FPGA-Design und deren Einsatz in Weltraumanwendungen

Masterarbeit

durchgeführt von

Reinhard Zeif
(r.zeif@student.tugraz.at)

Institut für Kommunikationsnetze und
Satellitenkommunikation
der Technischen Universität Graz

Begutachter: Univ.-Prof. Dipl.-Ing. Dr. O. Koudelka

Graz, am 11. April 2011

Betreuerin: Dipl. Ing. (FH) Dipl. Ing. Manuela Unterberger

Dankschreiben

Vorweg möchte ich Univ.-Prof. Dipl.-Ing. Dr. Otto Koudelka für die Möglichkeit danken, dass ich meine Diplomarbeit am Institut für Kommunikationsnetze und Satellitenkommunikation durchführen konnte. Ein großes Dankeschön geht an Manuela Unterberger für die vorzügliche Betreuung und der ständigen Bereitschaft mir mit Rat und Tat zur Seite zu stehen. Ohne ihre Hilfe wäre mir die Diplomarbeit in dieser Form nicht möglich gewesen.

In besonderer Weise möchte ich an dieser Stelle auch meinen Eltern danken. Sie haben mir bereits in frühen Jahren einen Sinn für die nötige Bildung vermittelt und mir alle Türen offen gehalten. Zudem haben sie mich stets in allen Phasen meines Studiums tatkräftig, teilweise bis zu den Grenzen ihrer Möglichkeiten, unterstützt. Ohne ihr Verständnis, ihr Vertrauen, den ab und zu nötigen Ansporn und ihre Sicht der Dinge hätte ich es bestimmt nicht so weit gebracht.

Ein besonderer Dank geht an meine Freundin Marina, die mich bei meinem Studium stets unterstützt und in den anstrengensten Phasen für die nötige Ablenkung oder Aufmunterung gesorgt hat. Vor allem danke ich ihr dafür, dass sie während meiner tagelangen, wenn nicht gleich wochenlangen, ununterbrochenen Beschäftigung mit der Diplomarbeit soviel Geduld aufgebracht hat. Manchmal musste sie ihre persönlichen Interessen, wegen meiner vielen Arbeit, hinten anstellen. All zu oft blieb nur sehr wenig Zeit für gemeinsame Aktivitäten. Für das Verständnis und Entgegenkommen, stehe ich sehr in ihrer Schuld.

Abschließend möchte ich denjenigen danken, die mich im Laufe meines Studium auf meinem Weg begleitet haben und für mich da waren.

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am …………………………… ………………………………………………..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…………………………… ………………………………………………..
date (signature)

Zusammenfassung

In dieser Masterarbeit werden die grundlegenden Eigenschaften von FPGAs und deren möglichen Einsatz bei Weltraumanwendungen beschrieben. Nachfolgend sind die Schwerpunkte dieser Masterarbeit zusammengefasst.

FPGAs bestehen aus vorgefertigten Logikkomponenten mit konfigurierbarer Funktion und Verdrahtung. Es gibt drei Arten von Konfigurationszellen zum Speichern der Konfiguration. SRAM- und Flash-Zellen können mehrmals programmiert werden, Antifuse-Zellen hingegen nur einmal. SRAM-Zellen müssen nach dem Ausschalten neu konfiguriert werden. Flash- und Antifuse-Zellen behalten danach ihre Konfiguration.
Für die Entwicklung werden häufig iterative Entwurfsmodelle mit verschiedenen Abstraktionsebenen und Entwurfsphasen verwendet. Die Ergebnisse der verschiedenen Phasen werden ständig überprüft und bei Fehlern überarbeitet. Die Implementierung der Schaltung erfolgt in der Praxis bevorzugt auf der RTL-Ebene. Durch Synthese kann die Schaltung automatisch auf die tieferen Abstraktionsebenen transformiert werden. Das fertige Hardware-Layout wird optimiert, geprüft, analysiert und als Konfigurationsdatei in den FPGA geschrieben.
FPGAs sind in einer extraterrestrischen Umgebung ständig der kosmischen Strahlung ausgesetzt. Die negativen Auswirkungen werden in langfristige TID-Effekte und kurzfristige SEEs unterteilt. SRAM-Zellen reagieren sehr empfindlich auf die Strahlung, wodurch es zu Störungen oder Ausfällen kommt. Flash- und Antifuse-Zellen werden viel weniger bzw. gar nicht beeinflusst. Spezielle Techniken, wie Readback und Scrubbing, ermöglichen die Erkennung oder Behebung der Störungen. Redundante Hardware oder Module verringern das Ausfallsrisiko erheblich.
Ein Vergleich zeigt, dass die Flash- und Antifuse-basierten FPGAs von Actel™ sehr gute Eigenschaften besitzen. Der Grund dafür liegt in der speziellen Architektur mit redundanten R- und C-Zellen. Die SRAM-basierten FPGAs von Xilinx™ sind weniger vor Strahlung geschützt, enthalten aber zahlreiche Komponenten zur Detektion oder Behebung von SEEs. Die FPGAs von Altera™ sind diesbezüglich schlechter ausgestattet.
Für die Programmierung der FPGAs wird bevorzugt VHDL oder Verilog-HDL verwendet. Eingebettete Prozessorkerne werden fast immer mit C oder C++ programmiert.

Abstract

This master thesis describes the fundamential properties of FPGAs and their possible use in space applications. In the following the key aspects of this master thesis are summerized.

FPGAs are made up of prefabricated logic components with configurable function and wiring. There are three types of configuration cells to store the configuration. SRAM and flash cells can be programmed several times, antifuse cells however only once. SRAM cells have to be reconfigured after a shutdown. Flash and antifuse cells keep their configuration.
For the design and development iterative design models are often used with different levels of abstraction and design phases. The results of the different phases are periodically verified and corrected in case of an error. In practice the implementation of the circuit takes place on the RTL level. By synthesis the circuit can be automatically transformed to an underlying abstraction level. The finished hardware layout is going to be optimized, verified, analyzed and written into the FPGA as configuration file.
FPGAs in an extraterrestrial environment are constantly exposed to the cosmic radiation. The negative consequences are divided into long-term TID effects and short-term SEEs. SRAM cells are very sensitive to radiation, which causes disruption or failure. Flash and antifuse cells are less affected, respectively immune to radiation. Special techniques such as readback and scrubbing are used to detect and eliminate disruptions. Redundant hardware or modules are very powerful to reduce the risk of a failure.
A comparison shows that flash and antifuse based FPGAs from Actel™ have very good properties. The reason for that lies in the special architecture with redundant R- and C-cells. The SRAM based FPGAs from Xilinx™ are less protected to radiation but they contain many components for the detection and elimination of SEEs. The FPGAs from Altera™ are less equipped in this regard.
For the programming of FPGAs is preferably used VHDL or Verilog-HDL. Embedded processor cores are almost always programmed with C or C++.

Inhaltsverzeichnis

1 Aufgabenstellung **21**

2 Application Specific Integrated Circuits **23**
2.1 Zellen basierte semi-custom ASICs (cell based) 24
2.1.1 Standardzellen 24
2.1.2 Macrozellen 24
2.2 Array basierte semi-custom ASICs (array based) 25
2.2.1 Pre-wired Architektur (CPLD, FPGA) 25
2.2.2 Pre-diffused Architektur (Gate Array) 25

3 Field Programmable Gate Arrays **27**
3.1 Konfigurationszellen (Configuration Cells) 28
3.1.1 Antifuse-Zellen 28
3.1.2 SRAM-Zellen 29
3.1.3 EEPROM und Flash-Zellen 29
3.1.4 Zusammenfassung 30
3.2 Logikzellen (Logic Cells) 31
3.3 Configurable Logic Block 31
3.4 Eingebauter RAM (Embedded RAM) 32
3.5 Zusätzliche Komponenten 32
3.6 Clock Tree und Clock Manager 33
3.7 Ein- und Ausgabe von Daten 34

4 Entwurf und Entwicklung integrierter Schaltungen **37**
4.1 Entwurfsmodelle und Abstraktionsebenen 38
4.2 Iterativer und traditioneller Entwurfsprozess 39
4.2.1 Der traditionelle Entwurf 39
4.2.2 Der iterative Entwurf 39
4.3 Typischer iterativer ASIC Entwurfsprozess 40
4.3.1 System Design 42
4.3.2 High-Level Design 42
4.3.3 Architektur Design 43
4.3.4 Software Design 44
4.3.5 Register Transfer Level Design 44
4.3.6 Front-End Design 45
4.3.7 Back-End Design 46
4.3.8 Fabrikation 47
4.3.9 Verpacken 48
4.3.10 Konfiguration 48
4.3.11 System Integration 49
4.4 Verifikation und Analyse 49
4.4.1 Überprüfung der Geometrie und Struktur 49
4.4.2 Verifikation durch Simulation 50

4.4.3 Layout-Überprüfung . . . 51
4.4.4 Vereinfachte Verifikation durch Teststrukturen . . . 51

5 FPGAs in extraterrestrischen Umgebungen **55**
5.1 Auswirkungen erhöhter Strahlung . . . 55
5.2 Beeinflussung durch langanhaltende Strahleneinwirkung . . . 56
5.3 Störungen durch einzelne energiereiche Teilchen . . . 58
5.3.1 Single Event Upset . . . 58
5.3.2 Single Event Latchup . . . 60
5.3.3 Single Event Functional Interrupt . . . 61
5.3.4 Single Event Burnout/Gate Rupture . . . 61
5.4 Vermeidung und Behebung von Single Event Effekten . . . 61
5.4.1 Dreifach redundante Module . . . 62
5.4.2 Redundanz auf Gatter-Ebene . . . 63
5.4.3 Rekonfiguration von FPGAs (Scrubbing) . . . 64
5.4.4 Kombination von TMR und Rekonfiguration . . . 65
5.4.5 Dreifach redundante Hardware . . . 65

6 Strahlungstolerante FPGA-Familien **67**
6.1 Xilinx space-grade FPGA-Familien . . . 67
6.1.1 Die Virtex-II QPro-Familie . . . 68
6.1.2 Die Virtex-4 QV-Familie . . . 73
6.1.3 Die Virtex-5 QV-Familie . . . 82
6.1.4 Zusammenfassung . . . 91
6.2 Actel radiation-hardened FPGA-Familien . . . 92
6.2.1 Die RTAX-S/SL-Familie . . . 92
6.2.2 Die RTAX-DSP-Familie . . . 97
6.2.3 Die RTSX-SU-Familie . . . 99
6.2.4 Die RT ProASIC3-Familie . . . 101
6.2.5 Zusammenfassung . . . 106
6.3 Altera FPGA-Familien . . . 107
6.3.1 Die Stratix-III-Familie . . . 107
6.3.2 Die Stratix-IV-Familie . . . 112
6.4 Zusammenfassung und Empfehlungen . . . 115

7 Programmiersprachen zur Entwicklung von FPGAs **117**
7.1 Grundlegende Einteilung . . . 117
7.2 Very High-Speed Integrated Circuit Hardware Description Language . . . 119
7.2.1 Grundlagen für den Entwurf mit VHDL . . . 119
7.2.2 Zustandsautomaten mit VHDL . . . 122
7.2.3 Realisierung eines TMR-Systemes . . . 126
7.2.4 Testen einzelner Logikkomponenten . . . 129
7.3 Zusammenfassung . . . 133

8 Anhang **135**

Literaturverzeichnis **157**

Abbildungsverzeichnis

2.1 Unterteilung von ASIC Bausteinen. 23
2.2 Merkmale der verschiedenen ASIC-Typen. 24
2.3 Architektur der Xilinx CoolRunner-II CPLD Familie [Xil08a]. 25
2.4 Vereinfachte Darstellung eines Channeled Gate Array. 26

3.1 Vereinfachte Darstellung einer FPGA Architektur [Max04]. 27
3.2 Darstellung einer programmierten Antifuse [Qui08]. 28
3.3 SRAM Speicher bestehend aus sechs Transistoren [Mia07]. 29
3.4 EEPROM Speicher bestehend aus Floating-Gate Transistoren [Mia07]. 30
3.5 Zusammenfassung der Eigenschaften von Konfigurationszellen [Max04]. 30
3.6 Vereinfachte Darstellung einer Logikzelle [Max04]. 31
3.7 Vereinfachte Darstellung eines Xilinx CLB [Max04]. 32
3.8 Eingebauter RAM zwischen den CLBs [Max04]. 32
3.9 Vereinfachter Aufbau eines Clock Trees [Max04]. 33
3.10 Clock Manager zur Korrektur von Jitter [Max04]. 34
3.11 Konfigurierbare I/O-Bänke in einem FPGA [Max04]. 34

4.1 Divide & Conquer Ebenenhierarchie mit jeweils eigener Beschreibung. 37
4.2 Y-Diagramm des Gajski-Kuhn Modells [Gro08]. 38
4.3 Ablauf einer iterativen Entwurfsphase. 40
4.4 Designphasen bei der Entwicklung einer integrierten Schaltung. 41
4.5 Endlicher Zustandsautomat mit Zustands- und Ausgangs-Logik [Tin00]. 43
4.6 Logiksynthese des RTL-Modells auf die Gatter-Ebene. 45
4.7 Schaltung und dazugehöriges Layout eines Inverters [RL08]. 46
4.8 Verdrahtung eines Rohchips mit Bond-Drähten [Her10]. 48
4.9 Zusammenhang der verschiedenen Verifikationsmethoden [JTA08]. 52
4.10 Aufbau eines IC mit eingebauter JTAG/Boundary-Scan Logik [JTA08]. 53

5.1 Ansammlung positiver Ladungsträger im Gate-Oxid [TRO03]. 56
5.2 Pfad des Leckstromes zwischen Source und Drain [TRO03]. 57
5.3 Verschiebung der Eingangskennlinie durch Leckströme [TRO03]. 57
5.4 Übersicht der einzelnen SEE-Effekte [Bau05]. 58
5.5 Auswirkung eines energiereichen Teilchens auf einen N-Kanal-FET [SF04]. 59
5.6 Beispiel für einen Ausfall durch einen Half-Latch [Caf02]. 60
5.7 Übersicht der gängigen Mitigation Techniken [Xil08b]. 62
5.8 Triple Module Redundancy (TMR)-System mit Voter [Hab02]. 62
5.9 Redundanz der Logikkomponenten auf Gatter-Ebene [Hab02]. 63
5.10 Störungskorrektur durch Rückkopplung auf Gatter-Ebene [Hab02]. 63
5.11 Realisierung der Readback- oder Scrubbing-Controller [Xil00]. 64
5.12 Triple Device Redundancy [Xil08b]. 65

6.1 Marktanteile von Altera und Xilinx bei programmierbaren Logikbausteinen [Xil10b]. 67
6.2 Virtex-4 und Virtex-5 FPGAs [Xilinx]. 68

6.3 Architektur der Virtex-2 QPro-FPGAs [Xil06]. 69
6.4 CLB eines Virtex-2 QPro-FPGAs [Xil06]. 70
6.5 Slice eines Virtex-2 QPro-FPGAs [Xil06]. 70
6.6 I/O-Block eines Virtex-2 QPro-FPGAs [Xil06]. 71
6.7 SelectRAM und Multiplizierer Blöcke eines Virtex-2 FPGAs [Xil06]. 72
6.8 Die acht Bereiche für die Taktsignale im XC4VLX15 [Xil08c]. 74
6.9 Anordnung von DCM und PMCD [Xil08c]. 75
6.10 CLB eines Virtex-4 QV-FPGAs [Xil08c]. 75
6.11 Vereinfachter Aufbau eines XtremeDSP Slice [Xil08d]. 76
6.12 Die Auxiliary Processor Unit des PPC405 [Xil10d]. 77
6.13 Struktur eines PowerPC 405 Prozessorkernes [Xil10d]. 78
6.14 Struktur eines Virtex-4 embedded Ethernet MAC Blockes [Xil10g]. 79
6.15 Die zwei Ports eines integrierten Block-RAM [Xil08c]. 80
6.16 Blöcke der Virtex-4 I/O-Beschaltung [Xil08c]. 80
6.17 Schritte bei der Konfiguration eines Virtex-4 FPGAs [Xil09b]. 81
6.18 CLB eines Virtex-5 QV-FPGAs [Xil10i]. 83
6.19 Verbindung der CLB Carry-Leitungen [Xil10i]. 84
6.20 Detaillierter Aufbau eines DSP48E Slice [Xil10j]. 84
6.21 Aufbau eines Virtex-5 CMT [Xil10i]. 85
6.22 Aufbau eines Virtex-5 PLL [Xil10i]. 86
6.23 Simple Dual-Port RAM mit integrierter Fehlerkorrektur [Xil10i]. 87
6.24 Grundlegender Aufbau eines Multi-Gigabit Transceivers [AC05]. 88
6.25 Aufbau eines RocketIO GTX Transceiver Sender-Blocks [Xil09c]. 88
6.26 Blöcke der Virtex-5 I/O-Beschaltung [Xil10i]. 89
6.27 Verifikation der Readback-Daten mit BIT-Datei [Xil10h]. 90
6.28 Sea-of-Modules-Architektur der Actel-FPGAs [Act10d]. 92
6.29 Architektur eines RTAX-FPGAs [Act10d]. 93
6.30 SEU hardened Flip-Flop eines RTAX-FPGAs [Act10d]. 94
6.31 SuperCluster-Block der RTAX-Architektur [Act10d]. 94
6.32 Verbindungsstruktur der RTAX SuperCluster [Act10d]. 95
6.33 Hauptverbindungen zwischen den einzelnen Logikkernen [Act10d]. 95
6.34 Verteilung der verschiedenen Taktsignale [Act10d]. 96
6.35 Aufbau der IO-Blöcke eines RTAX FPGAs [Act10d]. 97
6.36 Architektur eines RTAX-DSP FPGAs [Act10d]. 98
6.37 R- und C-Zelle der RTSX-FPGAs und die SuperCluster-Typen [Act10e]. 100
6.38 Struktur der Quadrant Clocks-Ressourcen [Act10e]. 100
6.39 Verbindungsstrukturen in den RTSX-SU-FPGAs [Act10e]. 101
6.40 Architektur der RT ProASIC3-Familie [Act08]. 102
6.41 Aufbau eines ProASIC3 VersaTile-Blocks [Act08]. 103
6.42 Struktur eines VersaNet-Netzwerkes [Act10b]. 104
6.43 Struktur eines CCC-Blocks mit eingebauter PLL [Act10b]. 105
6.44 UJTAG-Interface und Addressumwandlung für Flash-ROM [Act10b]. 105
6.45 Architektur des Stratix-III [Alt10a]. 108
6.46 Struktur eines ALM-Blockes [Alt10a]. 109
6.47 Aufbau eines Half-DSP-Blocks [Alt10a]. 110
6.48 Konfiguration mittels FPP-Methode [Alt10b]. 111
6.49 Architektur der Stratix-IV GX- und GT-FPGAs [Str10]. 112
6.50 SERDES-Block eines Startix-IV-FPGAs [Alt10b]. 114

7.1 Einteilung verschiedener Programmiersprachen [Gro08]. 118

7.2 Verschiedene Schritte bei der Synthese [Gro08]. 119
7.3 VHDL-Beispiel mit einem Zustandsautomaten. 122
7.4 Simulationsergebnis des kompletten Zustandsautomaten. 125
7.5 Struktur des TMR-Systems. 126
7.6 Simulationsergebnis der kompletten TMR-Ausgangslogik. 131
7.7 Simulationsergebnis für den Voter. 132

Abkürzungsverzeichnis

ADC	Analog-Digital-Umsetzer
AES	Advanced Encryption Standard
ALM	Adaptive Logic Module
ALU	Arithmetic Logic Unit
ALUT	Adaptive Look-up Table
APU	Auxiliary Processor Unit
AS	Active Serial
ASIC	Application Specific Integrated Circuit
ASMBL	Advanced Silicon Modular Block
BIT	Build-In Test
BPI	Byte Peripheral Interface
BS	Boundary-Scan
CDR	Clock Data Recovery
CE	Clock-Enable
CLB	Configurable Logic Block
CLK	Clock, Taktsignal
CMOS	Complementary Metal Oxide Semiconductor
CMT	Clock Management Tile
CP	Charge Pump
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
CRAM	Configuration Random Access Memory
CRC	Cyclic Redundancy Check
CTD	Clock Tile Distribution
DCI	Digital Controlled Impedance
DCM	Digital Clock Manager
DDD	Detailed Design Document
DDR	Double Data Rate
DES	Data Encryption Standard
D-FF	Data Flip-Flop
DPA	Dynamic Phase Alignment, Dynamic Phase Aligner
DRC	Design Rule Check
DRP	Dynamic Reconfiguration Port
DSP	Digital Signal Processing
ECC	Error Correction Code
EDAC	Error Detection And Correction
EEPROM	Electrically Erasable Programmable Read-Only Memory
EPC	Embedded Processor Core
EPDS	Electrical Power and Distribution System
ERC	Electrical Rule Check
FCM	Fabric Co-processor Module
FET	Field-Effect Transistor

FF	Flip-Flop
FIFO	First In First Out
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GCB	Global Clock Buffer
GEO	Geostationary Orbit
GMII	Gigabit Media Independent Interface
GN&C	Guidance, Navigation and Control
GPIC	General Purpose Integrated Circuit
GRM	General Routing Matrix
HCLK	Hardwired Clock
HDL	Hardware Description Language
I/O	Input/Output
IC	Integrated Circuit
ICAP	Internal Configuration Access Port
ICT	In-Circuit Test
IEEE	Institute of Electrical and Electronics Engineers
IIR	Infinite Impulse Response
IOB	Input/Output Buffer
ISC	In-System Configurable
ISP	In-System Programmable
JTAG	Jount Test Action Group
LAB	Logic Array Block
LAB	Logic Array Block
LF	Loop Filter
Lst.	Listing, Programmausdruck
LUT	Lookup Table
LVDS	Low Voltage Differential Signaling
LVS	Layout versus Schematic
MAC	Media Access Control
MGT	Multi-Gigabit Transceiver
MII	Media Independent Interface
MLAB	Memory Logic Array Block
MMU	Memory Management Unit
MTBF	Mean Time Before Failure
MUX	Multiplexer
NCC	Network Consistency Check
NS	Next State
OCM	On-Chip Memory
PLA	Programmable Logic Array
PFD	Phase-Frequency Detector
PGA	Pin Grid Array
PLL	Phase-Locked Loop
PMCD	Phase-Matched Clock Divider
ppb	part per billion
PPC	PowerPC Processor Core
PS	Passive Serial, Present State
RAM	Random Access Memory
RCB	Regional Clock Buffer

RCLK	Routed Clock
REM	Raster-Elektronen-Mikroskop
RHBD	Radiation Hardened by Design
RISC	Reduced Instruction Set Computer
ROM	Read-Only Memory
RTL	Register Transfer Layer, Register Transfer Level
RX	Receiver
SEB	Single Event Burnout
SEE	Single Event Effect
SEFI	Single Event Functional Interrupt
SEGR	Single Event Gate Rupture
SEL	Single Event Latchup
SERDES	Serializer-Deserializer
SEU	Single Event Upset
SIMS	Secondary Ion Mass Spectrometry
SoC	System on Chip
SOI	Silicon-On-Insulator
SOM	Sea-Of-Modules
SPI	Serial Peripheral Interface
SPL	Software Programming Language
SR	Schieberegister, Shift Register
SRAM	Static Random Access Memory
SRM	Spreading Resistance Method
ST	Self-Test
TAP	Test Access Port
TEM	Tunnen-Elektronen-Mikroskop
TID	Total Ionization Dose
TLB	Translation Look-aside Buffers
TMR	Triple Module Redundancy
TX	Transmitter
VCO	Voltage Controlled Oscillator
VHDL	Very High Speed Integrated Circuit Hardware Description Language

1 Aufgabenstellung

Die Diplomarbeit soll zu Beginn einen Einblick in die wichtigsten Merkmale und Arten von *Application Specific Integrated Circuits (ASICs)* geben. Daraus soll ersichtlich sein, was die wichtigsten Arten von ASICs sind und welcher Gruppe die *Field Programmable Gate Arrays (FPGAs)* zugeordnet werden.
Darauf aufbauend können die verschiedenen Typen von FPGAs und deren grundlegende Architektur beschrieben werden. Die unterschiedlichen Technologien sind anhand ihrer Eigenschaften zu vergleichen.

Auf Grund der Tatsache, dass integrierte Schaltungen in einer extraterrestrischen Umgebung anderen Einflüssen als auf der Erde ausgesetzt sind, sollen die damit verbundenen negativen Effekte und Auswirkungen erläutert werden. Die wichtigsten Effekte, deren Ursachen und die damit verbundenen Störungen, sind in einfacher Form zu erklären. Mögliche Maßnahmen zum Schutz der FPGAs und deren implementierten Schaltungen, vor den Auswirkungen der kosmischen Strahlung, sind zu beschreiben. Dabei sind besonders diejenigen Verfahren zu erwähnen, welche die Detektion oder Behebung der Störungen oder Ausfälle ermöglichen.

Weiters soll auf die wichtigsten Prinzipien für den Entwurf und die Simulation von FPGAs eingegangen werden. Die genauere Beschreibung eines iterativen ASIC-Entwurfsprozesses soll Aufschluss über die wesentlichen Abläufe bei der Entwicklung geben. Die einzelnen Phasen des Entwurfes und deren Ergebnisse sind zu erläutern.
Ein weiteres Ziel ist der Vergleich und die Beschreibung einiger bekannter FPGA-Familien. Dabei ist auf den Funktionsumfang und die grundlegende Architektur einzugehen. Anhand dieser Informationen sollen Aussagen über die Verwendbarkeit und Leistungsfähigkeit der verschiedenen FPGAs getroffen werden. Zudem ist deren Eignung für den Einsatz in extraterrestrischen Umgebungen zu erläutern.

Abschließend sollen die verschiedenen Programmiersprachen beschrieben werden, welche für FPGAs relevant sind. Dabei wird das Hauptaugenmerk auf die VHDL-Programmiersprache gelegt. Zur Verdeutlichung sind einige kurze Beispiele zu erarbeiten, welche eine Einführung in die Programmierung einfacher Schaltungen geben.

2 Application Specific Integrated Circuits

Die Abkürzung *ASIC* steht für *Application Specific Integrated Circuit*, was soviel heißt wie *anwendungsspezifische integrierte Schaltung*. Darunter versteht man eigens für spezielle Anwendungen entwickelte integrierten Schaltungen (ICs). Die Unterteilung von ASIC Bausteinen ist aus Abb. 2.1 ersichtlich.

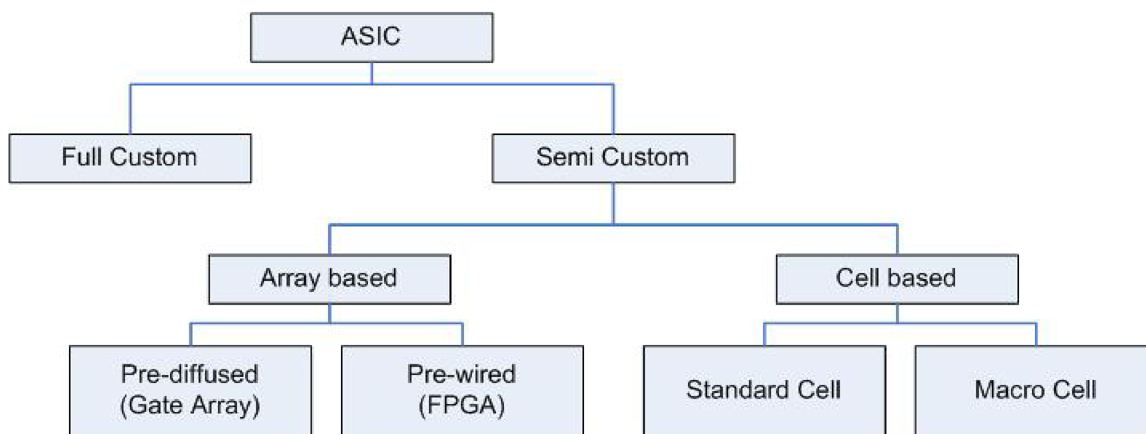

Abbildung 2.1: Unterteilung von ASIC Bausteinen.

Bei ASICs wird grundsätzlich zwischen den sogenannten *full custom* und *semi custom* Bausteinen unterschieden.
Ein full custom ASIC wird bei der Entwicklung komplett von Grund auf entworfen und anschließend gefertigt. Die Fertigung erfolgt in einzelnen Layern aus denen der IC schichtweise aufgebaut ist. Der Vorteil dabei ist, dass sehr aufwändige Layouts möglich sind und jeder Schaltungsteil speziell optimiert werden kann. Entsprechend können hohe Anforderungen, wie z.B. kleine Chipflächen und eine hohe Ausbeute, erreicht werden. Ein weiterer Vorteil ist, dass in einer einzigen integrierten Schaltung mehrere unterschiedliche Komponenten vereint werden können. Dadurch entstehen sehr komplexe Systeme auf einem Chip, welche man als *System on Chip (SoC)* bezeichnet.
Der Nachteil von full custom ASICs liegt darin, dass bei der Fertigung viele Prozessschritte nötig sind und dadurch hohe Kosten entstehen. Zudem ist die Entwicklung dieser ICs aufwändig, kostenintensiv und erst bei sehr großen Stückzahlen rentabel.

Ein semi custom ASIC enthält hingegen vom Hersteller bereits vorgefertigte Komponenten oder Layer. Vom Kunden können diese dann entsprechend vervollständigt werden. Semi custom ICs werden in die zwei Gruppen *Array basierend (array based)* und *Zellen basierend (cell based)* unterteilt. Diese werden auf den nachfolgenden Seiten ausführlicher beschrieben.
Die *Entwicklungskosten* von semi custom ASICs sind für Kunden geringer, weil die ICs teilweise vorgefertigt sind. Da der Hersteller die dafür nötige Entwicklung übernimmt, sind die Stückpreise relativ hoch. Zudem werden oft nicht alle Komponenten im Baustein benötigt, d.h. beim Kauf

ist es oft nötig einen Kompromiss einzugehen. Beim full custom ASIC sind hingegen nur die tatsächlich benötigten Komponenten auf dem IC enthalten. Die unterschiedlichen Merkmale der verschiedenen ASIC-Typen sind aus Abb. 2.2 ersichtlich.

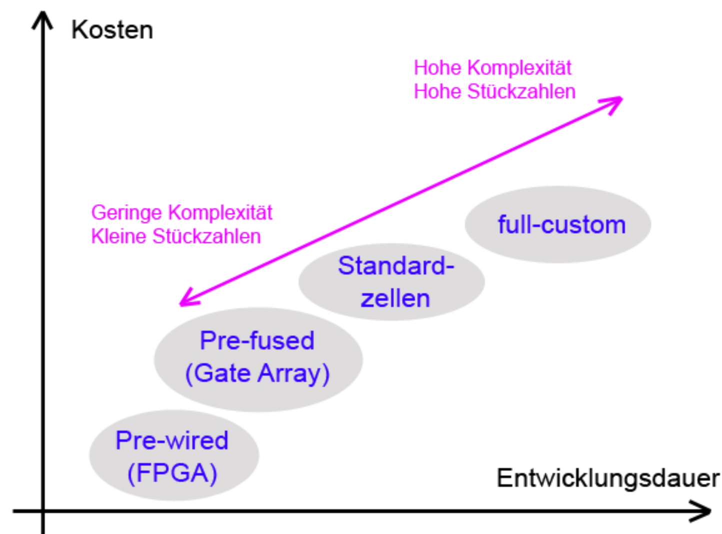

Abbildung 2.2: Merkmale der verschiedenen ASIC-Typen.

Im Gegensatz zu den ASICs gibt es eine große Zahl von Standard-ICs, den sogenannten *General Purpose ICs (GPICs)*. Diese werden von zahlreichen Herstellern für universelle Anwendungen und große Kundengruppen entwickelt und bestehen meist aus Standardkomponenten. Typische Bausteine sind z.B. Analog-Digital-Umsetzer (ADC), Speicher, Logik-ICs oder auch komplexe Mikroprozessoren.

2.1 Zellen basierte semi-custom ASICs (cell based)

2.1.1 Standardzellen

Als Standardzellen bezeichnet man die von einem ASIC-Hersteller vorgefertigten Grundelemente. Dabei handelt es sich um analoge oder digitale Grundschaltungen wie z.B. Logik-Gatter, Inverter oder Speicherelemente. Die Zellen werden vom Hersteller speziell auf den Fertigungsprozess zugeschnitten und als Bibliothek für die Entwicklungssoftware ausgeliefert. Diese Bibliotheken enthalten für jede Zelle zusätzliche Daten wie Symbole, Bauteilparameter und Layouts.

Die Standardzellen werden vom Entwickler im Chip in Reihen (cell rows) frei platziert und über den sogenannten Routingkanal (routing channel) miteinander verdrahtet.
Anschließend werden die Fertigungsdaten an den Hersteller übermittelt und dieser fertigt den Chip von Grund auf. Der Vorteil der Standardzellen liegt in der relativ guten Ausnutzung der Chipfläche. Die Fertigungskosten sind sehr hoch, da sämtliche Layer gefertigt werden.

2.1.2 Macrozellen

Macrozellen sind den Gate Arrays sehr ähnlich (siehe Kapitel 2.2.2). Allerdings handelt es sich hier vorwiegend um relativ komplexe Logikzellen, wie z.B. ganze Arithmetic Logic Units (ALU), Register oder spezielle Flip-Flops.

2.2 Array basierte semi-custom ASICs (array based)

2.2.1 Pre-wired Architektur (CPLD, FPGA)

Bei *pre-wired* Typen handelt es sich um ASICs mit komplett vorgefertigten Layern. Die ICs werden vom Hersteller entwickelt, gefertigt und getestet. Die Funktionalität wird durch Programmierung bzw. Konfiguration des Bausteines festgelegt.
Auf dem Markt gibt es mittlerweile eine große Anzahl von verschiedenen pre-wired Bausteinen. Diese unterscheiden sich vor allem durch ihre Komplexität, Geschwindigkeit, Architektur und die Art der Programmierung. Die Bandbreite reicht dabei von einmal bis mehrfach programmierbaren Bausteinen.
Zwei der bekanntesten Vertreter dieser ASIC-Typen sind das sogenannte *Complex Programmable Logic Device (CPLD)* und *Field Programmable Gate Array*. Bei CPLDs handelt es sich um programmierbare Logik mit einer Verbindungsmatrix und I/O-Blöcken.

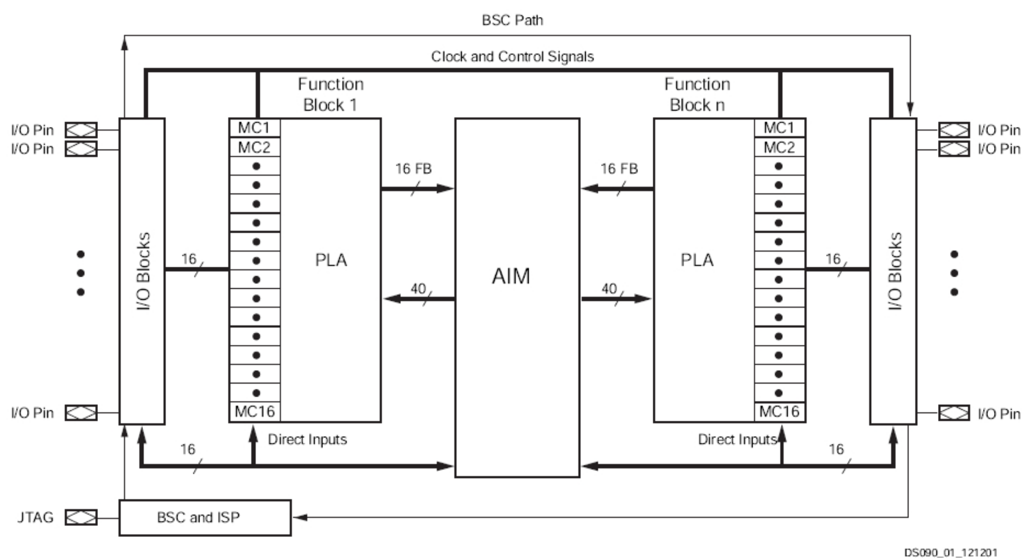

Abbildung 2.3: Architektur der Xilinx CoolRunner-II CPLD Familie [Xil08a].

Aus Abb. 2.3 ist die Architektur eines Xilinx CoolRunner-II CPLD Bausteines ersichtlich. Die Verbindungsmatrix (Advanced Interconnection Matrix, AIM) definiert die Verbindungen zwischen den zwei programmierbaren Logikblöcken (Programmable Logic Array, PLA) und den I/O-Blöcken. Für nähere Details siehe [Xil08a].

Auf die Funktionsweise und den Aufbau von FPGA Bausteinen wird in Kapitel 3 eingegangen.

2.2.2 Pre-diffused Architektur (Gate Array)

Als *Gate Array* bezeichnet man ICs die mehrere vorgefertigte Layer besitzen. Durch diese Layer werden Transistoren erzeugt, welche aber nicht miteinander verdrahtet sind. Die Transistoren werden vom ASIC-Hersteller als Reihen (Channeled Gate Array) oder flächendeckend (Sea-of-Gates Gate Array) angeordnet. Die Entwicklung einer Schaltung erfolgt, indem die Verdrahtung der einzelnen Transistoren definiert wird. Die vereinfachte Darstellung eines Channeled Gate Array ist aus Abb. 2.4 ersichtlich.

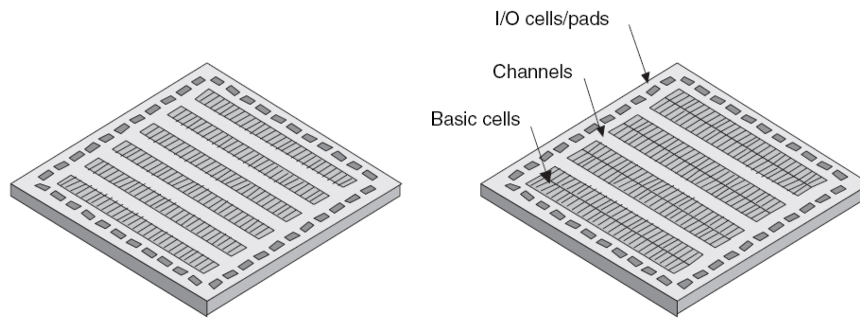

Abbildung 2.4: Vereinfachte Darstellung eines Channeled Gate Array.

Die Fertigungsdaten werden an den Hersteller gesendet und dieser vervollständigt den Chip mit den Verdrahtungs-Layern. Im Vergleich zu den Standardzellen (siehe Kapitel 2.1.1) sind diese ASICs bei der Fertigung günstiger. Ein Nachteil ist allerdings die vergleichsweise schlechte Ausnutzung der Chipfläche.

3 Field Programmable Gate Arrays

Bei *Field Programmable Gate Arrays (FPGAs)* handelt es sich um digitale Logikbausteine. Sie sind ein- oder mehrmals programmierbar und ermöglichen eine kostengünstige und rasche Entwicklung komplexer Schaltungen.
Im Gegensatz zu Gate Arrays (siehe Kapitel 2.2.2) besitzen FPGAs eine spezielle programmierbare Verdrahtung. Die Funktionalität kann somit direkt vom Entwickler, ohne teure Fertigungsschritte beim Hersteller, implementiert werden. Die Nachteile von FPGAs, im Vergleich zu full custom ASICs, sind jedoch die schlechtere Platzausnutzung und die Beschränkung auf vorwiegend digitale Schaltungen.

Als *Architektur* bezeichnet man die Art und Weise der Zusammenschaltung und die Ausprägung der vorhandenen Komponenten. Je nach Hersteller unterscheiden sich die Architekturen mehr oder weniger stark voneinander, jedoch lassen sich einige allgemein übliche *Grundkomponenten* definieren. Eine vereinfachte Architektur ist aus Abb. 3.1 ersichtlich.

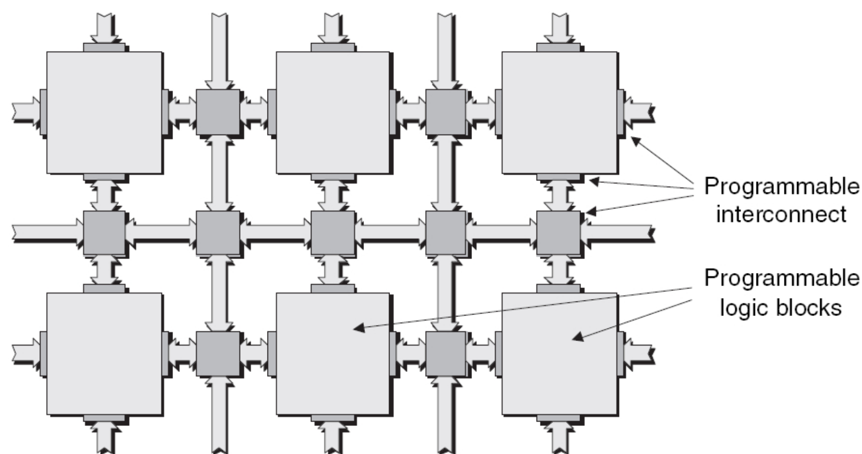

Abbildung 3.1: Vereinfachte Darstellung einer FPGA Architektur [Max04].

Die kleinsten programmierbaren Grundelemente sind die sogenannten *Logikzellen (logic blocks)*. Durch Konfiguration der Verbindungen und der Logikzellen wird vom Entwickler eine bestimmte Funktionalität definiert. Diese Programmierung wird in *Konfigurationszellen* gespeichert. Im allgemeinen werden Logikzellen zu größeren Einheiten, den sogenannten *Slices* und *CLBs*, zusammengefasst. Die meisten FPGAs besitzen zudem einen eingebauten *Speicher* und Komponenten für spezielle Anwendungen.

In den folgenden Abschnitten werden die wichtigsten Komponenten eines FPGA im Detail vorgestellt. Dabei werden die von der Firma Xilinx eingeführten Begriffe für die einzelnen Komponenten verwendet. Weitere Informationen und Abbildungen zu diesem Kapitel können aus [Max04], [Mia07] und [Qui08] entnommen werden.

3.1 Konfigurationszellen (Configuration Cells)

Ein wesentlicher Vorteil von FPGAs ist die Tatsache, dass ihre Funktionalität programmierbar ist. Dazu werden die eingebauten Logikkomponenten (siehe Kapitel 3.2 und 3.3) und die programmierbaren Verbindungen (interconnections) so konfiguriert, dass daraus eine gewünschte Funktionalität entsteht.

Mit Hilfe einer bauteilspezifischen Hersteller-Software wird beim Entwicklungsprozess eine Konfiguration für die interne Verdrahtung des FPGAs erzeugt. Die daraus entstandenen Daten werden in den FPGA geladen und in speziellen Konfigurationszellen gespeichert.
Je nach Art der im FPGA enthaltenen Konfigurationszellen, müssen diese Daten nur einmal (Antifuse, EEPROM) oder bei jedem Einschaltvorgang (SRAM) in den FPGA geschrieben werden.

Nähere Informationen dazu finden sich in den nachfolgenden Abschnitten und in [Max04].

3.1.1 Antifuse-Zellen

Bei den Antifuse-Zellen handelt es sich um nichtflüchtige (non-volatile) Konfigurationszellen. Bei der Programmierung werden zwei isolierte Verbindungen elektrisch miteinander verbunden. Dies geschieht z.B. indem ein nichtleitendes amorphes Silizium, durch Anlegen einer hohen Spannung, in Poly-Silizium umgewandelt wird. Der Zustand einer Antifuse-Zelle wird dadurch dauerhaft festgelegt, d.h. sie kann später nicht neu programmiert werden. In Abb. 3.2 ist eine Antifuse aus amorphem Silizium dargestellt. Nähere Informationen dazu sind in [Qui08] zu finden.

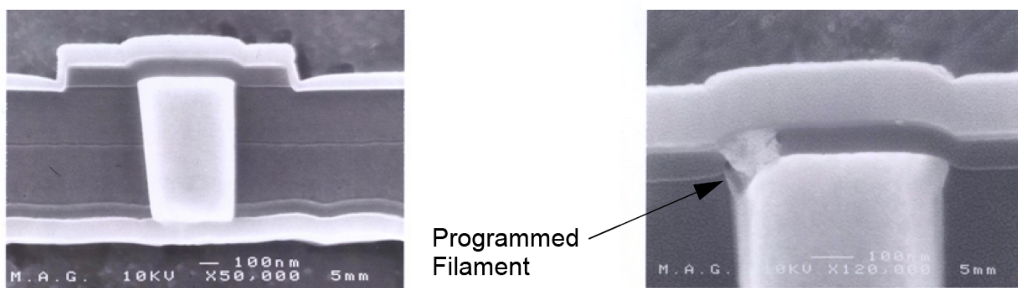

Abbildung 3.2: Darstellung einer programmierten Antifuse [Qui08].

Ein weiterer Vorteil liegt darin, dass kein externer Speicherbaustein mit den Konfigurationsdaten benötigt wird. Dadurch ist der FPGA sofort nach dem Einschalten einsatzbereit. Dies spart Platz und Kosten. Ein unerwünschter Zugriff auf die Konfiguration wird fast unmöglich, weil keine messbaren Datenleitungen existieren und das Auslesen der Konfiguration durch eine spezielle Sicherheits-Antifuse physikalisch verhindert werden kann.

Erwähnenswert ist auch die Tatsache, dass Antifuse-Zellen besonders gut für Umgebungen mit Strahlung geeignet sind. Dadurch eignen sie sich besonders für Raumfahrtsysteme. Der Grund dafür liegt in ihrem physikalischen Aufbau. Im Vergleich zu SRAM-Zellen sind die Antifuses kleiner. Daraus ergibt sich mehr Platz für die eigentliche Logik, höhere Geschwindigkeiten und weniger Standby-Stromaufnahme.
Ein Nachteil ist jedoch, dass FPGAs mit Antifuse-Zellen nur off-line programmiert werden können. Dazu muss der IC in ein spezielles Programmiergerät eingesetzt werden.

3.1.2 SRAM-Zellen

Diese Art von Konfigurationszellen ist am weitesten verbreitet. Die Abkürzung *SRAM* steht für *Static Random-Access Memory*. Die Zellen sind flüchtig (volatile), d.h. die gespeicherte Konfiguration geht beim Ausschalten verloren. Für gewöhnlich besteht eine SRAM-Zelle in einem FPGA aus vier bis sechs Transistoren. Sie ist also wesentlich größer als eine Antifuse Zelle, weshalb weniger Platz für Logik vorhanden ist (siehe Abb. 3.3).

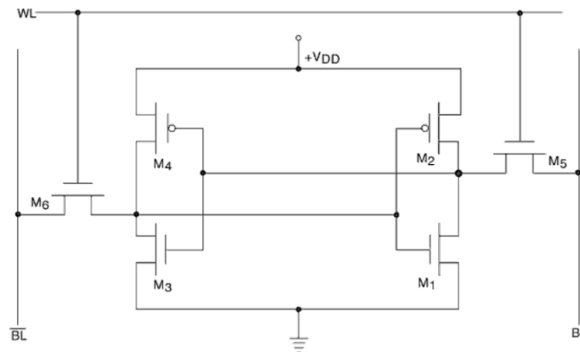

Abbildung 3.3: SRAM Speicher bestehend aus sechs Transistoren [Mia07].

Im Gegensatz zu den Antifuse-Zellen können SRAM-Zellen beliebig oft programmiert werden. Dies ermöglicht es nachträglich Änderungen an der Software (Firmware) vorzunehmen und somit Updates durchzuführen. Dadurch erspart sich das Unternehmen hohe Wartungskosten. Auf Grund der stetigen Entwicklung im Bereich von Speicherbausteinen sind SRAM gegenüber Antifuse-Zellen technologisch höher entwickelt und auch für neueste Prozesstechnologien verfügbar.

Ein Nachteil besteht allerdings darin, dass die Konfigurationsdaten bei jedem Einschaltvorgang in den FPGA geschrieben werden müssen. Dies sorgt nicht nur für höhere Bauteilkosten sondern auch für Sicherheitsrisiken. Spezialisten können den Chip leicht kopieren, indem sie die Konfigurationsdaten auslesen. Dementsprechend benutzen manche FPGAs verschlüsselte Übertragungsverfahren. Fehler im externen Speicherbaustein führen zum Ausfall des FPGAs.
Erwähnenswert ist auch die Tatsache, dass SRAM-Zellen viel empfindlicher auf die kosmische Strahlung reagieren als Antifuse-Zellen. Dementsprechend werden von manchen Herstellern spezielle Schutzmechanismen eingebaut und strahlungsresistentere Gehäusematerialien verwendet.

3.1.3 EEPROM und Flash-Zellen

Die Abkürzung *EEPROM* oder E^2PROM steht für *Electrically Erasable Programmable Readonly Memory*, was soviel bedeutet wie elektrisch lösch- und programmierbarer Lesespeicher. Die sogenannten *Flash* Zellen besitzen die gleichen Eigenschaften wie E^2PROM und unterscheiden sich nur im inneren Aufbau. Entsprechend handelt es sich dabei um nichtflüchtigen (non-volatile) Speicher der mehrmals elektrisch konfiguriert werden kann. Die Programmierung ist off-line mit Programmiergeräten oder on-line über spezielle Programmierschnittstellen (in circuit programmable) möglich.
Trotz der Ähnlichkeit zu SRAM-Zellen werden bei EEPROM und Flash-Zellen jeweils nur ein bis zwei *Floating-Gate Transistoren* benötigt. Diese werden in Reihen angeordnet, wodurch weniger Platz verbraucht wird und sich die Chipfläche verkleinert (siehe Abb. 3.4).

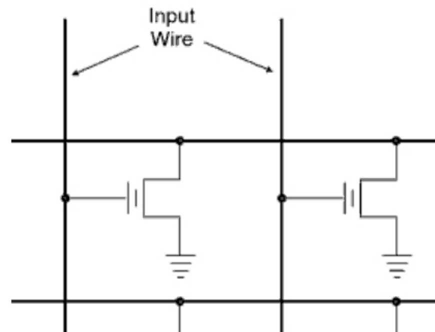

Abbildung 3.4: EEPROM Speicher bestehend aus Floating-Gate Transistoren [Mia07].

Externe Speicherbausteine sind nicht nötig. Einige Hersteller verwenden Verschlüsselungsverfahren zum Schutz der Konfiguration während der Übertragung. Updates und Auslesevorgänge können damit nur nach Eingabe des richtigen Schlüssels durchgeführt werden.
Der Fertigungsaufwand bei EEPROM und Flash ist im Vergleich zu SRAM höher. Deshalb sind sie durchschnittlich ein bis zwei Entwicklungsgenerationen älter als SRAM-Zellen.

3.1.4 Zusammenfassung

Abb. 3.5 zeigt eine Zusammenfassung mit den wichtigsten Eigenschaften der verschiedenen Konfigurationszellen. Nähere Informationen dazu können in [Max04] nachgelesen werden.

Feature	SRAM	Antifuse	E2PROM / FLASH
Technology node	State-of-the-art	One or more generations behind	One or more generations behind
Reprogrammable	Yes (in system)	No	Yes (in-system or offline)
Reprogramming speed (inc. erasing)	Fast	----	3x slower than SRAM
Volatile (must be programmed on power-up)	Yes	No	No (but can be if required)
Requires external configuration file	Yes	No	No
Good for prototyping	Yes (very good)	No	Yes (reasonable)
Instant-on	No	Yes	Yes
IP Security	Acceptable (especially when using bitstream encryption)	Very Good	Very Good
Size of configuration cell	Large (six transistors)	Very small	Medium-small (two transistors)
Power consumption	Medium	Low	Medium
Rad Hard	No	Yes	Not really

Abbildung 3.5: Zusammenfassung der Eigenschaften von Konfigurationszellen [Max04].

3.2 Logikzellen (Logic Cells)

Unter *Logikzellen (logic cell, Xilinx)* versteht man die programmierbaren logischen Grundkomponenten eines FPGAs. Jeder Hersteller verwendet traditionell andere Bezeichnungen, wie z.B. *Logikblock (logic block)* oder *Logikelement (logic element, Altera)*. Entsprechend weicht auch die Funktionalität zwischen den Herstellern ab.
Generell gibt es zwei Typen, die *Multiplexer basierten (MUX based)* und die *Lookup-Tabellen basierten (LUT based)* Logikzellen. Bei neueren FPGA Architekturen werden meist Mischformen eingesetzt. Eine Xilinx Logikzelle mit einer 4-fach LUT, einem Multiplexer und einem Flip-Flop ist aus Abb. 3.6 ersichtlich.

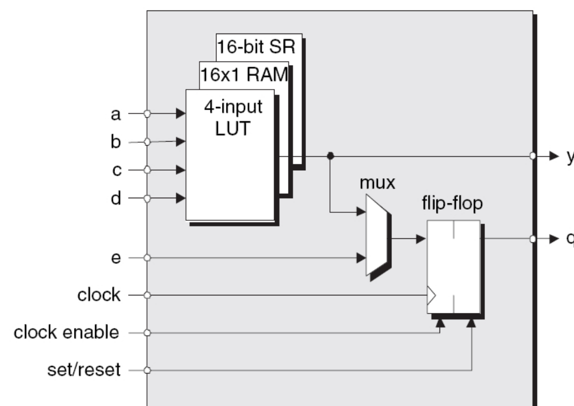

Abbildung 3.6: Vereinfachte Darstellung einer Logikzelle [Max04].

Die Lookup-Tabelle ist so gestaltet, dass sie zudem als Speicher (RAM) oder Schieberegister (SR) eingesetzt werden kann. Jede Logikzelle ist zusätzlich zu den I/O-Leitungen an eine *clock*, *clock enable* und *set/reset*-Leitung angeschlossen.

