Kaviani, A. S.; Bathala, K.: „Dynamic Power Consumption in Virtex™-II FPGA Family“, Proc. ACM/SIGDA 10th International Symposium on Field Programmable Gate Arrays (FPGA ’02), S. 157-164, 2002

[63] SimpleScalar: http://www.simplescalar.com

[64] Stretch Corporation.: http://www.stretchinc.com

[65] Tensilica Corporation: http://www.tensilica.com

[66] Veendrick, H.: „Deep-Submicron CMOS ICs: From Basics to ASICs“, Kluwer Academic Publishers, 2000, ISBN 90-440-01116

[67] von Sydow, T.: „Modellbildung und Analyse heterogener ASIP-eFPGA-Architekturen“, Dissertation am Lehrstuhl für Allgemeine Elektrotechnik und Datenverarbeitungssysteme, RWTH Aachen, 2010

[68] von Sydow, T.; Korb, M.; Neumann, B.; Blume, H.; Noll, T. G.: „Modelling and Quantitative Analysis of Coupling Mechanisms of Programmable Processor Cores and Arithmetic Oriented eFPGA-macros“, Proc. Reconfigurable Computing and FPGAs Conference (ReConFig ’06), S. 252-261, 2006

[69] von Sydow, T.; Neumann, B.; Blume, H.; Noll, T. G.: „Quantitative Analysis of embedded FPGA Architectures for Arithmetic”, Proc. 17th Application-Specific Systems, Architectures and Processors Conference (ASAP ’06), S. 125-131, 2006

[70] Weiss, O.; Gansen, M.; Noll, T. G.: „A flexible Datapath Generator for Physical Oriented Design“, Proc. 27th European Solid State Circuits Conference (ESSCIRC ’01), S. 408-411, 2001

[71] Wilton, S. J. E.; Ho, C. H.; Leong, P. H. W.; Luk, W.; Quinton, B.: „A Synthesizable Datapath-Oriented Embedded FPGA Fabric“, Proc. ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays (FPGA ’07), S. 33-41, 2007

[72] Xilinx Corporation: http://www.xilinx.com

[73] Ye, A.; Rose, J.: „Using Bus-Based Connections to Improve Field-Programmable Gate Array Density Implementing Datapath Circuits“, IEEE Trans. Very Large Scale Integration (VLSI) Systems, Vol. 14, Nr. 5, S. 462-473, 2006

[74] Zhong, Y.: „Implementierung und automatisierte Konfiguration von generischen eFPGA-Routing-Switch-Architekturen“, Masterarbeit am Lehrstuhl für Allgemeine Elektrotechnik und Datenverarbeitungssysteme, RWTH Aachen, Juli 2007

Lebenslauf

Persönliche Daten

Geburtsdatum	12. Mai 1976
Geburtsort	Kerpen

Schulausbildung

08/1982 - 06/1995	Grundschule Mutter Teresa, Burgau-Gymnasium und Gymnasium am Wirteltor Düren Abschluss: Allgemeine Hochschulreife

Wehrdienst

07/1995 - 04/1996	Fernmeldebataillon Gerolstein und Jägerbataillon Düren

Studium

09/1996 - 08/1999	Elektrotechnik - Fachhochschule Aachen, Abteilung Jülich, Fachrichtung Automatisierungstechnik/Mikrosystemtechnik Abschluss: Dipl.-Ing. (FH)
09/1999 - 06/2000	Electrical and Electronic Engineering - Coventry University, England Abschluss: B. Eng.
09/2000 - 09/2002	Elektrotechnik - RWTH Aachen, Fachrichtung Computer Engineering Abschluss: M. Sc.

Beruflicher Werdegang

08/2000 - 09/2000	Wiss. Hilfskraft bei Sihl GmbH, Düren
09/2001 - 12/2001	Stud. Hilfskraft am Lehrstuhl für Werkstoffe der Elektrotechnik II der RWTH Aachen
01/2002 - 05/2002	Stud. Hilfskraft am Lehrstuhl für Allgemeine Elektrotechnik und Datenverarbeitungssysteme der RWTH Aachen
10/2002 – 12/2007	Wiss. Angestellter am Lehrstuhl für Allgemeine Elektrotechnik und Datenverarbeitungssysteme der RWTH Aachen
seit 06/2008	Mitarbeiter der Robert Bosch GmbH im Bereich Automotive Electronics