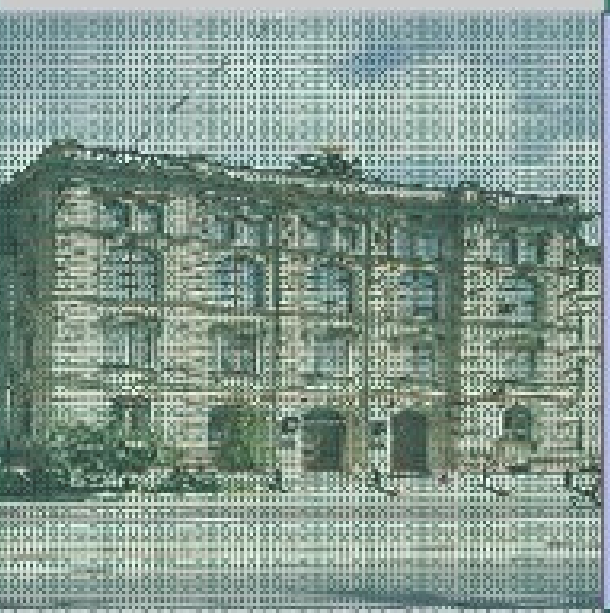
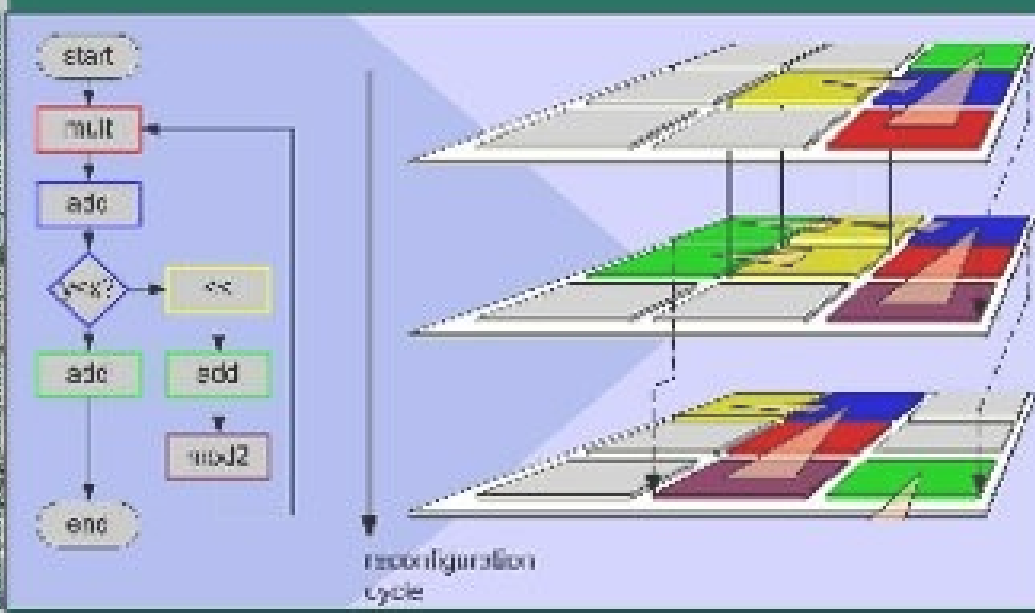
Wissenschaftliche Schriftenreihe
EINGEBETTETE, SELBSTORGANISIERENDE SYSTEME
Band 7

Dr. René Beckert

Untersuchungen zur Kostenoptimierung für Hardware Emulatoren durch Anwendung von Methoden der partiellen Laufzeitrekonfiguration

Prof. Dr. Wolfram Hardt (Hrsg.)





TECHNISCHE UNIVERSITÄT CHEMNITZ

TUD*press*

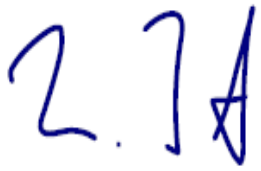
Vorwort zur Wissenschaftlichen Schriftenreihe “Eingebettete, Selbstorganisierende Systeme”

Der vorliegende Band der wissenschaftlichen Schriftenreihe Eingebettete selbstorganisierende Systeme widmet sich der Optimierung von Testverfahren, einem Thema, das im Entwurf von integrierten Schaltkreisen zunehmend Bedeutung gewinnt. Im Speziellen wird die Optimierung von FPGA-basierten Hardware-Emulatoren untersucht. Zum Einsatz kommen Methoden der partiellen, dynamischen Rekonfiguration. Das Ziel der Optimierung ist die möglichst hohe Auslastung der eingesetzten FPGAs. Dies reduziert insbesondere die Kosten für Emulatorsysteme für Anwendungen mittlerer Komplexität.

Dr. Beckert stellt zunächst das Gebiet der Hardware-Emulation und die für den Einsatz von partielle, dynamische Rekonfiguration notwendigen Verfahren zur Partitionierung und Steuerung der Systeme vor.

Im Schwerpunkt der Arbeit ein implementierter Entwurfsablauf vorgestellt, der zunächst die zu testende Entwurfsbeschreibung funktional partitioniert und diese ereignisgesteuert emuliert. Dazu wurde eine Laufzeitumgebung entwickelt, die den zeitlichen Aufwand für die notwendigen Rekonfigurationen minimiert und eine optimierte Platzierung der Hardware-Module ermöglicht. Die Platzierung berücksichtigt die Architekturmerkmale des zu emulierenden Entwurfs. Der Ablauf des gesamten Verfahrens wird in der vorliegenden Promotionsschrift von Herrn Beckert schrittweise dargestellt und umfassend erläutert. Die flexible Umsetzung und Implementierung der einzelnen Teilsysteme und Verfahren des Ansatzes lässt eine weitgehende Adaption in angrenzende Problemstellungen zu.

Der Leser erhält einen interessanten Einblick in Testmethodiken für integrierte Schaltungen und den Optimierungsansatz durch partielle Laufzeitrekonfiguration für FPGA-basierte Emulatoren.



Prof. Dr. Wolfram Hardt
Professur Technische Informatik
Dekan der Fakultät für Informatik
Wissenschaftlicher Leiter Universitätsrechenzentrum
Technische Universität Chemnitz
Oktober 2008



TECHNISCHE UNIVERSITÄT CHEMNITZ

Fakultät für Informatik

Untersuchungen zur Kostenoptimierung für Hardware Emulatoren durch Anwendung von Methoden der partiellen Laufzeitrekonfiguration

Dissertation

zur Erlangung des akademischen Grades

Doktoringenieur

(Dr.-Ing.)

vorgelegt

der Fakultät für Informatik

Technische Universität Chemnitz

von:	Dipl.-Ing. René Beckert
Gutachter:	Prof. Dr. rer. nat. Wolfram Hardt Prof. Dr.-Ing. Ulrich Heinkel Prof. Dr. rer. nat. Christophe Bobda

Chemnitz, den 23. Oktober 2008

Kurzfassung

An aktuelle Schaltkreis- und Systementwürfe werden zunehmend divergente Anforderungen gestellt. Einer sehr kurzen Entwicklungszeit für eine schnelle Markteinführung steht, um teure und aufwändige Re-Desings zu verhindern, eine möglichst umfangreiche Testabdeckung des Entwurfs gegenüber. Diese Anforderungen führen konventionelle Methoden für die Verifikation und den Test wie die Simulation an wirtschaftliche Grenzen.

Aus diesem Grund werden für den funktionalen Test zunehmend wieder verwendbare FPGA-basierte HW-Emulatoren eingesetzt. Nach der Abbildung der Entwurfsbeschreibung auf die Emulator-Plattform ist die funktionale Überprüfung des Entwurfs ca. im Faktor 10^3 schneller als mit einem Simulator durchführbar. Durch den Einfluss der steigenden Komplexität aktueller Entwürfe auf die Emulator-Plattform reduzieren sich jedoch signifikant die Performanz und die Effizienz des Ansatzes. Um dem entgegen zu wirken, werden zunehmend Multi-FPGA-Systeme eingesetzt. Diese Systeme bestehen aus einer großen Anzahl an FPGAs und können bis zu 30 Millionen Entwurfsgatter abbilden. Ein wesentlicher Nachteil dieses Ansatzes ist aber die geringe Anzahl an Verbindungsleitungen zwischen den FPGAs im Verhältnis zu den enthaltenen Logikgattern, was zu einer geringen Auslastung der programmierbaren Logik führt.

Die in Emulatoren eingesetzten FPGAs sind zunehmend partiell zur Laufzeit rekonfigurierbar. Dabei wird die Funktionalität einzelner Systemteile zu Laufzeit ausgetauscht, während andere Teile weiter aktiv bleiben.

Der in der vorliegenden Arbeit umgesetzte Ansatz behandelt die Anwendung von Methoden der Laufzeitrekonfiguration auf dem Gebiet der Hardware-Emulation.

Dafür ist zunächst eine Partitionierung des zu testenden Entwurfs in möglichst funktional unabhängige Systemteile notwendig. In der Arbeit ist die Implementierung von zwei Partitionierungsmethoden beschrieben, welche unter funktionalem und strukturalem Kontext eingesetzt werden.

Die entstehenden Module werden im Anschluss unter Verwendung des Xilinx Modular Design Flows zu implementierbaren HW-Modulen synthetisiert.

Für eine optimierte und ressourceneffiziente Platzierung der einzelnen HW-Module während der Emulation, ist ein ebenfalls auf dem FPGA platziertes Kommunikationsnetzwerk implementiert. Dieses Netzwerk wird abgestimmt auf die Entwurfsmodule synthetisiert und von einer angepassten Hardware/Software Kopplung gesteuert.

Die Arbeit schließt mit einer Bewertung des umgesetzten Ansatzes anhand von zwei Entwürfen. Dabei wird das Optimierungspotential des Ansatzes durch die Berücksichtigung von regulären Eigenschaften der zu testenden Entwürfe dargestellt. Ausblickend wird eine Anwendung des beschriebenen Ansatzes auf Multi-FPGA-Systeme beschrieben.

Inhaltsverzeichnis

Formelzeichen **XIV**

1 Einleitung **1**
1.1 Motivation . . . 2
1.2 Problemstellung . . . 3
1.3 Gliederung der Arbeit . . . 3

2 Methoden und Stand der Technik **5**
2.1 Funktionale Verifikationsverfahren . . . 5
2.1.1 Simulation . . . 5
2.1.2 Emulation . . . 7
2.2 Hardware-basierte Test- und Verifikationsverfahren . . . 8
2.2.1 Rapid-Prototyping . . . 9
2.2.2 Hardware-in-the-Loop . . . 10
2.2.3 HW-Emulation im IC-Entwurf . . . 11
2.2.4 Multi-FPGA Emulator-Systeme . . . 14
2.2.5 Ansätze zur Optimierung von HW-Emulation . . . 18
2.3 Partiell Dynamische Rekonfiguration . . . 21
2.3.1 Einordnung dynamisch rekonfigurierbarer Systeme . . . 22
2.3.2 SoC-Architekturen . . . 24
2.3.3 FPGA-Architekturen . . . 26
2.3.4 Werkzeuge und Design-Flow . . . 28
2.4 Relevante Methoden . . . 31
2.4.1 Methoden der strukturellen Entwurfspartitionierung . . . 31
2.4.2 Methoden der funktionalen Entwurfspartitionierung . . . 36
2.4.3 Methoden zur Laufzeitsteuerung (Scheduling) . . . 37
2.5 Zusammenfassung . . . 39

3 Konzeption und Architekturentwurf **40**
3.1 Allgemeine Einordnung und Abgrenzung des Ansatzes . . . 40
3.1.1 Ableitung des Optimierungspotentials . . . 40
3.1.2 Modellbildung . . . 41
3.2 Ablauf des Verfahrens . . . 47
3.2.1 Entwurfsabhängige Randbedingungen . . . 48
3.2.2 Entwurfspartitionierung . . . 50
3.2.3 Steuerung der HaHardwaremodule . . . 50
3.2.4 Synthese für partiell dynamisch rekonfigurierbare Hardware . . . 52

3.2.5 Kopplung der Emulationsergebnisse (Joining) . . . 53
3.3 Entwicklungsplattform . . . 54
3.4 Zusammenfassung . . . 55

4 Entwurfs-Partitionierung 57
4.1 Funktionale Entwurfspartitionierung . . . 57
4.2 Strukturale Netzlistenpartitionierung . . . 61
4.3 Anpassung der Designpartitionen an den XMDF . . . 68
4.4 Bewertung des implementierten Verfahrens . . . 69
4.5 Zusammenfassung . . . 70

5 Steuerung der Entwurfs-Partitionen 72
5.1 Einordnung des implementierten Verfahrens . . . 72
5.2 Hardware/Software Kopplung . . . 72
5.3 Implementierung der Rekonfigurationsumgebung . . . 75
5.3.1 Überblick und Anforderungen der Umgebung . . . 75
5.3.2 *Routing*-Strategie für die Implementierung des Verbindungsnetzwerks . . . 76
5.4 Task-Bildung für eine effiziente Ausnutzung des Verbindungsnetzwerks . . . 81
5.5 Implementierung der Hardware für die Ereignis-Detektion . . . 83
5.6 Ableitung von Szenarien für die Steuerung unterschiedlicher Architekturen . . . 85
5.6.1 Strukturiert Modulare Entwürfe . . . 86
5.6.2 Reguläre Architekturen . . . 87
5.6.3 Kombinatorische Logik . . . 89
5.6.4 Diskussion und Bewertung eines Ansatzes zur Vorhersage von HW-Modulen zur Optimierung des Schedul-Algorithmus . . . 91
5.7 Zusammenfassung . . . 93

6 Ergebnisse und Entwurfsbeispiele 94
6.1 Ergebnisse des Partitionierungsverfahrens . . . 94
6.1.1 *RS-Decoder* . . . 94
6.1.2 Signal-Prozessor-SoC . . . 97
6.2 Synthese- und Implementierungsergebnisse des Emulator-*Framework* . . . 101
6.2.1 Dimensionierung der *Routing*-Topologie . . . 101
6.2.2 Synthese-Ergebnisse für verschiedene Parametervarianten . . . 103
6.3 Laufzeitergebnisse . . . 106
6.3.1 RS-Decoder . . . 107
6.3.2 Signal-Prozessor-SoC . . . 108
6.4 Bewertung des Emulationsverfahrens . . . 111

7 Zusammenfassung und Ausblick 113
7.1 Ausblick . . . 114

8 Anhang 124

Abbildungsverzeichnis

2.1 Entwurfsablauf Rapid Prototyping . . . 10
2.2 Performancevergleich Verifikation [11] . . . 12
2.3 FPGA basiertes Emulations-System . . . 13
2.4 Einfluss der Auslastung eines FPGAs auf den kritischen Pfad . . . 13
2.5 Statisches Mesh-Routing für Multi-FPGA Systeme [52] . . . 15
2.6 Dynamisches Crossbar-Routing für Multi-FPGA Systeme [107] . . . 15
2.7 Einfluss des Routing auf die Latenz . . . 16
2.8 Hardware/Software Co-Simulation [92] . . . 19
2.9 Erlangen Slot Machine [2] . . . 20
2.10 Makimoto Wave [27] . . . 23
2.11 Syntheseprozess . . . 29
2.12 Xilinx Modular Design Flow . . . 31
2.13 Zellreplikation . . . 33
2.14 Cone-Algorithmus [86] . . . 34
2.15 Cluster-Bildung für Cone-Partitionierung [86] . . . 34
2.16 MaxFlow-MinCut . . . 35
2.17 Funktionale Entwurfspartitionierung . . . 36

3.1 Gegenüberstellung 2D-3D Platzierung des Entwurfs . . . 42
3.2 rekonfigurierbarer Xilinx FPGA . . . 43
3.3 Abläufe der partiellen Emulationen . . . 45
3.4 Ablaufplan pRTR-Emulation . . . 47
3.5 Reguläre Architekturen . . . 49
3.6 Emulator Ansatz ohne KommUniversums-Netzwerk . . . 51
3.7 Automatische Scripten-Generierung XMDF-Synthese . . . 53
3.8 Kopplung der TeileEmulationen . . . 54
3.9 Blockschaltbild FM480 Entwicklungsplattform . . . 55

4.1 Hierarchische Systembeschreibung [33] . . . 58
4.2 Emulation für reduzierte reguläre Architekturen am Beispiel SIMD Datenpfad . . . 59
4.3 Funktionale Partitionierung . . . 60
4.4 Kombinatorische Logik . . . 62
4.5 Strukturaler Ablauf für die Partitionierung . . . 63
4.6 Bildung komplementärer *Cluster* . . . 66
4.7 Strukturale Partitionierung Version A . . . 67

5.1 Hardware/Software Kopplung . . . 73

5.2 Blockschaltbild Rekonfigurationsumgebung . . . 76
5.3 Implementierte Routing-Topologie . . . 78
5.4 Statisches Routing innerhalb eines Knotens aus rekonfigurierbaren Flächen . . . 79
5.5 Zeitmultiplex-Bussystem auf On-Chip Ebene 2 . . . 80
5.6 Entwurfsbeispiel für Task-Bildung . . . 82
5.7 Task-Bildung für verteilte HW-Module . . . 83
5.8 Schnittstellen HW-Monitor . . . 84
5.9 Szenarien für rekursive Rekonfiguration eines SoC . . . 86
5.10 Szenario für die Rekonfiguration einer SIMD-Architektur . . . 89
5.11 Emulationsablauf für kombinatorische Logik . . . 90
5.12 Optimierungspotential durch Anwendung von Prefetching . . . 92

6.1 Blockschaltbild RS-Decoder . . . 95
6.2 Syntheseergebnisse RS-Decoder nach Partitionierung . . . 96
6.3 Syntheseergebnisse Strukturale Partitionierung (*KES-block*) . . . 97
6.4 HW-Module des Signal-Prozessor-SoC . . . 98
6.5 Funktionale Partitionierung Top_Level SoC . . . 99
6.6 Ressourcebedarf für die Top_Level Emulation des SoC . . . 99
6.7 Funktionale Partitionierung des VDSP-16 Modules . . . 100
6.8 Ressourcenbedarf für unterschiedliche Routing-Topologien . . . 102
6.9 Ressourcenaufwand für Multiplexer-Implementierung . . . 102
6.10 Ressourcenbedarf der umgesetzten heterogenen *Routing*-Topologie . . . 103
6.11 Ressourcenbedarf für Emulator-*Framework* (Wortbreite 4 Bit) . . . 104
6.12 Abhängigkeit der Frequenz zu den *Framework* Parametern . . . 105
6.13 Zeitbedarf für Synthese in Abhängigkeit der Parameter . . . 106
6.14 Laufzeit der Module des *Reed-Solomon-Decoder* . . . 107
6.15 Module Efficienz *RS-Decoder* . . . 108
6.16 Rekonfigurationszeiten HW-Module SoC . . . 109
6.17 Emulatorperformance im Vergleich zu einem flachen Emulator . . . 110
6.18 Emulationszeiten für verschiedene HW-Module . . . 110

8.1 Blockschaltbild *Nallatech* HW-Emulator (BenNUEY PCI) . . . 125
8.2 Blockschaltbild *TheDiniGroup* HW-Emulator (DN8000K10) . . . 126
8.3 Blockschaltbild *TheDiniGroup* HW-Emulator (DN8000K10PCI) . . . 127
8.4 Blockschaltbild RS-Decoder . . . 128

Tabellenverzeichnis

2.1 FPGA-basierte Emulator-Systeme 17
2.3 Rekonfigurierbare System-on-Chips 25
2.5 Rekonfigurierbare FPGA 27
2.6 Gegenüberstellung unterschiedlicher Partitionierungsansätze 37

4.1 Typ-Konventionen für Übergabe der Netzlisten-Parameter 64

5.1 Befehlssequenzen zwischen Software und Hardware [38] 74

6.1 Verhältnisse der *On-Chip*-Busbreite zum PCI-X-Bus 104

Formelzeichen

$\sum G_{Emulator}$	Anzahl der Gatter im Emulator
$\sum G_{Design}$	Anzahl der zu emulierenden Gatter Entwurf
$\sum G_{max_Module}$	Anzahl der Gatter im parameter-größten HW-Modul nach der Partitionierung
T_K	Konfigurationszeit für ein FPGA
x_E	Strukturelement eines FPGA
$T_{R(x_E)}$	Partielle Konfigurationszeit für ein Strukturelement eines FPGA
N	Anzahl der Strukturelemente einer Partition in einem FPGA
max_E	Maximale Anzahl der Elemente einer Rekonfiguration
$B(reco)$	Bandbreite der Rekonfigurationsschnittstelle
$sizeof(E)$	Größe einer Partition unter Berücksichtigung der geometrischen Vorgaben des FPGA
$T_{R(m)}$	Rekonfigurationszeit des HW-Moduls m
$min(t_{R(m)})$	Minimale Rekonfigurationszeit
T_E	Emulationszeit eines HW-Moduls
$max(T_E)$	maximale Emulationszeit
$t_{T(m)}$	Partielle Emulationszeit des HW-Moduls m
t_M	Absolute Emulationszeit des HW-Moduls
P	Grad der Parallelisierung während der Platzierung von HW-Modulen
E_M	Laufzeiteffizienz

1 Einleitung

Die zunehmende Durchdringung alltäglicher Bereiche mit neuester Informations- und Kommunikationstechnik und die damit einhergehende stetig steigende Nachfrage an integrierten Schaltkreisen (IC - Integrated Circuit) und eingebetteten Systemen (SoC - System-on-Chip) fordert von den Herstellern eine angepasste Entwicklung bei gleichzeitig sehr hohem Kostendruck. Durch fortschreitende technologische Entwicklungen wie z.B. der Ausnutzung von Signalreflektionen in MIMO-Systemen (MIMO - Multiple Input Multiple Output) oder der Erschließung neuer Anwendungsgebiete wie dem Einsatz von Videotechnik in mobilen Endgeräten (DVB-H - Digital Video Broadcast handheld applications) müssen diese Systeme eine sehr hohe Leistungsfähigkeit, zur Kostenreduzierung aber auch einen hohen Grad an Adaptivität haben. So werden z.B. für den Einsatz in mobilen Endgeräten die Anforderungen hinsichtlich einer sehr geringen Leistungsaufnahme bei einer sehr großen Integrationsdichte gestellt. Für andere Systeme, welche in sicherheitskritischen Umgebungen wie z.B. Medizintechnik oder Automobilbau eingesetzt werden, bedarf es zudem einer hohen, bestimmbaren Zuverlässigkeit und Ausfallsicherheit.

Diese konträren Parameter müssen in der Spezifikation, dem Architekturentwurf und der Verifikation eines dedizierten Schaltkreises (ASIC[59] - Application Specific Integrated Circuit) oder eingebetteten Systems berücksichtigt werden. Des Weiteren lassen die auftretenden hohen Kosten während der Entwurfsphasen (*ASIC Design Phases*) eines Schaltkreises grundlegende Überarbeitungen (*Re-Design*) nach der Maskenfertigung nicht mehr zu. Werden Entwurfsfehler erst im abschließenden Funktionstest oder unter Anwendungsbedingungen erkannt, müssen diese, wenn überhaupt möglich, mit Hilfe schaltkreisexterner Modifikationen wie z.B. externer Taktgeneratoren oder über die Applikationssoftware des Systems behoben werden. Diese Überarbeitungen stellen zusätzliche Kosten dar und sind nur in bestimmtem Maße und unter Berücksichtigung der gegebenen Spezifikation des Systems möglich.

War es vor 10 Jahren wenigen großen Firmen vorbehalten, komplexe IC-Entwürfe zu realisieren und zu fertigen, ist es auf Grund der deutlich verbesserten und automatisierten Entwurfssoftware zunehmend auch klein- und mittelständischen Unternehmen (KMU) möglich, dedizierte Schaltkreise und Systeme zu entwerfen. Durch die immensen Kosten einer eigenen Fertigung (Fab - Fabric) und durch die geringe Anzahl von Entwicklungen, erfolgt die Herstellung in diesem Fall vorwiegend extern bei Fertigungsdienstleistern. Die Entwürfe so genannter *Fab-less* Entwicklungsunternehmen werden außerdem in zunehmendem Maße vor dem eigentlichen Technologieprozess (*Silicon Run*) als separate Netzliste des Schaltkreis oder als komplettes Sub-Systems (IP - Intellectual Property) vertrieben. Dadurch hat sich der Konkurrenzkampf dieser Firmen auf die Ebene des Entwurfs verlagert, welcher durch die Entwicklungszeit eines Systems (*Time-to-Market*) und dessen Funktionalität bestimmt ist. Die Entwicklungszeit des Schaltkreises oder Systems hängt dabei wesentlich von der Testphase ab.

1.1 Motivation

Der Entwurf eines integrierten Schaltkreises wird in mehrere Phasen [59] eingeteilt. Neben dem Entwurf wird direkt mit dem Test bestehender Komponenten begonnen. Während dieser Tests werden die funktionalen und nicht-funktionalen Anforderungen der Komponenten *gegen* die Spezifikation überprüft [56]. Zu funktionalen Anforderungen zählt die Literatur [58], die direkt mit der Funktionalität des Testobjekts verbundenen Eigenschaften wie z.B. korrektes Systemverhalten bei der durch den Entwurf erreichten Taktfrequenz. Demgegenüber beschreiben nicht funktionale Eigenschaften Dienste oder Parameter, die das Testobjekt zusätzlich erfüllen kann, die aber den eigentlichen Systemanforderungen untergeordnet werden. Dazu werden z.B. der Flächenbedarf oder die Baugröße gezählt.

Während des Entwurfsablaufs (*Design Flow*) stehen unterschiedliche Möglichkeiten für den Test zur Verfügung. Die Literatur unterscheidet grundsätzlich zwischen formalen und nicht formalen Methoden [103]. Formale Methoden [111] basieren auf dem mathematischen Beweis für die Korrektheit einer Schaltfunktion. Auf Grund der Komplexität aktueller Schaltkreis- und Systementwürfe ist der Einsatz formaler Methoden mit sehr großen Einschränkungen verbunden und im Digitalentwurf wenig verbreitet. Aus diesem Grund werden formale Methoden in dieser Arbeit nicht betrachtet. Nicht-formale Methoden für Funktionstests basieren auf *kreativen* Prozessen [103] und unterscheiden Simulations- und Emulations- sowie anwendungsbasierte Verfahren. Alle diese Verfahren gehen im Gegensatz zu formalen Methoden von folgendem Ansatz aus:
Das zu testende Modul bzw. der Schaltkreis oder das System ist mit vertretbarem Zeitbedarf nicht vollständig überprüfbar. Des Weiteren nimmt der Aufwand für den Test bzw. die Verifikation mit der Komplexität des Testobjekts überproportional zu (*Simulation Gap*), d.h. der Anteil für den Test übersteigt ab einer bestimmten Komplexität den Anteil der eigentlichen Entwurfszeit. Für aktuelle Systeme wird ein Test- bzw. Verifikationsbedarf von ca. 50-80% der Entwicklungszeit eingeschätzt [32].

Die Abbildung des im Entwurf befindlichen Schaltkreises auf eine spezielle, aus FPGAs bestehende Testplattform bietet eine Möglichkeit, den steigenden Zeitbedarf für die funktionalen Tests zu verringern. Auf Grund der zunehmenden Größe der Entwürfe steigt jedoch analog die Größe der Plattform und damit deren Kosten.

Die vorliegende Arbeit leistet einen Beitrag, eine FPGA-basierte Testplattform zu nutzen, um die funktionalen Tests während der Schaltkreisentwicklung zu beschleunigen und unter Einbeziehung der Eigenschaften von FPGAs, wie z.B. partieller Rekonfiguration zur Laufzeit, gleichzeitig die Kosten für die notwendige Plattform zu reduzieren. Dafür werden im Schwerpunkt der Arbeit bekannte Methoden aus dem Gebiet der partiell dynamischen Rekonfiguration angepasst und auf das Gebiet der Hardwareemulation angewendet.

Das Ziel der Arbeit ist die temporale Ausnutzung einer reduzierten Emulatorplattform, welches dieses FPGA-basierte Testverfahren auch für klein- und mittelständige Unternehmen wirtschaftlich realisierbar macht. Des Weiteren ist durch den Ansatz eine Möglichkeit gegeben, den Zeitbedarf für die Tests ins Verhältnis zu den Hardwarekosten der Plattform setzen zu können. Die dadurch neuartige Skalierbarkeit des Emulators führt zu einer deutlich verbesserten Aufwandsabschätzung hinsichtlich der Kosten für den Test großer Entwürfe.

1.2 Problemstellung

Die Problemstellung der vorgelegten Arbeit leitet sich zunächst aus den Nachteilen eines ausschließlich simulationsbasierten Verifikationsansatzes ab. Der durch die geringe Performance von Simulatoren entstehende Zeitbedarf für die Überprüfung des Entwurfs, übersteigt sehr schnell eine wirtschaftliche Grenze des geforderten Umfangs der gesamten Verifikation. Funktionale Tests, welche als Teil der Gesamtverifikation zu betrachten sind, werden unter anderem mit der Abbildung der Entwurfsdaten auf eine wieder verwendbare FPGA-Plattform umgesetzt. Diese Plattformen führen gegenüber Simulatoren zu einer wesentlichen Verbesserung der Performanz. Eine Bedingung dieses Verfahrens ist jedoch, dass es möglich sein muss, die Anzahl an Gattern des Entwurfs auf der Emulatorplattform abbilden zu können. Das führt auf Grund einer zunehmenden Komplexität bei großen Entwürfen zu signifikant steigenden Anforderungen an die FPGA-Plattform.

Um die Emulation dennoch auch für große Entwürfe durchführen zu können, werden zunehmend Multi-FPGA-Systeme eingesetzt. Das führt wiederum zu einem technologiebedingten Performanzverlust und zu weiter steigenden Kosten der Plattform.

Aus der Motivation der Arbeit leiten sich unter Berücksichtigung der Anforderungen eines funktionalen Tests zwei primäre Probleme ab:

- Der Entwurf muss in einem ersten Schritt partitioniert werden.
- In einem zweiten Schritt müssen die entstandenen Partitionen in zeitlich und funktional korrekter Reihenfolge auf dem FPGA abgebildet werden.

Des Weiteren ist für den Einsatz des Verfahrens in einem industriellen Umfeld eine weitere sekundäre Anforderung, dass die einzelnen Schritte automatisiert und für den Anwender verdeckt durchzuführen sind.

1.3 Gliederung der Arbeit

Kapitel 2 gibt einen Überblick über den Stand der Technik auf dem Gebiet rekonfigurierbarer Architekturen sowohl im SoC- als auch für den FPGA-Bereich. Des Weiteren werden bestehende Werkzeuge und Entwurfsabläufe für die Synthese von FPGAs, ASICs und pRTR-Systemen untersucht und vorgestellt. In Abschnitt 2.2 wird auf hardwarebasierte Test- und Verifikationsverfahren eingegangen und deren Relevanz für diese Arbeit und das mögliche Optimierungspotential untersucht. Die grundlegende Theorie und bestehende Verfahren für die Partitionierung und die Steuerung der Rekonfigurationen sind Inhalt von Abschnitt 2.4.

In Kapitel 3 seht der Architekturentwurf für das zu implementierende Emulatorsystem im Mittelpunkt. Nach der Einordnung und Modellierung des Verfahrens wird dessen Ablauf in Abschnitt 3.2 beschrieben. Auf Grund der Vorbedingungen und übernommener Teile aus anderen Arbeiten wird an dieser Stelle eine Abgrenzung die eigenen Arbeiten vorgenommen. Die für diese Arbeit wichtigsten Baugruppen der Entwicklungsplattform sind Inhalt von Abschnitt 3.3. Des Weiteren wird der Einfluss der Plattform auf das Gesamtsystem und damit dessen Adaptierbarkeit abgeleitet.

Als ein der Arbeit zu Grunde liegendes Verfahren wird die umgesetzte Entwurfspartitionierung in Kapitel 4 vorgestellt. Dabei liegt der Fokus auf funktionalen und strukturellen Partitionierungsmethoden und

es wird aus den in Abschnitt 2.4 vorgestellten Methoden ein für die Umsetzung des Emulators relevantes Verfahren abgeleitet. Die Implementierung des Verfahrens ist Inhalt von Abschnitt 4.4.

Für die Steuerung des Emulators wurde eine generische Kommunikations- und Rekonfigurationsumgebung (*Framework*) auf einem FPGA implementiert. In Kapitel 5 steht zunächst die Motivation für die Auswahl der zu Grunde liegenden Algorithmen und deren Funktionsweise im Mittelpunk. Darauf folgend wird die umgesetzte und implementierte Laufzeitumgebung *(Framework)* beschrieben.

In Kapitel 6 werden die Test- und Entwurfsbeispiele vorgestellt. An verschiedenen Hardwarearchitekturen wird der in der Arbeit umgesetzte Ansatz verifiziert und bewertet. Neben der für die Umgebung notwendigen Hardware wird auch auf das erreichbare *Timing* eingegangen und dieses den Aussagen herkömmlicher FPGA-basierter Emulatoren aus Abschnitt 2.2 gegenübergestellt.

Kapitel 7 schließt die Arbeit und gibt eine Zusammenfassung des erreichten Standes. Im Ausblick werden alternative Möglichkeiten zur Lösung suboptimaler Implementierungen dargestellt und erläutert.

2 Methoden und Stand der Technik

Im folgenden Kapitel werden Verfahren und Methoden untersucht, welche der Arbeit zu Grunde liegen. Zunächst werden Verifikationsverfahren gegenübergestellt. Im darauf folgenden Teil wird auf die partiell dynamische Rekonfiguration eingegangen. Als angewendete Methode für die Optimierung eines Hardware-Emulators steht zunächst deren zeitliche Entwicklung im Mittelpunkt. Im Anschluss werden die für dieses Gebiet relevanten Begriffe eingeführt und erläutert. Der zweite Teil des Kapitels vergleicht unterschiedliche Implementierungsansätze von pRTR-Systemen aus den Bereichen SoC sowie FPGA und stellt relevante herstellerspezifische Entwurfsabläufe dar. Im Abschnitt 2.2 werden hardwarebasierte Verifikationsverfahren für Schaltkreise und Systeme gegenübergestellt. Der abschließende Teil 2.4 beschreibt und bewertet Methodiken aus den Gebieten Entwurfspartitionierung und Laufzeitsteuerung (*Schedule*), welche die Grundlage für den Einsatz von partieller dynamischer Laufzeitrekonfiguration bilden.

2.1 Funktionale Verifikationsverfahren

2.1.1 Simulation

Die Simulation ist auf Grund der einfachen Handhabung die am häufigsten eingesetzte Methode für die Verifikation im Schaltkreis- und Systementwurf. Die zu testende Komponente wird zunächst von einem *Compiler* in ein für den Simulator ausführbares Datenformat übersetzt. Der Simulator bildet die Basis für die Simulation und ist eine auf einer Rechnerplattform installierte Anwendungssoftware. In KMU werden Standard-PCs verwendet, die zur Reduzierung der Simulationszeit über einen sehr großen Hauptspeicher verfügen müssen. Große Unternehmen nutzen aus Performancegründen sehr kostenintensive dedizierte Serverplattformen, um die Simulationszeit zu verringern.

Kontinuierlich arbeitende Simulatoren berechnen die einzelnen Signaländerungen zu jedem Takt (*Clock*) sequentiell. Um die Nebenläufigkeiten (Parallelität) eines realen Schaltkreises berücksichtigen zu können, werden zwischen jeder Signaländerung die Abhängigkeiten neben läufiger Prozesse geprüft. Durch eine von der Komplexität des Entwurfs abhängigen Anzahl an Zwischenberechnungen (*Delta-Cycles*) wird die Nebenläufigkeit in der Simulation abgebildet. Mit einer zunehmender Anzahl an neben läufigen Prozessen berechnet der Simulator entsprechend mehr *Delta-Cycles.* Da diese untereinander ebenfalls Abhängigkeiten haben, führt diese Methodik bei großen, hoch parallelen Entwürfen zu sehr langen Berechnungszeiten des Simulators. Um die Berechnungszeit einer Simulation für den Anwender in akzeptabler Zeit abschließen zu können, werden die Zeitfenster für einen Simulationslauf sehr klein gewählt. Dadurch besteht die Gefahr, dass fehlerhafte Zustandsänderungen, welche erst nach einem sehr langen Zeitintervall auftreten, nicht erkannt werden.

Um diesem Nachteil entgegen zu wirken, werden längere Simulationszeiträume zunehmend durch eine ereignisorientierte Simulation abgebildet. Dieses Verfahren, welches eine Klasse der diskreten Simulation darstellt, basiert auf der Einschränkung des Zustandsmodells der Simulation. Auf der Basis eines Ereigniskalenders oder einer Ereignisliste, den Ereignisroutinen und einer Zeitbasis, bildet dieses Verfahren die

Simulation in langen Zeitschritten ab. Im Gegensatz zur kontinuierlichen Simulation werden nur die in der Ereignisliste eingetragenen Ereignisse *(Events)* berücksichtigt. Neben dem eigentlichen Ereignis ist in der Ereignisliste auch die Art und der Zeitpunkt des Eintretens eingetragen. Die Ereignisroutinen beschreiben im Zusammenwirken mit der Ereignisliste die Übergangsbedingungen zwischen den einzelnen Ereignissen. Für eine effiziente Umsetzung dieses Simulationsverfahrens ist ein entsprechendes Programmiermodell notwendig. Aus diesem Grund findet das Verfahren z.B. Anwendung in der Verifikation der *TransAction-Level*-Programmierung für *SystemC* [69]. Für die Simulation einer digitalen Schaltung ist der diskrete Ansatz auf Grund der zusätzlich notwendigen Informationen über das Zustandsmodell nur mit Einschränkungen geeignet. Aktuelle Simulatoren unterstützen aus diesem Grund eine Kombination aus der diskreten und der kontinuierlichen Simulation. Dieses Verfahren wird als hybride Simulation bezeichnet [90]. Ziel der Kombination aus den beschriebenen zwei Ansätzen ist eine hohe Auflösung der Simulationsdaten an den für den Anwender wesentlichen Systemteilen und eine Reduzierung der Auflösung an den nicht relevanten Teilen. Aus diesem Grund muss der Anwender in diesem Fall die für die jeweilige Anforderung notwendigen Informationen bzgl. der *Event*-Signale eigenständig festlegen. Damit setzt das Verfahren ein detailliertes Wissen bzgl. der zu verifizierenden Schaltung voraus und führt bei einer falschen Festlegung der *Event*-Signale zu einer fehlerhaften Gesamtsimulationen. Da die Feststellung diese Fehler auf Grund ihrer Verteilung über ein sehr langes Zeitintervall diffizil ist und die Fehler nicht direkt zu falschen Ausgaben des kontinuierlichen Anteils der Simulation führen, stellen sie ein großes Risiko für die Richtigkeit der Gesamtsimulation dar. In Abschnitt 2.2.5 wird ein weiteres Verfahren zur Kopplung eines hybriden Hardware-Software Simulators beschrieben und bewertet. In diesem Fall wird der kontinuierliche Anteil der Gesamtsimulation auf ein FPGA-System abgebildet und der diskrete Teil in Software simuliert.

Der Entwickler muss, um in akzeptabler Zeit aussagekräftige Ergebnis zu erlangen, die Verifikation planen und den Verifikationsraum [67] entsprechend einschränken. Dazu ist eine Testumgebung (TB - Testbench) notwendig, welche die erforderlichen Eingangsvektoren erzeugt.

Ausgehend von Testdaten (Stimulationsvektoren) wird die Komponente oder das *Design* (DuT - Design under Test) über ausgewiesene Eingänge mit Daten stimuliert. Die Testbench wird nach dem Entwurfsprozess aus dem Design entfernt und durch die realen Schnittstellen des Systems ersetzt. Für die Effizienz der Simulation kommt der Testbench eine entscheidende Rolle zu. Von der Generierung der für den Testfall relevanten Datenvektoren (Verifikationsraum) hängt zum einen die Richtigkeit der Testergebnisse, zum anderen der Testaufwand und die damit für die Verifikation benötigte Zeit ab. Auf Grund der Funktionsweise ist es für jeden Simulationslauf notwendig, den Simulator nach Änderungen im Entwurf erneut zu starten und das *Design* zu *compilieren*. Dadurch wird der Simulator an den Startpunkt des Tests zurückgesetzt. Dies hat zur Folge, dass für jede Änderung eine Simulation des gesamten System bis zu dem für die Verifikation relevanten Zeitpunkt durchgeführt werden muss.

Werden zu Beginn der Verifikationsphase sehr viele Fehler (*Bug*) erkannt und behoben, ist es mit zunehmender Überarbeitung des Systems aufwändiger, die relevanten Testdaten zu genieren und das DuT an dem entsprechenden Zeitpunkt *einschwingen* zu lassen. Ein weiterer Nachteil von Simulatoren ist die fehlende Echtzeitfähigkeit. Die erreichbaren Signaländerungen eines simulationsbasierten Ansatzes reichen nicht aus, um das Signal-*Timing* einer realen Kommunikationsschnittstelle (z.B. PCI-X, USB) abbilden zu können. Um dennoch Schnittstellen in die Simulation einbeziehen zu können, wird ein Makro, welches taktabhängig die Signaländerungen der Schnittstelle generiert, in die Simulation eingebunden. Dadurch ist die Berücksichtigung der Interaktion zwischen Schaltkreis und einem Schnittstellenprotokoll gegeben. Das Makro einer Schnittstelle kann auf Grund der angepassten Taktzyklen das Verhalten realer Geräte (*Device*)

an den Schnittstellen jedoch nicht abbilden. Dieses fehlende Interaktionsverhalten stellt ein sehr großes Problem während der Entwicklung von komplexen heterogenen Systemen dar, da die Interpretation von *weichen* Spezifikationen für die Protokolle von den Herstellen unterschiedlich ausgelegt werden. Oftmals wird erst im direkten Test eines *Device* an der Schnittstelle deutlich, ob das Protokoll auch hinsichtlich des Zeitverhaltens *(timing)* korrekt umgesetzt wurde. Da heterogene Systeme einschließlich Multiprozessor-SoC (MPSoC [18]) zunehmend den Schwerpunkt im Systementwurf bilden, ist die Echtzeitfähigkeit für einen Verifikationsansatz, welcher ein komplettes System berücksichtigt, anzustreben. Der Kostenaufwand für den Einsatz von Simulatoren ist abgesehen von den Lizenzkosten des eigentlichen Simulators gering. Für die Bedienung ist eine kurze Einarbeitungszeit notwendig. Der größte Kostenaufwand entfällt auf die Erstellung der Testbenches. Simulatoren bilden im Test einzelner Komponenten während des Entwurfsprozesses eine wichtige Unterstützung. Für Funktionstest in realen Anwendungsumgebungen komplexer heterogener MPSoCs sind Simulatoren jedoch nur bedingt geeignet.

2.1.2 Emulation

Emulation basiert auf der Abbildung des Testobjekts oder des Testverfahrens auf eine dedizierte Infrastruktur. Ein Emulator (lat. aemulari nachahmen) ahmt die Funktion des Zielgerätes nach. Die größte Anwendung erfahren Emulatoren in der hardwarenahen Programmierung von Mikroprozessoren oder digitalen Signalprozessoren (DSP - Digital Signal Processor). Diese Emulatoren werden als *In-Circuit*-Emulatoren bezeichnet und bilden alle Funktionen des zu testenden Prozessors ab. Die Programmierung erfolgt über eine direkte Schnittstelle (JTAG - Joint Test Action Group) zwischen dem Registersatz des Prozessors und der Entwicklungsumgebung. Der Anwender kann mit Hilfe des Emulators Programme erstellen und diese vor der direkten Implementierung auf dem Zielprozessor testen und aufgetretene Fehler beheben (*de-bug*). Da nicht korrekte Programme den Zielprozessor bzw. dessen Peripherie beschädigen können, wird durch Einsatz eines Emulators die Implementierungsphase erheblich sicherer und durch die besseren *Debug*-Möglichkeiten deutlich verkürzt. Der Anwender bekommt durch den Emulator zusätzlich einen direkten Zugriff auf die Ressourcen des Prozessors und kann Schrittweise die Reaktionen des Systems verfolgen. Im Schaltkreis- und Systementwurf kommen Emulatoren ebenfalls zum Einsatz. Man unterscheidet grundsätzlich zwei Verfahren:

Instruction-Set Emulation (ISE[12]) und Hardware-Emulation [74]. Die beiden Verfahren unterscheiden sich in der Abstraktionsebene, in der sie angewendet werden. Während mit der *Instuction-Set* Emulation Befehlsfolgen eines System verifiziert werden, bildet man für die Hardware Emulation den gesamten Entwurf auf die Emulationsplattform ab. Neben der ISE verweist die Literatur auch auf die *Instruktion-Set* Simulation (ISS [82]).

ISS-Verfahren ordnen sich in die Simulation ein, werden aber an dieser Stelle eingeführt, da sie im direkten Zusammenhang mit der ISE gesehen werden. Beide Verfahren werden in einer hohen Verifikationsebene angewendet und stellen eine Maßnahme zur Reduktion der Testkomplexität dar. Einer Reduzierung der Komplexität kommt, wie bereits beschrieben, sowohl in der Simulation als auch in der Emulation eine wesentliche Bedeutung zu. Während sich eine geringere zu verifizierende Komplexität in der Simulation direkt auf die Zeitkosten auswirkt, führt eine reduzierte Komplexität in der Emulation neben dem geringeren Zeitaufwand zu einer Kosteneinsparung für den Aufbau der Emulationsplattform. Ein weiterer Grund für eine ISE ist die kostengünstige Abbildung des Testobjekts auf die verwendete Plattform. Für eine *Instuction-Set*-Emulation lassen sich Standard-Prozessoren verwenden, auf denen der zu verifizierende Prozessor als emulierte Software abgearbeitet wird. ISE wird zusätzlich zum Test von Befehlsfolgen

bereits verifizierter Systeme genutzt, wenn neue Applikationssoftware implementiert wird oder Veränderungen am System vorgenommen wurden. Für die Verifikation eines im Entwurf befindlichen Systems ist die ISE nur bedingt geeignet.

Die Aufgabe der Verifikation oder des funktionalen Tests ist es, den Entwurf eines Systems zu unterstützen. Dafür werden an die Verifikationsmethoden komplexe Anforderungen gestellt. Während der Verifikation eines SoCs wird der Entwurf unter anderem auf der RTL-Ebene (RTL - Register Transfer Level) überprüft. Das macht den Einsatz einer Verifikationsplattform notwendig, welche in der Lage ist, die RTL-Ebene bzw. die Netzliste des Systems abbilden zu können. Dafür werden Hardware-Emulatoren eingesetzt, welche auf wieder verwendbaren Plattformen basieren. Diese kostenintensiven Testsysteme bestehen hauptsächlich aus programmierbaren Logikbausteinen (FPGAs - Field Programmable Gate Arrays).

Vergleichbar mit dem *Compile*-Vorgang in der Simulation wird in einem Syntheseprozess die Beschreibung des SoCs auf ein für einen FPGA nutzbares Datenformat (*bit-File*) abgebildet. Auf dem FPGA wird im Anschluss die logische Funktionalität der Zellen des SoCs emuliert. Durch die hohe Verarbeitungsgeschwindigkeit eines FPGAs und auf Grund der hohen Parallelität des *Gate Arrays* werden neben läufige Prozesse adäquat behandelt und der Testprozess gegenüber der Simulation signifikant beschleunigt.

Bedingt durch die stetig steigende Größe von SoC-Entwürfen und durch die Entwicklung von Multiprozessoren reicht die Anzahl von zur Verfügung stehenden Gattern (*Gate*s) eines einzelnen FPGAs nicht mehr aus. Aus diesem Grund verschaltet man die einzelnen FPGAs zu einer größeren Entwicklungsplattform. Verbunden mit dieser Art der Erweiterung der Gatter-Kapazität sind zum einen höhere Kosten durch die zusätzlichen FPGAs aber auch zusätzliche Kosten durch den notwendigen Einsatz spezieller Software, mit welcher der verteilte Syntheseprozess durchführt wird. Des Weiteren sinkt die Auslastung der einzelnen Bausteine durch eine Beschränkung der Verbindungsanschlüsse zwischen den FPGAs.

Hardware-Emulatoren eigenen sich auf Grund der erreichbaren Taktzyklen zur Integration in komplexere Umgebungen, welche sich auch für den Test von Schnittstellen und der Peripherie eines heterogenen SoCs eignen. Durch die großen Kosten der FPGAs und das zusätzliche *Know-How* für die FPGA-Synthese ist dieses Verfahren bislang nur in wenigen großen Unternehmen verbreitet.

2.2 Hardware-basierte Test- und Verifikationsverfahren

Im folgenden Absatz werden hardwareunterstützte Verfahren für den funktionalen Test von Schaltkreisen und Systemen beschrieben und gegenübergestellt. Der Fokus ist auf FPGA-basierte Ansätze für die Emulation von Netzlisten gerichtet.

Der Einsatz von FPGAs bietet eine flexible aber kostenintensive Möglichkeit, den Entwurf während der funktionalen Tests zu elaborieren und in bestimmtem Maße zu korrigieren. Durch die unterschiedlichen Anforderungen und den Grad der Produktreife gelten für den Test von Schaltkreisen oder komplexen Systemen konträre Randbedingungen. Diese lassen sich nicht in einem all umfassenden Test- oder Verifikationsablauf zusammenführen. Zum Beispiel kann der Funktionstest eines Systems erst erfolgen, wenn ein Muster oder Prototyp zu Verfügung steht. Auf Grund des Risikos und der hohen Kosten für eine nicht erfüllte Spezifikation oder ein nicht oder unzureichend funktionierendes Produkt muss der Prozess der Verifikation den gesamten Systementwurf begleiten. Zu intensive oder zu lange Verifikations- und Testphasen verursachen aber sehr hohe Kosten, zu denen konträr die Forderung steht, das Produkt sehr schnell zu entwickeln (*Time to Market*), um die Gesamtkosten zu reduzieren. Die Verifikationsmethode wird aus

diesem Grund hinsichtlich der Komplexität an die jeweilige Entwurfsphase angepasst. Der Verifikationsplan muss über alle Entwicklungszyklen des Entwurfs mitgeführt und ständig an den Entwicklungsstand angepasst werden. Die folgenden funktionalen Verfahren werden in unterschiedlichen Phasen eines Entwurfs eingesetzt. Mit Ausnahme der HW-Emulation, welche in Abschnitt 2.2.3 beschrieben wird, beziehen sich sowohl das *Rapid Prototyping* als auch die *Hardware-in-the-Loop* Verfahren im Schwerpunkt auf den Test von *embedded Systems.* Der Übergang vom funktionalen Test eines Schaltkreises oder SoCs auf ein *embedded System* ist in zunehmendem Maße bedingt durch die steigende Integrationsdichte fließend. Aus diesem Grund ist eine klare Abgrenzung der einzelnen funktionalen Testverfahren nicht möglich und hängt sehr stark vom Einsatzzweck ab.

2.2.1 Rapid-Prototyping

Rapid Prototyping ist ein Begriff, der aus dem Maschinenbau und der Fertigungstechnik abgeleitet ist. In diesem Bereich ist die Herstellung bestimmter Baugruppen sehr kostenintensiv, ein Modell der Baugruppe wird aber in einem sehr frühen Stadium des Entwicklungszyklus benötigt. Aus diesem Grund fertigt man aus günstigen Werkstoffen und den zur Verfügung stehenden CAD-Daten (CAD - Computer Aided Design) einen Prototypen. Die Fertigung wird ohne größeren Aufwand und ohne finanzielles Risiko mit speziellen Maschinen durchgeführt. Aus diesem Grund wird dieses Verfahren als schneller Prototypenbau (*rapid prototyping*) bezeichnet.

Im Entwurf elektronischer Systeme findet dieses Verfahren zunehmend Verbreitung [42]. Auf einer speziellen und an den Entwurf angepassten Plattform wird ein Prototyp des Systems implementiert. Diese Plattform umfasst alle für den späteren Einsatz des Systems relevanten funktionalen Komponenten. Dazu gehören Recheneinheiten (z.B. DSP, FPGA), Schnittstellen, Speicher oder bestimmte Peripheriekomponenten (z.B. mechanische Baugruppen). Nicht funktionale Randbedingungen wie z.B. Baugröße oder Form bleiben zunächst unbeachtet. Das *Rapid Prototyping* kann sowohl für Software als auch für Hardware angewendet werden [60]. Für die Entwicklung von Hardware-Komponenten für eingebettete Systeme (*embedded systems*) bietet eine *Rapid Prototyping*-Plattform eine geeignete Basis, da sie über den gesamten Entwurfsablauf für die Verifikation genutzt werden kann [88]. Die Heterogenität eingebetteter Systeme stellt für die Zusammenschaltung der Komponenten eine der größten Herausforderungen dar. Der Test und die schrittweise Inbetriebnahme des Gesamtsystems in Verbindung mit der parallel erfolgenden Weiterentwicklung und Anpassung der Komponenten führt sehr schnell zu einem funktionsfähigen Demonstrator. Durch das Zusammenspiel von Hardware und Software, welches die Grundlage moderner Systeme bildet und durch Entwurfsansätze aus dem Gebiet des Hardware/Software-CoDesign [44, 117, 42] unterstützt wird, eignen sich Prototypen-Plattformen für die zeitgleiche Verifikation beider Systemteile. Während dieser Verifikation ist es den Entwicklern möglich, das entstehende System auf laufzeitkritische- oder Echtzeitanforderungen zu überprüfen. Funktioniert der Demonstrator gemäß der Spezifikation, wird mit den gewonnenen Referenzwerten ein erstes Funktionsmuster des Systems erstellt.

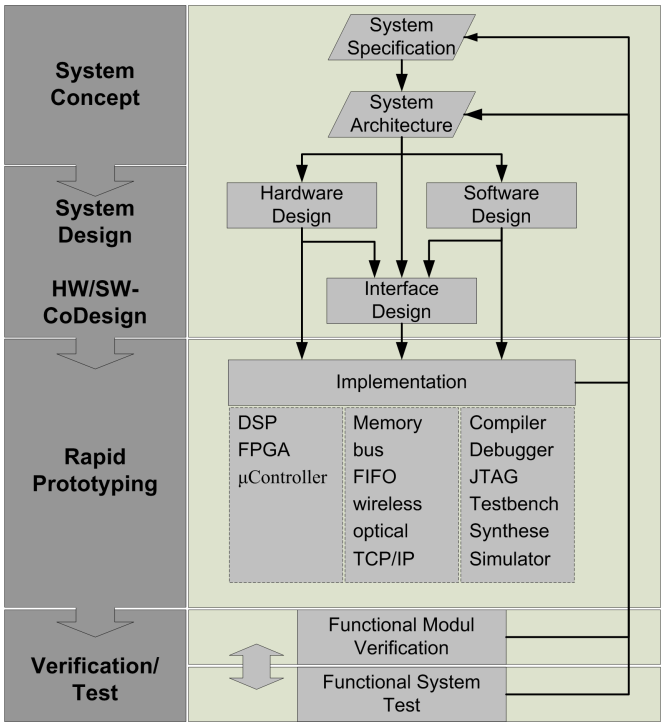


Abbildung 2.1: Entwurfsablauf Rapid Prototyping

Auf Grund der hohen Kosten für spezielle Systeme oder einzelner Baugruppen dieser Systeme, wie z.B. FPGAs, werden die Prototypen von einer zunehmenden Zahl von Anbietern durch stetige Weiterentwicklungen zu einem fertigen Produkt geführt. Das ist vor allem in der Fertigung von Datenübertragungssystemen mit sich ständig verändernden Parametern wie z.B. Empfangskarten für Mobilfunk [10] (*back bone*) der Fall.

