# Dynamisch und partiell rekonfigurierbare Hardwarearchitektur mit adaptivem hardwaregestützten Routing zur Laufzeit

Alexander Thomas

# Dynamisch und partiell rekonfigurierbare Hardwarearchitektur mit adaptivem hardwaregestütztem Routing zur Laufzeit

Zur Erlangung des akademischen Grades eines

**DOKTOR-INGENEURS (Dr.-Ing.)**

an der Fakultät für
Elektrotechnik und Informationstechnik
am Karlsruher Institut für Technologie (KIT)
genehmigte

**DISSERTATION**

von

**Dipl.-Ing. Alexander Thomas**

aus Dshambul/Kasachstan

Tag der mündlichen Prüfung:

27. April 2015

Hauptreferent Prof. Dr.-Ing. Dr. h. c. Jürgen Becker
Korreferent: Prof. Dr.-Ing. Jörg Henkel (Fakultät für Informatik, KIT)

Meiningen, den 13.04.2014

# Zusammenfassung

Die Vorliegende Arbeit befasst sich mit der Entwicklung und Synthese einer rekonfigurierbaren Hardwarearchitektur für dynamische Funktionsmuster, die im Rahmen des DFG-finanzierten Projektes AMURHA im Schwerpunktprogramm 1148 (spprr) für "Rekonfigurierbare Rechensysteme" entstanden ist. Hierbei war die Zielsetzung vor allem, neue und bestehende adaptive Konzepte in einer neuen Hardwarearchitektur, der HoneyComb-Architektur, zu vereinen und die Machbarkeit zu präsentieren. Zu den neuen Features dieser Architektur gehören Multikontextfähigkeiten, multigranulare Datentypen, hexagonale Zellstrukturen, programmierbare Ein-/Ausgabelogik und adaptives Routing zur Laufzeit.

Die Durchführung des Projekts hatte zwei wesentliche Schwerpunkte, die zum einen die Hardwareentwicklung und zum anderen die Softwareentwicklung einschlossen. In vorausgehenden Arbeiten und Studien bestehender rekonfigurierbarer Lösungen, wurden neue Ideen und Konzepte entwickelt und mit bestehenden Techniken kombiniert, um zur neuen Architektur zu gelangen. Daraus entstand ein parametrisierbares VHDL-Modell der neuen Architektur, welches an die Bedürfnisse der Anwendungen angepasst werden kann und auf diese Weise versucht, die Kosten der Rekonfigurationsressourcen zu optimieren.

Die Softwareentwicklung konzentrierte sich auf die Erstellung neuer Werkzeugprogramme, die sowohl die Entwicklung der HoneyComb-Architektur und der darauf laufenden Applikationen, als auch die Überwachung der darauf laufenden Anwendungen ermöglichten. Dazu gehören neben den Hardware-Konfigurationstools auch Compiler zur Übersetzung der Anwendungsbeschreibungen in Maschinencode.

Als Proof-of-Concept des neuen Ansatzes dienten ausgewählte Applikationen, die parallel in diesem Projekt entwickelt wurden. Hierbei war es wichtig, von den neuen Eigenschaften der Architektur zu profitieren und die Machbarkeit aufzuzeigen.

Im Anschluss an dieses Projekt entstand ein Demonstrationschip, der die realisierten Funktionen in realer Hardware umsetzte und die ausgewählten Applikationen ausführen konnte.

# Inhaltsverzeichnis

X

# Danksagung

Die vorliegende Dissertation entstand während meiner Tätigkeit am Institut für Technik der Informationsverarbeitung (ITIV) am Karlsruher Institut für Technologie (KIT). Zum erfolgreichen Abschluss der Arbeit haben viele Menschen direkt und indirekt beigetragen, denen ich an dieser Stelle meinen Dank aussprechen möchte.

Zuallererst möchte ich Herrn Prof. Dr.-Ing. Jürgen Becker meinen besonderen Dank für die Möglichkeit zur Mitarbeit am ITIV, den gelassenen Freiräumen zur wissenschaftlichen Entfaltung und die große Geduld bei der Anfertigung dieser Ausarbeitung aussprechen. Auch gilt mein Dank Herrn Prof. Dr.-Ing. K.-D. Müller-Glaser, der in kollegialer Leitung mit Prof. Dr.-Ing. Jürgen Becker das ITIV leitet und beide gemeinsam die Voraussetzungen für die Verfolgung wissenschaftlicher Tätigkeiten schufen. Mein besonderer Dank gilt auch Herrn Prof. Dr.-Ing. Jörg Henkel für die freundliche Übernahme des Korreferats der vorliegenden Arbeit.

Mein besonderer Dank gilt darüber hinaus auch allen Mitarbeitern des ITIV, die mir stets hilfsbereit und freundlich in wissenschaftlichen, technischen und Verwaltungsfragen zur Seite standen. Insbesondere zu erwähnen sind hier meine Kollegen Carsten Bieser, Jens Becker und Ralf König, die in allen Belangen ein offenes Ohr hatten und stets ein konstruktives wie auch kontroverses wissenschaftliches Gespräch begrüßten.

Ebenso gilt mein Dank den zahlreichen Studenten, die im Laufe meiner Arbeit am ITIV ihre Diplom-, Studien- und Seminararbeiten durchführten und durch ihre Kreativität und Engagement ihren Beitrag zum Gelingen dieser Arbeit leisteten. Michael Rückauer, als ehemaliger Student und späterer Kollege, sei an dieser Stelle insbesondere hervorzuheben. Sein großartiger Beitrag zu dieser Arbeit umfasst sowohl Teile der Softwareentwicklung, wie auch die spätere Durchführung des Layouts des Demonstrationschips.

Mein herzlichster Dank geht an meine Ehefrau Nina und meine Kinder, die durch ihre unendliche Geduld und ihre ausdauernde Unterstützung die vorliegende Arbeit erst ermöglichten. Ebenso gilt mein großer Dank meinen Eltern, die mir diesen akademischen Weg erst durch das vorausgegangene sorgenfreie Studium eröffneten. Auch danke ich meinen Schwiegereltern, Elvira und Helmut Kerl, für ihre stete Befürwortung und Unterstützung.

Zuallerletzt gilt mein großer Dank der Deutschen Forschungsgemeinschaft (DFG), die durch ihre Finanzierung des AMURHA-Projektes diese Arbeit möglich machte.

Meiningen, den 13. April 2014 Alexander Thomas

# Abkürzungen

| | |
|---|---|
| AES | Advanced Encryption Standard |
| ALU | Arithmetic Logic Unit |
| AMC | Adaptive Modulation and Coding |
| AMURHA | Adaptive multigranulare rekonfigurierbare Hardwarearchitektur |
| ANSI | American National Standards Institute |
| API | Application Programming Interface |
| ASIC | Application-Specific Integrated Circuit |
| ASIP | Application-Specific Instruction Set Processor |
| CD | Compact Disc |
| CISC | Complex Instruction Set Computing |
| CMOS | Complementary Metal Oxide Semiconductor |
| CPLD | Complex Programmable Logic Device |
| CPU | Central Processing Unit |
| CRC | Cyclic Redundancy Check |
| CSD | Circuit Switched Data |
| CUDA | Compute Unified Device Architecture |
| DDR | Double Data Rate |
| DECT | Digital Enhanced Cordless Telecommunications |
| DFG | Deutsche Forschungsgemeinschaft |
| DPHC | Datapath-HoneyComb-Cell |
| DSP | Digital Signal Processing |
| DVB | Digital Video Broadcasting |
| DVD | Digital Versatile Disc |
| DWT | Discrete Wavelet Transform |
| EDGE | Enhanced Data Rates for GSM Evolution |
| EPIC | Explicitly Parallel Instruction Computing |

| | |
|---|---|
| FFT | Fast Fourier Transform |
| FPGA | Field-Programmable Gate Array |
| GFLOP | Giga Floating-Point Operations Per Second |
| GMSK | Gaussian Minimum Shift Keying |
| GND | Ground |
| GPGPU | General Purpose Graphics Processing Unit |
| GPP | General Purpose Processor |
| GPRS | General Packet Radio Service |
| GPU | Graphics Processing Unit |
| GSM | Global System for Mobile Communications |
| HCA | HoneyComb-Assembler |
| HCL | HoneyComb-Language |
| HCLUT | HoneyComb-Lookup-Table |
| HCMEM | HomeyComb-Memory |
| HDTV | High-Definition Television |
| HEVC | High Efficiency Video Coding |
| HPC | High-Performance Computing |
| HSDPA | High Speed Downlink Packet Access |
| HSUPA | High Speed Uplink Packet Access |
| IEEE | Institute of Electrical and Electronics Engineers |
| iMDCT | Inversed Modified Discrete Cosine Transform |
| IMEC | Interuniversity Microelectronics Centre |
| IOHC | Input/Output-HoneyComb-Cell |
| ISDN | Integrated Services for Digital Network |
| ITRS | International Technology Roadmap for Semiconductors |
| LPC | Linear Predictive Coding |
| LTE | Long Term Evolution |
| LUT | Lookup Table |
| MEMHC | Memory-HoneyComb-Cell |
| MFLOP | Mega Floating-Point Operations per Second |
| MIMD | Multiple Instruction Multiple Data |
| MIMO | Multiple Input Multiple Output |

| | |
|---|---|
| MIPS | Million Instructions per Second |
| MMX | MultiMedia eXtension |
| MOSFET | Metal-Oxide Semiconductor Field Effect Transistors |
| MPEG | Moving Picture Experts Group |
| NIST | National Institute of Standards and Technology |
| NMOS | N-Type Metal-Oxide-Semiconductor |
| NOP | No Operation |
| NPC | Non-Deterministic Polynomial-Time Complete |
| OFDM | Orthogonal Frequency-Division Multiplexing |
| OpenCL | Open Computing Language |
| OpenMP | Open Multi-Processing |
| PC | Personal Computer |
| PCB | Printed Circuit Board |
| PLL | Phase-locked Loop |
| PMOS | P-Type Metal-Oxide-Semiconductor |
| PSK | Phase-Shift Keying |
| QAM | Quadrature Amplitude Modulation |
| RAM | Random Access Memory |
| RISC | Reduced Instruction Set Computing |
| SDRAM | Synchronous Dynamic Random Access Memory |
| SIMD | Single Instruction Multiple Data |
| SIMD | Single Instruction Multiple Data |
| SISD | Single Instruction Single Data |
| SRAM | Static Random Access Memory |
| TDP | Thermal Design Power |
| UMTS | Universal Mobile Telecommunications System |
| VLIW | Very Long Instruction Word |
| VLSI | Very-Large-Scale-Integration |
| WCDMA | Wideband Code Division Multiple Access |

# 1. Einführung

Im Laufe der letzten Jahrzehnte vollzog sich im Bereich der Halbleitertechnologien eine faszinierende Entwicklung, die für das Design mikroelektronischer Schaltungen viele Möglichkeiten bot, aber auch eine Reihe von Problemen aufwarf. Neben den technischen Herausforderungen ergaben sich auch viele gesellschaftliche Möglichkeiten und Konsequenzen, die bisher bei weitem nicht ausgeschöpft werden konnten und in Zukunft besonderer Aufmerksamkeit bedürfen. An dieser Stelle sei lediglich auf die Herausforderungen im Bereich des Datenschutzes hingewiesen. Diese betreffen sowohl die Rechtslage im Umgang mit digitalen Medien, als auch die anstehenden Veränderungen in der Gesellschaft, ausgelöst durch die ungeahnten Möglichkeiten der neuen Technologien.

## 1.1. Das Mooresche Gesetz

Die rasanten Fortschritte der hochintegrierten Halbleitertechnologien (Very-Large-Scale-Integration, VLSI) wurden bereits 1965 von Intel Corporation [5] Mitbegründer Gordon Moore durch das Mooresche Gesetz [1] formuliert. Seine anfängliche Voraussage lautete, dass die Komplexität von Schaltungen und damit die Anzahl der Schaltungskomponenten jedes Jahr verdoppelt werden wird. Dieses Gesetz wurde allerdings zehn Jahre später (1975) durch den Verfasser korrigiert und auf die Verdoppelung der Komponenten in zwei Jahren reduziert [2]. Trotz der Tatsache, dass Gordon Moore in seinem Gesetz Bezug auf die Schaltungskomplexität nahm, wird heute allgemein die Auffassung vertreten, dass die Anzahl der Transistoren und damit die Leistung einer Schaltung pro Fläche sich alle 18 Monate verdoppeln. Die Aktualisierung der Anzahl der Monate ergab sich aus den Beobachtungen der tatsächlichen Entwicklung der folgenden Jahre nach dem jeweiligen Postulat. Abbildung 1 veranschaulicht die tatsächliche Entwicklung der Transistordichte anhand der Intel Prozessoren. Hierbei zeichnet sich eine Verdopplung der Transistordichte alle 24 Monate ab.

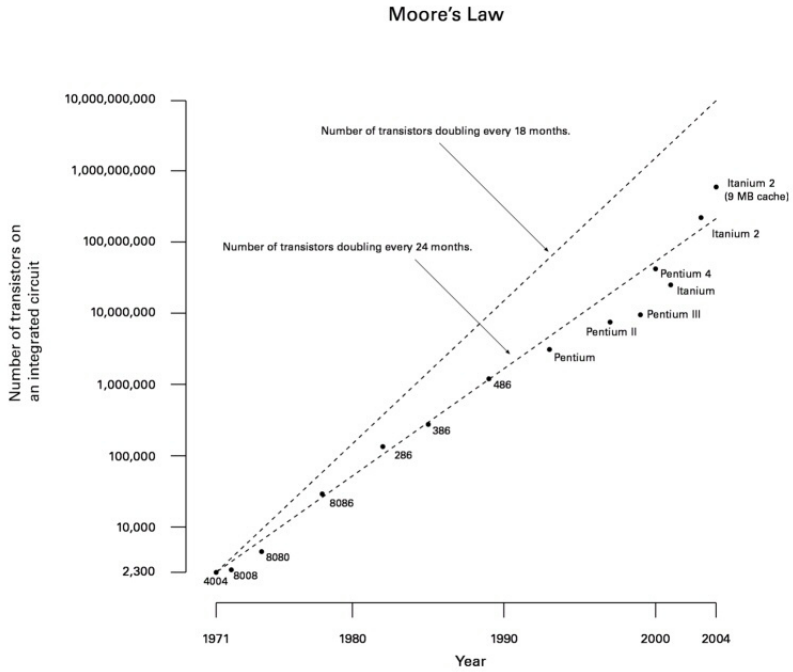



**Abbildung 1: Mooresches Gesetz aus dem Jahr 1975 im Vergleich zur tatsächlichen Entwicklung der Intel Prozessoren [5]**

Das exponentielle Wachstum der Chip-Komplexität, durch die dazu notwendige Verkleinerung der Schaltstrukturen, ermöglichte über Jahre einen beispiellosen Aufschwung der Chip-Industrie. Das Prinzip der kontinuierlichen Komplexitätserhöhung bietet sich geradezu an, wirtschaftlich genutzt zu werden, in dem in regelmäßigen zeitlichen Abständen neuere und leistungsfähigere Schaltkreise zum gleichen oder gar günstigeren Preis gefertigt werden können. Der Kunde erhält wiederkehrend den Anreiz neuere Geräte zu erwerben, da mit jeder Generation der Geräte neue Funktionen und höhere Leistung angeboten werden können. Durch das Mooresche Gesetz haben sich darüber hinaus Marktzyklen etabliert, die eingehalten werden müssen, um konkurrenzfähig zu bleiben. Schafft es ein Halbleiterhersteller nicht rechtzeitig die aktuelle Technologiestufe auszunutzen, so drohen ihm technologischer Rückstand und finanzielle Einbußen, die sich ebenfalls auf die Einführung der nächsten Technologiestufe auswirken können. Die Halbleiterwirtschaft orientiert sich dabei an der International Technology Roadmap for Semiconductors (ITRS)

[3][4], die regelmäßig eine Prognose für die zukünftigen Entwicklungen herausgibt. Diese Prognose wird jährlich von einem Gremium der Halbleiterhersteller aktualisiert und lehnt sich dabei im Wesentlichen an das Mooresche Gesetz an. Das Mooresche Gesetz darf in diesem Zusammenhang weniger als ein Gesetz verstanden werden, vielmehr als eine Richtlinie, die eine wirtschaftliche Ausschöpfung erlaubt.

## 1.2. Veranschaulichung der Miniaturisierung

Die Miniaturisierung der Schaltungsstrukturen erfolgt in Stufen, die in etwa die Verdopplung der Schaltungsdichte pro Flächeneinheit nach sich zieht. Die Zeit bis zur Einführung der nächsten Technologiestufe orientiert sich am Mooreschen Gesetz und erfolgt in etwa alle 24 Monate. Dieser Zeitraum scheint für den Markt ein genügend großes Fenster zu bieten, um eine erfolgreiche kommerzielle Ausbeute der aktuellen Technologien zu ermöglichen und darüber hinaus genügend finanzielle Mittel zu erwirtschaften, um die Fortführung der folgenden Zyklen zu finanzieren.

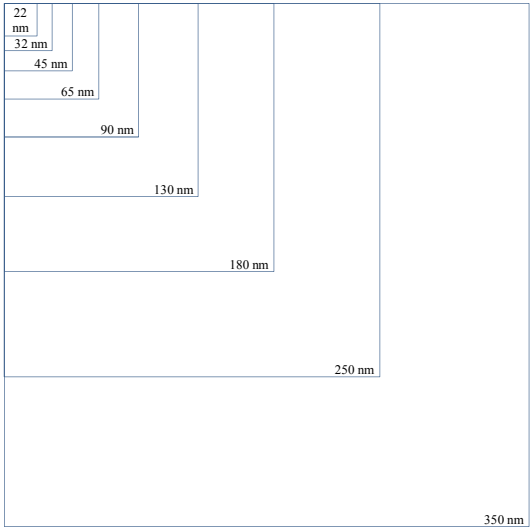



**Abbildung 2: Generationsübergreifende Darstellung der funktionalen und technologischen Dichtezunahmen mikroelektronischer Schaltungen**

Die Abgrenzung der Technologien erfolgt durch die Angabe der kleinsten Gate-Länge der verwendeten Transistoren. Hierdurch wird zudem das Auflösungsvermögen der verwendeten Technologien charakterisiert. Aktuelle Her-

stellungsprozesse bieten eine Gate-Länge von 22 nm und werden im Jahr 2014 durch 16 nm [5] abgelöst. Fasst man die Gate-Längen als Seitenlängen von Quadraten auf, so entsprechen die Flächen der Quadrate den Dichten der angesetzten Technologien. Setzt man diese Flächen ins Verhältnis zu einander, erhält man unter der Annahme, dass die übrigen Strukturen auf dem Silizium in gleichem Maße miniaturisiert werden, Faktoren für die Steigerung der Schaltungsdichte beim Übergang von einer Technologie zur nächsten. Abbildung 2 veranschaulicht diesen Sachverhalt graphisch und gibt eine Idee für die Größenverhältnisse während der Integrationssteigerung. Jedes Quadrat beherbergt somit die gleiche Anzahl der Transistoren. Obwohl die Verhältnisse der Gate-Längen im Schnitt einen Wert von etwa 1,41 liefern, ergibt sich durch das Quadrieren für das Verhältnis der Flächen ein ungefährer Faktor von zwei. Tabelle 1 fasst den dargelegten Zusammenhang für die letzten Jahre zusammen.

**Tabelle 1 Flächenzuwachs der VLSI Technologien**

| Jahr | Gate-Länge (nm) | Verhältnis der Gate-Längen | Gate-Länge im Quadrat (nm²) | Verhältnis |
|---|---|---|---|---|
| **1996** | 350 | - | 122500 | - |
| **1998** | 250 | 1,40 | 62500 | 1,96 |
| **2000** | 180 | 1,39 | 32400 | 1,92 |
| **2002** | 130 | 1,38 | 16900 | 1,97 |
| **2004** | 90 | 1,44 | 8100 | 2,09 |
| **2006** | 65 | 1,38 | 4225 | 1,92 |
| **2008** | 45 | 1,44 | 2025 | 2,09 |
| **2010** | 32 | 1,41 | 1024 | 1,98 |
| **2012** | 22 | 1,45 | 484 | 2,12 |

Diese Flächenverhältnisse geben jedoch nur eine relative Zunahme der Transistordichte an. Meist ist es in einem Design nicht möglich diese Dichte auszunutzen, da beispielsweise durch die Zunahme der Komplexität einer Schaltung auch mehr Verbindungen benötigt werden. Wird eine Schaltung nicht flach genug platziert, so führt dies zu einem überproportionalen Anstieg der Leitungsdichte und begrenzt somit die maximal erreichbare Dichte der Logikgatter. Diesem Problem begegnet man durch die kontinuierliche Erhöhung der Anzahl der Metal-Layer (derzeit sind 10 Metall-Layer Stand der Technik), wodurch dichteres Routing ermöglicht wird. Trotzdem sind die Herausforderungen, die maximale Dichte zu erreichen, enorm und ohne manuelle Platzierung und Verdrahtung wie im Falle des Full-Custom Designs nicht zu erreichen.

Beim Standardzellendesign ergibt sich durch die vorgegebenen Strukturen vordefinierter Gatter, der zeilenweisen Platzierung und der vereinfachten Verdrahtung durch horizontale und vertikale Leitungen vom Prinzip her ein Nachteil gegenüber einem Full-Custom-Design. Dieser Nachteil äußert sich in der Fläche, der Verlustleistung und der resultierenden Performance des Designs und wird in der Regel mindestens mit dem Faktor zwei beziffert [6]. Allerdings wird durch diesen Nachteil auf der anderen Seite eine bessere Unterstützung des computergestützten Entwurfs erkauft. Der Weg von funktionaler Beschreibung bis zum fertigen Chip-Design wird durch Synthese-, Layout- und Analyse-Tools geebnet. Der Designer kann sich auf diese Weise in seiner Arbeit verstärkt auf die zu realisierenden Funktionen konzentrieren und die Details des Chip-Layouts den Tools überlassen. Hierdurch werden Entwicklungskosten eingespart und Time-to-Market verkürzt.

Die große Zahl integrierbarer Transistoren und der daraus resultierender Gatter pro Flächeneinheit bietet ein großes Potential. So lassen sich bis zu 561 kgates/mm² bei einem 90 nm Standardzellen Prozess der Firma TSMC integrieren [7]. Als Referenz-Gatter wird hierbei ein NAND-Gatter-Äquivalent mit vier Transistoren angenommen. Nächste Stufe, der TSMC 65nm Prozess, liefert bei realen Anwendungen, hier G200 GPU der nVidia Corporation [8], eine Transistordichte von 1,4 Milliarden Transistoren auf einer Fläche von 576 mm², während die GPU eine maximale Verlustleistung von 236 Watt besitzt. Umgerechnet auf eine Fläche von 1 mm² ergibt das eine Dichte von 2,4 Millionen Transistoren bzw. 607,6 kgates. Der Spezifikation zufolge lassen sich beispielsweise beim TCBN65LP Prozess bis zu 854 kgates/mm² integrieren. Für das obige Beispiel bedeutet das eine Dichteauslastung von rund 71,1%. Die Zahlen verdeutlichen die Schwierigkeiten, die von Entwicklern überwunden werden müssen, um effiziente Designs zu realisieren.

## 1.3. Leistungsanforderungen moderner und zukünftiger Anwendungen

Moderne Anwendungen und Algorithmen haben in Folge des Mooreschen Gesetzes einen stetig wachsenden Bedarf an Rechenleistung. Dieser Umstand ist kein Zufall, sondern wird von Entwicklern der Algorithmen angesichts des erwarteten Rechenleistungsanstiegs in kommenden Gerätegenerationen eingeplant. Oftmals dauerte die Phase, bis die verfügbare Hardware die Anforderungen der Applikationen erfüllte, mehrere Jahre. Insbesondere die prozessorbasierten Rechensysteme holten meist einige Jahre später auf, so dass die Entwickler zunächst auf ASIC- oder DSP-Lösungen setzen mussten. Dieser Umstand zog höhere Kosten nach sich. Anhand von zwei Anwendungsbereichen soll im Folgenden gezeigt werden, wie die historische Entwicklung bis heute

ablief, um auf zukünftige Fortschritte und die Anforderungen zu schließen. Die ausgewählten Anwendungen gehören zum einen zu dem Bereich der Multimediaapplikationen und zum anderen zum Bereich der Telekommunikation.

### 1.3.1. Multimediaapplikationen

Im Bereich der Video-Kompression lassen sich die evolutionäre Entwicklung und der steigende Leistungsbedarf besonders gut darstellen. Die bekannten Video Standards der Motion Picture Experts Group [15] (MPEG-1, MPEG-2 und MPEG-4) repräsentieren hierbei eine zeitliche Entwicklung, die insbesondere auch die Leistungsfähigkeit und die Entwicklung der Rechensysteme damaliger Zeit wiedergeben. Der erste Standard aus dieser Reihe, MPEG-1 [16], wurde im Jahre 1991 eingeführt. Dies geschah zu einer Zeit, als die Leistungsfähigkeit der Personal Computer im Bereich von 20 MIPS (Intel 486er Prozessoren) lag. Compact Disc (CD) war Stand der Technik und somit die erste Wahl als Datenträger, was auch durch die gegebene Speicherkapazität von 650 MB die maximale Dateigröße auf diesem Medium und die mittlere Datenrate von 1,5 Mbit/s definierte. Neben der Videokompression bot bereits auch der MPEG-1 Standard Audiokodierungsverfahren. Diese boten unterschiedliche Qualitätsstufen und Komplexitäten für unterschiedliche Technologien, wobei der prominenteste Vertreter unter ihnen das MPEG1 Layer 3 Verfahren ist und heute besser bekannt ist als MP3.

**Tabelle 2 Übersicht der gängigsten Video/Audio-Standards**

| | MPEG-1 (1991) | MPEG-2 (1994) | MPEG-4 (2000) | H.261 (1993) | H.263 (1995) | H.264/ MPEG-4 AVC (2002) |
|---|---|---|---|---|---|---|
| **Datenraten** | 1,5 MBit/s | bis zu 50 MBit/s | | | | |
| **Transform** | 8x8 DCT | 8x8 DCT | 8x8 DCT | 8x8 DCT | 8x8 DCT | 4x4 |
| **MC Block Size** | 16x16 | 16x16, 8x16 | 8x8, 16x16 | 16x16 | 8x8, 16x16 | 16x16, 16x8, 8x8, 8x4, 4x4 |
| **MC Accuracy** | | | | 1-pel | | 1/8-pel |
| **Additional Motion Prediction Modes** | - B-Frames | - B-Frames - Interlace | - B-Frames - Interlace - GMC - SPRITE Coding | - | - B-Frames | - B-Frames - Long term frame memory - in-loop deblocking filter- CAVLC/CABAC |

Die eingeschränkte Datenrate von 1,5 Mbit/s bei voller PAL bzw. NTSC Auflösung (max. 720x576) machte MPEG-1 unattraktiv, da die daraus resultie-

rende Qualität bereits in absehbarer Zeit nicht mehr zeitgemäß sein würde. Bereits drei Jahre später (1994) wurde aus diesem Grund der MPEG-2 Standard [17] eingeführt. Ähnlich zu seinem Vorgänger enthält dieser Standard eine Reihe von Spezifikationen zum Transport, Multiplexing sowie Video- und Audiokodierung. Durch den Einsatz von MPEG-2 in DVDs (Digital Versatile Disc) und diversen DVB-Varianten (Digital Video Broadcasting) für die Übertragung von Fernsehbildern wurde der Siegeszug dieses Standards eingeläutet und ist noch heute kaum wegzudenken. Möglich wurde dies durch variable Datenraten für den Datenstrom, Modifikationen in der Bewegungserkennungstechnik (Motion Compensation), Unterstützung des Zeilensprungverfahrens und möglicher Auflösungen bis hin zu HDTV mit 1920x1080 Punkten. Das letztere wurde im Nachhinein in den Standard aufgenommen, um der Entwicklung des Fernsehsektors zu folgen. Zum Zeitpunkt der Einführung des Standards jedoch hätte noch kein PC (Pentium 60/66) eine DVD abspielen können, da die Leistungsfähigkeit nicht ausreichte. Erst einige Jahre später (ab 1998) mit der Verfügbarkeit von PCs mit mindestens 400MHz (Pentium II/III) mit Multimedia Extension Unterstützung (MMX) war das Abspielen dieser Medien möglich geworden.

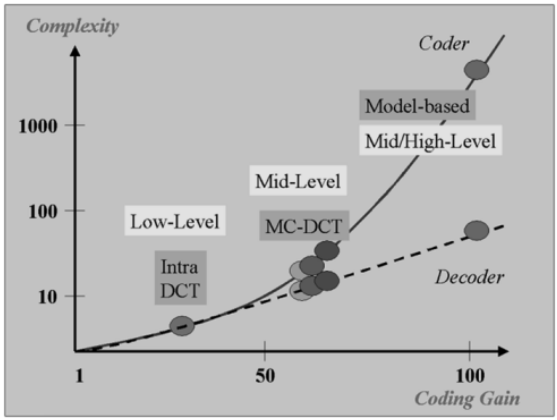



**Abbildung 3: Komplexitätszunahme und steigender Rechenleistungsbedarf der modernen Video-Kompressionsverfahren**

Im Jahre 2000 wurde der Folgestandard MPEG-4 [18] zur Reduktion der mittleren Datenrate bei gleichbleibender Qualität im Vergleich zu Vorgängern eingeführt (siehe Tabelle 2). Hierfür wurden im Wesentlichen komplexere Algorithmen zur Bewegungserkennung entwickelt, die eine enorme Qualitätssteigerung und Absenkung der durchschnittlichen Datenraten erlaubten. Bereits in dieser Form war dieser Standard eine Herausforderung für Rechenar-

chitekturen des Erscheinungsjahres (Pentium III, 1GHz). Durch die Ergänzung des Standards um die H.264 Eigenschaften im Jahre 2002, waren die aktuellen Architekturen bei weitem nicht in der Lage die H.264-kodierten Videos abzuspielen. Selbst heute (2012) hat eine CPU mit einem Kern mit 3 GHz Probleme einen H.264-Datenstrom in voller HDTV Auflösung ohne zusätzliche Beschleuniger-Hardware zu bewältigen. Variable Datenraten und daraus resultierender schwankender Performance-Bedarf liefern Spitzen bei der CPU-Auslastung, sodass meist nach einer kurzen Laufzeit des Videos die Bild- und Audiodekodierung nicht mehr synchron ablaufen. Zum flüssigen Genuss des Videoabends sollte auf jeden Fall eine Dual-Core CPU mit 3GHz verwendet werden.

Aktuell wird am Nachfolger des H.264-Verfahrens, dem H.265 oder High Efficiency Video Coding (HEVC) [19] genannt, gearbeitet. Basierend auf neuen Entropiecodierungstechniken soll dieses Verfahren eine 50 Prozent höhere Effizienz erreichen als der Vorgänger. Die Fertigstellung ist für Ende 2013 geplant. Es ist davon auszugehen, dass der Rechenbedarf für dieses Verfahren einen weiteren exponentiellen Anstieg erfährt, wie dies bei bisherigen Neuentwicklungen der Fall war.

Abbildung 4 verdeutlicht quantitativ den Rechenleistungsbedarf in Abhängigkeit der Qualität und des Kompressionsgewinns der verwendeten Transformationsalgorithmen, wie sie in den MPEG-Standards verwendet werden [33][34][45]. Unabhängig von diesen Algorithmen kommen noch Verfahren zur Bewegungskompensation, die einen großen Anteil an der resultierenden Qualität und ebenso einen großen Anteil an der Komplexität der Video-Standards haben. Durch den Einsatz von Video-Codecs im mobilen Bereich, werden Themen wie resultierende Verlustleistung verstärkt diskutiert und weiterentwickelt.

### 1.3.2. Telekommunikation

Die drahtlose sowie mobile Kommunikation im kommerziellen Bereich ist im Laufe der Zeit zu einem allgemeinen Lebensstandard geworden und aus unserem Lebensalltag kaum noch wegzudenken. Zur Realisierung mobiler Kommunikation wurden in letzten Jahrzehnten in Abhängigkeit der Anforderungen unterschiedliche Techniken entwickelt, um den Bedarf zu decken. Die wohl populärste Technologie stellen die schnurlosen Telefonsysteme dar, wobei sich selbst dieses Feld in unterschiedliche Bereiche einordnen lässt. Zum einen finden sich hier die älteren analogen, sowie modernere DECT (Digital Enhanced Cordless Telecommunications) Telefone zur schnurlosen Kommunikation über kurze Distanzen von unter 100 Metern, was zumeist innerhalb geschlossener Räume passiert. Zum anderen sind hier die mobilen

Geräte für unterwegs zu finden, angefangen bei einfachsten mobilen Telefonen bis hin zu Smartphones, die Zugriff auf Internet, Social Networks und eMails bieten. Hier sollen insbesondere mobilen Geräte für unterwegs betrachtet werden.

Angefangen hat die Entwicklung des Mobilfunks in Deutschland bereits 1958 mit analogen Verfahren mit der Einführung des A-Netzes. Die Weiterentwicklung folgte 1972 mit der Einführung des B-Netzes und schließlich 1986 durch die Einführung des C-Netzes. Bei allen diesen Verfahren handelte es sich ausnahmslos um analoge Verfahren mit stark begrenzter Teilnehmerzahl. Der wichtigste Unterschied zwischen den Netztypen liegt in der späteren Unterteilung in zellulären Aufbau, wodurch sich die Teilnehmerzahl von zuvor 27.000 im B-Netz auf nun 800.000 Teilnehmer im C-Netz erhöhen ließ. Zusammenfassend lassen sich die rein analogen A- und B-Netze zur Generation G0 einordnen, wohingegen das C-Netz bereits zur Generation G1 zählt. Die Generation G1 zeichnet sich insbesondere dadurch aus, dass die Verwaltung des Netzes bereits digital geschah, die Sprache allerdings weiterhin analog und frequenzmoduliert übertragen wurde. In diesem Kontext kann die Generation G1 als hybride Lösung bezeichnet werden.

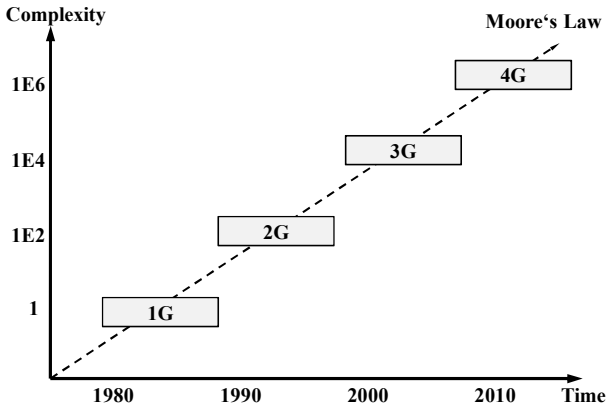



**Abbildung 4: Rechenleistungsbedarf der Mobilfunkgenerationen folgen den Voraussagen des Mooreschen Gesetzes**

Die ab 1992 in Deutschland folgenden Mobilfunknetze, wie das D-Netz und das E-Netz, basierten auf dem damals neu eingeführten GSM-Standard (Global System for Mobile Communications) [20] und waren vollständig digital. Der Übertragungsstandard des GSM verwendet mehrere Verfahren die zur Komprimierung der Sprache als auch zur Modulation auf dem Kanal dienen. Obwohl die Netto-Datenraten im Gegensatz zu ISDN (64.000 Bits/Sekunde) vergleichsweise niedrig sind (6.5 kbit/s bei halber Datenrate bzw. 13 kbit/s bei

voller Datenrate), wurde dennoch durch den Einsatz von speziellen Sprach-Codecs (wie dem linear predictive coding, LPC) eine moderate Sprachqualität erreicht. Im Laufe der Jahre wurden weitere Sprach-Codecs für unterschiedliche Datenraten spezifiziert und im GSM-Standard verankert. Vor der Modulation der Nutzdaten werden eine Reihe von Maßnahmen durchgeführt, um die Robustheit der Daten gegenüber Kanalfehlern zu erhöhen. Hierzu gehören Verfahren wie Bit-Interleaving, CRC-Prüfsummenbildung und Redundanzerhöhung durch Faltung. Für die senderseitige Modulation wird das GMSK (Gaussian Minimum Shift Keying) verwendet, welches 1 Bit pro Symbol auf dem Kanal überträgt. Auf der Empfängerseite wird ein Viterbi-Dekoder genutzt, um aus dem empfangenen Signal die Nutzdaten zu gewinnen. Das GSM lässt sich ebenfalls zur Übertragung von Daten nutzen. Die hierfür verwendete Technik wird CSD (Circuit Switched Data) genannt und ermöglicht Netto-Datenraten von 9600 Baud. Dabei wird eine festzugeordnete Verbindung aufgebaut und bis zur Terminierung aufrechterhalten.

Bedingt durch den Siegeszug des Internets wurde das GSM evolutionär weiterentwickelt und für die Datenvermittlung erweitert. Techniken wie GPRS (General Packet Radio Service) und EDGE (Enhanced Data Rates for GSM Evolution) gehören zu diesen Erweiterungen. GPRS baut im Wesentlichen auf derselben Technologie auf wie GSM, führt jedoch einen paketvermittelnden Dienst hinzu. Anstelle einer festen Verbindung wird eine virtuelle Verbindung für die Datenübertragung genutzt. Werden keine Daten übertragen, kann der Kanal von anderen Netzteilnehmern genutzt werden. Weiterhin können bis zu acht Zeit-Slots im GSM-Kanal verwendet werden, um eine höhere Bandbreite von bis zu 171,2 kbit/s zu erhalten. Im Falle von EDGE wird die Modulation mit GMSK um eine 8-PSK Modulation erweitert, sodass maximal 473 kbit/s erreicht werden können. Zusammenfassend lassen sich GSM und GSM-basierte Techniken der zweiten Generation (G2) zuordnen und stellen in diesem Kontext die erste digitale Generation dar. EDGE wird gelegentlich als die Generation 2,5 angeführt und bildet somit den Übergang zur folgenden dritten Generation.

Die dritte Generation des Mobilfunks wurde im Jahr 2004 in Deutschland in Form von UMTS (Universal Mobile Telecommunications System) eingeführt. Bei der Einführung des G3-Mobilfunks wurden weltweit unterschiedliche Ansätze gewählt. Es sollte erreicht werden, dass die Mobilfunkanbieter möglichst viele Komponenten des alten G2 Netzes nutzen konnten, um das neue Netz aufzubauen. So wurde in Schwellenländern wie Brasilien oder Argentinien das bewährte EDGE als G3 eingeführt. Westliche Staaten haben entweder reines UMTS oder eine Mischung aus beiden gewählt.

Das UMTS ist ein Mobilfunkstandard auf Basis von WCDMA (Wideband Code Division Multiple Access) [21] und arbeitet paketorientiert. Basierend

auf orthogonalen Spreiz-Codes [22] werden die Nutzerdaten gemeinsam auf einem Frequenzband übertragen. Mittels der Korrelation des Datenstroms mit den nutzerspezifischen Spreiz-Codes können auf der Empfängerseite die Daten der Netzteilnehmer getrennt werden. Diese Vorgehensweise erhöht die Effizienz des Kanals erheblich. Durch die hohe Bandbreite der UMTS-Kanäle von 5MHz (200kHz bei GSM), fällt der Verschnitt am Rande der Kanäle im Verhältnis zur verfügbaren Bandbreite deutlich geringer aus. Weiterhin können die erforderlichen Datenbandbreiten wesentlich flexibler den Nutzern zugeordnet werden. Die erforderliche Rechenleistung der Endgeräte zur Realisierung der UMTS-Kommunikation ist im Vergleich zu GSM/EDGE deutlich angestiegen und kann qualitativ der Abbildung 5 entnommen werden.

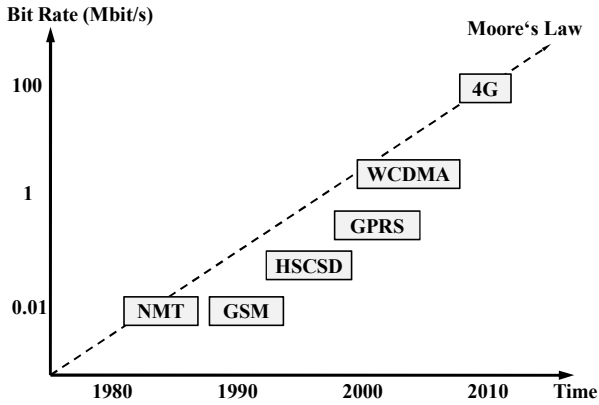



**Abbildung 5: Rechenleistungsbedarf der Mobilfunkgenerationen folgen den Voraussagen des Mooreschen Gesetzes**

Was anfänglich mit reiner Sprachübertragung und späterer Datenübermittlung bei GSM/GPRS begann, ist mit UMTS zu einem multimedialen Netzwerk angewachsen, welches neben Sprach-/Video-Kommunikation auch breitbandigen Internetzugang (bis zu 384 kbit/s im Downlink) und Fernsehen (DVB-H) bietet. Trotz der höheren Performance von UMTS wurde interessanterweise die maximale Sendeleistung von UMTS (0,25 W) im Vergleich zu GSM (2W) gesenkt, was neben der Spreiztechnik auch auf höhere Rechenleistung der Endgeräte und auf die technischen Fortschritte im RF-Frontend-Bereich zurückzuführen ist.

Auch UMTS hat im Laufe der Zeit Erweiterungen erfahren, die in Standards wie HSDPA (High Speed Downlink Packet Access) oder HSUPA (High Speed Uplink Packet Access) endeten. Hierbei handelt es sich um richtungsabhängige Erweiterungen von UMTS, die insbesondere auf die Steigerung der

Datenraten und niedrigere Latenzen abzielten. Mittels HSDPA sind im Idealfall bis zu 14,0 Mbit/s im Downlink erreichbar, während HSUPA bis zu 5,8 Mbit/s im Uplink ermöglicht. Erreicht werden diese Geschwindigkeitssteigerungen durch geänderte Modulationsverfahren und Verwaltungsstrategien. Insbesondere die adaptive Modulation (AMC) in Kombination mit 16-QAM bringt eine deutliche Leistungssteigerung.

Während die dritte Generation des Mobilfunkstandards (UMTS) langsam an ihre Grenzen stößt und ausgereizt ist, wird bereits seit 2010 die nächste Generation (G3.9) deutschlandweit eingeführt. Hierbei handelt es sich um Long Term Evolution (LTE) [25][29]. Dieses Übertragungsverfahren basiert auf der Orthogonal Frequency-Division Multiplexing Technologie (OFDM). Dabei wird eine zuvor festgelegte Anzahl an orthogonalen Unterträgern verwendet, so dass sich dieses Verfahren relativ leicht an die unterschiedlichen Frequenzbandbreiten anpassen lässt. Weiterhin wird durch den Einsatz der Multiple-Input-Multiple-Output-Technik (MIMO), d.h. durch den Einsatz mehrerer Antennen, eine weitere Robustheit auf dem fehleranfälligen Medium Luft erzielt. Das Resultat liefert Übertragungsraten von bis zu 300 Mbit/s im Downlink und bis zu 75 Mbit/s im Uplink. Weiterhin lassen sich sehr niedrige Latenzen erzielen, so dass eine Reihe neuer Anwendungen (IP/Video-Telefonie, Online-Gaming), für den mobilen Einsatz erschlossen werden und LTE somit eine echte Alternative für den Festnetzanschluss insbesondere auf dem Land wird.

Zum echten G3 Nachfolger wird das Advanced LTE [23][24] gezählt. Diese Technik soll nach aktuellem Stand über 1000 Mbit/s erreichen können und 2013 in Deutschland auf den Markt kommen. Die Kompatibilität zum LTE soll erhalten bleiben und selbst die gleichen Frequenzbänder sollen für beide Techniken parallel nutzbar sein.

An den Abbildungen Abbildung 4 und Abbildung 5 ist es unschwer zu erkennen, dass die Komplexitätszunahme sowie die Entwicklung der Datenraten im Mobilfunk von Generation zu Generation im Grunde dem Mooreschen Gesetz folgen. Ebenso wie bei Video-Codecs nimmt der Performance-Bedarf über die Jahre exponentiell zu und stellt die Hardware/Software-Entwickler bei jeder Neuentwicklung vor neue Herausforderungen. Es ist ebenfalls zu erwarten, dass diese Entwicklung im selben Maße weitergeht und die Herausforderungen im Entwicklungsbereich weiterhin bestehen werden [30][31]. Weitere Informationen zu Telekommunikationstechniken können [44] entnommen werden.

## 1.4. Vergleich der Datenverarbeitungstechniken

Das enorme Potential, welches durch die sehr große Zahl der Transistoren pro Flächeneinheit geboten wird, blieb in den letzten Jahren teilweise ungenutzt. Erkennbar ist dieser Umstand an den Leistungsunterschieden zwischen Prozessoren (CPUs) und applikations-spezifischen integrierten Schaltungen (ASICs). Die Gründe dafür sind sowohl in Hardware- als auch in Software-Aspekten zu suchen. Der wohl wichtigste Grund ist die geforderte Flexibilität zur Datenverarbeitung. Moderne Systeme setzen Prozessoren oder µController zur Berechnung von Algorithmen und Steuerungsaufgaben ein. Eine Alternative dazu wären ASICs, die in ihrer Funktion allerdings stark auf die vorgesehenen Applikationen ausgerichtet sind. Während Prozessoren auf einen vordefinierten Befehlssatz setzen und die Algorithmen flexibel in die Zeit abbilden, parallelisieren ASICs die Algorithmen in die Fläche. Im Falle der ASICs sind die Algorithmen allerdings in der mikroelektronischen Struktur kodiert und können nach der Fertigung nicht mehr verändert werden. Ein wichtiger Vorteil von ASICs gegenüber Prozessoren ist jedoch die Effizienz bzgl. Performanz, Verlustleistung und Fläche, siehe Abbildung 6 [32][35]. Die höheren Kosten von ASICs zahlen sich also aus, da dadurch höhere Performanz und ggf. längere Akkulaufzeiten erzielt werden. Hierbei kann die Leistungsfähigkeit der ASICs als ein Maß für die erreichbare Effizienz der zugrundeliegenden Halbleitertechnik angesehen werden.

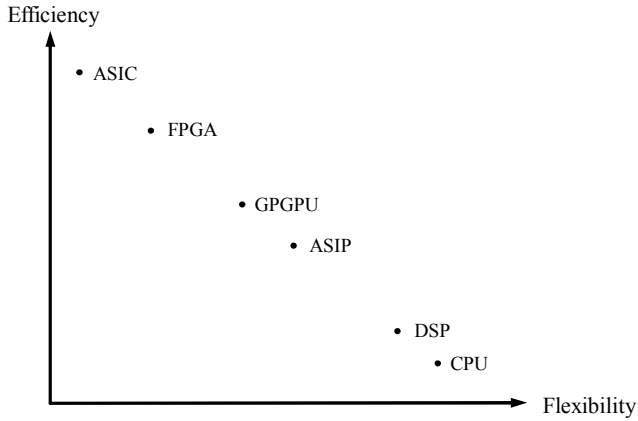



**Abbildung 6: Effizienz von unterschiedlichen Architektur-Ansätzen zur Datenverarbeitung**

Neben ASICs und CPUs lassen sich auch andere Zielplattformen zur Applikationsausführung nutzen: FPGAs, (GP-)GPUs, ASIPs oder auch DSPs.

Alle diese Ansätze weisen jeweils eine unterschiedliche Effizienz auf und sind in der Regel zwischen ASICs und CPUs eingeordnet. DSPs sind Hochleistungsprozessoren, die speziell zur digitalen Signalverarbeitung entwickelt und optimiert werden. Ihre Überlegenheit liegt insbesondere darin begründet, parallele Ausführung mehrerer Operationen in einem Takt zu unterstützen. Diese Operationen haben einen spezialisierten Charakter, die stark anwendungsabhängig sind. Ein weiteres wichtiges Merkmal der DSPs liegt in ihrer Unterstützung der Fließkomma- und Festkommaformate, was auch gleichzeitig eine Klassifizierung der DSPs darstellt. DSPs sind in ihrer Applikationsdomäne den CPUs weit überlegen, weisen jedoch eine geringere Flexibilität auf. Zu prominenten Vertretern der DSP-Familie gehören die Architekturen der Firma Texas Instruments [26].

ASIPs (Prozessoren mit applikationsspezifischem Befehlssatz) stellen eine weitere Klasse der Architekturen zur Datenverarbeitung dar. Es handelt sich hierbei um spezialisierte Prozessoren, die einen anwendungsspezifischen Befehlssatz besitzen. Dadurch sind diese Bausteine im Vergleich zu DSPs noch weiter eingeschränkt, haben jedoch zusätzlichen Vorteil bzgl. der Effizienz.

Eine weitere Klasse in dieser Übersicht stellen die GPUs (Graphics Processing Units) bzw. GPGPUs (General Purpose GPUs) dar. Anfänglich als reine Graphikbeschleuniger entwickelt und vermarktet, haben sie sich durch die Entwicklung der letzten Jahre zum allgemein nutzbaren Rechenkernen entwickelt. Sie eignen sich insbesondere für den High Performance Computing Bereich (HPC) für wissenschaftliche und technische Berechnungen, aber auch als Multimedia-Beschleuniger in Standard-PCs. Die massive Parallelität der GPUs, Unterstützung von Fließkomma-Formaten einfacher und doppelter Genauigkeit in Kombination mit großzügig bemessener Bandbreite der Speicheranbindung und einer praktikablen Methoden zur Programmierung lassen diese Architekturen in Leistungsbereiche vorstoßen, die bisher von keinen prozessor-basierten Ansätzen erreicht werden konnten. Bekannte GPU-Hersteller sind nVidia Corporation [8] und AMD Corporation [9].

Zuletzt werden hier die Field Programmable Gate Arrays (FPGAs) betrachtet. Diese repräsentieren eine ganze Reihe rekonfigurierbarer Architekturen. Es handelt sich hierbei um Bausteine, die auf einem niedrigen Level strukturell programmierbar sind. In diesem Zusammenhang wird von Rekonfiguration gesprochen. Es lassen sich boolesche Funktionen auf die Logikelemente der FPGAs abbilden und die aktiven Logikelemente mittels eines Kommunikationsnetzwerkes miteinander verbinden. Dieser Sachverhalt geht soweit, dass selbst physikalische Eigenschaften des Bausteins bzgl. der Signallaufzeit und des daraus resultierenden Timings im Design-Flow betrachtet werden müssen. Das Resultat ist eine digitale Schaltung, die sehr nahe an die Eigenschaften eines ASICs herankommt. Der Overhead der Rekonfiguration erzeugt jedoch

unweigerlich eine Lücke zwischen diesen beiden Ansätzen. Diese kann durchaus bis zum Faktor 30 betragen. Auf der Kostenseite steht allerdings bei kleinen Stückzahlen noch ein größerer Faktor zu Gunsten von FPGAs. Zu Herstellern von FPGAs gehören neben den beiden dominanten Marktteilnehmern wie Xilinx Corp. [10] und Altera Corp. [11] auch Hersteller mit deutlich kleineren Marktanteilen wie Lattice Semiconductor Corp. [12], Atmel Corp. [13] und Actel Corp. [14].

## 1.5. Rekonfigurierbare Systeme

Die Anwendungsbeispiele in den letzten Abschnitten haben deutlich die Herausforderungen der Entwickler aufgezeigt. Jede neue Algorithmen-Generation bringt neue Anforderungen mit sich, die zunächst mit flexiblen Ansätzen wie Prozessoren oder DSPs nicht zu bewältigen sind. Im ersten Schritt müssen zunächst neue ASIC-Lösungen entwickelt werden, die im Stande sind, die notwendige Leistung zu bringen. Hierbei ist es wichtig, die verfügbaren Ressourcen der zugrunde liegenden Technologie effizient auszunutzen. Der Nachteil der ASICs, nur im begrenzten Umfang und soweit im Vorab eingeplant Flexibilität zu bieten, besteht weiterhin und bedeutet ein Kostenrisiko für den Hersteller. Sollte im Design nach der Herstellung ein Fehler entdeckt werden, so lässt sich dieser Fehler nur nach einem Redesign und erneuter Fertigung beheben. Dieser Vorgang ist mit zusätzlichen Kosten verbunden und garantiert nicht, dass die nächste Revision die Qualitätskontrollen zu vollster Zufrieden passiert. Dieser Herausforderung begegnen die Hersteller durch aufwändige Verifikationsprozesse, die einen großen Anteil an den resultierenden Entwicklungskosten bedeuten. Eine weitere Gefahr droht dem Hersteller, wenn er durch die Notwendigkeit zum Redesign, das Zeitfenster für den rechtzeitigen Markteintritt verpasst.

Ansätze wie die FPGAs bieten hier eine erhöhte Flexibilität durch Rekonfigurierbarkeit und nutzen die Ressourcen relativ effizient aus. Damit bieten diese Bausteine eine Alternative zu ASICs. Allerdings ist der feingranulare Charakter der FPGAs zugleich ein Hindernis, da viele Ressourcen benötigt werden, um die feingranulare Konfigurierbarkeit zu ermöglichen. Dies hat zur Folge, dass die notwendige Chip-Fläche für identische Funktion beim FPGA deutlich höher ausfällt als beim ASIC und damit die erwartete Performance sowie Verlustleistung proportional zur Fläche nachteilig ausfallen. Eine Alternative zur feingranular rekonfigurierbaren Architektur stellen die grobgranular rekonfigurierbaren Ansätze dar. Doch während FPGAs bereits eine akzeptierte Technologie auf dem Markt sind, stecken die grobgranular rekonfigurierbaren Architekturen immer noch in ihren Anfängen. Zum großen Teil handelt es sich bei diesen Ansätzen um einen Gegenstand der Forschung und nur vereinzelt

gehen Hersteller ernsthaft der Entwicklung grobgranularer Architekturen nach. Prinzip bedingt ist jedoch zu erwarten, dass die Effizienz in diesem Umfeld gegenüber den FPGAs deutlich zunimmt, so dass viele Forschungseinrichtungen sich intensiv mit der Materie auseinandersetzen.

## 1.6. Das AMURHA Projekt

Das Projekt AMURHA (Adaptive multigranulare rekonfigurierbare Hardwarearchitektur für dynamische Funktionsmuster) das den Schwerpunkt der vorliegenden Arbeit darstellt, hatte zum Ziel, die Effizienz der Datenverarbeitung im Vergleich zu CPUs/DSPs oder gar FPGAs zu verbessern. Die verfügbaren Ressourcen auf dem Silizium sollten effizienter ausgenutzt werden, was zu einer kleineren Fläche und verbesserter Verlustleistung führen würde. Weiterhin sollte die Programmierbarkeit der resultierenden array-basierten Architektur verbessert und der praktische Nutzen erhöht werden. Die Laufzeit des Projektes war auf sechs Jahre angesetzt und lief vom Zeitraum Juni 2003 bis Mai 2009. Die anschließenden Route-and-Place-Tätigkeiten (P&R) für den angestrebten Prototyp wurden zwischen Oktober 2009 bis Februar 2011 durchgeführt und mit dem Tape-Out abgeschlossen. Die Produktion und das Packaging wurden im Juni 2011 abgeschlossen und die Prototypen ausgeliefert. Die Produktion erfolgte in 90 nm TSMC Standardzellen Technologie. Die Finanzierung des Projektes und die anschließende Protypenfabrikation erfolgten durch die Deutsche Forschungsgemeinschaft (DFG) [27] im Rahmen des Schwerpunktprogramms 1148 für rekonfigurierbare Rechensysteme (spprr) [28].

Um die gesetzten Ziele in diesem Projekt zu erreichen, wurde eine neuartige array-basierte adaptive dynamisch-rekonfigurierbare Architektur entwickelt. Die notwendigen Tools für die Programmierung und das Debugging waren ebenfalls Bestandteil des Projekts. Die resultierende Hardware-Architektur, mit dem Namen HoneyComb, hat einen multi-granularen Ansatz zur Datenverarbeitung und eine Reihe neuer und teils bei anderen Architekturen eingesetzter Techniken in einem Ansatz umgesetzt. Hierzu gehören Techniken wie dynamische und partielle Rekonfiguration, adaptives dynamisches Routing zu Laufzeit, Multi-Kontext-Fähigkeiten, multi-granulares Routing, grob- und fein-granulare Rechenstrukturen. Die genutzten Techniken erlaubten ebenfalls höher stehende Funktionen, wie beispielsweise die Konfiguration von fehlertoleranten Berechnungsstrukturen.

Da Flexibilität in der Hardware immer zusätzliche Kosten nach sich zieht, wurden Techniken entwickelt, um diese Kosten bei Bedarf zu reduzieren. So erlaubt es die HoneyComb-Architektur die unbenutzten Ressourcen und somit Hardware-Kosten zu reduzieren. Dies geschieht aus der Kenntnis der Anwen-

dungen heraus. Soll die resultierende Architektur nur eine vorgegebene Menge an Anwendungen unterstützen, so lassen sich die ungenutzten Schaltstrukturen aus der Architektur extrahieren und damit die Chip-Fläche reduzieren. Dieses Vorgehen führt zur Verringerung der Flexibilität bei gleichzeitig reduzierten Kosten.

## 1.7. Aufbau der Arbeit

Die vorliegende Arbeit ist in sechs Kapitel eingeteilt. Im ersten Kapitel wird die Motivation für dieses Projekt dargelegt und insbesondere ausgehend von anwendungsseitiger Anforderung die Notwendigkeit neuer Ansätze herausgestellt.

Das darauffolgende Kapitel präsentiert Grundlagen moderner Rechensysteme, die notwendig sind, um die Konzepte in dieser Ausarbeitung zu verstehen. Neben Zahlendarstellungen und arithmetischer Operationen werden Techniken zur Realisierung moderner Prozessoren als auch Grundlagen rekonfigurierbarer Architekturen vorgestellt. Zusätzlich werden die Abhängigkeiten moderner CMOS-Technik bezüglich ihrer Leistungsfähigkeit und Verlustleistung erläutert.

Kapitel drei stellt den aktuellen Stand der Technik vor. Dabei wird in groben Zügen auf den Stand klassischer Architekturen eingegangen und die Eigenschaften von ASIC-Designs vorgestellt. Der Hauptteil dieses Kapitels widmet sich den rekonfigurierbaren Architekturen sowohl kommerzieller als auch akademischer Herkunft.

Das vierte Kapitel beschreibt die HoneyComb-Architektur und ihre Hardwarekonzepte. Es wird detailliert auf den Aufbau eingegangen und die genutzten Techniken bis hin zu realisierten Zustandsmaschinen erläutert. Darüber hinaus wird der Template-Charakter der Architektur dargelegt, der die Architektur an die Bedürfnisse der Zielanwendungen anpassen kann.

Das vorletzte Kapitel widmet sich den Beispielanwendungen, die als Proof-of-Concept im AMURHA Projekt ausgesucht und umgesetzt wurden. Hierbei wurden vier Algorithmen auf die HoneyComb-Architektur abgebildet und die Funktionstüchtigkeit der Architektur demonstriert.

Das letzte Kapitel beinhaltet die Zusammenfassung, Verbesserungsvorschläge und das Fazit dieser Arbeit. Mit einem Schlusswort wird diese Ausarbeitung abgeschlossen.

# 2. Grundlagen der Datenverarbeitung

Moderne Prozessoren und anwendungsspezifische integrierte Schaltkreise (ASICs) folgen in ihrem Aufbau ähnlichen Regeln und Methoden. Vorteile hinsichtlich Leistungsfähigkeit, Verlustleistung und Platzbedarf der ASICs liegen insbesondere in der Tatsache begründet, dass die Implementierungen einen spezialisierten anwendungsabhängigen Charakter besitzen und somit keinen zusätzlichen Mehraufwand zum Erreichen der gewünschten Flexibilität benötigen. Dies resultiert in einem bzgl. der Fläche sehr dichten und sowohl das Timing als auch Verlustleistung betreffend effizienten Schaltkreis. Abgesehen von diesen konzeptionellen Unterschieden werden technologisch gesehen identische Komponenten (Technologieprimitive) zur Realisierung der Schaltungen in Form logischer wie auch arithmetischer Operationen verwendet, um die eigentlichen Applikationen umzusetzen. Rekonfigurierbare Architekturen besitzen dagegen einen Aufbau, der Eigenschaften beider Welten vereinen soll. Es wird versucht die Flexibilität der Prozessoren mit der Leistungsfähigkeit der ASICs zu verbinden bei gleichzeitiger Senkung der Verlustleistung. Während Prozessoren Anwendungen in die Zeit abbilden und zu diesem Zweck nur wenige Ausführungseinheiten besitzen, die der Reihe nach Operationen ausführen, nutzen ASICs das Potenzial der Mikroelektronik aus, indem Anwendungen in die Fläche abgebildet werden und somit die Parallelität ausnutzen. Die Zusammenführung beider Ansätze gestaltet sich nicht immer einfach, was insbesondere für grobgranulare rekonfigurierbare Architekturen zutrifft.

In diesem Kapitel werden Grundlagen und Techniken erläutert, die notwendig sind, um folgende Kapitel dieser Ausarbeitung zu verstehen. Beginnend mit der Vorstellung der klassischen Ansätze werden in groben Zügen Konzepte moderner Rechensysteme erläutert. Vergleiche zwischen Prozessor- und ASIC-Implementierungen werden verwendet, um das Verständnis für die Eigenschaften und Vorzüge jeweiliger Ansätze zu erleichtern und insbesondere Voraussetzungen für das Verständnis rekonfigurierbarer Architekturen zu schaffen, deren Grundlagen im zweiten Teil dieses Kapitels vorgestellt werden. Im letzten Teil dieses Kapitels werden Techniken zur Abbildung von Algorithmen vorgestellt. Hierbei wird die Perspektive des Programmierers auf die jeweiligen Ansätze verdeutlicht und das Verständnis weiter vertieft.

Für das Verständnis der vorgestellten Materie werden Kenntnisse der Digitaltechnik [41] vorausgesetzt. Leser, die mit Grundlagen der Datenverarbei-

tung und Rechnerarchitekturen vertraut sind, können dieses Kapitel überspringen.

## 2.1. Grundlagen digitaler Schaltungen

Beim Entwurf digitaler Schaltungen sind mehrere Parameter von Bedeutung, die einen Einfluss auf die Effizienz resultierender Systeme haben. Als erstes wäre hier die eingesetzte Halbleitertechnologie zu nennen, die ausschlaggebend die Fläche, Schaltgeschwindigkeit als auch die Herstellungskosten beeinflussen. Grundelemente mikroelektronischer Schaltungen stellen die Transitoren dar, insbesondere die lateralen MOSFETs (Metal-Oxide Semiconductor Field Effect Transistors) [43][47]. Basierend auf der verwendeten Technologie lassen sich mehrere Schaltungstechniken klassifizieren. Als die populärste Technik ist hier die CMOS-Technologie (Complementary Metal Oxide Semiconductor) zu nennen, die zwei komplementären Schaltnetze benötigt und die Eigenschaft besitzt, sehr stromsparend zu arbeiten. Die nächsten Pendants zu CMOS sind die NMOS-Logik, wie auch die PMOS-Logik.

Bei beiden Techniken wird jeweils ein Netz im Vergleich zu CMOS durch einen Widerstand realisiert, so dass bei der NMOS-Logik die Verbindung zur Core-Spannung ($V_{DD}$) und bei der PMOS-Logik die Verbindung zur Masse (GND) durch einen Widerstand realisiert wird. Beide Techniken haben Vor- als auch auch Nachteile. Die NMOS-Logik ist in der Lage schneller zu schalten, während die PMOS-Logik billiger in der Herstellung ist. Wird jedoch ein schnelles System mit großer Anzahl an Schaltelementen entworfen, so ist CMOS die erste Wahl. Während alleine die Flächenkosten dieser Schaltungstechnik nahezu das Doppelte der NMOS- wie auch PMOS-Logik betragen, sind die Eigenschaften hinsichtlich niedriger Verlustleistung und Schaltgeschwindigkeit zu bevorzugen. Im Folgenden soll die CMOS-Technik, wie auch Aufbau und ihre Eigenschaften in Kürze vorgestellt werden.

### 2.1.1. CMOS Technik

Moderne digitale Schaltkreise werden unter Verwendung der CMOS-Technologie [42] realisiert. Diese Technologie basiert auf der Idee zwei komplementäre Schaltkreise derart zu verschalten, dass immer nur ein Pfad entweder zur Spannung ($V_{DD}$) oder Masse (GND) niederohmig ist und den Ausgang der gesamten Schaltung treibt. In diesem Fall wird von „pull-up“ oder „pull-down“-Pfaden gesprochen. Die Realisierung der pull-up-Schaltung erfolgt mittels selbstleitender p-MOSFET-Transistoren, während die pull-down-Schaltungen mittels selbstsperrender n-MOSFET-Transistoren umgesetzt werden. p-MOSFET-Transistoren haben die Eigenschaft bei angelegter Spannung

$V_{DD}$ zu sperren. n-MOSFET-Transistoren haben die entgegengesetzte Eigenschaft, in dem sie bei angelegter Spannung öffnen, andernfalls sperren. Diese Eigenschaften lassen sich zur Realisierung von Logikschaltungen nutzen, wobei beispielsweise $V_{DD}$ als logische 1 und GND als logische 0 angesehen werden. In diesem Fall wird von positiver Logik gesprochen. Abbildung 7 stellt eine Schaltung zur Realisierung logischer Negation (NOT) dar. Bei angelegter 1 (Spannung = $V_{DD}$) wird eine 0 (Spannung = Gnd) ausgegeben, auf der anderen Seite führt das Anlegen einer 0 zur Ausgabe einer 1. Die resultierende Schaltung ist ein Nicht-Gatter (NOT Gate), das zum Aufbau digitaler Schaltungen genutzt werden kann.

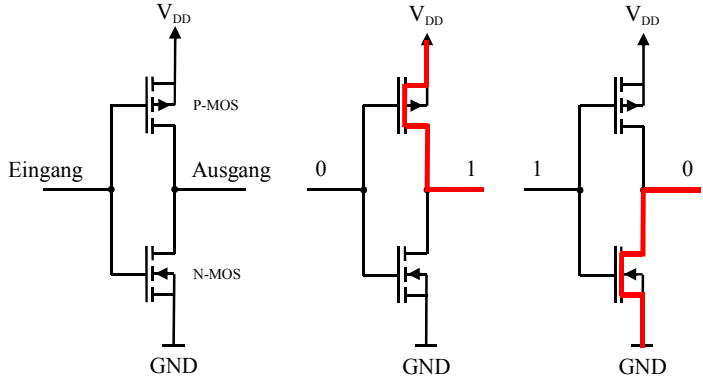



**Abbildung 7 NOT-Gatter in CMOS-Technik und möglicher Schaltzustände**

In ähnlicher Weise lassen sich komplexere Gatter entwerfen und Funktionen wie NAND (negierte Konjunktion) oder NOR (negierte Disjunktion) realisieren, wie in Abbildung 8 veranschaulicht. Insbesondere sei auf den komplementären Aufbau beider Gatter in Bezug auf komplementäre Netze (pull-up- und pull-down-Pfade) innerhalb der Gatter hingewiesen. Während das pull-down-Schaltnetz bestehend aus zwei n-MOSFET-Transitoren beim NAND-Gatter seriell geschaltet ist, ist das zugehörige pull-up-Schaltnetz bestehend aus zwei p-MOSFET-Transistoren parallel realisiert. Die Bildungsvorschrift für das pull-down-Schaltnetz ergibt sich direkt aus der zu realisierenden Funktion und entspricht in diesem Fall direkt der NAND-Operation:

$$F_{pull-down} = AB$$

Im Falle, dass an A und B $V_{DD}$ anliegt, öffnet das pull-down-Schaltnetz und der Ausgang wird auf GND gezogen. Die Bildungsvorschrift für das komplementäre pull-up-Schaltnetz ergibt sich aus der Negation der pull-down-Funktion:

$$F_{pull-up} = \overline{A\bar{B}} = \bar{A} + \bar{B}$$

Es gilt also eine Dualität zwischen diesen beiden Funktionen zu realisieren, damit sie sich beim Schalten gegenseitig ausschließen. Für weitere Infos bzgl. CMOS-Schaltungstechnik sei an dieser Stelle auf weiterführende Literatur verwiesen [42][43][47].

### 2.1.2. Verlustleistung in CMOS-Schaltungen

CMOS-Schaltungen haben den entscheidenden Vorteil, dass im aktiven Zustand entweder nur das pull-up-Schaltnetz oder das pull-down-Schaltnetz leitend ist. Damit sind im stabilen Zustand nahezu alle Kurzschlussströme eliminiert. Bei Schaltvorgängen, insbesondere beim Wechsel von einem Zustand in den anderen, treten kurzzeitig Kurzschlussströme auf und leisten einen Beitrag zur sogenannten dynamischen Verlustleistung. In diesem Zusammenhang spricht man von interner Verlustleistung ($P_{internal}$), die einen technologieabhängigen Charakter besitzt. Die dynamische Verlustleistung ist direkt proportional zur Taktrate der digitalen Schaltung, da nur dadurch Schaltvorgänge ausgelöst werden.

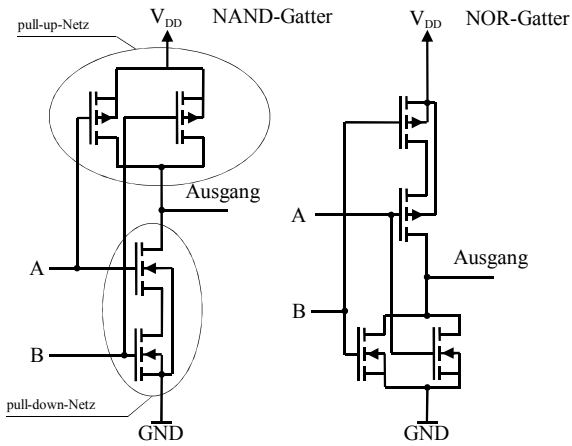



**Abbildung 8 NAND und NOR-Gatter realisiert mit CMOS-Technik**

Eine weitere Komponente der dynamischen Verlustleistung leisten die Schaltströme oder Umladeströme. Hierbei handelt es sich um Umladevorgänge an Kapazitäten in einer CMOS-Schaltung. Einen Beitrag dazu leisten Eingangskapazitäten von Gattern bzw. Transistoren aber auch Leitungen im

Schaltkreis, die je nach Länge und Verlauf unterschiedliche Kapazitäten und Widerstände besitzen. Clock-Bäume in digitalen Schaltungen leisten ebenfalls einen erheblichen Beitrag, die sich in größeren Designs mit einem Anteil von bis zu 40% bemerkbar machen können. Der Anteil der Verlustleistung, der von den Umladevorgängen abhängig ist, wird Schaltverlustleistung $P_{switching}$ genannt.

Einen zusätzlichen Beitrag zur gesamten Verlustleistung eines CMOS-Bausteins leistet die statische Verlustleistung ($P_{static}$). Sie setzt sich zusammen aus Leckströmen an Sperrflächen von Dioden und Transistoren. Mit der fortschreitenden Integrationsdichte der Halbleitertechnik werden die Silizium-Oxyd-Isolatoren ebenfalls verkleinert und mit jeder Generation dünner, so dass die statische Verlustleistung mit dem Fortschreiten der Technologie einen in Relation zur dynamischen Verlustleistung stetig höheren Stellenwert gewinnt. Für die Gesamtverlustleistung ergibt sich zusammenfassend:

$$P_{total} = P_{dynamic} + P_{static}$$
$$= P_{internal} + P_{switching} + P_{static}$$
$$= E_{internal} \cdot f + C_{load} \cdot V_{DD}^2 \cdot f + P_{static}$$

Wie bereits erwähnt, ist der Koeffizient $E_{internal}$ ein technologieabhängiger Parameter und setzt sich aus allen benutzten Gattern der verwendeten Technologie zusammen. Hersteller wie TSMC [7] charakterisieren diese Komponenten zur Entwicklungszeit und geben die Parameter für jedes Gatter explizit an. Zur Berechnung interner Verlustleistung müssen die einzelnen Anteile aufsummiert werden. Die Multiplikation mit der Arbeitsfrequenz ergibt die interne Verlustleistung. Ähnlich verhält es sich mit der statischen Verlustleistung, die für jedes Gatter spezifiziert ist. Sie bildet jedoch einen konstanten Anteil an der gesamten Verlustleistung und ist von der Arbeitsfrequenz unabhängig.

### 2.1.3. Stromspartechniken

Wie im letzten Abschnitt beschrieben, verursacht der Betrieb mikroelektronischer Schaltungen sowohl statische, als auch dynamische Verlustleistungen. Zur Reduktion der Verlustleistung existieren mehrere Techniken, die mit unterschiedlichem Aufwand realisiert werden können. Üblicherweise erfordern die im Folgenden beschriebenen Techniken eine zusätzliche Kontrolle seitens der Anwendung. Dies kann geschehen, indem die Steuerung in Abhängigkeit der Auslastung der Schaltung erfolgt. Diese Steuerung kann sowohl in Hardware umgesetzt werden, als auch durch Software erfolgen.

Am einfachsten kann die dynamische Verlustleistung reduziert werden. Dieser Anteil ist frequenzabhängig und steigt linear mit der Arbeitsfrequenz an. Zur Senkung der Verlustleistung empfiehlt es sich daher die Arbeitsfrequenz möglichst niedrig zu halten. Genau hier profitieren hoch-parallele ASIC-Implementierungen. Bei niedrigen Arbeitsfrequenzen lässt sich zudem die Spannung bei CMOS-Schaltungen senken, solange die technologiespezifische Mindestspannung nicht unterschritten wird. Da die Versorgungsspannung $V_{DD}$ quadratisch in die dynamische Verlustleistung eingeht, ist die erwartete Einsparung relativ hoch. In modernen Prozessoren werden Stromspartechniken eingesetzt, die nur bei Bedarf die Arbeitsfrequenzen erhöhen und im Leerlauf wieder senken, siehe Intels Speed-Step-Technologie [48]. In diesem Zusammenhang spricht man von Frequency Scaling.

Zusätzlich zur Senkung der Arbeitsfrequenz lassen sich auch komplette Module anhalten, indem das betreffende Modul mit dem Takt nicht mehr versorgt wird. In diesem Fall wird von Clock-Gating gesprochen. Diese Technik lässt sich bei ASICs nutzen, die Leerlaufphasen besitzen und in dieser Zeit keine Verlustleistung verursachen sollen. Aber auch bei mobilen Prozessoren wird diese Technik eingesetzt, wie dies beim ARM Cortex A9 Prozessor [49] der Fall ist.

Eine weitere Möglichkeit die dynamische Verlustleistung zu senken, insbesondere bei kombinatorischen Schaltnetzen, besteht in der Anwendung des Signal-Gating. Dies ist eine Technik zur Reduktion der Schaltaktivitäten beispielsweise bei komplexen Schaltnetzen, wie dies bei arithmetischen Operationen der Fall ist. Hier werden am Eingang der betreffenden Schaltung die Eingangssignale mittels AND-Gatter maskiert, so dass innerhalb der Schaltung keine Signalwechsel passieren können. Besitzen die Eingangssignale des maskierten Moduls eine große Zahl an internen Verbrauchern, so entkoppelt das Signal-Gating die resultierende Kapazität der Verbraucher vom externen Treiber.

Neben der dynamischen Verlustleistung trägt auch die statische Verlustleistung ihren Teil zur gesamten Verlustleistung bei. Im Wesentlichen wird die statische Verlustleistung durch Leckströme der miniaturisierten Schaltstrukturen bestimmt und wird durch die Versorgungsspannung getrieben. Die einzige Möglichkeit die statische Verlustleistung zu senken, liegt in der Möglichkeit die Versorgungsspannung zu senken oder gar abzuschalten. Zu diesem Zweck bieten Hersteller von Standardzellenbibliotheken spezielle Zellen zur Steuerung der Versorgungsspannung. Damit können Chip-Designer Spannungsdomänen definieren und physikalisch abgrenzen, um verschiedene Spannungen den Schaltungen oder Schaltungsteilen zuzuführen. Neben der Option die Spannung zu senken, lassen sich damit auch Schaltungen vollständig abschalten, was durch Absenkung der Spannung auf 0V erfolgt. Im letzten Fall lässt

sich die statische Verlustleistung einer Schaltung bei Bedarf vollständig eliminieren. Vom Senken der Versorgungsspannung profitiert auch die dynamische Verlustleistung. Die Technik zur Reduktion der Versorgungsspannung $V_{DD}$ wird Voltage Scaling [50] genannt. In Kombination mit dem dynamischen Anpassen der Arbeitsfrequenz lassen sich optimale Ergebnisse erreichen.

## 2.2. Zahlenformate in digitalen Systemen

Zur Realisierung der Datenverarbeitung müssen digitale Systeme vordefinierte Datentypen unterstützen, die in der Lage sind vom Benutzer vorgegebene Daten aufzunehmen und in Anwendungen zu verarbeiten. Dazu gehören neben Formaten für vorzeichenbehaftete und vorzeichenlose Ganzzahlen auch Dezimalzahlen als Substitut der Teilmengen für die natürlichen Zahlen N, ganze Zahlen Z und die reellen Zahlen R. Durch die endliche Genauigkeit der vektorbasierten Zahlendarstellungen in digitalen Systemen können nur Teilmengen der erwähnten Zahlenmengen dargestellt werden. Für die reellen Zahlen sind zwei gängige Formate definiert, die unterschiedliche Genauigkeiten und Komplexitäten für die Schaltungstechnik bieten. Nachfolgend werden aus technischer Sicht relevante Definitionen und Interpretationen der gängigen Datentypen vorgestellt. Für vollständige Definitionen wird hier ausdrücklich auf Fachliteratur [41][51] verwiesen.

### 2.2.1. Vorzeichenlose Ganzzahlen

Zur Darstellung von vorzeichenlosen Ganzzahlen wird direkt die duale Interpretation von Bitvektoren herangezogen. Dabei wird ähnlich wie im vertrauten Dezimalsystem, die rechte duale Stelle als der duale Wert mit niedrigster Wertigkeit angenommen, während der duale Wert ganz links die höchste Wertigkeit besitzt. Jeder duale Wert wird repräsentiert durch einen Bitvektor mit stellenabhängiger Wertigkeit:

$$b_{n-1}b_{n-2}\dots b_1 b_0 \qquad \text{mit } b_i \text{ Bit der Wertigkeit } i$$

Der Wert W, der durch einen dualen Vektor dargestellt wird, wird berechnet wie folgt:

$$W = \sum_{i=0}^{n-1} b_i 2^i$$

Am Beispiel des vorgegebenen Bitvektors „10010101“ ergibt das:

$$W = 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 149$$

Die kleinste Breite n eines dualen Vektors wird üblicherweise mit 8 Bits gewählt, was einem Byte entspricht. Die nächstgrößeren Bitvektoren in Rechenmaschinen entsprechen der doppelten Breite der vorangehenden Größe, was für die ersten vier Breiten der Reihe 8, 16, 32 und 64 Bits entspricht. Folgende Tabelle gibt eine Übersicht der darstellbaren Wertebereiche und verwendeten Datentypennamen am Beispiel der vier gängigen Bitvektorbreiten, wie sie in der Programmiersprache C nach dem ANSI-Standard [60] verwendet werden:

**Tabelle 3 Wertebereiche vorzeichenloser Datentypen nach ANSI C**

| Bitbreite | Wertebereich | Datentyp in C |
|---|---|---|
| **8** | 0 – 255 | unsigned char |
| **16** | 0 – 65.535 | unsigned short |
| **32** | 0 – 4.294.967.295 | unsigned long |
| **64** | $0 – 2^{64}-1$ | unsigned long long |

In Spezialanwendungen wie ASICs können beliebige Bitvektorgrößen gewählt werden, die der speziellen Anwendung genügen, Damit lässt sich die Schaltkreiskomplexität gemäß den Anforderungen optimieren.

### 2.2.2. Vorzeichenbehaftete Ganzzahlen

Zur Darstellung vorzeichenbehafteter Ganzzahlen wird üblicherweise die Zweier-Komplement-Darstellung (K2-Darstellung) verwendet. Sie hat den Vorteil, dass die Operationen Addition und Subtraktion in gleicher Weise angewendet werden können, wie bei vorzeichenlosen Bitvektoren. Die Anwendbarkeit der Addition und Subtraktion ergibt sich aus der erhaltenen Linearität der Wertigkeiten dieser Darstellung im Vergleich zur vorzeichenlosen Repräsentation. Es müssen allerdings die Über- bzw. Unterläufe nach den Operationen explizit durch Vorzeichenbetrachtung mit zusätzlichem Schaltungsaufwand erkannt werden.

Die positiven und negativen Zahlen in K2-Darstellung können leicht erkannt werden, indem das höchstwertigste Bit des Bitvektors betrachtet wird. Ist der Wert dieses Bits „0“, so handelt es sich um eine positive Zahl, andernfalls handelt es sich um eine negative Zahl. Bei einem Bitvektor mit n Bits werden abzüglich des Vorzeichenbits lediglich n-1 Bits genutzt, um die Wertigkeit auszudrücken. Damit ergibt sich der maximal darstellbare positive Wert mit $2^{n-1}-1$. Der folgende Wert im Bitvektor entspricht in der K2-Darstellung dem kleinsten negativen Wert, wohingegen der höchste duale Wert im Bitvektor eine -1 in der K2-Darstellung repräsentiert. Die Null wird in diesem Fall mit positivem Vorzeichen dargestellt, sodass der Bereich negativer

Zahlen immer einen Wert mehr besitzen als die positiven. Am Beispiel eines 8-Bit Vektors ergibt sich folgender Zusammenhang:

**Tabelle 4 Wertigkeiten der vorzeichenlosen und vorzeichenbehaften Darstellung**

| Bitvektor | 00000000 | 01111111 | 10000000 | 11111111 |
|---|---|---|---|---|
| **Vorzeichenlos** | 0 | 127 | 128 | 255 |
| **Vorzeichenbehaftet** | 0 | 127 | -128 | -1 |

Die Bildung aller positiven Werte in der K2-Darstellung erfolgt analog zur vorzeichenlosen Repräsentation. Die negativen Werte müssen jedoch durch folgende Rechenvorschrift gebildet werden:

$$W_{neg} = \overline{W_{pos}} + 1 = \overline{b_{n-1}b_{n-2}\dots b_1 b_0} + 1$$

Zunächst muss der Betrag des negativen Wertes in positiver Darstellung gebildet werden. Der gebildete Wert wird darauf hin bit-weise invertiert und anschließend eine 1 hinzuaddiert. Das Ergebnis ist die Repräsentation des gewünschten negativen Wertes in K2-Darstellung. Ein gegebener negativer Bitvektor in K2-Darstellung kann mit der gleichen Rechenvorschrift ebenso in die positive Darstellung überführt werden:

$$W_{pos} = \overline{W_{neg}} + 1 = \overline{b_{n-1}b_{n-2}\dots b_1 b_0} + 1$$

Das Ergebnis ist der Betrag des ursprünglichen negativen Wertes. Nachfolgende Tabelle gibt eine Übersicht der darstellbaren Wertebereiche und verwendeten Datentypennamen am Beispiel der vier gängigen Bitvektorbreiten, wie sie in ANSI C verwendet werden:

**Tabelle 5 Wertebereiche vorzeichenbehafteter Datentypen nach ANSI C**

| Bitbreite | Wertebereich | Datentyp in C |
|---|---|---|
| **8** | -128 – 127 | char |
| **16** | -32.768 – 32.767 | short |
| **32** | -2.147.483.648 – 2.147.483.647 | long |
| **64** | $-2^{63} – 2^{63}-1$ | long long |

### 2.2.3. Festkommadarstellung

Analog zu reeller Zahlendarstellung im Dezimalen lässt sich auch im Dualen eine Nachkommdarstellung definieren, dies aber unter dem Vorbehalt der endlichen Darstellung, da Bitvektoren in digitalen Systemen stets eine endliche Genauigkeit besitzen. Bei einem gegebenen Bitvektor lässt sich die Position des Kommas definieren. Alle Bits links von dieser Position repräsentieren

Ganzzahlen, alle Bits rechts davon stehen für Nachkommastellen. Folgende Wertigkeiten ergeben sich in dieser Darstellung:

$$b_{k-1} \dots b_0, b_{-1} \dots b_{-l}$$

mit $b_i$ Bit der Wertigkeit $2^i$ und Vektorbreite $n = k + l$

Durch die begrenzte Länge der Bitvektoren ist auch der darstellbare Zahlenbereich limitiert. Die kleinste darstellbare Zahl größer Null ist $2^{-l}$ und ist damit begrenzt durch die Länge des Nachkommateilvektors. Der größte repräsentierbare Wert ist $(2^n - 1)/2^l$. Die vorgestellte Festkommadarstellung ist zunächst als vorzeichenlos zu verstehen. Zur Erweiterung des Formats für die Unterstützung der vorzeichenbehafteten Darstellung kann wie im ganzzahligen Beispiel die K2-Darstellung genutzt werden.

Abgesehen von zusätzlicher Korrekturlogik ist die Arithmetik im Festkommaformat identisch mit dem Ganzzahlverfahren. Die Addition und Subtraktion können direkt mit demselben Addierer bzw. Subtrahierer ohne Beachtung der Kommaposition angewendet werden. Das Komma muss allerdings bei beiden Eingangswerten an derselben Position stehen. Bei Multiplikation hingegen muss die verschobene Kommaposition vom Ergebnis korrigiert werden, wie am dezimalen Beispiel zu erkennen ist:

$$(1{,}3 \cdot 100) \cdot (2{,}0 \cdot 10) = 2{,}6 \cdot 1000$$

Die Breite des resultierenden dualen Vektors entspricht der Summe der Breiten der beiden Eingangsvektoren. Die resultierende Kommaposition entspricht der Summe der Kommapositionen der Eingangswerte und muss je nach Bedarf korrigiert werden. Dies geschieht ähnlich wie im Dezimalen durch eine Schiebeoperation. Im Gegensatz zur Addition/Subtraktion muss bei der Multiplikation die Kommaposition der beiden Eingangsvektoren nicht identisch sein.

### 2.2.4. Fließkommadarstellung

Zur Erhöhung der Genauigkeit im Vergleich zur Festkommadarstellung lässt sich die Fließkommadarstellung nutzen. Es ist ein komplexes Format und es lässt sich dazu kein Analogon im Dezimalen finden. Die Definition des Formats erfolgte nach dem IEEE 754 Standard [61] und umfasst mehrere Bitbreiten. Jeder Bitvektor beinhaltet ein Bit für das Vorzeichen, jeweils einen Teilvektor für die Kodierung der Mantisse und des Exponenten. Tabelle 6 gibt eine Übersicht vordefinierter Grundformate.

**Tabelle 6 Grundformate der Fließkommadarstellung nach dem IEEE 754 Standard**

| Datentyp | Bitbreite | Exponent | Mantisse | Kleinste Zahl | Größte Zahl |
|---|---|---|---|---|---|
| **Single** | 32 | 8 | 23 | $1 \cdot 10^{-45}$ | $3{,}403 \cdot 10^{38}$ |
| **Double** | 64 | 11 | 52 | $5 \cdot 10^{-324}$ | $1{,}796 \cdot 10^{308}$ |

Beim 32-bit Format wird von einfacher Genauigkeit (single precison) gesprochen, während das 64-bit Format die doppelte Genauigkeit besitzt. Das binäre Format für die einfache Genauigkeit ist definiert wie folgt:

| Feld: | Vorzeichen (V) | Exponent (E) | Mantisse (M) |
|---|---|---|---|
| Bitbreite: | 1 | 8 | 23 |

Das Bit V repräsentiert das Vorzeichen, das negativ ist, wenn V den Wert 1 besitzt. Dieser Zusammenhang wird durch $(-1)^V$ wiedergegeben. Der Exponent E mit 8 Bits Genauigkeit besitzt einen positiven Wertebereich von 0 bis 255. Die Werte 0 und 255 sind reserviert und stehen somit der Fließkommadarstellung nicht direkt zur Verfügung. Zur Bestimmung des kodierten Exponenten E muss von diesem Wert der Bias (B) 127 subtrahiert werden, so dass man den tatsächlichen Wertebereich abzüglich der reservierten Werte von -126 bis 127 erhält. Die Mantisse M muss ebenfalls in die tatsächliche Darstellung umgerechnet werden, da im normalisierten Modus eine implizite 1 vor dem Komma steht und damit für die kodierte Mantisse $m = 1.M$ gilt. Für die normalisierte Darstellung ergibt sich damit:

$$(-1)^V \cdot 1, M \cdot 2^{E-B}$$

Darüber hinaus existieren weitere Betriebsmodi, die in folgender Tabelle aufgelistet sind:

**Tabelle 7 Betriebsmodi des Fließkommaformats einfacher Genauigkeit (32 bit)**

| Exponent E | Mantisse M | Wertebereich | Bezeichnung |
|---|---|---|---|
| E = 0 | M = 0 | $(-1)^S \cdot 0$ | Null |
| E = 0 | M > 0 | $(-1)^S \cdot 0.M \cdot 2^{1-B}$ | Denormalisiert |
| $0 < E < 2^r-1$ | M >= 0 | $(-1)^S \cdot 1.M \cdot 2^{E-B}$ | Normalisiert |
| $E = 2^r-1$ | M = 0 | $\infty$ | Unendlich |
| $E = 2^r-1$ | M > 0 | NaN | Keine Zahl (NaN) |

Wie der Tabelle 7 entnommen werden kann, unterstützt das Fließkommaformat neben dem normalisierten Standardmodus auch die Darstellung der Randbereiche, wie Unendlichkeit, oder „Keine Zahl“, um zu signalisieren, dass eine ungültige Operation ausgeführt wurde. Die de-normalisierte Darstellung wird verwendet, um sehr kleine Zahlen zu kodieren. Dabei wird keine implizi-

te eins vor dem Komma angenommen und der Exponent wird auf den kleinsten Wert gesetzt. Damit kann mit der Mantisse ein Wert angenommen werden, der durch die normalisierte Form nicht erreicht werden kann.

Neben den Grundformaten der Fließkommadarstellung definiert IEEE 754 auch erweiterte Formate, die keine festen Längen für den Exponenten und die Mantisse vorgeben, sondern lediglich Mindestgrößen definieren. Abgesehen davon wurden arithmetische Operationen, Rundungstechniken, Wurzelberechnung, Konversion und Fehlerbehandlung spezifiziert. Damit wird sichergestellt, dass Implementierungen auf verschiedenen Rechensystemen kompatibel bleiben. Im Rahmen dieser Arbeit wird darauf verzichtet weitere Details der Fließkommadarstellung zu erläutern und stattdessen auf die entsprechende Fachliteratur verwiesen.

## 2.3. Klassische Architekturen

Zu den klassischen Architekturen gehören Mikroprozessoren, die inzwischen eine Entwicklungszeit von rund 40 Jahren hinter sich haben und im Wesentlichen die Grundlagen der modernen Datenverarbeitung darstellen. Anfänglich war der Aufbau der Prozessoren relativ einfach und den Möglichkeiten der zugrunde liegenden Halbleitertechnologien angepasst. In den siebziger Jahren besaßen integrierte Schaltungen bestenfalls nur wenige zehntausend bis einhunderttausend Transistoren. Damit war die Komplexität der Prozessoren stark durch diesen Faktor limitiert. Moderne Schaltkreise dagegen können bereits mehrere Milliarden Transistoren beherbergen. Diese Zahl ist über die Jahre gewachsen und ermöglichte den Entwicklern den Einsatz neuer Techniken, die zur Steigerung der Leistungsfähigkeit führten. Daneben haben mehrere Rechenkerne in die Prozessoren Einzug gehalten.