Die nächstgrößere logische Komponente besteht aus mehreren miteinander verbundenen Logikzellen und wird von Xilinx als *Slice* bezeichnet. Die genaue Anzahl hängt dabei vom FPGA Modell ab. Im Allgemeinen teilen sich alle Slices auf dem Chip eine gemeinsame *clock*, *clock enable* und *set/reset*-Leitung.

3.3 Configurable Logic Block

Die größte logische Grundkomponente eines FPGAs wird im Allgemeinen als *Configurable Logic Block (CLB)* bezeichnet. Andere Hersteller wie Altera benutzen dafür auch die Bezeichnung *Logic Array Block (LAB)*.
Für einen CLB werden typischerweise zwei oder vier *Slices* zusammengefasst. Die genaue Anzahl hängt dabei von der Architektur des FPGAs ab. Ein vereinfachter CLB mit vier Slices ist aus Abb. 3.7 ersichtlich.

Wie im vorigen Kapitel erklärt wurde, sind Slices wiederum aus Logikzellen aufgebaut. Diese Hierarchie spiegelt sich auch in der Geschwindigkeit der Verbindungen wieder. Logikzellen sind mit den schnelleren Verbindungen ausgestattet als CLBs.

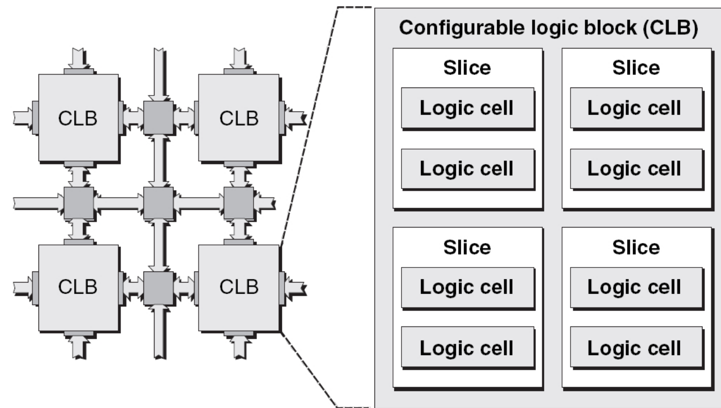

Abbildung 3.7: Vereinfachte Darstellung eines Xilinx CLB [Max04].

3.4 Eingebauter RAM (Embedded RAM)

Neben den bereits erwähnten Logikzellen und CLBs wird in viele FPGAs zusätzlich Speicher (embedded RAM) eingebaut. Viele Anwendungen benötigen große Mengen an Speicher, weshalb die Hersteller immer mehr RAM-Blöcke in die FPGAs integrieren. Dadurch wird eine Vielzahl neuer Anwendungen möglich. Eine beispielhafte Anordnung der eingebauten RAM-Blöcke ist aus Abb. 3.8 ersichtlich.

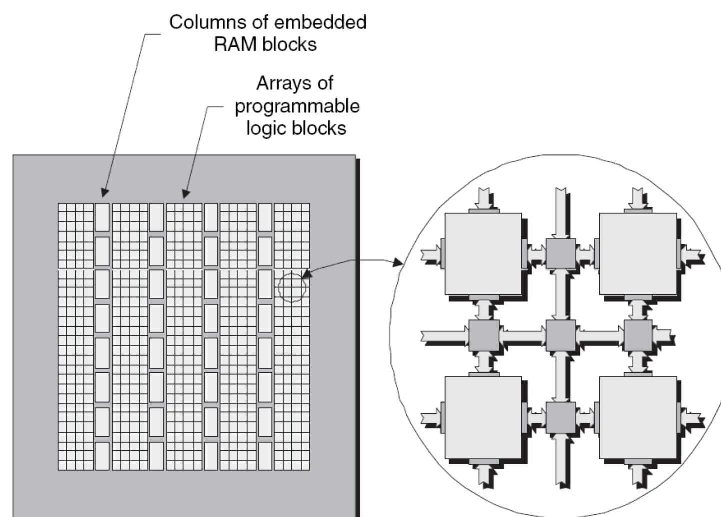

Abbildung 3.8: Eingebauter RAM zwischen den CLBs [Max04].

3.5 Zusätzliche Komponenten

In vielen Anwendungen kommen immer wieder ähnliche Schaltungen zum Einsatz. Deshalb gehen viele Hersteller dazu über diese häufig verwendeten Komponenten bereits von Anfang an in den FPGA einzubauen. Die Komponenten sind somit als reale Hardware vorhanden und müssen

nicht aus den CLBs programmiert werden. Der Vorteil liegt darin, dass diese Komponenten vom Hersteller optimiert sind und meist bessere Eigenschaften aufweisen als die programmierte Version. Die CLBs selbst stehen somit für andere Aufgaben zur Verfügung. Am häufigsten verbreitet sind spezielle *Multiplizierer*- und *Digital Signal Processing (DSP)*-Blöcke.

Einige FPGA besitzen zusätzlich eingebaute Prozessoren, sogenannte *EPCs (Embedded Processor Cores)*. Dabei handelt es sich um komplette Prozessorsysteme mit eigenem Befehlssatz. Die Firma Xilinx verwendet dafür meist *PowerPC* RISC-Prozessoren.

3.6 Clock Tree und Clock Manager

Ein Grundprinzip moderner FPGAs ist die Verwendung von *sequentieller Logik*. Darunter versteht man Schaltungen die ihren logischen Zustand nur zu exakt definierten Zeitpunkten ändern und somit ein speicherndes Verhalten aufweisen.
Diese Zeitpunkte werden durch ein extern erzeugtes rechteckförmiges *Taktsignal (Clock)* vorgegeben. Moderene FPGAs können mit Taktfrequenzen bis zu einigen hundert Megahertz betrieben werden.

Die wichtigste sequentielle Grundschaltung wird als *Flip-Flop (FF)* bezeichnet. Dieses übernimmt bei einer steigenden (oder fallenden) Taktflanke einen Eingangswert und setzt kurz darauf seinen Ausgangswert. Die Änderung kann von nachfolgenden FFs bei der nächsten Taktflanke übernommen werden. Daraus ergibt sich, dass Werte zwischen den einzelnen Taktflanken gespeichert werden.
Das Taktsignal muss deshalb an alle Komponenten im FPGA angeschlossen sein. Dies geschieht über eine verzweigte Verbindung, dem sogenannten *Clock Tree* (siehe Abb. 3.9).

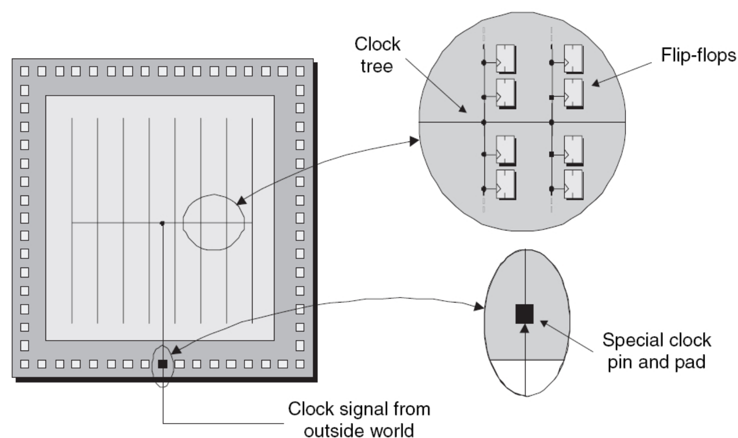

Abbildung 3.9: Vereinfachter Aufbau eines Clock Trees [Max04].

Wenn sich alle Signale immer nur bei einer fallenden oder steigenden Flanke ändern, dann spricht man von einer *synchronen Schaltung*. Falls Änderungen auch zwischen den Taktflanken möglich sind, handelt es sich um eine *asynchrone Schaltung*. Der Vorteil synchroner Schaltungen liegt in ihrer Einfachheit und hohen Zuverlässigkeit. Zur Vermeidung von Problemen sollten deshalb bevorzugt synchrone Schaltungen entworfen werden.

Auf Grund der verschiedenen Leitungslängen kann es am Clock Tree zwischen den einzelnen Strängen zu einem *Zeitversatz (Jitter)* kommen. Der *Taktmanager (clock manager)* detektiert und korrigiert diese unerwünschten Effekte. Zudem ermöglicht er die Erzeugung mehrerer unterschiedlicher Taktsignale. Dadurch können mehrere Clock Trees mit jeweils verschiedenen Taktfrequenzen verwendet werden (siehe Abb. 3.10).

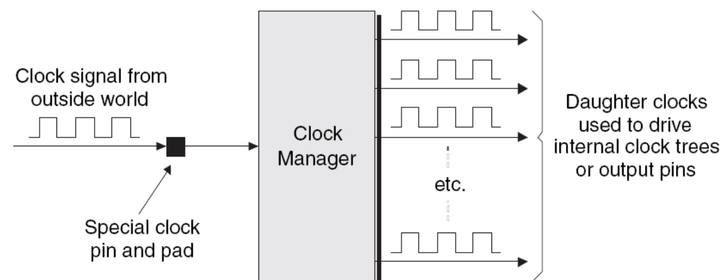

Abbildung 3.10: Clock Manager zur Korrektur von Jitter [Max04].

3.7 Ein- und Ausgabe von Daten

Ein FPGA besitzt zahlreiche *Anschlüsse (Pins)* für die Ein- und Ausgabe (Input/Output, kurz I/O) von Daten. Die Anzahl der Pins hängt dabei von der Architektur und der verwendeten Gehäuseform ab.

Moderne FPGAs mit sogenannten *Pin Grid Array (PGA)* Gehäusen können derzeit bis zu etwa 2000 Pins besitzen. Neben einigen Anschlüssen für die Versorgungsspannung und den Takt, werden für gewöhnlich viele Pins für den Daten-I/O und die Kommunikation mit anderen ICs benötigt.
In der Regel werden mehrere Anschlüsse zu einer größeren *I/O-Bank (I/O bank)* zusammengefasst. Eine vereinfachte Darstellung eines FPGAs mit acht I/O-Bänken ist aus Abb. 3.11 ersichtlich.

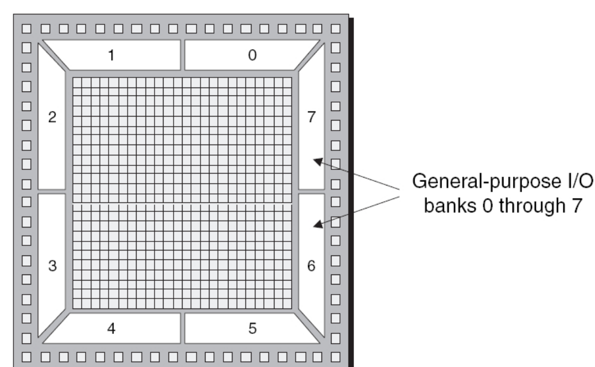

Abbildung 3.11: Konfigurierbare I/O-Bänke in einem FPGA [Max04].

In den letzten Jahrzehnten wurden immer neuere Logikfamilien entwickelt die teilweise unterschiedliche Spannungspegel verwenden. Dementsprechend gibt es eine Vielzahl an gängigen *I/O-Standards*.
Jede einzelne der I/O-Bänke kann so konfiguriert werden, dass alle zugehörigen Anschlüsse einen bestimmten I/O-Standard verwenden. Je nachdem ob ein Anschluss als Eingang, Ausgang oder beides verwendet wird, ändert sich die Konfiguration der internen Beschaltung. Dadurch wird es erst möglich, dass der FPGA mit einer Vielzahl unterschiedlicher Nachbar-ICs kommunizieren und Daten austauschen kann.

Auf Grund der oftmals hohen Signalfrequenzen von einigen hundert MHz kann es zu Fehlanpassung und damit zu Reflexionen kommen. Deshalb kann in einigen I/O-Bänken die Ein- bzw. Ausgangsimpedanz eingestellt werden.

4 Entwurf und Entwicklung integrierter Schaltungen

Die steigende Komplexität von FPGAs und anderer integrierter Schaltungen sorgt für immer aufwändigere Entwicklungsabläufe. Eine schlechte Planung kann schnell zu unerwarteten Zeitverzögerungen und rasch ansteigenden Entwicklungskosten führen. Im Gegensatz zu den Halbleitertechnologien ändern sich die gängigen Entwicklungsabläufe nur langsam. Für die Unternehmen wird ein effektiver Entwicklungsprozess zum immer wichtigeren Erfolgsfaktor. Je später Fehler entdeckt werden, umso höhere Kosten sind damit verbunden. In den letzten Jahren haben sich deshalb einige grundlegende *Designprinzipien* etabliert. Diese erleichtern die Entwicklung und sorgen für eine Effizienzsteigerung.

Das wohl wichtigste Designprinzip kann man als *hierarchische Unterteilung (Divide and Conquer)* bezeichnen. Darunter versteht man die Aufteilung eines Entwurfes in kleinere funktionale Module, welche über *Schnittstellen* miteinander verbunden sind. Jedes Modul ist wiederum aus noch kleineren Untermodulen aufgebaut. Dadurch entstehen unterscheidbare Ebenen mit einer jeweils eigenen Beschreibung (siehe Abb. 4.1).

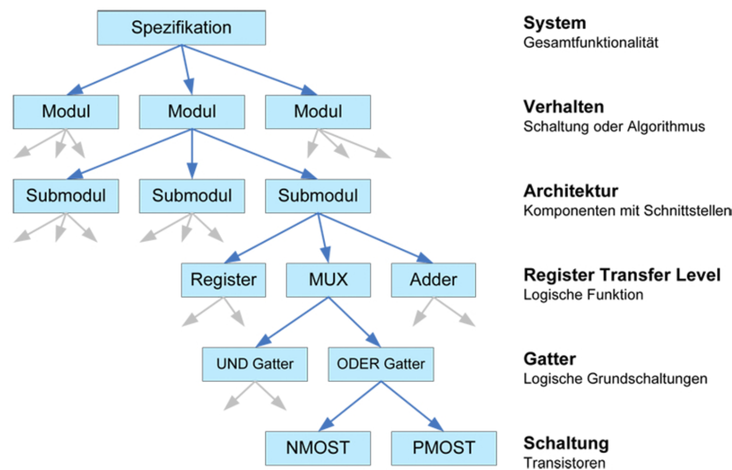

Abbildung 4.1: Divide & Conquer Ebenenhierarchie mit jeweils eigener Beschreibung.

Beim *Top-Down Entwurf* wird zuerst das System spezifiziert. Daraus werden die Module entworfen und anschließend immer mehr verfeinert. Diese Entwurfsmethode ist am weitesten verbreitet. Entsprechend werden beim *Bottom-Up Entwurf* zuerst die kleinsten Grundkomponenten entworfen und dann zum Gesamtsystem zusammengefasst.
Der Vorteil der hierarchischen Unterteilung liegt darin, dass die einzelnen Komponenten auf den unteren Ebenen relativ einfach sind und schnell entworfen werden können. Der Entwickler konzentriert sich auf ein Modul und muss nicht immer das gesamte System betrachten. Auf Grund der Einfachheit der Komponenten kann zudem die Funktionalität viel besser getestet werden.

Das zweite wichtige Designprinzip wird als *Wiederverwendbarkeit (Reusability)* bezeichnet. Die einzelnen Komponenten sollen möglichst einfache Aufgaben lösen und exakt definierte Schnittstellen aufweisen. Diese Komponenten werden oft zu einer Bibliothek zusammengefasst und später in weiteren ICs verwendet. Der Vorteil liegt darin, dass man diese Grundkomponenten nur einmal entwickeln und testen muss. Dadurch spart man sich viel Zeit und Aufwand.

Ein weiteres Prinzip ist die *Softwareunterstützung*. Bei der Entwicklung werden *spezielle Softwarepakete* verwendet. Diese vereinfachen die Arbeit erheblich und übernehmen die vielen komplizierten Berechnungen. Durch Simulation können aufwändige Analysen durchgeführt und Fehler frühzeitig erkannt werden. Für digitale Schaltungen erledigt die Software, bei Bedarf, mehrere Arbeitsschritte voll- oder halbautomatisch. Eine Entwicklung vieler ICs wäre ohne eine solche Entwicklungssoftware gar nicht vorstellbar.

Im nachfolgenden Kapitel 4.1 wird genauer auf die Entwurfsmodelle für die Entwicklung integrierter Schaltungen eingegangen. Eine detaillierte Beschreibung des Entwicklungsprozesses für ASICs bzw. FPGAs kann in Kapitel 4.3 nachgelesen werden.
Die Informationen und Abbildungen der folgenden Kapitel wurden teilweise aus [Gro08], [CH06], [Tin00], [RL08] und [Kil07] entnommen.

4.1 Entwurfsmodelle und Abstraktionsebenen

In den letzten Jahren wurden einige *Entwurfsmodelle* definiert. Diese Modelle versuchen einen Entwurfsprozess durch sogenannte *Abstraktionsebenen* zu beschreiben.
Beim Entwurf integrierter Schaltungen werden mehrere Abstraktionsebenen durchlaufen. Jede Ebene beschreibt den Aufbau des Systems mit einem definierten Detailgrad. Eine Abstraktionsebene steht generell in direktem Zusammenhang mit einer speziellen Entwicklungsphase. Ein häufig verwendetes Designmodell wurde erstmals von Gajski und Kuhn eingeführt. Eine grafische Darstellung des Modells ist aus Abb. 4.2 ersichtlich.

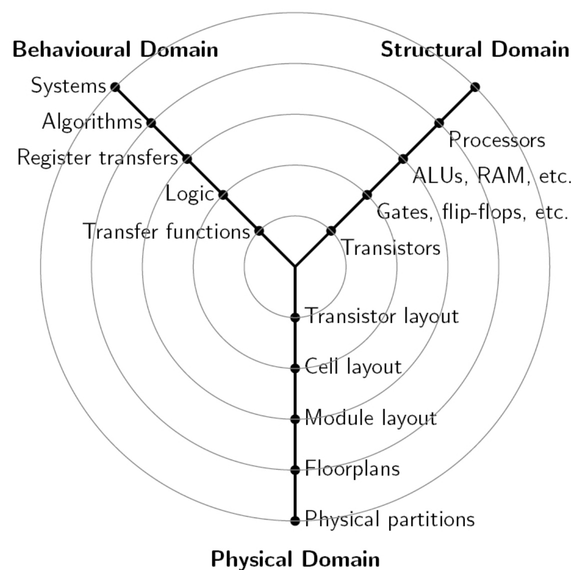

Abbildung 4.2: Y-Diagramm des Gajski-Kuhn Modells [Gro08].

Die einzelnen Abstraktionsebenen sind als Kreise dargestellt. Laut Gajski und Kuhn werden für integrierte Schaltungen mindestens *drei Beschreibungsarten* verwendet. Die *Verhaltensbeschreibung (behavioral domain)* erklärt die Funktionalität, während die *Strukturbeschreibung (structural domain)* bereits den Aufbau der Hardwarekomponenten beschreibt. Die *Geometriebeschreibung (geometric or physical domain)* bezieht sich auf den Aufbau und die Funktionsweise des Halbleitermaterials.
Bei einem Top-Down Entwurfsprozess werden die einzelnen Abstraktionsebenen von außen nach innen durchschritten. Jeder Entwurfsphase kann somit eine bestimmte Abstraktionsebene zugeordnet werden. Je näher man dem Zentrum kommt, umso detaillierter wird die integrierte Schaltung beschrieben. Umgekehrt wird ein Bottom-Up Entwurf von innen nach außen durchgeführt. Beim Entwurf einer integrierten Schaltung kann man natürlich beliebig zwischen den einzelnen Verhaltensbeschreibungen wechseln.

Die meisten dieser Modelle haben jedoch einen entscheidenden Nachteil. Sie enthalten keine konkreten Vorschläge zur Reihenfolge der einzelnen Entwurfsphasen und wie man in der Praxis von einer Ebene zur nächsten gelangt. Zudem ist nicht immer klar wie eine Abstraktionsebene in der Praxis genau definiert wird. Der tatsächlich gewählte Entwurfsprozess kann schlussendlich von vielen Faktoren abhängen.

Im nachfolgenden Kapitel 4.2 wird auf den traditionellen Entwurf und die Vorteile iterativer Methoden eingegangen. Ein typischer iterativer Entwurfsprozess für FPGAs wird detailliert in Kapitel 4.3 vorgestellt.

4.2 Iterativer und traditioneller Entwurfsprozess

4.2.1 Der traditionelle Entwurf

Der *traditionelle Ansatz* wird oft als *Wasserfall-Modell* bezeichnet. Darunter versteht man meist einen einfachen Top-Down Entwurf.
Ausgehend von der allgemeinen Verhaltensbeschreibung (Spezifikation) werden bei der Entwicklung die einzelnen Abstraktionsebenen nacheinander durchlaufen. Wichtig dabei ist, dass erst nach der Beendigung einer Entwurfsphase zur nächsten gewechselt wird. Dabei wird die integrierte Schaltung immer weiter verfeinert, bis schlussendlich der fertige IC entsteht.

Als großer Nachteil dieser Vorgehensweise erweist sich die Tatsache, dass Fehler in frühen Entwurfsphasen weitergereicht werden und schlussendlich zu erheblichen Problemen führen. In der Praxis wurden deshalb viele Zielvorgaben nicht erreicht oder Projekte sogar frühzeitig eingestellt. Aus diesem Grund wird heute meist ein iterativer Entwurf durchgeführt.

4.2.2 Der iterative Entwurf

Die Verwendung iterativer Methoden stellt eine wesentliche Verbesserung gegenüber den traditionellen Entwurfsprozessen dar.

Bei einem *iterativen Entwurfsprozess* werden die Ergebnisse nach jeder Entwurfsphase analysiert und kontrolliert. Dazu werden die Ergebnisse der Entwurfsphasen bzw. der verschiedenen Abstraktionsebenen miteinander verglichen. Fehler können dadurch frühzeitig erkannt und ausgebessert werden. Dieser Ablauf ist aus Abb. 4.3 ersichtlich.

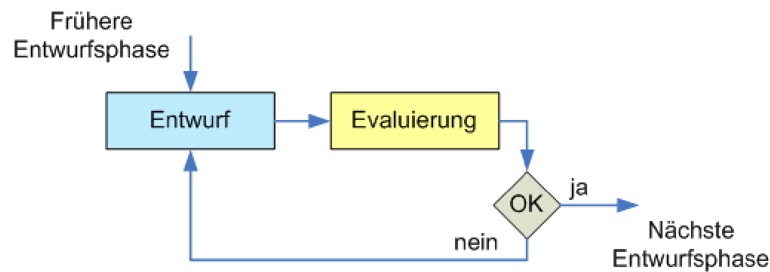

Abbildung 4.3: Ablauf einer iterativen Entwurfsphase.

Ist ein Problem in der aktuellen Entwurfsphase nicht lösbar, dann kann der Fehler meist auf einer höheren Abstraktionsebene ausgebessert werden. Es folgt daraus, dass bei Problemen die Ergebnisse bereits beendeter Entwurfsphasen nochmals überarbeitet werden. Somit kann sich z.B. die Spezifikation auch noch in späteren Phasen der Entwicklung ändern. Ziel ist es natürlich, dass man keinen oder möglichst jeweils nur einen Iterationsschritt zurück gehen muss.

In der Praxis zeigt sich, dass dadurch Entwicklungsfehler vermieden und Kosten eingespart werden. Auf Grund der ständigen Kontrolle kann der aktuelle Projektfortschritt einfacher eingeschätzt werden. Als Folge kommt es seltener zur Überschreitung der Entwicklungszeit. Im Durchschnitt wird eine größere Zahl an Projekten erfolgreich beendet oder wenigstens die Zielvorgabe besser eingehalten.

4.3 Typischer iterativer ASIC Entwurfsprozess

In diesem Kapitel wird ein *typischer iterativer Entwurfsprozess* vorgestellt. Dieser wird generell für die Entwicklung von ASICs und programmierbaren Bausteinen wie z.B. FPGAs oder CPLDs verwendet.

Jede Entwurfsphase liefert ein spezielles Ergebnis, welches von der nächsten Phase verwendet wird. Nach jeder Entwurfsphase wird eine *Evaluierung der Ergebnisse* durchgeführt. Bei dieser Überprüfung wird das aktuelle mit dem vorangegangenen Ergebnis verglichen. Die Ergebnisse beschreiben prinzipiell den gleichen ASIC. Sie tun dies aber auf unterschiedlichen Abstraktionsebenen. Es darf deshalb zwischen beiden keine funktionalen Unstimmigkeiten geben. Leider ist die Evaluierung in der Praxis nicht immer einfach, weshalb dafür vorwiegend die Entwicklungssoftware verwendet wird.

Ein Fehler kann gegebenenfalls in der aktuellen Entwurfsphase ausgebessert werden. Ist dies aus irgendwelchen Gründen nicht möglich, dann muss die Korrektur nachträglich in höher liegenden Abstraktionsebenen bzw. Entwurfsphasen durchgeführt werden. Anschließend kann zur nächsten Phase weiter gegangen werden.

Der Entwurfsprozess wird in Abb. 4.4 dargestellt. Aus Gründen der Übersichtlichkeit wurden die Iterationen nicht durch Pfeile dargestellt. Die Ergebnisse jeder Phase sind neben den Pfeilen vermerkt. Aus der Abbildung sind zusätzlich die wichtigsten Schritte jeder Entwurfsphase ersichtlich.

Die Übergänge zwischen den ersten vier Entwurfsphasen werden vorwiegend manuell durch den Entwickler durchgeführt. Daraus resultiert ein detailliertes Blockdiagramm mit allen Komponen-

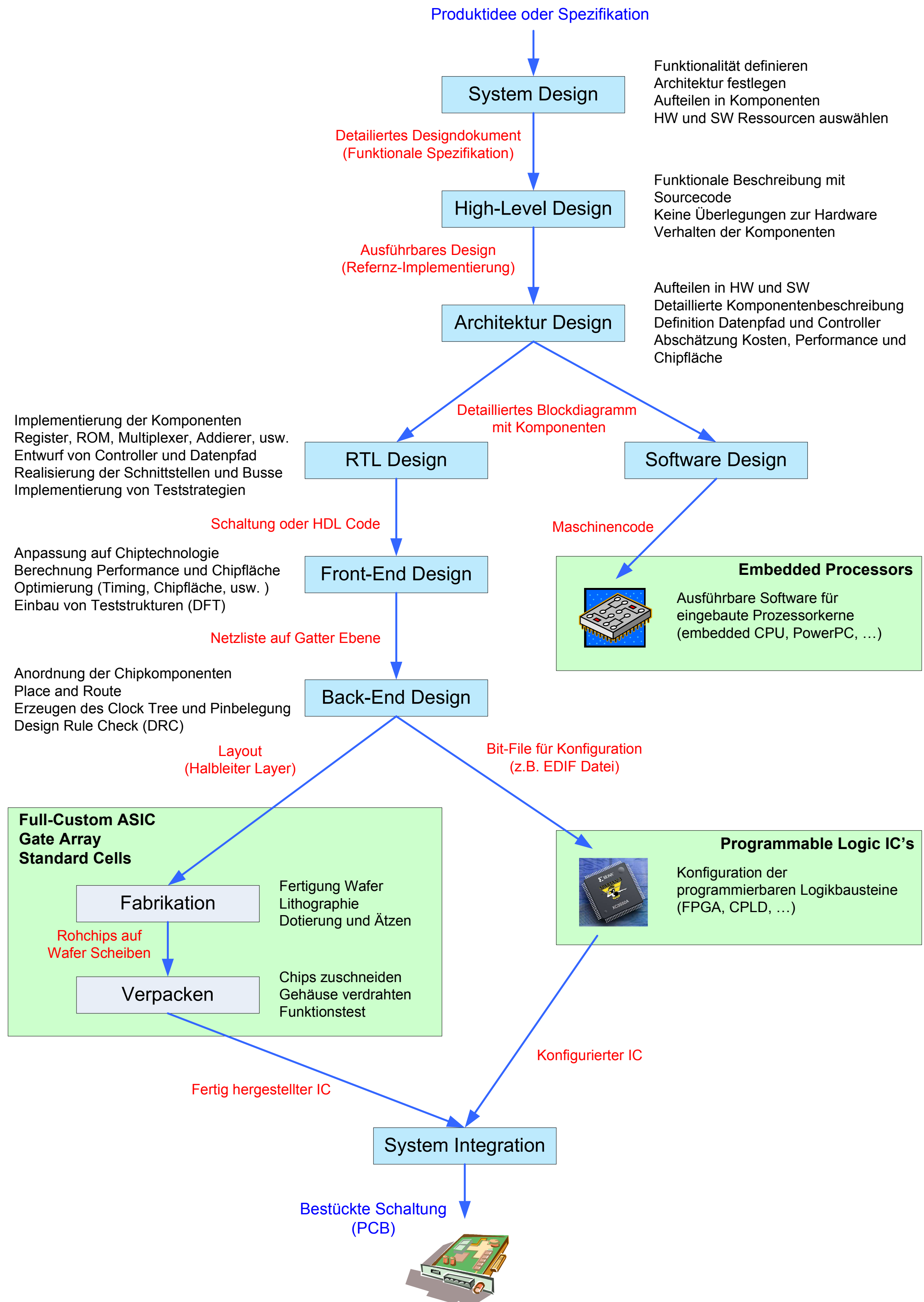

Abbildung 4.4: Designphasen bei der Entwicklung einer integrierten Schaltung.

ten der integrierten Schaltung. Zudem sind alle Funktionalitäten und Schnittstellen definiert. Im späteren Verlauf wird der Übergang, auf Grund der wachsenden Komplexität, vorwiegend mit Hilfe der Entwicklungssoftware durchgeführt. Diesen Vorgang bezeichnet man allgemein als *Synthese*.

In den nächsten Abschnitten folgt eine detaillierte Beschreibung der einzelnen Entwurfsphasen.

4.3.1 System Design

Input:	Produktidee oder Spezifikation
Output:	Detailliertes Designdokument mit funktionaler Spezifikation

Als *System Design* wird die erste und grundlegendste Phase des Entwurfs bezeichnet. Ausgehend von einer geforderten Zielsetzung, Idee oder Spezifikation wird die allgemeine Funktionsweise des ASICs formuliert. Das System wird nicht nur genau beschrieben sondern auch in sinnvolle Komponenten unterteilt. Daraus lässt sich dann bereits eine grobe Unterscheidung zwischen Hardware- und Softwarekomponenten ableiten.
Neben der Entwicklung einer funktionalen Spezifikation werden Überlegungen zu den benötigten Ressourcen getätigt. Oft entscheidet man sich bereits jetzt, ob fertige Komponenten eingekauft oder alles selbst entwickelt werden soll. Die Verwendung vorhandener Ressourcen spart oft viel Zeit und Geld.

Diese wichtigen Informationen und Überlegungen werden zu einem *Detaillierten Design Dokument (DDD)* zusammengefasst. Das DDD enthält die funktionale Spezifikation der integrierten Schaltung und bildet die Grundlage für den gesamten Entwurfsprozess.

Diese einleitende Phase wird auch dazu verwendet implizites Wissen zu generieren, d.h. sich Wissen anzueignen und Informationen einzuholen. Die Bandbreite reicht dabei von Überlegungen zur Projektplanung bis hin zur Aneignung von technischem Fachwissen und Schulung der Teammitglieder.

4.3.2 High-Level Design

Input:	Detailliertes Designdokument mit funktionaler Spezifikation
Output:	Ausführbare High-Level Referenz-Implementierung, Testvektoren

Ziel der *High-Level Design* Phase ist die Entwicklung einer funktionsfähigen High-Level *Referenz-Implementierung*. Diese soll die im DDD definierte Funktionalität besitzen. Einzelne Komponenten werden dabei als eigenes Unterprogramm implementiert. Häufig kommen dafür Hochsprachen wie C, C++ oder Java zum Einsatz. Ebenfalls denkbar ist die Verwendung von Programmen wie MATLAB. Bei der Implementierung sollen keine Überlegungen oder Annahmen zur später verwendeten Hardware gemacht werden.

Mit der Referenz-Implementierung kann überprüft werden, ob eine Spezifikation mit allen Komponenten so funktioniert, wie man es sich vorstellt. Zudem können alle Komponenten einzeln getestet und schrittweise analysiert werden. Meist werden zusätzlich sogenannte *Testvektoren* für die nächsten Entwurfsphasen erzeugt. Dabei handelt es sich um bestimmte Ausgabewerte der Referenz-Implementierung. In späteren Entwurfsphasen können dann die Ausgabewerte der integrierten Schaltung mit diesen Testvektoren verglichen werden. Abweichungen oder Fehler werden dadurch auf anderen Abstraktionsebenen rasch erkannt.

4.3.3 Architektur Design

Input: High-Level Referenz-Implementierung
Output: Detailliertes Blockdiagramm mit Komponenten und Schnittstellen

Beim *Architektur Design* werden, im Gegensatz zu den vorigen Entwurfsphasen, zum ersten Mal Überlegungen zur praktischen Realisierung als Hard- und Software durchgeführt. Hauptaugenmerk wird dabei auf die Architektur und den inneren Aufbau der Schaltung gelegt.

Basierend auf dem *Detaillierten Design Dokument (DDD)* und der Referenz-Implementierung, werden alle benötigten Blöcke definiert. Dies können z.B. Komponenten für die Ein- und Ausgabe von Daten, Speicher, ALUs und vieles mehr sein. Der Datenaustausch zwischen den Blöcken erfolgt über exakt definierte Schnittstellen. Dabei handelt es sich um mehrere Signalleitungen oder sogenannte Busse.
Der Entwickler muss entscheiden welche Blöcke als *Hardware oder Software* erstellt werden. Die Software wird z.B. für eingebaute Prozessoren verwendet (siehe Abschnitt 4.3.4). Das HW/SW Co-Design stellt dabei eine besondere Herausforderung dar.

In der Praxis wird die integrierte Schaltung oft als *endlicher Zustandsautomat (finite state machine, FSM)* realisiert. Der Vorteil dabei ist, dass eine FSM einen standardisierten und auch speziell organisierten Aufbau besitzt.
Eine FSM besteht prinzipiell aus einer *Zustandslogik (next state logic)* und einer *Ausgangslogik (output logic)*. Die Zustandslogik generiert dabei, in Abhängigkeit der Eingangsdaten und des aktuellen Zustandes (present state, PS), den nächsten Zustand (next state, NS). Die FSM merkt sich den aktuellen Zustand im Speicher (memory). Die Ausgangslogik erzeugt die für den aktuellen Zustand benötigten Ausgangsdaten. Anschließend wird der nächste Zustand zum aktuellen Zustand und der Vorgang wiederholt sich. Dieser Zusammenhang ist aus Abb. 4.5 ersichtlich.

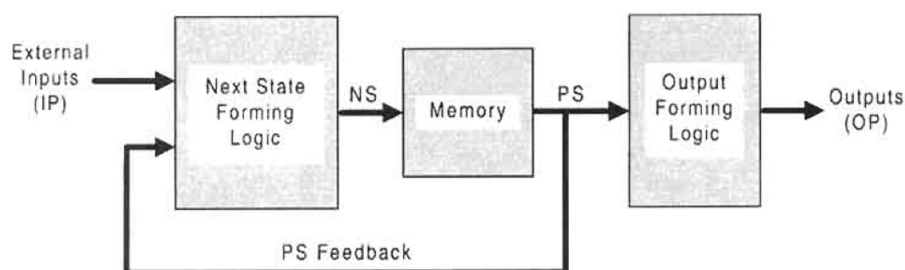

Abbildung 4.5: Endlicher Zustandsautomat mit Zustands- und Ausgangs-Logik [Tin00].

Neben einem Block für die Zustandslogik, müssen auch die Signalleitungen für die Zustände und die Schnittstellen zur Ausgangslogik eingeplant werden. Die Ausgangslogik enthält diejenigen Blöcke, welche zur Berechnung der Ausgangswerte benötigt werden. Nähere Details zu FSMs können aus [Tin00] entnommen werden.

In einem nächsten Schritt wird die interne Funktionsweise der einzelnen Blöcke beschrieben. Für die meisten Schaltungen werden Register, ROMs bzw. RAMs und verschiedene Grundschaltungen, wie z.B. Multiplexer, verwendet. Eine gegebenenfalls benötigte Software wird genauer spezifiziert. Nach Abschluss dieser Arbeiten ist die integrierte Schaltung komplett aus detailliert beschriebenen Blöcken aufgebaut. Daraus wird ein grafisches *Blockdiagramm*, mit allen wichtigen Informationen, erstellt.

Ein wichtiger Arbeitsschritt ist die *Abschätzung der Entwurfsparameter*. Jeder ASIC-Hersteller liefert Datenblätter in denen beispielsweise die *Chipfläche* und die *Leistungsaufnahme* für die verfügbaren Grundgatter aufgelistet sind. Alle integrierten Schaltungen bestehen am Ende aus diesen Gattern. Basierend auf dem Blockdiagramm, lässt sich bereits jetzt eine erste Abschätzung durchführen. Dies ist wichtig, weil z.B. eine zu hohe Leistungsaufnahme oder eine zu große Chipfläche unerwünscht ist.
Bei der Verwendung eines FPGAs kann bereits nach der Abschätzung entschieden werden, welcher Baustein verwendet werden soll. Je nach FPGA-Familie können vielleicht fertig eingebaute Komponenten verwendet werden. Diese müssen entsprechend beim Entwurf des Blockdiagrammes berücksichtigt werden.

4.3.4 Software Design

Input: Detailliertes Blockdiagramm mit Komponenten und Schnittstellen
Output: Maschinencode für eingebauten Prozessor (embedded processor core, EPC)

In der *Software* Entwurfsphase werden die im Blockdiagramm beschriebenen Softwarekomponenten implementiert. Diese werden z.B. für eingebaute Prozessorkerne benötigt. ASIC Bausteine wie z.B. FPGAs können mehrere dieser Prozessoren besitzen. Meist handelt es sich um *Reduced Instruction Set Computer (RISC)*-Prozessoren mit einem speziellen Befehlssatz.

Diese EPCs können spezielle Aufgaben im FPGA übernehmen, indem sie Programme aus dem Speicher abarbeiten. Dazu wird der benötigte Quellcode geschrieben und anschließend mit einem Compiler in den für den Prozessor verständlichen Maschinencode umgewandelt. Bei der Konfiguration des FPGA muss dieser dann in den Speicher geladen und für den Prozessor verfügbar gemacht werden. Die genaue Vorgehensweise hängt dabei von der verwendeten Entwicklungssoftware und der FPGA Familie ab.

4.3.5 Register Transfer Level Design

Input: Detailliertes Blockdiagramm mit Komponenten und Schnittstellen
Output: RTL Beschreibung - Schaltung oder HDL Quellcode

In der *Register Transfer Level (RTL)*-Designphase werden die im Blockdiagramm definierten Hardwarekomponenten, unter Einbeziehung des zeitlichen Verhaltens, implementiert. Der RTL ist die am häufigsten eingesetzte Abstraktionsebene bei der Entwicklung digitaler integrierter Schaltungen. In der Regel nimmt diese Entwurfsphase viel Zeit in Anspruch.

Das wichtigste Merkmal eines RTL-Entwurfes besteht darin, dass erstmals das *zeitliche Verhalten* in Abhängigkeit eines *Taktsignals (clock)* berücksichtigt wird. Die Schaltungen bestehen dementsprechend aus getakteter sequentieller Logik mit Flip-Flops und ungetakteter kombinatorischer Logik.
Oft steht der Begriff RTL auch für die Verwendung einer Hardwarebeschreibungssprache wie *Very High Speed Integrated Circuit Hardware Description Language (VHDL)* oder *Verilog HDL*.

Die *Implementierung der Hardware* erfolgt unter Zuhilfenahme einer entsprechenden Entwicklungssoftware. Mit dieser werden mehrere *Schaltungen (schematics)* oder *VHDL-Dateien* entworfen. Dabei erhält man eine Vielzahl an Modulen und Komponenten die miteinander über zahlreiche Schnittstellen hierarchisch verbunden sind. Gleichzeitig wird der im Blockdiagramm beschriebende Zustandsautomat, bestehend aus einer Zustands- und Ausgangslogik, erstellt. Als

Ergebnis erhält man schlussendlich ein vollständiges RTL-Modell der integrierten Schaltung. Die Funktionsweise muss dabei mit dem im Blockdiagramm definierten Verhalten übereinstimmen. Wichtig ist auch, dass beim Entwurf der später verwendete Halbleiterprozess unberücksichtigt bleibt.

Zur Gewährleistung des iterativen Prozesses werden die Ausgangsdaten des RTL-Modells durch Simulation ermittelt und mit den Ergebnissen höherer Abstraktionsebenen verglichen. Am einfachsten ist z.B. der Vergleich mit den Ausgangsdaten (Testvektoren) der Referenz-Implementierung.
Während des gesamten Entwurfes sollte außerdem das Prinzip der Wiederverwendbarkeit beachtet werden.

4.3.6 Front-End Design

Input: RTL Beschreibung - Schaltung oder HDL Quellcode
Output: Netzliste (netlist) auf Gatter-Ebene

Beim *Front-End Design* werden die Komponenten des RTL-Modells in eine *Logikschaltung auf Gatter-Ebene* umgewandelt. Dieser Vorgang wird ausnahmslos von der Entwicklungssoftware durchgeführt und als *Logiksynthese (logic synthesis)* bezeichnet. Dabei wird meist eine automatische Anpassung auf die verwendete Chiptechnologie durchgeführt. Der Synthesevorgang wird in Abb. 4.6 dargestellt.

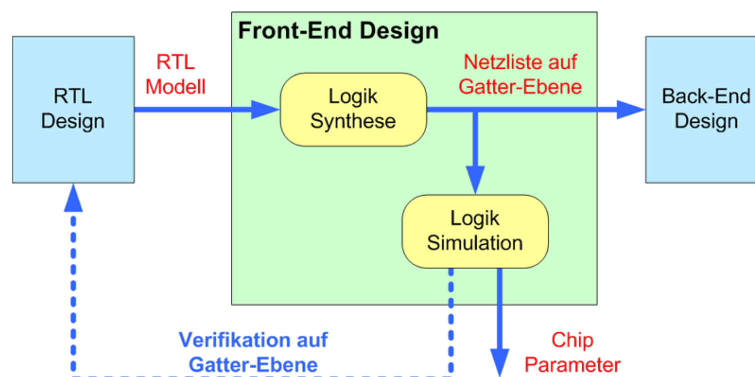

Abbildung 4.6: Logiksynthese des RTL-Modells auf die Gatter-Ebene.

Die daraus resultierende Schaltung besteht ausschließlich aus logischen Grundgattern und Flip-Flops. Der Vorteil einer Beschreibung auf Gatter-Ebene liegt darin, dass nahezu alle Eigenschaften der Schaltung simuliert werden können. Somit ist eine präzise Berechnung der *Chipfläche*, der *Laufzeiten* und der *Leistungsaufnahme* möglich.
Basierend auf diesen Parametern können mit der Entwicklungssoftware verschiedenste Optimierungen durchgeführt werden. Beispielsweise wird die Schaltung auf geringe Leistungsaufnahme oder Chipfläche optimiert.

Ein weiterer wichtiger Arbeitsschritt ist der *Einbau von Teststrukturen*. Diese werden zur Überprüfung der integrierten Schaltung verwendet. In den letzten Jahren haben sich diesbezügliche zahlreiche Konzepte bewährt (siehe Kapitel 4.4).

4.3.7 Back-End Design

Input: Netzliste auf Gatter-Ebene
Output: Halbleiter-Layout mit Transistoren oder Bit-File mit Konfigurationsdaten

Beim *Back-End Design* werden alle logischen Gatter aus der Netzliste in die entsprechende Halbleiterstruktur umgewandelt. Die exakte Vorgehensweise unterscheidet sich dabei erheblich zwischen full-custom ASICs bzw. Gate Arrays und programmierbaren Bausteinen.

Full-Custom ASICs und Gate Arrays

Bei diesen Bausteinen werden die Gatter durch *Struktursynthese (structural synthesis)* in die entsprechende Transistorschaltung umgewandelt. Als Ergebnis erhält man eine weitere Netzliste.

Ausgehend von der Transistorschaltung wird ein weiterer *Anordnen und Verdrahten (Place and Route)*-Syntheseschritt durchgeführt. Dabei werden die Transistoren, je nach Vorgabe, optimal angeordnet und miteinander verdrahtet. Benötigte Komponenten, wie z.B. der Clock-Tree, werden automatisch eingebaut. Daraus kann das endgültige *Layout* der integrierten Schaltung erstellt werden. Ein Layout für einen einfachen Inverter ist aus Abb. 4.7 ersichtlich.

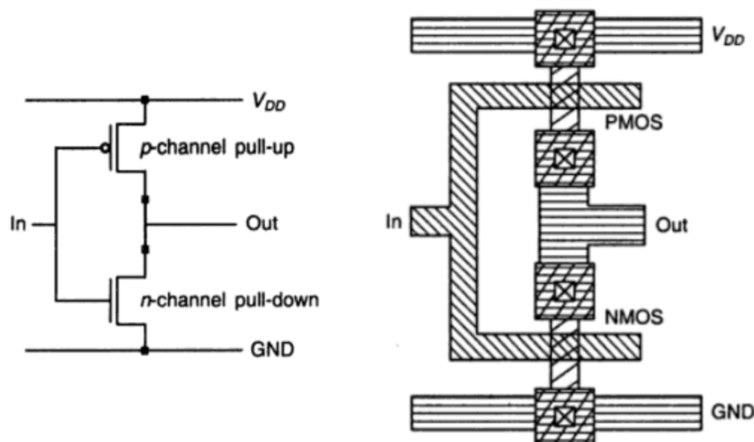

Abbildung 4.7: Schaltung und dazugehöriges Layout eines Inverters [RL08].

Wichtig dabei ist, dass das Layout bereits der tatsächlich erzeugten Halbleiterstruktur entspricht. Sämtliche Transistoren und Verbindungen bestehen aus mehreren verschieden dotierten Halbleitern oder Leitern. In der Abbildung sind diese durch unterschiedliche Schraffuren dargestellt. Weitere Informationen zu diesem Thema können in [RL08] nachgelesen werden.

Zusätzlich werden mehrere Simulationen und Analysen zur Verifikation der Funktionalität durchgeführt (siehe Kapitel 4.4). Besonders wichtig ist dabei die Kontrolle der geometrischen Abmessung mit Hilfe eines *Design Rule Check* (siehe Kapitel 4.4).

Die Daten für das Layout werden anschließend an den ASIC-Hersteller übermittelt, welcher in einem weiteren Syntheseschritt die sogenannten *Masken* erzeugt. Diese werden für die Fabrikation der Halbleiterstruktur verwendet.

Programmierbare Logikbausteine

Programmierbare Bausteine wie FPGAs besitzen bereits vorgefertigte logische Schaltungen. Diese Komponenten sind über zahlreiche Verbindungen (interconnections) miteinander verdrahtet (siehe Kapitel 3).
Eine Struktursynthese zu einer Transistorschaltung wird also nicht mehr benötigt. Ein weiterer Vorteil ist, dass durch den ASIC-Hersteller keine abschließende Fertigung der Halbleiterstrukturen nötig ist.

Die Gatter-Schaltung muss allerdings auf die vorhandene Logik angepasst werden. Dazu wird eine *Anordnen und Verdrahten (Place and Route)*-Synthese durchgeführt. Bei FPGAs werden dabei mehrere CLBs oder fertig eingebaute Logikkomponenten entsprechend miteinander verdrahtet. Als Ergebnis erhält man spezielle *Dateien für die Konfiguration (Bit-File)* der programmierbaren Bausteine.

4.3.8 Fabrikation

Input: Halbleiter-Layout mit Transistoren
Output: Rohchips (Dies) auf einer Siliziumscheibe (Wafer)

Als Grundmaterial werden bei der Fertigung sogenannte einkristalline *Siliziumscheiben (Wafer)* verwendet. Der ASIC-Hersteller erhält vom Kunden das fertige Layout der integrierten Schaltung. Daraus werden in einem weiteren Syntheseschritt die *Masken* für die Halbleiterherstellung erstellt. Masken definieren die Bereiche auf dem Wafer wo Halbleiterschichten dotiert, aufgebracht oder weggeätzt werden. Die dabei eingesetzten Techniken werden als *Lithographie* bezeichnet. Dadurch lassen sich die Transistoren und Verbindungen Schicht für Schicht erzeugen. Derzeit werden Wafer mit einem Durchmesser von bis zu 30 cm verwendet, wobei die Größe bis 2015 voraussichtlich auf 45 cm ansteigen wird.
Der Großteil aller ASICs basiert derzeit auf einem Complementary Metal Oxid Semiconductor (CMOS) Fertigungsprozess. Nähere Informationen zum CMOS-Prozess und zur Fabrikation können unter [RL08] nachgelesen werden.

Je nach Art der verwendeten Prozesstechnologie werden verschieden große Halbleiterstrukturen erstellt. Dadurch können hochintegrierte Schaltungen mit mehreren Millionen Transistoren entwickelt werden. Allerdings sind die Kosten für die Fertigung umso höher, je kleiner die Strukturen sind.

In der Regel besteht ein Wafer aus mehreren voneinander getrennten integrierten Schaltungen. Nach der Fertigung wird der ganze Wafer untersucht und getestet. Dabei wird nicht nur die Funktion der einzelnen Chips überprüft, sondern auch der physikalische Aufbau und die elektrischen Eigenschaften kontrolliert (siehe Kapitel 4.4).
Durch Zerschneiden des Wafers werden die fertigen *Rohchips (Dies)* erzeugt. Defekte oder über den Scheibenrand hinausstehende Chips werden dabei gleichzeitig aussortiert.

Je größer die Wafer sind, umso mehr Schaltungen können darauf erzeugt werden. Durch Fertigung eines einzigen Wafers kann sogar der Jahresbedarf eines ICs produziert werden. Dementsprechend führen Fertigungsfehler oder fehlerhafte Masken zu großen Ausfallraten und sehr hohen Kosten.

Für einen ASIC-Hersteller ist die Fabrikation in der Regel mit den größten Kosten verbunden. Minimale Fehler in Masken können den IC unbrauchbar machen. Zudem führen kleinste Ver-

unreinigungen zu schweren Fehlern. Entsprechend werden für die Herstellung hochintegrierter Schaltungen sehr komplexe Fertigungsanlagen benötigt.
Nicht jeder Hersteller kann jede Prozesstechnologie fertigen. So werden z.B. ASICs mit Strukturgrößen im Nanometerbereich nur von wenigen Unternehmen in großen *Fertigungsanlagen (Silicon Foundry)* hergestellt. Full-Custom ASICs, Gatearrays und Standardzellen ASICs werden somit vorwiegend für größere bis sehr große Stückzahlen produziert.

4.3.9 Verpacken

Input: Rohchips (Dies) auf einer Siliziumscheibe (Wafer)
Output: Fertig hergestellter IC mit Gehäuse

In der *Verpacken* Entwurfsphase werden die funktionierenden Rohchips (Dies) einzeln in spezielle Gehäuse eingebaut. Je nach Art der Anwendung, Anzahl der benötigten Anschlüsse und der Chipfläche stehen zahlreiche standardisierte Gehäuseformen und -materialien zur Verfügung.
Beim Einbau müssen die Ein- und Ausgänge des Rohchips und die Gehäuseanschlüsse (Pins) miteinander verbunden werden. Dies geschieht durch Anbringen sogenannter *Bond-Drähte (wire bonds)* oder *punktförmiger Verbindungen (bump connections)*. Den gesamten Ablauf bezeichnet man generell als *Bonding*. Ein Beispiel für die Verdrahtung mit Bond-Drähten ist aus Abb. 4.8 ersichtlich.

Abbildung 4.8: Verdrahtung eines Rohchips mit Bond-Drähten [Her10].

Jeder verpackte IC wird getestet und kontrolliert, da es beim Bonding bzw. Verpacken zu Fehlern kommen kann (siehe Kapitel 4.4). Das Gehäuse wird abschließend versiegelt, mit der Bauteilbezeichnung bedruckt und für den Transport verpackt.

4.3.10 Konfiguration

Input: Bit-File mit Konfigurationsdaten
Output: Konfigurierter (programmierter) IC

In der *Konfiguration* Entwurfsphase wird ein *programmierbarer Logikbaustein* so konfiguriert, dass er eine vorgegebene Funktion übernimmt. Diesen Vorgang bezeichnet man deshalb auch oft als Programmierung.
Dazu wird die gewünschte Funktionalität mit einer Entwicklungssoftware implementiert und die resultierende Schaltung mit einem spezifischen Compiler in eine *Konfigurationsdatei (Bit-File)*

umgewandelt. Der Inhalt der Konfigurationsdatei hängt dabei von der gewünschten Funktionsweise und der Architektur des programmierbaren Bausteines ab. In den letzten Jahren haben sich dafür einige standardisierte Dateiformate etabliert.

Die Konfiguration definiert die Verdrahtung der einzelnen internen Komponenten des Bausteines und wird in speziellen Konfigurationszellen (siehe Kapitel 3.1) gespeichert.
Die Programmierung der einzelnen ICs mit Hilfe spezieller externer Programmiergeräte wird immer seltener verwendet. Der Nachteil dabei ist, dass der IC nicht direkt im Endgerät programmiert werden kann und eine Aktualisierung (update) immer den nachträglichen Ausbau erfordert. Viele der programmierbaren Bausteine werden stattdessen im eingebauten Zustand über spezielle *In-System-Programmable (ISP)-Schnittstellen* programmiert. Dadurch kann die Funktionalität auch während des Betriebes leicht aktualisiert werden.