Während der Entwicklung der Funkkarten sind häufig die genauen Parameterwerte der zu Grunde liegenden Übertragungsstandards nicht klar definiert. Die Hersteller reagieren darauf mit flexibel programmierbaren Komponenten in den Systemen wie z.B. der Integration von FPGAs. Die entstehenden hohen Kosten eines Systems im Verhältnis zu den geringen Stückzahlen werden in diesem Fall durch den direkten Einsatz der Prototypen in den Übertragungssystemen reduziert. Die Daten für die funktionalen Tests werden dem System ähnlich der Testbench eines Simulators über die Peripherie zugeführt. Dadurch können spezielle Testfälle (*corner cases*) gezielt überprüft werden.

2.2.2 Hardware-in-the-Loop

Der *Hardware-in-the-Loop* (HIL - Hardware-in-the-Loop) Ansatz ist wie das *Rapid Prototyping* ein auf einem existierenden System basierendes Verfahren [84]. Das zu testende System wird in einem sehr späten Entwicklungsstadium in eine simulierte und dadurch reproduzierbare Anwendungsumgebung integriert. Dadurch ist mit einem HIL-System ein Test unter realen Anwendungsbedingungen möglich. Fehler, welche in der Spezifikation und den zur Erfüllung zu Grunde liegenden Tests nicht erkannt wurden können mir diesem Verfahren erkannt und korrigiert werden. Die Umgebung, wie z. B. ein Automobil, wird dabei

über Modelle simuliert (z.B. Simulink Modelle auf dSpace-Systemen), um die korrekte Funktion des zu testenden Systems wie z. B. die eines Motorsteuergerätes zu überprüfen. Die Eingänge des Steuergerätes werden mit Daten aus dem Umgebungsmodell stimuliert und schreiben Ergebnisse und Aktionen an diese Umgebung zurück. Die entstehende Schleife (*Loop*) führt zu einem iterativen Entwicklungsprozess.

Die HIL-Simulation wird dafür in Echtzeit durchgeführt. Durch das reproduzierbare Verhalten der Simulation können neue Entwicklungsversionen unter den gleichen Kriterien getestet werden wie die Vorgängerversion. Es wird dadurch detailliert und in Betrachtung des Zusammenspiels mit anderen Komponenten nachgewiesen, ob ein Fehler behoben wurde oder nicht. HIL-Systeme reduzieren den hohen Kostenaufwand für Tests in der realen Umgebung wie z.B. den Testreihen realer Automobile.

Der signifikanteste Unterschied zwischen der Verifikation mit *Rapid Prototyping*- und HIL-Systemen ist die Komplexität der Testumgebung [84]. Im *Hardware-in-the-Loop* Verfahren wird durch die umfangreichere Abbildung der Umgebung [110] ein wesentlich realeres Testverhalten erreicht als im *Rapid Prototyping*. Des Weiteren befindet sich das zu testende System für HIL-Verfahren in einem weiter fortgeschrittenen Stadium des Entwurfs als dies beim *Rapid Prototyping* der Fall ist.

In HIL-Systemen wird ein Test unter Echtzeitbedingungen angestrebt ist aber auf Grund der Komplexität der Umgebungsmodelle nicht in jedem Fall abzusichern. Durch eine Reduzierung der Testkomplexität für einzelne Baugruppen wird diesem Problem entgegengewirkt. Die Kosten für ein HIL-System sind bedingt durch den großen Rechenaufwand, die Echtzeitfähigkeit und den Lizenzkosten für die spezielle Software deutlich höher als im *Rapid Prototyping*.

2.2.3 HW-Emulation im IC-Entwurf

Für die Verifikation von Schaltkreisen und komplexen SoCs treten aus Gründen der Performanz zunehmend wieder verwendbare HW-Emulatoren in den Mittelpunkt. In der FPGA-basierten Hardware Emulation wird das entworfene digitale System nach der Synthese auf einen oder mehrere FPGAs abgebildet und funktional getestet. Hierbei lassen sich durch die hohe Taktrate eines FPGAs im Gegensatz zur Simulation wesentlich größere Zeitabschnitte verifizieren (vgl. Abs. 2.1.1). Der Performanzvorteil eines Emulators nimmt durch die parallele Verarbeitung eines FPGAs gegenüber der Simulation deutlich mit steigender Komplexität des zu testenden Systems signifikant zu (vgl. Abb. 2.2).

Eine weitere Motivation für die HW-Emulation ist die Integration von Schnittstellen an das FPGA. Dafür ist eine umfangreiche Prototypen-Umgebung (vgl. *Rapid Prototyping*) notwendig, durch welche sich das gesamte Systemverhalten unter Berücksichtigung von Interaktionen an den Schnittstellen abbilden lässt. Diese Verifikationskomplexität ist mit einem Simulationsansatz nur unter Einschränkungen und mit großem Ressourcenaufwand der zu Grunde liegenden Recheneinheit realisieren.

Vorraussetzung für einen herkömmlichen HW-Emulator ist die homogene Abbildung der Gatter-Netzliste auf einer wieder verwendbaren Plattform. Der folgende Abschnitt gibt einen Überblick über FPGA-basierte HW-Emulatoren. Zunächst werden unterschiedliche Ansätze vorgestellt und hinsichtlich Flexibilität und Effizienz bewertet. Im darauf folgenden Abschnitt stehen bekannte Möglichkeiten der Optimierung von HW-Emulatoren im Fokus. Daraus wird der Ansatz der Arbeit für die Optimierung von HW-Emulatoren durch den Einsatz von Methoden der partiellen Rekonfiguration zur Laufzeit abgeleitet.

In Abbildung 2.11 ist im linken Teil der Syntheseprozess für ein System dargestellt, welches auf einem FPGA realisiert wird. Im rechten Teil der Abbildung ist die Synthese für einen Entwurf zu sehen, welcher in weiteren technologischen Prozessen bis zu einem dedizierten Schaltkreis (ASIC) entwickelt wird. Zu erkennen ist weiterhin, dass beide Syntheseprozesse bis zur Abbildung des Entwurfs in einer festgelegten

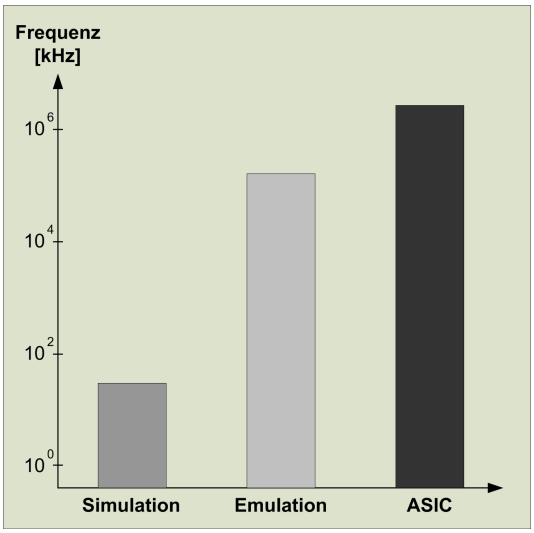


Abbildung 2.2: Performancevergleich Verifikation [11]

Technologie (FPGA oder ASIC) identisch verlaufen.

Daraus kann abgeleitet werden, dass auch die Verifikationsansätze bis zur Integration der Technologie identisch sind. Das ist die grundsätzliche Voraussetzung für den Einsatz von
Hardware-Emulatoren im technologieunabhängigen Test von Schaltkreisen und Systemen. Das Verifikationsergebnis wird auf RT-Ebene generiert, d.h. die Signaländerungen werden taktgenau ausgegeben. Die *back-annotierte* oder *Backend*-Verifikation für einen ASIC, in welcher die technologisch und physikalisch bedingten Effekte wie *Glitches*, Treiberleistungen, Laufzeitverhalten der Gatter (*Gate*s) usw. mit einfließen, ist mit diesem Ansatz nicht realisierbar.

Für den Einsatz eines HW-Emulators ist im ersten Schritt eine Synthese (vgl. Abb. 2.11) durchzuführen. Nach der Abbildung des Entwurfs auf eine Zielarchitektur und der Generierung der Konfigurationsdaten für die Programmierung der *LuTs* des FPGAs wird dieser programmiert. Das System, in welches das FPGA integriert ist (vgl. Abb. 2.3), wird nach Inbetriebnahme mit Eingangsdaten stimuliert. Dadurch werden die Schaltvorgänge der einzelnen CLBs im FPGA angeregt und der funktionale Test beginnt. Nach einer festgelegten Anzahl von Takten oder auf ein bestimmtes Ereignis wird der Testlauf beendet. Die generierten Ergebnisse des Entwurfs werden entweder über Ausgänge des FPGAs an die Testplattform zurück geschrieben oder direkt aus dem FPGA ausgelesen. Dieses Verfahren (z.B. *Xilinx ChipScope* [116]) greift auf die internen Speicherregister des FPGAs zu und schreibt die gespeicherten Werte über den JTAG-Bus zurück an ein Basissystem (*Host System*).

Die Ergebnisse des Syntheseprozesses für einen FPGA beeinflussen wesentlich die Performanz des gesamten Systems. Aus den Ergebnissen der *Place&Route*-Phase lassen sich die Taktrate und die Auslastung des FPGAs ableiten. Auf Grund des abnehmenden Optimierungspotentials und der begrenzten Anzahl an Leitungen innerhalb eines FPGAs wird der Algorithmus für die Logikoptimierung ab einem bestimmten Punkt keine Verbesserung mehr erreichen. Die begrenzte Anzahl an Leitungen zwischen den zu diesem Zeitpunkt platzierten Gattern beeinflussen den schnellst möglichen Takt des gesamten Systems. Die Verdrahtung weiterer Gatter auf dem FPGA führt dann zu einer Verlängerung des kritischen Pfads (CP - Critical Path). Aus diesem Grund sinkt die erreichbare Performanz eines Entwurfs auf einem FPGA signi-

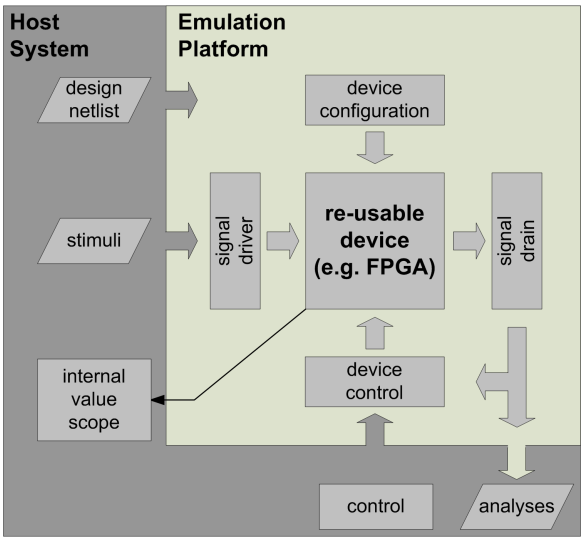


Abbildung 2.3: FPGA basiertes Emulations-System

fikant mit der zunehmenden Auslastung des FPGAs. Ab einer bestimmten FPGA- und entwurfsabhängigen Grenze (vgl. Abb. 2.4) reduziert sich der maximal mögliche Takt, obwohl das FPGA nicht vollständig ausgelastet ist.

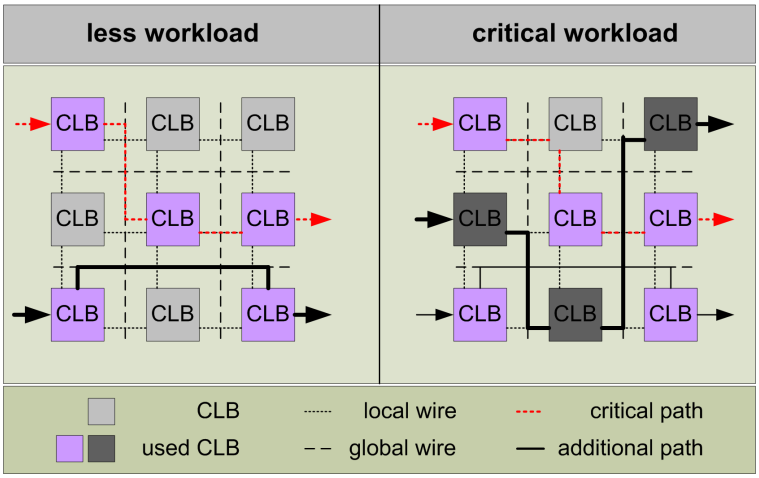


Abbildung 2.4: Einfluss der Auslastung eines FPGAs auf den kritischen Pfad

Aus diesem Grund, und um die generelle Grenze der zu testenden Entwürfe nicht zu beschränken, werden für große Emulator-Systeme nicht einzelne FPGAs sondern vernetzte Felder aus mehreren FPGAs eingesetzt werden.

2.2.4 Multi-FPGA Emulator-Systeme

Multi-FPGA Systeme lassen sich hinsichtlich der Anzahl an FPGAs grundsätzlich frei parametrisieren. Ein wichtiges Kriterium für den Einsatz dieser Systeme ist das Verhältnis von Ein- und Ausgängen der eingesetzten FPGA-Bausteine im Verhältnis zur Anzahl an programmierbaren Logikgattern innerhalb der Bausteine. Dieses Verhältnis beeinflusst wesentlich die Effizienz von Multi-FPGA Systemen und wird in Rent's Rule [29] mathematisch abgeschätzt:

$$P \propto G^f \qquad \begin{cases} f \sim 0.5\ (structured\ design), \\ f \geq 0.5\ (unstructured\ design), \end{cases} \tag{2.1}$$

$$P \widehat{=} \#\mathrm{I/O\ Pins}$$
$$G \widehat{=} \#\mathrm{Gates}$$

In Gleichung 2.1 [97] wird die synthetisierte Netzliste eines Entwurfs in *strukturiert* und *unstrukturiert* unterschieden. Die Struktur der Netzliste wird im Wesentlichen in der Entwurfsphase (vgl. Abb. 2.11) bestimmt. Strukturierte Entwürfe [51] werden vom Synthesewerkzeug in einem größeren Grad optimiert und sind dadurch unter Verwendung von weniger Ressourcen implementierbar. Durch die Reduzierung redundanter Verbindungen innerhalb einer Schaltung kann der *Place&Route*-Prozess das System optimaler auf das FPGA und auf die integrierte Standardlogik (vgl. Abb. 2.7) abbilden. Diese Randbedingung gewinnt in der Zusammenschaltung von einzelnen FPGAs zu Multi-FPGA Systemen noch größeren Einfluss. Aus Gleichung 2.1 ist abzuleiten, dass sich ein strukturierter Entwurf auf einem Multi-FPGA System deutlich günstiger abbilden lässt als ein unstrukturierter Entwurf. In Bezug auf das Verhältnis der der I/O-*Pins* zur Anzahl der Gatter innerhalb eines FPGAs ist festzustellen, dass mit einer zunehmenden Anzahl an I/Os bei gleich bleibender Anzahl an Gattern die Effizienz der Abbildung ansteigt. Dieser Zusammenhang hängt mit der größeren Anzahl an direkt verbundenen FPGAs zusammen. Sinkt die Anzahl der I/O-*Pins*, werden zunehmend mehr Leitungen über weitere FPGAs geleitet (vgl. Abb. 2.7). In Abbildung 8.3 ist die Komplexität hinsichtlich der Verbindungsstruktur aktueller Multi-FPGA Emulatoren zu erkennen.

Durch die deutlich geringere Anzahl an Verbindung zwischen den einzelnen FPGAs ist ein angepasstes Verbindungsnetzwerk (*Routing Network*) notwendig. Hierbei wird zwischen statischen und dynamischen *Routing*-Verfahren unterschieden. Werden die FPGAs statisch verschaltet, sind die Verbindungspunkte bereits während des Entwurfsprozesses derartiger Emulatoren festgelegt. Nachträglich ist es nicht mehr möglich, die Verbindungstopologie zu ändern. Die Hersteller von statisch verbundenen Multi-FPGA Systemen verfolgen anwendungsbedingt unterschiedliche *Routing*-Topologien.

In Abbildung 2.5 sind die Verbindungen als Maschen in regulären Strukturen zwischen den FPGAs platziert. Wird eine kurze Distanz für die Verbindungen gewählt (z.B. *8way single hop* Abb. 2.5), lässt sich ein Entwurf, in welchem die Gatter örtlich lokal platziert sind, mit einem geringen Grad an nicht genutzten Ressourcen platzieren. Enthält der Entwurf breite Busstrukturen, ist ein Verbindungsnetzwerk, welches große Distanzen auf der Entwicklungsplattform abbildet, am effizientesten. Auf Grund der hohen Kosten für eine Multi-FPGA Emulatorplattform trägt ein System, welches für spezielle Entwürfe optimiert wurde, ein hohes Risiko in der späteren Nutzbarkeit unter veränderten Randbedingungen.

Deshalb gehen die Hersteller von Emulatorsystemen zunehmend dazu über, das Verbindungsnetzwerk flexibel zu gestalten. Dazu werden die Leitungen nicht direkt zwischen den FPGAs verbunden, sondern

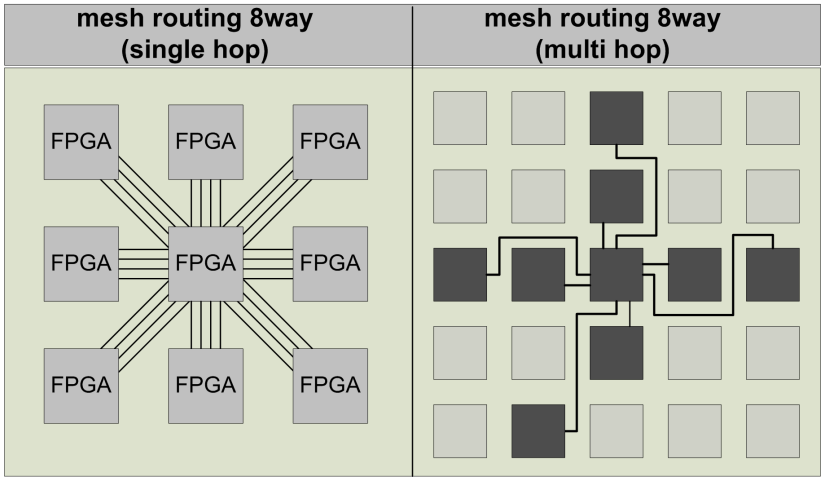


Abbildung 2.5: Statisches Mesh-Routing für Multi-FPGA Systeme [52]

sind an Netze von Schaltern (*Crossbar Switch*) angeschlossen. Die Schalter werden vor der Konfiguration der FPGAs beschaltet und stellen die Verbindungen in Abhängigkeit von der Netzliste des zu testenden Entwurfs (vgl. FPGA Emulator 2.2.4) her.

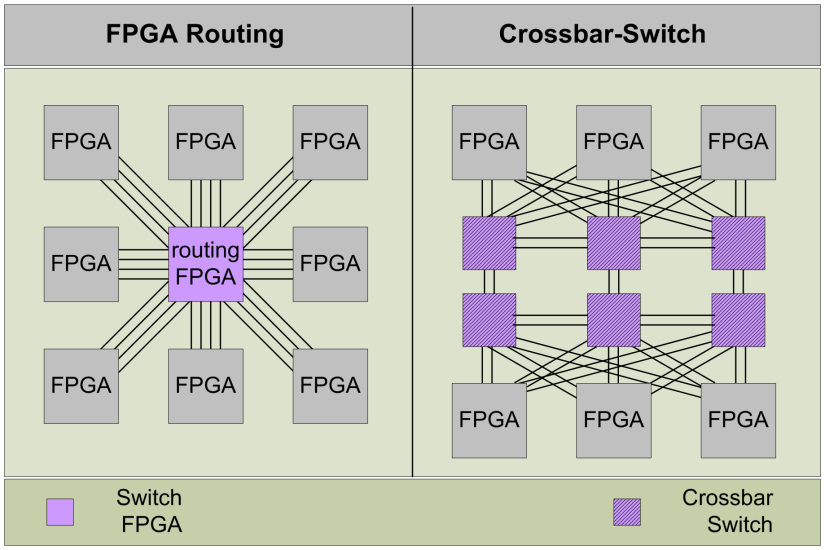


Abbildung 2.6: Dynamisches Crossbar-Routing für Multi-FPGA Systeme [107]

Eine weitere Möglichkeit, die FPGAs dynamisch zu verbinden, ist ein *Routing*-Netzwerk über ein zusätzliches FPGA. Dieses FPGA wird als flexibles, programmierbares Netzwerk für die entsprechenden Verbindungen konfiguriert (vgl. Abb.2.6).

Neben der zusätzlichen Flexibilität durch die variable Verdrahtung sind diese Systeme mit geringem Aufwand erweiterbar. Ein Nachteil des beschriebenen Ansatzes ist die durch die zusätzliche Latenz der

Schalter oder des *Switch*-FPGAs bedingte reduzierte Gesamtperformanz des Emulators. Diese Latenz führt in sehr komplexen Emulatorsystemen zu einer erheblichen Reduzierung der erreichbaren Taktfrequenzen. Für die Abbildung einer kombinatorischen Schaltung auf zwei FPGAs wird der kritische Pfad durch die Treiberstufen an den Ein- und Ausgängen auf die doppelte Anzahl an Takten verlängert. Auf Grund der geforderten Synchronität im gesamten System und dem damit notwendigen Einfügen von zusätzlichen Registerstufen in die verbleibenden Systemverbindungen (z.B. **G1**-**G2** in Abb. 2.7) führt ein *Cross-Routing*-Verfahren zu wesentlich größeren Latenzzeiten in der gesamten Emulation.

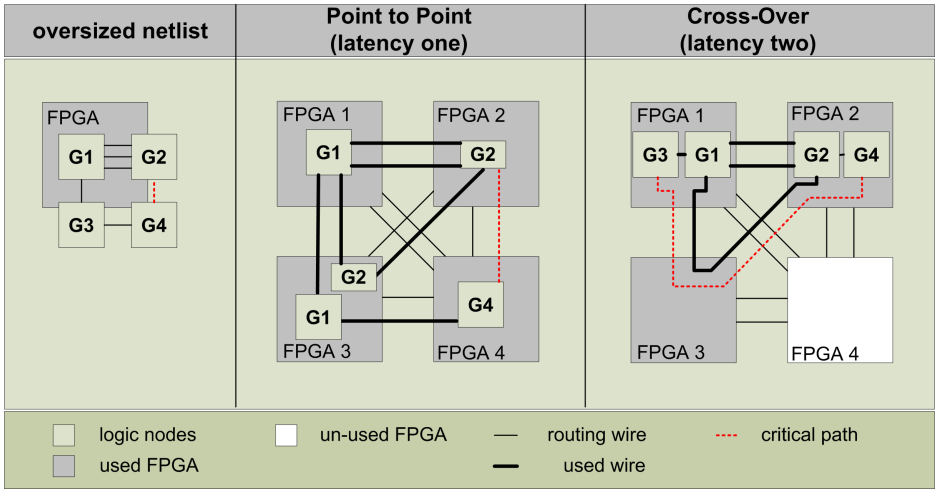


Abbildung 2.7: Einfluss des Routing auf die Latenz

In Abbildung 2.7 sind zwei unterschiedliche Ansätze für die Platzierung einer Netzliste dargestellt. Die Anzahl der Gatter in der Netzliste übersteigt die Anzahl der für die Emulation zur Verfügung stehenden Gatter der Plattform. Aus diesem Grund werden Teilnetzlisten (Abb. 2.7 **G1**-**G4**) auf einem Multi-FPGA-System verteilt platziert. Die angenommene Randbedingung des Beispiels in Abbildung 2.7 beschränkt die Verbindungen auf zwei Leitungen zwischen den einzelnen FPGAs und zwei platzierbare Teilnetzlisten auf jedem separaten FPGA. In der Teilabbildung 2.7 (*Point-to-Point*) wird die Netzliste unter Berücksichtigung der Anzahl der Verbindungen platziert. Die Teilnetzliste **G2** wurde zusätzlich in zwei kleinere Teilnetzlisten aufgeteilt. Der kritische Pfad verlängert sich um die Latenz einer Verbindung zwischen zwei FPGAs. Der Einfluss auf das gesamte System ist zwischen allen Verbindungen gleich. Die Anzahl der zur Verfügung stehenden Verbindungen reicht nicht aus, um alle Leitungen des Systems direkt zwischen den FPGAs zu verbinden. Aus diesem Grund sind die Verbindungen zwischen **G3** und **G4** bzw. **G1** und **G2** über ein drittes FPGA verdrahtet. Diese Anordnung reduziert einerseits die Anzahl der FPGAs auf drei, führt jedoch zu einer Verlängerung des kritischen Pfads um die Latenz von zwei FPGA-Übergängen. Das reduziert, wie bereits beschrieben, die gesamte Performanz erheblich.

Die Entscheidung für ein statisch oder ein dynamisch verdrahtetes Emulator-System ist abhängig von den Anforderungen der funktionalen Tests. Werden viele homogene Entwürfe verifiziert, sind statisch verdrahtete FPGA-Plattformen auf Grund der niedrigen Latenz und der damit verbundenen höheren erreichbaren Taktrate wesentlich besser geeignet. Sind die Anforderungen an die Plattform variabel, haben die Ansätze, welche auf *Crossbar*-Schaltern oder *Routing*-FPGAs basieren, deutliche Vorteile hinsichtlich der

Flexibilität. Diese Ansätze erreichen aber durch die Latenz der Verdrahtungen nicht die Performance von statisch verbundenen Systemen. Die Kosten dynamisch verdrahteter Systeme sind auf Grund der zusätzlichen Hardware außerdem wesentlich höher.

Die Hersteller reagieren auf die divergierenden Anforderungen mit heterogen verdrahteten Systemen (Tab. 2.1 Emulator der Firma *Nallatech).* Für lokale Verdrahtungen setzt die Firma statisch verdrahtete Aufsteckkarten mit jeweils zwei FPGAs ein. Diese Karten sind in ein komplexes Modulsystem integriert, welches Steckplätze für zusätzliche Einsteckkarten verwalten kann. Diese Einsteckplätze werden dann über ein flexibel programmierbares und synchrones Bussystem verbunden. Dadurch lassen sich unterschiedliche Szenarien für die Emulation von Gatter-Netzlisten aufbauen und dimensionieren.

Produkt	**BenNUEY PCI-X (vgl. Anhang A18.1)**	**DN8000K10 PCI**	**CHIPit Platinum Edition V5**	**DN8000K10 (vgl. Anhang A28.3)**
Hersteller	Nallatech	The DiniGroup	Pro Design Electronics Cooperation	The DiniGroup
Anzahl FPGA	flexibel (min 2 Virtex4)	1xVirtex FX60 2xLX100	3-21 Virtex5 FPGA XC5VLX330	14 Virtex LX 200, 2VirtexFX100
Routing	lokal (statisch) global (flexibel)	statisch	flexibel mit 3D Switching Technology	statisch
Anwendung	ASIC Emulator	ASIC-Prototyping	ASIC&SoC-Prototyping (28mil Logik Gatter)	ASIC-Emulation für große Entwürfe (25mil Logik Gatter)

Tabelle 2.1: FPGA-basierte Emulator-Systeme

Ein weiterer Ansatz für die Minimierung der Verdrahtungslatenz ist der Einsatz von seriellen Verbindungen zwischen zwei FPGAs. Dieser, von der Firma *Xilinx* in den FPGAs ab der *VirtexII*-Serie implementierte Ansatz, nutzt zwei I/O-Leitungen für die differentielle Übertragung von seriellen Protokollen. Die Verbindungen werden mit einer Taktrate, welche sehr viel höher ist als der *On-Chip*-Takt, betrieben. Mit diesem Verfahren (*RocketIO* [117]) sind Übertragungsraten von über 3Gbit/s erreichbar. Ein Nachteil des Ansatzes ist die zur Verwaltung des Protokolls zusätzlich zu implementierende Hardware. Dadurch reduziert sich die nutzbare Fläche innerhalb des FPGAs. Zusätzlich müssen die FPGAs synchronisiert sein, um die aus dem seriellen Protokoll geschriebenen Daten zu den akkuraten Werten des Entwurfs stimulieren zu können. Aus diesem Grund setzen verwandte Ansätze [99] zur Vergrößerung der Bandbreite zwischen den FPGAs ein Zeitmultiplex für die Nutzung der verfügbaren FPGA-Verbindungen ein. Dazu wird ein zusätzlicher HW-Rahmen, welcher die parallelen Datenkanäle des zu emulierenden Entwurfsteil serialisiert um diesen Entwurfsteil platziert. Dieser HW-Rahmen wird mit einem höheren Takt als der zu testende Entwurfsteil betrieben und überträgt die seriellen Daten an einen weiteren HW-Rahmen. An dieser Stelle werden die seriellen Daten wieder in ein paralleles Format gewandelt. Dadurch ist es möglich,

die Bandbreite zwischen den FPGAs ohne zusätzlichen Aufwand für die Synchronisation zu realisieren. Die Ansätze mindern das Problem der beschränkten Anzahl an Verbindungen zwischen den FPGAs, verringern aber ebenso deutlich die Performanz des Gesamtsystems, da die Übertaktung nur in bestimmten Grenzen zulässig ist. Die Grenzen hängen von der HW-Architektur des FPGAs ab.

In aktuellen Arbeiten [73, 87, 2] wird das Problem der durch die Bandbreitenbeschränkung ungünstigen Auslastung von FPGAs und den damit verbundenen hohen Gesamtkosten pro
Emulator-Gatter aufgegriffen. Eine signifikante Optimierung wird in den Ansätzen durch eine verbesserte Verbindung zwischen den FPGAs oder durch eine mehrfache zeitliche Ausnutzung der einzelnen FPGAs erreicht. Der folgende Abschnitt stellt unterschiedliche bereits publizierte Konzepte und Realisierungen für die Optimierung gegenüber und bewertet diese.

2.2.5 Ansätze zur Optimierung von HW-Emulation

Neben den genannten Verfahren für die optimale Ausnutzung der beschränkten Verbindungen zwischen den FPGAs existieren weitere Ansätze für die Optimierung von FPGA-basierten HW-Emulatoren. Diese Ansätze greifen grundlegend in die Wirkungsweise der Emulatoren ein und werden deshalb in diesem Abschnitt separat vorgestellt. Aus diesen Ansätzen wird im zweiten Teil das dieser Arbeit zu Grunde liegende Potential für eine Optimierung abgeleitet.

Wie bereits dargestellt, basieren HW-Emulatoren auf der Abbildung einer Entwurfsnetzliste auf einen oder mehrere FPGAs. Die dafür notwendige Partitionierung berücksichtigt keine funktionalen Aspekte des Entwurfs. Die Platzierung auf den FPGAs folgt den Randbedingungen des Emulatorsystems und nicht denen des Entwurfs.

Für die Optimierung auf Logikebene wird die Netzliste als flacher Graph von Knoten betrachtet [41]. Dadurch ist eine Berücksichtigung der Struktur des Entwurfs nur vor der Synthese vollständig möglich. Diese Eigenschaft wird in der Co-Simulation ausgenutzt, um kostengünstig die Vorteile der Simulation mit den Vorteilen der Emulation zu verbinden. Dazu ist eine Aufteilung des Verifikationsprozesses notwendig. Ein Teil der Verifikation wird durch einen Simulator berechnet, ein anderer Teil wird synthetisiert und auf einem FPGA emuliert. Es gibt unterschiedliche Ansätze für die Partitionierung. Zum einen ist es möglich, rechenintensive Systemteile unabhängig von deren Funktion im System auf den Emulator auszulagern. Zum anderen kann eine feste Verteilung vorgenommen werden wie z.B. die Auslagerung der ALU oder der DSP-Teile. In [73] und [72] wird ein System vorgestellt (*Apples*), welches auf einem Co-Simulationsansatz eines Simulators mit einem FPGA basiert. Die Kommunikation zwischen dem Simulator und dem FPGA erfolgt über einen auf dem FPGA implementierten *PowerPC*. Dieser kann direkt und taktgenau auf die Logik zugreifen und diese steuern. Das ist notwendig, da der FPGA als *Slave* im Gesamtsystem eingebunden ist. Die Effizienz von cosimulierenden Systemen hängt von der erreichbaren Zykluszeit des FPGAs ab. Die Zykluszeit gibt die Zeitspanne oder die Anzahl der Takte an, die der FPGA ohne Interkommunikation mit dem *Host*-System emulieren kann.

D.h.: besteht das zu emulierende System aus zusammenhängenden kombinatorischen Logikteilen, welche sowohl auf dem Simulator als auch auf dem Emulator verarbeitet werden, ist in jedem Taktzyklus ein Datenaustausch beider Systemteile notwendig. Vergleichbar mit den bereits beschriebenen Nachteilen eines HIL-Ansatzes hängt die also Effizienz wesentlich von der Interaktion des FPGAs mit der Umgebung ab, in diesem Fall dem Simulationsmodell. Aus diesem Grund setzten die Ansätze hauptsächlich auf die Integration von IPs in den Simulationsprozess (Abb. 2.8). Die IPs haben ein definiertes und in sich abgeschlossenen funktionales Systemverhalten, welches über Schnittstellen eingebunden wird. Dadurch ist es

möglich, lange von der Simulation unabhängige Emulationszyklen zu berechnen. Die Testdaten werden in einer für den Software- als auch den Hardwareteil gemeinsamen Testbench generiert.

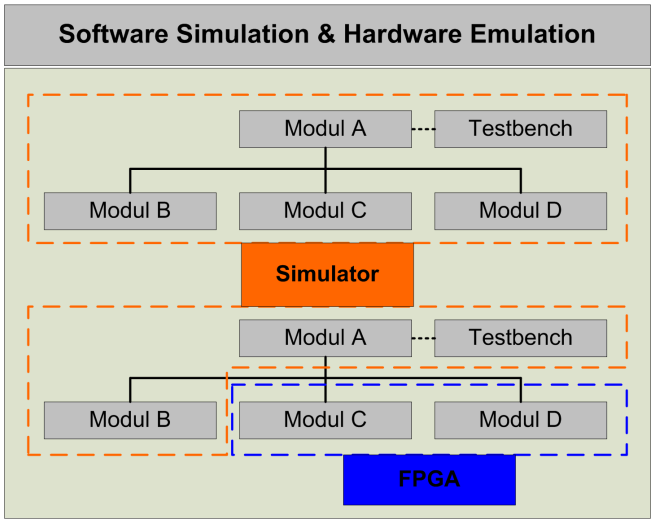


Abbildung 2.8: Hardware/Software Co-Simulation [92]

Weitere Simulator-Emulator Kopplungen für die Beschleunigung einer Simulation werden in [87] und [53] erläutert. In diesen heterogenen Ansätzen wird der Emulator jeweils vom weniger performanten Simulator gesteuert. Dadurch hängt die Performanz und die Auslastung des FPGAs zum einen von der Effizienz des Simulators und der Kopplung ab, zu anderen aber auch direkt von der Struktur des Entwurfs.

Für eine direkte Steigerung der Effizienz eines homogenen FPGA-basierten Emulator-Systems basieren verschiedene Plattformen auf dem Einsatz zur Laufzeit rekonfigurierbarer Strukturen. Die *Erlangen Slot Machine* (ESM - Erlangen Slot Machine[2]) ist ein rekonfigurierbares FPGA-System, welches für die Emulation bzw. das *Rapid Prototyping* entwickelt wurde. Bedingt durch die Struktur des FPGAs muss das zu testende System in *Slots* aufgeteilt werden. Diese *Slots* sind über Speicher und ein Verbindungsnetzwerk untereinander verbunden (vgl. Abb. 2.9). Das Netzwerk ist lokal über die internen Leitungen des FPGAs verbunden. Für eine verbesserte globale Kommunikation ist ein externer *Crossbar Switch* an jeden *Slot* angeschlossen.

Dadurch ist eine uneingeschränkte Kommunikation zwischen den *Slots,* unabhängig von deren Position auf dem FPGA, gegeben. Die externe Verbindungsstruktur beeinflusst und bestimmt signifikant die Performanz des gesamten Ansatzes. Dessen Effizienz hängt wiederum direkt mit dem Datenfluss des zu testenden Entwurfs zusammen und wird dem entsprechend mit unterschiedlichem Verhalten der Stimulidaten auch unterschiedliche Performanz erreichen.

In Abbildung 2.9 ist ein Blockschaltbild der ESM abgebildet. Durch die Verbindung der *Slots* über das extern platzierte *Crossbar Switch* reduziert sich die zur Verfügung stehende Anzahl an Ein- und Ausgängen des FPGAs. Mit zunehmender Anzahl an rekonfigurierbaren *Slots*, d.h. mit einer feineren Granularität, erhöht sich der Bedarf an I/O-*Pins*. Die ebenfalls für den Datenaustausch und die Zwischenspeicherung von Datenworten vorgesehenen SRAM-Speicher (SRAM - Static Random Access Memory) sind als *Shared Memory* mit allen *Slots* verbunden. Unter Berücksichtigung der erheblichen Latenz der Schreib- und Lesezyklen der SRAM-Speicher ist über diese ein lokaler Datenaustausch zwischen benachbarten *Slots*

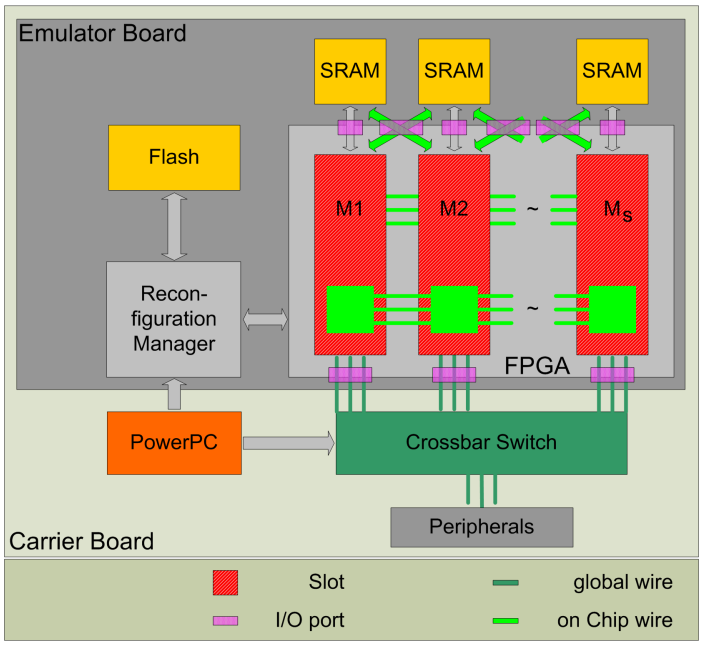


Abbildung 2.9: Erlangen Slot Machine [2]

realisierbar. Die effizientesten Verbindungen zwischen den *Slots* sind jedoch die lokalen auf dem FPGA platzierten Leitungen. Deren Anzahl ist aber FPGA-abhängig gering und im Falle großer Wortbreiten reduziert sich durch diese Begrenzung die Anzahl der möglichen *Slots.* [1]

Die Limitierung der Ein- und Ausgänge und die begrenzte Anzahl von internen Verbindungsleitungen reduzieren, abhängig vom verwendeten FPGA, signifikant die Performanz des Gesamtsystems. Das System erreicht die größte Performanz, wenn der zu testende Entwurf auf einem lokal gerichteten Datenfluss aufbaut. Für große kombinatorische Systeme wird auf Grund des erhöhten Kommunikationsbedarfs im System die Leistungsfähigkeit der ESM deutlich sinken. Die Anbindung einer Peripherie an das System über den *Crossbar Switch* lässt nur eingeschränkt ein System mit Echtzeitanforderungen zu.

Unter Ausnutzung eines gerichteten Datenflusses im Testentwurf, wie z.B. in einem Reed-Solomon-Decoder, wird im folgenden Ansatz [35] eine Optimierung für rekonfigurierbare FPGA-basierte Emulatoren erreicht. In Anlehnung an die *Slot*-Architektur der ESM wird bei diesen Systemen ein gerichteter Datenfluss durch den Emulator geführt. Dazu wird der Entwurf in separate Blöcke eingeteilt. Der Ausgang eines Blocks ist der direkte Eingang des folgenden Blocks. Ist die Berechnung in einem Block abgeschlossen und der Datenfluss in den folgenden Block übergegangen, wird der nicht mehr benötigte Bereich rekonfiguriert. Die zyklische Verarbeitung innerhalb des FPGAs führt zu einer sehr guten Ausnutzung der Emulatorplattform bei einem sehr geringen Mehraufwand für die Steuerung des Emulators. Ein Nachteil dieses Ansatzes ist die geringe Skalierbarkeit der nutzbaren FPGA-Bereiche. Wird die Plattform auf mehrere FPGAs verteilt, wirkt sich die Einschränkung der Ein- und Ausgänge analog zu Gleichung 2.1 auf die

[1] In einer mit der ESM vergleichbaren Arbeit, die der Verfasser im Rahmen der Implementierung eines rekonfigurierbaren Datenpfads auf einem vergleichbaren FPGA durchgeführt hat, war die Anzahl der Verbindungsleitungen bestimmend für die Anzahl der platzierbaren *"Slots" [14].*

Gesamtperformanz aus.

Die Einschränkung der Anzahl zur Verfügung stehender Ein- und Ausgänge eines FPGAs im Verhältnis zur Anzahl der Gatter innerhalb des FPGAs reduziert in aktuellen Multi-FPGA-Emulatoren die gesamte Effizienz. Konventionelle Lösungsansätze basieren auf dem Einsatz von Systemen mit einer sehr großen Anzahl an FPGAs (vgl. Anhang 8.3), um durch eine sehr weite Verteilung der Netzliste die *Pin*-Limitierung nicht zu überschreiten. Dadurch werden diese Systeme extrem teuer und sind nur für sehr wenige Firmen mit einer großen Anzahl an homogenen HW-Entwürfen interessant. Andere Ansätze nutzen die zur Verfügung stehenden Leitungen durch ein zeitliches Multiplex mehrfach aus bzw. setzten hoch-performante Bussysteme wie z.B. *RocketIOs* ein. Durch diese Verfahren lässt sich die Anzahl der FPGAs reduzieren, die Performanz des Gesamtsystem verringert sich aber entsprechend der Anzahl der Multiplexer bzw. der Performanz der seriellen Busverbindungen. Zusätzlich ist ein Modulator und ein De-Modulator auf jedem FPGA zu platzieren und eine sehr genaue Synchronisierung der Busstrukturen notwendig, um die Testergebnisse nicht zu verfälschen.

Eine Anwendung von Methoden der partiellen Rekonfiguration zu Laufzeit auf eine konventionelle FPGA-basierte Emulatorplattform lässt unter Verwendung bereits basierender Plattformen einen Lösungsansatz für deren Optimierung zu. Ausgehend von den dargestellten Ansätzen ist durch den Einsatz von Methoden der partiellen Rekonfiguration eines FPGAs zur Laufzeit eine signifikante Steigerung der Auslastung der einzelnen FPGAs zu erwarten. Dieser Ansatz basiert auf der Platzierung partieller Module des partitionierten Entwurfs auf dem FPGA [15]. Durch eine dem Datenfluss angepasste und auf Systemereignisse reagierende Steuerung wird eine kontinuierliche Emulation des gesamten Entwurfs auf einem FPGA erreicht. Das Ziel ist eine optimale Ausnutzung der Emulatorplattform unabhängig vom zu testenden Entwurf. In dieser Arbeit wird im Weiteren zunächst von einem homogenen Systementwurf ausgegangen. Die Einbeziehung von Schnittstellen und der Peripherie heterogener Systeme wird nur am Rand berücksichtigt.

2.3 Partiell Dynamische Rekonfiguration

Die größten Hersteller von FPGAs verfolgen seit etwa 10 Jahren das Ziel, die Konfiguration der FPGAs während der Laufzeit zu ändern. Der Einsatz der Methoden für eine Rekonfiguration zur Laufzeit (pRTR - partial RunTime Reconfiguration) hängt sehr stark von der Rekonfigurationszeit und dem zusätzlichen Bedarf an Hardware ab. Aktuelle Forschungsarbeiten beziehen sich auf die Partitionierung und die Steuerung der zu rekonfigurierenden Systeme. Diese Arbeiten haben jedoch nicht oder nur bedingt zur Integrierung der Methoden in bestehende Entwurfsabläufe und Systementwürfe geführt. Das ist zum einen mit der geringen erreichbaren Rekonfigurationszeit, zum anderen mit dem zusätzlichen Hardwareaufwand zu erklären. Ein weiterer Nachteil ist die speicherbasierte- und damit energiebedürftige Funktionsweise des Ansatzes. Der Einsatz von Methoden der partiellen dynamischen Rekonfiguration wird erst effizient eingesetzt, wenn die Anwendungsplattform besteht bzw. nicht angepasst werden muss und die Leistungsbilanz des Systems eine untergeordnete Rolle einnimmt. Diese Faktoren sind in der FPGA-basierten Hardware-Emulation gegeben.

Die Ausnutzung funktionaler Zusammenhänge eines SoCs und den damit verbundenen Abschaltphasen bestimmter Teile des Systems lässt die Möglichkeit einer zeitlichen Abfolge (*time multiplex*) der Tests dieser Funktionalitäten zu. Der folgende Abschnitt stellt eine Einführung in das Gebiet der Laufzeitrekonfiguration und der damit verbundenen Möglichkeiten für die Optimierung von Hardware-Emulatoren dar.

Grundsätzlich lässt sich die Rekonfiguration zur Laufzeit in zwei Ansätze einteilen. Ein Ansatz setzt voraus, dass jedes rekonfigurierbare Modul auf der Hardware platziert ist. Die Beschaltung und Verbindung der Module ist flexibel und über Multiplexer realisiert und lässt unterschiedliche Rekonfigurationen zu. Die Geschwindigkeit einer Rekonfiguration hängt dann direkt von der Schaltgeschwindigkeit der Multiplexer ab. Da diese Form der Laufzeitrekonfiguration zusätzliche Ressourcen der Anwendungsplattform voraussetzt, ist sie für den Ansatz dieser Arbeit nicht geeignet und wurde nicht weiter untersucht.

Eine weitere Realisierungsmöglichkeit ist das dynamische Rekonfigurieren von speicherbasierten Logikbausteinen wie z.B. FPGAs oder CPLDs (CPLD - Complex Programmable Logic Device). Diese Bauelemente speichern ihre Schaltfunktionalität (Logiktabelle) durch die Programmierung der internen Speicherzellen (LuT - Look up Table). Während der Rekonfiguration wird der Speicherinhalt der ausgewählten LuTs ausgetauscht. Nach der erneuten Zuschaltung arbeitet die rekonfigurierte Logikzelle mit der veränderten Funktionalität. Die Rekonfigurationszeit hängt in diesen Fall direkt von der Speicherzugriffzeit der Zelle ab. Diese ist zwar deutlich größer als die Schaltgeschwindigkeit der Multiplexer aus der zuvor beschriebenen Lösung, benötigt aber entgegen des ersten Ansatzes keine zusätzliche Hardware-Ressourcen der Plattform.

2.3.1 Einordnung dynamisch rekonfigurierbarer Systeme

Für die Entwicklung von integrierten Schaltkreisen und komplexen Systemen werden zwei grundsätzliche Lösungsansätze unterschieden. Zum einen können die Entwürfe als *customized* (kundenspezifisch) zum anderen als *standardized* (standardisiert) *Designs* entwickelt werden. Kundenspezifische Entwürfe werden typischerweise als ASICs gefertigt. In diesem Fall stehen den hohen Grundkosten des Entwurfs (NRI - Non Recurring Investment [6]) und der geringen Flexibilität eine sehr hohe Performance bei geringem Energiebedarf gegenüber. Standardisierte Entwürfe werden zur Kostenreduzierung über einen langen Zeitraum für einen sehr großen Kreis an Abnehmern gefertigt. Diese als ASIP (ASIP - Application Specific Instruction Set Processor) bezeichneten Mikro- /Prozessoren werden über einen Befehlssatz programmiert. Nach der Decodierung des Befehls vom Steuerwerk des Prozessors wird dieser von der integrierten ALU (ALU - Arithmetic Logical Unit) ausgeführt. Abhängig vom fokussierten Einsatzgebiet werden unterschiedliche Implementierungen von ASIPs angeboten. Bedingt durch die programmierbare Architektur sind ASIPs deutlich weniger performant als ASICs, dafür entstehen durch eine größere Anzahl an Anwendungen und damit verbundenen höheren Absatzzahlen wesentlich geringere NRI-Kosten pro Stück.

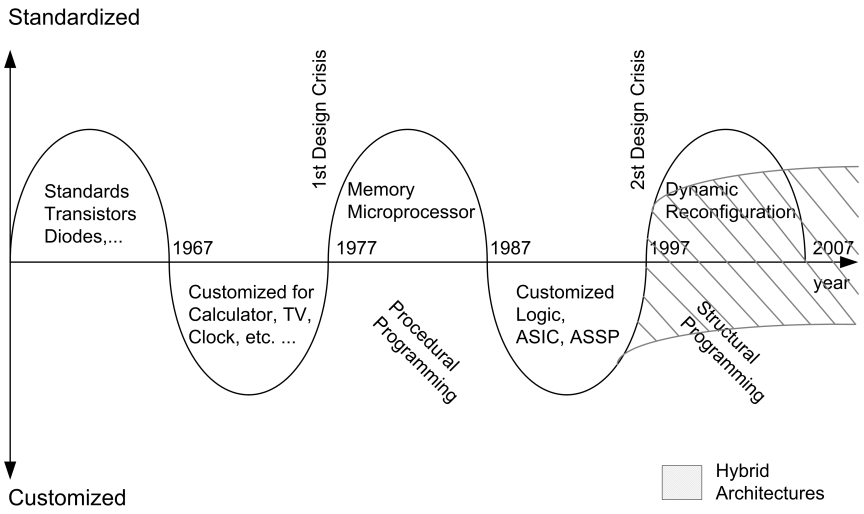


Abbildung 2.10: Makimoto Wave [27]

In Abbildung 2.10 ist der tendenzielle, technologiebedingte Wechsel von kundenspezifischen zu standardisierten Entwürfen im Wechsel von 10 Jahren zu erkennen. Dieser Wechsel setzt eine Abgrenzung zwischen beiden Entwurfsmöglichkeiten voraus. Seit Mitte der 90er Jahre werden zunehmend Architekturen angeboten, welche die Merkmale eines ASICs mit denen eines ASIPs oder eines FPGAs verbinden. In diesen Mischentwürfen werden aus Performancegründen bestimmte Hardwarekomponenten applikationsspezifisch umgesetzt und weniger laufzeitkritische Teile werden flexibel und programmierbar implementiert. Diese Systeme lassen sich grundsätzlich in SoC- und FPGA-basierte Systeme unterscheiden. Kriterium dafür ist der Anteil an programmierbarer Logik im Gesamtsystem. Die für diese Arbeit relevanten Parameter für die Bewertung der Systeme werden neben einer Vielzahl weiterer Kriterien eingeteilt in:

Kopplung: klassifiziert die Zusammenschaltung des flexiblen und des spezifischen Systemteils. Es wird zwischen loser Kopplung auf Systemebene und enger Kopplung auf Gatterebene unterschieden. In [45] wird das Thema hinsichtlich der Entwurfsunterstützung mit Hilfe von Hardware/Software-CoDesign Verfahren betrachtet. Die Kopplung der Systemteile hat wesentlichen Einfluss auf die erreichbare Gesamtperformanz.

Granularität: gibt an, inwieweit die Rekonfiguration in das System eingreift. In grob granularen Systemen ist es möglich, sehr große, zusammenhängende Blöcke (vgl. [22]) zu rekonfigurieren. Von fein granular wird ausgegangen, wenn die Rekonfigurationen sehr tief in die applikationsspezifische Hardware eingreifen. Die Blockgröße steht dabei in direktem Zusammenhang zur Rekonfigurationszeit [24].

Konfigurationszeit (T_K): gibt die notwendige Zeitspanne zum Konfigurieren eines ausgewählten Systembereichs an. Man unterscheidet zusätzlich zwischen Systemen, in welchen nur die gesamte konfigurierbare Fläche austauschbar ist und Systemen, die partiell rekonfigurierbar sind [70]. Die gesamte Rekonfigurationszeit setzt sich aus systemabhängigen Einzelgrößen zusammen. Für die Bestimmung wird zunächst das System oder zu konfigurierende Systembereich deaktiviert. Im Anschluss findet eine *Setup*-phase statt, in welcher die Konfigurationsdaten geladen (*upload*) bzw. die bereits konfigurierten Daten gespeichert werden (*download*). Nach der Rekonfiguration wird der konfigurierte Bereich erneut zugeschaltet. Abhän-

gig von der Architektur bzw. der Funktionalität des rekonfigurierten Bereichs und der damit verbundenen Anzahl an Registerstufen in diesem Bereich, benötigt das System entsprechend viele Takte bis gültige Datenwörter geschrieben werden. Eine ausführliche Beschreibung für die Zusammensetzung der Rekonfigurationszeit ist in [65] gegeben. Die Konfigurations- bzw. Rekonfigurationszeit ist eine wesentliche Größe für die Bestimmung der Effizienz eines pRTR-Systems.

Speicherverwaltung (Rekonfigurationsspeicher): beeinflusst in SoC-basierten Systemen die Systemgröße und die erreichbare Rekonfigurationszeit. Es existieren Realisierungen (vgl. Abs. 2.3.2), in denen die Speicherverwaltung in das System integriert ist. In diesem Fall sind sehr geringe Rekonfigurationszeiten erreichbar. Nachteilig ist die dadurch zunehmende Systemgröße. Externe Speicherverwaltungen reduzieren diesen Nachteil, erhöhen aber die Zykluszeit einer Rekonfiguration durch die längeren Speicherzugriffszeiten. In FPGA-Systemen, auf welche in Abschnitt 2.3.3 eingegangen wird, sind die Speicher für die Konfigurationsdaten ausnahmslos extern platziert.

Die folgenden beiden Absätze stellen Systeme aus dem SoC-Bereich und Realisierungen auf Basis von FPGAs gegenüber. Die rekonfigurierbaren Systeme wurden hinsichtlich der Anwendung von Methoden der partiell dynamischen Rekonfiguration unter den genannten Parametern für einen Hardware-Emulator untersucht und bewertet. Die Differenzierung zwischen einem SoC und einem FPGA ist nicht eindeutig festgelegt. Dieses Problem wird am Beispiel des FPSLIC deutlich. Die Einordnung der Systeme in dieser Arbeit ist aus diesem Grund primär unter dem Gesichtspunkt des Anteils an programmierbarer Logik und dem von dedizierten Komponenten erfolgt.

2.3.2 SoC-Architekturen

Im Rahmen von Forschungsarbeiten sind in den letzten Jahren eine Vielzahl unterschiedlicher Realisierungen für rekonfigurierbare SoCs entstanden. Im Mittelpunkt der Untersuchungen stand die Abschätzung für den Grad der Granularität in Verbindung mit einer optimalen Kopplung zwischen dem flexiblen und dem spezifischen Systemteil. Es wurden vier Systeme in die Untersuchung und Bewertung einbezogen:

Garp [54] (University of Berkeley): ist eine Kombination aus einem Standard Mikroprozessor und einem integrierten FPGA. Die Rekonfiguration erfolgt grob granular. Der im System integrierte FPGA ist spaltenweise partiell rekonfigurierbar. Die Rekonfigurationszeit ist abhängig vom externen Rekonfigurationsspeicher und beträgt maximal fünf Speicherzugriffszyklen. Der Mikroprozessor wird direkt über einen internen Befehlsspeicher programmiert. Es wurde eine direkte Kopplung zwischen dem FPGA und dem Mikroprozessor umgesetzt. Zusätzlich ist ein Datenaustausch über einen gemeinsamen internen Registerspeicher (*Cache*) vorgesehen.