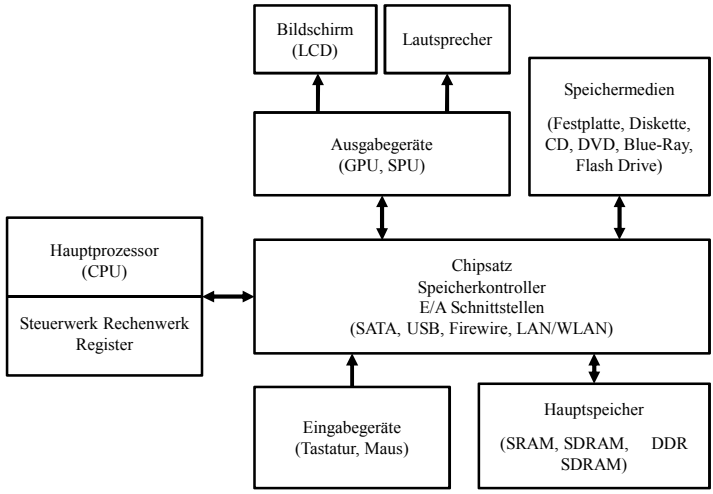



**Abbildung 9 Blockschaltbild eines modernen Rechensystems mit CPU als Hauptkomponente, Chipsatz, Hauptspeicher, diversen Ein-/Ausgabegeräten**

Die wichtigste Aufgabe von Prozessoren ist die Berechnung mathematischer Aufgaben. Zur Erfüllung dieser Aufgabe besitzen sie Komponenten wie das Rechenwerk und das Steuerwerk zur Ausführung von Instruktionen, die neben den arithmetischen und logischen Operationen zusätzlich diverse Kontroll- und Transportfunktionen bieten, um den Programmfluss zu steuern. Dazu gehören neben den Prozessorkontrollbefehlen auch Sprunginstruktionen zur Realisierung von Schleifen, bedingten oder unbedingten Sprüngen und Funktionsaufrufen. Für das Laden und Sichern der Daten sind Transportbefehle zu-

ständig. Beim Laden gelangen die Daten aus dem Speicher in die internen Prozessorregister und stehen den Operationen zur Verfügung. Berechnete Resultate werden beim Sichern aus den Registern wieder in den Speicher geschrieben. Der Speicher ist hierbei primär der Arbeitsspeicher des Systems, sekundär handelt es sich dabei um nichtflüchtige Massenspeichermedien wie Festplatte, Flash-Drives, CDs/DVDs oder Disketten. Abbildung 9 zeigt eine Übersicht der Grundkomponenten eines Datenverarbeitungssystems, wobei der Prozessor die zentrale Komponente darstellt.

Hauptsächlich tauscht der Prozessor Daten mit dem Hauptspeicher (Arbeitsspeicher) aus. Üblicherweise wird der Arbeitsspeicher beim Start des Systems durch Laden der Programme von nichtflüchtigen sekundären Speichermedien initialisiert und steht dann für die CPU zur Verfügung. Die geladenen Daten beinhalten den Betriebssystemkern, diverse Treiber (für die Ein-/Ausgabegeräte, GPU, SPU und selbst die CPU) und nicht zuletzt die eigentlichen Anwendungen. Die ersten beiden Teile ermöglichen den reibungslosen Betrieb des Systems, so realisiert das Betriebssystem über vordefinierte Softwareschnittstellen eine Verwaltung der Gerätetreiber und Anwendungen, während die Gerätetreiber dem Betriebssystem den Zugriff auf die Geräte ermöglichen. Treiber für die CPU geben dem Betriebssystem beispielsweise die Möglichkeit die CPU selbst in Stromsparmodi zu versetzen. Die Anwendungen werden während des Betriebs in Abhängigkeit der Benutzeraktivitäten bei Bedarf nachgeladen und zur Verfügung gestellt. Nicht benötigte oder gar beendete Anwendungen werden aus dem Speicher entfernt, Benutzerdaten dabei bei Bedarf auf den nichtflüchtigen Medien für die nächste Sitzung gesichert.

### 2.3.1. Rechenwerk / Datenpfad eines Prozessors

Das Rechenwerk bestimmt entscheidend die Eigenschaften eines Prozessors, da es im Wesentlichen alle Funktionen mittels seines Datenpfades durch integrierte Befehlsdecoder, Arithmetisch-Logische-Einheiten (ALUs), Registersätze, interne Puffer und Caches bestimmt. Das Steuerwerk nimmt Einfluss auf die Funktionen des Rechenwerks durch die Bereitstellung der notwendigen Steuersignale. Moderne Prozessoren besitzen jedoch kein separates Steuerwerk, da dieses als Teil des Datenpfades ausgeführt ist und wird daher im Folgenden nicht explizit betrachtet Abbildung 10 zeigt einen möglichen Aufbau des Rechenwerks anhand des hypothetischen 32bit RISC Prozessors DLX, von John L. Hennessy und David A. Patterson [62]. In ähnlicher Form ist diese Struktur auch in anderen Prozessoren zu finden [63][64].
Wie in Abbildung 10 dargestellt, wurde das Rechenwerk strukturell in fünf funktionale Bereiche unterteilt: Fetch, Decode, Execute, Memory und Write Back. Der erste Bereich (Fetch) lädt den nächsten Befehl aus dem Instruktions-

Cache anhand der Speicheradresse, die im PC-Register (program counter) gespeichert ist und errechnet die nächste Adresse (next program pointer, NPC) zum Laden der folgenden Instruktion. Der gelesene Befehl steht am Ausgang des Instruktion-Caches (IR) bereit und wird in der Decode-Stufe ausgewertet. Hier werden angeforderte Register aus dem Register-File gelesen oder aus Teilen der Instruktion zu vorzeichenbehafteten 32bit Werten erweitert, um beispielsweise neue Adressen für relative Sprünge zu berechnen. An das Register-File werden zwei Leseadressen und eine Schreibadresse angelegt. Dadurch können, je nach Instruktion, bis zu zwei Operanden (A/B) ausgelesen werden und ein Ergebnis in der Write-Back-Stufe gespeichert werden.

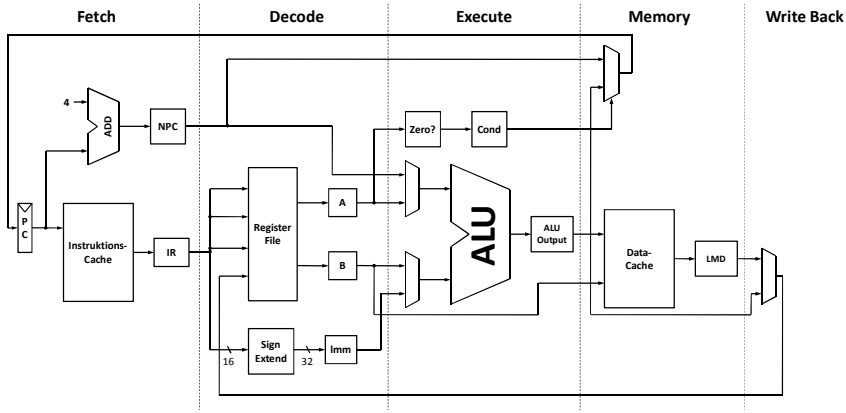



**Abbildung 10 Rechenwerk des DLX Prozessors mit funktionaler Aufteilung**

Die Execute-Stufe verrechnet bereitgestellte Werte aus dem Register-File, NPC oder Konstanten/Adressen in der arithmetisch-logischen Einheit (ALU). Die Menge und Art der Operationen innerhalb der ALU definieren entscheidend die Eigenschaften des Prozessors und werden von Prozessor-Entwicklern gezielt ausgewählt, mit der Absicht Zielapplikationen möglichst effizient zu verarbeiten. So kann beispielsweise das Fehlen der Multiplikation zu starken Leistungseinbußen führen, falls Applikationen in großem Maße davon gebraucht machen. In diesem Fall müsste die Multiplikation durch mehrere Additionsbefehle und logische Verknüpfungen realisiert werden, was enorme Leistungseinbußen nach sich ziehen würde, da statt einer Operation eine Sequenz von Instruktionen ausgeführt werden müsste. Auf der anderen Seite sind Multiplikationen sehr teuer in der Implementierung. Sollten Anwendungen keine Multiplikation benötigen, so lässt sich zu Gunsten der günstigeren Herstellung darauf verzichten. Dies gilt insbesondere für die Division, die in der Implementierung sehr teuer ist und enorme Laufzeiten mit sich bringt.

Zusätzlich zu ALU-Operationen wird in der Execute-Stufe der Operand A auf den Wert Null geprüft. Das Ergebnis ist eine Bedingung (Cond), die genutzt werden kann, um bedingte Sprünge zu realisieren. Die davon abhängige Auswahl der nächsten Instruktionsadresse geschieht in der Memory Stufe. Darüber hinaus können in dieser Stufe zuvor ausgelesene Werte aus dem Register-File in den Daten-Cache bzw. Arbeitsspeicher geschrieben werden.

In der letzten Stufe des Rechenwerks (Write-Back) werden berechnete Ergebnisse oder gelesene Daten aus dem Daten-Cache an das Register-File geleitet und nachfolgenden Instruktionen zur Verfügung gestellt.

Dieses relativ einfache Rechenwerk führt je eine Instruktion in einem Takt aus. Ausgehend vom PC-Register, werden alle Stufen in einem Takt verarbeitet. Die einfache Struktur führt jedoch zu langen kritischen Pfaden zwischen den Registern und beschränkt entscheidend die maximale Arbeitsfrequenz des Systems. Diese Einschränkung hat zur Folge, dass weniger Instruktionen pro Zeiteinheit verarbeitet werden können, d.h. der Durchsatz der Befehle pro Zeiteinheit bleibt niedrig und somit fällt die erreichbare Performance nur mäßig aus.

### 2.3.2. Pipelining

Zur Steigerung des Durchsatzes wird in der Prozessortechnik das Pipelining eingesetzt. Dabei wird versucht durch das Einfügen von Registern die kritischen Pfade zu verkürzen. Damit das Rechenwerk weiterhin die zuvor beschriebene Funktionalität erfüllen kann, muss das Einfügen nach einem festen Muster erfolgen. Im letzten Abschnitt wurden bereits funktionale Abschnitte im Rechenwerk beschrieben, die benutzt werden können, um das Rechenwerk in fünf Pipeline-Stufen einzuteilen. Abbildung 11 stellt eine derartige Struktur dar.

Wie aus der Abbildung ersichtlich wurden alle Pipeline-Stufen durch Register voneinander entkoppelt. Auf diese Weise beschränken sich die kritischen Pfade immer nur auf eine Stufe. Bei der Annahme, dass die Signallaufzeiten in allen Stufen identisch sind, lässt sich dadurch die Arbeitsfrequenz der Architektur um Faktor fünf erhöhen und damit der Durchsatz der Befehle um denselben Faktor anheben. Diese Annahme ist allerdings nur theoretischer Natur und lässt sich nur hypothetisch anwenden. In realen Systemen sind die RAM- bzw. Cache-Zugriffe und die Laufzeit durch die ALU die bestimmenden Faktoren und müssen bei der Einführung von Pipelines beachtet werden.
Im Falle der ALU ist die komplexe Netzliste für die Multiplikation oder gar die Division die bestimmende Komponente. Cache-Zugriffe dauern für gewöhnlich länger als einen Takt und verzögern die Verarbeitung zusätzlich. In Abbildung 11 wurden fünf Pipeline-Stufen eingefügt. Durch Einfügen weiterer

Stufen, wobei sich beispielsweise die Execute-Stufe über mehrere Takte erstrecken kann, lässt sich eine Pipeline in einem Prozessor weiter ausbalancieren und damit der Durchsatz zusätzlich steigern.

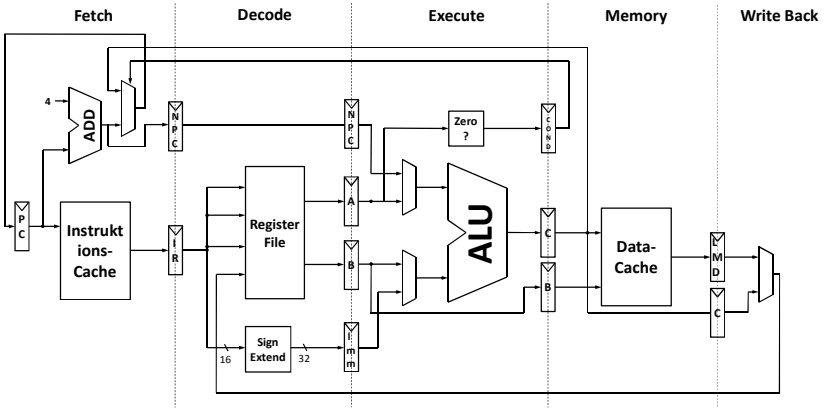



**Abbildung 11 Der erweiterte DLX-Datenpfad mit fünf Pipeline-Stufen**

Eine vollständig gefüllte fünfstufige Pipeline enthält fünf Befehle, d.h. ein Befehl pro Pipeline-Stufe. Die Idee ist, dass sich jede Instruktion innerhalb der Pipeline im eigenen funktionalen Abschnitt befindet und damit ungeteilt alle Ressourcen des betreffenden Abschnittes für sich nutzen kann. Im nächsten Takt werden alle relevanten Daten jeweils in die Eingangsregister der nächsten Pipelinestufe geschoben und im nächsten Takt für die nächste Stufe zur Verfügung gestellt. Dieser Vorgang lässt sich in einem Pipeline-Diagramm darstellen. Dabei lassen sich relativ leicht mögliche Probleme erkennen, die in einer Pipeline auftreten können.

Pipeline-Komponenten, die sich in einem Spalt des Pipeline-Diagramms befinden, werden gleichzeitig von den geladenen Befehlen in der Pipeline benötigt. Werden dieselben Pipeline-Komponenten in verschiedenen Pipelinestufen gebraucht, so besteht ein struktureller Konflikt, ein sogenannter Struktur-Hazard. Aus diesem Grund ist es beispielsweise von Vorteil zwei Caches zu nutzen: je ein Cache für Daten und Instruktionen. Der Instruktions-Cache wird in der Fetch-Stufe genutzt, der Daten-Cache in der Memory-Stufe. Bei gemeinsamer Ausführung der beiden Caches in einem Ein-Port-Speicher könnte immer nur eine der beiden Stufen auf den Cache zugreifen, die jeweils andere müsste solange warten. Ein Zwei-Port-Speicher ist in der Realisierung teurer, so dass auf diese Möglichkeit verzichtet wird. Generell lassen sich Struktur-Hazards durch Hinzufügen von zusätzlichen Ressourcen lösen.

Neben den Struktur-Hazards werden zwei weitere Hazards unterschieden: Daten- und Control-Hazards. Daten-Hazards treten auf, wenn ein Befehl Daten

anfordert, die der vorhergehende Befehl erst bereitstellen muss. Beispielsweise muss ein Lade-Befehl Daten in der Memory-Stufe der Pipeline aus dem Speicher lesen, bevor ein darauffolgender Befehl eine arithmetische Operation auf diesen Daten ausführen kann. Da die Memory-Stufe des Befehls i über der Execute-Stufe des Befehls i+1 steht und eine direkte Datenabhängigkeit gegeben ist, ist eine aufeinander folgende Ausführung nicht möglich. Zur Lösung dieses Problems werden Pipeline-Staus (Stalls) verwendet, d.h. in diesem Fall wird der arithmetische Befehl einschließlich aller folgenden Befehle angehalten, bis die notwendigen Daten durch die Memory-Stufe bereitgestellt wurden. Durch eine weitere Technik, genannt Forwarding, werden die Daten an die richtige Stufe innerhalb der Pipeline geleitet und im Voraus bereitgestellt. Im beschriebenen Szenario kann die Addition einen Takt später ausgeführt und damit die Programmausführung mit einem Takt Verzögerung fortgesetzt werden. Beim Einsatz von Compilern kann jedoch eine Optimierung diese Situation entschärfen, in dem durch Instruction-Reordering die Sequenz von Befehlen so angeordnet werden, dass keine Abhängigkeiten auftreten.

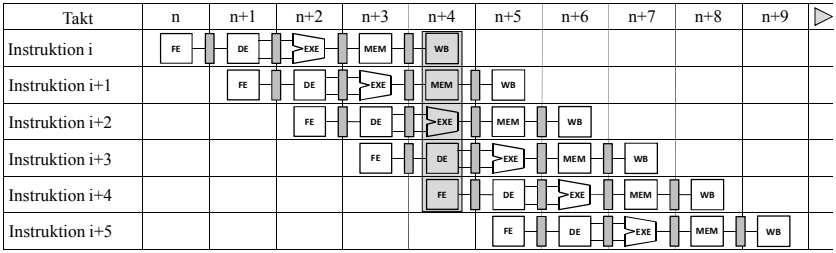



**Diagramm 1 Ablauf der Befehlsausführung in einer fünfstufigen DLX-Pipeline**

Control-Hazards treten auf, wenn der Programmfluss durch eine Sprunginstruktion geändert wird. Die Sprungadresse für diesen Fall kommt aus der Memory-Stufe und wird erst im nächsten Takt in der Fetch-Stufe ausgeführt. Vom Sprungbefehl bis zum ausgeführten Sprung werden also drei weitere Befehle in die Pipeline geladen, die nicht ausgeführt werden. Die Ergebnisse dieser Befehle werden verworfen, sodass die Pipeline einen dreifachen Stall ausführen muss. Nach dem Laden der neuen Adresse, wird die Pipeline regulär fortgesetzt. Auch hier existieren Techniken, um diesem Problem abzuhelfen. Moderne Prozessoren nutzen hier sogenannte Branch-Prediction-Verfahren (Sprungvorhersage), um basierend auf Wahrscheinlichkeiten und/oder vorherigem Verhalten an bestimmten Positionen im Code bei bedingten Sprüngen den gewünschten Programmpfad zu wählen, bevor das Ergebnis der Bedingung vorliegt.

### 2.3.3. Superskalarität

Durch die Einführung von Pipelines in der Prozessortechnik, wurde eine deutliche Leistungssteigerung erreicht. Diese Erweiterung erhöhte den Durchsatz der Prozessoren, da dadurch insbesondere höhere Taktraten erreichbar wurden. Die zusätzlichen Ressourcen für diese Erweiterung beschränken sich im Wesentlichen auf die Register, die notwendig sind, um die Pipeline-Stufen zu entkoppeln. Zusätzlicher schaltungstechnischer Aufwand für die Forwarding- und Stall-Mechanismen ist im Vergleich zum gesamten Prozessor klein. Die Grenzen dieser Erweiterung liegen in der maximalen Tiefe einer Pipeline, die noch sinnvoll realisiert werden kann. Tiefe Pipelines erlauben hohe Taktraten, dieser Vorteil wird jedoch durch Sprungbefehle, insbesondere bedingte Sprünge, wieder zunichte gemacht. Hier muss bei einer Programmflussänderung ein großer Teil der Pipeline verworfen und wieder gefüllt werden, was die Effizienz senkt. Der effektive Durchsatz der Instruktionen pro Zyklus (IPC) sinkt, da in dieser Zeit über mehrere Takte keine Instruktionen verarbeitet werden.

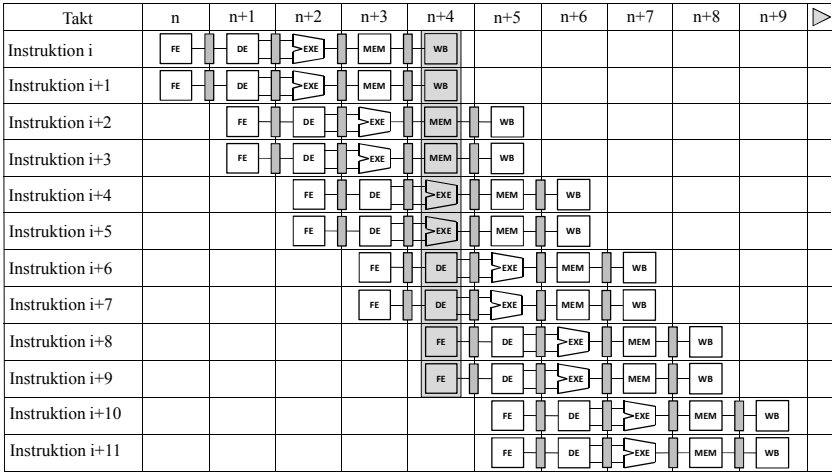



**Diagramm 2 Zweifach-superskalare Befehlsausführung eines DLX-Prozessors**

Zur Erzielung weiterer Leistungssteigerung wird eine Technik genutzt, die auf der Vervielfachung aller Funktionseinheiten und somit der Pipeline basiert. Dies führt zur parallelen Verarbeitung von Instruktionen, wie im Diagramm 2 für zweifache Superskalarität dargestellt. Die Kosten dieser Erweiterung betragen mehr als das Zweifache, da zusätzlicher Aufwand für den Scheduler aufgewendet werden muss, um die geladenen Befehle auf die Funktionseinheiten zu verteilen.

Es werden zwei Arten der Befehlsverteilung unterschieden, das statische und das dynamische Scheduling. Beim statischen Scheduling wird die Reihenfolge der Befehle im Programfluss nicht verändert (In-Order-Execution), sondern lediglich die Abhängigkeiten zwischen den Instruktionen beachtet. Beim dynamischen Scheduling hingegen verändert der Prozessor die Reihenfolge der Befehle (Out-Of-Order-Execution) zur Reduktion der Abhängigkeiten, um die Pipeline möglichst effizient auszulasten. Im ersten Fall hängt die resultierende Leistungsfähigkeit des Prozessors deutlich von der Code-Qualität und somit vom Compiler ab. Der Einfluss des Compilers ist im zweiten Fall deutlich reduziert.

### 2.3.4. RISC, CISC, VLIW und EPIC Ansätze

Die Ausprägungen der Prozessoren nahmen mit der Anzahl verschiedener Prozessormodelle im Laufe der Jahre zu. So lassen sich die verschiedenen Architekturen durch ihre Instruktionssätze und damit einhergehenden Architekturen klassifizieren. Die bekanntesten unter ihnen sind die RISC und CISC Architekturen. In den achtziger Jahren kamen dann die VLIW Architekturen hinzu und schließlich die EPIC Architekturen in den neunziger Jahren.

RISC-Architekturen (Reduced Instruction Set Computer) zeichnen sich durch ihren vereinfachten Instruktionssatz aus. So besitzen Maschinenbefehle einer RISC-Architektur eine feste Länge. Darüber hinaus beschränken sich die Maschinenbefehle auf einfache Operationen, wie Laden/Speichern von Werten zwischen Registern und Arbeitsspeicher. Alle Operationen werden auf interne Registerwerte angewendet. Aus diesem Grund werden RISC-Architekturen auch als Load/Store-Maschinen bezeichnen, da jede Operation ein Laden und abschließendes Speichern der Werte voraussetzt. Diese Vereinfachung spiegelt sich ebenfalls im Aufbau der RISC-Prozessoren wieder und erlaubt damit das Design von schnellen Pipelines, sodass RISC-Prozessoren im Allgemeinen mit höheren Taktraten betrieben werden können, als dies bei CISC der Fall ist.

CISC-Architekturen (Complex Instruction Set Computer) hingegen besitzen einen wesentlich komplexeren Aufbau, der sich vor allem in komplexeren Maschinenbefehlen wiederspiegelt. So besitzen ihre Befehle keine feste Länge, was ihrer Erweiterungsfähigkeit zu Gute kommt. In der evolutionären Entwicklung von Prozessoren können auf diese Weise immer mehr Befehle eingefügt werden, ohne dass der vorhandene Befehlsraum ausgeschöpft wird. Die resultierende Instruktionsmenge ist dann jedoch hinsichtlich ihrer Länge asymmetrisch. Die implementierten Befehle dieser Architekturen können weiterhin wesentlich komplexere Operationen ausführen. So sind arithmetische Operationen direkt auf Speicheradressen möglich, ohne zuvor einen Lade-Befehl ausführen zu müssen. Dieser Umstand lässt sich durch das Zusammen-

fassen von einfachen RISC-Befehlen zu komplexeren CISC-Befehlen lösen, womit auch die Vielfalt der einzelnen Operationen enorm ansteigt. Und genau hier ist auch der Nachteil dieser Architekturen begründet: da die Pipelines zu diesem Zweck stark aufgebläht werden, erhöht sich dadurch die schaltungstechnische Komplexität und begrenzt somit die maximal sinnvolle Pipeline-Tiefe. Moderne Prozessoren nutzen diesen Umstand jedoch zu ihrem Vorteil, indem CISC-RISC-hybride Architekturen aufgebaut werden. Ihre CISC-Instruktionssätze werden intern in einfache RISC-Befehle decodiert und erst dann ausgeführt. Das Ergebnis ist eine schnelle CISC-Maschine, die jedoch zusätzlichen schaltungstechnischen Aufwand für die Realisierung der Befehlsdecoder erfordert.

Seit der Einführung der superskalaren Konzepte, gibt es Bestrebungen die mehrfach vorhandenen Funktionseinheiten durch adäquate Instruktionssätze effizient auszunutzen. Die VLIW-Architekturen (Very Long Instruction Word) stellen einen derartigen Schritt dar und zeichnen sich durch eine Reihe von Innovationen aus. Die Besonderheit der VLIW-Instruktionen liegt darin begründet, dass mehrere einzelne Instruktionsbündel in eine große VLIW-Instruktion gepackt werden. Diese Instruktionen haben die Eigenschaft, dass sie ohne Konflikte von der Pipeline des Prozessors parallel ausgeführt werden können. Zu diesem Zweck muss der Compiler während der Erstellung des Maschinencodes alle Abhängigkeiten der Befehle prüfen und passende Gruppierungen erzeugen. Damit übernimmt der Compiler einen großen Teil der Code-Analyse und vereinfacht dadurch sowohl den Instruktionsdecoder als auch Dispatcher im Prozessor, die sich auf das statische Scheduling und In-Order-Execution beschränken. Die Anzahl paralleler Instruktionen sind bei VLIW-Architekturen zum Designzeitpunkt des Prozessors festgelegt und können nicht mehr verändert werden ohne Inkompatibilitäten in Kauf zu nehmen. Das bedeutet, dass neue Versionen eines Prozessors innerhalb einer Prozessorfamilie nicht mehr denselben Maschinencode ausführen können, wenn sich die Anzahl der parallelen Instruktionen innerhalb der VLIW-Instruktion ändert. Hierdurch ergeben sich sowohl Aufwärts- als auch Abwärtsinkompatibilitäten zwischen den Prozessor-Generationen. Ein weiteres Problem stellt die Code-Größe dar. Die üblichen VLIW-Instruktionslängen liegen bei 128 oder gar 256 Bits. Diese enthalten zwischen drei und acht parallele Instruktionen, die nicht immer vollausgenutzt werden können. Die leeren Instruktionen werden mit No-Operation-Instruktionen (NoP) aufgefüllt und haben somit keine Funktion, benötigen aber Platz im Speicher und damit auch Bandbreite bei Speicher-Prozessor-Transfers. Dieses Problem ist neben der Einschränkungen der Parallelisierbarkeit von Applikationen auch auf die VLIW-Compiler zurückzuführen, die oftmals in der Vergangenheit nicht die nötige Qualität lieferten. Eine Abhilfe stellt hier die Code-Kompression dar. Eine weitere Möglich-

keit liegt in der Erweiterung der VLIW-Architektur zur EPIC-Architektur (Explicitly Parallel Instruction Computing).

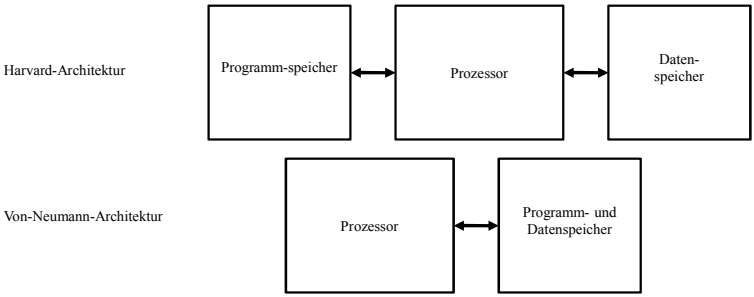



**Abbildung 12 Klassifizierung der Prozessoren bzgl. der Speicheranbindung**

EPIC-Architekturen verfolgen das Ziel die Nachteile der Code-Inkompatibilitäten der VLIW-Architekturen zwischen den Prozessor-Generationen zu eliminieren und darüber hinaus schonender mit der Bandbreite des Hauptspeichers umzugehen. Zu diesem Zweck besitzen sie keine statischen Instruktionsgruppenlängen im Vergleich zu VLIW-Architekturen, sondern Instruktionsgruppen unterschiedlicher Länge. Innerhalb einer Instruktionsgruppe können Befehle mit beliebigem Parallelitätsgrad enthalten sein, was auch die Ausführung in mehreren Schritten nicht ausschließt. Das Format der EPIC-Instruktionen ähnelt dem der VLIW-Instruktionen. Die Instruktionsgruppen werden allerdings durch Stopper voneinander getrennt und können sich über mehrere EPIC-Instruktionen erstrecken. Die Notwendigkeit NoPs in den Code einzufügen ist durch die flexible Gruppierung der Instruktionen nicht mehr vorhanden. Das verkleinert die Größe des Codes und schont zusätzlich die Bandbreite.

### 2.3.5. Harvard / von-Neumann Paradigmen

Beim Aufbau und der Funktionsweise von Prozessoren werden mehrere interne und externe Konzepte unterschieden. In vorangehenden Abschnitten wurden bereits mehrere interne Aspekte diskutiert. Die Art der Anbindung des Prozessors an den Hauptspeicher erlaubt eine weitere wichtige Klassifizierung. Wird der Programm-Code gemeinsam mit den Anwendungsdaten im selben Speicher vorgehalten, so spricht man von einer von-Neumann-Architektur, siehe Abbildung 12. Die Aufteilung des Programm-Codes und der Anwendungsdaten auf separate Speicher wird in einer Harvard-Architektur genutzt. Während die von-Neumann-Architektur eine transparente lineare Speicheradressierung ermöglicht, wird die Speicherbandbreite im Vergleich zur Harvard-

Architektur halbiert. Aus diesem Grund benutzen viele Prozessoren heute eine hybride Lösung, die intern auf Cache-Ebene auf einer Harvard-Architektur aufsetzt, extern jedoch von den von-Neumann-Vorteilen profitiert.

Diesen Ansatz verfolgt der DLX-Prozessor. Wird nur der Cache in der Pipeline betrachtet, handelt es sich um eine Harvard-Architektur. Der Grund für diesen Ansatz ist auf die Anforderungen der Pipeline zurückzuführen. Wie bereits diskutiert, müssen die Pipeline-Stufen Fetch und Memory gleichzeitig auf den Speicher zugreifen, was nur mit getrennten Speichermodulen kostengünstig realisiert werden kann. Der Aufwand für Speichermodule steigt mit der Anzahl der implementierten Ports. Abgesehen davon sind Programm- und Datenbereiche im Hauptspeicher gewöhnlich voneinander getrennt, so dass es wenig Sinn macht, einen gemeinsamen Speicher für beide Zwecke zu integrieren. Diesen Ansatz verfolgen die meisten modernen Prozessoren, wobei der interne Cache innerhalb der Prozessoren als SRAM-Speicher integriert und mit vollem Prozessor-Takt betrieben wird. Die Größe beschränkt sich meist auf wenige bis hin zu einigen hundert Kilobytes. Auf diese Weise werden häufig benötigte Daten oder Instruktionen in lokalen Caches gepuffert und der Prozessorkern wird damit vom relativ langsamen Arbeitsspeicher entkoppelt. Zur Berechnung der resultierenden Beschleunigung durch diese Technik lässt sich Amdahls Gesetz [67] nutzen und folgende Formel angeben:

$$F_{Gesamt.Beschleunigung} = \frac{1}{(1-A) + \frac{A}{F_{Cache.Beschleunigung}}}$$

$$F\text{: Faktor für die Beschleunigung,} \qquad A\text{: Anteil gepufferter Zugriffe}$$

Sollte der Cache 10x schneller sein als der Hauptspeicher und in 90% aller Speicherzugriffe eingesetzt werden, so ergibt das

$$F_{Gesamt.Beschleunigung} = \frac{1}{(1-0.9) + \frac{0.9}{10}} = \frac{1}{0.19} = 5{,}3$$

$$mit \qquad F_{Cache.Beschleunigung} = 10, \qquad A = 90\%$$

Die verbesserte Speicherhierarchie beschleunigt das System um den Faktor 5,3. Es sei an dieser Stelle angemerkt, dass diese Beschleunigung von den Eigenschaften der jeweiligen Anwendung abhängig ist. Beispielsweise sind Schleifen, die vollständig in den Cache passen, geeignete Kandidaten für die Beschleunigung.

Zusätzlich zum internen Cache in den Pipelines, werden bei sehr schnellen Prozessoren zusätzliche Cache-Level genutzt, siehe Sandy-Bridge-Prozessor im nächsten Kapitel. Der interne Cache wird dabei als Level-1-Cache bezeichnet. Dieser ist gleichzeitig der schnellste und kleinste Cache. Level-2-Cache ist

direkt hinter dem Level-1-Cache implementiert und besitzt eine Größe von einigen 100 Kilobytes bis hin zu einigen Megabytes. In Multi-Core-Prozessoren werden zusätzlich Level-3-Caches genutzt, die allen Kernen des Prozessors zur Verfügung stehen. Diese besitzen in modernen Prozessoren mehr als 10 Megabytes.

### 2.3.6. Multi-Core-Prozessoren

Neben dem Einsatz von Pipelines und Superskalarität gibt es auch Techniken, die naheliegender sind, um Leistungssteigerungen zu erreichen. Bisher wurden immer nur einzelne Prozessorkerne diskutiert. Leistungssteigerungen bis zum Anfang des ersten Jahrzehnts des neuen Jahrhunderts wurden durch den Einsatz vorgestellter Techniken erreicht. Hinzu kamen die Fortschritte der Halbleitertechnik, die enorme Frequenz- und Ressourcensteigerungen erlaubten und damit die Entwicklung vorantrieben. Allerdings stockte zu diesem Zeitpunkt die gewohnte Entwicklung der Taktratenerhöhungen, da die Verlustleistung schnell getakteter Prozessoren in kritische Bereiche vorstieß und eine Abführung der Wärme in gefordertem Maße technisch kostengünstig nicht realisierbar war. Die Stagnation zwang die Hersteller zum Umdenken und sie haben einen neuen Weg zur Erhöhung der Leistungsfähigkeit eingeschlagen.

Da die Steigerungen der Taktraten nicht mehr fortgesetzt werden konnten, wurden stattdessen mehrere Prozessorkerne auf einem Chip integriert. Durch diesen Schritt können heute relativ einfach Systeme mit mehreren Prozessorkernen aufgebaut werden. Dieser Umstand hat nicht nur den Vorteil erhöhter Leistung, sondern auch den Nachteil komplexerer Programmierung des Betriebssystems und der Applikationen. Möchte der Anwender die vorhandene Rechenleistung ausnutzen, so muss die Software multiprozessorfähig sein. Dazu muss der Programmierer im Voraus seine Software drauf auslegen, was nicht immer realisiert werden kann. In Abhängigkeit der verwendeten Algorithmen sind diese Möglichkeiten oft nur eingeschränkt möglich, so dass der Anwender schlussendlich nur begrenzt von einem Multi-Prozessor-System profitiert. Hauptsächlich wurde zunächst lediglich die Reaktionsfähigkeit des Systems verbessert.

### 2.3.7. Klassifizierung nach Michael Flynn

Vorgestellte Konzepte wie Superskalarität, VLIW- und EPIC-Maschinen zielen darauf ab, Leistungssteigerungen durch höhere Parallelität zu erreichen. Im Falle von RISC- und CISC-Maschinen werden je nach Modell zur Laufzeit Abhängigkeiten im Code untersucht und auf mehreren Funktionseinheiten zur Ausführung gebracht. Die Grundlage dafür sind einzelne Instruktionen, die

Operationen auf Daten anwenden. Gewöhnlich ist mit einer Instruktion jedoch eine Operation verbunden, wie die Addition oder die Multiplikation von zwei skalaren Werten. Das Ergebnis ist wieder ein skalarer Wert. Diese Art der Datenmanipulation wird Single Instruction Single Data (SISD) genannt.

Analog dazu lassen sich auch mehrere Datensätze durch eine Instruktion berechnen. Das Ergebnis ist eine Vektoroperation und es wird von Single Instruction Multiple Data (SIMD) gesprochen. Zur Realisierung dieser Technik, wird eine entsprechend der Anzahl der parallelen Operationen mehrfache Ausführung von Funktionseinheiten benötigt. Eine Pipeline-Architektur bleibt in ihrer Struktur nahezu gleich, muss jedoch in der Execute-Stufe mehrere ALUs beinhalten und eine entsprechende Anzahl an Ports im Register-File aufweisen.

**Tabelle 8 Klassifizierung der Rechnerarchitekturen nach Michael J. Flynn**

| | SingelInstruction | Multiple Instruction |
|---|---|---|
| Single Data | SISD | MISD |
| Multiple Data | SIMD | MIMD |

Die Idee von SISD und SIMD lässt sich weiter verfolgen und das Ergebnis ist zum einen Multiple Instruction Single Data (MISD) und zum anderen Multiple Instruction Multiple Data (MIMD). Im ersten Fall bedeutet das, dass mehrere Operationen auf einem Datensatz arbeiten. Dies können gleiche Operationen sein oder auch verschiedene. Gleiche Operationen würden eine Redundanz nach sich ziehen und mit einem möglichen Vergleich der Ergebnisse der Operationen eine fehlertolerante Rechenmaschine ermöglichen. MISD mit unterschiedlichen Operationen ist eher eine theoretische Konstruktion, zumal die reale Anwendbarkeit aus Kostengründen inakzeptabel ist. MIMD kann durch eine VLIW-Maschine realisiert werden, wenn einzelne Instruktionen SIMD-Operationen ausführen. Das Ergebnis wäre wieder eine MIMD-Maschine. Als MIMD lassen sich auch array-basierte Rechenarchitekturen in Form systolischer Arrays bezeichnen, wie sie in den folgenden Abschnitten vorgestellt werden. Tabelle 8 veranschaulicht die dargestellten Zusammenhänge, wie sie von Michael J. Flynn in seiner Publikation bereits 1966 [66] klassifiziert wurden.

## 2.4. Rekonfigurierbare Architekturen

Prozessoren, wie sie in vorangegangen Abschnitten diskutiert wurden, sind etablierte Konzepte, die über ein halbes Jahrhundert erforscht und entwickelt wurden. Die Idee, algorithmische Probleme in zeitlich festgelegte Abfolge von Befehlen umzusetzen, ist aus menschlicher Sicht intuitiv und daher naheliegend. Die Effizienz der resultierenden Rechensysteme ist jedoch verglichen mit ASICs relativ gering, wie dies an der Abbildung 3 erkennbar ist. Unter Effizienz sind in diesem Zusammenhang die Relation der Parameter Performance, Verlustleistung und die dafür benötigte Chipfläche zu verstehen. ASICs erreichen die höchstmögliche Effizienz, da sie den direktesten Weg zur Implementierung einer Anwendung auf Basis der verwendeten Technologie darstellen. Durch die Technologienähe erreichen sie die höchstmögliche Parallelität bei gleichzeitig höchster Komponentendichte und niedrigster Verlustleistung, sind jedoch fest verdrahtet und verlieren dadurch ihre Flexibilität. Prozessoren und DSPs verarbeiten Instruktionen sequentiell. Hier bieten superskalare und VLIW-Architekturen im Idealfall eine parallele Verarbeitung, stehen dennoch in ihrer Effizienz deutlich hinter den ASICs zurück. Doch wie gelingt es Vorteile beider Ansätze zu einem neuen Konzept zusammenzuführen? Die Idee ist die Realisierung hardwarenaher strukturell rekonfigurierbarer Architekturen. Sie sind seit über 25 Jahren Gegenstand der Forschung und die Entwicklung ist insbesondere für grobgranular rekonfigurierbare Architekturen noch nicht abgeschlossen.

### 2.4.1. Grundlegendes

Rekonfigurierbare Architekturen brechen mit dem gewohnten Paradigma der Rechensysteme hinsichtlich der sequentiellen Programmierung und Datenverarbeitung. Statt Algorithmen durch eine Sequenz von Befehlen zu beschreiben, wird hier versucht einen strukturellen Ansatz zu finden, der durch die Änderung der vorliegenden funktionalen Strukturen die Zielapplikation zur Ausführung bringt. Die neue Ausprägung der Hardware-Struktur für die Zielapplikation wird in diesem Zusammenhang als Konfiguration bezeichnet. Unter einer Rekonfiguration wird die Änderung einer vorliegenden Konfiguration verstanden. Die entstandene Hardware-Struktur ist in ihrer Art Daten zu verarbeiten einem ASIC sehr ähnlich, besitzt jedoch den zusätzlichen nicht zu vernachlässigbaren Hardware-Overhead für die gewünschte Flexibilität. Für den Entwickler der Architektur besteht die Herausforderung darin, einen Kompromiss zwischen Flexibilität, Kosten und Programmierbarkeit zu finden. Diese Aufgabe ist aber nicht leicht zu lösen, wie im nächsten Kapitel dargelegt wird und schon an der Vielfalt der Ansätze in diesem Bereich leicht zu erkennen ist.

In den achtziger Jahren kamen neue integrierte Schaltkreise der Firma Xilinx [10] auf den Markt: Feldprogrammierbare Gatteranordnung oder FPGAs (Field Programmable Gate Arrays). Hierbei handelte es sich um Schaltkreise, die keine vorgegebene Funktionalität besitzen, sondern vom Benutzer rekonfiguriert werden konnten. Dies erfolgte durch die Beschreibung der funktionalen Struktur, sehr ähnlich einem ASIC. Die Granularität der Beschreibung bewegte sich auf Bit-Ebene, wobei boolesche Gleichungen auf die FPGAs abgebildet werden können. Durch konfigurierbare Kommunikationsnetze und große Zahl von Registern, Lookup-Tabellen (LUTs) und in modernen FPGAs zusätzlichen dedizierten Einheiten für Arithmetik, Speicher und Ein-/Ausgabeeinheiten erhält der Benutzer ein mächtiges Werkzeug zur Realisierung hardwarenaher Anwendungen, die eine hohe Parallelität bieten. Man spricht von feingranular rekonfigurierbarer Logik, da sie auf Bit-Ebene arbeitet. Bzgl. der Effizienz können sich FPGAs durch den hardware-nahen Zugang der Anwendung deutlich von oben beschriebenen Prozessor-Architekturen absetzen. Allerdings kostet die feingranulare Natur des Bausteins zusätzliche Hardware-Ressourcen, um die resultierende Flexibilität zu bieten. Der Aufwand auf Bit-Ebene Rekonfiguration zu ermöglichen ist sehr groß, bietet allerdings die größte Flexibilität verglichen mit grobgranularen Architekturen. Durch die vergleichsweise höhere Funktionsdichte bieten grobgranulare Architekturen allerdings eine höhere Effizienz.

Aus diesem Grund versuchen viele Wissenschaftler bereits seit den achtziger Jahren grobgranular rekonfigurierbare Architekturen zu entwickeln. Die Ergebnisse waren beispielsweise systolische Arrays [67], die das Ziel verfolgten, auf Wort-Ebene flexible und hardware-nahe Ausführung der Applikationen zu ermöglichen. Statt der Verwendung von LUTs werden dedizierte ALUs benutzt, um die Flächeneffizienz zu steigern, in dem der Mehraufwand der Rekonfigurationsressourcen gesenkt wird. Allerdings büßt der Entwickler gleichzeitig einen großen Teil der Flexibilität ein, was die Anwendbarkeit seiner neuen Architektur einschränkt. Während bei FPGAs nahezu beliebige fehlende Funktionen im Nachhinein auf Bit-Level implementiert werden können, besteht diese Möglichkeit bei grobgranularen Architekturen nicht mehr. Hier muss der Entwickler im Voraus die notwendigen Funktionen vorsehen.

Neben den array-basierten Ansätzen existieren auch prozessorartige Architekturen, die unter die Kategorie rekonfigurierbarer Architekturen fallen. Hierbei handelt es sich um Rechenkerne, die nicht auf hohen Parallelitätsgrad setzen, sondern auf spezialisierte Befehle, die zur Laufzeit den Bedürfnisse der Anwendung angepasst werden. Dabei können beispielsweise rekonfigurierbare Strukturen als Teil des Prozessordatenpfads verwendet werden, die lose oder eng an den Prozessor gekoppelt sein können [68]. Eine weitere Architektur, wie sie im nächsten Kapitel vorgestellt wird und prozessorartig aufgebaut ist,

ist der Montium Prozessor. Dieser verarbeitet statt einer Befehlssequenz vorkompilierte Konfigurationsfolgen. Den Schwerpunkt dieser Arbeit bildet allerdings ein array-basierter Ansatz. Deshalb wird nachfolgend anhand einer einfachen Array-Architektur die Idee rekonfigurierbarer Architekturen dargelegt und erläutert. Den Abschluss des Kapitels bildet die Klassifikation rekonfigurierbarer Architekturen.

### 2.4.2. Funktionsweise

Wie bereits angedeutet, profitieren rekonfigurierbare Architekturen von hoher Anzahl verfügbarer Funktionseinheiten. Statt diese Einheiten fest zu verschalten, wird ein flexibles Kommunikationsnetzwerk genutzt, um bei Bedarf ausgewählte Funktionseinheiten zu verbinden. Jeder genutzten Funktionseinheit kann zugleich eine vorgegebene Funktion zugewiesen werden. Die resultierende Schaltung, bestehend aus aktiven Komponenten und den erstellten Verbindungen zwischen ihnen, realisiert eine Funktionsstruktur, die eine gewünschte Anwendung in der Hardware ausführen kann. Durch die parallele Natur der Hardware kann der Benutzer auf diese Weise von hohen Parallelitätsgraden der zugrunde liegenden Architektur profitieren, wobei die Komplexität seiner Applikationen durch den Umfang der verfügbaren Hardwareressource begrenzt wird.

### 2.4.3. Funktionale Primitive

Die Ausprägung rekonfigurierbarer Architekturen wird durch die Bitbreiten der verwendeten Funktionseinheiten bestimmt. Feingranulare Architekturen, wie FPGAs, arbeiten auf Bit-Ebene und sind in der Lage logische Funktionen umzusetzen. Zu diesem Zweck beherbergen FPGAs eine hohe Anzahl an LUTs (heute in der Altera Stratix V FPGA Serie: bis zu 718400 6:1 LUTs [69]). Hierbei handelt es sich um Speicher begrenzter Größe, die wie heute üblich logisch bis zu sechs Adressbits zur Adressierung des Inhalts besitzen. Die Anzahl der Adressbits bestimmt die Größe der LUTs und wird daher möglichst klein gehalten. Abbildung 13 zeigt ein Beispiel einer LUT mit sechs Eingängen und einem Register am Ausgang, das optional durch die Ansteuerung des Multiplexers genutzt werden kann. Zur Steigerung der funktionalen Flächendichte auf dem Chip werden die LUTs meist durch eine Kombination mehrerer kleinerer LUTs realisiert. So lassen sich beispielweise durch zwei 5:1 LUTs und einem Multiplexer ein 6:1 LUT realisieren. Die beiden kleineren LUTs lassen sich so unabhängig nutzen und bei Bedarf als vollständige 6:1 LUT konfigurieren.

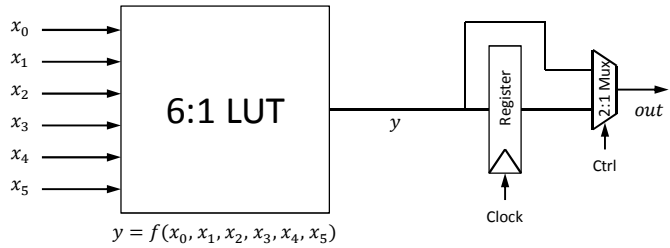
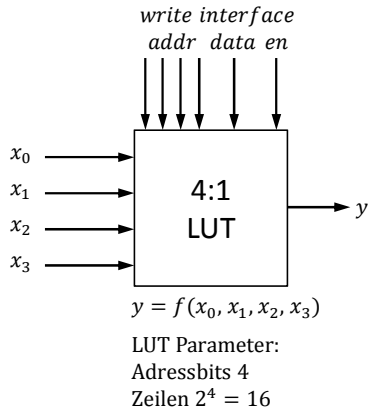



**Abbildung 13 LUT in der Anwendung mit abschließendem Register**

In Abhängigkeit des Inhalts des Speichers, lassen sich logische Funktionen umsetzen. Der Inhalt des Speichers wird dabei aus einer Wahrheitstabelle gewünschter boolescher Funktion abgeleitet. Die Adressbits werden in diesem Zusammenhang als Eingangsvariablen, die gelesenen Daten als Ausgangsvariable interpretiert. Besitzt der Speicher nur ein Ausgangsbit, so kann nur eine logische Funktion konfiguriert werden, die von maximal sechs Eingangsvariablen abhängen kann, siehe Abbildung 14. Bei mehreren Ausgangsbits kann damit die entsprechende Anzahl logischer Funktionen konfiguriert werden. In diesem Fall stehen diesen Funktionen dieselben Eingangsvariablen zur Verfügung, die in beliebiger Kombination genutzt werden können.




| x0 | x1 | x2 | x3 | y |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | *f*(0) |
| 1 | 0 | 0 | 0 | *f*(1) |
| 0 | 1 | 0 | 0 | *f*(2) |
| 1 | 1 | 0 | 0 | *f*(3) |
| 0 | 0 | 1 | 0 | *f*(4) |
| 1 | 0 | 1 | 0 | *f*(5) |
| 0 | 1 | 1 | 0 | *f*(6) |
| 1 | 1 | 1 | 0 | *f*(7) |
| 0 | 0 | 0 | 1 | *f*(8) |
| 1 | 0 | 0 | 1 | *f*(9) |
| 0 | 1 | 0 | 1 | *f*(10) |
| 1 | 1 | 0 | 1 | *f*(11) |
| 0 | 0 | 1 | 1 | *f*(12) |
| 1 | 0 | 1 | 1 | *f*(13) |
| 0 | 1 | 1 | 1 | *f*(14) |
| 1 | 1 | 1 | 1 | *f*(15) |

**Abbildung 14 Belegung einer LUT unter Verwendung einer Wahrheitstabelle und einer dedizierten Schreibschnittstelle**

Bei komplexen logischen Funktionen, die insbesondere von mehr Eingangsvariablen abhängen, als die implementierten LUTs an Adressbits bieten, lassen sich mehrere LUTs zusammenschalten, um die vollständige logische Funktion durch mehrstufige Logik zu realisieren. Zum Konfigurieren von LUTs wird das Schreib-Interface des verwendeten Speichers genutzt, welches parallel oder seriell ausgeführt sein kann. Der Speicher der LUTs selbst ist in diesem Fall der Konfigurationsspeicher, um die logische Funktion aufzunehmen.

In grobgranularen Architekturen werden statt einer Bit-Logik arithmetische Operationen auf Bitvektoren oder Worte angewendet. Hierzu werden ALUs eingesetzt, die einen vordefinierten Satz an arithmetischen Operationen besitzen. Im Gegensatz zu LUTs bestehen ALUs im einfachsten Fall aus kombinatorischen Netzlisten und können daher keine Konfigurationen speichern. Stattdessen muss jede ALU um externe Konfigurationsregister erweitert werden, siehe Abbildung 15. Diese werden an den Opcode-Port der ALU angeschlossen. Durch das Konfigurieren der Opcode-Register wird die ALU auf eine bestimmte Operation eingestellt. Diese Konfiguration ist statisch und bleibt für die Dauer der Konfigurationsausführung unverändert.

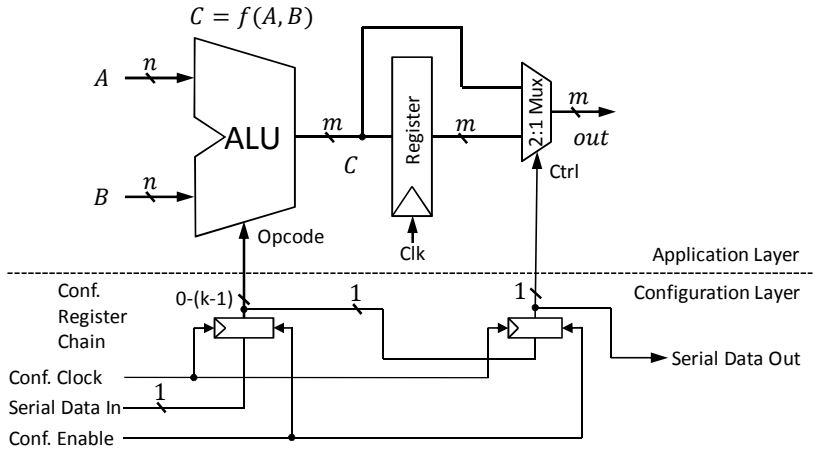



**Abbildung 15 Eine konfigurierbare ALU mit einem optionalen Ausgangsregister und der notwendigen Konfigurationsregister**

Sowohl LUTs als auch ALUs besitzen eine Signallaufzeit, die benötigt wird, bis angelegte Eingangsdaten stabile Ausgangsdaten liefern. Zusätzlich weisen auch die Leitungen zwischen diesen Funktionseinheiten eine Verzögerung bei der Signalausbreitung auf, die durch die Umladung der Leitungs- und Eingangskapazitäten hervorgerufen werden. Dies ist eine physikalische Eigenschaft der zugrunde liegenden Halbleiterschaltung, wodurch die maximale

Taktrate digitaler Schaltungen begrenzt wird. Ähnlich einer Pipeline können jedoch in bestimmten Abständen zwischen den Funktionseinheiten Register geschaltet werden, um abschnittsweise die Signallaufpfade zu verkürzen. Diese Register sind Teil der rekonfigurierbaren Architekturen und können von Applikationen optional genutzt werden. Abbildung 16 zeigt einen Datenflussgraphen, der die Abhängigkeiten der Operationen des zugrundeliegenden Polynoms aufzeigt. Er kann genutzt werden, um Abbildungen der Applikationen auf ein Array von Funktionseinheiten zu ermöglichen.

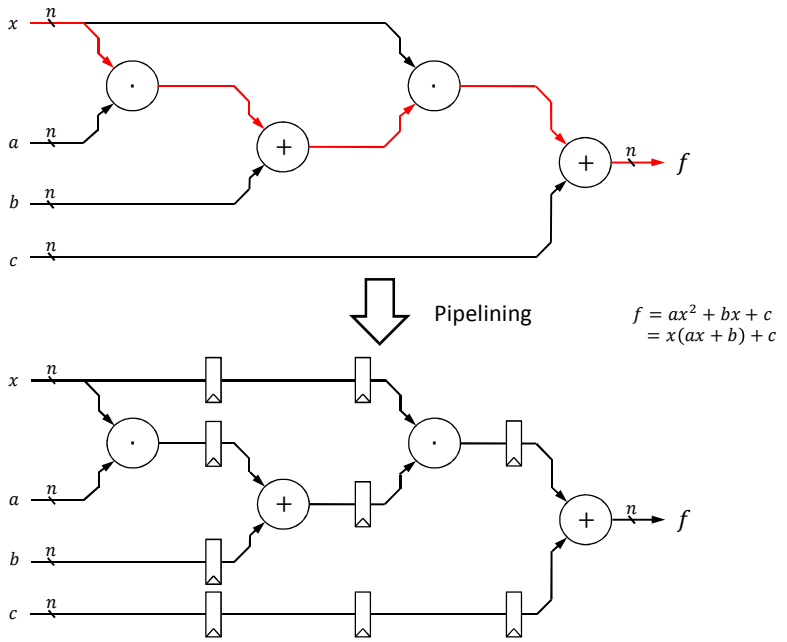



**Abbildung 16 Pipelining der ALUs am Beispiels eines quadratischen Polynoms wie es in ASICs oder rekonfigurierbaren Architekturen genutzt werden kann**

Neben den LUTs bzw. ALUs besitzen moderne rekonfigurierbare Architekturen dedizierte Einheiten zur effizienten Realisierung bestimmter Funktionen. Dazu gehören größere Speicherblöcke in Form von SRAMs, DSP-Blöcken und speziellen Hochgeschwindigkeitsschnittstellen. In besonderen Fällen werden auch High-Level-Funktionen direkt als dedizierte Module integriert, wie dies bei Forward Error Correction (FEC) oder Spread Data Path Units (SDPs) [70] der Fall ist. Im Falle der FPGAs macht die Integration dedizierter Einheiten besonders Sinn, da die Realisierung der Arithmetik mittels LUTs sehr aufwändig ist und viele Ressourcen benötigt. FPGAs bekommen

allerdings durch diese Erweiterungen wiederum einen spezialisierten Charakter, da die Anzahl und die tatsächliche Position dieser Einheiten auf dem FPGA anwendungsabhängig sind.

### 2.4.4. Kommunikationsnetze

Die Integration rekonfigurierbarer Einheiten in einer Rechenarchitektur erfordert ein flexibles Kommunikationsnetzwerk, um die Verbindungen zwischen den Funktionseinheiten zu realisieren. Idealerweise wird dabei die Möglichkeit geschaffen, beliebige Verbindungen zwischen den Funktionseinheiten zu ermöglichen. Dies ist insbesondere in array-basierten Architekturen notwendig, um keine Beschränkungen bzgl. der Erreichbarkeit bei der Programmierung der Architektur zu erhalten. Allerdings sind die Kosten für derartige Strukturen sehr hoch, so dass Forscher sich auf spezialisierte Lösungen konzentrierten, um die Kosten der Flexibilität gering zu halten.

#### 2.4.4.1. Kommunikationstopologien

Anstelle der Realisierung eines homogenen Netzwerks erlaubt die Aufteilung der Kommunikationsressourcen in globale und lokale Verbindungen eine Verbesserung der Laufzeiten und Reduktion der Kosten, siehe Abbildung 17. Die lokalen Verbindungen werden hierbei zur Anbindung naheliegender Nachbarzellen untereinander verwendet. Dadurch ist es nicht notwendig, benachbarte Zellen über das globale Kommunikationsnetz anzubinden und somit lässt sich die Anzahl der Leitungen dieser Kommunikationsressource reduzieren. Die Ausprägung der lokalen Verbindungen lässt sich weiterentwickeln und so beispielsweise nicht nur benachbarte Zellen anbinden, sondern weiter entfernte Zellen koppeln. Die Gefahr dieser Technik ist die resultierende Anwendungsabhängigkeit der Architektur. Die Kommunikationsstruktur, die dabei entsteht, weist eine Charakteristik auf, die der Zielanwendung entgegenkommen muss, damit diese effizient genutzt werden kann. Diese Charakteristik spiegelt sich im Datenflussgraphen einer Anwendung wieder.

Das globale Kommunikationsnetzwerk hat diese Einschränkungen nicht. Es bietet eine höhere Flexibilität bei gleichzeitig höheren Kosten. Ein idealer Aufbau der globalen Topologie würde jede beliebige Kombination von Verbindungen zwischen den Funktionseinheiten erlauben, allerdings wäre der Ressourcenbedarf enorm, mit der Anzahl der Funktionseinheiten stark ansteigen und den Ressourcenbedarf auf dem Chip dominieren. Insbesondere im Zentrum einer derartigen Architektur, wo sich die meisten Verbindungen kreuzen, wäre die notwendige Dichte nicht tragbar. Darüber hinaus stellt die Länge

dieser Leitungen als auch die Anzahl der Verbraucher ein Problem hinsichtlich der resultierenden Lautzeiten.

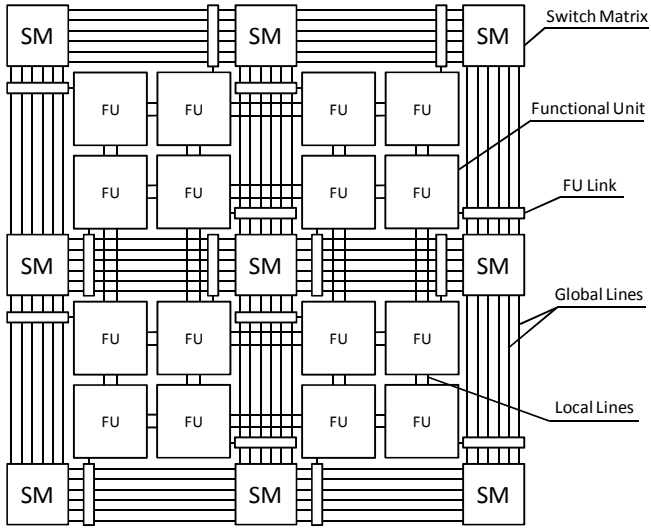



**Abbildung 17 Einfache array-basierte rekonfigurierbare Architektur mit globalen und lokalen Verbindungen, Schaltmatrizen und Funktionseinheiten.**

In anwendungsspezifischen Architekturen, wie dem Maia Chip der Universität Berkley [80], werden auch inhomogene Kommunikationstopologien genutzt, um die Effizienz der Architektur zu steigern. In diesem Fall wurden Funktionseinheiten in Form unterschiedlich ausgeprägter Makroblöcke bzgl. ihrer Größe und Funktion verwendet. Dadurch ist auch ein inhomogenes Kommunikationsnetzwerk notwendig, welches die Anwendungscharakteristik der Architektur wiederspiegelt.

### 2.4.4.2. Netzwerkressourcen

Die notwendigen Komponenten zur Realisierung der Netzwerktopologie wären zum einen Schaltmatrizen (SM) und zum anderen Link Module. Die Schaltmatrizen werden benutzt, um globale Leitungen, die horizontal und vertikal verlaufen, im Array zu routen. Dies kann im einfachsten Fall durch Multiplexer geschehen, was allerdings eine teure Lösung darstellt. Günstiger wäre hier die Nutzung von Transmission-Gates, wie sie in der Abbildung 18 dargestellt sind.

Hierbei handelt es sich um eine Schaltung, die aus je einem p-MOS und einem n-MOS Transistor aufgebaut ist. Der Gate-Eingang der Transistoren ist

dabei der Steuereingang, Source- und Drain-Eingänge werden an die Signalleitungen angeschlossen. Die Spannung am p-MOS-Gate wird zusätzlich durch einen Inverter gedreht, damit beide Transistoren zugleich leiten bzw. sperren. In realen Schaltungen ist ein Inverter nicht notwendig, da die Konfigurationsregister üblicherweise beide Signale liefern können.

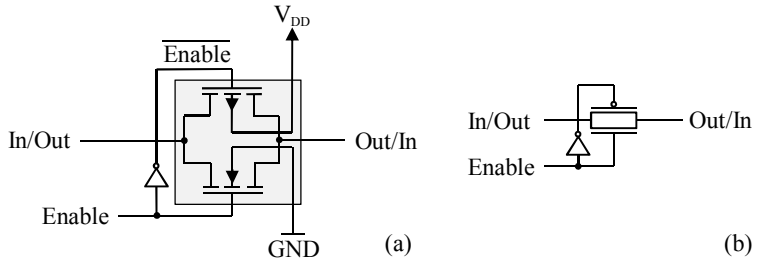



**Abbildung 18 Transmission-Gate als Transistorschaltung (a) und Symbol (b).**

Zwei Transistortypen sind notwendig, um sowohl eine logische Eins als auch eine logische Null transportieren zu können. Die resultierende Schaltmatrix basierend auf den Transmission-Gates kann wie in Abbildung 19 dargestellt aufgebaut werden, hängt aber im Detail von den Anforderungen der Architektur ab.

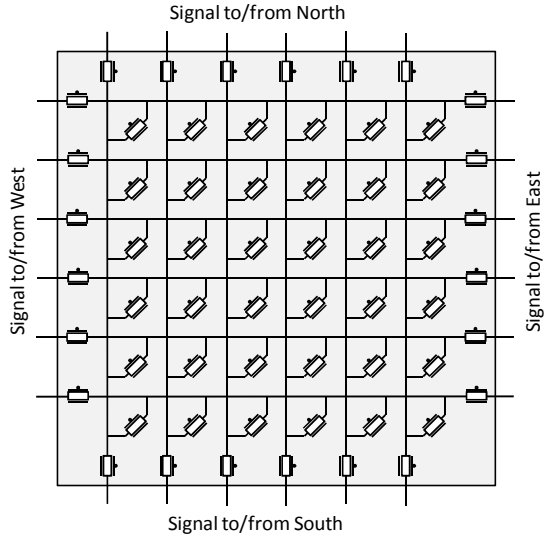



**Abbildung 19 Schaltmatrix für die globale Kommunikationstopologie**

Mit der dargestellten Schaltmatrix lassen sich zwei grundsätzliche Funktionen realisieren. Zum einem lassen sich Signale von einer Leitung auf eine andere leiten und damit Routing in einem großen Netzwerk auch mit mehreren Matrizen umsetzen. Zum anderen lassen sich mit den Transmission-Gates an den Eingängen der Matrizen die Weiterleitung der Signale unterbinden und damit eine Segmentierung der Leitungen erreichen. Dies geschieht durch das Sperren der betreffenden Transmission-Gates. Hiermit kann die Leitung im nächsten Abschnitt unabhängig Daten transportieren. Zusätzlich werden die Leitungskapazitäten der einzelnen Segmente entkoppelt, was der Signallaufzeit zugutekommt. Die genaue Ausprägung der Schaltmatrizen ist architekturabhängig. Bei modernen FPGAs werden beispielsweise zur Einsparungen der Ressourcen mehrere Segmente durch Schaltmatrizen zu Long-Lines zusammengefasst. Bei Aktivierung eines Treibers an einer derartigen Leitung, wird die komplette Leitung für die Signalübermittlung genutzt. Durch die Schaltmatrix wird jedoch nicht verhindert, dass Kurzschlüsse durch mehrere aktive Treiber möglich sind. Diese Aufgabe wird der Design-Software überlassen, die zum Synthesezeitpunkt der Konfiguration die notwendigen Prüfungen vornehmen muss.

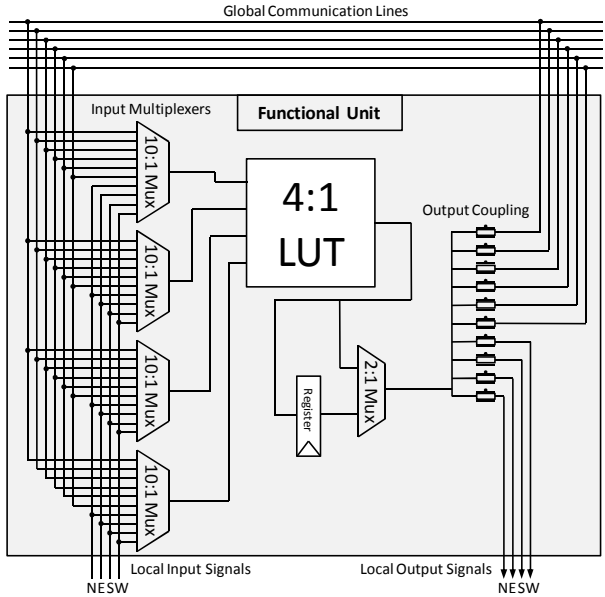



**Abbildung 20 Einfacher Aufbau einer Funktionseinheit mit Anbindung an das globale Kommunikationsnetzwerk**

Mit gleichen Ressourcen lassen sich Funktionseinheiten an das Kommunikationsnetzwerk anbinden. Abbildung 20 zeigt eine einfache Funktionseinheit, die über Transmission-Gates ihr Ausgangssignal in das globale Kommunikationsnetzwerk einspeist, während Eingangssignale der integrierten LUT mittels Multiplexer selektiert werden. Es sei an dieser Stelle angemerkt, dass der Einsatz von Transmission-Gates nicht ohne Signalregeneration auskommt und somit bei langen Signalwegen treibende Komponenten wie Verstärker eingesetzt werden müssen. Der Grund dafür ist der Widerstand der Transmission-Gates, der im leitenden Zustand zwar niedrig ist, jedoch weiterhin für einen Spannungsabfall sorgt. Darüber hinaus ist der Leitungswiderstand nicht zu vernachlässigen, der für zusätzlichen Spannungsabfall verantwortlich ist. Normalerweise werden in diesem Fall Gatter mit unterschiedlichen Treiberstärken zur Signalregeneration genutzt.

Neben den Signalnetzwerken müssen auch Clock-Netze in einer rekonfigurierbaren Architektur integriert sein. Speziell in großen feingranularen Arrays werden mehrere parallele Clock-Netze integriert, damit der Anwender komplexe Designs mit mehreren Clock-Domains entwerfen kann. Dazu werden zusätzlich Komponenten wie PLLs (Phase Locked Loop), Clock-Multiplexer und Treiber für die Taktbäume integriert, um in der Lage zu sein, designspezifische Frequenzen zu erzeugen. Integrierte PLLs vereinfachen das PCB-Design, weil die Anzahl notwendiger Taktquellen auf der Mutterplatine reduziert wird.

#### 2.4.4.3. Granularität der Kommunikationsnetze

Ähnlich zu Datenpfaden unterschiedlicher Granularität, werden auch Kommunikationsnetze mit entsprechender Anzahl einzelner oder paralleler Leitungen entworfen. Es wird in diesem Zusammenhang zwischen den feingranularen und grobgranularen Ansätzen unterschieden. Im zweiten Fall kommt es zusätzlich darauf an, für welche Wortbreite die Datenverarbeitung der Architektur ausgelegt ist. Welche Möglichkeiten der Signaleinspeisung und Abgriffe auf und von globalen Signalen bestehen, wurde bereits in letzten Abschnitten dargelegt. Weiterführend soll hier lediglich die Anwendung auf die verschiedenen Granularitäten aufgezeigt werden.

Feingranulare Signalleitungen haben die Eigenschaft, dass sie einzeln und unabhängig zwischen Quellfunktionseinheit und der Zielfunktionseinheit geroutet werden. Zur Beschreibung der Route werden pro Pfad und Bit separate Konfigurationsregister genutzt. Auf der Ressourcenseite werden genauso viele Transmission-Gates benutzt wie Konfigurationsregister. Im Falle der grobgranularen Architekturen, wird das Routing vektorbasiert gesteuert, d.h. bei einer 16-bit-Architektur genügt ein Konfigurationsregister um 16 Transmission-

Gates zu steuern. Die zusätzlichen Kosten für eine feingranulare Architektur sind damit enorm. Bei Heranziehen der Transistorzahl beider Komponenten als Gewichtung, ergibt sich eine Gewichtung der Transmission-Gates mit $T = 2$ und der Konfigurationsregister mit $R = 14$, entsprechend der TSMC Standardzellen Technologie. Beim Routing eines Bitvektors der Größe $G = 16$ durch eine Schaltstelle ergeben sich damit für 16 feingranulare Signale Gesamtkosten ($Kosten_f$) von

$$Kosten_f = G \cdot (T + R) = 16 \cdot (2 + 14) = 256,$$

während die Schaltkosten $(Kosten_g)$ für grobgranulare Signale mit

$$Kosten_g = G \cdot T + R = 16 \cdot 2 + 14 = 46$$

deutlich geringer ausfallen. Trotz der offensichtlichen Kostenvorteile grobgranularer Schaltstrukturen, büßen diese an Flexibilität ein, da ein Vektor immer nur als Ganzes und vollständig geroutet werden muss. Werden weniger Bits für eine programmierte Funktion benötigt, so muss dafür dennoch der vollständige Vektor von der Quelle bis zum Ziel durchgestellt werden. Feingranulare Kommunikationsnetze sind hier in der Lage angepasste Kommunikationsbandbreiten zur Verfügung zu stellen. Sie sind jedoch in der Realisierung, wie am Beispiel zu sehen, mit Abstand teurer grobgranulare Lösungen.

Es muss allerdings angemerkt werden, dass die verwendete Betrachtung stark vereinfacht ist und die tatsächlichen resultierenden Kosten anhand einer realen Umsetzung am Einzelfall bewertet werden muss.

### 2.4.5. Rekonfigurationstechniken

Die Bereitstellung von Konfigurationsregistern an Funktionseinheiten wie Multiplexern, Transmission-Gates oder ALUs ist eine teure Vorgehensweise, wie am Routing-Beispiel gezeigt wurde. Dies ist allerdings notwendig, wenn nach der Herstellung des Bausteins der Anwender in der Lage sein soll, die Funktion nach Bedarf zu ändern. In diesem Zusammenhang wird von Konfiguration bzw. bei einer Änderung bestehender Konfiguration von einer Rekonfiguration gesprochen. Im Folgenden sollen verschiedene Rekonfigurationstechniken vorgestellt werden.

#### 2.4.5.1. Statische Rekonfiguration

Neben der Umsetzung der konfigurierbaren Bausteine mittels Konfigurationsregistern existieren auch Lösungen basierend auf Fuses (Sicherungen). Dabei handelt es sich um einmal beschreibbare Strukturen, die durch die Verwendung einer Sicherung ihre Informationen auch nach dem ausschalten be-

halten. In diesem Zusammenhang wird von statischer Konfiguration gesprochen. Derart programmierte Bausteine sind nach dem Einschaltvorgang vollfunktionsfähig und benötigen kein Laden der Konfiguration. Weitere Vorteile sind schnellere Architektur durch das kompaktere Design, sowie geringere Verlustleistung. Als Nachteil wird die nach einmaligem Laden der Konfiguration festgelegte Funktion angesehen. Ein Beispiel für fuse-basierte Architekturen stellt die HardCopy-ASIC-Serie der Firma Altera dar.

#### 2.4.5.2. Dynamische Rekonfiguration

Die Verwendung von Registern zur Speicherung der Konfigurationsdaten und die Verbindung der Registerausgänge mit den Steuereingängen der Funktionseinheiten ermöglicht die Änderung der Funktion des Bausteins zum Betriebszeitpunkt. Die notwendigen Daten für die Konfiguration müssen vom Anwender erstellt oder von einem IP-Anbieter (Intellectual Property) erworben werden. Das einmal übertragene vorberechnete Datenmuster auf die Konfigurationsregister, bestimmt die Funktion des Bausteins. Durch die Fähigkeit des Bausteins immer wieder neue Konfigurationen aufzunehmen und damit seine Funktion zu ändern, wird in diesem Zusammenhang von dynamischer Rekonfiguration gesprochen.

Je nach Größe der Konfiguration und der Dauer der Rekonfiguration ergeben sich wieder neue Anwendungen, die analog zur Befehlsverarbeitung in einem Prozessor in einer sogenannten Konfigurationssequenz (Configuration Sequencing) resultieren können. Dabei werden Konfigurationen verhältnismäßig schnell auf den Baustein geladen, eine Funktion als Teil einer größeren Applikation ausgeführt und auf die darauf folgende Konfiguration gewechselt. Dieser Vorgang wiederholt sich in einer fortwährenden Sequenz, wie dies bei Prozessoren mit Instruktionen geschieht. Um diese Funktion zu ermöglichen, muss die Größe der Konfigurationen im Bereich von wenigen Kilobytes gehalten werden, damit die Rekonfigurationen im Bereich von wenigen bis maximal einigen hundert Mikrosekunden erfolgen können. Abzüglich der Laufzeit der zwischenzeitlichen Berechnungen, werden dadurch pro Sekunde tausendfach Rekonfigurationen ermöglicht und eine echte Konfigurationssequenzierung wird ermöglicht.

Eine Ausnahme bei den rekonfigurierbaren Bausteinen stellen die CPLDs (Complex Programmable Logic Devices) dar. Hierbei handelt es sich um rekonfigurierbare Architekturen, die anstelle von Konfigurationsregistern auf flash-basierte Techniken nutzen. Dadurch wird es möglich, den Inhalt der Konfigurationen nach einmaliger Programmierung dauerhaft zu speichern und nach dem Einschalten des Bausteins unmittelbar zur Verfügung zu stellen. Produkte für derartige Architekturen stellen alle großen Hersteller zur Verfü-

gung, jedoch sind insbesondere kleinere Marktteilnehmer wie Actel [14] auf diese Technik spezialisiert.

### 2.4.5.3. Multikontext-Rekonfiguration

Die Kombination der dynamischen Rekonfiguration mit einer Konfigurationssequenz legt eine weitere Technik der Rekonfiguration nahe, die sogenannte Multikontext-Rekonfiguration. Anstelle nur eines Registersatzes für die Konfigurationsregister (Abbildung 21) lassen sich zwei oder mehr Registersätze implementieren. Unter einem Kontext wird in diesem Zusammenhang ein vollständiger Registersatz für eine Konfiguration verstanden. Besitzt also eine Architektur mehrere Kontexte, so wird von einer Multikontext-Rekonfiguration gesprochen.

Wird nun zur Laufzeit einer der verfügbaren Registersätze ausgewählt, so steht die darin enthaltene Konfiguration unmittelbar zur Verfügung und die Applikation kann ausgeführt werden. Währenddessen können die übrigen Registersätze mit weiteren Konfigurationsdaten gefüllt werden. Bereits bei Verwendung von nur zwei Registersätzen lässt sich durch diese Vorgehensweise die Rekonfigurationszeit zum Großteil eliminieren, siehe Abbildung 22. Die Effektivität hängt in diesem Fall davon ab, inwieweit die Rekonfigurationszeiten unter den Ausführungszeiten der Anwendungen liegen, siehe Rekonfigurationszeit C2 zur Kalkulationszeit C1.

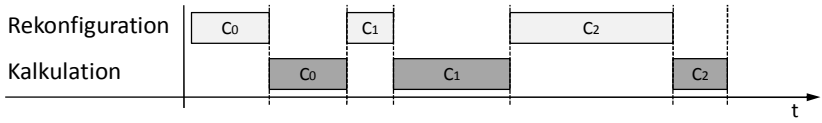



**Abbildung 21 Rekonfiguration mit einem Kontext mit wechselseitiger Rekonfiguration und Kalkulation der Anwendungen.**

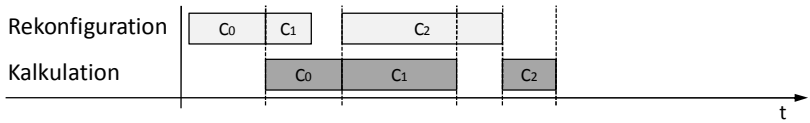



**Abbildung 22 Rekonfiguration mit zwei oder mehr Kontexten mit überlappender Rekonfigurationszeit und Kalkulation.**

Die beschriebene Anwendung der Multikontext-Rekonfiguration stellt allerdings nur einen Anwendungsfall dar, den diese Technik ermöglicht. Anstelle der Nutzung der einzelnen Kontexte pro Anwendung, lassen sich auch mehrere Kontexte innerhalb einer Anwendung selektieren. Diese Selektion kann

sogar von Takt zu Takt durch die Anwendung erfolgen, was für die vorliegende Architektur zu großen Vorteilen hinsichtlich der hinzugewonnen Flexibilität führt. Die Steuerung der Kontextauswahl kann entweder statisch durch eine vorprogrammierte Sequenz oder dynamisch in Abhängigkeit der Zwischenergebnisse der laufenden Applikation erfolgen. Damit lassen sich bedingte Konstrukte wie if-then-else oder Schleifen effektiv abbilden und damit zusätzlich Fläche einsparen, weil die genutzte Fläche mehrere Funktionen implementiert. In Architekturen ohne Multikontext-Unterstützung müsste zu diesem Zweck deutlich mehr Fläche aufgewendet werden, da beispielsweise sowohl der if- als auch der else-Pfad separate Funktionseinheiten und damit Flächen benötigen würden.

Üblicherweise werden Multikontext-Konfigurationen global in der vorliegenden Architektur angewendet. Das bedeutet, dass die Umschaltung des Kontextes für alle Funktionseinheiten zugleich erfolgt. Denkbar ist hier auch eine lokale Kontextsteuerung, welche die Umschaltung in einem begrenzten Bereich der Architektur vornimmt. Ein Beispiel für eine derartige Architektur stellt die HoneyComb-Architektur dar, die in Kapitel 4 vorgestellt wird und auf der der Hauptfokus dieser Ausarbeitung liegt.

Es ist relativ leicht nachvollziehbar, dass die Kosten für eine Multikontext-Erweiterung vergleichsweise hoch sind. Daher ist es auch nur sinnvoll ihre Anwendung bei grob-granularen Architekturen in Betracht zu ziehen. Bei diesen Architekturen ist die Zahl der Rekonfigurationsregister bereits reduziert, so dass eine Verdopplung oder gar Vervielfachung der Registersätze nicht zu einer Kostenexplosion führt. Die Konfigurationsgrößen bei feingranularen und grobgranularen Architekturen mit einem Kontext bewegen sich im Bereich von Megabytes im Falle von FPGAs und von wenigen hundert Kilobytes im Falle der grobgranularen Architekturen. Der Unterschied dieser Größenordnung beschränkt diese Technik damit auf grob-granulare Architekturen.

#### 2.4.5.4. Partielle Rekonfiguration

Die ersten rekonfigurierbaren Architekturen waren nur imstande die gesamte Architektur auf einen Schlag zu konfigurieren, wie dies bei den ersten FPGAs der Fall war. Dieser Umstand hat den Nachteil, dass ungenutzte Ressourcen der Architektur brachliegen, falls die ausgeführte Anwendung nicht die passende Größe aufweist und damit werden die Ressourcen nur unvollständig ausgenutzt. Je nach prozentualer Ausnutzung, wäre es daher sinnvoll beispielsweise eine weitere Anwendung auf die freien Ressourcen zu konfigurieren und damit die vorliegende Leistung der Architektur effizienter auszunutzen. Zu diesem Zweck wurde die partielle Rekonfiguration eingeführt. Eine Technik, die teilweise Rekonfiguration der Ressourcen erlaubt, die zu Anfang

der partiellen Rekonfiguration bei den Vertex 2 FPGAs von Xilinx [10] spaltenweise erfolgte und inzwischen die Kachel-Granularität erlaubt.

Die Anwendung der Technik ist allerdings mit einigen Schwierigkeiten verbunden. So müssen die Anwendungen, die gleichzeitig konfiguriert und ausgeführt werden, gemeinsam entwickelt worden sein. Dies ist notwendig, um Ressourcenkonflikte zwischen den Anwendungen zu vermeiden. Abbildung 23 zeigt ein einfaches Beispiel für einen Routing-Konflikt, der entstehen kann, falls zwei Anwendungen ausgeführt werden und auf dieselbe globale Leitungsressource konfiguriert wurden. Dieser Zustand kann vermieden werden, wenn das Synthese-Tool beide Anwendungen kennt und dadurch diese Konflikte auflöst. Moderne Design-Tools für FPGAs unterstützen diese Technik von Hause aus, wie dies beispielsweise bei der Software-Suite Quartus II der Firma Altera der Fall ist. Der Nachteil liegt klar auf der Hand: beide Anwendungen müssen aus einer Feder stammen und somit Kenntnis von einander haben.

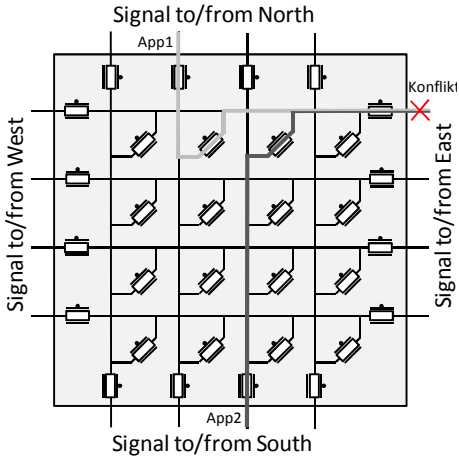



**Abbildung 23 Veranschaulichung des Konflikt beim Versuch die Anwendungen App1 und App2 auf dieselbe Routingressource zu konfigurieren.**

Es ist allerdings auch möglich ohne Tool-Unterstützung partielle Rekonfiguration mittels applikationsspezifischer Funktionalitäten zu realisieren. Zu diesem Zweck werden auf dem Applikationslevel Funktionen bereitgestellt mit vorspezifizierten Schnittstellen für partielle Konfigurationen. Dies erfolgte zunächst FPGA-bedingt spaltenweise [71] und später war sogar zweidimensionale partielle Rekonfiguration möglich, wie durch die Arbeit von Michael Hübner gezeigt wurde [72].

### 2.4.6. Klassifikation rekonfigurierbarer Architekturen

Analog zur Klassifikation der Prozessoren nach Michael J. Flynn ist es sinnvoll, rekonfigurierbare Architekturen nach der Art ihres Aufbaus und ihrer Funktionalität einzuteilen. Im Falle der Prozessoren erfolgt die Klassifikation nach der Methode der Befehlsausführung. Das ist in diesem Fall eine sinnvolle Einteilung, da Prozessoren von ihrer Grundfunktionalität her für die Befehlsausführung entwickelt und optimiert werden. Zwischen Prozessorfamilien bestehen zwar Unterschiede hinsichtlich des Befehlssatzes, der wesentliche Aspekt hier ist jedoch die Art der Befehlsausführung. Michael J. Flynn vereinfachte diese Sichtweise auf vier Klassen, die eine Einteilung hinsichtlich der eingehenden Befehle und der verarbeiteten Datensätze vornimmt. Auf der einen Seite werden Befehle entweder einzeln oder parallel ausgeführt, auf der anderen Seite berechnen diese Befehle entweder ein oder mehrere Datenworte. Da die Hauptfunktionalität der Prozessoren in der Verarbeitung von Instruktionen und der zugrunde liegenden Datensätze besteht, eignet sich diese Klassifikation für Prozessoren sehr gut, um den Charakter der Architektur festzustellen. In der Tat ist die Komplexität der Prozessoren viel zu hoch, so dass diese Klassifikation einen guten Kompromiss bzgl. des Abstraktionslevels darstellt.

Im Falle rekonfigurierbarer Architekturen lässt sich die Klassifikation nicht so einfach vornehmen. Während Prozessoren eine Folge von Befehlen verarbeiten, besitzen rekonfigurierbare Architekturen eine Vielfalt verschiedener hierarchischer und funktionaler Ausprägungen, die einen gravierenden Einfluss auf den resultierenden Charakter der Architektur haben. Daneben bestimmt die Kommunikationstopologie entscheidend die Bandbreite unterstützter Applikationen. Ebenso von entscheidender Bedeutung ist die eingesetzte Rekonfigurationstechnik, welche die Art und Weise der Nutzung der zugrunde liegenden Architektur entscheidend beeinflusst. Offensichtlich sind die Freiheitsgrade im Design rekonfigurierbarer Architekturen sehr umfangreich, so dass die eigentliche Klassifikation alle genannten Eigenschaften umfassen muss. Folgende Tabelle fasst die vier genannten Parameter zusammen und präsentiert die weitere Unterteilung, wie sie bereits in vorangegangen Abschnitten vorgestellt wurden.

**Tabelle 9 Klassifizierung rekonfigurierbarer Architekturen nach den wichtigsten Gesichtspunkten.**

<table>
<tr><th colspan="2">Hierarchie</th><th>Datenpfade</th><th colspan="2">Kommunikation</th><th colspan="3">Rekonfiguration</th></tr>
<tr><td colspan="2">prozessorartig</td><td>feingranular</td><td colspan="2">homogen - universell</td><td colspan="3">statisch</td></tr>
<tr><td rowspan="2">array-basiert</td><td>homogen</td><td>grobgranular</td><td rowspan="3">inhomogen</td><td>global/lokal</td><td rowspan="4">dynamisch</td><td colspan="3">Single-Context</td></tr>
<tr><td>hierarchisch</td><td>multigranular</td><td rowspan="2">anwendungs-spezifisch</td><td rowspan="3">Multi-Context</td><td colspan="2">Device -Level</td></tr>
<tr><td colspan="2">irreguläre Module</td><td rowspan="3"></td><td rowspan="2">Applicati-on-Level</td><td>global</td></tr>
<tr><td colspan="2" rowspan="2"></td><td>hierarchisch</td><td>partiell</td></tr>
<tr><td colspan="2"></td><td colspan="3">Hyper-Context</td></tr>
</table>

### 2.4.7. Programmierung rekonfigurierbarer Architekturen

Die technischen Unterschiede der vorgestellten Techniken, sowohl aus dem Prozessorbereich als auch aus dem Umfeld rekonfigurierbarer Architekturen, stellen sich für den Anwender am stärksten durch die Art der Programmierung dar. In den folgenden Abschnitten sollen diese Programmiertechniken erläutert werden.

### 2.4.8. Abbildung in die Zeit

Prozessoren folgen in ihrer Programmierung dem Menschen bekannter Denkweise, indem Probleme in Teilschritte zerlegt und auf eine Folge von Befehlen abgebildet werden. Dieser Ansatz ist für den Programmierer, abgesehen von der Kenntnis der Syntax/Semmantik und den Möglichkeiten der verwendeten Programmiersprache, sehr intuitiv und leicht erlernbar. Durch die Ausführung der Befehle in zeitlicher Abfolge wird auch von Abbildung in die Zeit gesprochen. Obwohl die interne Ausführung des erstellten Programms nicht immer nur einem strikt eindimensionalen Faden folgt, wie dies bei superskalaren Prozessoren mit der Out-Of-Order-Execution möglich ist. Für den Programmierer wird hier der Schein weiterhin gewahrt, dass seine Programme wie beabsichtigt strikt nach der vorgegebenen Befehlsreihenfolge ausgeführt werden.

### 2.4.9. Abbildung in die Fläche

Rekonfigurierbare Architekturen hingegen fügen einen weiteren wichtigen Aspekt in die Programmiermethodik ein: die Parallelität. Es reicht nicht mehr zeitliche Teilschritte zur Ausführung der Anwendung zu kennen, es muss auch die unter Umständen große Anzahl paralleler Einheiten ausgenutzt werden.

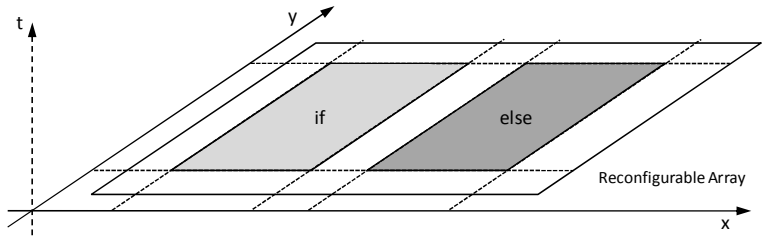



**Abbildung 24 Flächendisjunkte Abbildung einer if-then-else-Struktur in die Fläche unter Verwendung eines Kontextes.**

Dazu reicht es nicht mehr einer eindimensionalen Denkweise auf der Zeitachse zu folgen, sondern es bedarf einer zweidimensionalen Denkweise, die der Fläche der zugrunde liegenden Architektur gerecht wird. Jedoch auch hier ist der Faktor Zeit in der Flächenstruktur durch die Verknüpfungen zwischen den Funktionseinheiten kodiert und nicht wegzudenken. Abbildung 24 veranschaulicht die Abbildung einer if-then-else-Struktur in die Fläche einer Architektur mit einem Kontext. Hier muss jeder Funktion eine eigene Ressource zugewiesen werden.