4.3.11 System Integration

Input:	Fertiger IC
Output:	Elektronische Schaltung, PCB

Unter *System Integration* versteht man den Einbau der fertigen ICs in elektronische Schaltungen. Dieser Entwurfsprozess wird deshalb auch als Schaltungs- bzw. Geräteentwurf bezeichnet. Dabei handelt es sich um einen eigenen speziellen Entwurfsprozess der zu einem fertigen Endprodukt führt und hier nicht näher beschrieben wird.

4.4 Verifikation und Analyse

Im Sinne eines interativen Prozesses (siehe Abschnitt 4.2.2) sind die Resultate bei allen Entwurfsphasen gründlich zu überprüfen. Dadurch können Fehler frühzeitig erkannt und die Folgekosten für Korrekturen verringert werden.
Je nach Entwurfsphase kommen spezielle Verifikationsmethoden zum Einsatz. Diese reichen von der Kontrolle physikalischer Fertigungsparameter bis hin zu einer Überprüfung der endgültigen Funktionalität.
Die ICs werden in der Regel, während der Entwicklung und Fertigung, genau auf sämtliche Parameter überprüft. Sobald die anfänglichen Probleme und Fehler behoben sind, wird im späteren Verlauf meist auf eine kostengünstigere Überprüfung durch Stichproben übergegangen.
In den nachfolgenden Abschnitten werden die wichtigsten Verifikationsmethoden, in Abhängigkeit der einzelnen Entwurfsphasen, beschrieben.

4.4.1 Überprüfung der Geometrie und Struktur

Bei der Fertigung der einzelnen Materialschichten (Layer) können generell zahlreiche Probleme auftreten. Je nach Dauer und Temperatur werden, beim Auftragen von Materialschichten auf die Wafer, unterschiedliche Schichtdicken erzeugt. Bei den Ätztechniken kann eine Abweichung dazu führen, dass zu wenig oder zu viel Material entfernt wird. Eine falsche oder ungenaue Ausrichtung der Masken führt zu schweren Fehlern, weil die einzelnen Schichten zueinander versetzt sind.
Während und auch nach der Fertigung müssen deshalb der geometrische Aufbau und die Struktur einer integrierten Schaltung genau überprüft werden.

Die Kontrolle der Schichtdicken erfolgt vorwiegend optisch durch Mikroskopie. Dazu werden sogenannte Brüche oder Schrägschliffe erzeugt. Weitere Methoden wie die *Ionenstrahl-Massenspektroskopie* oder die *Spreading Resistance Method (SRM)* ermöglichen zudem eine Bestimmung der

chemischen Zusammensetzung der Schichten. Zusätzlich können Schichten mit Röntgenstrahlung durchleuchtet werden.

Mikroskopie

Durch Erzeugen von Brüchen oder Schrägschliffen werden die Schichten der integrierten Schaltung sichtbar gemacht und anschließend mit geeigneten Mikroskopieverfahren überprüft. Neben optischen Mikroskopen kommen für kleinste Strukturen auch Elektronenmikroskope zum Einsatz. Mit Hilfe sogenannter *Raster-Elektronen-Mikroskope (REM)* können verschiedene Oberflächenstrukturen visualisiert werden. Im Gegensatz dazu können dünne Schichten mit *Transmissions-Elektronen-Mikroskopen (TEM)* durchleuchtet werden. Dies erlaubt zusätzliche Aufschlüsse über den Aufbau der Schichten.

Spreading Resistance Method

Mit Hilfe von Prüfspitzen wird der lokale Widerstand der zu prüfenden Schichten gemessen. Dabei wird um den Messpunkt im Material ein bestimmter Feldverlauf erzeugt, welcher Aufschluss über die Dotierung des Materials gibt.

Ionenstrahl-Massenspektroskopie

Zur *Analyse der chemischen Zusammensetzung* von Materialien können verschiedene Methoden der Ionenstrahl-Massenspektroskopie verwendet werden. Ein häufig angewendetes Verfahren zur Analyse dünner Schichten ist die *Secondary Ion Mass Spectrometry (SIMS)*. Bei dieser Technik schießt ein fokusierter Ionenstrahl sogenannte Sekundärionen aus der Oberfläche des Materials, welche anschließend im Massenspektrometer untersucht werden. Der Vorteil der SIMS ist die sehr hohe Empfindlichkeit im Bereich weniger ppb (part per billion).

4.4.2 Verifikation durch Simulation

Am Markt erhältliche Entwicklungswerkzeuge verfügen über eine immer umfangreicher werdende Funktionalität und ermöglichen nahezu jede erdenkliche Simulation. Gut funktionierende digitale Simulatoren gibt es schon seit längerer Zeit, diese sind sehr weit verbreitet. Für analoge Schaltungen ist die Simulation wegen der benötigten Bauteilmodelle meist etwas aufwändiger. In den gängigen Entwurfsmethoden kommen besonders die Architektur-, Logik- und Schaltungssimulation zur Anwendung.

Die *Architektursimulation (behavioural simulation)* wird in den frühen Entwurfsphasen eingesetzt. Dabei wird das System durch abstrakte bzw. funktionale Blockdiagramme beschrieben. Die Eingabe erfolgt grafisch oder mit Hilfe einer geeigneten Programmiersprache. Als Blöcke werden z.B. Register, Speicher oder ALUs verwendet. Deshalb nennt man diese Art von Simulatoren auch gerne *Register Transfer Level Simulatoren.*

Als *Logiksimulation (logic simulation)* bezeichnet man die Simulation digitaler Logikschaltungen. Das Hauptaugenmerk wird dabei auf das zeitliche Verhalten der Grundgatter und Flip-Flops gelegt. Dies wird durch einfache Modelle der Gatter mit Anstiegs-, Abfalls- und Verzögerungszeiten erreicht. Als Ergebnis erhält man eine abstrakte zeitliche Darstellung der simulierten Ausgangssignale.

Bei der *Schaltungssimulation (circuit simulation)* werden hauptsächlich Schaltungen mit aktiven und passiven Bauelementen simuliert. Dazu sind komplexe Modelle der einzelnen Bauteile

nötig. Basierend auf einer meist grafischen Eingabe wird eine Netzliste erstellt und die Spannungen bzw. Ströme mit den Kirchhoffschen Gesetzen berechnet. Unterschiedliche Analysearten ermöglichen die Simulation des Zeit- und Frequenzverhaltens unter Einbeziehung verschiedenster Bauteiltoleranzen. Das verbreitetste Programm zur Schaltungssimulation ist *SPICE*.

4.4.3 Layout-Überprüfung

Jedes Layout muss vor der Fertigung auf seine korrekte Funktion überprüft werden. Bereits kleinste Fehler, welche erst nach der Fertigung des Wafers entdeckt werden, können zu erheblichen Kosten führen und sollten unbedingt verhindert werden. Dazu werden mehrere Methoden verwendet, die nachfolgend aufgelistet sind.

Electrical Rule Check

Zur Prüfung der Schaltung auf Kurzschlüsse, Leerläufe oder einer fehlerhaften Verdrahtung wird der sogenannte *Electrical Rule Check (ERC)* eingesetzt. Aus dem Layout wird dazu die äquivalente Schaltung extrahiert und als Netzliste gespeichert. Je nach Detailgrad der Extraktion, erfolgt eine Abbildung auf der Logik-, Gatter- oder Transistorebene. Mit Hilfe der erzeugten Netzliste können zudem auch eine LVS Überprüfung oder eine SPICE Simulation durchgeführt werden.

Design Rule Check

Beim *Design Rule Check (DRC)* wird das fertige Layout auf die Einhaltung spezieller geometrischer Abmessungen überprüft. Die Prüfkriterien umfassen vorwiegend bestimmte Abstände, Dicken, Breiten oder Flächen der vorhandenen Strukturen und hängen stark von der verwendeten Prozesstechnologie ab.

Layout versus Schematic

Die *Layout versus Schematic (LVS)* Überprüfung dient zur Kontrolle der Konsistenz zwischen Layout und Schaltung. Ähnlich wie beim ERC wird auch hier eine aus dem fertigen Layout extrahierte Netzliste verwendet. Mit LVS kann überprüft werden, ob bei der Synthese oder händischen Layouterstellung eine Inkonsistenz entstanden ist.
Oft wird anstelle von LVS auch der Begriff *Network Consistency Check (NCC)* verwendet.

4.4.4 Vereinfachte Verifikation durch Teststrukturen

Jede integrierte Schaltung muss mindestens einmal während und nach der Fertigung getestet werden. Bei fertigen ICs ist meist nur das Verhalten der Ein- und Ausgänge beobachtbar und deshalb keine exakte Aussage über den internen Zustand der Register möglich. Durch Einbau bestimmter *Teststrukturen* können alle internen Komponenten von außen zugänglich gemacht werden.
Dies ist ein wesentlicher Vorteil gegenüber herkömmlichen externen Messungen. Nachteilig wirkt sich lediglich die zusätzlich benötigte Chipfläche aus. In der Praxis wird jedoch, wegen der erheblichen Vorteile, ein Zuwachs der Chipfläche von bis zu 10% akzeptiert.

Self-Test bzw. Build-In Test

Eine weitere Möglichkeit ist der Einsatz sogenannter *Self-Test (ST)* oder *Build-In Test (BIT)* Schaltungen. Dabei handelt es sich im wesentlichen um eine interne Zusatzlogik zur Generierung

und Analyse bestimmter Testmuster. In der Regel werden dafür *rückgekoppelte Schieberegister* verwendet.
Der Vorteil von BIT liegt darin, dass die Überprüfung der integrierten Schaltung regelmäßig und automatisch durchgeführt werden kann. Dies kann z.B. nach dem Einschalten geschehen oder während sich der IC in einem Wartezustand (idle) befindet.

In-Circuit Test

Als *In-Circuit Test (ICT)* bezeichnet man die Überprüfung der integrierten Schaltung mittels mehrerer auf der Platine vorhandener Messpunkte. Diese sind in der Regel mit den Anschlüssen des ICs verbunden. Die eigentliche Messung erfolgt über kleine Federstifte.

Auf Grund des immer höheren Integrationsgrades moderner Schaltungen werden für eine vollständige Überprüfung sehr viele Messpunkte benötigt. Für die größten ICs, mit derzeit bis zu zweitausend Anschlüssen, würde die Gesamtfläche der einzelnen Messpunkte bereits viel größer sein, als die Chipfläche. Ein ICT ist oft nur bei der Fertigung möglich, da im eingebauten Zustand nicht mehr genügend Platz für die Prüfung der Platine vorhanden ist. Diese Nachteile führen dazu, dass ICTs immer mehr durch andere Methoden wie JTAGBS ersetzt werden.

JTAG/Boundary-Scan

Der von der *Joint Test Action Group (JTAG)* entwickelte *Boundary-Scan (BS)* IEEE 1149.x Standard ist das in den letzen Jahren meist eingesetzte All-in-One System zur Verifikation von integrierten Bausteinen. Hauptziel bei der Entwicklung war die Aufhebung der Beschränkungen gängiger ICTs. Der Boundary-Scan-Standard stellt somit eine Erweiterung zu anderen Verifikationsmethoden dar, dies ist aus Abb. 4.9 ersichtlich.

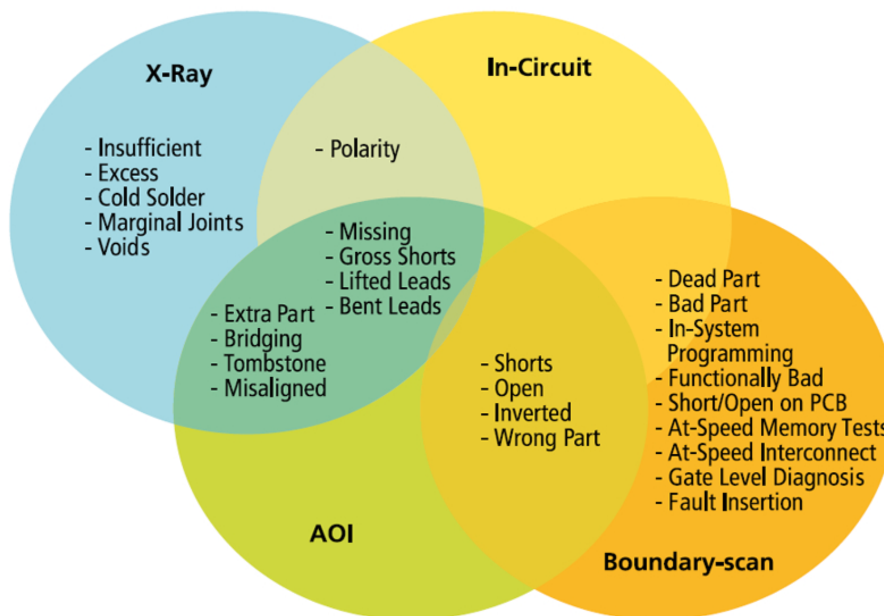

Abbildung 4.9: Zusammenhang der verschiedenen Verifikationsmethoden [JTA08].

Moderne programmierbare Bausteine sind häufig JTAG/BS konform. Die Kommunikation erfolgt über den seriellen *Test Access Port (TAP)*, welcher aus vier bzw. fünf Signalleitungen

besteht. Der TAP ermöglicht es den Baustein, mit Hilfe einer entsprechenden Software, zu prüfen und auch zu programmieren.

Ein spezieller *TAP Controller* übernimmt im Baustein die Steuerung und Überwachung des Boundary-Scans. Dieser verarbeitet alle eingehenden Befehle und retourniert die Ausgangsdaten. Zusätzlich wird für jeden Bausteinanschluss eine BS-Zelle benötigt. Der Aufbau einer integrierten Schaltung mit eingebauter JTAG/BS Logik kann aus Abb. 4.10 entnommen werden.

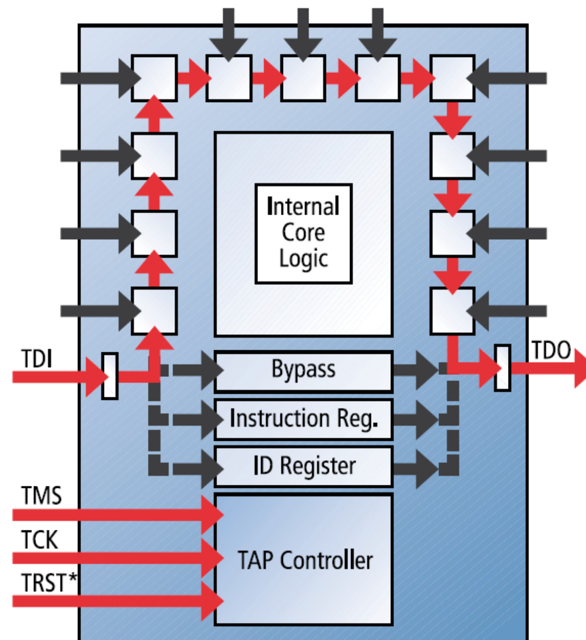

Abbildung 4.10: Aufbau eines IC mit eingebauter JTAG/Boundary-Scan Logik [JTA08].

Nähere Informationen zum JTAG/Boundary-Scan sind als BS Tutorial bei [JTA08] verfügbar.

5 FPGAs in extraterrestrischen Umgebungen

In den letzten Jahrzehnten ist die Eroberung des Weltalls stetig vorangeschritten. Dies wurde größtenteils durch neue Erkenntnisse im Bereich der Naturwissenschaften und der Technik ermöglicht. In den Anfangsjahren kamen vorwiegend analoge Schaltungen, in Raketen und Satelliten, zum Einsatz. Neben dem hohen Gewicht, der begrenzten Rechenleistung und einer hohen Leistungsaufnahme war eine komplexe Verdrahtung der On-Board Systeme nötig.
Diese wesentlichen Nachteile haben dazu geführt, dass in den letzten Jahren immer leistungsfähigere digitale Systeme verwendet werden. Dank ihrer hohen Rechenleistung und des geringen Gewichtes sind immer komplexere Systeme realisierbar. Programmierbare Logikbausteine sind zudem preiswert und ermöglichen im Betrieb eine Änderung der Funktionalität.

Die in einer *extraterrestrischen Umgebung (Weltall)* herrschenden Bedingungen unterscheiden sich wesentlich von denen auf der Erdoberfläche. Die Erdatmosphäre filtert bzw. verringert die für uns gefährliche Strahlung und schützt so vor zahlreichen schädlichen Einflüssen. Sie dient zudem als Wärmespeicher und ermöglicht eine relativ konstante Temperatur.
Durch Protuberanzen der Sonne werden unaufhörlich geladene Teilchen in unser Sonnensystem geschleudert. Diese Teilchenstrahlung wird auch als *kosmische Strahlung* bezeichnet. Es handelt sich dabei um Ionen und Elementarteilchen. Einige davon werden in den Strahlengürteln der Erde eingefangen oder gelangen über die Polkappen in die Atmosphäre. Die kosmische Strahlung kann außerhalb der schützenden Atmosphäre zum Ausfall integrierter Bausteine führen. Zusätzlich entsteht, durch den Kernfusionsprozess der Sonne, eine erhebliche Menge Gamma-Strahlung und anderer elektromagnetischer Wellen. Diese können sich ebenfalls negativ auf die Funktionsweise integrierter Bausteine auswirken. Dementsprechend stellt die Entwicklung zuverlässiger FPGAs für Luft- und Raumfahrtanwendungen eine permanente Herausforderung dar.

In den nachfolgenden Abschnitten werden die strahlungsbedingten Auswirkungen auf die Funktionsweise eines FPGAs im Detail erläutert.

5.1 Auswirkungen erhöhter Strahlung

Die im Weltall vorhandene kosmische Teilchenstrahlung oder eine elektromagnetische Strahlung kann zu einer Funktionsstörung im FPGA führen. Gegebenenfalls ist sogar eine irreversible Zerstörung der Halbleiterstrukturen möglich. Die Strahlungsanfälligkeit von FPGAs beruht auf einigen wesentlichen physikalischen Grundprinzipien. Die einzelnen Effekte hängen im Detail oft von der verwendeten Prozesstechnologie und der Architektur des integrierten Bausteines ab. Es folgt daraus, dass sämtliche Erscheinungen auch bei anderen CMOS Bausteinen beobachtet werden können.

Im Allgemeinen werden die Auswirkungen in mehrere Gruppen unterteilt. Sogenannte *Single Event Effects* beschreiben Störungen durch kurze Einzelereignisse. Dies wird dadurch hervorgerufen, dass einzelne energiereiche Teilchen auf das Halbleitermaterial aufschlagen und es durchdringen. In Abhängigkeit der betroffenen Schaltungsteile kann es zu Datenfehlern, Änderungen in der Konfiguration oder sogar zum kompletten Ausfall des FPGAs kommen.

Mit den *Total Ionization Dose*-Effekten können die längerfristigen Auswirkungen auf das Halbleitermaterial erklärt werden. Dabei handelt es sich im wesentlichen um die strahlenbedingte Entstehung von Löchern im Gate-Oxid, welche die Schwellspannung der Transistoren verändern. Damit können auch die im Feld-Oxid entstehenden Leckstrompfade, zwischen den Source- und Drain-Anschlüssen, erklärt werden.

Weitere Informationen zu diesem Thema können bei [Xil04], [SF04], [Ste08] und [Caf02] nachgelesen werden.

5.2 Beeinflussung durch langanhaltende Strahleneinwirkung

Unter dem Begriff *Total Ionization Dose (TID) Effects* versteht man Effekte, welche durch eine langanhaltende Einwirkung der kosmischen Strahlung hervorgerufen werden. Dabei handelt es sich vorrangig um Ionen und andere energiereiche Teilchen, wie z.B. Protonen und Neutronen. Die nachfolgenden Erläuterungen werden anhand eines selbstsperrenden N-Kanal-Feldeffekttransistors durchgeführt.

Beim Auftreffen von Ionen auf das Gate-Oxid eines Feldeffekttransistors (FET) werden mehrere Elektronen-Loch-Paare erzeugt. Die positiven Löcher wandern in Richtung des Kanals (hole transport) und setzen sich dort an bestimmten Stellen, den sogenannten *deep hole traps*, fest. Das Festsetzen wird demnach als *deep hole trapping* bezeichnet. Dies ist aus Abb. 5.1 ersichtlich.

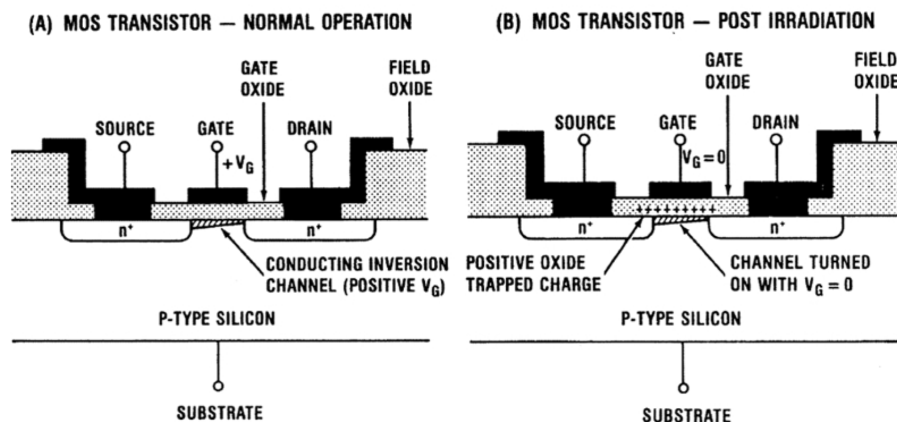

Abbildung 5.1: Ansammlung positiver Ladungsträger im Gate-Oxid [TRO03].

Die Löcher im Gate-Oxid sorgen bei einem selbstsperrenden N-Kanal Feldeffekttransistor (n-channel enhancement FET) für eine *Absenkung der Schwellenspannung* (threshold voltage, V_{TH}). Dies führt dazu, dass der FET bereits bei einer kleineren Gate-Source Spannung V_G leitend wird. Folglich kann es zu einem Fehler in der entsprechenden Schaltung kommen.
Bei der Festsetzung der Löcher und der damit verbundenen Verschiebung der Schwellenspannung handelt es sich um einen nicht umkehrbaren Effekt. Eine Rekombination findet zwar statt, diese hängt aber von mehreren Einflüssen wie z.B. der Temperatur ab und kann mehrere Jahre dauern.

Als zweiten Effekt kommt es auf Grund der Ionenstrahlung ebenfalls zu Veränderungen im dicken Feld-Oxid (field oxid), welches die einzelnen Transistoren voneinander trennt. Dabei entstehen *Leckstrompfade* zwischen den Source- und Drainanschlüssen (edge-leakage) oder den durch das

Feld-Oxid getrennten n^+-Anschlüssen (field-leakage). Der Verlauf des edge-leakage Leckstromes wird in Abb. 5.2 dargestellt.

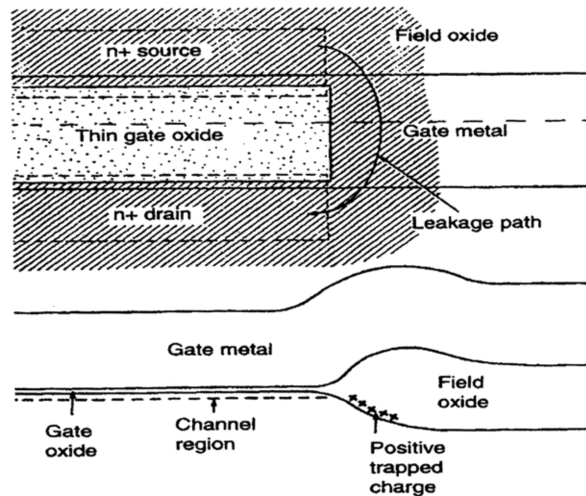

Abbildung 5.2: Pfad des Leckstromes zwischen Source und Drain [TRO03].

Da das Feld-Oxid im Vergleich zum Gate-Oxid sehr viel dicker ist, kommt es bei der gleichen Strahlendosis zu einer viel stärkeren Verschiebung der Schwellenspannung. Durch den resultierenden Leckstrom kann es im schlimmsten Fall zu einer Zerstörung des FET kommen. Die tatsächliche Stärke des Effektes hängt von zahlreichen Faktoren wie der Oxiddicke, der Temperatur und der Materialeigenschaften ab. Aus Abb. 5.3 sind die Auswirkungen der TID-Effekte auf die Eingangskennlinie des Transistors nochmals ersichtlich.

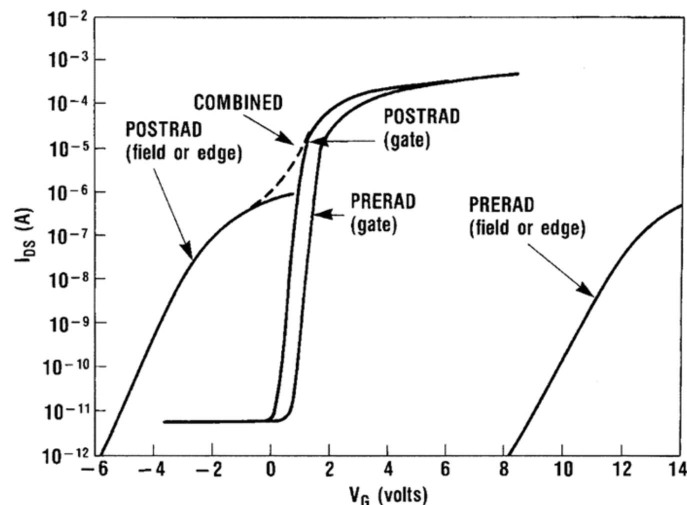

Abbildung 5.3: Verschiebung der Eingangskennlinie durch Leckströme [TRO03].

TID-Effekte können in jedem FPGA auftreten der auf einem CMOS-Prozess basiert. Eine vollständige Abschirmung der energiereichen Teilchen ist nicht möglich, da dies sehr aufwändig wäre. Einige FPGAs für Luft- und Raumfahrtanwendungen werden deshalb von den Herstellern

speziell konstruiert, um einen erhöhten Schutz vor der Strahlung zu gewährleisten. Beispielsweise reagiert die Silicon-On-Insulator (SOI) Technologie wesentlich unsensibler auf Strahlung, als gewöhnliche CMOS-Prozesse.
Weitere Informationen zu diesem Kapitel können in [TRO03] nachgelesen werden.

5.3 Störungen durch einzelne energiereiche Teilchen

Durch einzelne energiereiche Teilchen erzeugte Störungen werden in der Literatur meist als *Single Event Effects (SEE)* bezeichnet. Diese werden vorwiegend durch schwere Ionen hervorgerufen, welche auf das Halbleitermaterial aufschlagen und es durchdringen. Als Folge der SEE kommt es meist zu einer sofortigen Funktionsstörung im FPGA. Die einzelnen Effekte sind aus Abb. 5.4 ersichtlich.

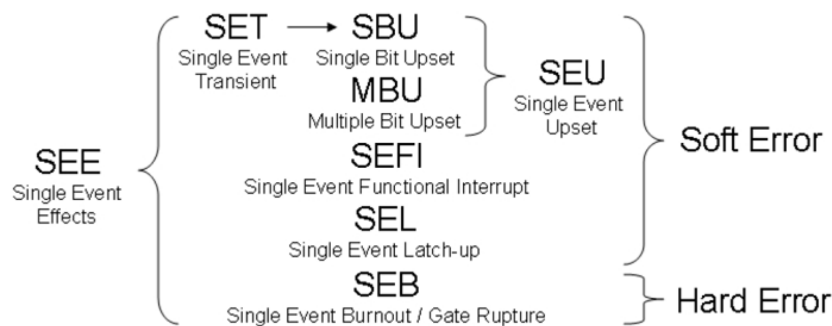

Abbildung 5.4: Übersicht der einzelnen SEE-Effekte [Bau05].

Prinzipiell beeinflussen alle SEE die Funktionsweise der Feldeffekttransistoren, welche als Grundelemente für die FPGAs verwendet werden. Die Häufigkeit der Ausfälle hängt in der Praxis von der verwendeten Prozesstechnologie und der Halbleiterstruktur der FETs ab.
Sämtliche Effekte können in sogenannte *Soft Errors* und *Hard Errors* unterteilt werden. Die Soft Errors beschreiben funktionale Störungen, welche z.B. zu Fehlern in den gespeicherten Daten oder einem Datenverlust führen. Die Hardware wird dadurch nicht beschädigt. Als Hard Errors werden permanente nicht korrigierbare Fehler bezeichnet, welche zu einer Beschädigung des FPGAs führen.

Die wichtigsten SEE-Effekte sind der *Single Event Upset (SEU)*, *Single Event Latchup (SEL)* und der *Single Event Functional Interrupt (SEFI)*. Dabei handelt es sich um Soft Errors. Der *Single Event Burnout (SEB)* ist hingegen ein Hard Error. Eine detaillierte Beschreibung erfolgt in den nachfolgenden Abschnitten.

5.3.1 Single Event Upset

Als *Single Event Upset* bezeichnet man den Ausfall eines FPGAs durch Veränderung der Konfiguration (configuration upset) oder der Daten in einem Speicherelement (data upset). Im Wesentlichen kann ein SEU auf die Beeinflussung eines einzelnen FETs zurückgeführt werden. Die verschiedenen Speicher der FPGAs sind davon besonders stark betroffen, da diese in der Regel viele FETs enthalten. Die MTBF (Mean Time Before Failure) für einen data upset liegt laut [Xil08b] bei wenigen Stunden.

Die Wahrscheinlichkeit für das Auftreten von Fehlern in der Konfiguration hängt vom Typ der verwendeten Konfigurationszellen ab. Besonders SRAM-Zellen (Static RAM) sind sehr anfällig für einen configuration upset, da diese aus typischerweise vier bis sechs FETs bestehen. Ein Ausfall kann in seltenen Fällen bereits bei geringer Strahlung, wie sie z.B. auf der Erdoberfläche herrscht, eintreten.
Flash-Zellen sind weniger empfindlich, weil sie nur aus einem einzelnen Floating Gate FET bestehen. Im Gegensatz dazu geht die Wahrscheinlichkeit eines configuration upsets bei Antifuse-Zellen gegen Null.

Bei FETs sind besonders die Drain-Anschlüsse sensibel auf Störungen, da sich dort ein p^+n bzw. ein n^+p-Übergang zum Substrat bzw. der Wanne ausbildet. Ein energiereiches Teilchen, typischerweise ein Ion, gibt beim Durchdringen des Halbleitermaterials Energie ab und erzeugt dadurch Elektronen-Loch-Paare. Als Folge entsteht eine *Ionenspur (ion track)* im FET, welche die Verarmungszone (depletion region) durchdringt. Anschließend beginnt der *Prozess der Ladungssammlung (charge collection).*
Die Sammlung der Ladungen und der daraus resultierende Verlauf des Stromimpulses sind in Abb. 5.5 dargestellt.

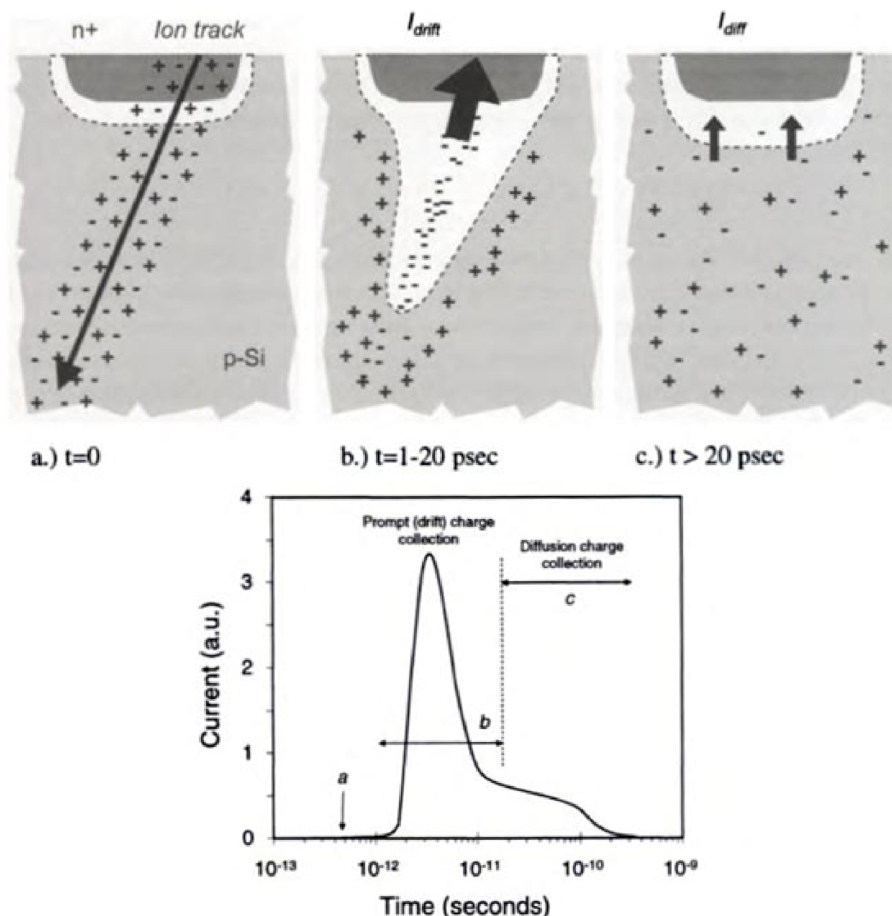

Abbildung 5.5: Auswirkung eines energiereichen Teilchens auf einen N-Kanal-FET [SF04].

Als *Drift* bezeichnet man das rasche Einsammeln der freien Ladungsträger durch das entstandene elektrische Feld. Dabei entsteht ein kurzer Stromimpuls in Richtung des n^+-Gebietes. Nachdem

die meisten Ladungen eingesammelt sind, verbleiben einige Elektronen-Loch-Paare im Substrat bzw. der Wanne. Die verbleibenden freien Ladungsträger werden durch *Diffusion* langsam zum n^+-Gebiet transportiert.

Ein solcher Stromimpuls kann von einem Flip-Flop detektiert werden und den Zustand der Schaltung ändern. Als Folge können Bits in Registern verändert werden, wodurch Datenfehler entstehen. In Konfigurationszellen kann der Stromimpuls zu einer Veränderung der Verbindungen führen, wodurch Kurzschlüsse bzw. Leerläufe entstehen. Zudem kann es dadurch zu Fehlern in den LUTs der Logikzellen kommen.

Nähere Informationen zu SEU können bei [SF04], [Caf02] und [Gra02] nachgelesen werden. Methoden zur Vermeidung oder Behebung dieser Störungen werden in Kapitel 5.4 behandelt.

5.3.2 Single Event Latchup

Beim Einsatz sogenannter *Half-Latches* sind ebenfalls Ausfälle durch einzelne energiereiche Teilchen möglich. Half-Latches werden zur Vorgabe konstanter Bit-Werte verwendet, welche von anderen Schaltungskomponenten als Eingangswert benötigt werden. Ein Beispiel für den Ausfall einer Schaltung, auf Grund einer Störung in einem Half-Latch, ist aus Abb. 5.6 ersichtlich.

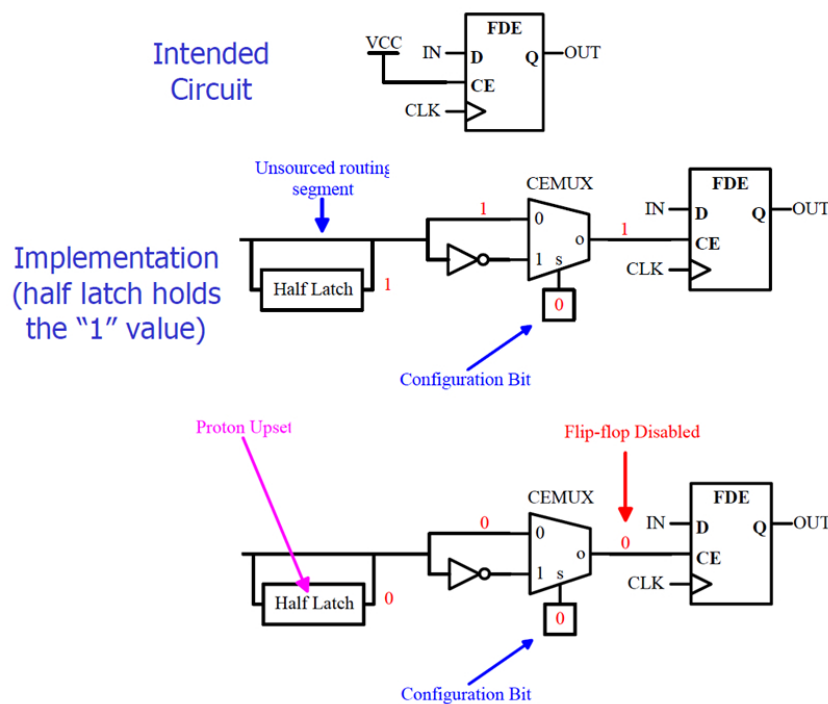

Abbildung 5.6: Beispiel für einen Ausfall durch einen Half-Latch [Caf02].

Das Beispiel zeigt ein Daten-Flip-Flop (D-FF) mit einem Clock-Enable (CE) Eingang. Durch Anlegen eines High-Signals am CE-Eingang wird das Flip-Flop aktiviert, damit es bei jeder steigenden Taktflanke (CLK) den Wert am Dateneingang (D) übernimmt. Das konstante High-Signal wird mit einem Half-Latch erzeugt. Ein energiereiches Teilchen kann dazu führen, dass der Ausgangswert des Half-Latches invertiert wird. Dadurch wird das FF deaktiviert und es kommt zum Ausfall der gesamten Schaltung.

Bei neuen FPGA-Familien ist dieses Problem mittlerweile weitgehend behoben. Anstelle der Half-Latch-Strukturen werden weniger kritische Schaltungen eingesetzt. Durch Verwendung geeigneter Synthesizer können die Half-Latches bei der Entwicklung automatisch durch andere Strukturen ersetzt werden. Im Fehlerfall kann das Problem durch eine erneute Programmierung des FPGAs behoben werden.

5.3.3 Single Event Functional Interrupt

Als *Single Event Functional Interrupt* bezeichnet man Störungen in der Kontrollhardware des FPGAs oder einer implementierten Zustandsmaschine. Dadurch kommt es zu einem Ausfall des gesamten Systems.

Bei Xilinx Virtex-4 FPGAs sind SEFIs in den folgenden Komponenten wahrscheinlich [Xil09a]:

- JTAG TAP Controller
- Frame Address Register (FAR)
- Configuration Control FSM
- SelectMAP Konfigurationsschnittstelle

Ein SEFI kann nicht mit dem FPGA direkt detektiert werden, weil dieser ausfällt. Dies ist nur von außen mit einer dafür vorgesehenen Hardware möglich, welche unter Umständen auch zur Behebung des SEFI verwendet werden kann. Eine effiziente Möglichkeit zum Schutz vor einem SEFI ist die Verwendung dreifach redundanter Hardware (siehe Abschnitt 5.4.5).

5.3.4 Single Event Burnout/Gate Rupture

Ein *Single Event Burnout* tritt dann auf, wenn es zum Zünden des für FETs typischen parasitären Thyristors kommt. Dadurch entsteht ein permanenter Kurzschluss zwischen Drain und Source. Dies führt zur thermischen Zerstörung des Bauteils.
Wird hingegen der Gate-Anschluss mit dem Kanal kurzgeschlossen, dann bezeichnet man dies als *Single Event Gate Rupture (SEGR)*. Dieser Effekt führt ebenfalls zur thermischen Zerstörung des FET. Somit handelt es sich bei beiden Effekten um Hard Errors.

5.4 Vermeidung und Behebung von Single Event Effekten

In den letzten Jahren wurden einige wichtige Techniken zur *Vermeidung von Ausfällen* auf Grund eines Single Event Effect entwickelt. Diese werden in der englischen Literatur allgemein als *mitigation techniques* bezeichnet. Bestimmte Verfahren ermöglichen zudem die *Erkennung und Behebung* bereits aufgetretener SEE.

Die *Erkennung von SEE* kann prinzipiell auf mehrere Arten erfolgen. Durch das zyklische *Auslesen der Konfiguration (readback)* und dem Vergleich mit den ursprünglichen Daten können Ausfälle der Konfigurationszellen (configuration upsets) detektiert werden. Die meisten Datenfehler (data upsets) sind ebenfalls detektier- und korrigierbar, sofern redundante Systeme zum Einsatz kommen.
Die gängigsten Maßnahmen zur *Vermeidung von Ausfällen* sind die *Erzeugung von redundanten Softwaremodulen (triple module redundancy)*, die periodische *Auffrischung der Konfiguration (Scrubbing)* und der *Einsatz redundanter Hardware*. Eine Übersicht der einzelnen Maßnahmen ist aus Abb. 5.7 ersichtlich.

SEU Rate (Low → High)	Operating Window			
Data Criticality		Low → High		
Error Persistence		No	Yes	
	Minutes	No Mitigation		XTMR
	Days	Scrubbing	Scrubbing XTMR	Redundant Devices
	Months	Scrubbing	Scrubbing XTMR	Redundant Devices
	Continuous	Scrubbing	Scrubbing XTMR	Redundant Devices

Abbildung 5.7: Übersicht der gängigen Mitigation Techniken [Xil08b].

Ein wesentlicher Faktor beim Schutz der FPGAs vor Strahlung ist der richtige Aufbau der Halbleiterstrukturen. FPGAs mit kleinen Strukturen, wie sie in neueren FPGA-Familien vorkommen, reagieren im Vergleich zu Modellen mit größeren Strukturen meist empfindlicher. Zudem spielen die verwendeten Materialien und das Gehäuse eine wesentliche Rolle. Die entsprechenden Techniken werden in der Literatur als *Radiation Hardening* bezeichnet.

5.4.1 Dreifach redundante Module

Eine erfolgreiche Maßnahme zur Vermeidung von Ausfällen ist die Erzeugung redundanter Systeme. Dazu werden drei identische Module im FPGA erzeugt. Diese Technik bezeichnet man als *Triple Module Redundancy (TMR)*.

Die redundanten Module können für einzelne Teilsysteme oder auch das Gesamtsystem erstellt werden. Die Ausgänge der einzelnen Module werden mit einem *Voter* verglichen. Dadurch kann festgestellt werden, ob in einem Modul eine Störung durch einen Single Event Effect (siehe Kapitel 5.3) aufgetreten ist. Der Voter kann den Fehler kompensieren, indem er die Ausgänge eines der funktionierenden Module weiterleitet. Der Aufbau eines solchen TMR-Systems ist in Abb. 5.8 dargestellt.

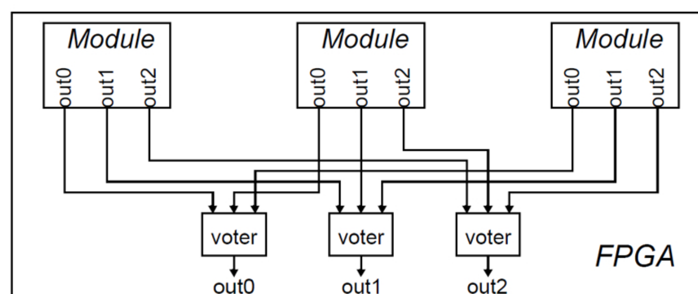

Abbildung 5.8: Triple Module Redundancy (TMR)-System mit Voter [Hab02].

Der Entwurf eines TMR-Systems ist in der Regel aufwändig, weil die Schaltungen dadurch komplexer und größer werden. Seit einiger Zeit stehen jedoch Programme zur Verfügung die eine

automatisierte Erzeugung ermöglichen. Als Beispiel ist hier das Xilinx *TMRTool* zu erwähnen mit welchem aus einem nichtredundanten System ein TMR-System synthetisiert werden kann. Weitere Informationen zum TMRTool sind unter [Xil09d] verfügbar.

Der Nachteil von TMR-Systemen besteht jedoch darin, dass die Störung im Modul durch die Voter nicht behoben wird. Ein Ausfall wird erst dann erkannt, wenn sich dieser auf die Ausgänge auswirkt. Der Voter verwendet dann die Ausgangswerte eines funktionierenden Modules. Dies funktioniert aber auch nur dann, wenn maximal eines der drei Module fehlerhaft ist. Der eigentliche Ausfall kann meist nur durch das Rücksetzen der Hardware (Reset) behoben werden, wodurch der FPGA zeitweise funktionsunfähig ist.
Ein weiterer Nachteil besteht darin, dass das TMR-System dreimal so viel Platz im FPGA benötigt. Zudem kann auch der Voter selbst durch einen SEE beeinflusst werden. Somit ist ein zusätzlicher Schutz des Voters notwendig.

5.4.2 Redundanz auf Gatter-Ebene

Die in FPGAs realisierten Module bestehen für gewöhnlich aus kombinatorischer und sequentieller Logik. Anstelle redundanter Module (TMR) können *redundante Schaltungen auf Gatter-Ebene* erzeugt werden. Dadurch wird ein direkter Schutz der programmierten Logik möglich. Die Struktur eines redundanten Systems auf Gatter-Ebene kann aus Abb. 5.9 entnommen werden.

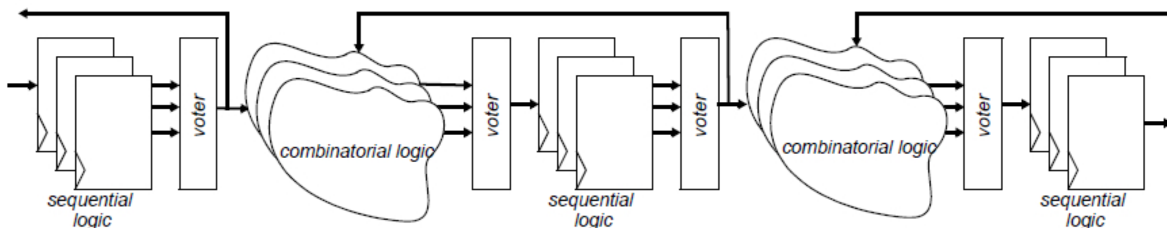

Abbildung 5.9: Redundanz der Logikkomponenten auf Gatter-Ebene [Hab02].

Die Resultate der Voter werden regelmäßig auf die kombinatorische Logik zurückgekoppelt. Dadurch wird der Zustand der redundanten Komponenten nach einer Störung wiederhergestellt und die Ausbreitung des Fehlers verhindert.
Zusätzlich sollten auch die Voter redundant realisiert werden. Der Aufbau einer solchen Schaltung wird in Abb. 5.10 dargestellt.

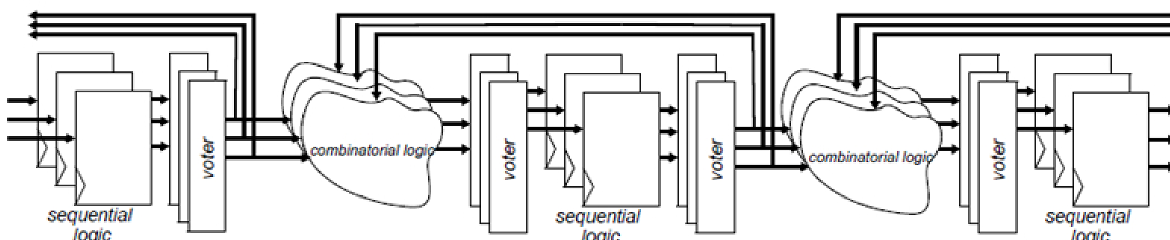

Abbildung 5.10: Störungskorrektur durch Rückkopplung auf Gatter-Ebene [Hab02].

5.4.3 Rekonfiguration von FPGAs (Scrubbing)

Als *Readback* bezeichnet man das Auslesen der im FPGA gespeicherten Konfiguration. Die ausgelesenen Daten werden anschließend mit den Originaldaten verglichen, um mögliche SEUs zu detektieren.
Readback ist die einfachste Methode zur Detektion und die Voraussetzung für eine erfolgreiche Korrektur. Bei einem aufgetretenen SEU wird der entsprechende Bereich (frame) neu konfiguriert. Dadurch fällt der FPGA nur für die kürzest mögliche Zeit aus und die Kontrolllogik der Konfigurationsschnittstelle bleibt meist im Lese-Modus.

Ein weiteres Verfahren wird als *Scrubbing* bezeichnet. Dabei wird die Konfiguration in regelmäßigen Abständen komplett erneuert. Der Vorteil ist, dass man sich die Detektion der SEUs und die dafür benötigte Logik erspart. Nachteilig wirkt sich jedoch die Tatsache aus, dass sich die Kontrolllogik größtenteils im Schreib-Modus befindet und somit anfälliger für einen SEU ist.

Für beide Techniken wird ein eigener Rekonfigurations-Controller benötigt. Dieser liest die Originaldaten aus dem Speicher und schreibt die Konfiguration in den FPGA. Der Controller kann entweder im FPGA selbst oder in einem zweiten externen Baustein realisiert werden. Moderne FPGAs sind mit Komponenten für die Rekonfiguration ausgerüstet. Zudem liefern einige Hersteller vorgefertigte Implementierungen und Hochsprachenmodelle. Die Controller können ebenfalls für die Erkennung eines SEFI benutzt werden. Der Aufbau der resultierenden Schaltung ist aus Abb. 5.11 ersichtlich.

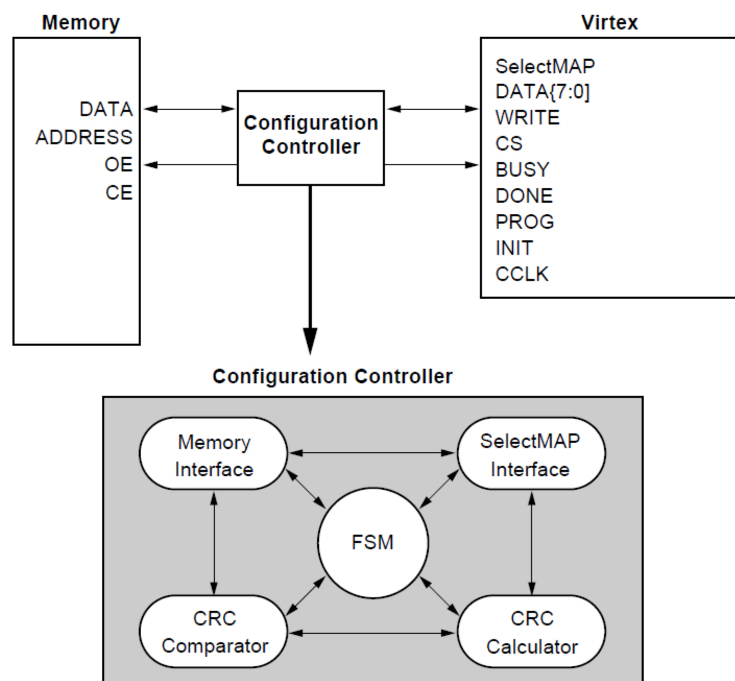

Abbildung 5.11: Realisierung der Readback- oder Scrubbing-Controller [Xil00].

Scrubbing und Readback können jedoch nur zur Detektion und Behebung bereits aufgetretener Störungen verwendet werden. Diese Verfahren eignen sich somit nur zum Schutz von Anwendungen mit relativ unkritischen Aufgaben (siehe Abb. 5.7). Für kritische Anwendungen müssen andere Techniken wie eine dreifach redundante Hardware verwendet werden.

Ein zusätzliches Problem kann bei der Verwendung von FPGAs mit flüchtigen (volatile) SRAM-Konfigurationszellen auftreten. Die Konfigurationsdaten müssen dann bei jedem Systemstart neu in den FPGA geschrieben werden. Die Daten werden dazu in einem eigenen Speicherbaustein hinterlegt. Durch einen SEE kann es zu Fehlern bei der Übertragung und damit zu einem Ausfall des FPGA kommen. Die meisten Systeme verwenden deshalb einen Cyclic Redundancy Check (CRC)-Algorithmus zur Kontrolle der Daten. Eine weitere Möglichkeit ist die Verwendung von Fehlerkorrekturverfahren. Zudem kann es durch einen SEE zum Ausfall des Speicherbausteines kommen. Aus diesem Grund werden spezielle strahlungsresistente Speicher verwendet.

Nähere Informationen zu diesem Thema können in [Xil10c] nachgelesen werden.

5.4.4 Kombination von TMR und Rekonfiguration

Zahlreiche Raumfahrtsysteme verwenden für ihre FPGAs eine Kombination aus TMR und Rekonfiguration (TMR Scrubbing). Durch TMR können Störungen durch SEU erkannt und deren Ausbreitung verhindert werden. Durch Scrubbing werden die einzelnen Blöcke der Konfiguration schrittweise wiederhergestellt.

Durch die Kombination beider Techniken können nahezu alle Störungen behoben werden. Diese Maßnahmen sind jedoch wirkungslos, wenn es zu einer Zerstörung des FPGAs oder des Speichers für die Konfiguration kommt. Dies kann z.B. durch einen Single Event Burnout geschehen.

5.4.5 Dreifach redundante Hardware

Die einzige Möglichkeit zur Gewährleistung eines unterbrechungsfreien Betriebes von FPGAs in extraterrestrischen Umgebungen ist der Einsatz einer dreifach redundanten Hardware. Die Ausgänge der drei FPGAs werden, wie bei einem TMR System, mit einem Voter auf Störungen überprüft. Der resultierende Aufbau ist aus Abb. 5.8 ersichtlich.