Chimaera [120] [119] (University of Washington): ist ein fein granularer rekonfigurierbarer *Scalar*-Prozessor. Die rekonfigurierbare Funktionseinheit ist eng gekoppelt und direkt in den Datenfluss der ALU integriert. Die ALU besteht nicht aus speicherprogrammierter Logik, sondern aus programmierbarer, kombinatorischer Logik. Die Programmierung ist in diesem Fall durch die Beschaltung der Logikzellen über Multiplexer realisiert. Der Zugriff auf den Rekonfigurationsspeicher erfolgt intern über die oberste (*level one*) *Cache*-Ebene. Dadurch wird eine sehr geringe Rekonfigurationszeit erreicht. Nachteil dieser Realisierung ist der durch den *Cache* bedingte große Flächenbedarf des Rekonfigurationsspeichers im System.

Triscend A7S [101] [100] wurde 2004 von *Xilinx* übernommen. Der A7S20 Schaltkreis ist ein auf Systemebene lose gekoppeltes *System-on-Chip,* welches kommerziell vertrieben wird. Die Rekonfiguration erfolgt über eine direkte Speicherschnittstelle aus einem externen Speicher. Ein 16 Kbyte großer interner Zwischenspeicher (*Scratch pad*), auf welchen die CPU (CPU - Central Processing Unit) und der konfigu-

rierbare Systemteil Zugriff haben, realisiert den Datenaustausch und reduziert während der Rekonfiguration die Latenz eines Speicherzugriffs.

XiRisc [26] (University of Bologna): ist ein für Signalverarbeitung (*signal processing*) optimierter *IP-Core* zur Integrierung in ein komplexeres SoC. Die Rekonfigurierbarkeit ist fein granular in einen VLIW-Prozessor (VLIW - Very Long Instruction Word) integriert. Es besteht eine enge Kopplung zwischen dem Prozessor und dem konfigurierbaren Datenpfad (*data path*). Die Steuerinformationen für eine Konfiguration des Datenpfads werden durch den Compiler erzeugt und zum Systemstart in den Programmspeicher geladen. Eine Rekonfiguration zur Laufzeit ist nicht möglich.

System	**Garp**	**Chimaera**	**Triscend (AS7)**	**XiRisc**
Granularität	grob	fein	fein	fein
Kopplung	direkt	eng	lose	eng
Rekonfigurations-bandbreite	Bit-parallel	1 cycle (gesamter Cache)	Byte-parallel	Bit-parallel
Speicher-verwaltung	extern über 4 Busse angebunden, intern Cache für config. cover von Spalten	4 Layer Cache (config.)	8 KByte Cache (data/config.)	extern
Rekonfigurations-zeit R_T	50us/32 Spalten	10ns/context-Speicher	7.4ms (40Mhz)	/

Tabelle 2.3: Rekonfigurierbare System-on-Chips

In Tabelle 2.3 sind die vier untersuchten rekonfigurierbaren SoCs zusammengefasst. Die unterschiedliche Methodik für Rekonfiguration und Speicherverwaltung wirkt sich direkt auf die Rekonfigurationszeiten aus. Während für das *XiRisc*-System keine Rekonfiguration zur Laufzeit vorgesehen ist, lassen sich die Systeme *Garp*, *Chimaera* und *Triscend* dynamisch rekonfigurieren. Die Effizienz der unterschiedlichen Ansätze hängt vom Anwendungsgebiet ab. Die *XiRisc*-Architektur ist auf Grund der compilerunterstützten Generierung der Konfigurationsdaten sehr leistungseffizient bei einer durchschnittlichen Flexibilität durch die grobe Granularität. Die *Triscend*-Produkte sind sehr flexibel, aber durch die Konfigurationsarchitektur auf einen externen Speicher angewiesen. Das *Garp*-System und der *Chimaera*-Prozessor stellen sich als gleichwertig dar. Der *Chimaera* hat ein flexibleres Konfigurationsmanagement und verwaltet mehrere Speicherbereiche (*context memory*). Das *Garp*-System hat auf Grund der direkten Kopplung die effizienteste Hardwarearchitektur, ist aber durch die relativ grobe Granularität (rekonfigurierbare Spalten) weniger flexibel.

Zusammenfassend ist festzustellen, dass SoC-basierte rekonfigurierbare Architekturen Vorteile vor allem in vor der Laufzeit bestimmbaren Anwendungsbedingungen wie z.B. der Signalverarbeitung oder der Steuerungstechnik haben. Für Anwendungen wie der Hardware-Emulation sind SoCs aber nur eingeschränkt geeignet, da diese heterogenen Systeme einen zu geringen Anteil an programmierbarer Logik im

Verhältnis zur genutzten *Chip*-Fläche haben.

2.3.3 FPGA-Architekturen

Neben den in Abschnitt 2.3.2 beschrieben Lösungen für die Integration von rekonfigurierbaren Bereichen in ein *System-on-Chip*, werden zur Steigerung der Performance parallel Ansätze verfolgt, rechenintensive Systemteile wie z.B. der ALU-, oder DSP-Funktionalität direkt in programmierbare Logikbaugruppen einzubinden. Die Ansätze lassen sich für speicherbasierte FPGAs in der Größe und dem Funktionsumfang der integrierten Baugruppen unterscheiden [108].

Die Systeme des größten FPGA-Herstellers *Xilinx* werden durch die Art und den Umfang der applikationsspezifischen Logikteile differenziert. *Xilinx* bietet FPGAs mit integrierten Mikroprozessoren wie z.B. einem von IBM entwickelten *PowerPC*[112] an. Diese, seit der *VirtexII-Pro*-Serie [114] implementierten ASIPs, sind als integrierter Prozessorkern (*Core*) direkt in das FPGA eingebunden. Dadurch wird eine heterogene Struktur mit sehr hoher Performance erreicht. Ein Nachteil ist die unterbrochene und verkleinerte Fläche für die Platzierung der programmierbaren Logik. Aus diesem Grund werden optimierte Prozessoren, wie z.B. ein *Micro Blaze*-Prozessor [62], als Netzliste für die Synthese auf ein FPGA angeboten. Diese Prozessoren erreichen nicht die Performance eines *PowerPC*, sind aber frei im FPGA platzierbar. Ab der *Virtex4-SX*-Serie werden die in der *FX*-Serie eingebundenen *PowerPC* durch integrierte *ExtremeDSP-Slices* ergänzt. Die *DSP-Slices* sind für Anwendungen der digitalen Signalverarbeitung optimiert und ihre Nutzung hat durch die feste Platzierung im FPGA direkten Einfluss auf die Platzierung des Entwurfs auf dem FPGA.

Alle FPGAs der Firma *Xilinx* sind seit der *Virtex-I*-Serie partiell dynamisch rekonfigurierbar. Die FPGAs der verschiedenen Serien unterscheiden sich grundsätzlich in der Form und dem Umfang der rekonfigurierbaren Fläche. Die *Virtex-I*- und *-II*-Serie muss in vollständigen Spalten rekonfiguriert werden (vgl. Abb. 3.2). Für ein *Virtex-I*-FPGA werden zusätzlich vier Spalten zu einem rekonfigurierbaren Modul zusammengefasst. Ab der *Virtex4*-Serie [113] ist die Beschränkung auf Spalten nicht mehr vorhanden. Aus diesem Grund sind die FPGAs der *Virtex4* und die der *Virtex5*-Serie mit wesentlich geringeren Einschränkungen hinsichtlich Form und Größe der Module rekonfigurierbar.

Die Rekonfiguration der FPGAs kann über unterschiedliche Schnittstellen erfolgen. Von *Xilinx* wird die ICAP-Schnittstelle [115] (ICAP - Internal Configuration Access Port) und die *Select-Map*-Schnittstelle [77] favorisiert. Während der *Select-Map*-Zugang extern erfolgt, wird die ICAP-Schnittstelle als Modul in den Entwurf integriert. Über dieses Modul werden während des Betriebs die neuen Konfigurationsdaten direkt an die Konfigurationsadressen der programmierbaren Speicher (*LuT*) geschrieben.

Ein weiterer Anbieter programmierbarer Logikbausteine mit integriertem Prozessor ist *Atmel*. Mit dem *AT94K FPSLIC* [31] wurde eine Kombination aus einem FPGA und einem integrierten SoC [7] realisiert. Beide Systemteile sind hinsichtlich des Einflusses auf das Gesamtsystem gleich gewichtet. Aus diesem Grund ordnet sich das System in das Gebiet der FPGAs ein, da *Atmel* das Anwendungsfeld hauptsächlich im Bereich der programmierbaren Logik sieht [31]. Das FPGA in einem *FPSLIC* lässt sich partiell zur Laufzeit rekonfigurieren [8]. Die Konfigurationen werden vor Inbetriebnahme des Systems generiert und im Speicher des AVR-*Controller* abgelegt. Zur Laufzeit schreibt der AVR-*Controller* die Konfigurationsdaten in ein Konfigurationsregister und programmiert das integrierte FPGA [81].

Altera ist ein dritter Anbieter für FPGAs. Die Produktpalette beinhaltet im Gegensatz zu *Xilinx* keine FPGAs mit integrierten Prozessoren, welche in dedizierter Form in das System eingebunden sind. *Altera* verfolgt für das Einbinden eines Prozessors einen mit dem *MicroBlaze*-Prozessor vergleichbaren Ansatz.

Dafür wird ein *NIOS-II*-Prozessor [4] direkt in den Entwurf eingebunden und frei auf dem FPGA platziert. *Altera* unterstützt nicht die dynamische Rekonfiguration von FPGAs der *Excalibur*-Serie [3].

Partiell dynamisch rekonfigurierbare Systeme werden auf Grund der FPGA-Architekturen in diesem Bereich intensiver eingesetzt als im SoC-Bereich. *Xilinx*, *Atmel* und *Altera* als die größten Anbieter speicherprogrammierbarer Logik unterstützen Verfahren und Methoden für die Rekonfiguration eines Systemteils zur Laufzeit durch angepasste Entwurfsabläufe (*Design Flows*). Tabelle 2.5 stellt die FPGAs der verschiedenen Hersteller gegenüber. Im Gegensatz zu den SoC-Systemen lassen sich die FPGAs neben Granularität und Rekonfigurationszeit in Rekonfigurationsbandbreite, erreichbare Taktrate des FPGAs, der Möglichkeit zur partiellen Rekonfiguration und der *Frame*-Größe einer Rekonfiguration unterscheiden. [2]

System	**VirtexII (Xilinx)** XC2VP7	**Virtex4 (Xilinx)** XC4VLX60	**FPSLIC (Atmel)** AT94k	**Excalibur (Altera)**
Granularität	eine Spalte	1 Frame (41*32bit)	Bit-Level	grob
Einschränkungen	durchgehende Spalte	keine	keine	gesamtes FPGA
partiell	ja	ja	ja	nein
R_T	Interface-abhängig	Interface-abhängig	$800ms$@1Mhz clk	Interface-abhängig
Rekonfigurations-bandbreite	$\frac{66MByte}{s}$ (Select Map)	$\frac{380MByte}{s}$ (ICAP)	Byte-parallel	Bit-parallel
Frame-Größe	1320 Frames	13868 Frames	< 36 KByte	FPGA-abhängig
max Taktfrequenz	400 Mhz	500 Mhz	25 Mhz (AVR)	SoC-abhängig

Tabelle 2.5: Rekonfigurierbare FPGA

Zusammenfassend erreichen FPGAs gegenüber den SoCs die deutlich größere Performanz. Die verfügbaren rekonfigurierbaren SoCs sind hauptsächlich für mobile Applikationen und für Signalverarbeitungssysteme entwickelt worden. Durch die Leistungsaufnahme von FPGAs sind diese nicht für den mobilen Einsatz geeignet. FPGAs werden auf Grund des hohen Grades an Parallelität und der Wiederverwendbarkeit hauptsächlich in Systemen mit großen Anforderungen hinsichtlich der Bandbreite in der Signalverarbeitung und in Prototypen-Systeme (vgl. Abs. 2.2) integriert.

Für die Anwendung von Methoden der partiellen dynamischen Rekonfiguration eignen sich FPGAs weitaus besser als die beschrieben SoCs. Die gegenübergestellten FPGAs (vgl. Tab. 2.5) lassen eine Rekonfiguration zur Laufzeit des Gesamtsystems zu. Dabei wird systemabhängig das vollständige FPGAs oder ein Teil des FPGA rekonfiguriert.

Die Hersteller unterstützen den Entwurf rekonfigurierbarer Systeme mit speziellen Entwurfsabläufen. Für die Testimplementierungen in der vorgelegten Arbeit wurden ausschließlich *Xilinx*-FPGAs verwendet. Der Grund sind die Erfahrungen der Arbeitsgruppe mit diesen Baugruppen und die bessere Unterstützung

[2] Für die SoC-Systeme sind die erweiterten Angaben der Klassifizierung von FPGAs vom Mikroprozessor und der Applikation abhängig und wurden nicht berücksichtigt.

des Herstellers für pRTR-Entwürfe. Der folgende Abschnitt geht aus diesem Grund auf die Abläufe für die Synthese eines rekonfigurierbaren Entwurfs für FPGAs der Firma *Xilinx* ein und stellt die verwendeten Werkzeuge vor.

2.3.4 Werkzeuge und Design-Flow

Die Synthese eines rekonfigurierbaren Systems unterscheidet sich in wesentlichen Punkten von der Synthese für ein statisch platziertes System. Zunächst werden die allgemeinen Entwurfsabläufe für einen Syntheseprozess auf unterschiedlichen Abstraktionsebenen beschrieben. Dabei wird auf die wichtigsten Unterschiede zwischen der Synthese für einen FPGA und der eines ASICs eingegangen. Am Entwurfsablauf der Firma *Xilinx* (XMDF - Xilinx Modular Design Flow) werden im zweiten Teil des Abschnitts die Erweiterungen für partiell rekonfigurierbare Entwürfe dargestellt.

Der Prozess der Abbildung eines Entwurfs auf eine Zielarchitektur [39] (z.B. FPGA oder ASIC) wird in mehrere Einzelschritte unterteilt und als Synthese (vgl. Abb. 2.11) bezeichnet:

- Ausgehend von einer Verhaltensbeschreibung (*Design Entry*) in einer Hardwarebeschreibungssprache (HDL - Hardware Description Language) wird der Entwurf im ersten Schritt *elaboriert* und in die Register-Transfer-Ebene (RTL - Register Transfer Level) abgebildet. Diesen Schritt, in welchem von den Synthesewerkzeugen ein erster Optimierungsschritt durchgeführt wird, bezeichnet man als Algorithmensynthese.

- Ein *Compiler* analysiert im Anschluss den Entwurf und übersetzt ihn in ein für einen Simulator ausführbares Format. Dadurch ist eine Verifikation der Entwurfsteile mit einem HDL-Simulator (z.B. Questa) durchführbar. Im Falle von auftretenden Fehlern oder Abweichungen von der Spezifikation können notwendige Änderungen direkt in der Verhaltensbeschreibung berücksichtigt werden. Mit einer Simulation auf RT-Ebene wird das Verhalten taktgenau abgebildet, d.h., Laufzeitabweichungen, die durch Verzögerungen von Logikelementen entstehen, werden nicht berücksichtigt.

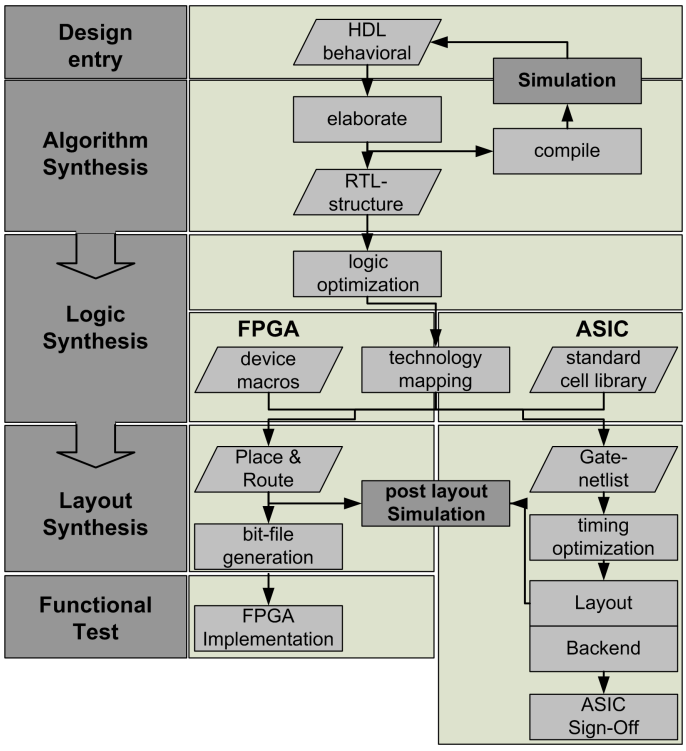


Abbildung 2.11: Syntheseprozess

- Der folgende Syntheseschritt optimiert die Schaltung auf Logikebene. Dabei wird vom Synthesewerkzeug eine Vereinfachung der Schaltung berechnet und die der Schaltung zu Grunde liegenden Gleichungen zusammengefasst oder algorithmisch reduziert. In der Logikoptimierung steht die Reduzierung der *boolschen* Gleichungen, welche die Schaltung repräsentieren, im Mittelpunkt. Im diesem Optimierungsprozess wird durch Faktorisierung die günstigste Balance für die eingesetzten Schaltnetze gemäß den Randbedingungen, welche durch den Entwurf vorgegeben sind, für eine Optimierung angestrebt.

- Nach der Logikoptimierung trennt sich der Entwurfsablauf für einen FPGA von dem eines ASICs ab. Die generische Netzliste, welche durch die Optimierung entstanden ist, wird für die FPGA-Synthese direkt durch die Abbildung der Makroblöcke auf die entsprechende Zielarchitektur durchgeführt. Die Makroblöcke repräsentieren die Abbildung der Hardwarestruktur des FPGAs für die komplementären Hardwarekonstukte des Entwurfs.

- Für den FPGA wird während der Layout-Synthese die Platzierung der Schaltung auf der Zielarchitektur durchgeführt. Nach Abschluss dieses Schrittes wird eine Konfigurationsdatei (*bit-file*) generiert. Diese enthält die Beschaltung der FPGA-internen CLBs (CLB - Configurable Logic Block). Nach der Generierung der Konfigurationsdatei ist eine funktionale Überprüfung der Schaltung unter Berücksichtigung der Randbedingungen des FPGAs möglich.

- Für die ASIC-Synthese wird während des *Technology Mapping* eine Zellbibliothek (*standard cell*

library) in den Syntheseschritt einbezogen. Diese Bibliothek enthält die in der Technologie vorhandenen Gatter der Zielarchitektur. Nach der Generierung der Gatternetzliste des ASICs werden auf diese Netzliste Optimierungsverfahren hinsichtlich des *Timing* der Schaltung (z.B. *re-timing*) angewendet.

- Für die Fertigstellung des ASICs ist im abschließenden Schritt ein *Layout*- und ein *Backend*-Prozess notwendig. Nach dem *Layout*-prozess des ASICs wird in einer *Post-layout*-Simulation eine Verifikation der Schaltung unter Berücksichtigung der Laufzeiten und physikalischen Eigenschaften der Gatterelemente, wie z.B. der Leitungskapazitäten, durchgeführt.

Der Syntheseprozess für ein FPGA und der eines ASIC unterscheiden sich nach dem *Technology Mapping* wesentlich voneinander. Die Bibliothek einer FPGA-Technologie enthält für die Reduzierung der Schaltungsfläche unter Berücksichtigung von Standardanwendungen zunehmend optimierte Hardwarekomponenten, welche direkt in das FPGA integriert sind. Diese Zellen werden nicht auf ein CLB abgebildet, sondern direkt in der *Place&Route*-Phase and den entsprechenden Stellen platziert. Das führt zu wesentlich besseren Laufzeitergebnissen und geringerem Flächenverbrauch, schränkt aber das Potential der Schaltungsoptimierung ein. Physikalische Laufzeiteffekte oder Probleme mit zu geringen Treiberleistungen der Gatter treten in einem FPGA nicht auf. Aus diesem Grund gestaltet sich der Syntheseschritte *Place&Route* für ein FPGA deutlich weniger aufwändig als für einen ASIC-Entwurf.

Die Synthese eines partiell dynamisch rekonfigurierbaren Systems für ein FPGA weicht durch die Möglichkeit der zeitlich mehrfachen Ausnutzung eines Teils des FPGAs von einer statischen Synthese ab. Der im Folgenden beschriebene Ablauf bezieht sich auf *Xilinx*-FPGAs und enthält die Schritte:

- Zunächst teilt der Systementwickler den Entwurf in einen statischen und in einen oder mehrere dynamische Teile ein.
- Im folgenden Schritt (Abb. *2.12 initial budgeting*) generiert ein Synthesewerkzeug für das *Top-level*-Modul des Entwurfs die Netzliste. Dabei werden die physikalischen Randbedingungen der Zielplattform (z.B. Anschlussbelegung) berücksichtigt. Des Weiteren werden die Takt- und die Leitungen für die Spannungsversorgung verbunden.
- Danach folgt im *Place&Route*-Prozess des *Top-level*-Moduls eine Festlegung der statischen und der dynamischen Bereiche. Diese Bereiche werden mit dedizierten Schnittstellen, welche FPGA abhängig sind, verbunden.
- Im Anschluss an die Generierung der *Top-level*-Netzliste werden die einzelnen statischen und dynamischen Module im zweiten Prozess (*active module implementation*) synthetisiert. Diese Module erfüllen einzeln nicht die Anforderungen, welche an einen vollständig synthetisierbaren Entwurf gestellt werden, da sie nicht über die physikalischen Verbindungen an die Zielarchitektur verfügen. Aus diesem Grund wird für die Synthese der dynamischen Module die Netzliste des *Top-level*-Moduls in den Syntheseprozess mit einbezogen.
- Der folgende Schritt der Platzierung wird konträr dazu für jedes Modul separat durchgeführt.
- Abschließend wird aus dem *Top-level*-Modul, dem statischen und einem dynamischen Modul eine ausführbare Konfiguration (*initial Configuration*) für das FPGA generiert. Dieser Prozess wird als *final assemble* bezeichnet.

Der verwendete Entwurfsablauf basiert aus Performance- und Kompatibilitätsgründen auf der Synthesesoftware *Precision* von *MentorGraphics*. Die Funktionalität des *Xilinx-Modular-Design-Flow* lässt die Verwendung eines zur ISE-Software von *Xilinx* abweichenden Synthesewerkzeugs zu.

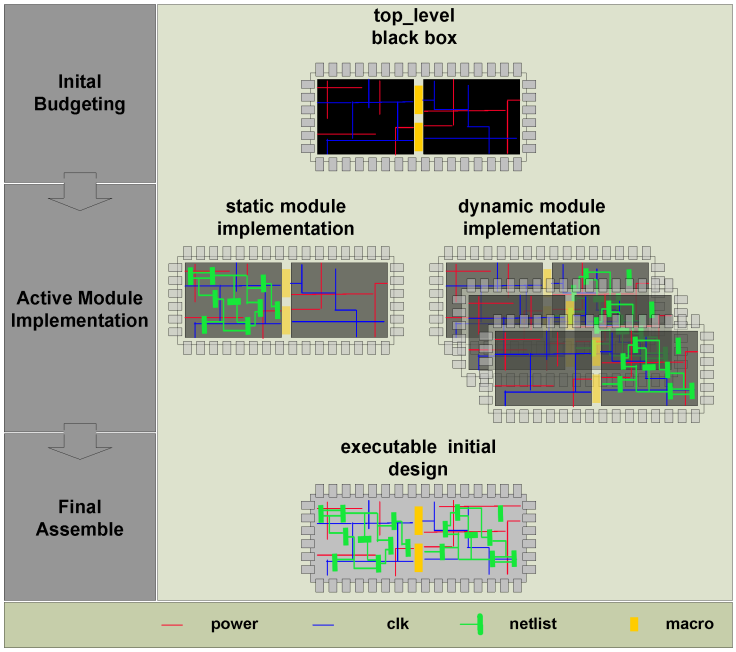


Abbildung 2.12: Xilinx Modular Design Flow

2.4 Relevante Methoden

Der in Absatz 3 beschriebene Ansatz für die Optimierung von Hardware-Emulatoren basiert im Wesentlichen auf der Partitionierung des Entwurfs und der funktional korrekten Steuerung der partiellen Systemteile. Inhalt der folgenden Absätze ist es, die grundlegenden Verfahren für die Partitionierung und die Ablaufsteuerung des neuen Ansatzes einer partiellen System-Emulation darzustellen und hinsichtlich des Implementierungsaufwands zu bewerten.

2.4.1 Methoden der strukturellen Entwurfspartitionierung

Die strukturalen Partitionierungsverfahren finden vorwiegend Anwendung in der Aufteilung einer Netzliste in Multi-FPGA Systemen und basieren auf Methoden der Graphentheorie, welche in [25] formalisiert ist.

Definition (Partition) [25, 78]: Eine Partition P des Graph $G = (V, E)$ mit der Knotenmenge V und der Kantenmenge E zwischen den Knoten wird partitioniert das gilt:

$$\sum_{\nu_i \in P_k} \leq A_k \mid A_k \text{ is limit on the area of } P_k \tag{2.2}$$

Für die weitere Anwendung in dieser Arbeit entsprechen nach der Synthese die Gatter einer Netzliste den Knoten und die Verbindungen zwischen den Gattern den Kanten des Graphen. Die Anwendung der Graphentheorie ist auf Grund des homogenen Charakters der synthetisierten Netzliste möglich, d.h., alle hierarchischen Ebenen und funktionalen Zusammenhänge des Entwurfs werden durch die Optimierung während der Synthese weitgehend aufgelöst. Eine Zielfunktion gibt die Randbedingungen für die Partitionierung vor (z.B. Anzahl der Verbindungen zwischen den FPGAs). Diese Randbedingungen führen zu einem Lösbarkeitsproblem [40] (NP - non-deterministic polynomial-time hard) d.h., die Menge der Lösungen ist mit Zunahme der Gatterzahl in keiner bestimmbaren Zeit vollständig berechenbar. Das Problem wird mit der Einführung dedizierter Heuristiken eingeschränkt. Es werden drei Klassen von Heuristiken [78] für die Partitionierung einer Netzliste unterschieden [45, 44]: *Group-Migration*, *metrische Allokationsmethoden*, *Simulated Annealing*.

Group-Migration: Methoden der *Group-Migration* gehen von einer Anfangspartition bestehend aus einer Menge von Knoten aus. Zu den bekanntesten Verfahren dieser Klasse wird die Bi-Partitionierung von Kernighan und Lin gezählt [63]. Während jedes Optimierungsschritts wird ein Knotenpaar im gegenseitigen Tausch in die jeweils gegenüberliegende Zielpartition übernommen. Dadurch bleibt die Anzahl an Knoten in beiden Partitionen gleich. Durch den Tausch ist eine Veränderung der Anzahl von Verbindungen zwischen den Partitionen möglich. Ist das Optimum für diese Partitionierung erreicht, werden die entstandenen Bi-Partitionen weiter aufgeteilt. Diese Partitionen und die einzelnen Iterationen müssen voneinander unabhängig betrachtet werden, um das Lösbarkeitsproblem zu verhindern. Nach der Bi-Partitionierung enthalten alle Teilpartitionen die gleiche Anzahl an Knoten. Der Algorithmus muss so oft auf die Teilpartitionen angewendet werden, bis die vorgegebene Anzahl an Verbindungen zwischen den Partitionen eingehalten wird. Das führt in der praktischen Anwendung auf Grund der Leitungsbeschränkungen zu einer ungünstigen Auslastung aktueller Multi-FPGA Systeme [78].

Im Ansatz von Fiduccia und Mattheyses [37] wird diese Einschränkung durch die Einführung eines Gewichtungsfaktors, welcher das Größenverhältnis zwischen den Partitionen bestimmt, reduziert. Die Knotenpaare werden nicht ausgetauscht sondern einzelne Knoten werden in die diametrale Partition verschoben. Diese Partition wird dadurch größer (gewichtiger). Der Gewichtungsfaktor begrenzt die Differenz zwischen der größten und der kleinsten Partition. Durch die größere Flexibilität ist diese Heuristik für die praktische Anwendung wesentlich besser geeignet als der Ansatz von Kernighan und Lin.

Eine Ergänzung der gewichteten Bi-Partitionierung ist das Zellreplikationsverfahren [61]. In der Zellreplikation wird durch das Einfügen einer zusätzlichen Zelle oder eines Gatters der Schnitt zwischen zwei Partitionen verkleinert.

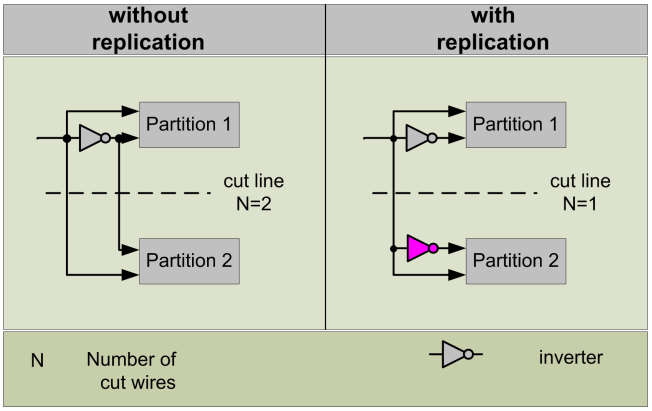


Abbildung 2.13: Zellreplikation

In Abbildung 2.13 ist das Verfahren der Zellreplikation dargestellt. Der Inverter wird durch Replikation in beiden Partitionen platziert. Dadurch entfällt die Ausgangsleitung, welche den Inverter mit den Partitionen verbindet. Der Schnitt reduziert sich auf eine Leitung. Dieses Verfahren vergrößert auf Grund der zusätzlichen Platzierung von Zellen die Partitionen und ist nur für die Optimierung einer abgeschlossenen Bi- Partitionierung sinnvoll.

Unter Berücksichtigung der Partitionen wird in der rekursiven asymmetrischen Bi- Partitionierung von *Kuznar, Brglez und Koziminski* [66] der Entwurf des Prototyping Systems an das Partitionierungsergebnis angepasst. Die Partitionierung erfolgt derart, dass jede Teilpartition der Netzliste in einen aus einer Bibliothek gegebenen FPGA passt [78]. Ein Vorteil dieses Verfahrens ist die exakte Dimensionierung der Plattform durch die angepasste Aufteilung des Entwurfs. Für eine kostengünstige Implementierung setzt dieser Ansatz jedoch eine sehr flexible Plattform voraus.

Die beschriebenen Bi-Partitionierungsverfahren berücksichtigen keinen Datenfluss oder strukturelle Zusammenhänge der Schaltungen. Des Weiteren wird der kritische Pfad während der Partitionierung verändert. Diese Nachteile führen zu einem suboptimalen Laufzeitverhalten innerhalb der Teilpartitionen.

Ein Ansatz für eine Partitionierung der Netzliste unter Berücksichtigung der hierarchischen Struktur des Entwurfs wurde von *Brasen, Hiol, Saucier* mit dem *Cone*-Algorithmus realisiert [86]. Ausgehend von einem Ausgang der Schaltung werden alle kombinatorischen Knoten, die zu diesem Ausgang führen, von einem Algorithmus in einem Kegel (*Cone*) zusammengefasst. Ein kombinatorischer Knoten ist eine aus Gattern bestehende Struktur, welche keine Register enthält. Diese Struktur propagiert eine Zustandsänderung in einem Taktzyklus. Die Latenz der Struktur ergibt sich aus der Summe der Laufzeiten der einbezogenen Gatter.

In Abbildung 2.14 ist die Knotenstruktur eines Entwurfs exemplarisch abgebildet. Ausgehend von einem Eingang folgt ein Kegel den weiteren Knoten bis zu einem Eingang. Die Kegel können sich in der Struktur überlappen, wenn ein Eingang Quelle mehrerer Kegel ist. Die Partitionierung der Kegel erfolgt durch Zuordnung der Knoten zu einzelnen *Clustern*.

Für die Partitionierung wird zunächst eine Liste aller Eingangsknoten gebildet. Diese Liste wird als *CRoot* bezeichnet und ordnet jedem primären Ausgang und jedem sequentiellen Element den Eingangsknoten zu. Die Anzahl der Knoten in *CRoot* entspricht der möglichen Anzahl an Kegeln in der Netzliste. Von jedem Knoten aus der Liste *CRoot*, werden alle Pfade über alle Knoten bzw. Gatter zurückverfolgt, bis

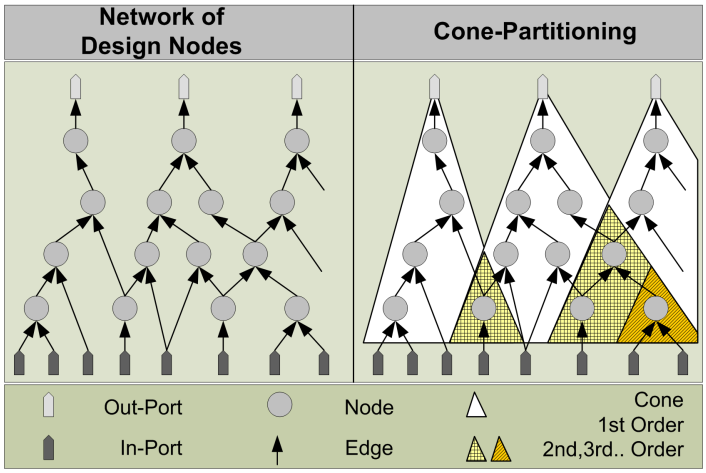


Abbildung 2.14: Cone-Algorithmus [86]

ein Eingang oder ein sequentielles Element (z.B. *Latch*, *Flip-Flop*) erkannt wird. Jeder selektierte Knoten wird mit der Nummer des Wurzelknotens aus *CRoot* markiert. Ist ein Knoten in mehreren Markierungslisten enthalten, wird er in die *CRoot*-Liste aufgenommen und als Wurzelknoten weitergeführt. Die überlappenden Kegel werden zu *Clustern* zusammengefasst. Existieren genauso viele Kegel wie *Cluster*, enthält der Entwurf keine überlappenden Kegel und lässt sich optimal partitionieren. Die sequentiellen Elemente werden den *Clustern* zugeordnet, um die ursprüngliche Schaltfunktion abzubilden. Wenn der Knoten des sequentiellen Elements in verschiedenen *Clustern* liegt, wird das Element dem *Cluster* zugeordnet, welches dadurch den größten Kostengewinn erfährt.

Durch die *Cone*-Partitionierung ist unter Berücksichtigung der Schaltfunktion eine variable Zusammenfassung der Knoten zu *Clustern* gegeben. Dadurch nähert sich das Partitionierungergebnis an die Randbedingungen der Emulatorplattform an.

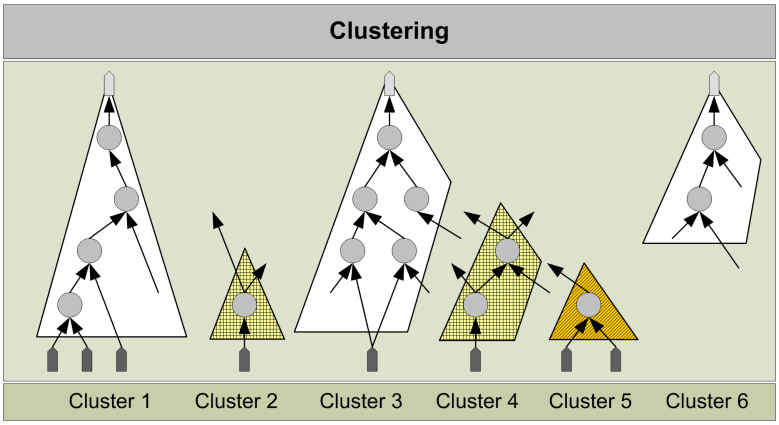


Abbildung 2.15: Cluster-Bildung für Cone-Partitionierung [86]

Unter Berücksichtigung des Flusses eines Netzwerks wurde von Yang, Wong eine balancierte Bi- Parti-

tionierungsheuristik vorgestellt [118]. Diese Heuristik basiert auf der *Maxflow-Mincut* Technik [30]. Den Kanten des Netzwerks wird dazu eine Kapazität zugewiesen und dem Datenfluss über diese Kanten eine Richtung. Durch die Bewertbarkeit des Flusses ist die Partitionierung unter Berücksichtigung des theoretisch möglichen Datendurchsatzes (*MaxFlow*) zwischen zwei Knoten möglich und dadurch lässt sich der kleinste Schnitt (*MinCut*) im Netzwerk berechnen. Der Schnitt repräsentiert im realen Multi-FPGA-System die Anzahl der notwendigen Leitungen bzw. die Bandbreite zwischen den FPGAs.

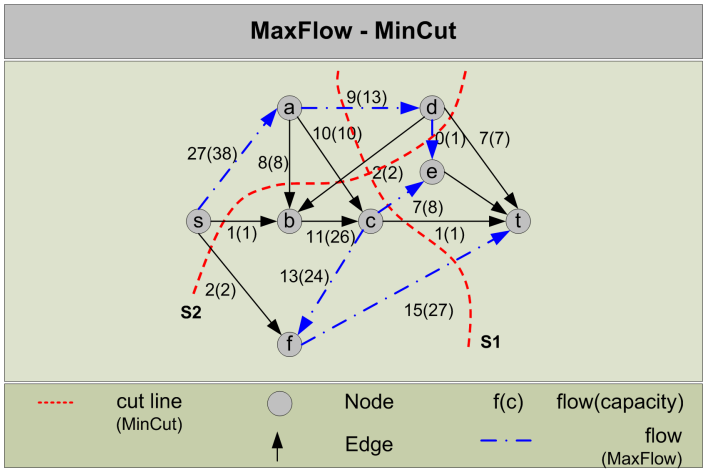


Abbildung 2.16: MaxFlow-MinCut

In Abbildung 2.16 ist ein Netzwerk mit gerichteten Kanten abgebildet. Der maximale Fluss durch das Netzwerk wird von zwei Schnittkanten **s1** und **s2** minimal geschnitten. Die Kapazität der Schnittkante **s1** beträgt 49, die der Schnittkante **s2** 31. Der *MaxFlow-MinCut* Algorithmus [78] geht zunächst von einem zufälligen Knotenpaar (z.B. **s** und **t**) aus und iteriert gerichtet über alle Flusskanten zwischen diesen Knoten. Ist der maximale Fluss berechnet, wird die minimale Schnittkante ausgewählt, die diesen Fluss schneidet. Dieses Verfahren wird iterativ fortgesetzt, bis kein weiterer maximaler Fluss zwischen zwei Knoten detektierbar ist.

Durch die Berücksichtigung des Flusses und des Datendurchsatzes wird mit dem *MaxFlow-MinCut* Algorithmus eine an die Plattform angepasste Partitionierung erreicht bzw. die Plattform kann auf die Partitionierung angepasst werden. Dadurch lassen sich die Kosten für den Aufbau eines Emulatorsystems wesentlich reduzieren. Der Nachteil des beschriebenen Verfahrens ist der große Aufwand der iterativen Berechnung des maximalen Flusses für einen minimalen Schnitt. Für eine Berücksichtigung des sich durch unterschiedliche Eingangsdaten ändernden Flusses im Netzwerk, muss das Verfahren iterativ zur Emulatorlaufzeit angewendet werden. Das führt zu einer signifikanten Reduzierung der Emulatorperformance. Des Weiteren erreicht das Verfahren bei sehr für großen Netzlisten die Lösbarkeitsgrenze (*NP-hard*).

Ein weiteres Partitionierungsverfahren, welches die Laufzeiteigenschaften des Entwurfs einbezieht, ist die *Re-Timing*-Partitionierung, welche in [80] beschrieben ist. Durch Verschiebung und Zusammenfassung von Knoten unter Berücksichtigung des kritischen Pfades zwischen den Knoten werden in diesem Verfahren analog zur Bi-Partitionierung *Cluster* gebildet, welche ein optimiertes Laufzeitverhalten aufweisen. Nach der Bestimmung des kritischen Pfades werden die komplementären Gatter zu einem *Cluster* kombiniert. Das gebildete *Cluster* wird aus der Berechnung des neuen kritischen Pfades ausgegrenzt. Der Algo-

rithmus berechnet in der nächsten Iteration den kritischen Pfad der verbleibenden Schaltung. Lassen sich auf Grund der ausgeglichenen Laufzeiten zwischen den Elementen der Netzliste keine weiteren *Cluster* bilden, bricht der Algorithmus ab. Eine Partitionierung unter Verwendung des das *Re-Timing*-Verfahrens führt zu optimalen Ergebnissen bzgl. der Laufzeit innerhalb eines *Clusters*. Ein Nachteil des Verfahrens ist das ungünstige Verhältnis der Laufzeiten zwischen den *Clustern* und die eingeschränkte *Clusterung* der verbleibenden Partitionen, wenn diese überwiegend aus kombinatorischer Logik bestehen.

Neben den Methoden der *Group-Migration* werden *Metrische Allokationsmethoden* und das *Simulated Annealing*-Verfahren für die Partitionierung auf strukturaler Ebene angewendet [23]. Diese Verfahren erfordern einen erheblich größeren Rechenaufwand gegenüber Verfahren der *Group-Migration*, erreichen jedoch ein hinsichtlich der gegebenen Randbedingungen genaueres Partitionierungsergebnis. Diese Genauigkeit ist für den Einsatz in einem zur Laufzeit rekonfigurierbaren HW-Emulator nicht notwendig. Aus diesem Grund werden in dieser Arbeit ausschließlich Verfahren der *Group-Migration* angewendet.

2.4.2 Methoden der funktionalen Entwurfspartitionierung

Neben den strukturellen Methoden für die Partitionierung einer synthetisierten Netzliste ist es auch möglich, einen Entwurf auf funktionaler Ebene [39] zu partitionieren. Im Gegensatz zu strukturellen Methoden kann diese Partitionierung nicht auf existierenden Kostenfunktionen aufbauen. Die Kosten, wie z.B. Anzahl der Gatter, welche erst durch die Synthese bestimmbar sind, werden in funktionalen Partitionierungsansätzen abgeschätzt oder durch eine Synthese nach der Partitionierung bestimmt. Die Partitionierung des Entwurfs auf der Beschreibungsebene wird des Weiteren als Architekturpartitionierung [68] bezeichnet. Der Vorteil dieses Partitionierungsverfahrens ist die Berücksichtigung funktionaler Zusammenhänge der Partitionen [106].

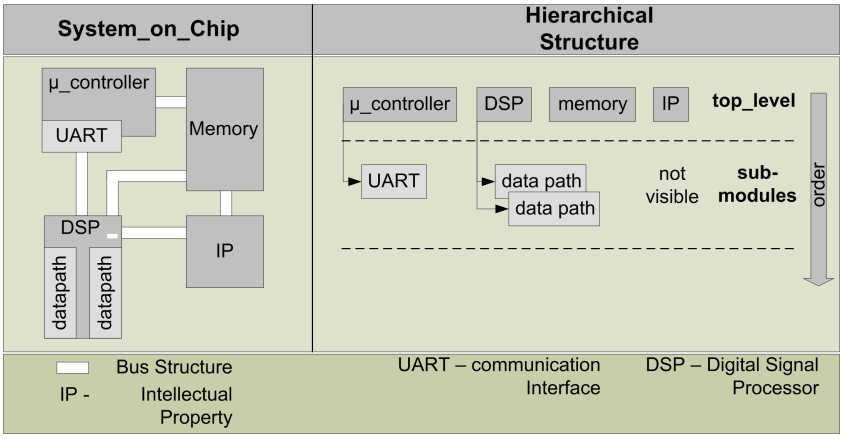


Abbildung 2.17: Funktionale Entwurfspartitionierung

In Abbildung 2.17 ist der Entwurf eines *System-on-Chip* funktional partitioniert. Die *Top-level*-Ebene des Systems enthält einen *Controller*, einen *DSP*, einen *IP-Core* und einen *Memory*. In der zweiten Hierarchieebene (*sub-modules*) sind die Sub-Module des *Controllers* (*UART*) und des *DSP* (*data path*) abgebildet. Im Gegensatz zur strukturellen Partitionierung werden mit der funktionalen Partitionierung Module generiert, welche nach der Synthese funktional unabhängig sind. Eine Gegenüberstellung von struktural

und funktionaler Partitionierung wurde in [105] vorgenommen.

Methode	**Bi-Partitionierung**	**Cone-Algorithmus**	**Funktionale Partitionierung**
Anwendungsebene	Netzliste	Netzliste	Verhaltensbeschreibung oder Netzliste
Systemstruktur	nicht berücksichtigt	berücksichtigt	berücksichtigt
Realisierung	erheblicher Aufwand bei vielen Knoten	gleichbleibend mit Knotenanzahl	gering auf Grund der hohen Abstraktionsebene

Tabelle 2.6: Gegenüberstellung unterschiedlicher Partitionierungsansätze

In Tabelle 2.6 werden drei relevante Partitionierungsverfahren gegenübergestellt. Die unterschiedlichen Eigenschaften haben grundlegende Auswirkungen auf die Anwendung der Partitionierungsmethode hinsichtlich der entstehenden Hardwaremodule. Die Relevanz einer angepassten Partitionierungsmethode für die Anwendung in einem partiell rekonfigurierbaren Hardware-Emulator wird in Kapitel 4 gegeben.

2.4.3 Methoden zur Laufzeitsteuerung (Scheduling)

Nach der Bildung der einzelnen Partitionen, werden diese unter Berücksichtigung der funktionalen und zeitlichen Abfolge gesteuert. Die der Steuerung zu Grunde liegenden Algorithmen können in Hardware oder in Software implementiert werden. Für eine Implementierung der Hardware lassen sich zwei Ansätze unterscheiden. Zum einen kann dezentral der für das *Schedule* angewendete Algorithmus direkt die Übergangsbedingungen eines Zustandsautomaten auslösen, welcher den Steuerungsalgorithmus abbildet. Zum anderen kann die Steuerung der Module extern und zentral realisiert sein. Das dezentrale Verfahren setzt für die Steuerung Kenntnisse bzgl. der Struktur und des Laufzeitverhaltens der Module voraus. Unter Anwendung dieser Kenntnisse schränkt das Verfahren die Flexibilität des Ansatzes ein.

Für eine dezentrale Steuerung ist eine Prozessoreinheit Voraussetzung, welche mit jedem Modul kommunizieren kann. Für die Umsetzung einer dezentralen Steuerung müssen konträre Anforderungen auf unterschiedlichen Systemebenen berücksichtigt werden. Für den neuen Ansatz ist eine direkte Einflussnahme auf die Hardwaremodule eine Steuerung auf RT-Ebene notwendig. Für die Berechnung des *Schedule*, wird hingegen ein flexibel umgesetzter Algorithmus gefordert. Dessen Berechnungen können zunehmend auf integrierten Prozessoren (vgl. Abs. 2.5) direkt auf dem FPGA umgesetzt werden. Im folgenden Abschnitt werden unterschiedliche Ansätze für die Implementierung einer steuerbaren Laufzeitumgebung aus dem Bereich rekonfigurierbarer Hardware vorgestellt.

In der Veröffentlichung [36] wird das *Schedule*-Problem aus den Anforderungen für Steuerung von Multiprozessorsystemen [93], [94] abgeleitet. Die Steuerung für ein System, welches mehrere Aufgaben (*Job*) zeitgleich bearbeitet, entspricht im Wesentlichen der gleichen Aufgabenstellung für die Platzierung der

Hardwaremodule für ein partiell rekonfigurierbares FPGA. Mit der Einschränkung, dass die Berechnung eines *Job* zurückgestellt werden kann, was für die Platzierung der Module nicht gilt. Ist die Platzierung eines Hardwaremoduls inkorrekt, führt das zu einer fehlerhaften Berechnung der folgenden Module. Die Daten für die Stimulierung der Module sind im Gegensatz zur Berechnung eines *Jobs* nicht mehr reproduzierbar. Für die Steuerung rekonfigurierbarer Hardware ist eine taktgenaue Reaktion auf eintretende Ereignisse Voraussetzung. Spekulative Verfahren für die Vorhersage des Auftretens und der Dauer von *Jobs,* wie z.B. dem *Greedy*-Algorithmus, führen bei der Steuerung von Hardwaremodulen zu einem sehr großen Risiko der fehlerhaften Platzierung von Modulen und damit der Beschädigung der HW-Plattform.

In [91] wird eine Methodik beschrieben, welche für die beschleunigte Berechnung eines Fehlermodells unter Verwendung eines hardwarebasierten Emulators konzipiert wurde. Zunächst wird der Modulgraph des Entwurfs partitioniert und die entstandenen HW-Module im Anschluss auf der FPGA-Plattform partiell emuliert. Die Partitionierung erfolgt nach der Synthese auf strukturaler Ebene. Die Module, welche emuliert werden, sind kombinatorische Logikknoten. Auf Grund der notwendigen Schaltvorgänge der Logikelemente in jedem Takt, werden an allen Ein- und Ausgängen eines Hardwaremoduls Speicherzellen platziert. Nach der Emulation des ersten Moduls werden dessen Ausgangswerte gespeichert. Das im Datenfluss folgende Modul wird mit den gespeicherten Werten in der folgenden Emulation stimuliert. Dieser Ansatz ist vergleichbar mit einem Emulatoransatz, welcher in [35] dargestellt ist. Die Steuerung dieses Ansatzes ist auf ein taktgesteuertes (*clock trigger*) Verfahren [28] zurückzuführen, welches die Hardwaremodule direkt vom zuvor berechneten Datenfluss bestimmt. Durch die Rekonfigurationen, die in jedem Takt notwendig ist, hängt die Effizienz des Ansatzes von der Perfomanz des Emulator-FPGAs ab. Dieser Ansatz ist im Wesentlichen für die Emulation von partitionierten Netzlisten geeignet. Die Leistungsfähigkeit und Flexibilität der Steuerung ist für die Emulation von strukturierten und funktional partitionierten Systemen nicht ausreichend.

Die Steuerung für einen bereits bewerteten Ansatz (vgl. Abb. 2.9) basiert auf einem heterogenen Hardware/Software Konzept. Die direkt für die Steuerung der Hardware notwendigen Systemteile sind auf dem Emulator-FPGA implementiert. Die globale Steuerung, welche das Zusammenspiel der gesamten Umgebung realisiert, ist eine flexibel programmierbare Softwarelösung, welche auf einem externen *PowerPC* implementiert ist. Ein wesentlicher Aspekt dieser Realisierung kommt der Synchronisierung der unterschiedlichen Systemteile zu. Der extern platzierte *PowerPC* verfügt über mehrere *Interrupt*-Eingänge, welche die Ereignisverwaltung der Peripherie detektieren und mit dem internen Steuerwerk austauschen. Die Effizienz dieses Steuerungsansatzes hängt dann wesentlich von der Performanz der Synchronisierung von Hardware und Software ab.

In [50] wird ein Ansatz beschrieben, welcher das Problem der Hardware/Software Synchronisierung für FPGAs unter Einsatz des *Xputer* [49] aufgreift. Der Ansatz nutzt die Eigenschaft des *Xputer* bzgl. der Kenntnis über die Laufzeit eines *Tasks* aus. Die Synchronisierung zwischen *Host*-System und *Xputer* erfolgt über einen gemeinsamen Speicher. Ist die Berechnung eines *Task* abgeschlossen, legt der *Xputer* das Ergebnis im Speicher ab. Das *Host*-System detektiert den Schreibzyklus und aktualisiert im Anschluss die Aufgabenliste für den *Xputer.* Daduch wird die Berechnung des folgenden *Task* ausgelöst. Dieser Ansatz ermöglicht eine Synchronisierung ohne zusätzlichen Hardwareaufwand für Steuer- und *Interrupt*-Leitungen. Die Kenntnisse bzgl. der Laufzeit eines *Task* kann für einen HW-Emulator jedoch nur eingeschränkt angenommen werden. Es ist für kombinatorische Logikelemente möglich, deren kritischer Pfad berechnet werden kann, eine Bestimmung der maximalen Laufzeit durchzuführen. Bei der funktionalen System-Emulation hängt die Laufzeit aber von der Vergangenheit des Systems und den Stimulidaten des Emulators ab und ist nicht deterministisch. Aus diesem Grund kann das beschriebene Verfahren für

den neuen Ansatz nicht eingesetzt werden.

Die untersuchten und bewerteten Ansätze für die Steuerung einer rekonfigurierbaren Hardwareplattform setzen überwiegend auf einer Verteilung des zu Grunde liegenden Algorithmus in einen flexiblen Softwareteil und einen taktgenauen Hardwareteil auf. In der vorliegenden Arbeit wurde ebenfalls eine solche Verteilung zu Grunde gelegt.

Neben den dargestellten Verfahren existieren Ansätze, welche spekulative Methoden und die Auswertung der Datenflussgraphen in die Steuerung mit einbeziehen. Für die Verifikation eines Systems, welches eine möglichst weitgehende Abdeckung des Zustandsraumes zum Ziel hat, sind diese Methoden aber nicht geeignet. Für unterschiedliche Eingangsdaten wäre für die Anwendung eines graphenbasierten Ansatzes in jedem Emulationslauf eine erneute Bestimmung des Datenflusses notwendig. Diese Berechnungen würden signifikant die Gesamtperformanz des Emulators reduzieren.

2.5 Zusammenfassung

Kapitel 2 stellt eine Bewertung zum Stand der Technik auf dem Gebiet der funktionalen Verifikation von Schaltkreisen auf der Basis von Hardwareemulatoren vor. Zunächst wurde das Gebiet von Architekturansätzen zur Umsetzung der partiellen Rekonfiguration zur Laufzeit als Grundlage der Arbeit untersucht und bewertet. Als Ergebnis ist festzustellen, dass FPGA-basierte Plattformen deutliche Vorteile hinsichtlich der verfügbaren Anzahl an Gattern und der Performanz gegenüber den SoC-basierten Architekturen haben.

In Abschnitt 2.2.4 standen Multi-FPGA-Systeme im Mittelpunkt der Untersuchungen. Unter Berücksichtigung der Nachteile konventioneller Multi-FPGA-Systeme, welche hauptsächlich im Bereich der Verbindungen zwischen den FPGAs liegen, wurde ein Ansatz für die Optimierung durch den Einsatz eines rekonfigurierbaren FPGAs abgeleitet.

Die dafür notwendige Partitionierung des Entwurfs wurde in Abschnitt 3.2.2 untersucht und bewertet. Die Partitionierung lässt sich in strukturale und funktionale Methoden unterscheiden. Die strukturalen Ansätze basieren auf einer graphentheoretischen Aufteilung der synthetisierten Netzliste unter Berücksichtigung der durch die Synthese bekannten Kosten wie z.B. der Anzahl der Gatter einer Partition. Die funktionale Partitionierung erfolgt hierarchisch in der Verhaltensbeschreibung des Entwurfs. Auf Grund der unbekannten Kosten ist eine iterative Heuristik notwendig, welche unter funktionalem Aspekt eine Aufteilung des Entwurfs mit anschließender Synthese vornimmt.

Ausgehend von den Nachteilen aktueller FPGA-basierter funktionaler Testmethoden, lässt sich eine Optimierung dieser Verfahren durch den Einsatz von Methoden der partiellen Rekonfiguration zur Laufzeit erreichen. Im folgenden Kapitel wird ein Architekturkonzept, basierend auf den Erkenntnissen bestehender Systeme für die Implementierung eines optimierten Hardwareemulators vorgestellt.