Die Beschreibung der Funktion erfolgt nicht mehr in einer Hochsprache wie C/C++ mit ihrem zeitlichen Charakter. Vielmehr werden Hardwarebeschreibungssprachen genutzt, um den erforderlichen Parallelitätsgrad wie im Falle der FPGAs zu erreichen.

### 2.4.10. Abbildung in die Fläche und Zeit

Die vorgestellte Multikontext-Technik für rekonfigurierbare Architekturen fügt einen weiteren Freiheitsgrad in die Programmierung ein. Neben der bereits beschriebenen Abbildung in die Fläche, kommt durch die Umschaltung der Kontexte innerhalb einer Applikation der Faktor Zeit auf Konfigurationslevel hinzu. Dadurch wird es möglich die Funktion der vorhandenen Funktionseinheiten auf zeitlicher Achse sehr schnell zu ändern, sodass es möglich wird, die Fläche effizienter zu nutzen. Allerdings steigt die Komplexität des Programmiermodells enorm an, da der Anwender in dieser Situation dreidimensional denken muss.

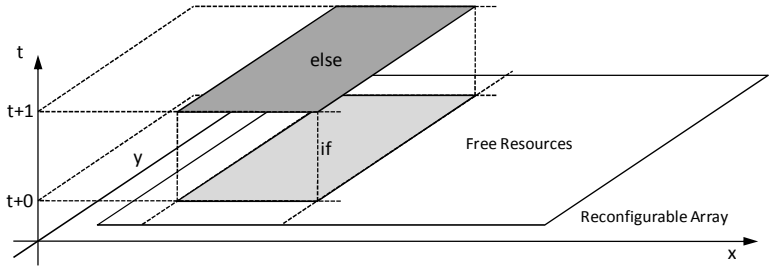



**Abbildung 25 Zeitlich überlappende Abbildung einer if-then-else-Struktur unter Verwendung von zwei Kontexten**

Realistisch betrachtet lassen sich dadurch per Hand keine komplexen Anwendungen abbilden, da dies aus Zeitgründen schlicht zu teuer wäre. Vielmehr wäre es notwendig auf Tool-Unterstützung zu setzen, um die eigentliche Abbildung automatisiert vorzunehmen. Die Entwicklung dieser Tools ist jedoch

für praktische Anwendungen noch nicht weit genug fortgeschritten und ist daher weiterhin Gegenstand der Forschung.

Vorteilhaft wirkt sich diese Technik auf die Auslastung der Hardware aus. Der Anwender ist damit imstande sehr fein den Parallelitätsgrad der Abbildung zu steuern, in dem er seine Anwendung je nach Anforderung mehr in die Fläche oder in die Zeit verschiebt.

Abbildung 25 stellt den Zusammenhang zwischen der Flächenabbildung und zeitlicher Umschaltung der Kontexte dar. Je nachdem welcher Teil der if-then-else-Struktur ausgeführt werden soll, schaltet die Architektur auf den entsprechenden Kontext um. Die Darstellung ist zur Veranschaulichung sehr einfach gewählt. In realen Anwendungen wäre eine Umschaltung mit feineren Stufen der beteiligten Funktionseinheiten von Vorteil, um einen Leerlauf der Funktionseinheiten bei blockweisem Umschalten zu vermeiden. Es ist auch erkennbar, dass die Architektur mittels dieser Technik mehr freie Ressourcen besitzt, die für weitere Applikationen nutzbar wären.

## 2.5. Zusammenfassung

Wie in diesem Kapitel beschrieben, haben sich die Konzepte für moderne Prozessoren mit großen Schritten weiterentwickelt. Ausgehend von einfachen Rechenkernen, sind moderne Prozessoren in der Lage, relativ effizient den Programmcode zu verarbeiten. Zu diesem Zweck besitzen sie sehr schnelle superskalare Pipelines in Form paralleler Funktionseinheiten, 64-bit Datenverarbeitung mit großen Adressräumen, SIMD-Befehlssätze, mehrstufige Caches und Prozessoren mit mehreren Prozessorkernen. Wie im nächsten Kapitel anhand realer Beispiele aufgezeigt wird, sind diese Lösungen noch nicht am Ende ihrer möglichen Leistungsfähigkeit angelangt. Dies lässt sich anhand umgesetzter Applikationen als ASICs oder FPGAs demonstrieren.

Zu diesem Zweck wird die Forschung im Bereich rekonfigurierbarer Systeme speziell im grobgranularen Segment vorangetrieben. Ein möglicher jedoch einfacher Aufbau einer rekonfigurierbaren Architektur wurde in diesem Kapitel eingeführt und die notwendigen Konzepte für die Realisierung einer derartigen Architektur vorgestellt. Die Abbildung von Applikationen soll möglichst hardwarenah auf rekonfigurierbare Architekturen erfolgen, damit vom parallelen Charakter der implementierten Funktionseinheiten profitiert werden kann. Die Grundidee eifert dem konzeptionellen Ansatz der ASIC-Implementierungen nach, die für eine vorliegende Anwendung eine Hardwarestruktur schaffen und somit eine Abbildung der Anwendungen in die Fläche vornehmen. Die dabei verwendeten Funktionseinheiten werden in derartigen Strukturen gleichzeitig genutzt und bieten damit den Vorteil paralleler Datenverarbeitung.

In diesem Kapitel vorgestellte Konzepte für rekonfigurierbare Architekturen stellen nur den ersten Schritt zur Realisierung einer rekonfigurierbaren Architektur dar. Bereits die Kombination der vorgestellten Komponenten mit Funktionseinheiten unterschiedlicher Granularitäten, globalen/lokalen Kommunikationsstrukturen und unterschiedlichen Rekonfigurationstechniken lassen eine große Anzahl möglicher Ansätze zu. Wie im nächsten Kapitel vorgestellt wird, ist bis heute bereits eine große Anzahl rekonfigurierbarer Architekturen entstanden. Die Vielfalt dieser Architekturen erstreckt sich von einfachen arraybasierten Ansätzen bis hin zu komplexen fraktalen Strukturen. Ebenso ist oftmals die Ausprägung der Architekturen mit applikationsspezifischen Merkmalen verbunden, was jedoch die Anwendbarkeit der Architekturen auf einen vorgegebenen Satz an Applikationen einschränkt.

# 3. Stand der Technik

Moderne Rechensysteme zur Datenverarbeitung umfassen ein breites Spektrum an unterschiedlichen Ansätzen, die letztendlich alle den Zweck verfolgen, effizient und möglichst performant ihre Zielanwendungen zur Ausführung zu bringen. Den Kern dieser Systeme stellen Prozessoren dar, die auch hauptsächlich dafür verantwortlich sind, die dem System zugedachten Rechenaufgaben zu erledigen. Die Grundlagen der Prozessortechnik wurden bereits im letzten Kapitel diskutiert und werden im vorliegenden Kapitel nur anhand aktueller Lösungen besprochen. Diese Ausarbeitung zielt auf die Beschreibung einer neuen rekonfigurierbaren Architektur und daher befasst sich auch der Schwerpunkt dieses Kapitels mit dem Stand der Technik im Bereich rekonfigurierbarer Systeme. Dennoch werden zur Vollständigkeit bekannte prozessorbasierte Lösungen besprochen, um den Stand der Technik in der Datenverarbeitung aufzuzeigen.

## 3.1. Einleitung

Durch die Fortschritte der Halbleitertechnik und der Mikroelektronik waren die Mikroprozessorhersteller in der Lage, in regelmäßigen Zyklen nach dem Mooreschen Gesetz immer schnellere Entwicklungen zu präsentieren. Die Anwender waren für die immer günstiger realisierbare Rechenleistung dankbar und verlangten für bestimmte Anwendungsgebiete nach Speziallösungen, die ihren Anforderungen gerecht werden sollten. So entstanden zunächst Single-Prozessor-Systeme für Heimanwender, Multi-Prozessor-Systeme für Workstations sowie Server und zu guter Letzt massiv-parallele Systeme, wie Großrechner für wissenschaftliche und militärische Anwendungen. Zusätzlich verlangen Kunden auch im mobilen Bereich nach mehr Leistung, die allerdings den stationären Systemen um ein Jahrzehnt bzgl. ihrer Leistungsfähigkeit hinterherhinken, jedoch sehr schnell aufholen.

Multi-Prozessor-Systeme basierten zunächst auf dem Einsatz von mehreren separaten Prozessoren mit jeweils einem Prozessorkern auf jedem Chip. Die Leistungssteigerungen wurden erzielt durch Architekturverbesserungen oder gar Neuentwicklungen und durch das Anheben der Taktfrequenzen. Allerdings haben die Entwickler zu Beginn des 21. Jahrhunderts festgestellt, dass Taktfrequenzen oberhalb von 3 GHz zu enormer Wärmeentwicklung geführt haben. Neben den Schwierigkeiten der Wärmeabfuhr senkte dieser Umstand durch die erhöhte Leistungsaufnahme auch deutlich die Effizienz der Prozesso-

ren. Dies führte zum Umdenken im Prozessordesign, so dass ab 2005 anstelle von einem schnell getakteten Prozessorkern zwei oder mehr Prozessorkerne mit einer langsameren Taktfrequenz auf einem Chip integriert wurden. Das Ziel war es nun, die gesamte Verlustleistung in einem beherrschbaren Bereich zu halten und dennoch einen Gewinn an paralleler Leistungsfähigkeit anzubieten. Jetzt, nach rund einem Jahrzehnt der Multiprozessorlösungen auf dem kommerziellen Markt, haben nur vereinzelt Prozessoren die 4 GHz Marke erreicht oder gar überschritten.

Der neue Ansatz führte zunächst zu kaum spürbarer Steigerung der Leistungsfähigkeit, da der Markt zum Zeitpunkt der Einführung, abgesehen von wenigen Prinzip bedingten Ausnahmen, kaum Softwareanwendungen geboten hatte, die multithreading-fähig waren. Erst nach fünf Jahren der Entwicklung waren Standard-Applikationen auf dem Markt zu finden, die im Stande waren, von mehreren Prozessorkernen zu profitieren. Die Software-Entwicklung hat nun gegenwärtig zum Stand der Technik der Prozessoren aufgeschlossen, erfordert jedoch immer noch zusätzlichen Aufwand, um die Software zu parallelisieren. Die Unterstützung seitens der Entwicklungstools ist nicht so weit fortgeschritten, dass die Programmierer mehrere Kerne ohne explizite Planung und Umsetzung ihrer Anwendungen nutzen können. Zur Vereinfachung existiert beispielsweise die OpenMP API [36], welche die Nutzung und die Verwaltung paralleler Threads in Form von Compiler-Direktiven vereinfacht, jedoch weiterhin expliziten Eingriff durch Programmierer erfordert.

Neben den Lösungen als universelle Prozessoren (GPPs) hielten seit Mitte der Neunziger Jahre 3D-Graphikbeschleuniger Einzug in die Standard-PCs. Zunächst wurden sie mit höchst-spezialisierten Datentypen und Funktionen zur Rasterisierung, z-Buffering zur Tiefenbestimmung sowie Filterung und Mapping von Texturen ausgestattet. Ende der neunziger Jahre übernahmen diese Beschleuniger bereits zusätzlich die Berechnungen der Graphikszenen mittels Matrixtransformationen sowie Verarbeitung der Lichtquellen (Transform and Lightning). Moderne Graphikprozessoren (GPUs) bieten inzwischen genügend Flexibilität, so dass mittels spezieller APIs wie CUDA [37] oder OpenCL [38] die verfügbare Rechenleistung für Anwendungen außerhalb der graphischen Domäne zur Verfügung steht. Während aktuelle Prozessoren Spitzenleistungswerte im Bereich von 100 MFLOPs bieten, erreichen aktuelle GPUs mehr als 1 GFLOP. Dieser Leistungsabstand macht die GPUs insbesondere für wissenschaftliche Berechnungen attraktiv. Die Programmierung ist jedoch deutlich aufwändiger verglichen mit GPPs, da der Code explizit für GPUs geschrieben und die Speicherhierarchie der Architektur beachtet werden muss, wenn der Programmierer von der enormen Leistungsfähigkeit profitieren möchte. Die GPUs werden in diesem Fall wie gewöhnliche Co-Prozessoren genutzt und unterstützen den Hauptprozessor bei seiner Arbeit. Diese zusätzli-

che Performance führt aber zu einem großen Nachteil, da moderne GPUs bei entsprechender Auslastung nicht selten mehr als 200 Watt Verlustleistung aufweisen.

Eine Alternative zu herkömmlichen Rechenmaschinen besteht in der Nutzung rekonfigurierbarer Architekturen. Die hardwarenahe Realisierung der Funktionalität einer Anwendung durch Einbeziehung möglichst vieler paralleler Funktionseinheiten bietet hier einen enormen Vorteil bzgl. Performanz, Verlustleistung und Fläche. Das Ziel rekonfigurierbarer Systeme besteht nicht darin, komplette Betriebssysteme zu unterstützen, sondern als Co-Prozessoren die rechenintensiven Aufgaben dem Host-System abzunehmen, ähnlich wie bei GPUs. Grundlagen dieser Architekturen wurden bereits in Kapitel 2 erläutert. Während feingranulare FPGAs bereits seit rund drei Jahrzehnten kommerziell vertrieben werden, sind grobgranulare Architekturen immer noch Gegenstand der Forschung und haben sich im kommerziellen Umfeld nicht durchgesetzt.

In diesem Kapitel wird eine Übersicht über die modernen Datenverarbeitungssysteme gegeben. Es soll der aktuelle Stand sowohl der Prozessoren als auch hochparalleler GPUs berücksichtigt werden, die derzeit sowohl in stationären wie auch mobilen Geräten Verwendung finden. Hier sind vor allem kommerzielle Ansätze führend und sind im folgenden Unterkapitel als klassische Architekturen zusammengefasst. Rekonfigurierbare Architekturen lassen sich in zwei Hauptklassen unterteilen: grobgranular und feingranular. Die feingranularen Architekturen sind im Wesentlichen durch FPGAs vertreten. Im Bereich der grobgranularen Architekturen sind einige kommerzielle Architekturen zu finden, die sich auf dem Markt zum Großteil nicht durchgesetzt haben. Die große Vielfalt der Architekturen kommt jedoch aus akademischem Umfeld. Eine Übersicht relevanter rekonfigurierbarer Architekturen wird in zwei Unterkapiteln beschrieben, die in feingranulare und grobgranulare Ansätze aufgeteilt ist. Zur Ergänzung wurde auch das Konzept der ASICs in diese Übersicht eingefügt, da die ASIC-Lösung anerkannter Weise die beste Performance bei gleichzeitig niedrigster Verlustleistung und Fläche liefert und daher zum Vergleich bei neuen Architekturen herangezogen werden sollte.

## 3.2. Klassische Rechenarchitekturen

Zur Kategorie der klassischen Rechenarchitekturen gehören Prozessoren, die in großer Zahl in modernen Systemen verbaut werden. Zu diesen Systemen gehören neben den üblichen Personal Computern (PCs) vor allem eingebettete Systeme und mobile Geräte. Bezogen auf die Anzahl implementierter Prozessoren stellen die PCs nur eine kleine Zahl dar. Vorwiegend finden die Prozessoren ihre Anwendung in eingebetteten Systemen, die in nahezu allen Bereichen der Maschinenbau- und Elektrotechnikindustrie genutzt werden. Im Ge-

gensatz zu PCs sind in diesen Bereichen oft sehr einfache Rechenwerke im Einsatz, die wenig Kosten verursachen sollen. Daneben nahm im letzten Jahrzehnt die Bedeutung der mobilen Geräte enorm zu. Neben den einfachen Mobiltelefonen haben inzwischen auch Smartphones bis hin zu mobilen Pads am Markt Fuß gefasst und bieten dem Anwender umfangreiche Dienste und Applikationen. Zu diesem Zweck besitzen diese Geräte Single- oder Multi-Prozessor-Lösungen, die speziell für mobile Anforderungen entwickelt wurden und eine Reihe von Stromsparfunktionen zum effektiveren Umgang mit den Akkus bieten. Die Prozessoren in diesen Bereichen gehören zur Klasse der SISD und/oder MIMD Prozessoren [40]. Letztere klassifizieren vor allem Digitale Signalprozessoren (DSPs), die beispielsweise in Mobiltelefonen zur Signalverarbeitung genutzt werden. Sie besitzen VLIW-Architekturen, die mehrere Instruktionen in einem Zyklus laden und auf mehrere Datensätze angewendet können. Daneben sind DSPs entweder für Festkomma- oder Fließkommadatentypen ausgelegt und lassen sich damit klassifizieren. Seit dem Einzug der Smartphones in die mobile Kommunikationslandschaft werden auch durch den Einsatz spezieller Hardware Beschleuniger [39] zusätzliche SIMD Funktion in diesem Umfeld geboten.

Neben den Standard- bzw. Universalprozessoren (GPP) für die oben beschriebenen Anwendungsfälle, gibt es eine Reihe von Speziallösungen, die beispielsweise in Vektorrechnern genutzt werden. Die Anwendung der Vektorprozessoren erfolgt in massivparallelen Systemen, die zur Berechnung von physikalischen Vorgängen wie Strömungssimulationen oder Wetterberechnungen genutzt werden. Die Anforderungen hier verlangen vor allem die parallele Anwendung von einzelnen Operationen auf viele Datensätze, daher wird diese Klasse von Prozessoren als SIMD bezeichnet. SIMD-Erweiterungen fanden auch ihren Weg in GPPs durch die Einführung von MMX, SSE oder AltiVec Erweiterungen und gehören inzwischen zu Standardfunktionen. Damit lässt sich die eigentlich sequentielle Natur der GPPs zu einer parallel arbeitenden Recheneinheit erweitern und damit die Effizienz steigern.

Wie bereits angedeutet, können moderne GPUs mittels spezieller APIs für wissenschaftliche, technische oder auch allgemein für Anwendung mit parallelem Charakter eingesetzt werden. In diesem Kontext arbeiten sie als Co-Prozessoren und unterstützen den Hauptprozessor bei seiner Arbeit. Da es sich bei GPUs grundsätzlich um massiv parallele Architekturen mit mehreren hundert einfachen Prozessoren handelt und diese bereits Einzug in moderne Anwendungen gehalten haben, werden sie ebenfalls im Abschnitt klassischer Architekturen beschrieben.

Da nicht alle Ausprägungen der Prozessoren in dieser Ausarbeitung diskutiert werden können, wird der Umfang exemplarisch auf zwei Typen beschränkt: je ein Beispiel für eine GPP und eine GPU.

### 3.2.1. Universalprozessoren

Universalprozessoren (GPPs) bieten die größte Flexibilität verglichen mit anderen Konzepten aus der Datenverarbeitung. Diese Eigenschaft ist vorwiegend der Tatsache zu verdanken, dass die Prozessorkerne wahlfreien Zugriff auf den Arbeitsspeicher (RAM) haben und somit in Kombination mit internen Registern, ALUs und Adressgeneratoren Funktionen wie Sprünge, Schleifen und Funktionsaufrufe nach Belieben realisieren können. Wie im letzten Kapitel beschrieben, lässt sich der Aufbau der Prozessoren nach mehreren Gesichtspunkten klassifizieren. Diese Aspekte sollen hier nicht wiederholt werden, sondern werden als bekannt vorausgesetzt.

Die wahlfreien Speicherzugriffe bieten Vorteile für die Prozessoren und ermöglichen dieser Technologie eine flexible Nutzung. Das Laden jedes einzelnen Befehls und zusätzlich das Laden und Speichern der notwendigen Daten haben allerdings auch Nachteile: Die Bandbreite der Speicherschnittstelle wirkt beschränkend auf die maximale Leistungsfähigkeit des Prozessors. Durch Techniken wie Caching mit einer oder mehreren Stufen lässt sich dieser Effekt reduzieren, aber nicht beseitigen. Mittels SIMD Instruktionen lässt sich eine zusätzliche Leistungssteigerung erreichen, da eine Instruktion mehrere Operationen auslöst und damit das Laden mehrerer Instruktionen vermeidet. Um eine Vorstellung vom Stand der Technik moderner Prozessoren zu vermitteln wird nunmehr anhand eines modernen Prozessors der Firma Intel die Technik beschrieben.

### 3.2.2. Intel Sandy-Bridge-Mikroarchitektur

Die derzeit leistungsfähigsten Prozessoren bezogen auf die Leistung einzelner Prozessorkerne stellt die Firma Intel [5] her, wobei das Prozessorgeschäft seit rund 40 Jahren auch gleichzeitig ihr Kerngeschäft darstellt. Bereits 1971 stellte Intel den ersten vollintegrierten 4004 Mikroprozessor mit 2300 Transistoren vor. In den darauffolgenden Jahren wurden mehrere Architekturen mit jeweils unterschiedlichem Erfolg und steil ansteigender Komplexität entwickelt. In den Jahren 2000 bis 2006 hat Intel auf die Netburst-Architektur (Pentium 4) gesetzt, die sich durch eine sehr lange Pipeline von 31 Stufen auszeichnete. Die Entwickler hofften, dass bei sehr hohen Frequenzen oberhalb von 3 GHz die Nachteile der langen Pipeline wieder aufgehoben sein würden. Zum Einführungszeitpunkt wurde der erste Netburst Prozessor mit 1,4 GHz betrieben und besaß 42 Millionen Transistoren hergestellt in 180nm Technologie, erreichte jedoch nur mittelmäßigen Instruktionsdurchsatz (IPC), wodurch er mit seiner Konkurrenz bei gleichen Taktfrequenzen nicht mithalten konnte. Die lange Pipeline war schlussendlich zu teuer. Bei auftretenden Programmsprüngen hat die zusätzliche Logik für Sprungvorhersagen (Branch

Prediction) den Nachteil der langen Pipeline nicht wettmachen können. Dieser Umstand senkte das IPC deutlich. Bei hohen Taktraten stieß Intel zusätzlich auf das Problem der hohen Wärmeentwicklung, was zusätzlich die Erhöhung der Taktfrequenzen begrenzte. Erst nach zwei Jahren der evolutionären Entwicklung der Halbleitertechnik erreichte Intel bei 130nm die 3GHz Grenze. Die resultierende Leistung der Architektur stand gleichwohl weiterhin in keinem Verhältnis zu resultierender Verlustleistung. Der Grund hierfür lag zuallererst auch in der Tatsache begründet, dass die interne Pipeline dieser Architektur mit doppelter Frequenz betrieben wurde.

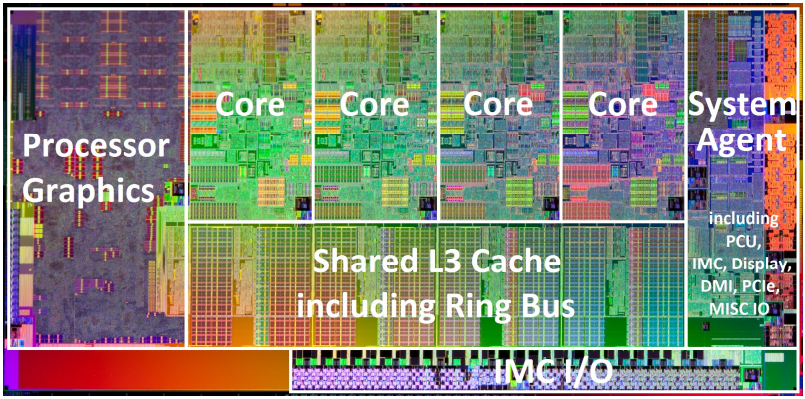



**Abbildung 26 Die-Photo des Sandy-Bridge-Prozessors mit vier Kernen, einem Graphikprozessor und dem System Agent**

Nach den Erfahrungen mit der Netburst-Architektur hat Intel das Konzept der langen Pipelines aufgegeben und kehrte zu bewährten Konzepten des Pentium Pro (P6-Mikroarchitektur) aus den Neunzigern zurück. Bereits 2003 wurde der Pentium M für den Mobilen Sektor eingeführt, der im Kern auf der alten P6-Architektur basierte. Sein Erfolg und die Probleme mit der Netburst-Architektur veranlassten Intel das Konzept weiter zu verfolgen und schließlich 2006 die Intel-Core-Mikroarchitektur einzuführen. Alle aktuellen Prozessoren von Intel basieren im Kern auf diesem Ansatz und werden von Generation zu Generation weiterentwickelt. Eine der aktuellen Mikroarchitekturversionen trägt den Namen „Sandy Bridge“ und wird nachgehend im Detail erläutert.

Die Sandy-Bridge-Mikroarchitektur ist zum Einsatz in einem Mehrkernprozessor konzipiert und kann maximal acht Kerne auf einem Chip unterbringen. Jeder Kern besitzt zwei Cache-Stufen: Einen geteilten Level1-Cache für Daten und Instruktionen mit jeweils 32KB und einem Level2-Cache mit 256 KB. Alle Kerne auf einem Mehrkernprozessor teilen sich den Level3-Cache, der modular aufgebaut ist und mit der Anzahl der Kerne an Größe zunimmt, da

jeder Kern einen ihm zugeordneten L3-Cache (Last Level Cache, LLC) fester Größe zusätzlich beisteuert. Die einzelnen Teile des L3-Cache sind über einen leistungsfähigen Ring-Bus verbunden, der unter anderem zum Datentransport dient und Cache-Snooping-Funktionen zum Synchronisieren der L3-Cache-Teile (LLCs) bietet. Die Bandbreite des 256bit breiten Datenbusses liegt im Bereich von 100GB/s und hängt direkt vom verwendeten Takt des Prozessors ab. Zusätzlich zu Prozessorkernen sind der System Agent und der optionale Graphikprozessor an den Ringbus gekoppelt, siehe Abbildung 26. Dadurch bekommen diese Komponenten Zugriff auf die LLCs. Der System Agent bietet folgende Funktionen: integrierten Speicher-Controller (IMC), PCI Express Interface (PCIe), Direct Media Interface (DMI), Cache-Controller und Power Control Unit (PCU).

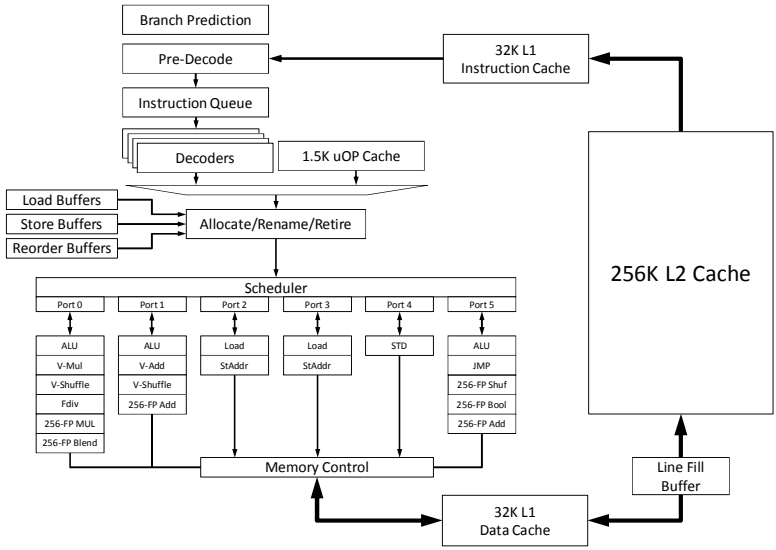



**Abbildung 27 Datenpfad der Intel Sandy-Bridge-Mikroarchitektur**

Abbildung 27 zeigt den internen Aufbau eines Sandy-Bridge-Prozessorkerns. Jeder Kern beinhaltet zwei Cache-Stufen und den eigentlichen Datenpfad einschließlich Steuerlogik, die in mehrere Module aufgeteilt ist. Der komplette Datenpfad lässt sich in zwei Bereiche einteilen: der erste Bereich mit In-Order-Execution und den zweiten Bereich mit Out-Of-Order Superscalar Execution. Die Befehle im ersten Teil werden der Reihe nach, wie sie im Speicher zu finden sind, abgearbeitet, während im zweiten Teil die Befehlsreihenfolge geändert wird, um die Effizienz der Ausführung zu steigern. Zu-

sätzlich ist die Ausführung sechs-fach superskalar ausgelegt, so dass bis zu sechs Mikrooperationen in einem Zyklus ausgeführt werden können.

Die angeforderten Befehle (Instruction Fetch) werden zunächst aus dem L1-Cache geladen. Falls ein Cache-Miss vorliegt, so werden die übrigen Caches in der aufsteigenden Reihenfolge abgefragt und zuletzt auf den Hauptspeicher zugegriffen. Ein Cache-Miss hat eine höhere Latenz zur Folge, so dass der Prozessor länger auf seine Instruktionen oder Daten warten muss. Die Pre-Decode-Stufe untersucht zunächst den eingehenden Befehl auf seine Größe hin und bestimmt alle längenbestimmenden Präfixe der Instruktion. Da es sich bei Sandy-Bridge-Prozessoren um CISC-Prozessoren handelt, sind die Längen von Befehl zu Befehl unterschiedlich und müssen explizit behandelt werden. Das Ergebnis des Pre-Decode wird in der Instruction Queue abgelegt und den vierfach ausgeführten Decodern zugeführt. Die Decoder haben die Aufgabe die CISC-Instruktionen in Mikrooperationen (µOps) zu übersetzen. Einer der vier Decoder ist im Stande alle eingehenden Instruktionen zu dekodieren, während die übrigen drei nur einfache CISC-Befehle in eine Mikrooperation dekodieren können. Zusätzlich zum Dekodieren, werden in dieser Stufe sowohl MicroFusion als auch MacroFusion Operationen angewendet. MicroFusion untersucht die resultierenden Mikrooperationen und fasst nach bestimmten Regeln mehrere Mikrooperationen zu einer zusammen. MacroFusion fasst mehrere CISC-Befehle zu einer Mikrooperation zusammen.

Die ausgehenden Mikrooperationen werden im Decoder-Cache (1,5K µOp Cache) gepuffert und durch die µOp-Queue an die Allocate/Rename/Retire-Einheit geleitet. Der Decoder-Cache bietet einen zusätzlichen Puffer von bis zu 1536 Mikrooperationen und erlaubt einen schnellen Zugriff auf die Befehle, was speziell im Falle von Schleifen zu einer enormen Verlustleistungseinsparung des Frontends und Beschleunigung der Ausführung führt. An dieser Stelle ist auch die Branch-Prediction-Einheit (Sprungvorhersage) von Interesse, die entscheidet an welcher Stelle im Programm-Code die Ausführung weitergehen soll. Dies erfolgt nicht nur anhand des aktuellen Programmzeigers (IP), sondern auch zusätzlich anhand des Pfades im Programm, der zur Verzweigung führt und greift somit insbesondere bei Schleifen oder häufig durchlaufenen Verzweigungen.

Die µOp-Queue bietet eine zusätzliche Optimierung für sehr kleine Schleifen mit maximal 28 Mikrooperationen und sorgt dafür, dass Schleifen vollständig aus dieser Queue ausgeführt werden. Verantwortlich für die Erkennung der Schleifen ist die Loop-Stream-Detector-Einheit (LSD), die ebenfalls Bestandteil der µOp-Queue ist.

Die folgende Out-Of-Order-Engine mit den Operationen Allocate/Rename/Retire hat die Aufgabe die Mikrooperationen zur Out-Of-Order-Ausführung vorzubereiten. Zu diesem Zweck werden Operanden-Abhängigkeiten zwischen

den Mikrooperationen aufgelöst und auf interne Register überführt (Allocate). Der Scheduler sorgt dafür, dass Befehle erst zur Ausführung kommen, wenn alle Quelloperanden zur Verfügung stehen. Bis zu diesem Zeitpunkt werden betroffene Mikrooperationen gepuffert. Weiterhin sorgt die Retire-Funktion für ordnungsgemäßes Verwerfen ausgeführter Mikrooperationen und behandelt Fehler und Ausnahmen.

Die Out-Of-Order-Ausführung ist der anschließende Teil im Datenpfad und hat die Aufgabe die eingehenden Befehle tatsächlich auszuführen. Zu diesem Zweck stehen drei unterschiedliche Ausführungseinheiten zur Verfügung, die mit unterschiedlichen Operationen und Datentypen ausgestattet sind. Die erste kann Integer-Operationen ausführen, die zweite SIMD Integer und Floating Point Operationen, während die dritte Operationen des mathematischen Coprozessors x87 bietet. Der vorgeschaltete Scheduler verteilt bis zu sechs Mikrooperationen pro Takt und Port gemäß ihrer Funktionen und Abhängigkeiten und sorgt damit für die effiziente Ausführung des Programm-Codes. Die Ausführungseinheiten haben eine Verbindung zum L2-Data-Cache und sind im Stande Daten zu laden oder zu speichern, siehe Ports 2, 3 und 4. Somit können Lade/Speicher-Operationen parallel zur Rechenoperationen ausgeführt werden, was enorme Vorteile für die Leistungsfähigkeit des Prozessors bringt.

Die Zugriffe des Datenpfads auf die Caches erfolgt über die Übersetzungspuffer (Translation Lookaside Buffer, TLB,). Diese Einheiten übersetzen die eingehenden linearen Adressen des aktuellen Tasks in die tatsächlichen physikalischen Adressen des Hauptspeichers und erlauben es den Programmen in exklusiven Adressbereichen mit jeweils eigenen Adressräumen im Hauptspeicher zu residieren. Sowohl der L1 Instruction Cache aus auch der L1 Data Cache besitzen zu diesem Zweck eigene TLBs. Hinter dem L2-Cache sind die Prozessorkerne mittels des Ringbusses an die gemeinsamen LLCs (L3 Caches) angeschlossen.

Die aktuellen Sandy-Bridge-Prozessoren werden derzeit mit einem Takt von bis zu 3,6 GHz bei einem Vierkernprozessor angeboten. Maximal sind jedoch Achtkernprozessoren verfügbar, die aber einen geringeren Takt von 2,9GHz bieten. Die Intel-Turbo-Technologie bietet allerdings zusätzlich die Möglichkeit bei Auslastung nur weniger Kerne die Taktrate je nach Modell teils bis zu 33% zu steigern, solange die gesamte thermische und stromtechnische Belastung (TDP) das spezifizierte Maximum des Prozessors nicht überschreitet. Im April 2012 hat Intel eine neue Prozessor-Reihe mit dem Namen Ivy Bridge eingeführt. Hierbei handelt es sich im Kern weiterhin im die Sandy-Bridge-Architektur, die geringe Modifikationen erfahren hat und in Intels neuer 22nm Technologie unter der Verwendung der neuen 3D-Tri-Gate-Transistoren [73] gefertigt wird. Bis dato ist die Sandy-Bridge-Architektur

insbesondere im Serverbereich mit bis zu 8 Kernen weiterhin im Einsatz und von der Ivy-Bridge nicht abgelöst.

### 3.2.3. GPGPU - nVidias Fermi Architektur

Nach einer Reihe technischer Schwierigkeiten in der Fertigung brachte nVidia im April 2010 die bis zu diesem Zeitpunkt komplexeste und leistungsstärkste GPU-Reihe auf den Markt, basierend auf der Fermi-Architektur [74], die nach dem italienischen Kernphysiker Enrico Fermi benannt war. Primärer Einsatz dieser GPU ist die Graphikverarbeitung in PC-basierten Systemen. Die 512 parallelen Prozessoren in der größten Ausführung der Fermi-Architektur machen sie gleichzeitig wegen hoher Verlustleistung unbrauchbar für den mobilen Einsatz. So entwickelte nVidia zusätzlich GPU-Varianten mit deutlich weniger Prozessoren bei gleichzeitig geringerer Leistungsfähigkeit, jedoch auch mit annehmbarer Verlustleistung, die heute in mobilen Geräten Verwendung finden. Die bereits erwähnte CUDA API erlaubt es den Programmierern mit bekannten Programmiersprachen wie C/C++ Anwendungen für die Fermi-Architektur zu entwickeln. Damit ist es jedem Nutzer möglich die enorme Leistungsfähigkeit der GPU auch zu Hause zu nutzen und somit seinen PC in einen Hochleistungscomputer zu verwandeln. Inzwischen bieten viele namhafte Software-Hersteller Lösungen ihrer Produkte an, die von installierten GPUs im PC profitieren. Dazu gehören insbesondere Hersteller für Multimediaanwendungen für den Heimkinobereich wie Corel [75] mit WinDVD oder CyberLink [76] mit PowerDVD. Beide Applikationen können DVDs und Blurays abspielen unter Zuhilfenahme CUDA-kompatibler GPUs und damit die CPU im System entlasten.

Um die Funktionalität der Fermi-Architektur zu verstehen, muss zunächst betrachtet werden, wie CUDA hardware- und softwareseitig Applikationen auf der GPU zur Ausführung bringt. Die kleinste Einheit in der Ausführung stellt dabei ein Thread dar, der neben Registern einen lokalen privaten Speicher besitzt. Mehrere parallele Threads werden zu einem Thread-Block zusammengefasst, der wiederum einen unter den Threads geteilten Speicher besitzt. Somit können Threads in diesem Block Daten austauschen. Mehrere solcher Thread-Blöcke werden in Grids zusammengefasst. Jedes Grid besitzt seinerseits einen globalen Applikationsspeicher, der durch seinen globalen Charakter Datenaustausch zwischen mehreren Grids ermöglicht. CUDA ist mit dieser Struktur im Stande ein oder mehrere Grids auf einmal auf der GPU auszuführen, wobei jeder Streaming-Multiprozessor (SM, siehe Abbildung 28) ein oder mehrere Thread-Blöcke seinerseits ausführen kann. Jeder SM bearbeitet dabei gruppenweise 32 Threads, die Warp genannt werden. Um das CUDA-Programmiermodell effizient zu nutzen, muss der Programmierer die vorlie-

gende GPU-Architektur genau kennen, sonst läuft er Gefahr durch unnötig entstandene Latenzen wertvolle Rechenleistung zu verschenken.

Für die massive Parallelität, die notwendig ist, um das CUDA Programmiermodell zu verarbeiten, bietet die Fermi-GPU insgesamt 512 Rechenkerne (CUDA Cores), die jeweils in der Lage sind sowohl Integer als auch 32bit Fließkomma-Operationen nach dem IEEE 754-2008 Standard [61] einschließlich Fused Multiply-ADD-Operationen (FMA) anzubieten. Für Fließkomma-Operationen doppelter Genauigkeit lassen sich zwei CUDA-Cores zusammenfassen, jedoch mit halbem Durchsatz. Eine FMA-Operation ähnelt den MAC-Operationen, rundet das Ergebnis jedoch erst nach der Addition ab. Jeweils 32 CUDA-Cores werden zu einem Streaming-Prozessor (SM) zusammengefasst. Neben den CUDA-Cores besitzt jede SM weitere Funktionseinheiten, wie Special Function Units (SFUs) für transzendentale Funktionen und Interpolationen, Load/Store Einheiten, vier Textur-Einheiten, eine PolyMorph-Einheit und gemeinsam nutzbaren L1-Cache mit 64KB, der nach bestimmten Regeln als Cache oder Shared Memory konfigurierbar und nutzbar ist. Daneben besitzen SMs zwei Warp-Scheduler und zwei Dispatcher-Einheiten, die im Stande sind zwei Warps gleichzeitig zur Ausführung zu bringen. Die Beschreibung der Instruktionen in einem Warp geschieht für alle Threads auf einmal und hat zur Folge, dass identische Instruktionen in den Threads parallel, während unterschiedliche Instruktionen nacheinander ausgeführt werden. In dieser Eigenschaft ähneln die GPUs den Vektorprozessoren, was bei Graphikanwendungen von Vorteil ist, da viele identische Operationen auf die Graphikpunkte angewendet werden und massive-parallele Verarbeitung gewünscht ist.


**Abbildung 28 Streaming Multiprozessor (SM) mit integrierten Komponenten**

Insgesamt 16 SMs sind in einer Fermi-GPU integriert und um einen L2-Cache mit 768KB angeordnet, siehe Abbildung 29. Daneben sind weitere vier Raster Engines integriert, um die eigentliche Ausgabe auf dem Monitor zu erzeugen. Die gesamte Architektur kommt damit auf insgesamt 512 Rechen-

einheiten und 64 SFUs, die allgemein von Anwendungen genutzt werden können. Allen SMs steht der gemeinsame L2-Cache Verfügung. Der gesamten Architektur stehen sechs 64bit GDDR5 Speicher-Controller zur Verfügung, die bei einem Takt von 2,5GHz eine Bandbreite von 240 GB/s erreichen. Für professionelle Anwendungen ermöglichen die Speicher-Controller optional die Unterstützung von ECC-Speichern. Daneben sind weiterhin ein Host-Interface und die Giga Thread Engine integriert. Letztere ist verantwortlich für die schnellen Kontext-Wechsel und das globale Scheduling einschließlich Verteilung von Anwendungen auf die SMs.


**Abbildung 29 Fermi-Architektur mit 16 Stream-Multiprozessoren (SMs), L2 Cache, Memory Controllern, Giga Thread Engine und Host Interface**

Die GigaThread Engine verteilt die Warps auf die SMs, wobei SM interne Scheduler- und Dispatcher-Einheiten die Operationen auf die Ausführungseinheiten verteilen. Dabei ist ein Dispatcher in der Lage 16 Threads pro Zyklus zu starten, was einem halben Warp entspricht. Die andere Hälfte kann entweder vom selben SM oder einem anderen ausgeführt werden. Im Gegensatz zu früheren GPU-Modellen kann die Fermi Architektur erstmals mehrere Kernels parallel zur Ausführung bringen, falls genügend Ressourcen vorhanden sind. Bei früheren GPU Modellen musste ein Kernel beendet werden, bevor der nächste starten konnte. Auf diese Weise wird die Auslastung der Architektur

zusätzlich erhöht und Anwendungen können effizienter ausgeführt werden. Der L2-Cache hat eine sehr wichtige Funktion innerhalb der Architektur. Da die Speicherzugriffe sehr teuer sind und sich im Bereich von mehreren hundert Zyklen bewegen, ist die entkoppelnde Funktion des L2-Caches umso wichtiger und greift bei der Verarbeitung kleiner Datenmengen. Die teuren Speicherzugriffe sind bei Graphikanwendungen nicht so wichtig, da die hohe Anzahl an Threads es dennoch erlaubt die SMs zu beschäftigen, indem sehr schnelle Kontextwechsel vollzogen werden.

Der komplette Fermi-Chip besitzt insgesamt 3 Millionen Transistoren und stellte im Jahr 2010 den absoluten Rekord dar. Produziert wurde die GPU in 40nm TSMC CMOS Standardzellentechnologie und benötigte eine Fläche von 529 mm². Die Leistungsfähigkeit dieser GPU wird vom Hersteller mit rund 1,3GFLOPS bei einfacher Genauigkeit angegeben und sinkt um die Hälfte bei Nutzung der doppelten Genauigkeit. Damit ist die Fermi-Architektur den modernen Prozessoren um den Faktor 10 überlegen und stellt eine sinnvolle Alternative für wissenschaftliche Berechnungen dar.

Im März 2012 stellte nVidia den Nachfolger der Fermi-Architektur vor mit dem Namen Kepler [77]. Die neue Architektur ähnelt sehr stark der Fermi-Architektur, wurde jedoch in vielen Punkten verbessert. So beinhaltet die gesamte Architektur 15 Streaming-Multiprozessoren (SMX), wobei jeder SMX 192 CUDA Cores sowie 32 SFUs enthält. Der gesamte Chip kommt auf 3,5 Milliarden Transistoren und wird in 28nm TSMC CMOS Standardzellen-Technologie gefertigt.

## 3.3. Applikationsspezifische Integrierte Schaltkreise

Integrierte Schaltkreise, die vollständige Applikationen oder Teilalgorithmen ausführen können, werden als Applikationsspezifische Integrierte Schaltkreise (Application Specific Integrated Circuits, ASICs) bezeichnet. In der heutigen Welt sind ASICs aus dem Alltag praktisch nicht wegzudenken, da sie in den meisten Geräten mit elektronischen Komponenten verwendet werden. Diese Schaltstrukturen werden speziell für ausgewählte Applikationen entwickelt und sind nicht im Stande andere Anwendungen auszuführen. In dieser Hinsicht sind sie den Prozessoren unterlegen, da sie keinerlei Flexibilität bieten. Auf der anderen Seite sind ASICs sehr effizient in der Ausführung der unterstützten Anwendungen. Sie bieten die bestmögliche Effizienz hinsichtlich der Performanz, Fläche und Verlustleistung.

Doch wie lässt sich die überlegene Effizienz von ASICs erklären? Während komplexere Ansätze wie Prozessoren oder gar rekonfigurierbare Architekturen nur die Grundlagen ihrer Funktionalität in einer Schaltung definieren und ihre Anwendungen weiterhin in Form von Informationen und damit Speicherinhal-

ten (sprich als Software oder Configware) in diese Architekturen eingebracht werden, um die internen Schaltvorgänge zu steuern, wird dieser Umstand bei ASICs strukturell ausgeführt. Die vollständige Funktionalität ist in der Schaltung kodiert und benötigt somit keinen zusätzlichen Aufwand, so dass die gesamte Schaltung nur für den Zweck der Applikation optimiert werden kann. Es werden also keine Komponenten in der Schaltung dazu verwendet Verwaltungsaufgaben zu übernehmen. Weiterhin können Applikationen ähnlich einer Pipeline als eine Kette von Funktionseinheiten realisiert werden und die teuren Speicherzugriffe bzgl. Bandbreite und Verlustleistung zwischen den Funktionseinheiten können auf ein Minimum reduziert werden.

Die verwendete Basistechnologie für einen ASIC ist die gleiche wie bei Prozessoren und anderen modernen Rechenarchitekturen. Sie alle setzen auf Halbleitertechnologien von unterschiedlichen Herstellern, die heute mit Strukturbreiten von bis zu 22 nm verfügbar sind. Die Unterschiede dieser Technologien sind aus Sicht des Anwenders vernachlässigbar und sind vorwiegend technologischer Art. Intel beispielsweise nutzt inzwischen 3D-Tri-Gate-Transistoren, die von der üblichen planaren Struktur der früheren Ansätze abweichen. Die Basis bei allen Herstellern sind Wafer, kreisförmige Siliziumscheiben, die als Träger für die zu produzierenden mikroelektronischen Schaltungen dienen. Die benötigte Fläche pro Chip legt maßgeblich die Höhe der Kosten fest und lässt sich auf die Anzahl der Chips auf diesem Wafer umrechnen. Weiterhin hängt die Wahrscheinlichkeit für einen fehlerfreien Chip von dessen Größe ab, so dass sie maßgeblich die effektive Ausbeute bestimmt und damit die Stückkosten. Bei Nutzung größerer Wafer (heute üblicherweise 300mm im Durchmesser) fällt der Verschnitt am Rand des Wafers im Verhältnis zur Nutzfläche geringer aus als bei kleineren Wafern, was sich kostensenkend auf die Chips auswirkt. Alternativ zum Silizium kann auch Germanium als Halbleitermaterial genutzt werden, was zu Anfangszeit der Halbleitertechnologien auch üblich war. Heute wird Germanium nur noch bei Spezialanwendungen wie Sensoren oder Hochfrequenzschaltungen genutzt.

Mittels Halbleitertechnik werden auf dem Wafer durch Auftrage-, Belichtungs- und Ätztechniken Transistorstrukturen gebildet. Üblicherweise geschieht dies durch Masken für eine sehr große Anzahl an Transistoren in einem Durchgang. Mehrere Transistoren setzen sich zu logischen Gattern oder Registern zusammen, wovon mehrere bereits eine mikroelektronische Digitalschaltung bilden können. Das Interconnect zwischen den Transistoren wird mittels Metalllagen, heute Kupfer, realisiert, von denen bei aktuellen Technologien meist 10 nutzbar sind. Neben anderen Verfahren kann der Entwurf von ASICs wahlweise als Full-Custom-Design oder Standard-Cell-Design erfolgen. Beim Letzteren stehen Werkzeuge zur Verfügung, die in Kombination mit Hardware-Beschreibungssprachen (Hardware Description Language, HDL), wie

VHDL [54] oder Verilog [55] einen automatisierten Design Flow bieten, der bei einem ausgereiften Flow nur wenige manuelle Eingriffe benötigt. Die notwendigen Schritte für ein komplettes ASIC-Design schließen die Verifikation der funktionalen Beschreibung, Synthese, Place and Route, Timing Analyse, Layout-versus-Schematic Analyse, Design Rules Checks und funktionale Post-Layout-Verifikation. Im Falle des Full-Custom-Designs ist der Entwickler deutlich stärker gefordert, da die finale Schaltung manuell auf Transistorebene gelayoutet wird. Die Tool-Unterstützung ist in diesem Fall deutlich eingeschränkt und erfordert daher große Erfahrung seitens des Entwicklers. Die wichtigste Unterstützung erhält der Entwickler bei seiner Arbeit in diesem Fall durch Design-Rule-Checker (DRC), damit die technologische Realisierung mit geringerem Risiko gelingt.

Trotz der wichtigen Vorteile des ASICs gegenüber Prozessoren hinsichtlich der Performanz, Fläche und Verlustleistung existieren auch wichtige Nachteile, die in diesem Zusammenhang von großer Bedeutung sind. Ein wichtiger Nachteil liegt in der geringen Flexibilität des resultierenden Schaltkreises. ASICs sind nicht in der Lage Anwendungen auszuführen, die zum Entwicklungszeitpunkt nicht vorgesehen waren. Ebenso ist die Komplexität zur Erstellung dieser Anwendungen im Falle des ASICs wesentlich höher und bedarf eines erheblichen monetären Aufwands. Der Kostenfaktor stellt eine große Hürde dar. Die Fertigung von Halbleiterstrukturen benötigt den Einsatz von Belichtungsmasken, die sehr teuer sind. Bei der Massenfertigung mikroelektronischer Schaltungen rechnet sich dieser Maskenpreis, da er sich auf die große Zahl der Chips umrechnen lässt. Bei geringen Stückzahlen oder Design-Änderungen fällt der Maskenpreisanteil deutlich ins Gewicht und bestimmt den Verkaufspreis des Produkts. Des Weiteren stellt die benötigte Zeit zur Produktion ein großes Problem dar, da im Falle eines Fehlers oder einer Design-Änderung bis zu einem halben Jahr vergehen kann, bis ein neuer Chip verfügbar ist.

## 3.4. Feingranulare rekonfigurierbare Architekturen

Zur Kategorie der feingranularen rekonfigurierbaren Architekturen gehören insbesondere FPGA-Bausteine. Die bekanntesten und größten Vertreter werden von den Firmen Xilinx [10] und Altera [11] produziert, die gemeinsam den Markt für FPGAs dominieren. Ursprünglich zwecks ASIC-Prototyping und für Interconnect-Aufgaben entwickelt, haben sich FPGA-Bausteine in den letzten 30 Jahren als Allzwecktalente entpuppt. Sie werden in sehr vielen Bereichen der Elektrotechnikbranche eingesetzt, insbesondere bei Anwendungen, wo keine großen Stückzahlen anfallen. Bei hochwertigen Produkten wird die Entscheidung für den Einsatz von FPGAs auch aus anderen Gründen getroffen, die beispielsweise in dem breiten Sortiment an Kommunikationsschnittstellen in modernen FPGAs zu finden sind. Dadurch entfällt bei Firmen die Notwendigkeit der Entwicklung eigener Schnittstellen, insbesondere bei Firmen, die über keine großen Entwicklungsressourcen verfügen. Durch dieses Vorgehen verringert sich das Investitionsrisiko und nimmt den zusätzlichen Druck aus den Entwicklungsabteilungen raus.

Moderne FPGAs beherbergen bis zu 1,9 Millionen Logikzellen, wie dies im Falle des neuen Virtex 7 [78] der Fall ist. Hierbei handelt es sich um einen Baustein für die höchsten Leistungsansprüche, der allerdings auch im Bereich der Verlustleistung an der Spitze liegt. Nach Erfahrungen der letzten Jahre dürfte die Spitzenverlustleistung jenseits der 100 W liegen. Dafür bekommt der Entwickler allerdings auch eine Reihe von Features geboten. So beherbergt der Baustein neben den herkömmlichen Logikzellen dedizierte Speicherblöcke unterschiedlicher Größe, DSP-Blöcke mit Akkumulatoren und Multiplizierern sowie eine Reihe von Standardschnittstellen für PCIe oder DDR Implementierungen. Daneben sind 28G-Tranceiver im Baustein integriert, die relativ einfach für proprietäre Hochgeschwindigkeitsübertragungen genutzt werden können. Der Virtex-7 wird in verschiedenen Größen und Konfigurationen angeboten, die sich in der Anzahl der Logikzellen, dedizierter Blöcke, sowie Schnittstellen unterscheiden. Die exakten Spezifikationen der FPGAs werden durch Gespräche mit Kunden identifiziert und sorgen dafür, dass jeder Kunde eine möglichst effiziente Lösung für sein Problem findet.

Das Innovative an dem Virtext-7 ist die neue Technik genannt Stacked Silicon Interconnect (SSI). Hierbei wird der Virtex 7 nicht als ein komplettes Die in unterschiedlichen Varianten entwickelt und produziert, sondern aus einzelnen Teil-Dies zum gesamten Chip im Package zusammengesetzt. Als Träger dient hierzu ein Silicon Interposer, vorzugsweise in einer günstigeren Technologie. Die SSI Technik sorgt für eine transparente und schnelle Kommunikation zwischen den Teilabschnitten, die dem Anwender in seinem Design nicht auffallen. Um das zu erreichen, werden mehr als 10.000 Verbindungen zwischen den Teilabschnitten realisiert. Der Vorteil dieser Technik liegt in der

verbesserten Ausbeute für den gesamten Chip. Einzelne Segmente des Virtex 7 werden separat gefertigt und besitzen eine kleinere Fläche im Vergleich zum gesamten Chip. Die Ausbeute in der Chip-Fertigung ist aber exponentiell abhängig von der benötigten Fläche pro Die und sinkt mit der steigenden Fläche, daher ergeben sich auch niedrigere Fertigungskosten für den kompletten Virtex 7. Weiterhin hat der Hersteller die Möglichkeit durch die Verwendung der modularen Standardsegmente unterschiedlich große Modelle des Virtex-7 anzubieten, ohne diese explizit zu entwickeln.

Die Hersteller der FPGAs haben sehr früh erkannt, dass die Implementierung rein feingranularer Strukturen die FPGAs sehr schnell an ihre Grenzen bringt. FPGA-Designs, die sich mit Bit-Logik befassen und arithmetische Operationen außen vor lassen, sind für derartige Lösungen ausreichend. Moderne Anwendungen in fast allen Bereichen kommen allerdings nicht ohne arithmetische Operationen aus, so dass FPGA-Hersteller schnell dazu übergingen dedizierte vektorbasierte Einheiten in FPGAs anzubieten. Diese Tatsache macht es schwierig FPGAs als reine feingranulare Ansätze zu sehen, sondern legt die Klassifizierung als hybride Lösung nahe. Die konkrete Mischung und das Verhältnis der dedizierten Einheiten zueinander und zu Logikzellen sind jedoch applikationsspezifisch. Daher bieten Hersteller oftmals zusätzlich zu unterschiedlichen Größen der FPGAs auch unterschiedliche Ausprägungen bzgl. der erwähnten Verhältnisse.

Die Programmierung der FPGAs ist im Frontend dem Design Flow der ASICs sehr ähnlich. In beiden Fällen stellen aktuell HDLs den Einstieg in den Design-Flow dar. Die genutzten Technologie-Primitive unterscheiden sich allerdings deutlich, da im ASIC insbesondere Gatter genutzt werden, bei FPGAs hingegen stellen Look-Up-Tables (LUTs) in den Logikzellen und die vorhandenen dedizierten Einheiten die eigentlichen Primitiven dar. In beiden Fällen geschieht die Abbildung der HDL-Beschreibung auf die Primitiven mittels einer Synthese, die neben den Design-Constraints auch Technologie-Bibliotheken nutzen. Das Ergebnis ist in beiden Fällen eine Netzliste mit den genutzten Primitiven als Knoten. Das Backend, d.h. Place and Route des Designs auf den Zielbaustein, ist beim FPGA stark abhängig vom verwendeten Modell, das die notwenigen Ressourcen bietet. Durch die Einstellung des Backend-Tools auf den entsprechenden Typ, übernehmen diese weitgehend automatisiert die notwendigen Schritte zur fertigen Konfiguration.

## 3.5. Grobgranulare rekonfigurierbare Architekturen

Die Klasse der grobgranularen rekonfigurierbaren Architekturen zeichnet sich vor allem dadurch aus, dass arithmetische Operationen als Primitive verwendet werden. Dies geschieht zunächst einmal mittels ALUs, wie sie auch in klassischen Architekturen verwendet werden. Je nach Ausprägung der Architekturen, besitzen die verwendeten ALUs jedoch unterschiedliche Datentypen, d.h. unterschiedliche Bitbreiten, Datenformate und damit Genauigkeiten. Dieser Sachverhalt ist analog zu klassischen Architekturen zu sehen. Allein die große Zahl der integrierten ALUs in rekonfigurierbaren Architekturen, hebt das Potenzial zur Leistungsfähigkeit deutlich hervor. Dass dieses Potenzial nicht so einfach genutzt werden kann, zeigt die jahrzehntelange Arbeit der Unternehmen und zahlreicher Forscher in diesem Segment. Bandbreitenprobleme bei direktem Speicherzugriff (DMA), teure Interconnect-Strukturen, mangelnde Flexibilität, eingeschränkte Programmierbarkeit und nicht zuletzt Akzeptanz der neuen Technologien durch den Kunden sind nur einige der aufgetretenen Probleme in diesem Umfeld.

Herkömmliche Prozessoren haben es an dieser Stelle einfacher. Sie beherbergen meist nur wenige ALUs. Der Prozessor hat den direkten Speicherzugriff nach Art der von-Neumann oder Harvard-Architekturen. Die Abbildbarkeit von Algorithmen gestaltet sich in gewohnter Weise, da sie dem menschlichen Denken durch die Zerlegung der Probleme in eine zeitlich festgelegte Abfolge von Einzelschritten sehr nahe kommt. Das halbe Jahrhundert des Vorsprungs im Compilerbau ist in der Qualität der Entwurfswerkzeuge deutlich erkennbar. Allerdings haben die Prozessoren im Laufe der Zeit an Funktionalität dazugewonnen. Die Vielfalt an Speicherzugrifftechniken, Out-Of-Order-Execution, Speculative Execution, komplizierte Mikro-Code-Dekoder und/oder mehrstufige Caches fordern ihren Tribut. Die zusätzlichen Optimierungen ließen die modernen Prozessoren in der Fläche stark anwachsen und haben gleichzeitig die Verlustleistung in die Höhe getrieben. Die Hersteller versuchen mit Stromspartechniken diesen Nachteil in den Griff zu bekommen, jedoch wirken Erhöhungen der Taktrate zusätzlich verlustleistungssteigernd. Grobgranulare Architekturen versuchen diese Probleme durch Parallelität in den Griff zu bekommen. Statt in die Zeit werden Algorithmen in die Fläche abgebildet bei gleichzeitig niedrigen Taktraten.

Eine Möglichkeit den Entwurfsraum einer rekonfigurierbaren Architektur zu beschränken, ist die Reduktion unterstützter Zielanwendungen. Durch diesen Schritt lässt sich die Komplexität der resultierenden Hardware deutlich reduzieren, was unter anderem durch die Tatsache zu begründen ist, dass die notwendigen ALUs einen geringeren Umfang an Operationen benötigen und die arithmetischen Vektoren auf das notwendige Minimum schrumpfen. Weiterhin ist die Vielfalt der notwendigen Schaltmöglichkeiten zwischen den

implementierten Funktionseinheiten dadurch reduziert, so dass die Interconnect-Netzwerke ebenfalls eine Reduktion hinsichtlich der Anzahl der Ein- und Ausgänge erfahren.

Beliebte Applikationsdomäne für rekonfigurierbare Architekturen ist die digitale Signalverarbeitung. In diesem Umfeld dominieren seit Jahren die digitalen Signalprozessoren (DSPs), die einen optimierten Instruktionssatz und spezialisierte Datentypen mit sich bringen. Neben Fließkommaformaten bieten sie auch Festkommaformate, die in Hardware günstiger implementierbar sind. Durch die Spezialisierung steigt hinsichtlich der Leistungsfähigkeit und Verlustleistung die Effizienz der DSPs bei der Signalverarbeitung, wodurch sie insbesondere attraktiv für mobile Geräte sind. Alternativ zu DSPs werden auch ASICs zur Verarbeitung in diesem Umfeld genutzt. Die Leistungsfähigkeit der ASICs stellt in diesem Zusammenhang die Leistungsspitze dar und wird häufig als Referenz zum Leistungsvergleich herangezogen. ASIC Implementierungen stellen die direkteste Umsetzung einer Applikation auf der gegebenen Technologie dar, was zugleich die resultierende Qualität erklärt.

Eine wichtige Eigenschaft der DSP-Algorithmen zeichnet sich durch den hohen Grad der Parallelität aus, wie dies bei FFT oder FIR Algorithmen der Fall ist. Diese Eigenschaft wird in Hardware-Implementierungen, wie den ASICs und auch den DSPs ausgenutzt, um die Leistungssteigerungen zu erreichen. Bei rekonfigurierbaren Architekturen wird ebenfalls versucht, durch eine große Anzahl von Funktionseinheiten die Parallelität auszunutzen. Die mögliche Nebenläufigkeit der Anwendungen, wie dies in der Telekommunikation oder bei Multimediaanwendungen der Fall ist, ist jedoch kein Zufall, sondern wird von entsprechenden Standardisierungsgremien bei IEEE oder ITU gezielt durch die Auswahl der Algorithmen ausgewählt und standardisiert. Diese Gremien sind durch zahlreichen Persönlichkeiten aus der Wirtschaft und universitärem Umfeld besetzt, so das neben politischen Gründen für die Auswahl der Anwendungen das Augenmerk auch auf die Qualität der Implementierungen gelegt wird. Insbesondere ist es von Bedeutung, dass die Anwendungen in Hardware effizient implementierbar sind.

Der Erfolg einer neuen Architektur ist wesentlich davon abhängig, ob die Programmierbarkeit gewährleistet ist. Eine angemessene abstrakte Programmiersprache in Kombination mit einem Compiler ist die ideale Ausgangslage. Vorzugsweise sollte hier die Hochsprache C genutzt werden, die den meisten Software-Entwicklern bekannt ist. Auf diese Weise wäre die Akzeptanz für die Architektur höher und gleichzeitig würden sich eine Reihe von Anwendungen erschließen, denn schließlich liegen bereits viele Applikationen in Form von C-Beschreibungen vor. Die Übersetzung der C-Beschreibungen in effiziente Konfigurationen für rekonfigurierbare Architekturen gestaltet sich allerdings nicht einfach. Denn selbst eine abstrakte Programmiersprache wie C bietet eine

Reihe von Funktionen, deren Anwendung nicht immer leicht übersetzbar ist. Zu dieser Kategorie gehören insbesondere Zeiger, die dynamisch in Programmen genutzt werden können und deshalb ihr Verhalten nicht leicht extrahierbar ist. Die einzige Möglichkeit das exakte Verhalten zu bestimmen, wäre durch das Profiling der Anwendung, was wiederum viele Voraussetzungen erfüllen muss, beginnend mit einem passenden Satz an Anwendungsdaten. Bei Prozessoren mit direktem Speicherzugriff (DMA) ist eine dynamische Speicherallokation kein Problem, da Variablen bzw. Speicherplatz beliebig reserviert und wieder freigegeben werden kann. Speziell bei arraybasierten Architekturen muss die Speicherallokation im Voraus bekannt sein, da diese Tatsache einen erheblichen Einfluss auf die Struktur und die Ausführung der Anwendungen hat.

Die Notwendigkeit eines Compilers für eine Architektur hat neben dem Komfortaspekt und der technischen Anwendbarkeit auch marktstrategische Konsequenzen. In unserer modernen Zeit ist der Faktor time-to-market von zentraler Bedeutung. Marktteilnehmer, die nicht rechtzeitig mit ihren Produkten auf den Markt kommen, unterliegen ihrer Konkurrenz im Wettbewerb. In diesem Zusammenhang sind technische Lösungen gefragt, die dabei helfen, die Entwicklung zeitnäher marktreif zu realisieren. Wenn die Entwickler sich mit den Details der Architekturinterna beschäftigen müssen, da beispielsweise kein C-Compiler vorliegt und stattdessen eine Low-Level-Sprache wie Assembler genutzt werden muss, verlieren sie wertvolle Zeit, die den Unterschied zwischen Erfolg und Misserfolg ausmachen kann.

### 3.5.1. Übersicht

Im Umfeld rekonfigurierbarer Systeme sind in den letzten Jahrzehnten mehrere Architekturen entstanden, die das Ziel verfolgen, hardwarenah mit größtmöglicher Effizienz und Flexibilität gewünschte Applikationen auszuführen. Dies ist nur möglich, wenn so wenig Abstraktion von der Hardware erfolgt wie absolut nötig und die Applikation wie ein ASIC möglichst direkt auf der Hardware mit minimalem Overhead abläuft. Dieser Umstand führt dazu, dass pro ausgeführter Operation möglichst wenig Schaltaktivität erfolgt. Es ist also wichtig, dass die Anzahl der Schaltvorgänge reduziert wird. Das gewünschte Resultat ist eine sehr sparsame Architektur, die sehr nah an die Eigenschaften eines ASICs herankommt. Nunmehr wird eine Übersicht zu den bestehenden Architekturen gegeben. Dabei wird zunächst eine Auswahl an Architekturen in Kurzform beschrieben, um der Vielfalt der Lösungen gerecht zu werden. Eine Auswahl der Ansätze wird anschließend detailliert beschrieben.

Das Pleiades Projekt [79] bietet ein Template für ultra-low-power high-performance rekonfigurierbares Computing. Es beherbergt einen zentralen Host-Prozessor in Kombination mit einem heterogenen Array von autonomen anwendungsabhängigen Satellit-Prozessoren. Der resultierende Aufbau ist stark auf die Zielapplikation zugeschnitten und bietet unter Umständen wenig Support für neue Applikationen. Ein bekannter Vertreter für die Anwendung des Pleiades Templates ist der Maia Chip [80], der speziell auf die Anforderungen der Sprachkodierung zugeschnitten ist. Der Nachteil dieser Architektur besteht im komplizierten heterogenen Aufbau, der kaum Raum für die gewünschte Flexibilität bietet.

Die PipeRench Architektur [81] basiert auf einer mehrfach ausgeführten Pipeline-Struktur. Die nebenläufigen Pipelinestufen sind durch ein Interconnect-Netzwerk verbunden, so dass Daten zwischen den Pipelines getauscht werden können. Weiterhin ist ein globales Netzwerk implementiert, damit der Datentransport entgegen des Pipeline-Flusses erfolgen kann. Die genaue Ausführung der Architektur ist parametrisierbar und kann den Gegebenheiten der Anwendungen angepasst werden.

Bei PADDI [85] handelt es sich um eine MIMD-Architektur. Hier finden sich mehrere Nano-Prozessoren um eine Switch-Struktur angeordnet, die jeweils VLIW-ähnliche Befehle verarbeiten können. Die Switch-Struktur ist darauf ausgelegt, konfliktfrei Prozessoren kommunizieren zu lassen, besitzt allerdings einen komplexen Aufbau, welcher sich auf die Skalierbarkeit negativ auswirkt.

Die RaPiD-Architektur [82] ist ein lineares Array funktionaler Einheiten. Sie besitzt lokale Speicher und wird wie eine lineare Pipeline konfiguriert. Sie eignet sich gut für irreguläre Anwendungen, besitzt aber Schwächen in der Verarbeitung blockorientierter Algorithmen.

MorphoSys [83] bietet alle notwendigen Komponenten zur Kontrolle und Verarbeitung in einem Design. Ein TinyRISC Prozessor ist zuständig für die Kontrolle, während ein Array aus Processing Elements (PEs) die Verarbeitung übernimmt. Das Array zeichnet sich durch ein multi-level Interconnect-Netzwerk zum optimalen Datenaustausch aus. Lokale Speicher für Daten und Konfigurationen versorgen das Array mit nötigen Informationen und entkoppeln gleichzeitig das Array vom Host-Interface. Streaming-Anwendungen stellen das Array vor externe Bandbreitenprobleme, wie dies häufig bei Array-Ansätzen der Fall ist.

MATRIX [84] ist eine MorphoSys-ähnliche Architektur, besitzt jedoch keinen integrierten Prozessor für die Kontrolle des Arrays. Dadurch ist die

Steuerung des Arrays nicht so komfortabel wie dies beispielsweise bei MorphoSys der Fall ist. Die Datenverarbeitung geschieht mit 8-bit Genauigkeit.

REMARC [86] ist ein Array basierend auf einfachen 16-bit Nano-Prozessoren, die durch lokale Verbindungen kommunizieren. Das Array wird gesteuert durch eine Global Control Unit. Sie übernimmt den Daten- und Konfigurationstransport und bietet nur eine geringe Bandbreite zum Host-System. Die Architektur ist für Multimedia-Anwendungen ausgelegt, besitzt jedoch keine integrierten Multiplizierer, was eine deutliche Schwäche darstellt.

RAW [87] ist ein weiteres Beispiel für eine RISC-Prozessor-basierte Architektur. Die Kommunikation zwischen den Prozessoren ist realisiert durch eine switch-basierte Struktur, die Punkt-zu-Punkt-Verbindungen aufbauen kann. Der Ansatz ähnelt ebenfalls einem Multiprozessor und seine Nachteile liegen in der komplizierten Programmierung.

Der Umfang der Ansätze im Bereich rekonfigurierbarer grobgranularer Architekturen der letzten fünfzehn Jahren ist enorm und es können nicht alle Architekturen hier ihre Beachtung finden, da dies den Rahmen dieser Ausarbeitung sprengen würde. Nachfolgend werden Architekturen beschrieben, die durch ihre innovativen Ideen aufgefallen sind und deshalb besondere Beachtung verdienen.

### 3.5.2. QuickSilver ACM

Ein prominenter Vertreter der kommerziellen rekonfigurierbaren Architekturen ist die Adaptive Computer Machine (ACM) [88] der Firma QuickSilver Technology [89]. Während die Konkurrenz in diesem Umfeld vorzugweise auf homogene Arrays oder prozessorartige Architekturen setzt, basiert der ACM-Ansatz auf einem völlig ungewöhnlichen Konzept. Basierend auf Anwendungsanalysen erkannten die Schöpfer dieser Architektur, dass Anwendungen zumeist inhomogene Anforderungen besitzen und infolge dessen auch die Architektureigenschaften ähnliche Parameter aufweisen müssen. Während arithmetische Operationen grobgranulare Ressourcen benötigen, profitieren Kontrollmechanismen von feingranularen Strukturen. Weiterhin ist es von Vorteil Zustandsmaschinen kostengünstig umzusetzen. Die genaue Mischung der benötigten Ressourcen für jede betrachtete Anwendung ist sehr spezifisch und dennoch versprach QuickSilver Technology eine sehr breite Abdeckung durch die ACM.

Das Erkennen der inhomogenen Natur von Anwendungen durch QuickSilver führte zu einem Mischaufbau der ACM, mit dem Ziel sowohl grobgranulare als auch feingranulare Verarbeitung zu bieten. Der Grundaufbau der ACM besteht aus Knoten unterschiedlichen Typs, die fraktal miteinander verbunden

sind, siehe Abbildung 30. Die verwendeten Knoten definieren die Art der Datenverarbeitung und sind in fünf verschiedenen Ausführungen implementiert: arithmetisch, bit-level, FSM, skalar und konfigurierbare Ein-/Ausgabe. QuickSilver stellt in diesem Zusammenhang heraus, dass jeder Knoten im Stande ist, komplette algorithmische Elemente auszuführen. Als algorithmisches Element bezeichnet QuickSilver hierbei algorithmische Einheiten einer Anwendung, die in sich geschlossen definiert sind, wie FIR, FFT oder DCT.

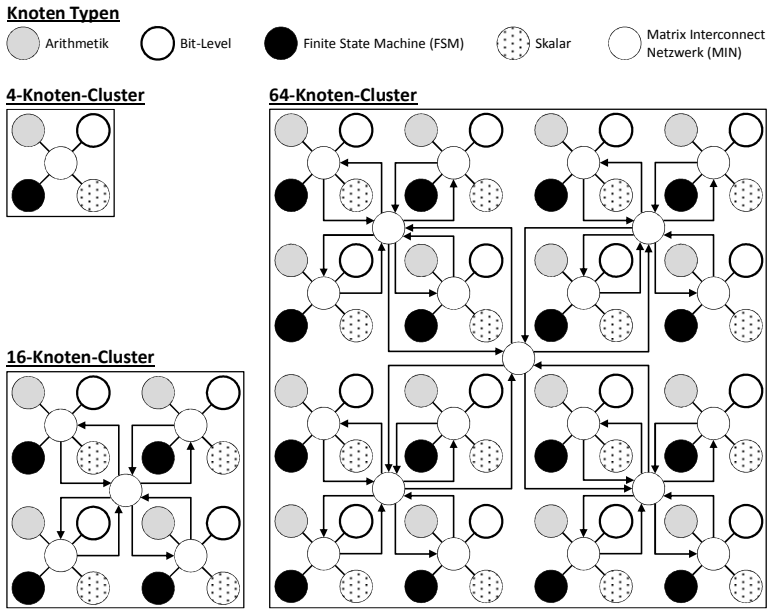



**Abbildung 30 Fraktale Struktur der ACM Architektur mit drei Ausbaustufen**

Arithmetische Knoten lassen sowohl die Realisierung von linearen als auch nicht linearen Funktionen zu. Dazu besitzen diese Knoten mehrere MAC-Einheiten und ALUs, die auf Vektor-Level arbeiten. Bit-Level-Knoten sind im Stande Bit-Level-Operationen unterschiedlicher Breite zu realisieren. Dazu gehören ebenfalls Funktionen wie linear rückgekoppelte Schieberegister (LSFP), Walsh-Code und GOLD-Code Generatoren, die eine Kombination von LSFR Funktionen voraussetzen. FSM-Knoten lassen die Abbildung von Zustandsmaschinen unterschiedlicher Klassen zu. Wird eine Zustandsmaschine zu komplex, so lassen sich mehrere FSM-Knoten der ACM zu ihrer Realisierung nutzen oder ihre Realisierung in eine Folge von Zeitschritten zerlegen. Skalare Knoten beinhalten einen RISC Prozessor und sind in der Lage herkömmlichen Prozessor-Code auszuführen. Konfigurierbare Ein-/Ausgabe

Knoten sind zusätzlich im Stande Bus-Interfaces zu emulieren. Dies schließt Standard-Interfaces wie PCI, USB und Firewire ein. Darüber hinaus lassen sich auch proprietäre Hoch-Performance-Interfaces umsetzen. In der Abbildung 30 ist dieser Knotentyp nicht dargestellt.

Eine wichtige Eigenschaft der ACM-Knoten besteht darin, die auszuführende Funktion von Zyklus-zu-Zyklus umzuschalten. Dies ist eine wichtige Multi-Context-Eigenschaft der Architektur, die zum Zeitpunkt der Einführung ebenfalls in der DRP-Architektur [90] Anwendung fand. Dadurch lassen sich nicht nur Anwendungen einzelner Knoten, sondern auch die der kompletten Architektur schnell umschalten. Die Umschaltgeschwindigkeit erlaubt hundertfachen oder gar tausendfachen Wechsel der Anwendungen pro Sekunde und eröffnet eine neue Sicht auf die Anwendung der Architektur. Das Resultat ist ebenfalls eine deutliche Verbesserung der Ausnutzung der Architektur-Ressourcen, die laut den Autoren sogar ASICs im Hinblick auf die Gatter-Ausnutzung Konkurrenz bieten kann.

Jeder Knoten besitzt sowohl die erforderliche Logik zur Datenmanipulation als auch lokale Speicher für Anwendungsdaten sowie Konfigurationsdaten. Der Konfigurationsspeicher ist mit einer Interface-Breite von 32 bis 128 Bits ausgestattet, was eine schnelle Übertragung der Konfigurationsdaten zulässt. Zusätzlich können mehrere Konfigurationen in diesem Speicher enthalten sein und von Takt zu Takt umgeschaltet werden. Die Größe der Speicher beansprucht ca. 75% der gesamten Knotenfläche. Das Konzept der ACM sieht vor, Daten einer Anwendung möglichst lokal an einem Ort vorzuhalten, während die Konfiguration dieser Anwendung durch schnelles Umschalten am selben Ort angepasst wird. Hiermit wird eine aufwändige Verdrahtung der Architektur vermieden. Das Konzept geht noch einen Schritt weiter und stellt die Forderung auf, dass Knoten, die weiter auseinander sind, auch durch weniger Leitungen verbunden sind. Im Umkehrschluss sind Knoten, die näher zu einander liegen, mit einer höheren Bandreite verdrahtet. Diese Forderung legt den Aufbau des ACM-Netzwerks ähnlich dem eines Generalized FAT Trees [91] nahe und entspricht im Wesentlichen der fraktalen Natur der Anwendungen, die sich im heterogenen Aufbau der ACM wiederspiegelt, siehe Abbildung 30.

Die kleinste Einheit der ACM stellt das 4-Node-Cluster dar, welches die oben beschriebenen Knotentypen (arithmetisch, bit-level, skalar und FSM) mit je einem Vertreter enthält. Diese Knoten sind durch ein Matrix-Interconnect-Network (MIN) miteinander verbunden und bilden die Grundeinheit der ACM Architektur. Das MIN-Netzwerk ist sowohl für den Transport von Anwendungsdaten als auch Konfigurationsdaten konzipiert. Vier 4-Node-Cluster können zu einem 16-Node-Cluster zusammengesetzt werden, indem die MIN-Knoten mit einem übergeordneten MIN verbunden werden. Dieses Vorgehen

lässt sich beliebig wiederholen und somit bei Bedarf auch ein sehr großes ACM-Array aufbauen. Die eigentliche Beschränkung bildet in diesem Zusammenhang die maximale wirtschaftlich nutzbare Chip-Größe.

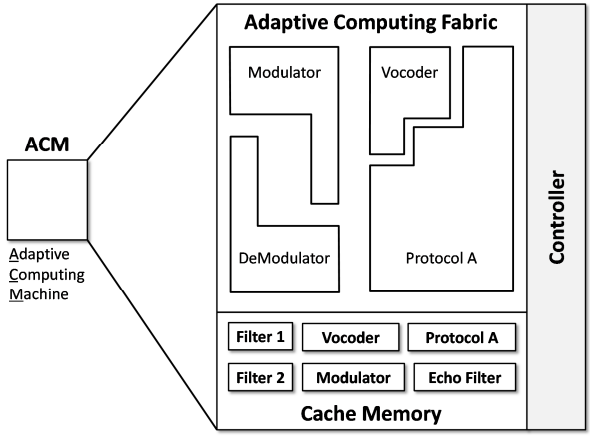



**Abbildung 31 Virtualisierte Hardware der ACM Architektur**

Das Programmiermodel der ACM setzt auf eine eigene Hochsprache mit dem Namen SilverC™. Dabei handelt es sich im weiteren Sinne um die Programmiersprache C, die um zusätzliche Sprachkonstrukte erweitert wurde. Sie erlauben es dem Programmierer den nötigen Einfluss auf die temporalen und räumlichen Parameter der entwickelten Anwendung zu nehmen. Kompilierte Anwendungen für die ACM werden als SilverWare™ bezeichnet und sind in der Form von der ACM ausführbar. Abbildung 31 zeigt die virtualisierte Sicht der Applikationen auf die ACM Hardware. Einmal geladen, enthält der Cache Memory auszuführende Anwendungen, die vom Architektur-Controller aufgerufen werden können. Der Controller steuert den Ladevorgang, die Ausführung und den Transport der Daten. Die eigentliche Ausführung der Anwendung erfolgt durch die Adaptive Computing Fabric. Das Vermögen der ACM Algorithmen und Applikationen auszuführen hat QuickSilver Technology in mehreren Demonstrationen und Publikationen belegt [92][93]. Hierzu gehören AlgoPAErithmen wie RSA [94] und komplette Applikationen wie MPEG4 [53] und W-CDMA [95].

Die Realisierbarkeit der Konzepte dieser Architektur hat QuickSilver Technology mittels des gefertigten Prototyps QS2412 gezeigt. Dieser Chip beinhaltete 12 Knoten und stellte einen funktionsfähigen Demonstrator dar. Das Geschäftsmodell von QuickSilver Technology basierte auf der Idee, die ACM als IP (Intelectual Property) an seine Geschäftspartner zu lizensieren.

Trotz der vielversprechenden Resultate musste QuickSilver Technology seinen Betrieb im Jahr 2005 einstellen. QuickSilver Technology besaß zu diesem Zeitpunkt den Status eines StartUps und war auf externe Geldgeber (Venture-Capitals) angewiesen. Offiziell wird das Scheitern der Verhandlungen mit Geldgebern als Grund für die Einstellung des Betriebs angegeben.

### 3.5.3. PACT XPP-III Prozessor

Ein weiterer bekannter Vertreter der kommerziellen rekonfigurierbaren grob-granularen Architekturen ist die XPP-Architektur [97] der Firma PACT XPP Technologies AG [96]. Es handelt sich hierbei um ein homogenes Array von Processing Array Elements (PAE) unterstützt durch eine Reihe hochoptimierter Prozessoren, sogenannter Function-PAEs (FNC). Das Array war dazu gedacht hochparallel Anwendungen mit regulärem Datenfluss zu verarbeiten, wohingegen die FNCs irregulären Code mit vielen Sprüngen und Verzweigungen zum Ziel hatten. Idealerweise werden Datenflussgraphen direkt auf das XPP-Array abgebildet, so dass komplette Pipeline-Strukturen darauf ablaufen können.

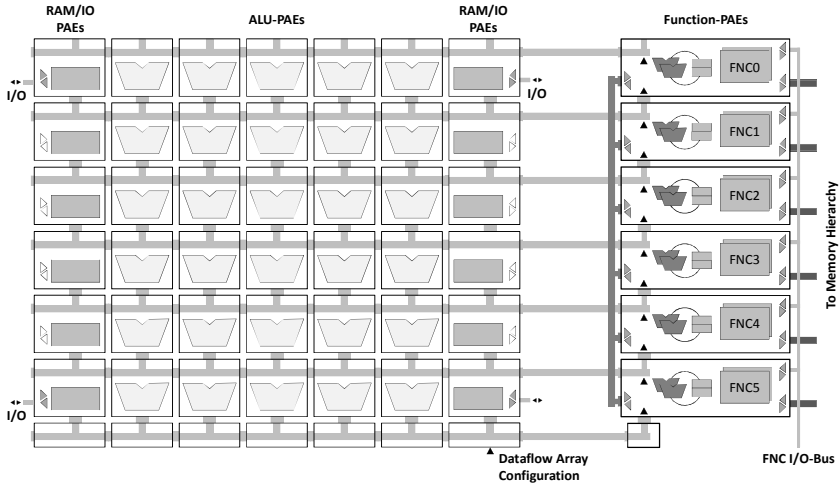



**Abbildung 32 Exemplarischer Aufbau des XPP-III Kerns mit einer 5x6 Array-Konfiguration**

Der Aufbau des XPP-Arrays ist zellbasiert und enthält im Wesentlichen zwei Zelltypen: ALU-PAEs und RAM-PAEs. ALU-PAEs befinden sich im Inneren des Arrays. Links- und rechtsseitigen Abschluss des Arrays stellen die RAM-PAEs dar. Rechts außerhalb des geschlossenen Arrays ist ein zusätzli-

cher Spalt mit FNCs zu finden. Abbildung 32 zeigt den Aufbau des XPP-Arrays. ALU-PAEs enthalten drei ALUs zur Datenverarbeitung, während RAM-PAEs zwei ALUs, ein Speichermodul und ein optionales I/O-Objekt beinhalten.

Die Kommunikationsstruktur der XPP-Arrays ist eine optimierte Komponente und stellt eine Besonderheit dar. Während konkurrierende Architekturen das Routing als selbständiges Transportmittel betrachten, ist im Falle der XPP das Routing integrierter Bestandteil der Recheneinheiten. Die horizontalen Busse verlaufen am oberen Rand jeder PAE innerhalb des Arrays. Sie verbinden jede PAE einer Zeile miteinander, können jedoch zusätzlich segmentiert werden. Jedes Segment kann dabei unabhängig Daten transportieren, sodass jeder segmentierte Bus mehrfach nutzbar wird. Vertikale Busse verlaufen durch die Komponenten der ALU-PAEs und RAM-PAEs mit jeweils zweifacher Verbindung nach unten und einfacher nach oben. Der vertikale Transport ist so mit der Datenverarbeitung verknüpft und erledigt zwei Schritte in einem. Die verwendete Busbreite kann anwendungsabhängig gewählt werden und erlaubt drei Größen: 16, 24 oder 32 Bits. Neben den Datenvektoren transportieren die Kommunikationsbusse unabhängig 1-Bit-breite Event-Daten. Zwischen den FNCs existiert ein weiterer vertikaler Bus. Die Übertragung der Daten im Kommunikationsnetzwerk wird mit einem Hand-Shake-Protokoll gesichert, so dass der Anwender keine Rücksicht auf den Datenfluss nehmen muss. Eine Einheit führt ihre Operation erst dann aus, wenn alle Eingänge gültige Daten aufweisen.

Die ALU-PAEs bestehen aus drei XPP-Objekten: FREG (forward Register), ALU und BREG (backward Register). Abbildung 33 zeigt die Struktur einer ALU-PAE. Jedes XPP-Objekt ist imstande Datenoperationen auszuführen. Eingangsseitig werden die Daten in Registern gepuffert. Um das Ungleichgewicht der Pipelines auszugleichen, werden 1-stage-FIFOs genutzt, die optional zusätzlichen Puffer bieten. Bei Bedarf können die Register auch als Konstanten konfiguriert werden und liefern dauerhaft die vorgegebenen Werte. Ausgangseitig dienen DF-Register zum Sichern der Daten im Falle eines Pipelinestalls und verursachen im Normalbetrieb keine zusätzlichen Verzögerungen. Die eingangs- und ausgangsseitigen Register/FIFOs sind optional an die horizontalen Busse angebunden und können bei Bedarf mit dem gewünschten Bus verbunden werden. Diese Aufgabe übernehmen die Switch-Objekte, die auch die Segmentierung der horizontalen Busse realisieren.

Das ALU-Objekt in der Mitte der ALU-PAE bietet logische und arithmetische Operationen. Operationen wie Multiplikation und Komparatoren sind ebenfalls enthalten. Der Eingang für das Event-Signal (gestrichelt dargestellt) dient zur Steuerung der ALU-Operationen, kann aber auch als Carray-In/-Out-Flag genutzt werden. Zusätzlich kann das Event-Signal als Trigger für die

anschließende Rekonfiguration dienen. Das XPP-Array unterstützt prinzipiell partielle Rekonfiguration. Diese Funktion muss allerdings im Vorab geplant werden, da sich sonst Überschneidungen oder Doppelbelegungen ergeben können. Durch den statischen Charakter des Routings müssen belegte Ressourcen zuvor bekannt sein und während der Kompilation von der Nutzung ausgeschlossen werden. Auch sind einmal kompilierte Anwendungen in ihrer Struktur und Position statisch. Sind Änderungen an diesen Parametern erwünscht, so ist eine Rückkehr zum Compiler unumgänglich.

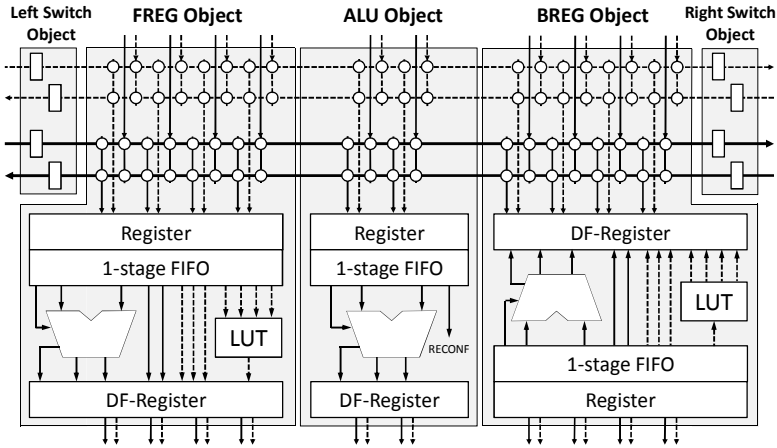



**Abbildung 33 Struktur und Komponenten einer ALU-PAE mit horizontalen Bussen**

Die BREG- und FREG-Objekte haben einen ähnlichen Aufbau wie das ALU-Objekt, weisen allerdings einen eingeschränkten Operationsumfang auf. Die Aufgaben dieser Objekte bestehen darin, vertikal Daten zwischen den horizontalen Bussen zu transportieren, sowie Multiplex/Demultiplex- und einfache arithmetische Operationen zur Verfügung zu stellen. Darüber hinaus sind LUTs integriert, die boolesche Funktionen auf Event-Signale anwenden können. Zusätzlich beinhalten BREG-Objekte Barrel Shifter, sowie Packing/Unpacking und Clipping Operationen. FREG-Objekte hingegen besitzen zusätzliche Zähler und Akkumulatoren. In Kombination mit dem ALU-Objekt können somit MAC-Operationen realisiert werden. Die Anzahl der vertikalen Verbindungen innerhalb dieser Objekte ist auf RTL parametrisierbar. RAM-PAEs beinhalten ebenfalls die FREG- und BREG-Objekte. Anstelle des ALU-Objektes werden jedoch ein RAM-Objekt und ein I/O-Objekt genutzt. Mit dem I/O-Objekt lassen sich Verbindungen nach außen realisieren, während das RAM-Objekt einen Zwei-Port-SRAM zur Verfügung stellt und zwei Be-

triebsmodi erlaubt: RAM und FIFO. Die Größe des SRAM ist ein RTL-Parameter.

Der Datentransport der ALU-, FREG-, RAM- und I/O-Objekte erfolgt abwärts, während das BREG die Daten aufwärts leitet. Durch die passende RTL-Parametrisierung der FREG-/BREG-Objekte kann dennoch ein Gleichgewicht der vertikalen Verbindungen erreicht und somit der augenscheinlich vorwiegende Transport der Daten nach unten ausgeglichen werden.

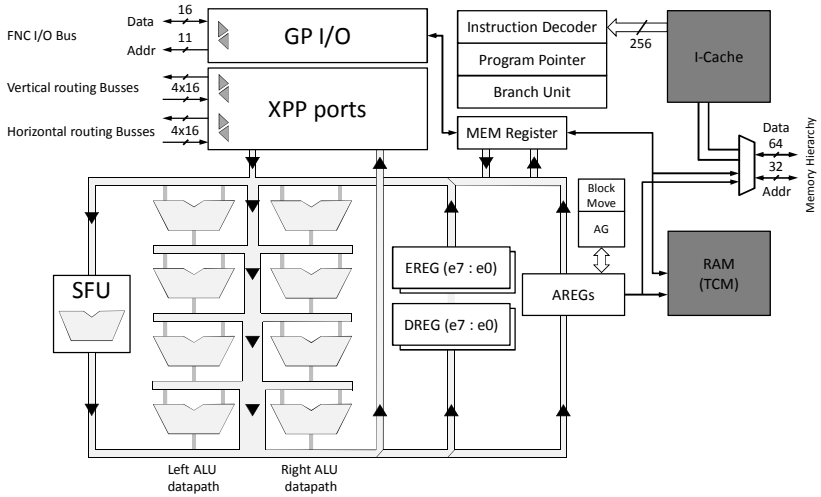



**Abbildung 34 Aufbau der Function-PAE (FNC)**

Abbildung 34 zeigt den Aufbau der FNC mit folgenden Komponenten: 2x4-Array von 16-bit ALUs, Special Function Unit (SFU), 16-bit Register File (EREG/DREG), 32-bit Adressgenerator und Adress-Registerfile (AREG), Instruktions-Cache, lokalem Speicher (TCM), Instruktionsdekoder, Branch-Unit, Speicher-Port Ein-/Ausgangs-Register (MEM Register), FNC und XPP I/O-Ports. Die Funktionsweise des FNC ist vergleichbar mit einem VLIW-Prozessor.