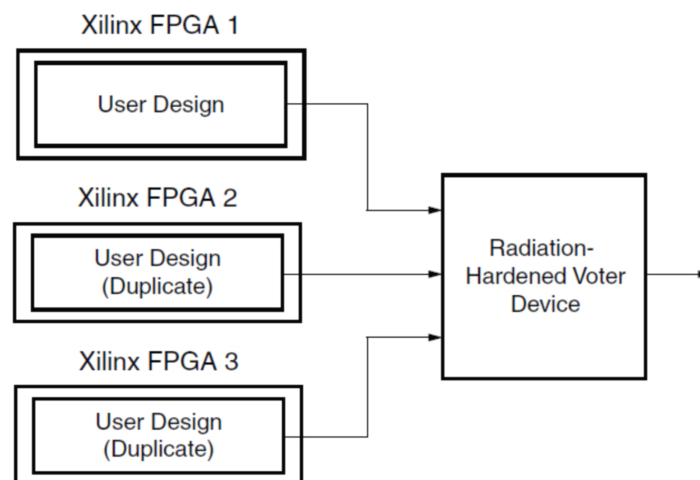

Abbildung 5.12: Triple Device Redundancy [Xil08b].

Bei einem Single Event Functional Interrupt in einem der drei FPGAs kann dieser durch Zurücksetzen (Reset) oder andere Eingriffe von Außen behoben werden. Dadurch würde es in einem

nichtredundanten System zu einem kurzzeitigen Ausfall kommen. Dies ist hier jedoch nicht der Fall, da die zwei anderen FPGAs ohne Unterbrechung weiter arbeiten. Wie bei einem TMR-System kommt es erst dann zum Ausfall des Gesamtsystems, wenn mehr als ein FPGA von einem SEFI betroffen ist. Selbstverständlich muss auch der Voter, wie bei den TMR-Systemen, vor der Strahlung geschützt werden.

Die wesentlichen Nachteile von Systemen mit dreifach redundanter Hardware sind die hohen Kosten und der wesentlich größere Platzbedarf. Zusätzlich muss die dadurch bedingte Gewichtszunahme in Kauf genommen werden.

6 Strahlungstolerante FPGA-Familien

Die *Xilinx* Unternehmensgruppe ist derzeit der weltweit erfolgreichste Hersteller von programmierbaren Logikbausteinen. In den letzten Jahren verzeichnet Xilinx einen leicht ansteigenden Marktanteil, welcher momentan knapp über 50 Prozent liegt. Diese Führungsposition verdankt das Unternehmen unter anderem der großen Produktvielfalt und ihrem großen Know-How. Der größte Konkurrent, mit etwa 35 Prozent Marktanteil, ist *Altera*. Die Marktanteile der letzten Jahre sind zusammenfassend in Abb. 6.1 dargestellt.

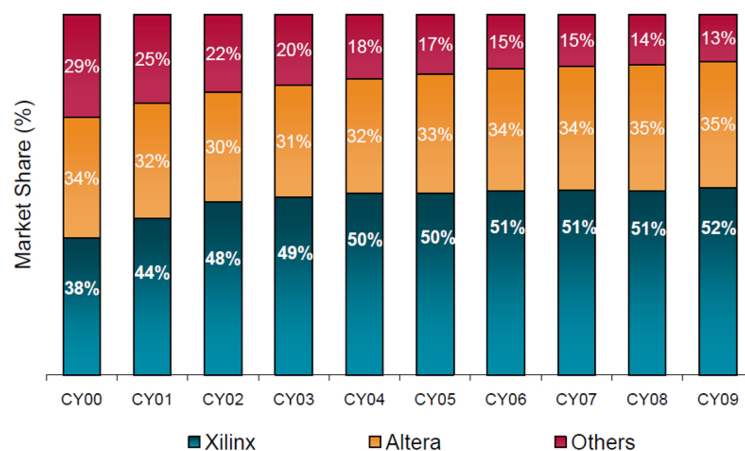

Abbildung 6.1: Marktanteile von Altera und Xilinx bei programmierbaren Logikbausteinen [Xil10b].

Altera beherrscht zusammen mit Xilinx über 85 Prozent des Marktes. Jedoch existieren zahlreiche kleinere Hersteller, welche sich auf spezielle Anwendungen spezialisiert haben. Beispielsweise produzieren *Actel* und *Atmel* eine Reihe von FPGAs die für Luft- und Raumfahrtanwendungen verwendet werden können.

In den nachfolgenden Abschnitten werden diejenigen FPGA-Familien von Xilinx, Actel und Altera genauer erläutert, welche für Raumfahrtsysteme geeignet erscheinen.

6.1 Xilinx space-grade FPGA-Familien

Besonders hervorgetan hat sich Xilinx bei der Entwicklung von strahlungsresistenten (radiation hardened) FPGA-Familien für Luft- und Raumfahrtsysteme. Zudem ist das Unternehmen der weltweite Marktführer im Bereich der programmierbaren Logikbausteine.

Zu den space-grade FPGAs gehören die *Virtex-II Qpro*, *Virtex-4 QV* und die *Virtex-5 QV*-Modelle. Dies sind spezielle radiation-hardened FPGA-Varianten der entsprechenden high-performance Virtex-Familien. Die Virtex-4QV- und Virtex-5QV-FPGAs sind in Abb. 6.2 dargestellt.

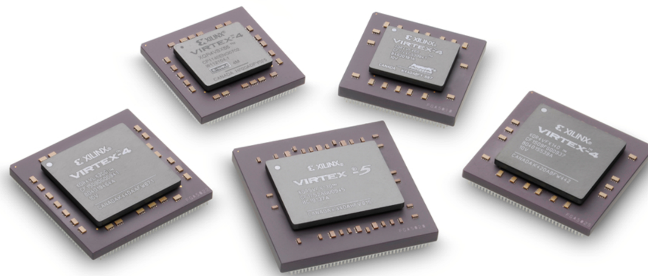

Abbildung 6.2: Virtex-4 und Virtex-5 FPGAs [Xilinx].

Die Merkmale der einzelnen Virtex-Familien unterscheiden sich teilweise stark voneinander. Dementsprechend ist die Wahl des richtigen FPGAs nicht nur ein wesentlicher Kosten- und Erfolgsfaktor, sondern auch aus technischer Sicht eine Notwendigkeit. Eine Übersicht der wichtigsten Merkmale ist aus Tabelle 6.1 ersichtlich.

	Xilinx space-grade FPGAs		
	Virtex-5QV XQR5VFX130	Virtex-4QV XQR4VFX60	Virtex-II XQR XC2V3000
Core Voltage	1.0 V	1.2 V	1.5 V
Logic Cells	130.000	56.880	32.256
Total BlockRAM (kBit)	10.728	4.176	1.728
Max. Single-Ended I/Os	836	576	720
Enhanced DSP Slices	320	-	-
DSP Slices	-	128	-
18 x 18 Multiplier	-	-	96
PowerPC	-	2	-
TID	700 krad	300 krad	200 krad
SEL Immunity	>100	> 100	> 160

Tabelle 6.1: Merkmale einiger Virtex FPGAs für Luft- und Raumfahrtanwendungen [Xil10a]

Die aufgelisteten Merkmale liefern einen kleinen Überblick über die einzelnen Familien. In der Praxis sind aber zahlreiche Varianten der einzelnen FPGAs erhältlich. Diese unterscheiden sich meist in ihrem Funktionsumfang und den vorgesehenen Einsatzbereichen.

Zusätzlich produziert Xilinx eine Reihe von speziellen Speicherbausteinen für Umgebungen mit einer erhöhten Strahlung. Diese können z.B. als externe Speicher für SRAM-basierte FPGAs verwendet werden.

6.1.1 Die Virtex-II QPro-Familie

Die *Virtex-II QPro*-Familie wurde erstmals im Jahr 2004 eingeführt und ist eine Weiterentwicklung der erfolgreichen Virtex-Architektur. Diese high-performance FPGAs zeichneten sich besonders dadurch aus, dass ihre Architektur für kritische Anwendungen in strahlenden Umgebungen entwickelt wurde. Dementsprechend wurde auf eine hohe TID- und SEE-Immunität großen Wert gelegt.

Die wichtigsten Eigenschaften der Virtex-II QPro-Familie sind [Xil06]:

- 1.5 V Versorgungsspannung
- Garantierte TID von 200 krad
- Latch-up Immunität für LET > 160 $MeV cm^2/mg$
- SEU in GEO (SEFI) < 1.5E-6 upsets/device/day
- 0.15 μm 8-Layer Prozess
- SRAM-basierte In-System-Konfiguration
  - Partielle Rekonfiguration und Readback Fähigkeit
  - Optionale DES Verschlüsselung für Konfigurations-Bitstream
- Max. 33.792 Slices
- SelectRAM - 2.5Mb dual-port RAM
- 18 x 18 Bit Multiplizierer
- Max. je 67.584 Register und LUT/Shift-Register
- 12 Digital Clock Manager
- SelectIO-Ultra Technology
  - Max. 824 I/O-Anschlüsse
  - Digitally Controlled Impedance
- IEEE 1149.1 - JTAG/Boundary-Scan Support

Die Virtex-II QPro-Familie besitzt drei FPGAs, welche mit XQR2V1000, XQR2V3000 und XQR2V6000 bezeichnet werden. Die letzten vier Ziffern, multipliziert mit Tausend, ergeben die Anzahl der verfügbaren Logikgatter. Als Konfigurationszellen werden flüchtige SRAM-Zellen verwendet. Entsprechend muss ein externer Speicher für die Konfiguration vorhanden sein.

Grundlegende Architektur

Die Architektur eines Virtex-II QPro-FPGAs besteht aus zahlreichen Komponenten. Der grundlegende Aufbau ist aus Abb. 6.3 ersichtlich.

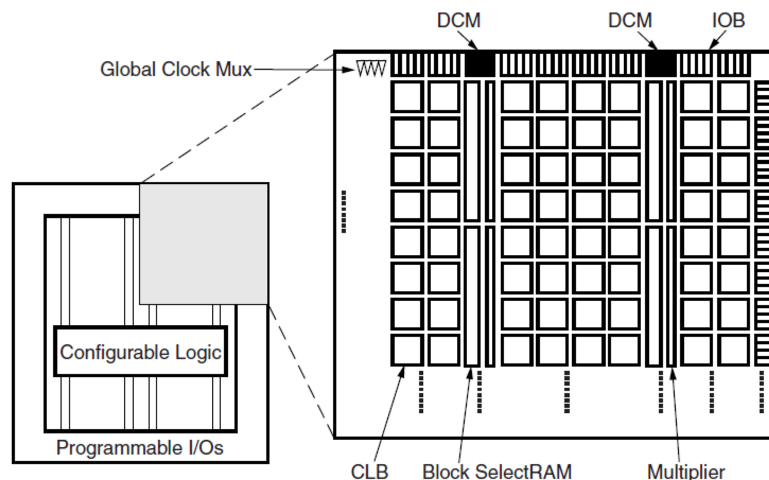

Abbildung 6.3: Architektur der Virtex-2 QPro-FPGAs [Xil06].

Die *Configuration Logic Blocks* enthalten die eigentliche Logik. Diese werden über die konfigurierbare *General Routing Matrix (GRM)* miteinander verbunden. Die programmierbaren *I/O*

Blocks bilden die Schnittstelle zwischen den Anschlüssen (pins) und der internen Logik. Des weiteren enthält der FPGA mehrere *Block SelectRAM*-Speichermodule und einige *Multiplizierer (multiplier)*. Die *Digital Clock Manager*-Blöcke erzeugen und steuern die Taktsignale.

Grundlegende Informationen zu den Komponenten können aus Kapitel 3 entnommen werden.

Configuration Logic Block

Ein *Configuration Logic Block (CLB)* der Virtex-II QPro-Familie besteht aus vier *Slices*, welche über schnelle Datenleitungen miteinander verbunden sind. Die einzelnen CLBs sind über eine *Verbindungsmatrix (switch matrix)* mit der *General Routing Matrix* verbunden. Die GRM definiert die komplexe Verbindungsstruktur zwischen den einzelnen CLBs und den verschiedenen Komponenten im FPGA (siehe Abb. 6.4).

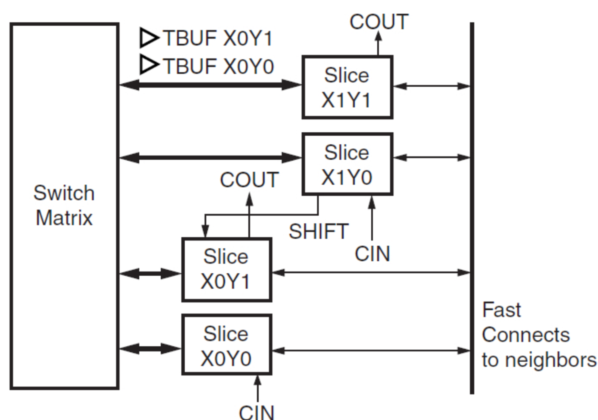

Abbildung 6.4: CLB eines Virtex-2 QPro-FPGAs [Xil06].

Der interne Aufbau eines *Slice* ist aus Abb. 6.5 ersichtlich.

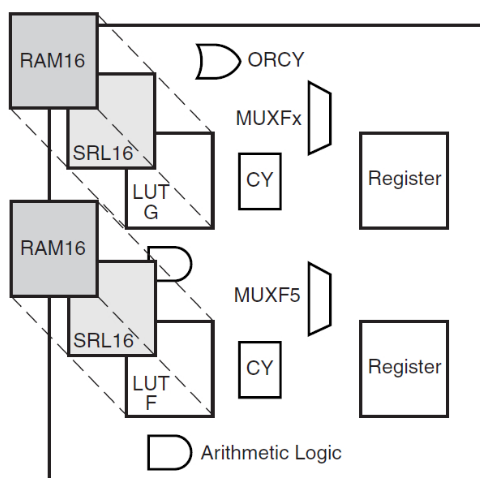

Abbildung 6.5: Slice eines Virtex-2 QPro-FPGAs [Xil06].

Jeder *Slice* besteht aus zwei Funktionsgeneratoren (function generators). Diese können einzeln als *4-input Lookup-Table (LUT)*, *16-Bit Schieberegister (SRL16)* oder alternativ als *16-Bit distributed SelectRAM (RAM16)*-Speicher konfiguriert werden. Zusätzlich stehen zwei *Speicherelemente (register)*, *arithmetische Logikgatter*, *Fast Carry Logic* und *Multiplexer* zur Verfügung.

Input/Output-Block

Ein *Input/Output Block (IOB)* kann als Ein- oder Ausgang konfiguriert werden. Insgesamt stehen 19 I/O-Standards zur Auswahl. Zwei IOBs können zu einen differentiellen Eingang kombiniert werden. Eine Verwendung als einstellbare Stromsenke ist ebenfalls möglich. Die Konfiguration erfolgt mit Hilfe der sechs Register des IOBs (siehe Abb. 6.6).

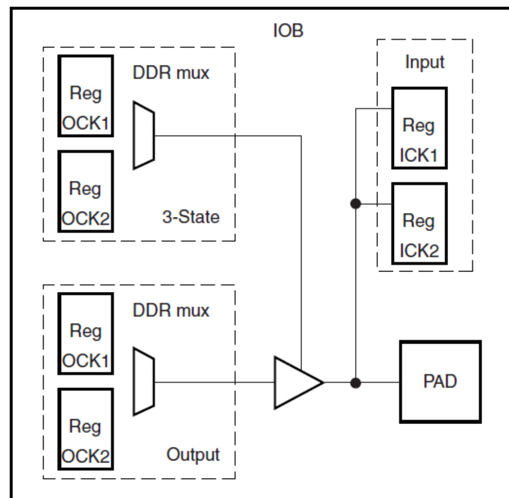

Abbildung 6.6: I/O-Block eines Virtex-2 QPro-FPGAs [Xil06].

Alle IOBs werden in insgesamt acht *I/O-Bänke* unterteilt. Jede Bank enthält mehrere V_{CC0} Anschlüsse, welche mit einer bestimmten Spannung versorgt werden müssen. Diese hängt vom verwendeten Ausgangs-Standard ab.

Digital Controlled Impedance

Die Virtex-II QPro-FPGAs können I/O-Daten mit bis zu 622 Mbit/s übertragen. Bei hohen Frequenzen kann es zu einer Fehlanpassung zwischen den Anschlüssen des FPGAs und den Leitungen kommen, wodurch unerwünschte Reflexionen entstehen. Diese können mit Hilfe der *Digital Controlled Impedance (DCI)*-Module beseitigt werden, indem man für jede I/O-Bank eine bestimmte Impedanz vorgibt.

Digital Clock Manager

Insgesamt stehen in jedem FPGA zwölf *Digital Clock Manager (DCM)*, für die Steuerung der Taktsignale, zur Verfügung. Die wichtigste Aufgabe eines DCM ist die Vermeidung von *Phasenabweichungen (Jitter)* zwischen den Taktsignalen. Diese entstehen z.B. durch die unterschiedlichen Leitungslängen im FPGA. Die DCMs können ebenfalls zur *Frequenzsynthese*, d.h. zur Abwandlung der eigentlichen Taktfrequenz, verwendet werden. Diese neuen Taktsignale können

intern oder extern verwendet werden. Auf Wunsch kann auch eine bestimmte *Phasenverschiebung (phase shift)* zwischen den Taktsignalen erzeugt werden.

Block SelectRAM

Jeder FPGA enthält eine größere Menge an 18-kBit großen *SelectRAM-Blöcken*. Dabei handelt es sich um einen konfigurierbaren synchronen Speicher. Je nach Wunsch kann man die Blöcke als single- oder dual-port RAM betreiben. Eine Wahl der Wortbreite von 16 x 1-Bit bis 512 x 36-Bit ist ebenfalls möglich.

Multiplizierer

Bei den *Multiplizierer (multiplier)*-Modulen handelt es sich um 18 x 18-Bit 2's-complement signed Multiplizierer. Die einzelnen Multiplizierer können separat verwendet oder über eine Verbindungsmatrix mit einem SelectRAM-Block verknüpft werden (siehe Abb. 6.7).

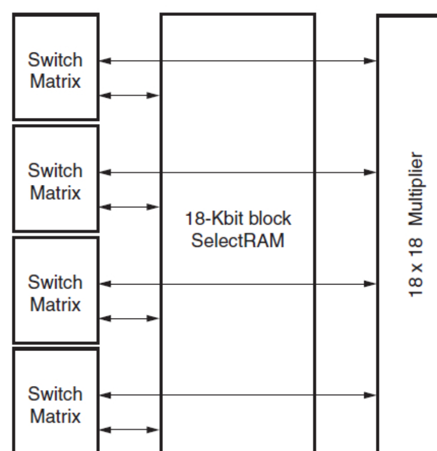

Abbildung 6.7: SelectRAM und Multiplizierer Blöcke eines Virtex-2 FPGAs [Xil06].

Der Vorteil liegt darin, dass sie im Vergleich zu CLBs schneller und energiesparender arbeiten.

Konfiguration

Die Konfiguration der Virtex-II QPro-FPGAs wird in flüchtigen SRAM-Konfigurationszellen gespeichert. Die Konfiguration geht beim Ausschalten verloren und muss nach dem Einschalten neu in den FPGA geschrieben werden.

Von den FPGAs werden die *Master/Slave-Serial*, die *Master/Slave-SelectMAP* und die *IEEE 1149.1 JTAG/Boundary-Scan*-Konfigurationsmethoden unterstützt. Die Wahl der Methode erfolgt über die *Mode-Pins* und hängt vom Aufbau des Systems ab.
Bei der *Slave-Serial*-Methode empfängt der FPGA die Konfigurationsdaten aus dem externen Speicher als seriellen Bit-Stream. Werden hingegen die Daten vom FPGA selbst aus dem externen Speicher angefordert, dann handelt es sich um die *Master-Serial*-Methode. Eine wesentlich schnellere Konfiguration durch einen Byte-Stream erlaubt die *SelectMAP*-Schnittstelle. Auch hier kann zwischen Slave und Master unterschieden werden.
Die Programmierung mit der IEEE 1149.1 JTAG/Boundary-Scan-Schnittstelle erfolgt dementsprechend über den eingebauten TAP Controller. Zusätzlich wird der IEEE 1532 In-System

Configurable (ISC)-Standard unterstützt. Neben der Programmierung werden dadurch auch die Rekonfiguration und der Test des FPGAs ermöglicht.
Die Konfigurationsdaten können bei der Übertragung verschlüsselt werden. Dazu wurde in den FPGAs ein *Data Encryption Standard (DES)*-Algorithmus implementiert.

Für den eigentlichen Konfigurationsvorgang werden spezielle Anschlüsse am FPGA verwendet. Auf Wunsch (persist option) können diese auch nach der Konfiguration für die partielle Rekonfiguration (Scrubbing) und das Auslesen der Daten (Readback) verwendet werden.
Dies ist allerdings nur im SelectMAP- oder Boundary-Scan-Modus möglich. Bei einem Readback können zusätzlich auch die Register, der SelectRAM und der BlockRAM ausgelesen werden. Bei der partiellen Rekonfiguration wird nur die betroffene Logik beeinflusst, d.h. der FPGA muss nicht neu gestartet oder sein Betrieb unterbrochen werden.

6.1.2 Die Virtex-4 QV-Familie

Bei der *Virtex-4 QV-Familie* handelt es sich um eine strahlungstolerante space-grade Variante der Virtex-4 FPGAs. Die Architektur weist im Vergleich zu ihren Vorgängern einige wesentliche Erneuerungen auf.

Die wichtigsten Eigenschaften der Virtex-4 QV-Familie sind [Xil10f]:

- 1.2 V Versorgungsspannung
- Garantierte TID von 300 krad
- Latch-up Immunität für LET > 100 $MeVcm^2/mg$
- SEFI < 1.5E-6 upsets/device/day
- SEU Mitigation-Unterstützung mit TMRTool Software
- 90 nm Kupfer CMOS-Prozess
- SRAM-basierte In-System Konfiguration
  - Optionale AES Verschlüsselung für Konfigurations-Bitstream
  - SelectMAP und Serial Mode
- Max. 89.088 Slices
- Xesium Clock Technologie
  - Max. 20 Digital Clock Manager
  - Zusätzliche phase-matched Frequenzteiler
- Max. 512 XtremeDSP Slice
- SelectIO Technologie
  - Max. 960 I/O-Anschlüsse
  - Digitally Controlled Impedance
- Smart RAM Speicherhierarchie
  - Max. 6 Mb RAM mit 18-kBit Blöcken
  - High-speed Speicher-Schnittstellen
- IEEE 1149.1 - JTAG/Boundary-Scan Support
- IBM PowerPC RISC-Prozessorkerne (nur FX)
- Multiple Ethernet Media Access Controller (nur FX)

Grundlegende Architektur

In den Virtex-4 QV-FPGAs wird eine spezielle *Advanced Silicon Modular Block (ASMBL)-Architektur* verwendet. Ziel der ASMBL ist die Entwicklung kostengünstiger off-the-shelf FPGAs mit einem breiten Funktionsumfang. Die wesentlichen Erneuerungen gegenüber der Virtex-II

Familie sind eine *high-performance Logic*, die *Xesium Clock Technology*, überarbeitete *CLBs* sowie zusätzliche *XtremeDSP Slices* zur Signalverarbeitung.

Die verschiedenen FPGAs sind als LX-, SX- und FX-Varianten verfügbar. Diese unterscheiden sich in der Größe und dem Funktionsumfang. Die FX-Bausteine verfügen über die größte Funktionalität und beinhalten zusätzlich zwei eingebettete *PowerPC-Prozessoren* und einen *tri-mode Ethernet Media Access Control (MAC)*-Block.

Xesium Clock Technologie

Die Virtex-4 QV-FPGAs enthalten maximal 20 *Digital Clock Manager*. Diese steuern alle Taktsignale und korrigieren unerwünschte Phasenverschiebungen (Jitter). Weiters können Taktsignale mit einer definierten Frequenz oder Phasenverschiebung erzeugt werden. Ein wesentlicher Unterschied zum Virtex-II besteht darin, dass jeder DCM über eine spezielle Schnittstelle im laufenden Betrieb dynamisch konfiguriert werden kann.

Generell kann zwischen sogenannten *globalen* und *lokalen* Taktsignalen (clocks) unterschieden werden. Insgesamt sind 32 globale Taktleitungen (clock lines) vorhanden, welche von jeweils einem *Global Clock Buffer (GCB)* betrieben werden. Der GCB wird meist vom DCM betrieben und benutzt diesen als Taktquelle. Die globalen Taktsignale werden für die Taktung aller auf dem FPGA vorhandenen sequentiellen Logikkomponenten verwendet. Beim Ausfall eines DCM wechselt der GCB automatisch auf eine funktionierende Taktquelle.

Der FPGA wird für die Verteilung der Taktsignale in mehrere Bereiche (regions) unterteilt. Die Anzahl der Bereiche hängt von der Größe des FPGAs ab, wobei ein Bereich aus 16 CLBs besteht. Als Beispiel sind die acht Regionen eines XC4VLX15 in Abb. 6.8 dargestellt.

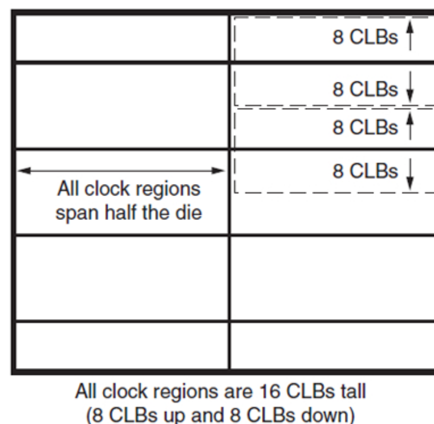

Abbildung 6.8: Die acht Bereiche für die Taktsignale im XC4VLX15 [Xil08c].

Ein lokales Taktsignal ist unabhängig von den globalen Taktsignalen und wird innerhalb der verschiedenen Regionen verwendet. Der *Regional Clock Buffer (RCB)* ändert dazu das globale Taktsignal auf die gewünschte lokale Taktfrequenz ab. Der Wirkungsbereich des lokalen Taktes ist allerdings auf maximal drei Regionen beschränkt.

Die *Phase-Matched Clock Dividers (PMCDs)* besitzen Frequenzteiler zur Erzeugung von bis zu vier Taktsignalen. Zusätzlich können phasenverschobene Signale erzeugt werden.

Die DCMs sind zusammen mit den PMCDs symmetrisch in der mittleren Spalte des FPGAs angeordnet (siehe Abb. 6.9).

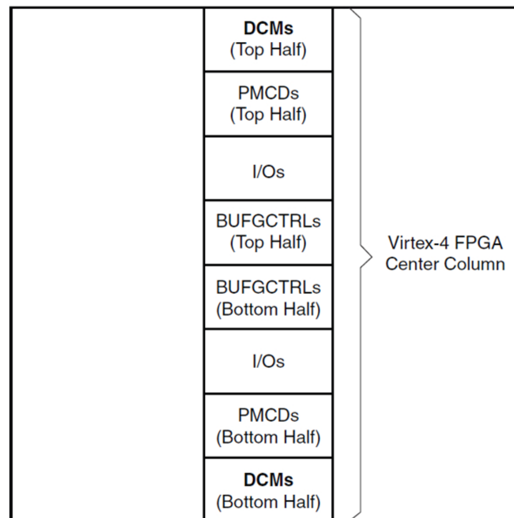

Abbildung 6.9: Anordnung von DCM und PMCD [Xil08c].

Configuration Logic Block

Die *CLBs* bestehen aus insgesamt vier Slices. Allerdings werden jeweils zwei davon spaltenweise zu einem Paar zusammengefasst. Der Aufbau ist in Abb. 6.10 dargestellt.

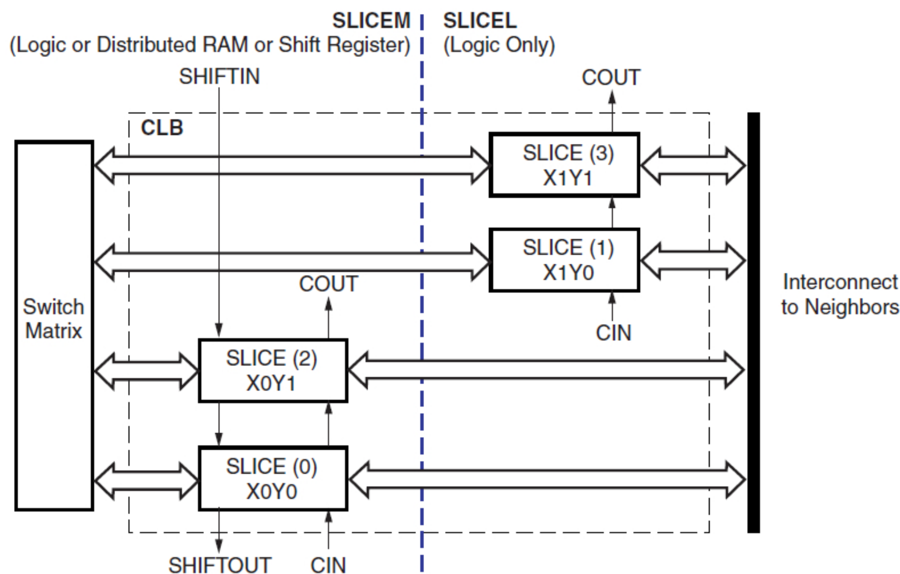

Abbildung 6.10: CLB eines Virtex-4 QV-FPGAs [Xil08c].

Beide Slice-Paare werden als *SLICEL* und *SLICEM* bezeichnet und besitzen jeweils eine unabhängige Leitung für die Übergabe der Übertrags-Bits (carry chain). Zusätzlich können die

beiden Slices in SLICEM als Schieberegister verkettet werden (shift chain). Wie beim Virtex-II sind auch hier die Slices über schnelle Datenleitungen miteinander verbunden. Der Datenaustausch zwischen den verschiedenen CLBs erfolgt über die *Verbindungsmatrix (switch matrix)*, welche mit der *General Routing Matrix* verbunden ist.

Ein einzelner Slice besteht aus zwei *Funktionsgeneratoren (function generators)*, zwei *Speicherelementen (registers)*, *arithmetischen Logikgattern*, einer *Fast Carry Logic* und *Multiplexern*. Jeder Funktionsgenerator kann als LUT mit vier Eingängen konfiguriert werden. Die zwei LUT in SLICEM können zudem zu einem 16-Bit Schieberegister oder 16-Bit RAM kombiniert werden. Folglich besitzt ein CLB insgesamt acht Flip-Flops, acht LUTs bzw. alternativ einen 64-Bit RAM oder ein 64-Bit Schieberegister.

XtremeDSP Slices

Die *XtremeDSP Slices (DSP48)* wurden als neue Komponenten in die ASMBL-Architektur integriert. Sie erweitern den Funktionsumfang der Virtex-4 QV-Familie und ermöglichen die Realisierung neuer *Digital Signal Processing (DSP)*-Anwendungen.

Jeder DSP48 besteht aus einem 18 x 18-Bit Multiplizierer, 48-Bit Addierer/Subtrahierer, Schieberegister, Multiplizier-Akkumulator, Komparator, mehreren Multiplexern und einem 48-Bit Zähler. Durch Vorgabe eines OPMODE können die einzelnen Komponenten mit Hilfe der Multiplexer zu einer gewünschten Funktionalität verknüpft werden. Zur Verbesserung der Leistung sind zusätzliche Pipeline-Register vorhanden. Der vereinfachte Aufbau eines XtremeDSP Slice ist aus Abb. 6.11 ersichtlich.

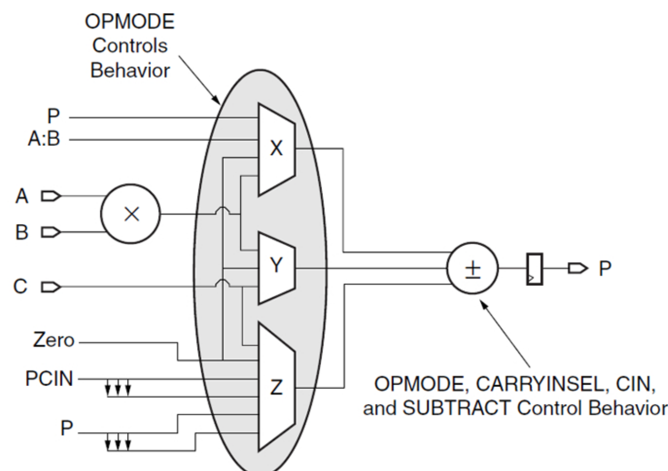

Abbildung 6.11: Vereinfachter Aufbau eines XtremeDSP Slice [Xil08d].

Zwei XtremeDSP Slices können im FPGA zu einem sogenannten *XtremeDSP Tile* zusammengefasst werden. Durch die Verknüpfung können weitere komplexe Komponenten, z.B. größere Multiplizierer oder spezielle Finite Impulse Response (FIR)-Filter, erzeugt werden.

Weitere Informationen zu den XtremeDSP Slices können aus [Xil08d] entnommen werden.

Eingebettete PowerPC-Prozessorkerne

Bei der Virtex-4 QV-Familie sind die FX-Varianten mit zwei *eingebetteten PowerPC 405-Prozessorkernen (PPC405)* ausgestattet. Bei den PPC405 handelt es sich um einen *Reduced Instruction Set Computer (RISC)*-Prozessor mit einer 32-Bit Architektur. Der PPC405 besteht unter anderem aus dem *Prozessorkern (core)*, der *On-Chip Memory Logic (OCM)* und der *Auxiliary Processor Unit (APU)*.

Die OCM fungiert als Interface zwischen dem Prozessorkern und den Daten- bzw. Befehlsspeichern im FPGA.
Mit Hilfe der APU kann der bestehende Befehlssatz durch benutzerdefinierte Befehle erweitert werden. Diese können von einem *FPGA Fabric Co-processor Module (FCM)* ausgeführt werden. Die Kommunikation zwischen dem PPC405-Prozessorkern, dem APU und dem FCM zur Abarbeitung der Befehle ist in Abb. 6.12 dargestellt.

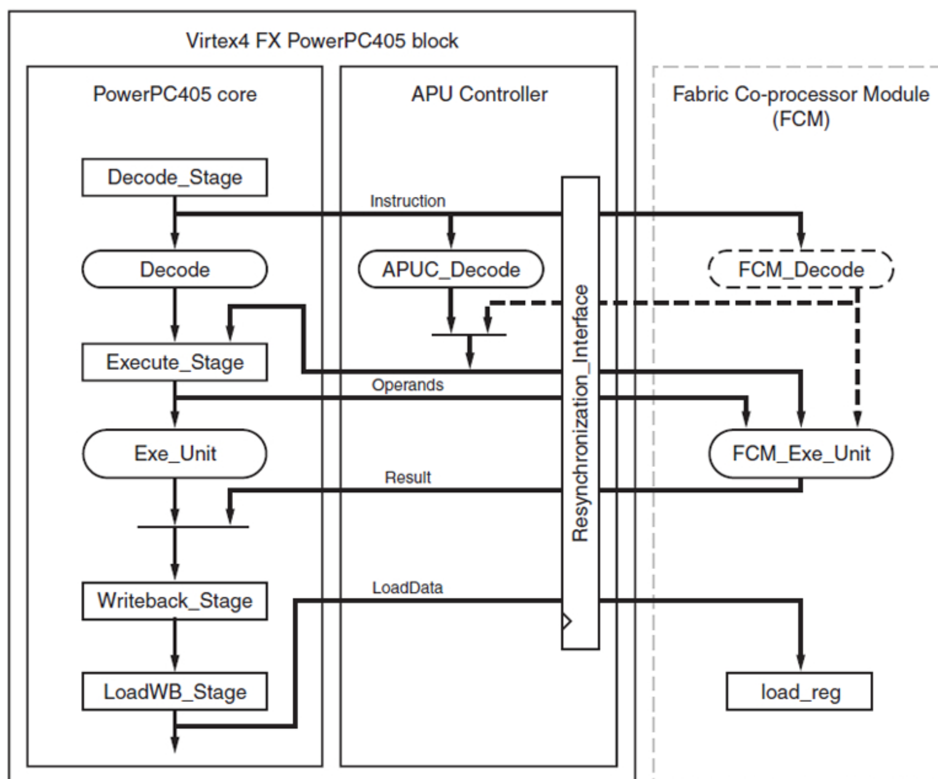

Abbildung 6.12: Die Auxiliary Processor Unit des PPC405 [Xil10d].

Der Datenaustausch mit dem PPC405-Prozessorkern erfolgt über verschiedene Register. Diese können in *Benutzer-Register (user registers)* und *System-Register (privileged register)* unterteilt werden.
Daten können mit Hilfe einiger Lade-Befehle vom Speicher in die Register geschrieben oder von dort gelesen werden. Viele Befehle verwenden Daten aus den Registern oder speichern dort die Resultate. Einige Register ermöglichen den Zugriff auf die internen Komponenten des PPC405 oder liefern wichtige Informationen über den aktuellen Zustand. Der Aufbau des Prozessorkerns ist aus Abb. 6.13 ersichtlich.

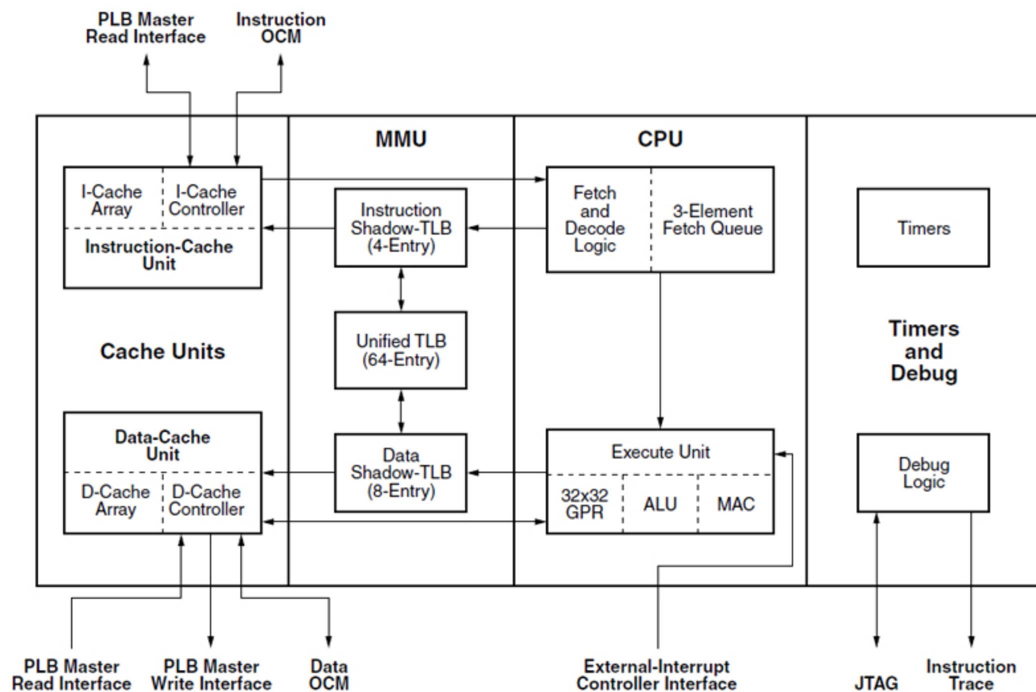

Abbildung 6.13: Struktur eines PowerPC 405 Prozessorkernes [Xil10d].

Bei der Software für den PPC405 wird zwischen den *privileged-mode* und *user-mode* Betriebsarten (operation modes) unterschieden. Auf die Benutzer-Register kann mit jeder Software zugegriffen werden. Im Gegensatz dazu können System-Register nur mit Hilfe der *privileged Software* ausgelesen oder verändert werden. Die System-Register sind somit vor einem Zugriff durch eine normale Benutzersoftware (user-mode software) geschützt.
Die Befehle und Daten werden beim Ausführen der Software in den *Cache Units* zwischengespeichert. Dazu steht jeweils ein 16-kByte großer Befehls-Cache (instruction cache) und Daten-Cache (data cache) zur Verfügung. Zur Steuerung und Überwachung wird eine Cache-Logik (Cache Controller) eingesetzt. Dieser befüllt die Caches und ersetzt gegebenenfalls einzelne, nicht mehr benötigte, Einträge.

Der *Central Processing Unit (CPU)* holt die einzelnen Befehle aus dem Cache und arbeitet diese mit Hilfe eines *Fetch-Execute-Algorithmus (fetch and decode logic)* ab. Dabei werden Speicheradressen und Befehlscodes (Opcodes) ermittelt. Sofern virtuelle Adressen verwendet werden, müssen diese mit Hilfe der *Memory Management Unit (MMU)* in Adressen des physikalischen Speichers umgewandelt werden. Anschließend werden die einzelnen Befehle mit der *Execute Unit* der CPU ausgeführt.
Der *Translation Look-aside Buffer (TLB)* beinhaltet bereits übersetzte Zuordnungen von virtuellen Speicherseiten mit dem physikalischen Speicher. Häufig werden in Folge der Abarbeitung Daten aus den Registern verarbeitet und Resultate in diese gespeichert. Der CPU ist deshalb über den *Instruction Shadow-TLB* und den *Data Shadow-TLB* mit der Cache Unit verbunden. Dadurch wird eine schnellere Abarbeitung der Befehle ermöglicht.

Weitere Informationen zu den PowerPC-Prozessoren sind bei [Xil10d] verfügbar.

Tri-Mode Ethernet Media Access Controller

Der *Tri-Mode Ethernet Media Access Controller (Ethernet MAC)*-Block ermöglicht die Verwendung des IEEE 802.3 Ethernet-Standards. Es werden Datenraten von 10, 100 und 1000 Mbit/s unterstützt und automatisch detektiert. Das Blockdiagramm ist aus Abb. 6.14 ersichtlich.

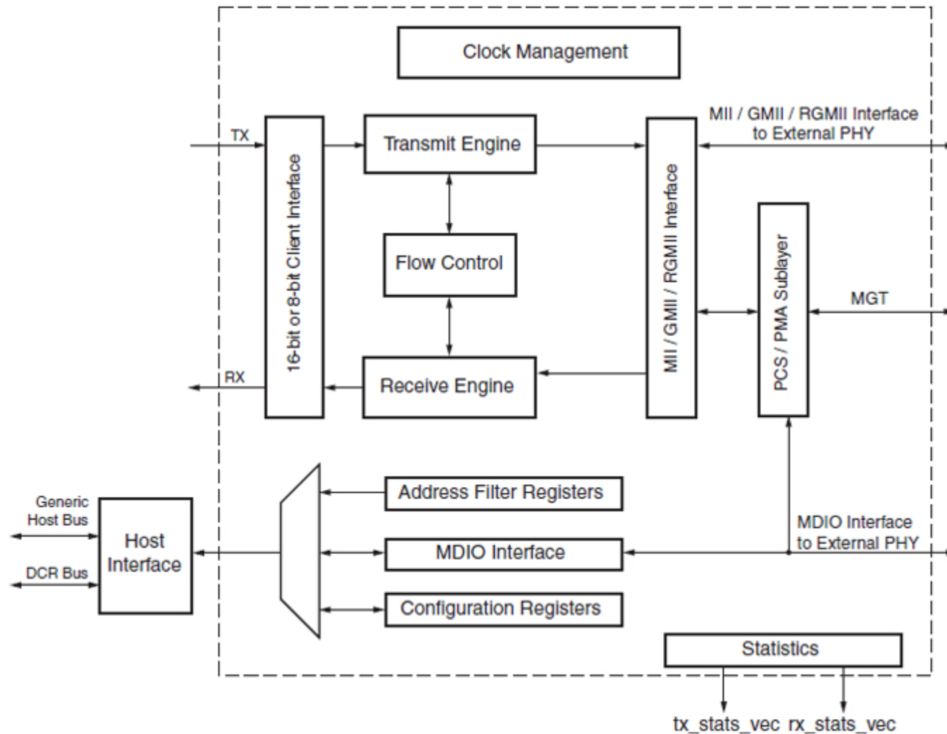

Abbildung 6.14: Struktur eines Virtex-4 embedded Ethernet MAC Blockes [Xil10g].

Das Blockdiagramm besteht aus zwei MAC-Blöcken, welche sich ein gemeinsames *Host-Interface (host interface)* teilen. Ein *physikalisches Interface* für die MII, GMII und GRMII Protokolle (MII/GMII/GRMII interface) steht ebenfalls zur Verfügung. Das *Adressfilter (address filter)* entfernt anhand der Frame-Adresse diejenigen eingehenden Datenpakete (frames), die nicht für den MAC-Block bestimmt sind. Dies ist besonders dann relevant, wenn Systeme mit mehreren MACs verwendet werden.

Die *RX/TX-Schnittstelle* besteht aus einem Sende- und Empfangsmodul (RX/TX engines), welche von einem Kontrollflusssystem (flow control) gesteuert werden. Das Managementmodul für die Ein- und Ausgabe von Daten (management data I/O) ermöglicht den Zugriff auf die Status-Register der externen Schnittstelle und das PCS/PMA-Sublayer-Modul.
Zusätzlich können verschiedene Statistiken (statistics) im Ethernet MAC abgerufen werden.

Weitere Informationen zu den Ethernet MAC sind bei [Xil10g] verfügbar.

Integrierter Block-RAM

Alle FPGAs der Virtex-4 QV-Familie besitzen einen integrierten Block-RAM mit 18 kBit großen Blöcken. Diese können als 16k x 1 bis 512 x 36-Bit Speicher konfiguriert werden. Jeder Speicher-

block verfügt über zwei unabhängige Schnittstellen (dual-port block RAM), welche in Abb. 6.15 dargestellt sind.

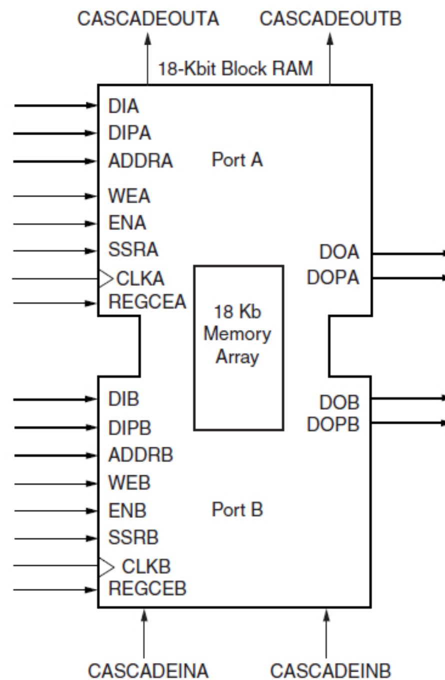

Abbildung 6.15: Die zwei Ports eines integrierten Block-RAM [Xil08c].

Durch Zusammenhängen mehrerer Block-RAMs können größere Speicher erzeugt werden. Optional kann jeder Speicherblock auch als First-In-First-Out (FIFO)-Speicher konfiguriert werden.

SelectIO Ressourcen

Die I/O-Schaltungen der Virtex-4 QV-FPGAs bestehen aus einem *Input/Output Block (IOB)* und jeweils einem *Eingangs- und Ausgangslogik (ILOGIC, OLOGIC)*-Block (siehe Abb. 6.16).

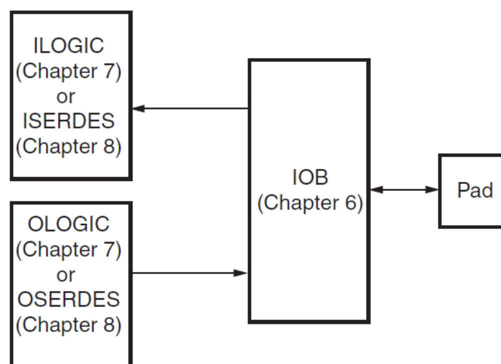

Abbildung 6.16: Blöcke der Virtex-4 I/O-Beschaltung [Xil08c].

Ein IOB verfügt über einen I/O-SelectIO-Treiber der mit dem einem Anschluss (pad) verbunden

ist. Damit können für die einzelnen Anschlüsse bestimmte I/O-Standards vorgegeben werden. Die ILOGIC- und OLOGIC-Blöcke beinhalten alle Logik-Ressourcen der Virtex-II QPro-Familie. Dies sind unter anderem Logikschaltungen für den *combinatorial I/O*, *registered I/O*, *3-state I/O* und *Double Data-Rate (DDR) I/O*.

Des weiteren sind mehrere DCI-Blöcke in den Virtex-4 QV-FPGAs enthalten. Diese ermöglichen die Einstellung der Ein- und Ausgangsimpedanzen. Dadurch können unerwünschte Reflexionen auf Grund hochfrequenter Datensignale behoben werden.

Konfiguration

Wie bei der Virtex-II QPro-Familie sind auch die Virtex-4 QV-FPGAs mit flüchtigen SRAM-Konfigurationszellen ausgestattet. Beim Einschalten müssen deshalb die Konfigurationsdaten in den FPGA geschrieben werden. Dementsprechend benötigt man zusätzlich einen externen Speicher für die Konfiguration.

Bei der Konfiguration kann zwischen der *Master/Slave-Serial*, der *Master/Slave-SelectMAP 8/32-Bit* und der *JTAG/Boundary-Scan*-Methode gewählt werden. Eine Erneuerung gegenüber dem Virtex-II ist die 32-Bit Variante des SelectMAP-Modus. Dadurch können die FPGAs noch schneller konfiguriert werden.

Bei den Master-Methoden übernimmt der FPGA die Konfiguration, indem er für den externen Speicher ein Taktsignal erzeugt (configuration clock). Der Speicherbaustein erkennt dies und liefert dem FPGA anschließend die benötigten Daten. Wenn hingegen der externe Speicher das Taktsignal erzeugt und den FPGA auffordert die Konfigurationsdaten zu übernehmen, dann handelt es sich um eine Slave-Methode.

Zur Fertigstellung der Konfiguration müssen insgesamt acht Schritte durchlaufen werden. Diese sind in Abb. 6.17 dargestellt.

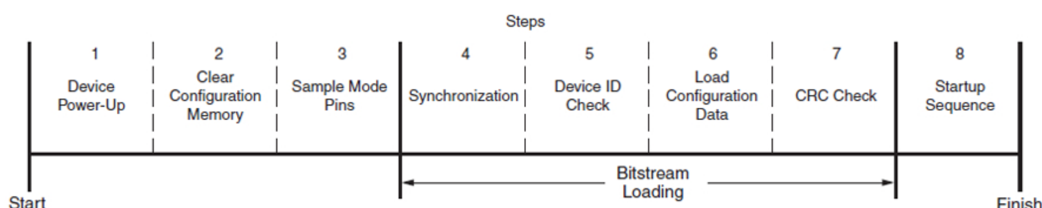

Abbildung 6.17: Schritte bei der Konfiguration eines Virtex-4 FPGAs [Xil09b].

Im ersten Schritt wird die Versorgungsspannung V_{CC} an den FPGA angelegt. Im zweiten Schritt wird die alte Konfiguration gelöscht. Dies geschieht bei jedem Neustart eines FPGAs. Anschließend werden die MODE-Pins überprüft und im Master-Modus das Taktsignal für den externen Speicher erzeugt.
Bei der Synchronisierung wird ein spezielles Datenwort gesendet, welches den Beginn des Daten-Streams signalisiert. Bevor jedoch der tatsächliche Konfigurationsvorgang beginnt, wird die ID-Nummer des FPGAs überprüft. Dadurch wird gewährleistet, dass der richtige FPGA in einem System programmiert wird. Im sechsten Schritt werden die Konfigurationsdaten in den FPGA geladen. Danach wird zur Kontrolle die CRC-Summe berechnet. Als letzter Schritt wird der FPGA gestartet, wodurch er seine vorgesehene Funktion übernimmt.

Advanced Encryption Standard Verschlüsselung

Zum Schutz der Konfigurationsdaten kann der Bitstream mit einem *Advanced Encryption Standard (AES)-Algorithmus* verschlüsselt werden. Der AES-Algorithmus verschlüsselt 128-Bit Datenpakete mit einem 256-Bit Schlüssel.

Rekonfiguration

Die Rekonfiguration des FPGAs kann über den *Internal Configuration Access Port (ICAP)*, die *JTAG/Boundary-Scan-* oder die *SelectMAP-Schnittstelle* erfolgen. Zudem können dabei Techniken wie Readback oder Scrubbing eingesetzt werden.
Der Inhalt der Konfigurationszellen definiert, neben der vom Entwickler gewünschten Logik, auch die Funktionsweise der funktionalen Logikblöcke (functional block). In diese Kategorie fallen z.B. die Digital Clock Managers. Jeder funktionale Block besitzt jedoch einen eigenen addressierbaren Speicher für die Konfiguration.
Als *Dynamic Reconfiguration of Functional Blocks* bezeichnet man die dynamische Rekonfiguration der funktionalen Logikblöcke im laufenden Betrieb. Dies erfolgt über den speziellen *Dynamic Reconfiguration Port (DRP)*. Bei der Virtex-4 QV-Familie können dadurch funktionale Blöcke zur Laufzeit, ohne Beeinflussung der eigentlichen Logikkomponenten, rekonfiguriert werden.

6.1.3 Die Virtex-5 QV-Familie

Bei der *Virtex-5 QV-Familie* wird eine überarbeitete *Advanced Silicon Modular Block (ASMBL)-Architektur* eingesetzt. Die FPGAs werden mit einem 65 nm Kupfer-CMOS-Prozess gefertigt. Derzeit ist nur der XQR5VFX130-FPGA als space-grade radiation-hardened Baustein verfügbar.