3 Konzeption und Architekturentwurf

Die Anwendung von Methoden der partiellen Rekonfiguration zur Laufzeit auf eine konventionelle FPGA-basierte Emulatorplattform lässt nach Auffassung des Verfassers unter Verwendung basierender Emulatorsysteme einen Lösungsansatz für deren Optimierung zu. Ausgehend von den dargestellten bekannten Ansätzen ist durch den Einsatz von Methoden der Laufzeitrekonfiguration eine signifikante Steigerung der Auslastung der einzelnen FPGAs zu erwarten. Dieser Ansatz basiert auf der Platzierung partieller Module eines partitionierten Entwurfs auf einem einzelnen FPGA [15]. Durch eine dem Datenfluss angepasste und auf Systemereignisse reagierende Steuerung, wird eine kontinuierliche Emulation des gesamten Entwurfs erreicht.

Zwei wesentliche Ziele des neuen Ansatzes sind zum einen eine optimierte Ausnutzung der Emulatorplattform und eine damit bei Multi-FPGA Systemen verbundene Reduzierung der
FPGAs. Zum anderen wird durch die Berücksichtigung der Besonderheiten von regulären Architekturen in einem weiteren Optimierungsschritt die Anzahl der für den funktionalen Test notwendigen Rekonfigurationen reduziert.

Das folgende Kapitel gibt einen Überblick über den in der Arbeit verfolgten Ansatz und stellt die einzelnen zu Grunde liegenden Schritte dar. Zunächst erfolgt eine allgemeine Einordnung des Verfahrens in die Verifikation für integrierte Schaltkreise. Darüber hinaus wird eine Abgrenzung des in dieser Arbeit umgesetzten Anteils zu herangezogenen Beiträgen durchgeführt. Der zweite Teil enthält die Beschreibung eines Modells. Die Modellierung dient einer verbesserten Anschaulichkeit des gesamten Verfahrens und der Bestimmung relevanter Parametergrößen. Im vierten Abschnitt wird die eingesetzte Entwicklungsplattform schematisch erläutert und die für den Ansatz relevanten Randbedingungen, welche durch die Plattform gegeben sind, herausgestellt.

3.1 Allgemeine Einordnung und Abgrenzung des Ansatzes

Den Schwerpunkt dieser Arbeit bildet die Untersuchung der Anwendbarkeit von Methoden für die Partitionierung und Laufzeitsteuerung rekonfigurierbarer FPGAs unter Bezugnahme eines pRTR-Emulators für homogene HW-Entwürfe. Hierbei wird der Ansatz auf einem FPGA-gestützten Emulatorsystem experimentell umgesetzt. In Kombination mit einem Simulator ist weiterführend die Verwendung von bestehenden Multi-FPGA Emulatoren vorgesehen. Auf Grund der Verwendung eines FPGAs und der damit gegeben Möglichkeit, reelle Schnittstellen zu implementieren, lässt sich der Ansatz sowohl für die klassische homogene HW-Emulation als auch im heterogenen *Rapid Prototyping* einsetzen. Der Ansatz ordnet sich in die funktionalen Verifikationsmethoden ein.

3.1.1 Ableitung des Optimierungspotentials

Der größte Anteil an den Kosten einer FPGA-basierten Emulatorplattform kommt den eingesetzten FPGAs zu. Durch die Limitierung der Ein- und Ausgänge im Verhältnis zur programmierbaren integrierten Logik

sinkt die effiziente Auslastung der Logikressourcen in den einzelnen FPGAs [9]. Die zunehmend komplexeren Entwürfe aus dem SoC-Bereich führen durch umfangreiche Busstrukturen und einer stark ausgeprägten Heterogenität zu einer weiteren Reduzierung der Emulatoreffizienz. Aktuelle Hardwareemulatoren müssen aus diesem Grund mit einer zunehmenden Anzahl an FPGAs realisiert werden. Dabei wird von den Herstellern der Emulatoren versucht, der Beschränkung der Bandbreiten mit dem Einsatz von differenziellen Verbindungen (z.B. *RocketIO*) zu begegnen. Wie in Abschnitt 2.2.5 dargestellt, führen diese Ansätze zu einem erheblichen Mehraufwand für die Synchronisierung der FPGAs und reduzieren die verfügbare Anzahl an Gattern für die Emulation.

Aktuelle ASIC-Entwürfe folgen in der Struktur weitgehend einer funktionalen Zusammenschaltung von Teilentwürfen (Modulen) und *IP-Cores*. Diese funktionalen Zusammenhänge werden in konventionellen Emulatorplattformen unzureichend oder nicht berücksichtigt.

3.1.2 Modellbildung

Das größte Potential für eine Optimierung von FPGA-basierten HW-Emulatoren ist durch eine größere Auslastung der FPGAs durch eine zeitlich multiple Nutzung gegeben. Dafür ist eine neuartige Betrachtung der funktionalen System-Emulation notwendig.

Aktuelle HW-Emulatoren gehen von einer permanenten Abbildung der vollständigen Netzliste aus. Daraus ergibt sich für das Verhältnis von Gattern des Emulator $G_{Emulator}$ und den Gattern des zu testenden Entwurfs G_{Design}:

$$\frac{\sum G_{Emulator}}{\sum G_{Design}} \leq 1 \tag{3.1}$$

Dieses Verhältnis ist durch die beschriebenen Randbedingungen der Emulator-Plattform in der praktischen Anwendung sehr viel geringer. In [99] wird eine realistische, vom zu testenden Entwurf abhängige Auslastung von ca. 20% gegeben. Unter Berücksichtigung der funktionalen Zusammenhänge und einer angepassten Partitionierung des Entwurfs wird es möglich, die Abbildung der funktionalen Blöcke auf einem einzelnen FPGA zu realisieren.

Für die Umsetzung des Ansatzes werden Methoden der partiellen Rekonfiguration eines FPGAs zur Laufzeit (pRTR) eingesetzt. Für deren Anwendung ist es notwendig, die durch die Partitionierung entstehenden Hardwaremodule in einer funktionalen und zeitlichen Sequenz auf dem FPGA zu konfigurieren.

Das Verhältnis für die Effizienz der Emulatorgatter $G_{Emulator}$ zu den Gattern des Entwurfs ändert sich dahingehend, dass nicht mehr der gesamte Entwurf in die Betrachtung einfließt, sondern das größte Modul des Entwurfs G_{max_Module}. Für das notwendige Verhältnis des neuen Ansatzes gilt dem entsprechend:

$$\frac{\sum G_{Emulator}}{\sum G_{max_Module}} \leq 1 \tag{3.2}$$

Die Gleichungen 3.1 und 3.2 geben keine Aussage bzgl. der zeitlichen Effizienz des Verfahrens. Der durch die Rekonfigurationen zwischen den Modulen entstehende Zeitverlust, wirkt sich direkt auf den Zeitbedarf der gesamten Emulation aus.

Eine Partitionierung des Entwurfs in funktional weitgehend unabhängige Partitionen und deren separate

Implementierung auf das FPGA erweitert den Abbildungsraum um eine weitere Dimension. Dadurch ist es neben der Reduzierung der Plattform-Komplexität möglich, die Synthese des Entwurfs auf Einzel- statt auf Multi-FPGA Systeme zu optimieren.

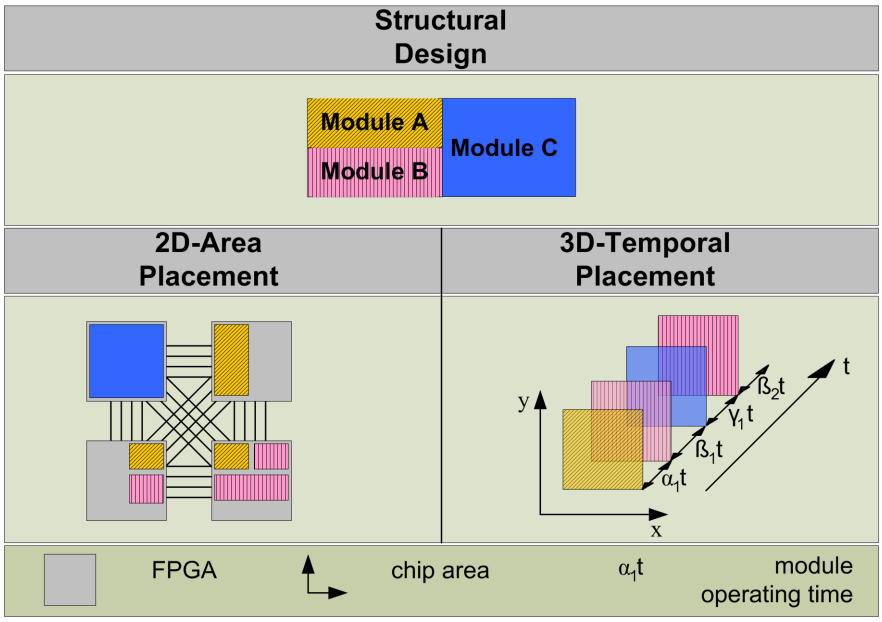


Abbildung 3.1: Gegenüberstellung 2D-3D Platzierung des Entwurfs

In Abbildung 3.1 ist der Ansatz für die Platzierung der Module auf einem einzelnen FPGA in zeitlicher Folge gegenüber der gesamten Platzierung der Netzliste auf einem Multi-FPGA System dargestellt. Die Emulation der Module erfolgt gegenüber der konventionellen Methode iterativ unter Berücksichtigung des funktionalen Systemverhaltens des zu verifizierenden Entwurfs.

Die Effizienz des Ansatzes hinsichtlich des Zeitverhaltens hängt wie dargestellt direkt von den Rekonfigurationszeiten der Module ab. Die Konfigurationszeit T_K gibt die Zeitspanne vom Beginn bis zum Ende des vollständigen Konfigurierens eines FPGAs an. Technologie- und geometrieabhängig berechnet sich aus der gesamten Konfigurationszeit T_K des FPGAs die Zeit für die Konfiguration einer Untermenge $T_{R(x_E)}$ von N-Elementen x_E des FPGAs. Aus dem Verhältnis der zu rekonfigurierenden Anzahl an Elementen N und der gesamten Anzahl an Elementen max_E bestimmt sich die partielle Rekonfigurationszeit $T_{R(x_E)}$ eines Moduls der Größe N-Elemente wie folgt:

$$T_{R(x_E)} = T_K * \frac{\sum_{x=1}^{N} x_E}{\sum_{x=1}^{\max(E)} x_E} \; [s] \tag{3.3}$$

wobei für die gesamte Rekonfigurationszeit T_K unter Berücksichtigung der Konfigurationsbandbreite

$B(reco)$ gilt:

$$T_K = \frac{\sum_{x=1}^{\#E}(x_E * sizeof(E))}{B(reco)} \left[\frac{MByte}{\frac{MByte}{s}}\right] \tag{3.4}$$

Neben den technologiebedingten Randbedingungen für die Konfiguration schränken zusätzliche geometrische Vorgaben der FPGAs die Freiheitsgrade der Rekonfiguration ein. In Abschnitt 2.3.3 wurden FPGA- Lösungen für die Anwendung von Methoden der partiellen Rekonfiguration vorgestellt. Die FPGA-Baureihen der Firma *Xilinx* unterscheiden sich in der Form und dem Umfang der rekonfigurierbaren Fläche, was mit den Entwicklungszyklen der FPGAs zusammenhängt.

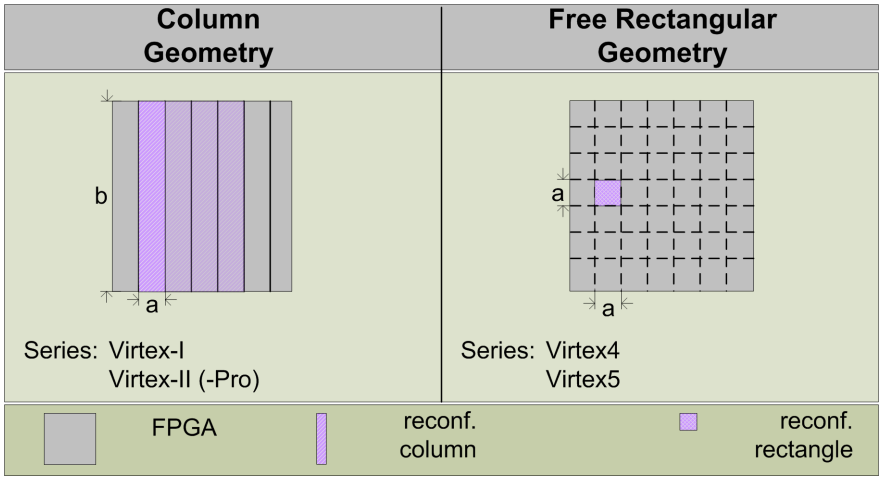


Abbildung 3.2: rekonfigurierbarer Xilinx FPGA

In Abbildung 3.2 sind unterschiedliche geometrische Randbedingungen für *Xilinx*-FPGAs abgebildet. Die *Virtex4* und *-5* Serien sind frei auf dem FPGA rekonfigurierbar. Der Anwender legt in diesem Fall die Größe der Elemente E und deren Platzierung unter Beachtung der kleinsten rekonfigurierbaren Datengröße (*Frame*) frei fest.

Dabei gilt für die Berechnung der Größe von E :

$$sizeof(E)\ [\text{Byte}] = \begin{cases} n * a^2 + (sizeof(\text{frame}) - 1) & | \text{ Virtex4, 5} \\ n * sizeof(\text{column}) & | \text{ VirtexII} - (\text{Pro}) \\ n * 4 * sizeof(\text{column}) & | \text{ VirtexI} \end{cases} \tag{3.5}$$

Aus den Gleichungen 3.2 und 3.3 leitet sich der Zeitbedarf für eine Gesamtemulation ab. Anhand des Beispiels aus Abbildung 3.1 ergibt sich die gesamte Emulationszeit T_E aus der Summe der *download*- und *upload*-Konfigurationen $2 * T_{R(m)}$ und der Laufzeit t_M der einzelnen HW-Module *m* für alle partiellen Emulationen *#P_E*. Dabei ist es zulässig, dass ein HW-Modul datenflussbedingt mehrfach konfiguriert wird.

Es gilt:

$$T_E = \sum_{\text{m=1}}^{\text{\# p_E}} 2 * T_{R(m)} + t_{T(m)} \tag{3.6}$$

- Das Modul wird zunächst in der Zeit $T_{R(m)}$ konfiguriert und im Anschluss für die Laufzeit t_M emuliert.
- Abhängig von den Anforderungen des Emulationslaufes können die Konfigurationsdaten in einem weiteren Schritt mit den gespeicherten Registerwerten vom FPGA heruntergeladen werden. Die benötigte Zeit entspricht der Konfigurationszeit $T_{R(m)}$.

Der Einsatz von Methoden der Rekonfiguration eines FPGAs zur Laufzeit lässt sich grundlegend durch eine Rekonfiguration des gesamten FPGAs oder durch eine partielle Rekonfiguration umsetzen. Die Entscheidung für eine gesamte Rekonfiguration oder für den partiellen Austausch von Modulen hängt von der Granularität der Partitionierung und dem Grad der Parallelisierung *P* ab. Mit einer zunehmenden parallelen Emulation der HW-Module sinkt deren Einfluss der Verarbeitungszeit t_M auf die Gesamtemulation. Die Konfigurationszeit eines Moduls bleibt demgegenüber konstant.

In Gleichung 3.6 wird zunächst nur die Emulationszeit von einem konfigurierten HW-Modul ($P = 1$) berechnet. Werden mehrere HW-Module zeitgleich platziert, reduziert sich die Emulationszeit. Dafür lassen sich zwei unterschiedliche Strategien für den Zeitpunkt der Platzierung und der Zeitspanne der partiellen Emulation unterscheiden.

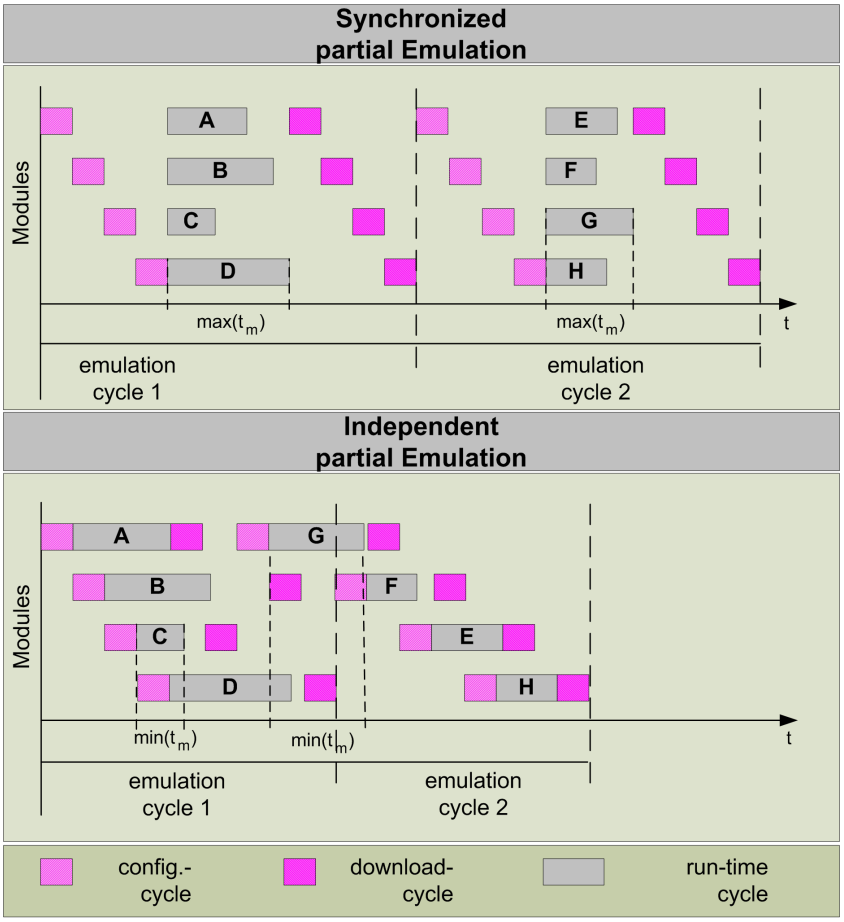


Abbildung 3.3: Abläufe der partiellen Emulationen

Als Voraussetzung beider Ansätze wird angenommen, dass alle HW-Module an jeder beliebigen Stelle auf dem FPGA platziert werden können. In Abbildung 3.3 ist im oberen Teil ein Verfahren abgebildet, welches die Emulation aller platzierten HW-Module synchron durchführt. Im unteren Teil erfolgt der *download*-Vorgang asynchron bereits nach Abschluss der Emulation mit der kürzesten Laufzeit des entsprechenden HW-Moduls. Dabei ist die Bandbreite des Konfigurations-*Interfaces* zu berücksichtigen. Im dargestellten Fall kann jeweils ein HW-Modul geladen bzw. vom FPGA entfernt werden.

- In Gleichung 3.7 ist der Einfluss der Parallelität P unter der Bedingung berücksichtigt, dass alle parallelen Emulationen $\#m_{(P_E)}$ zum gleichen Zeitpunkt begonnen bzw. beendet werden (vgl. Abb. 3.3). Die Zeitspanne zwischen Laden und Entfernen der HW-Module ist durch die Laufzeit des am längsten emulierten HW-Moduls $max(t_{R(m)})$bestimmt.
  Es gilt:

$$T_E = \sum_{m=1}^{\#m_{(P_E)}} (2 * T_{R(m)}) + max(t_{R(m)}) \tag{3.7}$$

Der Einfluss der Parallelität wirkt sich bei der synchronisierten Emulationen der HW-Module nur auf die Laufzeit aus und geht über alle partiellen Emulationen $\#P_E$ in die Gesamtzeit der Emulation mit dem Faktor:

$$\sum_{\#\text{ p_E}} \frac{max(t_{R(m)})}{\#m_{(P_E)}} \tag{3.8}$$

ein.

- Im zweiten Fall erfolgt der Austausch der HW-Module in Abhängigkeit von dessen separater Emulationszeit (vgl. Abb. 3.3). Zunächst wird eine Grundkonfiguration der möglichen HW-Module auf dem FPGA platziert. Die Anzahl der zeitgleich emulierbaren HW-Module $\#m_{(p_E)}$ hängt wiederum vom Grad der Parallelität *P* ab. Die Grundkonfiguration ergibt sich als:

$$\sum_{\text{m=1}}^{\#m_{(P_E)}} T_{R(m)} \tag{3.9}$$

Der Zeitpunkt für eine weitere Konfiguration hängt von der kürzesten Laufzeit $min(t_{R(m)})$ eines platzierten HW-Moduls ab. Vor der erneuten Konfiguration wird das emulierte HW-Modul mit der minimalen Laufzeit vom FPGA entfernt und das folgende HW-Modul $T_{R(\#m_{(P_E)}+1)}$an dessen Stelle konfiguriert.

$$T_E \quad = \quad \sum_{\text{m=1}}^{\#m_{(P_E)}} T + min(t_{R(m)}) + T_{R(min(t_{R(m)}))} + T_{R(\#\, m_{(P_E)}+1)} \tag{3.10}$$

Im zweiten Ansatz (vgl. Gln. 3.10) wird durch die vollständige Berücksichtigung der Parallelität eine Verkürzung der gesamten Emulationszeit erreicht. Das Laden neuer HW-Module erfolgt bereits während der Emulation vorangehender HW-Module. Dadurch erhöht sich die Effizienz mit zunehmender Parallelität der partiellen Emulationen. Eine genaue Bestimmung der gesamten Effizienz ist durch die Abhängigkeit zwischen einzelnen HW-Modulen nicht möglich.

Eine weitere Kenngröße, welche für die Platzierung der HW-Module von Bedeutung ist, ist die Laufzeiteffizienz eines Moduls E_M, welche sich aus dem Verhältnis der Rekonfigurationszeit $T_{R(M)}$ und der zugehörigen partiellen Laufzeit $t_m$ergibt als:

$$E_M \quad = \quad \frac{T_{R(m)}}{t_m} \tag{3.11}$$

Während sich die Rekonfigurationszeit eines Moduls deterministisch aus dessen Größe ableiten lässt, ist die Laufzeit des HW-Moduls mit dem in der Arbeit verfolgen Ansatz nicht vorhersagbar. Die genaue Bestimmung der Laufzeit eines Moduls t_M ist auf Grund der Abhängigkeit der Laufzeit von den Eingangsdaten des Moduls abhängig. Aus diesem Grund wird die Effizienz einer Modulrekonfiguration erst nach dessen Laufzeit bestimmbar. Ein Algorithmus für die Entscheidung und Bestimmung einer Rekonfiguration wird demnach durch die Auswertung der Effizienz eines Moduls im Laufe der Verifikation zunehmend genauer. Dieses auf *Prefetching*-Mechanismen basierende Verfahren ist in Abschnitt 5.6.4 erläutert und sekundär in der Implementierung des Emulators berücksichtigt.

3.2 Ablauf des Verfahrens

Das in Abbildung 3.4 dargestellte Schema veranschaulicht den vollständigen Ablauf des partiell rekonfigurierbaren Emulators unter Einbeziehung des neuen Ansatzes. Folgende Phasen werden abgearbeitet:

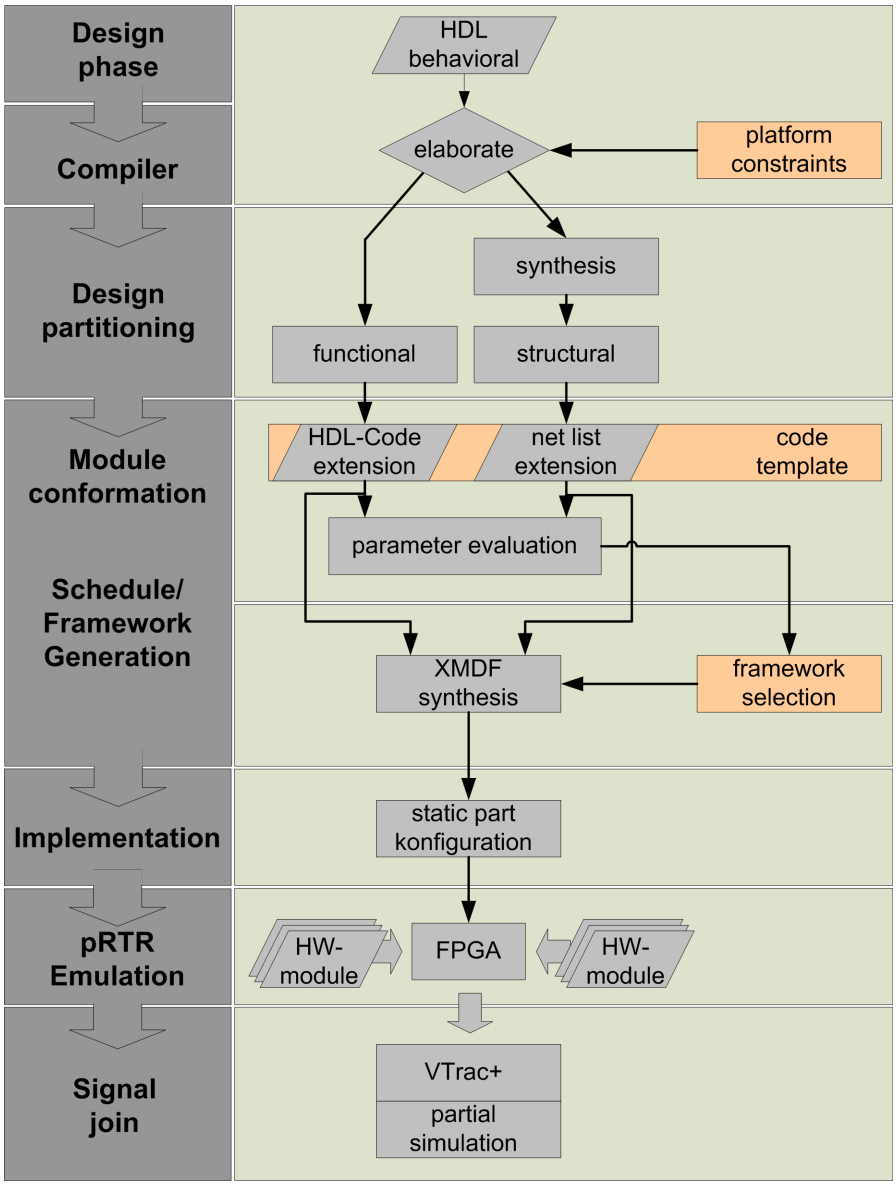


Abbildung 3.4: Ablaufplan pRTR-Emulation

- Im Anschluss an die *Design*-Phase, in welcher die Module des Entwurfs durch eine Simulation verifiziert werden, erfolgt die Partitionierung des Entwurfs. Diese wird sowohl auf funktionaler als auch strukturaler Ebene durchgeführt. Welches Verfahren zur Anwendung kommt, entscheidet sich

nach dem *Elaborate*-Vorgang und hängt direkt von der Struktur und der Größe des zu verifizierenden Entwurfs und den durch die Plattform gegebenen Randbedingungen ab.

- In der darauf folgenden Phase werden die entstehenden Partitionen gemäß den Anforderungen des Emulator -FPGAs zusätzlich um Schnittstellen erweitert und zu rekonfigurierbaren HW-Modulen synthetisiert. Für die Erweiterung der Partitionen kommt ein bestehendes und für diese Anwendung modifiziertes Analyse-*Framework* (*Regatta*) zum Einsatz. Als ein weiteres Ergebnis der Partitionierung werden die exakten Parameter für die Dimensionierung des Emulator-*Frameworks* abgeleitet.
- Nach der Synthese der Laufzeitumgebung wird diese als statisches Modul auf dem FPGA konfiguriert.
- Im darauf folgenden Schritt startet die Emulation der partiellen Entwurfsteile. Die generierten Daten aus den partiellen Emulationen schreibt das *Framework* zurück an einen Simulator, welcher auf dem *Host*-System implementiert ist. Der Simulator setzt unter Verwendung eines bestehenden Kopplungsverfahrens für Simulatoren (*VTrac+*) die partiellen Ergebnisse der Emulationen zusammen.

Eine detaillierte Beschreibung der einzelnen Schritte des Ablaufplans erfolgt in den nächsten Abschnitten. Zunächst werden unterschiedliche Architekturen, welche Schaltkreis- und Systementwürfen zu Grunde liegen können, dargestellt. Aus diesen Applikationen für den Emulator leiten sich in den folgenden Abschnitten die Anforderungen für die Implementierung der Emulatorsteuerung ab.

3.2.1 Entwurfsabhängige Randbedingungen

Neben Standardarchitekturen mit einem einzelnen Steuerwerk und einer angeschlossenen funktionalen Berechnungseinheit (ALU) werden im Schaltkreis- und Systementwurf zunehmend Multi-Prozessor Systeme entworfen. Für die klassische, emulatorgestützte Verifikation dieser Systeme sind auf Grund der zunehmenden Komplexität immer größere Ressourcen hinsichtlich der abbildbaren Gatter notwendig. Des Weiteren berücksichtigen konventionelle Emulatoren keine funktionalen Aspekte des zu verifizierenden Entwurfs wie Nebenläufigkeiten im Datenfluss oder mehrfache Instantiierungen von Hardwareblöcken wie sie z.B. in Multi-Prozessor Systemen umgesetzt werden.

Ein wesentlicher Ansatz dieser Arbeit ist die Berücksichtigung regulärer Strukturen von Entwürfen und der damit verbundenen Reduzierung von notwendigen Emulatorgattern. Für die Erkennung dieser Strukturen werden zwei Wege betrachtet.

- Zum einen kann die Information bereits aus dem Entwurf abgeleitet werden. Dies ist der Fall, wenn der Entwurf unter Verwendung von Entwurfsmustern (*design macro*) erstellt wurde.
- Im anderen Fall werden diese Informationen zusätzlich in den Entwurf eingebracht. Wenn keine Makros verwendet wurden, muss der Partitionierungsalgorithmus während der Analyse die regulären Strukturen erkennen. Diese Erkennung kann auf Grund der Optimierung und Ausflachung der Hierarchie des Entwurfs während der Synthese nur in der funktionalen Beschreibungsebene erfolgen. Die regulären Muster werden in diesem Fall über das Analyse-*Framework Regatta [83]* detektiert.

Das Analyse-*Framework Regatta* weist nach einer Strukturanalyse jedem Befehl (Schlüsselwort) der verwendeten Syntax einen Schlüssel (*Rule-Key*) zu. Die Schlüsselwörter der gesamten Entwurfsbeschreibung

werden zu Ketten aus *Rule-Keys* zusammengefasst und die regulären Muster des Entwurfs in separaten Ketten selektiert. Dafür stehen in dem Analyse-*Framework* zahlreiche Funktionen zu Verfügung, welche an dieser Stelle nicht weiter erläutert werden.

Nach der Extraktion der Ketten transformiert das Analyse-*Framework* diese zurück in die ursprüngliche Beschreibungsform. Als Ergebnis entstehen separate Beschreibungen der Teilentwürfe, welche nach der Synthese hinsichtlich des Verhaltens identische Hardwaremodule abbilden.

Der Ausnutzung von regulären Eigenschaften im Test von Multi-Prozessor Systemen kommt hinsichtlich Effizienz dieser Arbeit eine sehr große Bedeutung zu. Durch das nur einmalig notwendige Konfigurieren der mehrfach instantiierten Entwurfsstruktur, lässt sich der Einfluss der Rekonfigurationszeit für das FPGA signifikant reduzieren.

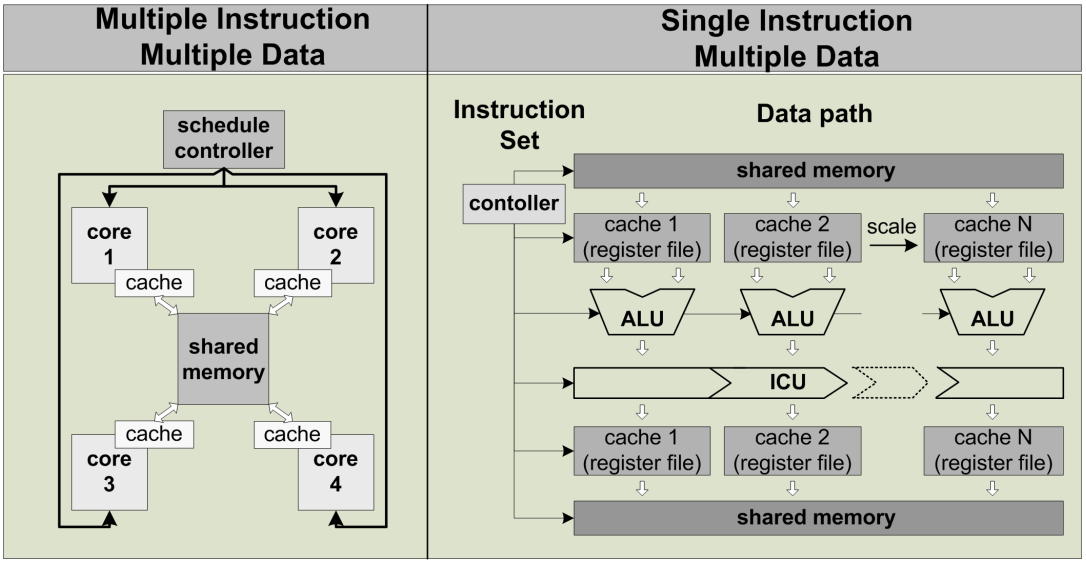


Abbildung 3.5: Reguläre Architekturen

In [57] werden unterschiedliche Ansätze für die parallele Ausführung von Operationen beschrieben. Neben Architekturen, welche die Ausführung einer Instruktion auf unterschiedliche Daten (SIMD - Single Instruction Multiple Data) anwenden, existieren Multi-Prozessor Systeme, in denen mehrere Befehle gleichzeitig auf unterschiedliche Daten angewendet werden (MIMD - Multiple Instruction Multiple Data). In Abbildung 3.5 sind die Unterschiede beider Architekturen dargestellt. Während bei den MIMD-Systemen eine vollständige Parallelität umgesetzt wurde, verarbeitet die SIMD-Architektur, welche z.B. in signalverarbeitenden Vektordatenpfaden zum Einsatz kommt, den gleichen Befehl in jedem Taktzyklus auf jedem einzelnen Prozessor-*Slice*. Die Rechenwerke bzw. Prozessoren der Systeme werden in beiden Architekturen gleichermaßen in der Hardware abgebildet. Die Systeme unterscheiden sich jedoch wesentlich in der Realisierung des Datenaustauschs zwischen den einzelnen Prozessoren bzw. Rechenwerken. In einer MIMD-Architektur erfolgt der Datenaustausch asynchron über einen von allen Prozessoren gemeinsam genutzten Speicher (*shared memory*).

Im Vektordatenpfad werden Datenwörter direkt und synchron zwischen den *Slice* kommuniziert. Dafür ist eine gemeinsam genutzte Verbindungseinheit (ICU - Inter Connectivity Unit) zwischen allen *Slice* plat-

ziert. Diese Verbindungsstruktur ist an das Steuerwerk des Datenpfads angeschlossen und wird analog zu den ALUs über einen Befehlssatz programmiert.

3.2.2 Entwurfspartitionierung

Die Partitionierung stellt unter Berücksichtigung des gesamten Ansatzes einen wesentlichen Schwerpunkt der vorliegenden Arbeit dar. Der umgesetzte Algorithmus wird in Kapitel 4 ausführlich erläutert. Dieser Abschnitt gibt einen Überblick über die Anforderungen an den Partitionierungsalgorithmus. Abhängig von der Struktur und der zu Grunde liegenden Architektur des Entwurfs ist eine Anforderung, das angewendete Partitionierungsverfahren auf unterschiedlichen Ebenen anwenden zu können.

- Die Effizienz eines pRTR-Emulators hängt wesentlich von der optimalen Auslastung des Emulators ab. Diese steigt durch HW-Module, welche über einen sehr großen Zeitraum unabhängig von anderen HW-Modulen emuliert werden können. Insbesondere bei funktionalen HW-Modulen ist das der Fall. Aus diesem Grund wird eine Partitionierung der Entwurfsbeschreibung auf funktionaler Ebene angestrebt.

- Ist es auf Grund der Struktur des Entwurfs oder eines Sub-Moduls nicht möglich, auf funktionaler Ebene zu partitionieren, muss der Algorithmus die synthetisierte Netzliste auf struktureller Ebene aufteilen. Dazu ist vor der Anwendung des Partitionierungsalgorithmus eine Synthese des Entwurfs oder des funktionalen HW-Moduls aus dem ersten Partitionierungsschritt durchzuführen.

Die Auswahl und exakte Parametrisierung des Emulator-*Frameworks* wird auf der Basis der Ergebnisse der Partitionierung durchgeführt. Um alle HW-Module in allen für die Rekonfiguration vorgesehenen Bereichen (rekonfigurierbare Flächen) auf dem FPGA platzieren zu können, werden die Parameter des hinsichtlich Fläche und Wortbreite größten HW-Moduls als Grundlage herangezogen. Die Effizienz des Ansatzes hängt dementsprechend von der Differenz des größten zum kleinsten HW-Modul ab. Eine wesentliche Forderung an den Partitionierungsalgorithmus ist aus diesem Grund eine Aufteilung des Entwurfs unter der Maßgabe, parameteräquivalente Module zu realisieren.

Die durch die Partitionierung entstandenen Module sind auf Grund fehlender Schnittstellen nicht synthetisierbar. Aus diesem Grund ist eine weitere Forderung des Verfahrens die Vervollständigung der Teilbeschreibungen zu synthesefähigen HW-Modulen. Diese Anpassungen müssen analog zur Partitionierung auf funktionaler als auch auf struktureller Entwurfsebene realisierbar sein.

3.2.3 Steuerung der HaHardwaremodule

Nach der Partitionierung des Entwurfs werden die Hardwaremodule auf dem FPGA der Emulatorplattform konfiguriert. An die zeitliche und funktionale Steuerung der partiellen Hardwaremodule (*schedule*) werden unterschiedliche Anforderungen gestellt. Die Steuerung hat zum Ziel, einen Ablaufplan für die

- Platzierung,

- Inbetriebnahme,

- Verbindung und den

- Datenaustausch der separaten Module nach vorgegebenen Parametern zu realisieren.

Zunächst wurde im Rahmen der Arbeit ein Ansatz für die Steuerung der HW-Module untersucht, welcher die Module ohne zusätzliche Steuerungsumgebung verbindet. Dazu wurden an den Ein- und Ausgangsstufen der HW-Module Speicherregister platziert, welche die Ein- und Ausgangsdaten während der Rekonfiguration der HW-Module speichern. Das Ziel des Ansatzes war es, beginnend am Eingang des Datenflusses die Daten modulweise über das FPGA zu propagieren. Dieses Verfahren entspricht grundsätzlich der Emulatorlösung, welche in [35] bzw. [46] umgesetzt wurde mit dem Unterschied, dass der neue Ansatz eine zwei dimensionale Ausbreitung des Datenflusses zulässt.

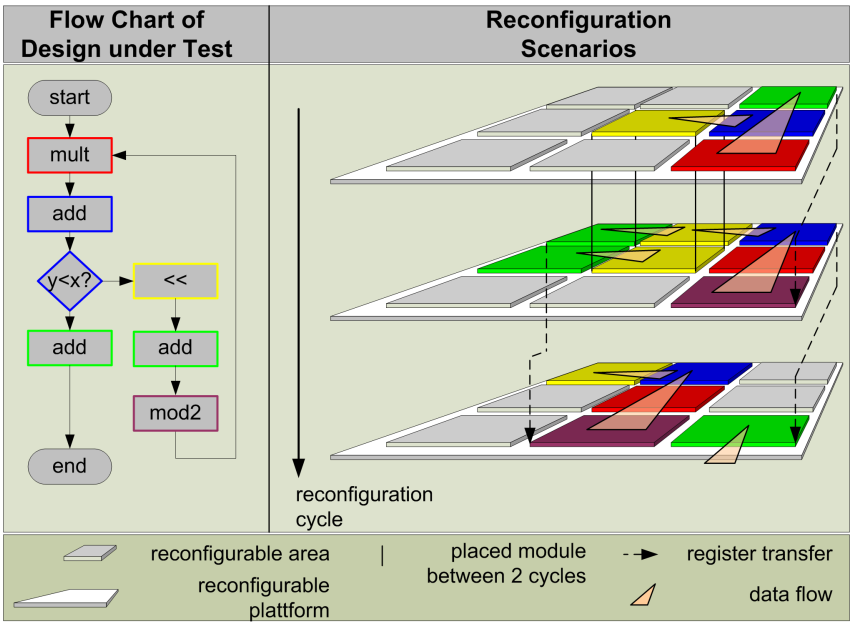


Abbildung 3.6: Emulator Ansatz ohne KommUniversums-Netzwerk

Für die Umsetzung ist es notwendig, die HW-Module um identische Schnittstellen [21, 24] zu erweitern. Des Weiteren ist eine umfangreiche Manipulation der Konfigurationsdateien notwendig, um den Syntheseablauf ohne einen statischen Systemteil realisieren zu können. Neben den hohen Implementierungskosten hängt die Effizienz dieses Ansatzes direkt von der Rekonfigurationszeit ab. Für eine iterative Verarbeitung in einem zu verifizierenden System ist es in dem Ansatz notwendig, die für den Datenfluss erforderlichen HW-Module ebenfalls iterativ auf dem FPGA zu platzieren. Durch die fehlende Kommunikationsstruktur ist ein reaktiver Datenfluss ohne mehrere Rekonfigurationen der HW-Module nicht realisierbar.

Während der Untersuchungen wurde von diesem Ansatz auf Grund des Implementierungs-, Steuerungs- und Syntheseaufwands abgesehen. Für den Ansatz steigt die Effizienz mit einer feineren Granularität. Dadurch ist die simultane Platzierung mehrerer HW-Module realisierbar, was zu einer Abnahme der Rekonfigurationen führt. Des Weiteren sinkt die Rekonfigurationszeit pro Modul.

Für eine effiziente Ausnutzung der Entwurfsstruktur während der Verifikation unter Berücksichtigung einer möglichst geringen Anzahl an Rekonfigurationen ist eine Abbildung der Kommunikationsstruktur des Entwurfs im Emulator-*Framework* notwendig. Aus diesem Grund muss das Emulator-*Framework* hin-

sichtlich der Kommunikation zwischen den einzelnen HW-Modulen die unterschiedlichen Ansätze für Architekturkonzepte berücksichtigen und ausreichend flexibel sein.

Die Steuerung der Laufzeitumgebung rekonfigurierbarer Hardwaremodule beeinflusst ebenfalls wesentlich die Funktionsweise des angestrebten Ansatzes für einen optimierten HW-Emulator auf der Basis partieller Rekonfiguration. Zum einen muss die Steuerung auf Hardwareebene taktgenau eingreifen, zum anderen muss der zu Grunde liegende Algorithmus flexibel programmierbar sein. Diese konträren Parameter lassen sich durch eine angepasste Aufteilung der Hardware- und Softwarekomponenten für die Steuerung direkt auf einem FPGA vereinen. Die Synchronisierung der unterschiedlichen Ebenen ist für eine optimale Ausnutzung der Performanz wesentlich. In Kapitel 5 wird der umgesetzte Ansatz für die Implementierung einer hierarchischen Laufzeitsteuerung detailliert vorgestellt.

3.2.4 Synthese für partiell dynamisch rekonfigurierbare Hardware

An die Partitionierung des Entwurfs schließt sich die Synthese der HW-Module an. Wurde der Entwurf auf strukturaler Ebene partitioniert, ist die Synthese bereits abgeschlossen und die Module entsprechen nach der Erweiterung um die Schnittstellen den Anforderungen der rekonfigurierbaren Plattform. Die Synthese für partiell rekonfigurierbare HW-Module bezieht sich aus diesem Grund auf die funktional partitionierten Entwurfsteile.

Nachdem diese an die Anforderungen des XMDF angepasst wurden (vgl. Abs. 3.2.2), wird der Syntheseablauf durchgeführt. Dafür ist eine Unterteilung der Module in ein statisch platziertes Modul (*top_level*) und die rekonfigurierbaren HW-Module (*modules*) notwendig. Das statische *top_level*-Modul, über welches die physikalischen Schnittstellen des FPGAs mit den rekonfigurierbaren Bereichen (RA - Reconfigurable Area) verbunden werden, muss über die gesamte Laufzeit des Emulators platziert werden.

Die Synthese für eine rekonfigurierbare Plattform wird von jedem Hersteller unterschiedlich umgesetzt. In dieser Arbeit wird auf eine Testplattform zurückgegriffen, auf welcher zwei FPGAs der Firma *Xilinx* implementiert sind. Von *Xilinx* wird für die Synthese eines rekonfigurierbaren Systems ein optimierter Syntheseablauf bereitgestellt (XMDF vgl. Abschnitt 2.3.4). Dieser Ablauf besteht aus den in Abschnitt 2.3.4 beschriebenen drei Phasen:

- *initial budgeting*,
- *active module implementation* und
- *final assemble*.

Für die Anwendung in einem FPGA-basierten Verifikationssansatz ist ein automatisierter Ablauf der Synthese eine wesentliche Anforderung für die Integration des Systems. Der *Xilinx-Modular-Design-Flow* erfüllt in der ursprünglichen Form nicht diese Anforderung.

Aus diesem Grund wurde im Vorfeld dieser Arbeit ein skriptenbasierter Ablauf entworfen, welcher die für die einzelnen Teilschritte notwendigen Syntheseskripte generiert [13]. Ausgehend von einer Beschreibung des Systems für das *top_level* (*design.*tops) und für die Module (*top*.mods) (Abb. 3.7) wird die Synthese für ein partiell rekonfigurierbares System wesentlich vereinfacht. Neben den Syntheseskripten ist die automatische Generierung der .ucf-Datei (.ucf - User Constraint File) eine Voraussetzung für die automatisierte Anwendung des Emulators. Diese Datei hängt direkt vom verwendeten FPGA ab und legt die physischen Randbedingungen des FPGAs zu dessen Peripherie (Belegung der Ein- und Ausgänge) als auch

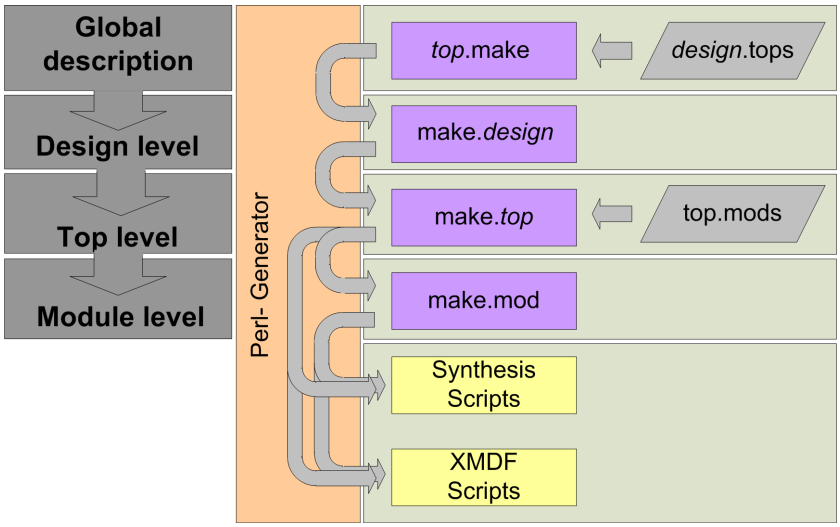


Abbildung 3.7: Automatische Scripten-Generierung XMDF-Synthese

nach innen (Festlegung der rekonfigurierbaren Flächen) fest. Die automatische Erstellung der .ucf-Datei ist Inhalt einer Vielzahl von Veröffentlichungen [34] [21].

Von der Firma *Xilinx* wird für die Erstellung der Datei die Software *PlanAhead* angeboten. Diese Werkzeug wurde in der Arbeitsgruppe evaluiert, aber auf Grund einer unzureichenden Unterstützung für eine Skriptensteuerung als ungeeignet bewertet. Die .ucf-Datei ist für die Erstellung und Konfiguration des Emulator-*Frameworks* im aktuellen Stand der vorliegenden Arbeit nicht automatisch generierbar. Auf Grund der prototypischen Umsetzung des Ansatzes ist der Einfluss auf den gesamten Ansatz jedoch vernachlässigbar.

3.2.5 Kopplung der Emulationsergebnisse (Joining)

Die dem Ansatz zu Grunde liegende Aufteilung des Entwurfs und der partiellen Emulation der Entwurfsteile führt zu einer Vielzahl von Ergebnissen, deren Berechnung zu unterschiedlichen Zeitpunkten stattfindet. Die Zusammenführung der einzelnen Signalwerte wird auf dem *Host*-System unter Verwendung eines Standardsimulators realisiert. Der Simulator wurde um das Programm *VTrac+* [14] erweitert. Diese Programmerweiterung wurde am *Fraunhofer-Institut für Integrierte Schaltungen in Dresden* für die Kopplung von Simulatoren entwickelt. Eine Funktion der Software ermöglicht es, dedizierte Signale von einer externen oder internen Signalquelle direkt in einem Simulationslauf zu stimulieren. Dabei muss im Gegensatz zu vergleichbaren Werkzeugen der Simulationslauf nicht unterbrochen werden. Weitere Funktionen, welche in dieser Arbeit nicht verwendet wurden, ermöglichen das Auslesen von Signalwerten und den direkten Vergleich unterschiedlicher Signaltabellen während der Simulation.

Für die Verwendung von *VTrac+* wird eine Simulation gestartet, welche alle für die Verifikation relevanten Signale enthält. Das Programm greift im Anschluss direkt auf die vom Simulator verwendete Datenbasis zu und manipuliert die enthaltenen Signalwerte.

In Abbildung 3.8 ist das Verfahren für ein Multi-FPGA System dargestellt. Die unabhängig voneinander agierenden Emulator-FPGAs schreiben die Ergebniswerte an die Datenbasis des Simulators, welche auf

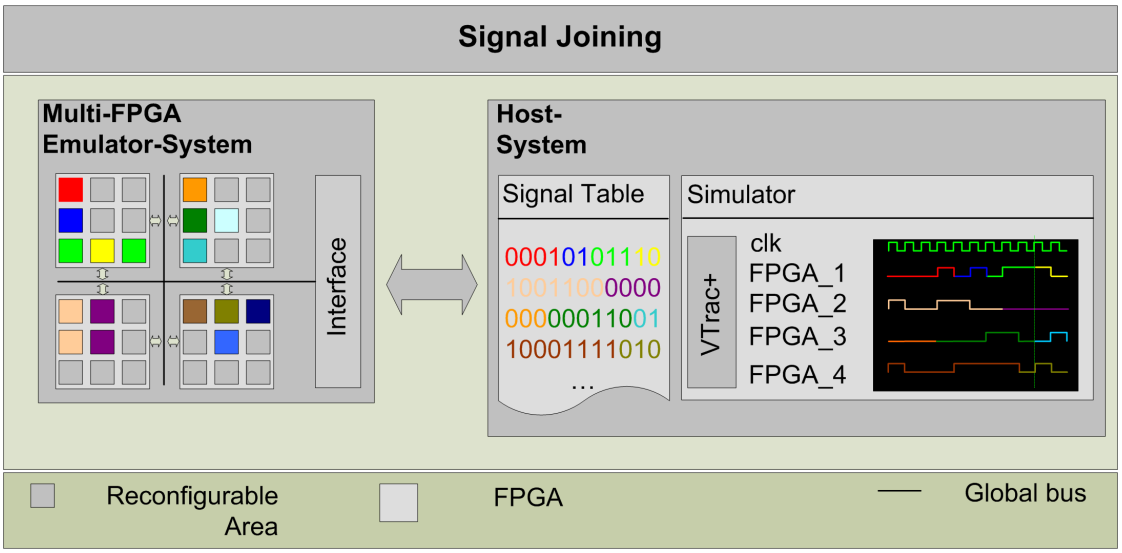


Abbildung 3.8: Kopplung der TeileEmulationen

dem *Host*-System abgelegt ist. In der Signaltabelle entspricht jede Spalte einem Abtastzeitpunkt.

Die Umsetzung und der Test dieses Ansatzes wurden in [19] und [16] für die Kopplung mehrerer Simulatoren ausführlich bearbeitet und bewertet. Aufbauend auf den Ergebnissen aus [19] wurde eine Erweiterung realisiert [14], welche externe Signale mit einbezieht. Dadurch ist die Kopplung extern gespeicherter Signallisten von unterschiedlichen Plattformen mit einem Simulator während der Laufzeit realisierbar. Diese Kopplung stellt die Grundlage für das Zusammenführen der einzelnen Emulationsergebnisse in dieser Arbeit dar. Durch eine globale Synchronisation der FPGAs ist die Einbeziehung mehrerer FPGAs als Emulatorplattform möglich. Die Ergebnisse der vorgelegten Arbeit beziehen sich aber ausschließlich auf ein Einzel-FPGA System.

3.3 Entwicklungsplattform

Die erarbeiteten Methoden und Verfahren wurden für eine Bewertung des Ansatzes prototypisch auf einem FPGA-Entwicklungssystem umgesetzt. Das zu Grunde liegende System umfasst zwei FPGAs der Firma *Xilinx*. Ein *VirtexII-Pro*-FPGA fungiert als Kommunikationsschnittstelle des Systems mit der Umgebung. Ein weiteres FPGA der *Virtex4-LX*-Serie wird im Rahmen dieser Arbeit als Emulator eingesetzt.

In Abbildung 3.9 ist die Entwicklungsplattform schematisch dargestellt. Das FM480-System der Firma *4DSP* [1] wird als PMC-Karte (PMC - Peripheral Mezzanine Card) in ein *Host*-System integriert. Zur Unterstützung unterschiedlicher Standards für die Kommunikation mit dem *Host*-System ist der Protokoll-*Controller* als IP-Core auf dem *VirtexII-Pro*-FPGA implementiert. Dieses FPGA verfügt zusätzlich über einen integrierten *PowerPC,* auf welchem der Algorithmus für die Steuerung der Hardwaremodule implementiert wurde. Auf Grund der Auslastung des *VirtexII-Pro*-FPGA ist dieser als zusätzlicher Emulator-FPGA ungeeignet.