Das 2x4-ALU-Array ist rein kombinatorisch aufgebaut, wobei jede ALU einfache arithmetische, logische, Vergleichs- und Shift-Operationen beinhaltet und kann sie bedingt ausführen. Jede ALU ist auch in der Lage ihre Operanden von allen Register-Files sowie Ergebnisse der ALUs in den darüber liegenden Zeilen des ALU-Arrays unabhängig zu lesen. Somit können Eingangsdaten bis zu vier ALU-Stufen in einem Takt passieren und erhöhen damit den IPC. Selbst kleine If-Strukturen mit Abhängigkeiten können von der FNC in einem Takt ausgeführt werden, solange die Anzahl der notwendigen ALUs die vorhandene Anzahl nicht übersteigt. Hierfür haben die Entwickler spezielle Kon-

trollmechanismen in den Instruktionsdekoder und die Branching-Unit eingebaut, die im Detail bisher nicht publiziert wurden. Letztere erlaubt das Auswerten von bis zu drei Zieladressen in einem Takt. Die SFU arbeitet parallel zum ALU-Array. Sie bietet zwei 16-bit Multiplikationen in einem Takt und weitere Funktionen wie Bit-Field-Extraction. Die Eingangswerte der SFU kommen aus den Register-Files Die Ergebnisse werden wieder in die Register-Files zurückgeschrieben und stehen im nächsten Takt wieder zur Verfügung. Ein lokaler 256x256-bit 4fach assoziativer Instruktions-Cache sorgt für die ausreichende Bandbreite der FNC. Spezielle Funktionen wie Line-Lock helfen dabei den Cache zu steuern. Die Daten werden im 16-bit breiten TCM vorgehalten. Der Zugriff auf den externen Speicher erfolgt über ein 32-bit Interface mit 64-bit breiten Datenwörtern. Zur Beschleunigung der Zugriffe können die Transfers in Blöcken (Bursts) organisiert werden.

Nach dem Reset der Architektur bootet zunächst die erste FNC (FNC0). Sie kann nach Bedarf weitere FNCs starten und/oder einen DMA-Controller veranlassen, das XPP-Array zu konfigurieren. Die Konfigurationsdaten werden durch die Konfigurationsschnittstelle (unten rechts) in das Array übertagen und durch einen internen Konfigurationsbus im Array verteilt. Durch die Abbildung mehrerer Applikationen und Algorithmen [98], wurden die Fähigkeiten der XPP-Architektur demonstriert.

Das Programmiermodel des XPP-III-Prozessors setzt auf die Hochsprachen C/C++. Zur Übersetzung der Anwendungen bietet PACT XPP Technologies Design-Flow-Tools mit dem Namen PACT Software Design System. Das Mapping auf das XPP-Array und FNCs setzt derzeit das manuelle Partitionieren voraus [99][100]. Voraussetzung dafür sind die regulären bzw. irregulären Eigenschaften der Anwendungen. Der aufgeteilte Code kann mit einem Zwischenschritt in den Assembler-Code übersetzt werden und schließlich in den Maschinen Code. Bei der FNC werden hierzu eine spezielle Version des GCC-Compilers (FNC-CC) und ein eigenes Assembler-Tool (XFNCASM) herangezogen. Im Falle des XPP-Array wird zum Übersetzen XPP-VC genutzt und zum Assemblieren von NML nach Maschinen-Code XMAP.

Das Geschäftsmodell der PACT XPP Technologies AG basiert auf IP-Lizensierung und Vermarktung der Patentrechte. Den Beweis der Realisierbarkeit ihrer Architektur in Silizium lieferte PACT XPP Technologies zuletzt mit der Teilnahme im EU-Projekt Morpheus und anschließender Fertigung eines Silizium-Prototyps [101].

### 3.5.4. DRP - Dynamisch Rekonfigurierbarer Prozessor

Der Dynamically Reconfigurable Processor (DRP) [90] wurde 2002 von NEC Electronics (heute Renesas Electronics) [102], entwickelt und vorgestellt.

Das Ziel der Entwicklung war es, einen flexiblen rekonfigurierbaren Prozessorkern für zukünftige SoC-Designs zur Verfügung zu stellen, um die entwicklungsintensiven ASIC-Designs zu ersetzen. Die Schöpfer der Architektur haben allerdings einen anderen Weg zum Erreichen dieses Ziel eingeschlagen und sind nicht den üblichen Trends der 'statischen' Konfigurationen gefolgt. Geprägt wurden diese Trends vor allem durch die Vorgaben der feingranularen Architekturen, wie der FPGAs. Allerdings sind die Möglichkeiten grobgranularer Architekturen vielfältiger, da die Datenmengen für die notwendigen Rekonfigurationen deutlich kleiner ausfallen. Eine Möglichkeit von diesen Eigenschaften zu profitieren zeigt die DRP-Architektur auf.

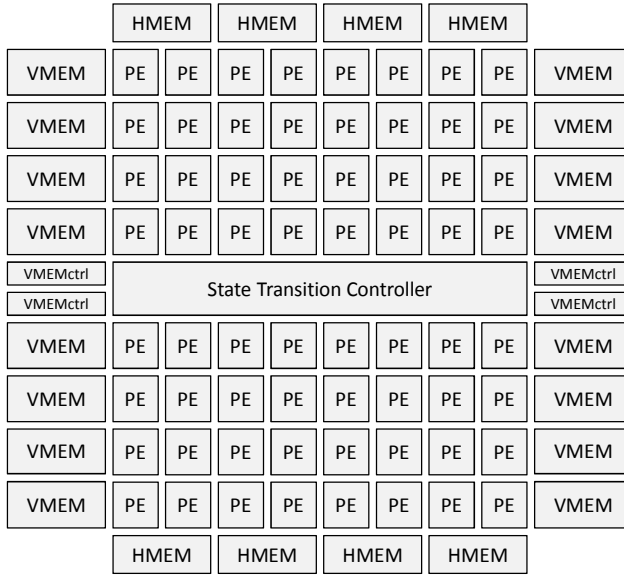



**Abbildung 35 Tile der DRP-Architektur mit 64 PEs, lokalen Speichern und Zustandkontrollmechanismen**

DRP ist ein grob-granularer dynamisch rekonfigurierbarer Prozessorkern für die Integration in ASICs oder SoCs. Es teilt sich in ein oder mehrere „Tiles“ auf. Ein Tile stellt somit eine Primitive des DRP dar, siehe Abbildung 35. Die rudimentären Komponenten eines Tile sind Processing Elements (PEs), State Transition Controller (STC), vertikale 2-Port-Speicher (VMEMs), Speichercontroller (VMCtrl) und horizontale 1-Port-Speicher (HMEMs).

Ein Tile beherbergt 64 PEs die in einem Array von zweimal 8x4 PEs angeordnet sind. Zwischen den Teil-Arrays befindet sich der STC. Innerhalb der PEs finden sich eine 8-bit ALU, 8-bit Data Management Unit (DMU), 16x8-bit

Register-File (RFU), ein 8-bit Register und ein Kontext-Speicher. Die DMU ist dazu gedacht, Schiebe- und logische Operationen zur Verfügung zu stellen. Das Interface der PEs besitzt zum einen zwei 8-bit Eingänge und einen 8-bit Ausgang für Daten und zum anderen Eingänge und Ausgänge für ALU-Flags, siehe Abbildung 36.

Die Verbindungen und Funktionen aller Einheiten sind programmierbar. Dazu muss zunächst ein Kontext aus dem Kontext-Speicher ausgewählt werden. Die Daten im gewählten Kontext spezifizieren das Verhalten der Einheiten und der Kommunikationsnetze. Beispielsweise werden Ein- und Ausgänge aktiviert oder in das Zielregister geschrieben. Der Kontext-Speicher besitzt 16 Einträge. Der Pointer für den Kontext-Speicher wird vom STC geliefert. STC ist ein programmierbarer Sequenzer, der für die PEs die Pointer generiert. Dazu wird eine vorberechnete FSM mit maximal 64 Zuständen in den STC programmiert und übernimmt die Steuerung des Tile. Eingehende Flags aus dem Tile beeinflussen die Zustandsübergänge des STC. Die maximale Anzahl der Verzweigungen aus einem Zustand ist mit vier angegeben. Jeder Zustand der FSM ist in diesem Zusammenhang mit einem Kontext des Tile verknüpft.

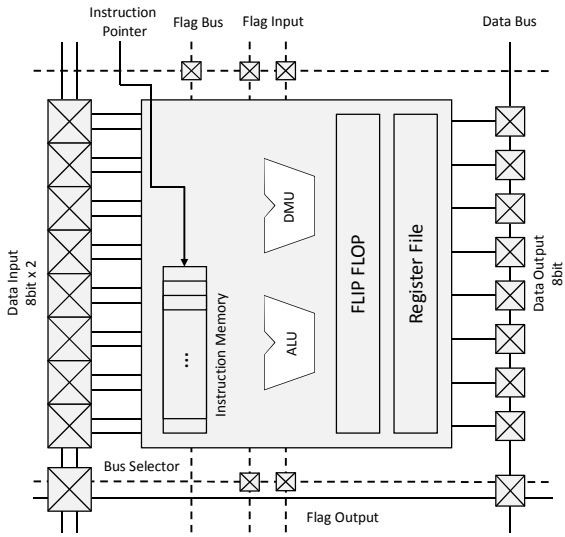



**Abbildung 36 8-Bit Processing Element (PE) der DRP-Architektur**

Jedes Tile besitzt 16 VMEMs und 8 HMEMs. Beide Speicher besitzen eine Wortbreite von 8 Bit. VMEMs besitzen weiterhin 256 Einträge, HMEMs hingegen 8182 Einträge. VMEMs können zusätzlich als FIFOs genutzt werden. Weiterhin können alle Speicherinhalte, Register- und Register-File-Werte von

allen Kontexten geteilt werden. In einem DRP-Kern mit mehreren Tiles steuert eine zentrale STC-Einheit (CSTC) die FSMs der anderen Tiles, siehe Abbildung 37. Bei Bedarf können die lokalen STCs jedoch auch unabhängig zum CSTC laufen.

Ähnlich zur XPP-Architektur gestaltet sich die Anwendung der partiellen Rekonfiguration auf die DRP-Architektur als schwierig. Anwendungen, die nicht darauf ausgelegt sich mit anderen Anwendungen auf der DRP-Architektur ausgeführt zu werden, würden Ressourcen-Konflikte verursachen. Diese Funktion muss also im Vorab geplant und betreffende Anwendungen gemeinsam kompiliert werden.

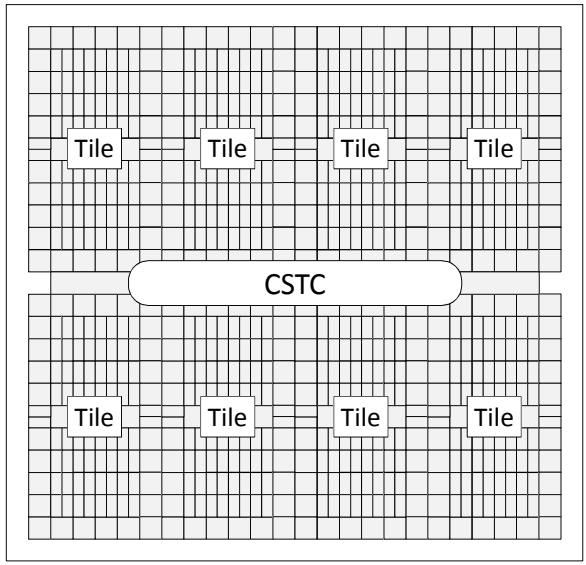



**Abbildung 37 DRP-Architektur bestehend aus acht Tiles und dem zentralen CSTC.**

Vergleichbar mit den Argumenten der ACM-Schöpfer, verfolgen die Entwickler des DRP die Strategie der Datenlokalität. Dabei wird versucht die Anwendungsdaten im Array möglichst wenig zu bewegen und stattdessen auf Rekonfiguration der umgebenden Funktionalität zu setzen. Diesem Ansatz kommt das Konzept der Multikontext-Rekonfiguration entgegen. Das Programmiermodel des DRP bietet einen High-Level-Compiler, der C-Programme direkt in ausführbaren Code übersetzen kann [103]. Ursprünglich wurde DRP gemäß den Anforderungen der C-Sprache konzipiert, so dass dieser Ansatz naheliegend ist. In diesem Design-Flow wird C mittels High-Level-Synthese (HLS) zunächst in mehrere Verilog-Beschreibungen übersetzt und anschlie-

ßend mittels des eigenen Technology Mapping Tools und eines weiteren Place & Route Tools in die finalen STC- und PE-Beschreibungen überführt. Die Thematik der HLS im Zusammenhang mit DRP wird auch von externen universitären Institutionen auch außerhalb Japans verfolgt und erforscht [104]. Die Anwendbarkeit der DRP-Architektur wurde in mehreren Publikationen gezeigt und schließt eine große Zahl an Algorithmen und Applikationen ein [90][105].

### 3.5.5. MONTIUM Prozessor

Im Rahmen des Chameleon Projektes der Universität Twente [106] wurde ein Framework für heterogene SoCs entwickelt. Das Ziel dieses Projektes war es unterschiedliche Komponenten, wie Prozessoren, DSPs, FPGAs, ASICs und/oder grobgranular rekonfigurierbare Architekturen in einem Framework zusammenzufassen. Dieses Framework erlaubte es den Beteiligten die Eigenschaften derartiger Systeme zu untersuchen und die Entwicklung voranzutreiben. Die Laufzeit des Projektes war für vier Jahre angesetzt und endete im Jahr 2004, allerdings sind daraus weitere nationale und EU-finanzierte Projekte hervorgegangen.

Für die Berechnung typischer DSP-Algorithmen mit 16bit Genauigkeit entstand in diesem Projekt der grobgranular rekonfigurierbare Prozessorkern Montium [107][108]. Der kompakte Aufbau und die resultierende Fläche von 2 mm² bei Verwendung der Phillips CMOS12 130nm Technologie erlaubt es mehrere Kerne auf einem Die unterzubringen und somit die Performance applikationsgerecht anzupassen. Abbildung 38 zeigt den Aufbau des rekonfigurierbaren Prozessors. Im unteren Bereich der Abbildung findet sich die Communication und Configuration Unit (CCU), im oberen Teil ist der eigentliche Prozessorkern (Tile Processor, TP) zu sehen.

Die CCU realisiert das Interface zum System durch die Anbindung an einen System-Bus oder Network-On-Chip (NoC). Die Interface-Spezifikation dieser Komponente ist damit umgebungsabhängig und muss an die Systemgegebenheiten angepasst werden. Die Hauptfunktion besteht darin, Applikationsdaten und Prozessorkonfigurationen zu transportieren. Zu diesem Zweck empfängt die CCU Instruktionen vom Centralized Configuration Manager (CCM) und steuert dadurch den TP. Es sind vier Zustände für die Kommunikation zwischen CCU und TP definiert. Soll TP zunächst konfiguriert werden, werden Konfigurationsdaten an die CCU durch den CCM verschickt und per DMA im Konfigurationsspeicher abgelegt. In dieser Phase werden die Aktivitäten des TP angehalten, um inkonsistente Resultate zu vermeiden. Im zweiten Schritt werden Applikationsdaten empfangen und lokal im TP gespeichert. Danach beginnt die Berechnungsphase. Währenddessen wartet die CCU auf

das Ende der Berechnungen, welches durch den internen Sequencer des TP signalisiert wird. Im letzten Zustand transportiert die CCU die Ergebnisse der Berechnungen aus dem TP zum vorgegebenen Ziel, wie Systemspeicher oder System-Interface. Danach besteht die Möglichkeit durch das Laden neuer Applikationsdaten und anschließendem Berechnungsstart bestehende Konfiguration nochmal zu verwenden oder eine Rekonfiguration des TP zu veranlassen. Darüber hinaus bietet die CCU spezielle Modi, die insbesondere Überlappungen bzw. Vertauschungen der obengenannten Zustände erlauben.

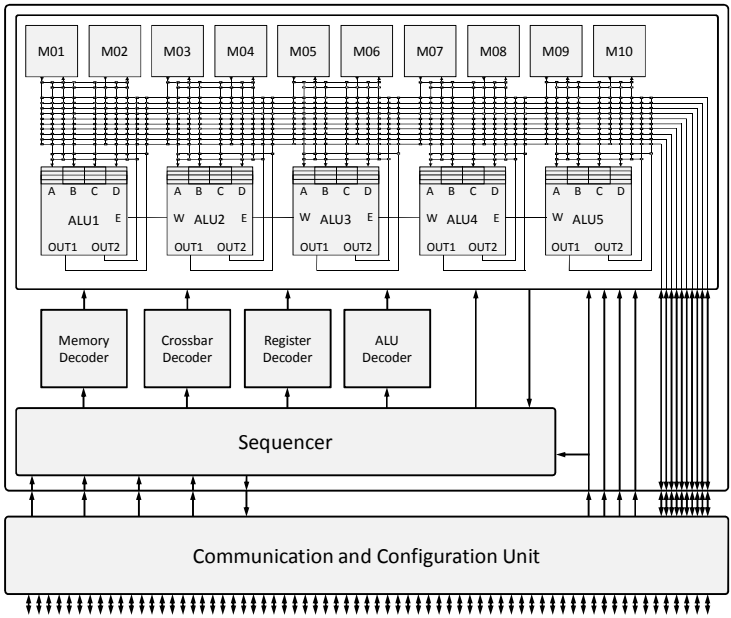



**Abbildung 38 Montium Prozessorkern mit fünf ALUs, zehn lokalen Speichern und Sequenzer-Mechanismen**

Der TP beherbergt einen Sequenzer, vier Decoder und das Processing Part Array (PPA). Das PPA ist die wichtigste Komponente des TP und beinhaltet vier ALUs, zehn lokale Speicher und ein programmierbares Interconnect. Die Kontrolle der Abläufe im TP wird durch den relativ einfachen Sequenzer realisiert, der durch eine vorprogrammierte Sequenz die vier nachfolgenden Decoder ansteuert und damit die Funktionalität des PPA bestimmt. Jeder Decoder steuert ihm zugeordnete Einheiten: lokale Speicher, Interconnect, Register und ALUs im PPA. Die maximale Speichergrößer für die vollständige Konfiguration des Montium beträgt 2,6 KB. Reale Konfigurationen liegen allerdings klar

unter diesem Wert und stellen deutlich die Stärke der grobgranularen Architekturen im Vergleich zu feingranularen Ansätzen heraus.

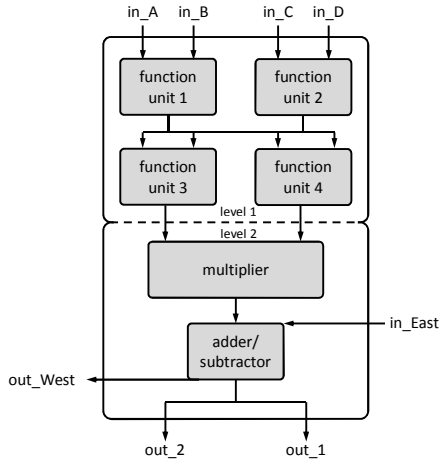



**Abbildung 39 Montium ALU-Interna mit zwei-stufigem Aufbau**

Der Aufbau des PPA ist regulär. Die implementierten 16bit ALUs besitzen vier Eingänge und zwei Ausgänge, siehe Abbildung 39. Angelehnt an die Tatsache, dass DSP typische Operationen, wie MAC, mehr als zwei Eingänge besitzen, profitieren die ALUs von einem komplexeren Aufbau und beschleunigen diese Operationen. Zusätzlich besitzen die ALUs einen East-Eingang und einen West-Ausgang, die beide 32 Bits besitzen. Damit können Ergebnisse einer ALU an die benachbarten ALUs weiterleitet und zur Beschleunigung komplexer MAC-Operationen genutzt werden. Interna der ALUs lassen sich in zwei Bereiche einteilen: Level-1 und Level2. Der Erstere beinhaltet Function Units (FUs), die grundsätzlich C-ähnliche Funktionen realisieren. Das Interconnect zwischen diesen Einheiten ist fest verdrahtet, bei Bedarf können diese Einheiten jedoch passiert werden und damit ungenutzt bleiben. Level-2 bietet einen Multiplizierer und einen Addierer/Subtrahierer. Letzter arbeitet mit 32bit Genauigkeit zur Realisierung der MAC-Operationen. Zusätzlich können in Level-2 der East-Eingang genutzt und/oder Ergebnisse an den West-Ausgang weitergeleitet werden. Da diese Einheiten keine Pipeline besitzen, verringert diese Kaskade die maximale Taktfrequenz der Architektur. Ähnlich zum Level-1 können die Operationen in Level-2 leicht passiert werden. Dies kann mittels Software geschehen oder die Einheiten können zum RTL-Zeitpunkt vollständig entfernt bzw. geändert werden. Diese Funktion bietet die Möglichkeit die Montium-Architektur an die Bedürfnisse der Applikationen anzupassen. Eingangsseitig besitzen die ALUs jeweils vier lokale Register, die wahl-

weise zum Verzögern der Eingangsdaten um ein oder mehrere Takte dienen können und nicht passierbar sind.
Die zehn lokalen Speicher des PPA sind als SRAM mit einem Lese- und einem Schreib-Port ausgeführt. Sie besitzen je einen konfigurierbaren Adressgenerator (AGU), der neben einfachen Adressfolgen auch bitinverse Folgen erzeugen kann. Die zehnfache Ausführung der Speicher bietet nach Ansicht der Entwickler genügend Bandbreite, um die fünf ALUs auszulasten.

Das programmierbare Interconnect des PPA besitzt 10 globale Busse, weitere lokale Busse zwischen jeweils zwei Speichermodulen und einer ALU, sowie lokale Speicherbusse. Die globalen Busse sind im Stande zwischen allen Ausgängen und Eingängen des PPA Verbindungen herzustellen. Weiterhin ist auch die CCU an die globalen Busse angeschlossen und hat somit direkten Schreib- und Lesezugriff auf die lokalen Speicher, die bei den oben beschriebenen Zuständen der CCU genutzt werden.

Das Programmiermodell des Montium-Prozessors setzt auf die Hochsprache C auf, besitzt jedoch auch Einschränkungen, wie die Verwendung von Zeigern in Programmen. Die vorliegende C-Beschreibung wird zunächst in einen Kontrolldatenflussgraphen (CDFG) transformiert. Diese Struktur beinhaltet sowohl Kontrollinformationen als auch Datenflussinformationen und eignet sich sehr gut zur Realisierung von vollständigen Abbildungen. Nach der Transformation des CDFG in eine Montium-spezifische Zwischendarstellung, wird die Beschreibung in Cluster partitioniert. Jedes Cluster kann von einer Montium-ALU in einem Takt ausgeführt werden. Nach dem folgenden Scheduling der Cluster und der Abbildung in die Zeit erfolgt die Ressourcen-Allokation. Das Ergebnis ist eine zyklusgenaue Beschreibung des Montium-Programms, die schließlich in ausführbaren binären Code überführt wird. Die Einschränkungen des C-Sprachumfangs betreffen die Nutzung von Zeigern im Programm-Code.

In mehreren Publikationen haben die Entwickler dieser Architektur die Anwendbarkeit demonstriert [109][110]. Trotz der verhältnismäßig begrenzten Parallelisierbarkeit aufgrund der geringen Anzahl an ALUs sind die Resultate vielversprechend und lassen sich durchaus mit hoch-parallelen Ansätzen wie XPP oder DRP vergleichen. Zugleich besitzt der Montium-Prozessor eine höhere Flexibilität als beispielsweise der XPP und stellt damit eine vielversprechende Architektur dar.

### 3.5.6. Asynchronous Array of simple Processors (AsAP)

Eine Forschergruppe am VLSI Computation Laboratory an der Universität von Kalifornien entwickelte eine Rechenarchitektur mit einer großen Anzahl von einfachen Prozessoren angeordnet in einem zwei-dimensionalen Array.

Die verwendeten Prozessoren sind sehr einfach in ihrer Funktionalität und beinhalten einen lokalen Speicher. Der Speicher ist in zwei Module aufgeteilt. Das Erste ist für Instruktionen vorgesehen und kann 64 Instruktionen aufnehmen. Das zweite Modul kann zur Speicherung von Daten verwendet werden und bis zu 128 16bit Wörter beinhalten. Die Kommunikation zwischen den Prozessoren geschieht über lokale Verbindungen zwischen den Zellen. Dafür sind alle direkten Nachbarn miteinander verbunden und können Daten austauschen. Eingangsports werden mittels dual-clocked FIFOs entkoppelt und erlauben den Prozessoren den Betrieb mit unterschiedlichen Taktfrequenzen. Hierzu beinhaltet jeder Prozessor einen lokalen Oszillator, der entsprechend den Anforderungen der Applikationen angepasst oder auch vollständig abgeschaltet werden kann. Jeder Prozessor kann seinen Oszillator unabhängig steuern, so dass der komplette Chip global gesehen asynchron läuft und die Daten-Transfers zwischen den Prozessoren über FIFOs an den Eingängen synchronisiert werden.

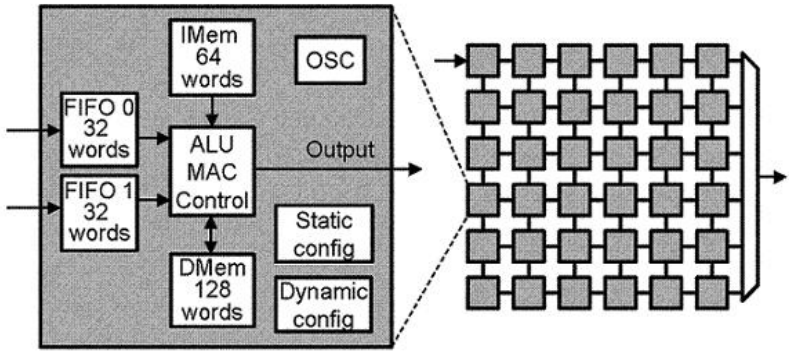



**Abbildung 40 Aufbau des Asynchronous Array of simple Processors (AsAP)**

Der vereinfachte Prozessorkern ist eine RISC-Architektur mit lediglich einer FFT-spezifischen und 53 applikationsunabhängigen Instruktionen. Er enthält eine 16bit ALU und eine 40bit MAC Einheit, die in einer neunstufigen Pipeline integriert sind. Diese Einheiten können allerdings nicht gleichzeitig bedient werden, so dass keine superskalare Architektur vorliegt. Die Pipeline ist einmal notwendig, um dem Prozessor Zeit für die Speicherzugriffe zu geben und zudem zur Erhöhung der maximalen Taktfrequenz, indem beispielsweise die MAC Einheiten über drei Stufen verteilt ist. Aus der Sicht des Prozessors erscheinen die Ein/Ausgangsports sowie der Datenspeicher wie gewöhnliche Register und können vom Benutzer transparent angesprochen werden.

Die erste ASIC Realisierung (AsAP1) [111] basierend auf dieser Architektur wurde im Jahre 2005 in 0,18µm Technologie hergestellt. Dabei wurden 36 Prozessoren in einem 6x6 Array auf dem Chip integriert und benötigte pro Kern eine Fläche von 0,66 mm². Bei einer Core-Spannung von 1,8V erreicht die Architektur bis zu 540MHz und verbraucht dabei im Durchschnitt 32mW pro Kern. Bei Steigerung der Spannung auf 2,0V waren 600 MHz möglich.

Während die erste Version noch einen relativ einfachen Aufbau hatte, wurde die zweite Version der Architektur [112] in mehreren Punkten weiterentwickelt. Zur Skalierung der Taktfrequenz kam die Skalierung der Spannung hinzu. Jeder Prozessorkern kann dabei durch die Auswahl der gewünschten Spannungsnetze die eigene Spannung auf eine der zwei möglichen Stufen ($V_{high}$ und $V_{low}$) einstellen. Das Spannungsnetz wurde zu diesem Zweck in einer Gitteranordnung mit vier $V_{DD}$ Spannungsleitungen und zwei GND Leitungen verschränkt realisiert. In speziellen Spannungszellen wird die Selektion der $V_{DD}$ Spannung vorgenommen, während die GND Leitungen fest mit ihren Verbrauchern verbunden sind. Die einfachen Prozessorkerne erhielten eine 6-Stufen-Pipeline und behielten ansonsten grundsätzlich den gleichen Aufbau wie die erste Version.

Eine weitere große Änderung erfuhr das Kommunikationsnetzwerk. Die lokalen Verbindungen zwischen den benachbarten Prozessoren wurden durch ein globales Kommunikationsnetzwerk ersetzt. Dazu erhielt jeder Prozessor eine Communication Matrix mit vier Duplex-Verbindungen zu jedem direkten Nachbarn. Jede Matrix kann die eingehenden Daten durch statische Konfiguration in Richtung der Zielzelle durchstellen. Die Datentransfers selbst werden durch ein Handshake-Protokoll gesichert. Prozessoren sind mit zwei Eingangsleitungen und acht Ausgangsleitungen an die lokalen Matrizen angeschlossen. Eingehende Daten werden wir bei AsAP1 mit Dual-Clocked-FIFOs synchronisiert. Der Nachteil dieses Verfahrens liegt in der Abhängigkeit des Datentransfers von der verwendeten Frequenz. Sind zwei Teilnehmer weit voneinander entfernt, so muss die Taktfrequenz des Senders gesenkt werden, um Signalen genügend Zeit zur Ausbreitung zu geben und das Protokoll einzuhalten. Messungen ergaben, dass benachbarte Zellen mit bis zu 1,3 GHz kommunizieren können, während ein Abstand von 11 Zellen bereits zu einer Höchstfrequenz von unter 600 MHz führt. Neben dem Datenaustausch zwischen zwei Teilnehmern besteht auch die Möglichkeit zur Realisierung eines Broadcasts. Dafür werden beim Sender die Request-Signale der Empfänger konjunktiv verknüpft, so dass erst gesendet werden kann, wenn alle Empfänger bereit sind. In diesem Fall aktiviert der Sender das Valid-Signal und legt zugleich die Daten auf den Bus.

Neben den allgemeinen Prozessorkernen wurden weiterhin programmierbare, aber in der Anwendung spezialisierte Prozessorkerne (Viterbi, FFT und

Motion Estimation) in das Array integriert. Daneben bekam das Array drei verteilte Speichermodule mit jeweils 16KB. Der AsAP2 wurde in 65nm CMOS Technologie hergestellt und lieferte beachtenswerte Ergebnisse. So erreichte der Chip eine maximale Frequenz von 1,2GHz bei einer Spannung von 1,2V und 62mW je Kern. Selbst bei einer Spannung von 0,675V konnte die Architektur 66MHz erreichen und verbrauchte dabei lediglich 608µW pro Prozessorkern, wobei 164 derartiger Prozessoren auf dem Chip Platz fanden. Der gesamte Flächenbedarf lag bei 39,4mm² bei einer Komplexität von 55 Millionen Transistoren. Der Durchsatz der Architektur liegt bei 196,8 GMACs pro Sekunde bei 1,2GHz.

## 3.6. Zusammenfassung

Die Vielfalt der Rechenarchitekturen ist in heutiger Zeit schier enorm und kaum überschaubar. Die Ansätze erstrecken sich über Prozessoren, spezialisierte DSPs, moderne GPUs, ASIC-Lösungen, fein- und grobgranulare Architekturen. Im Falle der letzteren ist die Auswahl sehr beschränkt, da die Programmierbarkeit bzw. die Anwendbarkeit noch nicht den Stand der Technik erreicht hat, der den heutigen Ansprüchen genügen würde. Im Übrigen haben die Entwickler die Qual der Wahl und müssen genau wissen, welche Anforderungen sie an ihre Systeme haben, um die passenden Komponenten angemessen auszuwählen.

Prozessoren besitzen bei weitem die größte Flexibilität, müssen jedoch diesen Vorteil durch den Nachteil der relativ niedrigen Performance erkaufen. Die Flexibilität wird in diesem Fall durch die Tatsache erreicht, dass durch den direkten Speicherzugriff sowohl auf Daten als auch Instruktionen beliebig zugegriffen werden kann. Diese Tatsache ist jedoch auch gleichzeig der Nachteil des Ansatzes, da die Bandbreitenlimitierung des Speichers die Leistungsfähigkeit begrenzt. Auf der anderen Seite steht eine Reihe von Entwicklungswerkzeugen bereit, die den Entwickler bei seiner Arbeit unterstützen und die Software-Entwicklung vereinfachen. Insbesondere sind die Compiler durch die jahrzehntelange Forschung und Entwicklung ausgereift und sind im Stande die Prozessoren sehr effektiv auszunutzen.

DSPs sind dem Grunde nach sehr prozessorähnlich, besitzen allerdings hochspezialisierte Datenpfade. Der Fokus der Optimierung der DSPs liegt vor allem auf einer Teilmenge von Anwendungen, die für die digitale Signalverarbeitung von Bedeutung sind. Dazu gehören Algorithmen wie FFT, FIR, DCT, Matrix-Operationen und ähnliche. Die parallelen Fähigkeiten der DSPs rühren vom VLIW-Ansatz und bedürfen spezieller Beachtung bei der Programmierung. Die Analyse und Extraktion von parallelen Operationen direkt aus einer Hochsprache wie C ist aber schwierig. Es ist daher auch gegenwärtig noch

üblich hochoptimierten Code für DSPs in Maschinensprache zu formulieren, was jedoch mehr Zeit in Anspruch nimmt.

GPUs sind relativ neue Teilnehmer im Umfeld der programmierbaren Rechenarchitekturen. Ihr eigentliches Einsatzgebiet ist die graphische Datenverarbeitung, sie sind aber auch dank der mittlerweile großen Flexibilität für High-Performance-Computing sehr gut geeignet. Sie bieten Fließkommaoperationen einfacher und doppelter Genauigkeit. Die große Zahl der vorhandenen Recheneinheiten heben die Leistungsspitzen dieser Architekturen über die Grenze von einem TFLOP/s. Die Programmierbarkeit gestaltet sich jedoch nicht so einfach wie im Falle der Prozessoren, da spezielle Techniken wie die CUDA genutzt werden müssen. Die Mehrleistung kann nur erreicht werden, wenn kein sequentieller Code ausgeführt wird und stattdessen möglichst viele Threads in Form von kurzen Schleifen genutzt werden.

ASICs sind sehr effizient bzgl. Performance, Verlustleistung und Fläche. Sie sind von Natur aus nicht programmierbar, da die eigentliche Zielanwendung bereits in der Struktur der mikroelektronischen Schaltung implementiert ist. Das Resultat bietet keinerlei Flexibilität, es sei denn, der Entwickler hat zum Entwicklungszeitpunkt gewünschte Einstellmöglichkeiten vorgesehen. Ein weiterer wichtiger Aspekt der ASICs ist der Kostenpunkt, der bei der Fertigung von einzelnen Modulen einen sechs bis sieben stelligen Stückpreis in US$ erreichen kann. Dies liegt vor allem an den notwendigen Masken zur Belichtung in der Fertigung, die teuer in der Herstellung sind. Einmal erzeugte Masken können jedoch in der Massenfertigung erneut verwendet werden und so den Stückpreis bis hin in den Cent-Bereich drücken. Fehler in der Schaltung oder Änderungen der Anwendungen zwingen den Hersteller jedoch zurück zum Reißbrett und bedürfen des erneuten Durchlaufs des kompletten Design Flows einschließlich der Fertigung, die alleine 6-8 Wochen in Anspruch nehmen kann.

Feingranulare Architekturen wie FPGAs arbeiten auf Bit-Ebene indem logische Funktionen mittels LUTs und Registern hochparallel realisiert werden. Zu diesem Zweck muss auf einem FPGA jedes Signal separat den LUTs zugeführt werden, was mittels eines umfangreichen konfigurierbaren Netzwerks geschieht. Dieser Vorgang ist hochkomplex und bedarf exakter Kenntnis der Bausteineigenschaften. Da jedes FPGA-Modell unterschiedliche Eigenschaften bzgl. Signallaufzeiten, Routing-Ressourcen, LUT-Aufbau und Register-Verfügbarkeit besitzt, ist die Entwicklung der Applikationen eine sehr spezifische Aufgabe. Auch wenn die Entwicklungswerkzeuge einen Teil der physikalischen Eigenschaften der Bausteine vor dem Anwender verbergen, muss dieser dennoch Kenntnisse der Bausteineigenschaften haben. Die feingranulare Arbeitsweise der FPGAs hat den Nachteil einer großen Konfigurationsdatenmenge, wodurch in der Anwendung auf häufige Rekonfiguration verzichtet

werden muss. Bei kleinen Stückzahlen stellen FPGAs eine Alternative zu ASICs dar, da sie in diesem Fall mit Abstand günstiger sind. Ursprüngliche Anwendungen der FPGAs zum Prototyping oder als Interconnect-Bausteine auf PCBs sind der allgemeinen Anwendung als leistungsfähige Rechenbausteine gewichen. Unterstützt wird diese Entwicklung durch die Integration dedizierter Einheiten für arithmetische Funktonen oder lokale Speicher durch die Hersteller.

Grobgranulare Architekturen besitzen einen geringeren Konfigurationsoverhead. Dies ist dem Umstand zu verdanken, dass das vektorbasierte Routing und Funktionseinheiten vergleichsweise wenige Konfigurationsdaten zur Beschreibung ihrer Funktionen benötigen. Dies eröffnet auch die Möglichkeit, die dynamische Rekonfiguration zu nutzen, wobei in wenigen Mikrosekunden Konfigurationen gewechselt werden. Gleichzeitig ist die erreichbare funktionale Dichte den FPGAs überlegen. Eine große Herausforderung für grobgranulare Architekturen stellt die Programmierbarkeit dar. Insbesondere bei arraybasierten Ansätzen ist durch die eingeschränkte Speicherzugriffsmöglichkeit die Anwendung der Hochsprache C nur eingeschränkt möglich. Die Leistungsfähigkeit grobgranularer Architekturen ist zwischen den FPGAs und ASICs angesiedelt. In diesem Kapitel wurden fünf Architekturen im Detail vorgestellt, die im Folgenden zusammenfassend rekapituliert werden.

Die ACM wurde als Template für heterogene SoCs entwickelt, mit dem Ziel die Anwendungen sowohl in die Fläche als auch in die Zeit abzubilden. Die fraktale Hierarchie der ACM wird der Struktur der Anwendungen gerecht und kann zusätzlich durch dedizierte Funktionen angepasst werden. Der hierarchische Ansatz aufbauend auf MIN-Netzen stellt zugleich auch eine Schwäche dar und skaliert mit der Anzahl der Knoten nur unzureichend. Die Programmierung mittels SilverC™ bedarf manueller Eingriffe durch den Programmierer und dies stellt eine zusätzliche Hürde in der Anwendung dar.

Die XPP-III Architektur ist für 16-bit Anwendungen optimiert und besitzt ein XPP-Array für reguläre und FNC-Prozessoren für irreguläre Programmteile. Das Kommunikationsnetzwerk ist durch ein Protokoll selbstsynchronisierend, was auch die Anbindung der Architektur an das System vereinfacht. Der heterogene Aufbau der Architektur macht ihre Anwendung schwierig. Dies äußert sich in der Notwendigkeit der manuellen Partitionierung der Anwendung auf das Array als auch die FNCs.

Die DRP-Architektur wurde als array-basierter effizienter Beschleuniger für SoCs wie auch als Standalone-Lösung entwickelt. Der Gedanke hinter DRP propagiert die Idee, dass mehrere Konfigurationen oder Kontexte zugleich im Array existieren und beliebig von Takt zu Takt bedingt oder unbedingt umgeschaltet werden können. Die Programmierung der DRP erfolgt in der Hochsprache C, ist jedoch bzgl. der Zeigernutzung eingeschränkt. Durch die globale

Auswahl der Kontexte, muss die Ein- und Ausgabe der Architektur mit zusätzlichem Aufwand realisiert werden. Können Daten nicht sofort geliefert oder abgeholt werden, so muss auch die Berechnung im gesamten Array angehalten werden. Weiterhin ist die partielle Rekonfiguration innerhalb eines Tiles praktisch nicht möglich.

Die MONTIUM-Architektur bietet einen Prozessorkern, der in seiner Funktionsweise einem VLIW-Prozessor ähnelt. Anstelle von Instruktionen verarbeitet der Kern aber Konfigurationssequenzen, die zuvor in den Prozessorkern geladen werden. Mit einer rekonfigurierbaren Schaltstruktur, die zehn interne Speicher und fünf ALUs miteinander verbindet, ist der Prozessor sehr effizient. Neben der Unterstützung von C-Operatoren können auch applikationsabhängige Funktionen implementiert werden. Durch Mehrfachinstanzen lassen sich auch mehrere MONTIUM-Prozessoren in einem SoC unterbringen. Dennoch lassen sich in diesem Fall die Kerne nicht nutzen, um beispielsweise eine FFT mit hoher Punkt-Zahl zu beschleunigen. Der limitierende Faktor ist hier die Anzahl der ALUs pro Kern und der verfügbare Speicher innerhalb desselben.

Die AsAP-Architektur ist im eigentlichen Sinne keine rekonfigurierbare Architektur. Das AsAP-Array besteht aus einer Vielzahl einfacher Prozessoren, die lokale Speicher für die Instruktionen und Daten besitzen. Die Prozessoren führen lokal diese Instruktionen aus und tauschen ihre Daten über die lokalen Verbindungen mit den Nachbarn. Von Bedeutung dabei ist die Tatsache, dass die Daten asynchron zwischen den Prozessoren getauscht werden. Jeder Prozessor kann somit mit der absolut minimalen Arbeitsfrequenz betrieben werden. Je nach Arbeitsfrequenz, kann auch die Versorgungsspannung der Prozessoren individuell angepasst werden. Diese Techniken senken die Verlustleistung für jeden Prozessor auf ein Mindestmaß. Die Programmierbarkeit dieser Architektur hat mit ähnlichen Problemen zu kämpfen wie moderne Multiprozessor-Systeme. Es ist weiterhin nicht möglich transparent ein C-Programm zu formulieren, dass die große Anzahl an Prozessoren effizient ausnutzt.

# 4. HoneyComb-Architektur

Bei näherer Betrachtung der vorgestellten Architekturen im letzten Kapitel, wird dem Leser aufgefallen sein, dass trotz der Vielfallt der interessanten Ansätze im Bereich grobgranular-rekonfigurierbarer Architekturen, viele wünschenswerte Eigenschaften nicht implementiert wurden. Dazu gehören insbesondere adaptive Features, welche die Nutzbarkeit der Architekturen stark verbessern. So sind beispielsweise die meisten Architekturen nicht in der Lage parallel mehrere Anwendungen gleichzeitig auszuführen. Ein Versuch, dies zu bewerkstelligen, endet im Ressourcenkonflikt im Routing-Bereich oder bzgl. der Funktionseinheiten in den Arrays, insbesondere wenn die Anwendungen unabhängig voneinander entwickelt und kompiliert wurden. Denn trotz der Tatsache, dass es sich hier um dynamisch rekonfigurierbare Architekturen handelt, sind die Abläufe in den Architekturen statischen Mustern unterworfen, die im Voraus berechnet werden. Eine Ausnahme stellt hier die ACM Architektur dar, die durch ihre Multikontext-Fähigkeit auf Anwendungsebene mehrere Anwendungen parallel ausführen kann. Im Falle der FPGAs ist ein zusätzlicher Hardware-Layer notwendig [72], um diese Flexibilität zu bieten, wodurch wertvolle Ressourcen auf dem Chip aufgebraucht werden. Die große Menge vorhandener Ressourcen moderner Chips bietet eine Menge Möglichkeiten neue Funktionen direkt in Silizium zu implementieren, die dynamische Abläufe in rekonfigurierbaren Architekturen erlauben. Tabelle 10 stellt eine Übersicht der gewünschten Features dar, die notwendig sind, um dynamische Abläufe in Hardware zu ermöglichen:

**Tabelle 10 Liste der Anforderungen an eine neue Architektur**

| Nr | Eigenschaft |
|---|---|
| 1 | Flexible Konfigurationsstrukturen zur Laufzeit |
| 2 | Verschiebbare Konfigurationen zur Laufzeit |
| 3 | Konfliktfreie Überlagerung/Verschränkung von Konfigurationen |
| 4 | Kurze Re-/Konfigurationszeiten |
| 5 | Programmierbare Ein/Ausgabe Module/Funktionen |
| 6 | Verbesserte Erreichbarkeit innerhalb des Arrays |
| 7 | Unterstützung von Energiesparfunktionen im Array |
| 8 | Flexible Datentypen |
| 9 | RTL-Anpassungsfähigkeit an die Applikation |

Die Liste der gewünschten Eigenschaften entstand aus den Erfahrungen durch die Arbeit mit rekonfigurierbaren Architekturen, insbesondere mit

FPGAs, PACT XPP und die Beteiligung an der Entwicklung der DReAM Architektur während meiner Zeit als Diplomand in Darmstadt. Durch zusätzliche Analysen der Publikationen weiterer Architekturen, wie im letzten Kapitel vorgestellt, entstand der Wunsch nach mehr Dynamik und Flexibilität bei dynamisch rekonfigurierbaren Architekturen.

Der erste Punkt in der Tabelle fällt beim Einsatz von FPGAs sehr schnell auf, da FPGA-Konfigurationen in ihrer strukturellen Ausprägung ein statisches Muster bilden. Eine Veränderung dieser Struktur nach der Synthese ist insbesondere bei FPGAs kaum zu bewerkstelligen, da hierfür ein erneuter Synthesedurchlauf notwendig wäre. Bei grobgranularen Architekturen lässt sich dieser Gedanke eher aufgreifen, da der zellbasierte Aufbau eine gute Granularität hierfür bietet. Allerdings werden bei aktuellen Architekturen die ebenfalls statischen Strukturen in die Konfigurationen eingeprägt, die zur Laufzeit nicht veränderbar sind. Das Problem ist deshalb von Interesse, da bei Ausführung mehrerer Konfigurationen auf einem Array durch partielle Rekonfiguration nicht sichergestellt werden kann, dass die zu konfigurierenden Anwendungen passende strukturelle Ausprägung besitzen, um gleichzeitig und ohne Überschneidungen auf der Architektur Platz zu finden.

Der zweite Punkt bezieht sich auf die Tatsache, dass Konfigurationen fest an ihre Positionen gebunden sind. Die Erfahrungen mit XPP zeigen, dass diese Funktion grundsätzlich implementiert werden kann, jedoch durch das Format des Konfigurationsdatenstroms zu viele Änderungen an diesen Daten vorausgesetzt. Abgesehen davon, muss die neue Position der Konfiguration im Array identische Strukturen bieten, damit die Kompatibilität gewahrt bleibt. Dieser Umstand schließt inhomogene Architekturen von vorherein aus bzw. schränkt ihre Anwendbarkeit deutlich ein. Dies gilt auch im Falle der XPP, da ihre Speicherzellen nur an äußeren Rändern zu finden sind.

Ein weiterer interessanter Punkt ist die Überlagerung bzw. Verschränkung von Konfigurationen. Bei Verwendung partieller Rekonfiguration beim Einsatz mehrerer Anwendung kann es im Array zur Segmentierung kommen. Dieses Problem lässt sich zum einen lösen, indem von Zeit zu Zeit das Array leer gemacht wird und die Ausführung erneut startet. Eine Möglichkeit wäre hier, eine Hardware-Unterstützung anzubieten, um die neuen Positionen von einzelnen Teilen der Konfiguration mit dem Rest der Applikation zu verbinden. Dies wäre ein neues Level der Flexibilität in diesem Kontext.

Der vierte Punkt betrifft vor allem FPGAs, da hier moderne Versionen bis zu 50MBytes für eine Konfiguration benötigen. Die Gründe hierfür sind zunächst in der Natur der feingranularen Bausteine zu suchen, die deutlich mehr Daten für die Beschreibung der Applikationen brauchen. Das Problem dabei ist allerdings, dass durch diesen Umstand gleichzeitig die Möglichkeit der schnellen Rekonfiguration genommen wird und Techniken wie Configuration Se-

quencing nicht sinnvoll angewendet werden können. Dies ist deshalb wünschenswert. weil die verfügbaren Ressourcen auf einem rekonfigurierbaren Array limitiert sind und eine zeitliche Aufteilung des Array zwischen Applikationen eine wirtschaftlichere Nutzung ermöglichen würde.

Üblicherweise beschränken sich die I/O-Module bei rekonfigurierbaren Architekturen auf einen Satz vordefinierter Funktionen, wie DMA oder Blocktransfers. Zur sinnvollen Nutzung sind hier programmierbare Schnittstellen zweckmäßig, die basierend auf Ereignissen aus dem Array oder dem Host-System programmierte Aktionen ausführen können.

Punkt sechs adressiert die etablierten Array-Strukturen heutiger Architekturen. Zumeist sind die Ausprägungen heute vierkantig und besitzen damit eine vertikale und eine horizontale Verbindungsrichtung zu ihren Nachbarn. Dies hat einen großen Nachteil, insbesondere wenn diagonale Verbindungen gewünscht sind. In diesem Fall ist die Anzahl der Hops vom Start zum Ziel bis zu 40% länger als eine direkte diagonale Verbindung und erhöht die Latenz der Kommunikation. Zusätzlich werden auch im selben Maße mehr Ressourcen gebraucht, um die Verbindung zu realisieren.

Der nächste Punkt betrifft Stromspartechniken, die sich bei zellbasierten Architekturen anbieten, indem nicht aktive Zellen beispielsweise durch Clock-Gating oder gar Voltage-Gating deaktiviert werden. Das Clock-Gating ist eine relativ einfache Technik und lässt sich auch bei kleinen Zellen kostengünstig anwenden. Das Voltage-Gating hingegen benötigt eine Mindestfläche auf dem Silizium, da die Spannungsversorgung zwischen unabhängigen Gebieten aufwendig entkoppelt werden muss, so dass sich diese Technik erst ab einer gewissen Größe der Zellen rechnet.

Punkt acht betrifft die Flexibilität der Architekturen hinsichtlich ihrer Granularität. FPGAs sind in dieser Hinsicht sehr flexibel und lassen die Abbildung unterschiedlicher Granularitäten auf ihre feingranularen Strukturen zu. Grobgranulare Architekturen hingegen sind meist auf einen Datentypen festgelegt und geraten bei abweichenden Anwendungen ins Hintertreffen. Gewünscht wäre hier ein Mindestmaß an Flexibilität, um eine Wahl bei der Abbildung von Anwendungen zu haben.

Trotz der Tatsache, dass das Hauptthema dieser Ausarbeitung rekonfigurierbare Systeme betrifft, ist eine Anwendungsabhängigkeit erstrebenswert und notwendig. Flexibilität kostet nun mal zusätzliche Ressourcen, die im Worst-Case nie eingesetzt werden. Beschränkt sich die Anwendung einer Architektur auf einen Satz von Anwendungen, so liegt der Wunsch nach einer Kostensenkung sehr nahe. Dies kann erfolgen, indem auf RTL die nicht verwendeten Ressourcen entfernt werden. Das Resultat wäre ein Architektur-Template, das sich an die Gegebenheiten ihrer Anwendungen anpassen lässt und somit Kosten spart.

## 4.1. Einleitung

Aus der Idee heraus, neue adaptive Features und innovative Ansätze zu nutzen, entstand die Idee eine neue adaptive, dynamisch rekonfigurierbare Architektur, die HoneyComb-Architektur, im Rahmen eines Forschungsvorhabens zu entwickeln. Dieses Forschungsvorhaben zielte auf die Entwicklung, Implementierung und Synthese einer neuartigen dynamisch rekonfigurierbaren und multigranularen Rechenarchitektur. Weiterhin sollte die Problematik der Adaptierbarkeit moderner Systeme während der Laufzeit adressiert werden. Ein Schwerpunkt dieses Vorhabens lag im dynamischen Ablauf von Zielapplikationen mit Operationen unterschiedlicher Granularitäten, wie sie in den meisten Algorithmen mit unterschiedlicher Gewichtung vorkommen. Hierbei sollte eine möglichst flexible und effiziente Realisierungsmethodik untersucht und implementiert werden, die im Gegensatz zur Mehrheit bisheriger Ansätze, Operationen dynamisch auswählen kann. Das führt zu Flächenersparnissen und ermöglicht zudem ein effizienteres Einsetzen der Architektur, indem weitere Applikationen parallel zur gleichen Zeit ablaufen können. Darüber hinaus sollen die physikalischen Kommunikationsverbindungen auf der geplanten multigranularen Array-Architektur erst zur Laufzeit festgelegt und allokiert werden, so dass zum Zeitpunkt des Entwurfs und der Kompilierung der Anwendung der genaue Kommunikationsverlauf der Funktionsmuster nicht bekannt sein muss. Die optimierten Kommunikationspfade sollen erst nach der Konfiguration adaptiv durch die Zielhardware realisiert werden. Während dynamischer Rekonfiguration werden demnach die Freigaben und Belegungen von Hardwareressourcen (inkl. Kommunikationspfade) hardwaretechnisch automatisiert durchgeführt, so dass keine statischen vorab generierten Konfigurationscodes und Szenarien mehr notwendig sind, sondern zur Laufzeit höchst dynamisch und hardwareunterstützt Ressourcen allokiert und wieder freigegeben werden können. Für exemplarisch ausgewählte Betriebssituationen sollten anwendungsspezifisch passende Architekturvarianten mit genügend Hardwareressourcen und Kommunikationsbandbreiten identifiziert werden.

Die Durchführung des Projekts unter dem Akronym AMURHA wurde durch die DFG im Rahmen des Schwerpunktprogramms 1148 bewilligt, finanziert und über die Dauer von 6 Jahren unterstützt. Am Ende des Projekts entstand ein Demonstrationschip, der die Funktionstüchtigkeit der HoneyComb-Architektur aufzeigte und die entwickelten Konzepte bestätigte. Die Herstellung des ASICs erfolgte in TSMC 90nm Standardzellentechnologie und hatte eine Fläche von insgesamt 16mm². Eine Reihe von Tools und Applikationen wurden parallel zur Hardware entwickelt, um die neue Architektur zu programmieren und ihre Funktionstüchtigkeit zu demonstrieren. Diese Entwicklung schloss sowohl Programmierwerkzeuge wie auch Debugging Tools ein.

### 4.1.1. Grundlagen der HoneyComb-Architektur

Für einen hohen Parallelitätsgrad innerhalb der HoneyComb-Architektur wurde ein Array-Ansatz gewählt, der Funktionseinheiten in großer Zahl in sich vereint. Zudem lässt sich durch ihre massiv-parallele Verarbeitung die Arbeitsfrequenz senken und damit die Verlustleistung in Abhängigkeit der Auslastung reduzieren. Abbildung 41 veranschaulicht die Struktur der HoneyComb-Architektur, wobei deutlich die hexagonale Struktur der Zellen erkennbar ist. Damit erhält das Array eine dritte Routing-Richtung und ermöglicht im Durchschnitt den Aufbau kürzerer Verbindungen. Weiterhin teilt sich jede Zelle in zwei Hauptbestandteile auf: Routing Unit (RU) und die Functional Unit (FU), siehe Abbildung 42.

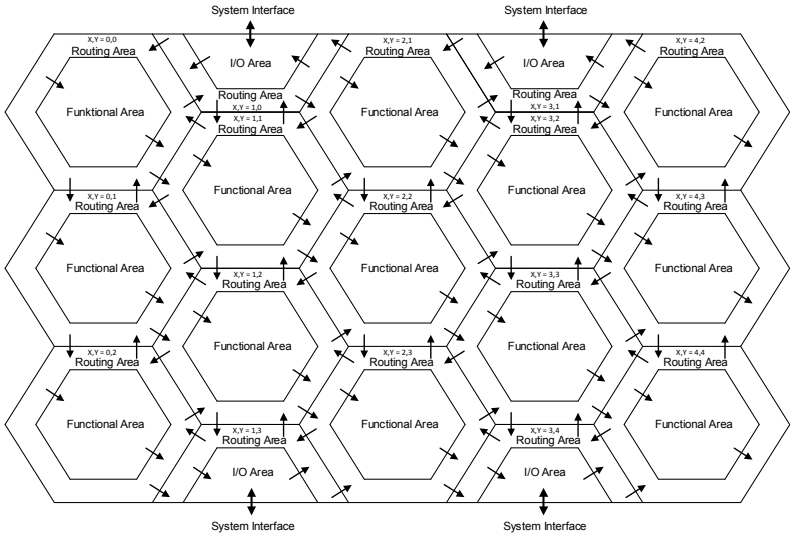



**Abbildung 41: Hexagonaler Aufbau der HoneyComb-Architektur mit koordinatenbasiertem Routing zur Laufzeit**

Das erstere befindet in der Abbildung 41 im Bereich der Routing Area, während das letztere im Bereich der Functional Area zu finden ist. Die FUs haben im Array spezifische Aufgaben und teilen sich in drei Klassen auf: Datapath-FU (DPFU), Memory-FU (MEMFU) und Input/Output-FU (IOFU). Die HoneyComb-Architektur unterstützt zwei grundlegende Datentypen: grobgranular und feingranular. Im ersten Fall werden 32-bit Vektoren genutzt, um Instruktionen, Konfigurationen und Daten zu übertragen und zu verarbeiten, im letzten werden 1-bit oder n-bit Signale transportiert, für boolesche Berechnungen herangezogen und mit der grobgranularen Domäne ausge-

tauscht. Bei der Übertragung zwischen den FUs mittels des Kommunikationsnetzwerks werden feingranulare Signale optional zu Bündeln mit mehreren Bits zusammengefasst und als eine Einheit konfiguriert und anschließend übertragen.

Die DPFUs führen die eigentlichen Kalkulationen aus, indem sie für die Architektur ALUs, LUTs, Register und Ein-/Ausgangsports realisieren. In ihrer Struktur bieten DPFUs ein lokales Netzwerk, welches sich in zwei Domänen aufteilen lässt: grobgranular und feingranular. Die grobgranulare Domäne ist dafür zuständig arithmetische Operationen auf unterschiedlichen Datentypen mittels ALUs auszuführen und in grobgranularen Registern zu speichern. Diese Werte können im nächsten Takt lokal in der DPFU verwendet werden oder an andere Zellen weitergeleitet werden. Unterstützte Datentypen sind hier vorzeichenbehaftete/vorzeichenlose Ganzzahlen, Festkomma- und Fließkommazahlen. Die feingranulare Domäne unterstützt boolesche Logik, die für logische Funktionen und Steueraufgaben für den Konfigurationsablauf in derselben DPFU oder auch außerhalb genutzt werden kann. Generierte feingranulare Signale können einzeln oder in Gruppen zur Operationsselektion bei ALUs genutzt werden, um auf diese Weise Kontextwechsel zu adressieren. ALUs ihrerseits können Status-Flags wie Überlauf oder Vorzeichenbits an die die feingranulare Domäne zur Steuerung von FSMs oder einfacher Logik verwenden. Außerdem lassen sich aus grobgranularen Vektoren einzelne Bits extrahieren und in die feingranulare Domäne überführen. Darüber hinaus lassen sich feingranulare Signale nutzen, um die Register für Datenpfade zu steuern, um Funktionen wie Löschen, Speichern oder Halten zu realisieren. Zur Speicherung der feingranularen Daten sind eigens vorgesehene Register vorhanden. Wie bei allen FUs besitzen die grobgranularen wie auch die feingranularen Domänen eigene Ein- und Ausgangsports, die sich nach ihrer Granularität unterteilen lassen. Diese Ports erlauben den Zugang zu betreffenden FUs. Zusätzlich ist ein weiterer dedizierter Port für den Transfer von Konfigurationsdaten vorhanden, der für die Programmierung der FUs genutzt wird, indem Daten vom Host-System zur diesem Port übertragen werden. Zellen, welche die DPFUs enthalten, werden Datapath HoneyComb Cells (DPHCs) genannt.

Die MEMFUs haben die primäre Aufgabe, Datenspeicher im Array anzubieten. Hierfür werden ein oder mehrere Speichermodule in die betreffende MEMFU integriert und zusätzlich ein Speicher-Controller zur Steuerung der Datentransfers zwischen den Anwendungen und dem Speichermodul eingesetzt. Die Speicherblöcke arbeiten grundsätzlich grobgranular. Sollen feingranulare Daten gespeichert werden, so werden ein oder mehrere Bits zu einem 32-bit Vektor zusammengesetzt und in dieser Form gespeichert. Invers funktioniert der Vorgang ähnlich und erlaubt die Extraktion der ursprünglichen Daten. Der Speichercontroller bietet die Möglichkeit eines normalen RAM-

Zugriffs auf den lokalen Speicher, indem die Applikationen Daten und die Adressen für den Zugriff liefern. Zusätzlich lässt sich der Speicher als FIFO (First-In-First-Out) oder LIFO (Last-In-First-Out) konfigurieren. Weiterhin lassen sich mehrere Speichermodule zur Erhöhung der Kapazität durch die Speichercontroller transparent koppeln und somit den Anwendungen größere RAM-Module oder FIFOs suggerieren. Zellen, welche die MEMFUs enthalten, werden als Memory HoneyComb Cells (MEMHCs) bezeichnet.

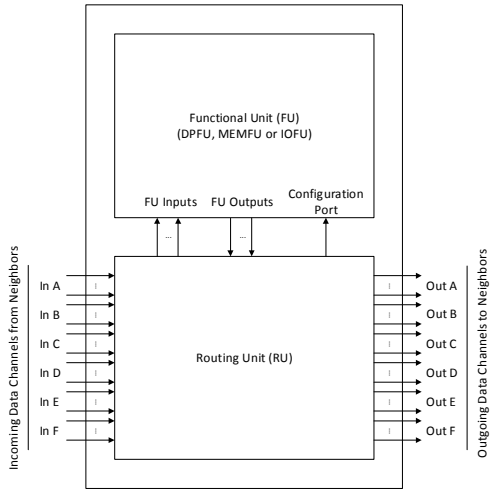



**Abbildung 42: Vereinfachte Darstellung der universellen HoneyComb-Zellen**

Die IOFUs bieten dem Array die Möglichkeit, die Anbindung an das Host-System zu realisieren. Hierzu besitzen sie auf der externen Seite ein generisches Interface, das zur Anbindung an das Host-System via AMBA AHB/AXI, WishBone oder ähnlichem On-Chip-Bus genutzt werden kann. Das Innere der IOFU bildet ein einfacher Controller basierend auf einer 5-stufigen Pipeline-Architektur, die neben Berechnungen mit 32-bit Integer-Datentypen insbesondere den Transfer der Daten zwischen dem Array und dem Host-System bewerkstelligt. Zu diesem Zweck stehen dem Controller-Kern weitere dedizierte Blocktransfermodule zur Verfügung, die parallel und unabhängig voneinander Daten in verschiedenen Modi übertragen können. Dies geschieht wahlweise per DMA im Host-System oder durch einfache Datenübertragung an den Host-Controller (Master-/Slave-Konzept). Die Datenübermittlung von IOFUs zum Array wird mittels Dual-Clock-FIFOs realisiert und somit die Arbeitsfrequenzen des Arrays vom Host-System entkoppelt. Damit lässt sich das Arbeitstempo der HoneyComb-Architektur unabhängig von Anforderungen der aktuell auszuführenden Anwendungen einstellen. Neben den grobgranularen Daten ist

die IOFU auch in der Lage feingranulare Daten in das Array und zurück zu transferieren, in dem zwei Packing-Modi beim Zugriff auf den Hauptspeicher angeboten werden. Zellen, welche die IOFUs enthalten, werden als Input/Output HoneyComb Cells (IOHCs) bezeichnet.

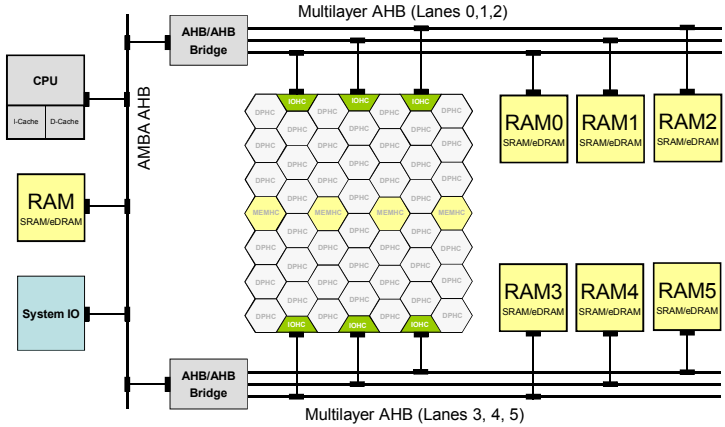



**Abbildung 43: Exemplarische Anbindung der HoneyComb-Architektur in das Host-System mittels breitbandiger Multilayer AHB Topologie**

Die Aufgaben der RUs, die in jeder Zelle zu finden sind und die Grundlage des globalen Kommunikationsnetzes bilden, bestehen darin, Punkt-zu-Punkt-Verbindungen zwischen den FU-Ausgängen und -Eingängen zweier Zellen zur Laufzeit zu etablieren. Zum einen sind hier alle RUs benachbarter Zellen miteinander verbunden, zum anderen geschieht dies mittels spezifizierter Instruktionen, die beim Eintritt in die Quell-Routing-Unit durch den Routing-Controller erkannt wird und den Routing-Prozess anstoßen. Zu diesem Zweck besitzt jede RU eindeutige Koordinaten, anhand derer sie die Routing-Richtung für den weiteren Verlauf der zu etablierenden Verbindungen bestimmen kann. Auf diese Weise kann jede Zelle den weiteren Verlauf der Verbindung ermitteln, bis die Zielzelle erreicht ist. Ist dies erfolgt, verbindet die Ziel-RU den eingehenden Pfad mit dem Eingangsport der FU. Während die Quell-RU den ausgehenden Pfad mit dem Ausgang ihrer FU verknüpft. Ist der Pfad zwischen den beiden FUs etabliert, können diese Daten ungestört über die entstandene Punkt-zu-Punkt-Verbindung austauschen. Der Datentransfer erfolgt hierbei mit voller Array-Arbeitsfrequenz, wobei bei jedem Passieren einer RU die Daten in einem Register optional gepuffert werden und auf diese Weise ein langer Pfad zwischen zwei FUs entkoppelt werden. Neben den üblichen Datentransfers zwischen den FUs, kennen RUs zusätzlich spezielle Kon-

figurationsverbindungen, die immer von einer IOFU zum dedizierten Konfigurationsport einer DPFU oder MEMFU geführt werden. Damit lassen sich Konfigurationsdaten zu dieser Zelle übertragen und neben der Programmierung der beiden FU-Typen Ausgangsverbindungen von dieser Zelle aus programmieren, indem Routing Instruktionen in den Konfigurationsdatenstrom eingebettet werden.

Neben den Routing-Aufgaben besitzen die RUs zur Realisierung des Power-Managements eine zusätzliche Logik. Im Falle der HoneyComb-Architektur beschränkt sich diese Funktion intern auf das unabhängige Clock-Gating der RUs und FUs. Beide Module können je nach Aktivität unabhängig voneinander abgeschaltet werden. Im Falle der RUs werden diese Module von Nachbarzellen aktiviert, sobald eine eingehende Verbindung erstellt wird. Daraufhin wird das Clock-Gating der betreffenden RU deaktiviert und die RU kann ihre Aufgaben übernehmen. Abschaltung der RUs erfolgt erst, wenn keine internen Verbindungen durch diese RU mehr bestehen. Ähnlich verhält es sich mit FUs. Bestehen keine Verbindungen zwischen den RUs und FUs, so werden die FUs abgeschaltet und erst im Falle einer eingehenden oder ausgehenden Verbindung (Daten oder Konfiguration) wieder aktiviert. Dieses Verhalten gilt nicht für IOFUs, da diese Module als Bindeglied zum System immer aktiv gehalten werden.

Zusätzlich zu den dargestellten Zellen in Abbildung 41 besitzt das Array einen globalen Controller. Dieses Modul besitzt einen einfachen Aufbau, indem eingehende und ausgehen Signale vom bzw. zum Array durch ein internes Register-File dem System über ein separates Interface zum Auslesen bzw. Schreiben angeboten werden. Hierbei handelt es sich primär um Statussignale der Zellen, Error Codes der RUs und zu Debug-Zwecken der aktuelle Status der IOFUs hinsichtlich der Pipelineaktivitäten, Status der Blocktransfermodule, etc. Damit lässt sich von außen ein rudimentärer Überblick über den Status des Array verschaffen und im Fehlerfall eine einfache Analyse aufbauen. Nach einem Reset lässt sich mittels des globalen Controllers die Ausführung in IOHCs anstoßen, indem die Startadresse des auszuführenden Programms mit einem Startsignal an die betreffende IOFU-Pipeline übertragen wird.
Die Anbindung der HoneyComb-Architektur an das Host-System kann wie in Abbildung 43 dargestellt erfolgen. In diesem Beispiel wurde ein größeres Array dargestellt, als es in Abbildung 41 der Fall ist. Idealerweise sollte ein derartiges System ein breitbandiges Bussystem aufweisen und beispielsweise durch eine Multilayer-AHB/AXI-Implementierung [121] ausgeführt sein. Durch die Realisierung des Hauptspeichers durch mehrere SRAM-Module und exklusiver Anbindung dieser Module an die separaten Bus-Layer ließe sich ein ungestörter Betrieb der IOHCs der HoneyComb-Architektur erreichen.

### 4.1.2. Prinzipieller Ablauf eines Programms

Detaillierte Erläuterungen für die Vorgänge innerhalb der HoneyComb-Architektur werden in den folgenden Abschnitten durch die Beschreibungen der technischen Realisierung erfolgen. Der aktuelle Abschnitt soll dem Leser zunächst eine Idee vermitteln, wie die Abläufe in dieser Architektur einzuordnen sind, um das Verständnis für die Details zu erleichtern. Beschrieben werden exemplarisch die Vorgänge, die nötig sind, um die HoneyComb-Architektur zu konfigurieren, die Anwendungen ablaufen zu lassen und wieder zu löschen. Beginnen wir nach einem Reset, da in diesem Fall keine Randbedingungen zu beachten sind.

Nach dem Reset der HoneyComb-Architektur sind alle Zellen inaktiv. DPHCs und MEMHCs sind durch aktiviertes Clock-Gating vom Clock-Baum abgetrennt. Die RUs in IOHCs sind ebenfalls inaktiv, während die IOFUs auf eine Anforderung vom globalen Controller warten. Durch die Übertragung der Startadresse an den globalen Controller durch das Host-System, werden vom globalen Controller ein Startsignal und die Startadresse des auszuführenden Programms an die gewählte IOFUs gesendet. Daraufhin nimmt die Pipeline ihre Arbeit auf und beginnt ab der Startadresse die Instruktionen in die Pipeline zu laden. Werden im Zuge der Verarbeitung Blockbefehle geladen und ausgeführt, so stößt die Pipeline Blocktransferoperationen mittels der dedizierten Blocktransfermodule an. Jedes dieser Module lädt zunächst Konfigurationsdaten aus dem Hauptspeicher und transferiert die Daten über die integrierten Dual-Clock-FIFOs in das Array. Sobald die FIFOs aktiviert werden, wird ein Signal an die RUs derselben Zelle gesendet und diese dadurch aktiviert. Eingehende Daten, allem voran Routing Instruktionen, sorgen dafür, dass zunächst Verbindungen von aktueller IOFU zu Zielzellen aufgebaut und mit dedizierten Konfigurationsports verbunden werden. Dieser Vorgang aktiviert alle durch das Clock-Gating inaktiven RUs auf dem Weg zum Ziel, einschließlich der Zielzellen. Daraufhin werden Konfigurationsdaten ins Array über diese Konfigurationsverbindungen übertragen und sorgen dafür, dass zum einen die Ziel-FUs konfiguriert werden und zum anderen ausgehende Verbindungen zu anderen FUs und IOFUs im Array aufgebaut werden. Letzteres wird gebraucht, um Ergebnisse der Berechnungen aus dem Array zu transportieren. Nach dem Konfigurationsprozess der Zellen werden die Konfigurationsverbindungen von der IOFU zu den Zielzellen wieder abgebaut. Zu diesem Zeitpunkt sind die internen Zellen im Array konfiguriert und alle Verbindungen zwischen den Zellen und zu IOFUs etabliert. Nun werden von der IOFU noch die letzten Routing-Instruktionen zur eigenen RU übertragen, damit die Verbindungen für die eigentlichen Datenübertragungen zu den Ziel-FUs aufgebaut werden. Nach diesem abschließenden Vorgang ist die Konfiguration beendet und die Anwendungsausführung kann gestartet werden.

Die Übertragung der Daten kann von den IOFUs einzeln oder als Block erfolgen, wobei letzteres deutlich effizienter geschieht, da keine zusätzlichen Wartezyklen anfallen, sobald der Vorgang gestartet wurde. Während und nach der Übertragung der Daten in das Array, überwacht die IOFU die Aktivitäten an den Ausgangsports und überträgt die ankommenden Daten bzw. Ergebnisse aus dem Array zum Host-System in den Hauptspeicher oder direkt zum Controller. Die eigentliche Aktion wird hier durch das Programm für die IOFU beschrieben. Sind beide Vorgänge abgeschlossen, das heißt Eingangs- und Ausgangsdaten sind übertragen, kann erneut eine Berechnung erfolgen oder die Konfiguration gelöscht werden. Das erstere kann ja nach Anwendung nahtlos erfolgen. Das letztere macht die Ressourcen für andere Anwendungen frei, indem zunächst Verbindungen von aktueller IOFU zu Ziel-FUs abgebaut werden. Danach werden Konfigurationsverbindungen zu Zellen dieser Anwendung aufgebaut und die Reset-Instruktionen an die FUs übermittelt. Dies löscht alle aktiven Konfigurationen in den FUs. Der vorletzte Schritt schließlich ist das Löschen aller Verbindungen von diesen FUs zu anderen FUs der Anwendung, einschließlich der IOFUs. Zuletzt müssen noch die aktiven Konfigurationsverbindungen von der IOFU gelöscht werden, so dass nach diesem Vorgang wieder alle Zellen nach und nach bei erreichen kompletter Inaktivität durch das Clock-Gating deaktiviert werden. Zum Schluss kann die IOFU durch die Ausführung einer Exit-Instruktion sich selbst in Leerlauf-Zustand versetzten und auf weitere Anfragen vom globalen Controller warten.

Währende des gesamten Ablaufs kann über die Zugriffe auf den globalen Controller der Status der aktiven IOHCs als auch der RUs der aktiven Zellen abgefragt werden. In Fehlersituationen stoppen die RUs und melden Fehler an den globalen Controller, der wiederum vom System abgefragt werden kann, um das Problem zu erkennen und ggf. einzugreifen.

### 4.1.3. Hierarchisches Programmiermodell

Aufbauend auf der Hardware-Hierarchie der HoneyComb-Architektur und dem grundsätzlichen Programmablauf lässt sich ein für den Programmierer sich darstellendes Programmiermodell definieren, das den Umgang und das Verständnis für diese Architektur erleichtert, siehe Abbildung 44. Es lassen sich an dieser Stelle drei Hierarchieebenen identifizieren: Transport Layer, Communication Layer und Configuration Layer. Weiterhin sind mehrere Abhängigkeiten der unteren Layer zu darüber liegenden erkennbar, die nunmehr erläutert werden.

Die erste Schicht des Programmiermodells stellt der Transport Layer dar. Er beinhaltet den Programmcode für die IOHCs bzw. IOFUs und besitzt den Zweck, Konfigurationen und Daten zwischen dem System und dem Array

auszutauschen. Darüber hinaus besteht die Möglichkeit auf Ereignisse aus dem Array oder dem System zu reagieren und den Programmablauf zu steuern. Diese Steuerung betrifft vor allem den Zeitpunkt der Programmausführung und hat somit einen zeitlichen Einfluss auf die darunter liegenden Schichten des Programmiermodells.

Die zweite Schicht bildet der Communication Layer. Im Array wird diese Schicht vor allem durch die RUs aller Zellen realisiert und steuert damit das Kommunikationsnetz, welches adaptives Routing zur Laufzeit bietet. Die letztgenannte Funktion wird durch vier definierte Routing-Instruktionen gesteuert und durch RUs ausgeführt, siehe Abschnitte 4.3.2 und 4.3.3. Das Routing sorgt dafür, dass sowohl Konfigurations- als auch Datenpfade im Kommunikationsnetz hardwaregestützt zur Laufzeit berechnet werden. Die Koordinaten in den Routing-Instruktionen legen in diesem Kontext das Ziel der Pfade fest und spezifizieren damit ebenfalls die Zielzelle der transportierten Konfigurationen und Daten.

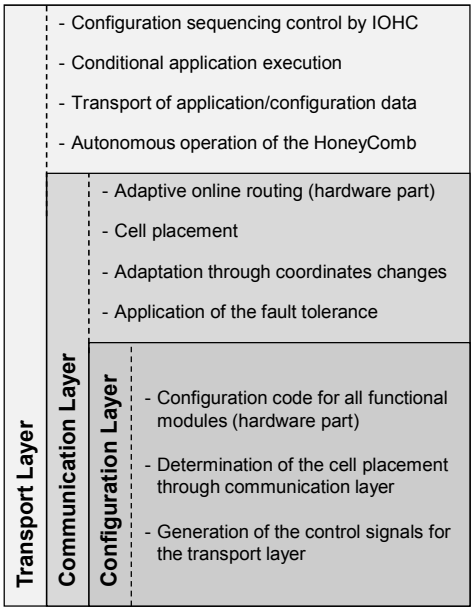



**Abbildung 44: Programmiermodell der HoneyComb-Architektur angelehnt an die Hardware-Hierarchie und ihrer Funktionen**

Der Configuration Layer stellt die dritte Schicht dar und beinhaltet die eigentlichen strukturellen Konfigurationsdaten für die FUs, die über den Konfigurationsport in die DPFUs und MEMFUs gelangen. Diese Daten sind in ih-

rem Aufbau fix und können zur Laufzeit nicht verändert werden. Die darüber liegenden Schichten bestimmt für den Configuration Layer wann und wo er ausgeführt werden soll, so dass der Code für die FUs in diesem Sinne neutral formuliert ist und damit auf allen Zellen mit identischen FU-Ausprägungen, wie zum Compile-Zeitpunkt definiert, ausgeführt werden kann.

Zur Laufzeit bedeutet dies, dass die Änderung der Koordinaten im Communication Layer zur Neuplatzierung der Zellen führt und beispielsweise defekte oder bereits belegte Zellen bei der Anwendung auslässt. Der Transport Layer kann bei mehreren Anwendungen auch das Scheduling übernehmen und damit den zeitlichen Ablauf steuern.

Programmierseitig besitzen alle drei Schichten einen unterschiedlichen Maschinencode. Der Transport Layer adressiert die IOFUs und wird in den integrierten Pipelines ausgeführt. Ihr Maschinencode ähnelt dem einer modernen 32-bit RISC-Maschine, besitzt jedoch einige spezielle Befehle zur Ansteuerung der Blocktransfer-Module. Der Communication Layer besitzt ebenfalls binäre Befehle und zwar vier 32-bit Routing Instruktionen. Diese werden verteilt von allen RUs im Kommunikationsnetzwerk ausgeführt und bilden die gesamte Funktionalität für den Communication Layer. Der Configuration Layer bietet eine Reihe von Instruktionen zur Beschreibung von FU-internen Modulen und des lokalen Netzwerks. Das Prinzip ist hier stark an die Struktur der verwendeten FUs angelehnt und kann nicht von strukturell abweichenden FUs verwendet werden. Zur programmierseitigen Unterscheidung der FU-Ausführungen werden MD5-Checksummen genutzt, um keine Verwechslungen zu erlauben. Diese Checksummen werden zum RTL-Zeitpunkt definiert und können später nicht geändert werden. Kompilierter Code für FUs kann später nur auf FUs mit ihm zugeordneter MD5-Checksumme zur Ausführung kommen.

### 4.1.4. Design-Entwurf und das VHDL-Modell

Das Design der Architektur erfolgte vollständig in VHDL nach dem Bottom-Up-Prinzip. Der Entwurf begann mit den Hardware-Strukturen und mündete in der aktuell höchsten Beschreibungssprache für die HoneyComb-Architektur, die HoneyComb-Language (HCL). Der Entwurfsprozess erstreckte sich über weitere Instanzen mit unterschiedlichsten Aufgabenstellungen, wie dem Low-Level-Assembler (HoneyComb-Assembler, HCA), Super-Configuration-Generator und HCViewer.

Um die Architektur möglichst flexibel zu halten und die Anpassungsfähigkeit des resultierenden Modells an Anwendungen zu gewährleisten, wurde zunächst eine Grundstruktur für alle Komponenten wie DPFUs, MEMFUs, IOFUs und RUs definiert. Mittels der Generics und Konstanten in VHDL

wurden diese Strukturen soweit wie möglich parametrisierbar realisiert, um bei Bedarf die notwendigen Ressourcen für die Hardwarerealisierung zu minimieren. Die Parametrisierbarkeit erstreckt sich über die Definition der Anzahl und Typ der Zellen, die Anzahl und Granularitäten der Datenports der FUs, Anzahl und Umfang der Operationen und Funktionen der ALUs, Anzahl und Größen der LUTs, Anzahl/Typ/Funktionen der Register, Ausprägung des Interconnects in den MEMFUs und DPFUs, Größen interner FIFOs in RUs und FUs, Multikontextfähigkeiten der ALUs und LUTs und weiterer Details, die in folgenden Abschnitten erläutert werden.

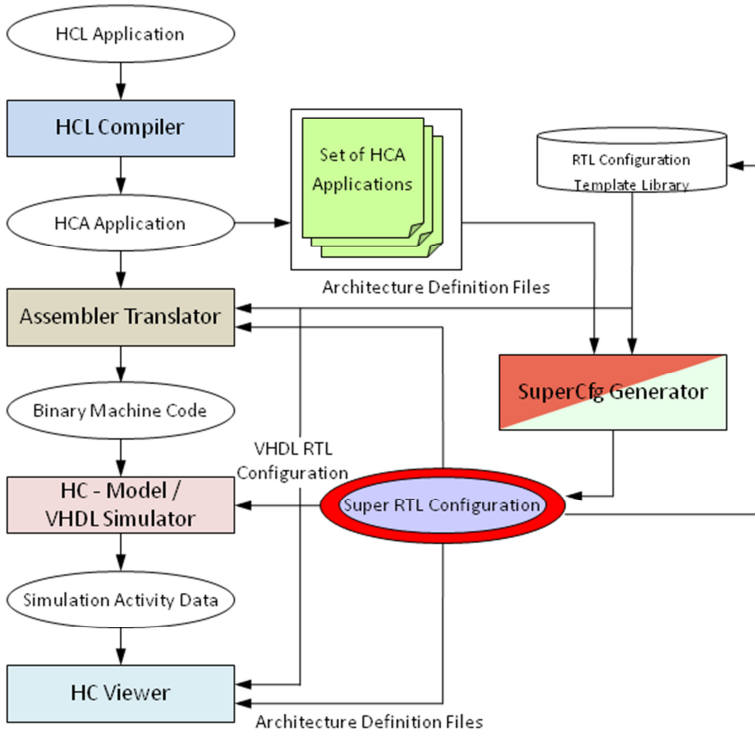



**Abbildung 45: Design-Flow für die Anwendungsentwicklung und optionaler RTL-Konfiguration-Generierung der HoneyComb-Architektur**

Die Idee war zunächst, ein RTL-Template zu schaffen, das sich an die Anwendungen anpassen lässt. Die resultierende Architektur besitzt eine derart hohe Vielzahl an möglichen Einstellungen, dass zusätzliche Tools entstanden sind, um die Generierung der finalen Architektur zu erleichtern. Im Falle der

DPHCs sind es bis zu 60.000 Parameter, die auf die finale Ausprägung der Architektur Auswirkungen haben. Manuelle RTL-Konfigurationen lassen sich dennoch leicht erzeugen, da dieser Prozess toolbasiert begleitet und die strukturelle Konsistenz überwacht wird. Zum besseren Verständnis der Architektur wird im Weiteren Bezug auf die ASIC-Realisierung der HoneyComb-Architektur genommen, um die zusätzliche Komplexität durch die Parametrisierbarkeit zu entschärfen.

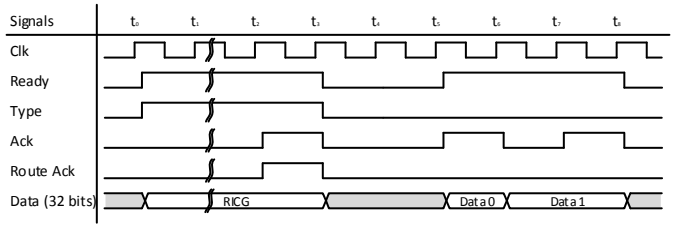



**Abbildung 46: Protokoll-Signale der grobgranularen CG-Links auf RU-Ebene zur Übertragung von Instruktionen/Konfigurationen und Daten**

Zur Programmierung der Architektur können derzeit zwei Programmiersprachen herangezogen werden HCL und HCA, siehe Abbildung 45. HCL ist eine an VHDL angelehnte Programmiersprache mit einem prozessbasierten Konzept, die zellweises Programmieren der Architektur auf einem vergleichbaren Level wie VHDL ermöglicht. HCA ist stattdessen eine reine Low-Level-Beschreibungsform, die deutlich fundierteres Wissen der Architektur voraussetzt. Der HCViewer kann während der Programmierung und der Simulation des Arrays herangezogen werden, um die internen Vorgänge zyklusgenau zu visualisieren. Hat der Anwendungsentwickler einen Satz an Anwendungen für seinen Anwendungsfall der HoneyComb-Architektur entwickelt, so lässt sich der Super-Configuration-Generator nutzen, um eine reduzierte Ausprägung der Architektur zu erzeugen. Dies erfolgt durch die Analyse der verwendeten Ressourcen wie Operationen der ALUs, LUTs, Speichermodule, Register, des Interconnects und ihre Ausprägungen. Das Resultat ist eine RTL-Konfiguration der HoneyComb-Architektur, die die notwendige Obermenge an Funktionen bieten, um alle Anwendungen ausführen zu können. Abbildung 45 veranschaulicht den Designflow, der sich dem Anwendungsentwickler derzeit bietet.

## 4.2. Universeller Zellaufbau

Wie bereits im Abschnitt 4.1.1 einleitend beschrieben, ist der Aufbau einer HoneyComb-Zelle einheitlich gestaltet, siehe Abbildung 49. Zum einen ist hier

die RU zu finden und zum anderen die FU. Die RU ist über mehrere Links mit den RUs der Nachbarzellen verbunden. Alle RUs ihrerseits sind mit den lokalen FUs verknüpft und bedienen auf diese Weise ihre Ein- und Ausgänge. Zur effizienteren Ausnutzung der Kommunikationsressourcen sind Links mit drei verschiedenen Granularitäten definiert. Die Darstellung ist generisch aufgebaut, so z.B. ist die Anzahl der Verbindungen parametrisierbar und diese Tatsache ist durch Pünktchen zwischen den Verbindungen angedeutet.

Zunächst einmal ist hier der grobgranulare (CG) Link zu finden. Hierbei handelt es sich um einen 32-bit Datenkanal, der primär die grobgranularen Daten übermittelt und sekundär die Konfigurationsdaten für die FU und Routing-Verbindungen transportiert. Abbildung 48 veranschaulicht die notwendigen Signale eines CG Links, siehe Signalbündel I. Wie alle Datentransfers innerhalb der HoneyComb-Architektur, sind auch diese Links durch ein Handshake-Protokoll gesichert. Auf diese Weise stellt die Architektur sicher, dass keine Daten verloren gehen, definiert, erzeugt und verbraucht werden. Abbildung 46 zeigt das genutzte Protokoll für die Datenübertragung auf diesen Links. Als primäre Signale für das Handshake-Protokoll sind hier 'Ready' und 'Ack' zu finden. Beide Signale sind an der Quelle und Senke gepuffert und die eigentliche Übernahme erfolgt, wenn beide Signale gleichzeitig auf high sind. Der Vektor 'Data' enthält in diesem Fall die zu transportierenden Daten oder Instruktionen, wie dies am Beispiel mit der Instruktion RICG (mehr später) angedeutet wird. Die Werte Data0 und Data1 repräsentieren gewöhnliche Datenvektoren, die im aufgezeigten Verlauf nach der erfolgreich ausgeführten Instruktion übertragen werden. Die übrigen Signale werden in den folgenden Abschnitten erläutert.

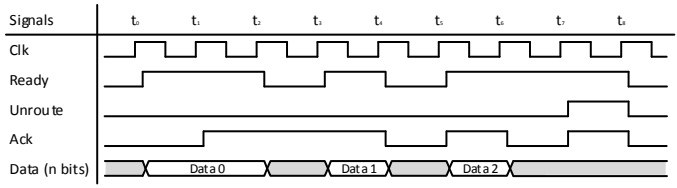



**Abbildung 47: Protokoll-Signale der multigranularen MG-Links auf RU-Ebene zur Übertragung von Daten und Unroute-Signalen**

Auf der anderen Seite existieren zwei unterschiedliche multigranulare Links (MG1, MG2), die zum Transport feingranularer Daten aus den FUs gedacht sind. Hierbei handelt es sich um zwei Links mit je einer vordefinierten Vektorbreite, die zum RTL-Zeitpunkt festgelegt sein müssen. Im Falle des ASICs wurde je eine 1-bit / 4-bit Aufteilung vorgenommen, um den vorliegenden Applikationen gerecht zu werden. Dies ist ein Erfahrungswert, der stark applikationsabhängig ist und am besten anhand vorliegender Anwendungen

bestimmt wird. Abbildung 48 zeigt mit dem Bündel II/III den Aufbau der MG1/MG2 Kanäle. Abbildung 47 veranschaulicht das genutzte Protokoll auf diesen Links. Das 'Unroute' Signal wird in diesem Fall genutzt, um die MG1/MG2 Ressourcen freizugeben. Mehr dazu in späteren Abschnitten.

Neben der RU und FU in den Zellen ist zusätzliche Logik für die Realisierung des Clock-Gating vorhanden. Hierfür finden sich hier 'Activity Input' Signale für je eine Nachbarzelle, die durch aktive Verbindungen in Richtung dieser Zelle diese aktivieren und das Clock-Gating ausschalten. Ebenso ausgehend von Signalen 'Routing Activity' innerhalb der RU wird zum einen das eigene Clock-Gating gesteuert und zum anderen Signale für die Nachbarzellen erzeugt. Für die Deaktivierung des Clock-Gating reicht es, wenn nur ein eingehendes oder internes Aktivitätssignal auf '1' gesetzt wird. Ist keins der Signale auf '1', so wird das Clock-Gating mit einer Verzögerung von zwei Takten wieder aktiviert. Die Verzögerung ist notwendig, um den internen FSMs innerhalb der RU genügend Zeit zu geben, um in einen definierten Zustand zu kommen. Das Clock-Gating für die FUs wird ebenfalls hier erzeugt. In diesem Fall hängt die Aktivierung von aktiven internen Ports der FU selbst ab.