Die wichtigsten Eigenschaften der Virtex-5 QV-Familie sind [Xil10e]:

- 1.0 V Versorgungsspannung
- Garantierte TID von 700 krad
- Latch-up Immunität für LET > 100 $MeVcm^2/mg$
- SEU gehärtete Konfigurationszellen mit typ. 3.8E-10 upsets/device/day
- SEU gehärtete IOBs und DCIs, SEU und SET gehärtete CLBs
- SEFI < 2.76E-7 upsets/device/day
- 65 nm Kupfer CMOS-Prozess
- SRAM-basierte In-System-Konfiguration
  - Serial/Byte Peripheral Interface Flash Konfiguration
  - Optionale AES Verschlüsselung für Konfigurations-Bitstream
  - SelectMAP, Serial und JTAG/Boundary-Scan Mode
- Max. 20.480 CLB Slices
- RocketIO GTX Transceiver
- Advanced XtremeDSP Slices
- SelectIO Technologie
  - Max. 836 I/O-Anschlüsse
  - Digitally Controlled Impedance
- Integrierter Block-Speicher
  - Max. 10.5 Mb RAM mit 36-kBit Blöcken
  - High-speed Speicher-Schnittstellen
- IEEE 1149.1 - JTAG/Boundary-Scan Support
- Tri-mode Ethernet Media Access Controller

Grundlegende Architektur

Durch Einsatz der *Radiation Hardened By Design (RHBD)*-Technologie wird eine höhere Strahlungsimmunität erreicht. Dadurch werden die geschützten Komponenten um den Faktor 1000 unempfindlicher gegenüber SEU.

Die Virtex-5 FPGAs verfügen über SEU- und SET-gehärtete CLB-Flip-Flops. Die SEU-Immunität wird dadurch erreicht, dass die Flip-Flops aus RHBD dual-node Latches aufgebaut sind. Zur Vermeidung der SET werden spezielle Filter verwendet, welche die unerwünschten Spannungsspitzen auf den Daten-, Enable-, Set/Reset- und Taktleitungen verhindern.
Ebenfalls gegen SEU geschützt werden die Flip-Flops der IOBs und die DCI-Controller. Bei den IOBs kommen RHBD dual-node Latches zum Einsatz. Der DCI-Controller wurde hingegen als TMR-System implementiert.

Als wesentliche Erneuerungen sind die größeren *25 x 18-Bit Multiplizierer*, die *36-kBit Block-RAMs* und FIFOs, die *Phase-Looked-Loop Clock-Generatoren*, überarbeitete *Enhanced Clock Management*-Blöcke und neue *Enhanced XtremeDSP Slices* zu nennen.

Configuration Logic Block

Die CLBs der Virtex-5 QV-Familie bestehen aus zwei Slices. Diese sind nicht, wie bei den Virtex-4 QV-FPGAs, über schnelle Datenleitungen direkt miteinander verbunden, sondern ausschließlich über die Verbindungsmatrix (switch matrix). Eine Verbindung der verschiedenen CLBs erfolgt über die GRM. Der Aufbau der CLBs ist aus Abb. 6.18 ersichtlich.

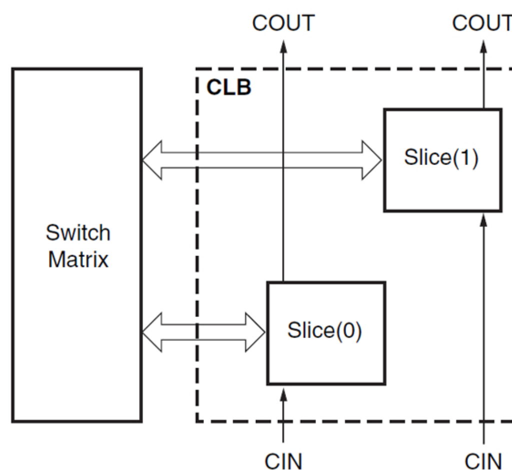

Abbildung 6.18: CLB eines Virtex-5 QV-FPGAs [Xil10i].

Jeder *Slice* besteht aus vier *6-input LUT*, vier *Speicherelementen (register)*, *Multiplexern*, einer *Vorzeichenlogik (carry logic)* sowie einer *arithmetischen Logik*. Bei einigen Slices (SLICEM) können die LUTs optional als Schieberegister oder Speicher (distributed RAM) konfiguriert werden.

Alle in einer Spalte angeordneten Slices besitzen eine eigene Leitung zur Weitergabe der Übertrag-Bits (carry chain). Bei der Verwendung mehrerer CLBs sind diese entsprechend miteinander verbunden (siehe Abb. 6.19).

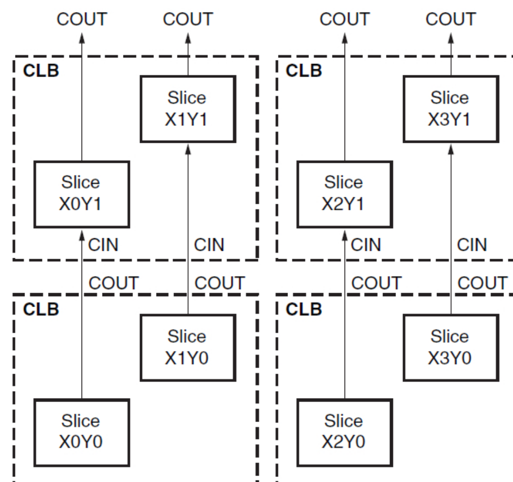

Abbildung 6.19: Verbindung der CLB Carry-Leitungen [Xil10i].

Enhanced XtremeDSP Slices

Die *Enhanced XtremeDSP Slices (DSP48E)* wurden, im Vergleich zu den DSP48 der Virtex-4 QV-Familie, um einige Komponenten erweitert. Der detaillierte Aufbau eines DSP48E ist aus Abb. 6.20 ersichtlich.

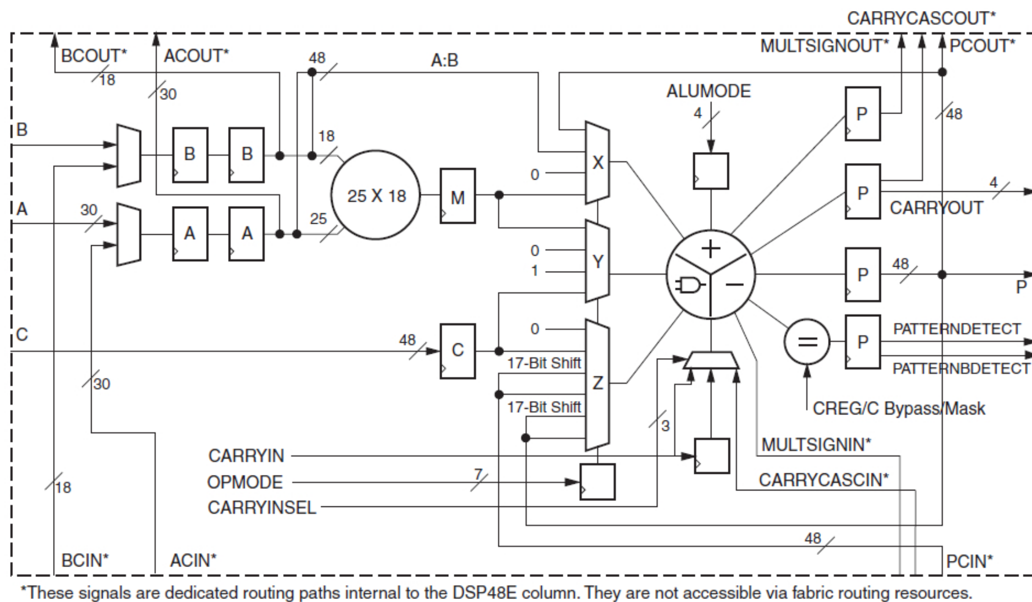

Abbildung 6.20: Detaillierter Aufbau eines DSP48E Slice [Xil10j].

Ein DSP48E Slice besteht aus einem 25 x 18-Bit Multiplizierer, Addierer/Subtrahierer, Schieberegister, Multiplizier-Akkumulator, Zähler, Komparator, Pattern-Detektor und mehreren Multiplexern. Mit den OPMODE-Steuerleitungen und den Multiplexern können die einzelnen Komponenten zu einer gewünschten Gesamtfunktion kombiniert werden. Die Subtrahierer/Addierer-Schaltung wurden um eine Komponente zur Berechnung von Bitoperationen erweitert. Der so-

genannte Patterndetektor (pattern detector) kann zum Runden, der Detektion eines Übertrags (overflow/underfolw) und zum automatischen Rücksetzen der Zähler und Akkumulatoren verwendet werden.
Durch die Verknüpfung mehrerer DSP48E können weitere komplexe Komponenten, z.B. größere Multiplizierer oder spezielle FIR-Filter, erzeugt werden. Optional können Pipeline-Mechanismen zur Leistungssteigerung eingesetzt werden.
Weitere Informationen zu den XtremeDSP Slices können aus [Xil10j] entnommen werden.

Clock Management Technologie

In den Virtex-5 QV-FPGAs werden eigene *Clock Management Tiles (CMTs)* für die Erzeugung und Steuerung der Taktsignale (clocks) eingesetzt. Der XQR5VFX130 besitzt insgesamt sechs CMTs, welche in der Mitte des FPGAs in einer Spalte angeordnet sind.

Ein CMT besteht aus zwei *Digital Clock Managers* und einem *Phase-Locked Loop (PLL)*. Der Aufbau eines CMTs ist aus Abb. 6.21 ersichtlich.

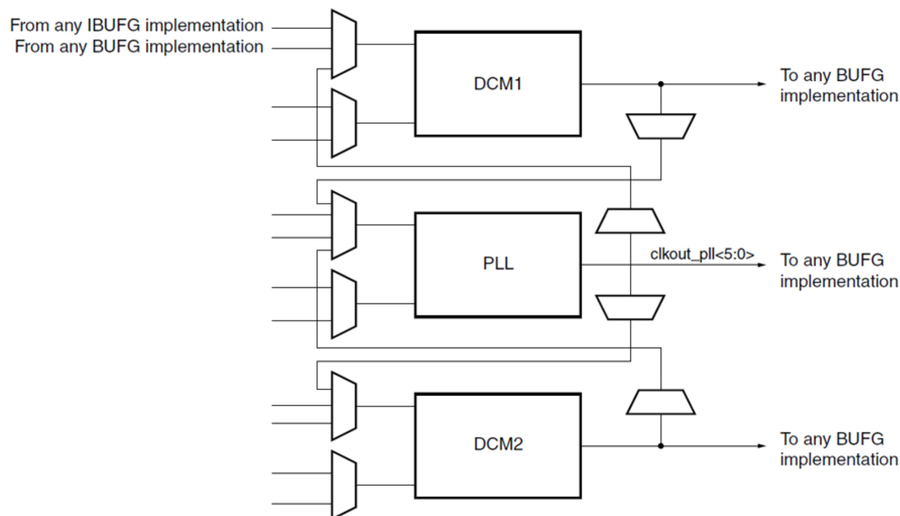

Abbildung 6.21: Aufbau eines Virtex-5 CMT [Xil10i].

Die DCMs und die PLL können als eigenständiger Taktgenerator eingesetzt werden. Optional können zwei Taktsignale des PLLs als Referenz für die zwei DCMs verwendet werden. Zudem kann das Ausgangssignal eines der beiden DCMs als Referenz für den PLL eingesetzt werden. Dazu sind die DCMs und die PLL miteinander über Multiplexer verbunden.

Bei der Verteilung der Taktsignale können im FPGA, z.B. auf Grund unterschiedlicher Leitungslängen, zeitlich versetzte Taktflanken entstehen. Zur Kompensation dieses Effektes besitzt jeder DCM einen *Delay-Locked Loop*. Ein spezieller *Frequenzteiler* ermöglicht es dem DCM ein neues Taktsignal zu generieren. Dank der *Phasenschieber* können Signale zudem mit einer gewünschten Phasenverschiebung erzeugt werden.
Ein Vorteil der DCMs besteht darin, dass sie einen eigenen Speicher für die Konfiguration besitzen und deshalb dynamisch über den DRP rekonfiguriert werden können. Der Inhalt normaler Konfigurationszellen wird dabei nicht verändert.

Die PLL erzeugt aus einem gegebenen Taktsignal ein neues Signal mit einer gewünschten Frequenz. Generell kann sie dabei einen großen Frequenzbereich abdecken und gleichzeitig die Taktflanken der Ein- und Ausgangssignale miteinander synchronisieren (jitter filter). Die einzelnen Blöcke des PLLs sind aus Abb. 6.22 ersichtlich.

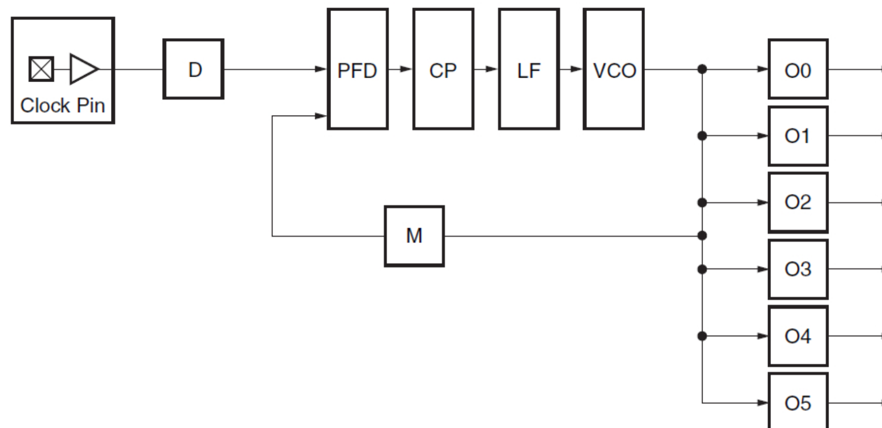

Abbildung 6.22: Aufbau eines Virtex-5 PLL [Xil10i].

Das Eingangs- bzw. Referenztaktsignal wird an einen *Phase-Frequency Detector (PFD)* weitergeleitet. Dieser vergleicht die Phase und die Frequenz des Eingangssignals mit dem rückgekoppelten Ausgangssignals. Das Ausgangssignal des PFD ist proportional zur Phasen- und Frequenzdifferenz beider Signale. Die *Charge Pump (CP)* und das nachfolgende *Loop Filter (LF)* erzeugen daraus eine Referenzspannung für den *Voltage-Controlled Oscillator (VCO)*.

Der VCO erzeugt, in Abhängigkeit dieser Referenzspannung, ein Ausgangssignal mit einer entsprechenden Frequenz. Steigt die Referenzspannung, dann erzeugt der VCO eine höhere Frequenz. Der PFD regelt den VCO solange nach, bis die Frequenz des Ein- und Ausgangssignals miteinander übereinstimmen. Zusätzlich kann die Ausgangsfrequenz mit einem *Zähler (M)* um einen gewünschten Faktor verkleinert oder vergrößert werden. Verkleinert man die Frequenz des rückgekoppelten Ausgangssignals beispielsweise auf die Hälfte, dann wird die Ausgangsfrequenz des VCO doppelt so groß wie die Eingangsfrequenz des PFD.

Insgesamt werden vom VCO acht Ausgangssignale mit gleicher Frequenz, aber unterschiedlicher Phasenverschiebung erzeugt, welche für die FPGA-Logik zur Verfügung stehen.

Integrierter Block-RAM

Die FPGAs der Virtex-5 QV-Familie besitzen einen integrierten Block-RAM mit jeweils 36-kBit großen Blöcken. Diese bestehen aus zwei getrennten 18-kBit RAMs und können z.B. als 32k x 1 bis 1k x 36-Bit Speicher konfiguriert werden. Jeder Speicherblock (36k und 18k) verfügt über zwei unabhängige Schnittstellen (dual-port block RAM).

Ein einzelner Block kann zudem im sogenannten *Simple Dual-Port RAM*-Modus betrieben werden. Der wesentliche Vorteil des Simple-Modus besteht darin, dass der Speicherblock mit einem *Error Detection And Correction (EDAC)*-Modul ausgestattet ist. Alle EDAC-Module verwenden einen *(72,64) Hamming-Code* mit acht Parity-Bits (siehe Abb. 6.23).

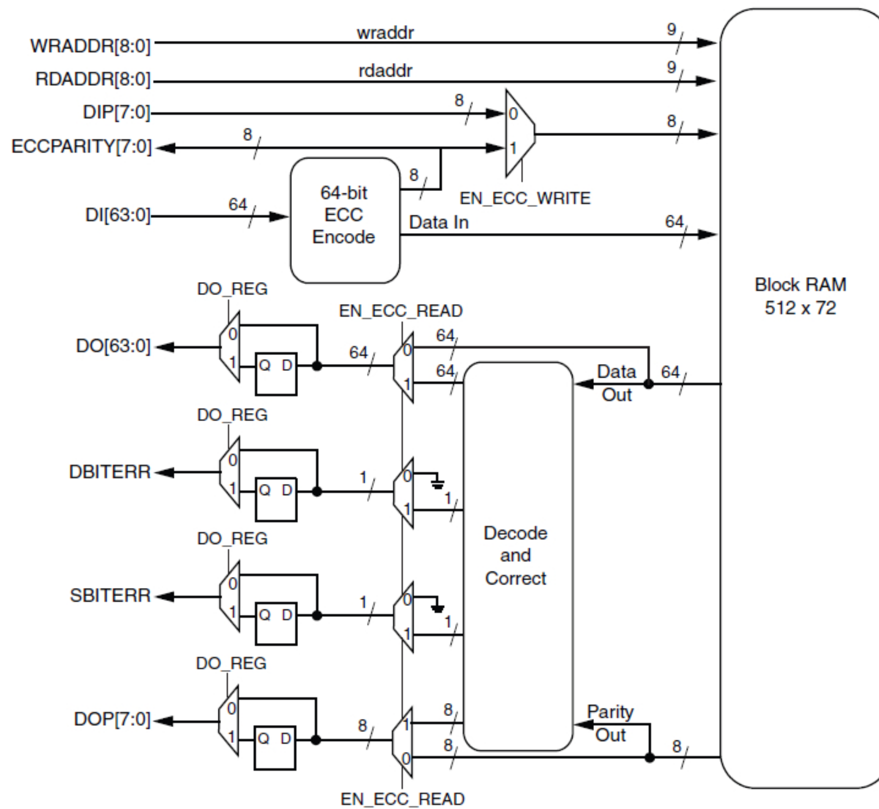

Abbildung 6.23: Simple Dual-Port RAM mit integrierter Fehlerkorrektur [Xil10i].

Die Partity-Bits können optional bei jedem Schreibvorgang vom EDAC-Modul (ECC encode) generiert und über die DIP-Leitungen in den Speicher geschrieben werden. Mit dem Hamming-Code kann ein einzelner Bitfehler korrigiert und maximal zwei Bitfehler detektiert werden. Darüberhinaus ist keine Detektion oder Korrektur möglich. Das Auslesen der Daten- und Parity-Bits erfolgt entsprechend über die DO- und DOP-Leitungen.

Auf Wunsch kann der Block-RAM auch als FIFO-Speicher konfiguriert werden. Dabei wird jeweils ein Port als Schreib- bzw. Leseschnittstelle verwendet.

RocketIO GTX Transceiver

Der XQR5VFX130-Baustein enthält insgesamt 18 *RocketIO GTX Transceiver*. Dabei handelt es sich um spezielle *Multi-Gigabit Transceiver (MGT)*. Sie ermöglichen den Einsatz verschiedener Standards für die *serielle high-speed Datenübertragung*, wie z.B. SATA, PCIe oder 1-Gbit Ethernet. Die RocketIO GTX Transceiver verfügen über eine variable Datenrate von 150 Mbit/s bis 3.125 Gbit/s, welche vom verwendeten Übertragungsstandard abhängt.
Die wichtigste Funktion eines MGT ist die Umwandlung paralleler Bits in einen seriellen Datenstrom. Deshalb wird ein MGT oft auch als *Multi-Gigabit Serializer/Deserializer (SERDES)* bezeichnet.

Grundsätzlich lässt sich ein MGT in einen *Sender (TX)* und *Empfänger (RX)* unterteilen. Im Sender werden n parallele Bits mit der Datenrate r aus dem FPGA übernommen und mit einem

Serializer zu einem seriellen Bitstream mit der Datenrate $r * n$ umgewandelt. Entsprechend werden die empfangenen seriellen Bits wieder vom Empfänger mit einem *Deserializer* in parallele Bits rückgewandelt. Der grundlegende Aufbau eines MGT ist aus Abb. 6.24 ersichtlich.

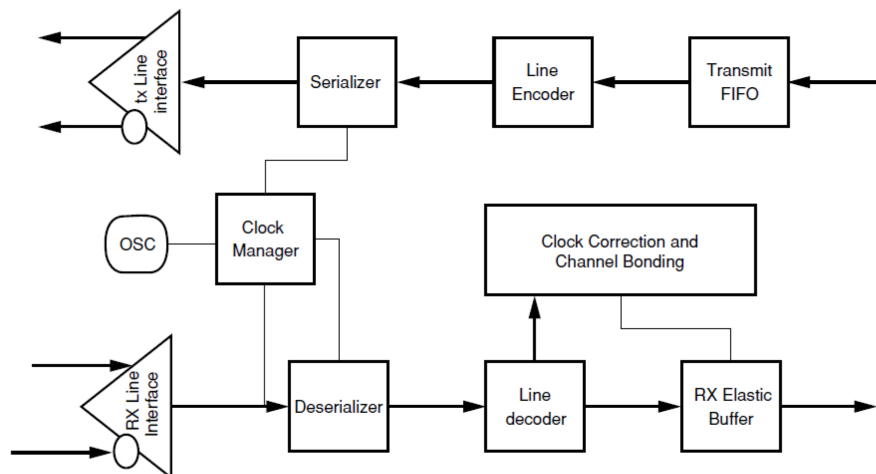

Abbildung 6.24: Grundlegender Aufbau eines Multi-Gigabit Transceivers [AC05].

Der Clock Manager erzeugt und steuert die benötigten Taktsignale. Entsprechende Korrekturen und Anpassungen der RX/TX-Taktsignale werden vom *Clock Correction and Channel Bonding*-Block durchgeführt. Zur Zwischenspeicherung der Daten werden FIFO-Caches verwendet. Die Umwandlung der analogen Signale des Kanals und der seriellen Bits übernehmen die *RX/TX Line Interfaces*. Zusätzlich können konfigurierbare *Line Decoder/Encoder* verwendet werden.

Der tatsächliche Aufbau der RocketIO GTX Transceiver weicht jedoch teilweise von den genannten Komponenten eines MGTs ab. Dies liegt daran, dass sein Funktionsumfang durch zusätzliche Blöcke erweitert wird. Zur Verdeutlichung dieses Umstandes ist der Sender des RocketIO GTX Transceivers in Abb. 6.25 dargestellt.

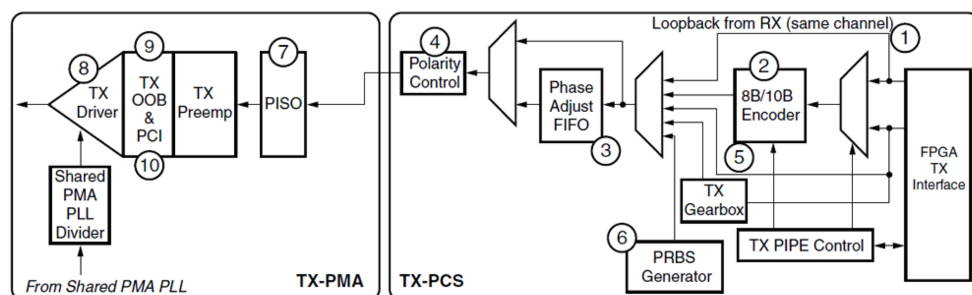

Abbildung 6.25: Aufbau eines RocketIO GTX Transceiver Sender-Blocks [Xil09c].

Eine detaillierte Beschreibung der Funktionsweise sowie der verschiedenen Blöcke kann aus [AC05] und [Xil09c] entnommen werden.

SelectIO Ressourcen

Die I/O-Beschaltung eines Virtex-5 QV-FPGAs besteht aus jeweils einem IOB-, ILOGIC-, OLOGIC- und IODELAY-Block (siehe Abb. 6.26).

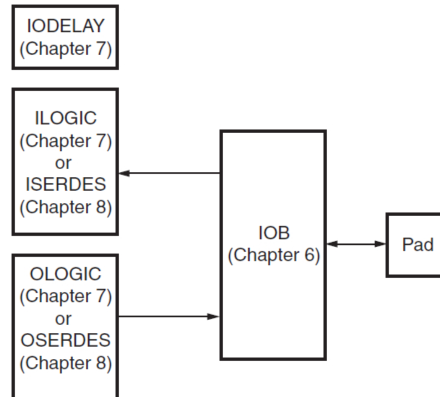

Abbildung 6.26: Blöcke der Virtex-5 I/O-Beschaltung [Xil10i].

Die IOBs der Virtex-5 QV-FPGAs enthalten einen DCI-Controller zur Konfiguration der Impedanzen. Die verschiedenen Ein- und Ausgangstreiber und der 3-State-Treiber ermöglichen die Verwendung zahlreicher I/O-Standards.

Die ILOGIC- und OLOGIC-Blöcke sind direkt mit dem IOB verbunden und beinhalten die Logikschaltungen zur Signalerzeugung und -erkennung. Alternativ können beide Blöcke als Seriell-Parallel-Wandler eingesetzt werden. Ein zusätzlich vorhandener IODELAY-Block kann optional mit dem ILOGIC- oder OLOGIC-Block verbunden werden. Dieser ermöglicht die Verzögerung (delay) der I/O-Signale.

Konfiguration

Die Virtex-5 QV-FPGAs sind mit flüchtigen SRAM-Konfigurationszellen ausgestattet. Beim Einschalten müssen deshalb die Konfigurationsdaten in den FPGA geschrieben werden. Dementsprechend wird ein externer Speicher für die Konfiguration benötigt.

Zur Konfiguration stehen standardmäßig die *Master/Slave-Serial*, die *Master/Slave-SelectMAP* und die *JTAG/Boundary-Scan*-Methoden zur Auswahl. Als Erweiterung gegenüber der Virtex-4 QV-Familie können zusätzlich die *Master Serial Peripheral Interface (SPI) Flash* und die *Master Byte Peripheral Interface Up/Down (BPI Up/Down)*-Methoden verwendet werden.

Die Konfiguration kann, wie bei den Virtex-4 FPGAs, in insgesamt acht Schritte unterteilt werden.

Rekonfiguration

Bei den Virtex-5 QV-FPGAs kann ein *Readback* über den *ICAP*-, die *JTAG/Boundary-Scan*- oder die *SelectMAP*-Schnittstelle durchgeführt werden.

Beim sogenannten *Readback Verify* werden alle Zustände aus den Konfigurationszellen, dem Block-RAM und anderen Speicherelementen ausgelesen. Zusätzlich können die Register aller CLBs und IOBs bei einem *Readback Capture* ausgelesen werden, wodurch sich diese Methode besonders für die Verifikation (debugging) eignet.
Die beim Readback ermittelten Daten (readback data stream) werden anschließend mit der Originalkonfiguration verglichen. Dabei können einige Bits ausgelassen werden, weil diese z.B. einem leeren Speicher entsprechen. Die Positionen dieser unwichtigen Bits werden in sogenannten MSK- oder MSD-Dateien gespeichert. Die eigentliche Verifikation kann auf zwei Arten erfolgen.

Bei der ersten Methode muss eine spezielle *Golden Readback (RBD)*-Datei mit den Readback-Daten der Originalkonfiguration vorhanden sein. Die RBD-Datei wird dann mit den Daten der MSD-Datei verglichen. Der Vorteil dabei ist, dass beide Dateiformate miteinander nahezu übereinstimmen und leicht verglichen werden können. Bei dieser Methode muss die RBD-Datei in einem Speicher hinterlegt werden, was wiederum ein Nachteil ist.
Die zweite Methode vergleicht die MSK-Datei mit den originalen Konfigurationsdaten aus der BIT-Datei. Die Struktur beider Dateien ist aus Abb. 6.27 ersichtlich.

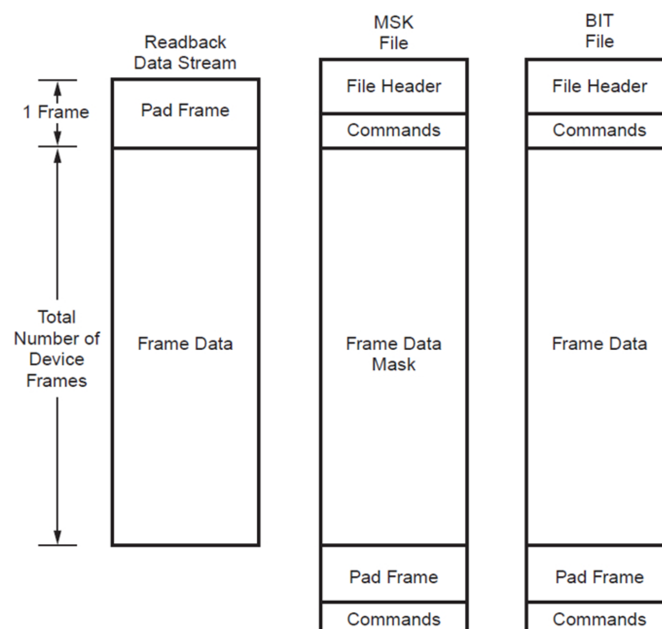

Abbildung 6.27: Verifikation der Readback-Daten mit BIT-Datei [Xil10h].

Der Vorteil dieser Methode liegt darin, dass nur die BIT, MSK und Readback-Befehle gespeichert werden müssen.

Zusätzlich zu den bereits besprochenen Readback-Methoden wurde in die Virtex-5 QV-FPGAs eine *Error Correction Code (ECC)*-Logik und ein *Readback CRC*-Algorithmus implementiert. Der ECC kann bei einem Readback verwendet werden, um einzelne Bitfehler zu detektieren und anschließend zu korrigieren. Dazu wird der Readback kurzzeitig angehalten. Der CRC-Code kann ebenfalls zur Überprüfung der Konfiguration eingesetzt werden.

Der Inhalt der Konfigurationszellen definiert, neben der vom Entwickler gewünschten Logik, auch die Funktionsweise der funktionalen Logikblöcke (functional blocks). In diese Kategorie

fallen z.B. die Digital Clock Managers. Jeder funktionale Block besitzt jedoch einen eigenen adressierbaren Speicher für die Konfiguration.
Als *Dynamic Reconfiguration of Functional Blocks* bezeichnet man die dynamische Rekonfiguration der funktionalen Logikblöcke im laufenden Betrieb. Dies erfolgt über den speziellen *Dynamic Rekonfiguration Port (DRP)*. Bei der Virtex-5 QV-Familie können dadurch funktionale Blöcke zur Laufzeit, ohne Beeinflussung der eigentlichen Logikkomponenten, rekonfiguriert werden.

6.1.4 Zusammenfassung

Die Architektur der Xilinx space-grade Virtex-FPGAs basiert gänzlich auf SRAM-Konfigurationszellen. Diese reagieren prinzipiell sehr empfindlich auf die Strahlung, wodurch gehäuft SEEs auftreten.
Als Gegenmaßnahme sind die Virtex-4 QV- und Virtex-5 QV-FPGAs mit zahlreichen Komponenten zur Vermeidung oder Behebung der Störungen ausgestattet. Dazu zählen Methoden wie Readback, Scrubbing oder ECC. Erweiterte Maßnahmen wie z.B. TMR können teilweise mit einer Entwicklungssoftware von Xilinx (TMRTool) realisiert werden.

Die wichtigsten Eigenschaften der einzelnen Xilinx-Familien nochmals im Überblick:

	Xilinx space-grade FPGAs		
	Virtex-5 QV	Virtex-4 QV	Virtex-II XQR
Zellentyp	SRAM	SRAM	SRAM
Logic Slices	40.960	< 89.088	< 33.792
Block-RAM (kBit)	10.728	< 9.936	< 2.592
TID	700 krad	300 krad	200 krad
SEL Immunität	>100	> 100	> 160
SEFI in GEO	2.76E-7	1.5E-6	< 1.5E-6
Fehlerdetektion/-korrektur	ja	ja	nein
SEU-gehärtete Flip-Flops	ja	nein	nein
Core Voltage	1.0 V	1.2 V	1.5 V
DSP Slices	DSP48E	DSP48	-
18 x 18 Multiplier	-	-	< 144
PowerPC 405	-	2	-

Tabelle 6.2: Die wichtigsten Eigenschaften der space-grade Virtex-FPGAs.

Weitere Eigenschaften können aus den entsprechenden Datenblättern entnommen werden.

6.2 Actel radiation-hardened FPGA-Familien

Mit einem weltweiten Marktanteil von nur wenigen Prozent ist *Actel* ein vergleichsweise kleiner Produzent. Das Unternehmen hat sich allerdings sehr stark auf die Entwicklung strahlungsresistenter FPGAs für Luft- und Raumfahrtanwendungen spezialisiert.
Sämtliche strahlungsgehärtete FPGA-Familien basieren auf nichtflüchtigen Antifuse- oder Flash-Zellen. Diese reagieren viel unempfindlicher auf Strahleneinflüsse und vermindern die Häufigkeit von Störungen durch SEEs.

Ein weiterer Unterschied zu den bereits vorgestellten FPGA-Familien von Xilinx besteht darin, dass statt der klassischen FPGA-Architektur eine neuartige *Sea-of-Modules-Architektur* verwendet wird. Diese besteht nicht mehr aus einzelnen Logik-Blöcken mit Verbindungsmatritzen, sondern aus komplett vernetzten Logikmodulen. Die beiden unterschiedlichen Ansätze sind aus Abb. 6.28 ersichtlich.

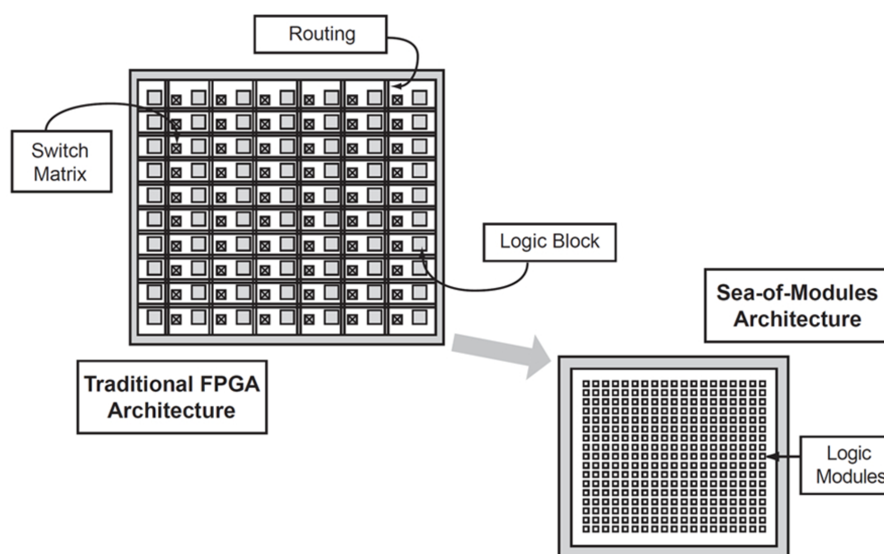

Abbildung 6.28: Sea-of-Modules-Architektur der Actel-FPGAs [Act10d].

6.2.1 Die RTAX-S/SL-Familie

Die *RTAX-S/SL*-Familie basiert auf einer high-performance Sea-of-Modules (SOM)-Architektur. Die FPGAs zeichneten sich besonders dadurch aus, dass sie für kritische Anwendungen in strahlenden Umgebungen entwickelt wurden. Aus diesem Grund wurden die empfindlichen Register speziell vor Störungen geschützt und großer Wert auf eine hohe TID- und SEE-Immunität gelegt.

Die wichtigsten Eigenschaften der RTAX-Familie sind [Act10d]:

- 1.5 V Versorgungsspannung
- 0.15 μm 7-Layer CMOS Antifuse-Prozess
- SEU-gehärtete Register (Flip-Flops)
- Garantierte TID von 300 krad
- Latch-up Immunität für LET > 117 $MeV cm^2/mg$

- EDAC Fehlerdetektierung und -korrektur für RAM mit Scrubber
- SEU in GEO < 10E-10 upsets/bit/day mit aktiviertem EDAC
- Max. 20.160 SEU-hardened Flip-Flops
- Max. 540 kBit SRAM
- Max. 840 SEU-geschützte I/O-Anschlüsse
- Antifuse-basierte Konfiguration mit Programmiergerät
- IEEE 1149.1 - JTAG/Boundary-Scan Support
- Max. 120 Embedded Multiply/Accumulate Blocks (nur RTAX-DSP)

Die RTAX-S/SL-Familie besteht aus insgesamt vier FPGAs, welche sich in ihrer Größe voneinander unterscheiden. Bei den SL-Typen handelt es sich um eine Low-Power-Variante der RTAX-S-FPGAs.

Grundlegende Architektur

Als Hauptkomponenten werden bei den RTAX-FPGAs sogenannte *Logikkerne (Core Tiles)* verwendet, welche die konfigurierbare Logik beinhalten. Diese sind entsprechend der SOM-Architektur regelmäßig über die gesamte Chipfläche angeordnet. Die kleineren *I/O-Blöcke (I/O Structures)* dienen als Schnittstelle zwischen den Logikkernen und den Anschlüssen. Sie sind deshalb an den vier Seiten des FPGAs, in einer Reihe, angeordnet.
Jeder Logikkern besteht zudem aus einer Vielzahl an *SuperCluster*- und einigen *4-kBit RAM/-FIFO*-Blöcken. Die Anzahl der einzelnen Blöcke hängt von der Größe des verwendeten FPGAs ab. Ein RTAX1000S/SL-Baustein besitzt beispielsweise neun Logikkerne mit jeweils 176 Super-Clusters und drei SRAM/FIFO-Blöcken. Die soeben genannten Blöcke der RTAX-Architektur sind in Abb. 6.29 dargestellt.

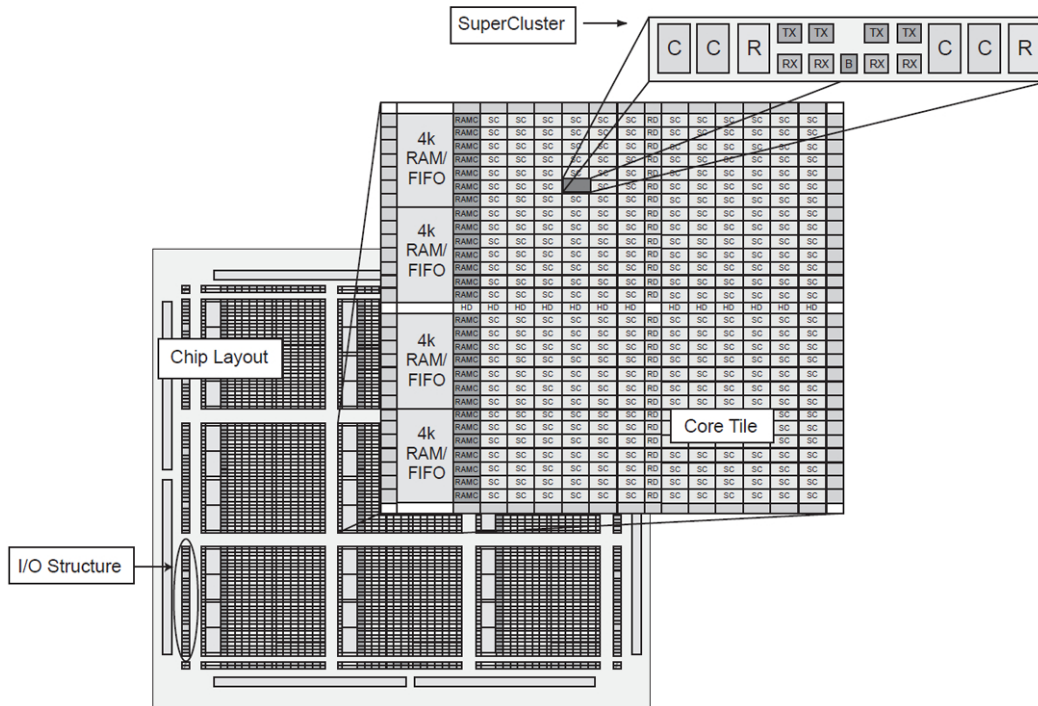

Abbildung 6.29: Architektur eines RTAX-FPGAs [Act10d].

Die einzelnen SuperCluster bestehen aus mehreren *R*- und *C-Zellen*, sowie verschiedenen *RX*- und *TX-Blöcken*. Mit ihnen können die gewünschten Logikfunktionen implementiert werden. Dazu werden die einzelnen SuperCluster über Antifuse-Zellen und spezielle Verbindungsstrukturen miteinander verbunden.

Zusätzlich sind in den RTAX-FPGAs sämtliche Flip-Flops (register) gegen SEUs geschützt. Diese sogenannten *SEU Enhanced Flip-Flops* bestehen aus drei speziellen Flip-Flops die zu einem TMR-System verbunden sind (siehe Abb. 6.30).

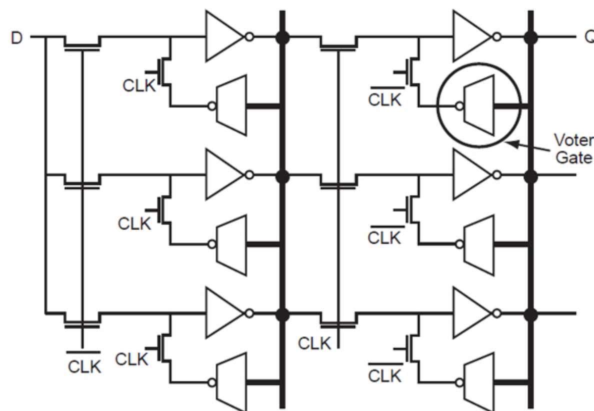

Abbildung 6.30: SEU hardened Flip-Flop eines RTAX-FPGAs [Act10d].

SuperCluster-Block

Die sogenannten *Cluster* bestehen aus zwei *C-Zellen (combinatorial cells)*, einer *R-Zelle (register cell)* und jeweils zwei *RX/TX Routing Buffers*. In der RTAX-Architektur werden zwei Cluster und ein zusätzliches *independent Buffer*-Modul zu einem großen *SuperCluster*-Block zusammengefasst. Die einzelnen Teile der SuperCluster sind aus Abb. 6.31 ersichtlich.

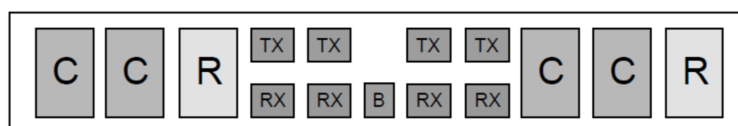

Abbildung 6.31: SuperCluster-Block der RTAX-Architektur [Act10d].

Mit einer *C-Zelle* können verschiedene kombinatorische Logikfunktionen, mit bis zu fünf Eingängen, erzeugt werden. Sie beinhalten ebenfalls eine von außen zugängliche Carry-Logik für arithmetische Funktionen. Durch Verknüpfung der einzelnen C-Zellen können somit größere arithmetische Funktionen erzeugt werden.
Eine *R-Zelle* ermöglicht die Implementierung der sequentiellen Logik und enthält ein gegen SEUs geschütztes Flip-Flop. Jede R-Zelle kann beispielsweise als einzelnes Flip-Flop konfiguriert oder mit benachbarten C-Zellen verbunden werden. Die RX/TX Routing-Buffer sind an die Verbindungsstrukturen angeschlossen und können zur Verknüpfung verschiedener SuperCluster verwendet werden.

Verbindungsstruktur

Bei allen SOM-Architekturen sind die einzelnen Module über eine ausgeklügelte *Verbindungsstruktur* miteinander verbunden, welche in der Literatur meist als *Routing Structure* bezeichnet wird.
Betrachtet man die SuperCluster-Blöcke, dann sind diese über *FastConnect*-, *DirectConnect*- und *CarryConnect*-Strukturen miteinander verbunden. Als DirectConnect bezeichnet man eine direkte und sehr schnelle Verbindung zwischen R- und C-Zellen, welche keine Antifuse beinhaltet. Zur direkten Verknüpfung der Carry-Logic (carry chain) werden die CarryConnect-Strukturen verwendet. Mit FastConnect-Strukturen werden die Module des SuperClusters, über jeweils eine einzelne Antifuse-Zelle, mit den horizontalen *Output Tracks* verbunden. Die genannten Verbindungsstrukturen sind in Abb. 6.32 dargestellt.

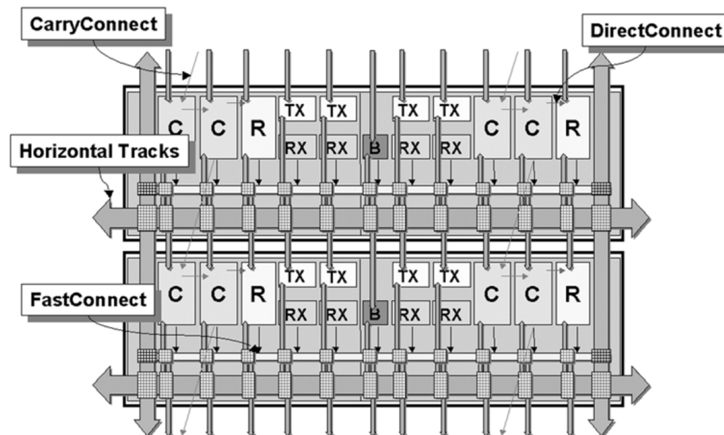

Abbildung 6.32: Verbindungsstruktur der RTAX SuperCluster [Act10d].

Die Kommunikation zwischen den SuperCluster-Blöcken und den anderen Logikkernen (core tiles) erfolgt zudem über horizontale und vertikale Hauptleitungen (tracks). Zur Verknüpfung der Komponenten und Leitungen werden mehrere Antifuse-Zellen verwendet (siehe Abb. 6.33).

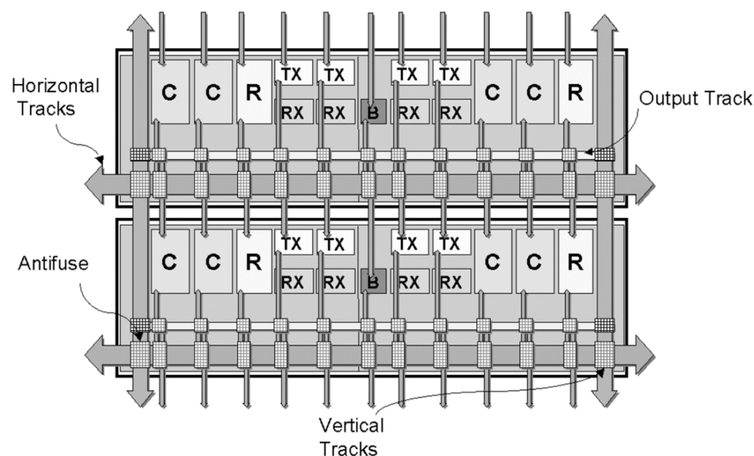

Abbildung 6.33: Hauptverbindungen zwischen den einzelnen Logikkernen [Act10d].

Mit den RX/TX Routing Buffers können die Hauptleitungen mit den Output Tracks innerhalb der SuperCluster verbunden werden.
Dank dieser komplexen Strukturen sind zahlreiche Logikfunktionen implementierbar. Einzelne Zellen der SuperCluster können so beliebig mit anderen Zellen, Logikkernen oder den I/O-Strukturen verbunden werden.

Eingebetteter RAM/FIFO-Speicher

Die RTAX-Architektur beinhaltet zusätzlich einige 4-kBit SRAM-Blöcke, welche optional als FIFO-Speicher betrieben werden können. Die einzelnen Blöcke können als 128 x 36-Bit bis 4k x 1-Bit Speicher konfiguriert werden. Eine Verknüpfung zu einem größeren Speicher ist ebenfalls möglich. Für den FIFO-Speicher wurde eine eigene Steuerung (controller) implementiert.

Als Erweiterungen zum normalen Funktionsumfang, enthalten alle RTAX-FPGAs einen *EDAC*- und *Background Memory-Refresher (Scrubber)*-Algorithmus. Der EDAC verwendet einen verkürzten Hamming-Code, wodurch einzelne Bitfehler zur Laufzeit korrigiert und zwei fehlerhafte Bits detektiert werden können.

Taktsignale

In der Literatur von Actel werden die Taktsignal-Komponenten meist als *globale Ressourcen (global ressources)* bezeichnet. Diese Ressourcen lassen sich in eine sogenannte *Hardwired Clock (HCLK)* und eine *Routed Clock (CLK)* unterteilen.
Jeder RTAX-S/SL-FPGA verfügt über vier HCLK- und vier CLK-Blöcke. Eine HCLK wird für die Taktung der gesamten sequentiellen Logik verwendet und mit einem Netzwerk im ganzen FPGA verteilt. Die CLKs können zusätzlich zur Steuerung von kombinatorischen Eingängen, Set/Reset-Eingängen und Enable-Eingängen verwendet werden. Sie verfügen somit über einen größeren Funktionsumfang und ermöglichen eine flexiblere Verwendung. Zur Verteilung der Taktsignale werden spezielle Netzwerke verwendet (siehe Abb. 6.34).

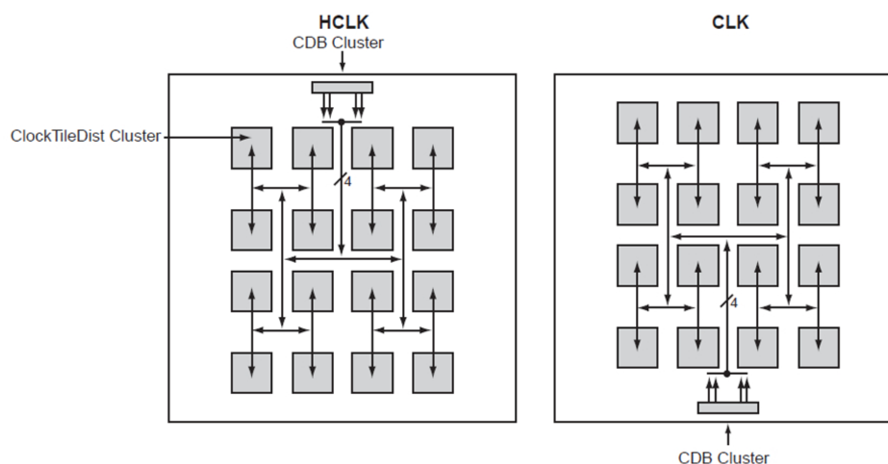

Abbildung 6.34: Verteilung der verschiedenen Taktsignale [Act10d].

Dazu befinden sich am Ausgangspunkt für die HCLK und CLK jeweils vier *Clock Distribution Buffer (CDB)*, welche über das Netzwerk mit den *Clock Tile Distribution (CTD) Clusters* verbunden sind. Von dort werden die Taktsignale in den Logikkernen verteilt.

Input/Output-Strukturen

Die I/O-Blöcke der RTAX-FPGAs werden als *I/O-Cluster* bezeichnet. Diese bestehen aus zwei *I/O-Modulen*, vier *RX*-Modulen, zwei *TX*-Modulen und einem *Buffer*.
Jedes I/O-Modul besteht aus einem Input-, Output- und Enable-Register. Für die Register werden, wie bei den R-Zellen, spezielle SEU-hardened Flip-Flops verwendet. Der Aufbau eines I/O-Clusters ist aus Abb. 6.35 ersichtlich.

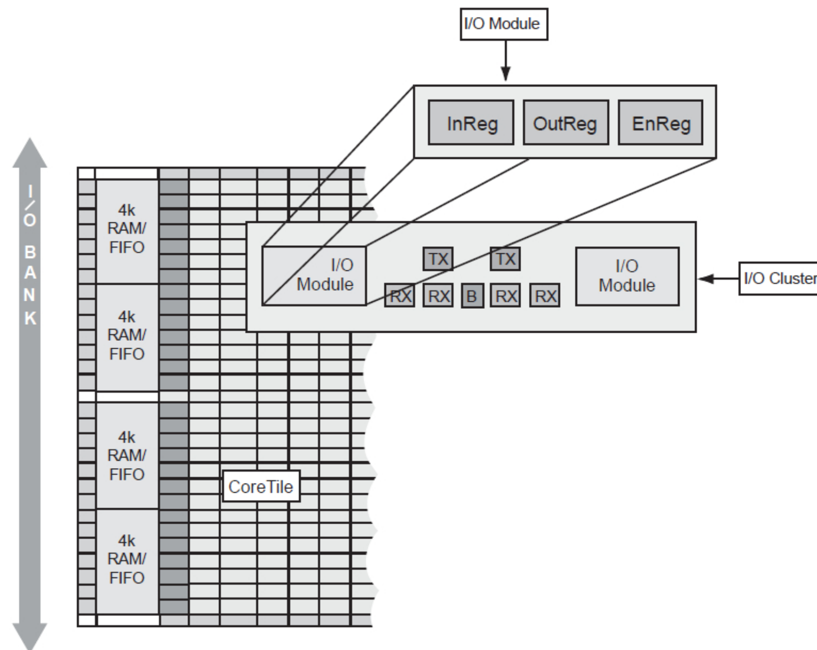

Abbildung 6.35: Aufbau der IO-Blöcke eines RTAX FPGAs [Act10d].

Die Anschlüsse des FPGAs und die damit verbundenen I/O-Cluster werden in insgesamt acht I/O-Bänke unterteilt. Jede Bank kann auf einen von insgesamt 14 verschiedenen I/O-Standards konfiguriert werden. Die Standards umfassen unsymmetrische (single-ended), symmetrische (differential) und spannungsreferenzierte (voltage-referenced) Signale.