Das zweite auf der Testplattform implementierte FPGA ist ein *Virtex4 LX60* mit einer Anzahl von 6656 CLB. Dieses FPGA wird für die Konfiguration des Emulator-*Frameworks* eingesetzt. Zusätzlich sind an

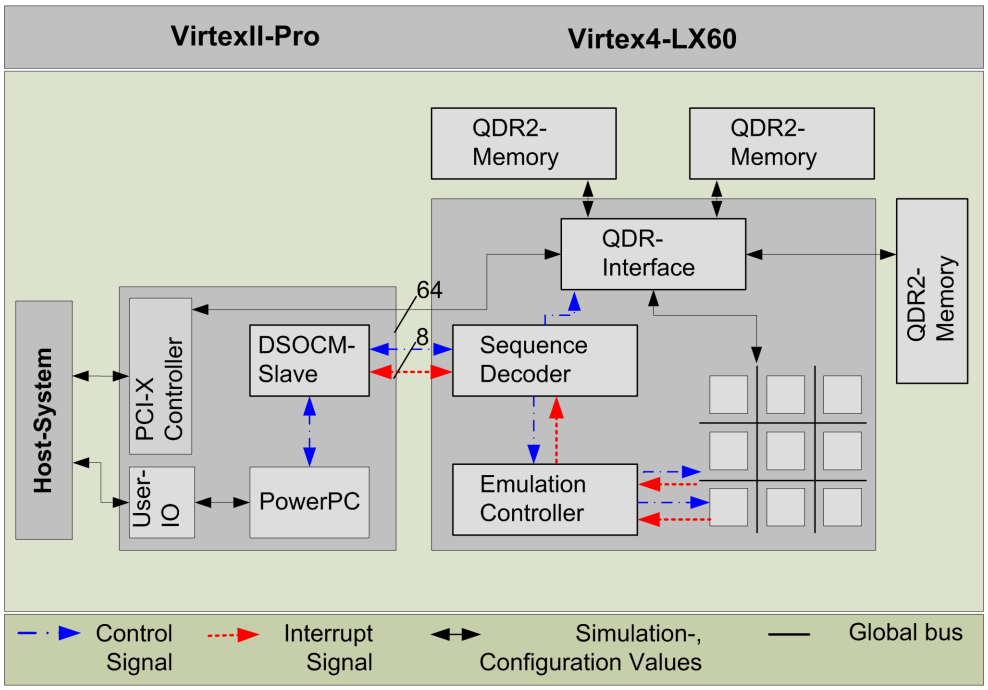


Abbildung 3.9: Blockschaltbild FM480 Entwicklungsplattform

das *Virtex4*-FPGA drei QDR-SRAM-Speicherbänke (QDR - Quad Data Rate), (SRAM - Static Random Access Memory) angeschlossen. Diese Speicher haben zwei getrennte Lese- und Schreibkanäle, welche beide mit doppelter Datenrate arbeiten (DDR - Dual Data Rate). Dadurch sind vier Speicherzyklen pro Takt möglich. Die Wortbreite der Speicher beträgt 64 Bit, welche sich auf zwei Zugriffe von jeweils 32 Bit aufteilen lässt. Die Verbindung zwischen dem FPGA für die Kommunikation und dem Emulator-FPGA besteht aus einem Bus mit einer Wortbreite von 88 Datenbit und mehreren unabhängigen Takt- und Steuerleitungen.

Für eine optimale Performance wurden die QDR-SRAM-*Controller* in den Entwurf des Emulator-*Frameworks* mit einbezogen. Eine detaillierte Beschreibung der Komponenten und der Umsetzung des *Frameworks* wird in Kapitel 5 gegeben.

3.4 Zusammenfassung

In Kapitel 3 wird das Konzept und der Ablauf des neuartigen Ansatzes vorgestellt. Durch den Einsatz von Methoden der partiellen Rekonfiguration eines FPGAs zur Laufzeit werden Auslastung und Performance der Emulation gesteigert. Zunächst wird der neue Ansatz modelliert. Ziel dieser Modellierung ist eine Bestimmung der relevanten Systemgrößen und Zusammenhänge, welche für den Ablauf der partiellen Emulation entscheidend sind.

Für die Umsetzung des Ansatzes ist im ersten Schritt eine Partitionierung des Entwurfs unter gegebenen Randbedingungen notwendig. Aus den Ergebnissen der Partitionierung werden die separaten HW-Module generiert. Der zweite notwendige Schritt umfasst die Ablaufsteuerung der HW-Module während der parti-

ellen Emulationen.

In der Arbeit wurden zwei Ansätze für die Platzierung und die Kommunikation der HW-Module untersucht. Der ersten Ansatz basiert auf einer direkten Kommunikation der HW-Module untereinander. Dieses Verfahren führt auf Grund der fehlenden Kommunikationsstruktur für reaktive Entwürfe zu einem signifikanten Aufwand für die Rekonfiguration der HW-Module. Aus diesem Grund werden die weiteren Untersuchungen unter Betrachtung einer Kommunikationsstruktur auf dem Emulator-FPGA durchgeführt. Die Bestimmung wesentlicher Parameter für das notwendige Emulator-*Framework* werden aus den Ergebnissen der Partitionierung abgeleitet. Das *Framework* wird in Verbindung mit den *Design*-Beschreibungen der partiellen HW-Module durch Verwendung des *Xilinx-Modular-Design-Flow* synthetisiert. Dafür ist die Anwendung eines automatisierten Syntheseablaufs erforderlich. Dieses Verfahren wurde aus weiteren Arbeiten übernommen, welche sich an diese Implementierungen anlehnen. Die während der Emulation berechneten Verifikationsergebnisse werden im letzten Schritt in einem Standardsimulator zusammengeführt. Dafür ist die Anwendung eines zusätzlichen Programms notwendig, welches in mehreren Arbeiten zu dieser Problematik untersucht und bewertet wurde.

Des Weiteren wird die eingesetzte Testplattform für die Implementierung des Emulator-*Frameworks* dargestellt. Diese aus zwei FPGAs bestehende Plattform ist in ein *Host*-System integriert, welches den gesamten Ablauf des Verfahrens steuert und überwacht. Für eine optimale Steuerung des Emulator-FPGAs ist eine Aufteilung der Steuerung in einen flexiblen Softwareteil und einen taktgenauen Hardwareteil notwendig. Der Hardwareteil ist als *Controller* direkt auf dem Emulator-FPGA implementiert. Dieser wird durch den *Schedule*-Algorithmus gesteuert, welcher auf dem *PowerPC* des Kommunikations FPGAs implementiert ist. Das für die taktgenaue Steuerung umgesetzte Protokoll ist synchron und wird über *Interrupt*-Leitungen mit dem Emulator-*Controller* verbunden.

Den methodischen Schwerpunkt der Arbeit bildet die Partitionierung des Entwurfs und die Steuerung der HW-Module. Das folgende Kapitel 4 beschreibt zunächst den realisierten Algorithmus für die Partitionierung auf unterschiedlichen Entwurfsebenen. Im darauf folgenden Kapitel 5 wird der Entwurf des Emulator-*Frameworks* vorgestellt.

4 Entwurfs-Partitionierung

Eine wesentliche Anforderung für die Anwendung des in Kapitel 3 vorgestellten Ansatzes für die Optimierung von FPGA-basierten HW-Emulatoren ist eine angepasste Partitionierung des Entwurfs. In Abschnitt 3.2.2 wurden die Anforderungen an den Algorithmus der Partitionierung spezifiziert. Im folgenden Kapitel wird das umgesetzte Verfahren für die Aufteilung eines Entwurfs in unterschiedlichen Ebenen erläutert.

Zunächst wird ein Überblick des gesamten Partitionierungsverfahrens gegeben. Im Weiteren wird auf ein Verfahren eingegangen, welches für die Partitionierung auf funktionaler Ebene umgesetzt wurde. Ist eine Partitionierung auf funktionaler Ebene nicht möglich oder erfüllen die funktional partitionierten Module nicht die geforderten Randbedingungen, welche durch das Emulator-*Framework* vorgegeben sind, wird die Partitionierung auf strukturaler Ebene fortgesetzt. Dieses Verfahren wird in Abschnitt 4.2 erläutert.

4.1 Funktionale Entwurfspartitionierung

Die Partitionierung der Entwurfsbeschreibung auf funktionaler Ebene basiert auf der Auswertung von Informationen eines standardisierten Entwurfswerkzeugs (z.B. *HDL-Designer ©MentorGraphics*). Die folgenden Ausführungen entsprechen dem dedizierten und von *MentorGraphics* favorisierten Entwurfsablauf für die Integrierung von *IP-Core* in FPGA-Entwürfen [33]. In diesem werden wesentliche und für die Partitionierung notwendigen Schritte erläutert.

- Zunächst wird der Entwurf in einer Hardwarebeschreibungssprache (HDL) spezifiziert und im Anschluss für die Analyse unter Verwendung des in den *HDL-Designer* integrierten *Compiler* in ein simulierbares Programmformat übersetzt. Unter direkter Verwendung eines HDL-Simulators (z.B. *Questa, ModelSim*) erfolgt die Übersetzung des Entwurfs durch den internen *Compiler* des Simulators. Während der Übersetzung führt das verwendete Werkzeug im ersten Schritt eine syntaktische und funktionale Analyse des Entwurfs durch. Diese Prüfung testet den Entwurf zunächst hinsichtlich der Fehlerfreiheit der verwendeten Syntax (*VHDL* bzw. *Verilog*). Die funktionale Analyse kontrolliert die umgesetzten Restriktionen der verwendeten Sprachsyntax. In diesem Schritt werden unter anderem Fehler erkannt, die z.B. durch eine falsche oder ungenaue Zuordnung von Signalen entstehen. Die Anforderungen bzgl. der einzuhaltenden Restriktionen hängen von der eingesetzten Beschreibungssprache ab. In der vorliegenden Arbeit basieren die Implementierungen auf der Beschreibungssprache *VHDL*. Durch die Überprüfungen der Entwurfswerkzeuge werden unter Berücksichtigung der HDL-Syntax Beschreibungsfehler bereits in dieser Phase der Systementwicklung ausgeschlossen. Des Weiteren wird die Vollständigkeit des Entwurfs in Bezug auf alle instantiierten Module, Sub-Module und *IP-Cores* geprüft. Fehlende Komponenten werden durch das Entwicklungswerkzeug angezeigt.

- Der Beschreibung (*discovery process* [33]) folgend übersetzt ein *Assembler* den Entwurf. Mit dem Entwicklungswerkzeug *HDL-Designer* ist es im Anschluss möglich, die Struktur des Entwurfs zu

visualisieren. Die Abbildung der dem Entwurf zu Grunde liegenden Hierarchie der funktionalen Blöcke ermöglicht einen Überblick bzgl. der Beschaltung und Instantiierungen einzelner Ressourcen in der *Design*-Hierarchie (vgl. Anhang 8.4). Die Entwurfssoftware von *MentorGraphics* wird durch ein dediziertes Datenformat (IBD - Interface Based Design) die Modulpartitionierung unterstützt. Das IBD-Format [85] repräsentiert die Hierarchie eines strukturierten Entwurfs unter Berücksichtigung der implementierten Module, den Verbindungen zwischen den Modulen und den *top_level*-Schnittstellen (*Ports*) des Entwurfs. Diese Informationen werden graphisch und in Form einer Textdatei zu Verfügung gestellt. Durch Verarbeitung des Textformats ist eine automatisierte Verarbeitung der Entwurfsbeschreibung gegeben.

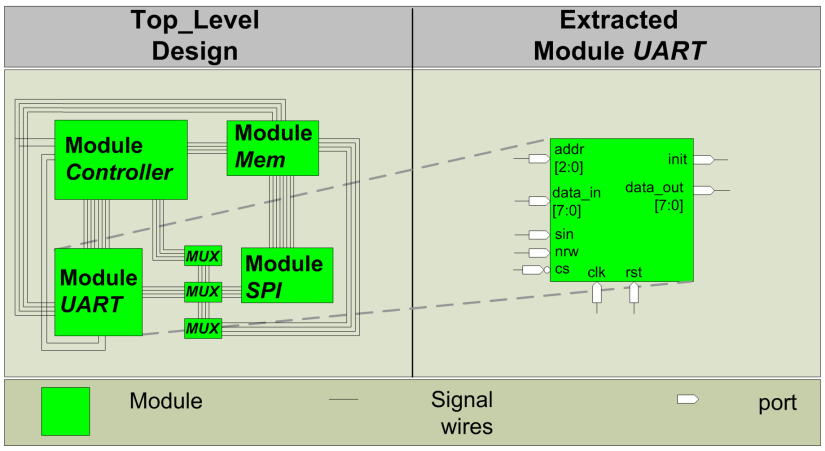


Abbildung 4.1: Hierarchische Systembeschreibung [33]

- In der vorgestellten Arbeit wird die Partitionierung zunächst auf Basis der graphischen Repräsentation unter Verwendung des IBD-Formats durchgeführt. In Abbildung 4.1 ist die Hierarchie eines *top_level*-Entwurfs visualisiert. Die Blöcke repräsentieren die Module, welche unter Einbeziehung von Signalen miteinander verbunden sind. Eine Extraktion der Module ist unter Verwendung der Benutzeroberfläche durch eine Kopie des Moduls (z.B. *UART*) in ein separates Arbeitsfenster umgesetzt. Eine automatisierte Verarbeitung ist durch die Auswertung der Textdatei vorbereitet, im aktuellen Stand der Arbeit wird dieser Schritt auf Grund der abnehmenden Transparenz einer automatischen Verarbeitung manuell umgesetzt.
- Die dem Modul zu Grunde liegende HDL-Beschreibung enthält nach der Entfernung aus dem gesamten Entwurf die Verhaltensbeschreibung und die Schnittstellen des Moduls. Die Verbindungsleitungen zu anderen Modulen entfallen.

Eine weitere Umsetzung der Partitionierung unter funktionalen Gesichtspunkten ist die Berücksichtigung von Entwurfsmustern für reguläre Architekturen. Entwurfsmuster werden eingesetzt, um den Programmieraufwand während der Schaltkreis- und Systementwicklung zu reduzieren. Dabei werden regulär implementierte Module generisch beschrieben und während der Implementierung parametrisiert instantiiert. Durch die Auswertung des IBD-Formats der Systembeschreibung werden die mehrfach instantiierten Mo-

dule selektiert und deren Kostenbewertung und Synthese auf Grund der äquivalenten Parameter nur für ein HW-Modul des parallelen Entwurfsteils durchgeführt.

Ein weiterer Ansatz für eine ressourcenoptimierte Anwendung des neuen Verfahrens ist die Erweiterung der Spezifikation und des Architekturentwurfs um Informationen bzgl. der Entwurfsarchitektur. Aus diesen Informationen werden vom Partitonierungsalgorithmus definierte Muster für die Emulation abgeleitet. Dadurch ist eine Partitionierung und eine Generierung des Emulator-*Frameworks* möglich, welche genau auf den Entwurf abgestimmt ist. Diese Vorgehensweise wird von IP-Entwicklern bereits in ähnlicher Form praktiziert [47].

In diesem Verfahren werden große reguläre Entwürfe, welche sich nicht auf einzelnen FPGAs platzieren lassen, partitioniert und in verkleinerter Form sukzessive emuliert. Unter Berücksichtigung der für die Funktion notwendigen Komponenten und der Reduzierung der parallelen Strukturen lässt sich der Ressourcenbedarf für die gesamte Entwicklungsplattform reduzieren.

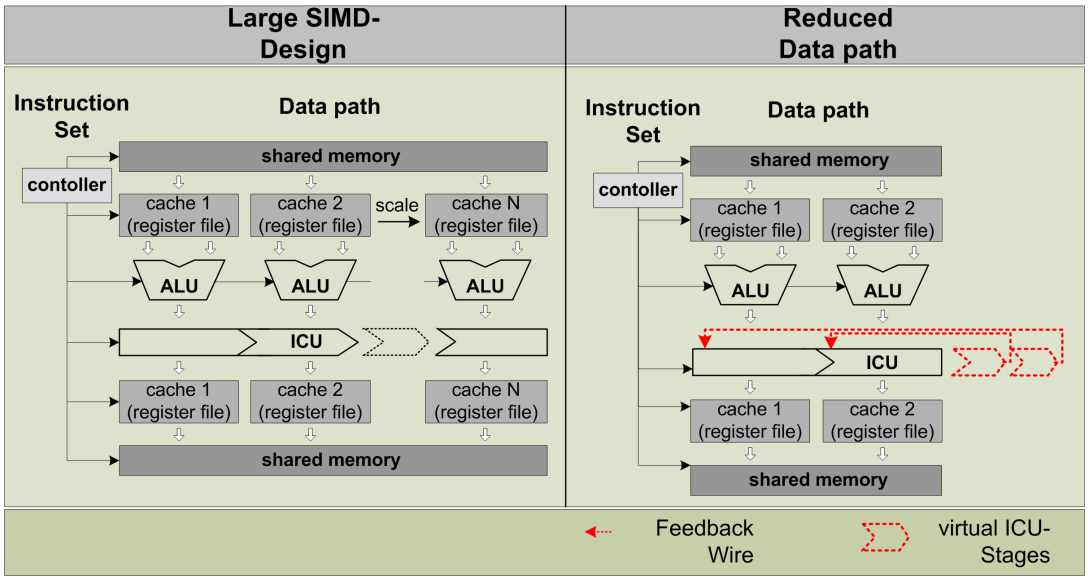


Abbildung 4.2: Emulation für reduzierte reguläre Architekturen am Beispiel SIMD Datenpfad

In Abbildung 4.2 ist ein SIMD-Datenpfad abgebildet, welcher für die Emulation auf zwei instantiierte *Slice* reduziert wurde. Eine Herausforderung während der Emulation ist die Abbildung der Funktionalität des Verbindungsmoduls (*ICU*). Aus diesem Ansatz lässt sich der notwendige Aufwand für die zwingend vom Emulator abgebildete Hardware ableiten. Neben dem Befehls-*Decoder* (*Instruction Set Controller*) muss die gesamte ICU-Funktionalität auf dem FPGA konfiguriert sein. Dem gegenüber ist der Aufwand für die Implementierung der Datenpfade geringer. Diese werden für einen Emulationszyklus mehrfach durchlaufen. Für eine vollständige Abbildung der ICU-Funktionalität ist die Abbildung von mindestens zwei Datenpfaden ohne eine Rekonfiguration erforderlich.

Grundsätzlich wird für die Partitionierung regulärer Architekturen in Module für die Datenverarbeitung und in Module für die Datensteuerung unterschieden. Die Steuermodule werden durch das *Framework* abgebildet. Die Module, welche die Datenverarbeitung repräsentieren, sind abhängig vom Grad der Parallelität in zeitlicher Abfolge darstellbar. An dieser Stelle überschneiden sich Verfahren für die Partitionierung

und Verfahren für die Steuerung der Module (*schedule*), welche in Kapitel 5 beschrieben werden.

Die Verwendung der Entwurfsdaten regulärer Architekturen, welche während der funktionalen Partitionierung durch die Anwendung verschiedener Verfahren und Werkzeuge (*HDL-Designer*) erkannt und ausgewertet werden, führt zu einem erheblichen Optimierungspotential des dargestellten Ansatzes. Die Anwendung stellt vor allem für die Generierung des Emulator-*Frameworks* eine signifikante Verbesserung dar und wird aus diesem Grund in Abschnitt 5.6.2 übergreifend betrachtet.

Nach der Separierung der funktionalen Module ist auf Grund der fehlenden Verbindungsleitungen (siehe Abb. 4.1) und der dadurch nicht mehr erreichbaren Stimulidaten der *Testbench* eine weitere Simulation der Module nicht mehr möglich. Die für die Bestimmung der Kosten eines Moduls notwendige Synthese bleibt demgegenüber unbeeinträchtigt. Für die Synthese der Module wird die Software *Precision* von *MentorGraphics* eingesetzt.

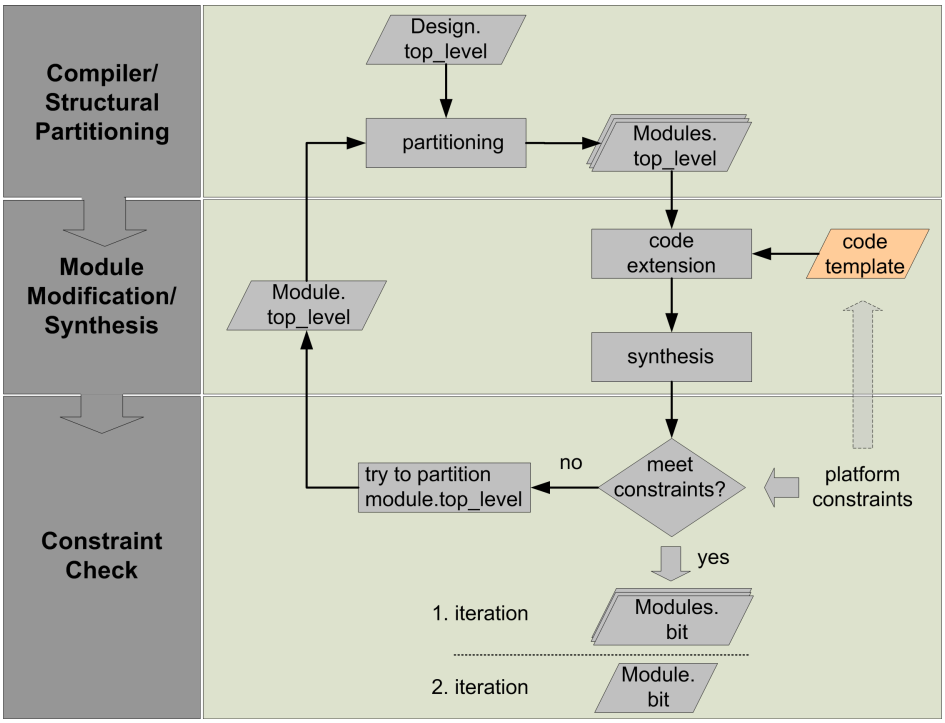


Abbildung 4.3: Funktionale Partitionierung

Auf Grund der Verwendung eines Verbindungsnetzwerks für die Kommunikation zwischen den rekonfigurierbaren HW-Modulen sind die Randbedingungen für die Generierung der Module weitgehend festgelegt. Im aktuellen Stand der Arbeit erfolgt der Datenfluss durch die Module ausschließlich von der Oberkante eines Moduls zu dessen Unterkante. Diese Einschränkung ist für eine Reduzierung der Implementierungskosten des im *Framework* enthaltenen Verbindungsnetzwerks notwendig. Dadurch reduziert sich ebenfalls der Aufwand für die Implementierung der *slice macro*, welche die dynamisch platzierbaren HW-Module mit statischen Emulator-*Framework* verbinden. In Abbildung 4.3 ist der umgesetzte Ablaufplan für die funktionale Entwurfspartitionierung dargestellt.

Nach der Synthese der Module wird die entstandene Netzliste hinsichtlich der Kosten bewertet. Die Kostenfunktion ist durch die Größe des FPGAs und des implementierten Emulator-*Frameworks* vorgegeben. In der umgesetzten Arbeit wurden unterschiedliche Varianten für die Parametrisierung des *Frameworks* untersucht. Als Ergebnis werden Funktionswerte für die Bestimmung der Kosten wie die Anzahl der CLBs des gesamten *Frameworks* für eine bestimmte *Framework*-Konfiguration abgeleitet. Diese sind abhängig von der Anzahl generierbarer HW-Module und deren lokaler Zusammengehörigkeit. Für die Kostenfunktion, welche den Partitionierungsverfahren zu Grunde liegt, sind folgende Parameter relevant:

- Anzahl der parallel rekonfigurierbaren Flächen (#RA),
- Wortbreite der Verbindungen zwischen den rekonfigurierbaren HW-Modulen.

Auf Grund der funktionalen Abhängigkeiten eines Entwurfs, ist eine Partitionierung, welche äquivalente HW-Module als Zielfunktion hat, nur mit sehr großem Aufwand oder gar nicht möglich. Aus diesem Grund ist eine näherungsweise Bestimmung der *Framework*-Parameter für die funktionale Partitionierung ausreichend. Die Synthesedaten für relevante *Framework*-Variationen sind in Kapitel 6 beschrieben.

4.2 Strukturale Netzlistenpartitionierung

Ist eine Partitionierung des Entwurfs unter funktionalen Gesichtspunkten nicht möglich, wird im folgenden Schritt die synthetisierte Netzliste des Entwurfs oder einzelner Module auf strukturaler Ebene partitioniert. Für die Umsetzung wurde ein Verfahren, welches auf dem *Cone*-Algorithmus basiert, an die Anforderungen des neuen Ansatzes angepasst.

Die Emulation eines struktural partitionierten Entwurfs unterscheidet sich wesentlich von der Emulation funktional interagierender Module. Die Hierarchie einer Schaltung wird während der Synthese für die Optimierung weitgehend ausgeflacht. Die einzelnen Schaltelemente sind nach der Synthese untereinander mit Signalleitungen verbunden. Die Gatter, welche auf physikalischer Ebene die logische Abbildung der Schaltung realisieren, propagieren die Signaländerung kontinuierlich auf jede Änderung des Eingangs. Die Laufzeit der Signale durch ein Gatter divergiert und hängt vom Umfang der Schaltfunktion und der technologischen Integration eines Gatters ab. Eine Aneinanderreihung mehrerer Gatter führt dementsprechend zu unterschiedlichen Verzögerungen der Signallaufzeit. Dadurch wird die Schaltfunktion nachfolgender und paralleler Gatter beeinflusst. Zur Vermeidung falscher Schaltvorgänge werden zwischen die Signalpfade der Gatter getaktete Speicherelemente geschaltet, welche eine Synchronisierung der Gatter realisieren. Jedes getaktete Element der Netzliste (*FlipFlop, Register*) hängt direkt an einer Taktleitung (*Clock*). Der Taktgeber synchronisiert dadurch die gesamte Schaltung. Auf eine Taktflanke propagiert das getaktete Element den Speicherinhalt an den Ausgang. Dadurch reduziert sich der Einfluss der Signalverzögerung auf die ungetakteten Gatter zwischen den Speicherelementen. Die Menge der Gatter zwischen getakteten Elementen wird im Folgenden als kombinatorischer Knoten bezeichnet (vgl. Abb. 4.4).

Die Information in der gesamten Schaltung wird zu jedem Takt zwischen den Speicherelementen propagiert. Die Ausbreitung beginnt an den stimulierten Eingängen der Schaltung und wird zur gesamten Laufzeit der Schaltung ungerichtet durch alle angeschlossenen Schaltelemente entsprechend dem Datenfluss weitergeführt. Für eine fehlerfreie Emulation der Schaltung müssen die Signalübergänge durch den Emulator abgebildet werden.

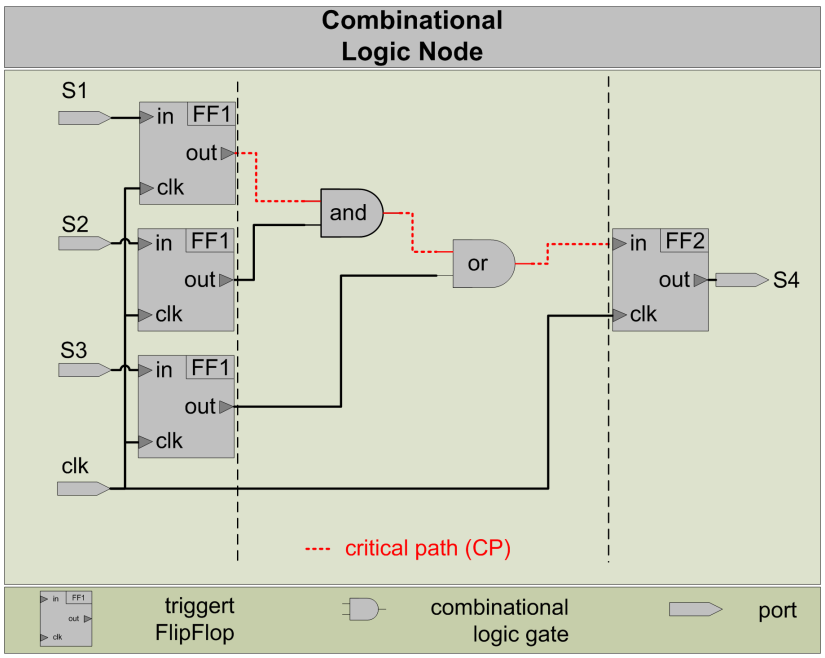


Abbildung 4.4: Kombinatorische Logik

Für die Umsetzung ist es notwendig, die synthetisierte Netzliste so zu partitionieren, dass die resultierenden HW-Module mit einer geringen Anzahl an Rekonfigurationen effektiv die Schaltfunktion abbilden. Demgegenüber steht die Anforderung einer sequentiellen Schaltung, zu jedem Takt die Ausgangswerte unabhängig von deren Gültigkeit und Zustand zu propagieren.

Der dieser Arbeit zu Grunde liegende Algorithmus für die strukturale Partitionierung einer synthetisierten Netzliste eines Entwurfs basiert auf einem nicht kontextabhängigen Verfahren, d.h. Informationen, welche sich durch eine Datenflussanalyse (DFG - Data Flow Graph) des Entwurfs gewinnen lassen, werden auf Grund unterschiedlicher Ergebnisse für verschiedene Stimulidaten nicht berücksichtigt. Im folgenden Abschnitt werden die für diese Arbeit relevanten Schwerpunkte hervorgehoben und erläutert. Das Partitionierungsverfahren ist in [64] umfassend beschrieben und in [20] wird die Implementierung des Algorithmus detailliert erläutert und bewertet. Das implementierte Verfahren zu Bildung der HW-Module basiert auf den folgenden drei Schritten:

- Die Partitionierung auf Gatterebene basiert zunächst auf der Zerlegung in die kleinsten kombinatorischen Knoten der Netzliste (vgl. Abb. 4.5).
- Diese Knoten werden im Anschluss zu komplexeren Konstrukten (*Cluster*) zusammengefasst.
- Aus den *Clustern* werden durch eine Synthese die HW-Module generiert.

Ein kombinatorischer Knoten besteht im Gegensatz zu einem *Cluster* einzig aus verbundenen Schaltelementen ohne einem im Signalverlauf platzierten Speicherelement (*Register, FlipFlop* bzw. *Latch*), d.h. die Latenz eines kombinatorischen Knoten ist bestimmt durch die Summe der Gatterlaufzeiten des längsten

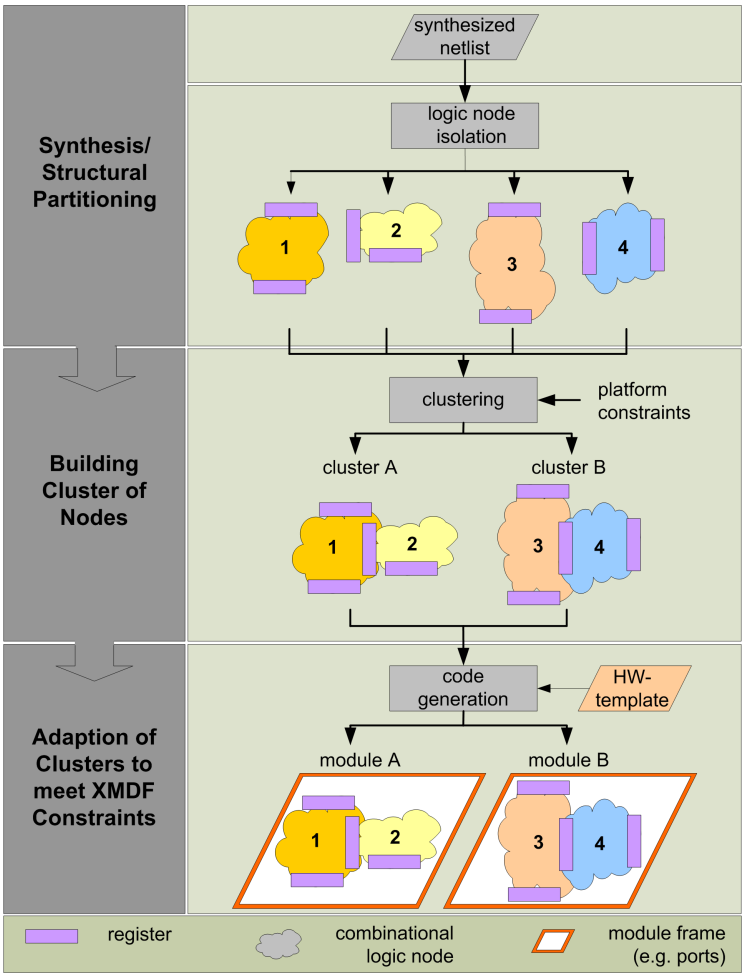


Abbildung 4.5: Strukturaler Ablauf für die Partitionierung

Pfades im Knoten. Die Selektierung der Gatter erfolgt durch die Anwendung eines Algorithmus, welcher auf der Basis eines modifizierten *Cone*-Algorithmus implementiert wurde. Basierend auf einer Analyse der Netzliste folgt der Algorithmus vom Ausgang eines Gatters beginnend den Signalpfaden. Für die Analyse der Netzliste wird die Software *Leonardo Spectrum* eingesetzt. Dieses Synthesewerkzeug lässt sich unter Verwendung von Skripten steuern. Im umgesetzten Verfahren für die strukturale Partitionierung wird die Software durch den TCL-Interpreter (TCL - Tool Command Language) manipuliert. Folgende Schritte sind für die strukturale Partitionierung durchzuführen:

- Zunächst wird die Verhaltensbeschreibung des Entwurfs synthetisiert. Enthaltene *Tristate*-Logik kann auf Grund des bidirektionalen Charakters nicht eindeutig einer Signalflussrichtung zugeordnet werden. Aus diesem Grund werden Konstrukte, welche *Tristate*-Logik enthalten, während der Synthese durch multiplexergesteuerte Logikelemente ersetzt [64]. Dieser Schritt ist für die Anwendung des Partitionierungsverfahrens notwendig, hat jedoch auf das zu verifizierende Systemverhalten keinen Einfluss. Die Steuerung des Multiplexers ist abhängig von der geforderten Signalflussrichtung

und wird direkt mit der Logik verbunden, welche an die Leitung angeschlossen ist. Das Synthesewerkzeug unterstützt diesen Schritt durch Aktivierung eines entsprechenden Schalters.

- Im Anschluss erfolgt eine Analyse der Netzliste hinsichtlich aller enthaltenen Elemente. Im Zuge der Netzlistenanalyse wird die Datenbasis generiert, welche dem Algorithmus zu Grunde liegt. Die Datenbasis enthält die Gatter und deren Verbindungsleitungen, welche die Schaltfunktion des Entwurfs repräsentieren.
- An die Analyse schließt sich ein Suchalgorithmus an, welcher selektierten Modulen *Alias*-Namen zuordnet (vgl. Tab. 4.1). Der Aufbau der Netzliste und die zugehörigen Ein- und Ausgänge werden in einem Datenkonstrukt aus *String*-Variablen abgelegt. Dadurch kann die Datenbasis in andere Verarbeitungsprogramme konvertiert werden. Vorteil der programmübergreifenden Datenbasis ist eine Verarbeitung der Netzlistendaten über die Funktionalität der TCL-Skriptensprache hinaus.
- Nach der Überführung der Netzliste in die Datenbasis wird mit dem Partitionierungsalgorithmus begonnen.

***Alias*-Parameter**	**Zuordnung der Parameter zu Netzlisten-Elementen**
DsgnName	*Ttop_Level* Identifikation
clk_Lst	Liste aller Takteingänge
CRoot_Lst	Liste aller Pin- und Registereingänge
connection_Lst	Liste aller Verbindungen in der Netzliste (1. Element ist ein treibender Eingang)
CmpInst_Lst	Matrix für die Zuordnung der *Alias*-Namen zur Zell-Komponente
IO2Inst_Lst	Matrix für die Zuordnung der Ein- und Ausgangsport zu den Zell-Komponenten
LuTFkt_Lst	Liste der Boolschen Funktion, welche als LuT abgebildet sind

Tabelle 4.1: Typ-Konventionen für Übergabe der Netzlisten-Parameter

Der Partitionierungsalgorithmus ist aus Gründen der Performance in C/C++ programmiert. Untersuchungen während der Implementierung zeigten, dass eine Umsetzung des Algorithmus in TCL für eine Netzliste mit 800 Elementen ca. 30 Minuten benötigte. Eine äquivalente Implementierung des Algorithmus in C/C++ reduzierte den Zeitaufwand auf wenige Sekunden. Nachteil der Konvertierung und Verarbeitung in C/C++ ist der zusätzliche Speicherbedarf für die gesamte Abbildung der Netzlisteninformationen in eine zweite Datenbasis, welche notwendig ist, um die für den TCL-Interpreter bestehende Datenbasis nicht zu verändern. Dieser Aufwand tritt bei einer internen Verarbeitung und Anwendung des Algorithmus

für die Partitionierung der Netzliste im Synthesewerkzeug mit TCL-Kommandos nicht auf. Der Nachteil des Speicherbedarfs wird durch die Weiterverarbeitung der Daten in einer Hochsprache relativiert und der Einsatz unterschiedlicher Software, welche nicht über ein TCL-Interface verfügt (z.B. Matlab, Mathcad), möglich. Für die Implementierung der Partitionierung in C/C++ wird auf eine umfangreiche Bibliothek von verfügbaren Funktionen zurückgegriffen. Neben dem Einsatz optimierter Suchalgorithmen wird eine dynamische Datenstruktur generiert, welche die selektierten Knoten zusammenfasst. Durch diese Struktur ist eine effiziente Bewertung der kombinatorischen Knoten hinsichtlich der Anzahl enthaltener Gatter, Leitungen und Latenzzeiten möglich. Während der Bildung der *Clusters* wird diese Bewertung iterativ auf die möglichen Knotenkombinationen angewendet. Durch den Einsatz mathematisch optimierter Algorithmen, welche eine Bibliothek in C/C++ zur Verfügung stellt, reduziert sich der erforderliche Zeitaufwand signifikant.

Der Ablauf des Algorithmus für die Partitionierung der Netzliste besteht aus folgenden drei Schritten:

1. Zuordnung der Treiberkegel[1] für jedes Element der Netzliste (vgl. 2.14):
   Während der Zuordnung wird ausgehend von einem Ausgang jedes Element der Netzliste, welches durch eine Verbindung direkt oder über ein weiteres Element indirekt zu diesem Ausgang verbunden ist, in eine Liste aufgenommen. Dadurch entstehen die für den *Cone*-Algorithmus typischen Kegel (vgl. Abbildung 2.14). Konträr zur ursprünglichen Implementierung des *Cone*-Algorithmus ist auf Grund der größeren Relevanz für den Einfluss der Schaltaktivität an den Ausgängen eines Entwurfs für diese Arbeit ein inverser Suchlauf implementiert worden, d.h. die Suche beginnt nicht an den Eingängen des Entwurfs, sondern an dessen Ausgängen. Die Signalrichtung ist demgegenüber unverändert vom treibenden Element zum getriebenen Element. Nach der Erfassung aller in der Netzliste enthaltenen Elemente in einer Liste ist es möglich, alle Verbindungen in den Suchalgorithmus einzubeziehen. Wenn ein Element detektiert wird, welches bereits als Treiber eines weiteren Kegels in der Liste *CRoot* eingetragen wurde, wird der entsprechende Kegel sofort in die Liste des aktuellen Treiberbaums hinzugefügt. Dadurch reduziert sich in dicht vermaschten Netzwerken mit einem großen Anteil an kombinatorischer Logik die Anzahl der Suchläufe. In [20] ist der iterative Algorithmus für die Bestimmung der Kegelelemente unter Beachtung kombinatorischer Logikteile abgebildet. Das Ergebnis des ersten Schrittes der Partitionierung ist eine Liste aller möglichen Kegelstrukturen innerhalb der Netzliste und eine weitere Liste, welche den Kegeln die Treiberelemente also Eingang oder Registerausgang zuordnet.

2. Erweiterung der *CRoot*-Liste für Elemente, welche Verbindungen in andere Kegel der Netzliste haben:
   Um während der Bildung der *Cluster* eine funktionale Diskrepanz durch die Trennung abhängiger Kegel zu verhindern, werden diese Kegel im zweiten Schritt selektiert. Dafür iteriert der Algorithmus über die Liste der Kegelstrukturen, und wählt die Elemente aus, welche Ressourcen mit anderen Kegeln teilen (vgl. Abb. 2.15). Kegelstrukturen, welche keine Elemente mit anderen Kegeln teilen, können unter Berücksichtigung der Verdrahtung in jedem *Cluster* platziert werden. Dieser Schritt der Partitionierung bedingt auf Grund der iterativen Suche einen großen Berechnungsaufwand.

3. Bildung der *Cluster* für die kombinierbaren Elemente:
   Während der Bildung der *Cluster* werden die Elemente ausgewählter Kegel zusammengefasst. Un-

[1] Kegel die direkt an einem Eingang der Schaltung beginnen

ter Beachtung der Abhängigkeiten zwischen den Kegeln und den Randbedingungen des Emulator-*Frameworks*, fasst der Algorithmus geeignete Kegel zusammen. Dafür iteriert der Algorithmus über alle eingetragenen Elemente in der Liste die in Schritt zwei generiert wurde. In Abbildung 4.6 ist

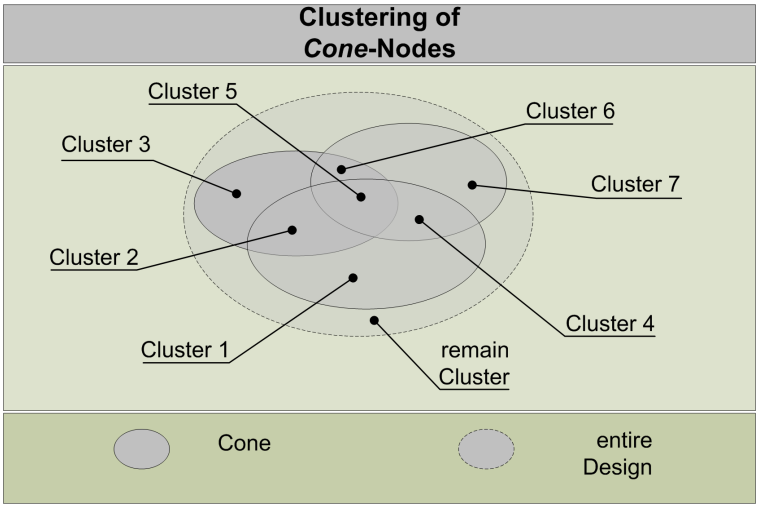


Abbildung 4.6: Bildung komplementärer *Cluster*

die Zusammenfassung der Schnittmenge von drei Kegeln abgebildet. Im dargestellten Fall lassen sich alle komplementären Kegel zu *Clustern* verbinden. Knoten, die keinem direkten Kegel zugeordnet werden, bilden ein separates *Cluster* (*remain Cluster*). In diesem *Cluster* ist hauptsächlich die Treiberlogik der Ein- und Ausgänge des Entwurfs enthalten. Dadurch stellt dieses *Cluster* die Verbindung der internen Logikressourcen mit der Umgebung her. Für eine Reduzierung der Rekonfigurationen, welche für dieses Modul zwingend in jedem Takt notwendig sind, ist eine statische Platzierung des HW-Moduls im Emulator-*Framework* vorgesehen.

Der Algorithmus wird in der Arbeit für zwei Implementierungen einer strukturalen Entwurfspartitionierung mit unterschiedlichen Zielfunktionen eingesetzt. Für eine effiziente Realisierung unter Berücksichtigung der beschriebenen konträren Anforderung ist der Einsatz von Speicherelementen und einem angepassten Netzwerk für die Kommunikation zwischen den Elementen notwendig. Diese Implementierung setzt für eine ausreichende Performanz der Emulation eine sehr effiziente Speicherschnittstelle voraus.

In dem zu Grunde liegenden Emulatorsystem (vgl. Abs. 3.3) ist ein breitbandiger *QDR*-Speicher integriert. Dieser ermöglicht zwei unabhängige Schreibe- und Lesezyklen pro Takt des Emulators. Die HW-Module schreiben nach jedem erfolgten Taktzyklus die propagierten Ausgangswerte an eine von Steuerwerk des Emulators zugewiesene Speicherschnittstelle. Im folgenden Takt des Emulators wird mit diesem Wert das im Datenfluss folgende Element stimuliert. In dieser Umsetzung ist das Ziel für die Partitionierung die Generierung von HW-Modulen mit gleichen Parametern wie z.B. die Anzahl der Gatter oder die Wortbreite der Schnittstellen. Divergieren die Gatter bzgl. der Parameter untereinander sehr weit, entsteht für die Implementierung und Effizienz des Emulator-*Frameworks* eine Reduzierung an nutzbaren Ressourcen. Der Grad der Reduzierung richtet sich nach der Abweichung der Parameter des größten und damit für die Dimensionierung des *Frameworks* zu Grunde liegenden HW-Moduls zum Ressourcenverbrauch des kleinsten Moduls.

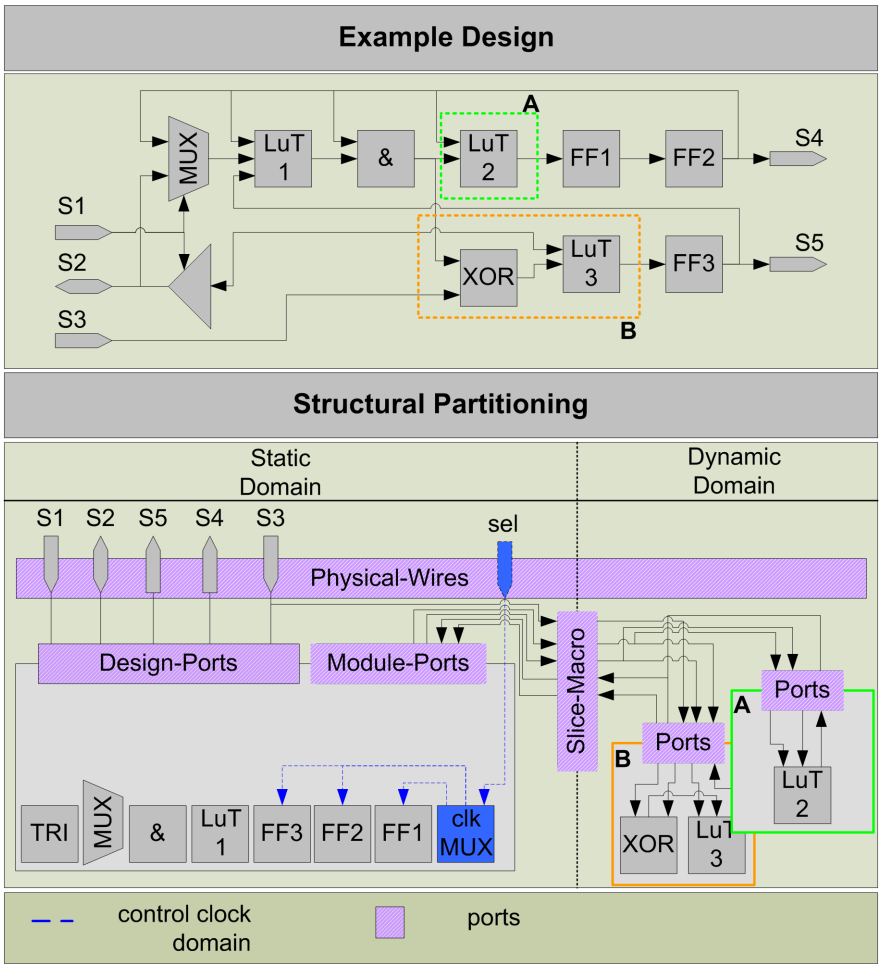


Abbildung 4.7: Strukturale Partitionierung Version A

Eine weitere untersuchte Realisierungsmöglichkeit ist die Aufteilung der Netzliste in kombinatorische und sequentielle Elemente. Nach der Partitionierung werden die sequentiellen Elemente während der Emulation im statischen Teil des Emulators platziert. Dadurch ist der Erhalt der internen Zustände der Schaltungsregister während den Rekonfigurationsphasen gewährleistet. Die kombinatorische Logik wird unter Berücksichtigung des Datenflusses zu HW-Modulen zusammengefasst und während der Emulation dynamisch rekonfiguriert.

In Abbildung 4.7 ist die Aufteilung eines Entwurfs in kombinatorische Logik und den entsprechenden sequentiellen Logikbausteinen abgebildet. Der Datenfluss teilt sich in zwei unabhängige Zweige *A* und *B* auf. Das *&*-Gatter und das *LuT1*-Element sind im gemeinsamen Teil des gesamten Datenflusses platziert. Während die Gatter der Signalpfade *A* und *B* zu separaten HW-Module zusammengefasst werden, ist die Auslagerung des *LuT1*-Elements und des *&*-Gatters in ein rekonfigurierbares HW-Modul auf Grund der fehlenden eindeutigen Zuordnung zu einem Signalpfad nicht möglich.

Dieses Verfahren für die strukturale Partitionierung führt für große kombinatorische Logikkonstrukte zu

effizienten Ergebnissen. Enthält die Netzliste einen großen Anteil an sequentieller Logik, entsteht während der Partitionierung ein dominierendes statisches Modul. Gleichzeitig reduziert sich die für die Rekonfiguration zur Verfügung stehende Logikfläche.

4.3 Anpassung der Designpartitionen an den XMDF

Für eine Integration der HW-Module in einem partiell rekonfigurierbaren System ist die Beachtung der durch die Plattform vorgegeben Randbedingungen notwendig [76]. Die Makros sind für jedes eingesetzte FPGA unterschiedlich und werden vom Hersteller der FPGAs bereitgestellt. Inhalt der Makros (z.B. *BusMacro* bzw. *SliceMakro* für *Xilinx* FPGAs) ist die Abbildung einer dedizierten Schnittstelle für die Verbindungsleitungen, welche während der Rekonfiguration unterbrochen werden. Die Makros werden als HDL-Beschreibung im Entwurf oder als synthetisierte Netzlistenelemente an dessen Schnittstellen angebunden.

Für die Platzierung der Makros gelten unterschiedliche geometrische Anforderungen [102]. Während die *Virtex*-FPGA der *I, -II* und *-II-Pro* Serie spaltenweise rekonfigurierbar sind, werden *Virtex4-* und *5-*FPGA sowohl spalten- als auch zeilenweise rekonfiguriert (vgl. Abb. 3.2). Auf Grund der Verwendung eines *Virtex4*-FPGAs für die *Framework*-Implementierungen (vgl. Abs. 3.3) kommen *SliceMakro* zum Einsatz. Diese unterscheiden linke und rechte Seite eines rekonfigurierbaren Moduls sowie Schnittstellen an der Ober- und Unterseite. Für die Platzierung wird das entsprechende Makro ausgewählt und mit den Schnittstellen des Entwurfs verbunden [75]. Die geometrische Ausrichtung erfolgt durch die Auswahl des entsprechenden Makros und der Generierung der .ucf-Datei, in welcher die Randbedingungen der Plattform und die für die Platzierung notwendige Angabe der zu verwendenden FPGA-Ressourcen angegeben werden. In [64] ist die Auswahl und die Strategie für die Platzierung der *SliceMacro* erläutert.

Die Platzierung der Makros erfolgt auf Grund der unterschiedlichen Ebenen, in denen die Partitionierung durchgeführt wird, sowohl in der Verhaltensbeschreibung des Entwurfs als auch durch Einfügen synthetisierter Elemente in die Netzliste [43].

Für die Implementierung der Makroelemente in die Verhaltensbeschreibung des Entwurfs wird auf den *Code*-Generator des *Regatta-Frameworks* zurückgegriffen. Zunächst werden die Makros in das *Rule Key* Format transformiert und im Anschluss an den entsprechenden Stellen eingefügt. Die Einfügemarken sind durch die Schlüsselwörter der Schnittstellen gekennzeichnet. Nach einer Rücktransformation des Schlüsselbaums sind die HW-Module um die Makros erweitert.

Der *Parser* des *Regatta-Frameworks* unterstützt im derzeitigen Stand teilweise die Beschreibungssprache *VHDL. Verilog* wird vom *Parser* nicht unterstützt. Aus diesem Grund ist die Implementierung in der Beschreibungsebene nicht vollständig automatisiert umgesetzt. Nach der Erweiterung der HW-Module erfüllen diese die Anforderungen des Syntheseablaufs und genügen den Randbedingungen des XMDF. Zusammen mit dem Emulator-*Framework* als statischem Modul werden die HW-Module synthetisiert.

Die Implementierung der Makros in die struktural partitionierte Netzliste erfolgt teilautomatisiert. Die Elemente werden zunächst synthetisiert und im Anschluss in die Netzliste eingefügt. Dafür sind in der Datenbasis die entsprechenden *Alias*-Namen der Schnittstellen und der aufgetrennten Leitungen separat abgelegt. Abschließend werden die Anschlüsse der Makros mit den ausgewählten Signalleitungen der HW-Module verbunden, um den Anforderungen des XMDF zu genügen.

4.4 Bewertung des implementierten Verfahrens

Ausgehend von einer Beschreibung des Entwurfs in *VHDL* oder *Verilog* ist mit dem Einsatz des standardisierten Entwicklungswerkzeugs *HDL-Designer* ein einheitlicher Einstieg für alle HDL-Beschreibungen gegeben. Die funktionale Partitionierung des zu verifizierenden Entwurfs in der Beschreibungsebene ermöglicht unter Ausnutzung einer strukturierten Beschreibung die Selektion der HW-Module des Entwurfs. Der Einsatz kommerzieller Werkzeuge für die notwendige Analyse sichert die Integration des Ansatzes und reduziert durch die lange Entwicklungsphase des Werkzeugs die Anzahl der Fehler.

Nach der Auftrennung des Entwurfs findet eine Kostenbewertung der enthaltenden Module hinsichtlich der Anzahl der Gatter und der enthaltenen Leitungen statt. Diese Bewertung ist in der Beschreibungsebene des Entwurfs nicht realisierbar. Aus diesem Grund werden die Module in einem Syntheseschritt auf die Gatterebene abgebildet. Ausgehend von Richtwerten, die durch das Emulator-*Framework* vorgegeben sind, werden die Module mit den bestehenden Kosten akzeptiert oder wenn möglich iterativ weiter partitioniert.

Dieser Algorithmus hat auf Grund der notwendigen Syntheseschritte einen hohen Zeitbedarf. Des Weiteren führt die Anforderung, den Entwurf hinsichtlich gleicher Kosten der Module zu partitionieren zu einer großen Anzahl an Iterationen. Der konkrete Aufwand hängt direkt von der umgesetzten Granularität des entworfenen Systems und der geforderten Granularität der HW-Module ab. Unter Berücksichtigung regulärer Architekturen reduziert sich des Weiteren signifikant der Aufwand für die funktionale Partitionierung. Nach der Selektion der multiplen Instanzen wird nur ein Modul für die Synthese herangezogen. Dadurch verringert sich die Zeit für die Abarbeitung des Algorithmus zur Bestimmung der Modulkosten. Des Weiteren entfällt der iterative Annährungsprozess zur Bestimmung der Kosten für zur Selektion äquivalenter Module. Auf Grund der Partitionierung der Systembeschreibung ist die Anpassung der einzelnen Partitionen an die Anforderungen des XMDF mit geringem Aufwand realisierbar.

Der Algorithmus für die strukturale Partitionierung auf Gatterebene, welcher in [20] beschrieben ist, basiert auf der Bildung von Clustern mit äquivalenten Kosten. Die Performanz des implementierten Verfahrens hängt direkt vom Aufwand für die Generierung der für die Bewertung der Kosten notwendigen Datenbasis ab. Durch die Auflistung der Netzliste in einer angepassten Datenbasis werden die für den Partitionierungsprozess notwendigen Eigenschaften zusammengefasst. Dabei werden abhängige und dadurch kombinierbare Knoten markiert. Diese angepasste Untermenge der gesamten Datenbasis reduziert den Aufwand für die Suche passender Knoten erheblich. Enthält der zu partitionierende Entwurf eine große Anzahl an Knoten (<1000), potenziert sich die Zeit für die *Clusterung* der Knoten. Dieser Prozess führt ohne die optimierte Suche der relevanten Knoten zur Erreichung der Lösbarkeitsgrenze (vgl. *3.2.2 NP-hardness*).