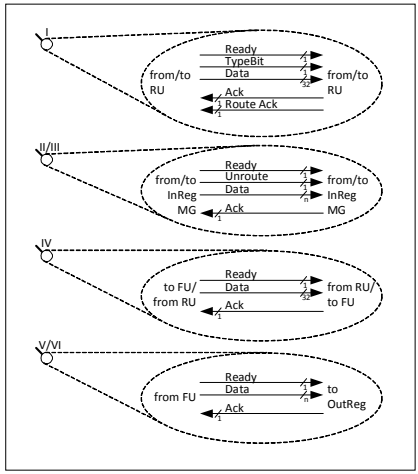



**Abbildung 48: Legende aller externen Signale auf RU Ebene**

Zusätzlich zur Steuerung des Clock-Gating besteht die Möglichkeit die Zellen von außen durch den globalen Controller zu deaktivieren. Hierfür dient das 'Enable' Signal, dass im aktiven Fall auf eine '1' gesetzt ist. Sollte im Fehlerfall die Zelle deaktiviert werden, sei es durch einen aufgetretenen Defekt in der Herstellung, Designfehler oder zu Testzwecken, so lässt sich das durch das Löschen des 'Enable' Signals bewerkstelligen.

Weiterhin bietet jede RU die Möglichkeit dem globalen Controller ihren Status mitzuteilen. Hierfür dienen die Signale 'Error' und 'ErrorCode'. Im Falles eines Fehlers wird hierfür 'Error' auf '1' gesetzt und 'ErrorCode' liefert einen der Situation entsprechenden Fehlerwert. Diese Daten lassen sich für jede Zelle separat aus dem globalen Controller auslesen und eine Fehleranalyse betreiben.

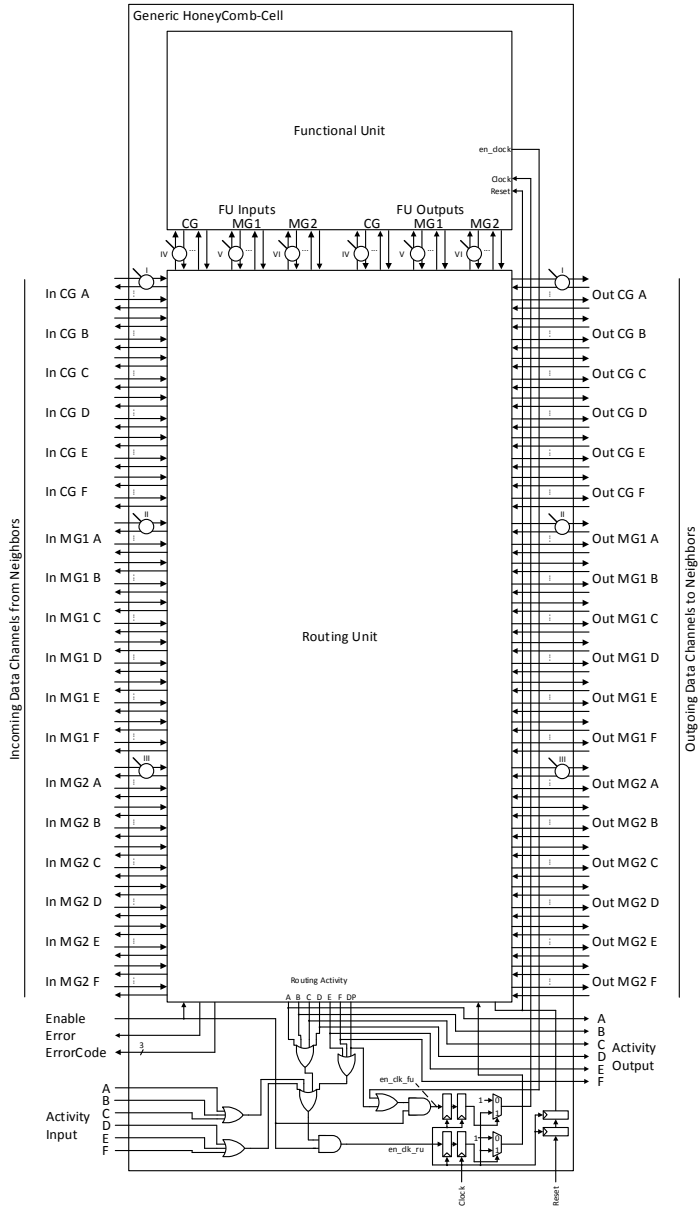



**Abbildung 49: Komplettes Blockdiagramm einer generischen HoneyComb-Zelle einschließlich Clock-Gating-Logik zur Realisierung des Power-Managements**

## 4.3. Routing Unit

Routing Units (RUs) sind die Hauptkomponenten zur Realisierung des globalen Kommunikationsnetzes. Praktisch alle Daten, wie Anwendungsdaten und Instruktionen/Konfigurationen, werden über dieses Netzwerk übertragen. Die Entscheidung für ein einheitliches Netzwerk wurde getroffen, um die große Bandbreite, die den Anwendungsdaten im Array zur Verfügung steht auch zur Beschleunigung der Konfigurationsübermittlung zu nutzen. Dies ist in diesem Kontext auch sinnvoll, da für das adaptive Routing zur Laufzeit, eine Art der Instruktionsübermittlung über dieses Netzwerk ohnehin notwendig war.

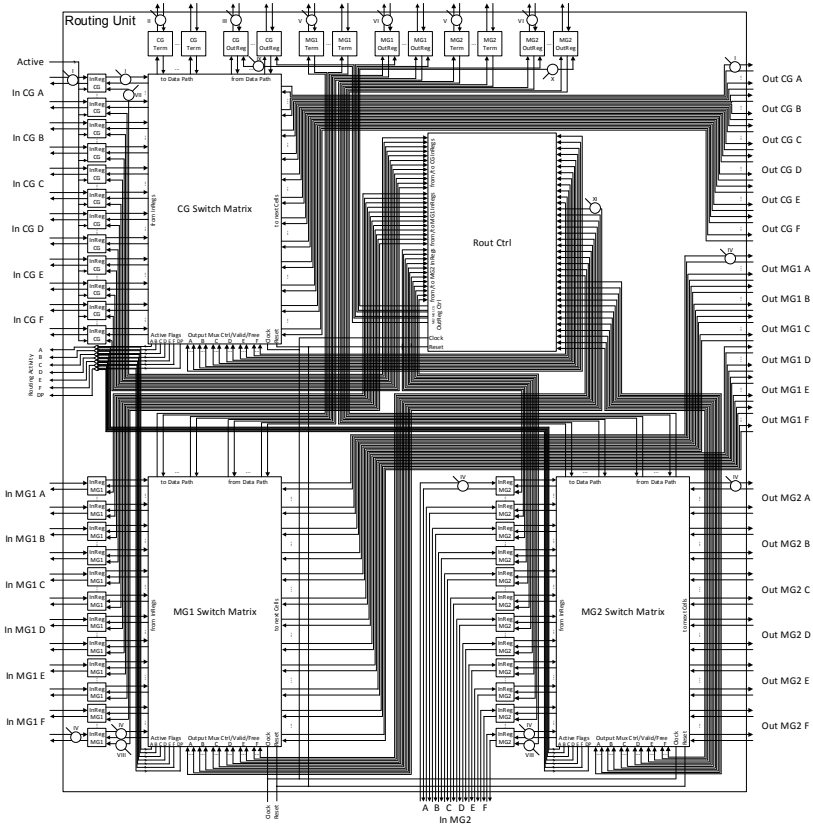



**Abbildung 50: Blockdiagramm des internen Aufbaus der Routing Unit**

Die RU enthält eine Reihe von Komponenten im Top-Level. Dazu gehören Input Register (InRegs) für CG und MG Links, CG und MG Output Register (OutRegs) für eingehende Links von FUs, Terminierungsmodule für ausgehende Links zur FU, eine CG und zwei MG Schaltmatrizen und der Routing Controller (RoutCtrl), siehe Abbildung 50. Die generische Darstellung verdeutlicht bereits die Komplexität der gesamten Struktur, die im Falle des gefertigten ASICs ca. 50% der Chip-Fläche vereinnahmte.

### 4.3.1. Funktionen

Die Funktionen der RU decken mehrere Bereiche ab: Auf- und Abbau von Routingverbindungen für Konfigurationen und Daten zur Laufzeit, den Daten- und Konfigurationstransport sowie Kommunikation des aktuellen Status zum globalen Controller. Der Auf- und Abbau von Verbindungen bezieht sich immer auf Punkt-zu-Punkt-Verbindungen zwischen ausgehenden und eingehenden FU Ports zwischen zwei unterschiedlichen Zellen. Die Steuerung des Routings erfolgt mittels spezifizierter Routing-Instruktionen, die zu Beginn des Routing-Prozesses zur Startzelle übertragen werden müssen. Zu diesem Zweck muss von einer aktiven IOHC ein Konfigurationspfad zu dieser Zelle aufgebaut werden, der in der Lage ist, Instruktionen zu transportieren. Wird eine Routing-Instruktion am eingehenden und in dieser Zelle terminierten, d.h. am dedizierte Konfigurations-Port der FU angeschlossenen, Konfigurationspfad detektiert, so schickt das betroffene CG InReg Modul die Anforderung an den RoutCtrl. Dieses Modul berechnet den möglichen Pfad zur Weiterleitung und speichert die Instruktion im OutReg Modul des zugehörigen Quellports der FU, der als Quelle für die neue Verbindung genutzt werden soll.

Routing-Instruktionen und Konfigurationsdaten werden nur über CG Links übertragen. Das Routing von MG Verbindungen kann nur unter Zuhilfenahme der CG Links erfolgen. In diesem Fall assistieren CG Link während des Aufbaus der MG Verbindungen und werden nach diesem Vorgang abgebaut, so dass die betreffenden CG-Ressourcen für andere Anwendungen frei werden.

Während der Aufbau von Verbindungen immer einen freien CG-Link voraussetzt und durch den RoutCtrl gesteuert wird, wird der Abbau von den CG und MG Links alleine durch die Quell-RU initiiert und danach selbstständig durchgeführt. Dazu muss zur Quellzelle eine Konfigurationsverbindung von IOHC kommend aufgebaut und eine Löschinstruktion zur RU transportiert werden. Hier erkennt das InReg die eingehende Instruktion und schickt eine Anforderung an den RoutCtrl, welcher umgehend die Anfrage an das betroffene OutReg weiterleitet. Dieses Modul nutzt die vorhandenen Signale in den jeweiligen CG und MG Links, um den Löschvorgang zu starten. Das genaue Protokoll wird in späteren Abschnitten erläutert.

Nachfolgend werden die Instruktionen und Techniken zum Routing im Detail erläutert.

### 4.3.2. Routing Algorithmus

Das grundsätzliche Vorgehen bei der Erstellung einer Verbindung in der HoneyComb-Architektur ähnelt dem Lee-Algorithmus [122] zum Finden eines Pfades zwischen Quelle und Ziel. Hierbei wird die Tiefensuche zur Analyse der Umgebung eingesetzt und sukzessive ein Weg zum Erreichen des Ziels ermittelt.

Die Vereinfachung im vorliegenden Fall liegt in der Tatsache begründet, dass ein homogenes Koordinatensystem bereits vorliegt, das einem die Bestimmung des kürzesten Pfades erlaubt. Mittels des Backtracking-Algorithmus ist die Architektur außerdem in der Lage, aus einer Sackgasse oder bei nichtvorhandener Möglichkeit zum Finden des optimalen Pfades zur vorherigen Zelle zurückzukehren. Zu diesem Zweck werden von jeder Zelle während des Routing-Prozesses bereits getestete Richtungen gespeichert und damit vor weiteren Versuchen ausgenommen. Dies garantiert allerdings nicht, dass bei erneutem Eintreffen des selben Routing-Prozesses an einer Zelle, alle Richtungen erneut geprüft werden, da beim Weiterleiten der Routing-Anfrage an eine bereits geprüfte Zelle alle Informationen diese Instruktion betreffend bereits gelöscht wurden.

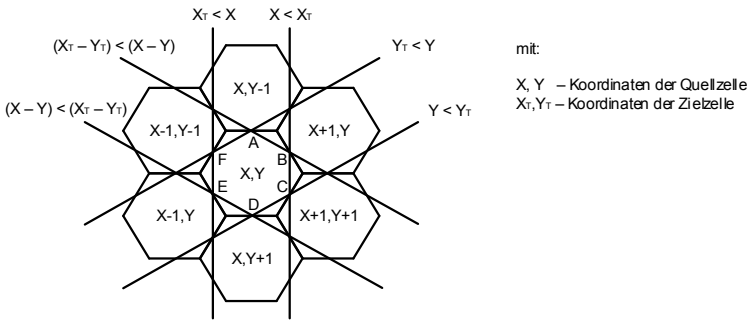



**Abbildung 51: Routenberechnung basierend auf den Koordinaten der aktuellen Zelle**

Basierend auf den (X, Y)-Koordinaten der Quellzelle versucht die Architektur, mit Vergleichsoperationen (siehe Abbildung 51) zunächst die optimale Richtung zu bestimmen. Gelingt dies, wird die Richtung für optimales Routing ausgewählt, welche die meisten Ressourcen, hinsichtlich der freien Links, zu diesem Zeitpunkt aufweist. Gelingt dies nicht, wird eine freie Richtung ausgewählt und in beiden Fällen der Routing-Vorgang bei der nächsten Zelle fortge-

setzt. Wird jedoch eine optimale Route gewünscht, so bricht der Routing-Vorgang nach der misslungenen Suche einer optimalen Richtung bereits ab. Selbiges geschieht, wenn überhaupt keine neue Richtung gefunden wird. In beiden Fällen wird der Routing-Vorgang an die Vorgängerzelle mittels Back-Tracking zurückgegeben und der Prozess in dieser Zelle fortgesetzt.

Eine wichtige Eigenschaft des Algorithmus besteht darin, dass jede Zelle nur lokale Entscheidungen treffen kann. Dieser Umstand garantiert nicht die Qualität der erstellten Verbindung, jedoch wird ein Pfad gefunden, sofern einer existiert. Im Falle der Suche nach dem optimalen Pfad, wird dieser ebenfalls gefunden, wenn ein solcher existiert. Über die Zeit der Suche kann aber im Voraus keine exakte Angabe gemacht werden. Ein Worst-Case für die Dauer des Routing-Prozesses kann indes definiert werden. Dafür muss die Annahme getroffen werden, dass alle möglichen Pfade bis zum Finden der eigentlichen Route abgesucht werden. Im Allgemeinen lassen sich die Routing-Zeiten niedrig halten, wenn der Programmierer bei seiner Arbeit strukturiert vorgeht und sich an die Regeln des hierarchischen Programmiermodells hält.

### 4.3.3. Routing Instruktionen

Zur Steuerung des Routingprozesses wurden vier Instruktionen definiert, die diese Funktionen bieten: Aufbau von CG Verbindungen mittels 'Routing Instruction CG' (RICG), Aufbau von MG Verbindungen mittels 'Routing Instruction MG1' und '-2' (RIMG1, RIMG2), Selektives lokales und globales Löschen von CG und MG Verbindungen mittels 'Endpacket Instruction' (EPI), siehe Abbildung 52.

#### 4.3.3.1. Routing Instruction CG (RICG)

Die Instruktion RICG bietet primär die Möglichkeit CG Verbindungen zwischen zwei Zellen aufzubauen. Zu Optimierungszwecken bietet sie eine Reihe von Optionen, die hier in Kürze vorgestellt werden.

Das Bit "ComBit" dient dazu bei bestehenden Konfigurationsverbindungen Instruktionen ohne Effekt auf diese Verbindung zu transportieren. Bei 'ComBit' = '1' wird nur die terminierende Zelle einer Konfigurationsverbindung die Auswertung der Instruktion vornehmen, alle übrigen Zellen lassen die Instruktion passieren. Andernfalls kann beispielsweise eine 'EPI' Anweisung während sie getunnelt wird eine bestehende Verbindung löschen, ohne dass es gewollt ist. Das 'CfgBit' wird von InReg genutzt um bei Routingprozessen zwischen einer Daten- und einer Konfigurationsverbindung zu unterscheiden. Der eigentliche Effekt greift bei der Zielzelle, wo entweder eine neue Verbindung

aufgebaut wird ('CfgBit' = '0') oder eine Terminierung zum Konfigurationsport der FU erfolgt.

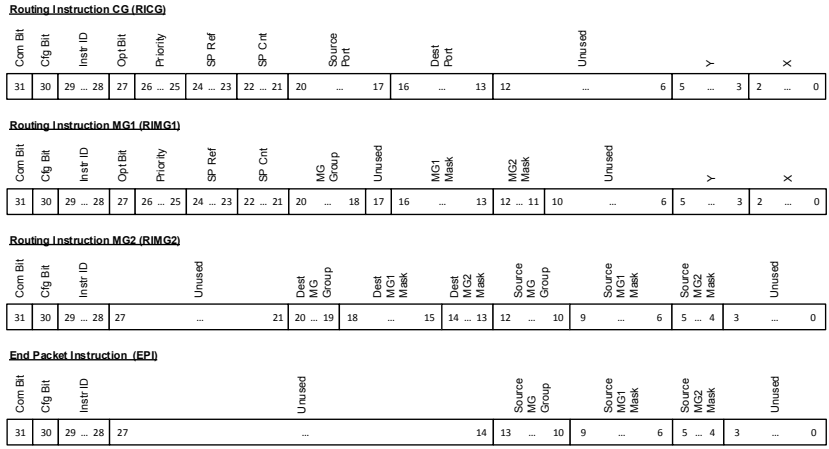



**Abbildung 52: Definition der Routing-Instruktionen für die Steuerung des adaptiven Routings zur Laufzeit innerhalb der HoneyComb-Architektur**

Das 'OptBit' sorgt dafür, dass eine Verbindung zwischen zwei Zellen den kürzesten Pfad darstellt ('OptBit' = '1'), andernfalls wird ein möglicher Weg zum Ziel gesucht, ohne Rücksicht auf seine Qualität. Der kürzeste Pfad lässt sich in einem homogenen Koordinatensystem relativ einfach bestimmen, siehe Abbildung 53. Dafür dürfen jeweils die Koordinaten der nächsten Zelle nur in die Richtung der X und Y Zielkoordinaten inkrementiert oder dekrementiert oder beim aktuellen Wert belassen werden. Ist beispielsweise die Zielkoordinate größer als die Quellkoordinate, so darf die nächste Zelle den gleichen oder einen um eins höheren Wert aufweisen. Dies gilt für beide Koordinatenkomponenten unabhängig. Grundsätzlich nutzt der RoutCtrl die in Abbildung 51 dargestellte Sicht des Arrays, um über erläuterte Vergleiche der Koordinaten zwischen der aktuellen Zelle und der Zielzelle die Ausgangsrichtung zu finden.

Das 'Priority' Feld dient der RU zur Priorisierung der eingehenden Requests bei gleichzeitigem Eintreffen. In diesem Fall wird der Request mit höherer Priorität bevorzugt behandelt und als erster ausgeführt.

Die Felder 'SPRef' und 'SPCnt' dienen zum Aufbau von Verbindungen mit einer Bypass-Funktion. Die 'SPRef' definiert einen Referenzwert, während 'SPCnt' von jedem InReg Module inkrementiert wird und bei Erreichen des Wertes von 'SPRef' auf null zurückgesetzt wird. Besitzt 'SPCnt' den Wert 'SPRef', so werden die Daten dieser Verbindung im aktuellen InReg gepuffert, andernfalls nicht. Damit lässt sich ein Pfad aufbauen, der weniger gepufferte

Hops besitzt als er InReg-Module passiert. So hat diese Option eine Speed Path Funktionalität, um Latenzen im globalen Kommunikationsnetzwerk zu senken. Die Anwendung dieser Funktion ist allerdings als kritisch einzustufen, da das Timing bei zu langen ungepufferten Pfaden durch die Timing Analyse der HoneyComb-Architektur nicht abgedeckt ist. Es muss also die Taktfrequenz der Architektur gesenkt werden, um die resultierenden Timing-Anforderungen zu erfüllen. Aus Kostengründen besitzt der Demonstrationschip diese Funktion nicht, da der Chip auf eine maximale Chipfläche von rund 16mm² beschränkt war und auf einige Funktionen in diesem Zusammenhang verzichtet werden musste.

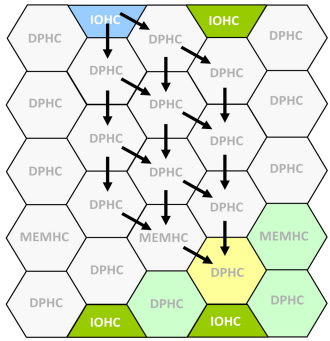



**Abbildung 53: Vollständige Routing-Karte für optimale Pfade zwischen der blauen IOHC und gelben DPHC mit insgesamt 10 möglichen Pfaden**

Die beiden Felder 'SourcePort' und 'DestPort' spezifizieren die zu verbindenden Quell- und Zielports der involvierten FUs einer Verbindung.

Die Koordinaten der Zielzelle werden durch die Felder X und Y definiert. Quellkoordinaten sind implizit durch die notwenige Routinginstruktion für die Konfigurationsverbindung, die zur Quellzelle vorausgehen muss, vorgegeben.

### 4.3.3.2. Routing Instruction MG 1 und 2 (RIMG1, RIMG2)

Der wichtigste Unterschied zwischen RICG und RIMGn liegt in der Differenzierung bzgl. der Routing-Granularität. Zum Aufbau der MG Verbindungen werden beide Instruktionen RIMG1 und RIMG2 benötigt, da ein 32-bit Wort nicht genügend Bits für alle Daten bereitstellt. Der Routing Vorgang wird mit RIMG1 gestartet, und mit RIMG2 beendet. Während bei CG FU-Ports nur einfach durchnummeriert wird, werden MG1 und MG2 Ports in Gruppen RTL-parametrisierbarer jedoch global-einheitlicher Größen eingeteilt. MG1 und MG2 Subgruppen können dabei unterschiedliche Anzahl und Granularität

aufweisen. Sie werden gruppenweise und gemeinsam als ein Vektor von RIMG1/2 Befehlen programmiert, siehe Abbildung 54.

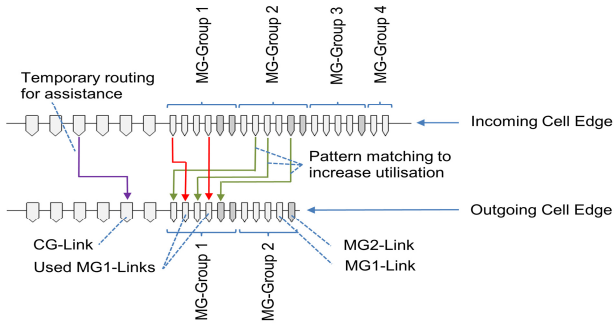



**Abbildung 54: Nach Gruppen eingeteilte multigranulare (MG) Ports während des Aufbaus einer Routing Verbindung mit assistierendem CG-Link.**

Während die ersten MG-Gruppen alle vollständig gefühlt sein müssen, ist es möglich, dass die letzte MG Gruppe nur unvollständig gefüllt ist. In der Abbildung dargestellte rote Verbindungen sind bereits belegte Ressourcen. Grüne Verbindungen hingegen werden in diesem Beispiel durch ein CG Link assistierend geroutet. Dazu transportiert der CG Link die RIMG1 Instruktion, welche die drei Zusatzfelder 'MGGroup', 'MG1Mask' und 'MG2Mask' definiert. MGGroup definiert hierbei die eingehende Gruppenummer (im Beispiel 'MGGroup'='1') und mittels 'MG1Mask' und 'MG2Mask' die genau Belegung der aktiven eingehenden Links (im Beispiel 'MG1Mask' = "0110", 'MG2Mask' = "10"). Nun wird der Request mit diesen Daten an den RoutCtrl weitergeleitet. Der RoutCtrl versucht für die neue Richtung eine MGGroup mit genügend Ressourcen für MG1 und MG2 zu bestimmen und erzeugt im Abschluss eine Gruppennummer, sowie MG1 und MG2 Belegungsmasken für die nächste Zelle. Diese Daten werden vor dem Weiterleiten an die nächste Zelle durch das InReg in RIMG1 eingesetzt, so dass die folgende RU diese Informationen aus der RIMG1-Instruktion entnehmen und auf die aktiven Links schließen kann.

Ist die Zielzelle erreicht, so quittiert dessen eingehendes InReg die RIMG1 Instruktion. Diese Quittierung wird sukzessive durch alle InRegs auf dem Weg zur Quellzelle bis zum beteiligten OutReg einschließlich bestätigt. Dieses OutReg löscht daraufhin die RIMG1 Instruktion und sendet stattdessen die folgende RIMG2 Instruktion. Diese Instruktion definiert Quell- und Zielgruppen (DestMGGroup und SourceMGGroup), sowie die zugehörigen Masken (DestMG1Mask, DestMG2Mask, SourceMG1Mask und SourceMG2Mask) analog zur RIMG1 Definition, allerdings auf die Ausgangs- und Eingangsports der Quell- und Ziel-FUs bezogen. Sobald die Instruktion RIMG2 durch den

kompletten Pfad bis zur Zielzelle propagiert ist, werden die MG Verbindungen wie im Zielteil der RIMG2-Instruktion spezifiziert verbunden. Daraufhin wird die RIMG2 Instruktion durch ein weiteres 'Ack'='1' am CG Link bestätigt und der CG Pfad ausgehend von der Quellzelle gelöscht. Ab diesem Zeitpunkt werden die CG-Ressourcen dieses Pfades für andere Routing-Prozesse freigemacht.

#### 4.3.3.3. Endpacket Instruction (EPI)

Zum Löschen aktiver CG und MG Verbindungen werden Endpacket Instruktionen (EPIs) genutzt, siehe Abbildung 52. Während beim Verbindungsaufbau alle RUs auf dem Pfad von der Quelle zum Ziel am Routing-Prozess beteiligt werden, sind beim Löschprozess lediglich die RUs der Quellzellen involviert. Dazu muss zunächst eine EPI-Instruktion durch eine Konfigurationsverbindung von einer IOHC zu dieser Zelle transportiert werden. Das empfangende InReg erkennt die Instruktion und schickt eine Anforderung an den RoutCtrl. Dieser wertet die Anforderung aus und schickt an die adressierten CG oder MG OutRegs der Zelle die Löschanweisungen. Über die CG und MG Links verschicken die OutRegs ein Löschsignal an die nachfolgenden Zellen und versetzen sich anschließen in den Leerlauf. Die beteiligten Multiplexer-Module besitzen genügend zusätzliche Logik, um die Löschsignale zu erkennen und die aktuelle Selektion zu löschen, sobald das nachfolgende InReg das Löschsignal akzeptiert hat. Dieser Vorgang setzt sich fort bis zum Eingangsport der FU der Zielzelle, indem sich alle beteiligten Multiplexer und InRegs zurücksetzen und die genutzten Ressourcen freigeben.

Die Datenfelder 'ComBit', 'CfgBit' und 'InstrID' der EPI-Instruktion haben die gleiche Funktion wie im Falle der 'RICG' und 'RIMG1' Instruktionen. Zum Adressieren der OutRegs werden die Felder 'SourceMGGroup', 'SourceMG1Mask' und 'SourceMG2Mask' genutzt. Das Feld 'SourceMGGroup' wird zur Adressierung sowohl von MG Gruppen als auch CG-Ports genutzt. Sind die Masken, welche durch die beiden Datenfelder

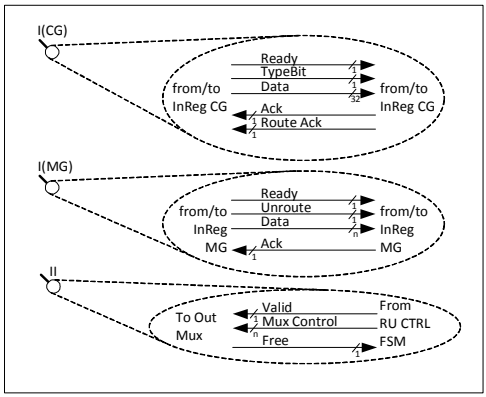



**Abbildung 55: Legende der Signale in Switchmatrizen**

SourceMG1Mask und SourceMG2Mask beschrieben sind, gleich null, so werden durch 'SourceMGGroup' CG-Ports adressiert, andernfalls repräsentiert dieser Wert eine MG Gruppe. Die Umrechnung auf die MG-Ports erfolgt analog zur Beschreibung im Abschnitt 4.3.3.2.

## 4.3.4. RU Komponenten

Im Folgenden werden die Module der RUs im Detail beleuchtet, um die Funktionen deutlicher herauszustellen.

### 4.3.4.1. Input Register CG (InRegCG)

Input Register CG (InRegCG) sind elementare Komponenten in der Realisierung des adaptiven Routings zur Laufzeit. Sie empfangen und detektieren eingehende Instruktionen an CG-Eingangslinks, weisen sie ab oder stellen eine Anforderung an den RoutCtrl, transportieren Instruktionen und Daten unter Verwendung von zwei oder mehr internen Registern, detektieren EPI-Anweisungen und führen die Löschung bestehender Routen aus. Abbildung 56 veranschaulicht den Aufbau eines InRegCG am Falle der finalen ASIC Implementierung mit vier internen Datenregistern. Die gestrichelten Komponenten in der Darstellung wurden im ASIC nicht umgesetzt.

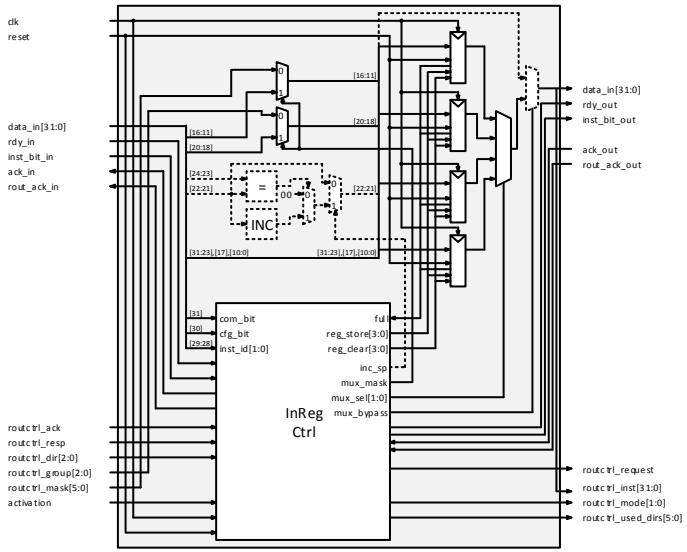



**Abbildung 56: Aufbau des InRegCG am Beispiel des ASICs mit vier internen Registern**

Der größte Teil der eingehenden Daten wird unverändert an die internen Datenregister geführt und in einem leeren Register gespeichert. Die Bits [20:18] und [16:11] beinhalten im Falle der RIMG1 Anweisung die MG-Group-Nummer als auch die MG-Mask-Bits und können bei Bedarf durch den ersten Multiplexer oben mit den Daten vom RoutCtrl kommend ersetzt werden. Dies erfolgt nach dem abgeschlossenen Routing-Prozess im RoutCtrl und die Änderung wird dabei im selben Datenregister gespeichert wie die Original-Anweisung. Daraufhin kann die veränderte Anweisung durch die passende Schaltung der Ausgangsmultiplexer an die nächste Zelle gesendet werden.

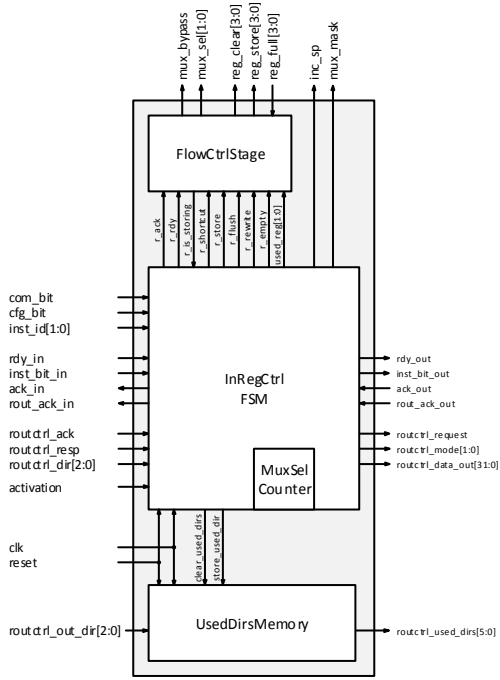



**Abbildung 57: Innere Struktur des InReg Controllers**

Bits [24:23] und [22:21] beziehen sich auf die Speed Path Parameter der RICG und RIMG1 Anweisungen. Hier wird das Feld 'SPCnt' inkremeniert und gleichzeitig mit 'SPRef' verglichen, siehe Abschnitt 4.3.3.1. Der inkrementierte Wert wird abschließend analog zum vorherigen Feld im aktiven Datenregister gespeichert. Erreicht 'SPCnt' den Wert von 'SPRef' dann werden die Eingangsdaten im aktuellen InReg gepuffert, andernfalls wird das InReg durch das Schalten des gestrichelten Ausgangsmultiplexers durchgeschleift. Diese Funktion ist im ASIC aus Kostengründen nicht umgesetzt.

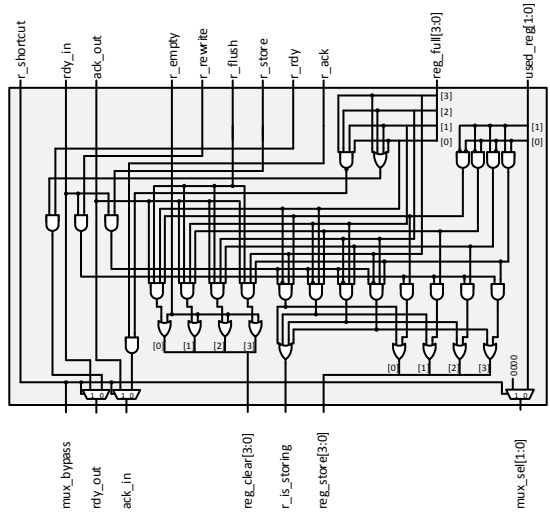



**Abbildung 58: FlowCtrlStage - Kombinatorische Schaltung zur Steuerung der Multiplexer, Register und Realisierung des Handshake-Protokolls**

Die vier Datenregister werden im Betrieb abwechselnd mit Daten gefüllt und geleert. In der Implementierung muss ein InReg immer mindestens zwei Register besitzen, damit Eingangslinks und Ausgangslinks zeitgleich Daten transportieren können. Bei der Verwendung von einem Register mit der Anforderung der gleichzeitigen Datentransfers auf beiden Seiten würde sich der kritische Pfad von der Eingangsseite zur Ausgangsseite durchziehen und die maximale Frequenz der Architektur deutlich bestimmen. Ab zwei Registern lässt sich das Handshake-Protokoll alleine basierend auf dem aktuellen Füllstand der Register steuern, ohne Bezug auf die Aktivitäten auf der jeweils anderen Seite des InReg zu nehmen.

**Tabelle 11 Steuersignale der FSM an die FlowCtrlStage**

| Steuersignal | Funktion |
|---|---|
| **r_empty** | löscht alle Register |
| **r_flush** | löscht den aktuellen Ausgangswert auf dem Register |
| **r_store** | speichert aktuellen Eingangswert |
| **r_rewrite** | überschreibt aktuellen Wert |
| **r_rdy** | aktiviert den Ausgang falls Daten vorhanden |
| **r_ack** | bestätigt den Eingang falls ein Register frei ist |
| **r_shortcut** | aktiviert die Bypass-Funktion |

Neben den Multiplexer- und Registerstrukturen besitzen InRegCGs einen Controller, der zur Erfüllung der beschriebenen Aufgaben die notwendigen Steuersignale erzeugt, siehe Abbildung 57. Um die notwendige FSM nicht zu kompliziert werden zu lassen, wurden zwei Aufgaben des Controllers in separate Module ausgelagert: FlowCtrlStage und UsedDirsMemory Module.

Das erste Modul ist eine rein kombinatorische Schaltung, welche die FSM bei der Steuerung der Register unterstützt, in dem Multiplexer- und Register-Steuersignale sowie das Handshake-Protokoll unabhängig von FSM-Aktivitäten geschaltet werden, siehe Abbildung 58. Im Wesentlichen ist dieses Modul dafür verantwortlich, die Datenein- und -ausgänge zu steuern und dafür zu sorgen, dass der Datenfluss reibungslos abläuft. Zu diesem Zweck besitzt dieses Modul von der FSM kommend mehrere Steuersignale, die bestimmte Betriebsmodi erzwingen, siehe Tabelle 11. Das Signal 'r_is_storing' zeigt an, dass aktuell ein Datenwort am Eingang in einem der Register gespeichert wird. Die übrigen Signale dieses Moduls werden zur Ausgangsmultiplexer-Steuerung und direkt zur Handshake-Protokoll Signalerzeugung genutzt. Um diese Aufgabe erfüllen zu können, werden durch die Signale wie 'reg_full' der aktuelle Status der Register diesem Modul zugeführt. Darüber hinaus zeigt das Signal 'used_reg' das aktuelle aktive Datenregister an, welches am Ausgang seine Daten anliegen hat. Dieser Wert wird vom MuxSel Counter zirkulär hochgezählt, wenn der Wert am Ausgang vom nachfolgenden Modul entgegengenommen wird.

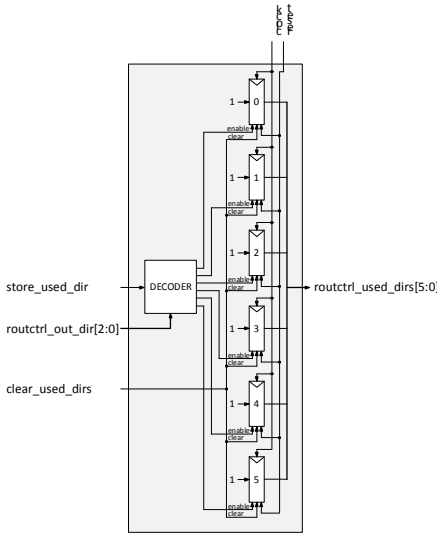



**Abbildung 59: UsedDirsMemory Modul zum speichern der beim Routing geprüfter Zellrichtungen**

Das Modul UsedDirsMemory hält die bereits geprüften Richtungen der Zelle vor, die beim RoutCtrl für den Routing-Vorgang geprüft wurden. Während dieses Vorgangs liefert RoutCtrl eine dual kodierte Richtungsnummer an das InRegCG, welche dekodiert im zugehörigen Register gespeichert wird. Jedes Register in diesem Modul repräsentiert eine Zellrichtung. Der Ausgangsvektor wird dabei vom RoutCtrl genutzt, falls weitere Routing-Versuche nötig sein sollten. Das Signal 'store_used_dir' wird von der

FSM erzeugt und damit die geprüfte Richtung dokumentiert. Im Leerlauf löscht die FSM die gespeicherten Richtungen mit dem Setzen des 'clear_used_dirs'.

Die Hauptsteueraufgabe im InRegCG kommt der Zustandsmaschine im InRegCtrl zu, siehe Abbildung 60. Dazu besitzt diese FSM 16, die hier erläutert werden.

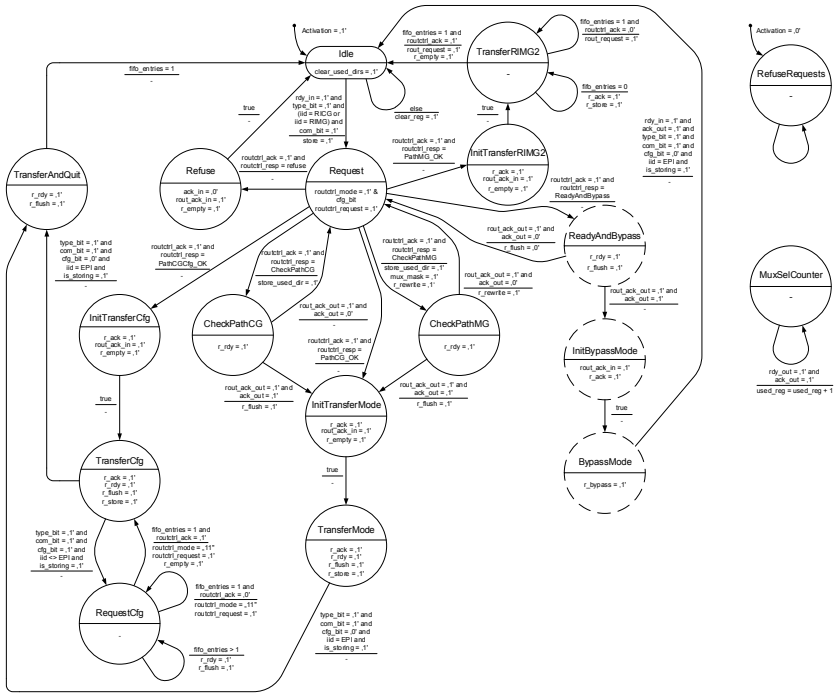



**Abbildung 60: Zustandsmaschine der InRegCGs zum Management eingehender Instruktionen, Datentransport, Auf- und Abbau von Verbindungen**

Im Leerlauf oder nach einem Reset befindet sich die Zustandsmaschine im 'Idle'-Zustand. Dieser wird nur verlassen, falls das Signal 'Activation' auf eins gesetzt wird. Dies ist ein zellglobales Signal und wird im globalen Controller der Architektur gesetzt oder gelöscht. Ist die Zelle deaktiviert, so wird jeder Kommunikationsversuch von außen bereits vom InRegCG abgewiesen. Ist die Zelle aktiv ('Activation' = '1'), dann sind Zustandsübergänge möglich und das InRegCG kann seine Aufgabe erfüllen.

Bei Ankunft einer RICG oder RIMG1 Instruktion wechselt die FSM in den Zustand 'Request'. Dabei wird die ankommende Instruktion im Datenregister 0

gespeichert. In diesem Zustand wird der RoutCtrl basierend auf den Parametern der gespeicherten Instruktion veranlasst, die nächste Richtung für das Routing zu bestimmen. Die Antwort des RoutCtrl veranlasst die FSM, in einen der nächsten Zustände zu wechseln.

Falls es sich um eine Konfigurationsverbindung handelt und das Ziel ist mit dieser Zelle erreicht, antwortet der RoutCtrl mit 'routctrl_resp' = 'PathCGCfg_OK' und erstellt eine Verbindung zwischen InRegCG-Ausgang und dem dedizierten FU-Konfigurationsport. Die FSM wechselt dabei in den 'InitTransferCfg'-Zustand. Dabei wird eingangsseitig die eingehende Instruktion quittiert, so dass der Pfad bis zum Ziel bestätigt und damit etabliert wird. Im nächsten Takt wechselt die FSM in den Zustand 'TransferCfg' und ist bereit Konfigurationen oder Instruktionen zu transportieren. Instruktionen sind in diesem Kontext mit 'TypeBit' = '1' markiert, andernfalls handelt es sich um Konfigurationsdaten für die FU der Zelle. Weiterhin müssen diese Instruktionen mit 'cfg_bit' = '1' markiert sein, um als Solche erkannt zu werden. Ankommende Instruktionen werden beim Wechsel in den 'RequestCfg' Zustand gespeichert. In diesem Zustand wird RoutCtrl veranlasst, die Instruktion zu verarbeiten, in dem im Falle der RICG oder RIMG1 ein neuer Routing-Prozess gestartet, im Falle der RIMG2 die Instruktion zum zugehörigen OutReg übertragen und im Falle der EPI Instruktion ein Löschvorgang einer CG oder MG Route veranlasst wird. Ist der Request verarbeitet, antwortet RoutCtrl mit 'routctrl_ack' = '1' und die FSM wechselt in den 'TransferCfg' Zustand. Beim Empfang einer EPI-Instruktion ohne die 'CfgBit' = '1' Markierung wird der Konfigurationspfad abgebaut. Die FSM wechselt in den 'TransferAndQuit' Zustand. Hier wird EPI durch den nächsten Multiplexer propagiert und damit die Verbindung zur FU gelöst. Daraufhin wechselt die FSM in den Zustand Idle und wartet auf neue Instruktionen.

Beim Routen einer CG-Verbindung gibt es beim Wechsel aus dem Request-Zustand zwei weitere Möglichkeiten. Ist die Zielzelle noch nicht erreicht und die Richtungssuche ist erfolgreich, so antwortet RoutCtrl mit 'CheckPathCG' und die FSM wechselt in den gleichnamigen Zustand. Dabei wird die aktuelle Richtung im 'UsedDirsMemory' Module gespeichert und steht dem RoutCtrl beim nächsten möglichen Routing-Versuch zur Verfügung. Weiterhin wird in diesem Zustand ausgangseitig ein 'Ready' = '1' Signal erzeugt und auf eine Antwort der folgenden InRegs gewartet. Ist die Antwort negativ ('ack' = '0' und 'rout_ack' = '1'), dann wechelt die FSM in den 'Request' Zustand und veranlasst den RoutCtrl eine neue Routing-Richtung zu bestimmen. Im Falle einer positiven Antwort ('ack' = '1' und 'rout_ack' = '1') wechelt die FSM in den 'InitTransferMode' Zustand. Dabei wird den Vorgänger-InRegs auf die gleiche Weise das erfolgreiche Routing signalisiert. Im nächsten Takt wechselt die FSM in den 'TransferMode' Zustand und ist bereit Daten und auch Instruktio-

nen zu übertragen. Wichtig bei den letzteren ist es, dass die Instruktionen mit 'cfg_bit' = '1' markiert sind. Dies signalisiert, dass Daten dieser Art durch die Route getunnelt werden und keine Verarbeitung stattfinden soll. Bei Ankunft einer EPI-Instruktion mit 'cfg_bit' = '0' wechselt die FSM in den 'TransferAndQuit' Zustand, während diese Instruktion in einem Datenregister gespeichert wird. Hier wartet die FSM bis alle Daten bis auf diese Instruktion zum nächsten Modul übertragen wurden und wechselt anschließend in den 'Idle' Zustand, während das letzte Wort übertragen wird und anschließend alle Register leer sind.

Eine weitere Möglichkeit, wie RoutCtrl auf eine CG-Routing-Anfrage antworten kann, ist gegeben, wenn tatsächlich die Zielzelle für eine reine Datenverbindung erreicht ist. In diesem Fall antwortet RoutCtrl mit 'PathCG_OK' und die FSM wechselt direkt in den 'InitTransferMode' Zustand. RoutCtrl stellt dabei eine Verbindung mit dem in der RICG Instruktion definierten CG-Eingangsport der FU, so dass Daten direkt an diesen Port übertragen werden können. Das weitere Vorgehen der FSM entspricht der vorherigen Beschreibung, es werden lediglich keine Unterscheidungen zwischen den Instruktionen und Daten gemacht, da alle Instruktionen bis auf EPI an die FU übertragen werden.

Auch im Falle des Routings von MG-Verbindungen gilt es zwei Fälle zu unterscheiden. Analog zu CG-Verbindungen ist zunächst einmal eine Durchleitung möglich. In diesem Fall antwortet der RoutCtrl mit CheckPathMG auf eine RIMG1-Instruktion und die FSM wechselt in den gleichnamigen Zustand. Beim Wechsel wird die neue Richtung gespeichert, die Ausgangsmaske in RIMG1 übernommen ('mux_mask' = '1') und die neue RIMG1 Instruktion im zuvor genutzten Register überschrieben ('r_rewrite' = '1'). In diesem Zustand wird die gültige RIMG1-Instruktion am Ausgang signalisiert ('r_rdy' = '1') und anschließend wartet die FSM auf eine Antwort vom folgenden InRegCG. Ist die Weiterleitung erfolgreich, wechselt die FSM in den 'InitTransferMode' Zustand und nach dem Versand der RIMG2-Instruktion wechselt die FSM über den 'TransferAndQuit' Zustand in den 'Idle' Zustand.

Ist beim MG-Routing die Zielzelle mit dem aktuellen InRegCG erreicht, so wechselt die FSM vom 'Request' Zustand in den 'InitTransferRIMG2' Zustand. Hier bestätigt die FSM den vorgehenden InRegCGs dieser Route das erfolgreiche Erreichen der Zielzelle und wechselt in den 'TransferRIMG2' Zustand. Hier wartet das InRegCG auf den Versand der RIMG2-Instruktion durch die Quellzelle und das Durchqueren der Instruktion durch alle beteiligten InRegCGs bis sie im aktuellen InRegCG ankommen. Hier leitet das InRegCG die Instruktion an den RoutCtrl weiter, der im Abschluss die Verbindungen zwischen den eingehenden MG1 und MG2 Links und den Links zur FU desselben Typs herstellt. Die Zielmasken für diesen Vorgang liefert die RIMG2-

Instruktion, die Quellmasken wurden zuvor von der RIMG1-Instruktion geliefert.

Die letzte Funktion dieser FSM bezieht sich auf die Bypass-Option. Soll dieses InRegCG seine transferierten Daten nicht puffern, so antwortet der RoutCtrl mit 'ReadyAndBypass' und versetzt die FSM in den gleichnamigen Zustand. Hier sendet das InRegCG den folgenden InRegCGs die Routing-Anfrage durch das Weiterleiten der RICG oder RIMG1 Instruktion. Bei erfolgreichem Routing antwortet der direkte Nachfolger mit 'ack_out' = '1' und 'rout_ack' = '1', die FSM wechselt in diesem Fall in den 'InitBypassMode'. Im Falle eines Fehlschlags wechselt die FSM in den 'Request' Zustand und wiederholt den Vorgang. Im 'InitBypassMode' Zustand wird dem vorgehenden InRegCG das erfoglreiche Routing signalisiert und gleich in den Zustand 'BypassMode' gewechselt. In diesem Zustand wird tatsächlich in den Bypass-Modus mit 'r_bypass' = '1' geschaltet. Wird in diesem Zustand die Übertragung einer EPI-Instruktion detektiert ('rdy_in' = '1', 'ack_out' = '1' und 'type_bit' = '1'), wird der Bypass-Modus durch den Wechsel in den 'Idle' Zustand verlassen.

Findet der RoutCtrl keine Möglichkeit eine Route zu erstellen, so meldet dieser mit 'routctrl_resp' = 'Refuse' einen Fehler und die FSM wechselt in den 'Refuse' Zustand und im nächsten Takt sofort in den 'Idle' Zustand. Dabei meldet InRegCG an das Vorgänger-InRegCG das Misslingen des Aufbaus einer Verbindung und bricht die Operation ab.

Nachfolgende Tabelle fasst nochmal alle sinnvollen Signalkombinationen an CG-Links zusammen und gibt eine Übersicht realisierbarer Funktionen:

**Tabelle 12 Handshake-Protokoll an CG-Links auf RU-Ebene**

| Ready | Ack | Rout_Ack | TypeBit | Funktion |
|---|---|---|---|---|
| 0 | 0 | 0 | - | keine Aktivitäten |
| 1 | 0 | 0 | 0 | Daten liegen an, keine Annahme |
| 1 | 1 | 0 | 0 | Daten werden übertragen |
| 1 | 0 | 0 | 1 | Instruktionen liegen an, keine Annahme |
| 1 | 1 | 0 | 1 | Instruktionen werden übertragen |
| 1 | 1 | 1 | 1 | Route wird bestätigt |
| 1 | 0 | 1 | 1 | Route wird abgewiesen |

#### 4.3.4.2. Input Register MG (InRegMG)

Analog zu InRegCG Modulen bieten InRegMG Module Verwaltungsfunktionen, um das Routing von MG1- und MG2-Verbindungen zu ermöglichen, aber auch bestehende Verbindungen zu löschen und die Ressourcen wieder

freizugeben. In ihrer Komplexität sind diese Module im Vergleich deutlich einfacher aufgebaut, was auch daran liegt, dass die eigentliche Logik für den Verbindungsaufbau in den InRegCG-Modulen implementiert ist. Darüber hinaus ist die Datenbreite der MGn-Links auf ein oder einige wenige Bits beschränkt, was auch deutlich die Chip-Fläche im Vergleich zu InRegCG reduziert. Abbildung 61 veranschaulicht das Blockschaubild des InRegMG.

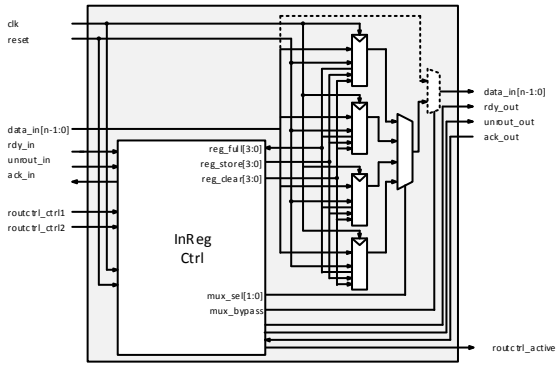



**Abbildung 61: Blockschaubild des InRegMG**

Es ist augenscheinlich die reduzierte Anzahl der Steuerleitungen im Vergleich zu InRegCG erkennbar. Ähnlich verhält es sich mit der Anzahl der Datenregister, die wie im CG Fall auf vier festgelegt wurde. Eingehende Daten werden der Reihe nach in den Datenregistern gespeichert und durch den Ausgangsmultiplexer selektiert. Der Datenfluss funktioniert nach dem Prinzip eines FIFOs. Das Handshake-Protokoll für die Eingangs- und Ausgangsseite wird vom InReg-Controller und damit hauptsächlich von der internen FSM gesteuert. Die Kommunikation des RoutCtrl mit den InRegMG-Modulen erfolgt über drei Signale, zwei in Richtung des InRegMG (routctrl_ctrl1 und routctrl_ctrl2) und ein Statussignal zurück zum RoutCtrl (routctrl_active). Damit kann der RoutCtrl Fehlbelegungen erkennen und dadurch Fehler an den globalen Controller melden.

Die FSM des InRegMG ist im Ruhezustand oder nach einem Reset im 'Idle' Zustand, siehe Abbildung 62. Soll das Modul gepufferte Daten transferieren, so signalisiert RoutCtrl dies mit 'routctrl_ctrl1' = '1' und 'routctrl_ctrl2' = '1' und die FSM wechselt in den Zustand 'TransferMode'. Hier bedient die FSM das Handshake-Protokoll und steuert die Datenregister sowie Multiplexer des InRegMG. Die Selektion des Ausgangsregisters wird dabei über einen zusätzlichen Zähler (MuxSelCounter) realisiert, der hochzählt, sobald am Ausgang ein aktiver Transfer detektiert wird ('rdy_out' = '1' und 'ack_out' = '1'). Die Erzeugung der $f_{store}$ und $f_{clear}$ Signale wird durch eine zusätzliche Schaltung

realisiert, siehe Abbildung 63. Hierbei werden externe Handshakeprotokolle genutzt, um die Steuerung der Register umzusetzen. Verlassen wird der 'TransferMode' Zustand durch drei Ereignisse.

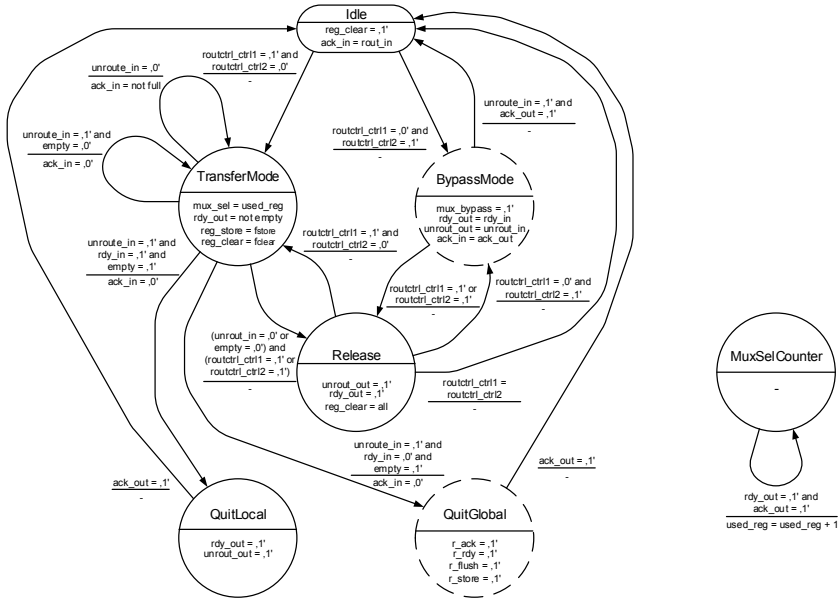



**Abbildung 62: Relativ überschaubare FSM des InRegMG Moduls**

Der erste Fall tritt ein, wenn ein Löschsignal ('unroute_in' = '1' und 'rdy_in' = '1') am Eingang erkannt wird. Zwei Aktionen können in diesem Fall passieren. Erstens, das InRegMG wartet bis alle Daten übertragen wurden und die internen Datenregister leer sind. Zweitens, die FSM wechselt in den 'QuitLocal' Zustand und speichert dabei das Löschsignal nicht einem seiner Datenregister. Dies geschieht, wenn alle Datenregister bereits leer sind oder sobald dieser Fall nach einer Wartezeit eintritt. Die Weiterleitung des Löschsignals an das nachfolgende Modul erfolgt dabei durch die Zustandskodierung 'rdy_out' = '1' und 'unrout_out' = '1'). Wird daraufhin ein Transfer am Ausgang detektiert mit 'ack_out' = '1', dann wechselt die FSM in den 'Idle' Zustand und ist damit im Leerlauf. Dieser Pfad der FSM wird beschritten, wenn ein lokales Löschsignal am Eingang detektiert wird. Für den gesamten Pfad hat dies zur Folge, dass ein Pfad vom OutRegMG einer FU bis zum Eingang einer anderen FU gelöscht wird, ohne seine bis dahin aktive Funktion zu beeinflussen.

Eine weitere Möglichkeit besteht hier beim Empfang eines globalen Löschsignals mit 'unroute_in' = '1' und 'rdy_in' = '0'. Dabei handelt es sich um ein

globales Löschsignal und bewirkt, dass alle Pfade einer Konfiguration gelöscht werden. In diesem Fall wechselt die FSM in den 'QuitGlobal' Zustand, sobald alle internen Datenregister leer sind. Hier wird erneut durch den Zustand kodiert, das globale Löschsignal an das nachfolgende Modul geleitet und bei aktivem Transfer am Ausgang wechselt die FSM in den 'Idle' Zustand und wartet damit auf neue Anforderungen seitens des RoutCtrl. Das globale Löschsignal war ein Konzept zum schnellen Löschen der bestehenden Konfiguration, dass allerdings im finalen ASIC aus Kostengründen keine Anwendung fand. Dabei sollten alle Verästelungen einer Konfiguration durch das Löschen der Wurzeln sukzessive auf RU und FU Eben gelöscht werden.

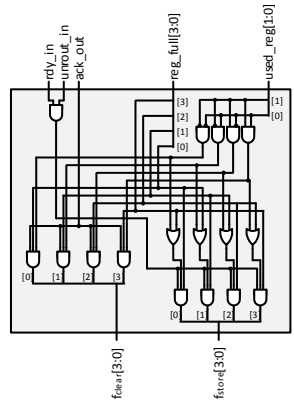



**Abbildung 63: Generierung der 'Clear' und 'Store' Signale in Abhängigkeit des Handshake-Protokolls am Ein- und Ausgang**

Die letzte Möglichkeit den 'TransferMode' Zustand zu verlassen besteht darin ein Signal vom RoutCtrl zu erhalten ('routctrl_ctrl1' = '1' oder 'rout_ctrl_ctrl2' = '1'), welches dem InRegMG das Deaktivieren des Pfades meldet, falls beim Aufbau einer MG-Verbindung eine getestete Route nicht erfolgreich war. Die FSM wechselt in diesem Fall in den 'Release' Zustand. Durch das Senden des Löschsignals durch das InRegMG an die nachfolgenden Module mit 'unrout_out' = '1' und 'rdy_out' = '1' werden damit alle Multiplexer-Einstellungen dieser RU gelöscht. Durch den Wechsel in den 'Idle' Zustand hat der RoutCtrl hat weiterhin die Möglichkeit das InRegMG vollständig in den Ruhezustand zu versetzten oder für weitere Routing-Aktivitäten zurück in den 'TransferMode' Zustand zu wechseln.

Ein weiterer Betriebsmodus des InRegMG besteht darin, den 'BypassMode' Zustand zu nutzen. Hierbei werden keine Daten in diesem Modul gepuffert und der Eingang wird direkt an den Ausgang durchgestellt. Ein eingehendes

aktives Löschsignal ('unroute_in' = '1' und 'ack_out' = '1') wird in diesem Zustand erkannt und die FSM wechselt in den 'Idle' Zustand. Eine weitere Möglichkeit diesen Zustand zu verlassen besteht wie im 'TransferMode' Zustand durch ein Signal vom RoutCtrl, welches während des Routing-Prozesses genutzt wird, um den MG-Verbindungsaufbau zu steuern.

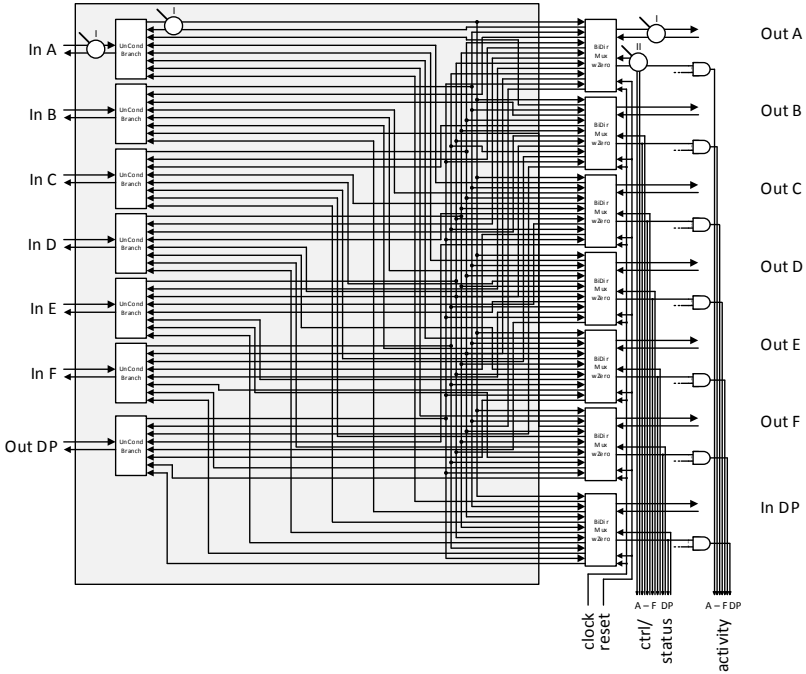



**Abbildung 64: Multiplexer- und Brancher-Strukturen innerhalb der Switch Matrizen mit integrierter Logik zum Löschen bestehender Verbindungen**

Ein InRegMG Modul realisiert lediglich ein Link, welches im Routing-Prozess Bestandteil einer MG-Gruppe ist. Während der Instanziierung in der RU wird aus einem InRegMG durch Parametrisierung ein InRegMG1 oder InRegMG2 Module realisiert. Hierbei werden insbesondere Datenbusbreiten spezifiziert. Das Handling einer MG-Gruppe durch das RoutCtrl erfolgt analog zum Gruppeaufbau durch die Ansteuerung mehrerer Links einer MG-Gruppe zugleich. Interne Prozesse, welche diese Steuerung auslösen, werden durch die 'RIMG1' und 'RIMG2' Instruktionen ausgelöst. In der ASIC-Implementierung wurden die beiden Zustände 'BypassMode' und 'QuitGlobal' aus Kostengründen nicht integriert.

### 4.3.4.3. Switch Matrizen

Den größten Anteil an der RU haben die Switch-Matrizen, siehe Abbildung 64. Sie beinhalten die bidirektionale Multiplexer-Logic (BiDirMuxwZero, siehe Abbildung 65), und Unconditional Brancher Module (UnCondBranch, siehe Abbildung 66), um bidirektionale Signale des Handshake-Protokolls zu transportieren. Letztere stellen eine zusätzliche Logik bereit, um rücklaufende 'Ack' und, falls wie im Falle von CG vorhanden, 'Route Ack' Signale disjunktiv zu verknüpfen und den InRegs zuzuführen. InRegs werden hierbei an den Ports in Richtung A-F angeschlossen, siehe Abbildung 64. Die Abbildung veranschaulicht sowohl die CG als auch MG Versionen der Switch-Matrizen ab. Zur Übersicht verwendeter Signale siehe Abbildung 55.

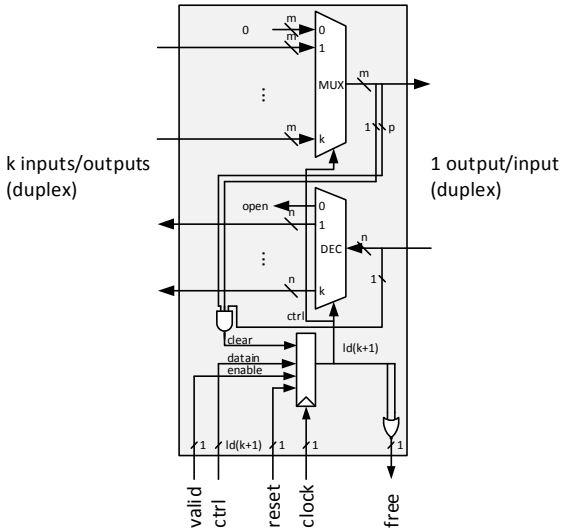



**Abbildung 65: BiDirMuxwZero - Bidirektionalle Multiplexer-Struktur mit integrierten Registern zur Speicherung der Einstellungen und Löschfunktion**

BiDirMuxwZero Module sind im Aufbau komplexer als einfache Multiplexer. Zusätzlich zu Multiplexern für die ausgehende Richtung der Links und Decodern für eingehende 'Ack' und 'RoutAck' Signale, beherbergen diese Module zusätzliche Register zur Speicherung der Multiplexer- und Decoder-Ansteuerung. Die Belegung dieser Module erfolgt durch den RoutCtrl. Dazu wurden zusätzliche Steuersignale 'Valid' und 'MuxControl' zwischen dem RoutCtrl und dem Multiplexer-Modul implementiert. 'MuxControl' transportiert hierbei den Wert für die Multiplexer. 'Valid' zeigt an, wann die neue Belegung in die internen Register übernommen werden kann. Welcher Wert in

den internen Registern gespeichert werden soll, wird zuvor vom RoutCtrl im Zuge der Routing-Berechnung bestimmt.

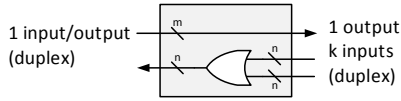



**Abbildung 66: UnCondBranch - Disjunktive Zusammenfassung rücklaufender Ack-Signale zur Rückführung an die InRegCG/MG Module**

BiDirMuxwZero Module enthalten zusätzlich Erkennungslogik für EPI Instruktionen und abgewiesene Routing-Versuche an CG-Links und Löschsignale an MG-Links. Falls diese Bedingungen detektiert werden, so wird der Inhalt der internen Register auf Null gesetzt und die Multiplexer gesperrt. Zu diesem Zweck sind die internen Multiplexer und Decoder so gebaut, dass sie in Null-Stellung keine Signale durchlassen und erst bei anliegender eins und darüber einen der anliegenden Eingangskanäle bidirektional durchstellen. Das Modul enthält Logik für beide Formate (CG und MG), die durch Parameter bei der Instanziierung spezifiziert wird.

Abbildung 64 veranschaulicht jeweils nur einen Multiplexer pro ausgehender Richtung. Somit kann diese Struktur nur einen Link pro Seite unterstützen. In der realen Anwendung kann die Anzahl verwendeter Links zwischen null und 16 variieren, so dass diese Struktur stark anwachsen kann. Jeder Datentyp (CG, MG1, MG2) besitzt eine eigene Switch-Matrix mit vordefinierter Anzahl Links pro Seite und Breiten der Datenbusse.

Neben den Datensignalen liefern Switch-Matrizen Status-Informationen zum Zustand aktueller Einstellungen. So wird das 'free' Signal der BiDirMuxwZero genutzt, um jeweils pro Multiplexer-Struktur dem RoutCtrl den Status zu signalisieren. Dieses Signal wird durch disjunktive Verknüpfung und anschließender Invertierung aus der Selektion des Multiplexers gebildet, so dass die Stellung null auf einen ungenutzt Multiplexer hinweist. Bündel der 'free' Signale werden im Vektor 'status' zusammengefasst und dem RoutCtrl zugeführt.

Pro Richtung werden 'free' Signale erneut disjunktiv verknüpft. Dies wird genutzt, um den aktuellen RUs und Nachbarzellen Aktivitäten zu melden, um die Steuerung des Clock-Gating zu ermöglichen.

#### 4.3.4.4. Output Register CG (OutRegCG)

Die Schnittstelle zwischen den FUs und RUs stellen die Output Register und die Output Module dar. Die ersteren nehmen dabei eine aktive Rolle insbesondere bei den Routing-Prozessen ein. Während InRegCG Module als Initiatoren der Routing-Vorgänge durch den Empfang der Routing-

Instruktionen fungieren und auch die Weiterleitung der Instruktionen bei laufenden Routing-Vorgängen übernehmen, bilden die OutRegCG Module die Quelle der Routing-Prozesse, siehe Abbildung 67. Jeder Routing Prozess im Array wird durch den Empfang einer entsprechenden Instruktion am InRegCG initiiert. Der berechnete FU-Ausgang wird dabei durch RoutCtrl an den Switch-Matrizen geschaltet und die empfangenen Routing-Instruktionen vom InRegCG zum aktivierten OutRegCG übertragen und gespeichert. Alle weiteren Vorgänge des Routing-Prozesses werden ab diesem Zeitpunkt vom OutRegCG initiiert, insbesondere wenn Nachbarzellen erfolglose Routing-Versuche melden. Schlagen alle Versuche fehl, eine Verbindung herzustellen, so wird die letzte Anforderung des OutRegCG vom RoutCtrl unbeantwortet bleiben und stattdessen meldet RoutCtrl einen Fehler an den globalen Controller. Es wurde kein Mechanismus vorgesehen, dass die RU sich von diesem oder anderen Fehlern erholt.

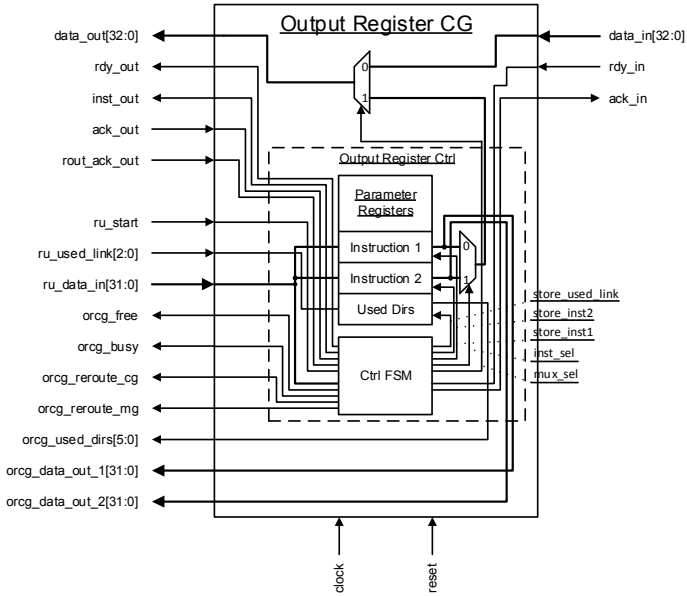



**Abbildung 67: Blockdiagramm des OutRegCG-Moduls**

Neben den zwei Routing-Instruktionen speichert OutRegCG bereits geprüfte Richtungen durch den Routing-Prozess (UsedDirs). Ausgehende FU-Daten werden in diesem Modul nicht gespeichert, stattdessen lediglich zu den Switch-Matrizen durchgestellt. Während der Routing-Prozesse werden gespeicherte Instruktionen selektiert (inst_sel) und durch den Ausgangsmultiplexer

weitergeleitet (mux_sel). Zur Kommunikation zwischen dem OutRegCG und RoutCtrl werden eine Reihe von Signalen genutzt, die eine Routing-Anforderung sowie eine Rückmeldung vom RoutCtrl erlauben. Signalnamen mit dem Präfix 'ru_' laufen in Richtung OutRegCG. Präfix 'orcg_' kennzeichnet Signale, die in Richtung RoutCtrl verlaufen, siehe Abbildung 67. Die Steuerung der Register und Multiplexer wird durch die OutRegCG-FSM übernommen, die im Weiteren erläutert wird.

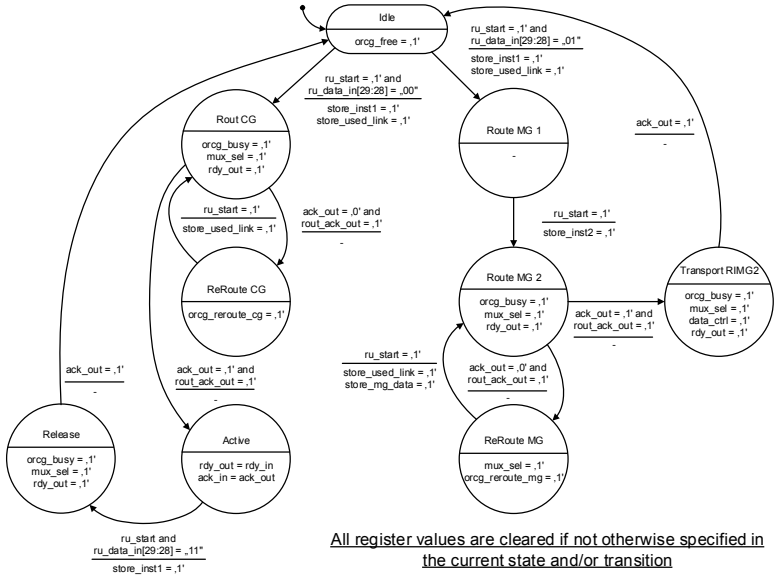



**Abbildung 68: Multiplexerstrukturen innerhalb der Switch Matrizen mit integrierter Logik zum Löschen bestehender Verbindungen innerhalb der Multiplexer**

Im Ruhezustand befindet sich die FSM im 'Idle' Zustand und meldet dies dem RoutCtrl durch das 'orcg_free' = '1' Signal. Meldet RoutCtrl in diesem Fall mit 'r_start' = '1' eine neue Anforderung, so erkennt die FSM die Art des Routing-Vorgang anhand der mit übertragenen Opcodes im 'ru_data_in' Vektor, welcher die komplette Instruktion (RICG oder RIMG1) beeinhaltet. Daraufhin speichert die FSM die neue Instruktion (stor_inst1) sowie die verwendete Ausgangsrichtung 'store_used_link' = '1' und wechselt in den 'RoutCG' Zustand. In diesem Zustand werden Ausgangsmultiplexer des OutRegCG geschaltet, so dass die gespeicherte Instruktion weitergeleitet wird und die Routing-Anforderung an die nächste Zell mit 'rdy_out' = '1' signalisiert wird.

Gleichzeitig schaltet RoutCtrl die Switch-Matrix, so dass eine Verbindung zum gewünschten InRegCG der nächsten Zelle entsteht. Danach wartet die FSM auf eine Antwort der folgenden Zelle. Ist die Antwort ein Scheitern der Routing-Verbindung ('ack_out' = '0' und 'rout_ack_out' = '1'), so wechselt die FSM in den 'ReRoutCG' Zustand und meldet eine neue Routing-Anforderung an RoutCtrl ('orcg_reroute_cg' = '1'). Sobald RoutCtrl die Anfrage verarbeitet hat, wird dieser Umstand mit 'ru_start' = '1' gemeldet und die FSM wechselt zurück in den 'RoutCG' Zustand.

Erfolgreiches Routing ('ack_out' = '1' und 'rout_ack_out' = '1') führt zum Wechsel in den 'Active' Zustand. In diesem Fall ist eine Verbindung erfolgreich hergestellt, so dass die FSM die Eingangssignale von FU zur nächsten InRegCG durchstellen kann. Wird im weiteren Verlauf ein Request zum Löschen der bestehenden Verbindung detektiert, so wechselt die FSM in den 'Release' Zustand. Das Löschen wird durch die Übertragung der EPI Instruktion zum OutRegCG initiiert. Bei Bestätigung der EPI-Annahme durch das nächste Modul, geht die FSM in den 'Idle' Zustand und wartet auf neue Anforderungen seitens des RoutCtrl.

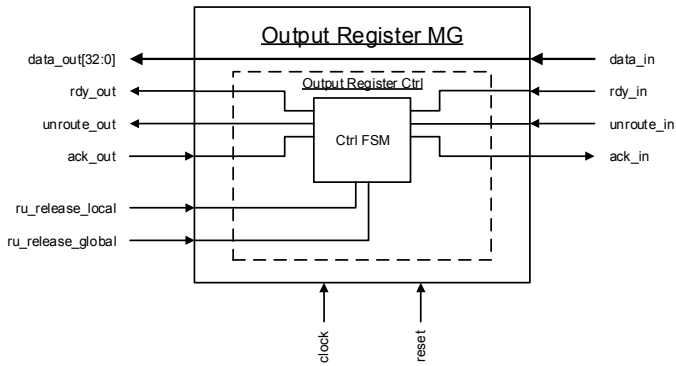



**Abbildung 69: Blockschaltbild des OutRegMG Moduls**

Im Falle des MG-Routings überträgt der RoutCtrl die RIMG1 Instruktion an den OutRegCG, welches in den Zustand 'RoutMG1' wechselt und gleichzeitig die erste Instruktion speichert. Hier wartet die FSM auf die Übertragung von RIMG2 und wechselt daraufhin in den 'RouteMG2' Zustand. Zugleich wird die RIMG2 Instruktion im OutRegCG gespeichert. Der Wechsel zwischen 'RoutMG2' und "ReRoutMG' geschieht in ähnlicher Weise, wie dies beim CG-Routing zwischen den Zuständen 'RoutCG' und 'ReRoutCG' erfolgt. Erfolgreiches Routing versetzt die FSM schließlich in den 'TransportRIMG2' Zustand, der für die Übertragung der RIMG2-Instruktion ('data_ctrl' = '1')

sorgt. Bei Bestätigung der Übertagung geht die FSM in den Ruhezustand über und kann für weitere Routing-Anforderungen in Anspruch genommen werden.

#### 4.3.4.5. Output Register MG (OutRegMG)

Im Vergleich zur Funktionalität des OutRegCG ist ein OutRegMG Modul recht einfach aufgebaut. Gegenüber RoutCtrl verhält sich dieses Modul vollständig passiv, da keine Statusinformationen oder Anforderungen geschickt werden können. Über zwei Signale (ru_release_local und ru_release_global) kann RoutCtrl lediglich ein Löschsignal an OutRegMG absetzen, um bestehende Verbindungen zu lösen. Abbildung 69 veranschaulicht das Blockschaltbild eines OutRegMG Moduls, welches im Wesentlichen aus einer Kontroll-FSM besteht. Die Datenleitungen werden ohne Zuschalten eines Multiplexers durchgeleitet, da Datenleitungen keinen Einfluss auf die Löschfunktion besitzen.

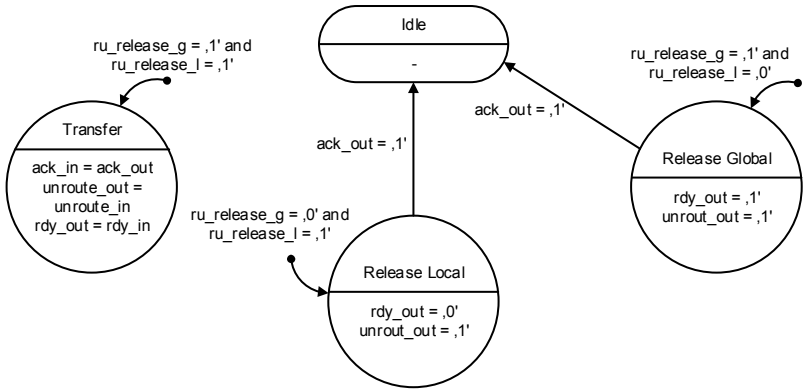



**Abbildung 70: Zustandsmaschine eines OutRegMG Controllers**

Aus dem Ruhezustand wird die FSM durch das Setzen der Steuersignale von RoutCtrl kommend direkt in drei mögliche Zustände versetzt. Im 'Transfer' Zustand (Eintritt bei 'ru_release_g' = '1' und 'ru_release_l' = '1') wird das Handshake-Protokoll des Ein- und Ausgangs des OutRegMG bedient und damit Datenübertragungen ermöglicht. Im 'ReleaseLocal' Zustand (Eintritt bei 'ru_release_g' = '0' und 'ru_release_l' = '1') sendet die FSM ein lokales Löschsignal an die folgenden Module und löscht die bestehende Verbindung bis zum abschließenden 'Output Module' der RU, siehe folgender Abschnitt.

Im 'ReleaseGlobal' Zustand (Eintritt bei 'ru_release_g' = '1' und 'ru_release_l' = '0') sendet die FSM ein globales Löschsignal an die folgenden

Module und kann das Löschen einer kompletten Konfiguration zur Folge haben. Beide Zustände werden verlassen, wenn die gesendeten Löschsignale mit 'ack_out' = '1' bestätigt wurden.

### 4.3.4.6. Output Module CG und MG (OutModCG/MG)

Output Module haben die Aufgabe die Übertagungssignale auf RU-Ebene an die Anforderungen der FUs anzupassen. Im Falle der CG-Daten besitzen diese auf RU-Ebene ein zusätzliches Signal (type_in) zur Unterscheidung zwischen Daten und Instruktionen. Beim Eintritt in OutModCG (siehe Abbildung 71) wird dieses Signal terminiert, in dem eine eingehende Instruktion ('type_in' = '1') durch logische Verknüpfung für das Ausblenden des 'rdy_out' Signals sorgt. Zusätzlich wird die Übertragung dieser Instruktionen dem Vorgänger bestätigt ('ack_in' = '1'). Die Verwaltung der Routing-Ressourcen innerhalb der FUs obliegt der FU-Konfigurationslogik.

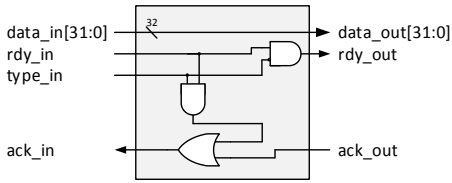



**Abbildung 71: OutModCG zur Terminierung der CG-Datenkanäle bei Eintritt in FUs**

Ähnlich verhält es sich bei OutModMG zur Terminierung der MG-Datensignale, siehe Abbildung 72. Hier wird insbesondere das 'unrout_in' Signal terminiert und dem Sender bestätigt ('ack_in' = '1'). FUs sind an diesen Vorgängen nicht beteiligt und müssen auf diese Instruktionen bzw. Signale nicht reagieren.

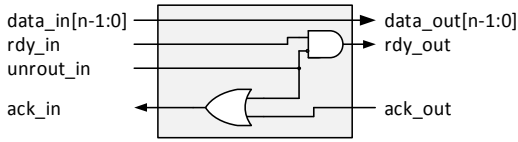



**Abbildung 72: OutModMG zur Terminierung der MG-Datenkanäle bei Eintritt in FUs**

### 4.3.4.7. Routing Controller

Die zentrale Komponente zur Verarbeitung der Anforderungen von InRegCG und OutRegCG kommend, d.h. zur Berechnung der neuen Routen und schalten der Switch-Matrizen, stellt der Routing Controller (RoutCtrl) dar. Er

lässt sich in ihrem Aufbau in weitere Komponenten aufteilen, siehe Abbildung 74. Verwendete Signale in diesem Blockschaltbild sind in Abbildung 73 zusammengefasst.

Eingehende Routing-Anforderungen von InRegCG Modulen (InReg CG A-F) werden zunächst in zwei Multiplexer-Stufen selektiert und anschließend der RoutCtrl-FSM zugeführt. Die erste Multiplexer-Stufe fasst zunächst seitenweise die Signale zusammen und anschließend werden die sechs Seiten der Zelle durch einen Multiplexer erneut selektiert. Auf diese Weise lässt sich eine überschaubare Ansteuerung der Multiplexer realisieren und die Anforderungen den Seiten der Zelle leichter zuordnen. Bevor die Verarbeitung der Anforderungen erfolgen kann, muss zunächst eine Auswahl der anliegenden Anforderungen erfolgen. Dieser Schritt wird durch das Modul 'DetectRequest' realisiert. Hier wird nach dem Prioritätsprinzip unter Beachtung der anliegenden Instruktionen und Prioritäten die nächste Anforderung in der Liste ausgewählt und an die RoutCtrl-FSM durchgestellt.

Die Ansteuerung der 'InRegMG1 A-F' und 'InRegMG2 A-F' Multiplexer-Stufen erfolgt hingegen rein nach der Vorgabe der RIMG1 und RIMG2 Instruktionen. Durch die Kodierung der MG Gruppen und der Masken innerhalb der Instruktionen werden diese Multiplexer direkt angesteuert und die InRegMG Module programmiert.

OutRegCG und OutRegMG Module werden durch einstufige Multiplexer selektiert. Das Auswahlverfahren erfolgt hier nach dem Round-Robin Verfahren. Während alle bisher besprochenen Multiplexer in diesem Abschnitt bidirektional Signale von und zu Modulen durchstellen, handelt es sich bei den OutRegMG Signalen um unidirektionale Signale.

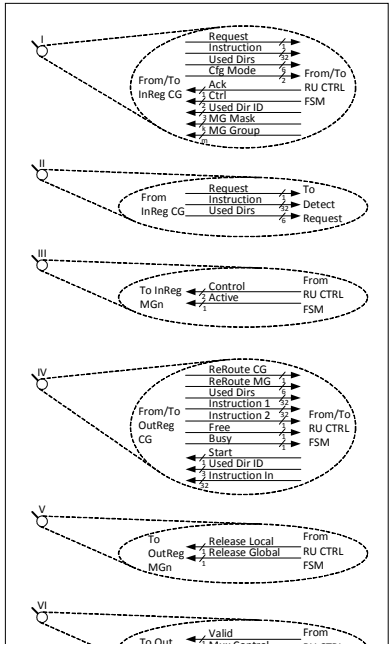



**Abbildung 73: Zusammenfassung genutzter Signale im Blockschaltbild des Routing Controllers**

Die eigentliche Routen-Kalkulation im RoutCtrl-Modul wird durch das Modul 'CalculateRoute' umgesetzt. Zu diesem Zweck erhält dieses Modul kompletten Zugriff auf die Statussignale der Switch-Matrizen und ist dadurch in der Lage freie Ressourcen und Belegungen in die Kalkulation mit einzube-

ziehen. Die Berechnungsanforderungen erhält das Modul direkt von RoutCtrl-FSM und sendet die Ergebnisse der Berechnung an die FSM zurück.

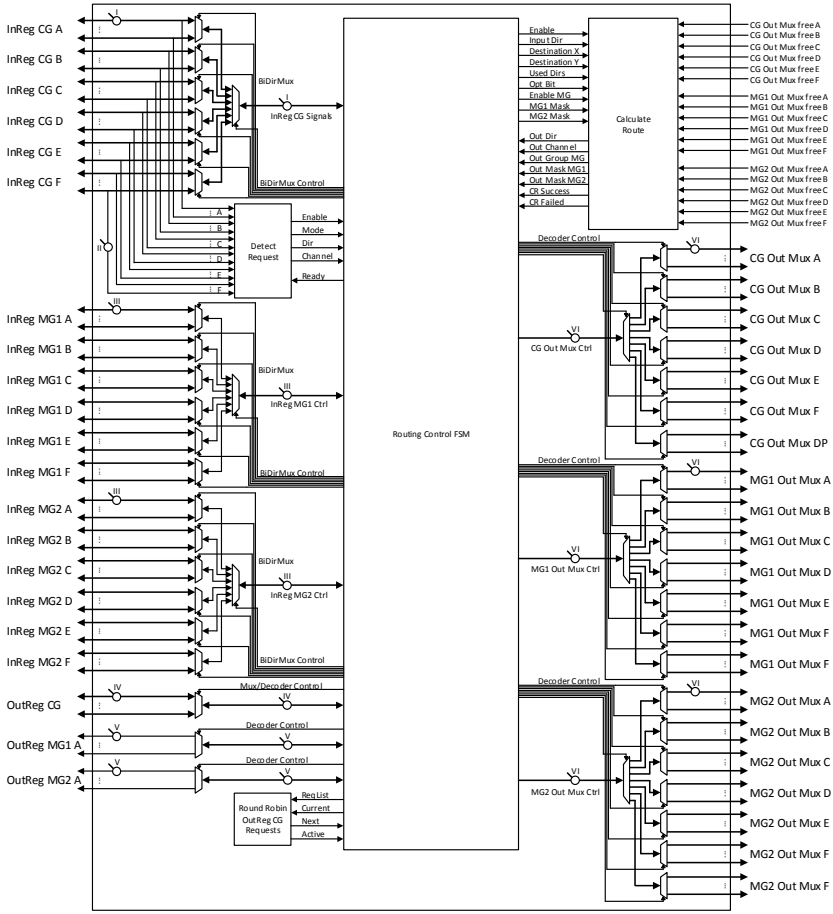



**Abbildung 74: Blockschaltbild des Routing Controllers (RoutCtrl) mit Multiplexer- und Decoderstrukturen, Request Detection, Route Calculation und Routing Control FSM Modulen**

Die 'CGOutMux A-F' Signale werden ebenfalls durch zweistufige Decoder an die Zielmodule in den Switch-Matrizen zugeführt. In gleicher Weise verhält es sich mit den 'MG1-' und 'MG2OutMux A-F' Signalen.

Der Kern des RoutCtrl besteht aus einer FSM, die alle Anforderungen entgegennimmt und die vorhandenen Ressourcen nutzt, um die Berechnung und

das Weiterleiten der Ergebnisse zu bewerkstelligen. Nunmehr werden die Funktionen der Komponenten im Detail beschrieben.

#### 4.3.4.7.1. Calculate-Route-Module

Die Routen-Berechnung erfolgt durch eine vorwiegend parallele kombinatorische Schaltung, dem Calculate-Route-Modul (CRM), die am Ende in einer Register-Stufe mündet. Der gesamte Algorithmus zur Berechnung der Route wurde zu diesem Zweck in kleine Schritte zerlegt, um die Komplexität überschaubar zu halten. Die eigentliche Aufgabenstellung lässt sich wie folgt beschreiben:

Basierend auf XY-Koordinaten der Zielzelle und eigener Position berechne die Richtung für den weiteren Verlauf der Route unter Einbeziehung der aktuellen Auslastung der Ressourcen und der optimalen Richtungen.