Konfiguration

Die Konfiguration der RTAX-FPGAs erfolgt über spezielle Programmiergeräte. Auf Grund der Antifuse-Zellen kann der FPGA nur ein einziges mal programmiert werden. Eine Rekonfiguration im herkömmlichen Sinn ist somit nicht möglich.
Der Nachteil der fixen Programmierung wird allerdings dadurch kompensiert, dass die Antifuse-Zellen gegenüber den möglichen Strahlungseinflüssen (SEEs) unempfindlich sind.

6.2.2 Die RTAX-DSP-Familie

Die *RTAX-DSP*-Familie besteht aus lediglich zwei FPGAs. Diese verfügen über den kompletten Funktionsumfang der *RTAX-S/SL*-Familie und besitzen auch prinzipiell die selbe, auf Antifuse-Zellen basierende, SOM-Architektur.

Der einzige wesentliche Unterschied besteht darin, dass die FPGAs der RTAX-DSP-Familie, zusätzlich zum normalen Funktionsumfang, einige spezielle *DSP Mathematik-Blöcke* für die Signalverarbeitung besitzen. Diese sind in den Logikkernen (Core Tiles), zwischen den RAM/FIFO- und den SuperCluster-Blöcken, angeordnet (siehe Abb. 6.36).

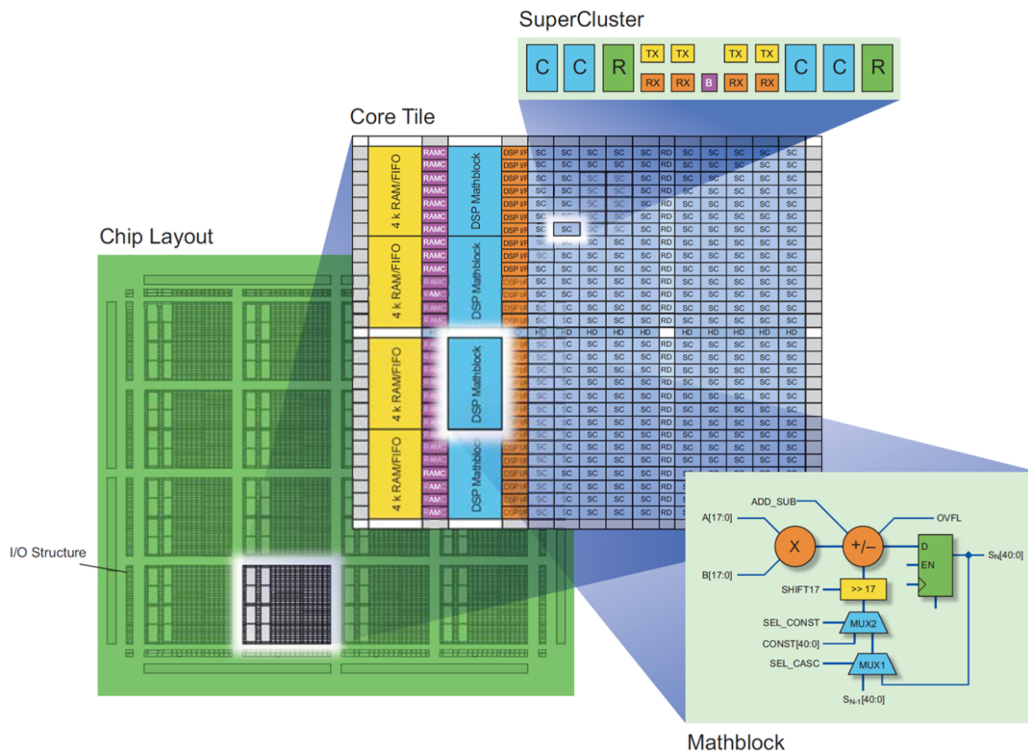

Abbildung 6.36: Architektur eines RTAX-DSP FPGAs [Act10d].

DSP-Mathematikblock

Ein *DSP-Mathematikblock (DSP Mathblock)* kann zur Implementierung komplexer Systeme für die Signalverarbeitung verwendet werden. Jeder dieser DSP-Blöcke besteht aus einem 18 x 18-Bit Multiplizierer (multiplier), einem Addierer/Subtrahierer-Block (add/subtract) und einem Ausgangs-Register (output register). Die DSP-Blöcke sind, im Vergleich zu den Xilinx DSP48E, relativ klein und einfach aufgebaut.

Mit den zwei zusätzlichen Multiplexern (MUX1, MUX2) kann der Ausgangswert des Registers zum Addierer/Subtrahierer-Block rückgekoppelt werden, wodurch eine Multiplizierer/Akkumulator (multiply/accumulate)-Funktion entsteht. Das Schieberegister wird zur Erzeugung eines präziseren Multiplikators verwendet.

Komplexere DSP-Funktionen, wie beispielsweise FFT/IFFT-Funktionen oder FIR/IIR-Filter, lassen sich durch die Verknüpfung mehrerer DSP-Blöcke erzeugen. Dazu wird der Ausgang eines anderen DSP-Blocks mit dem Eingang des Verknüpfungs-Multiplexers (MUX1) verbunden.

6.2.3 Die RTSX-SU-Familie

Bei der *RTSX-SU*-Familie handelt es sich um eine strahlungsresistente SOM-Architektur, welche auf der Actel SX-A-Familie basiert. Bei diesen FPGAs wurden die empfindlichen Register (Flip-Flops) ebenfalls vor strahlungsbedingten Störungen geschützt und entsprechend auf eine relativ hohe TID- und SEE-Immunität geachtet.

Die wichtigsten Eigenschaften der RTSX-SU-Familie sind [Act10e]:

- 3.3 - 5 V Versorgungsspannung
- SEU-gehärtete Register (Flip-Flops)
- Garantierte TID von 100 krad
- SEL Immunität für LET > 104 $MeVcm^2/mg$
- SEU in GEO < 10E-10 upsets/bit/day
- 0.25 μm CMOS-Antifuse-Prozess
- Max. 4.024 kombinatorische Zellen (C cells)
- Max. 2.012 SEU-hardened register cells (Flip-Flops)
- Max. 360 I/O-Anschlüsse
- Antifuse-basierte Konfiguration mit Programmiergerät
- IEEE 1149.1 - JTAG/Boundary-Scan Support

Die RTAX-S/SL-Familie besteht aus dem RTSX32SU- und dem RTSX72SU-Baustein. Diese unterscheiden sich im Wesentlichen in ihrer Größe voneinander. Die RTSX-SU-FPGAs verfügen, im Vergleich zur RTAX-Familie, über einen geringeren Funktionsumfang.

Grundlegende Architektur

Die RTSX-SU-Familie verfügt über eine SOM-Architektur, welche auf einem 0.25 μm CMOS-Antifuse-Prozess basiert. Mehrere *Logik-Blöcke* beinhalten die konfigurierbare sequentielle und kombinatorische Logik und sind regelmäßig über die gesamte Chipfläche angeordnet. Die Logik-Blöcke bestehen aus einer Vielzahl an *SuperCluster*-Blöcken. Die Anzahl der einzelnen Super-Cluster hängt von der Größe des verwendeten FPGAs ab.
Die *I/O-Blöcke* dienen als Schnittstelle zwischen den Logik-Blöcken und den Anschlüssen. Zur Übertragung der Signale werden in den RTSX-SU-FPGAs verschiedene *Verbindungsstrukturen* verwendet.

SuperCluster-Block

Ähnlich wie bei der RTAX-Familie, bestehen die *SuperCluster*-Blöcke der RTSX-SU-Familie aus zwei *Clusters*. Die FPGAs verfügen über zwei unterschiedliche Cluster-Typen, welche wiederum aus insgesamt drei C- oder R-Zellen aufgebaut sind.

Die C-Zellen (conbinatorial cells) werden zur Realisierung der kombinatorischen Logik verwendet. Mit den R-Zellen (register cells) wird hingegen die sequentielle Logik erzeugt. Sie bestehen prinzipiell aus einem speziellen gegen SEUs geschützten (SEU hardened) Flip-Flop. Der Schutz wird dadurch erreicht, dass diese Flip-Flops als TMR-System implementiert sind.
Der erste Cluster-Typ besteht aus einer R- und zwei C-Zellen (C-R-C) und der zweite Typ aus einer C- und zwei R-Zellen (C-R-R). Zusätzlich kann zwischen zwei SuperCluster-Typen unterschieden werden. Die verschiedenen Zellen und SuperCluster sind aus Abb. 6.37 ersichtlich.

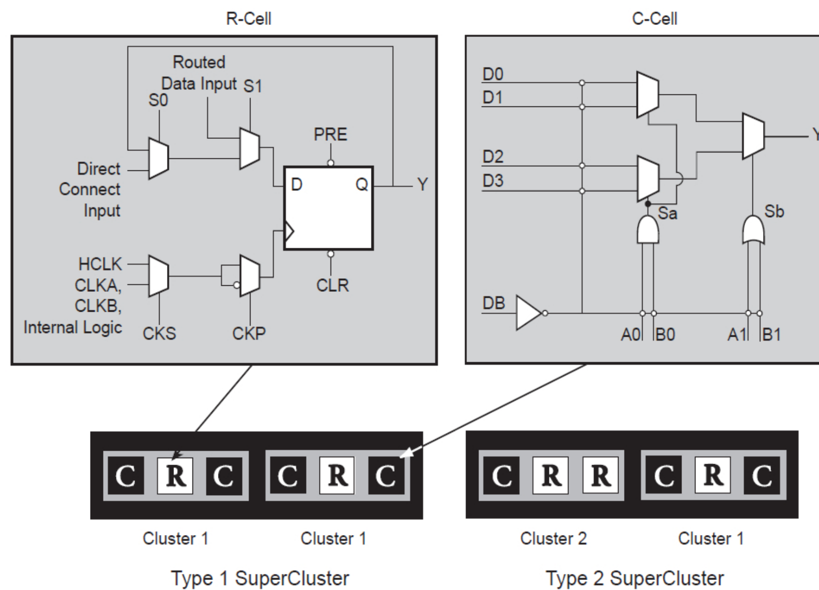

Abbildung 6.37: R- und C-Zelle der RTSX-FPGAs und die SuperCluster-Typen [Act10e].

Taktsignale

Alle RTSX-SU-FPGAs verfügen über eine *Hardwired Clock (HCLK)*- und zwei *Routed Clock (CLK)*-Ressourcen. Der RTSX72SU-Baustein besitzt zusätzlich vier *Quadrant Clocks (QCLKs)* (siehe Abb. 6.38).

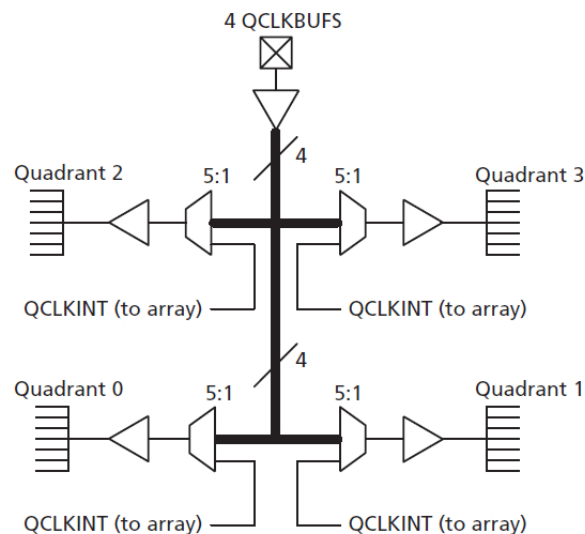

Abbildung 6.38: Struktur der Quadrant Clocks-Ressourcen [Act10e].

Die QCLKs bieten die Möglichkeit, dass in jedem der vier Quadranten des FPGAs ein eigenes Taktsignal zur Verfügung steht. Die HCLK-Ressourcen werden als Taktsignal für die R-Zellen eingesetzt. Die CLKs können zusätzlich zu den R-Zellen auch für die Eingänge der C-Zellen verwendet werden.

Input/Output-Strukturen

Die *I/O-Strukturen* ermöglichen die Verwendung mehrerer 3.3 V und 5 V I/O-Standards. Somit sind die FPGAs mit den gängigsten TTL- oder CMOS-Pegeln kompatibel. Jeder Anschluss kann als Eingang, Ausgang oder im Tri-State-Modus betrieben werden.

Verbindungsstruktur

Zur Verbindung der einzelnen R- und C-Zellen werden *FastConnect*- und *DirectConnect*-Ressourcen verwendet. Mit den DirectConnect-Leitungen werden die R- und C-Zellen innerhalb eines Clusters direkt miteinander verbunden. Verbindungen zwischen den Clusters erfolgen mit den FastConnect-Ressourcen. Diese benötigen für jede Verbindung eine einzelne Antifuse-Zelle, wodurch sie geringfügig langsamer sind.
Für längere Verbindungen werden zusätzliche horizontal und vertikal verlaufende Verbindungsstrukturen verwendet. Diese globalen Netze verbinden die einzelnen SuperCluster miteinander und verwenden zwei bis fünf Antifuse-Zellen. Die verschiedenen Verbindungsstrukturen sind zusammengefasst in Abb. 6.39 dargestellt.

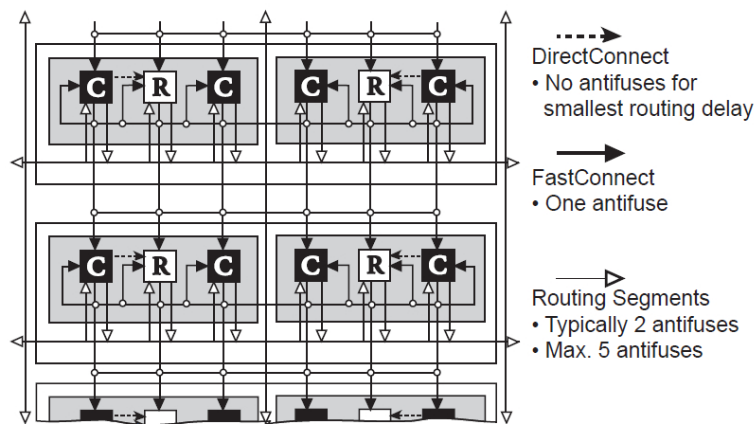

Abbildung 6.39: Verbindungsstrukturen in den RTSX-SU-FPGAs [Act10e].

Konfiguration

Die Konfiguration der RTSX-SU-FPGAs erfolgt mit einem speziellen Programmiergerät. Auf Grund der Antifuse-Zellen kann der FPGA nur ein einziges mal programmiert werden. Eine Rekonfiguration im herkömmlichen Sinn ist somit nicht möglich.

Der Nachteil der fixen Programmierung wird allerdings dadurch kompensiert, dass die Antifuse-Zellen gegenüber den möglichen Strahlungseinflüssen (SEEs) unempfindlich sind.

6.2.4 Die RT ProASIC3-Familie

Bei der RT ProASIC3-Familie handelt es sich um eine strahlungstolerante Variante der low-power ProASIC3EL-Familie. Sie weisen gegenüber den normalen ProASIC3-Bausteinen eine etwa 40% verringerte dynamische und bis zu 90% verringerte statische Leistungsaufnahme auf. Die RT ProASIC3-Familie besteht aus den RT3PE600L- und RT3PE3000L-FPGAs.

Die wichtigsten Eigenschaften der RT ProASIC3-Familie sind [Act08]:

- 1.2 bis 1.5 V Versorgungsspannung
- TID Immunität von 15 krad
- Latch-up Immunität für LET > 70 $MeVcm^2/mg$
- Clock Conditioning Circuits mit eigenem PLL
- 0.13 μm 7-Layer CMOS-Flash-Prozess
- Max. 75.264 VersaTiles
- Max. 504 kBit dual-port SRAM Speicher
- 1kBit Flash-ROM Speicher
- Max. 620 I/O-Anschlüsse
- Flash-basierte In-System-Konfiguration
  - 128-Bit AES-Verschlüsselung
  - FlashLock-Technologie zum Schutz der Konfiguration
- IEEE 1149.1 - JTAG/Boundary-Scan Support

Weitere Informationen zu den Auswirkungen der verschiedenen Strahlungseffekten können aus [Act10a] und [Act10c] entnommen werden.

Grundlegende Architektur

Die Architektur der RT ProASIC3-Familie besteht aus mehreren miteinander verbundenen *VersaTile*-Blöcken. Für die Speicherung der Konfiguration werden nichtflüchtige Flash-Zellen verwendet, welche weniger anfällig für Strahleneinflüsse sind als SRAM-Zellen. Zusätzlich verfügen die FPGAs über einen flüchtigen *SRAM/FIFO*-Speicher, die *Clock Conditioning Circuits und PLLs* sowie spezielle *I/O-Strukturen.*
Die Architektur der RT ProASIC3-Familie, mit den wichtigsten Komponenten, ist aus Abb. 6.40 ersichtlich.

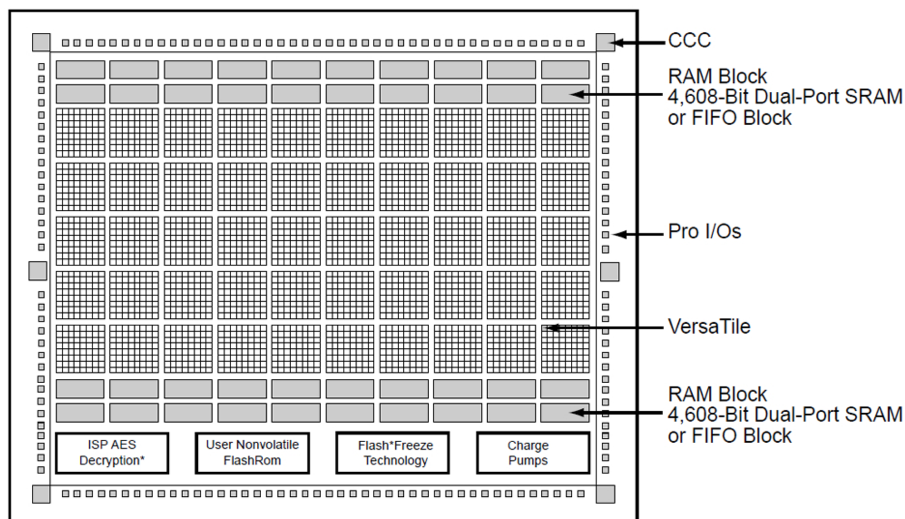

Abbildung 6.40: Architektur der RT ProASIC3-Familie [Act08].

Zur Minimierung der Leistungsaufnahme sind die FPGAs mit der Actel *Flash*Freeze*-Technologie ausgestattet. Damit kann von einem dynamischen zu einem energiesparenderen statischen

Betriebsmodus gewechselt werden. Der *ISP AES Decryption*-Block wird, bei der verschlüsselten Übertragung von Konfigurationsdaten, zur Entschlüsselung des Bitstreams verwendet.

Eingebetteter RAM/FIFO-Speicher

Die RT ProASIC3-FPGAs enthalten mehrere 4.608-Bit große SRAM-Blöcke mit einer konfigurierbaren Wortbreite von 256 x 18-Bit bis 4k x 1-Bit. Durch den zusätzlichen FIFO-Controller können diese auch als FIFO-Speicher verwendet werden. Eine Verknüpfung zu größeren Speicherblöcken ist ebenfalls möglich.

VersaTile-Block

Ein *VersaTile*-Block besteht aus der konfigurierbaren sequentiellen und kombinatorischen Logik. Mit Hilfe der Flash-Zellen kann jeder VersaTile als *3-input LUT*, *Enable Data-Flip-Flop* oder *Latch* konfiguriert werden. Die Komponenten eines VersaTile-Blockes sind aus Abb. 6.41 ersichtlich.

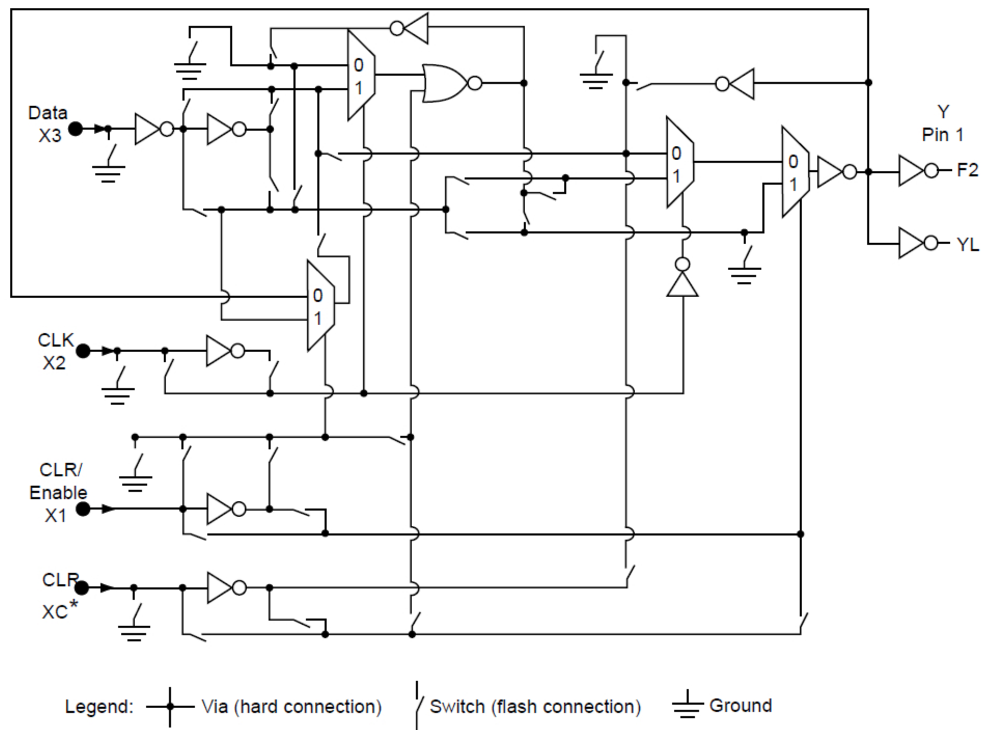

Abbildung 6.41: Aufbau eines ProASIC3 VersaTile-Blocks [Act08].

Verbindungsstrukturen

Die *Verbindungsstrukturen (routing structures)* werden zum Datenaustausch zwischen den einzelnen Komponenten verwendet. Insgesamt stehen vier Arten von Verbindungsstrukturen zur Verfügung, welche horizontal und vertikal durch den gesamten FPGA verlaufen.
Die *ultra-fast local resources* verbinden die Ausgänge eines VersaTiles mit den Eingängen seiner acht nächsten Nachbarn. Für längere Verbindungen stehen die *efficient long-line resources* zur Verfügung. Diese können VersaTiles über einen Bereich (span) von ein, zwei oder vier Blöcken

miteinander verbinden. Mit den *high-speed very-long-line recources* kann jeder beliebige VersaTile innerhalb des gesamten FPGAs erreicht werden.

Als weitere Verbindungsstruktur beinhalten die ProASIC3-FPGAs ein *VersaNet-Netzwerk*, welches die VersaTiles mit den I/O-Blöcken und dem SRAM-Speicher verbindet. Damit kann z.B. die Verteilung der Taktsignale oder der globalen Reset-Signale durchgeführt werden. Der Vorteil der VersaNets besteht darin, dass es sich um schnelle Ressourcen, mit sehr geringem Jitter und einer Vielzahl an möglichen Verzweigungen, handelt.
Für die Ein- und Ausgänge eines VersaTile-Blocks stehen neun VersaNets zur Verfügung. Jedes davon kann mit einem Taktgenerator oder einem anderen VersaTile verbunden sein.
Das Netzwerk wird in sogenannte *Spines*, *Ribs* und mehrere *Spine-Selection*-Multiplexer unterteilt. Die Multiplexer verbinden das globale VersaNet im Zentrum des FPGAs mit den einzelnen vertikal verlaufenden Spines. Diese werden entsprechend ihrer Position in Top- und Bottom-Spines unterteilt. Die Ribs verbinden schlussendlich die einzelnen VersaTiles mit dem nächstgelegenen Spine. Die Struktur des VersaNets ist in Abb. 6.42 dargestellt.

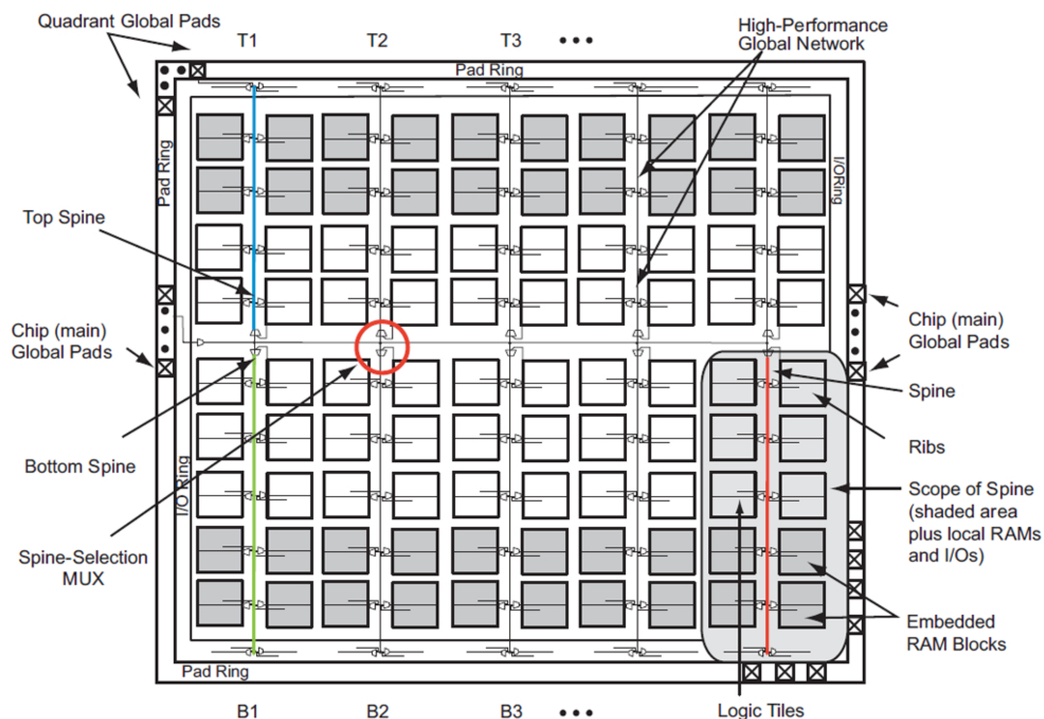

Abbildung 6.42: Struktur eines VersaNet-Netzwerkes [Act10b].

Clock Conditioning Circuits

Insgesamt ermöglichen sechs *Clock Conditioning Circuits (CCCs)* die Erzeugung der benötigten Taktsignale mit eigener Frequenz, Phasenverschiebung oder einer Zeitverzögerung. Deshalb besteht ein CCC aus einem *Phase-Locked Loop*, fünf konfigurierbaren *Multiplizierer/Dividierer*-Blöcken, drei *Phasenselektierer (phase selectors)*, einem *dynamischen Schieberegister (dynamic shift register)* und sechs konfigurierbaren *Verzögerungsglieder (delay blocks)*.
Die PLL ist in der Lage Taktsignale mit einer Frequenz zwischen 40 bis 250 MHz zu erzeugen. Der Entwickler kann somit bei jedem CCC aus mehreren Ausgangs-Taktsignalen wählen. Der

vereinfachte Aufbau eines CCCs, ohne Multiplizierer/Dividierer-Blöcke, ist aus Abb. 6.43 ersichtlich.

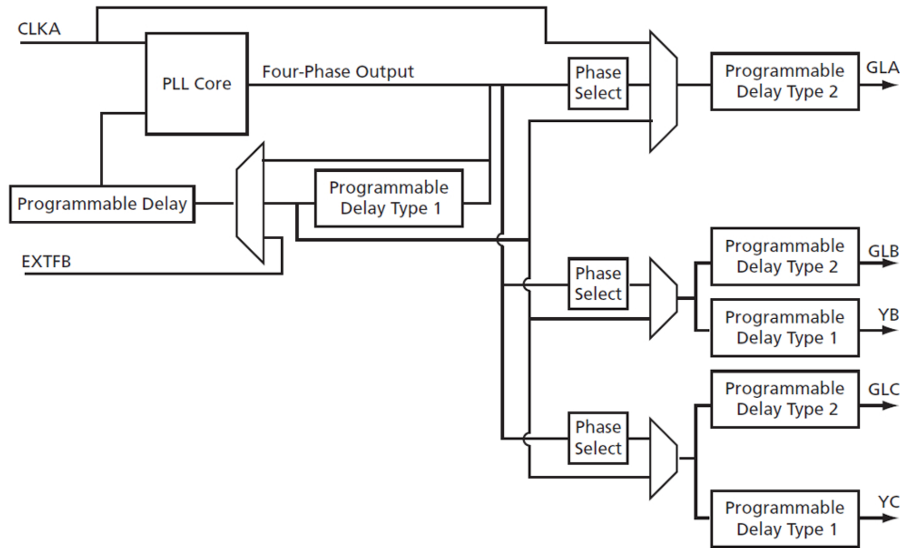

Abbildung 6.43: Struktur eines CCC-Blocks mit eingebauter PLL [Act10b].

Konfiguration

Die *Konfiguration* der Flash-Zellen wird ausschließlich über die JTAG-Schnittstelle des FPGAs durchgeführt. Zum Auslesen des FlashROM-Speichers können die JTAG-, das UJTAG-Interface oder die FPGA-Logik verwendet werden.
Die FPGA-Logik (core fabric) liest den FlashROM-Speicher durch direkte Adressierung, wobei dafür eine 7-Bit-Adresse verwendet wird. Die obersten drei Bits werden zur Auswahl der insgesamt acht Speicherseiten (pages) verwendet. Mit den restlichen vier Bits können die 16 Bytes jeder Speicherseite einzeln adressiert und gelesen werden. Die Komponenten zum Zugriff über die UJTAG-Schnittstelle und der FPGA-Logik sind aus Abb. 6.44 ersichtlich.

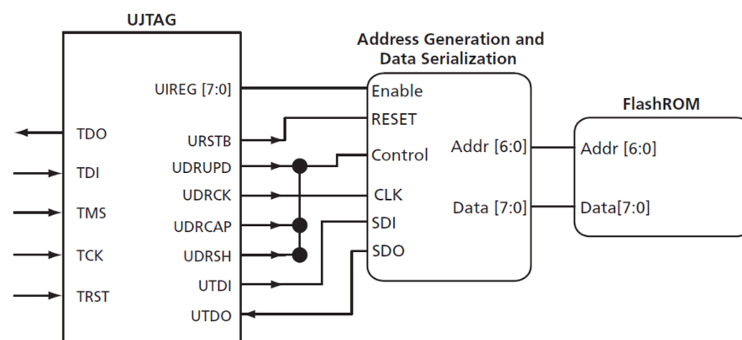

Abbildung 6.44: UJTAG-Interface und Addressumwandlung für Flash-ROM [Act10b].

Ein 128-Bit AES-Algorithmus ermöglicht die Entschlüsselung von verschlüsselt übertragenen

Konfigurationsdaten. Nach der Programmierung kann die Konfiguration der Flash-Zellen mit der *FlashLock*-Technologie vom Entwickler gesperrt werden. Dadurch wird die Veränderung oder das Auslesen der Konfiguration verhindert.

Rekonfiguration

Die ProASIC3-Familie unterstützt keine Rekonfiguration der FPGA-Logik mit z.B. Readback- oder Scrubbing-Verfahren. Es können nur die vorhandenen Flash-Zellen ausgelesen und neu programmiert werden.

6.2.5 Zusammenfassung

Die Architektur der Actel RTAX- und RTSX-FPGAs basiert gänzlich auf Antifuse-Konfigurationszellen. Diese reagieren praktisch unempfindlich auf die Strahlung, wodurch es nur sehr selten zu Ausfällen durch SEEs kommt. Beide FPGA-Familien beinhalten zudem spezielle gegen SEU geschützte Flip-Flops. Dank dieser Kombination sind die RTAX- und RTSX-FPGAs sehr gut für Raumfahrtanwendungen geeignet.
Die RTAX-Familie verfügt, im Vergleich zu einigen Xilinx-FPGAs, über einen zwar geringeren, aber für zahlreiche Anwendungen durchaus ausreichenden Funktionsumfang. Bei der RTSX-SU-Familie handelt es sich um vergleichsweise kleine FPGAs mit einem relativ geringen Funktionsumfang.

Im Gegensatz dazu sind die RT ProASIC3-FPGAs mit Flash-Konfigurationszellen ausgestattet, welche einen etwas geringeren Schutz vor SEUs bieten. Die Flip-Flops sind hier nicht speziell gegen SEUs geschützt. Die Architektur unterscheidet sich zudem relativ stark von den anderen FPGA-Familien.

Die wichtigsten Eigenschaften der einzelnen Actel-Familien nochmals im Überblick:

	Actel radiation-tolerant FPGAs			
	RTAX-S/SL	RTAX-DSP	RTSX-SU	RT ProASIC3
Zellentyp	Antifuse	Antifuse	Antifuse	Flash
SEU-enhanced FF’s	ja	ja	ja	nein
Block-RAM (kBit)	< 540	< 540	-	< 504
TID	300 krad	300 krad	100 krad	15 krad
SEL Immunität	>117	> 117	> 104	> 70
SEU in GEO	10E-10	10E-10	$<$ 10E-10	-
EDAC	ja	ja	nein	nein
Core Voltage	1.5 V	1.5 V	3.3 - 5 V	1.2 - 1.5 V
Readback	nein	nein	nein	nur FlashROM
DSP-Blöcke	nein	ja	nein	nein

Tabelle 6.3: Die wichtigsten Eigenschaften der radiation-tolerant Actel-FPGAs.

Weitere Eigenschaften können aus den entsprechenden Datenblättern und Benutzerhandbüchern entnommen werden.

6.3 Altera FPGA-Familien

Die Firma Altera ist mit einem Marktanteil von etwa 35 % der weltweit zweitgrößte Hersteller von programmierbaren Logikbausteinen. Im Wesentlichen handelt es sich dabei um verschiedene FPGAs, CPLDs und ASICs.
Im Gegensatz zu den bereits beschriebenen Herstellern, produziert Altera keine speziellen FPGA-Varianten für Raumfahrtanwendungen bzw. Umgebungen mit einer erhöhten Strahlung. Einige FPGAs der high-end Startix-Familie wurden jedoch für militärische Anwendungen entwickelt und verfügen über eine gewisse Strahlungstoleranz. Sie besitzen dementsprechende Komponenten für die Behebung von SEUs und zur Durchführung einer partiellen Rekonfiguration.

In den nachfolgenden Abschnitten werden zur kurzen Übersicht die, für Raumfahrtsysteme relevanten, Stratix-III- und Stratix-IV-Familien genauer beschrieben.

6.3.1 Die Stratix-III-Familie

Bei der *Stratix-III-Familie* handelt es sich um high-performance FPGAs mit SRAM-Konfigurationszellen. Ihre SOM-Architektur basiert auf einem 65 nm Kupfer-CMOS-Prozess und besteht aus mehreren Logik-Blöcken, verschiedenen RAM-Speichern, DSP-Blöcken mit Multiplizierern, Phase-Locked Loops und anderen Komponenten.

Die wichtigsten Eigenschaften der Stratix-III-Familie sind [Alt10a]:

- 1.1 V Versorgungsspannung
- Latch-up Immunität
- SEU in GEO (SEFI) nicht spezifiziert (< 1 SEFI pro 100 Jahre)
- 65 nm Kupfer-CMOS-Prozess
- SRAM-basierte In-System-Konfiguration
  - Optionale 256-Bit AES Verschlüsselung
  - Parallel, Seriell und JTAG/Boundary-Scan
- Automatisierte CRC-Prüfung der Konfiguration
- Max. 135.000 Adaptive Logic Modules
- Max. 12 Phase-Locked Loops
- Max. 112 DSP-Blöcke mit Multiplizierern
- Max. 1.104 I/O-Anschlüsse
- Max. 20.497-kBit TriMatrix-Speicherblöcke
  - 9-kBit dual-port RAM- und FIFO-Speicher
  - 144-kBit Speicherblöcke mit Error Correction Coding
- Dynamic On-Chip Termination
- High-Speed Datentransfer mit Serializer/Deserializer bis 1.6 Gbit/s
- IEEE 1149.1 - JTAG/Boundary-Scan Support

In den nachfolgenden Abschnitten wird ein Überblick, über die grundlegende Architektur und die verschiedenen Module, gegeben.

Grundlegende Architektur

Die wichtigsten Komponenten der Stratix-III-Architektur sind die *Logic Array Blocks (LABs)* und die *Memory Logic Array Blocks (MLABs)*. Mit ihnen kann die gewünschte kombinatorische und sequentielle Logik implementiert werden.

Ein MLAB verfügt über den selben Funktionsumfang wie ein LAB, enthält aber einen zusätzlichen 320-Bit großen SRAM-Speicherblock. Die MLAB und LAB sind immer paarweise angeordnet und somit gleich oft im FPGA vorhanden.

Jeder LAB oder MLAB besteht aus insgesamt zehn *Adaptive Logic Modules (ALMs)*, welche über die *lokalen Verbindungsstrukturen (lokal interconnect)* miteinander direkt verbunden sind. Spezielle *horizontale und vertikale Verbindungen (interconnections)* werden für den Datenaustausch zwischen den LAB/MLAB verwendet. Mehrere Leitungen für Übertrag-Bits (carry chains), arithmetische Logikfunktionen (arithmatic chains) und Register (register chains) ermöglichen die Verknüpfung der verschiedenen LABs zu größeren Komponenten.
Die Architektur eines Stratix-III-FPGAs, mit den LABs, MLABs und den verschiedenen Verbindungsstrukturen, ist aus Abb. 6.45 ersichtlich.

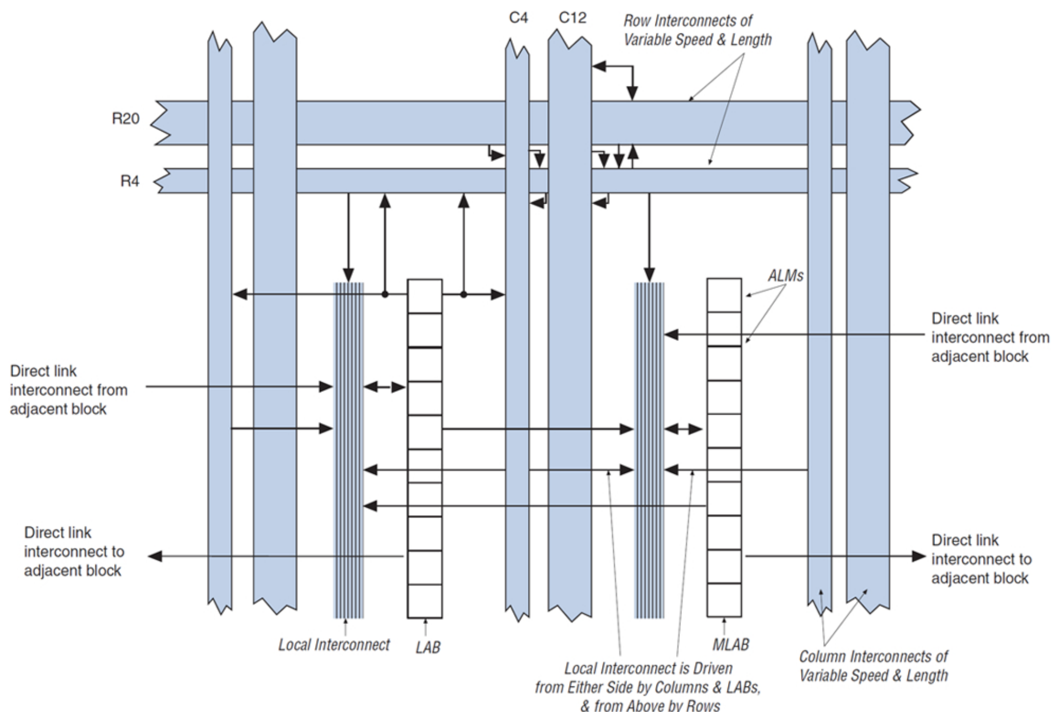

Abbildung 6.45: Architektur des Stratix-III [Alt10a].

Adaptive Logic Module

Jedes *Adaptive Logic Module* besteht im wesentlichen aus zwei *Adaptive LUTs (ALUTs)* und zwei *Registern (registers)*. Die ALUTs bestehen wiederum jeweils aus einer *6-input LUT* und einem *Volladdierer (full adder)*.

Die beiden ALUTs besitzen insgesamt acht Eingänge und können zur Implementierung verschiedener Funktionen verwendet werden. Dies können beispielsweise arithmetische Funktionen, LUTs oder Schieberegister sein.
Zu diesem Zweck kann jeder ALM in einem *Normal-*, *Extended LUT-*, *Arithmetic-*, *Shared Arithmetic-* oder *LUT-Register*-Modus betrieben werden. Die Wahl des Modus erfolgt über spezielle Steuerleitungen. Im Normal- und Extended LUT-Modus stehen beispielsweise kleine

3-input LUTs, bis hin zu großen 7-input LUTs, zur Verfügung. Der Aufbau und die einzelnen Komponenten eines ALMs sind in Abb. 6.46 dargestellt.

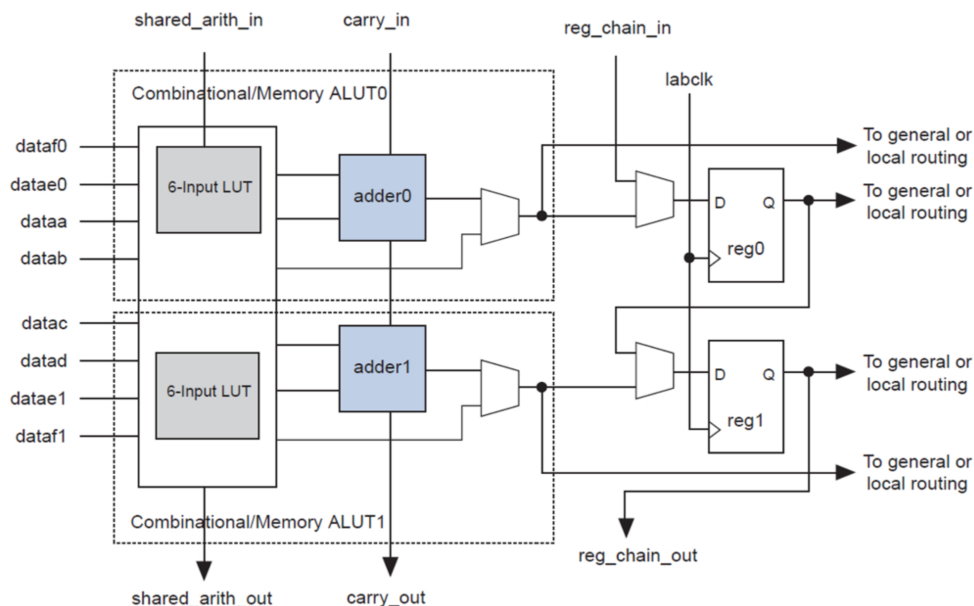

Abbildung 6.46: Struktur eines ALM-Blockes [Alt10a].

TriMatrix Speicher-Block

Die Stratix-III-FPGAs besitzt drei unterschiedliche *TriMatrix RAM-Speicherblöcke*, welche sich in ihrer Größe und Vielseitigkeit voneinander unterscheiden. Die MLABs beinhalten einen 320-Bit großen Speicher, welcher laut Altera bevorzugt als FIFO oder Schieberegister eingesetzt werden soll. Die 9-kBit großen M9K-Speicher sind hingegen für normale Anwendungen vorgesehen. Zusätzlich stehen 144-kBit große M144K-Blöcke zur Verfügung, welche z.B. für die Zwischenspeicherung großer Datenmengen verwendet werden können.
Im Unterschied zu den anderen Speichern, besitzen die M144K-Blöcke einen speziellen *Build-in Error Correction Coding (ECC)*-Algorithmus. Dieser wird zur Fehlerdetektion und -korrektur eingesetzt.
Unabhängig davon können alle Speicherblöcke wahlweise als single-port oder dual-port RAM, ROM oder Schieberegister konfiguriert werden. Ebenfalls möglich ist die Verknüpfung der Blöcke zu größeren Speichern.

Taktsignale und PLL

Zur Erzeugung der verschiedenen Taktsignale werden in den FPGAs drei bis zwölf PLLs mit jeweils bis zu zehn Ausgängen verwendet, wobei alle Signale vom Entwickler einzeln konfiguriert werden können. Die PLL-Blöcke sind an den Außenkanten des FPGAs angebracht.
Zusätzlich sind für die Erzeugung und Steuerung der Taktsignale einige Multiplizierer, Dividierer, Phasenschieber und Oszillatoren im FPGA vorhanden.

Bei der Stratix-III-Familie werden, zur Verteilung der Taktsignale, maximal 16 *Global Clock Networks (GCLKs)*, 88 *Regional Clock Networks (RCLKs)* und zusätzlich 116 *Periphery Clock*

Networks (PCLKs) eingesetzt. Die tatsächliche Anzahl hängt dabei vom verwendeten FPGA ab. Während die Taktsignale der GCLKs im gesamten FPGA zu Verfügung stehen, sind die Signale der RCLKs und PCLKs ausschließlich auf die verschiedenen Quadranten beschränkt.

DSP-Block

Jeder *DSP-Block* ist aus zwei identischen *Half-DSP-Blöcken* aufgebaut. Diese bestehen aus einigen Registern, vier Multiplizierern, mehreren Addierern/Subtrahierern, einem Schieberegister und einem Multiplexer. Damit lassen sich prinzipiell verschiedene 9-, 12-, 18- und 36-Bit Multiplizierer erzeugen, welche optional mit einem Addierer/Subtrahierer oder Akkumulator zu einer gewünschten Funktion kombiniert werden können. Der Aufbau eines Half-DSP-Blocks ist in Abb. 6.47 dargestellt.

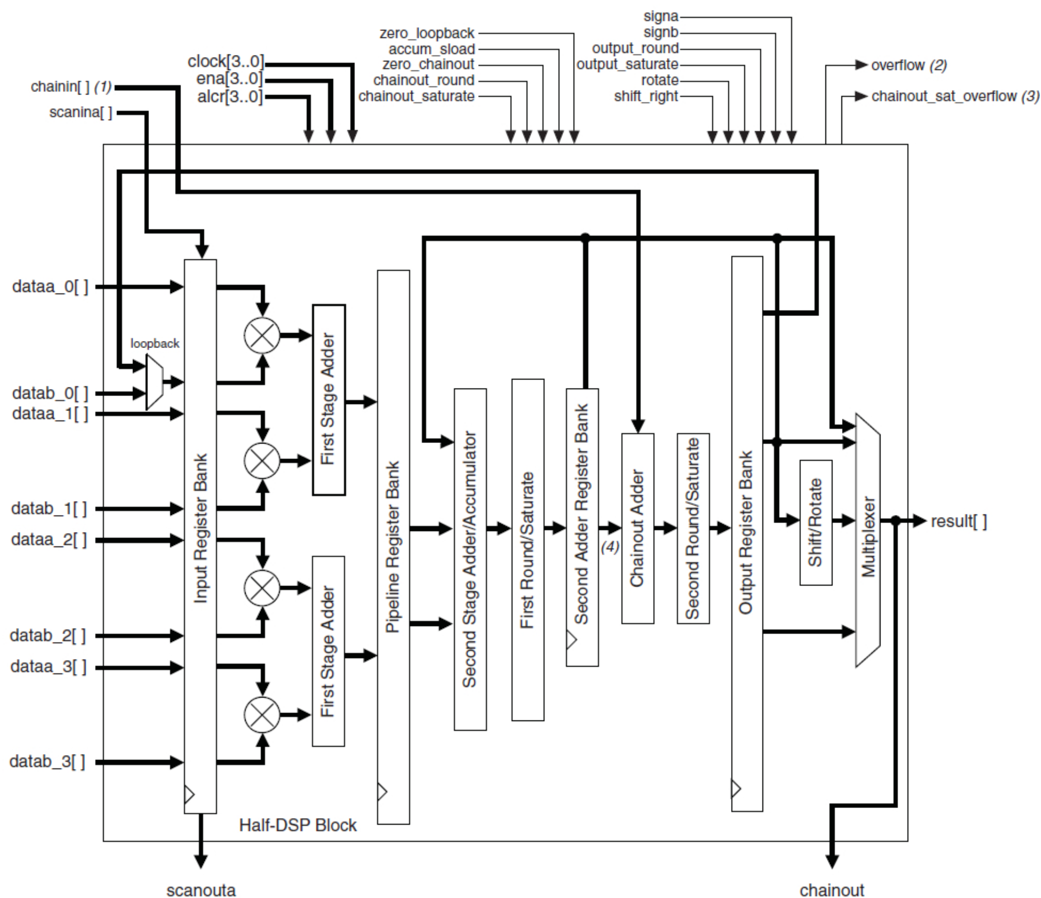

Abbildung 6.47: Aufbau eines Half-DSP-Blocks [Alt10a].

Mit Hilfe der DSP-Blöcke lassen sich zudem zahlreiche komplexere Komponenten realisieren. Dies sind beispielsweise FFT/IFFT-Blöcke und FIR/IIR-Filter.

I/O-Block

Die relativ komplexen *I/O-Blöcke* der Stratix-III-Familie bieten zahlreiche Möglichkeiten. Verschiedenste symmetrische, unsymmetrische und spannungsreferenzierte *I/O-Standards* vereinfachen die Kommunikation mit anderen Hardwarekomponenten. Zur Vermeidung von Reflexionen

wird die *On-Chip Termination (OCT)* verwendet. Zudem können verschiedene Parameter wie z.B. die Slew-Rate, I/O-Verzögerung (delay) und der Ausgangsstrom konfiguriert werden.

High-Speed differential Transceiver

Jeder FPGA beinhaltet mehrere *Transceiver*-Blöcke für die source-synchrone differentielle high-speed Datenübertragung. Diese enthalten, neben einem SERDES-Block, mehrere *Dynamic Phase Alignment (DPA)*-Blöcke und *Clock Data Recovery (CDR)*-Blöcke.
Im DPA-Modus werden zuerst die seriellen Daten aus dem Eingangsbuffer gelesen. Danach bestimmt der DPA, aus acht phasenverschobenen Taktsignalen, das am Besten für die Abtastung der Eingangsdaten geeignete Signal. Der sogenannte Soft-CDR-Modus ermöglicht zudem die Rückgewinnung des Taktsignals aus den empfangenen seriellen Daten.

Konfiguration

Auf Grund der SRAM-Zellen muss die Konfiguration bei jedem Einschaltvorgang neu in den FPGA geladen werden. Zur Speicherung der Daten wird ein statischer RAM, der sogenannte *Configuration RAM (CRAM)*, verwendet. Für die Übertragung der Konfigurationsdaten stehen die *Fast Passive Parallel (FPP)*-, *Fast Active Serial (AS)*-, *Passive Serial (PS)*-Methoden und die *JTAG/Boundary-Scan*-Schnittstelle zur Verfügung.

Die JTAG-Schnittstelle kann zusätzlich zum Testen und Debuggen des FPGAs verwendet werden. Sie stellt deshalb die einfachste und flexibelste Lösung dar.
Bei der Fast AS-Methode liest der FPGA die benötigten Konfigurationsdaten über eine serielle Schnittstelle aus einem externen Speicherbaustein. Schreibt hingegen eine externe Komponente die seriellen Daten in den FPGA, dann handelt es sich um die PS-Methode. Bei der FPP-Methode wird eine parallele Schnittstelle verwendet (siehe Abb. 6.48).

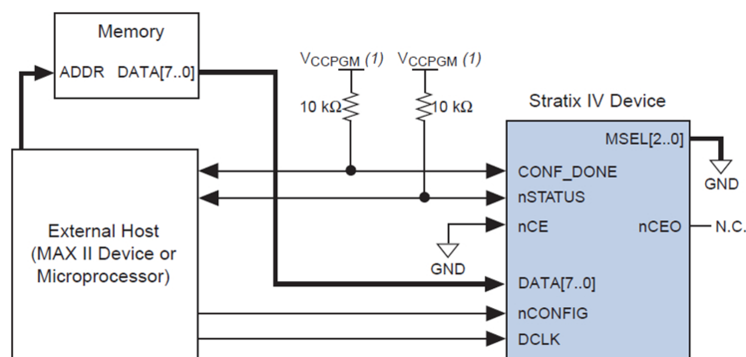

Abbildung 6.48: Konfiguration mittels FPP-Methode [Alt10b].

Damit die Konfigurationsdaten bei der Übertragung nicht gelesen werden können, werden diese mit einem 256-Bit AES-Algorithmus verschlüsselt. Die Stratix-III-FPGAs enthalten dementsprechend einen *AES-Decryption*-Block zur Entschlüsselung der Daten. Zur Vermeidung von Übertragungsfehlern wird zusätzlich die CRC-Summe der Konfigurationsdaten überprüft.

Zur Überprüfung der bestehenden Konfiguration kann ebenfalls der CRC-Algorithmus eingesetzt werden. Dieser Berechnet die CRC-Summe aus den Daten des CRAMs und vergleicht diese mit dem Originalwert. Dadurch kann ein SEU rasch und einfach erkannt und dessen Position im

CRAM ermittelt werden. Die Behebung der Störung muss allerdings von außen, d.h. mit einem zweiten integrierten Baustein, durchgeführt werden.

6.3.2 Die Stratix-IV-Familie

Bei der *Stratix-IV-Familie* handelt es sich um high-performance FPGAs mit SRAM-Konfigurationszellen. Ihre SOM-Architektur basiert auf einem 65 nm Kupfer-CMOS-Prozess und besteht aus mehreren Logik-Blöcken, verschiedenen RAM-Speichern, DSP-Blöcken mit Multiplizierern und Phase-Locked Loops.