Zur Vermeidung eines ungeordneten und dadurch zeitaufwändigen Suchprozesses werden die dem Synthesewerkzeug zu Grunde liegenden Informationen über den Datenfluss mit einbezogen. Die Daten werden vom Synthesewerkzeug während der Optimierung generiert. Im Laufe des Optimierungsschritts verschiebt der Algorithmus die sequentiellen Logikbausteine innerhalb des kombinatorischen Datenpfads. Unter Berücksichtigung der Taktabhängigkeiten, welche zwischen den Gattern bestehen, werden zur Synchronisierung zusätzliche Register eingefügt. Dieses Verfahren wird als *re-timing* [71] bezeichnet. Durch *re-timing* entstehen kombinatorische Knoten, welche durch äquivalente Laufzeiten zwischen den Speicherelementen (vgl. 4.4) charakterisiert sind. Unter Ausnutzung der dadurch entstehenden *Pipeline*-Struktur und durch eine effiziente Aufteilung des zu verifizierenden Entwurfs in die durch die sequentielle Logik vorgegebenen Stufen der *Pipeline* ist eine deutliche Reduzierung des Partitionierungsaufwands und der Anzahl der Rekonfigurationen erreichbar. Die Effizienz dieses Verfahren hängt vom zu testenden Entwurf ab. Unter

Berücksichtigung der Ergebnisse des Optimierungsschritts ist die Auswahl einer freien *Clusterung* bzw. der *Clusterung* der *Pipeline*-Elemente möglich. Für Entwürfe, welche einem gerichteten Datenfluss folgen, ist die Effizienz der *Pipeline-Clusterung* am größten. Für reaktive Systeme mit einem rückgekoppelten Datenfluss erreicht die freie *Clusterung* bessere Ergebnisse (vgl. Kapitel 6).

4.5 Zusammenfassung

Der Algorithmus für die Partitionierung des Entwurfs wurde in zwei Abstraktionsebenen umgesetzt. Zunächst ist eine Partitionierung unter funktionalen Gesichtspunkten vorgesehen. Dieser Algorithmus basiert auf der Auswertung der Entwurfsbeschreibung vor der Synthese. Für die Ermittlung der Modulkosten wird im Anschluss eine Synthese durchgeführt. Die Selektierung der Module ist durch ein standardisiertes Entwurfswerkzeug realisiert. Dadurch wird zum einen eine Integration des Ansatzes in bestehende Abläufe erleichtert, zum anderen werden Fehler durch Eigenentwicklungen vermieden.

Erfüllt ein Modul oder der gesamte Entwurf nicht die durch das Emulator-*Framework* vorgegebenen Randbedingungen, wird im zweiten Schritt der Partitionierung ein modifiziertes Verfahren, welches auf einer Strukturpartitionierung basiert eingesetzt. Das Verfahren ist vom *Cone*-Algorithmus abgeleitet. Im ersten Schritt des Verfahrens wird der Entwurf oder das zu partitionierende Modul synthetisiert und analysiert. Dazu werden die Gatter der Netzliste und die Verbindungsleitungen in eine angepasste Datenbasis überführt. Aus dieser selektiert der Algorithmus im zweiten Schritt die kombinatorischen Knoten der Netzliste. Im dritten Schritt erfolgt eine *Cluster*-Bildung kombinierbarer Knoten unter Berücksichtigung der Randbedingungen des Emulator-*Framework*s.

Unter Anwendung des Algorithmus für die strukturale Partitionierung auf Gatterebene wurden zwei unterschiedliche Lösungsansätze umgesetzt. In der ersten Implementierung wird eine Aufteilung der Netzliste in kombinatorische und sequentielle Elemente angestrebt. Die sequentiellen Elemente der Netzliste werden im Anschluss im statischen Teil des Emulators platziert. Die kombinatorischen Elemente bilden unter Berücksichtigung des Datenflusses dynamisch platzierbare HW-Module. Auf Grund der Implementierungsgröße der sequentiellen Elemente wird der statische Teil in dieser Umsetzung deutlich größer als der kombinatorische Teil. Das führt zu einer Reduzierung der Effizienz des gesamten Ansatzes. Ein Vorteil dieser Lösung ist die statische Platzierung der speichernden Elemente und dadurch eine signifikante Reduzierung der Rekonfigurationszeiten.

In der zweiten Implementierung wird die *Clusterung* der Knoten unabhängig von der Art des Knotens durchgeführt. Ziel der Zusammenfassung von Knoten in diesem Verfahren ist die Generierung von *Clustern* mit äquivalenten Kosten. Auf Grund der Ausrichtung des Emulator-*Frameworks* am größten HW-Modul, werden Kosten für dessen Generierung im Falle unterschiedlich großer HW-Module reduziert.

Eine weitere Optimierung ist durch die Einbeziehung der Daten während des Optimierungsschrittes der Netzliste durch die Synthesewerkzeuge gegeben. Dabei wird das *Re-timing*-Verfahren für die Ausbalancierung der *Pipeline*-Stufen eingesetzt, welches in allen Synthesewerkzeugen implementiert ist. Der Einsatz dieses Verfahrens wurde in der vorliegenden Arbeit theoretisch in der Betrachtung der Modulsteuerung untersucht.

Die partitionierten Teile des zu testenden Entwurfs werden im Anschluss um Schnittstellen für das FPGA erweitert. Dieser Vorgang wird im Falle der funktionalen Partitionierung in der Entwurfsebene durchgeführt. Für die Untersuchung zur Automatisierung dieses Vorgangs wurde ein im Institut entwickeltes *Framework* zur statischen *Code*-Analyse einbezogen. Durch eine Erweiterung des *Regatta-Parsers* um

eine vollständige Sprachbibliothek für *VHDL* und *Verilog* ist eine umfassende Automatisierung der funktionalen Partitionierung und der Anpassung der Module an den XMDF zu erwarten. Im derzeitigen Stand ist es möglich, das *Regatta-Framework* in begrenztem Umfang für die Generierung der Module einzusetzen. Die Platzierung der Schnittstellen in die partitionierte Netzliste ist teilautomatisiert umgesetzt.

Zusammenfassend ist die Partitionierung des Entwurfs auf unterschiedlichen Ebenen realisierbar. Die Anforderungen, welche durch den generellen Ansatz gefordert sind, werden durch die implementierten Algorithmen der Partitionierung erreicht.

5 Steuerung der Entwurfs-Partitionen

Das beschriebene Emulationsverfahren basiert auf der Zerlegung des zu verifizierenden Entwurfs in separate HW-Module. Die HW-Module werden nach der Synthese auf ein rekonfigurierbares FPGA geladen und in funktionaler und zeitlicher Abfolge emuliert. Inhalt des folgenden Kapitels ist die Beschreibung der für die zeitliche und funktionale Steuerung (schedule) notwendigen Umgebung.

Im ersten Abschnitt des Kapitels wird eine allgemeine Einordnung des Ansatzes in Verfahren zur Steuerung von pRTR-Entwürfen vorgenommen. Im zweiten Teil wird der Algorithmus zur Berechnung der Modulfolge und dessen Aufteilung in einen Software- und Hardwareteil erläutert. Die Funktionsweise des auf dem *Virtex4*-FPGA implementierten Emulator-*Frameworks* und dessen Zusammenspiel mit der Steuerung ist in Abschnitt 3 dargestellt.

5.1 Einordnung des implementierten Verfahrens

Das in der Arbeit umgesetzte Verfahren zur Steuerung rekonfigurierbarer HW-Module wertet die Aktivitäten an den Ausgängen der HW-Module aus [109]. Daraufhin berechnet der Algorithmus an Hand der bereits konfigurierten HW-Module und des freien Platzes auf dem FPGA die zeitliche Abfolge für die Rekonfiguration. Aus dem Algorithmus für die Partitionierung ist der Zusammenhang bekannt, in welcher funktionalen Abfolge die HW-Module auf dem FPGA rekonfiguriert werden müssen.

Die Anforderungen an die Steuerung divergieren zum einen auf Grund der geforderten taktgenauen Reaktion auf die Ereignisse an den Ausgängen der HW-Module und zum anderen auf Grund eines großen Grades an Flexibilität für die Implementierung des Steueralgorithmus.

Die Umsetzung des Steuerwerks wurde aus diesem Grund in einen Hardware- und in einen angepassten Softwareteil untergliedert. Die Steuerung der HW-Module erfolgt zentral und ereignisbedingt. Für eine Optimierung des zusätzlichen Ressourcenbedarfs wurde eine auf dem Emulator-FPGA implementierte Laufzeitumgebung realisiert. Dieses Emulator-*Framework* besteht aus einem flexiblen Kommunikationsnetzwerk und den für die Steuerung notwendigen *Controllern.* Durch die direkte Umsetzung des *Frameworks* auf dem FPGA ist die geforderte taktgenaue Steuerung der HW-Module realisierbar. Das Kommunikationsnetzwerk hat die Aufgabe, den Aufwand für die notwendigen Rekonfigurationen zu reduzieren. Die für die Steuerung des Emulator-*Frameworks* notwendige Software ist auf einem *PowerPC* implementiert. Der als dedizierter IC integrierter *PowerPC* auf dem *VirtexII*-FPGA ermöglicht eine direkte Beeinflussung der Gatter, ohne die Logikressourcen des FPGAs für zusätzliche Steuer-*Controller* zu reduzieren.

5.2 Hardware/Software Kopplung

Die Aufteilung des umgesetzten Verfahrens in einen Hardware- und einen Softwareteil folgt den Anforderungen einer verteilten Steuerung [48]. Dadurch ist eine Entkopplung von den Berechnungen des Ablaufplans der Emulation von der direkten ereignisbasierten Steuerung der HW-Module gewährleistet. Durch

diese Aufteilung ist des Weiteren die Zusammenfassung mehrerer Emulator-FPGAs in einem System möglich.

In Abbildung 3.9 wird die gesamte Entwicklungsumgebung schematisch dargestellt. Durch die Aufteilung der Plattform in einen Kommunikations-FPGA (*Virtex-II-Pro*) und ein separates Emulator-FPGA (*Virtex4*) sind der Hardwareteil und der Softwareteil zudem auf zwei unterschiedlichen FPGAs implementiert. Das *Virtex-II-Pro*-FPGA ist die Kommunikationsschnittstelle zwischen dem *Host*-Prozessor und der gesamten Entwicklungshardware. Unter Ausnutzung des *PowerPCs* für die Implementierung der Steuersoftware des Emulators wird eine direkte Verbindung zum *Host*-System und zum Emulator-FPGA erreicht. Der auf dem *PowerPC* implementierte *Schedule*-Algorithmus steuert die Laufzeitumgebung auf dem Emulator-FPGA (vgl. Abb. 5.1).

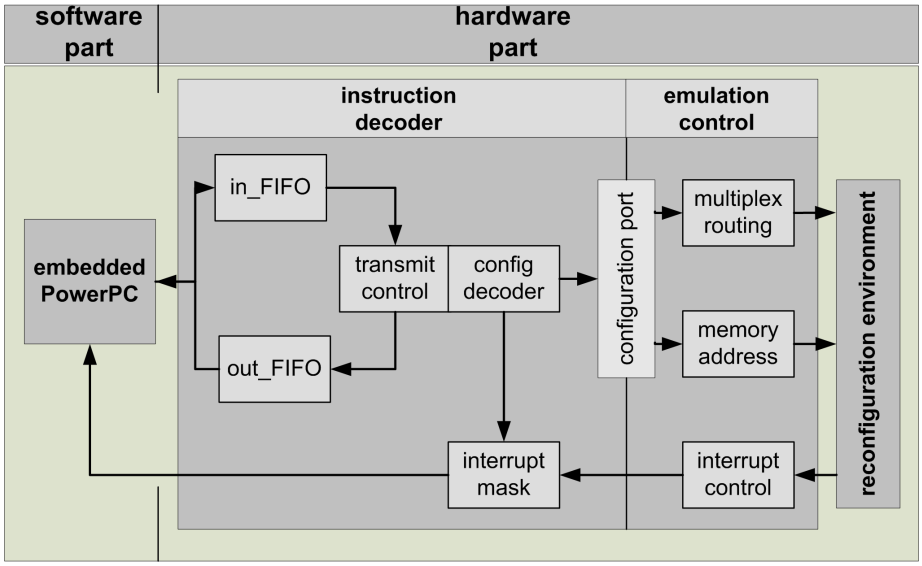


Abbildung 5.1: Hardware/Software Kopplung

Um eine Verringerung der Effizienz des signifikant schnelleren *Virtex4*-FPGAs durch den langsameren *VirtexII-Pro*-FPGA zu verhindern, wurde ein asynchrones Protokoll zwischen den FPGAs umgesetzt. Für die Verarbeitung der Protokolldaten zwischen den FPGAs und für die direkte Steuerung des Emulator-*Frameworks* ist auf dem *Virtex4*-FPGA ein *Controller* implementiert, welcher direkt durch den *PowerPC* programmiert wird.

In Abbildung 5.1 ist das Steuerwerk des *Controllers* in Verbindung mit der Kopplung zum *PowerPC* und zum Rekonfigurations-*Framework* abgebildet. Der *Controller* verarbeitet die Befehle der Software weiter und setzt diese in komplexe Befehlssequenzen zur Steuerung des Emulators um.

- Zunächst werden die eingehenden Befehlssequenzen in einem FIFO-Speicher abgelegt. Das Kommunikationsmodul (*transmit control*) steuert abhängig vom Verarbeitungsstand des *Frameworks* den Eingangspuffer (*in_FIFO*) und den Ausgangspuffer (*out_FIFO*).

- Der Instruktionskoder (*config coder*) setzt im Anschluss die eingehenden Befehle in komplexe Steuersequenzen für das Steuerwerk des Emulators um. Eine Sequenz entspricht der Programmierung

des gesamten Emulator-*Frameworks* unter der Berücksichtigung vollständiger Emulationszyklen.

Dafür ist die Einbeziehung des gesamten Steuerwerks notwendig. Neben dem Instruktionskoder gehören ein *multiplex routing Controller* und ein *memory address Controller* zum Steuerwerk.

- Mit dem *multiplex routing Controller* werden die Multiplexer für die lokalen Verbindungen innerhalb eines rekonfigurierbaren Knotens (*reconfiguration node*) vor Beginn einer Emulation programmiert.
- Für die Allokation und Adressierung der physikalischen QDR-RAM-Speicherbänke ist der *memory address Controller* implementiert. Die Ansteuerung der einzelnen Speicherbänke erfolgt über einen für jede Speicherbank separaten Speicher *Controller* mit einer Wortbreite von 32 Bit in zwei Schreib- und Lesezugriffen. Zusätzlich werden die Schreib- und Lesezyklen in *Burst*-Zugriffen eines vollständigen Datenworts von 64 Bit durchgeführt.

Auf Grund der großen Anzahl an Speicherzugriffen während einer Emulation, welche jedoch wesentlich geringerer Bandbreiten bedingen, reduzieren die Randbedingungen der QDR-Speicher signifikant die Flexibilität des Ansatzes. Aus diesem Grund wurde in der Arbeit ein Verfahren zur Verwaltung virtueller Speicherinstanzen eingeführt. Die detaillierte Umsetzung der Speicherverwaltung ist im Abschnitt 5.3.2 beschrieben.

Durch die konsequente Trennung der zeitlichen Berechnung einer Emulationssequenz und der taktgenauen Steuerung der HW-Module wird ein Ablauf von Emulationen verschiedener HW-Module über einen langen Zeitraum unabhängig von der Verarbeitungsgeschwindigkeit der *Schedule* Software erreicht. Für die Übertragung der Ereignisse und Statusinformationen des Emulator-*Frameworks* kommuniziert das auf dem *Virtex4*-FPGA implementierte Steuerwerk mit dem *PowerPC* ebenfalls über ein asynchrones Protokoll.

In der Implementierung sind 12 programmierbare Sequenzen in der Zustandsmaschine des *Controllers* realisiert, welche vom *PowerPC* gesteuert werden können und 2 Befehlssequenzen für die Kommunikation des Steuerwerks mit dem *PowerPC*.

Sequenz (PPC->Controller)	Funktionalität	Sequenz (Controller->PPC)	Funktionalität
WROPC_GET	Leseanforderung vom PPC	RDOPC_GET_ STATE	schreibt Zustandsinformation an PowerPC
WROPC_SET_ TASL_SEL	selektiert FolgeTask	RDOPC_GET_ TS	Liste der aktuellen Taktzyklen der TASK
WROPC_SET_ TS_MAX	setzt Taktzyklen für ausgewählten TASK		
WROPC_SET_ RT_CONFIG_ DATA_ADDR	setzt Konfiguration für lokale Routing Multiplexer		

Tabelle 5.1: Befehlssequenzen zwischen Software und Hardware [38]

In Tabelle 5.1 sind ausgewählte Befehlssequenzen der Kommunikation zwischen *PowerPC* und *Controller* bzw. in entgegengesetzter Richtung beschrieben. Für die Übertragung des Protokolls wurden 8 Signalleitungen zwischen dem *VirtexII-Pro* und dem *Virtex4* für die Implementierung reserviert (vgl. Abb. 3.9). Für die Steuerung der partiellen Emulationen sind folgende Schritte vom Emulator abzuarbeiten:

- Nachdem der Algorithmus eine Sequenz berechnet hat, schreibt der *PowerPC*, welcher auf dem Bus als *Master* fungiert, das Befehlswort an den Eingangspuffer des Instruktionsdekoders (*instruction decoder* Abb. 5.1). Ein Befehlswort enthält die Kodierung für die Ansteuerung der verschiedenen Sub-*Controller* (*multiplex routing*, *memory address*) und zusätzliche Steuerinformationen für den Instruktionskoder (*config coder*). Diese Informationen geben an, wie oft ein HW-Modul in einem Emulationszyklus verbleibt.
- Die Sub-*Controller* für die Beschaltung der Multiplexer beschalten im Anschluss die lokalen Verbindungen innerhalb der Knoten.
- Danach werden über den PCI-Bus und die angeschlossene ICAP-Konfigurationsschnittstelle die ersten HW-Module (*Tasks*) konfiguriert.
- Nach der Bestätigung der Konfigurationen startet das Steuerwerk die partiellen Emulationen.

Ein *Task* besteht aus einer Gruppe von HW-Modulen (vgl. Abs. 5.4), welche in lokaler Nachbarschaft partitioniert wurden. Unter Ausnutzung der Lokalität von HW-Modulen wird im verfolgten Ansatz eine Reduzierung der Anzahl an Rekonfigurationen erreicht. Des Weiteren wird durch eine angepasste *Routing*-Strategie, welche im folgenden Abschnitt beschrieben ist, der Ressourcenbedarf für das Verbindungsnetzwerk signifikant reduziert, ohne die Performanz der gesamten Emulation zu reduzieren.

5.3 Implementierung der Rekonfigurationsumgebung

5.3.1 Überblick und Anforderungen der Umgebung

Die ressourceneffiziente Umsetzung des neuen Ansatzes setzt auf dem Emulator-FPGA eine Rekonfigurations- und Laufzeitumgebung (vgl. Abb. 5.1 *reconfiguration environment*) voraus. Diese Umgebung besteht aus einem Kommunikationsnetzwerk und der für die Steuerung des Netzwerks notwendigen Hardwarekomponenten. Des Weiteren sind für die Überwachung und Erkennung der Ereigniszustände an den Ausgangsschnittstellen der HW-Module Beobachtungskomponenten platziert. Diese Komponenten (vgl. Abschnitt 5.5) detektieren einen Schreibvorgang und speichern die ersten Datenwörter bis eine Veränderung am Ausgang des HW-Moduls erkannt wird.

Für eine langfristige Nutzung des Ansatzes als funktionales Verifikationsverfahren wurde des Weiteren die Möglichkeit der Integrierung von HW-Monitoren vorgesehen. Diese Monitore werden aus der Beschreibung einer *Assertion* von einer speziellen Software synthetisiert und zur Laufzeit im Entwurf platziert. Eine *Assertion* bildet eine bestimmte Bedingung (*Property*), die im Entwurf auftreten muss, in einer implementierbaren Datenbasis ab. Diese Datenbasis wird vom Verifikationswerkzeug unter Berücksichtigung der Zustandsänderungen des Entwurfs ausgewertet. Für die Implementierung auf einem FPGA ist die Abbildung der Funktionalität einer *Assertion* in einer synthetisierbaren Zustandsmaschine notwendig.

Für diese Aufgabe wird von der Firma *Temento Systems* die Software *DiaLite* [98] zur Verfügung gestellt. Dadurch ist die automatische Synthese der HW-Monitore unter Berücksichtigung der funktionalen Korrektheit gegeben.

Die Integration eines HW-Monitors belastet zusätzlich den Ressourcenbedarf der Implementierung des zu verifizierenden Entwurfs auf einer FPGA-basierten Testplattform. Der in dieser Arbeit verfolgte Ansatz der Optimierung eines FPGA-basierten Emulators durch den Einsatz von Methoden der dynamischen Rekonfiguration lässt sich ebenfalls für eine ressourcenoptimierte Integrierung der HW-Monitore adaptieren.

Die ausgewählte Gruppe von HW-Monitoren, die auf ein bestimmtes HW-Modul angewendet werden sollen, wird an den Ein- und Ausgängen des Moduls platziert. Während der Verifikation werden die HW-Monitore entsprechend des geforderten Verifikationsplans dynamisch ausgetauscht. Das Ergebnis dieses Verfahrens ist der Test mehrerer funktionaler Eigenschaften auf das gleiche HW-Modul. Durch den Austausch zur Laufzeit ist der zusätzliche Verbrauch an Ressourcen gering.

Der beschriebene Ansatz ist im Umfeld der vorliegenden Arbeit entstanden, bildet aber nicht deren Schwerpunkt. Eine detailliertere Beschreibung und umfassende Ergebnisse der dynamischen Platzierung von HW-Monitoren sind im BMBF-Projekt *URANO*S im Rahmen des Arbeitspaketes *"3.1.2 On-Chip Monitoring"* [104] entstanden.

5.3.2 *Routing*-Strategie für die Implementierung des Verbindungsnetzwerks

In Abbildung 5.2 ist die Laufzeitumgebung dargestellt. Zunächst sind Gruppen, welche aus rekonfigurierbaren Flächen (*Reconfigurable Areas* - RA) bestehen, zu einem Netzwerk zusammengefasst und untereinander über ein globales Bussystem verbunden.

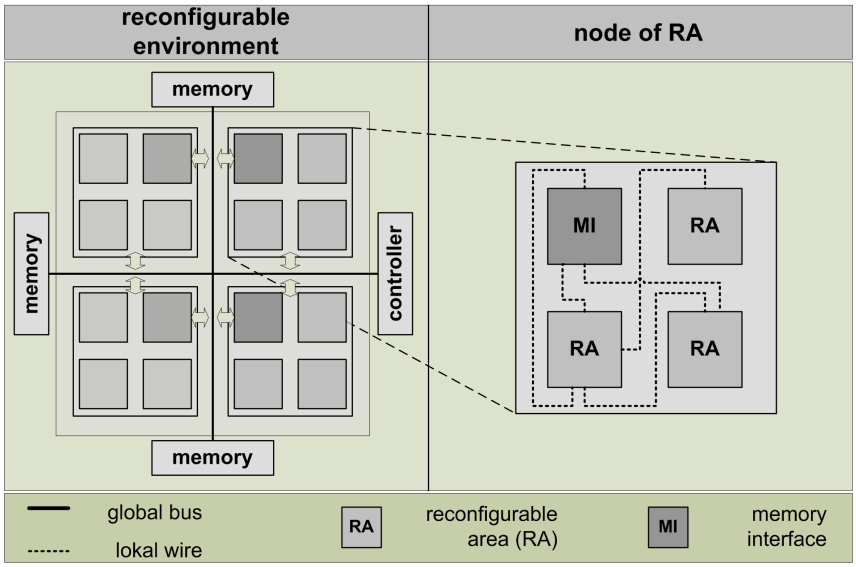


Abbildung 5.2: Blockschaltbild Rekonfigurationsumgebung

Eine Gruppe (vgl. Abb. 5.2) besteht des Weiteren aus einer parametrisierbaren Anzahl an rekonfigurierbaren Flächen und einem Netzwerk aus statisch verbundenen Leitungen zwischen den Flächen. Im

folgenden Abschnitt werden die unterschiedlichen Topologien und Netzwerkebenen und deren explizite Umsetzung beschrieben.

Für die Verbindung und Kommunikation der HW-Module ist ein angepasstes *Routing* notwendig. Im Rahmen der Arbeit wurden unterschiedliche Konzepte und Implementierungen für Verbindungsnetzwerke erarbeitet. Zunächst wurde ein Verfahren favorisiert, welches ohne ein zusätzliches Kommunikationsnetzwerk die HW-Module miteinander verbindet (vgl. Abs. 3.2.3). Dieses Verfahren ist auf Grund des erheblichen Aufwands für die Rekonfigurationen nicht für den favorisierten Ansatz für eines pRTR-Emulators geeignet. Aus den Ergebnissen dieses Ansatzes sind folgende Anforderungen für eine *Routing*-Topologie abgeleitet, welche die Grundlage für die Dimensionierung und Implementierung des Verbindungsnetzwerks bilden:

1. Um eine optimale Performanz der Emulation auch für struktural partitionierte Entwürfe zu erreichen, ist eine direkte und taktgenaue Kommunikation ausgewählter HW-Module notwendig [55].

2. Um die Anzahl von Rekonfigurationen und den Steueraufwand der Emulationen zu reduzieren, muss möglich sein, dass jedes konfigurierte HW-Module mit allen weiteren konfigurierten HW-Modulen Daten austauschen kann.

3. Für eine effiziente Speicherung der Datenwörter müssen alle platzierten HW-Module über eine breitbandige Schnittstelle an die externen QDR-Speicher verfügen.

Diese Anforderungen stellen konträre Randbedingungen für eine standardisierte *Routing*-Topologie dar. Aus diesem Grund wurde eine Kombination aus unterschiedlichen Verbindungsnetzwerken realisiert.

In Abbildung 5.3 ist die implementierte Topologie dargestellt. Auf den für die Emulation genutzten FPGAs besteht das Kommunikationsnetzwerk aus drei Ebenen:

- Die erste Ebene verbindet eine parametrisierbare Anzahl an rekonfigurierbaren Flächen (RA - Reconfigurable Area) und Speicherschnittstellen, welche direkt miteinander benachbart sind.

- In der zweiten Ebene werden diese Knoten aus rekonfigurierbaren Flächen über ein Bussystem verbunden. Des Weiteren findet in dieser Ebene der Zugriff auf die Speicherschnittstelle statt.

- In der dritten Ebene ist eine Kommunikation der einzelnen Busse der zweiten Ebene über ein *On-Chip*-Bussystem realisiert. Dieses dient hauptsächlich der Überbrückung langer Distanzen auf großen FPGAs.

- Für die Verbindungen zwischen mehreren FPGAs (vgl. Abb. 3.8) und der Integrierung des Emulator-FPGAs in das *Host*-System ist die vierte Kommunikationsebene vorgesehen. In dieser Ebene werden dedizierte Protokolle (z.B. *RocketIO*) eingesetzt um die FPGAs auch plattformübergreifend einzubinden.

Für die Kommunikation lokal benachbarter HW-Module werden direkte Verbindungen (*Point-to-Point*) platziert. Ausgehend und unter Ausnutzung von lokalen Nachbarschaften zwischen HW-Modulen werden diese bevorzugt in einem Knoten konfiguriert. Innerhalb dieses Bereichs werden die Leitungen statisch

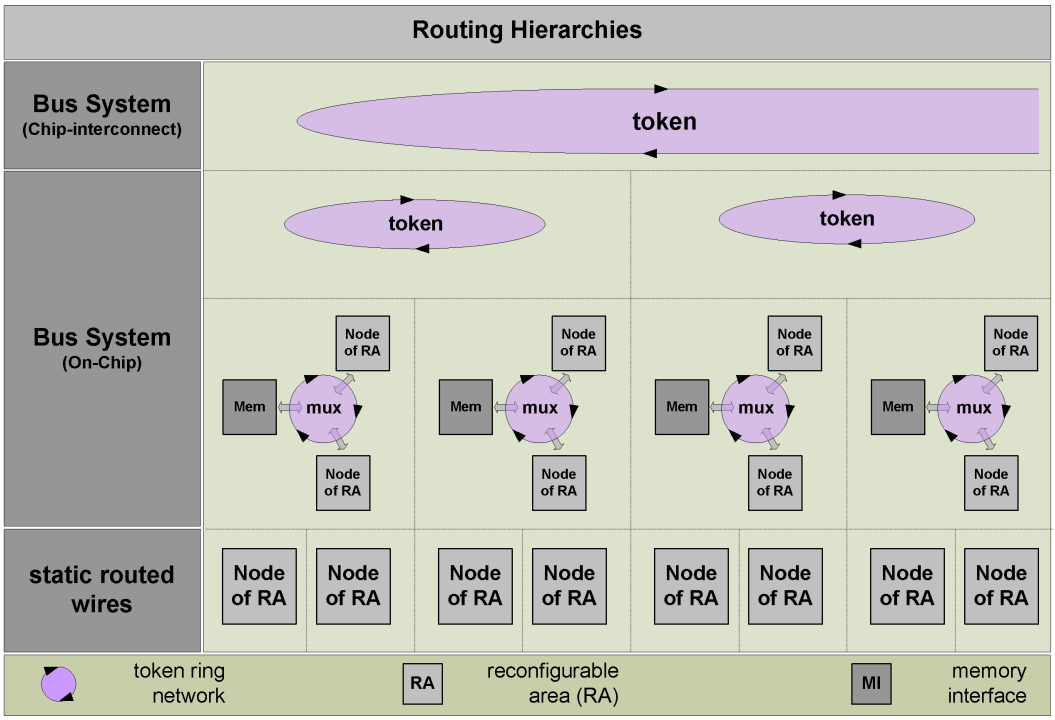


Abbildung 5.3: Implementierte Routing-Topologie

während der Generierung des Emulator-*Framework* beschaltet (*routing*). Einer bestimmten und parametrisierbaren Anzahl an rekonfigurierbaren Flächen wird eine Speicherschnittstelle (MI) zugewiesen (vgl. Abb. 5.2 3 RA, 1 MI).

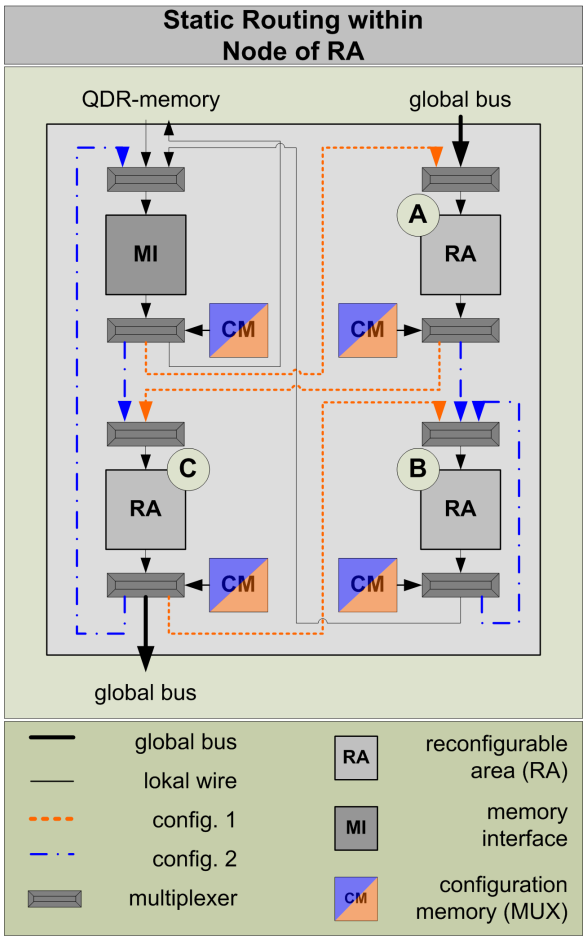


Abbildung 5.4: Statisches Routing innerhalb eines Knotens aus rekonfigurierbaren Flächen

In Abbildung 5.4 ist die Festlegung statischer Leitungen innerhalb eines Knotens abgebildet. Die statischen Verbindungen sind auf Grund der taktgenauen Kommunikation benachbarter HW-Module notwendig. Dadurch wird eine Kommunikation von HW-Modulen ohne zusätzlichen Aufwand für eine Synchronisierung erreicht. Generell reduziert sich aber durch eine statische Festlegung signifikant die Flexibilität des Ansatzes. Für eine optimale Relation zwischen einem voll vermaschten Netz zwischen den *RAs* (*Peer-to-Peer*) und der Festlegung auf ein einziges Szenario, werden im implementierten Ansatz bestimmte Kombinationen für den Datenfluss festgelegt. Die Umschaltung der einzelnen Szenarien erfolgt über Multiplexer. Jedem Multiplexer wird dafür über einen Konfigurationsspeicher (CM - configuration memory) eine bestimmte Beschaltung zugewiesen. Die Speicher der Multiplexer werden vor der Emulation geladen. Es ist auf Grund einer gemeinsamen Programmierschnittstelle der Speicher (*CM*) nicht möglich, das Schaltverhalten einzelner Multiplexer während der Laufzeit zu ändern. Im Falle der Umschaltung werden die Steuersignale aller Multiplexer verändert. In der aktuellen Implementierung sind vier Szenarien programmierbar (vgl. Abb. 5.4 2 Szenarien).

In der Abbildung 5.4 ist eine Verbindung der Speicherschnittstelle in die rekonfigurierbaren Flächen *A* (rotes Szenario) oder *C* (blaues Szenario) realisiert. Des Weiteren kann das Modul *C* Daten an das Mo-

dul *B* (dem roten Datenfluss folgend), an die Speicherschnittstelle (blauer Datenfluss) oder Datenwörter an den globalen Bus schreiben. Mit einer zunehmenden Anzahl an rekonfigurierbaren Einheiten innerhalb eines Knotens führt dieser Ansatz zu einer großen Performanz bei gleichzeitigem geringem Ressourcenverbrauch. Nachteilig ist die frühzeitige Festlegung auf bestimmte Szenarien. Unter Betrachtung aller Knoten ist für die Festlegung der statischen Kommunikationsszenarien eine große Abdeckung der möglichen Datenflüsse zu erreichen. Dieser Kommunikationsebene kommt auf Grund der direkten und taktgenauen Verbindung in der Emulation kombinatorischer Logikmodule eine große Bedeutung zu. Die Bildung der benachbarten HW-Module zu abhängigen Task wird in Abschnitt 5.4 erläutert.

Die zweite Kommunikationsebene verbindet eine bestimmte Anzahl an Knoten mit einem Bussystem. Dieses Bussystem basiert auf einem festen Zeitmultiplex und verwaltet die Datenwörter der HW-Module innerhalb der verbundenen Knoten. In dieser Kommunikationsebene werden die physikalischen Speicherschnittstellen allokiert und den entsprechenden Knoten zugewiesen. Für eine Nutzung der zweiten Ebene für die synchronisierte Kommunikation von HW-Modulen wurde ein Bussystem mit einem festen Zeitmultiplex implementiert. Die Länge der Zeitfenster für die Zugriffe t_A und die Reihenfolge der Zugriffe ist statisch festgelegt (vgl. Abb. 5.5).

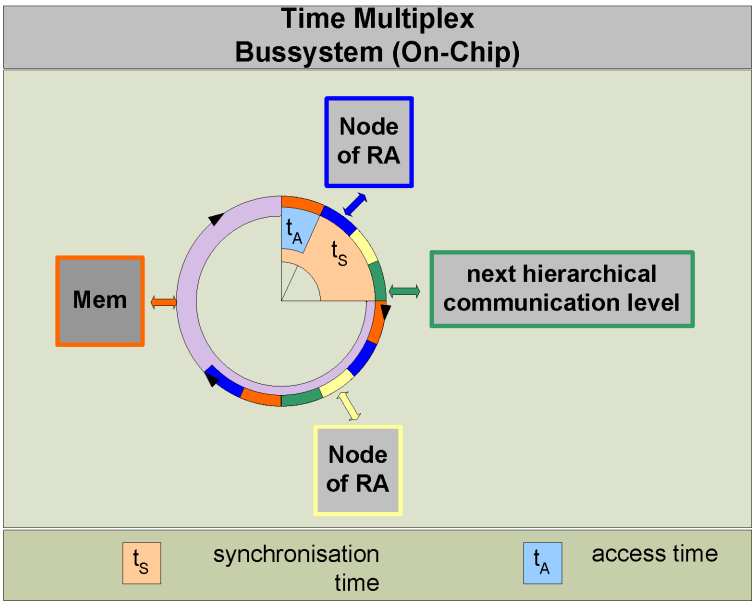


Abbildung 5.5: Zeitmultiplex-Bussystem auf On-Chip Ebene 2

In Abbildung 5.5 ist ein Netzwerk mit zwei Knoten und einer physikalischen Speicherschnittstelle abgebildet. Für die Kommunikation in die nächste Ebene ist eine weitere Schnittstelle angebunden. Aus der Anzahl der möglichen Zugriffe $N_A$und der Zugriffszeit $t_A$berechnet sich die Zeit der Synchronisierung t_S.

Dabei gilt für die Berechnung der Synchronisationszeit:

$$t_S \quad = t_A \quad * N_A \, [ms] \tag{5.1}$$

Unter Beachtung eines Austausch von Daten mit dem globalen *On-Chip* Kommunikationsnetzwerk (Ebene 3) ist die obere Schranke für die maximale Synchronisationsszeit t_{Smax} gegeben als:

$$t_{Smax} \quad \leq \quad N_N * max(t_S)\,[ms] \tag{5.2}$$

wobei $N_N$die Anzahl der Knoten (*Nodes N*) im *On-Chip* Netzwerk ist. Gleichung 5.2 gilt unter Berücksichtigung, dass alle lokalen Netzwerkknoten (Ebene 2) innerhalb der Synchronisationszeit des lokalen Netzes mit der längsten Synchronisationszeit auf das globale *On-Chip*-Netz zugreifen. Anderenfalls findet ein asynchroner Zugriff auf das globale Netz statt. In diesem Fall reduziert sich die Synchronisationszeit proportional zur Anzahl der zugreifenden lokalen Netzknoten.

Der Synchronisierung des gesamten *On-Chip*-Netzwerkes auf einem FPGA oder eines Netzwerkes zwischen FPGAs kommt auf Grund der temporalen Gültigkeit der Datenwörter während der Emulation eine große Bedeutung zu. Für eine Verteilung kommunizierender HW-Module, welche in unterschiedlich synchronisierten lokalen Netzknoten platziert wurden, ist eine übergreifende Synchronisierung notwendig. Konträr dazu steht die Forderung, dass die lokalen Emulationen unabhängig und nicht beeinflusst von der Taktrate anderer Funktionen der Emulationsumgebung agieren sollen.

Dafür wurde für die Ring-Netzwerke der Ebenen 3 und 4 eine Synchronisierung implementiert, welche auf Zeitstempeln und der Erfassung absoluter Adressen der HW-Module basiert. Im Falle eines Schreibvorgangs wird das Datenwort, der Zeitstempel, die Nummer des Zielknotens und die Nummer des Quellknotens zu einem Paket verbunden und auf dem Bus gesendet.

Der Zeitstempel wird durch ein globales Signal generiert und an den Schnittstellen der lokalen Knoten abgegriffen. Dadurch ist zu jedem Zeitpunkt an jedem Knoten ein einheitlicher Vergleichswert verfügbar. Ein Problem dieses Verfahrens ist der Abgleich des Synchronisationssignals mit weiteren FPGAs. In diesem Fall muss der am geringst performante FPGA das Signal für alle weiteren FPGAs generieren. Für diese Arbeit wurde für die Implementierung nur ein FPGA betrachtet. Die Zeitstempel werden analog für die im Speicher abgelegten Datenwörter verwendet. Dadurch ist gewährleistet, dass die Datenwörter zu einem angeforderten Emulationszyklus geladen und später zugeordnet werden können. Die Auswertung der Zeitstempel unterstützt ebenfalls die Herstellung der zeitlichen Ordnung der Datenwörter auf dem *Host*-System.

Der Einsatz eines zusätzlichen *Controllers* für eine Verwaltung der Datenwörter auf dem FPGA wurde auf Grund der Reduzierung der FPGA-Ressourcen nicht verfolgt. Im Rahmen der Arbeit [38] wurde die beschriebene Methode für das Verbindungsnetzwerk umgesetzt, welches in einer Kombination die unterschiedlichen Eigenschaften verschiedener Netztopologien zulässt.

5.4 Task-Bildung für eine effiziente Ausnutzung des Verbindungsnetzwerks

Für eine effiziente Ausnutzung der lokalen statisch verdrahteten Verbindungen ist eine Auswahl geeigneter HW-Module notwendig. Diese Module werden als zusammenhängende Einheit innerhalb eines rekonfigurierbaren Knotens platziert. Der Algorithmus für die Partitionierung des Entwurfs übergibt zunächst eine Liste der zusammenhängenden HW-Module. Diese Liste wird aus der Funktion, welche für die *Clusterung* herangezogen wird, abgeleitet.

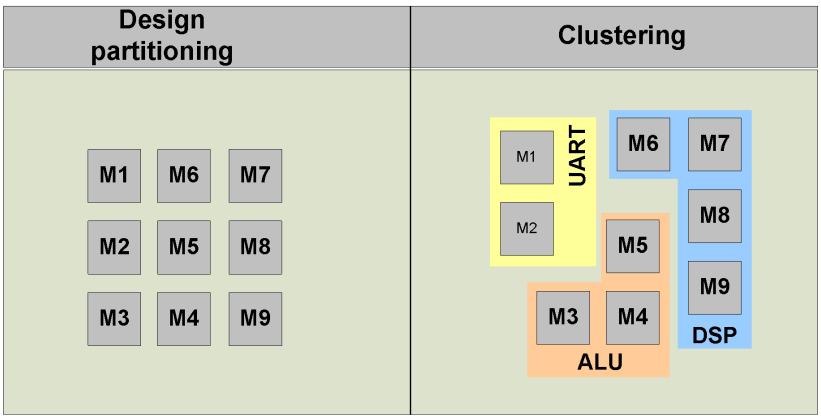


Abbildung 5.6: Entwurfsbeispiel für Task-Bildung

In Abbildung 5.6 ist ein Beispiel für die Partitionierung und anschließende *Clusterung* eines zu emulierenden *System-on-Chips* veranschaulicht. Folgende Schritte sind für die *Clusterung* notwendig:

- Das SoC wird zunächst in die HW-Module M1-M9 partitioniert.
- Nach der Modulauswahl beginnt die Bildung der *Cluster*. Zunächst wird dafür die *Cluster*-Größe festgelegt. Diese hängt von der Anzahl der rekonfigurierbaren Flächen innerhalb eines Knotens ab. Im Beispiel in Abbildung 5.6 existiert ein *Cluster,* welches 2 HW-Module (*UART*) enthält, ein *Cluster* mit 3 HW-Modulen (*ALU*) und eines mit 4 HW-Modulen (*DSP*).
  Für die Festlegung der Anzahl an rekonfigurierbaren Flächen innerhalb eines Knotens wird der Mittelwert der *Cluster*-Größen, im Beispiel das *Cluster*, welches 3 HW-Module enthält, zu Grunde gelegt.
- Im Anschluss werden die statischen Leitungen innerhalb der Knoten für bestimmte Szenarien festgelegt. Im Beispiel (vgl. Abb. 5.6) ist das Ziel für die Platzierung die *Cluster UART* und *ALU* innerhalb eines einzelnen Knotens zu verdrahten. Das *Cluster DSP* wird auf Grund der Größe auf zwei Knoten aufgeteilt. Die Kommunikation zwischen den HW-Modulen *M8* und *M9* erfolgt in diesem Fall über das lokale Kommunikationsnetzwerk (vgl. Abb. 5.3 Ebene 2).

Abbildung 5.7 stellt die Bildung der *Tasks* für das Beispiel aus Abbildung 5.6 dar. Durch die Festlegung und Zusammenfassung abhängiger HW-Module ist eine effiziente Platzierung der HW-Module gewährleistet. Der Austausch von Modulen wird unter Beachtung der gebildeten *Task*-Grenzen durchgeführt. Dadurch werden die notwendigen Rekonfigurationen für kleine *Tasks* (vgl. Abb. 5.7 *UART, ALU*), welche innerhalb eines Knotens platzierbar sind, signifikant reduziert.

Eine konträre Strategie, welche für die Platzierung der HW-Module den externen Speicher mit einbezieht, reduziert zwar die Anzahl notwendiger Knoten, vergrößert jedoch wesentlich die Anzahl der notwendigen Rekonfigurationen. Der Anwendung dieser Strategie kommt für die Platzierung von *Task*s, welche auf Grund der Größe bzw. der Anzahl der HW-Module über die Grenzen von Knoten hinaus platziert werden, besondere Bedeutung zu. Im Beispiel in Abbildung 5.7 ist die Rekonfiguration aus dem externen Speicher für das *DSP-Task* in Betracht zu ziehen, da dieser die Grenze eines Knotens überschreitet und

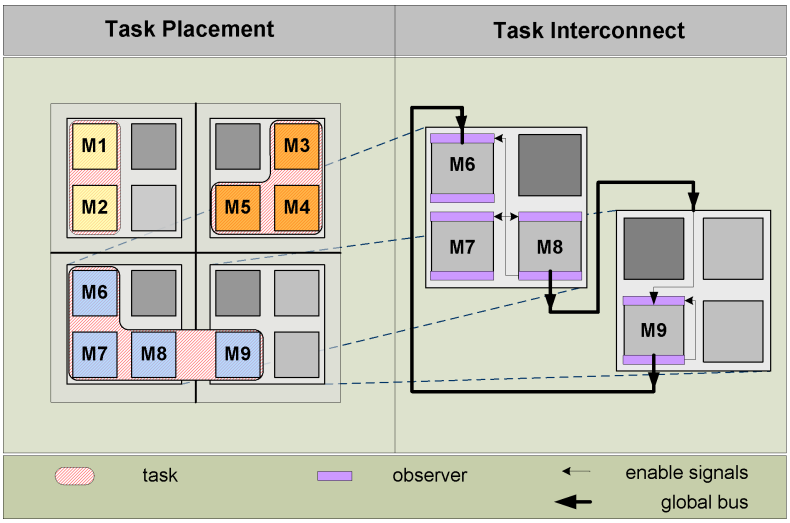


Abbildung 5.7: Task-Bildung für verteilte HW-Module

damit während der Rekonfiguration durch den permanenten Datenaustausch zwischen den Modulen *M8* und *M9* den globalen Bus sehr stark auslasten wird.

Unter Anwendung des speicherbasierten Verfahrens schreibt HW-Moduls *M8* die Datenworte direkt an die Speicherschnittstelle. Nach einer Rekonfiguration aller HW-Module im entsprechenden Knoten wird das HW-Modul *M9* dem Datenfluss folgend mit den Datenwörtern aus dem Speicher in der folgenden partiellen Emulation stimuliert.

Die beschriebene Verfahrensweise wird analog für die Emulation kombinatorischer HW-Module eingesetzt, bei welcher ein taktgenauer Signalaustausch einer aufgeteilten Netzliste abgebildet werden muss (vgl. Abs. 5.6.3).

Die Auswahl der HW-Module wird in der aktuellen Implementierung automatisch durchgeführt. Die Nachbarschaftsgrenzen zusammengehörender HW-Module werden dafür in einer Matrix eingetragen, welche Bestandteil der Parameterbeschreibung des Emulator-*Frameworks* ist. Die Gewichtungen der Matrixelemente bilden während der Generierung des *Frameworks* die Grundlage für die Bildung der *Tasks*. Die gewichtetsten Nachbarn eines Elements der Matrix werden in den neuen *Task* einbezogen. Dieser Vorgang wird bis zur Erreichung der vorgegebenen maximalen Anzahl an HW-Elementen in einem *Task* fortgesetzt. Die weiteren benachbarten Elemente bilden den folgenden *Task*. Weist ein HW-Modul keine Nachbarn auf, bildet es einen eigenständigen *Task*. Durch die Berücksichtigung der Nachbarschaftsgrenzen zwischen den HW-Modulen ist neben der Ausnutzung lokaler Abhängigkeiten zwischen den HW-Modulen eine optimale Ausnutzung und Parametrisierung des Emulator-*Frameworks* gegeben.

5.5 Implementierung der Hardware für die Ereignis-Detektion

Wie bereits dargestellt, setzt sich die Steuerung der HW-Module aus den zwei gundsätzlichen Informationen zusammen, zu welchem Zeitpunk und über welche Schnittstellen die HW-Module Daten untereinander austauschen. Auf Grund der Partitionierung des Entwurfs ist die Information transparent, welche

HW-Module über welche Schnittstellen miteinander kommunizieren. Auf der Basis dieses Musters wird die Entscheidung für die Platzierung der HW-Module getroffen.

Der genaue Zeitpunkt eines Zugriffs auf eine Schnittstelle ist vor der Emulation ansatzbedingt und auf Grund unterschiedlicher Stimulidaten nicht deterministisch. Aus diesem Grund wird auf HW-Monitore zurückgegriffen, welche das Schreibereignis auf eine Schnittstelle detektieren und selbstständig das Modul und alle weiteren HW-Module in dem *Task* deaktivieren. Die Ablaufsteuerung des Emulator-*Frameworks* basiert auf einer ereignisbasierten Erkennung der Zustände einer laufenden partiellen Emulationen. Für die Steuerung des *Frameworks* ist aus diesem Grund eine angepasste Erkennung der Schreibvorgänge an den Schnittstellen der rekonfigurierbaren Flächen notwendig.

Grundsätzlich ist eine Unterscheidung von gültigen gegenüber ungültigen Schreibvorgängen an einer Schnittstelle ohne zusätzliche Informationen wie der Gültigkeitsangabe eines Schreibzugriffs (*valid bit*) nicht möglich. Des Weiteren unterscheidet sich der Schreibvorgang eines funktional partitionierten HW-Moduls vom Schnittstellenzugriff eines struktural partitionierten HW-Moduls. Während die Daten, welche an eine Schnittstelle zwischen zwei funktional partitionierten HW-Modulen erst im Falle gültiger Datenwörter geschrieben werden, propagiert ein kombinatorisches Modul fortlaufend relevante Informationen an den Ausgang. Daraus resultiert eine grundsätzliche Unterscheidung in der Emulation beider Varianten. Während für funktional bestimmte HW-Module die Erkennung eines Schreibvorgangs relevant ist, ist für kombinatorische HW-Module zusätzlich die Anzahl der geschriebenen Datenwörter von Bedeutung. Dafür wurde im Rahmen der Arbeit ein HW-Monitor entworfen, welcher an den Eingangs- und Ausgangsschnittstellen einer rekonfigurierbaren Fläche platziert wird.

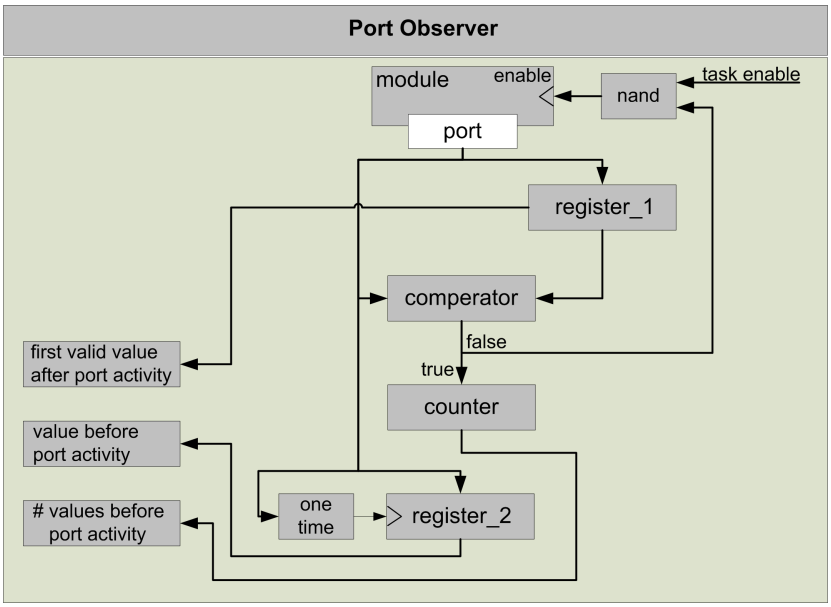


Abbildung 5.8: Schnittstellen HW-Monitor

In Abbildung 5.8 ist das Blockschaltbild des HW-Monitors dargestellt. Zur Vereinheitlichung wurde ein gemeinsamer HW-Monitor zur Erkennung der Schnittstellenaktivität für funktional- und struktural partitionierte Entwürfe implementiert. Der HW-Monitor arbeitet die folgenden Schritte ab um den Schreibvorgang

zu detektieren:

- Ein an die Schnittstelle der rekonfigurierten Fläche des HW-Moduls (vgl. Abb. 5.8 *port*) geschriebenes Datenwort wird im ersten Schritt in beiden Registern abgelegt.
- Der *enable*-Eingang von *Register_2* wird im Anschluss für einen weiteren Schreibvorgang gesperrt (*one time*).
- Im folgenden Taktzyklus vergleicht der *Comperator* das aktuell an der Schnittstelle anliegende Datenwort mit dem in *Register_1* gehaltenen Datenwort. Sind diese Datenwörter unterschiedlich, ist eine Aktivität an der Schnittstelle detektiert und der *enable*-Eingang des Moduls wird deaktiviert. Dadurch können keine weiteren Datenwörter an die Schnittstelle propagiert werden. Sind die verglichenen Datenworte äquivalent, wird der Zähler (*counter*) um den Wert eins inkrementiert.
- Nach einer Detektierung speichert der HW-Monitor das aktuelle Datenwort (*first valid value*), die Anzahl vor der Schnittstellenaktivität geschriebener Datenworte und das Datenwort selbst. Mit diesen Informationen ist eine Wiederherstellung der Aktivität im Beobachtungszeitraum gegeben.
- Wenn die Daten aus dem Speicher in ein Modul zurück geschrieben werden, gibt der *Controller* zur Steuerung des Emulator-*Frameworks* die Informationen bzgl. der Anzahl an Schreibvorgängen an die Speicherschnittstelle. Diese lädt entsprechend der Werte die adressierten Daten aus dem Speicher und übergibt sie an den Datenbus. Über diesen werden die Daten direkt an das zu stimulierende HW-Modul geschrieben. Dadurch ist eine taktgenaue Stimulation der Eingänge eines im Datenfluss folgenden HW-Moduls möglich.

Für die Steuerung eines vollständigen *Tasks* wird das *Task_enable*-Signal mit den *enable*-Eingängen der im *Task* zusammengefassten HW-Module verbunden (vgl. Abb. 5.7). Diese dezentrale Steuerung gewährleistet eine Datensicherheit für alle HW-Module des *Tasks* im Fall eines Schreibvorgangs auf eine Schnittstelle, auf deren Gegenseite noch kein HW-Modul konfiguriert wurde. Die HW-Monitore werden während einer Rekonfiguration mit dem HW-Modul ausgelesen und abgespeichert. Dadurch ist eine vollständige Wiederherstellung der inneren Registerzustände eines *Tasks* nach einer Rekonfiguration möglich. Die *Task_enable*-Signale sind des Weiteren mit dem *Controller* für die Steuerung des *Frameworks* verbunden. Dieser setzt die detektierten Ereignisse in eine Befehlssequenz um und sendet den Status der verschiedenen *Tasks* an den *PowerPC*. Für die Auswahl und Festlegung eines bestimmten *Tasks* wird die *Interrupt_mask* (vgl. Abb. 5.1) entsprechend kodiert. Im folgenden Abschnitt werden Emulationsszenarien für bestimmte Architekturen beschrieben.