Der erste Schritt der Berechnung besteht in der Bestimmung der aktuellen Auslastung für jeden Link-Typ und Richtung der Zelle. Dafür werden basierend auf den Status-Signalen der Switch-Matrizen (CGfreevectorA-F) die Anzahl der freien Ausgänge je Richtung und ein 1-Bit breites Signal zur Anzeige freier Links bestimmt. Auf der MG Seite ist die Berechnung dieser Stufe deutlich komplexer. Hier werden die verfügbaren Ressourcen nicht per Link sondern für jede freie Gruppe berechnet. Zu diesem Zweck werden MG1 und MG2 Input Masks herangezogen und basierend auf den Daten der 'MG1 und MG2 free vector A-F' Vektoren die freien Ressourcen bestimmt. Am Ausgang stehen drei Parameter bereit: 'MG1/MG2count', 'MGGroupfreeflagA-F' und 'MGfreeGroupID'. Die ersten beiden Signale beschreiben wie viele MG1/MG2 Links in der aktiven Gruppe benötigt werden. Die 'MGGroupfreeflagA-F' geben mit je einem Bit pro Richtung an, ob freie Gruppen verfügbar sind. 'MGfreeGroupIDA-F' geben die Anzahl der MG-Gruppen an, die zum Routing in die jeweilige Richtung frei und nutzbar sind.

Nun beginnt die eigentliche Berechnung der Ausgangsrichtung, wie dies im Abschnitt 4.3.2 beschrieben wurde. Basierend auf den Koordinaten und der Eingangsrichtung (Input Direction Signal) werden zunächst mögliche Routing-Richtungen bestimmt (Check Routability for each Direction) und anschließend in zwei Klassen eingeteilt (Determine Optimum/All Directions). Wird das OptBit-Signal auf eins gesetzt, so werden hier nur optimale Richtungen betrachtet, andernfalls auch alternativen erzeugt. Im nächsten Schritt wird für jede Richtung eine Überdeckung gebildet (Check Available Directions), indem ressourcenseitig mögliche, berechnete und bereits geprüfte Richtungen übereinander gelegt werden. Hierbei werden Seiten mit keinen freien Ressourcen und bereits abgeschlossenen Routing-Versuchen vom weiteren Verlauf ausge-

schlossen. Geprüfte Seiten werden über den Vektor 'UsedDirections' von In-RegCG oder OutRegCG geliefert.

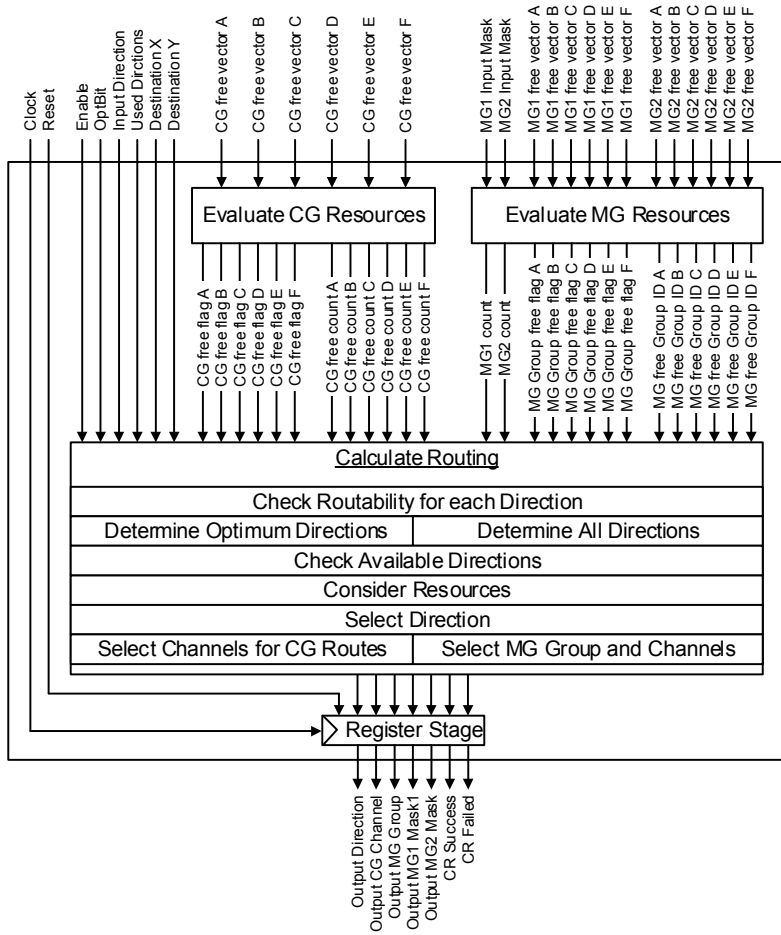



**Abbildung 75: Logischer Aufbau des Clacukate Route Moduls zur Berechnung der Routing-Richtung und Reservierung notwengier CG und MG Ressourcen**

Im nächsten Schritt werden für die verbliebenen Richtungen die Anzahl freier Ressourcen betrachtet und daraufhin die Richtung mit den meisten freien Links ausgewählt. Dies geschieht zunächst für die optimalen Richtungen, anschließend für die verbliebenen, falls das OptBit nicht gesetzt ist. Die derzeitige Version des RoutCtrl betrachtet lediglich die Auslastung der CG-Ressourcen. Auf der MG-Seite wird mindestens eine freie Gruppe für das

Routing vorausgesetzt. Im letzten Schritt wird in Abhängigkeit der ausgewählten Richtung ein freier CG-Link und falls notwendig eine freie MG-Gruppe selektiert.

Die Ausgangsdaten werden abschließend im Ausgangsregister gespeichert. Nach der Berechnung erhält RoutCtrl-FSM eine Reihe von Signalen, die folgende Informationen enthält: Ausgangsrichtung (OutputDirection), CG-Ausgangslink (OutputCGChannel), MG-Gruppennummer (OutputMGGroup), MG1/MG2 Gruppenmasken (OutputMG1/MG2Mask), Statusbit zum Erfolg der Berechnung (CRSuccess) und Fehlerbit (CRFailed). Das letzte Signal wird genutzt, falls keine Richtung für das Routing gefunden wurde. Dies kann passiert, wenn bereits alle Richtungen geprüft wurden, keine freien Ressourcen oder keine optimale Richtung mit freien Ressourcen vorhanden ist.

#### 4.3.4.7.2. Detect-Request-Module

Zur Selektion der eingehenden Anforderungen wird im RoutCtrl das Detect-Request-Modul (DRM) verwendet, um eine priorisierte Auswahl der Anforderungen zu realisieren, siehe Abbildung 76. Auf eine Round-Robin-Steuerung wurde hier verzichtet, da die überschaubare Anzahl der eingehenden CG-Links in Kombination mit verschiedenen Instruktionstypen und möglicher Prioritäten bereits starke Entscheidungskriterien liefert, so dass keine Anforderung in diesem Kontext aushungern wird. Abgesehen davon, sind die Routing-Anforderungen keine Vorgänge die eine hohe Frequenz aufweisen, so dass jede RU genügend Zeit besitzt, um alle Anforderungen zeitnah zu verarbeiten.

Alle ankommenden Signale von InRegCG Modulen werden zunächst direkt an das DRM angeschlossen und hier verarbeitet. Dies schließt ein ein-bit-breites Request-Signal (InRegRequestA-F), die Routing-Instruktion (RICG oder RIMG2) und die Information über bereits geprüfte Richtungen ein (InRegCG UsedDirsA-F) ein. Die verwendeten Informationen auf den Routing-Instruktionen beschränken sich auf die Instruction-ID und die Priorität, die in 'GetRequestA-F'-Modulen seitenweise und für alle Links zugleich extrahiert werden. Dies ist ein formaler Vorgang der in Abhängigkeit der globalen RTL-Parameter die notwendigen Informationen aus den Vektoren rauszieht und dem Selektionsmodul im nächsten Schritt zur Verfügung stellt.

Das Selektionsmodul schaut sich der Reihe nach die Richtungen A-F an und vergleicht die Instruction-IDs und Prioritäten. Hierbei gibt es ein festes Schema, dass die ausgewählte Reigenfolge klar festlegt. Erstes Kriterium hier ist die Instruktion selbst. EPI Instruktionen für MG und CG Routen werden am höchsten priorisiert, da sie wichtige Ressourcen für weitere Routing-Prozesse freigeben. Danach werden die MG-Routing-Anforderungen verarbeitet, da sie temporär CG-Links nutzen und anschließen wieder freigeben. Zum Schluss

werden die CG-Routing-Anforderungen selektiert. In den letzten beiden Fällen können zusätzlich Prioritäten zur weiteren Unterteilung herangezogen, jedoch nicht Instruktionsübergreifend angewandt werden.

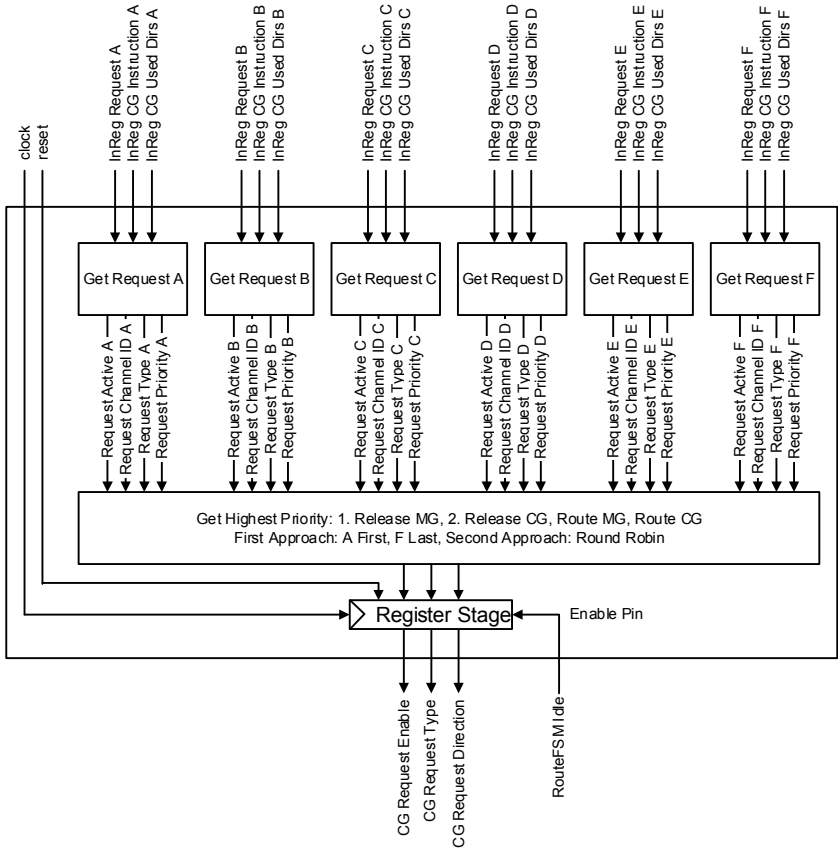



**Abbildung 76: Detect-Request-Module zur Priorisierung und Selektion eingehender Anforderungen von externen CG-Links**

Alle diese Vorgänge sind parallel in einer kombinatorischen Netzliste umgesetzt und benötigen einen Takt zur Verarbeitung. Wenn die RoutCtrl-FSM im 'Idle' Zustand ist, dann wird der Inhalt des Ausgangsregisters bei einer Änderung der Anforderungen erneuert, andernfalls bleiben die Werte bis zum Ende der FSM-Aktivitäten erhalten. Über das Signal 'CG Request Enable' melden das DRM der RoutCtrl-FSM. dass eine aktive Anforderung vorliegt. 'CG Request Type' hilft der FSM rasch in den richtigen Zustand zu wechseln und mittels 'CGRequestDirection' und 'CGRequestLink' die Multiplexer-Stufen

für die Anforderungen zu schalten, damit diese im nächsten Takt zur Verfügung stehen.

Die Anforderungsauswahl bei OutRegCG Modulen ist im Vergleich zur DRM Realisierung einfach gehalten. Hier werden alle Anforderungen per Round-Robin Verfahren der Reihe nach bedient. Das liegt vor allem in der Tatsache begründet, dass hier keine EPI-Anforderungen vorliegen können und die Unterscheidung zwischen CG- und MG-Routing-Anforderungen nicht notwendig ist. Abgesehen davon sind die Aktivitäten von OutRegCG ausgehend als sehr niedrig einzustufen, da die meisten Vorgänge über InRegCG Module initiiert werden. Gegenüber InRegCG Routing-Anforderungen haben OutRegCG-Anforderungen Priorität und werden zuerst von der RoutCtrl-FSM bedient. EPI-Anforderungen haben in beiden Fällen Vorrang.

#### 4.3.4.7.3. Routing FSM

Die FSM des RoutCtrl ist die Hauptkomponente in der Behandlung der anliegenden Anforderungen seitens der InRegCGs und OutRegCGs. Sie ist dafür verantwortlich eingehende Anforderungen zu untersuchen und die notwendigen Aktionen anzustoßen, um die Verarbeitung korrekt auszuführen. So werden beispielsweise Initiale-Routing-Anforderungen direkt verarbeitet, während weitere Versuche zunächst das Löschen bestehender Verbindungen voraussetzen. Ebenso müssen Fehlersituationen während des Betriebes erkannt und dem globalen Routing-Controller gemeldet werden. Für ihren Betrieb besitzt die RoutCtrl-FSM keinerlei Informationen über den historischen Verlauf der vergangen Aktionen, sondern bekommt diese Informationen von InRegCGs oder OutRegCGs geliefert, welche dezentral dieselben speichern. Lediglich die Koordinaten der aktuellen Zelle sind im RoutCtrl hart-kodiert, so dass die Routing-Berechnungen durchgeführt werden können. Nachfolgend wird die FSM-Funktionalität im Detail erläutert. Allerdings wird an dieser Stelle darauf verzichtet, jedes Detail und Bedingung der FSM exakt zu beschreiben, um dem Leser ein besseres Verständnis zu liefern. So sind bereits die Aktionen in der FSM in der Abbildung 77 abstrakt formuliert.

Die FSM beginnt ihre Arbeit im 'Idle' Zustand, wo sie auf Anforderungen von InRegCGs oder OutRegCGs wartet. Beim Eintreffen einer CG-Routing-Anforderung von InRegCGs ('CG_DP_active' = '0' und 'CG_request' = '1') wechselt die FSM in den 'RoutCG' Zustand. Hier wird zunächst geprüft, ob die Zielzelle bereits erreicht ist. Falls nein, wird zunächst die Routenberechnung über das CRM angestoßen. Einen Takt später liegt das Ergebnis der Berechnung vor und die FSM prüft den Modus der Routing Instruktion. In Abhängigkeit des Modus generiert nun die FSM unterschiedliche Antworten an das InRegCG Modul und wechselt in den Zustand 'Wait'. Nachstehende Tabelle

gibt eine Übersicht möglicher Antworten an InRegCG Module, wobei nur die Antworten 1, 2, 3, 4 und 7 aus diesem Zustand heraus möglich sein:

**Tabelle 13 Mögliche Antworten der RoutCtrl-FSM an InRegCG Module**

| | Signal | Beschreibung |
|---|---|---|
| 1 | **RoutCtrl_PathCG_Cfg** | Eine Konfigurationsverbindung hergestellt |
| 2 | **RoutCtrl_PathCG_OK** | Eine CG-Datenverbindung hergestellt |
| 3 | **RoutCtrl_CheckPathCG** | Versuch zum Etablieren einer MG-Verbindung |
| 4 | **RoutCtrl_RdyAndBypass** | Schalte in den Bypass-Modus |
| 5 | **RoutCtrl_PathMG_OK** | Eine MG-Datenverbindung hergestellt |
| 6 | **RoutCtrl_CheckPathMG** | Versuch zum Etablieren einer MG-Verbindung |
| 7 | **RoutCtrl_Refuse** | Routing nicht möglich, Backtracking zum Vorgänger |

Detektiert die FSM eine ankommende Routing-Verbindung zu dieser Zelle, so prüft sie ob der Konfigurationsport der FU frei ist und stellt eine Verbindung her. Im Fehlerfall wird der Fehlercode 'RCEC_UsingBusyInputCG' an den globalen Controller generiert und in den Zustand 'Error' gewechselt, welcher nur durch ein Reset verlassen werden kann. Ein weiterer Fehlerfall liegt vor, wenn bei einer IOHC versucht wird den Konfigurationsport anzusteuern, da es in diesem Fall keinen gibt. Hier wird der Fehler 'RCEC_NotRoutAbleCG' erzeugt. Im Falle eines Erfolges melden die FSM Antwort 1 an das ausgewählte InRegCG und geht in den 'Wait' Zustand. In allen Fällen verweilt die FSM nur einen Takt im 'Wait' Zustand, um dem DRM zu signalisieren, dass die FSM für weitere Anforderungen bereit ist. Auf diese Weise wird die FSM beim Eintreffen im 'Idle' Zustand bereits die nächste Anforderung vorfinden, falls eine anliegt.

Im Falle einer bestehenden Konfigurationsverbindung ('cfg_mode' = "11"') muss die RICG Instruktion an die OutRegCGs kopiert werden. Hier wird zusätzlich geprüft, ob der gewünschte Ausgang in Benutzung ist und ggf. eine Fehlermeldung generiert (RCEC_UsingBusyOutputCG). Andernfalls wird das OutRegCG aktiviert, speichert die neue Instruktion, Ausgangsmultiplexer der Switch-Matrix wird programmiert und ein einfaches 'Acknowledge' an das InRegCG gesendet.

Weitere Antworten aus dem 'RoutCG' Zustand schließen normale Datenverbindungen ein, die im Erfolgsfall beim Erreichen der Zielzelle mit 'RoutCtrl_PathCG_OK' beantwortet werden. Bei erfolgreicher Durchleitung antwortet die FSM mit 'RoutCtrl_CheckPathCG', da zu diesem Zeitpunkt noch nicht abzusehen ist, ab die gewählte Richtung endgültig ist.

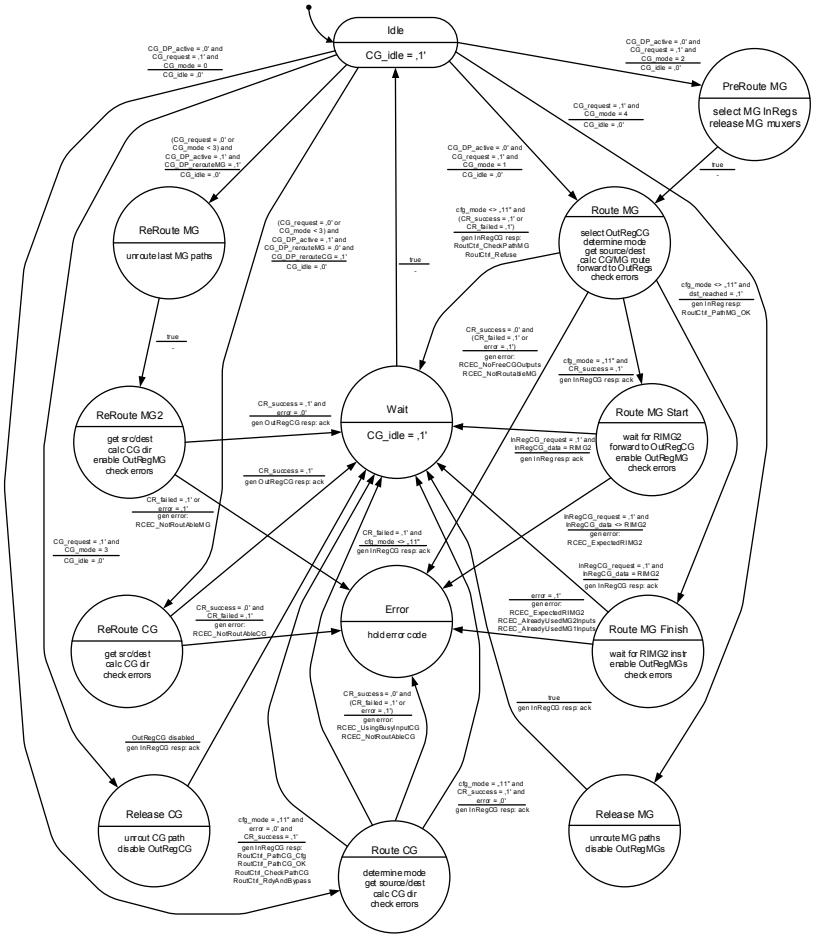



**Abbildung 77: RoutCtrl-FSM zur Steuerung und Koordination der Routing-Prozesse**

CG-Anforderungen an die FSM von OutRegCGs kommend veranlassen die FSM zum Wechseln vom 'Idle' in den 'ReRoutCG' Zustand. Der Name des Zustands deutet an, dass diese Anforderung bereits wiederholt wird, was im Falle der OutRegCG-Anforderungen immer der Falls ist. Nach der Berechnung der nächsten Routing-Richtung, schalten der Switch-Matrizen und mit dem Senden einer Antwort an das OutRegCG, wechselt die FSM in den 'Wait' Zustand. Sollten bei der Berechnung der neuen Routing-Richtung Fehler auftreten, so geht die FSM in den 'Error' Zustand und melden einen Fehler (RCEC_NotRoutAbleCG) an den globalen Controller. Da OutRegCGs die

Quelle für Routing-Verbindungen sind, kann sich eine RU von einem derartigen Fehler nicht erholen.

Soll eine bestehende CG-Verbindung gelöscht werden, so wechselt die FSM vom 'Idle' in den 'ReleaseCG' Zustand, sobald eine EPI-Anforderung für CG-Verbindungen detektiert wird. In diesem Fall wird die Anforderung an das adressierte OutRegCG durchgestellt und in den 'Wait' Zustand gewechselt. Fehler werden in diesem Fall nicht generiert.

Bei MG-Routing-Prozessen ist die Steuerung seitens der RoutCtrl-FSM deutlich komplizierter. Durch die Tatsache, dass in diesem Fall zwei Instruktionen benötigt werden, erhöht sich bereits der Aufwand zur Realisierung. Wird im 'Idle' Zustand der FSM ein RIMG1-Request detektiert, so wechselt die FSM in den 'RoutMG' Zustand. Ähnlich wie im 'RouteCG' Zustand müssen nun mehrere Fallunterscheidungen zum weiteren Vorgehen gemacht werden. Handelt es sich um eine Durchleitung, so wird die neue Ausgangsrichtung für CG und MG Links berechnet, die Switch-Matrizen für alle Datentypen geschaltet, für InRegCG und InRegMG Module Antworten generiert und in den Zustand 'Wait' gewechselt.

Im Falle einer Anforderung von einer Konfigurationsverbindung, wird ein neuer Routing-Prozess initiiert. Dafür werden ebenfalls die neuen Ausgangsrichtungen berechnet, Switch-Matrizen geschaltet und die RIMG1 Instruktion an das gewählte und im weiteren Verkauf assistierende OutRegCG gesendet. Die zur routenden MG-Gruppe gehörenden OutRegMG Module werden zusätzlich in den 'Transfer' Zustand versetzt und können Daten übertragen. Die RoutCtrl-FSM wechselt daraufhin in den 'RouteMGStart' Zustand und wartet auf die Übertragung der RIMG2 Instruktion vom selektierten InRegCG zum OutRegCG Modul. Wird hier keine RIMG2 Instruktion bei Empfang detektiert, so wird ein Fehler (RCEC_ExpectedRIMG2) an den globalen Controller gesendet. Andernfalls wechselt die FSM in den 'Wait' Zustand und schließt den Vorgang ab. Im 'RouteMG' Zustand können ebenfalls zwei Fehler erzeugt werden, falls kein freies OutRegCG zur Verfügung steht (RCEC_NotFreeCGOutputs) und bei ungültiger Adressierung der MG-Gruppen oder fehlgeschlagenem Routing-Versuch (RCEC_NotRoutableMG).

Werden Anforderungen von InRegCG wiederholt, d.h. der Vektor 'UsedDirs' signalisiert in diesem Fall bereits geprüfte Richtungen, so wechselt die FSM in den 'PreRouteMG' Zustand. Hier veranlasst die FSM zunächst beteiligte InRegMG Module aufgebaute Verbindungen in den MG-Switch-Matrizen zu löschen und in den Zustand 'RoutMG' zu wechseln. Ab hier entspricht der Verlauf dem MG-Routing-Prozess, wie oben beschrieben.

Ist bei einer MG-Routing-Anforderung die Zielzelle erreicht, so wechselt die FSM von 'RouteMG' ausgehend nicht in den 'Wait' Zustand, sondern in den 'RoutMGFinish' Zustand. Denn während durchleitende RUs sich nicht um den

Transfer der RIMG2 Instruktionen kümmern müssen, muss die Zielzellen-RU aktiv an diesem Vorgang teilnehmen. Im 'RoutMGFinish' Zustand wartet die FSM den Transfer der RIMG2 Instruktion ab und aktiviert danach die Ausgangsports der RU in Richtung der FU, denn die Beschreibung der MG-Zielgruppe erfolgt in der RIMG2 Instruktion. In diesem Zustand kann die FSM mehrere Fehler erzeugen. Beim Empfang einer anderen Instruktion als RIMG2 melden die FSM den Fehler 'RCEC_ExpectedRIMG2' an den globalen Controller. Sind entweder adressierte MG1- oder MG2-Links nicht frei, so melden die FSM entweder 'RCEC_AlreadyUsedMG1Inputs' oder/und 'RCEC_AlreadyUsedMG2Inputs'. Bei erfolgreichem Abschluss der RIMG2-Übertragung wechselt die FSM in den 'Wait' Zustand und meldet dem In-RegCG den erfolgreichen Abschluss mit einem einfachen 'Acknowledge'.

**Tabelle 14 Fehlercodes der RoutCtrl-FSM für den globalen Controller**

| | Fehler | Beschreibung |
|---|---|---|
| 1 | **RCEC_NotRoutAbleCG** | Keine CG-Routing aus der Zelle heraus möglich |
| 2 | **RCEC_NoFreeCGOutputs** | Alle OutRegCG Module sind belegt |
| 3 | **RCEC_NotRoutAbleMG** | Keine MG-Routing aus der Zelle heraus möglich |
| 4 | **RCEC_ExpectedRIMG2** | RU erwartet eine RIMG2 Instruktion, empfängt stattdessen falsche Instruktion |
| 5 | **RCEC_UsingBusyOutputCG** | Ein aktives OutRegCG wird erneut durch eine RICG Instruktion andressiert |
| 6 | **RCEC_UsingBusyInputCG** | Ein aktiver CG-Eingang der FU wird erneut durch eine RICG Instruktion adressiert |
| 7 | **RCEC_AlreadyUsedMG1Inputs** | Ein aktiver MG1-Eingang der FU wird erneut durch eine RIMG2 Instruktion adressiert |
| 8 | **RCEC_AlreadyUsedMG2Inputs** | Ein aktiver MG2-Eingang der FU wird erneut durch eine RIMG2 Instruktion adressiert |

Eine weitere Möglichkeit zum Anstoßen der MG-Routing-Prozesse besteht in der Widerholungsanfrage vom assistierenden OutRegCG kommend. Dabei wechselt die FSM vom 'Idle' Zustand in den 'ReRoutMG' Zustand. Hier löscht die FSM zunächst bestehende Verbindungen in den MG-Switch-Matrizen und wechselt zum nächsten Zustand (ReRouteMG2). Nach erfolgreicher Berechnung der nächsten Ausgangsrichtung, werden die OutRegCG Module mit aktualisierten Gruppen- und Mask-Informationen versorgt, OutRegMG Module aktiviert und die CG/MG-Switch-Matrizen entsprechend der Routing-Berechnungen geschaltet. Die FSM wechselt dabei in den 'Wait' Zustand. Im

Fehlerfall bei der Routenberechnung wird beim Wechsel in den 'Error' Zustand der Fehler 'RCEC_NotRoutAbleMG' gemeldet.

Die letzte Funktion der FSM erlaubt bestehende Verbindungen von OutRegMG Modulen ausgehend aktiv zu löschen. Beim Eintreffen einer EPI-Anforderung für MG-Verbindungen, wechselt die FSM in den 'ReleaseMG' Zustand. Hier wird den durch EPI adressierten MG-Links ein lokales oder globales Löschsignal geschickt. OutRegMG löschen daraufhin ihre Verbindungen und gehen in den 'Idle' Zustand. Die RoutCtrl-FSM bestätigt diesen Vorgang dem aktiven InRegCG und geht in den 'Wait' Zustand über.

Die ursprüngliche Implementierung der RU, insbesondere des RoutCtrl, erlaubte den Aufbau einer Verbindung innerhalb einer Zelle in drei Takten. Während der Optimierung des ASICs wurde diese Zahl auf vier erhöht, indem eine weitere Registerstufe in das CRM eingefügt wurde. Dieser Vorgang hatte zusätzliche Flexibilität beim Place-And-Route des ASICs gewährt. Im Falle des MG-Routing-Prozesses, insbesondere bei erneuten Routing-Versuchen, erhöht sich die Dauer um einen zusätzlichen Takt, da hier ein separater Löschvorgang für die MG-Switch-Matrizen notwendig ist. Die ist der Fall, da MG-Verbindungen weitgehende passiv durch die Assistenz der parallel verlaufenden CG-Verbindung aufgebaut werden und von den eigentlichen Vorgängen des Routing-Prozesses nicht direkt tangiert sind.

Die finale ASIC-Implementierung enthält nur die Funktion zum lokalen Löschen der CG/MG Verbindungen. Auf die Implementierung der globalen Löschfunktionen wurde in diesem Zusammenhang aus Kostengründen verzichtet. Da diese Funktion im Betrieb der HoneyComb-Architektur nur unzureichend getestet wurde, wird in dieser Ausarbeitung auf detaillierte Beschreibung weitgehend verzichtet.

## 4.4. Datapath-Functional-Unit (DPFU)

Functional Units (FUs) in der Ausprägung zur arithmetischen und logischen Datenverarbeitung werden bei der HoneyComb-Architektur Datapath-Functional-Units (DPFUs) genannt. Die komplette Zelle heißt in diesem Zusammenhang Datapath-HoneyComb-Cell (DPHC). Diese Zellen enthalten eine Reihen von HoneyComb-ALUs (HCALUs), HoneyComb-LUTs (HCLUTs), grobgranulare 32-bit- und feingranulare 1-bit- Register, FG2CG- und CG2FG-Converter, CG/FG Brancher und ein Konfigurationsnetzwerk, siehe Abbildung 79. Dadurch besitzen diese FUs sowohl eine feingranulare als auch eine grobgranulare Domäne und können arithmetische wie auch logische Operationen ausführen. Im Gegensatz zur RU-Eben finden sich in der DPFU nur rein feingranulare Signale, die nach außen durch Splitter- und Merger-Module in MG1/MG2-Signale überführt werden.

Alle Datenübertragungen innerhalb der DPFUs sind durch ein Handshake-Protokoll gesichert, so dass keine Daten ungewollt gelöscht oder verloren gehen können. An einem Kontenpunkt wie einer ALU wird zu diesem Zweck gewartet bis alle Eingänge gültig sind und erst dann der Ausgangwert erzeugt und kontrolliert weitergereicht.

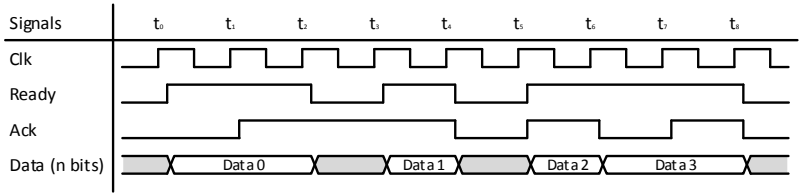



**Abbildung 78: Handshake-Protokol innerhalb der DPFUs und MEMFU**

Neben den Datenleitungen setzt sich das Handshake-Protokoll innerhalb der DPFUs aus zwei zusätzlichen Signalen zusammen: Ready und Ack, siehe Abbildung 78. Das Ready-Signal wird vom Sender erzeugt und signalisiert den Zeitpunkt, wann der Sender bereit ist seine Daten abzugeben. Das Ack-Signal kommt vom Empfänger und deutet seinerseits an, wenn die Übermittlung erfolgen kann. Das Besondere an diesem Protokoll ist der Umstand, dass beide Signale je nach Implementierung gepuffert oder ungepuffert sein können. Die eigentliche Übermittlung erfolgt erst zu dem Zeitpunkt, wenn beide Signale auf eins gesetzt sind. Innerhalb der FUs findet dieses Protokoll sowohl bei CG wie auch FG-Signalen seine Anwendung. Das Bedeutet allerdings auch, dass der zusätzliche Aufwand zur Realisierung des Handshake-Protokolls bei FG-Signalen enorm ist.

### 4.4.1. Grundlegende Struktur und Funktion

Der grundlegende Aufbau der DPFUs wird in der Abbildung 79 veranschaulicht. Hier finden sich unterschiedliche Module, die es erlauben sowohl grobgranulare (recht Seite) als auch feingranulare (linke Seite) Daten zu verarbeiten. Die genaue Anzahl verwendeter Komponenten wird bei der HoneyComb-Architektur erst zum RTL-Zeitpunkt festgelegt und kann entweder vom Entwickler oder anwendungsgetrieben definiert werden.

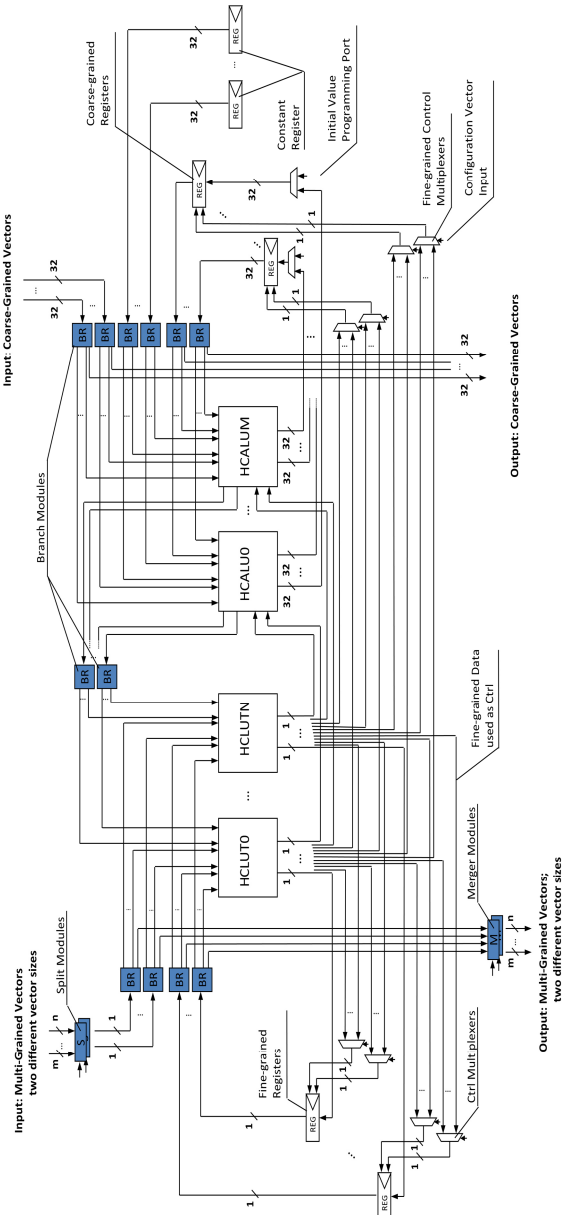



**Abbildung 79: Generisches Blockdiagramm der DPFU mit CG/FG Domänen und funktionalen Einheiten**

Die Struktur in Abbildung 79 besitzt einen generischen Charakter und drückt den flexiblen Aufbau der Architektur aus. Buchstäblich jede Komponenten in diesem Modul kann hinsichtlich ihrem innerem Aufbau und der Anzahl ihrer Instanzen innerhalb der DPFU parametrisiert werden. Insbesondere das Kommunikationsnetzwerk zwischen den Modulen ist sehr flexibel und jede einzelne Verbindung kann hinzugefügt oder entfernt werden. Die grundsätzliche Struktur wie sie in dieser Abbildung dargestellt ist, wird jedoch in jedem Fall erhalten. Doch wie funktioniert der grundlegende Ablauf einer Konfiguration auf einer derartigen Struktur?

Nach dem Neustart der HoneyComb-Architektur sind zunächst alle DPFUs ohne jegliche Funktion und obendrein mittels Clock-Gating inaktiv. Soll eine DPFU eine Applikation ausführen, muss sie zunächst konfiguriert werden. Hierfür muss von IOHC kommend eine Konfigurationsverbindung zur dieser Zelle aufgebaut und eine passende Konfiguration übertragen werden.

Die Programmierung kann bei DPFUs zweidimensional erfolgen: strukturell und zeitlich. Dafür besitzen insbesondere HCALUs optional mehrere Kontexte, die von Takt zu Takt durch die Ansteuerung aus den HCLUTs oder FG-Registern ihre Operationen wechseln können. In ähnlicher Weise lassen sich Operationsmodi von CG/FG Registern umschalten und optional Werte protokollgetrieben verbrauchen, halten oder explizit löschen. Der komplette Umfang der Funktionen dieser Module wird in folgenden Abschnitten erläutert. Zusätzlich zur zeitlichen Programmierung kommt die strukturelle hinzu, die durch möglichst viele implementierte Komponenten eine hohe Parallelität bei der Verarbeitung bietet.

Der feingranulare Teil der DPFU eignet sich, um direkt logische Operationen auszuführen oder komplexere FSMs zu konfigurieren. Eingaben dieser Operationen können von extern über das MG-Netzwerk der RUs kommen oder direkt aus dem grobgranularen Teil stammen. In diesem Fall können HCALUs Status-Flags wie Zero, Overflow oder Sign liefern oder aber durch Bit-Extraktion aus 32Bit-Vektoren bestimmte Bits gewonnen werden.

Eingehende CG-Daten lassen sich zunächst über Brancher an bestimmte HCALUs weiterleiten und führen die ersten Operationen aus. Das Ergebnis der Berechnungen wird in gewünschten Registern gespeichert und steht entweder für weitere Kalkulationen dieser Zelle bereit oder kann an andere Zellen gesendet werden. Dabei kann dieser Wert nur einmal genutzt oder durch die Steuerung des Registers mehrfach verwendet werden. Zusätzlich stehen spezielle Register für Konstanten (ConstantRegister) bereit, die Koeffizienten oder einfache Konstanten für Applikationen bereithalten können. Diese Werte können von der Applikation nicht überschrieben werden und stehen für die gesamte Dauert der Applikationsausführung zur Verfügung.

### 4.4.2. Universelle Module

Um die Komplexität der HoneyComb-Architektur beherrschbar zu halten, wurden im Laufe der Entwicklung universelle Module entwickelt, die möglichst oft an verschiedenen Stellen der Architektur Verwendung fanden. Dies betrifft insbesondere das Handling von Protokollen an Kontenpunkten, Signalverzweigungen und Multiplexing von bidirektionalen Signalen mit speziellen Funktionen, die im Weiteren erläutert werden.

#### 4.4.2.1. Protocol-Handler

Durch die Nutzung des Handshake-Protokolls nach Abbildung 78 ist es notwendig an Knotenpunkten, wie sie im Falle der ALUs oder Verzweigungen derartiger Signale auftreten, durch eine entsprechende Behandlung beteiligter Signale die korrekte Übertragung sicherzustellen. Im Falle der ALUs ist es beispielsweise notwendig auf die Validität aller aktiver Eingänge zu warten ('rdy' = '1'), um einen korrekten Ausgang an die folgenden Module zu signalisieren. Ebenso werden die Eingänge erst übernommen, d.h. mit 'ack' = '1' bestätigt, wenn der Ausgang dies auch tut und somit ein Transfer stattfinden kann. Gleichzeitig werden die Daten durch die ausgewählte Operation ALU und das nachfolgende Modul weitergegeben.

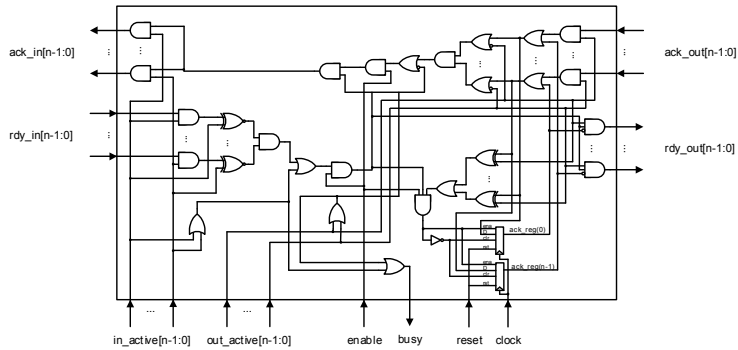



**Abbildung 80: Schaltung eines einfachen Protocol-Handlers mit aktivierbaren Ein- und Ausgängen sowie Enable und Busy Signalen**

Bei Verzweigungen von Signalen darf der Eingang erst bestätigt werden, wenn alle Verzweigungen und damit Ausgänge dies ebenso tun oder bereits getan haben. D.h. in diesem Fall muss eine Speicherfunktion dafür sorgen, dass bereits aktivierte Ausgänge vermerkt werden und sobald alle Ausgänge aktiv waren der Transfer eingangsseitig bestätigt wird. Hier haben wir ein Szenario mit einem Eingang und mehreren Ausgängen, während im Falle der

ALU ein Ausgang und mehrere Eingänge benutzt werden. Auch sind in diesem Zusammenhang Variationen denkbar, die beide Fälle kombinieren, so dass mehrere Eingänge und Ausgänge genutzt werden.

Abbildung 80 zeigt die Schaltung eines Protocol-Handlers, welcher in den beschriebenen Szenarien angesetzt wird. Zur Handhabung des Protokolls werden eingangsseitig 'rdy_in' und 'ack_in' Signale genutzt, während auf der Ausgangseite 'rdy_out' und 'ack_out' Signale bereit stehen. Die jeweilige Anzahl der Ein- und Ausgänge ist in diesem Modul parametrisierbar. Zu Laufzeitaktivierung der Ein- und Ausgänge werden 'in_active' und 'out_active' Vektoren genutzt. Hierbei steht jedes Bit für ein Handshake-Signal, das beim Setzen auf eins aktiviert wird. Damit sind bei komplexeren Modulen die Ein- und Ausgänge beliebig programmierbar, indem beispielsweise Konfigurationsregister die Werte für die 'in_active' und 'out_active' Eingänge liefern. Zusätzlich steht der Eingang 'enable' bereit, um das Modul global zu aktivieren, während der Ausgang 'busy' die Aktivität des Moduls und damit des Knotens anzeigt.

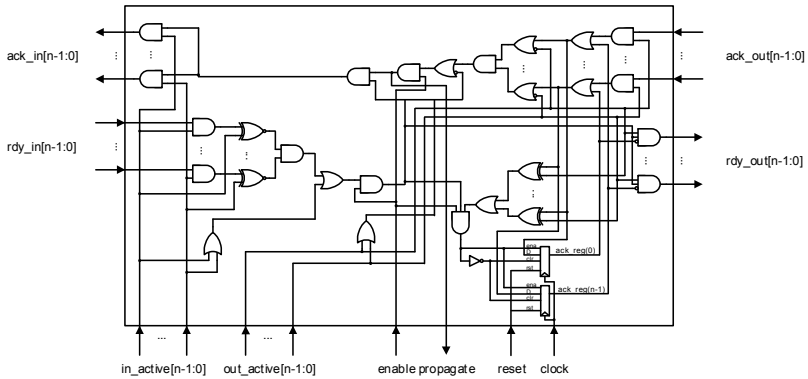



**Abbildung 81: Protocol-Handler-Variante mit einem 'propagate' Ausgang zur Signalisierung eines aktiven Transfers**

Insgesamt wurden vier Varianten des Protocol-Handlers genutzt, um die Funktionen innerhalb der HoneyComb-Architektur zu realisieren. So besitzt eine weitere Variante statt des 'busy' Signals den Ausgang 'propagate' zur Signalisierung der aktiven Transfers durch den Knoten, in dem Moment wenn eingangsseitig die Daten bestätigt werden. Auf diese Weise kann eine Multikontext-Einheit mittels dieses Signals ihren Kontext umschalten und die nächste Funktion für den folgenden Takt auswählen.

Zur Realisierung der globalen Löschfunktion wurde eine dritte Variante des Protocol-Handlers realisiert, siehe Abbildung 82. Zusätzlich zu 'rdy' und 'ack' Signalen bietet sie 'unrout_in' und 'unrout_out' Signale, welche nach Abbildung 47 zum Löschen bestehender Routen genutzt werden. Während diese

Abbildung MG Signale auf RU-Ebene adressiert, wird im Falle der FUs und der globalen Löschfunktion das gleiche Protokoll sowohl für CG wie auch FG Signale genutzt. Es sei an dieser Stelle angemerkt, dass der Demonstrationschip diese Funktion nicht unterstützt, so dass nur die ersten beiden Varianten des Protocol-Handlers zum Einsatz kommen.

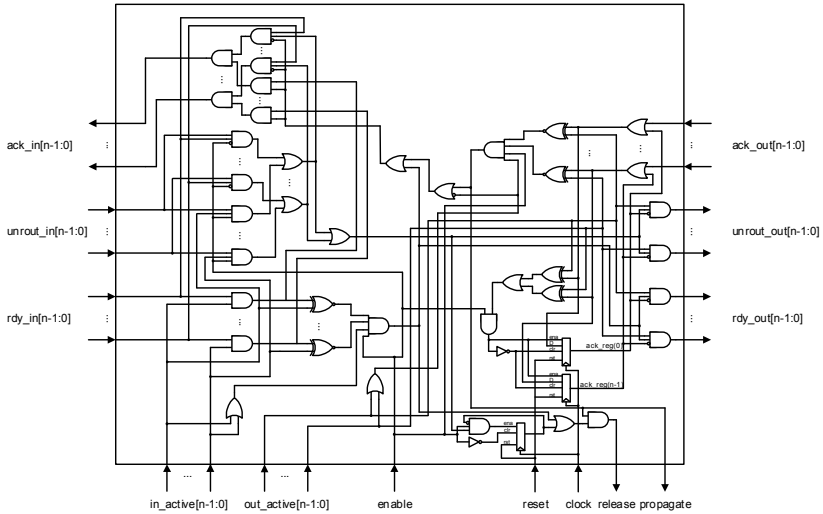



**Abbildung 82: Protocol-Handler mit Unterstützung der Unrout-Funktion zur Realisierung der globalen Routen-Löschfunktion**

Auch hier müssen am Eingang empfangene Unrout-Signale an die Ausgänge propagiert werden. Haben alle nachfolgenden Module den Empfang bestätigt, so wird auch am Eingang der Transfer quittiert und zugleich durch den Ausgang 'release' = '1' das Auflösen der Konfiguration signalisiert. Dadurch kann das Modul, in welchem der Protocol-Handler genutzt wird, die notwendigen Schritte einleiten, um die eigene Konfiguration zu löschen. Die vierte Variante des Protocol-Handlers bietet hier statt des 'propagate' Signals lediglich einen 'busy' Ausgang und ist damit in der Unterscheidung analog zu ersten beiden Varianten zu sehen.

Alle Protocol-Handler bieten eine Reihe von Status und Steuersignalen, die im Einsatz je nach Ausprägung der Zielmodule nicht immer genötigt werden. Für diesen Fall sind Protocol-Handler derart beschrieben, dass sie diese Funktionen rausnehmen und dadurch Ressourcen einsparen. Zusätzlich können Steuersignale wie 'in_active' oder 'out_active' mit Konstanten belegt werden, so dass sie durch Optimierungen in der Synthese weniger Logikelemente benö-

tigen werden. Ebenso kann auf diese Weise bei Bedarf der Eingang 'enable' fest aktiviert werden, falls das Zielmodul diese Funktion nicht benötigt.

#### 4.4.2.2. Brancher-Modul

Basierend auf den Protocol-Handlern werden an Verzweigungen von protokollbasierten Signalen Brancher-Module eingesetzt. Der Kern dieser Module besteht aus einem Protocol-Handler, der die Steuerung der Transfers übernimmt. Abbildung 66 veranschaulichte bereits einen Brancher, der unbedingt ankommende Ack-Signale disjunktiv verknüpft, da hier die Annahme gilt, dass nur ein Ausgang zugleich aktiv sein darf. Diese Annahme trifft innerhalb der FUs nicht zu. Hier kann beispielsweise ein Register seine Werte an mehrere ALUs/LUTs oder FU-Ausgänge weiterleiten und dieser Vorgang muss auf Protokollebene koordiniert werden.

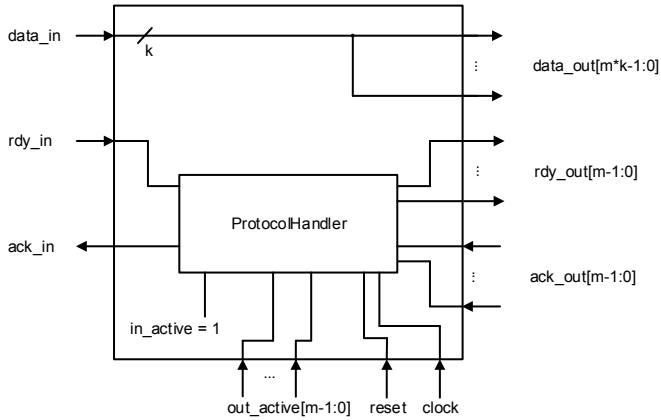



**Abbildung 83: Brancher-Modul zur Verzweigung protokoll-getriebener Signale zur koordinierten und programmierbaren Verteilung**

Der Protocol-Brancher leitet in seiner Funktion eingehende Daten lediglich durch und jeder Ausgang erhält exakt den gleichen Datenwert. Die Vektorbreite m wird im Falle der CG-Signale auf 32 und im Falle der FG-Signale auf eins gesetzt. Die Behandlung des Protokolls erfolgt durch den Protocol-Handler. Eingangsseitig wird nur ein Signal unterstützt, während ausgangsseitig entsprechend der Brancher-Ausführung m Signale unterstützt werden. Der Steuereingang 'in_active' wird konstant auf eins gesetzt, während der Steuereingang 'out_active' durch ein Konfigurationsregister gesteuert wird. Damit lassen sich zur Laufzeit gewünschte Ausgänge aktivieren.

### 4.4.2.3. Multiplexer-Module

Neben den Verzweigungen protokollbasierter Signale müssen auch Multiplexer-Funktionen in diesem Zusammenhang eine besondere Behandlung erfahren. Beim einfachen Durchschalten müssen lediglich eingehende Protokollsignale bidirektional zum Ausgang durchgeschaltet werden. Die übrigen Eingänge senden immer 'ack' = '0' an den Sender und signalisieren damit, dass kein Transfer stattfinden soll. Abbildung 84 (a) zeigt die Realisierung eines bidirektionalen Multiplexer (BiDirMultiplexer), der neben einem einfachen Multiplexer in Vorwärtsrichtung zusätzlich einen Dekoder in die rückführende Richtung besitzt. Der Steuereingang 'ctrl' wird in der Anwendung von einem Konfigurationsregister oder einer Multikontexttabelle belegt und beschreibt welches Eingangssignal an den Ausgang durchgestellt werden soll.

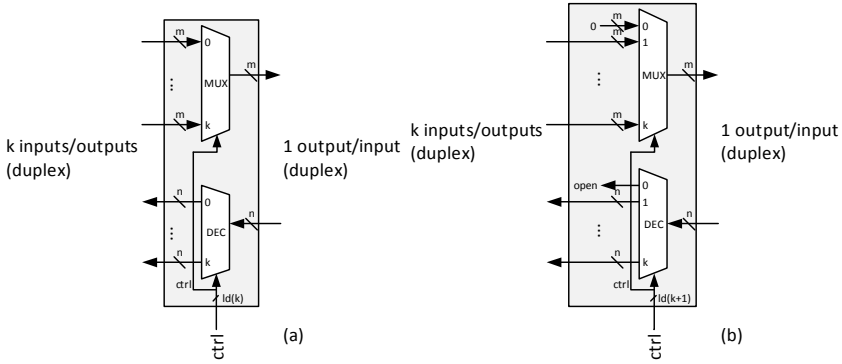



**Abbildung 84: Bidirektionale Multiplexer zur Selektion protokoll-getriebener Signale; a - Standardverhalten, b - mit einer Null-Sperr-Stellung**

Neben der Standardvariante wurde eine weitere Variante mit einer definierten Sperrstellung implementiert, die keine Signale bei 'ctrl' = '0' Belegung durchstellt. Dadurch lässt sich Signal-Gating-Funktionalität im Design nutzen, ohne zusätzliche Gates zu implementieren. Durch die Dualkodierung der Steuersignale der Multiplexer ist die Implementierung auf diese Weise günstiger als durch ein direktes Gate mit einem Steuerbit.

In der Nutzung der Multiplexer in Kombination mit protokollgetriebenen Signalen lassen sich leicht weitere Möglichkeiten des Einsatzes vorstellen. So ist es in bestimmten Anwendungsszenarien von Vorteil bei der Auswahl von einem Eingang eines Multiplexers einen zweiten ebenso zu quittieren, indem dessen Sender ein 'ack' = '1' gesendet wird. In diesem Fall bedeutet dieser Vorgang die Löschung eines Wertes, ohne diesen zu konsumieren. Insbesondere bei bedingten Konstrukten wie if-Statements ist diese Situation regulär und sollte betrachtet werden. Bei Nutzung protokollbasierter Signale ist es in die-

sem Fall notwendig eine separate Behandlung der Signale zu realisieren, wobei bei Bedarf eine gesonderte Antwort für selektierte Eingänge generiert werden soll.

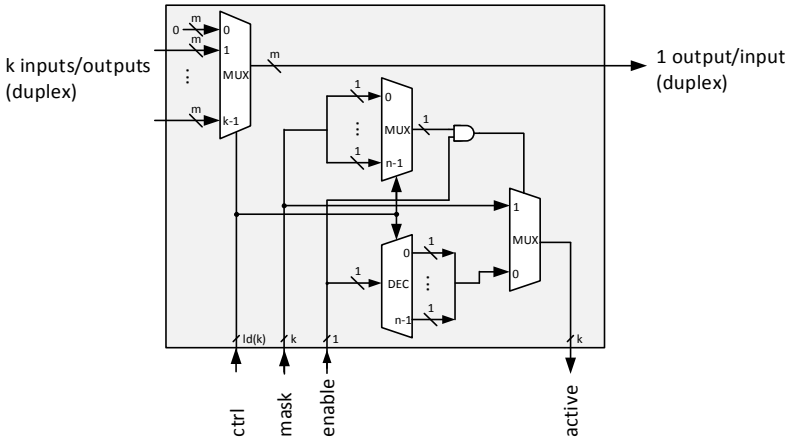



**Abbildung 85: Unidirektionaler Multiplexer mit speziellen Steuersignalen zum Dekodieren, Maskieren und Aktivieren der Protocol-Handler**

Abbildung 85 veranschaulicht eine unidirektionale Multiplexer-Struktur mit zusätzlicher Logik zur Realisierung des beschriebenen Verhaltens. Die Protokollsignale werden hierbei von einem zusätzlichen Protocol-Handler gehandhabt und müssen nicht separat durch einen Multiplexer geführt werden. Der Eingang 'ctrl' beschreibt die dualkodierte Nummer des Eingangs, der durchgestellt werden soll. Mit dem Signal 'mask' wird beschrieben, welcher Eingang beim Transfer quittiert werden soll. Jedes Bit in 'mask' repräsentiert dabei einen Eingang und kann damit direkt adressiert werden. Der Ausgangsvektor 'active' gibt die aktivierten Eingänge wieder und ist dabei ebenso dualkodiert wie 'mask'. In der Anwendung kann 'active' direkt an Protocol-Handler angeschlossen werden. In diesem Szenario würde der Nutzer die Protokollsignale 'rdy' und 'ack' der Ein- und Ausgänge ebenso direkt an einen Protocol-Handler anschließen und dadurch das Handling des Protokolls ermöglichen. Das Modul, welches sowohl den beschriebenen Multiplexer als auch den Protocol-Handler beherbergt, wird hierbei als ein Knoten betrachtet. Der Eingang 'enable' fungiert als ein zusätzlicher Schalter, der die 'active' Ausgabe deaktivieren kann.

Wird nun in der Anwendung der Vektor 'mask' auf null gesetzt oder nur das 'mask'-Bit des aktiven Eingangs auf eins, so verhält sich das Modul wie ein gewöhnlicher Multiplexer. Über den Vektor 'active' wird dem Protocol-Handler signalisiert, welcher Eingang selektiert ist, so dass dieses Signal beim

Transfer quittiert wird. Werden nun mehrere Bits auf eins gesetzt, so hängt das Verhalten davon ab, ob der aktuell ausgewählte Eingangs ebenso durch 'mask' selektiert ist. Wenn nein, so verhält sich dieses Modul wieder wie zuvor beschrieben. Falls jedoch das 'mask'-Bit des Eingangs aktiv ist und damit auf eins gesetzt ist, so werden alle 'active'-Bits der mit 'mask' selektierten Eingänge auf eins gesetzt und beim Transfer quittiert. Währenddessen wird nur der durch den Multiplexer selektierte Eingang zum Ausgang transportiert, während die anderen Daten gelöscht werden und damit verloren gehen. Durch diesen Mechanismus ist es möglich, gezielt Eingänge zu löschen oder für weitere Verarbeitung zu erhalten. Bei Nutzung der Multikontextfunktionalität kann 'mask' für jeden Kontext separat beschrieben werden und somit ein differenziertes Verhalten beschreiben.

### 4.4.3. CG / FG Register

In der DPFU verwendete CG und FG Register sind in ihrer Funktionalität sehr ähnlich aufgebaut. Sie beherbergen zwei oder mehr interne Register, die nach außen wie FIFOs funktionieren. Als besondere Funktion bieten diese Register zwei optionale FG-Eingänge, die als Steuersignale genutzt werden. Das erste Signal wird dabei zum Löschen verwendet (Clear) und löscht den am Ausgang des Registers anliegenden Wert, falls dieses FG-Signal den Wert eins hat, andernfalls hängt das Verhalten vom zweiten FG-Signal ab, welches eine Halten-Funktion (Hold) ermöglicht. Ist dieser Wert nun eine Eins, so wird eine Kopie des aktuellen Wertes an die folgenden Module weitergegeben bei gleichzeitigem Halten des aktuellen Wertes im Register. In allen Fällen werden die beiden FG-Signale beim Registertransport quittiert und damit verbraucht. Die Steuerung bezieht sich dabei immer auf den aktuellen Ausgangswert des Registers. Falls aktuell kein Wert im Register vorliegt, so wartet das Register bis alle Werte (beide FG-Steuersignale, soweit genutzt, und der eigentliche Registerdatensatz) vorliegen und führt dann den Transfer durch. Das beschriebene Verhalten gilt für CG- wie auch FG-Register. Folgende Tabelle fasst den Einfluss der FG-Steuersignale nochmal zusammen:

**Tabelle 15 Steuerung der CG/FG-Register mittels der optionalen FG-Steuersignale**

| | Clear | Hold | Beschreibung |
|---|---|---|---|
| **1** | **1** | - | Ausgangswert wird gelöscht |
| **2** | **0** | 1 | Ausgangswert wird kopiert und transportiert |
| **3** | **0** | 0 | Ausgangswert wird nur transportiert |

### 4.4.4. HCALU Module

Die HoneyComb-ALUs (HCALUs) sind die primären Komponenten zur arithmetischen Datenverarbeitung mit Multikontextunterstützung, siehe Abbildung 86. Die bieten zuallererst eine ALU mit einem vordefinierten Satz arithmetischer Operationen, der zum RTL-Zeitpunkt ausgewählt werden kann. Das Besondere an diesem Modul ist die Kombination mit einer LUT, welche einen oder mehrere Kontexte speichern kann. In Abhängigkeit der eingehenden Werte der HCLUTs oder eigener Flags kann nun von Takt zu Takt der Kontext und damit beispielweise die Operation gewechselt werden. Auf diese Weise kann die ALU auf Ereignisse bei der Applikationsausführung reagieren und zusätzlich die Hardware effizienter ausnutzen.

Neben der Opcode-Selektion können die HCLUTs auch Eingänge und Ausgänge, sprich Zielregister der HCALU, von Takt zu Takt selektieren. Darüber hinaus lässt sich die Zusammensetzung der Adresse für die interne LUT taktweise ändern und so den aktuellen Operationen anpassen.

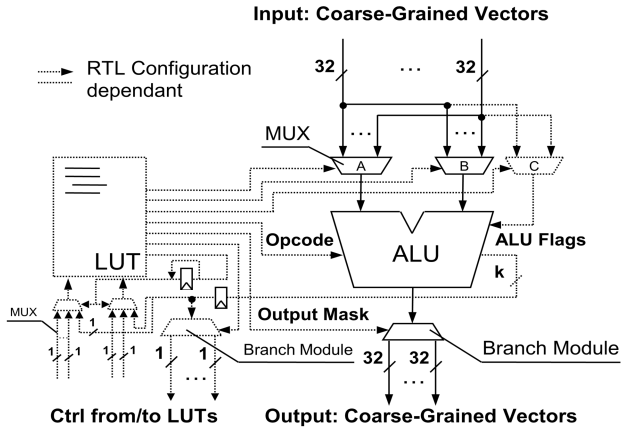



**Abbildung 86: Generisches Blockdiagramm der HoneyComb-ALU (HCALU)**

Zur Steuerung des Handshake-Protokolls wird zusätzlich ein Protocol-Handler innerhalb der HCALU implementiert, in Abbildung 86 nicht dargestellt, welcher alle grobgranularen und feingranularen Ein- und Ausgänge als ein Knoten behandelt. Auf diese Weise muss sich der Programmierer keine Gedanken über den Ablauf des Programms machen, da dies direkt in Hardware protokollgestützt geregelt wird. Der Protocol-Handler wartet mit der Weiterleitung der Daten bis alle Eingänge bereit sind und erst dann werden Ergebnisse an die folgenden grobgranularen/feingranularen Register oder HCLUTs aktiv geleitet.

**Tabelle 16 Verfügbare und auf RTL parametrisierbare HCALU-Operationen**

| | Operation | Beschreibung |
|---|---|---|
| 1 | **add** | Vorzeichenlose Addition |
| 2 | **sub** | Vorzeichenlose Subtraktion |
| 3 | **mul** | Vorzeichenlose Multiplikation |
| 4 | **mulsh** | Vorzeichenlose Multiplikation mit anschließendem Schieben |
| 5 | **div** | Vorzeichenlose Division |
| 6 | **divsh** | Vorzeichenlose Division mit anschließendem Schieben |
| 7 | **sadd** | Vorzeichenbehaftete Addition |
| 8 | **ssub** | Vorzeichenbehaftete Subtraktion |
| 9 | **smul** | Vorzeichenbehaftete Multiplikation |
| 10 | **smulsh** | Vorzeichenbehaftete Multipl. mit anschließendem Schieben |
| 11 | **sdiv** | Vorzeichenbehaftete Subtraktion |
| 12 | **sdivsh** | Vorzeichenbehaftete Subtraktion mit anschließendem Schieben |
| 13 | **fadd** | Fließkommaaddition |
| 14 | **fsub** | Fließkommasubtraktion |
| 15 | **fmul** | Fließkommamultiplikation |
| 16 | **fdiv** | Fließkommadivision |
| 17 | **nota** | Bitweise Negation des Operanden A |
| 18 | **notb** | Bitweise Negation des Operanden B |
| 19 | **notc** | Bitweise Negation des Operanden C |
| 20 | **and** | Bitweise Konjunktion von A und B |
| 21 | **nand** | Bitweise negierte Konjunktion von A und B |
| 22 | **or** | Bitweise Disjunktion von A und B |
| 23 | **nor** | Bitweise negierte Disjunktion von A und B |
| 24 | **xor** | Bitweises Exklusiv-Oder von A und B |
| 25 | **eqv** | Bitweise Äquivalenz von A und B |
| 26 | **slr** | Logisches Schieben nach rechts |
| 27 | **sll** | Logisches Schieben nach links |
| 28 | **sar** | Arithmetisches Schieben nach rechts mit Erhalt des Vorzeichens |
| 29 | **sal** | Arithmetisches Schieben nach links mit Erhalt des Vorzeichens |
| 30 | **ror** | Logisches Rotieren nach rechts |
| 31 | **rol** | Logisches Rotieren nach links |
| 32 | **nop** | Keine Operation |
| 33 | **stoa** | Einfaches Durchleiten von A |
| 34 | **stob** | Einfaches Durchleiten von B |
| 35 | **stoc** | Einfaches Durchleiten von C |

Tabelle 16 gibt eine Übersicht unterstützter Operationen der HCALUs wieder. Dabei muss beachtet werden, dass RTL-seitig eine Untermenge dieser Operationen ausgewählt werden kann und zur Laufzeit zur Verfügung steht. Die interne ALU unterstützt grundsätzlich bis zu drei Operanden und unter-

stützt somit Operationen wie Multiplikation mit anschließendem Schieben, die für Festkommazahlen notwendig sind, um die durch die Multiplikation oder Division entstandene Kommaverschiebung zu korrigieren. Optional sind auch Fließkommazahlen Teil der ALU. Die verwendeten Operationen wurden nicht extra für die HoneyComb-Architektur entwickelt. Es wurde auf angebotene Operationen von Synopsys Design Compiler [57] zurückgegriffen. Der Fokus dieser Arbeit lag vielmehr im Architekturbereich, so dass auf Entwicklung eigener Schaltungen für die Realisierung der Operationen verzichtet wurde. Die unterstützen Flags der ALU sind auf fünf Typen beschränkt und in folgender Tabelle zusammengefasst:

**Tabelle 17 Verfügbare und auf RTL parametrisierbare HCALU-Operationen**

| | Flag | Beschreibung |
|---|---|---|
| 1 | **carry** | Übertrag von einer vorzeichenlosen Addition |
| 2 | **zero** | Das Ergebnis ist eine Null |
| 3 | **overflow** | Vorzeichenabhängiger Überlauf |
| 4 | **sign** | Vorzeichen des Ergebnisses |
| 5 | **parity** | Gerade Parität des Ergebnisses |

Eingänge der HCALUs können optional je nach Anwendungsanforderung Eingangsmultiplexer mit 'mask'-Funktionen nutzen, siehe Abschnitt 4.4.2.3. Auch hier gilt, dass sowohl 'ctrl'- wie auch 'mask'-Steuersignale der Eingangsmultiplexer entweder direkt über Konfigurationsregister oder über die interne LUT angesteuert werden können. Dadurch wird die HCALU in die Lage versetzt, sehr flexibel mit den Eingängen umzugehen und die Steuerung hinsichtlich des Signalflusses selbst zu realisieren. In Kombination mit den CG-Registern und ihren FG-Steuersignalen wird es dadurch möglich vielseitige Szenarien aufzubauen, um verschachtelte 'if-then-else' Konstrukte auf die Hardware abzubilden.

Alle gestrichelten Verbindungen in der Abbildung 86 repräsentieren optionale Funktionen. So lässt sich eine HCALU auf RTL im einfachsten Fall ohne Multikontextunterstützung implementieren, um Ressourcen zu sparen. Oder nur ein Teil der Multikontext-Funktionalität wird benötigt, so dass die LUT ausgewählte Komponenten ansteuert, die restlichen werden über einfache Konfigurationsregister, wie dies in Abbildung 88 dargestellt ist, statisch für die Dauer einer Konfiguration belegt.

### 4.4.5. HCLUT Module

Boolesche Operationen können in DPFUs mittels HoneyComb-LUTs (HCLUT) realisiert werden, siehe Abbildung 87. Auch hier werden einfache Module, in diesem Fall LUTs, durch zusätzliche Beschaltung zu komplexen

Funktionsmodulen erweitert. Eingangssignale bei HCLUTs werden mittels Eingangsmultiplexern den vorgesehenen LUT-Eingängen zugeführt und adressieren dadurch in der LUT gespeicherten Inhalt, um eine vorgegebene Anzahl an booleschen Funktionen am Ausgang zu realisieren. Wie im Grundlagenkapitel erläutern, stellt der Inhalt der LUT eine Wahrheitstabelle dar. Eingehende Signale werden dabei als Adressen genutzt, um auf bestimmte Zellen der LUTs zuzugreifen und das Ergebnis der Funktion auszugeben.

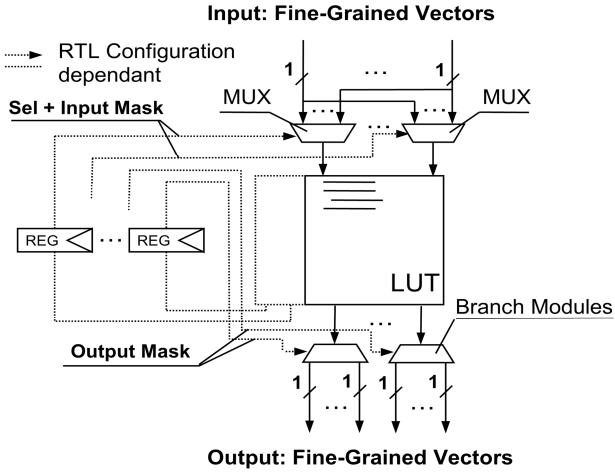



**Abbildung 87: Generisches Blockschaltbild einer HoneyComb-LUT (HCLUT)**

Am Ausgang können optional Ausgangs-Brancher verwendet werden, um die Ausgänge an einen oder mehrere Empfänger wie FG-Zielregister, FG2CG-Module, Steuereingänge der FG/CG-Register oder HCALUs zu senden. Alle Ein- und Ausgangsignale sind gemäß dem Kommunikationsprotokoll der DPFUs abgesichert. Ein zusätzlicher Protocol-Handler in der HCLUT sorgt für reibungslosen Ablauf zwischen den Eingangssignalen und den Ausgangs-Branchern oder direkt den Empfängern im Falle eines einfachen Ausgangs.

Das besondere an den HCLUTs stellen die rückgekoppelten Ausgangsbits der integrierten LUTs dar. Hierbei handelt es sich um zusätzliche Ausgangsbits der integrierten LUT, die über rückgekoppelte Register oder im Bypass-Modus im nächsten Takt oder direkt im gleichen Takt Steuersignale für Eingangsmultiplexer oder Ausgangs-Brancher zur Verfügung stellen zu können. Durch diese Funktionalität ist es möglich kontextsensitiv durch einige wenige Eingangsvariablen den Rest der der Ein-/Ausgänge der booleschen Funktion dynamisch aufzubauen. Folgende Code-Beispiele verdeutlichen den Sachverhalt:

$if\ (a)\ then$ $if\ (a)\ then$

$b = f(c,d)$
*else*
$b = f(c,e)$
*end if*

$b = f(d,e)$
*else*
$c = f(d,g)$
*end if*

Während das Beispiel links auch mittels einer booleschen Funktion realisierbar ist, ist mit einer einfachen LUT das rechte Beispiel schwieriger zu bewerkstelligen. Hier wird in Abhängigkeit einer Bedingung (a) eine der beiden booleschen Funktionen ausgeführt. In realer Anwendung würde das zwei einfache LUTs brauchen oder zumindest zwei separate Ausgänge einer größeren LUT (ab 4:2 LUTs). Bei einer HCLUT kann in diesem Fall ein Ausgang für beide Funktionen genutzt werden. Durch das Handshake-Protokoll muss allerdings explizit definiert sein, was mit ungenutzten Signalen geschehen soll. So muss beispielsweise gesondert eine Variable gelöscht werden, falls diese von der HCLUT nicht gebraucht wird und auch in Zukunft nicht benötigt wird. Dieser Mechanismus wird in ähnlicher Weise wie bei HCALUs durch Eingangsmultiplexer mit der 'Mask'-Funktionalität ermöglicht. Auch hier können die Steuersignale dieser Multiplexer entweder direkt an Konfigurationsregister oder als Feedbacksignale aus der LUT stammen.

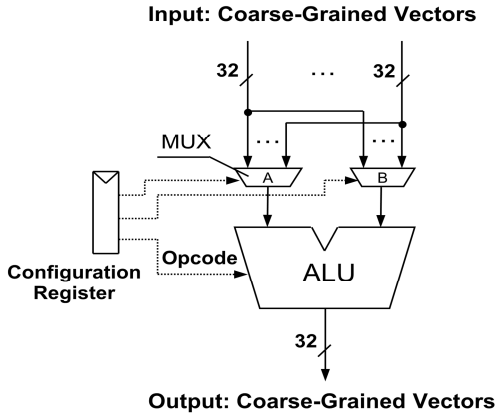



**Abbildung 88: Einfacher Aufbau einer HCALU ohne Multikontextunterstützung**

Wie bei einer HCALU ist auch eine HCLUT stark generisch modelliert und kann auf RTL parametrisiert werden. Gestrichelte Strukturen wie sie in Abbildung 87 dargestellt sind, stellen optionale Ressourcen dar und können bei Bedarf durch Parameter entfernt oder hinzugefügt werden. Im einfachsten Fall erhält der Entwickler dadurch eine einfache HCLUT mit einer Reihe von Eingängen und Ausgängen, die sich wie eine Standard-LUT verhält, siehe Abbil-

dung 88. Das Gegenstück dazu stellt eine HCLUT mit rückgekoppelten Steuersignalen dar, die eine kontextsensitive Realisierung boolescher Funktionen ermöglicht.

### 4.4.6. CG2FG / FG2CG Module

Zum Datenaustausch zwischen den beiden unterschiedlich-granularen DPFU-Domänen wurden zwei weitere Module entwickelt und implementiert. Durch die enorme Größe der CG-Vektoren im Verhältnis zu FG-Signalen, ist an dieser Stelle nur eine Bit-Extraktion und Bit-Reimplantation sinnvoll anwendbar. Im Falle der Extraktion bedeutet das, dass aus einem vorliegenden 32-bit-Vektor bestimmte Bits in FG-Signale gewandelt und der FG-Domäne zur Verfügung gestellt werden. Diese Aufgabe übernimmt das CG2FG-Modul, siehe Abbildung 89.

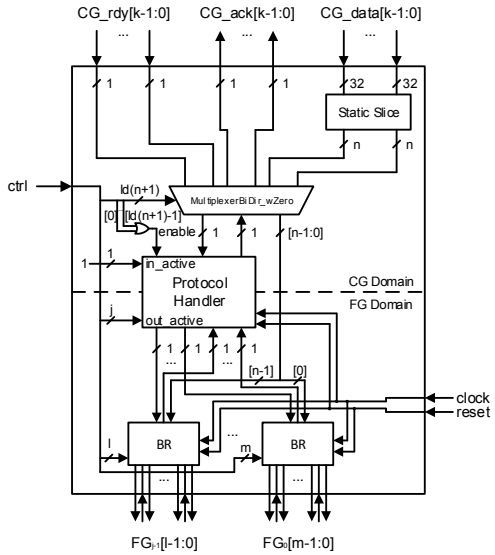



**Abbildung 89: CG2FG-Modul zur Bitextraktion und Transport in die FG-Domäne**

Mehrere eingehende CG-Signale werden hierbei im ersten Schritt durch einen Eingangsmultiplexer selektiert. Im zweiten Schritt werden die ersten n-Bits des 32-bit-Vektors durch den Protocol-Handler um die Handshake-Protokoll-Signale ergänzt und entweder direkt oder über Ausgangs-Brancher an die weiteren Module weitergeleitet. Die Ausgangs-Brancher kommen nur zum Einsatz, falls mehrere Empfänger in der DPFU pro extrahiertem Bit vorhanden sind und dadurch eine Verzweigung notwendig wird.

Das Kontrollsignal 'ctrl' des CG2FG-Moduls beinhaltet zum einen die Daten für die Selektion des Eingangsmultiplexers wie auch die 'in_active' und 'out_active' Steuervektoren für den Protocol-Handler. Falls Ausgangs-Brancher im Modul enthalten sind, so liefert das Signal 'ctrl' auch die Steuerdaten für diese Module. Die Belegung des kompletten Kontrollsignals 'ctrl' erfolgt direkt über ein Konfigurationsregister, wie es im folgenden Abschnitt beschrieben wird.

In umgekehrter Richtung zur Extraktion wird die Bit-Implantation bei Bedarf durchgeführt. Hier werden bestimmte FG-Signale zu einem CG-Vektor zusammengesetzt. Dabei muss der resultierende Vektor nicht die vollen 32-Bits besitzen. Die ungenutzten Bits werden dabei auf Null gesetzt. Zur Realisierung dieser Funktion wird das FG2CG-Modul genutzt, welches in Abbildung 90 dargestellt ist.

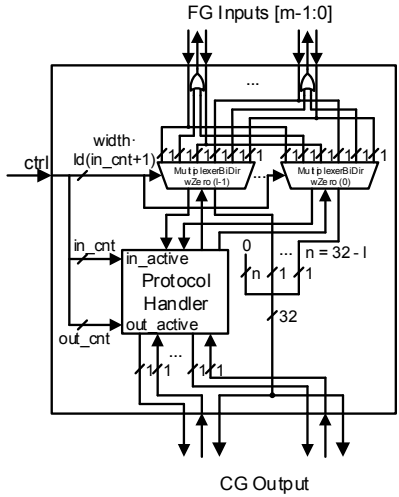



**Abbildung 90: FG2CG-Modul zur Bitreimplantation und Transport in die CG-Domäne**

Die eingehenden FG-Signale können bei Bedarf auch mehrfach implantiert werden. Dafür werden sie jeweils an alle Eingangsmultiplexer angeschlossen und die rücklaufenden 'ack' Signale disjunktiv verknüpft. Die selektierten Signale werden zunächst zu einem Vektor konkateniert und um die notwendige Anzahl 'leerer' Bits zu einem 32-bit-Vektor ergänzt. Durch die Verwendung bidirektionaler Eingangsmultiplexer mit Sperrstellung (MultiplexerBiDi_wZero) muss nicht jedes belegbare Bit im finalen 32-bit Vektor auch tatsächlich durch ein FG-Signal repräsentiert werden. Eingangsmultiplexer auf Nullstellung erzeugen für diesen Fall eine Null, so dass die entsprechende Stelle im finalen CG-Vektor konstant auf null gesetzt bleibt.

Das Kontrollsignal des FG2CG-Moduls beinhaltet sowohl die Steuervektoren für die Eingangsmultiplexer wie auch die 'in_active' und 'out_active' Steuersignale für den Protocol-Handler. Wie auch im Falle des CG2FG-Moduls wird das Kontrollsignal des FG2CG-Modul direkt an ein Konfigurationsregister angeschlossen.

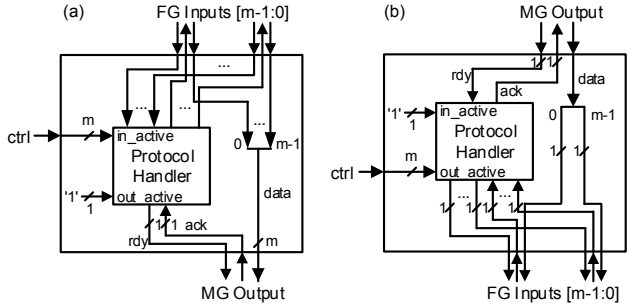



**Abbildung 91: MG-Merger (a) und MG-Splitter (b) Module zur Transfarmation der FG-Signale in MG-Vektoren der Breite m und umgekehrt**

Die beiden Module CG2FG und FG2CG sind in ihrem Aufbau sehr einfach gehalten und unterliegen dadurch einigen Einschränkungen. So können beispielsweise keine Lücken in den CG-Vektoren durch die RTL-Parameter beschrieben werden. Wird nun von einer Anwendung das Bit 31 aus einem CG-Vektor extrahiert, so müssen auch Bits 0-30 bei der Extraktion berücksichtigt werden. Dieser Umstand gilt auch in umgekehrter Richtung. Dies erzeugt zusätzliche Kosten, wurde aber durch die seltene Anwendung der Module und zur Vermeidung des zusätzlichen Aufwands in der RTL-Parametrisierung im Falle einer Erweiterung in Kauf genommen.

### 4.4.7. MG-Merger/Splitter Module

Während auf RU-Ebene der Transport feingranularer Signale in Form multigranularer Vektoren stattfindet, werden feingranulare Signale in FUs direkt transportiert und genutzt. Für den Übergang zwischen diesen Signalformen bzw. Vektorbreiten werden MG-Merger und MG-Splitter Module genutzt. Hierbei handelt es sich um einfache Module zum Zusammenfassen und Aufteilen von FG-Signalen zu MG-Vektoren und umgekehrt. Beide Module beinhalten hauptsächlich je einen Protocol-Handler, so dass diese Module sich wie Knoten im System verhalten und in das gesamte Gefüge der Architektur fügen.

In den MG-Merger Modulen finden sich keine Multiplexer oder Brancher, sondern lediglich eins zu eins Verbindungen der eingehenden FG-Signaldaten zu MG-Signaldaten, siehe Abbildung 91. Der Protocol-Handler koordiniert

den Transport der FG-Signale in die MG-Domäne, indem erst nach Bereitstellung aller aktiven FG-Signale ein MG-Vektor bereitgestellt und transportiert wird. In diesem Fall werden die FG-Signale quittiert und der Zyklus beginnt von neuem. Die aktiven FG-Signale werden durch den 'ctrl' Steuervektor der Breite m beschrieben, was auch dem 'in_active' Vektor des Protocol-Handlers entspricht.

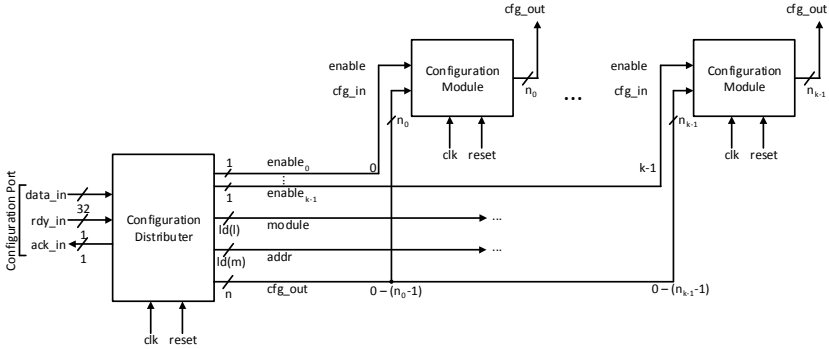



**Abbildung 92: Konfigurationsnetzwerk mit dem Configuration Distributer als Bus-Master und einer Reihe von Konfiguratrionsmodulen**

In ähnlicher Weise, nur in umgekehrter Richtung, verhält es sich mit dem MG-Splitter Modul. Der eingehende MG-Vektor wird statisch in mehrere FG-Signale aufgeteilt. Die Koordination des Handshake-Protokolls wird auch hier durch den Protocol-Handler realisiert. Ein gültiger MG-Vektor erzeugt m gültige FG-Signale. Sobald die Empfänger dieser Signale ihren Empfang quittiert haben, wird auch der MG-Vektor quittiert und der Zyklus beginnt von neuem. Im Idealfall passiert dieser Vorgang in Modulen innerhalb eines Zyklus. Im Einzelnen hängt dies allerdings von aktuellen Anwendungsgegebenheiten ab.

### 4.4.8. Konfigurationsnetzwerk

Die Charakterisierung der Funktionen einer FU erfolgt strukturell mittels einer zuvor erstellten Beschreibung. In binärer Form werden diese Daten als Konfigurationsdaten bezeichnet. Die Übertragung der Konfigurationsdaten auf RU-Ebene erfolgt über die gleichen Netzwerkressourcen wie die Applikationsdaten, wie dies in vorigen Abschnitten beschrieben wurde. Der Eintritt in die FU zur Aktivierung applikationsrelevanter Strukturen und Funktionen erfolgt über den dedizierten Konfigurationsport. Hinter diesem Port innerhalb der FUs, insbesondere im Falle einer DPFU und auch MEMFU, ist der Configuration-Distributer implementiert, siehe Abbildung 92. Dieses Modul nimmt

die eingehenden Daten entgegen, untersucht das Format und leitet die Konfigurationsdaten über den internen Konfigurationsbus an die internen Konfigurationsregister weiter. Im einfachsten Fall werden hier Configuration-Module genutzt, um verteilte Konfigurationsdaten direkt in Registern zu speichern und über den 'cfg_out' Port Funktionsblöcken innerhalb der FU zur Verfügung zu stellen. Das 'enable' Signal wird in diesem Zusammenhang innerhalb des Configuration-Distributer aus dem Datenfeld 'UnitID' dekodiert und jedem Configuration-Module exklusiv zugeordnet, siehe Abbildung 93. Hier wird es direkt als Register-Enable genutzt, um eingehende Daten zu speichern.

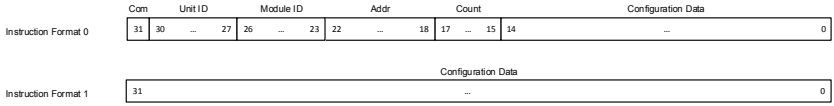



**Abbildung 93: Datenformat für eingehende Konfigruationsdaten und -instruktionen**

Neben den 'enable' Signalen zur Selektion der Configuration-Module und dem 'cfg_out' Datenvektor zum Transport der Konfigurationsdaten enthält der Konfigurationsbus zwei weitere Signale, 'ModuleID' und 'Addr', zur Ansteuerung der komplexeren Zielmodule, siehe Abbildung 94. Hierbei handelt es sich um Module wie HCALUs oder HCLUTs, die intern mehrere Komponenten wie Multiplexer oder Brancher besitzen und darüber hinaus komplette LUTs beherbergen, die separat konfiguriert werden. Mittels des 'ModuleID' Vektors können in diesem Fall die internen Komponenten dieser Module adressiert werden. Integrierte LUTs besitzen in diesem Zusammenhang eine eigene 'ModuleID' Kennzeichnung und nutzen darüber hinaus den 'Addr' Vektor zur Adressierung der LUT-Zeilen. Zur Kategorie der komplexeren Konfigurationsmodule gehören ebenso HCMEM Module, die in MEMHC Zellen zum Einsatz kommen. Hier werden große SRAM-basierte Speicher eingesetzt, die durch Konfigurationen vorbelegt werden können.

Der Aufbau der Konfigurationsdaten kann der Abbildung 93 entnommen werden. Das erste 32-bit Datenwort, welches den Configuration Distributer erreicht, muss das 'InstructionFormat0' besitzen. Die Datenfelder dieses Formates beschreiben das Zielmodule innerhalb der FUs ('UnitID', 'ModuleID' und 'Addr' Vektoren) und die gesamte Länge der Konfigurationsinstruktion (Count Vektor). Insbesondere die Länge definiert wie viele 32-bit-Datenworte nötig sind, um den kompletten 'cfg_out' Vektor zu erhalten. Sobald die definierte Anzahl der Datenworte empfangen wurde, transferiert der Configuration Distributer die Daten in einem Takt an das adressierte Modul.

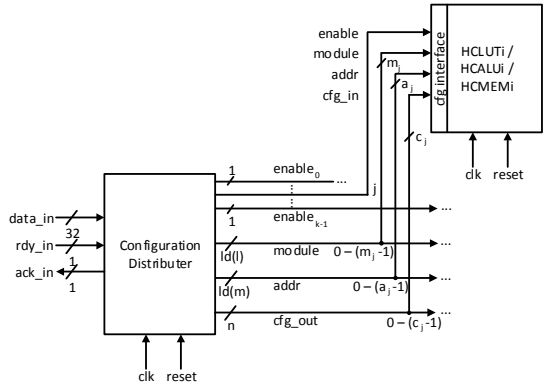



**Abbildung 94: Aufbau des Konfigurationsnetzwerks am Beispiel einer LUT-basierten Komponente wie HCALU, HCLUT oder HCMEM**

Im Hinblick auf die verfügbaren Technologieprimitive insbesondere der erhältlichen SRAM-Module wurde zu Beginn des Designentwurfs die Entscheidung zur parallelen Konfigurationsbusrealisierung getroffen. Durch diesen Ansatz werden auch keinerlei Verzögerungen am Konfigurationsport der FU erzeugt, da die Daten durch den Configuration-Distributer ohne zusätzliche Zyklen durchgereicht werden.

## 4.5. Memory-Functional-Unit (MEMFU)

Während DPFUs Schaltungsstrukturen zur Datenmanipulation anbieten, werden in Memory-Functional-Units (MEMFUs) Speichermodule für array-interne Speicherung und Bereitstellung der Daten implementiert, siehe Abbildung 95. Ebenso wie DPFUs sind MEMFUs generisch formuliert und können dem Bedarf an die Applikationen angepasst werden. So lassen sich die Anzahl der Ein- und Ausgänge, der CG-Register, der FG2CG und CG2FG-Module sowie Anzahl und Größe der Speichermodule parametrisieren. Weiterhin ist das Kommunikationsnetzwerk sehr flexibel gehalten und das Interconnect zwischen den Modulen kann zum RTL-Zeitpunkt konfiguriert werden und steht danach in integrierter Form den Anwendungen zu Verfügung. Während ein vollständiges Interconnect zwischen allen Modulen kostenintensiv ist, lässt sich auf diese Weise wie auch bei den DPFUs Kostenreduktion erreichen. Dieses Vorgehen macht die resultierende MEMFU- sowie DPFU-Strukturen jedoch anwendungsspezifisch und sollte mit Bedacht angewendet werden. Erfahrungen bei der Programmierung der HoneyComb-Architektur zeigten aber, dass beim Einhalten definierter Zugriffsmuster beispielweise beim Zugriff auf die Register sich eine Struktur abzeichnet, die Anwendungen gemein-

sam haben. Dieser Umstand kann zur Kostenreduktion durch Umsetzung regulärer und immer wiederkehrender Strukturen als Basis für die Zellrealisierung ausgenutzt werden. Die Flexibilität der Zellen erhöht sich in diesem Fall bei gleichzeitiger Kostensenkung.

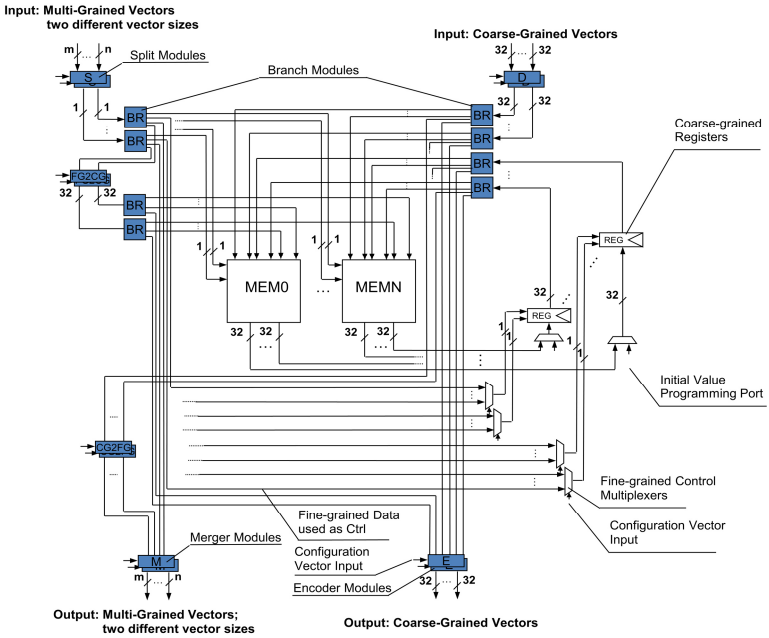



**Abbildung 95: Generische Struktur der Memory-Functional-Unit (MEMFU) mit CG und MG Interfaces, CG-Registern, Datentypenkonvertern und Speichermodulen**

MEMFUs besitzen keine gesonderte FG-Domäne, sind jedoch in der Lage FG-Daten zu speichern. Zu diesem Zweck werden FG-Signale über FG2CG-Module in CG-Datenformate gewandelt und den Speichermodulen zugeführt. Gelesene CG-Daten können wiederum durch CG2FG-Module in FG-Signale rücktransformiert werden und stehen in dieser Form der Anwendung auch außerhalb dieser Zelle zur Verfügung. Neben der Speicherung können FG-Signale auch zur Steuerung der CG-Register genutzt werden, wie dies im Falle der DPFUs möglich ist.