Die wichtigsten Eigenschaften der Stratix-IV-Familie sind [Alt10b]:

- Latch-up Immunität
- SEU in GEO (SEFI) nicht spezifiziert ($<$ 1 SEFI pro 100 Jahre)
- 40 nm CMOS-Prozess
- SRAM-basierte In-System-Konfiguration
- Automatische CRC-Prüfung der Konfiguration
- Programmierbare TX pre-emphasis und RX equilization
- Digital Signal Processing
  - 9-Bit, 12-Bit, 18-Bit und 36-Bit Multiplizierer
  - Schnelle Pipeline-Architektur
- High-Speed Differential I/O mit DPA und Soft-CDR
  - CDR-basierte Transceivers mit 8.5 bis 11.3 Gbit/s
- Max. 325.220 Adaptive Logic Modules
- Max. 112 DSP-Blöcke mit Multiplizierern
- Max. 1.120 I/O-Anschlüsse
- Max. 33.294-kBit Enhanced TriMatrix-Speicherblöcke
  - 9-kBit M9K, 144-kBIT M144K und 640-Bit MLAB
- Dynamic On-Chip Termination
- IEEE 1149.1 - JTAG/Boundary-Scan Support

Grundlegende Architektur

Die Architektur der Stratix-IV-Familie ähnelt sehr stark dem Stratix-III-Vorgängermodell. Ein Stratix-IV-FPGA besteht dementsprechend aus den selben Komponenten (siehe Abb. 6.49).

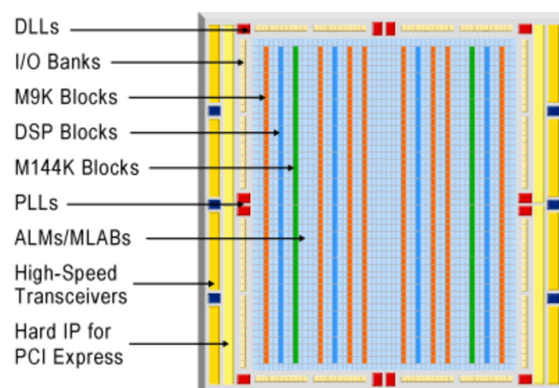

Abbildung 6.49: Architektur der Stratix-IV GX- und GT-FPGAs [Str10].

Die einzelnen Komponenten wurden jedoch teilweise vergrößert und im Funktionsumfang erweitert. Wie bei der Stratix-III-Familie sind auch hier mehrere LABs als Grundbausteine vorhanden. Diese bestehen aus zehn ALMs und können für die Implementierung der gewünschten Logik verwendet werden. Die FPGAs sind als E-, GX- und GT-Varianten verfügbar, welche sich teilweise im Funktionsumfang voneinander unterscheiden.
Als Hauptunterschied zum Vorgängermodell, besitzt die GX- und GT-Bausteine der Stratix-IV-Familie mehrere high-speed Transceiver-Blöcke. Diese ermöglichen die Verarbeitung und Übertragung serieller Daten mit einer Datenrate von maximal 11.3 Gbit/s.

Adaptive Logic Module

Die Adaptive Logic Modules der Stratix-IV-Familie wurden praktisch unverändert von der Stratix-III-Familie übernommen. Jeder ALM besteht im Wesentlichen aus verschiedenen konfigurierbaren LUTs, zwei Register und zwei Volladdierern.

TriMatrix Speicher-Block

Der *TriMatrix-Speicher* besteht bei der Stratix-IV-Familie aus 9-kBit großen M9K- und 144-kBit großen M144K-Speicherblöcken. Zusätzlich beinhalten die MLABs einen 640-Bit großen Speicher.
Jeder Speicherblock kann als RAM, FIFO, Schieberegister oder ROM konfiguriert werden. Die M144K-Blöcke verfügen zudem über einen ECC-Algorithmus. Dieser ermöglicht die Korrektur eines einzelnen Bitfehlers in einem 64-Bit Datenwort. Zusätzlich können zwei Bitfehler in jedem Datenwort detektiert werden.

DSP-Block

Die *DSP-Blöcke* der Stratix-IV-Familie wurden nahezu unverändert von der Stratix-III-Familie übernommen. Somit sind sämtliche Schnittstellen der DSP-Blöcke mit dem Vorgängermodell identisch. Die enthaltenen Pipeline-Architektur ermöglicht die schnelle Multiplikation von 9-Bit, 12-Bit, 18-Bit und 36-Bit Datenwörtern. Verschiedene Addierer-, Subtrahierer-, Schieberegister- und Akkumulator-Strukturen erweitern zusätzlich den Funktionsumfang des DSP-Blockes. Wie bei anderen FPGAs, können damit beispielsweise verschiedene IIR- oder FIR-Filter realisiert werden.

High-speed serial and differential I/O

Neben den normalen Standard-I/Os, können bei der Stratix-IV-Familie auch verschiedene differentielle Ein- und Ausgänge verwendet werden. Diese werden im FPGA in sogenannte *Row- und Column-I/Os* unterteilt.

Jeder FPGA besitzt, zur seriellen high-speed Datenübertragung, mehrere *SERDES-Blöcke*. Diese sind mit *Low Voltage Differential Signaling (LVDS)-Schnittstellen* verbunden und erlauben Datenraten von maximal 1.6 GBit/s. Mit den SERDES-Blöcken können zahlreiche synchrone und asynchrone Kommunikationsstandards zur Datenübertragung eingesetzt werden.

Die GT- und GX-Bausteine der Stratix-IV-Familie verfügen zusätzlich über bis zu 48 high-speed *Clock Data Recovery (CDR)*- und mehreren *Dynamic Phase Alignment (DPA)*-Blöcken. Diese speziellen *LVDS SERDES-Transceivers* sind mit den Row-I/Os verbunden und erreichen Datenraten von 11.3 GBit/s (GT) bzw. 8.5 Gbit/s (GX).

Der Aufbau eines solchen LVDS SERDES-Transceivers, mit Sender (LVDS Transmitter) und Empfänger (LVDS Receiver), ist aus Abb. 6.50 ersichtlich.

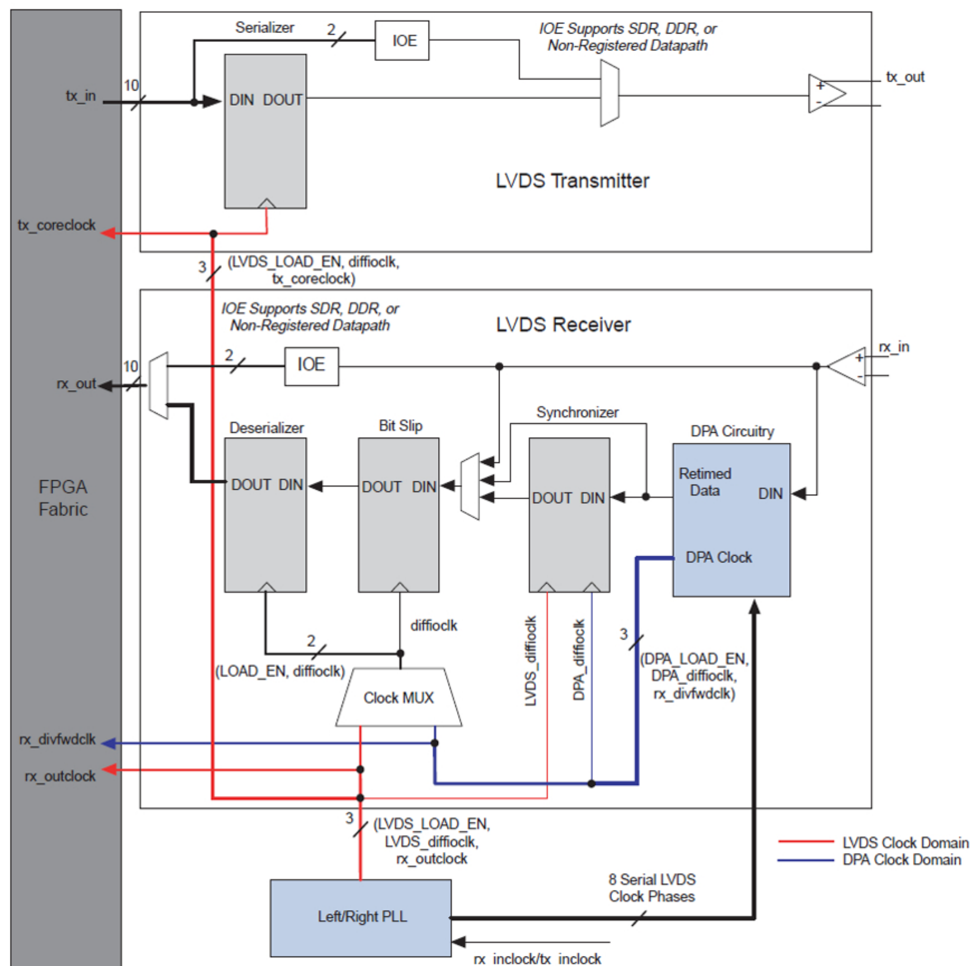

Abbildung 6.50: SERDES-Block eines Startix-IV-FPGAs [Alt10b].

Der LVDS Receiver-Block unterstützt drei verschiedene Betriebsarten, den *Non-DPA Mode*, *DPA Mode* und den *Soft-CDR Mode*.
Bei der Datenübertragung wird durch das physikalische Medium und den Jitter bzw. Skew der verschiedenen Taktsignale im FPGA häufig eine Phasenverschiebung zwischen dem Datensignal und dem Taktsignal des Empfänger-Blocks erzeugt. Mit Hilfe der verschiedenen Betriebsarten können die Phasenverschiebungen zwischen den Daten und dem source-synchronen Taktsignal (Non-DPA und DPA) bzw. dem Referenz-Taktsignal (Soft-CDR) kompensiert werden.
Mit der *Non-DPA Mode* kann eine günstige Einstellung manuell vorgegeben werden. Die *DPA Mode* und *Soft-CDR Mode* wählt automatisch die beste Einstellung.

Konfiguration

Für die Konfiguration können mehrere Verfahren verwendet werden. Wie bei anderen Stratix-FPGAs stehen dafür die *Fast Passive Parallel (FPP)*-, *Fast Active Serial (AS)*-, *Passive Serial (PS)*-Methoden und die *JTAG/Boundary-Scan*-Schnittstelle zur Verfügung.

6.4 Zusammenfassung und Empfehlungen

In diesem Kapitel wurden einige nennenswerte strahlungstolerante FPGA-Familien vorgestellt. Es handelt sich hierbei natürlich um keine vollständige Übersicht aller Hersteller und FPGAs. Die genannten Hersteller und deren FPGAs wurden auf Grund ihres Marktanteiles und der gegebenen Relevanz für Luft- und Raumfahrtsysteme ausgewählt. Im Fall eines konkreten Projektes wird man sich dementsprechend nochmals mit den verschiedenen kleineren und oft auf eine Marktnische spezialisierten Herstellern beschäftigen müssen, um den für die Anwendung am Besten geeigneten FPGA zu finden.

Wie aus den vorangegangenen Abschnitten bereits hervorgeht, können FPGAs mit Antifuse-Zellen nur einmal programmiert werden. Sie sind allerdings gegenüber den Strahlungseinflüssen praktisch unempfindlich. Im Gegensatz dazu sind die SRAM-Zellen mehrmals programmierbar, aber sehr anfällig für strahlungsbedingte Störungen. Diese benötigen zudem einen externen Speicherbaustein für die Programmierung.

Aus diesem Grund sind, von den in den vorigen Abschnitten aufgezählten FPGA-Familien, in den meisten Fällen die FPGAs von der Firma Actel für Raumfahrtanwendungen zu empfehlen. Diese sind wegen ihrer Architektur relativ gut vor SEEs geschützt, da sie mit speziellen SEU-hardened Registern und Flip-Flops ausgestattet sind. Wünscht man allerdings einen tendentiell neueren Fertigungsprozess, einen teils höheren Integrationsgrad oder einen FPGA mit mehreren speziellen Komponenten wie z.B. großen DSP-Blöcken, dann sind die Xilinx Virtex-FPGAs eine durchaus vertretbare Wahl. Diese basieren zwar auf SRAM-Zellen, unterstützen aber das Readback- und Scrubbing-Verfahren zur Behebung und Detektion von SEEs. Die Entwicklung eines TMR-Systems kann zudem mit der Hersteller-Software automatisiert durchgeführt werden.

Laut den bei [Spa01] und [Spa02] veröffentlichten Zahlen, sind zwischen den Jahren 1990 und 2001 zahlreiche Störungen und Komplettausfälle von Satelliten und Raketen aufgetreten. Der vorwiegende Grund dafür waren Fehler der *Antriebe (Propulsion)*, der *Guidance, Navigation and Control (GN&C)-Komponenten* und dem *Electrical Power and Distribution System (EPDS)*. Leider sind in den mir zugänglichen Quellen nur sehr spärliche Informationen über den tatsächlichen Grund für die Ausfälle enthalten. Es kann dementsprechend nur schwer festgestellt werden, ob der Ausfall oder Fehler im Detail durch einen FPGA oder andere Hardwarekomponenten verursacht wurde.
Tendentiell werden in den letzten Jahren allerdings immer häufiger komplexe elektronische Systeme mit integrierten Schaltungen in Raumfahrtsystemen eingesetzt. Dementsprechend übernehmen ASICs immer wichtigere Funktionen in den verschiedenen Raumfahrtsystemen. Bei der Entwicklung muss deshalb besonders auf eine hohe Qualität und Ausfallssicherheit der elektronischen Komponenten geachtet werden.

Zusätzlich spielen, bei der Wahl der FPGA-Familie, die Kosten für die Entwicklungssoftware und die damit verbundenen Lizenzen der Hersteller eine Rolle. Diese kosten, je nach gewünschtem Funktionsumfang, bis zu mehrere tausend Euro. Die FPGAs unterscheiden sich teilweise ebenfalls stark im Preis.

7 Programmiersprachen zur Entwicklung von FPGAs

Auf Grund der unterschiedlichen Anforderungen bei der Entwicklung von FPGAs, haben sich in den letzten Jahrzehnten verschiedene Programmiersprachen etabliert. In Kapitel 4 wurde bereits auf die verschiedenen Abstraktionsebenen und die Entwurfsphasen bei der Entwicklung von FPGAs eingegangen. Die Programmiersprachen decken in der Regel mehrere unterschiedliche Abstraktionsebenen ab. Beispielsweise kann eine ausführbare Software, für die eingebetteten Prozessorkerne des FPGAs, mit verschiedenen *Software-Programmiersprachen* erstellt werden. Zur Entwicklung der gewünschten Hardware werden hingegen spezielle *Hardware-Programmiersprachen* eingesetzt. In der Praxis kommt eine Programmiersprache allerdings vorwiegend auf derjenigen Abstraktionsebene zum Einsatz, für welche sie die größten Vorteile besitzt.
Die nachfolgenden Abschnitte sollen einen kurzen Überblick über die wichtigsten Programmiersprachen für FPGAs geben und eine Einsicht in deren Anwendungsgebiete vermitteln. Auf die Programmiersprache VHDL wird in einem Abschnitt, zur Verdeutlichung der Konzepte einer Hardware-Programmiersprache, etwas näher eingegangen.
Weitere Informationen und Abbildungen zu diesem Kapitel können aus [Gro08] und [Ash08] entnommen werden.

7.1 Grundlegende Einteilung

Grundsätzlich lassen sich nahezu alle relevanten Programmiersprachen in sogenannte *Software Programming Languages (SPLs)* und *Hardware Description Languages (HDLs)* unterteilen.
Es soll allerdings erwähnt werden, dass die Unterteilung der Programmiersprachen von der gewählten Betrachtungsweise abhängt. Dies zeigt sich beispielsweise dadurch, dass die Software-Programmiersprachen *C* und *C++* in der Praxis auch für die Programmierung von Mikrocontrollern oder Signalprozessoren verwendet werden. Man könnte also fälschlicherweise annehmen, dass diese ebenfalls zur Gruppe der HDLs gehören. Bei genauer Betrachtung wird man allerdings feststellen, dass damit lediglich die Funktionsweise der bereits vorhandenen Hardware-Komponenten festgelegt wird. Mit einer SPL kann also keine Hardware entworfen oder erstellt werden.

Bei FPGAs kommen SPLs vorwiegend für die Entwicklung einer eingebetteten Software zum Einsatz. Diese Programme können von EPCs, wie z.B. dem PPC405, ausgeführt werden. *C* und *C++* werden in der Praxis am häufigsten verwendet. Dies begründet sich unter anderem darauf, dass beide Programmiersprachen zahlreiche indirekte oder direkte Eingriffe in den Speicher erlauben. Bei der Kompilierung wird der Quellcode, für einen bestimmten Prozessorkern, in den ausführbaren Maschinencode umgewandelt. Somit ist diese Software nur auf dem ausgewählten EPC lauffähig.

Andere SPLs, wie z.B. *Java*, benötigen hingegen eine vorgefertigte Basissoftware mit einem Interpreter. Dies könnte z.B. das Betriebssystem eines EPCs sein. Der wesentliche Vorteil dieser Programme liegt in ihrer Plattformunabhängigkeit, da sie nur indirekt über die Basissoftware mit der Hardware kommunizieren.

Einige der wichtigsten Programmiersprachen und deren generelle Unterteilung sind aus Abb. 7.1 ersichtlich.

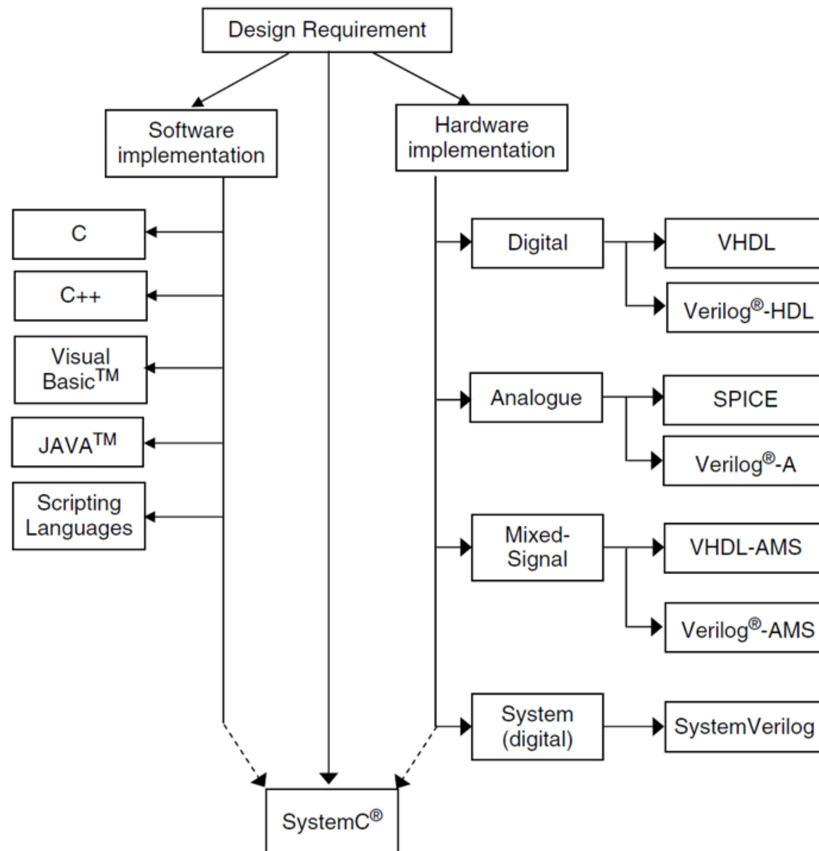

Abbildung 7.1: Einteilung verschiedener Programmiersprachen [Gro08].

Für den Entwurf und die Simulation von digitalen und analogen Hardwaresystemen werden verschiedene HDLs eingesetzt. In den letzten Jahren kommen bei der Entwicklung digitaler Schaltungen vorwiegend *VHDL* und *Verilog-HDL* zum Einsatz. Für analoge Schaltungen wird häufig *SPICE* und *Verilog-A* verwendet. Im Gegensatz zu den SPLs, werden dabei die Schaltungen für die gewünschte Hardware entworfen und implementiert. Die verschiedenen HDLs ermöglichen und vereinfachen zusätzlich die Simulation und Analyse der Schaltungen.

Bei Raumfahrtanwendungen kommen häufig sehr komplexe Systeme zum Einsatz, welche aus Hardware- und Softwarekomponenten bestehen. Hier stellt sich die Frage, wie die einzelnen Teilsysteme zu einem Gesamtsystem zusammengefasst werden können. Auf Grund der damit verbundenen Problemstellungen wurde der sogenannte *SystemC*-Standard eingeführt. Dabei handelt es sich um eine C++ Klassen-Bibliothek, welche speziell für den Entwurf von kombinierten Hardware- und Softwaresystemen entwickelt wurde. Das Hauptanwendungsgebiet liegt dabei auf der Simulation und der Verifikation solcher komplexer Systeme.

Der nachfolgende Abschnitt enthält eine kurze Einführung in die VHDL-Programmiersprache, da diese für digitale Systeme sehr häufig zum Einsatz kommt. Die wichtigsten Merkmale von VHDL werden dabei, anhand einiger einfacher Beispiele, verdeutlicht.

7.2 Very High-Speed Integrated Circuit Hardware Description Language

Die Anforderungen für die VHDL-Programmiersprache wurden erstmals nach 1980 vom US Department of Defense definiert und einige Jahre später als IEEE-Standard 1076 übernommen. Ziel war die Entwicklung einer neuen Designmethode für digitale Schaltungen. In den letzten Jahren wurde die VHDL-Spezifikation mehrmals an neue Anforderungen angepasst und im ursprünglichen Funktionsumfang erheblich erweitert.

Bei allen *Hardware Description Languages* handelt es sich im Wesentlichen um eine textbasierte Programmiersprache, zum Entwurf von Schaltungen oder Systemen. Die Programmierung kann dabei auf mehreren verschiedenen Abstraktionsebenen erfolgen. Neben der textbasierten Eingabe eines Quellcodes, werden auf den obersten Abstraktionsebenen häufig grafische Beschreibungen, wie Blockdiagramme oder Schaltpläne (schematics), verwendet.
Der besondere Vorteil der HDLs besteht darin, dass die Schaltungen durch *Synthese* von einer höheren Abstraktionsebene auf eine tiefere transformiert werden können. Dadurch kann der Entwickler das System auf einer für ihn überschaubaren und verständlichen Ebene entwerfen. Anschließend kann die Schaltung in die tatsächlich vorhandenen Hardwarekomponenten des FPGAs, wie z.B. Gatter oder Transistoren, umgewandelt werden.
In der Regel werden bei der Synthese spezielle *Netzlisten* erzeugt. Diese enthalten eine textbasierte Beschreibung der benötigten Komponenten und deren gegenseitige Verknüpfung. Ein weiterer Vorteil der Synthese liegt darin, dass die Hardwareentwicklung erheblich beschleunigt und vereinfacht wird. Neben der Erleichterung beim Entwurf, ermöglicht dies gleichzeitig die automatisierte Simulation und Optimierung der Schaltungen. Eine prinzipielle Darstellung der üblichen Syntheseschritte ist aus Abb. 7.2 ersichtlich.

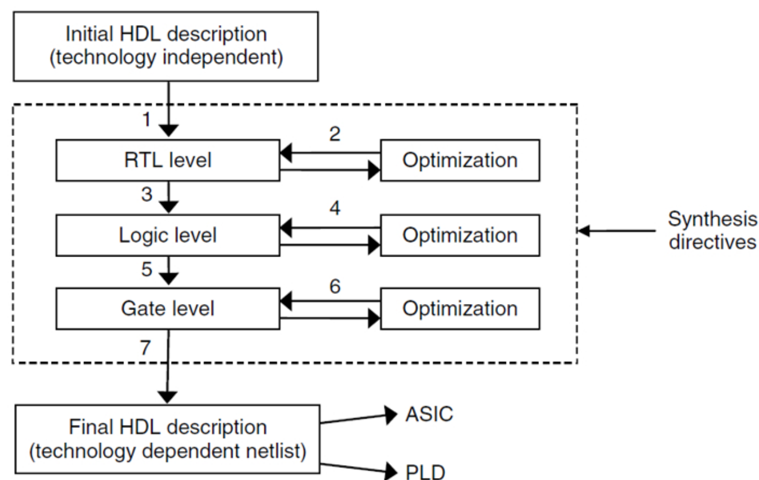

Abbildung 7.2: Verschiedene Schritte bei der Synthese [Gro08].

Weitere Informationen, zum Entwurf von FPGAs, können aus Kapitel 4 entnommen werden.

7.2.1 Grundlagen für den Entwurf mit VHDL

Laut [Gro08] kann eine digitale Schaltung mit einem VHDL-Code, grundsätzlich in Form einer *Struktur (structural)*-, *Datenfluss (dataflow)*- und *Verhaltens (behavioral)*-Beschreibung, im-

plementiert werden. Mit der Struktur-Beschreibung wird die Schaltungsstruktur, d.h. die Verdrahtung der verschiedenen Logikkomponenten, festgelegt. Eine Datenfluss-Beschreibung definiert hingegen die Verknüpfung der Ein- und Ausgänge einer Schaltung. Mit der Verhaltens-Beschreibung wird die Funktionsweise einer Logikkomponente, mit Hilfe verschiedener Algorithmen, implementiert.

Zur Realisierung der Schaltungen werden spezielle *Entity*-, *Architecture*- und *Process*-Deklarationen verwendet. Der Begriff Entity bedeutet übersetzt soviel wie Funktionseinheit oder Instanz. Die Entity-Deklaration ist somit die Grundkomponente jeder VHDL-Datei. Sie spezifiziert eine eigenständige Schaltung oder Funktionseinheit und enthält unter anderem die Definition der verwendeten Ein- und Ausgänge. Die Architecture-Deklaration wird anschließend, für die Implementierung der eigentlichen Funktionsweise des Entity, eingesetzt. Sie enthält dementsprechend eine der drei oben genannten Beschreibungen. Aus Lst. 7.1 ist der Zusammenhang zwischen Entity und Architecture, anhand eines 4-Bit Subtrahierers, dargestellt.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;

ENTITY FourBitSub IS
    PORT( IN1 : IN   STD_LOGIC_VECTOR(3 downto 0);
          IN2 : IN   STD_LOGIC_VECTOR(3 downto 0);
          SUB : OUT  STD_LOGIC_VECTOR(3 downto 0)
        );
END ENTITY FourBitSub;

ARCHITECTURE SubFunctionality OF FourBitSub IS
    -- Signal declaration, Constant declaration, Components
BEGIN
    -- dataflow description
    SUB <= IN1(3 downto 0) - IN2(3 downto 0);

END ARCHITECTURE SubFunctionality;
```

Listing 7.1: 4-Bit Subtrahierer mit Entity- und Architecture-Deklaration

Vor der eigentlichen Entity-Deklaration werden die benötigten Bibliotheken angegeben. Das dargestellte *FourBitSub*-Entity enthält die Definition der beiden 4-Bit Eingänge *IN1*, *IN2* und des 4-Bit Ausganges *SUB*. Anschließend folgt in der Architektur-Deklaration die Implementierung der Funktionsweise. Hier wird *IN2* von *IN1* abgezogen, wobei ein Unterlauf nicht explizit signalisiert wird.
Für eine Verhaltensbeschreibung verwendet man in der Architecture-Deklaration zusätzlich ein oder mehrere Process-Deklarationen. Laut VHDL-Spezifikation werden alle Prozesse parallel nebeneinander ausgeführt.
Jedes Entity realisiert somit eine eigenständige Logikkomponente mit Ein- und Ausgängen und einer vorgegebenen Funktionsweise. Die verschiedenen Entities können wiederum in anderen Entity-Deklarationen, als eigene Logikkomponenten, verwendet werden. Dafür steht eine spezielle *Component*-Deklaration zur Verfügung. Diese Herangehensweise erlaubt es mehrere Logikkomponenten zu einem größeren System zusammenzufassen. Zur Verdeutlichung zeigt Lst. 7.2 eine Halbaddierer-Schaltung, bestehend aus jeweils einem als Komponente definierten 2-Input XOR- und AND-Gatter.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
ENTITY HalfAdder IS
    PORT( A :   IN    STD_LOGIC;
          B :   IN    STD_LOGIC;
          SUM : OUT   STD_LOGIC;    -- sum output
          CR :  OUT   STD_LOGIC     -- carry output
        );
END ENTITY HalfAdder;

ARCHITECTURE HalfAdderStructure OF HalfAdder IS
    -- signal declaration: none
    COMPONENT LogicXORGate      -- component declaration of XOR
        PORT( IN1  : IN     STD_LOGIC;
              IN2  : IN     STD_LOGIC;
              Q :    OUT    STD_LOGIC
            );
    END COMPONENT;

    COMPONENT LogicANDGate      -- component declaration of AND
        PORT( IN1  : IN     STD_LOGIC;
              IN2  : IN     STD_LOGIC;
              Q :    OUT    STD_LOGIC
            );
    END COMPONENT;

BEGIN                           -- structural description of half adder
    A1: LogicXORGate            -- instantiate XOR
        PORT MAP( IN1 => A,
                  IN2 => B,
                  Q   => SUM
                );
    A2: LogicANDGate            -- instantiate AND
        PORT MAP( IN1 => A,
                  IN2 => B,
                  Q   => CR
                );
END ARCHITECTURE HalfAdderStructure;
```

Listing 7.2: Halbaddierer aufgebaut aus einer XOR- und AND-Komponente

Die beiden Gatter werden als jeweils eigenständiges Entity implementiert und sind in Lst. 7.3 dargestellt.

```
-- AND Gate Entity
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY LogicANDGate IS
    PORT( IN1  : IN    STD_LOGIC;
          IN2  : IN    STD_LOGIC;
          Q :    OUT   STD_LOGIC
        );
END ENTITY LogicANDGate;

ARCHITECTURE AndBehaviour OF LogicANDGate IS
BEGIN     -- behavioural description as process
    AndProcess: PROCESS(IN1, IN2)
    BEGIN
        Q <= IN1 and IN2;
    END PROCESS AndProcess;

END ARCHITECTURE AndBehaviour;

-- XOR Gate Entity
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
ENTITY LogicXORGate IS
    PORT( IN1 : IN     STD_LOGIC;
          IN2 : IN     STD_LOGIC;
          Q  :  OUT    STD_LOGIC
        );
END ENTITY LogicXORGate;

ARCHITECTURE XorBehaviour OF LogicXORGate IS
BEGIN     -- behavioural description as process
    XorProcess: PROCESS(IN1, IN2)
    BEGIN
        Q <= IN1 xor IN2;
    END PROCESS XorProcess;

END ARCHITECTURE XorBehaviour;
```

Listing 7.3: 2-Input AND- und XOR-Gatter als Entity mit Process-Deklaration

Beim beschriebenen Halbaddierer werden ausschließlich kombinatorische Logikgatter verwendet. In vielen Systemen werden allerdings kombinatorische und sequentielle Schaltungen miteinander kombiniert. Dabei sollten zusätzliche Entwurfsprinzipien berücksichtigt werden. Bei einer synchronen Logik ist es besonders wichtig Signalübergänge nur bei Taktflanken zuzulassen.

7.2.2 Zustandsautomaten mit VHDL

In den meisten FPGAs kommen Zustandsautomaten mit einer *Zustands (Next State)*- und *Ausgangs (Output)*-Logik zum Einsatz.
Der VHDL-Code für ein solches Systems wird nachfolgend, anhand eines einfachen Beispiels, genauer erklärt. Das dazugehörige Blockdiagramm ist aus Abb. 7.3 ersichtlich.

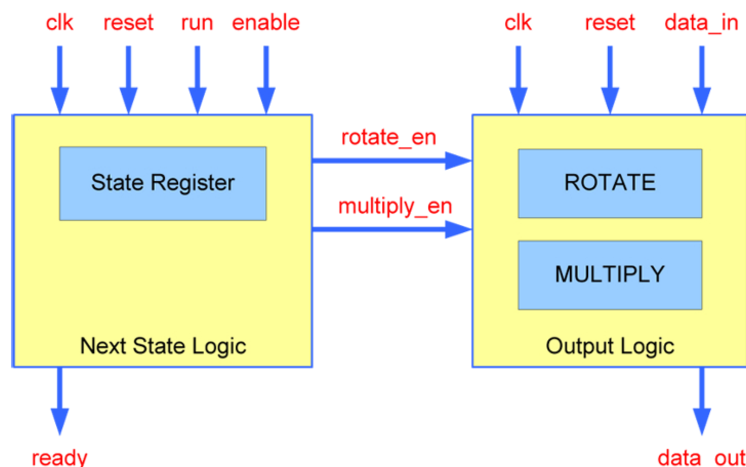

Abbildung 7.3: VHDL-Beispiel mit einem Zustandsautomaten.

Aus dem Blockdiagramm sind der Aufbau des Systems und die benötigten Signalleitungen ersichtlich. Die Zustandslogik besitzt insgesamt drei Zustände. Im ersten Zustand *IDLE* prüft die Zustandslogik, ob von außen ein Startsignal (run) gesetzt wurde. Ist dies der Fall, dann wird auf den zweiten Zustand *ROTATE* gewechselt. Hier wird der Ausgangslogik signalisiert, dass eine Rotation der Eingangsdaten durchgeführt werden soll. Dabei werden die zwei Byte der Eingangsdaten miteinander vertauscht. Nach dem Übergang zum Zustand *MULTIPLY* werden die vertauschten Daten mit den angelegten Eingangsdaten multipliziert und an den Ausgang weiter gegeben. Anschließend wechselt die Zustandslogik wieder in den ersten Zustand zurück.

Die Zustandslogik, mit den drei Zuständen, ist aus Lst. 7.4 ersichtlich. Die entsprechende Ausgangslogik ist in Lst. 7.5 dargestellt. Die darin verwendeten Komponenten, für die Rotation und Multiplikation, werden in Lst. 7.6 abgebildet.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY control IS
      PORT( -- to external interface:
           clk :            in  STD_LOGIC;
           reset :          in  STD_LOGIC;
           run :            in  STD_LOGIC;
           ready :          out STD_LOGIC;
           -- internal:
           rotate_en :      out STD_LOGIC;
           multiply_en :    out STD_LOGIC
      );
END ENTITY control;

ARCHITECTURE control_behaviour OF control IS

type STATE_TYPE is (IDLE, ROTATE, MULTIPLY);
signal current_state, next_state: STATE_TYPE;

BEGIN    -- Finite State Machine
    comb: PROCESS(current_state, run)
    BEGIN
        -- initialize outputs with default values
        ready        <= '0';
        multiply_en  <= '0';
        rotate_en    <= '0';

        CASE current_state IS
            when IDLE =>
                ready <= '1';
                IF run = '1' THEN
                    ready <= '0';
                    next_state <= ROTATE;
                ELSE
                    next_state <= IDLE;
                END IF;
            when ROTATE =>
                rotate_en <= '1';            -- enable rotate operation
                next_state <= MULTIPLY;
            when MULTIPLY =>
                multiply_en <= '1';          -- enable add operation
                next_state <= IDLE;
        END CASE;
    END PROCESS comb;

    sync: PROCESS(clk, reset)
    BEGIN
        IF reset = '1' THEN
            current_state <= IDLE;
        ELSIF (clk'event and clk = '1') THEN
            current_state <= next_state;
        END IF;
    END PROCESS sync;

END ARCHITECTURE control_behaviour;
```

Listing 7.4: Zustandslogik zur Steuerung der Ausgangslogik

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
```

```
ENTITY datapath IS
    PORT( -- to interface
          clk :         in  STD_LOGIC;
          reset :       in  STD_LOGIC;
          data_in :     in  STD_LOGIC_VECTOR(15 downto 0);
          data_out :    out STD_LOGIC_VECTOR(31 downto 0);
          -- internal
          rotate_en :   in  STD_LOGIC;
          multiply_en : in  STD_LOGIC
        );
END ENTITY;

ARCHITECTURE structure OF datapath IS
    COMPONENT rotator_16bit
      PORT( clk :    in  STD_LOGIC   := '0';
            reset :  in  STD_LOGIC   := '0';
            enable : in  STD_LOGIC   := '0';
            input :  in  STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
            output : out  STD_LOGIC_VECTOR(15 downto 0) := (others => '0')
          );
    END COMPONENT;

    COMPONENT multiplier_16bit
      PORT( clk :    in  STD_LOGIC := '0';
            reset :  in  STD_LOGIC := '0';
            enable : in  STD_LOGIC := '0';
            in1 :    in  STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
            in2 :    in  STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
            output : out  STD_LOGIC_VECTOR(31 downto 0) := (others => '0')
          );
    END COMPONENT;

    signal rotate_out: STD_LOGIC_VECTOR(15 downto 0) := (others => '0');

BEGIN
    rot: rotator_16bit
        PORT MAP( clk    => clk,
                  reset  => reset,
                  enable => rotate_en,
                  input  => data_in,
                  output => rotate_out
                );

    mul: multiplier_16bit
        PORT MAP( clk    => clk,
                  reset  => reset,
                  enable => multiply_en,
                  in1    => rotate_out,
                  in2    => data_in,
                  output => data_out
                );

END ARCHITECTURE structure;
```

Listing 7.5: Ausgangslogik mit Rotation und Multiplikation

Die Komponenten der Ausgangslogik werden jeweils mit einer dazugehörigen Enable-Signalleitung aktiviert. Sobald dies geschieht, wird die Rotation bzw. Multiplikation durchgeführt. Die Übernahme der Ausgangswerte erfolgt bei der nächsten Taktflanke.

```
-- 16 x 16-Bit multiplication
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY multiplier_16bit IS
  PORT( clk :    in  std_logic;
        reset :  in  std_logic;
```

```
          enable  : in   std_logic;
          in1  :    in   std_logic_vector(15 downto 0);
          in2  :    in   std_logic_vector(15 downto 0);
          output:   out  std_logic_vector(31 downto 0)
        );
END ENTITY multiplier_16bit;

ARCHITECTURE MultBehaviour OF multiplier_16bit IS
BEGIN
    mul: PROCESS(clk, reset)
    BEGIN
        IF reset='1' THEN
            output <= (others => '0');
        ELSIF (clk'event and clk ='1') THEN
            IF enable='1' THEN
                output <= in1 * in2;
            END IF;
        END IF;
    END PROCESS mul;

END ARCHITECTURE MultBehaviour;

-- byte-wise rotation of 16-bit
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY rotator_16bit IS
  PORT( clk  :    in   std_logic;
        reset :   in   std_logic;
        enable :  in   std_logic;
        input :   in   std_logic_vector(15 downto 0);
        output :  out  std_logic_vector(15 downto 0)
      );
END ENTITY rotator_16bit;

ARCHITECTURE RotBehaviour OF rotator_16bit IS
BEGIN
    rot: PROCESS(clk, reset)
    BEGIN
        IF reset='1' THEN
            output <= (others => '0');
        ELSIF (clk'event and clk ='1') THEN
            IF enable='1' THEN
                output(15 downto 8) <= input(7 downto 0);
                output(7 downto 0) <= input(15 downto 8);
            END IF;
        END IF;
    END PROCESS rot;

END ARCHITECTURE RotBehaviour;
```

Listing 7.6: Die Komponenten der Ausgangslogik

Die Funktionsweise der resultierenden Gesamtschaltung ist aus Abb. 7.4 ersichtlich.

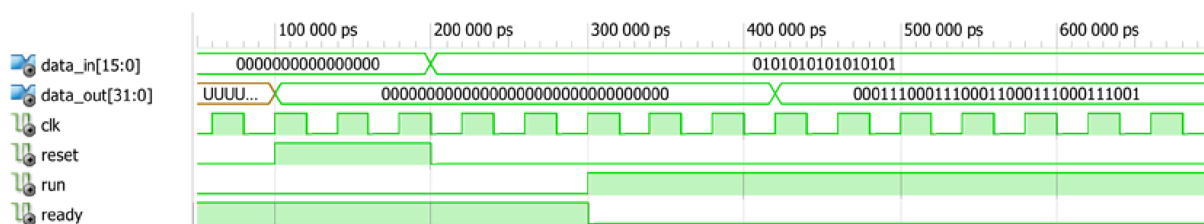

Abbildung 7.4: Simulationsergebnis des kompletten Zustandsautomaten.

Aus der Simulation ist erkennbar, dass die Eingangsdaten nach dem Reset an die Schaltung angelegt werden. Die Ausgangslogik übernimmt die Daten, sobald *run* logisch high wird. Nach drei Takten ist die Berechnung der Rotation und Multiplikation beendet und die Ausgangsdaten werden ausgegeben.

7.2.3 Realisierung eines TMR-Systemes

Wie bereits in Kapitel 5.4 beschrieben wurde, können TMR-Systeme zur Detektion von Ausfällen oder Störungen verwendet werden. Dabei werden die kritischen Module des Systems dreifach implementiert und deren Ausgangsdaten ständig mit einem Voter überprüft.
Im folgenden Beispiel sollen insgesamt drei 16 x 32-Bit RAM-Blöcke und ein Voter zu einem TMR-System kombiniert werden. Die Struktur der dafür notwendigen Ausgangslogik ist in Abb. 7.5 dargestellt.

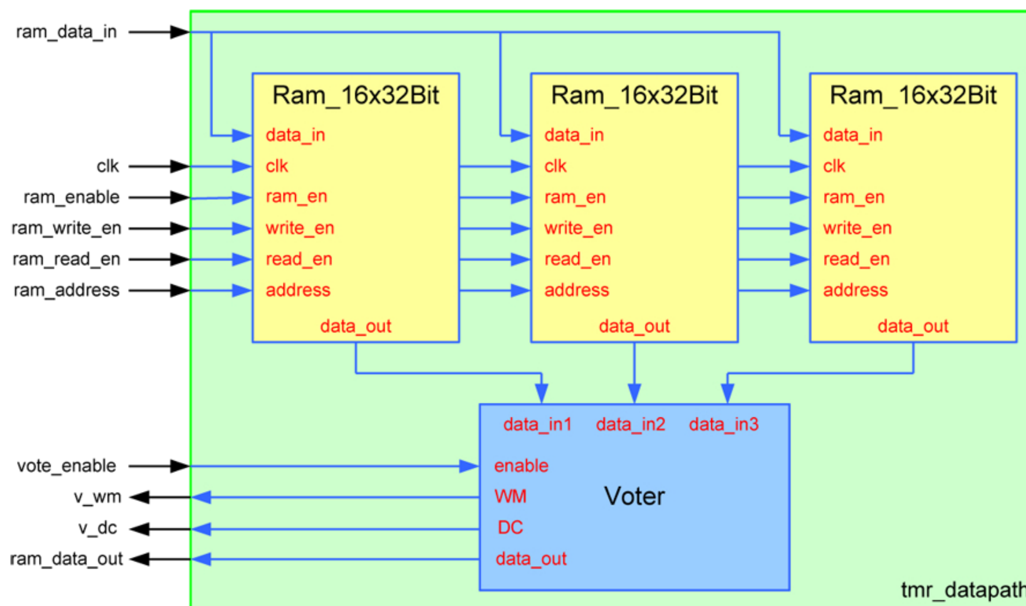

Abbildung 7.5: Struktur des TMR-Systems.

Die Schnittstellen des RAM-Blockes und des Voters sind aus der Abbildung und den implementierten Komponenten-Deklarationen ersichtlich. Zur Steuerung der Blöcke werden die verschiedenen Enable-Signale verwendet. Der vollständige VHDL-Code der Ausgangslogik ist aus Lst. 7.7 ersichtlich.

```
-- TMR-System datapath with three 16 x 32-Bit RAMs and Voter
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY tmr_datapath IS
    PORT( clk :            in    STD_LOGIC;
          ram_enable :     in    STD_LOGIC;
          ram_address :    in    INTEGER range 0 to 15;
          ram_write_en :   in    STD_LOGIC;
          ram_read_en :    in    STD_LOGIC;
          vote_enable :    in    STD_LOGIC;
```

```
            ram_data_in  :   in    STD_LOGIC_VECTOR(31 downto 0);
            ram_data_out :   out   STD_LOGIC_VECTOR(31 downto 0);
            v_wm :           out   STD_LOGIC;
            v_dc :           out   STD_LOGIC
        );
END ENTITY tmr_datapath;

ARCHITECTURE structure OF tmr_datapath IS

    COMPONENT ram_16x32bit
    PORT( clk       :   in   STD_LOGIC := '0';
          ram_en    :   in   STD_LOGIC := '0';
          write_en  :   in   STD_LOGIC := '0';
          read_en   :   in   STD_LOGIC := '0';
          address   :   in   INTEGER range 0 to 15 := 0;
          data_in   :   in   STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
          data_out  :   out  STD_LOGIC_VECTOR(31 downto 0) := (others => '0')
        );
    END COMPONENT ram_16x32bit;

    COMPONENT voter
    PORT( enable    :   in   STD_LOGIC := '0';
          data_in1  :   in   STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
          data_in2  :   in   STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
          data_in3  :   in   STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
          WM :          out  STD_LOGIC := '0';  -- warning message
          DC :          out  STD_LOGIC := '0';  -- data corruption
          data_out  :   out  STD_LOGIC_VECTOR(31 downto 0) := (others => '0')
        );
    END COMPONENT voter;

    signal v_in1 : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
    signal v_in2 : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
    signal v_in3 : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');

BEGIN

    ram1: ram_16x32bit
    PORT MAP( clk       => clk,
              ram_en    => ram_enable,
              write_en  => ram_write_en,
              read_en   => ram_read_en,
              address   => ram_address,
              data_in   => ram_data_in,
              data_out  => v_in1
             );

    ram2: ram_16x32bit
    PORT MAP( clk       => clk,
              ram_en    => ram_enable,
              write_en  => ram_write_en,
              read_en   => ram_read_en,
              address   => ram_address,
              data_in   => ram_data_in,
              data_out  => v_in2
            );

    ram3: ram_16x32bit
    PORT MAP( clk       => clk,
              ram_en    => ram_enable,
              write_en  => ram_write_en,
              read_en   => ram_read_en,
              address   => ram_address,
              data_in   => ram_data_in,
              data_out  => v_in3
            );

    vot: voter
    PORT MAP( enable    => vote_enable,
              data_in1  => v_in1,
              data_in2  => v_in2,
```

```
                data_in3    => v_in3,
                WM          => v_wm,
                DC          => v_dc,
                data_out    => ram_data_out
            );

END ARCHITECTURE structure;
```

Listing 7.7: Ausgangslogik eines TMR-Systems für einen RAM

Aus dem Beispiel ist erkennbar, dass die RAM-Blöcke als Komponenten implementiert sind. Dafür wurde im voraus bereits ein eigenes *Ram_16x32Bit*-Entity erstellt. Aus Gründen der Übersichtlichkeit wird dieses hier jedoch nicht im Detail dargestellt. Der entsprechende VHDL-Code kann allerdings aus dem Anhang entnommen werden.

Die drei RAM-Blöcke erhalten jeweils die identischen 32-Bit Eingangsdaten und speichern diese, sofern *ram_enable* und *write_enable* auf logisch high gesetzt sind. Das Aktivieren von *read_enable* sorgt dafür, dass die 32-Bit Ausgangsdaten bei der nächsten Taktflanke aus dem Speicher gelesen werden.
Die Daten werden anschließend direkt an den Voter übertragen, welcher diese auswertet, sobald *vote_enable* logisch high ist. Das Ergebnis wird entsprechend mit den Signalleitungen *WM (warning message)* und *DC (data corruption)* signalisiert. Sobald die Daten von mindestens einem der drei RAM-Blöcke abweichen, wird *WM* logisch high gesetzt. Die Ausgangsdaten werden dann von einem der zwei konsistenten RAM-Blöcke übernommen und am Ausgang des Voters ausgegeben. Es wird also in diesem Beispiel angenommen, dass die zwei RAM-Blöcke mit den identischen Daten auch tatsächlich fehlerfrei sind. Ein wirklich schwerer Datenfehler ist erst dann aufgetreten, wenn alle RAM-Blöcke unterschiedliche Daten enthalten. In diesem Fall wird *DC* zusätzlich auf logisch high gesetzt. Für die Ausgangsdaten wird entsprechend Null ausgegeben.

Die Implementierung des verwendeten Voters ist aus Lst. 7.8 ersichtlich.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY voter IS
    PORT( enable :    in   STD_LOGIC;
          data_in1 :  in   STD_LOGIC_VECTOR(31 downto 0);
          data_in2 :  in   STD_LOGIC_VECTOR(31 downto 0);
          data_in3 :  in   STD_LOGIC_VECTOR(31 downto 0);
          WM :        out  STD_LOGIC;  -- warning message
          DC :        out  STD_LOGIC;  -- data corruption
          data_out :  out  STD_LOGIC_VECTOR(31 downto 0)
        );
END ENTITY voter;

ARCHITECTURE VoterBehaviour OF voter IS

BEGIN
     vote: PROCESS(enable, data_in1, data_in2, data_in3) IS
     BEGIN
         -- initialize with default values
         WM <= '0';
         DC <= '0';
         data_out <= (others => '0');

         IF (enable = '1') THEN
             IF (data_in1 = data_in2) THEN
                 data_out <= data_in1;
                 IF (data_in1 /= data_in3) THEN     -- only 2 inputs equal
                     WM <= '1';
```

```
            END IF;
        ELSE
            WM <= '1';
            IF (data_in1 = data_in3) THEN      -- only 2 inputs equal
                data_out <= data_in1;
            ELSIF (data_in2 = data_in3) THEN   -- only 2 inputs equal
                data_out <= data_in2;
            ELSE                               -- nothing equal, corrupted
                data_out <= (others => '0');
                DC <= '1';
            END IF;
        END IF;
    END IF;
END PROCESS vote;

END ARCHITECTURE VoterBehaviour;
```

Listing 7.8: Voter für RAM-Block

Zur Steuerung der TMR-Schaltung werden die vorhandenen Enable-Signale verwendet. Dazu muss ein weiteres Entity mit der Zustandslogik implementiert werden. Dieses wird hier allerdings, aus Gründen der Übersichtlichkeit, nicht genauer beschrieben. Der entsprechende VHDL-Code kann allerdings aus dem Anhang entnommen werden.

Im nachfolgenden Abschnitt wird gezeigt, wie eine Logikkomponente getestetet werden kann.

7.2.4 Testen einzelner Logikkomponenten

Auf den folgenden Seiten soll gezeigt werden, wie während der Entwicklung einfache Tests der implementierten Logikkomponenten mit VHDL erfolgen können. Im Gegensatz zu Tests an der fertigen Hardware, werden dabei die implementierten digitalen Schaltungen durch Simulation überprüft.

In den meisten Fällen kommt dafür eine spezielle *VHDL Testbench*-Datei zum Einsatz. Dabei wird die zu testende Schaltung (Entity) als Komponente in ein spezielles Test-Entity eingefügt. Durch Vorgabe verschiedener Eingangswerte, kann anschließend die Funktionsweise, anhand der resultierenden Ausgangswerte, überprüft werden.
Einige Hersteller bieten, mit ihrer Entwicklungssoftware, die Möglichkeit solche Testbench-Dateien teilweise automatisiert zu erstellen. Dadurch können die Tests relativ einfach und schnell implementiert werden.

Der VHDL-Code in Lst. 7.9 zeigt ein solches Test-Entity, für die in Lst. 7.7 beschriebene Ausgangslogik eines TMR-Systems.