5.6 Ableitung von Szenarien für die Steuerung unterschiedlicher Architekturen

Eine wesentliche Anforderung für das Erreichen einer optimalen Effizienz des Ansatzes ist die Ableitung von angepassten Szenarien für die partiellen Emulationen. Diese Szenarien sind bestimmt durch die Architektur und den Aufbau des zu emulierenden Entwurfs. Ausgehend von einem strukturiert entworfenen *System-on-Chip* wird im Abschnitt 5.6.1 die Ablaufsteuerung für das Emulator-*Framework* beschrieben. Unter Berücksichtigung regulärer Architekturen, welche in Abschnitt 3.5 beschrieben sind, wird der

angepasste Ablauf der Emulation in Abschnitt 5.6.2 dargestellt. Die Emulation eines aufgeteilten kombinatorischen Systems stellt an die Steuerung auf Grund der taktgenauen Synchronisierung der separaten HW-Module eine besondere Herausforderung. In Abschnitt 5.6.3 wird die Umsetzung der Ablaufsteuerung für kombinatorische HW-Module beschrieben.

5.6.1 Strukturiert Modulare Entwürfe

Die Verifikation eines modularen Entwurfs stellt die Basis für die Anwendung des neuen Ansatzes dar. Ausgehend von einer auf modularer Ebene partitionierten Beschreibung des zu verifizierenden Systems, werden die daraus synthetisierten HW-Module in funktionalem Kontext unter Verwendung des Emulator-*Frameworks* auf das Emulator-FPGA konfiguriert. Durch die Heterogenität eines SoC und der damit einhergehenden Divergenz der Modulparameter lassen sich keine Optimierungen durch die Berücksichtigung regulärer HW-Module vornehmen. Der zeitliche Aufwand für die notwendigen Rekonfigurationen ist aus diesem Grund bestimmend für die Gesamtzeit der Emulation. Die Rekonfiguration der HW-Module wird im implementierten Ansatz direkt über die ICAP-Schnittstelle durchgeführt. Dadurch hängt die Rekonfigurationszeit vom erreichbaren Takt auf dem Emulator FPGA ab.

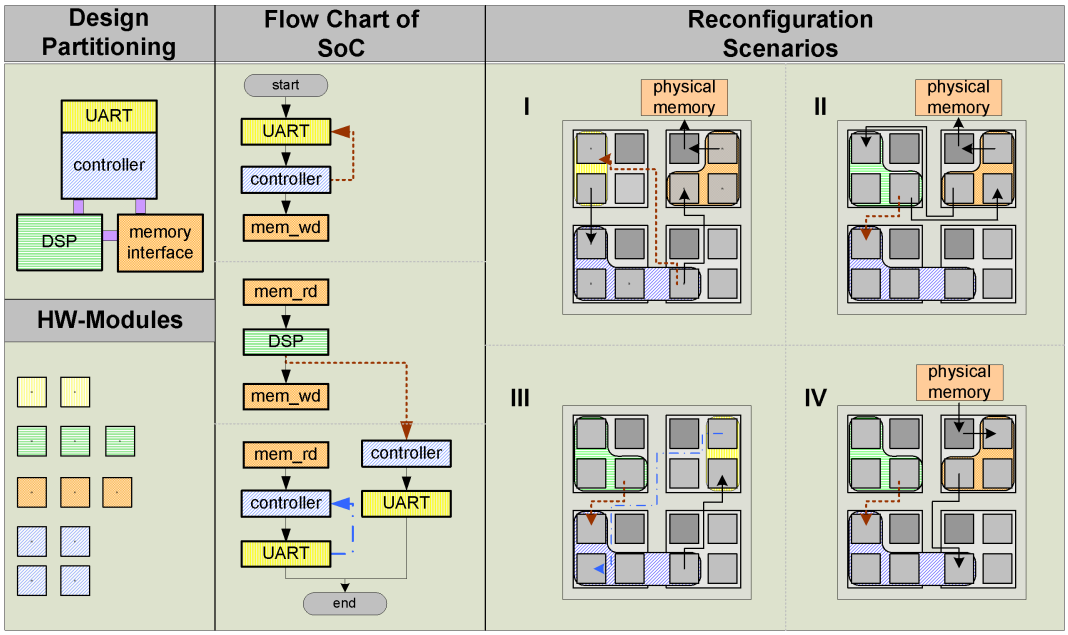


Abbildung 5.9: Szenarien für rekursive Rekonfiguration eines SoC

Die partielle Emulation teilt sich in folgende Schritte ein:

- Im ersten Schritt trägt der *PowerPC* die zu konfigurierenden HW-Module in eine Liste ein. Der *Task,* welcher das aktuelle HW-Modul enthält, bekommt die höchste Priorität.
- Im Anschluss wird der Initialzustand für den Beginn der Emulation konfiguriert. Dazu werden die *Tasks*, welche die hoch priorisierten HW-Module enthalten, auf dem Emulator-FPGA platziert, bis alle RA belegt sind.

- Darauf folgend lädt das Steuerwerk die Stimulidaten in den Speicher und initialisiert die Emulation.
- Der *PowerPC* sendet fortlaufend Befehlssequenzen an den *Controller*, bis ein *Interrupt* zurück gesendet wird oder der Eingangs-FIFO (*in_FIFO*) im Steuerwerk gefüllt ist.
- Das Steuerwerk überwacht ab dem Start der Emulation kontinuierlich den Zustand der HW-Module. Löst ein HW-Monitor an einer Schnittstelle ein Ereignis aus, sendet der *Controller* eine entsprechende Sequenz an den *PowerPC*.
- Der Steueralgorithmus berechnet die folgende Sequenz und entscheidet im Anschluss, ob der *Task* ausgetauscht wird oder auf dem FPGA verbleibt. Die Entscheidung hängt von der Position des *Task* in der Liste mit verbleibenden HW-Modulen ab. Wird ein HW-Modul, welches zu dem *Task* gehört mit hoher Priorität geführt, verbleibt der *Task* zunächst auf dem FPGA. Im Falle einer niedrigen Priorität erfolgt die Abspeicherung des gesamten *Tasks*.
- Die ICAP-Schnittstelle schreibt im Anschluss die entsprechende Konfigurationsdatei zurück an das *Host*-System.
- Sind alle verfügbaren Knoten im *Framework* belegt und es wird ein Schnittstellenzugriff erkannt, wird der *Task*, welcher den Schnittstellenzugriff ausgelöst hat, durch den im Datenfluss folgenden *Task* ausgetauscht.

Da die im Datenfluss vor dem ausgetauschten *Task* platzierten HW-Module mit hoher Wahrscheinlichkeit einen Schreibvorgang an den HW-Monitoren detektieren werden, führt dieses Szenario zum rekursiven Rekonfigurieren der abhängigen HW-Module. Im optimalen Fall besteht der zu emulierende Entwurf aus unabhängigen HW-Modulen, welche über einen Speicher mit einander kommunizieren. In diesem Fall entstehen *Tasks*, welche über einen langen Zeitraum im *Framework* platziert bleiben und die Daten über den globalen Bus mit den Speicher austauschen können.

5.6.2 Reguläre Architekturen

Die Kosten für das Rekonfigurieren der HW-Module eines Entwurfs lassen sich unter Beachtung des Datenflusses in regulärer Architekturen signifikant reduzieren. Ausgehend von einer Partitionierung des Entwurfs in regulär implementierte HW-Module und in singulär implementierte Verbindungsmodule, wird der Entwurf entsprechend der Funktionalität der HW-Module emuliert. Die verhaltensgleichen regulären HW-Module weisen während der Emulation ein identisches Laufzeitverhalten auf, welches durch einen angepassten Ablauf des Datenflusses während der partiellen Emulationen entsprechend der Parallelität mehrfach durchlaufen wird. Dafür werden nur zwei multipel instantiierte HW-Module abgebildet und entsprechend dem Grad der Parallelität unterschiedlich oft mit den zugehörenden Stimulidaten emuliert. Der Datenaustausch zwischen den Modulen erfolgt über einen Speicher oder das Kommunikationsmodul der parallelen Hardware.

Im Falle einer hoch parallelen SIMD-Architektur (vgl. Abb. 3.5 und 5.10) wird von jedem regulären HW-Modul eine identische Instruktionsfolge abgearbeitet. Diese ist durch das Befehlsregister mit dem aktuell auszuführenden Befehl des Steuerwerks bestimmt. Für die Emulation wird das Steuerwerk als HW-Modul mit auf dem Emulator abgebildet. Zusätzlich muss während der Kommunikation der Datenpfade das HW-Modul für den Datenaustausch auf dem Emulator platziert werden.

In Abbildung 5.10 ist der Ablauf für die Emulation einer N-fach parallelen SIMD-Architektur dargestellt. Zunächst wird eine Partitionierung der Architektur unter Berücksichtigung der regulären HW-Module vorgenommen. Dafür werden die Datenpfade und die zugehörigen Register entsprechend der Parallelität einem identischen HW-Modul zugeordnet. Das programmierbare Verbindungsnetzwerk (ICU) und das Steuerwerk (*instruction controller*) bilden zusätzliche HW-Module. Im zweiten Teil ist der Datenfluss für den Vektor-Datenpfad abgebildet. Folgende Schritte sind für die Emulation durchzuführen:

- Nach der Konfiguration des Emulators beginnt die partielle Emulation der HW-Module.
- Während der Emulation der initialen Konfiguration (Szenario I) dekodiert das Steuerwerk die Befehlssequenz und schreibt den Befehl in die Befehlsregister der Datenpfade. Zusätzlich werden Datenworte in den Registern abgelegt.
- In Sequenz II werden zwei HW-Module, welche die Datenpfade abbilden, rekonfiguriert. Aus dem gespeicherten Befehlswort und den Daten, welche in den Registern abgelegt wurden, berechnen die Datenpfade die ersten gültigen Datenworte. Dieser Vorgang wird entsprechend der Parallelität der Architektur P und der Anzahl konfigurierbarer Datenpfade N_P wiederholt (N_i). Auf Grund der Voraussetzung, dass zwei HW-Module eines Datenpfades für einen ICU-Befehl konfiguriert sein müssen gilt:

$$N_i \quad = \quad 2 * \frac{P}{N_P} \tag{5.3}$$

- In der dritten Sequenz (vgl. Abb. 5.10) erfolgt ein Austausch der Registerwerte der einzelnen *Slice* durch das HW-Modul der ICU.
- Im Anschluss werden die Datenwörter zweier Datenpfade entsprechend dem ICU-Befehlswort zwischen den *Slice* verschoben.
- Nach dem Datenaustausch der Sequenz II erfolgt eine erneute Konfiguration mit den Datenwörtern der weiteren zwei Datenpfade und den zugehörenden Registern.
- Nach dem Laden der Registerwerte für die Datenpfade wird Sequenz III konfiguriert. Über die ICU werden analog die Datenworte verschoben. Dieser Vorgang wird wiederholt, bis alle ICU-Befehle abgearbeitet sind.

Für die zeitlichen Kosten der Emulation T_{ICU} eines ICU-Befehls gilt abhängig vom Grad der Parallelität (vgl. Gln. 5.3) und unter Berücksichtigung von der zu rekonfigurierenden Größe der HW-Module mit den Registern $sizeof(Reg)$ und der ICU $sizeof(ICU)$ und den daraus resultierenden Rekonfigurationszeiten $T_{R(Reg)}$ und $T_{R(ICU)}$ (vgl. Gln. 3.3 und 3.4):

$$T_{ICU} \quad = \quad \frac{N_i}{2} * \left(2 * T_{R(Reg)} + T_{R(ICU)}\right)[s] \tag{5.4}$$

Handelt es sich um eine MIMD-Architektur, ist zusätzlich zum Datenfluss der Instruktionssatz des Prozessors zu berücksichtigen. Der exakte Ablauf der Emulation hängt von der jeweiligen Architektur ab.

Grundsätzlich verlagern sich die Kosten für die Emulation von der Rekonfigurationszeit zu den Kosten für den Datentransfer aus dem Speicher in die entsprechenden rekonfigurierten HW-Module.

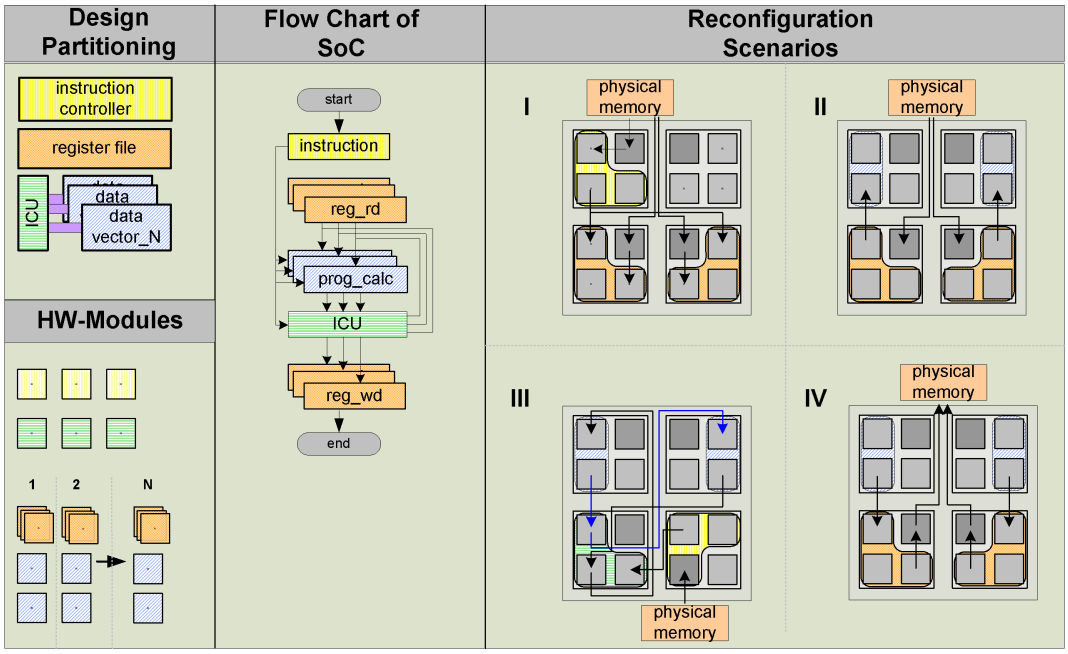


Abbildung 5.10: Szenario für die Rekonfiguration einer SIMD-Architektur

Für eine Reduzierung der Anzahl an Rekonfigurationen ist eine dauerhafte Platzierung der ICU auf dem Emulator-FPGA vorzusehen. Ein signifikantes Problem während der Emulation der einzelnen *Slices* stellt die Berücksichtigung der Datenabhängigkeiten zwischen den einzelnen *Slice* dar. Bestimmte Schiebebefehle für den Austausch eines Datenwortes zu einem anderen *Slice* verursachen eine temporale Abhängigkeit bereits emulierter Vektoren. Aus diesem Grund ist für die Abbildung des zyklischen Datenaustausches zwischen den Datenvektoren über die ICU eine simultane Platzierung von zwei Vektoren erforderlich.

Während für die Emulation eines modularen SoCs (vgl. Abs. 5.6.1) auf Grund der Heterogenität die Anzahl der Rekonfigurationen im Mittelpunkt stehen, ist das Kommunikationsnetzwerk für den iterativen Datenfluss der Schwerpunkt für eine effiziente Umsetzung einer Emulation für reguläre Architekturen. Dafür sind Kosten für eine Reduzierung der Rekonfigurationen zu den Kosten für das Kommunikationsnetzwerk zu verschieben.

5.6.3 Kombinatorische Logik

Die Emulation von HW-Modulen, welche nicht in funktionalem, sondern in strukturalem Kontext zueinander stehen, basiert im Wesentlichen auf einer alternierenden Sicherung der Registerzustände im externen Speicher und dem darauf folgenden Austausch der HW-Module. Diese Vorgehensweise ist notwendig, um einen Datenverlust während der Rekonfigurationen zu verhindern. Eine weitere Alternative für eine Reduzierung der Rekonfigurationen ist die Aufteilung des zu verifizierenden Entwurfs in sequentielle und kombinatorische HW-Module (vgl. Abb. 4.7). Dadurch ist es möglich, ohne ein Sichern der Registerzu-

stände die HW-Module der kombinatorischen Logik auszutauschen. Während die Emulation von HW-Modulen, welche in funktionalem Kontext zueinander stehen, eine Emulation über lange Laufzeitperioden ermöglicht, ist in der Emulation struktural partitionierter Entwürfe nach jedem Takt ein Abgleich der Registerzustände notwendig. Durch die HW-Monitore (vgl. Abs. 5.5) an den Schnittstellen der Module ist es möglich, in Abhängigkeit der Anzahl an Registerstufen innerhalb des HW-Monitors durch eine Zwischenspeicherung der Ausgangswerte die Anzahl der Rekonfigurationen zu reduzieren.

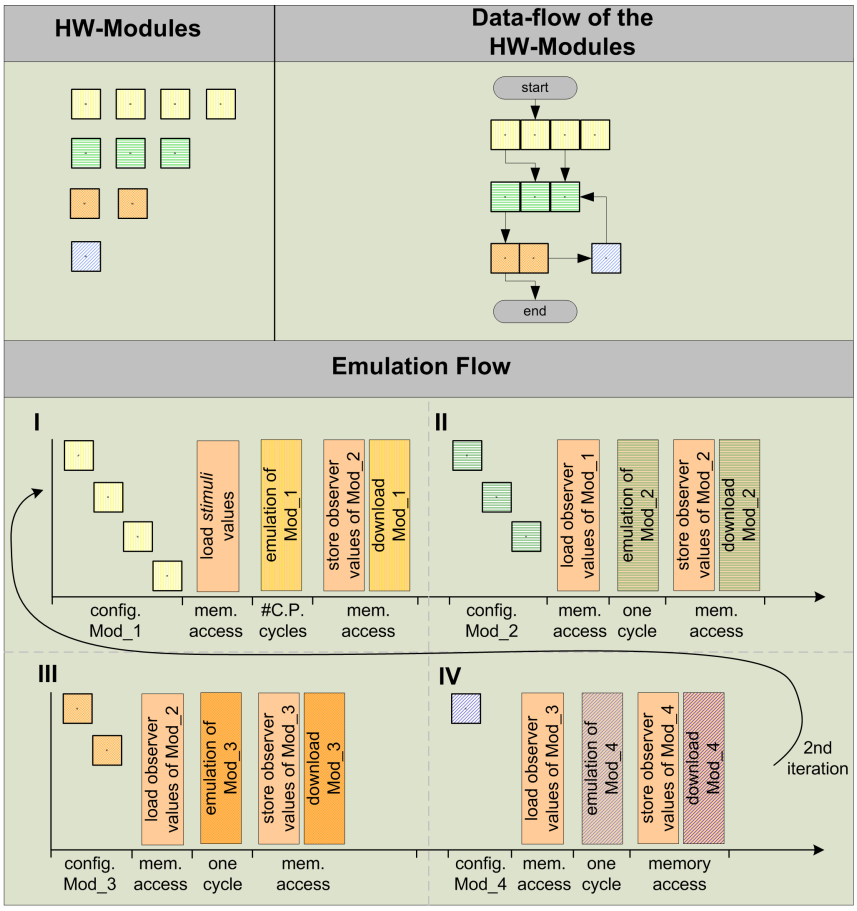


Abbildung 5.11: Emulationsablauf für kombinatorische Logik

Der Ablauf einer Emulation für kombinatorische HW-Module erfolgt im implementierten Ansatz unter Verwendung des externen Speichers und es sind folgende Schritte durchzuführen:

- Zunächst wird das HW-Modul, welches am Beginn den Datenflusses steht, in das Emulator-*Framework* geladen.
- Nach dem Beginn der Emulation propagiert die Logik innerhalb des HW-Moduls sofort unbestimmte Werte an die Schnittstelle.

- Der HW-Monitor registriert die Datenwörter und speichert das erste Wort und die Anzahl der äquivalenten folgenden Datenwörter. Ändert sich das Datenwort, stoppt der HW-Monitor die Funktionalität des betreffenden Moduls oder des gesamten *Tasks*. Die zwei gespeicherten Datenwörter und der zugehörige Zähler verbleiben im Emulator-*Framework.*
- Die Konfigurationsdatei des HW-Moduls wird mit den eingestellten Registerzuständen abgespeichert.
- Das im Datenfluss folgende HW-Modul wird geladen und entsprechend den im Speicher abgelegten Datenwörtern stimuliert.

In der umgesetzten Implementierung verbleibt der HW-Monitor für strukturale Emulationen im *Framework*. Dadurch wird ein taktgenauen Übergang zwischen zwei HW-Modulen während einer Rekonfiguration gesichert. Auf Grund der großen Anzahl an Rekonfigurationen, die für die Emulation eines struktural partitionierten Entwurfs notwendig sind, stellt dieser Ansatz allerdings den Flaschenhals dar, da für alle HW-Module ein separater HW-Monitor benötigt wird.

5.6.4 Diskussion und Bewertung eines Ansatzes zur Vorhersage von HW-Modulen zur Optimierung des Schedul-Algorithmus

Unabhängig vom Partitionierungsverfahren hängt die Effizienz des implementierten Ansatzes im Wesentlichen von der Anzahl der Rekonfigurationen ab. Durch die Möglichkeit, partiell HW-Module auf dem FPGA auszutauschen, während weitere HW-Module aktiv bleiben, ist eine Optimierung durch eine parallele Rekonfiguration möglich. Der Zeitpunk der Rekonfiguration wird im umgesetzten Ansatz durch die Detektion eines Schreibereignisses auf eine Schnittstelle bestimmt. Unter Berücksichtigung vorausgegangener Emulationen ist auf Grund eines zu erwartenden äquivalenten Systemverhaltens, bei gleichen Stimulidaten, in den folgenden Emulationensläufen eine Vorhersage von Rekonfigurationen der HW-Module möglich. In [17] wurde ein Verfahren umgesetzt, welches auf einem *Hidden-Markov*-Modell (HMM) basiert. Das *Markov*-Modell wird durch eine Matrix abgebildet, welche alle Zustände des Programmzählers eines Steuerwerks enthält. Während der Berechnung äquivalenter Algorithmen durch das Steuerwerk werden in der Matrix die Zustandsübergänge des Programmzählers gespeichert. Ein Zustand, welcher in der Matrix z.B. durch Sprünge mehrfach erreicht wird, inkrementiert den Wert in diesem Feld der Matrix. Dadurch stellt sich im Laufe mehrerer Programmdurchläufe eine Gewichtung bestimmter Systemzustände ein. Ausgehend von diesen Daten lässt sich eine Vorhersage von Folgen des Programmzählers unter Berücksichtigung der größten Wahrscheinlichkeit für den folgenden Wert eines Programmzählers treffen.

Dieses Verfahren kann ebenso auf die Vorhersage der zu rekonfigurierenden HW-Module angewendet werden. In der Matrix werden für im Falle des partiellen Emulators die Übergänge der HW-Module während der Rekonfigurationen eingetragen.

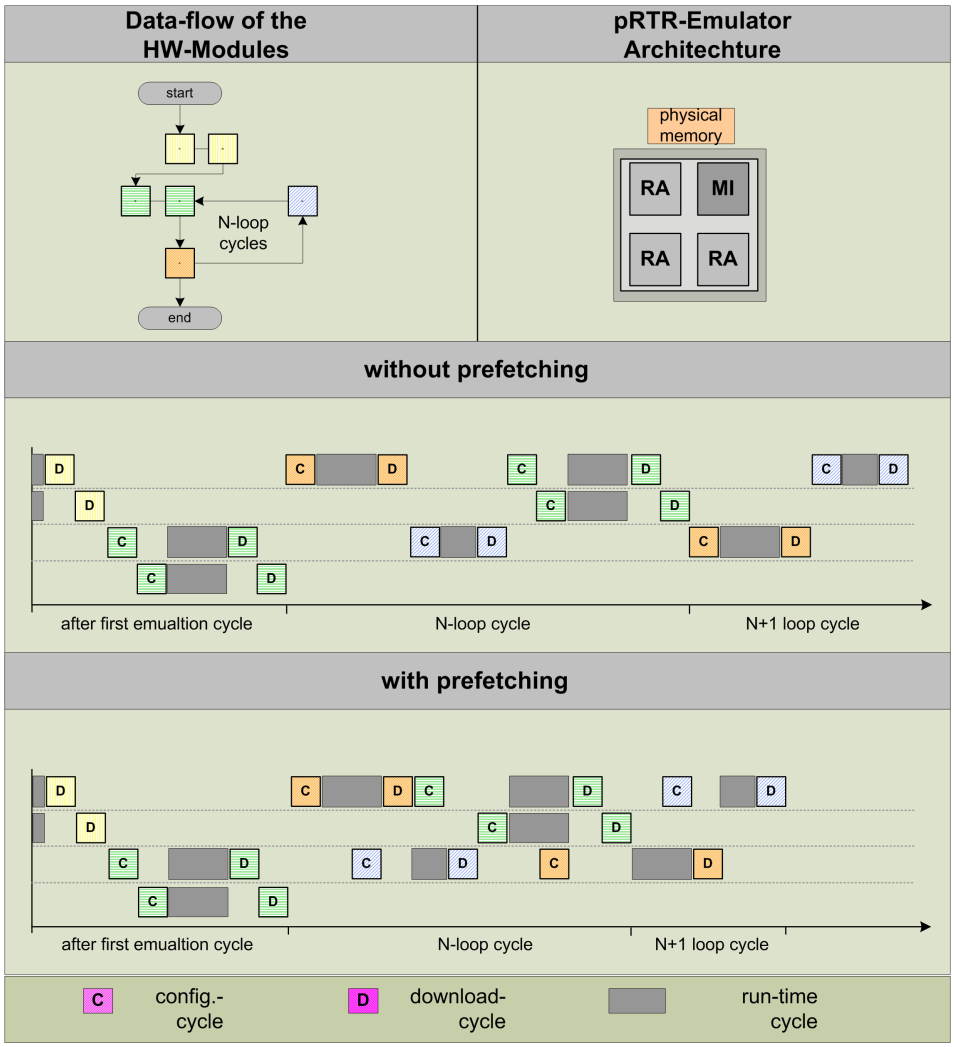


Abbildung 5.12: Optimierungspotential durch Anwendung von Prefetching

In den folgenden Emulationen (vgl. Abb. 5.12) erfolgt die Bestimmung und Rekonfiguration für ein HW-Module bereits während sich die aktiven HW-Module in der Emulationsphase befinden. Dieser Ansatz zur Reduzierung des Rekonfigurationsflaschenhalses dient der Anregung für weitere Arbeiten und wurde nur in der Implementierung des Softwaremodells für den Emulator berücksichtigt. Die Effizienz des Ansatzes steigt signifikant mit einer zunehmenden Anzahl an rekonfigurierbaren Flächen und langen partiellen Emulationsphasen der HW-Module. Im optimalen Fall ist es möglich, die Rekonfigurationszeit vollständig in die Emulationsphasen zu verlagern. Die dadurch gesteigerte Performanz für die Emulation eines HW-Moduls nähert sich der Performanz, die ein konventioneller HW-Emulators für ein äquivalentes HW-Modul benötigen würde (vgl. Abs. 6.3).

5.7 Zusammenfassung

Die Steuerung des partiell rekonfigurierbaren Emulators basiert auf einer angepassten Hardware-Softwarekopplung. Der Softwareteil wird auf einem *PowerPC* berechnet, welcher auf einem zweiten FPGA implementiert ist. Die Befehlssequenzen werden im Anschluss an ein Steuerwerk auf dem Emulator-FPGA gesendet. Das Steuerwerk kontrolliert ein Emulator-*Framework*, welches taktgenau die Emulation überwacht und aus einem hierarchischen Kommunikationsnetzwerk besteht. Dieses verbindet auf der untersten Ebene die für die Rekonfiguration notwendigen Flächen direkt miteinander. Die Festlegung der Verbindungen wird vor der Generierung des Emulator-*Frameworks* getroffen. Durch die Möglichkeit, eine Auswahl mehrerer Verbindungen festlegen zu können, bleibt die Flexibilität bei gleichzeitigem optimierten Ressourceneinsatz gewährleistet. Die Umschaltung zwischen den getroffenen Verbindungen wird durch Multiplexer erreicht. In der zweiten Ebene des Kommunikationsnetzwerks werden mehrere statisch verbundene rekonfigurierbare Flächen über ein Bussystem verbunden. Dieses Bussystem verarbeitet die angeschlossenen Knoten durch ein Zeitmultiplexverfahren, in welchem alle angeschlossenen Knoten gleichberechtigt sind. In dieser zweiten Ebene erfolgt ebenfalls der Zugriff auf die physikalischen Speicherbänke. Für die weiteren Ebenen des Kommunikationsnetzwerks wurde ein globales Bussystem implementiert, welches auf einer *Token*-Ring-Topologie basiert. Das globale Bussystem ist hinsichtlich der Wortbreite skalierbar und verbindet in der umgesetzten Implementierung das Emulator-*Framework* mit dem *Host*-System.

Für die Detektion der zeitlichen Ereignisse zum Rekonfigurieren der HW-Module wurde ein HW-Monitor entworfen und implementiert, der an den Schnittstellen der HW-Module einen Schreibvorgang erkennt. Der HW-Monitor schaltet selbstständig die Funktion der HW-Module ab und sendet ein *Interrupt*-Signal an das Steuerwerk. Aus diesen Informationen und einer Liste weiter zu bearbeitender HW-Module wird durch einen Algorithmus, welcher auf dem *PowerPC* berechnet wird, entschieden, ob ein HW-Modul ausgetauscht wird oder im Emulator-*Framework* verbleibt.

Unter Berücksichtigung spezieller Hardwarearchitekturen wird eine deutliche Erhöhung der Effizienz des *Frameworks* erreicht. Durch die Ausnutzung der regulären Eigenschaften von Multi-Prozessor Systemen, wird die Anzahl der notwendigen Rekonfigurationen deutlich reduziert. Für eine funktional vollständige Emulation dieser Architekturen wird der Datenfluss mehrfach durch den hinsichtlich Komplexität reduzierten parallelen Systemteil geführt. Die Steuerung setzt sich aus angepassten Befehlssequenzen für die Emulation von speziellen HW-Architekturen zusammen, welche durch das Steuerwerk des Emulator-*Framework* im Hardwareteil unterstützt werden.

Die bisher umgesetzten Teile des implementierten *Frameworks* umfassen alle für die Steuerung notwendigen HW-Komponenten. Darüber hinaus wurden unterschiedliche *Routing*-Topologien hinsichtlich des Ressourcenverbrauchs und der Flexibilität untersucht (vgl. Abs. 6.10). Des Weiteren ist durch einen optimierten Vorhersagealgorithmus für den Zeitpunkt einer Rekonfiguration eine signifikante Reduzierung des zeitlichen Aufwands der Rekonfiguration möglich. Durch Anwendung von *Prefetching*-Verfahren wird eine parallele Ausführung von Rekonfigurationen bei zeitgleicher Emulation weiterer HW-Module effizienter.

Zusammenfassend haben die Implementierungen des Emulator-*Frameworks* gezeigt, dass durch ein angepasstes Kommunikationsnetzwerk und eine optimierte Laufzeitsteuerung die Effizienz des gesamten Ansatzes gesteigert wird. Eine detaillierte Bewertung der Implementierungen des Kommunikationsnetzwerkes und des gesamten neuen Ansatzes wird im folgenden Kapitel 6 diskutiert.

6 Ergebnisse und Entwurfsbeispiele

Im folgenden Kapitel werden die Ergebnisse der Implementierung und Umsetzung dargestellt. Zunächst werden Aussagen über das Partitionierungsverfahren anhand zweier Beispiele diskutiert. Im zweiten Teil stehen die Ergebnisse des Entwurfs und der Synthese des Emulator-*Frameworks* für verschiedene Parametervariationen im Mittelpunkt. Ausgehend von diesen Ergebnissen werden im dritten Abschnitt Laufzeitergebnisse der zwei Beispiele beschrieben. Zum Abschluss wird das neue Emulationsverfahren hinsichtlich der Effizienz bewertet.

6.1 Ergebnisse des Partitionierungsverfahrens

Im folgenden Abschnitt werden Ergebnisse aus der Partitionierung von zwei HW-Entwürfen dargestellt. Die Partitionierung der Entwürfe erfolgt zunächst unter funktionalem Kontext. Als Beispielentwürfe wurden ein *Reed-Solom-Decoder* (RS - Reed Solomon) [79] und ein Signal-Prozessor-SoC herangezogen.

Das erste Beispiel ist ein *RS-Decoder*, welcher einen kontinuierlichen Datenfluss aufweist (vgl. Abb. 8.4). Als zweites Beispiel wurde ein Signal-Prozessor-SoC bewertet, welches aus einer heterogenen Vielzahl von Spezialprozessoren besteht, welche über den Speicher untereinander kommunizieren. Für die Bewertung des Emulationsverfahrens wurde das SoC hinsichtlich der Komplexität zunächst auf wesentliche Komponenten reduziert.

6.1.1 *RS-Decoder*

Der *RS-Decoder* hat die Parameter (31,19,6). Dieser *RS-Code* ist charakterisiert durch eine Zahl von 31 Symbolen in einem Block. In diesen 31 Symbolen sind 19 Datenworte und 12 Symbole für die Redundanz des *Codes* enthalten.

Aus der Differenz der Symbole und der enthaltenen Datenworte lässt sich die maximale Anzahl erkennbarer Fehler von 12 Fehlern ableiten. Aus den erkannten fehlerhaften Datenworten ist der *RS-Decoder* in der Lage, 6 Fehlerworte zu korrigieren. Die Wortbreite eines Symbols beträgt 5 Bit. In Abbildung 8.4 ist der *RS-Decoder* mit den enthaltenen *Top-level*-Modulen schematisch dargestellt.

Zunächst wird für den eingehenden Block (*Frame in*) das Syndrom im Modul *SC-block* bestimmt. Ist das Syndrom ungleich 0, wird der Block im internen Speicher (*FIFO_register*) abgelegt. Nach der Berechnung des Fehlerpolynoms im *KES-block* werden dessen Nullstellen, welche die Fehlerstellen repräsentieren im *CSEE-block* bestimmt. Die Korrektur der verfälschten Symbole wird im Modul *Error Correction* durchgeführt. Dieses Modul führt unter Beachtung des für den *Code* aufgespannten *Galois*-Feldes eine Addition des Fehlerwertes mit dem fehlerhaften Symbol durch. Der korrigierte Block (*Frame out*) wird im Anschluss ausgegeben.

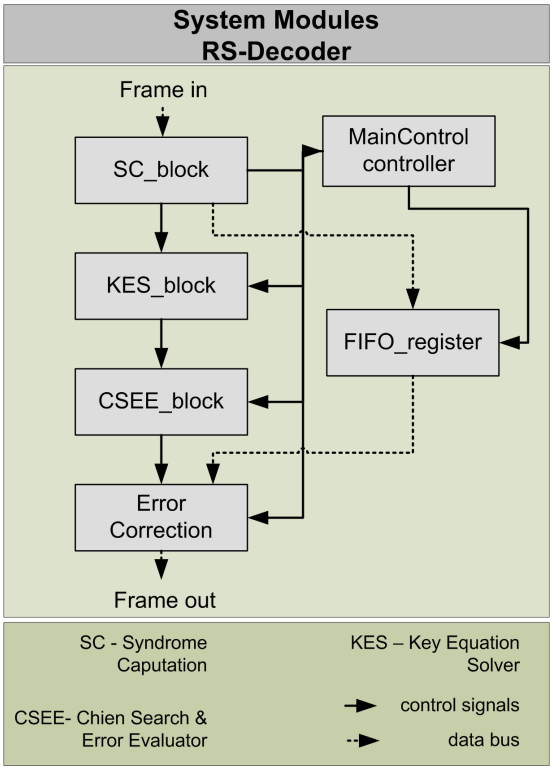


Abbildung 6.1: Blockschaltbild RS-Decoder

Der Entwurf ist mit dem Entwicklungswerkzeug *HDL-Designer* elaboriert und mit der Software *Precision* synthetisiert worden. Während der Elaborierungsphase hat die Software 40 Komponenten detektiert. Zusätzlich zu den funktionalen Komponenten ist im Entwurf eine *Testbench* enthalten. Die Komponenten sind in fünf *Top-level*-Modulen integriert. In Anhang 8.4 ist das elaborierte Blockschaltbild des *RS-Decoder* abgebildet, welches zusätzlich ein sechstes Modul (*gf_adder*) enthält. Dieses Modul, welches in Abbildung 8.4 dem Modul *Error Correction* entspricht, ist nicht als *Top-level*-Modul ausgewiesen und geht während der Synthese des gesamten Entwurfs mit in den *GSEE-block* ein. Wie in Abbildung 4.3 dargestellt, werden zunächst die *Top-level*-Module separiert und nach einer Erweiterung um die *slice macro* als einzelne Module synthetisiert.

Als Zielarchitektur für die Synthese wurde die *Virtex4*-Technologie ohne Optimierung zu Grunde gelegt. In Abbildung 6.2 sind die Ergebnisse der Synthese nach der funktionalen Partitionierung dargestellt.

Für die Synthese des Moduls *KES-block* wird die größte Anzahl an Ressourcen benötigt. Dieses Modul enthält keine weiteren Sub-Module und ist aus diesem Grund unter funktionalen Gesichtspunkten nicht weiter partitionierbar. Die Parameter für die Dimensionierung des Emulator-*Frameworks* müssen von diesem Modul abgeleitet werden, da es sich um das hinsichtlich der CLB-Anzahl größte Modul handelt. Eine rekonfigurierbare Fläche innerhalb des *Frameworks* muss also für die Platzierung aller HW-Module des Beispiels mindestens 548 CLB-*Slices* enthalten.

Ein weiterer relevanter Parameter ist die Anzahl der Eingangs- und Ausgangsleitungen der rekonfigurierbaren Flächen. Das Modul mit dem zahlenmäßig größten CLB-Anteil muss nicht zwingend das Module

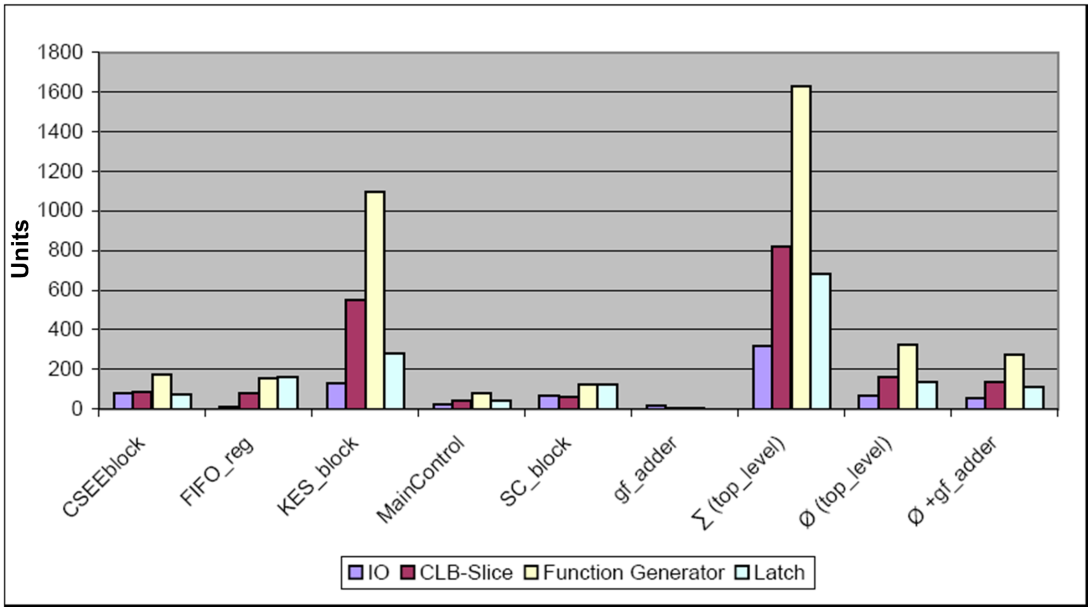


Abbildung 6.2: Syntheseergebnisse RS-Decoder nach Partitionierung

mit der größten Anzahl an Ein- und Ausgängen sein. Des Weiteren kann sich das Modul mit der größten Anzahl an Eingängen vom Modul mit der größten Anzahl an Ausgängen unterscheiden.

Im Beispiel des *RS-Decoders* ist der *SC-block* bestimmend für die Anzahl der Eingänge und das Modul *Main_Controller* für die Anzahl der Ausgänge. Für die Synthese des Emulator-*Frameworks* werden für das *RS-Decoder* Beispiel 12 Eingangsdatenleitungen und 16 Ausgangsdatenleitungen einer rekonfigurierbaren Fläche festgelegt.

Zur Bewertung des Partitionierungsverfahrens auf strukturaler Ebene wurde das Modul *KES-block* zu Grunde gelegt. Dieses Modul weicht nach der Bewertung der Kosten hinsichtlich der Anzahl an CLB am deutlichsten von den weiteren Modulen des *Reed-Solom-Decoders* ab.

Zunächst erfolgte eine Synthese mit der Software *Leonardo Spectrum*, welche in die Skriptensteuerung des Partitionierungsablaufs integriert ist (vgl. Abs. 4.2). Als Zielarchitektur wurde ein *Virtex*-FPGA ausgewählt.

In Abbildung 6.3 sind die Analyseergebnisse des gesamten Entwurfs und des separaten *KES-block*-Moduls dargestellt. Ausgehend von den *Ports*, welche die Spitzen der partitionierbaren Kegel (*Cone*) repräsentieren, wurde die strukturale Aufteilung des HW-Moduls vorgenommen. Folgende Ergebnisse sind während der strukturalen Partitionierung des *KES-block*-Moduls berechnet worden:

- Unter Verwendung aller enthaltenen *Cone*-Strukturen ist die Bildung von 209 teilweise abhängigen Knoten realisierbar.
- Aus diesen Knoten lassen sich 10 unabhängige *Cluster* bilden.
- Eine Zusammenführung von *Clustern,* ohne der Beachtung von gegenseitigen Abhängigkeiten zu anderen HW-Modulen, führt während der Emulation zu einer großen Anzahl an Rekonfigurationen.

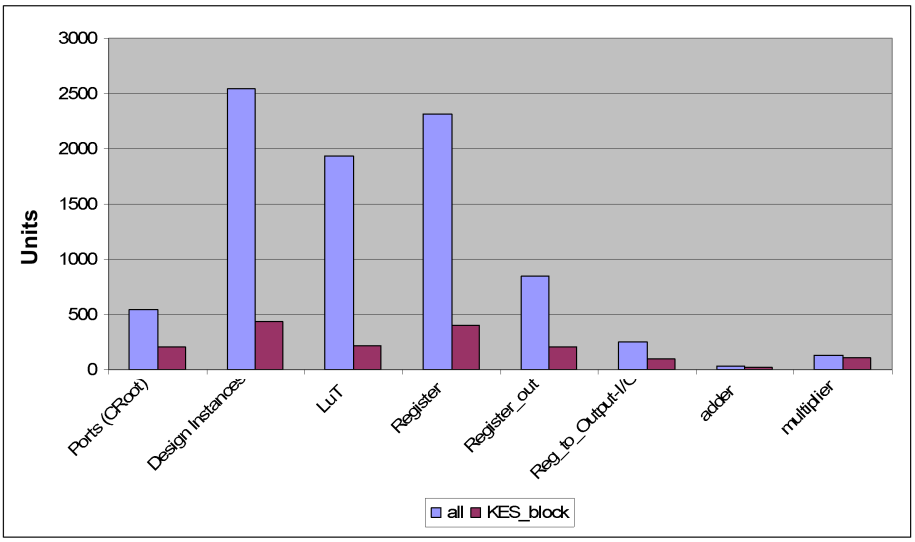


Abbildung 6.3: Syntheseergebnisse Strukturale Partitionierung (*KES-block*)

Aus diesem Grund wurde als Einschränkung eine Kostenfunktion zu Grunde gelegt, welche die Bildung von Modulen mit äquivalenten Kosten funktional partitionierter HW-Module anstrebt. Die Kostenfunktion wird durch den Algorithmus mit der Bildung von zwei unabhängigen HW-Modulen erfüllt.

Nachteilig für die Anwendung des strukturalen Partitionierungsverfahrens ist der große Zeitbedarf, der für die Bildung der *Cluster* benötigt wird. Während ein strukturierter Entwurf, welcher unter Verwendung einer *Virtex*-Technologie (vgl. *stream_codec*) einen Ressourcenbedarf von fünf CLB hat in ca. 20 ms partitioniert wird, benötigt der Algorithmus für eine Vergleichspartitionierung des *Reed-Solomon-Decoders* auf struktureller Ebene sieben Minuten und für das Modul *KES_block* ca. 1,5 Minuten.

Der Zeitbedarf für die strukturale Partitionierung hängt neben der Größe des Entwurfs direkt mit dessen Struktur zusammen. Enthält der Entwurf z.B. eine große Anzahl kombinatorischer Logikteile, entsteht eine beträchtliche Anzahl an Kegelspitzen (vgl. Abs. 3.2.2), welche maßgeblich bestimmend für die Komplexität der Partitionierungsaufgabe ist.

6.1.2 Signal-Prozessor-SoC

Die Partitionierung des Signal-Prozessor-SoC wird zunächst auf der *Top-level*-Ebene durchgeführt. Als Synthesesoftware für die Berechnung der Kosten wurde wieder *Precision* von *Mentor Graphics* eingesetzt. Die Entwurfsdaten des SoC sind von einem externen Entwickler zur Verfügung gestellt worden, weshalb im Unterschied zur Bewertung des *RS-Decoders* zunächst ein *StratixII-EP2S180*-FPGA von *Altera* zu Grunde liegt. Dieses FPGA ist vergleichbar mit einem *Virtex4-LX200*-FPGA der Firma *Xilinx*. Um die Syntheseergebnisse weiterhin auf die *Virtex4*-Architektur beziehen zu können, wurden eine Umrechnung unter Beachtung der Äquivalenz beider FPGA-Architekturen [5] durchgeführt.

Der Algorithmus hat ausgehend von der *Top-level*-Ebene des Entwurfs (vgl. Abb. 6.4) zwei weitere Sub-Modul-Ebenen elaboriert. In der ersten Ebene enthält der Entwurf alle Komponenten für die interne

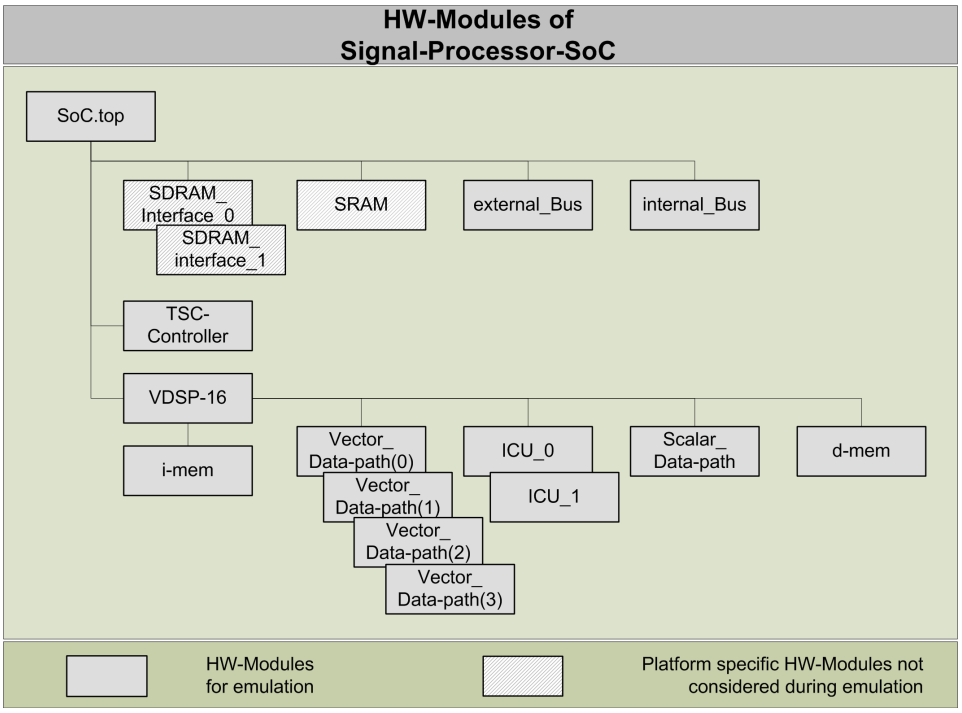


Abbildung 6.4: HW-Module des Signal-Prozessor-SoC

und externe Busanbindung, die Speicherschnittstellen (*SDRAM_interface*), und den internen SRAM. Des Weiteren sind die Module *TSC-Controller,* und *VDSP-16 Top-level* Module.

Das Modul *VDSP-16* enthält die weiteren Sub-Module *Vector Datapath (0..3)*, *ICU(0-1)*, den Skalar Datenpfad (*Scalar Data-path*) und den Datenspeicher (*d-mem*). Zusätzlich zu den Sub-Modulen ist der Befehlsspeicher (*i-mem*) als weiteres Modul integriert.

Die externen Speicher, deren Controller in die *Top-level*-Ebene integriert sind, werden in der Emulation nicht als HW-Module berücksichtigt. Die Funktion wird durch die QDR-RAM-Speicher des Emulators ersetzt.

In Abbildung 6.5 sind die elaborierten Ressourcen nach der Partitionierung dargestellt. Das Bussystem, welches die *Top_level*-Module miteinander verbindet, wird auf Grund der Abhängigkeiten zwischen dem internen- und externen Bus in einem HW-Modul zusammengefasst. Der *TSC-Controller,* welcher die die *Top-level*-Module und die Interaktion des SoC mit der Peripherie steuert, bildet zunächst ein separates HW-Modul. Dieser Controller ist als IP-Core in das System integriert. Aus diesem Grund ist eine weitere funktionale Partitionierung nicht möglich.

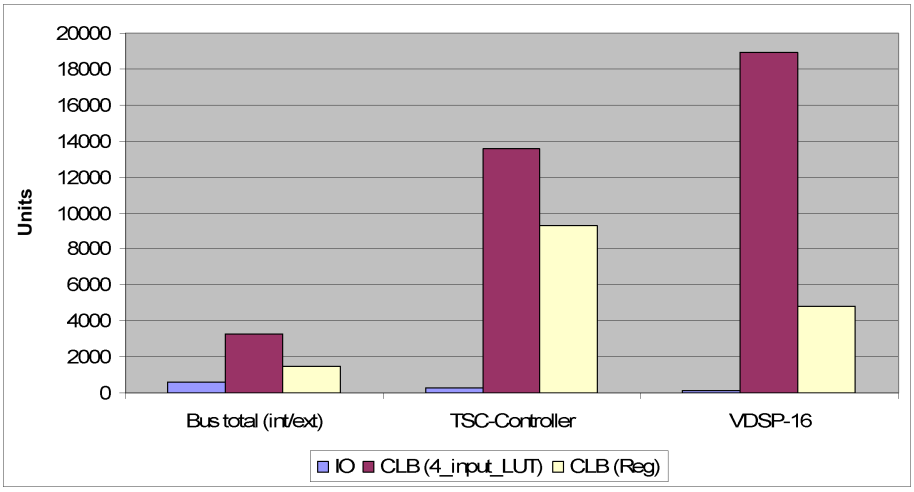


Abbildung 6.5: Funktionale Partitionierung Top_Level SoC

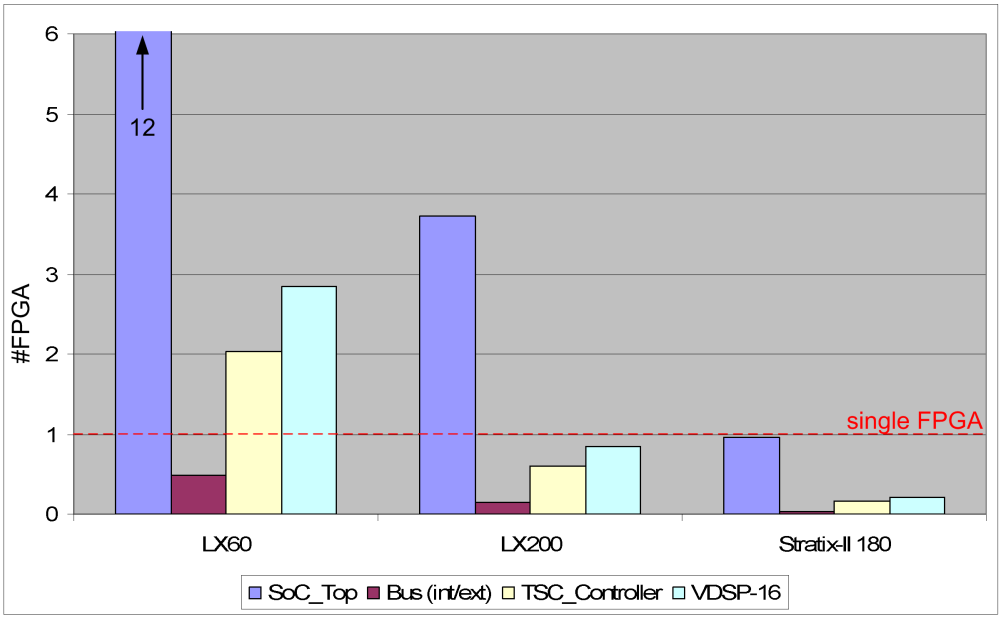


Abbildung 6.6: Ressourcebedarf für die Top_Level Emulation des SoC

In Abbildung 6.6 ist der Bedarf an FPGAs für die Emulation der *Top_level*-Module des SoC abgebildet. Die Synthese wurde für einen *Static-II-180* FPGA optimiert, was den signifikanten Unterschied für den Ressourcenbedarf des vergleichbaren *Virtex4-LX200* zum *Static-II 180* FPGA begründet.

Das mit Abstand größte HW-Modul in der *Top-level*-Ebene bildet das *DSP-16*-Modul. Dieses Modul wird in einem weiteren funktionalen Partitionierungsschritt in die in dem Modul integrierten Sub-Module unterteilt. Wie in Abbildung 6.4 dargestellt, enthält das *VDSP-16*-Modul mehrere *Data-path*-Sub-Module,

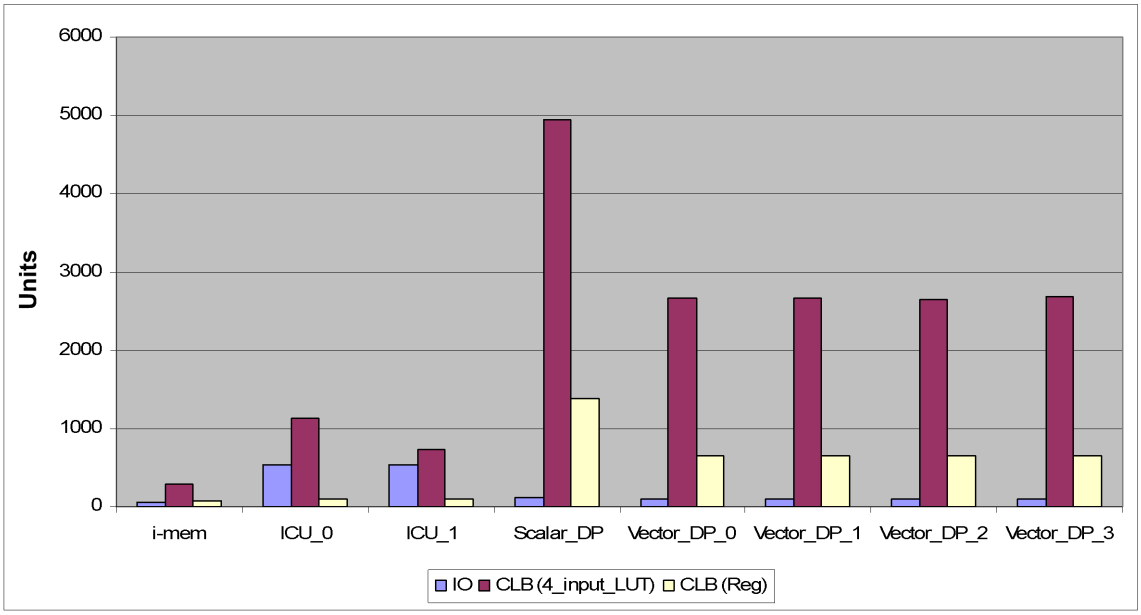


Abbildung 6.7: Funktionale Partitionierung des VDSP-16 Modules

zwei *ICU*-Module und zwei Speicher-Module. Die Ergebnisse der nach der Partitionierung und Synthese bewerteten HW-Module sind in Abbildung 6.7 abgebildet.