Die meisten Komponenten der MEMFU wurden bereits in vorhergehenden Abschnitten im Zusammenhang mit der DPFU beschrieben und erläutert. Auch der Konfigurationsbus der MEMFU nutzt das gleiche Konzept und wird hier nicht mehr explizit beschrieben. Eine Ausnahme bilden hier die Memory Module (MEM0-MEMN), die exklusiv in den MEMFUs zur Verfügung stehen

und im nachgehenden Abschnitt erläutert werden. Wie auch in DPFUs sind alle CG- und FG-Signale in MEMFUs durch ein Handshake-Protokoll gesichert.

### 4.5.1. Memory Module

Die Memory Module (MEM0-MEMN) der MEMFUs bilden den Kern zur Speicherung der Daten. Hierfür besitzen sie neben einem SRAM-Modul eine zusätzliche Logik, um den Zugriff zu ermöglichen. Anwendungen haben hier die Möglichkeit entweder im RAM-Modus wahlfrei durch generieren eigener Schreib- und Leseadressen auf den Speicher zuzugreifen oder einen der vordefinierten First-In-First-Out- (FIFO) oder Last-In-First-Out-Modi (LIFO) zu nutzen.

Im FIFO-Modus läuft die Speicherung völlig autonom ab, ohne dass sich die Anwendung um die Steuerung kümmern muss. Im LIFO-Modus hingegen, muss die Anwendung zwischen Schreib- und Lesezugriffen umschalten. Dies ist notwendig, um kontrollierten Ablauf im LIFO sicherzustellen, da sonst keine definierte Reihenfolge beim Auslesen des Puffers sichergestellt werden kann. Zur Steuerung des LIFO wird ein eingehendes FG-Signal genutzt, welches durch eine '1' den Schreibzugriff aktiviert und durch eine '0' in den Lesemodus umschaltet.

Zu Vergrößerung der nutzbaren Speichergröße lassen sich zur Laufzeit benachbarte Speichermodule innerhalb der MEMFUs zusammenschalten und damit transparent für die Anwendung ein größerer Speicherbereich erreichen.

## 4.6. Input/Output-Functional-Unit (IOFU)

Die Input/Output-Functional-Unit (IOFU) ist die Hauptkomponente zum Transport der Konfigurations- und Anwendungsdaten zwischen dem Host-System und dem HoneyComb-Array. Zu diesem Zweck besitzt die IOFU neben dem µController weitere Komponenten, die den Datenstrom steuert und je nach Modus konvertiert, siehe Abbildung 96. Der µController hat dabei die Aufgabe, durch Adressgenerierung die Daten blockweise zu Transferieren und die tatsächlich Datenübermittlung zu überwachen. Durch Interrupts kann dieser Fehlsituationen an das Host-System melden oder eigenständig auf Situationen reagieren. Die Programmierung des µControllers ist hierbei entscheidend und legt die Qualität des I/O-Verhalten dieser Komponente fest.

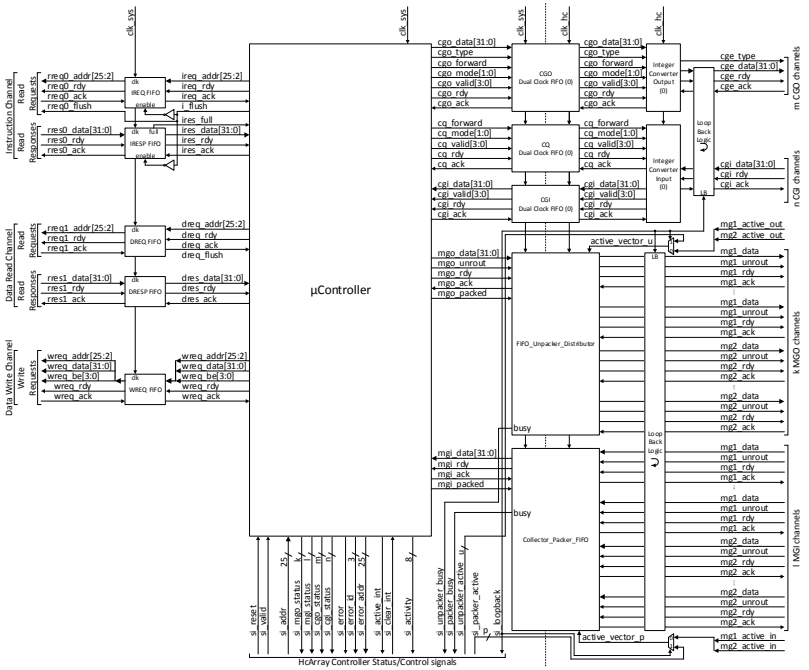



**Abbildung 96: Generisches Blockschaltbild der Input/Output-Functional-Unit (IOFU)**

Auf der linken Seite der Abbildung 96 finden sich die Schnittstellen des µControllers an das Host-System. Dafür müssen drei getrennte Kanäle unterschieden werden. Als erstes wäre hier der Instruction-Channel zu finden. Dieser Kanal steht exklusiv der Übermittlung der Instruktionen aus dem Hauptspeicher zum µController zur Verfügung. Hierbei werden über den Anforde-

rungskanal 'rreq0' (ReadRequests) die Leseadressen an das System übermittelt und über den Antwortkanal 'rresp0' (ReadResponses) die eigentlichen Instruktionen zurück erwartet. Über die Eingangs- und Ausgangs-FIFOs lassen sich zum einen mehrere Adressen im Voraus generieren und zum anderen eine Instruktionen-Queue zur schnelleren Verarbeitung realisieren. Im Falle eines Programmsprungs besteht hier die Möglichkeit den Inhalt dieser FIFOs zu löschen, um an einer neuen Adresse die Ausführung fortzusetzen.

Datenübermittlung in Lese- und Schreibrichtung sind ihrerseits ebenfalls in zwei getrennte Kanäle aufgeteilt (DataReadChannel und DataWriteChannel). Analog zum Instruktionskanal ist der Datenlesekanal in zwei Unterkanäle unterteilt. Zum einen werden hier über den Unterkanal 'rreq1' (ReadRequests) an das Host-System übermittel und zum anderen über den Unterkanal 'rres1' (ReadResponses) die gelesenen Daten zurück erwartet. Auf den ersten Blick fällt die Ähnlichkeit zwischen dem Instruktionskanal und dem Datenlesekanal auf und legt eine einheitliche Implementierung nahe. Dies ist jedoch nicht realisierbar, da beide Kanäle in der Regel Warteschlangen bilden und der Datenstrom dieser Datentypen sich gegenseitig blockieren würde.

Den letzten Kanal zum System stellt der Schreibkanal dar, der alle Informationen zum Schreiben der Daten aus dem HoneyComb-Array beinhaltet (Daten, Adressen, Byte-Enable) und in einem Takt übermitteln kann.

Auf der rechten Seite der Abbildung 96 finden sich die Komponenten zur Übertragung der Daten zum und aus dem HoneyComb-Array. Das CG-Output-Interface (CGO) beschreibt die CG-Schnittstelle zum HoneyComb-Array hin. Die Steuersignale aus dem µController kommend werden hierbei zunächst mittels Dual-Clocked-FIFOs in die Clock-Domäne des HoneyComb-Arrays übertragen und vom 'IntegerConverterOutput' Module für das Array aufbereitet. Dieses Modul bietet die Möglichkeit eingehende 32-bit Datenworte direkt oder in 8/16-bit Worten zum Array zu übertragen. Mehr Details zu dieser Funktion bietet der Abschnitt 4.6.4. In der Gegenrichtung sorgen die CG-Input-Kanäle (CGI) für den Datentransfer aus dem Array zum Host-System. Zur Steuerung dieser Schnittstelle wird allerdings der 'CommandQueue' Kanal eingesetzt, in dem zunächst Steuerinformationen vom µController kommend zum 'IntegerConverterInput' Modul übertagen werden und das Format der ausgehenden Daten in Richtung Host-System beschreiben. Sowohl die CGI- als auch CGO-Schnittstellen sind in der Demonstrationschipumsetzung der HoneyComb-Architektur jeweils zweifach ausgeführt und können parallel genutzt werden. Dies ist möglich, da die Taktfrequenz des Host-Systems üblicherweise höher angesiedelt ist und damit die Bandbreite auf dieser Seite als ausreichend angesehen wird. Darüber hinaus benötigen HoneyComb-Applikationen separate Schnittstellen zum Array hin und können auf RU-Ebene keine Multiplex/Demultiplex-Funktionen realisieren. Zu Debugging-

Zwecken besitzen CGI- und CGO-Schnittstellen paarweise eine systemseitige Loop-Back-Funktion, welche bei der Inbetriebnahme der HoneyComb-Architektur bei der Fehlersuche genutzt werden kann.

Neben den CG-Schnittstellen sind in den IOFUs auch direkte MG-Schnittstellen implementiert, die es erlauben ohne Umwege MG-Daten vom und zum Host-System zu übertagen. Zum einen findet sich hier das FIFO-Unpacker-Distributer-Modul, welches neben dem Clock-Domänenübergang ebenso Unpack-Funktionen und die MG-Gruppensteuerung für dieses Datenformat bietet, näheres siehe Abschnitt 4.6.4. In der Gegenrichtung sorgt das Collector-Packer-FIFO-Modul für die Umkehrfunktion durch das Packing und Datentransfers zum Host-System. In Anlehnung an das BCD-Format (Byte Coded Decimal) werden bei der MG-Übertagung auf Bit-Ebene optional die Daten ausgepackt und beim Rücktransport zum System wieder eingepackt. Ähnlich wie bei CG-Schnittstellen ist auch bei diesen MG-Kanälen zum Debuggen eine Loopback-Logik implementiert.

Um den µController zu steuern und seinen Status abzufragen, wurden mehrere Signale vom und zum globalen Controller zur IOFU integriert. Damit lässt sich einem ruhenden µController die Startadresse für das nächste Programm übermitteln oder ein Fehlerfall erkennen und seine Ursachen untersuchen. Eine vollständige Übersicht der möglichen Funktionen bietet diese Tabelle:

**Tabelle 18 Status- und Kontrollsignale der IOFU zum globalen Controller**

| | Signal | I/O | Beschreibung |
|---|---|---|---|
| 1 | **si_reset** | I | Zurücksetzen des µControllers |
| 2 | **si_valid** | I | Anzeige Gültigkeit von **si_addr** |
| 3 | **si_addr** | I | Startadresse des µController-Programms |
| 4 | **si_mgo_status** | O | Status der MG-Output-Adressgeneratoren |
| 5 | **si_mgi_status** | O | Status der MG-Input-Adressgeneratoren |
| 6 | **si_cgo_status** | O | Status der CG-Output-Adressgeneratoren |
| 7 | **si_cgi_status** | O | Status der CG-Input-Adressgeneratoren |
| 8 | **si_error** | O | Indikator für einen internen µController-Fehler |
| 9 | **si_error_id** | O | Fehler-Code für den Fall '**si_error**' = '**1**' |
| 10 | **si_addr** | O | Adresse für den aufgetretenen Fehler |
| 11 | **si_active_int** | O | Interrupt des µControllers |
| 12 | **si_clear_int** | I | Löschsignal für den Interrupt |
| 13 | **si_activity** | O | Anzeige der aktiven µController-Komponenten |
| 14 | **si_unpacker_busy** | O | Indikator für die Aktivität des MGO-Unpackers |
| 15 | **si_packer_busy** | O | Indikator für die Aktivität des MGI-Packers |
| 16 | **si_unpacker_active** | I | Bitbreitendefinition für den Loopback-Modus |
| 17 | **si_packer_active** | I | Bitbreitendefinition für den Loopback-Modus |

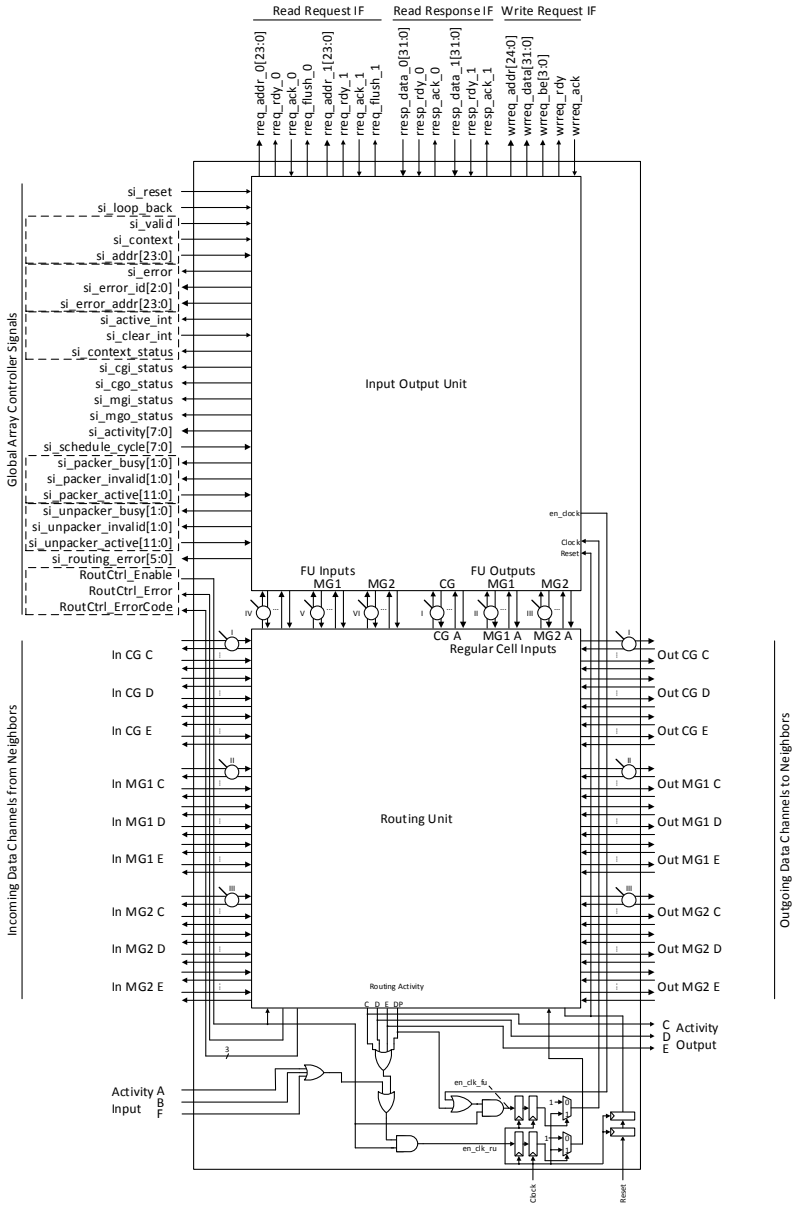



**Abbildung 97: Anbindung der IOFU an die RU über reguläre Zelleneingänge (hier Port A) und dedizierte FU-Ports**

### 4.6.1. Anbindung an die Routing Unit (RU)

Die Anbindung der IOFU an die RU derselben Zelle erfolgt wegen ihrer Sonderstellung als Quelle für Daten und Instruktionen auf eine unkonventionelle Art und Weise. Während DPFUs und MEMFUs dedizierte FU-Ports für ihre Anbindung nutzen, funktioniert dieser Ansatz nicht bei IOFUs. Die Ursache hierfür liegt in der Tatsache begründet, dass IOFUs Instruktionen generieren können, die von der RU dieser Zelle geroutet werden muss. Dies funktioniert nicht, wenn OutRegCGs als Eingangsmodule zu RUs genutzt werden, da diese Module nicht selbstständig Routing-Anfragen starten können. Diese Aufgabe entfällt auf die InRegCGs, so dass eingangsseitig zu RUs hin die Anbindung der IOFUs über Seitenkanten der Zelle erfolgen muss, siehe Abschnitt 4.3.4.1.

Da die Standardplatzierung der IOHCs am Rande der HoneyComb-Architektur erfolgt, ist dieser Umstand in dieser Aufgabenstellung entgegenkommend. Durch die Platzierung am Rande verbleibt immer mindestens eine Kante der Zelle frei und kann damit als Schnittstelle zwischen IOFU und RU dienen. Dies kann aber nur in Richtung RU erfolgen, da eine Zellkante ausgangsseitig nicht direkt angesteuert werden kann. Dafür werden weiterhin die dedizierten FU-Eingänge verwendet, so dass die Routing-Instruktionen weiterhin nach vorgesehenem Muster funktionieren und diese Ports direkt ansteuern können. Abbildung 97 zeigt das Blockschaltbild und die beschriebene Besonderheit einer IOFU-Anbindung an die RU. Die Signalkodierung dieser Abbildung findet sich in der Abbildung 48.

Aus diesen Umständen ergibt sich auch der Unterschied der CGO- und CGI-Kanäle aus der Abbildung 96. Während CGI-Schnittstellen ein FU-internes Handshake-Protokoll (CGI, CG-intern) nutzen, besitzt die CGO-Schnittstelle wie die CG-Signale auf RU-Ebene ein zusätzliches 'type'-Bit (CGE, CG-extern). Über dieses Bit legt der µController fest, welche Daten welchen Status besitzen. Dafür sind im Instruktionssatz des IOHC-Controllers dedizierte Befehle definiert, die diese Differenzierung ermöglichen.

### 4.6.2. µController Aufbau und Funktionalität

Der IOFU-µController besteht aus einer Pipeline-Struktur, die in der Lage ist einen vereinfachten RISC-Programmcode auszuführen und weiteren Komponenten zur Realisierung von Blocktransfers. Die Ansteuerung der Blocktransferlogik erfolgt ebenfalls durch die Instruktionen der Pipeline und fügt sich damit transparent in die Programmierung der IOFUs ein. Neben CG und MG Datentypen, werden auch mehrfache Ausführungen der notwendigen Adressgeneratoren beider Datentypen und Richtungen unterstützt, so dass quasi parallele Blocktransfers durch IOFUs programmierbar sind.

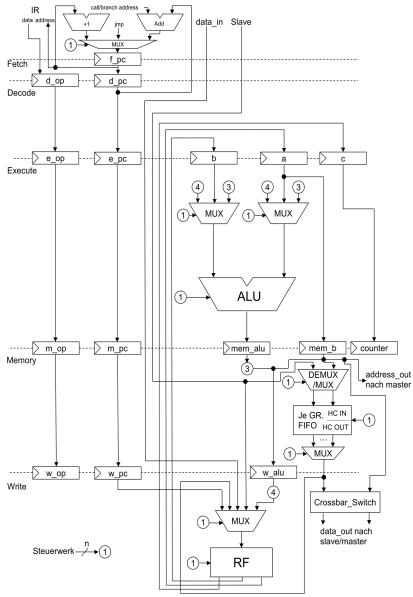



**Abbildung 98: DLX-angelehnte sechsstufige Pipeline-Struktur des µControllers**

Abbildung 98 veranschaulicht die entwickelte Pipeline-Struktur in vereinfachter Form, wie sie im µController eingesetzt wird. Dabei handelt es sich um eine sechsstufige Pipeline, die alle notwendigen Programmflusskontrollmechanismen, Speicherzugriffe, Operationen, Interrupts, 16 Register und zusätzliche Befehle für IOFU-Logik wie Blocktransfers bietet. Zur Steuerung des Programmflusses werden Vergleichsbefehle, relative und absolute Sprungbefehle, die bedingt und unbedingt auch in Abhängigkeit der Blocktransferlogik ausgeführt werden können, geboten. Über MOV-Befehle lassen sich Speicherwerte für weitere Operationen in die Register der Pipeline kopieren und anschließend ebenso wieder zurückschreiben. Zusätzlich lassen sich damit einzelne Werte in die CGO/MGO-Schnittstellen übertragen und damit zum HoneyComb-Array transportieren. Ebenso ist der Weg aus den Arrays für beide Datentypen möglich, wobei das Ziel sowie Quelle in die entgegengesetzte Richtung sowohl in ein internes Register wie auch eine Speicherstelle sein kann. Interrupts müssen in dieser Struktur über separate Befehle ausgelöst werden und der Status kann ebenso durch weitere Instruktionen geprüft werden. Das Löschen der Interrupts muss über den globalen Controller erfolgen, damit sichergestellt ist, dass das Host-System das Event verarbeitet. Unterstützte Operationen der Pipeline schließen arithmetische und logische Operationen ein. Ebenso werden Multiplikation und Division im VHDL-Modell opti-

onal unterstützt. Arithmetische Operationen machen keine Unterscheidung zwischen vorzeichenbehafteten und vorzeichenlosen Zahlendarstellungen, so dass der Programmierer hier durch zusätzliche Operationen Abhilfe schaffen muss.

Blocktransferbefehle sind in ihrer Implementierung und Anwendung vielseitig konzipiert. So unterstützen sie neben relativen und absoluten Adressierungsarten auch inkrementelle und iterative Zugriffsarten. Die letzten beiden Modi legen fest, ob die Adresse bei anschließenden Zugriffen inkrementiert wird oder nicht. Dafür wird die Schrittweite in der Instruktion definiert und es lassen sich größere Schritte als eins angeben. Weiterhin unterscheiden die Blocktransfers die Art der Datentransfers. So lassen sich gewöhnliche CG-Applikationsdaten mit den Befehlen BTCR und BTCW zum und aus dem Array transportieren, während der Befehl BTCFG die transportierten Daten über das CG-Bit 'type' als Instruktionen markiert. Dies geschieht beim Transport mittels des BTRI Befehls ebenso, allerdings wird in diesem Fall der Transport einer Routing Instruktion (RICG oder RIMG1) zur dieser IOHC ausgehend von einer beliebigen Zelle im Array angenommen und die Zielkoordinaten in der Instruktion durch Koordinaten der ausführenden IOHC ersetzt. Dieses Vorgehen ist wichtig, da zum Compile-Zeitpunkt nicht festgelegt wird, welche IOHC die Applikation ausführen soll und dies sich ohnehin zur Laufzeit ändern darf. Auf diese Weise müssen alle Routing Instruktionen in Richtung IOHCs mittels des BTRI-Befehls transportiert werden. Daneben gibt es auch MG-Transportbefehle wie BTFR und BTFW, die sich abgesehen von transportierten Datentypen in ähnlicher Weise verhalten wie BTCR und BTCW Befehle.

Neben den beschriebene Befehlen, die eine relative Adressierung für den Speicherzugriff nutzen, wurden auch Befehle für die absolute Adressierung definiert, die eine andere Mnemonik besitzen, in der Anwendung jedoch identisch sind: BTACR, BTACW, BTACFG, BTARI, BTAFR, BTAFW. Durch den zusätzlichen Buchstanden A in der Mnemonik wird hierbei auf die absolute Adressierung hingewiesen. Die anschließende Tabelle fasst die Blocktransferbefehle der relativen Adressierung zusammen und gibt die verwendeten Parameter an:

**Tabelle 19 Blocktransferbefehle und Optionen für relative Adressierung**

| | Befehl | Parameter |
|---|---|---|
| 1 | BTCFG | R1, R2 \| <Val4>, [R3 \| <Val4>], R4 \| <Val4> |
| 2 | BTRI | R1, R2 \| <Val4>, [R3 \| <Val4>], R4 \| <Val4> |
| 3 | BTCR | R1, R2 \| <Val4>, [R3 \| <Val4>], R4 \| <Val4>, [INT8 \| INT16 \| INT32] |
| 4 | BTCW | R1 \| <Val4>, R2 \| <Val4>, [R3 \| <Val4>], R4, [INT8 \| INT16 \| INT32] |
| 5 | BTFR | R1, R2 \| <Val4>, [R3 \| <Val4>], R4 \| <Val4> |
| 6 | BTFW | R1 \| <Val4>, R2 \| <Val4>, [R3 \| <Val4>], R4 |

Die Lesebefehle in dieser Zusammenstellung verwenden ein identisches Format. Der erste Parameter gibt die Startadresse an, die in einem Register gespeichert erwartet wird (R1). Der zweite Parameter gibt die Anzahl der zu lesenden 32-Bit Datenworte an, der entweder direkt oder durch einen Wert im Register spezifiziert wird, wie dies auch bei den folgenden Parametern der Fall ist. Der nächste Parameter definiert die Schrittweite, die optional angegeben werden kann. Die Voreinstellung nutzt den Wert eins. Der letzte Parameter definiert die Zielschnittstelle, welche zur Übertagung zum Array genutzt werden soll. Im Falle des BTFR Befehls wird hierbei die MGO-Schnittstelle adressiert. Bei Schreibbefehlen (BTCW und BTFW) sind der erste und vierte Parameter im Vergleich zur Lesebefehlen vertauscht, so dass zuerst das Quell-FIFO des entsprechenden Datentyps erwartet wird und zuletzt die Zieladresse im angegebenen Register.

Während die CG-Blocktransferbefehle alle notwendigen Parameter zur Ansteuerung der Blocktransferlogik mitbringen, muss im Falle der MG-Transfers auf zusätzliche Befehle zurückgegriffen werden.

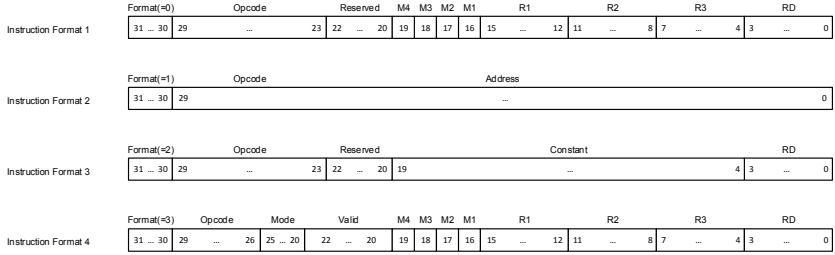



**Abbildung 99: RISC Instruktionsformate der entwickleten Pipeline einschließlich der Unterstützung der Blocktransfermodule**

Abbildung 99 zeigt die genutzten Instruktionsformate, die für die Pipeline-Ausführung definiert wurden. Die ersten drei Formate werden für Standardbefehle innerhalb der Pipeline genutzt, während das vierte Format insbesondere für CG-Blocktransferbefehle benötigt wird. Hier sind insbesondere die zusätzlichen Parameter ‚Mode' und ‚Valid' definiert, die zur Ansteuerung der Blocktransferschaltkreise notwendig sind. Die Felder M1, M2, M3 und M4 werden von der Pipeline genutzt, um zwischen direkter und indirekter Parametrisierung über Register zu unterscheiden. Das zweite Format wird genutzt, um den unbedingten Sprungbefehle ‚JMP' zu realisieren. Durch die verfügbaren 30-bit für die Adresse und einem zwingenden Alignement auf 32-bit-Grenzen im Hauptspeiche, lässt sich jede Speicherzelle im 32-bit-Adressraum über diesen Befehl erreichen. Die Adressierung wird hierbei relativ zum aktuellen Programm-Pointer berechnet, wobei der angegebene Adresswert als Offset zur aktuellen Position interpretiert wird.

Das dritte Format erlaubt das Vorladen der internen Register mittels Konstanten, wobei zwischen Low- und High-Anteilen unterschieden wird. Durch zwei Instruktionen lässt damit ein 32-bit-Register vollständig vorbelegen und in der Applikation nutzen.

### 4.6.3. Block-Transfer-Datenpfad

Der Datenpfad zur Realisierung der Blocktransfers wurde bei der Realisierung in zwei Teile eingeteilt. Der erste Teil befasst sich mit der Generierung der Schreib- und Leseadressen, die Anfragen an das Host-System leitet und koordiniert und die Antworten an das Array weitergibt. Dies wurde als Teil des µControllers realisiert, da hier eine enge Bindung durch eine Reihen von Steuer- und Statussignalen an die Pipeline besteht. Der zweite Teil wurde bereits im Abschnitt 4.6 eingeführt und besteht aus Integer Convertern und MG Packer/Unpacker-Logik, die im folgenden Abschnitt detaillierter beschrieben werden. Zur besseren Übersicht wurde die Blocktransferlogik innerhalb des µControllers in zwei Teil aufgeteilt, die weitgehend unabhängig arbeiten und jeweils eine Transferrichtung in die des HoneyComb-Arrays und des Host-Systems realisieren.

Abbildung 100 veranschaulicht den Datenpfad für Datentransfer vom Host-System in Richtung des HoneyComb-Arrays. Darüber hinaus sind Mechanismen zum Transfer der Daten zu Pipeline-Registern vorgesehen, sowie Transfers zwischen den Registern und dem HoneyComb-Array. Die Steuersignale von der Pipeline kommend werden von links zunächst an die CG/MG Adressgeneratoren (CG/MG AG Output Module) über bidirektionale Multiplexer geleitet und stoßen Adressgenerator-Funktionen an, die durch Blocktransferbefehle in der Pipeline gesteuert werden. Dazu gehören Signale wie 'fifo_out_addr', 'fifo_out_step' und 'fifo_out_count', die eine Startadresse, Schrittweite und die Anzahl der zu lesenden Datenworte vorgeben. Das Signal 'fifo_out_ag_start' stößt den ausgewählten Adressgenerator an. Die Selektion des Adressgenerators erfolgt über die CG-FIFO-Nummer, die in einem der Blocktransferbefehle spezifiziert wird und am Ausgang der Pipeline das Selektionssignal 'fifo_out_cg_sel' bestimmt. Zu jeder CGO-Schnittstelle in dieser Darstellung gehört ein CG-Adressgenerator. Einmal angestoßen erzeugt der aktive CG-Adressgenetor eine Sequenz von Adressen am Ausgang 'cg_master_addr', die über einen weiteren Multiplexer an den Read Data Request Channel geleitet werden. Die Steuerung des Multiplexers übernimmt dabei der Priority-Manager, der bei Aktivität der Adressgeneratoren nach dem Round-Robin-Verfahren der Reihe nach ihre Ausgänge an den Request Channel durchstellt. Gleichzeitig wird eine Kopie der Anforderung und aller zusätzlicher Parameter im Request FIFO gespeichert.

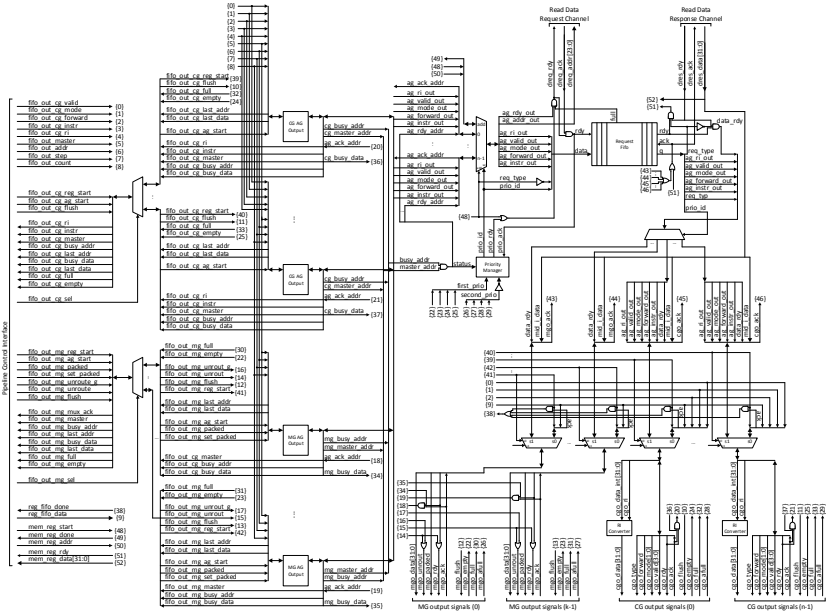



**Abbildung 100: CG/MG Adressgeneratoren und Datenpfad der in Richtung HoneyComb-Array fließenden Daten (CGO, MGO)**

Am selben Multiplexer hat die Pipeline die Möglichkeit Speicherlesezugriffe über den Eingang 'add' an den Read Data Request Channel abzusetzen. Dazu wird das Signal 'mem_reg_start' gesetzt und die AG-Anfragen übergangen. Die gelesenen Daten für die Pipeline Register werden direkt am Eingang des Read-Data-Response-Channel abgriffen und in die Pipeline über den Eingang 'mem_reg_data' geleitet.

Der µController sorgt damit dafür, dass Leseanfragen an das Host-System in der gleichen Reihe bearbeitet und über den Read Data Response Channel die gelesenen Daten zurückgeliefert werden können. Auf diese Weise ist die Reihenfolge der gespeicherten Anfragen im Request FIFO die gleiche wie auch die Reihenfolge der Antworten am Read Data Response Channel. Damit werden die eingehenden Datensätze um die zugehörigen Parameter aus dem Request FIFO ergänzt und in Richtung CGO-Schnittstelle geleitet. Die letzte Multiplexerstufe am CGO-Interface erlaubt die direkte Durchleitung der Registerdaten, falls es sich um einen direkten Transfer zwischen Register und HoneyComb-Array handelt. Zu diesem Zweck muss das Steuersignal 'fifo_out_cg_reg_start' gesetzt werden und der Vektor 'reg_fifo_data' die Daten für den Transfer bereitstellen.

Nach der Multiplexerausgangsstufe besteht weiterhin die Möglichkeit im Falle einer Routing-Instruktion die Zieladresse anzupassen, wie dies für den Blocktransferbefehl BTRI im Abschnitt 4.6.2 beschrieben wurde. Hierbei wird nicht versucht eine Routing-Instruktion zu erkennen, sondern lediglich die vorliegenden Datenvektoren an den Datenfeldern X und Y entsprechende der RICG und RIMG1 Definitionen zu ändern. Handelt es sich bei dem Blocktransfer um Konfigurationsdatenübertagung, wie sie mit dem BTCFG-Befehl angestoßen werden kann, so wird zusätzlich das Bit 'cgo_type' in Abhängigkeit des Signals 'ag_ri_out' und damit am Steuerinterface von der Pipeline kommend das Signal 'fifo_out_cg_instr' gesetzt. Viele der von der Pipeline ankommenden Signale werden in dieser Struktur bis zum Ausgang durchgeleitet, um in nachfolgenden Integer-Convert-Modulen die eigentliche Konvertierung der Daten vorzunehmen. Die Adressgeneratoren speichern diese Signale zunächst ab und leiten sie dann mit den erzeugten Adressen in Richtung des Request-FIFO.

Um den aktuellen Status der CGO- wie auch MGO-Schnittstellen abfragen zu können, liefern die nachfolgenden FIFOs ihren Status an das CGO- bzw. MGO-Interface und können über den bidirektionalen Multiplexer auf der linken Seite der Abbildung zur Pipeline durchgestellt werden. Das gleiche gilt für die Adressgeneratoren, die ihren Status über 'busy'-Signale anzeigen. So werden beispielweise Register nur dann in das HoneyComb-Array übertragen, wenn sowohl der zugehörige Adressgenerator im Leerlauf ist und die Schnittstelle bereit ist. Für die Pipeline wurden Befehle definiert, die auf diese Informationen zugreifen und damit den Programmfluss beeinflussen können.

Im Falle der MGO-Schnittstellen funktioniert die Übertragung im Wesentlichen vergleichbar zum CGO-Ansatz. Hier besteht jedoch keine Möglichkeit eine Schrittweite für die Adressberechnung anzugeben, so dass der Reihe nach ein Datenwort nach dem anderen gelesen wird. Stattdessen bietet die MG-Schnittstelle der Pipeline eine Reihe weitere Steuersignale, die das Packing/Unpacking oder Unroute-Signale über die MGO-Schnittstelle weitergeben können. Im letzten Fall werden Löschbefehle für bestehende Routen am MG1-/MG2-Interface der RU dieser Zelle erzeugt und diese gelöscht.

Im Vergleich zum Leseteil der Blocktransferlogik ist der Schreibteil etwas einfacher aufgebaut, siehe Abbildung 101. Hier besteht das Host-System-Interface nur aus einer Richtung, so dass hier keine Synchronisationsmechanismen bereitgestellt werden müssen. In ähnlicher Weise stößt die Pipeline ihre Anfragen über ihre Steuersignale an, welche über bidirektionale Multiplexer an die CG/MG-Adressgeneratoren geleitet werden. Einmal angestoßen, erzeugen sie im CG-Fall zum einen eine Sequenz von Steuersignalen für die Integer Converter und zum anderen eine Sequenz von Adressen. Die Steuersignale werden über separate Command-Channels (CGI_CQ) an die Integer-

Converter geleitet und steuern dadurch den Rückfluss der Daten aus dem HoneyComb-Array, indem spezifizierte Datenpacking-Funktionen aktiviert werden. Ausgangsmultiplexer an Command Channels erlauben der Pipeline einzelne Lesebefehle an das Array abzusetzen, um das Ergebnis in die eigenen Register zu kopieren.

Eingehende Daten von Integer-Convertern an CGI-Schnittstellen werden entweder an die Register der Pipeline geleitet, oder aber zum Data-Write-Request-Channel durchgestellt. Hier werden zuvor die anliegenden Adresssequenzen und die aktiven CGI-Schnittstellen durch den Priority-Manager miteinander verglichen und falls beide aktiv sind, an den Data-Write-Request-Channel durchstellt. Bei mehreren validen Anfragen wird nach dem Round-Robin-Verfahren der Reihe nach durchgeschaltet. Daneben beachtet der Priority-Manager den Zustand der Eingangs-FIFOs an CGI und MGI Schnittstellen und favorisiert zusätzlich die Eingänge mit höherem FIFO-Stand.

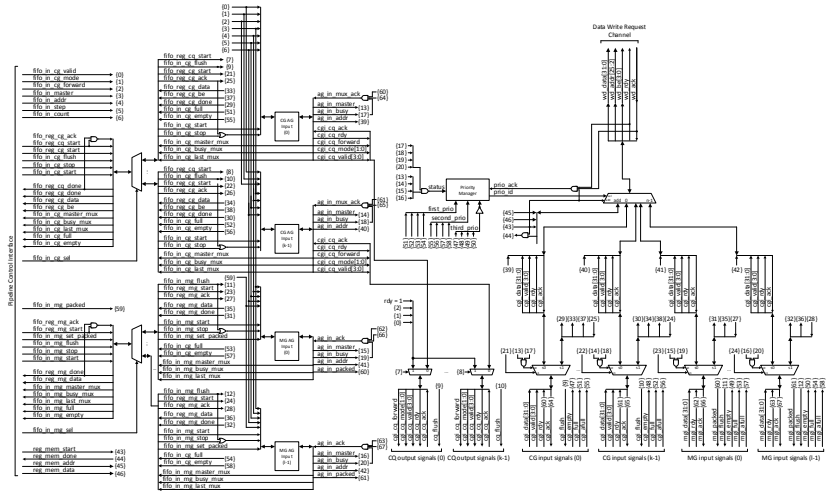



**Abbildung 101: CG/MG Adressgeneratoren und Datenpfad der in Richtung Host-System fließenden Daten (CQ, CGI, MGI)**

Am Ausgangsmultiplexer des Data-Write-Request-Channels besteht weiterhin die Möglichkeit für die Pipeline Registerwerte zum Host-System zu übertragen. Dazu müssen die Vektoren 'reg_mem_data' und 'reg_mem_addr' die Daten bzw. Schreibadresse bereitstellen und durch das Signal 'reg_mem_start' den Vorgang anstoßen. Bei Beendigung des Transfers wird dies über den Eingang 'reg_mem_done' signalisiert.

In ähnlicher Weise zu CG-Block- und Registertransfers unterstützt diese Struktur die Übertragung von MG-Daten. Auch hier liegen die Unterschiede in

den vorliegenden Steuersignalen. Weiterhin sind keine Command-Channels für die MGI-Schnittstellen vorgesehen, da in diesem Fall nur ein quasi statischer Vektor 'mgi_packed' benötigt wird, welcher für die Dauer einer kompletten Blocküübertragung unverändert bleibt. Sowohl CGI- als auch MGI-Schnittstellen liefern Byte-Enable-Signale (cgi/mgi_valid), die byte-selektives Schreiben im Hauptspeicher des Host-Systems ermöglichen.

### 4.6.4. CG Packing/Unpacking Module

Die Integer-Converter-Module in der IOFU bieten eine Möglichkeit, die 32-bit-Daten an der Lese- und Schreibschnitte zum Host-System effizienter auszunutzen. Dafür lassen sich die in der HoneyComb-Architektur genutzten Datentypen mit kleiner Bitbreite von 8/16-bit mehrfach in ein 32-bit Datenwort packen.

Beim Lesen wird das eingehende 32-bit-Datenwort je nach verwendetem internen Datentyp unterschiedlich zerlegt bzw. entpackt, siehe Abbildung 102. Bei Verwendung von 32-bit-Datenworten im Array wird der Eingang einfach durchgestellt, ohne die Daten zu ändern. Bei einem 16-bittigem Datenwort im Array wird das eingehende Datenwort in zwei 16-bit-Werte aufgeteilt. Die Reihenfolge der MSB bzw. LSB Anteile bei der Weiterleitung hängt in diesem Fall vom Parameter 'forward' ab und lässt sich nutzen, um auf vorliegende Gegebenheiten von Endianness (little endian, big endian) innerhalb des Host-Systems zu reagieren.

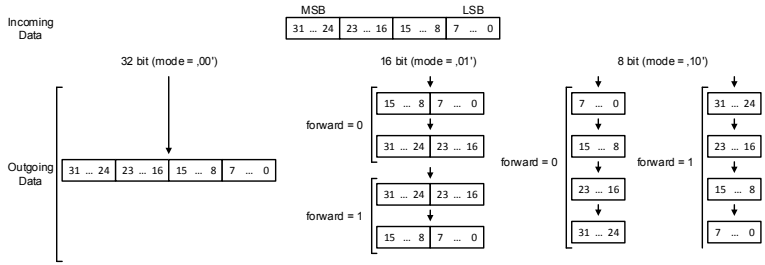



**Abbildung 102: Unpacking-Verfahren basierend auf den Parametern 'forward' und 'mode'**

Im 8-bit-Fall wird das Eingangsdatenwort in vier einzelne Bytes zerlegt und in Abhängigkeit des 'forward' Parameters der Reihe nach an das Array geleitet. Der Datentyp der Array-Daten wird in Integer-Convertern durch den Parameter 'mode' definiert, wie dies in Abbildung 102 veranschaulicht wird. Zusätzlich wird ein dritter Parameter 'valid' mit der Breite von 4 Bits verwendet, um die Validität jedes Bytes des 32-bit-Datenworts am Eingang anzuzei-

gen. Damit lassen sich zusätzlich Bytes oder 16-bit Werte beim Unpacking/Packing überspringen.

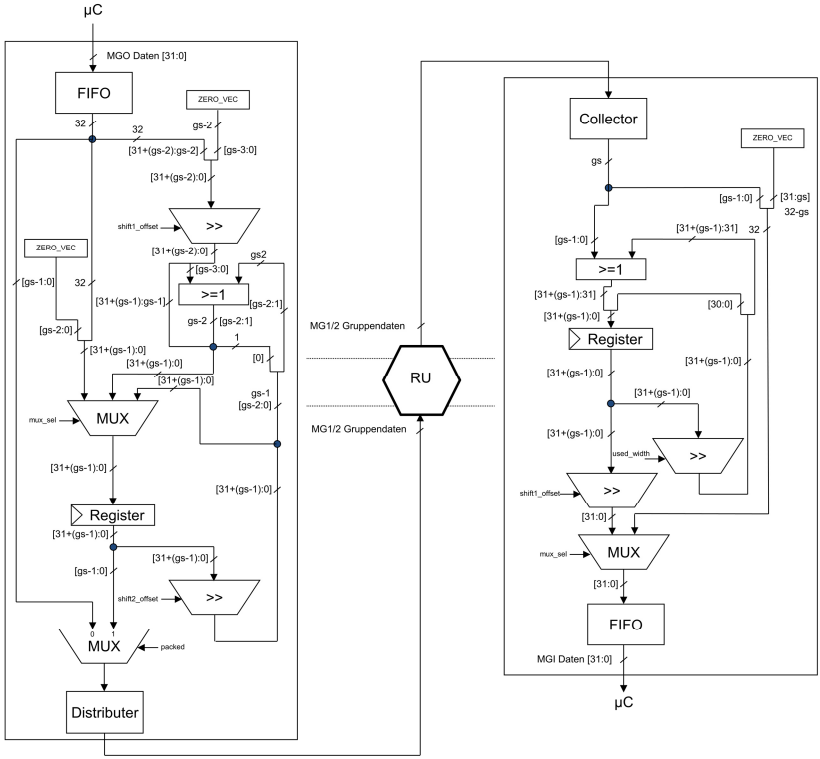



**Abbildung 103: Der gesamte Datenpfad der FIFO-Unpacker-Distributer und Collector-Packer-FIFO Module**

Analog zum CG-Unpacker, wie dies in Abbildung 102 dargestellt ist, funktioniert auch der CG-Packer in umgekehrter Richtung. Diese Funktion wird durch das Module Integer-Converter-Input, wie in Abbildung 96 dargestellt, realisiert.

### 4.6.5. MG Packing/Unpacking Module

Im Falle der MG-Daten ist der Spielraum bei Packing/Unpacking-Funktionen deutlich größer, da die Bitbreite der verwendeten Vektoren kein Vielfaches von acht sein muss, sondern alle Größen von eins bis zur maximalen MG-Gruppengröße annehmen kann. Daher ergibt sich auch eine für 32-bit-

Datenwortgrenze übergreifende Packing-Methode, um keinen Verschnitt der verbleibenden Bits zuzulassen, falls die 32 durch die aktuell genutzte Vektorgröße nicht teilbar ist. Für diesen Fall wurde der Packing-Schaltkreis entwickelt, der es erlaubt die geforderte Packing-Funktionalität zu bieten und im Modul FIFO-Packing-Distributer eingesetzt wird, siehe Abbildung 103.

Hier besteht die Möglichkeit die Daten direkt durchzulassen, was den Nachteil hat, dass ein Großteil der 32-bit-Daten ungenutzt bleibt. Dazu muss der Ausgangsmultiplexer auf der linken Seite der Abbildung auf 'packed' = '0' geschaltet werden. In gepackter Form werden die Eingangsdaten durch Shift-Operationen geschoben, indem von rechts nach links immer ein Teil des Vektors der aktiven MG-Gruppengröße entsprechend abgeschnitten und weitergeleitet wird, während der restliche Teil des ursprünglichen Datenwortes durch eine weitere Iteration dieselbe Prozedur durchläuft. Ist der verbleibende Teil des Datenvektors kleiner als die aktive MG-Gruppengröße, wird das nächste eingehende Datenwort links an den Rest angehängt und die Prozedur fortgesetzt. Dies wird wiederholt, bis die gewünschte Anzahl der MG-Vektoren erzeugt wurde. Anschließend wird die Struktur explizit durch Pipeline-Instruktionen zurückgesetzt.

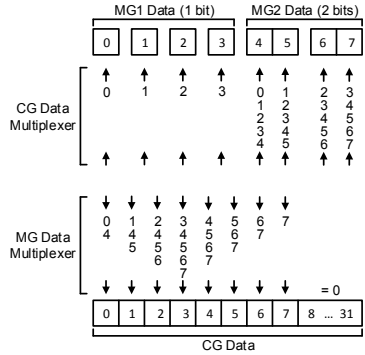



**Abbildung 104: MG Collector/Distributer Konzept zur Verteilung der MG-Daten auf einen statischen Bitvektor der selben Größe**

Die ausgehenden MG-Gruppendaten werden im Distributer-Modul auf die tatsächlich aktiven MGO-Channels verteilt, siehe Abbildung 104 oben. Dazu werden pro MG1 und MG2 Datenkanal je ein Multiplexer verwendet, die alle zusammen in geordneter Form die eingehenden Bits der Daten (hier als CG Data bezeichnet) den MG-Datenbits zuordnen. Die Ordnung bzw. Einschränkung besteht hier darin, dass die eingehenden Bits der Reihe nach von links nach rechts aufsteigend angeordnet sein müssen. Lücken innerhalb derselben Kanaltypen sind nicht erlaubt, so dass ein inaktiver MGX-Kanal zur Folge hat,

dass ab seiner Position alle weiteren Kanäle desselben Typs ungenutzt bleiben. Diese Einschränkung wurde getroffen, um den Multiplexaufwand so gering wie möglich zu halten, insbesondere weil auf RU-Ebene flexible Multiplex-Möglichkeiten bestehen und diesen Nachteil ohne weiteres ausgleichen kann.

In umgekehrter Richtung erfüllt der Collector die umgekehrte Funktion und sammelt mittels einer Reihe von Multiplexern eingehende MG1/MG2-Daten zu einem statischen CG-Vektor nach den gleichen Regeln der möglichen Bitreihenfolge, siehe Abbildung 104 unten. Die ausgehenden Daten des Collectors werden in den Packer geleitet und können hier entweder direkt durchgestellt werden oder in ähnlicher nur in umgekehrter Abfolge wie beim Unpacker in 32-bit-Datenworte eingepackt. Dazu werden die eingehenden MG-Daten der Reihe nach links an den neuen Vektor angehängt und bei Erreichen von 32-bit zum FIFO transportiert. Ist der neue Vektor größer als 32-bit, so werden die rechten 32-Bits zum FIFO geleitet, während die restlichen linken Bits im Puffer des Packers verbleiben und bei der fortgesetzten linkseitigen Konkatenation der eingehenden Daten verwendet werden.

## 4.7. Anbindung an das Host-System

Bei Verwendung mehrerer IOHCs in einem HoneyComb-Array besteht die Gefahr, dass die Anzahl der Schnittstellen zum System (siehe Abbildung 96 links) stark ansteigt und durch begrenzte Bandbreite des Systems keinen zusätzlichen Nutzen bringt. Die Verwendung mehrerer IOHCs macht jedoch in einem derartigen Szenario dennoch Sinn, da dadurch die Ausführung mehrerer Applikation auf einem HoneyComb-Array vereinfacht wird. Andernfalls müssen IOFU-Programme geschrieben werden, die mehrere Applikationen im Array steuern, was zusätzlichen Aufwand für den Entwickler mit sich bringt und sich als enorm unkomfortabel erweist.

Durch die Vielzahl der Datenpins der externen IOHC-Schnittstelle (123 Pins bei einer IOHC) steigt auch die Anzahl der Pins mit der Anzahl der IOHCs stark an. Insbesondere für den Demonstrationschip mussten Verfahren entwickelt werden, um die Anzahl der externen Pins zu reduzieren. Hier war die Anzahl der verfügbaren Pins auf maximal 132 Datenpins beschränkt, da von den verfügbaren 256 Pins des ausgewählten Packages 124 Pins für die Spannungsversorgung des Chipkerns und der IO-Pins benötigt wurden.

Da jede IOHC zwei Read-Request-Data-Channels und zwei Read-Response-Data-Channels besitzt, die in ihrem Aufbau ähnlich sind und nur im Falle der Instruktionen eine zusätzliche Flush-Funktion bietet, lassen sich diese Kanäle auch für mehrere IOHCs durch Mutiplexer und Decoder bündeln. Dazu bedarf es eines Schedulers, der im realisierten Fall die Umschaltung zwischen den eingehenden Kanälen nach dem Round-Robin-Verfahren umsetzt und dem

ausgehenden Kanal eine zusätzliche ID verpasst. Diese ist notwendig, um beim Erhalt der Leseantwort vom System, die Quelle der Anfrage zur Weiterleitung ausmachen zu können. Ausgangsseitig wird weiterhin ein komplexes FIFO verwendet, welches in der Lage ist, bei einer Flush-Anfrage selektiv in Abhängigkeit der gespeicherten ID Leseanfragen die Einträge zu löschen.

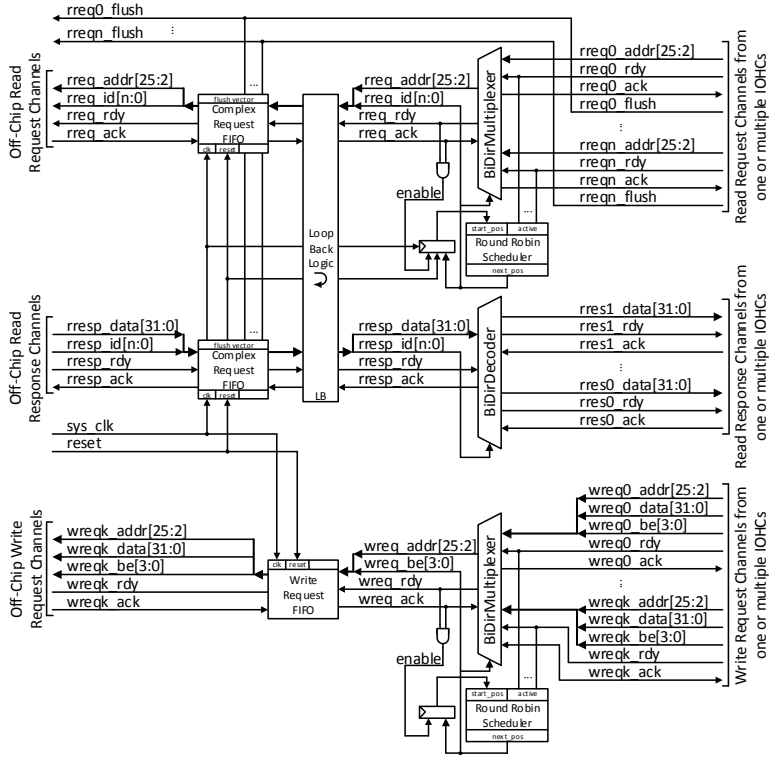



**Abbildung 105: Externe Schnittstellen der HoneyComb-Architektur realisiert durch Mutiplexing der Schreib- und Lesekanäle multipler IOHCs**

Bei der Verarbeitung der Leseranfragen muss das System die ID der Anfrage mit der Antwort an die HoneyComb-Architektur zurückschicken. Eingehende Lesedaten werden zunächst im ebenfalls komplexen Eingangs-FIFO gespeichert und anschließend durch den bidirektionalen Decoder entsprechend der ID an die Quell-IOHC durchgestellt. Auch hier wurde zum Testen der HoneyComb-Schnittstellen eine Loopback-Funktion realisiert, um die Inbetriebnahme zu erleichtern und die Timing-Gegebenheiten des verwendeten PCB und des gefertigten Chips einfacher zu bestimmen.

Auf der Schreibseite ist der Sachverhalt deutlich einfacher. Ein bidirektionaler Multiplexer selektiert mit Hilfe eines Round-Robin-Schedulers einen der aktiven Write-Request-Data-Channels und leitet die Anfragen an das Ausgangs-FIFO. Eine ID wird in diesem Falle nicht benötigt, da keine Antwort seitens der HoneyComb-Architektur auf eine Schreibanfrage erwartet wird. Nach dem Weiterleiten der Anfrage an das Host-System ist dieser Vorgang abgeschlossen. Das Host-System muss daraufhin die gelieferten Daten an die angegebene Adresse unter Beachtung der Byte-Enable-Signale schreiben.

Wir auch innerhalb des HoneyComb-Array sind alle externen Schnittstellen des Demonstrationschips durch ein Handshake-Protokoll geschützt.

## 4.8. Zusammenfassung

Das vorliegende Kapitel beschreibt das Konzept und den Aufbau der dynamisch rekonfigurierbaren HoneyComb-Architektur, die im Rahmen des DFG Schwerpunktprogramms 1148 im Projekt AMURHA entwickelt wurde. Basierend auf eigenen Erfahrungen und Studium weiterer akademischer und kommerzieller rekonfigurierbarer Architekturen, insbesondere array-basierter Ansätze, wurden neue Algorithmen und Strukturen in Hardware umgesetzt und anhand ausgewählter Testapplikationen getestet. Der innovative Ansatz dieser Architektur beginnt bereits in der Definition der Zellstruktur, die statt vier Seiten auf eine hexagonale Zellstruktur aufsetzt. Dadurch ist es möglich mit bis zu 40% weniger Ressourcen für Verbindungen zwischen Zellen zu verbrauchen und auch von der geringen Zahl der Hops zwischen verbundenen Zellen zu profitieren.

Insgesamt wurden drei Zelltypen für diese Architektur definiert. Die Datenpfadzellen (DPHCs) sorgen für die Verarbeitung der Daten und besitzen dafür eine Reihe von ALUs, LUTs und Registern. Das lokale Netzwerk ist in zwei Teile unterteilt und unterstützt sowohl grobgranulare 32-Bit-Datenworte als auch feingranulare 1-Bit-Signale. Dies ermöglicht sowohl die Realisierung arithmetischer wie auch boolescher Funktionen und deren Interaktion innerhalb einer Datenpfadzelle. Dadurch lassen sich mächtige Steuerabläufe programmieren und durch die Multikontextfähigkeit der ALUs Probleme sowohl in die Fläche als auch in die Zeit abbilden. Durch den Austausch der Daten zwischen den Zellen lassen sich komplexe Applikationen im Array aufbauen, die von diesen Eigenschaften profitieren. Zelleinterne ALUs unterstützen vorzeichenbehaftete wie auch vorzeichenlose Datentypen und bei Bedarf auch Fließkommazahlen. Spezielle Multiply-Shift und Divide-Shift Operationen erlauben die Behandlung von Festkommazahlen.

Die Speicherzellen (MEMHCs) bieten lokalen Speicher im Array an, die teils durch mehrere implementierte Speichermodule ihren Dienst als RAM,

FIFO oder LIFO anbieten. so dass die Applikationen im Array von der hohen Bandbreite profitieren können. Bei Bedarf können mehrere kleine Speichermodule einer Speicherzelle zu einer größeren Einheit logisch zusammengefasst und einer Anwendung transparent zur Verfügung gestellt werden. Innerhalb der Speicherzelle können sowohl grobgranulare auch feingranulare Daten gespeichert werden.

Der dritte Zelltyp, die Ein-/Ausgabezelle (IOHC), stellt die Schnittstelle zum System dar und erlaubt den Transfer von Daten und Konfigurationen zwischen dem HoneyComb-Array und dem Host-System. Zu diesem Zweck wurde ein spezialisierter µController basierend auf einer sechsstufigen Pipelinearchitektur entwickelt, die neben der Verarbeitung von RISC-Befehlen auch zusätzliche Logik zur Realisierung von Blocktransfers für grob- und feingranulare Daten ermöglicht. Daneben werden auch Packing/Unpacking-Funktionen für beiden Datentypen angeboten, um die Effizienz der Speicherplatzausnutzung zu steigern.

Eine der innovativen Entwicklungen dieser Architektur stellt das adaptive Routing zur Laufzeit dar, welches direkt in Hardware umgesetzt wurde. Dafür besitzt jede Zelle dieser HoneyComb-Architektur eine Routing-Unit (RU), die direkt mit ihren sechs Nachbarn über mehrere Links und dem Kern der Zellen, den funktionalen Einheiten (FUs) verbunden ist. Die funktionalen Einheiten (FUs) sind in Form der Datenpfadzellen (DPFUs), Speichermodule (MEM-FUs) oder Systemschnittstellen (IOFUs) ausgeführt. Die Routing-Units sind in dieser Struktur in der Lage, basierend auf den eigenen Koordinaten und den Koordinaten der Zielzellen, den Verlauf der Routen zwischen zwei Zellen zu kalkulieren und etablieren. Auf diese Weise können zwischen den funktionalen Einheiten der Zellen Punkt-zu-Punkt-Verbindungen aufgebaut werden, die alle verwendeten Ressourcen für diese Kommunikation exklusiv nutzen und anschließend wieder freigeben können. Die Steuerung der Routing-Prozesse erfolgt über spezielle Routing Instruktionen, die sowohl für grobgranulare als auch multigranulare (n-Bit) Verbindungen nutzbar sind.

Die Beschreibung der HoneyComb-Architektur erfolgte in VHDL. Das resultierende Modell bietet eine Vielzahl an generischen Parametern, die nahezu alle Eigenschaften der Architektur verändern und den zugrundeliegenden Applikationen anpassen können. Die Grundstruktur der HoneyComb-Architektur bleibt jedoch durch die Parametrisierung unberührt. Zur Entwicklung von Applikationen wurden zwei Programmiersprachen entwickelt, der HoneyComb-Assembler und die HoneyComb-Language. Die erste Programmiersprache ist eine Low-Level-Programmiersprache, während die zweite eine höhere Abstraktion bietet und eine VHDL-ähnliche Semantik aufweist. Für beide Sprachen wurden Compiler entwickelt und für Demonstrationsapplikationen genutzt. Zur Verifikation der HoneyComb-internen Abläufe wurde ein

zyklusgenaues Visualisierungswerkzeug, der HoneyComb-Viewer, entwickelt, das auf Log-Dateien aus VHDL-Simulationen aufsetzt und die internen Abläufe visualisiert.

Im Zuge der Entwicklung der HoneyComb-Architektur wurden Demoapplikationen erstellt, die die Machbarkeit der Architektur demonstrieren. In der letzten Phase des Projektes ist ein Demonstrationschip designt worden und wurde dann in TSCM 90nm Standardzellentechnologie gefertigt.

Im anschließenden Kapitel werden anhand der ausgewählten Demoapplikationen die Ergebnisse der HoneyComb-Architektur präsentiert und die Verlustleistung, Flächen und Performance mit anderen Technologien verglichen.

# 5. Anwendungen der HoneyComb-Architektur

Zur Verifikation der Funktionstüchtigkeit und als Proof-of-Concept wurden auf die HoneyComb-Architektur mehrere Applikationen abgebildet und sowohl durch funktionale Simulation als auch Post-Synthese/Layout-Simulationen die Umsetzbarkeit und Anwendbarkeit bestätigt. Am Ende des AMURHA Projekts stand das fertige Silizium zur Verfügung, welches erfolgreich zu abschließenden Tests herangezogen wurde.

Im Laufe der Projektbearbeitung entstanden eine Reihe von Tools, die zur Unterstützung der Anwendungsentwicklung genutzt und bereits im letzten Kapitel erwähnt wurden. Da diese Arbeit sich hauptsächlich mit den Hardwareaspekten der HoneyComb-Architektur befasst, wurde auf die tiefere Beschreibung der verwendeten Tools verzichtet. An dieser Stelle soll lediglich die Vorgehensweise bei der Tool Entwicklung aufgezeigt werden. Es wurden viele Ideen entwickelt, die teilweise ihren Weg in die Umsetzung gefunden haben und teilweise aus Zeitgründen verworfen werden mussten. Es wurden viele Kompromisse geschlossen, um den Abschluss des Projekts nicht zu gefährden und zum Schluss erfolgreich einen Demonstrationschip vorweisen zu können.

Im Folgenden wird ein Überblick über die Methodik der Anwendungsentwicklung gegeben und der Designflow aufgezeigt. Anschließend werden die Demonstrationsapplikationen vorgestellt und die Ergebnisse präsentiert. Dazu gehören Applikationen wie Advanced Encryption Standard (AES), Fast-Fourier-Transformation (FFT), inverse modifizierte diskrete Kosinustransformation (iMDCT) und diskrete Wavelet-Transformation (DWT). Zum Schluss werden die Ergebnisse für den Demonstrationschip diskutiert und eine Zusammenfassung gegeben.

## 5.1. Einleitung

Das AMURHA-Projekt hatte hauptsächlich die Entwicklung einer Hardware-Architektur zum Ziel, die mehrere bestehende und bereits erprobte als auch neue Konzepte in einer Architektur vereint. Um jedoch einen Proof-of-Concept zu liefern, war es notwendig eine formale Programmiermethode zu entwickeln, die es erlaubte Anwendung auf diese Architektur abzubilden. Damit entstanden im Wesentlichen zwei Programmiersprachen: HoneyComb-

Assembler (HCA) und HoneyComb-Language (HCL). Während die erste Programmiersprache grundsätzlich auf die strukturelle Beschreibung der Anwendungen ausgerichtet war, setzte HCL auf eine abstraktere Methode, die ähnlich wie VHDL prozessbasiert aufgebaut ist. Hierbei wird jeder Zelle einer HoneyComb-Applikation ein Prozess zugeordnet, wobei die Beschreibung innerhalb des Prozesses von den Zellgegebenheiten unabhängig ist. Es war an dieser Stelle nicht wichtig, welche Funktionseinheiten die beschriebenen Operationen schlussendlich ausführen, denn diese Zuordnung übernahm der HCL-Compiler, der ebenfalls parallel im Projekt entwickelt wurde. Die Ausgabe des HCL-Compilers was eine HCA-Beschreibung, die durch den Assembler in den Maschinencode übersetzt wurde. Dies schloss alle im vorigen Kapitel beschriebenen Layer der HoneyComb-Architektur ein. Das finale Programm sorgte dafür, dass die IOHCs die Daten für das Array lieferten (Transport Layer), die notwendigen Routen im Array aufgebaut als auch zu gegebener Zeit wieder abgebaut wurden (Communication Layer) und die Functional Units in den Zellen die notwendigen Berechnungen ausführten (Configuration Layer), wie im vorigen Kapitel vorgestellt

Bei der Ausführung des Programm-Codes in einer Simulation konnte das VHDL-Modell der HoneyComb-Architektur Zustandsinformationen in Log-Dateien schreiben, die im HoneyComb-Viewer visualisiert werden konnten. Dadurch bekam der Entwickler die Möglichkeit, das massiv-parallele Geschehen innerhalb des Arrays einzusehen und seine Applikationen gegebenenfalls anzupassen. Die Funktionalität des HC-Viewers ist weitreichend und eignet sich zur detaillierten zyklusgenauen Darstellung der Abläufe im Array. Darüber hinaus lassen sich die Zustände der Funktionseinheiten und Register abfragen.

Der aktuelle Stand der Tools erlaubt es aber noch nicht vollständig ohne Eingriff des Entwicklers von HCL in Maschinencode zu kompilieren. Der HCL-Compiler liefert noch nicht die notwendige Qualität des Codes und ebenso ist der spezifizierte Funktionsumfang noch nicht erreicht. Jedoch eignet sich der erzeugte HCA-Code des Compilers als gute Vorlage, um die notwendigen Änderungen und Optimierungen vorzunehmen, ohne die komplette Applikation in HCA schreiben zu müssen. Dies war auch der Ansatz für die Entwicklung der Demo-Applikationen zum Ende des AMURHA-Projektes.

Die erste Version der Applikationen entstand in diesem Zusammenhang in HCL unter Berücksichtigung einfacher Komplexitätsregeln. So sollten die finalen DPHCs nicht mehr als 4-5 unabhängige Multiplikationen bieten, damit die Größe der Zellen im Rahmen bleiben. Stattdessen wurde auf zeitlich aufeinanderfolgende Multiplikationen durch die Multikontext-Funktionalität gesetzt. Auch die übrigen Operationen wurden während der Entwicklung im Auge behalten. Ähnliches gilt für die Speichernutzung in MEMHCs. Die An-

zahl paralleler Speichermodule wurde auf acht festgelegt und auf 1024x32Bits begrenzt. Diese Tatsache schränkte die maximale Punktzahl der FFT- und iMDCT-Applikationen, wohingegen AES hinsichtlich der Fläche von kleineren Speichermodulen mit 256 Einträgen profitieren würde.

Nach der Kompilierung der HCL-Programme lassen sich in HCA weitere Optimierungen vornehmen. So lassen sich die Multiplikationen immer auf die unteren HCALU-Module (0,1, ...) abbilden. Darüber hinaus lassen sich die Zugriffmuster der HCALUs/HCLUTs auf die Register systematisieren und somit den Overhead durch reduzierte Flexibilität einschränken. Bei diesem Modell können die HCALUs auf die Register mit dem gleichen Index zugreifen und darüber hinaus auf einen Registerblock oberhalb der maximalen HCALU-Zahl schreiben. Ein ähnliches Vorgehen wurde bei den HCLUTs definiert und durchgezogen. Basierend auf den HCA-Beschreibungen wurden mit dem Super-CFG-Generator jeweils die HoneyComb-Architektur-Versionen erzeugt und damit die Tests der Applikationen durchgeführt. Beim Einsatz des Super-CFG-Generators wird zugleich auch die Konfiguration für den HC-Viewer generiert, so dass die Aktivitäten in Array visualisiert werden konnten.

Nachdem die vier Applikationen erstellt und getestet wurden, wurde mittels des Super-CFG-Generators die finale Version des VHDL-Modells für den Demonstrationschip erzeugt. Zunächst wurde auf eine homogene Lösung hingearbeitet, wobei sich herausgestellt hat, dass die resultierende Fläche des Siliziums die maximal-verfügbare Fläche des Mini-ASIC-Programms von Interuniversity Microelectronics Centre (IMEC) [123] von 16mm² übersteigt. Daher wurde im Abschluss auf die inhomogene Lösung umgeschwenkt, und die Fläche unter die maximale Fläche gedrückt.

Die inhomogene Lösung hatte den Preis, dass die Applikationen nicht mehr beliebig auf das Array platziert werden können. Die Funktionalität der IOHCs/DPHCs/MEMHCs wie auch die Routing-Ressourcen wurden stark auf die Zielapplikationen zugeschnitten. Das Array ist immer noch symmetrisch, so dass für jede Zelle eine alternative Platzierung existiert. Abbildung 118 veranschaulicht die finale HoneyComb-Konfiguration auf dem Chip.

Nachfolgend werden die erstellten Applikationen vorgestellt und gegen äquivalente Implementierungen im ASIC bzw. FPGA, als auch CPU verglichen. Diese Applikationen wurden zwecks Vergleichbarkeit erstellt und basieren auf 90nm Standardzellentechnologie, was im Falle des FPGA auf den Stratix II (Modell EP2S180F1508C3) von Altera zutrifft.

## 5.2. Fast-Fourier-Transformation (FFT)

Die Fast-Fourier-Transformation befasst sich mit der effizienten Berechnung der Fourier-Transformationen, die allerdings mit diskreten Zahlen rechnet. Sie rechnet Samples aus dem Zeitbereich in den Frequenzbereich um und ermöglicht eine Reihe von Anwendungen, die von Signalanalyse, Telekommunikation, Bildverarbeitung bis hin zur Lösung mathematischer Probleme reichen. Durch die Vielfalt der Anwendungen ist dieser Algorithmus sehr beliebt bei Abbildungen auf neue Architekturen, da die Beschleunigung oder auch die sehr effiziente Implementierung auf Zuspruch stößt. Die Abbildung auf die HoneyComb-Architektur wurde ausgewählt, da die Implementierung sowohl grobgranulare als auch feingranulare Aspekte bietet und gepaart mit Multikontext-Fähigkeiten der Architektur eine platzsparende Abbildung erreicht werden konnte.

### 5.2.1. Grundlagen

Die Theorie hinter der Fourier-Transformation besagt, dass jedes Signal sich durch eine Überlagerung von Sinus-Schwingungen darstellen lässt. Diese Idee hat weitreichende Konsequenzen, wie ein Ingenieur seine Systeme sieht und die bestehenden Probleme heutzutage löst. Durch eine Fourier-Transformation lassen sich Signalverläufe mit folgender Gleichung vom Zeitbereich in den Frequenzbereich umrechnen:

$$S(f) = \int_{-\infty}^{\infty} s(t) \cdot e^{-i2\pi f t} dt$$

Die Ausgangsfunktion gibt die Frequenzanteile des Eingangssignals wieder und hilft dem Ingenieur, eine frequenzabhängige Analyse durchzuführen. Die vorliegende Funktion hat allerdings den Nachteil äußerst rechenintensiv zu sein und eignet sich daher nicht direkt für eine diskrete Betrachtung wie sie heute in digitalen Systemen meist vorzufinden sind. Zu diesem Zweck wurde die diskrete Fourier-Transformation (DFT) eingeführt, die auf folgender Gleichung aufsetzt:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi k \frac{n}{N}}$$

Diese Gleichung erlaubt es Berechnungen mit diskreten Werten durchzuführen, bietet allerdings eine hohe Komplexität von $O(N^2)$. Um diesem Problem zu begegnen haben J.W. Cooley und John Tukey den Cooley-Tukey-Algorithmus eingeführt, der es erlaubt die Fourier-Transformation mit einer

Komplexität von O(*N log(N)*) rekursiv durchzuführen. Auf die Wiedergabe der Herleitung soll hier verzichtet und stattdessen auf die Publikation der Autoren verwiesen werden [116]. Die entscheidenden Gleichungen für eine Radix-2-Implementierung, wie dies bei verwendeten Knoten (Butterflys) mit zwei Ein-/Ausgängen (Radix-2) genutzt wird, lauten wie folgt:

$$X_k = E_k + e^{-\frac{i2\pi}{N}k}\mathrm{O_k}$$

$$X_{k+\frac{N}{2}} = E_k - e^{-\frac{i2\pi}{N}k}\mathrm{O_k}$$

Diese Gleichungen beschreiben die Funktion eines Butterflys innerhalb eines Netzwerks von Butterflies mit N Eingängen und N Ausgängen. Wichtig hierbei ist, dass die Anzahl der Eingänge einer Zweierpotenz entspricht. Die Variablen $E_k$ und $O_k$ entsprechen den geraden und ungeraden Anteilen der rekursiven Funktion und definieren die Zusammenschaltung des gesamten Netzwerks. Folgende Abbildung repräsentiert exemplarisch eine Radix-2-Implementierung für eine FFT mit 8 Ein-/Ausgängen:

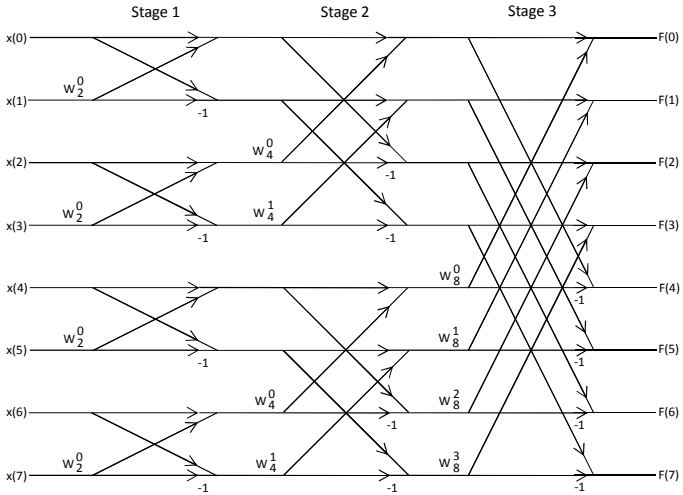



**Abbildung 106: Radix-2 Butterfly-Netzwerk für eine FFT mit N=8 Ein-/Ausgängen**

Die Umsetzung der FFT Applikation auf der HoneyComb-Architektur ist in der Lage bis zu 1024-Punkte zu berechnen, was alleine durch die verwendeten Speichermodule vorgegeben ist und jeder Zeit auf RTL angepasst werden kann.

### 5.2.2. Realisierung

Die Limitierungen der vorgegebenen Chip-Fläche für den Demonstrator erlaubt es nur eine iterative Variante der Applikation auf die HoneyComb abzubilden. Bei diesem Ansatz wird nur ein Butterfly für die komplette Applikation genutzt und die Daten in einem Speicher zwischengespeichert, siehe Abbildung 107. Durch einen Controller werden aus dem Speicher die für den nächsten Rechenschritt notwendigen Daten gelesen, zweifach parallelisiert und dem Butterfly zugeführt. Dies betrifft sowohl die Koeffizienten W wie auch die für diesen Rechenschritt notwendigen Daten. Durch die Verwendung von Ein-Port-Speichern in dieser Version der HoneyComb-Architektur, müssen die zwei benötigten Datensätze nacheinander aus dem Speicher gelesen und im Deinterleaver parallelisiert werden. Infolgedessen sinkt die Rechenleistung um die Hälfte. Dies war aber eine Designentscheidung, um den Projektfortschritt nicht zu gefährden. Dem erfolgten Proof-of-Concept tat es keinen Abbruch und kann in der nächsten Version der Architektur als Verbesserung eingeführt werden.

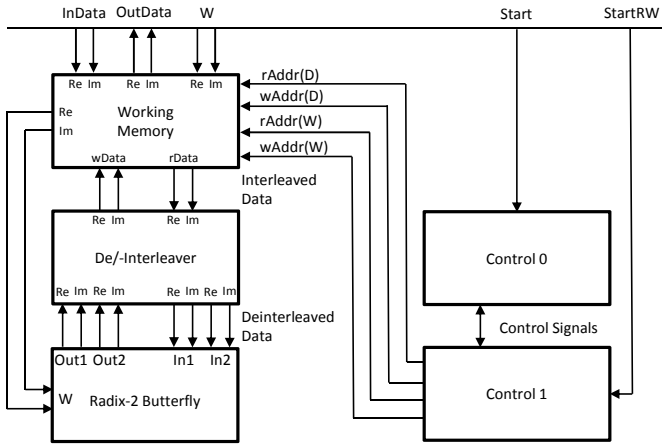



**Abbildung 107: Funktionale Aufteilung der Funktionsblöcke auf die HoneyComb-Zellen und der Kommunikationsverbindungen zwischen den Zellen**

Die komplette Abbildung auf die HoneyComb-Architektur nahm vier DPHCs, eine MEMHC und eine IOHC in Anspruch. Der Ablauf der Applikation sieht wie folgt aus: Nach der Konfiguration der Applikation werden zunächst einmal die vorberechneten Koeffizienten in den dafür vorgesehenen Speicher übertragen. Daraufhin werden vom System die zu transformierenden Daten geliefert oder von der IOHC aus dem Arbeitsspeicher aktiv gelesen und

im array-internen Speicher abgelegt. Mit dem Start-Flag signalisiert die IOHC das Ende der Übertragung, woraufhin der Controller (bestehend aus zwei Zellen) beginnt die Adressen für die Daten und Koeffizienten zu berechnen. Die gelesenen Daten werden an das Butterfly gesendet und die Ergebnisse im Speicher abgelegt. Die notwendigen Zugriffsmuster werden vom Controller erzeugt und somit der Ablauf der Applikation gesteuert.

Folgende Abbildung veranschaulicht die Aufteilung der Funktionen auf dem Array. Die linke Bildhälfte enthält zusätzlich Markierungen, um die Funktionsblöcke herauszustellen, während die rechte Bildhälfte die Abbildung direkt aufzeigt. Der in beiden Darstellungen dargestellte Ausschnitt des Array repräsentiert den rechten Teil der HoneyComb-Architektur und stellt eine der beiden Möglichkeiten der FFT-Abbildung. Durch die gewählte Symmetrie der Architektur ist es trotz des inhomogenen Charakters möglich, die Anwendung auf der linken Seite des Array auszuführen.

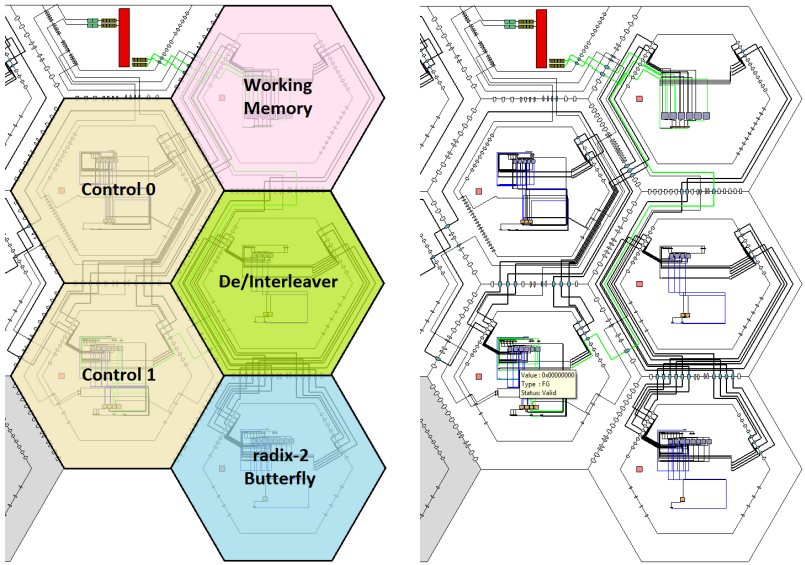



**Abbildung 108: Ausschnitt der Abgebildeten FFT-Applikation mit markierten (links) und nicht markierten (rechts) Funktionsblöcken**

Das gewählte Datenformat für die Applikation ist das Fixed-Point-Format, welches ebenfalls in der Referenzapplikation und den ASIC/FPGA-Lösungen verwendet wurde. Die Position des Dezimalpunktes kann nach Belieben und geforderter Genauigkeit in HCA eingestellt werden. Es wurde viel Wert darauf gelegt, dass die Implementierung auf der HoneyComb-Architektur den Re-

chenoperationen in C der Referenz-Implementierung entspricht und somit vergleichbar ist.

### 5.2.3. Ergebnisse

Die Ergebnisse dieser Anwendung auf der HoneyComb-Architektur bei 100MHz gibt folgende Tabelle wieder:

**Tabelle 18 HC-Daten für FFT1024 bei 100MHz**

| Application | DPHCs | MEMHCs | Config. Time | Performance |
|---|---|---|---|---|
| FFT1024 | 4 | 1 | 7,65 µs | 10850 blocks/s |

Die Verlustleistung verwendeter Zellen liegt in diesem Fall bei 126,139 mW und benötigt eine Fläche von 4430674,71 µm². Diese Angaben schließen alle beteiligten Zellen dieser Anwendung ein. Die Referenzapplikation in C eignet sich nicht direkt zu einem Leistungsvergleich, da die Implementierung nicht optimiert ist. In diesem Zustand erreicht ein moderner Intel Core i7-3930K Prozessor 10417,75 Blöcke/s, ausgeführt auf einem der sechs verfügbaren Kerne des Prozessors. Umgerechnet würde das eine maximale Verlustleistung pro Kern von 21,6W bedeuten, wenn man 130W TDP als Grundlage zur Aufteilung auf die sechs Kerne nimmt.

Im Vergleich dazu erreicht eine vergleichbare FPGA-Implementierung des iterativen FFT-Ansatzes ca. 98 MHz und benötigt dafür 1019 ALUTs, 569 Register, 131092 Speicherbits und 36 DSP Elemente bei gesamter Verlustleistung von 1651,89 mW. Die erzielbare Rechenleistung liegt hierbei bei 7975,26 Blöcken/s.

Wie erwartet fällt die ASIC Lösung im Vergleich deutlich effizienter aus. Bei einer Fläche von 615930.236 µm², die nur 1/7 der HoneyComb-Fläche entspricht, und einer Gesamtverlustleistung von 25.14 mW bei 500MHz, sind maximal 934,579 MHz erreichbar. Die Rechenleistung liegt in diesem Falle bei 40690,1 Blöcke/s bei 500MHz und 76056,2 Blöcke/s bei 934,579 MHz.

Wie erwartet ist die ASIC-Lösung mit Abstand die effizientere Umsetzung. Flächenvergleiche zwischen der HoneyComb-Architektur und der FPGA-Lösung lassen sich nur schwer bewerkstelligen, da die genauen Flächendaten des verwendeten Stratix II FPGAs nicht publiziert sind. Wichtig ist hierbei, dass der FPGA ebenfalls mit 90nm Technologie hergestellt ist. Hinsichtlich der Verlustleistung bringt die HoneyComb-Architektur hier deutliche Vorteile.

Um die Vergleichbarkeit der Lösungen sicherzustellen, arbeiten alle Kandidaten mit gleichen Datentypen und identischer Genauigkeit. Selbst der iterative Ansatz wurde in ASIC und FPGA umgesetzt, um den Flächenlimitierun-

gen der HoneyComb-Architektur Rechnung zu tragen. Ein ähnliches Vorgehen wurde bei den anderen drei Applikationen umgesetzt.

## 5.3. Inverse Diskrete Cosinus-Transformation (iMDCT)

Die modifizierte diskrete Cosinus-Transformation [117] gehört zur Klasse der diskreten Cosinus-Transformationen mit dem Unterschied, dass sie auf fortlaufende überlappende große Datensätze angewendet werden soll, insbesondere um Randartefakte zu vermeiden. Dieser Algorithmus wird in Audio-Codecs wie OggVorbis [119] eingesetzt, um eine hohe Kompression mit geringen hörbaren Verlusten zu erreichen. Zum Dekodieren wird hier die inverse modifizierte Cosinus-Transformation genutzt, um einen komprimierten Datenstrom für unser Ohr zugänglich zu machen. Folgende Gleichung beschreibt die inverse Transformation:

$$x(n) = \sum_{k=0}^{M-1} X(k) cos\left[\left(n + \frac{M+1}{2}\right)\left(k + \frac{1}{2}\right)\frac{\pi}{M}\right], mit\ n\ =\ 0, 1, \ldots, N-1$$

Da die direkte Berechnung dieser Gleichung, ebenso wie FFT eine Komplexität von O($N^2$) aufweist, ist es ratsam zuvor eine Umformung vorzunehmen. Die Lösung von Nikolajevic und Fettweis [118] bietet einen nützlichen rekursiven Ansatz, der in der folgenden Abbildung durch ein Datenflussdiagramm dargestellt ist:

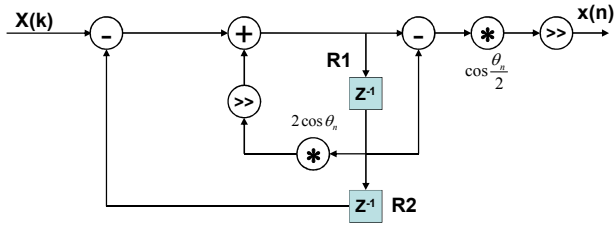



**Abbildung 109: Rekursiver Ansatz zur Berechnung der iMDCT nach Nikolajevic und Fettweis**

Die resultierende Komplexität ist vergleichbar mit der Ausgangsgleichung, allerdings lässt sich diese Lösung in Hardware praktisch umsetzen. Im Gegensatz zur FFT arbeitet iMDCT mit reellen Werten.

### 5.3.1. Realisierung

Der vorgeschlagene Ansatz lässt sich nicht direkt auf die HoneyComb-Architektur abbilden, ohne die Arbeitsfrequenz deutlich zu senken. Bei mehreren Operationen zwischen Registern, würden zu lange kritische Pfade entstehen. Eine mögliche Lösung dafür besteht in einer weiteren Transformation des Datenflussgraphen, in dem weitere Register eingefügt werden, siehe Abbildung 110. Durch die Umformung entstehen drei Kontexte in der Schleife, die durch eine vollständige Implementierung genutzt werden können oder durch eine Abbildung auf zwei HCALUs platzsparend platziert werden kann. In der aktuellen Implementierung wurde auf die letzte Möglichkeit zurückgegriffen und die Multikontext-Funktionalität der HCALUs ausgenutzt.

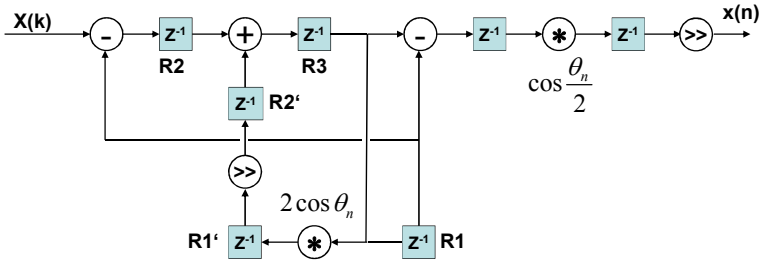



**Abbildung 110: Modifizierte Version des rekursiven Ansatzes zur Abbildung auf die HoneyComb-Architektur**

Der Aufbau der abgebildeten Anwendung ist ähnlich gestaltet wie im Falle der FFT. Es gibt einen Speicher für Koeffizienten und Arbeitsdaten. Das Data-Spread-Modul sorgt für die Verteilung der Daten an die Finger, welche jeweils die Implementierung des in Abbildung 110 dargestellten Datenflussdiagramms realisiert. Das Control-Modul steuert die Abläufe, indem Schreibadressen erzeugt werden, während die Applikation mit vorberechneten Koeffizienten sowie neuen Arbeitsdaten geladen wird und Leseadressen während der Durchführung der Berechnungen generiert werden. Die Finger schicken die Ergebnisse direkt an die IOHC und damit das System, da ihre Ausgangswerte direkt die Ergebnisse darstellen, siehe Abbildung 111.

Das Besondere an dieser Abbildung ist die Fähigkeit der Applikation Finger bei Bedarf hinzuzufügen und wieder zu löschen. Zu diesem Zweck sind die Controller und das Data-Spread-Module entsprechend ausgelegt und können nach erfolgter Rekonfiguration durch das System zur Nutzung zusätzlicher Finger veranlasst werden.

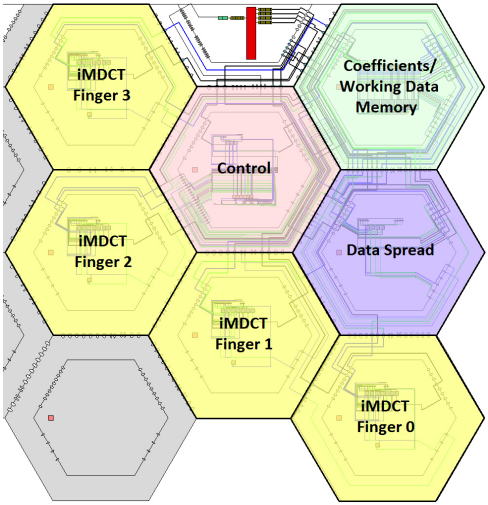



**Abbildung 111: Abbildung der iMDCT am rechten Rand der HoneyComb-Architektur mit markierten Funktionseinheiten**

Die Umsetzung auf der HoneyComb-Architektur arbeitet wie bei FFT mit Fixed-Point-Datentypen, die zur Compile-Zeit einstellbar sind. Die Koeffizienten können zur Steigerung der Genauigkeit eine passende Punktposition aufweisen und liegen bei aktueller Implementierung an Position 29.

Im Falle des OggVorbis-Dekoders werden 1024-Spektralwerte in 2024-Samplewerte umgerechnet. Ein derartiger Datensatz wird im Folgenden als ein Block bezeichnet.

### 5.3.2. Ergebnisse

Nachstehende Tabelle gibt die Leistungsfähigkeit von iMDCT auf der HoneyComb-Architektur wieder. Zum einen ist eine 1-Finger-Variante vertreten und zum anderen eine 7-Finger-Variante, die neben den vier möglichen Fingern an einem Controller, einen weiteren Controller auf der anderen des HoneyComb-Arrays voraussetzt, der drei weitere Finger für seine Berechnungen nutzt. Die Summe entspricht in diesem Fall der Zahl sieben und stellt für den Demonstrationschip die maximale Leistungsfähigkeit hinsichtlich der iMDCT-Applikation dar. Ebenso sind in diesem Fall die Verlustleistung und die Fläche beträchtlich, da fast das gesamte Array zum Einsatz kommt:

**Tabelle 19 HC-Daten für die iMDCT Anwendung**

| Application | DPHCs | MEMHCs | Config. Time | Performance |
|---|---|---|---|---|
| iMDCT 1xfinger | 3 | 1 | 24,06 µs | 47,6 blocks/s |
| iMDCT 7xfingers | 11 | 2 | 25,60 µs | 333,46 blocks/s |

Die gesamte Fläche für die ein-Finger-Variante liegt bei 3775158,357 µm² und schließt lediglich Finger 0 mit ein. Die Verlustleistung in diesem Fall liegt bei 118,357 mW. Die 7-Finger-Variante benötigt hingegen bereits 9816255,928 µm² und verbraucht dabei 195,02 mW.