```
-- Test-Entity for TMR-System
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY test_tmr_datapath IS
END ENTITY test_tmr_datapath;

ARCHITECTURE TestBehaviour OF test_tmr_datapath IS

    COMPONENT tmr_datapath
        PORT( clk :             in    STD_LOGIC;
              ram_enable :      in    STD_LOGIC;
              ram_address :     in    INTEGER range 0 to 15;
              ram_write_en :    in    STD_LOGIC;
```

```
                ram_read_en  :   in    STD_LOGIC;
                vote_enable  :   in    STD_LOGIC;
                ram_data_in  :   in    STD_LOGIC_VECTOR(31 downto 0);
                ram_data_out :   out   STD_LOGIC_VECTOR(31 downto 0);
                v_wm :           out   STD_LOGIC;
                v_dc :           out   STD_LOGIC
              );
    END COMPONENT;

    signal clk :            STD_LOGIC := '0';
    signal ram_enable :     STD_LOGIC := '0';
    signal ram_address :    INTEGER range 0 to 15 := 0;
    signal ram_write_en :   STD_LOGIC := '0';
    signal ram_read_en :    STD_LOGIC := '0';
    signal vote_enable :    STD_LOGIC := '0';
    signal ram_data_in :    STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
    signal ram_data_out :   STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
    signal v_wm :           STD_LOGIC := '0';
    signal v_dc :           STD_LOGIC := '0';

BEGIN
    tmr_test: tmr_datapath
    PORT MAP( clk               => clk,
              ram_enable        => ram_enable,
              ram_address       => ram_address,
              ram_write_en      => ram_write_en,
              ram_read_en       => ram_read_en,
              vote_enable       => vote_enable,
              ram_data_in       => ram_data_in,
              ram_data_out      => ram_data_out,
              v_wm              => v_wm,
              v_dc              => v_dc
            );

    clk_proc: PROCESS
    BEGIN
        Wait for 0 ns;    clk <= '0';
        Wait for 20 ns;   clk <= '1';
        Wait for 20 ns;   clk <= '0';
    END PROCESS clk_proc;

    test_proc: PROCESS
    BEGIN
        -- initialize
        Wait for 100 ns; ram_enable  <= '0'; vote_enable  <= '0';
                         ram_address <= 1;   ram_write_en <= '0';
                         ram_read_en <= '0';
        -- write to ram
        Wait for 100 ns; ram_enable  <= '1'; vote_enable  <= '0';
                         ram_address <= 1;   ram_write_en <= '1';
                         ram_read_en <= '0';
                         ram_data_in <= "01010101010101010101010101010101";
        -- read ram over voter
        Wait for 100 ns; ram_enable  <= '1'; vote_enable  <= '1';
                         ram_address <= 1;   ram_write_en <= '0';
                         ram_read_en <= '1'; ram_data_in  <= (others => '0');
        -- read voter with deactivated ram
        Wait for 100 ns; ram_enable  <= '0'; vote_enable  <= '1';
                         ram_address <= 0;   ram_write_en <= '0';
                         ram_read_en <= '0'; ram_data_in  <= (others => '0');
    END PROCESS test_proc;

END ARCHITECTURE TestBehaviour;
```

Listing 7.9: Test-Entity für Ausgangslogik des TMR-Systems

Die zeitliche Abfolge, der zu simulierenden Eingangswerte, wird am Ende der Testbench-Datei festgelegt. Das Simulationsergebnis, mit den resultierenden Signalen für die gesamte Ausgangslogik des TMR-Systems, ist aus Abb. 7.6 ersichtlich.

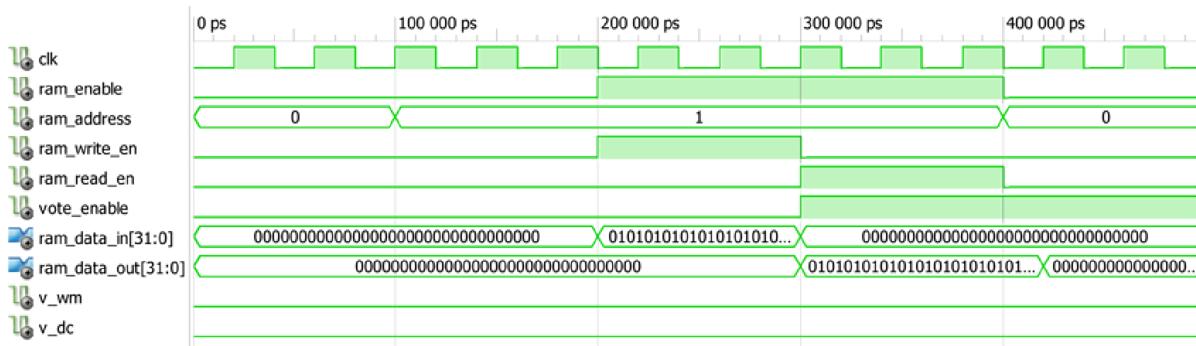

Abbildung 7.6: Simulationsergebnis der kompletten TMR-Ausgangslogik.

Man erkennt, dass die Eingangsdaten durch Setzen von *ram_read_en* übernommen werden. Die Daten können anschließend über den Voter, nach Aktivieren von *ram_write_en* und *vote_enable*, wieder aus dem RAM gelesen werden.

Mit diesem Test-Entity können allerdings auch die Gefahren von solchen Tests erläutert werden. Der Voter wird nämlich nur dann einen Fehler signalisieren, wenn die Daten in mindestens einem RAM-Block verändert wurden. Die Struktur des getesteten TMR-Systems (tmr_datapath) definiert aber, dass alle drei RAM-Blöcke am selben Datenbus *ram_data_in* hängen. Beim Test erhalten deshalb alle RAMs die selben Eingangsdaten. Mit dem Test-Entity kann also kein Fehler in den Daten eines RAMs hervorgerufen werden, weil nur auf die Ein- und Ausgänge des gesamten TMR-Systems Einfluss genommen werden kann. In der Schaltung müsste stattdessen ein eigener Eingangsbus für jeden RAM-Block vorhanden sein, damit beim Test ein Block mit falschen Daten beschrieben werden kann. Dies widerspricht allerdings der gewünschten Schaltungsstruktur.

Zur Vermeidung solcher Probleme müssen deshalb immer erst die kleinen Grundkomponenten eines Systems getestet werden. Erst wenn diese funktionieren werden die größeren Systeme überprüft. Als Beispiel dafür, ist die Testbench-Datei des Voters in Lst. 7.10 dargestellt.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY voter_test IS
END voter_test;

ARCHITECTURE TestBehaviour OF voter_test IS
    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT voter
    PORT(
            enable :   IN   std_logic := '0';
            data_in1 : IN   std_logic_vector(31 downto 0) := (others => '0');
            data_in2 : IN   std_logic_vector(31 downto 0) := (others => '0');
            data_in3 : IN   std_logic_vector(31 downto 0) := (others => '0');
           WM :        OUT  std_logic := '0';
           DC :        OUT  std_logic := '0';
            data_out : OUT  std_logic_vector(31 downto 0) := (others => '0')
        );
    END COMPONENT;

    signal enable : std_logic := '0';
    signal data_in1 : std_logic_vector(31 downto 0) := (others => '0');
```

```vhdl
    signal data_in2 : std_logic_vector(31 downto 0) := (others => '0');
    signal data_in3 : std_logic_vector(31 downto 0) := (others => '0');
    signal WM : std_logic;
    signal DC : std_logic;
    signal data_out : std_logic_vector(31 downto 0);

BEGIN
    uut: voter
    PORT MAP( enable => enable,
              data_in1 => data_in1,
              data_in2 => data_in2,
              data_in3 => data_in3,
             WM => WM,
             DC => DC,
              data_out => data_out
            );

    -- Stimulus process
    test_proc: PROCESS
    BEGIN
        wait for 100ns;  enable <= '1';
                         data_in1 <= "00001111000011110000111100001111";
                         data_in2 <= "00001111000011110000111100001111";
                         data_in3 <= "00001111000011110000111100001111";
        wait for 90ns;   enable <= '0';
        wait for 10ns;   enable <= '1';
                         data_in1 <= "11111111000011110000111100001111";
                         data_in2 <= "00001111000011110000111100001111";
                         data_in3 <= "00001111000011110000111100001111";
        wait for 90ns;   enable <= '0';
        wait for 10ns;   enable <= '1';
                         data_in1 <= "00001111000011110000111100001111";
                         data_in2 <= "11111111000011110000111100001111";
                         data_in3 <= "00001111000011110000111100001111";
        wait for 90ns;   enable <= '0';
        wait for 10ns;   enable <= '1';
                         data_in1 <= "00001111000011110000111100001111";
                         data_in2 <= "00001111000011110000111100001111";
                         data_in3 <= "11111111000011110000111100001111";
        wait for 90ns;   enable <= '0';
        wait for 10ns;   enable <= '1';
                         data_in1 <= "11001111000011110000111100001111";
                         data_in2 <= "00111111000011110000111100001111";
                         data_in3 <= "00001111000011110000111100001111";
   END PROCESS test_proc;

END ARCHITECTURE TestBehaviour;
```

Listing 7.10: Test-Entity für den Voter des TMR-Systems

Das Simulationsergebnis des Voters ist in Abb. 7.7 dargestellt. Man erkennt, dass der Voter bei abweichenden Eingangsdaten eine Unstimmigkeit (WM high) oder einen Datenfehler (DC high) signalisiert.

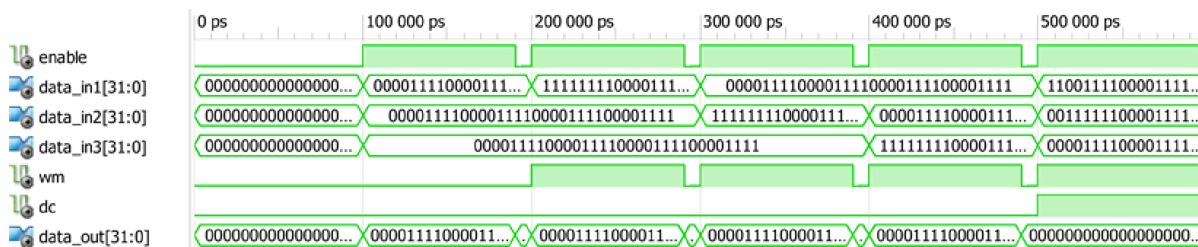

Abbildung 7.7: Simulationsergebnis für den Voter.

Die oben beschriebene Realisierung eines TMR-Systems ist allerdings relativ aufwändig, da aus einem einzigen RAM-Block eine viel größere Schaltung erstellt werden muss. Bei Systemen mit sehr vielen Logikkomponenten ist der Aufwand, für die manuelle Realisierung, rasch nicht mehr zu bewältigen. Aus diesem Grund stellen viele Hersteller eine spezielle Entwicklungssoftware zur Verfügung, welche automatisch die kritischen Teile einer Schaltung in ein TMR-System umwandelt. Dies spart viel Zeit und Aufwand, jedoch hängt die genaue Struktur und die Qualität des resultierenden Systems von der verwendeten Software ab.

7.3 Zusammenfassung

Zur Programmierung von FPGAs wird in der Regel eine spezielle Entwicklungssoftware eingesetzt. Während die Schaltungen auf den höheren Abstraktionsebenen oft grafisch und textbasiert eingegeben werden, kommen auf niedrigeren Abstraktionsebenen in der Praxis vorwiegend die textbasierten Programmiersprachen VHDL und Verilog-HDL zum Einsatz. Zur Entwicklung einer, auf eingebetteten Prozessorkernen, lauffähigen Software wird bevorzugt C und C++ herangezogen.
Die Wahl der tatsächlich verwendeten Programmiersprache hängt allerdings von zahlreichen Faktoren ab. Beispielsweise unterstützt die verwendete Entwicklungssoftware nicht alle Programmiersprachen oder der Entwickler ist mit einer anderen Technik vertrauter.

Durch automatisierte Synthese, Optimierung und Simulation wird der Entwicklungsaufwand drastisch minimiert. Die Systeme können somit größtenteils im voraus getestet und auf mögliche Schwächen überprüft werden. Dank der immer besseren Entwicklungsumgebungen können zunehmend mehr Prozesse automatisiert durchgeführt werden.
Viele Hersteller bieten eine kostenlose Basissoftware mit den wichtigsten Werkzeugen zur FPGA-Entwicklung. Jedoch wird während der Entwicklung meist eine entsprechende Infrastruktur für die Hardware benötigt. Dies sind bevorzugt spezielle Entwicklungs-Platinen, welche mit den notwendigen Peripheriebausteinen ausgestattet sind.

Die gezeigten VHDL-Beispiele sollen einen kurzen Einblick in das Konzept der Hardware-Programmierung vermitteln. Darüber hinaus gibt es natürlich eine Vielzahl anderer Entwurfsprinzipien und Strategien in VHDL. Weitere Informationen dazu sind entsprechend der einschlägigen Fachliteratur zu entnehmen.

8 Anhang

```
1        -- Project: 4-Bit subtraction
2        -- Entity:  subtractor
3        -- Autor:   Reinhard Zeif
4        -- Date:    Feb. 2011
5        ------------------------------------------------------------
6        LIBRARY IEEE;
7        USE IEEE.STD_LOGIC_1164.ALL;
8        USE IEEE.STD_LOGIC_UNSIGNED.ALL;
9        USE IEEE.STD_LOGIC_ARITH.ALL;
10
11       ENTITY FourBitSub IS
12           PORT( IN1 : IN    STD_LOGIC_VECTOR(3 downto 0);
13                 IN2 : IN    STD_LOGIC_VECTOR(3 downto 0);
14                 SUB : OUT   STD_LOGIC_VECTOR(3 downto 0)
15               );
16       END ENTITY FourBitSub;
17
18       ARCHITECTURE SubFunctionality OF FourBitSub IS
19           -- Signal declaration, Constant declaration, Components
20       BEGIN
21           -- dataflow description
22           SUB <= IN1(3 downto 0) - IN2(3 downto 0);
23
24       END ARCHITECTURE SubFunctionality;
```

```
 1  -- Project: Half adder
 2  -- Entity:  Half adder
 3  -- Autor:   Reinhard Zeif
 4  -- Date:    Feb. 2011
 5  ----------------------------------------------------------
 6  LIBRARY IEEE;
 7  USE IEEE.STD_LOGIC_1164.ALL;
 8  USE IEEE.NUMERIC_STD.ALL;
 9  USE IEEE.STD_LOGIC_UNSIGNED.ALL;
10
11  ENTITY HalfAdder IS
12      PORT( A :    IN    STD_LOGIC;
13            B :    IN    STD_LOGIC;
14            SUM : OUT    STD_LOGIC;    -- sum output
15            CR :  OUT    STD_LOGIC     -- carry output
16          );
17  END ENTITY HalfAdder;
18
19  ARCHITECTURE HalfAdderStructure OF HalfAdder IS
20      -- signal declaration: none
21     COMPONENT LogicXORGate    -- component declaration of XOR
22          PORT( IN1 : IN    STD_LOGIC := '0';
23                IN2 : IN    STD_LOGIC := '0';
24                Q :   OUT   STD_LOGIC := '0'
25              );
26     END COMPONENT;
27
28     COMPONENT LogicANDGate    -- component declaration of AND
29          PORT( IN1 : IN    STD_LOGIC := '0';
30                IN2 : IN    STD_LOGIC := '0';
31                Q :   OUT   STD_LOGIC := '0'
32              );
33     END COMPONENT;
34
35  BEGIN                  -- structural description of half adder
36      A1: LogicXORGate    -- instantiate XOR
37          PORT MAP(IN1 => A, IN2 => B, Q => SUM);
38      A2: LogicANDGate    -- instantiate AND
39          PORT MAP(IN1 => A, IN2 => B, Q => CR);
40
41  END ARCHITECTURE HalfAdderStructure;
```

LogicANDGate.vhd

```
1        -- Project: Half adder
2        -- Entity:  2-Input AND gate
3        -- Autor:   Reinhard Zeif
4        -- Date:    Feb. 2011
5        ----------------------------------------------------------
6        LIBRARY IEEE;
7        USE IEEE.STD_LOGIC_1164.ALL;
8        USE IEEE.STD_LOGIC_UNSIGNED.ALL;
9
10       ENTITY LogicANDGate IS
11           PORT( IN1 : IN    STD_LOGIC;
12                 IN2 : IN    STD_LOGIC;
13                 Q :   OUT   STD_LOGIC
14               );
15       END ENTITY LogicANDGate;
16
17       ARCHITECTURE AndBehaviour OF LogicANDGate IS
18       BEGIN    -- behavioural description as process
19           AndProcess: PROCESS(IN1, IN2)
20           BEGIN
21               Q <= IN1 and IN2;
22           END PROCESS AndProcess;
23
24       END ARCHITECTURE AndBehaviour;
```

LogicXORGate.vhd

```
1   -- Project: Half adder
2   -- Entity:  2-Input XOR gate
3   -- Autor:   Reinhard Zeif
4   -- Date:    Feb. 2011
5   ----------------------------------------------------------
6   LIBRARY IEEE;
7   USE IEEE.STD_LOGIC_1164.ALL;
8   USE IEEE.STD_LOGIC_UNSIGNED.ALL;
9
10  ENTITY LogicXORGate IS
11      PORT( IN1 : IN    STD_LOGIC;
12            IN2 : IN    STD_LOGIC;
13            Q :   OUT   STD_LOGIC
14          );
15  END ENTITY LogicXORGate;
16
17  ARCHITECTURE XorBehaviour OF LogicXORGate IS
18  BEGIN    -- behavioural description as process
19      XorProcess: PROCESS(IN1, IN2)
20      BEGIN
21          Q <= IN1 xor IN2;
22      END PROCESS XorProcess;
23
24  END ARCHITECTURE XorBehaviour;
25
26
```

control.vhd

```
1        -- Project: State machine with rotation and multiplication
2        -- Entity:  Next state logic (control)
3        -- Autor:   Reinhard Zeif
4        -- Date:    Feb. 2011
5        ------------------------------------------------------------
6        LIBRARY IEEE;
7        USE IEEE.STD_LOGIC_1164.ALL;
8        USE IEEE.NUMERIC_STD.ALL;
9
10       ENTITY control IS
11             PORT( -- to external interface:
12                 clk :           in  STD_LOGIC;
13                 reset :         in  STD_LOGIC;
14                 run :           in  STD_LOGIC;
15                 ready :         out STD_LOGIC;
16                 -- internal:
17                 rotate_en :     out STD_LOGIC;
18                 multiply_en :   out STD_LOGIC
19             );
20       END ENTITY control;
21
22       ARCHITECTURE control_behaviour OF control IS
23           type STATE_TYPE is (IDLE, ROTATE, MULTIPLY);
24           signal current_state, next_state: STATE_TYPE;
25
26       BEGIN    -- Finite State Machine
27           comb: PROCESS(current_state, run)
28           BEGIN
29               -- initialize outputs with default values
30               ready         <= '0';
31               multiply_en   <= '0';
32               rotate_en     <= '0';
33
34               CASE current_state IS
35                   when IDLE =>
36                       ready <= '1';
37                       IF run = '1' THEN
38                           ready <= '0';
39                           next_state <= ROTATE;
40                       ELSE
41                           next_state <= IDLE;
42                       END IF;
43                   when ROTATE =>
44                       rotate_en <= '1';          -- enable rotate operation
45                       next_state <= MULTIPLY;
46                   when MULTIPLY =>
47                       multiply_en <= '1';        -- enable add operation
48                       next_state <= IDLE;
49               END CASE;
50           END PROCESS comb;
51
52           sync: PROCESS(clk, reset)
53           BEGIN
54               IF reset = '1' THEN
55                   current_state <= IDLE;
56               ELSIF (clk'event and clk = '1') THEN
57                   current_state <= next_state;
58               END IF;
59           END PROCESS sync;
60
61       END ARCHITECTURE control_behaviour;
```

```
1       -- Project: State machine with rotation and multiplication
2       -- Entity:  Datapath (output logic)
3       -- Autor:   Reinhard Zeif
4       -- Date:    Feb. 2011
5       ------------------------------------------------------------
6       LIBRARY IEEE;
7       USE IEEE.STD_LOGIC_1164.ALL;
8       USE IEEE.NUMERIC_STD.ALL;
9
10      ENTITY datapath IS
11          PORT( -- to interface
12                  clk :         in  STD_LOGIC;
13                  reset :       in  STD_LOGIC;
14                  data_in :     in  STD_LOGIC_VECTOR(15 downto 0);
15                  data_out :    out STD_LOGIC_VECTOR(31 downto 0);
16                  -- internal
17                  rotate_en :   in  STD_LOGIC;
18                  multiply_en : in  STD_LOGIC
19                );
20      END ENTITY;
21
22      ARCHITECTURE structure OF datapath IS
23          COMPONENT rotator_16bit
24            PORT( clk :    in  STD_LOGIC   := '0';
25                  reset :  in  STD_LOGIC   := '0';
26                  enable : in  STD_LOGIC   := '0';
27                  input :  in  STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
28                  output : out  STD_LOGIC_VECTOR(15 downto 0) := (others => '0')
29                );
30          END COMPONENT;
31
32          COMPONENT multiplier_16bit
33            PORT( clk :    in  STD_LOGIC := '0';
34                  reset :  in  STD_LOGIC := '0';
35                  enable : in  STD_LOGIC := '0';
36                  in1 :    in  STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
37                  in2 :    in  STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
38                  output : out  STD_LOGIC_VECTOR(31 downto 0) := (others => '0')
39                );
40          END COMPONENT;
41
42          signal rotate_out: STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
43
44      BEGIN
45          rot: rotator_16bit
46              PORT MAP( clk    => clk,
47                        reset  => reset,
48                        enable => rotate_en,
49                        input  => data_in,
50                        output => rotate_out
51                      );
52
53          mul: multiplier_16bit
54              PORT MAP( clk    => clk,
55                        reset  => reset,
56                        enable => multiply_en,
57                        in1    => rotate_out,
58                        in2    => data_in,
59                        output => data_out
60                      );
61
```

```
62        END ARCHITECTURE structure;
```

rotator_16bit.vhd

```
1        -- Project: State machine with rotation and multiplication
2        -- Entity:  16-Bit byte-by-byte rotator
3        -- Autor:   Reinhard Zeif
4        -- Date:    Feb. 2011
5        ------------------------------------------------------------
6        LIBRARY IEEE;
7        USE IEEE.STD_LOGIC_1164.ALL;
8        USE IEEE.NUMERIC_STD.ALL;
9
10       ENTITY rotator_16bit IS
11         PORT( clk :    in  std_logic;
12               reset :  in  std_logic;
13               enable : in  std_logic;
14               input :  in  std_logic_vector(15 downto 0);
15               output : out std_logic_vector(15 downto 0)
16             );
17       END ENTITY rotator_16bit;
18
19       ARCHITECTURE RotBehaviour OF rotator_16bit IS
20       BEGIN
21           rot: PROCESS(clk, reset)
22           BEGIN
23               IF reset='1' THEN
24                   output <= (others => '0');
25               ELSIF (clk'event and clk ='1') THEN
26                   IF enable='1' THEN
27                       output(15 downto 8) <= input(7 downto 0);
28                       output(7 downto 0) <= input(15 downto 8);
29                   END IF;
30               END IF;
31           END PROCESS rot;
32       END ARCHITECTURE RotBehaviour;
```

multiplier_16bit.vhd

```
1         -- Project: State machine with rotation and multiplication
2         -- Entity:  16-Bit multiplier
3         -- Autor:   Reinhard Zeif
4         -- Date:    Feb. 2011
5         ----------------------------------------------------------
6         LIBRARY IEEE;
7         USE IEEE.STD_LOGIC_1164.ALL;
8         USE IEEE.STD_LOGIC_UNSIGNED.ALL;
9         USE IEEE.NUMERIC_STD.ALL;
10
11        ENTITY multiplier_16bit IS
12          PORT( clk :    in  std_logic;
13                reset :  in  std_logic;
14                enable : in  std_logic;
15                in1 :    in  std_logic_vector(15 downto 0);
16                in2 :    in  std_logic_vector(15 downto 0);
17                output:  out std_logic_vector(31 downto 0)
18              );
19        END ENTITY multiplier_16bit;
20
21        ARCHITECTURE MultBehaviour OF multiplier_16bit IS
22        BEGIN
23            mul: PROCESS(clk, reset)
24            BEGIN
25                IF reset='1' THEN
26                    output <= (others => '0');
27                ELSIF (clk'event and clk ='1') THEN
28                    IF enable='1' THEN
29                        output <= in1 * in2;
30                    END IF;
31                END IF;
32            END PROCESS mul;
33        END ARCHITECTURE MultBehaviour;
34
```

system.vhd

```
1        -- Project: State machine with rotation and multiplication
2        -- Entity:  Complete system - combines next state logic and
3        --          output logic with multiplication and rotation
4        -- Autor:   Reinhard Zeif
5        -- Date:    Feb. 2011
6        ------------------------------------------------------------
7        LIBRARY IEEE;
8        USE IEEE.STD_LOGIC_1164.ALL;
9        USE IEEE.STD_LOGIC_UNSIGNED.ALL;
10       USE IEEE.NUMERIC_STD.ALL;
11
12       ENTITY system IS
13           PORT( clk : IN  std_logic;
14                 reset : IN  std_logic;
15                 run : IN  std_logic;
16                 ready : OUT  std_logic;
17                 data_in :     in  STD_LOGIC_VECTOR(15 downto 0);
18                 data_out :    out STD_LOGIC_VECTOR(31 downto 0)
19               );
20       END system;
21
22       ARCHITECTURE SysBehaviour OF system IS
23           -- Component Declaration for the Unit Under Test (UUT)
24           COMPONENT control
25           PORT( clk : IN  std_logic := '0';
26                 reset : IN  std_logic := '0';
27                 run : IN  std_logic := '0';
28                 ready : OUT  std_logic := '0';
29                 rotate_en : OUT  std_logic := '0';
30                 multiply_en : OUT  std_logic := '0'
31               );
32           END COMPONENT;
33
34           COMPONENT datapath
35           PORT( -- to interface
36                 clk :         in  STD_LOGIC := '0';
37                 reset :       in  STD_LOGIC := '0';
38                 data_in :     in  STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
39                 data_out :    out STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
40                 -- internal
41                 rotate_en :   in  STD_LOGIC := '0';
42                 multiply_en : in  STD_LOGIC := '0'
43               );
44           END COMPONENT;
45
46          signal rot_enable : std_logic := '0';
47          signal mul_enable : std_logic := '0';
48
49       BEGIN
50
51          -- Instantiate the Unit Under Test (UUT)
52          uut: control
53              PORT MAP( clk => clk,
54                        reset => reset,
55                        run => run,
56                        ready => ready,
57                        rotate_en => rot_enable,
58                        multiply_en => mul_enable
59                      );
60
61          dp: datapath
```

```
62              PORT MAP( clk => clk,
63                        reset => reset,
64                        data_in => data_in,
65                        data_out => data_out,
66                        rotate_en => rot_enable,
67                        multiply_en => mul_enable
68                      );
69
70          END ARCHITECTURE SysBehaviour;
```

system_test.vhd

```
1        -- Project: State machine with rotation and multiplication
2        -- Entity:  Test for complete System
3        -- Autor:   Reinhard Zeif
4        -- Date:    Feb. 2011
5        ------------------------------------------------------------
6        LIBRARY IEEE;
7        USE IEEE.std_logic_1164.ALL;
8        USE IEEE.numeric_std.ALL;
9
10       ENTITY test_system IS
11       END test_system;
12
13       ARCHITECTURE TestBehaviour OF test_system IS
14           -- Component Declaration
15           COMPONENT system
16               PORT( clk : IN  std_logic;
17                     reset : IN  std_logic;
18                     run : IN  std_logic;
19                     ready : OUT  std_logic;
20                     data_in :     in  STD_LOGIC_VECTOR(15 downto 0);
21                     data_out :    out STD_LOGIC_VECTOR(31 downto 0)
22                   );
23           END COMPONENT;
24
25           signal data_in :  STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
26           signal data_out : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
27           signal clk : std_logic := '0';
28           signal reset : std_logic := '0';
29           signal run : std_logic := '0';
30           signal ready : std_logic;
31
32       BEGIN
33           -- Component Instantiation
34           uut: system
35               PORT MAP( clk => clk,
36                         reset => reset,
37                         run => run,
38                         ready => ready,
39                         data_in => data_in,
40                         data_out => data_out
41                       );
42
43           clk_proc: PROCESS
44           BEGIN
45               clk <= '0';
46               wait for 20ns;
47               clk <= '1';
48               wait for 20ns;
49           END PROCESS clk_proc;
50
51           test_proc: PROCESS
52           BEGIN
53               wait for 100ns;  reset <= '1';  run <= '0';
54               wait for 100ns;  reset <= '0';  data_in <= "0101010101010101";
55               wait for 100ns;  run <= '1';
56               wait for 100ns;  run <= '1';
57               wait for 100ns;  run <= '1';
58               wait for 100ns;  run <= '1';
59           END PROCESS test_proc;
60
61       END ARCHITECTURE TestBehaviour;
```

```
1         -- Project: TMR-System with three RAMs and Voter
2         -- Entity:  Datapath of TMR system
3         -- Autor:   Reinhard Zeif
4         -- Date:    Feb. 2011
5         ----------------------------------------------------------
6         LIBRARY IEEE;
7         USE IEEE.STD_LOGIC_1164.ALL;
8         USE IEEE.STD_LOGIC_UNSIGNED.ALL;
9         USE IEEE.NUMERIC_STD.ALL;
10
11        ENTITY tmr_datapath IS
12            PORT( clk :           in    STD_LOGIC;
13                  ram_enable :    in    STD_LOGIC;
14                  ram_address :   in    INTEGER range 0 to 15;
15                  ram_write_en :  in    STD_LOGIC;
16                  ram_read_en :   in    STD_LOGIC;
17                  vote_enable :   in    STD_LOGIC;
18                  ram_data_in :   in    STD_LOGIC_VECTOR(31 downto 0);
19                  ram_data_out :  out   STD_LOGIC_VECTOR(31 downto 0);
20                  v_wm :          out   STD_LOGIC;
21                  v_dc :          out   STD_LOGIC
22                );
23        END ENTITY tmr_datapath;
24
25        ARCHITECTURE structure OF tmr_datapath IS
26
27            COMPONENT ram_16x32bit
28            PORT( clk      :    in    STD_LOGIC := '0';
29                  ram_en :      in    STD_LOGIC := '0';
30                  write_en :    in    STD_LOGIC := '0';
31                  read_en :     in    STD_LOGIC := '0';
32                  address :     in    INTEGER range 0 to 15 := 0;
33                  data_in :     in    STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
34                  data_out :    out   STD_LOGIC_VECTOR(31 downto 0) := (others => '0')
35                );
36            END COMPONENT ram_16x32bit;
37
38            COMPONENT voter
39            PORT( enable :      in    STD_LOGIC := '0';
40                  data_in1 :    in    STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
41                  data_in2 :    in    STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
42                  data_in3 :    in    STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
43                  WM :          out   STD_LOGIC := '0';  -- warning message
44                  DC :          out   STD_LOGIC := '0';  -- data corruption
45                  data_out :    out   STD_LOGIC_VECTOR(31 downto 0) := (others => '0')
46                );
47            END COMPONENT voter;
48
49            signal v_in1 : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
50            signal v_in2 : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
51            signal v_in3 : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
52
53        BEGIN
54
55            ram1: ram_16x32bit
56            PORT MAP( clk       => clk,
57                      ram_en    => ram_enable,
58                      write_en  => ram_write_en,
59                      read_en   => ram_read_en,
60                      address   => ram_address,
61                      data_in   => ram_data_in,
```

```
62                    data_out   => v_in1
63                  );
64
65          ram2: ram_16x32bit
66          PORT MAP( clk        => clk,
67                    ram_en     => ram_enable,
68                    write_en   => ram_write_en,
69                    read_en    => ram_read_en,
70                    address    => ram_address,
71                    data_in    => ram_data_in,
72                    data_out   => v_in2
73                  );
74
75          ram3: ram_16x32bit
76          PORT MAP( clk        => clk,
77                    ram_en     => ram_enable,
78                    write_en   => ram_write_en,
79                    read_en    => ram_read_en,
80                    address    => ram_address,
81                    data_in    => ram_data_in,
82                    data_out   => v_in3
83                  );
84
85          vot: voter
86          PORT MAP( enable     => vote_enable,
87                    data_in1   => v_in1,
88                    data_in2   => v_in2,
89                    data_in3   => v_in3,
90                    WM         => v_wm,
91                    DC         => v_dc,
92                    data_out   => ram_data_out
93                  );
94
95      END ARCHITECTURE structure;
```

```
1        -- Project: TMR-System with three RAMs and Voter
2        -- Entity:  16 x 32-Bit RAM-block
3        -- Autor:   Reinhard Zeif
4        -- Date:    Feb. 2011
5        ----------------------------------------------------------
6        LIBRARY IEEE;
7        USE IEEE.STD_LOGIC_1164.ALL;
8        USE IEEE.STD_LOGIC_UNSIGNED.ALL;
9        USE IEEE.NUMERIC_STD.ALL;
10
11       ENTITY ram_16x32bit IS
12           PORT( ram_en :    in     STD_LOGIC;
13                 clk :       in     STD_LOGIC;
14                 write_en :  in     STD_LOGIC;
15                 read_en :   in     STD_LOGIC;
16                 address :   in     INTEGER range 0 to 15;
17                 data_in :   in     STD_LOGIC_VECTOR(31 downto 0);
18                 data_out :  out    STD_LOGIC_VECTOR(31 downto 0)
19               );
20       END ENTITY ram_16x32bit;
21
22       ARCHITECTURE RamBehaviour OF ram_16x32bit IS
23
24       BEGIN
25           PROCESS(clk, address, write_en, read_en, ram_en)
26
27           TYPE mem_array IS ARRAY(0 to 15) OF STD_LOGIC_VECTOR(31 downto 0);
28           variable ram_memory: mem_array;
29           BEGIN
30               IF (clk'event and clk ='1') THEN
31                   IF (ram_en = '1') THEN
32                       IF (write_en = '1') THEN
33                           ram_memory(address) := data_in(31 downto 0);
34                       ELSIF (read_en = '1') THEN
35                           data_out(31 downto 0) <= ram_memory(address);
36                       END IF;
37                   ELSE
38                       data_out(31 downto 0) <= (others => '0');
39                   END IF;
40               END IF;
41           END PROCESS;
42
43       END ARCHITECTURE RamBehaviour;
```

voter.vhd

```
1         -- Project: TMR-System with three RAMs and Voter
2         -- Entity:  Voter
3         -- Autor:   Reinhard Zeif
4         -- Date:    Feb. 2011
5         ----------------------------------------------------------
6         LIBRARY IEEE;
7         USE IEEE.STD_LOGIC_1164.ALL;
8         USE IEEE.NUMERIC_STD.ALL;
9
10        ENTITY voter IS
11            PORT( enable :    in   STD_LOGIC;
12                  data_in1 :  in   STD_LOGIC_VECTOR(31 downto 0);
13                  data_in2 :  in   STD_LOGIC_VECTOR(31 downto 0);
14                  data_in3 :  in   STD_LOGIC_VECTOR(31 downto 0);
15                  WM :        out  STD_LOGIC;  -- warning message
16                  DC :        out  STD_LOGIC;  -- data corruption
17                  data_out :  out  STD_LOGIC_VECTOR(31 downto 0)
18                );
19        END ENTITY voter;
20
21        ARCHITECTURE VoterBehaviour OF voter IS
22
23        BEGIN
24            vote: PROCESS(enable, data_in1, data_in2, data_in3) IS
25            BEGIN
26                -- initialize with default values
27                WM <= '0';
28                DC <= '0';
29                data_out <= (others => '0');
30
31                IF (enable = '1') THEN
32                   IF (data_in1 = data_in2) THEN
33                       data_out <= data_in1;
34                       IF (data_in1 /= data_in3) THEN      -- only 2 inputs equal
35                           WM <= '1';
36                       END IF;
37                   ELSE
38                       WM <= '1';
39                       IF (data_in1 = data_in3) THEN       -- only 2 inputs equal
40                           data_out <= data_in1;
41                       ELSIF (data_in2 = data_in3) THEN    -- only 2 inputs equal
42                           data_out <= data_in2;
43                       ELSE                                -- nothing equal, corrupted
44                           data_out <= (others => '0');
45                           DC <= '1';
46                       END IF;
47                   END IF;
48               END IF;
49            END PROCESS vote;
50
51        END ARCHITECTURE VoterBehaviour;
```

```
 1        -- Project: TMR-System with three RAMs and Voter
 2        -- Entity:  Test for voter
 3        -- Autor:   Reinhard Zeif
 4        -- Date:    Feb. 2011
 5        ----------------------------------------------------------
 6        LIBRARY ieee;
 7        USE ieee.std_logic_1164.ALL;
 8        USE ieee.std_logic_unsigned.all;
 9        USE ieee.numeric_std.ALL;
10
11        ENTITY voter_test IS
12        END voter_test;
13
14        ARCHITECTURE behavior OF voter_test IS
15
16            -- Component Declaration for the Unit Under Test (UUT)
17
18            COMPONENT voter
19            PORT( enable : IN  std_logic;
20                  data_in1 : IN  std_logic_vector(31 downto 0);
21                  data_in2 : IN  std_logic_vector(31 downto 0);
22                  data_in3 : IN  std_logic_vector(31 downto 0);
23                  WM : OUT  std_logic;
24                  DC : OUT  std_logic;
25                  data_out : OUT  std_logic_vector(31 downto 0)
26                );
27            END COMPONENT;
28
29            signal enable : std_logic := '0';
30            signal data_in1 : std_logic_vector(31 downto 0) := (others => '0');
31            signal data_in2 : std_logic_vector(31 downto 0) := (others => '0');
32            signal data_in3 : std_logic_vector(31 downto 0) := (others => '0');
33            signal WM : std_logic;
34            signal DC : std_logic;
35            signal data_out : std_logic_vector(31 downto 0);
36
37        BEGIN
38            uut: voter
39            PORT MAP( enable => enable,
40                      data_in1 => data_in1,
41                      data_in2 => data_in2,
42                      data_in3 => data_in3,
43                      WM => WM,
44                      DC => DC,
45                      data_out => data_out
46                    );
47
48            -- Stimulus process
49            test_proc: PROCESS
50            BEGIN
51                wait for 100ns; enable <= '1';
52                                data_in1 <= "00001111000011110000111100001111";
53                                data_in2 <= "00001111000011110000111100001111";
54                                data_in3 <= "00001111000011110000111100001111";
55                wait for 90ns;  enable <= '0';
56                wait for 10ns;  enable <= '1';
57                                data_in1 <= "11111111000011110000111100001111";
58                                data_in2 <= "00001111000011110000111100001111";
59                                data_in3 <= "00001111000011110000111100001111";
60                wait for 90ns;  enable <= '0';
61                wait for 10ns;  enable <= '1';
```

```
62                                  data_in1 <= "00001111000011110000111100001111";
63                                  data_in2 <= "11111111000011110000111100001111";
64                                  data_in3 <= "00001111000011110000111100001111";
65                  wait for 90ns;  enable <= '0';
66                  wait for 10ns;  enable <= '1';
67                                  data_in1 <= "00001111000011110000111100001111";
68                                  data_in2 <= "00001111000011110000111100001111";
69                                  data_in3 <= "11111111000011110000111100001111";
70                  wait for 90ns;  enable <= '0';
71                  wait for 10ns;  enable <= '1';
72                                  data_in1 <= "11001111000011110000111100001111";
73                                  data_in2 <= "00111111000011110000111100001111";
74                                  data_in3 <= "00001111000011110000111100001111";
75              END PROCESS;
76          END ARCHITECTURE;
77
```

```
1        -- Project: TMR-System with three RAMs and Voter
2        -- Entity:  Test for TMR system
3        -- Autor:   Reinhard Zeif
4        -- Date:    Feb. 2011
5        ----------------------------------------------------------
6        LIBRARY IEEE;
7        USE IEEE.STD_LOGIC_1164.ALL;
8        USE IEEE.STD_LOGIC_UNSIGNED.ALL;
9        USE IEEE.NUMERIC_STD.ALL;
10
11       ENTITY test_tmr_datapath IS
12       END ENTITY test_tmr_datapath;
13
14       ARCHITECTURE TestBehaviour OF test_tmr_datapath IS
15
16           COMPONENT tmr_datapath
17               PORT( clk :           in   STD_LOGIC;
18                     ram_enable :    in   STD_LOGIC;
19                     ram_address :   in   INTEGER range 0 to 15;
20                     ram_write_en :  in   STD_LOGIC;
21                     ram_read_en :   in   STD_LOGIC;
22                     vote_enable :   in   STD_LOGIC;
23                     ram_data_in :   in   STD_LOGIC_VECTOR(31 downto 0);
24                     ram_data_out :  out  STD_LOGIC_VECTOR(31 downto 0);
25                     v_wm :          out  STD_LOGIC;
26                     v_dc :          out  STD_LOGIC
27                   );
28           END COMPONENT;
29
30           signal clk :           STD_LOGIC := '0';
31           signal ram_enable :    STD_LOGIC := '0';
32           signal ram_address :   INTEGER range 0 to 15 := 0;
33           signal ram_write_en :  STD_LOGIC := '0';
34           signal ram_read_en :   STD_LOGIC := '0';
35           signal vote_enable :   STD_LOGIC := '0';
36           signal ram_data_in :   STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
37           signal ram_data_out :  STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
38           signal v_wm :          STD_LOGIC := '0';
39           signal v_dc :          STD_LOGIC := '0';
40
41       BEGIN
42           tmr_test: tmr_datapath
43           PORT MAP( clk          => clk,
44                     ram_enable   => ram_enable,
45                     ram_address  => ram_address,
46                     ram_write_en => ram_write_en,
47                     ram_read_en  => ram_read_en,
48                     vote_enable  => vote_enable,
49                     ram_data_in  => ram_data_in,
50                     ram_data_out => ram_data_out,
51                     v_wm         => v_wm,
52                     v_dc         => v_dc
53                   );
54
55           clk_proc: PROCESS
56           BEGIN
57               Wait for 0 ns;    clk <= '0';
58               Wait for 20 ns;   clk <= '1';
59               Wait for 20 ns;   clk <= '0';
60           END PROCESS clk_proc;
61
```

```
62          test_proc: PROCESS
63          BEGIN
64              -- initialize
65              Wait for 100 ns; ram_enable  <= '0'; vote_enable  <= '0';
66                               ram_address <= 1;   ram_write_en <= '0';
67                               ram_read_en <= '0';
68              -- write to ram
69              Wait for 100 ns; ram_enable  <= '1'; vote_enable  <= '0';
70                               ram_address <= 1;   ram_write_en <= '1';
71                               ram_read_en <= '0';
72                               ram_data_in <= "01010101010101010101010101010101";
73              -- read ram over voter
74              Wait for 100 ns; ram_enable  <= '1'; vote_enable  <= '1';
75                               ram_address <= 1;   ram_write_en <= '0';
76                               ram_read_en <= '1'; ram_data_in  <= (others => '0');
77              -- read voter with deactivated ram
78              Wait for 100 ns; ram_enable  <= '0'; vote_enable  <= '1';
79                               ram_address <= 0;   ram_write_en <= '0';
80                               ram_read_en <= '0'; ram_data_in  <= (others => '0');
81          END PROCESS test_proc;
82
83      END ARCHITECTURE TestBehaviour;
84
```

Literaturverzeichnis

[AC05] Athavale, A. ; Christensen, C.: *High-Speed Serial I/O Made Simple - A Designer's Guide with FPGA Applications*. Xilinx Inc., 2005 http://www.xilinx.com/publications/archives/books/serialio.pdf

[Act08] Actel Ltd. (Hrsg.): *Radiation-Tolerant ProASIC3 Low Power Space-Flight Flash FPGAs with Flash*Freeze Technology*. Actel Ltd., 2008. http://www.actel.com/documents/RTPA3_DS.pdf, Abruf: 21.10.2010

[Act10a] Actel Ltd. (Hrsg.): *Radiation-Tolerant ProASIC3 FPGAs Radiation Effects*. Actel Ltd., 2010. http://www.actel.com/documents/RT3P_Rad_Rpt.pdf, Abruf: 23.10.2010

[Act10b] Actel Ltd. (Hrsg.): *Radiation-Tolerant ProASIC3 Low Power Space-Flight FPGA Fabric User's Guide*. Actel Ltd., 2010. http://www.actel.com/documents/RTPA3_UG.pdf, Abruf: 21.10.2010

[Act10c] Actel Ltd. (Hrsg.): *Radiation-Tolerant ProASIC3 Single-Event Latch-Up*. Actel Ltd., 2010. http://www.actel.com/documents/RT3P_SEL_Rpt.pdf, Abruf: 23.10.2010

[Act10d] Actel Ltd. (Hrsg.): *RTAX-S/SL and RTAX-DSP Radiation-Tolerant FPGAs*. Actel Ltd., 2010. http://www.actel.com/documents/RTAXS_DS.pdf, Abruf: 20.10.2010

[Act10e] Actel Ltd. (Hrsg.): *RTSX-SU RadTolerant FPGAs (UMC)*. Actel Ltd., 2010. http://www.actel.com/documents/RTSXSU_DS.pdf, Abruf: 28.10.2010

[Alt10a] Altera Corp. (Hrsg.): *Stratix III Device Handbook*. Altera Corp., 2010. http://www.altera.com/literature/hb/stx3/stratix3_handbook.pdf, Abruf: 04.11.2010

[Alt10b] Altera Corp. (Hrsg.): *Stratix IV Device Handbook*. Altera Corp., 2010. http://www..altera.com/literature/hb/stratix-iv/stratix4_handbook.pdf, Abruf: 04.11.2010

[Ash08] Ashenden, Peter J.: *The designers guide to VHDL*. Elsevier Inc., 2008. – ISBN 978–0–12–088785–9

[Bau05] Baumann, R. (Hrsg.) ; IEEE NSREC Short Course, Seattle, WA (Veranst.): *Single-Event Effects in Advanced CMOS Technology*. 2005

[Caf02] Caffrey, M. (Hrsg.) u. a. ; MAPLD Conference 2002 (Veranst.): *Single-Event Upsets in SRAM FPGAs*. http://klabs.org/richcontent/MAPLDCon02/papers/session_p/p8_caffrey_p.pdf. Version: 2002, Abruf: 18.09.2010

[CH06] Cofer, R. C. ; Harding, B. F.: *Rapid System Prototyping with FPGAs*. Elsevier Ltd., 2006. – ISBN 978–0–7506–7866–7

[Gra02] Graham, P. (Hrsg.) u. a. ; MAPLD Conference 2003 (Veranst.): *Consequences and Categories of SRAM FPGA Configuration SEUs*. http://klabs.org/richcontent/MAPLDCon03/papers/c/c6_graham_p.pdf. Version: 2002, Abruf: 29.09.2010

[Gro08] Grout, I.: *Digital Systems Design with FPGA and CPLD*. Elsevier Ltd., 2008. – ISBN 978–0–7506–8397–5

[Hab02] Habinc, S.: Functional Triple Modular Redundancy (FTMR) / ESA, Gaisler Research. Version: 2002. http://klabs.org/richcontent/fpga_content/DesignNotes/seu_hardening/functional_tmr_fpga_003_01-0-2.pdf, Abruf: 30.09.2010. 2002. – Forschungsbericht

[Her10] Heraeus: *Heraeus Contact Materials - Bonding Produktfolder*. Version: 2010. http://www.heraeus-contactmaterials.com/media/webmedia_local/media/downloads/documentsbw/brochure/HERAEUS_BondingWire_Brochure.pdf, Abruf: 03.08.2010

[JTA08] JTAG: *When does boundary-scan make sense*. Version: 2008. http://www.jtag.de/de/Lernen/Boundary-scan_tutorial/When_does_boundary-scan_make_sense_%28ENG%29?cm_loggedin=1, Abruf: 05.09.2010

[Kil07] Kilts, S.: *Advanced FPGA Design: Architecture, Implementation, and Optimization*. John Wiley and Sons Inc., 2007. – ISBN 978–0–470–05437–6

[Max04] Maxfield, C.: *The Design Warrior's Guide to FPGA*. Elsevier Ltd., 2004. – ISBN 0–7506–7604–3

[Mia07] Miani, A. K.: *Digital Electronics: Principles, Devices and Applications*. John Wiley and Sons Inc., 2007. – ISBN 978–0–470–03214–5

[Qui08] Quicklogic (Hrsg.): *Whitepaper - Reliability of the Amorphous Silicon Antifuses*. Quicklogic, 2008. http://www.quicklogic.com/assets/pdf/white_papers/QLReliabilityoftheAmorphousSiliconAntifuseWPRevA.pdf, Abruf: 15.07.2010

[RL08] Raj, A. A. ; Latha, T.: *VLSI Design*. PHI Learning Ltd., New Dehli, 2008. – ISBN 978–81–203–3431–1

[SF04] Schrimpf, R. D. ; Fleetwood, D. M.: *Radiation Effects and Soft Errors in Integrated Circuits and Electronic Devices*. World Scientific Publishing Co. Pte. Ltd., 2004. – ISBN 981–238–940–7

[Spa01] *Aerospace Mission Failure Analysis for NASA Ames Research Center*. Version: 2001. http://science.ksc.nasa.gov/shuttle/nexgen/Nexgen_Downloads/ROCKET_FAILURES_FailureCauses_Peinemann.pdf, Abruf: 05.12.2010

[Spa02] *Satellite GN&C Anomaly Trends*. Version: 2002. http://klabs.org/DEI/lessons_learned/gsfc_lessons/satellite_anomaly_br.pdf, Abruf: 05.12.2010

[Ste08] Sterpone, L.: *Electronics System Design Techniques for Safety Critical Applications*. Springer Sience and Business Media B. V., 2008. – ISBN 978–1–4020–8978–7

[Str10] *Stratix IV FPGA Family Architecture*. Version: 2010. http://www.altera.com/products/devices/stratix-fpgas/stratix-iv/overview/architecture/stxiv-architecture.html, Abruf: 11.11.2010

[Tin00] Tinder, R. F.: *Engineering Digital Design*. 2nd. Elsevier Ltd., Academic Press, 2000. – ISBN 0–12–691295–5

[TRO03] T. R. Oldham, F. B. M.: Total Ionizing Dose Effects in MOS Oxides and Devices. In: *IEEE TRANSACTIONS ON NUCLEAR SCIENCE* Bd. 50. IEEE, 2003

[Xil00] Xilinx Inc. (Hrsg.): *Correcting Single-Event Upsets Through Virtex Partial Configuration*. Xilinx Inc., 2000. http://www.xilinx.com/support/documentation/application_notes/xapp216.pdf, Abruf: 29.09.2010

[Xil04] Xilinx Inc. (Hrsg.): *Radiation Effects and Mitigation Overview*. Xilinx Inc., 2004. http://www.xilinx.com/esp/mil_aero/collateral/presentations/radiation_effects.pdf, Abruf: 16.09.2010

[Xil06] Xilinx Inc. (Hrsg.): *QPro Virtex-II 1.5V Radiation-Hardened QML Platform FPGAs - Product Specification*. Xilinx Inc., 2006. http://www.xilinx.com/support/documentation/data_sheets/ds124.pdf, Abruf: 05.10.2010

[Xil08a] Xilinx Inc. (Hrsg.): *CoolRunner-II CPLD Family Documentation*. Xilinx Inc., 2008. http://www.xilinx.com/support/documentation/data_sheets/ds090.pdf, Abruf: 15.07.2010

[Xil08b] Xilinx Inc. (Hrsg.): *Single-Event Upset Mitigation Selection Guide*. Xilinx Inc., 2008. http://www.xilinx.com/support/documentation/application_notes/xapp987.pdf, Abruf: 30.09.2010

[Xil08c] Xilinx Inc. (Hrsg.): *Virtex-4 FPGA User Guide*. Xilinx Inc., 2008. http://www.xilinx.com/support/documentation/user_guides/ug070.pdf, Abruf: 05.10.2010

[Xil08d] Xilinx Inc. (Hrsg.): *XtremeDSP for Virtex-4 FPGAs*. Xilinx Inc., 2008. http://www.xilinx.com/support/documentation/user_guides/ug073.pdf, Abruf: 11.10.2010

[Xil09a] Xilinx Inc. (Hrsg.): *Correcting Single-Event Upsets in Virtex-4 FPGA Configuration Memory*. Xilinx Inc., 2009. http://www.xilinx.com/support/documentation/application_notes/xapp1088.pdf, Abruf: 29.09.2010

[Xil09b] Xilinx Inc. (Hrsg.): *Virtex-4 FPGA Configuration User Guide*. Xilinx Inc., 2009. http://www.xilinx.com/support/documentation/user_guides/ug071.pdf, Abruf: 13.10.2010

[Xil09c] Xilinx Inc. (Hrsg.): *Virtex-5 FPGA RocketIO GTX Transceiver User Guide*. Xilinx Inc., 2009. http://www.xilinx.com/support/documentation/user_guides/ug198.pdf, Abruf: 16.10.2010

[Xil09d] Xilinx Inc. (Hrsg.): *Xilinx TMRTool Product Brief*. Xilinx Inc., 2009. http://www.xilinx.com/publications/prod_mktg/XTMRTool_ssht.pdf, Abruf: 30.09.2010

[Xil10a] *Space-Grade Virtex FPGA Product Table*. Version: 2010. http://www.xilinx.com/publications/prod_mktg/virtex5qv-product-table.pdf, Abruf: 05.10.2010

[Xil10b] *Xilinx Corporate Overview*. Version: 2010. http://phx.corporate-ir.net/External.File?item=UGFyZW50SUQ9NTQyODN8Q2hpbGRJRD0tMXxUeXBlPTM=&t=1, Abruf: 02.10.2010

[Xil10c] Xilinx Inc. (Hrsg.): *Partial Reconfiguration User Guide.* Xilinx Inc., 2010. http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_2/ug702.pdf, Abruf: 02.10.2010

[Xil10d] Xilinx Inc. (Hrsg.): *PowerPC Processor Reference Guide.* Xilinx Inc., 2010. http://www.xilinx.com/support/documentation/user_guides/ug011.pdf, Abruf: 11.10.2010

[Xil10e] Xilinx Inc. (Hrsg.): *Radiation-Hardened, Space-Grade Virtex-5QV Device Overview.* Xilinx Inc., 2010. http://www.xilinx.com/support/documentation/data_sheets/ds192_V5QV_Device_Overview.pdf, Abruf: 16.10.2010

[Xil10f] Xilinx Inc. (Hrsg.): *Space-Grade Virtex-4QV Family Overview - Product Specification.* Xilinx Inc., 2010. http://www.xilinx.com/support/documentation/data_sheets/ds653.pdf, Abruf: 07.10.2010

[Xil10g] Xilinx Inc. (Hrsg.): *Virtex-4 FPGA Embedded Tri-Mode Ethernet MAC User Guide.* Xilinx Inc., 2010. http://www.xilinx.com/support/documentation/user_guides/ug074.pdf, Abruf: 13.10.2010

[Xil10h] Xilinx Inc. (Hrsg.): *Virtex-5 FPGA Configuration User Guide.* Xilinx Inc., 2010. http://www.xilinx.com/support/documentation/user_guides/ug191.pdf, Abruf: 16.10.2010

[Xil10i] Xilinx Inc. (Hrsg.): *Virtex-5 FPGA User Guide.* Xilinx Inc., 2010. http://www.xilinx.com/support/documentation/user_guides/ug190.pdf, Abruf: 05.10.2010

[Xil10j] Xilinx Inc. (Hrsg.): *Virtex-5 FPGA XtremeDSP Design Considerations - User Guide.* Xilinx Inc., 2010. http://www.xilinx.com/support/documentation/user_guides/ug193.pdf, Abruf: 16.10.2010