Die Software-Version des Algorithmus eignet sich nicht zum direkten Leistungsvergleich, da auch hier keine optimierte Version des Algorithmus vorlag. Hingegen konnte diese Implementierung zur Verifikation der Ergebnisse herangezogen werden und erfüllte die Äquivalenz der Resultate zu 100%.

Die FPGA-Version des Algorithmus lieferte eine maximale Arbeitsfrequenz von 285MHz. Dabei betrug die Verlustleistung 1831.92 mW. Es wurden weiterhin folgende Ressourcen gebraucht: 1087 ALUTS, 563 Register, 98304 Speicherbits und 16 DSP Blöcke. Dazu muss angemerkt werden, dass die Finger ebenso wie beim ASIC nach Abbildung 109 umgesetzt wurden und analog zur HoneyComb-Implementierung einen Controller und zusätzlichen Speicher beinhalten und dadurch einen höheren Ressourcenbedarf aufweisen. Die Rechenleistung des FPGA liegt bei maximal 134,4 Blöcken/s.

Im Falle des ASIC beträgt die resultierende Fläche 423456,355 µm² und die Verlustleistung 83.390 mW bei 1GHz. Die maximal erreichbare Arbeitsfrequenz liegt bei 1,724 GHz. Die erreichbare Rechenleistung beträgt 471,8 Blöcke/s bei 1GHz und 821,6 Blöcke/s bei 1,724 GHz.

Die Tendenz ist klar erkennbar. Die ASIC-Lösung ist hinsichtlich Fläche, Verlust- wie auch Rechenleistung deutlich überlegen. Die 1-Finger-Variante der HoneyComb-Architektur steht im Vergleich zur FPGA-Lösung gut dar, lediglich die maximale Leistung des FPGA konnte nicht erreicht werden, was aber in keinem Verhältnis zum Verlustleistungsvergleich zwischen den beiden Architekturen steht.

## 5.4. Diskrete Wavelet Transformation (DWT)

Bei Wavelet Transformationen handelt es sich um eine weitere mathematische Zeit-Frequenz-Transformation, die jedoch ihre Frequenzinformationen lokale vorhält und sich daher hervorragend für die Bildkompression eignet. Aus diesem Grund wird dieses Verfahren insbesondere in JPEG2000 eingesetzt, um eine hohe Qualität bei gleichzeitig hoher Kompression zu erreichen.

Folgendes Bild veranschaulicht die Anwendung der Wavelet-Transformation auf ein Bild, welches zweidimensional in High- und Low-Anteile zerlegt wird. Dabei wird die absolute Punktzahl nicht verkleinert, lediglich die beiden Frequenzkomponenten pro Pixel voneinander getrennt.


**Abbildung 112: Zweidimensinale Wavelet-Transformation eines Bildes mit gefilterten High- und Low-Anteilen**

Zur Abbildung der Anwendung auf die HoneyComb-Architektur wurde die JPEG200-Referenzimplementierung herangezogen und die Wavelet-Funktion extrahiert. Die anfänglich in JAVA-vorliegende Funktion wurde in Programmiersprache C implementiert und gegen das Original verifiziert. In erster Linie handelt es sich um einen Hochpass- und einen Tiefpassfilter, welche die eingehenden Daten in zwei Datenströme aufteilen: die hochfrequenten und niederfrequenten Daten mit jeweils halber Datenrate. Wird ein Bild darauf zweidimensional angewendet, so entsteht die obere Darstellung mit vier Kacheln. Mittels Quantisierung und folgendem Kompressionsverfahren lassen sich derart transformierte Bilddaten effizient komprimieren.

Abbildung 113 zeigt die fertige Abbildung der Applikation auf die HoneyComb-Architektur. Zur Beschleunigung der Berechnungen wurde eine dreifache Abbildung für je eine Farbkomponente gewählt. Die Applikation zeichnet sich durch einen relativ geringen Ressourcenbedarf aus, ist aber sehr rechenintensiv, da ein Bild für die zweidimensionale Transformation zweimal durch die Filter muss und in der JPEG2000 Anwendung die verbleibenden LL-Anteile je nach Kompressionsfaktor zusätzlichen Durchgängen unterzogen werden können.

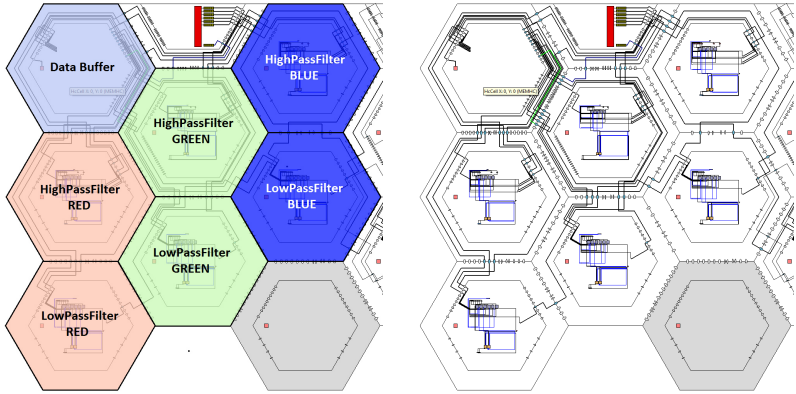



**Abbildung 113: Abbildung der Wavelet-Transformation für die parallele Verarbeitung dreier Farbkomponenten (rot, grün blau)**

### 5.4.1. Ergebnisse

Nachstehende Tabelle gibt die Leistung der Applikation auf der HoneyComb-Architektur wieder:

**Tabelle 20 HC-Daten für die Wavelet-Transformation**

| Application | DPHCs | MEMHCs | Config. Time | Performance |
|---|---|---|---|---|
| Wavelet | 6 | 1 | 3,15 µs | 60 MPixel/s |

Die hierfür benötigte Fläche beträgt unter Berücksichtigung der verwendeten Zellen 5385581,218 µm² und die Verlustleistung liegt bei 141,015 mW.

Ein Intel Core i7-3920K Prozessor mit 6 Kernen und 3,2GHz erreicht bei Nutzung nur eines Kerns 11,076 MPixel/s. Seine Verlustleistung liegt hierbei bei maximal 21,6 Watt unter Berücksichtigung der TDP von 130W für den gesamten Prozessor.

Ein FPGA der Stratix II Familie erreicht hingegen eine Arbeitsfrequenz von 294 MHz und erzielt eine maximal Rechenleistung von 294 MPixel/s. Hierfür verwendet der FPGA sehr wenig Ressourcen von 59 ALUTs und 66 Registern. Die Verlustleistung beträgt 1507,78 mW.

Im Falle des ASICs wird eine Fläche von 4364,606 µm² benötigt und eine Verlustleistung von 2.236 mW bei 1GHz umgesetzt. Die erreichbare maximale Arbeitsfrequenz liegt bei 1,428GHz. Die erzielbare Rechenleistung liegt bei 1GPixel/s bei 1GHz und 1,428GPixel/s bei 1,428 GHz.

## 5.5. Advanced Encryption Standard (AES)

Das Verschlüsselungsverfahren AES [114][115] wurde vom National Institute of Standards and Technology (NIST) [113] durch eine öffentliche Ausschreibung Ende der neunziger Jahre und schließlich im Jahr 2000 definiert und spezifiziert. Zu den Gewinnern zählen bei dieser Ausschreibung die beiden belgischen Forscher Joan Daemon und Vincent Rijmen, die ein Blockcode-Verfahren einreichten, welches die Forderungen von NIST erfüllte und eine Reihe von Tests in dieser Ausschreibung hinsichtlich der Sicherheit und Robustheit überstand. Zu wichtigen Forderungen von NIST zählten unter Anderem die effiziente Implementierbarkeit in Software wie auch in Hardware. Angesichts der Datenmengen der Zukunft sollte die Performanz des neuen Verschlüsselungsverfahrens skalierbar sein, bei gleichzeitig niedrigen Kosten. Darüber hinaus sollte AES frei sein und die Autoren auf Gebühren bei der Nutzung verzichten.

Heute ist AES aus dem täglichen Leben kaum noch wegzudenken, da die Verschlüsselung in unglaublich viele Bereiche hauptsächlich zum Datenschutz Einzug gehalten hat. Kompressionsarchive, Telekommunikationsprotokolle, IP-Telefonie, eMails, und viele mehr zählen zu einer Vielzahl von Anwendungen, die heute AES nutzen. Zur effektiven Umsetzung in Software bieten moderne Prozessoren, beispielsweise von Intel und AMD, spezielle Maschinenbefehle, um die Ausführung zu beschleunigen.

In den nachfolgenden Ausführungen wird zusammenfassend der Aufbau des AES Algorithmus vorgestellt und die Umsetzung auf der HoneyComb-Architektur präsentiert.

### 5.5.1. Grundlagen

Das AES-Verfahren basiert auf einer Blockchiffre von einer vorgegeben Blockgröße, die durch NIST auf die Größe von 128 Bits bzw. 16 Bytes mit einer 4x4 Bytes Organisation festgeschrieben wurde, siehe Abbildung 114. Die angewendete Schlüssellänge kann hingegen 128, 192 oder 256 Bits betragen, die ähnlich wie die Datenblöcke angeordnet sind.

Bevor der Schlüssel auf die Daten angewendet werden kann, muss er expandiert werden. Dies geschieht mittels des Expansionsalgorithmus (Key Expansion), der ebenso Teil von AES ist, hier jedoch nicht betrachtet wird. Das Ergebnis der Expansion ist der expandierte Schlüssel (Round Key, K), der aus einer definierten Anzahl von 128-Bit-Blöcken besteht. Die Anzahl ist durch die definierte Schlüssellänge von 128, 192 oder 256 Bits definiert und entspricht in der Reihenfolge 10, 12 und 14 Schlüsselblöcken und im Folgenden ebenso vielen Runden mit jeweils der Länge von 128 Bits.

Im Falle von AES handelt es sich grundsätzlich um eine Permutations-Substitutions-Blockchiffre. Hierbei wird durch eine vordefinierte Abfolge von Permutations- und Substitutionsfunktionen sowie mehrfacher XOR-Verknüpfungen mit den Schlüsselblöcken der eingehende Datenstrom verschlüsselt und für einen außenstehenden ohne die Kenntnis des Schlüssels unlesbar gemacht. Die Dechiffrierung erfolgt in umgekehrter Reihenfolge, und der expandierte Schlüssel wird erneut auf den ehemals verschlüsselten Datenstrom angewendet. Es handelt sich somit um ein symmetrisches Verfahren, bei dem der Sender und der Empfänger mit demselben Schlüssel arbeiten.

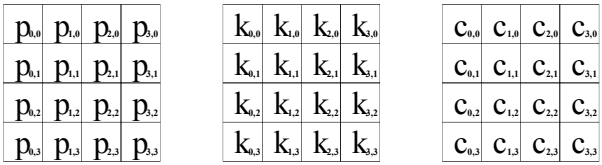



**Abbildung 114: Aufbau in AES verwendeter 128-Bit-Blöcke mit $p_{i,j}$ = Eingangsdaten, $k_{i,j}$ = Schlüsseldaten und $c_{i,j}$ = Ausgangsdaten, mit j = 0...3 , i = 0...3**

Linke Seite der Abbildung 115 zeigt den Ablauf des AES-Encoders mit seinen vier Grundfunktionen: Add Round Key, Sub Bytes, Shift Rows und Mix Columns. Durch die Schlüsselexpansion werden aus dem eingehenden Schlüssel N+2 expandierte Schlüsselblöcke erzeugt und beim Ablauf des Algorithmus den Add Round Key-Funktionen der Reihe nach zugeführt. Im Falle der Schlüssellänge mit 256 Bits entstehen daraus 15 Schlüsselblöcke, so dass die innere Schleife 13 mal ausgeführt wird, während am Eingang und Ausgang jeweils ein weiterer Schlüsselblock gebraucht wird.

Die Funktion Add Round Key realisiert die XOR-Verknüpfung der Eingangsdaten mit dem aktuellen Schlüsselblock, was byteweise erfolgt und der Funktion

$$c_{i,j} = p_{i,j} \oplus k_{i,j}$$

entspricht. Die nachstehende Funktion Sub Bytes substituiert die Daten ebenfalls byteweise durch eine vorberechnete in AES-spezifizierte Substitutionsmatrix (SBox) mit

$$c_{i,j} = SBox(p_{i,j})$$

Dies lässt sich mit einer Look-Up-Tabelle realisieren, während die Matrix den Inhalt der Look-Up-Tabelle bildet. Die Funktion Shift Rows sorgt für ein zeilenweises Vertauschen innerhalb der Datenblöcke. Dies wird nach folgendem Muster bewerkstelligt:

$$c_{i,0} = p_{i,j}$$
$$c_{i,1} = p_{(i+3)\ mod\ 4,j}$$
$$c_{i,2} = p_{(i+2)\ mod\ 4,j}$$
$$c_{i,3} = p_{(i+1)\ mod\ 4,j}$$

D.h. die erste Zeile wird belassen, wie sie zuvor war. Die zweite Zeile wird um eine Position nach links verschoben, die dritte Zeile um zwei Positionen, die vierte Zeile schließlich um drei Positionen.

Die letzte Funktion Mix Columns vermischt spaltenweise die eingehenden Daten und erzeugt die neuen Spalten des Ausgangsblocks. Hierbei wird eine modulo Multiplikation mittels des irreduziblen Polynoms $x^8+x^4+x^3+x+1$ im Galois-Körper GF($2^8$) angewendet. Die Funktion lässt sich mit einer Matrix veranschaulichen:

$$\begin{matrix} c_{i,0} \\ c_{i,1} \\ c_{i,2} \\ c_{i,3} \end{matrix} = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \begin{pmatrix} p_{i,0} \\ p_{i,1} \\ p_{i,2} \\ p_{i,3} \end{pmatrix} = \begin{pmatrix} 2p_{i,0} \oplus 3p_{i,1} \oplus 1p_{i,2} \oplus 1p_{i,3} \\ 1p_{i,0} \oplus 2p_{i,1} \oplus 3p_{i,2} \oplus 1p_{i,3} \\ 1p_{i,0} \oplus 1p_{i,1} \oplus 2p_{i,2} \oplus 3p_{i,3} \\ 3p_{i,0} \oplus 1p_{i,1} \oplus 1p_{i,2} \oplus 2p_{i,3} \end{pmatrix}$$

mit

$$1p_{i,j} = p_{i,j}$$
$$2p_{i,j} = \begin{cases} 2 \cdot p_{i,j} & \text{wenn } p_{i,j} < 2^7 \\ 2 \cdot p_{i,j} \oplus (11b)_{hex} & \text{wenn } p_{i,j} \geq 2^7 \end{cases}$$
$$3p_{i,j} = 2p_{i,j} \oplus p_{i,j}$$

Eine wichtige Eigenschaft des AES-Algorithmus ist die Tatsache, dass gänzlich auf die Verwendung von Multiplikationen verzichtet wird. Die Multiplikation mit zwei lässt sich im dualen durch eine Schiebeoperation realisieren und erzeugt somit geringe bis keine zusätzlichen Kosten, wie im Falle einer festverdrahteten unbedingten Multiplikation mit zwei.

### 5.1.1. Realisierung

Die Umsetzung des AES Algorithmus auf der HoneyComb-Architektur erfolgte durch einen iterativen Ansatz. Das heißt die Hauptschleife des Algorithmus wurde nicht aufgerollt, wie dies im FPGA oder ASIC für die maximale Leistung notwendig wäre, sondern beibehalten. Zusätzlich wurden der Initialblock (Add Round Key) und die abschließenden Funktionsblöcke (Sub Bytes, Shift Rows und Add Round Key) durch eine Umformung in die Schleife aufgenommen, siehe rechte Hälfte in Abbildung 115. Die neue Schleife beginnt immer mit der Add Round Key Funktion und endet mit Mix Columns. Im

letzten Schleifendurchgang wird Mix Columns nicht mehr ausgeführt, sondern der Datenstrom mittels der Komponenten MC Demux und MC Mux daran vorbeigeführt. Nach dem abschließenden Ausführen der Add Round Key Funktion im letzten Schleifendurchgang, werden die verschlüsselten Daten über den Ausgangsdemultiplexer (Out Demux) ausgegeben. Der Eingangsmultiplexer (In Mux) sorgt am Anfang der Berechnung, dass Eingangsdaten in die Schleife gelangen und schließt anschließend die Schleife, damit die Iterationen ausgeführt werden können. Die komplette Konfiguration benötigt zwei MEMHCs, 11 DPHCs und eine IOHC Zelle. Damit ist diese Applikation das Maß für die maximale Größe der ASIC Version dieser Architektur.

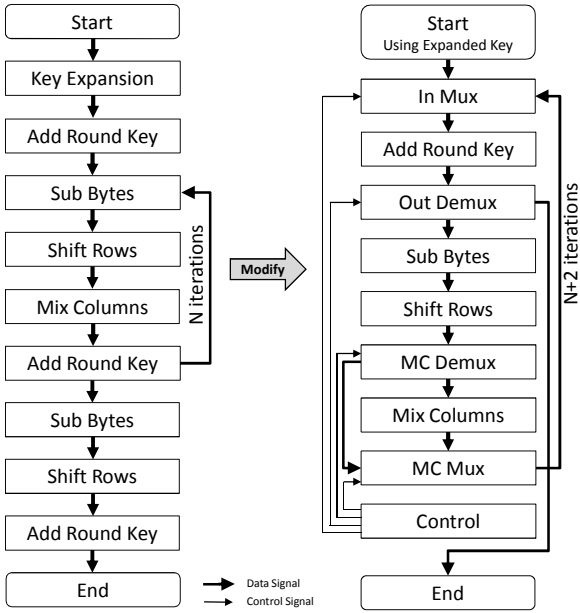



**Abbildung 115: Ablaufdiagramm des AES-Algorithmus mit links Originalablauf und rechts modifizierte Version der HoneyComb-Architektur**

Die maximale Leistung kann dieser Ansatz nur erreichen, wenn die Hardwareschleife vollständig mit Daten gefüllt ist und alle Blöcke aktiv arbeiten. Durch den iterativen Ansatz hat man den Nachteil, dass die Lade- und Entleerungsphasen die maximalerreichbare Leistung drücken, da während dieser Zeit die Schleife in Hardware nicht vollständig gefüllt ist. Eine Abhilfe stellt hier ein Pipeline-FIFO-Modul dar, welches durch eine hohe Tiefe das Verhältnis der Ladephasen zu Berechnungsphasen zu Gunsten der Berechnungsphasen verschiebt. Abbildung 116 veranschaulicht die Abbildung des iterativen An-

satzes auf die HoneyComb-Architektur mit dem erläuterten Pipeline-FIFO zwischen den Funktionsblöcken Output Demux und Shift Rows. Die Funktion Sub Bytes wurde ebenfalls in der Zelle mit den Pipeline FIFOs mittels der vorhandene Speicher realisiert. Hierfür wurden vier HCMEM-Module verwendet. Vier weitere werden für die FIFOs genutzt und bieten einen Puffer mit bis zu 1024 Einträgen.

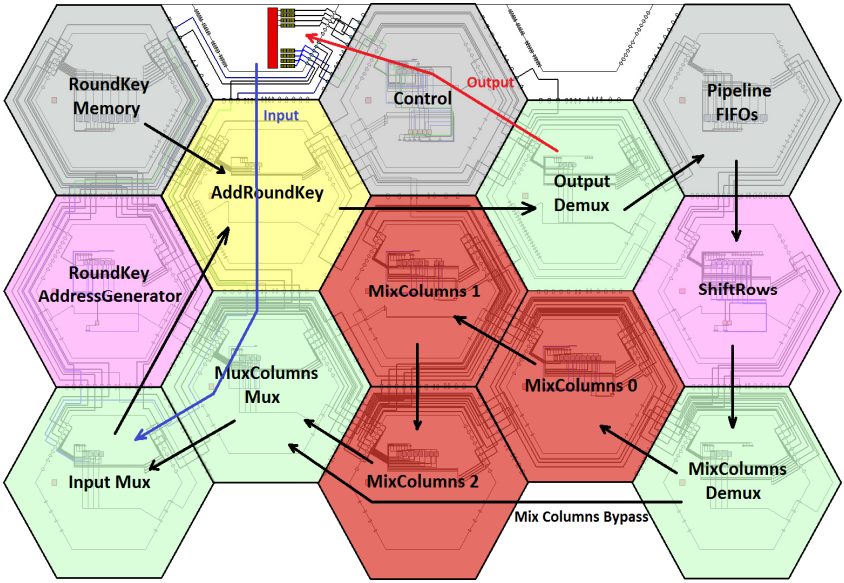



**Abbildung 116: Datenfluss der AES-Applikation des iterativen Ansatzes mit einem zusätzlichen Pipeline-FIFO zur Leistungssteigerung**

Das Control-Modul dieser Anwendung liefert die Steuersignale für die vier Multiplexer/Demultiplexer-Module und den RoundKey-Adressgenerator. Im ersten Fall wird jeweils ein Bit benötigt, das die Schaltstellung der jeweiligen Zielmodule bestimmt. Im zweiten Fall wird ein 32-Bit-Wert übertagen, welches den Adressgenerator auf die Anzahl der Datensätze in der Schleife einstellt. Angestoßen wird das Control-Modul von außen, indem die Anzahl der eingehenden Datenblöcke durch die Übertagung vorgegeben wird. Das Control-Modul startet darauf seine Arbeit und ermöglicht damit den Datenfluss in der Anwendung. Nach einem kompletten Durchlauf der Anwendung wartet das Control-Modul auf eine erneute Eingabe von außen.

Der Funktionsblock Add Round Key besteht aus drei Zellen. Der RoundKey-Adressgenerator erzeugt basierend auf der Anzahl der Datenblöcke für den Speicher mit den gespeicherten Round Keys die mit den Daten korrelie-

renden Adressen. Diese Adressen werden in der RoundKey-Memory-Zelle genutzt, um Schlüsselblöcke auszulesen und an die AddRoundKey-Zelle zu schicken. Hierfür werden vier unabhängige HCMEMs genutzt, die jeweils den kompletten Satz der Round Keys enthalten. Innerhalb der AddRoundKey-Zelle erfolgt die eigentliche XOR-Verknüpfung mit den durchlaufenden Datenblöcken.

Auch die Funktion Mix Columns benötigt insgesamt drei Zellen, da die oben beschriebenen Gleichungen sich nicht mit der erforderlichen Leistungsfähigkeit auf eine DPHC abbilden ließen, ohne die Zellgröße von vorgegebenen 9 HCALUs zu übersteigen.

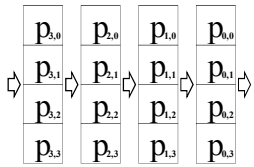



**Abbildung 117: Übertragung eines Datenblocks erfolgt in vier Taktzyklen, wobei jedes Byte einen eigenen 32-Bit-Kanal nutzt**

Das verwendete Datenformat dieser Anwendung in der HoneyComb-Architektur ist in Abbildung 117 dargestellt. Hier ist erkennbar, dass hierfür ein hoher Verschnitt hinsichtlich der Datenbreite in Kauf genommen wird. Jedes Byte in dieser Anwendung wird über einen 32-Bit-Kanal übertragen, so dass 24-Bits effektiv ungenutzt bleiben. Durch dieses Format ergibt sich auch die Zahl vier für die Anzahl der Speicher/FIFOs und die Anzahl der 32-bit-Kanäle zwischen den Funktionsblöcken.

Die Ausführung der Anwendung teilt sich in drei Phase auf: Ladephase, Ausführungsphase und die Entladephase. In der ersten Phase werden Daten in das Array zur Berechnung übertragen. In der folgenden Phase wird die Berechnung ausgeführt und schließlich in der letzten Phase die Daten an das System zurückübertagen. Danach kann die erste Phase wieder von neuem starten.

Die Anwendung läuft wie folgt ab: Nachdem die Konfiguration in das Array übertragen wurde und die Anwendung bereit ist, werden zunächst die Daten des Round Keys an die RoundKey-Memory-Zelle übertragen. Hierfür werden dynamisch acht 32-Bit-Verbindungen zwischen IOHC und der betroffenen MEMHC aufgebaut: jeweils ein Daten- und ein Adresskanal pro Speichermodul innerhalb der MEMHC. Daraufhin werden die Daten übertragen: je eine Zeile der Datenblöcke pro Speicherkanal. Nach der Übertragung werden die Verbindungen wieder gelöscht. Analog werden die Daten für die SBOX zur Zelle Pipeline FIFOs übertragen.

Soll eine Verschlüsselung gestartet werden, so wird zunächst die Menge der Daten signalisiert und der Wert über das IOHC an das Control-Modul übertragen. Das Control-Modul beginnt daraufhin die Steuersignale zu generieren. Das OutputDemux-Modul leitet die Daten von der AddRoundKey-Zelle zur PipelineFIFOs-Zelle durch, während das Input Mux-Modul zunächst die Eingangsdaten in die Schleife einspeist. Die eingehenden Daten füllen bei der Übertragung die Pipeline, die in dieser Phase vor dem InputMux gestaut werden und der Rückstau sich rückwärts durch die Pipeline propagiert und schließlich im FIFO aufgefangen wird. Die Daten wandern anschließend in dieses FIFO, bis die Übertragung abgeschlossen ist.

In der Ausführungsphase wird das InputMux-Modul auf den Schleifenbetrieb gestellt und leitet Daten vom MixColumnsMux-Modul zum AddRoundKey-Modul. Damit ist die Schleife geschlossen und die Berechnung wird Runde um Runde durchgeführt. Das Control-Modul zählt die Runden durch und erzeugt ebenso die für den Abschluss wichtigen Steuersignale.

In der letzten Runde werden die Daten an den MixColumns Zellen 0, 1 und 2 durch die MixColumnsDemux- und MixColumnsMux-Zellen verbeigeleitet. Zusätzlich wird zum gegebenen Zeitpunkt das OutputMux-Modul zur Umleitung der Daten zum IOHC und damit dem System eingestellt. Sobald alle Daten zum IOHC durchpropagiert sind, ist die aktuelle Berechnung abgeschlossen und eine neue kann gestartet werden.

### 5.1.2. Ergebnisse

Die erzielbare Leistung der HoneyComb-Architektur kann der folgenden Tabelle entnommen werden:

**Tabelle 21 HC-Daten für die AES-Applikation**

| Applikation | DPHCs | MEMHCs | IOHCs | Config. Time | Performance |
|---|---|---|---|---|---|
| AES256 | 11 | 2 | 1 | 6,85µs | 25,6MBytes/s |

Hierbei wird mit einer Ausnahme das vollständige Array genutzt. Nur eine IOHC ist in diesem Fall an den Berechnungen nicht beteiligt. Die gesamte Fläche des aktiven Array beträgt 9409465,524 µm² und die Verlustleistung liegt bei 206,146 mW. Zum wichtigsten Nachteil der HoneyComb-Abbildung gehört die Tatsache, dass es sich um eine 32bit-Architektur handelt, die keine speziellen 8-bit-Befehle bietet. Dadurch fällt beim Byte-Transport zu viel Aktivität an, was sich negativ auf die resultierende Verlustleistung auswirkt. Durch die limitierte Größe des Arrays besteht ebenfalls keine Möglichkeit AES als eine vollständige Pipeline abzubilden. In diesem Fall wäre die Effizienz deutlich höher, da alle Mux/Demux-Module wie auch das Control-Modul

nicht nötig wären. Die maximal erreichbare Rechenleistung für eine vollständige AES-Pipeline liegt bei 400MBytes/s. Hierfür muss die HoneyComb-Architektur 15 MEMHCs und 87 DPHCs besitzen.

Die genutzte Referenzimplementierung für AES erreicht auf dem Test-PC mit einem Intel Core i7-3930K 6-Kernprozessor bei 3,2 GHz 0,4566 MBytes/s auf einem genutzten Kern. Dies ist nicht überraschend, da auch dieser Code nicht optimiert ist. Im Vergleich liegt die schnellste dem Autor bekannte Implementierung [120] bei 436,5 MBytes/s auf 6 Kernen bzw. 72,75 MBytes/s pro Kern. Maximale geschätzte Verlustleistung pro Kern liegt auch hier bei 21,6 W.

Im Falle des FPGA und des iterativen Ansatzes ist eine maximale Arbeitsfrequenz von 140MHz erreichbar. Der Ressourcenverbrauch liegt dabei bei 1072 ALUTs und 173 Registern. Die Verlustleistung setzt 1752,16 mW um. Maximale Rechenleistung beträgt hierbei 149,333MByte/s.

Ebenfalls auf dem FPGA unter Verwendung des Pipeline-Ansatzes benötigt die Applikation 12925 ALUTs und 3840 Register. Die maximale Arbeitsfrequenz liegt bei 192MHz und 1877,55 mW an Verlustleistung werden dabei umgesetzt. Die maximale Rechenleistung liegt bei 3,072 GBytes/s.

Der iterative Ansatz beim ASIC benötigt eine Fläche von 694153.624µm² und setzt eine Verlustleistung in Höhe von 16.737 mW bei 500MHz um. Die maximal erzielbare Rechenleistung liegt bei 533,33MBytes/s bei einer Arbeitsfrequenz von 500MHz und 987,073MBytes/s bei 952,38MHz.

Bei Nutzung einer vollständigen Pipeline im ASIC ergeben sich eine Fläche von 697699.860 µm² und eine Verlustleistung von 298.336mW bei 1GHz. Die maximale Rechenleistung erreicht 16GBytes/s bei 1GHz und dementsprechend 32Gbytes/s bei 2GHz.

Bei dieser Applikation hat die HoneyComb-Architektur leistungsbezogen das Nachsehen gegenüber dem FPGA und CPU. Punkten kann sie hingegen im Bereich der Verlustleistung, da sie hier nach wie vor deutlich vor FPGA und CPU liegt.

## 5.6. Ergebnisse am Beispiel des Demonstrationschips

Der gesamte Demonstrationschip besitzt insgesamt 2 IOHCs, 2 MEMHCs und 11 DPHCs, wie dies an der Abbildung 116 erkennbar ist. Drei der vier Applikationen (FFT, iMDCT und Wavelet) können im Array gemeinsam mit anderen Applikationen zur Ausführung gebracht werden, da genügend Ressourcen zur Verfügung stehen. Lediglich die AES-Anwendung nimmt das komplette Array in Anspruch und definierte damit die Mindestgröße des Siliziums.

Abbildung 118 veranschaulicht das Layout des finalen Chips. Hier ist gut zu erkennen, dass die ursprüngliche hexagonale Struktur der HoneyComb-Architektur den rechteckigen Verläufen der zugrunde liegenden Technologie gewichen ist. Durch den Umstand, dass aktuelle Standardzellentechnologien keine diagonalen Strukturen zulassen, muss mit einer rechteckige Zellstruktur mit spaltenweisem Versatz gearbeitet werden, um die Distanzen und damit Signallaufzeiten zwischen verbundenden Zellen gering zu halten.

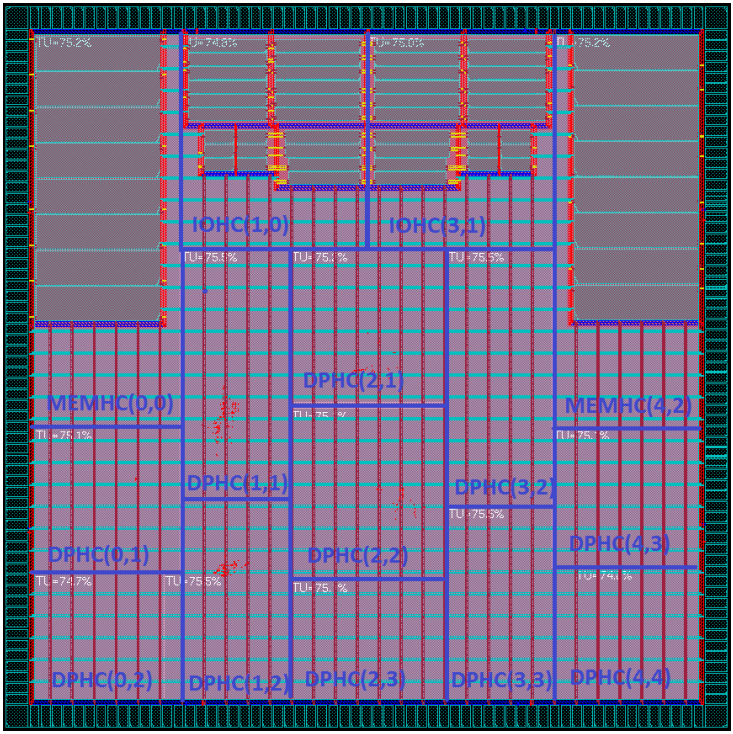



**Abbildung 118: Layout des Demonstrationschips mit Cadence SoC-Encounter dargestellt**

Die resultierenden Flächen der Architektur, der Zellen und sowohl der RUs als auch FUs können der nachfolgenden Tabelle 22 entnommen werden. Interessant zu sehen ist die Tatsache, dass das Routing in Form von RUs ca. 32% der Architektur ausmacht. Trotz der komplexen Funktionen wie dynamisches Routing zur Laufzeit sind diese Flächen nur moderat angewachsen. Untersuchungen zufolge, die während der Entwicklung der Architektur angestellt wurden, lassen sich die Multiplexer-Strukturen in den RUs wesentlich effekti-

ver durch Crossconnect-Bars oder Fat-Trees realisieren. Hier sind Einsparungen im Bereich von bis 50% zu erwarten. Dies muss in folgenden Arbeiten an der HoneyComb-Architektur genau untersucht und eine Optimierung in Betracht gezogen werden.

**Tabelle 22 HC-Flächendaten für den Demonstrationschip**

| | Total (µm²) | FU (µm²) | FU (%) | RU (µm²) | RU (%) |
|---|---|---|---|---|---|
| HoneyComb Array | 11710748,425 | | | | |
| MEMHC(0,0) | 1311531,967 | 1110461,525 | 84,7% | 197345,736 | 15,0% |
| DPHC(0,1) | 539894,071 | 297550,577 | 55,1% | 237830,163 | 44,1% |
| DPHC(0,2) | 396563,426 | 274143,160 | 69,1% | 118505,676 | 29,9% |
| IOHC(1,0) | 916692,007 | 779640,183 | 85,0% | 135337,373 | 14,8% |
| DPHC(1,1) | 641344,142 | 319323,669 | 49,8% | 316822,789 | 49,4% |
| DPHC(1,2) | 552048,345 | 275439,425 | 49,9% | 271981,987 | 49,3% |
| DPHC(2,1) | 481635,659 | 277569,867 | 57,6% | 199851,557 | 41,5% |
| DPHC(2,2) | 545871,601 | 282593,582 | 51,8% | 258622,000 | 47,4% |
| DPHC(2,3) | 489649,785 | 282346,622 | 57,7% | 202980,265 | 41,5% |
| IOHC(3,1) | 896440,189 | 777648,588 | 86,7% | 117172,093 | 13,1% |
| DPHC(3,2) | 655516,353 | 313049,238 | 47,8% | 337424,192 | 51,5% |
| DPHC(3,3) | 587245,084 | 283377,268 | 48,3% | 299256,250 | 51,0% |
| MEMHC(4,2) | 1328922,341 | 1111166,185 | 83,6% | 214033,646 | 16,1% |
| DPHC(4,3) | 555160,041 | 302403,067 | 54,5% | 248095,467 | 44,7% |
| DPHC(4,4) | 407390,702 | 274811,598 | 67,5% | 128556,949 | 31,6% |
| Cells Total | 10305905,714 | 6961524,555 | 67,5% | 3283816,141 | 31,9% |

Hinsichtlich der Verlustleistung ist der Anteil der Routing-Ressourcen sogar noch geringer ausgefallen: 16,1%, siehe Tabelle 23. Dies deutet darauf hin, da die große Zahl der Gatteraktivitäten in den DPHCs passiert und Kommunikationsnetze verhältnismäßig geringere Anzahl an Gatter-Aktivitäten aufweisen. Nichtsdestotrotz würde auch die Verlustleistung von oben erwähnten Optimierungen profitieren und ihren Anteil weiter reduzieren können.

Insgesamt haben IOHCs und MEMHCs durch die große Zahl der implementierten Speichermodule einen hohen Anteil an der Gesamtverlustleistung im Array. Dieser Umstand muss ebenfalls genauer untersucht werden und

gegebenenfalls Maßnahmen zur Reduktion der Verlustleistung ergriffen werden.

**Tabelle 23 HC-Verlustleistungsdaten für den Demonstrationschip**

| | Dynamic Power Total (mW) | Dynamic Power FU (mW) | Dynamic Power FU (%) | Dynamic Power RU (mW) | Dynamic Power RU (%) |
|---|---|---|---|---|---|
| MEMHC(0,0) | 27,333 | 24,380 | 89,2% | 2,953 | 10,8% |
| DPHC(0,1) | 7,480 | 4,570 | 61,1% | 2,909 | 38,9% |
| DPHC(0,2) | 6,407 | 4,332 | 67,6% | 2,076 | 32,4% |
| IOHC(1,0) | 68,563 | 65,848 | 96,0% | 2,715 | 4,0% |
| DPHC(1,1) | 8,732 | 5,174 | 59,3% | 3,558 | 40,7% |
| DPHC(1,2) | 7,880 | 4,328 | 54,9% | 3,552 | 45,1% |
| DPHC(2,1) | 7,084 | 4,322 | 61,0% | 2,762 | 39,0% |
| DPHC(2,2) | 7,536 | 4,329 | 57,4% | 3,206 | 42,6% |
| DPHC(2,3) | 7,136 | 4,329 | 60,7% | 2,807 | 39,3% |
| IOHC(3,1) | 68,144 | 65,825 | 96,6% | 2,319 | 3,4% |
| DPHC(3,2) | 9,010 | 5,182 | 57,5% | 3,828 | 42,5% |
| DPHC(3,3) | 7,782 | 4,332 | 55,7% | 3,450 | 44,3% |
| MEMHC(4,2) | 27,385 | 24,387 | 89,1% | 2,998 | 10,9% |
| DPHC(4,3) | 7,769 | 4,577 | 58,9% | 3,192 | 41,1% |
| DPHC(4,4) | 6,049 | 4,334 | 71,6% | 1,715 | 28,4% |
| Cells Total | 274,288 | 230,248 | 83,9% | 44,041 | 16,1% |

Die in Tabelle 23 präsentierten Verlustleistungswerte sind alle der statischen Analyse geschuldet. Hier wird je nach Modell eine 50% Aktivität angenommen und daraus die mögliche Verlustleistung berechnet. Eine dynamische Verlustleistungsanalyse ergibt hier einen präziseren zeitlichen Verlauf. Abbildung 119 veranschaulicht die Ausführung der beschrieben Applikationen in folgender Reihenfolge:

| | | |
|---|---|---|
| 1100 – 25000 Zyklen | FFT + iMDCT | parallel Ausführung |
| 30100 – 82300 Zyklen | AES | exklusive Ausführung |
| 82300 – 105500 Zyklen | Wavelet + FFT | parallele Ausführung |
| 105500 - >117100 Zyklen | iMDCT | exklusive Ausführung |

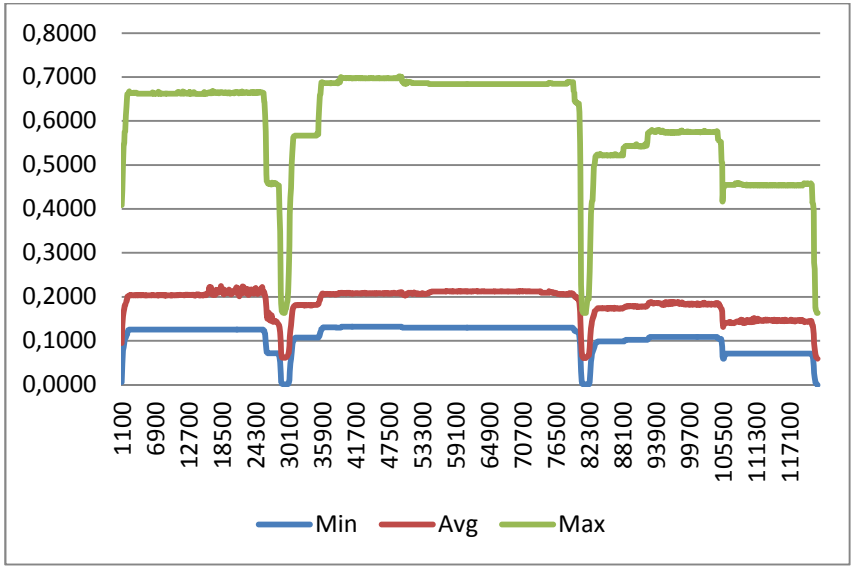



**Abbildung 119: Ergebnisse der dynamische Timing-Analyse basiertend auf den Aktivitäten der oben beschriebene Anwendungen**

Es ist gut zu erkennen, wie unterschiedliche Applikationen unterschiedliche Verlustleistung zur Folge haben, was natürlich an unterschiedlichen Konfigurationen und Aktivitäten innerhalb der Applikationen liegt. Auch deutlich zu erkennen sind die Pausen, wobei die Verlustleistung eine deutliche Reduktion erfährt. Im Falle von AES ist auch die Ladephase erkennbar, bevor die Berechnungen starten und die Applikation die maximale Verlustleistung erzeugt.

Die präsentierten Ergebnisse der HoneyComb-Architektur sind zwecks Vergleichbarkeit bei einer Arbeitsfrequenz von 100 MHz ermittelt worden. Die maximale Arbeitsfrequenz, die bei der Synthese und Layout ermittelt wurde, lag bei 166 MHz im Array und 250 MHz in den IOHCs. Die real gemessene Arbeitsfrequenz am Chip lag indes leicht darüber und erreichte im Array bis zu 200 MHz.

## 5.7. Zusammenfassung

Im Zuge der Entwicklung der HoneyComb-Architektur wurden mehrere Applikationen als Proof-of-Concept realisiert und darauf zur Ausführung gebracht. Darunter sind Applikationen wie FFT, iMDCT, DWT und AES zu finden. Diese Applikationen wurden weiterhin genutzt, um den Demonstrati-

onschip zu spezifizieren, indem zu Gunsten dieser Anwendungen die Ressourcen auf diesem Chip selektiert wurden. Um die maximale Chip-Fläche, die zur Herstellung zur Verfügung steht, nicht zu überschreiten, wurde weiterhin ein inhomogener Aufbau des Array ausgewählt. In dieser Konfiguration wurde schlussendlich der Chip fabriziert und abschließend erfolgreich getestet. Abbildung 120 zeigt den fertigen Chip in einem PGA-Package mit freier Sicht auf das Die.

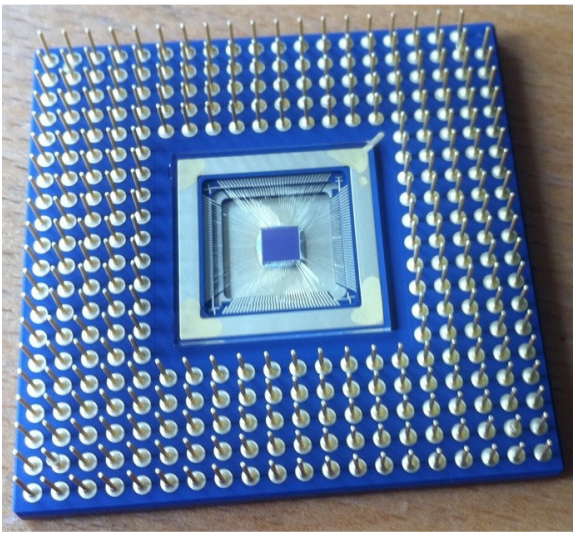


**Abbildung 120: Photo des fertigen Demonstratonschips im PGA-Gehäuse mit durchsichtigem Unterboden mit Blick auf den TSMC 90nm 16mm² Die**

In diesem Kapitel wurden die Applikationen und ihre Umsetzung auf die HoneyComb-Architektur erläutert und Vergleiche mit FPGA-, ASIC- und CPU-Implementierungen dieser Applikationen angestellt. Wie erwartet schnitt die ASIC-Implementierung mit deutlichem Abstand zum Feld am besten ab, gefolgt von der HoneyComb-Architektur, dem FPGA und schließlich der CPU. Bis auf die AES-Applikation, konnte sich die HoneyComb-Architektur sowohl leistungsbezogen als auch hinsichtlich der Fläche und Verlustleistung zwischen ASIC und FPGA einordnen. Im Falle der AES-Anwendung konnte insbesondere die Leistung der HoneyComb-Architektur nicht mithalten. Hier greift der Nachteil der 32-bit Architektur und des grobgranularen Charakters der HoneyComb-Architektur.

# 6. Zusammenfassung und Ausblick

Im Rahmen des DFG Schwerpunktprogramms 1148 für "Rekonfigurierbare Rechensysteme" (spprr) entstand im Projekt "Entwicklung und Synthese einer adaptiven multigranularen rekonfigurierbaren Hardwarearchitektur für dynamische Funktionsmuster" (AMURHA) die HoneyComb-Architektur. Basierend auf Erfahrungen und Studien vorhandener akademischer und kommerzieller Architekturansätze wurden neue Konzepte herausgearbeitet, implementiert und schließlich in Silizium hergestellt. Daneben entstand eine Reihe von Entwicklungswerkzeugen, die neben der Entwicklung der Anwendungen für diese Architektur auch die Konfiguration des Template-Charakters derselben erlaubten. Hierbei war es möglich auf einem Satz von Anwendungen die Architektur auf die besonderen Anforderungen anzupassen und Kosten in der Fertigung zu sparen.

Die Laufzeit des Projekts war auf sechs Jahre ausgelegt und enthielt neben der Entwicklung der Hardware auch die erforderlichen Software-Komponenten zur Realisierung eines abschließenden Demonstrators, um die Ergebnisse des Projektes zu präsentieren. Dies geschah durch die Fabrikation eines Demonstrationschips in TSMC 90nm Standardzellentechnologie. Aus Kostengründen und wegen vorhandener Flächenlimitierungen durch die Nutzung des Multi-Project-Wafers im miniASIC-Programm von IMEC, die auf rund 16mm² beschränkt war, wurde die HoneyComb-Architektur hauptsächlich auf vier ausgewählte Applikationen (FFT, iMDCT, DWT und AES) beschränkt und die vorhandenen Ressourcen darauf ausgelegt.

Die Technik des dynamischen Routings zur Laufzeit wurde in Zusammenarbeit mit XPP Technologies AG zum Patent angemeldet, siehe EP2004009640.

## 6.1. Einleitung

Moderne Rechensysteme basieren nachwievor auf altbekannten Architekturen wie CPUs, und DSPs, oder wahlweise auf FPGA- und/oder ASIC-Technologien. Insbesondere in mobilen Geräten wird eine feine Unterteilung der Algorithmen auf diese Architekturen vorgenommen, um den besten Trade-Off zwischen Flexibilität, Performance und Verlustleistung einzugehen. Hierbei sind auch die Parameter wie verfügbarer Platz in Geräten und Kosten von höchster Bedeutung, die nicht vernachlässigt werden können. Hinsichtlich der

Flexibilität haben hier die CPUs/DSPs einen deutlichen Vorteil, gefolgt von FPGAs und schließlich den anwendungsspezifischen integrierten Schaltkreisen (ASICs). Während die Flexibilität in dieser Reigenfolge deutlich abnimmt, nimmt die Leistungsfähigkeit hingegen enorm zu. Diese Tatsache und auch der Umstand, dass die Verlustleistung in dieser Reihenfolge ebenfalls abnimmt, macht ASICs enorm attraktiv für spezielle rechenintensive Algorithmen, die in Front/Backend-Implementierungen von Kommunikationssystemen zu finden sind. Aber auch zur Beschleunigung von graphischen Rechenanwendungen eignen sich anwendungsspezifische Schaltkreise in Form von GPUs, die in den letzten zehn Jahren bei mobilen Geräten ebenfalls Einzug gehalten haben.

Einen Kompromiss hinsichtlich Flexibilität, Verlustleistung und Performance stellen in diesem Vergleich grobgranulare rekonfigurierbare Architekturen dar. Durch die grobgranulare Natur dieser Architekturen ist die funktionale Dichte auf dem Silizium sehr hoch, da Highlevel-Funktionen wie Arithmetik nicht durch einzelne LUTs realisiert werden müssen, sondern als dedizierte dichtgepackte Funktionseinheiten ihren Platz auf dem Silizium finden. Um die notwendige Flexibilität zu erreichen, werden zusätzliche Komponenten in die Architekturen eingebracht, um die Programmierbarkeit zur ermöglichen. Die Art und Weise wie dies geschieht, wurde im Kapitel 3 an existierenden Architekturen verdeutlicht und zeigt, dass hier noch enormes Forschungspotential besteht.

Die funktionale Dichte grobgranular rekonfigurierbarer Architekturen lässt diese Architekturen in unserem Vergleich zwischen FPGAs und ASICs einordnen. Das Potential diesen Platz einzunehmen ist vorhanden, allerdings mangelt es noch an flexiblen Lösungen. Mit den vorgestellten Architekturen wie der Quicksilver ACM und der DRP-Architektur existieren bereits vielversprechende Kandidaten, die im Detail jedoch auch ihre Schwächen zeigen, siehe Kapitel 3.

## 6.2. Zusammenfassung

Durch das Studium vorhandener Architekturen und genutzter Techniken, wurde ein Satz an Anforderungen herausgearbeitet, um die Machbarkeit einer hochflexiblen dynamisch rekonfigurierbaren Architektur zu demonstrieren, siehe Kapitel 3. Es wurden Techniken wie dynamisches Routing zu Laufzeit, Multikontext-Fähigkeit, Multigranularität, programmierbare IOs, kurze Rekonfigurationszeiten, Anpassungsfähigkeit der Architektur an Applikation auf Register-Transfer-Level (RTL) und konfliktfreie Überlagerung von Applikationen im Array identifiziert und als Ziel festgelegt. Das Ergebnis der Umsetzung dieser Anforderungen ist die HoneyComb-Architektur, die alle diese Eigenschaften vereint, aber nicht nur Vorteile hat und noch Potential zu Verbesserung bietet.

Durch die Realisierung des dynamischen Routings zu Laufzeit, erhält die Architektur die Fähigkeit zur Laufzeit komplette Konfigurationen oder einzelne Zellen von Applikationen auf dem Array zu verschieben. Es ist zusätzlich möglich, Zellen von verschiedenen Applikationen zu verschränken, ohne sich zum Entwurfszeitpunkt um die Routingkonflikte Gedanken machen zu müssen. Diese Fähigkeit ist gewissen Regeln unterworfen. So können beispielsweise Anwendungen mit mehreren Zellen und Feedback-Signalen nicht nach Belieben auseinander gezogen werden, ohne dass man Performance-Einbußen erleidet. Auch erzeugt das dynamische Routing zusätzliche Kosten in der Hardware, die sich im Demonstrationschip im Bereich von 30% bewegen, jedoch durch weitere Optimierungen reduziert werden können.

Multikontextdatenpfade erlauben dem Programmierer der HoneyComb-Architektur Anwendungsteile entweder in die Fläche oder in die Zeit abzubilden. Dies kann natürlich individuell je nach Anwendungseigenschaft entschieden werden und zu einer Flächeneinsparung führen, in dem langsame Teile vorwiegend in die Zeit und rechenintensive Teile der Anwendung in die Fläche abgebildet werden. Durch die multigranularen Datentypen dieser Architektur ist es möglich, sowohl arithmetische wie auch kontrollaste Applikationsteile effizient abzubilden. Natürlich ist das Verhältnis der dafür notwendigen feingranularen und grobgranularen Komponenten applikationsabhängig, es lässt sich jedoch eine Balance finden, wie dies durch die vier Demonstrationsapplikationen gezeigt wurde.

Um die HoneyComb-Architektur von System-Komponenten unabhängig zu machen, wurden programmierbare IOs implementiert, die in der Lage sind unabhängig DMA-Funktionen auszuführen und Daten zwischen dem System und dem Array auszutauschen. Hierfür wurden spezialisierte µController entwickelt, die sehr schnelle Transferfunktionen bieten und indirekt die Abläufe im Array anstoßen können.

Nach Vorbild der XPP-Architektur wurde die Kommunikation innerhalb der HoneyComb-Architektur vollsynchron ausgelegt. Hierfür wird ein Handshake-Protokoll genutzt, welches dafür sorgt, dass eine Komponente erst ihre Operation ausführen kann, wenn alle Eingänge einen gültigen Wert aufweisen. Daraufhin werden ein oder mehrere gültige Ausgänge generiert und an die nachfolgenden Komponenten geleitet. Diese Technik stellt eine praktische Möglichkeit für den Programmierer dar, eine strukturelle Anwendungsbeschreibung zu erzeugen, verursacht aber auch einen enormen Kostennachteil, der sich bei grobgranularen Komponenten im Bereich von 20-30% und im Bereich der feingranularen Logik im Bereich von bis zu 50% bewegt und darüber hinaus wegen doppelter Laufzeiten der Protokollsignale längere kritische Pfade nach sich zieht.

Durch die hohen Kosten einer rekonfigurierbaren Architektur, die durch zusätzliche Hardwarekomponenten zur Realisierung der geforderten Flexibilität entstehen, ist es sinnvoll eine Möglichkeit zur Kostenreduktion vorzusehen. Daher wurde die HoneyComb-Architektur als ein Template ausgeführt, welches auf RTL an die Anforderungen der Zielapplikationen angepasst werden kann. Dies wurde auch im Falle der Demonstrationsapplikationen durchgeführt, um die engen Grenzen der zur Verfügung stehenden Fläche des Multiproject-Wafers des miniASIC-Programms zu erfüllen. Das Ergebnis ist eine inhomogene Hardware-Architektur, die alle notwendigen Funktionen zur Ausführung der Zielapplikationen bietet. Es ist aber auch weiterhin möglich darüber hinaus Applikationen abzubilden. Der Programmierer muss indessen mit den zur Verfügung stehenden Ressourcen auskommen.

Das Programmiermodell der HoneyComb-Architektur ist derzeit auf zwei proprietäre Programmiersprachen beschränkt. Zum einen kann der Low-Level-Assembler (HoneyComb-Assembler, HCA) genutzt werden, um strukturelle Applikations-Beschreibungen zu erstellen. Zum anderen bietet die abstraktere HoneyComb-Language (HCL) eine prozessbasierte Beschreibungsmethodik, die jeder Zelle innerhalb des Arrays einen Prozess mit abstrakter Funktion bietet. Eine Anbindung an eine Hochsprache C oder ähnliche existiert derzeit nicht.

Zur Entwicklung von Anwendungen und zum Management des Architektur-Templates sind mehrere Entwicklungswerkzeuge entstanden, die neben der Übersetzung der HCA/HCL Beschreibungen, auch die zyklusgenaue graphische Visualisierung der Vorgänge innerhalb des Arrays ermöglichen (HC-Viewer). Dadurch hat der Anwender die Möglichkeit, seine Applikation zu debuggen und zu analysieren, um gegebenenfalls gezielt Modifikationen vorzunehmen.

Zur Demonstration der Funktionstüchtigkeit der HoneyComb-Architektur und als Proof-of-Concept wurden vier ausgewählte Applikationen auf die Architektur abgebildet: FFT, iMDCT, DWT und AES. Hierbei handelt es sich um bekannte und vielgenutzte Applikationen aus den Bereichen der Telekommunikation, Signalanalyse, Bildverarbeitung und Datensicherheit, die hohe Anforderungen an Rechenarchitekturen stellen und häufig als Benchmarks herangezogen werden. Zur Realisierung der Applikationen wurden die meisten der oben genannten Eigenschaften der HoneyComb-Architektur genutzt und ihre Funktion demonstriert. Details und die Ergebnisse der Applikationen finden sich im vorigen Kapitel. Die Ergebnisse bestätigen die Einordnung der HoneyComb-Architektur als Vertreter der grobgranularen Architekturen zwischen den FPGAs und ASICS.

Im Projekt AMURHA wurde eine adaptive dynamisch rekonfigurierbare Architektur, die HoneyComb-Architektur, entwickelt, synthetisiert, funktional

verifiziert und die Funktionstüchtigkeit durch die Fabrikation eines realen Demonstrator-Chips bewiesen. Die Integration der Fülle neuer Funktionen, die in ihrer Art und in genutzter Kombination einmalig ist, macht die Architektur effizient, äußerst flexibel und für den Einsatz im fehlerredundanten Umfeld geeignet. Die gesteckten Ziele dieses Projekts wurden zu 100% erreicht und können in weiteren Projekten verbessert und ausgebaut werden.

## 6.3. Ausblick

Die Entwicklung der HoneyComb-Architektur wurde in sechs Jahren in einem Ein-Mann-Projekt mit Unterstützung zahlreicher Studenten durchgeführt und im Nachgang in einem weiteren Jahr mit dem Layout des Demonstrationschips abgeschlossen. In dieser Zeit sind sehr viele Ideen entstanden und mussten auch wieder verworfen werden, da der Zeitplan des Projekts die breitgefächerte Verfolgung aller Möglichkeiten nicht zuließ. So wurden viele Kompromisse geschlossen, die nicht immer zur bestmöglichen Lösung führten, aber den erfolgreichen Abschluss des Projektes ermöglichten. Deshalb werden im Folgenden vier wichtige Ideen zur Verbesserung der HoneyComb-Architektur vorgestellt, die in der möglichen weiteren Fortführung des Projektes betrachtet werden sollten.

*Optimierungen der Multiplexer-Strukturen/Leitungslängen*

Um die Komplexität der Routing Units (RUs) im ersten Entwurf klein zu halten und die Durchführbarkeit des Projektes nicht zu gefährden, wurden zunächst einfache Multiplexer-Strukturen genutzt und im Enddesign belassen. Nichtsdestotrotz wurden Untersuchungen angestellt, inwieweit sich weitere Kosten in diesem Bereich einsparen lassen. Hierbei hat sich herausgestellt, dass durch den Einsatz von Crossconnect-Netzwerken anstelle von einfachen Multiplexern bereits eine Kostenreduktion von bis zu 50% möglich ist. Das Risiko in der Implementierung lag insbesondere in der komplexen Ansteuerung dieser Netzwerke und wurde daher verworfen. Eine weitere mögliche Einsparung bieten die Generalized FAT Trees, die durch ihre hierarchischen Schaltstrukturen einen minimalen Kostenaufwand erzeugen. In beiden Fällen lassen sich die Leitungslängen in den RUs optimal ausbalancieren und dadurch zusätzlich Verlustleistung einsparen.

*Selbstsynchronisation in den Datenpfadmodulen (DPHCs,MEMHCs)*

Durch den Einsatz der Handshakeprotokolle in der gesamten HoneyComb-Architektur bietet sich dem Programmierer eine komfortable Sicht auf den Ablauf von Anwendungen. Allerdings erzeugt diese Technik einen hohen Overhead und kann durch weitere Arbeiten an der Architektur reduziert werden. Diese Reduktion kann wie folgt aussehen: Während die globale Kommunikation auf RU-Ebene und in den IOHCs weiterhin mit einem Handshakepro-

tokoll arbeitet, wird das Handshakeprotokoll aus den FUs von DPHCs und MEMHCs vollständig entfernt. Stattdessen wird der Ablauf in diesen Zellen vollständig durch Multikontext-Steuerung nach dem Vorbild der DRP übernommen. Es werden weiterhin mehrere ALUs und LUTs integriert, wobei ihre Interaktion durch einen Sequenzspeicher gesteuert wird. Das Interface zum globalen Kommunikationsnetzwerk muss die Synchronisation zwischen den Handshakeprotokollen und der Multikontextsteuerung bewerkstelligen und auf einander abstimmen. Durch das Entfernen der Handshakeprotokolle sind in den Datenpfaden deutlich höhere Frequenzen von mehr als 350 MHz zu erwarten. Auch die Reduktion der Verlustleistung bei gleichen Arbeitsfrequenzen im Vergleich zur aktuellen Version dürfte spürbar sein. Darüber hinaus wird die Programmierbarkeit und die mögliche Anbindung an eine Hochsprache wie C/C++vereinfacht.

*Anbindung an eine Hochsprache*

Grundsätzlich stellt es sich als schwierig heraus, rekonfigurierbare Architekturen durch Hochsprachen programmierbar zu machen. In der Vergangenheit gab es viele Versprechen, die im Endeffekt auf eine Teilmenge der Hochsprachen gesetzt haben und den Programmierer einschränkten. Eine vollständige Unterstützung einer Hochsprache würde in diesem Kontext der Zielarchitektur eine Unmenge an neuen Applikationen eröffnen.

Im Falle der HoneyComb-Architektur verhält es sich in ähnlicher Weise: Nach aktuellem Stand ist es schwierig eine direkte Anbindung zu realisieren. Der Hauptgrund hierfür liegt in dem Synchronisationsnetzwerk, welches ein ungewöhnliches Paradigma für eine Hochsprache wie C darstellt und keine direkte Funktionen zur Steuerung dieser Eigenschaft bietet. Dennoch ist eine eingeschränkte Anbindung an dieser Stelle denkbar und sollte in Betracht gezogen werden. Eine völlig andere Situation bietet sich, falls die Selbstsynchronisation aus den Datenpfaden entfernt wird. Dann stellt sich der Datenpfad dem Compiler wie ein VLIW-Prozessor dar und es können bekannte Algorithmen zur Umsetzung genutzt werden.

*Optimierung der Speichermodule*

Bei Betrachtung der Verlustleistung der HoneyComb-Architektur fällt dem Betrachter sofort die Tatsache ins Auge, dass speicherintensive Module (MEMHCs, IOHCs) einen hohen Anteil an der gesamten Verlustleistung aufweisen. Hier sollten weitergehende Untersuchungen angestellt und die Ursache für dieses Problem ermittelt werden. Erste Hinweise deuten darauf hin, dass die integrierten Speichermodule durch ihre Fläche eine hohe Verlustleistung erzeugen. In diesem Fall wären kleinere und abschaltbare Speichermodule von Vorteil, die bei Bedarf hinzu geschaltet werden können (Clock Gating, Voltage Gating).

# Referenzen


[1] Moore, Gordon. "Cramming More Components onto Integrated Circuits," *Electronics Magazine* Vol. 38, No. 8, pp. 4 (April 19, 1965)

[2] Moore, Gordon. "Progress in Digital Integrated Electronics" *IEEE, IEDM Tech Digest* (1975) pp.11-13.

[3] International Technology Roadmap for Semiconductors, www.itrs.net

[4] International Technology Roadmap for Semiconductors, 2013 Edition, "Process Integration, Devices and Structures

[5] Intel Corporation, Santa Clara, USA, www.intel.com

[6] Eriksson, H.; Larsson-Edefors, P.; Henriksson, T.; Svensson, C.; , "Full-custom vs. standard-cell design flow - an adder case study," Design Automation Conference, 2003. Proceedings of the ASP-DAC 2003. Asia and South Pacific , vol., no., pp. 507- 510, 21-24 Jan. 2003

[7] Taiwan Semiconductor Manufacturing Company Limited, Hsinchu, Taiwan, www.tsmc.com

[8] nVidia Corporation, Santa Clara, USA, http://www.nvidia.com

[9] AMD Corporation, Sunnyvale, USA, http://www.amd.com

[10] Xilinx Corporation, San José, USA, http://www.xilinx.com

[11] Altera Corporation, San José, USA, http://www.altera.com

[12] Lattice Semiconductor Corporation, Hillsboro, Oregon, USA, http://www.latticesemi.com

[13] Atmel Corporation, San José, USA, http://www.atmel.com

[14] Actel Corporation, Mountain View, California, USA, http://www.atmel.com

[15] MPEG - The Moving Picture Experts Group, http://mpeg.chiariglione.org

[16] D. J. Le Gall, "MPEG: A video compression standard for multimedia applications," Commun. of the ACM, vol. 34, pp. 47-58, April 1991.

[17] A. Puri, "Video Coding using the MPEG-2 compression standard," SPIE/VCIP, Cambridge, MA, vol. 2094, pp. 1701-1713, Nov. 1993.

[18] Battista, S.; Casalino, F.; Lande, C., "MPEG-4: a multimedia standard for the third millennium. 2," MultiMedia, IEEE , vol.7, no.1, pp.76,84, Jan-Mar 2000

[19] Sullivan, G.J.; Ohm, J.; Woo-Jin Han; Wiegand, T., "Overview of the High Efficiency Video Coding (HEVC) Standard," Circuits and Systems for Video Technology, IEEE Transactions on , vol.22, no.12, pp.1649,1668, Dec. 2012

[20] Redl, Siegmund M.; Weber, Matthias K.; Oliphant, Malcolm W (February 1995). *An Introduction to GSM*. Artech House Mobile Communication Series, ISBN 978-0890067857
[21] Cox, Christopher; *Essentials of UMTS*, The Cambridge Wireless Essentials Series, 978-0521889315
[22] Souto, N.; Silva, J.C.; Rodrigues, A.; Cercas, F.; Correia, A., "Enhanced UMTS CS-CDMA uplink transmission using turbo super-orthogonal codes," Vehicular Technology Conference, 2004. VTC 2004-Spring. 2004 IEEE 59th , vol.1, no., pp.357,361 Vol.1, 17-19 May 2004
[23] TR 36.806, "Evolved Universal Terrestrial Radio Access (E-UTRA)"; Relay architectures for E-UTRA (LTE-Advanced)
[24] Nakamura, Takeshiro; "Proposal for Candidate Radio Interface Technologies for IMT-Advanced Based on LTE Release 10 and Beyond (LTE-Advanced)", 3GPP IMT-Advanced Evaluation Workshop, Beijing, China, 17-18 December, 2009
[25] Abeta, S., "Toward LTE commercial launch and future plan for LTE enhancements (LTE-Advanced)," *Communication Systems (ICCS), 2010 IEEE International Conference on* , vol., no., pp.146,150, 17-19 Nov. 2010
[26] Texas Instruments Inc, Dallas, USA, www.ti.com
[27] DFG-Deutsche Forschungsgemeinschaft, www.dfg.de
[28] DFG Schwerpunktprogramm 1148, http://www12.informatik.uni-erlangen.de/spprr
[29] Rohde & Schwarz; "UMTS Long Term Evolution (LTE) Technology Introduction", Application Note, 1MA111
[30] Woh, M.; Sangwon Seo; Mahlke, S.; Mudge, T.; Chakrabarti, C.; Flautner, K.; , "AnySP: Anytime Anywhere Anyway Signal Processing," Micro, IEEE , vol.30, no.1, pp.81-91, Jan.-Feb. 2010
[31] Raivio, Y.; , "4G-hype or reality [mobile communications]," 3G Mobile Communication Technologies, 2001. Second International Conference on (Conf. Publ. No. 477) , vol., no., pp.346-350, 2001
[32] Safadi, M.S.; Ndzi, D.L.; , "Digital Hardware Choices For Software Radio (SDR) Baseband Implementation," Information and Communication Technologies, 2006. ICTTA '06. 2nd , vol.2, no., pp.2623-2628, 0-0 0
[33] Sheu, B.; Ismail, M.; Wang, M.; Tsai, R.; "Current and Future Trends in Multimedia Standards", Wiley-IEEE Press, 1998, ISBN 9978-0470545348
[34] Sikora, T., "Trends and Perspectives in Image and Video Coding," Proceedings of the IEEE , vol.93, no.1, pp.6,17, Jan. 2005

[35] Hartenstein, R.; , "Trends in reconfigurable logic and reconfigurable computing," Electronics, Circuits and Systems, 2002. 9th International Conference on , vol.2, no., pp. 801- 808 vol.2, 2002
[36] The OpenMP® API specification for parallel programming; www.openmp.org
[37] nVidia CUDA Runtime API, http://docs.nvidia.com/cuda/cuda-runtime-api/
[38] The OpenCL Specification v1.2; Khronos OpenCL Working Group; http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf
[39] AnandTech; "Apple's iPhone Dissected", http://www.anandtech.com/show/2264
[40] "Pentium(R) Processor with MMX Technology"; Intel Corporation, http://download.intel.com/design/archives/processors/mmx/docs/24318504.pdf
[41] Lipp, H. M., Becker, J., "Grundlagen der Digitaltechnik", Oldenbourg Wissenschaftsverlag, 2010, ISBN 978-3486597479
[42] Baker, R. J., "CMOS - Circuit Design, Layout and Simulation", IEEE Press, 2005, ISBN 0-471-70055
[43] Tietze, U., Schenk, Ch., "Halbleiter-Schaltungstechnik", Springer Verlag, Springer, Auflage: 14., 2012, ISBN 978-3642310256
[44] Jung, V., Warnecke, H.-J., "Handbuch der Telekommunikation", Springer, 2. Auflage, 2002, ISBN 3-540-4295-3
[45] Strutz, T., "Bilddatenkompression", Vieweg Praxiswissen, 3. Auflage, ISBN 3-528-23922-0
[46] Nekoogar, F., "Timing Verification of ASICs", Prentice Hall, 1999, ISBN 0-13-794348-2
[47] Saint, C., Saint, J., "IC Layout Basics", McGraw-Hill, 2001, ISBN 0-07-138625-4
[48] "Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor", White Paper, March 2004, Intel Corp., http://download.intel.com/design/network/papers/30117401.pdf
[49] "Cortex-A9 MPCore", Technical Reference Manual, 2012, ARM Ltd., http://infocenter.arm.com/help/topic/com.arm.doc.ddi0407i/DDI0407I_cortex_a9_mpcore_r4p1_trm.pdf
[50] J. M. Rabaey. Digital Integrated Circuits. Prentice Hall, 1996
[51] Bryant, E.R., O'Hallaron, D.R., "Computer Systems: A Programmer's Perspective", Addison-Wesley, 2nd Edition, 2010, ISBN 978-0136108047
[52] ISO/IEC JTCI/SC29 WG1, "JPEG2000 Part I", Final Committee Draft Version 1.0, March 2000
[53] ISO/IEC JTCI/SC29 WG11, "MPEG-4 Overview", V.21, March 2002
[54] IEEE, "VHDL Language Reference Manual 2002", IEEE Standard 1076-2002, 2002, ISBN 0-7381-3247-0

[55] IEEE, "Verilog Hardware Description Language", IEEE Standard 1364-2001, 2001, ISBN 0-7381-2827-9
[56] Cadence Design Systems Corporation, San Jose, USA, www.cadence.com
[57] Synopsys Inc., Mountain View, USA, www.synopsys.com
[58] Aldec Inc., Henderson, USA, www.aldec.com
[59] Mentor Graphics Corporation, Wilsonville, USA, www.mentor.com
[60] ISO/IEC 9899:2011, "Programming Languages - C", 2011
[61] IEEE 754-2008: Standard for Floating-Point Arithmetic, IEEE Standards Association, 2008, doi:10.1109/IEEESTD.2008.4610935
[62] Patterson, D., Hennessy, J.L., "Rechnerorganisation und Rechnerentwurf", Oldenbourg Wissenschaftsverlag, 2011, ISBN 978-3486591903
[63] Leon3 Processor by Aeroflex Gaisler, www.gasler.com
[64] OpenSPARC Project, http://www.oracle.com/technetwork/systems/opensparc/index.html
[65] Amdahl, Gene M., "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, AFIPS Conference Proceedings, AFIPS Press, Reston, Va., 1967, pp. 483–485
[66] Flynn, M., "Some Computer Organizations and Their Effectiveness," Computers, IEEE Transactions on , vol.C-21, no.9, pp.948,960, Sept. 1972
[67] Hartenstein, R.W.; Kress, R., "A datapath synthesis system for the reconfigurable datapath architecture," Design Automation Conference, 1995. Proceedings of the ASP-DAC '95/CHDL '95/VLSI '95., IFIP International Conference on Hardware Description Languages. IFIP International Conference on Very Large Scal , vol., no., pp.479,484, 29 Aug-1 Sep 1995
[68] Becker, J.; Thomas, A.; Scheer, M.: "Datapath and Compiler Integration of Coarse-grain Reconfigurable XPP-Arrays into Pipelined RISC Processors", Proceedings of IFIP International Conference on Very Large Scale Integration (IFIP VLSI-SOC 2003), Darmstadt, Germany, Dezember 1-3, 2003
[69] "Stratix V Device Handbook", www.altera.com
[70] Becker, J.; Pionteck, T.; Habermann, C.; Glesner, M., "Design and implementation of a coarse-grained dynamically reconfigurable hardware architecture", VLSI, 2001. Proceedings. IEEE Computer Society Workshop on , vol., no., pp.41,46, May 2001
[71] Braun, L.; Paulsson, K.; Kromer, H.; Hubner, M.; Becker, J., "Data path driven waveform-like reconfiguration," Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on , vol., no., pp.607,610, 8-10 Sept. 2008

[72] Hübner, M., "Dynamische und partiell rekonfigurierbare Hardwaresystemarchitektur mit echtzeitfähiger On-Demand Funktionalität", Dissertation, Verlag Karlsruhe, 2007

[73] Jan, C.-H.; Bhattacharya, U.; Brain, R.; Choi, S.-J.; Curello, G.; Gupta, G.; Hafez, W.; Jang, M.; Kang, M.; Komeyli, K.; Leo, T.; Nidhi, N.; Pan, L.; Park, J.; Phoa, K.; Rahman, A.; Staus, C.; Tashiro, H.; Tsai, C.; Vandervoorn, P.; Yang, L.; Yeh, J.-Y.; Bai, P., "A 22nm SoC platform technology featuring 3-D tri-gate and high-k/metal gate, optimized for ultra low power, high performance and high density SoC applications," Electron Devices Meeting (IEDM), 2012 IEEE International , vol., no., pp.3.1.1,3.1.4, 10-13 Dec. 2012

[74] Fermi-Architecture, Whitepaper, nVidia Corp.

[75] Corel Corp., Ottawa, Kanada, www.corel.com

[76] CyberLink Corp., Neu-Taipeh, Taiwan, www.cyberlink.com

[77] Kepler GK110 Architecture, Whitepaper, nVidia Corp.

[78] Mehta, N., "Xilinx redefines Power, Performance, and Design Productivity with Three Innovative 28nm FPGA Families: Virtex-7, Kintex-7, Artix-7 Devices", White Paper: 7 Series FPGAs, Xilinx Corp.

[79] Abnous, A.; Zhang, H.; Wan, M.; Varghese; G., Prabhu; V. and Rabaey, J.; “The Pleiades Architecture”, in “The Application of Programmable DSPs in Mobile Communications”, John Wiley & Sons, Ltd, Chichester, UK, 2002

[80] Chen, D.C.; Rabaey, J.M.; , "A reconfigurable multiprocessor IC for rapid prototyping of algorithmic-specific high-speed DSP data paths," Solid-State Circuits, IEEE Journal of , vol.27, no.12, pp.1895-1904, Dec 1992

[81] Goldstein, S.C.; Schmit, H.; Budiu, M.; Cadambi, S.; Moe, M.; Taylor, R.R.; "PipeRench: a reconfigurable architecture and compiler," Computer , vol.33, no.4, pp.70-77, Apr 2000

[82] Ebeling, C.; Fisher, C.; Guanbin Xing; Manyuan Shen; Hui Liu; "Implementing an OFDM receiver on the RaPiD reconfigurable architecture," Computers, IEEE Transactions on , vol.53, no.11, pp. 1436- 1448, Nov. 2004

[83] Guangming Lu; Singh, H.; Ming-Hau Lee; Bagherzadeh, N.; Kurdahi, F.J.; Filho, E.M.C.; Castro-Alves, V.; "The MorphoSys dynamically reconfigurable system-on-chip," Evolvable Hardware, 1999. Proceedings of the First NASA/DoD Workshop on , vol., no., pp.152-160, 1999

[84] Mirsky, E.; DeHon, A.; "MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources," FPGAs for Custom Computing Machines, 1996.

Proceedings. IEEE Symposium on , vol., no., pp.157-166, 17-19 Apr 1996

[85] Chen, D.C.; Rabaey, J.M.; , "A reconfigurable multiprocessor IC for rapid prototyping of algorithmic-specific high-speed DSP data paths," Solid-State Circuits, IEEE Journal of , vol.27, no.12, pp.1895-1904, Dec 1992

[86] Miyamori, T.; Olukotun, U.; , "A quantitative analysis of reconfigurable coprocessors for multimedia applications," FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on , vol., no., pp.2-11, 15-17 Apr 1998

[87] Babb, J.; Frank, M.; Lee, V.; Waingold, E.; Barua, R.; Taylor, M.; Kim, J.; Devabhaktuni, S.; Agarwal, A., "The RAW benchmark suite: computation structures for general purpose computing," Field-Programmable Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on , vol., no., pp.134,143, 16-18 Apr 1997

[88] Master, P.; "The next big leap in reconfigurable systems," Field-Programmable Technology, 2002. (FPT). Proceedings. 2002 IEEE International Conference on , vol., no., pp. 17- 22, 16-18 Dec. 2002

[89] QuickSilver Technology, www.quicksilvertech.com

[90] Suzuki, M.; Hasegawa, Y.; Yamada, Y.; Kaneko, N.; Deguchi, K.; Amano, H.; Anjo, K.; Motomura, M.; Wakabayashi, K.; Toi, T.; Awashima, T.; , "Stream applications on the dynamically reconfigurable processor," Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on , vol., no., pp. 137- 144, 6-8 Dec. 2004

[91] Ohring, S.R.; Ibel, M.; Das, S.K.; Kumar, M.J.; , "On generalized fat trees," Parallel Processing Symposium, 1995. Proceedings., 9th International , vol., no., pp.37-44, 25-28 Apr 1995

[92] Puttegowda, K.; Athanas, P.; "RSA encryption using extended modular arithmetic on the Quicksilver COSM adaptive computing machine," Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on , vol., no., pp. 305- 307, 9-11 April 2003

[93] Plunkett, B.; Chou, D.; "Computational efficiency: adaptive computing vs. ASICs," Electronics, Circuits and Systems, 2002. 9th International Conference on , vol.2, no., pp. 819- 822 vol.2, 2002

[94] Buchmann, J., "Einführung in die Kryptographie", Springer-Verlag, Berlin 1999, ISBN 3-540-66059-3

[95] Chevallier, C., Brunner, C., Garavaglia, A., Murray, K.P., Baker, K.R., "WCDMA (UMTS) Deployment Handbook. Planning and

Optimization Aspects", John Wiley & Sons, New York NY 2006, ISBN 0-470-03326-6

[96] PACT XPP Technologies AG, www.pactxpp.com

[97] „XPP-III Processor Overview“, White Paper, PACT XPP Technologies AG

[98] Ganesan, M.K.A.; Singh, S.; May, F.; Becker, J.; "H. 264 Decoder at HD Resolution on a Coarse Grain Dynamically Reconfigurable Architecture," Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on , vol., no., pp.467-471, 27-29 Aug. 2007

[99] Schuler, E.; Vorbach, M.; May, F.; Weinhardt, M.; , "Dynamic Reconfiguration for Irregular Code Using FNC-PAE Processor Cores," Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on , vol., no., pp.244-249, 16-20 May 2011

[100] J. M. P. Cardoso and M. Weinhardt: Chapter 9: “Compilation and Temporal Partitioning for a Caorse-Grain Reconfigurable Architecture”, in “New Algorithms, Architectures and Applications for Reconfigurable Computing” (editors: P. Lysaght, W. Rosenstiel), Springer, Dordrecht, NL, 2005

[101] Campi, F.; Konig, R.; Dreschmann, M.; Neukirchner, M.; Picard, D.; Juttner, M.; Schuler, E.; Deledda, A.; Rossi, D.; Pasini, A.; Hubner, M.; Becker, J.; Guerrieri, R.; , "RTL-to-layout implementation of an embedded coarse grained architecture for dynamically reconfigurable computing in systems-on-chip," System-on-Chip, 2009. SOC 2009. International Symposium on , vol., no., pp.110-113, 5-7 Oct. 2009

[102] NEC Electronics, heute: Renesas Electronics, www.renesas.eu

[103] Toi, T.; Awashima, T.; Motomura, M.; Amano, H.; , "Time and space-multiplexed compilation challenges for dynamically reconfigurable processors," Circuits and Systems (MWSCAS), 2011 IEEE 54th International Midwest Symposium on , vol., no., pp.1-4, 7-10 Aug. 2011

[104] Oppold, T.; Schweizer, T.; Kuhn, T.; Rosenstiel, W.; , "A Design Environment for Processor-Like Reconfigurable Hardware," Parallel Computing in Electrical Engineering, 2004. PARELEC 2004. International Conference on , vol., no., pp. 171- 176, 7-10 Sept. 2004

[105] Suzuki, N.; Kurotaki, S.; Suzuki, M.; Kaneko, N.; Yamada, Y.; Deguchi, K.; Hasegawa, Y.; Amano, H.; Anjo, K.; Motomura, M.; Wakabayashi, K.; Toi, T.; Awashima, T.; "Implementing and evaluating stream applications on the dynamically reconfigurable processor," Field-Programmable Custom Computing Machines, 2004.

FCCM 2004. 12th Annual IEEE Symposium on , vol., no., pp. 328-329, 20-23 April 2004
[106] Chameleon Project, chameleon.ctit.utwente.nl
[107] Heysters, Paul M. and Smit, Gerard J.M. and Molenkamp, Egbert (2004) Energy-Efficiency of the Montium Reconfigurable Tile Processor. In: International Conference on Engineering of Reconfigurable Systems and Algorithms, ERSA 2004, Las Vegas, Nevada, USA, June 21-24, 2004
[108] Smit, G.J.M.; Heysters, P.M.; Rosien, M.; Molenkamp, B.; "Lessons learned from designing the MONTIUM - a coarse-grained reconfigurable processing tile," System-on-Chip, 2004. Proceedings. 2004 International Symposium on , vol., no., pp. 29- 32, 16-18 Nov. 2004
[109] Smit, L.T.; Rauwerda, G.K.; Molclerink, A.; Wolkotte, P.T.; Smit, G.J.M.; "Implementation of a 2-D 8x8 IDCT on the Reconfigurable Montium Core," Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on , vol., no., pp.562-566, 27-29 Aug. 2007
[110] Heysters, P.M.; Rauwerda, G.K.; Smit, G.J.M.; "Implementation of a HiperLAN/2 receiver on the reconfigurable Montium architecture," Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International , vol., no., pp. 147, 26-30 April 2004
[111] Zhiyi Yu; Meeuwsen, M.J.; Apperson, R.W.; Sattari, O.; Lai, M.; Webb, J.W.; Work, E.W.; Truong, D.; Mohsenin, T.; Baas, B.M.; , "AsAP: An Asynchronous Array of Simple Processors," Solid-State Circuits, IEEE Journal of , vol.43, no.3, pp.695-705, March 2008
[112] Truong, D.N.; Cheng, W.H.; Mohsenin, T.; Zhiyi Yu; Jacobson, A.T.; Landge, G.; Meeuwsen, M.J.; Watnik, C.; Tran, A.T.; Zhibin Xiao; Work, E.W.; Webb, J.W.; Mejia, P.V.; Baas, B.M.; , "A 167-Processor Computational Platform in 65 nm CMOS," Solid-State Circuits, IEEE Journal of , vol.44, no.4, pp.1130-1144, April 2009
[113] NIST - National Institute of Standards and Technology, www.nist.gov
[114] Daemen, J.; Rijmen, V.; "AES Proposal: Rijndael"; www.nist.gov, 2003
[115] Daemen, J.; Rijmen, V.; "The Design of Rijndael", Springer Verlag, Springer, 2002, ISBN 978-3540425809
[116] Cooley, J.W.; Tukey, J.W.; "An algorithm for the machine calculation of complex Fourier series". Mathematics of Computation, Vol. 19, No. 90, pp. 297–301, 1965
[117] Princen, J.; Johnson, A.; Bradley, A., "Subband/Transform coding using filter bank designs based on time domain aliasing cancellation,"

Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP '87. , vol.12, no., pp.2161,2164, Apr 1987
[118] Nikolajevic, V.; Fettweis, G., "New recursive algorithms for the forward and inverse MDCT," Signal Processing Systems, 2001 IEEE Workshop on , vol., no., pp.51,57, 2001
[119] Ogg Vorbis, www.vorbis.com
[120] TrueCrypt, Disk Encryption Software, www.truecrypt.org
[121] AMBA, Advanced Microcontroller Bus Architecture, www.arm.com
[122] Lee, C. Y., "An Algorithm for Path Connections and Its Applications," Electronic Computers, IRE Transactions on, vol.EC-10, no.3, pp.346,365, Sept. 1961
[123] Interuniversity Microelectronics Centre (IMEC), www.imec.be

# Eigene Publikationen


[1] Thomas, A.; Rückauer, M.; Becker, J.; “HoneyComb: An Application-Driven Online Adaptive Reconfigurable Hardware Architecture” International Journal of Reconfigurable Computing, vol. 2012, Article ID 832531, 17 pages, 2012. doi:10.1155/2012/832531

[2] Thomas, A.; Ruckauer, M.; Becker, J., "HoneyComb: an application-driven online adaptive reconfigurable hardware architecture", SBCCI2011, pp. 173-178. ACM, (2011)

[3] Thomas, A.; Ruckauer, M.; Becker, J., "HoneyComb: A multi-grained dynamically reconfigurable runtime adaptive hardware architecture", SOC Conference (SOCC), 2011 IEEE International, pp.335-340, 26-28 Sept. 2011

[4] Thomas, A.; Becker, J.: "Development and Synthesis of Adaptive Multi-grained Reconfigurable Hardware Architecture for Dynamic Function Patterns", Dynamically Reconfigurable Systems - Architectures, Design Methods and Applications, Springer, Heidelberg, February 2010, pp. 3-23

[5] S. Eisenhardt, T. Schweizer, J. Oliveira, T. Oppold, W. Rosenstiel, A. Thomas, J. Becker, F. Hannig, D. Kissler, H. Dutta, J. Teich, H. Hinkelmann, P. Zipf, M. Glesner, "SPP1148 Booth: Coarse-Grained Reconfiguration", In Proceedings of the International Conference on Field Programmable Logic and Applications (FPL), p.349, Heidelberg, Germany

[6] Koenig, R.; Thomas, A.; Kuehnle, M.; Becker, J.; Crocoll, E.; Siegel, M.: "A Mixed-Signal System-on-Chip Audio Decoder Design for Education", RCEducation 2007, Porto Alegre, Brasil, 2007

[7] Thomas, A.; Becker, J.: "New Adaptive Multi-grained Hardware Architecture for Processing of Dynamic Function Patterns (Neue adaptive multi-granulare Hardwarearchitektur).", it - Information Technology 49(3): 165, 2007

[8] Thomas, A.; Becker, J.: "Multi-grained Reconfigurable Hardware Architecture with Online-Adaptive Routing Techniques", IFIP International Conference on Very Large Scale Integration (IFIP VLSI-SOC 2005), Perth, Western Australia, October 17-19, 2005

[9] Thomas, A.; Becker, J.: "Online-adaptive Reconfigurable Hardware Architecture and Runtime Environment", IEEE International SOC

Conference (SOCC2005), Dulles Airport, Washington, September 25-28, 2005

[10] Thomas, A.; Becker, J.: "Multi-grained Reconfigurable Datapath Structures for Online-Adaptive Reconfigurable Hardware Architectures", IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2005), Tampa, Florida, May 11-12, 2005

[11] Becker, J.; Hübner, M.; Paulsson, K.; Thomas, A.: "Dynamic Reconfiguration On-Demand: Real-time Adaptivity in Next Generation Microelectronics.", ReCoSoc2005, Montpellier, France, 2005

[12] Thomas, A.; Zander, T.; Becker, J.: "Adaptive DMA-based I/O Interfaces for Data Stream Handling in Multi-grained Reconfigurable Hardware Architectures", Symposium on Integrated Circuits and Systems Design (SBCCI 2004), Porto de Galinhas, Pernambuco, Brazil, September 7-11, 2004

[13] Thomas, A.; Becker, J.: "Dynamic Adaptive Routing Techniques In Multigrain Dynamic Reconfigurable Hardware Architectures", Field-programmable Logic and its applications (FPL 2004), Antwerpen, August 2004

[14] Thomas, A.; Becker, J. E.; Becker, J.: "Anwendungsspezifische IP-Generierung für zukünftige SoC-Implementierungen in mobilen Kommunikationssystemen", Tagungsband, Dresdner Arbeitstagung Schaltungs- und Systementwurf (DASS´2004) und IP2 Workshop, Dresden, April 19-20, 2004

[15] Thomas, A.; Becker, J.; Heinkel, U.; Winkelmann, K.; Bormann, J.: "Formale Verifikation eines Sonet/SDH Framers", Tagungsband "Dresdner Arbeitstagung Schaltungs- und Systementwurf (DASS´2004) und IP2 Workshop", Dresden, April 19-20, 2004

[16] Thomas, A.; Becker, J.: "Aufbau- und Strukturkonzepte einer adaptiven multigranularen rekonfigurierbaren Hardwarearchitektur", in Kongressbericht "17th International Conference on Architecture of Computing Systems (ARCS 2004)", Workshop "Dynamically Reconfigurable Systems", Augsburg, März 23-26, 2004

[17] Thomas, A.; Becker, J.; Heinkel, U.; Winkelmann, K.; Bormann, J.: "Formale Verifikation eines Sonet/SDH Framers", in Tagungsband "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen", GI/ITG/GMM Workshop, Kaiserslautern, Februar 24-25, 2004

[18] Bieser, Carsten; Becker, Jens; Thomas, Alexander; Müller-Glaser, Klaus D.; Becker, Jürgen: "Hardware/Software Co-Training by FPGA/ASIC Synthesis and programming of a RISC Microprocessor-

Core", International Conference on Microelectronic Systems Education (MSE); Anaheim/Los Angeles, USA; Juni 2003

[19] Becker, J.; Thomas, A.; Scheer, M.: "Datapath and Compiler Integration of Coarse-grain Reconfigurable XPP-Arrays into Pipelined RISC Processors", Proceedings of IFIP International Conference on Very Large Scale Integration (IFIP VLSI-SOC 2003), Darmstadt, Germany, Dezember 1-3, 2003

[20] Becker, J.; Thomas, A.; Scheer, M.: "Efficient Processor Instruction Set Extension by Asynchronous Reconfigurable Datapath Integration", Proc. of 10th XVI Brazilian Symposium on Integrated Circuit Design (SBCCI 2003) , Sao Paulo, Brazil, September 6-10, 2003

[21] Bieser, Carsten; Becker, Jens; Thomas, Alexander; Müller-Glaser, Klaus D.; Becker, Jürgen: "Hardware/Software Co-Training by FPGA/ASIC Synthesis and programming of a RISC Microprocessor-Core", "International Conference on Microelectronic Systems Education (MSE)" , Anaheim/Los Angeles, USA, June 2003

[22] Becker, J.;Thomas, A.; Vorbach, M.; Baumgarte, V.; "An Industrial/Academic Configurable System-on-Chip Project (CSoC): Coarse-grain XPP-/Leon-based Architecture Integration", Design, Automation and Test in Europe Conference (DATE 2003), München, März 2003

[23] Becker, J.; Thomas, A.; Vorbach, M.; Ehlers, G.; "Dynamically Reconfigurable Systems-on-Chip: A Core-based Industrial/Academic SoC Synthesis Project.", IEEE Workshop Reconfigurable SoC , Hamburg, April 2002, Kongressbericht, 2002

# Patente

EP2004009640; Data processing device and method, Anmeldetag: 30.08.2004