Die vier funktionsgleichen *Vector_Data-path*-Module bilden nach der funktionalen Partitionierung hinsichtlich Größe und Anzahl der *IOs* äquivalente HW-Module. Der *Scalar_Data-path* weicht dagegen wesentlich in der Anzahl der benötigten CLBs von den weiteren HW-Modulen in der Sub-Modul-Ebene des *VDSP-16*-Moduls ab. Die *ICUs* sind bezüglich der Kosten vergleichbar, haben aber eine unterschiedliche Funktionalität was eine verhaltensgleiche Emulation dieser Module nicht zulässt. Die Speicher werden als eigenständige HW-Module emuliert.

Aus den Ergebnissen der funktionalen Partitionierung lassen sich folgende Parameter für die Dimensionierung des Emulator-*Frameworks* für einen *Virtex4-LX60* FPGA ableiten:

- In die funktionale Emulation der *Top-level*-Module kann unter Verwendung eines *Virtex4-LX60* FPGA nur das HW-Modul des Bussystems einbezogen werden. Der *TSC-Controllers* lässt sich auf einem *LX60*-FPGA ohne eine weitere Partitionierung nicht platzieren. Diese Partitionierung ist nur noch auf strukturaler Ebene durchführbar, da der TSC-Controller als synthetisierter *IP-Core* funktional nicht weiter zu partitionieren ist.

- Die strukturale Partitionierung des *TSC-Controllers* führt zu 17730 unabhängigen Kegeln. Aus diesen werden drei hinsichtlich Größe und Anzahl der IOs vergleichbare HW-Module in Clustern zusammengefasst und zu HW-Modulen erweitert.

- Eine rekonfigurierbare Fläche des Emulator-*Frameworks* zur partiellen Emulation des SoC hat eine Größe von 2570 CLB. Auf dem Virtex4-LX60 lassen sich demnach 2 rekonfigurierbare Flächen platzieren.

- Die Wortbreite für das *Framework* beträgt 64 Signalleitungen zwischen den RAs.

Die funktionale Partitionierung des Signal-Prozessor-SoC veranschaulicht die Ergebnisse der Partitionierung einer wesentlichen Gruppe von Entwürfen. Neben den heterogenen HW-Modulen in der *Top-level*-Ebene sind in der Ebene der Sub-Module eine Reihe von regulären Architekturen elaboriert worden. Diese unterschiedlichen Architekturteile müssen, um eine effiziente funktionale Emulation des Entwurfs zu erreichen, während der Emulation berücksichtigt und durch das Kommunikationsnetzwerk des Emulators abgebildet werden.

6.2 Synthese- und Implementierungsergebnisse des Emulator-*Framework*

Die Bewertung der Implementierungsergebnisse ist in zwei Abschnitte untergliedert. Im ersten Abschnitt werden die Ergebnisse zur Ermittlung einer optimalen und an die Problemstellung angepassten *Routing*-Topologie dargestellt. Die Ergebnisse aus der konkreten Implementierung unter Berücksichtigung verschiedener Parameter für die Dimensionierung des Emulator-*Frameworks* sind Teil des zweiten Abschnitts.

6.2.1 Dimensionierung der *Routing*-Topologie

In Vorbereitung der Implementierung des Emulator-*Frameworks* wurden Untersuchungen bzgl. des Kommunikationsaufwands durchgeführt. Den Untersuchungen liegt ein Modell der Zielarchitektur (*Virtex4*) und möglicher Verbindungsstrategien zu Grunde.

Als ein Ergebnis wurde eine Abschätzung des Ressourcenaufwands für unterschiedliche Kommunikationstopologien und *Routing*-Verfahren bestimmt. Das umgesetzte *On-Chip*-Netzwerk (vgl. Abs. 5.3.2) ist aus den Parametern dieser Bewertung abgeleitet. Zunächst gingen die Untersuchungen von uneingeschränktem *Routing* zwischen den rekonfigurierbaren Flächen aus. Dieses Verfahren wurde mit *Clos*-Netzwerken unterschiedlicher Topologien verglichen. In Abbildung 6.8 ist zu erkennen, dass der Aufwand für uneingeschränktes Routing zwischen allen rekonfigurierbaren Flächen (RA) die Ressourcen des zu Grunde liegenden *Virtex4*-FPGA (LX60) ab einer Anzahl von 10 Signalen bei mehr als 4 RAs übersteigt. Aus der Abbildung geht des Weiteren hervor, daß die Anzahl der Signale eine größere Rolle hinsichtlich des Ressourcenbedarfs einnimmt, als die Anzahl der rekonfigurierbaren Flächen. Dieser Zusammenhang ist wesentlich durch die Platzierung der Multiplexer gestützt, welche für das *Routing* der Signale notwendig sind. Der Ressourcenbedarf der Multiplexer steigt linear mit zunehmender Anzahl an Signalen. Wird zusätzlich die Anzahl an rekonfigurierbaren Flächen erhöht, steigt der Ressourcenbedarf exponentiell. Dieser Zusammenhang ist in Abbildung 6.8 am Beispiel des uneingeschränkten *Routing* zu erkennen. Der Bedarf an CLB für die Implementierung der Multiplexer ist in Abbildung 6.9 dargestellt. Das Verhältnis an notwendigen CLB zum Verbinden einer bestimmten Anzahl von Signalleitungen wurde für eine *Virtex*-Architektur ohne Einbeziehung der auf dem FPGA fest platzierten Multiplexer ermittelt.

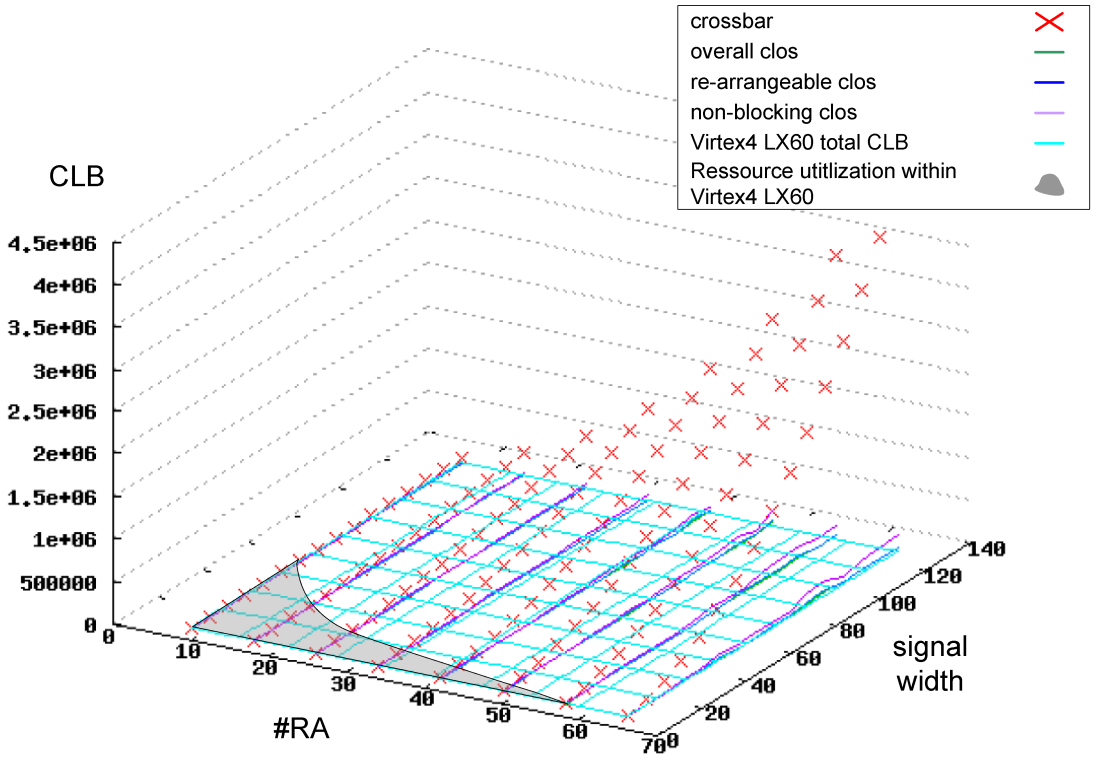


Abbildung 6.8: Ressourcenbedarf für unterschiedliche Routing-Topologien

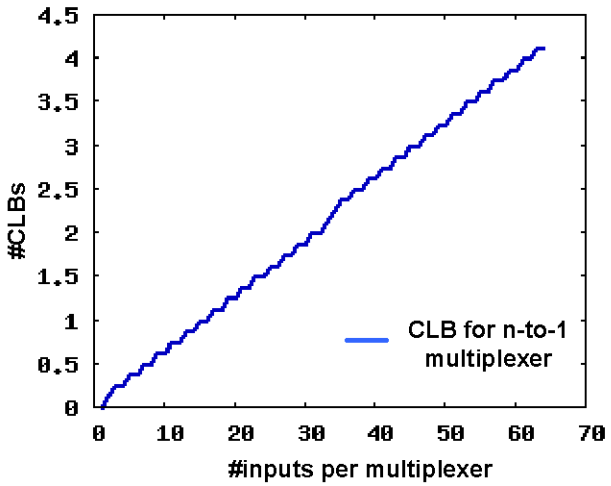


Abbildung 6.9: Ressourcenaufwand für Multiplexer-Implementierung

Der Bedarf an Verbindungs- und Kommunikationsressourcen wird durch die Einbeziehung eines hierarchischen Netzwerks für die Informationsübertragung signifikant gesenkt. In Abbildung 6.10 ist die Res-

sourcenallokation für unterschiedliche Topologien unter Einbeziehung von zwei Realisierungen, welche auf *Routing*-Hierarchien basieren (vgl. Abb.6.10) dargestellt.

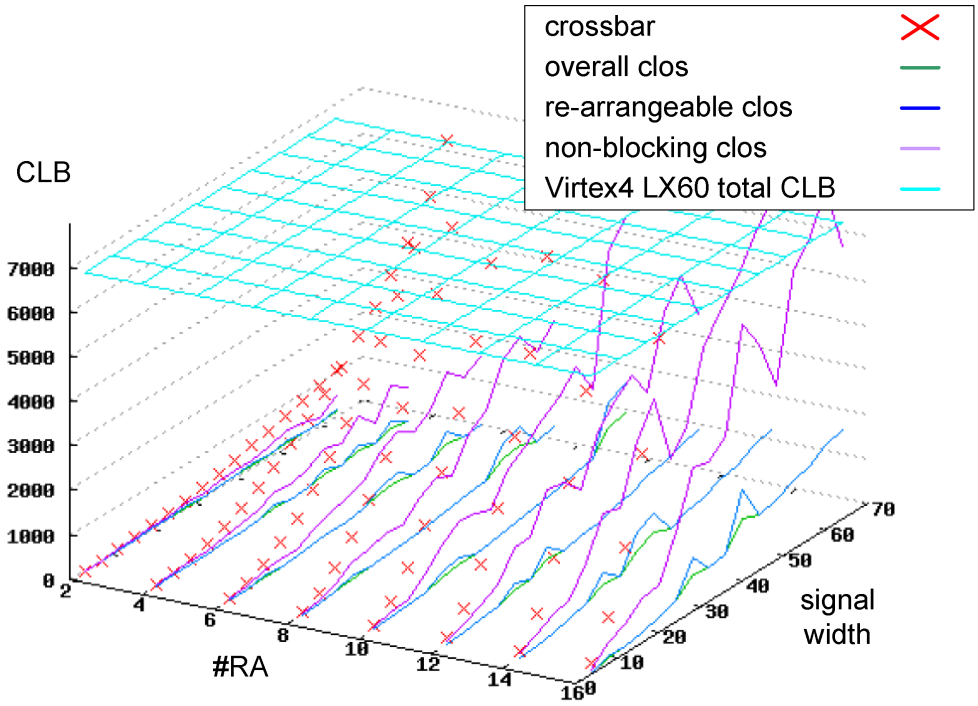


Abbildung 6.10: Ressourcenbedarf der umgesetzten heterogenen *Routing*-Topologie

Für eine ressourceneffiziente Umsetzung der *Routing*-Topologie wurde ausgehend von den Ergebnissen der Simulation ein heterogenes Kommunikationsnetzwerk implementiert, welches aus einem lokal verbundenen HCGP-Netzwerk und einem global kommunizierenden *Partial-Crossbar* besteht. In Abbildung 6.10 ist der deutliche Vorteil dieser Implementierung zu erkennen. Der gesamte Ressourcenbedarf der implementierten Topologie wird unter Berücksichtigung einer äquivalenten Erreichbarkeit der einzelnen rekonfigurierbaren Flächen gegenüber klassischen *Clos*-Netzwerken (*non-blocking clos networks*) und übergreifenden *Partial-Crossbar* um ca. 71% gesenkt.

6.2.2 Synthese-Ergebnisse für verschiedene Parametervarianten

Für die Implementierung und Inbetriebnahme des Emulator-*Frameworks* sind unterschiedliche Parametervarianten synthetisiert worden. Dafür wurde die HDL-Beschreibung des *Frameworks* hinsichtlich der Anzahl der rekonfigurierbaren Flächen, der Signalbreite und der Topologien des *On-Chip*-Netzwerk skalierbar umgesetzt. Die Breite der Signalbusse steht in direktem Verhältnis zur Busbreite des PCI-X-Busses, welcher mit 64-Bit festgelegt ist. In Tabelle 6.1 sind die Faktoren und die daraus resultierenden Wortbreiten des *On-Chip*-Busses gegenübergestellt. Für repräsentative Ergebnisse ist die Synthese des Emulator-*Framework* für die Busverhältnisse 1, 2 und -16 umgesetzt worden.

In den Diagrammen in Abbildung 6.11 ist der Ressourcenbedarf für eine unterschiedliche Anzahl an rekonfigurierbaren Flächen im Verhältnis zu den festgelegten Signalbreiten abgetragen. Es wurden Syn-

Faktor im Verhältnis zu 64-Bit	-16	-8	-4	-2	1	2	3	4
resultierende Wortbreite *On-Chip*-Bus	4	8	16	32	64	128	196	256

Tabelle 6.1: Verhältnisse der *On-Chip*-Busbreite zum PCI-X-Bus

theseergebnisse bis zu einer Anzahl von 64 RAs bestimmt.

Grundsätzlich steigt mit einer Verdopplung der rekonfigurierbaren Flächen der Ressourcenbedarf bei konstanter Wortbreite exponentiell an. Das Verhältnis an verwendeten HW-Komponenten zu einer steigenden Anzahl an Signalleitungen ist linear. Mit zunehmender Signalbreite (vgl. 6.11 128 und 256 Bit) ist die Synthese für eine größere Anzahl von RAs nicht mehr durchführbar. In diesem Fall ist eine Abbildung der HDL-Beschreibung durch das Synthesewerkzeug für den zu Grunde liegenden FPGA nicht realisierbar. In drei Diagrammen ist die Abhängigkeit der *Framework*-Parameter von der Anzahl der eingesetzten *LuT* und der *Slice-FlipFlop* dargestellt. Das Verhältnis der *LuT*- und der Anzahl der *Slice-FlipFlop* zu einem *CLB* der *Virtex4*-Technologie beträgt acht. Die Zunahme sowohl der *LuT*, in welchen die Multiplexer abgebildet werden, als auch der *Slice-FlipFlop*, welche die Register abbilden ist konstant.

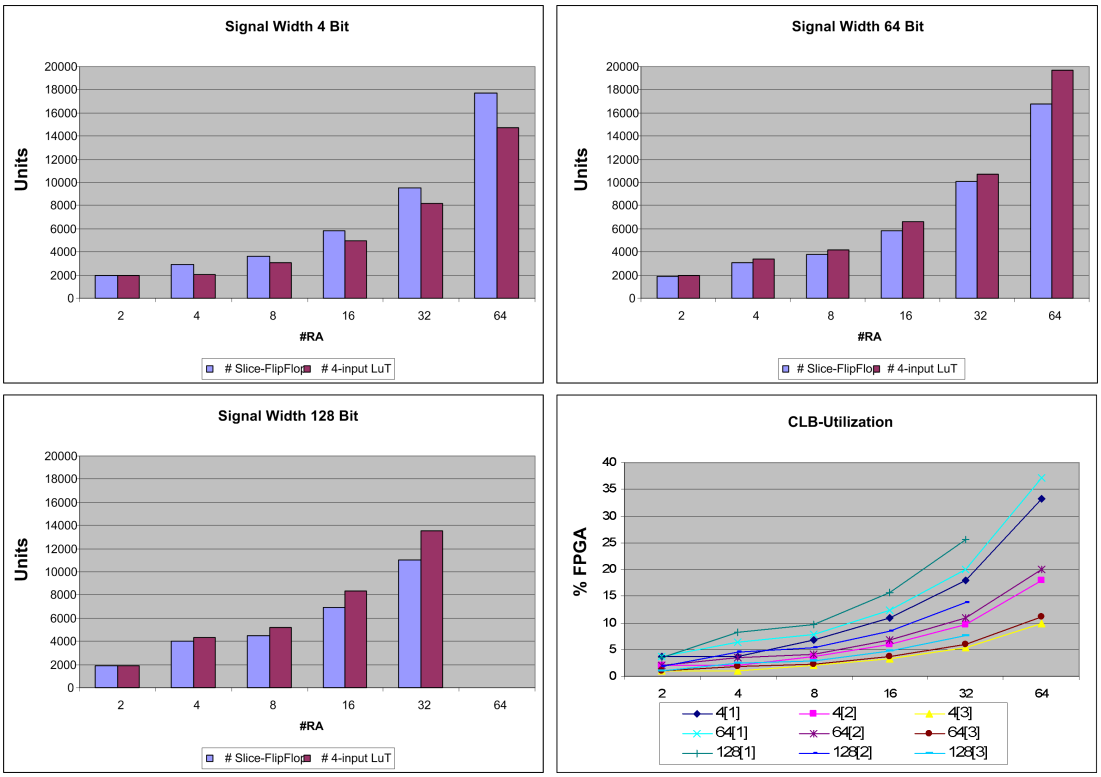


Abbildung 6.11: Ressourcenbedarf für Emulator-*Framework* (Wortbreite 4 Bit)

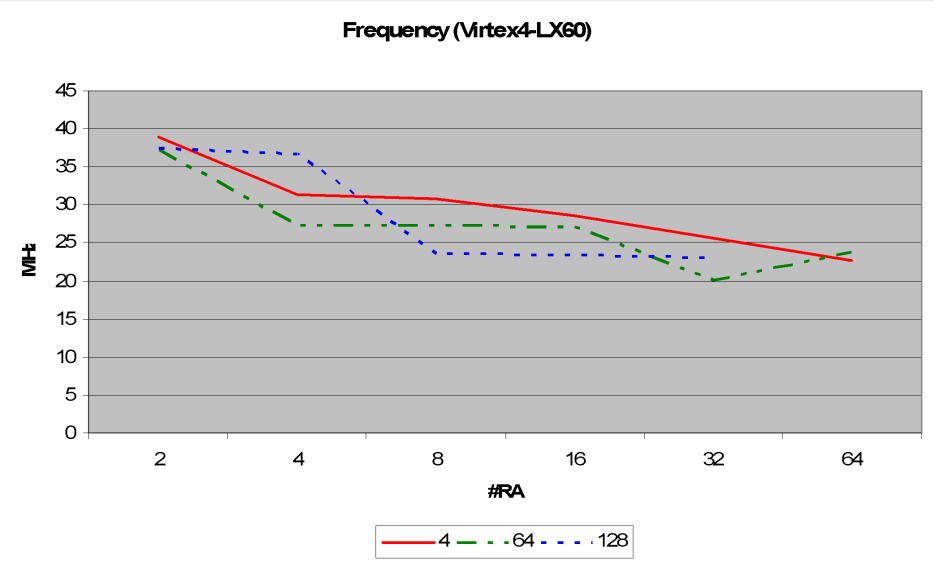


Abbildung 6.12: Abhängigkeit der Frequenz zu den *Framework* Parametern

Im vierten Diagramm in Abbildung 6.11 ist die Auslastung unterschiedlich großer FPGAs der *Virtex4-LX*-Serie gegenübergestellt. Die Gesamtauslastung der FPGAs ist analog zur Darstellung des Ressourcenbedarfs der *Routing*-Topologie in CLB angegeben. Die Auslastung des für die Implementierung eingesetzten *Virtex4-LX60*-FPGA beträgt in der größten Dimensionierung (64 RA bei 64 Bit Signalbreite vgl. Abb.6.11 *64[1]*) des *Frameworks* 37% des gesamten FPGAs.

Neben den Kosten für die Implementierung des Emulator-*Frameworks* ist die erreichbare Taktfrequenz für die Emulation eine entscheidende Größe zur Bestimmung der Effizienz des Ansatzes. Die Taktfrequenz wird im Wesentlichen durch die Anbindung der externen *QDR*-Speicher beeinflusst. Der Speicherzugriff wird über ein FIFO zwischen der Speicherbank und dem
Schnittstellen-*Controller* synchronisiert. Diese Synchronisierung führt zu einer erheblichen Latenzzeit und mindert die erreichbare Taktfrequenz um mehr als 50%. Für den synchronen Datenaustausch zwischen den RAs und für eine Reduzierung der Komplexität ist in der implementierten Version des Emulator-*Frameworks* eine einheitliche Taktdomäne verwendet worden. Dadurch beeinflusst das *Framework*-Element mit der größten Latenzzeit die gesamte Performance.

Die erreichbare Systemfrequenz des *Frameworks* hängt unter Berücksichtigung des Diagramms in Abbildung 6.12 unwesentlich von den Parametern des *Framework* ab. Dies hat zwei Gründe:

- Erstens wird die erreichbare Frequenz des FPGAs von ca. 400-500 MHz deutlich unterschritten. Dadurch ist die Beeinflussung der Auslastung des FPGAs auf die Taktfrequenz sehr viel geringer als bei der maximal erreichbaren FPGA-Frequenz.

- Ein weiterer Grund ist die geringe Auslastung des FPGAs, welche 35% nicht übersteigt. Dadurch kann das Synthesewerkzeug die Netzliste des *Frameworks* optimal auf dem FPGA platzieren und verdrahten (*place&route*).

Die Dimensionierung und Synthese des Emulator-*Frameworks* wird gemäß des Ablaufplans des neuen Ansatzes (vgl. Abs. 3.4 nach der Partitionierung des Entwurfs durchgeführt. Die dafür benötigte Zeit muss

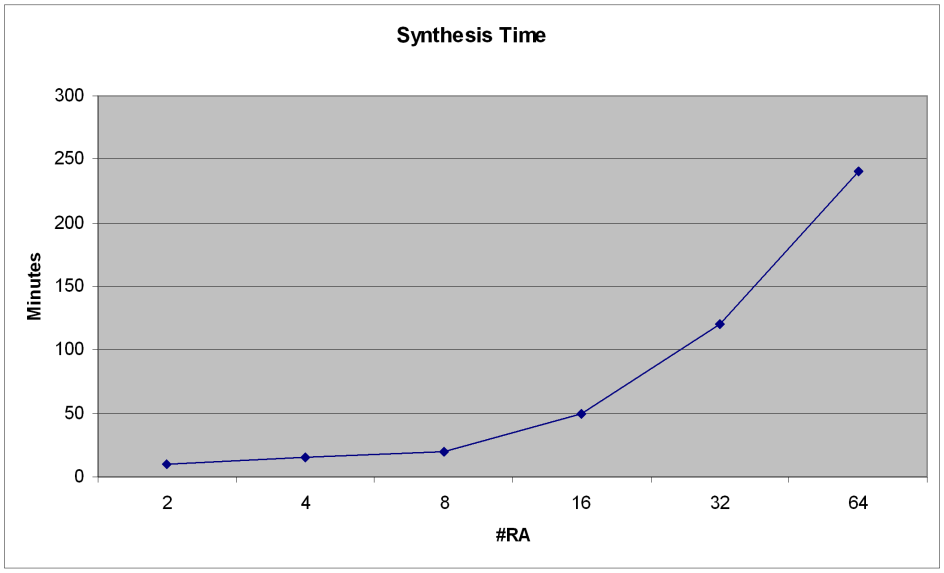


Abbildung 6.13: Zeitbedarf für Synthese in Abhängigkeit der Parameter

in der gesamten Verifikationszeit berücksichtigt werden und ist ebenfalls bestimmend für die Effizienz des gesamten Ansatzes. In Abbildung ist der Zeitbedarf 6.13 für die Synthese des *Frameworks* für eine unterschiedliche Anzahl an rekonfigurierbaren Flächen abgetragen. Es ist ein quadratischer Zusammenhang zwischen den Größen zu erkennen. Die gemessene Zeit wurde für die Synthese auf einem *AMD Athlon 64 X2 6400+* mit der Synthesesoftware *ISE* der Firma *Xilinx* ermittelt.

Der Zeitbedarf ist für eine industrielle Anwendung des Ansatzes wesentlich zu groß. Aus diesem Grund muss das *Framework* mit vorbestimmten Parametern synthetisiert werden. Aus einer Datenbank wird nach der Partitionierung des Entwurfs die entsprechende Umgebung ausgewählt und implementiert. Dies beeinflusst wiederum die Kostenfunktion der Partitionierung, welche dadurch unter Berücksichtigung gegebener Parameter durchgeführt werden kann.

6.3 Laufzeitergebnisse

Neben den statischen Ergebnissen für die Bewertung des Ansatzes werden im folgenden Abschnitt Ergebnisse zu den Laufzeiteigenschaften dargestellt. Für eine Reduzierung des Zeitaufwands wurden die Ergebnisse unter Verwendung eines in *Matlab* implementierten Simulators für die Abbildung rekonfigurierbarer Architekturen ermittelt. Der Simulator basiert auf den Gleichungen aus Abschnitt 3. Die zu Grunde liegenden Parameter für die Abbildung der Ereignisse innerhalb des Emulators sind aus den Synthesedaten und aus Ergebnissen von Systemsimulationen des *Frameworks* abgeleitet.

6.3.1 RS-Decoder

Zunächst wurde der in Abbildung8.4 dargestellte *Reed-Solomon-Decoder* mit relevanten Parametern zur Bestimmung der Effizienz des neuen Verfahrens emuliert. Im folgenden Diagramm (vgl. Abb. 6.14) sind die Laufzeitparameter der Module des *RS-Decoders* dargestellt.

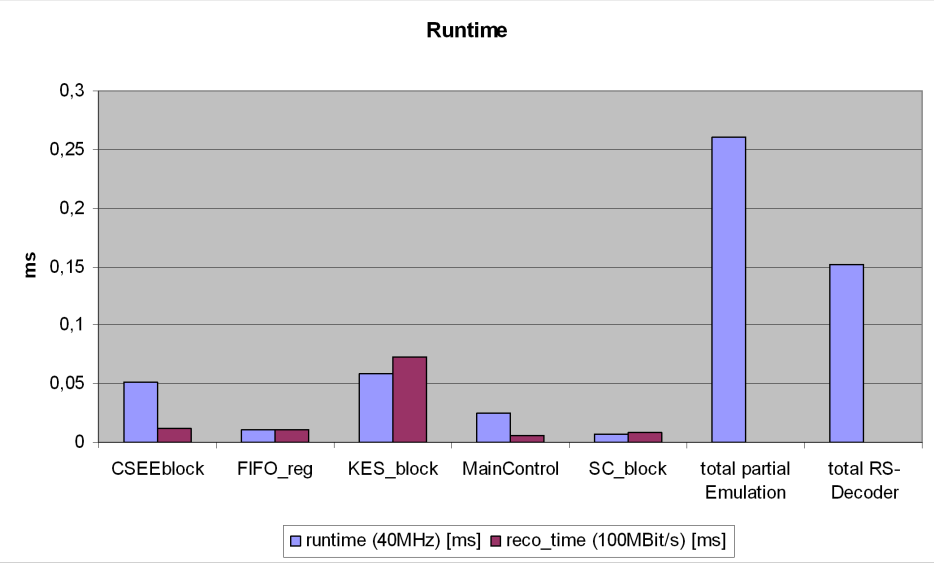


Abbildung 6.14: Laufzeit der Module des *Reed-Solomon-Decoder*

Die Konfigurationszeit ist auf Grund der festgelegten Modulgröße statisch und vor Beginn der Emulation bekannt. Für die Bewertung des Verfahrens wurde eine Konfigurationsbandbreite von 380MByte/s zu Grunde gelegt. Das entspricht der vollen Auslastung einer ICAP-Schnittstelle für die Konfiguration. Die Laufzeiten und damit die Dauer der Modulemulationen sind abhängig vom Datenfluss des jeweiligen Moduls. Für die Simulation des Emulators wurden die Laufzeiten der Module aus einer Gesamtsimulation des *RS-Decoders* abgeleitet. Das Zeitverhalten der partiellen Emulationen ist unter Anwendung des neuen Ansatzes in Diagramm 6.15dargestellt.

Unter Beachtung der Gleichung 3.11 lässt sich für die folgenden Emulationsläufe eine wirksame Effizienz der Module ableiten. Dadurch ist ein gezieltes *Prefetching* von Modulaktivitäten innerhalb bestimmter Grenzen möglich. Die Module *CSEE-block* und *Main Control* weisen eine hohe Effizienz für die zu Grunde gelegten Stimulationsdaten auf, während die Module *FIFO_reg*, *KES-block* und *SC-Block* eine sehr kurze Laufzeit im Verhältnis zu den Rekonfigurationszeiten haben.

Die Werte in Diagramm 6.15 sind für den einmaligen Durchlauf der Module ermittelt worden.

Eine Ausnahme bildet das Modul *FIFO_reg,* dessen Zugriff sowohl während der Syndromberechnung (*SC-Block*) als auch während der Fehlerkorrektur (*CSEEblock*) erfolgt. Das Modul *MainControl*, welches aktiv die Steuerung kontrolliert, interagiert zu Beginn und am Ende der Berechnungen eines jeden Moduls im Ablaufplan des Fehlerkorrekturverfahrens. Wird die Anzahl der rekonfigurierbaren Flächen auf vier festgelegt, ist es möglich, die Module *FIFO_reg* und *MainControl* statisch zu platzieren. Für die gesamte Emulation des *RS-Decoders* wurde ein Zeitbedarf von 3,7 Sekunden ermittelt.

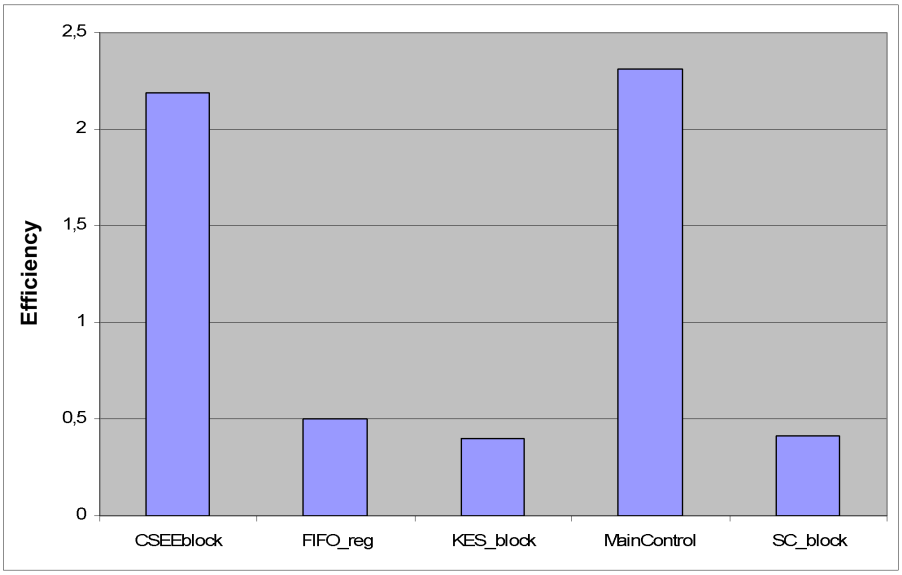


Abbildung 6.15: Module Efficienz *RS-Decoder*

6.3.2 Signal-Prozessor-SoC

Die Bewertungen der Laufzeitergebnisse des Signal-Prozessor-SoC unter Verwendung des neuen Ansatzes beziehen sich auf eine äquivalente Implementierung des SoCs auf einem *StratixII-180* FPGA von Altera. In dieser Implementierung ist der gesamte SoC-Entwurf auf einem einzelnen FPGA abgebildet worden.

Nach der Partitionierung des SoC wurden mit einer Synthese die Kosten der HW-Module bestimmt. Aus diesen Kosten sind die Konfigurationszeiten der HW-Module abgeleitet, welche in 6.16 dargestellt sind.

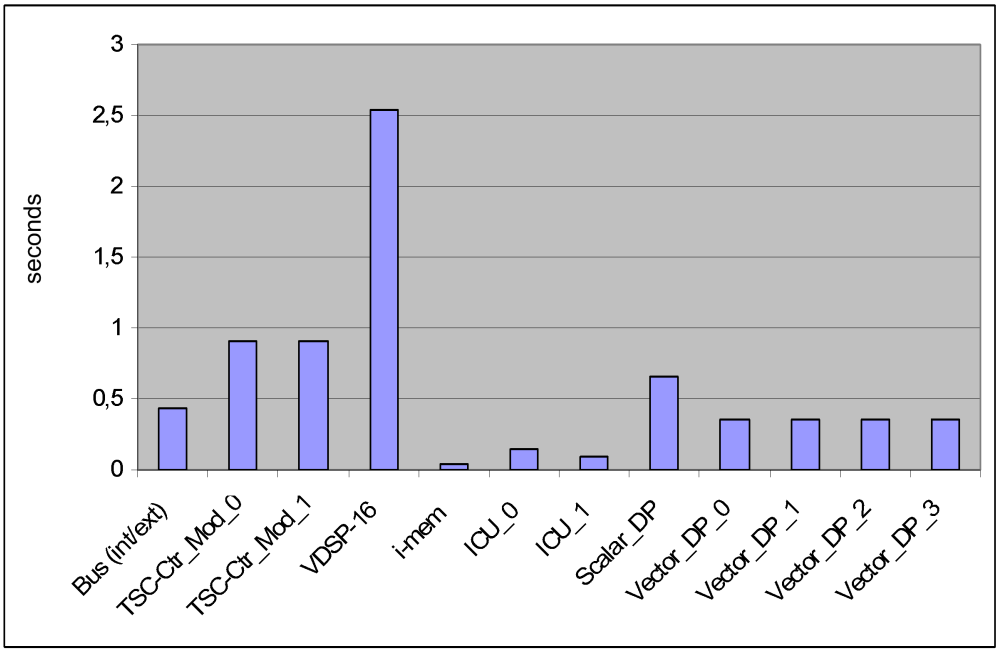


Abbildung 6.16: Rekonfigurationszeiten HW-Module SoC

Aus den Rekonfigurations- und partiellen Laufzeiten der HW-Module wird die effektive Frequenz des Emulators abgeleitet. In Abbildung 6.17 sind unterschiedliche Ansätze für die Emulation des SoCs hinsichtlich der erreichbaren Frequenz bei einer unterschiedlichen Anzahl an Emulatortakten dargestellt. Für die Rekonfiguration nach jedem Takt des HW-Moduls, was der Emulation eines struktural partitionierten Entwurfs entspricht, sinkt die effektive Frequenz des Emulators in den Hertzbereich. Mit einer zunehmenden Anzahl an emulierten Takten der HW-Module im Verhältnis zu den Konfigurationszeiten nähert sich die Frequenz der 40MHz-Grenze. Diese Grenze ist sowohl für die Vergleichsimplementierung auf dem *StratixII-180* FPGA als auch für das Emulator-*Framework* maßgebend. Für die partielle Emulation des SoCs mit jeweils nur einem zeitgleich platzierten HW-Modul reduziert sich die effektiv erreichbare Frequenz der Emulation auf 3.3Mhz. Des Weiteren sind die Ergebnisse für 3 und 12 rekonfigurierbare Flächen berücksichtigt. Mit einer steigenden Anzahl an partiell platzierten HW-Modulen während der Emulation nähert sich die Funktion schneller der Obergrenze von 40MHz. Ein weiterer Vergleichswert ist die erreichbare Frequenz eines Multi-FPGA-Systems, welches aus *Virtex4*-FPGAs besteht. Dieses erreicht bei einer mittleren Auslastung der FPGAs eine effektive Frequenz für die Emulation von 8MHz [96].

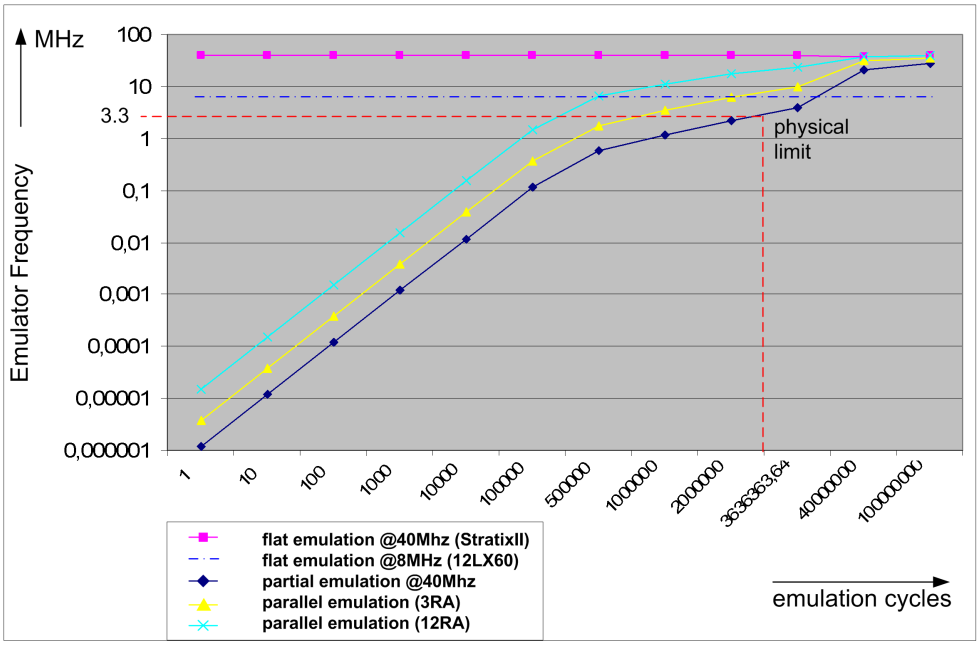


Abbildung 6.17: Emulatorperformance im Vergleich zu einem flachen Emulator

In Abbildung 6.18 sind die erreichten Emulationszeiten für die HW-Module *TSC-Controller* und *Vector-DP* dargestellt. Der in drei HW-Modulen struktural partitionierte *TSC-Controller* veranschaulicht den großen Zeitbedarf für die Emulation eines strukturalen HW-Moduls. Die gesamte Emulationszeit übersteigt schon für sehr kurze Laufzeiten des HW-Moduls eine Zeitspanne, die für den Test eines Entwurfs wesentlich zu lang ist.

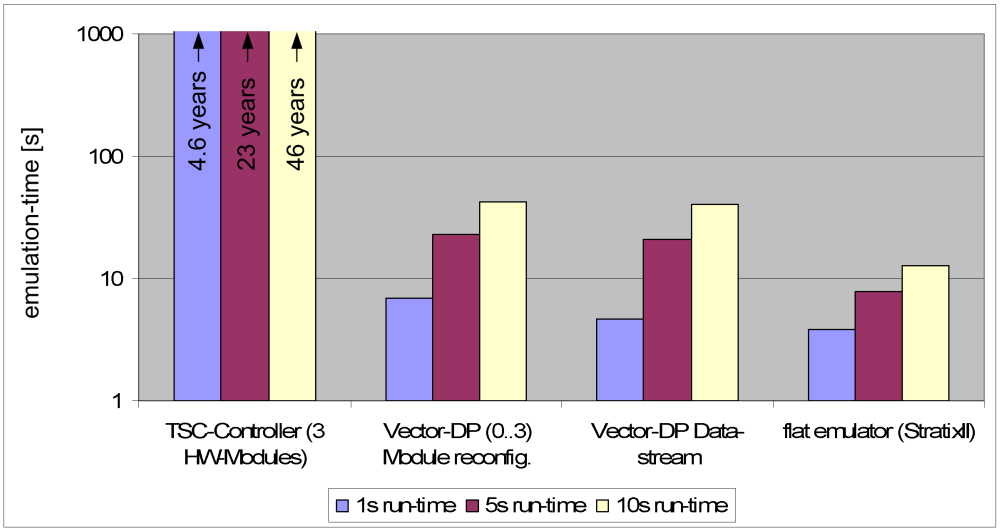


Abbildung 6.18: Emulationszeiten für verschiedene HW-Module

Das HW-Modul *VDSP-16* enthält des Weiteren 4 äquivalente *Vector-DPs*. Diese regulären Architekturen sind in zwei unterschiedlichen Variationen emuliert worden. In der ersten Variante wurden alle vier Datenpfade sequentiell konfiguriert und emuliert. In der zweiten Variante erfolgte ein zyklischer Datenaustausch nach der Konfiguration eines einzelnen Datenpfades. In Abbildung 6.18 ist zu erkennen, dass die unterschiedlichen Varianten mit einer längeren Modullaufzeit eine vergleichbare Gesamtemulationszeit erreichen. Durch die wachsende Effizienz der HW-Module mit einer zunehmenden Modullaufzeit sinkt der Einfluss der Rekonfigurationszeiten. Die Emulation von regulären Architekturen nähert sich dann der Effizienz eines konventionellen Emulators, welcher die gesamte Architektur zeitgleich abbildet. Die gesamte Emulationszeit steigt linear mit der Anzahl an regulären HW-Modulen die sequentiell emuliert werden.

6.4 Bewertung des Emulationsverfahrens

Das beschriebene Emulationsverfahren basiert auf der Partitionierung eines zu verifizierenden Hardware Entwurfs und der anschließenden partiellen Emulation der separaten HW-Module. Das vorgestellte Partitionierungsverfahren, welches einen Schwerpunkt der vorgelegten Arbeit darstellt, setzt sich aus zwei unabhängigen Verfahrensteilen zusammen. Der erste Schritt unterteilt den Entwurf auf funktionaler Ebene. Zunächst ist dafür eine Analyse der Entwurfsbeschreibung notwendig. Hierfür wurde auf ein Standardwerkzeug (*HDL-Designer ©MentorGraphics*) zurückgegriffen. Dies hat den Vorteil, dass umfangreiche und verifizierte Funktionen zur Analyse der Entwürfe zur Verfügung stehen. Die Synthese wurde mit dem Werkzeug *Precision* durchgeführt. Im Anschluss an die Synthese ist eine Bewertung der Kosten möglich. Der Ablauf zur funktionalen Partitionierung ist teilweise automatisiert implementiert. Die generische Anbindung der für die partielle Rekonfiguration notwendigen und plattformabhängigen Makros (z.B. *slice macro* für *Virtex4*) ist im derzeitigen Stand nicht automatisch realisiert.

Werden während der Kostenbewertung, welche unter Einbeziehung des Emulator-*Frameworks* durchgeführt wird, Module selektiert, welche extreme Abweichungen aufweisen oder generell nicht die Vorgaben der Plattform erfüllen, wird eine weitere Partitionierung auf strukturaler Ebene durchgeführt. Dafür wurde ein Verfahren implementiert, das auf dem *Cone*-Algorithmus basiert. Der Ablauf für die strukturelle Partitionierung ist vollständig automatisiert. Unter Festlegung systemrelevanter Parameter, welche vom verwendeten FPGA abhängen, analysiert und partitioniert der Algorithmus einen vollständigen Entwurf oder ein einzelnes HW-Modul aus einem Entwurf. Ein signifikanter Nachteil des umgesetzten Verfahrens ist der erhebliche Zeitbedarf für die strukturale Partitionierung. Dieser ist abhängig von der Struktur und der Größe des zu partitionierenden Entwurfsteils. Durch die einmalige Aufteilung des gesamten Entwurfs zu Beginn der Emulation fallen die zeitlichen Kosten der Partitionierung jedoch im gesamten Ablauf des Verfahrens nur geringfügig ins Gewicht.

Für die Ablaufsteuerung der Emulation wurde eine Laufzeitumgebung implementiert, welche zum einen die direkte Kontrolle über den Ablauf regelt und zum anderen den Ressourcenbedarf der Laufzeitsteuerung reduziert. Das Emulator-*Framework* wird abhängig von den partitionierten Modulen generiert. Dafür ist eine Parametrisierung berücksichtigt worden, welche eine Einstellung aller relevanten Größen des *Frameworks* zulässt. Die anschließende Synthese wird in Zusammenwirkung mit den partitionierten HW-Modulen durchgeführt. Abhängig von der Zunahme an rekonfigurierbaren Flächen im *Framework*, ist eine exponentielle Zunahme des Ressourcenbedarfs für das gesamte *Framework* messbar. Eine Vergrößerung der Signalbreite führt demgegenüber zu einer linearen Ressourcenzunahme. Der Zeitbedarf für die Synthese der kompletten Laufzeitumgebung ist ein signifikanter Faktor für die Zeitdauer der gesamten Emulation.

Dieser Kostenfaktor kann durch eine Bibliothek, welche eine Auswahl aus vorsynthetisierten Umgebungen enthält, reduziert werden.

Die erreichbaren Ergebnisse bzgl. der Laufzeit basieren auf einer prototypischen Implementierung der wesentlichen Teile des *Frameworks* auf einem *Virtex4-LX60*-FPGA. Im direkten Vergleich mit einem *StratixII*-FPGA, auf welchem die vollständige Implementierung eines heterogenen *SoC* realisiert wurde, ist bzgl. der erreichbaren Taktfrequenz eine Äquivalenz bei einer Signalbreite von 64 Leitungen festzustellen. Durch den hohen Grad der Auslastung des *StratixII*-FPGA wird dessen Gesamtperformanz gegenüber dem geringer ausgelasteten *Virtex4*-FPGA deutlich reduziert. Für den neuen Ansatz ist gegenüber der Auslastung die erreichbare Taktfrequenz des Emulator-*Frameworks* entscheidend für die maximal erreichbare Taktfrequenz.

Eine Abbildung und Emulation unter Anwendung des neuen Ansatzes vergrößert zunächst den Zeitbedarf für das Gesamtverfahren, reduziert aber wesentlich den Hardwarebedarf des Emulators. Im direkten Vergleich hinsichtlich der verfügbaren Ressourcen eines *StratixII*- und eines *Virtex4*-FPGAs ohne der Berücksichtigung von Optimierungen innerhalb der Technologie (z.B. *DSP blockset* eines *Virtex4*) kann auf einem *StratixII-FPGA* die doppelte Anzahl an äquivalenten Gattern abgebildet werden.

Ein weiterer Vorteil ist die optimierte Auslastung des einzelnen FPGA. Da die Effizienz von Multi-FPGA-Emulatoren wesentlich von der Bandbreite der Verbindungen abhängt, sinkt für zu testende Entwürfe, welche sehr breite Kommunikationsstrukturen beinhalten, die Auslastung nach Erfahrungen relevanter EDA-Hersteller auf etwa 20%. Zusätzlich beeinflussen die aus dem FPGA herausgeführten Leitungen durch eine deutlich geringere Performanz als im FPGA verdrahtete Signalleitungen negativ die Länge des kritischen Pfads des gesamten Entwurfs. Eine Folge dieser Einschränkung ist eine notwendige Reduzierung des gesamten Taktes der Emulation. Aus Erhebungen der Hersteller von Multi-FPGA-Systemen geht hervor, dass die effektive Frequenz einer Emulation realistisch im Kilo- oder unteren Megahertzbereich durchgeführt werden muss, d.h., die erreichbare Maximalfrequenz der FPGA wird nicht ausgenutzt.

Der große Zeitaufwand für die Rekonfigurationen zwischen dem Austausch der Module stellt für den neuen Ansatz die z die bedeutendste Einflussgröße hinsichtlich der Effizienz dar. Für Entwürfe, welche aus HW-Modulen aufgebaut sind die über einen sehr langen Zeitraum unabhängig von weiteren HW-Modulen des Gesamtentwurfs Berechnungen durchführen, steigt die Effizienz signifikant an. Ist ein Modul auf struktureller Ebene partitioniert worden, ist es notwendig, die internen Zustände der einzelnen Modulteile in jedem Taktzyklus abzugleichen. Dadurch sinkt die Effizienz direkt proportional mit der Größe und der Anzahl der enthaltenen HW-Module.

7 Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde ein neuartiges Verfahren für die Emulation von Hardwareentwürfen vorgestellt. Der Ansatz ordnet sich in die funktionalen Verifikationsverfahren ein und basiert auf der Abbildung des zu testenden Entwurfs auf einem FPGA. Konträr zu bekannten Emulationsverfahren wird der Entwurf nicht vollständig sondern teilweise emuliert. Zwischen den Teilemulationen erfolgt ein Datenaustausch, was in der Gesamtheit aller partiellen Emulationen zu einer umfassenden Entwurfsemulation führt.

Die Partitionierung des Entwurfs erfolgt im Fokus unter funktionalen Gesichtspunkten. Die der Partitionierung zu Grunde liegende Kostenfunktion orientiert sich an der Emulatorplattform und hat wesentlichen Einfluss auf die Effizienz des Ansatzes. Im Anschluss an die Partitionierung werden die Entwurfsteile zu funktionsfähigen HW-Modulen synthetisiert. Während der Untersuchungen zur Steuerung der einzelnen Emulationsläufe wurde eine Laufzeitumgebung abgeleitet, welche die Kosten in zeitlicher- als auch hinsichtlich des Ressourcenbedarfs optimiert. Dieses Emulator-*Framework* wird parametrisierbar an die Kosten der zu emulierenden HW-Module angepasst. Zur Reduzierung der zeitintensiven Rekonfigurationen verbindet ein Netzwerk, welches Teil des *Frameworks* ist, die für die Rekonfiguration vorgesehenen Flächen miteinander. Die entstehenden Kommunikationswege berücksichtigen sowohl lokale als auch globale Abhängigkeiten der HW-Module. Dadurch verbleiben direkt abhängige HW-Module über einen längeren Emulationszeitraum auf der Plattform, was die Anzahl der notwendigen Rekonfigurationen signifikant reduziert. Die Daten der partiellen Emulationen werden datenflussabhängig für die Stimulierung einer der folgenden Modulemulationen herangezogen oder an ein *Host*-System zurück geschrieben. Unter Verwendung eines komplementären Werkzeugs (*VTrac+*) wird die Abbildung des gesamten Emulationslaufes in einem standardisierten HW-Simulator erreicht.

Einer der signifikantesten Vorteile des neuen Ansatzes ist die effektive Auslastung der einzelnen FPGAs. Dadurch werden die Nachteile der beschränkten Bandbreite zwischen den FPGAs in konventioneller Multi-FPGA-Systemen weitgehend reduziert. Die Optimierung der Synthese wird durch den neuartigen Ansatz für nur einen einzelnen FPGA und nicht ein Feld aus FPGAs mit dazwischen liegenden Verbindungsleitungen durchgeführt.

Das implementierte Emulator-*Framework* erreicht abhängig von der zu Grunde gelegten Signalbreite und der Anzahl der rekonfigurierbaren Flächen eine Taktfrequenz von ca. 20-40 Mhz. Die effektive Frequenz der Emulation sinkt auf Grund der notwendigen Rekonfigurationen entwurfs- und datenflussabhängig auf etwa 1-5 Mhz. Die Auslastung der FPGA nimmt aber gegenüber Multi-FPGA-Systemen deutlich zu.

Zusammenfassend stellt der neue Ansatz eine Alternative zu konventionellen Ansätzen dar, in welchen die Anzahl der FPGA erhöht wird, um dem wachsenden Bedarf an Ressourcen gerecht werden zu können.

Unter Ausnutzung des Optimierungspotentials, welches während der Arbeiten eingegrenzt werden konnte, lässt sich nach einer Abschätzung die Performance des Emulator-*Frameworks* auf ca. 50-100 Mhz erhöhen. Damit steigt die effektive Frequenz der Emulation auf bis zu 50 Mhz. Die Skalierbarkeit und die Möglichkeit, das neue Verfahren auch für bestehende Multi-FPGA-Systeme einzusetzen, lässt eine konti-

nuierliche Weiterentwicklung bestehender Systems zu und ist damit für kleine und mittelständische Unternehmen eine kostengünstige Alternative zu wenig skalierbaren konventionellen Multi-FPGA-Systemen.

7.1 Ausblick

Bezug nehmend auf die Implementierungen für den neuen Ansatz unter Einbeziehung eines einzelnen FPGAs, ist das Verfahren auch auf mehrere FPGAs anwendbar. Diese werden dann als einzelne Emulatoren betrachtet und zentral gesteuert. Die Synthese und Optimierung erfolgt gemäß des neuen Ansatzes unter Berücksichtigung der Umgebung eines einzelnen FPGAs. Dadurch ist eine flexible Möglichkeit gegeben, ein komplexes und parametrisierbares Emulatorsystem aufzubauen, welches die Bedürfnisse und Anforderungen zukünftiger heterogener *SoC*-Entwürfe berücksichtigt. Durch die deutlich gesteigerte Performanz können dabei insbesondere die für diese Systeme typischen Schnittstellen und Protokolle zur Umgebung einbezogen werden. Des Weiteren reduziert sich der in den Kosten und weitgehend geringer Flexibilität begründete Nachteile FPGA-basierter HW-Emulatoren.

Neben dem Anwendungsgebiet der funktionalen Verifikation ist eine Anwendung des neuen Ansatzes in Prototypensystemen bis hin zu Produkten möglich. Unter Beachtung der Grenzen und Einschränkungen des Verfahrens ist eine signifikante Reduzierung der HW-Ressourcen und damit den entstehenden Kosten zu erwarten. Da in diesen Gebieten die genauen Anwendungsparameter durch eine Spezifikation festgelegt sind, ist eine weitere Optimierung der Partitionierung und des *Frameworks* für das spezifizierte System möglich.

Durch die Anwendung eines *Prefetching*-Verfahrens ist eine weitere Optimierung des Steueralgorithmus möglich. Dadurch sinkt der Einfluss der Rekonfigurationszeiten der HW-Module auf die Gesamtzeit einer Emulation.

Die Inhalte der Arbeit beziehen sich auf die Optimierung eines FPGA-basierten Emulationsverfahrens zum Test integrierter Schaltkreise. Die Effizienz bestehender Verfahren wird sich mit der weiter zunehmenden Komplexität zukünftiger Entwürfe deutlich reduzieren. Das vorgestellte Verfahren wurde unter dem Gesichtspunkt der industriellen Anwendung betrachtet und stellt eine Möglichkeit dar, die Nachteile konventioneller FPGA-Plattformen zu vermeiden. Gerade unter Einbeziehung bestehender Multi-FPGA-Systeme ist eine signifikante Steigerung der Performance für den funktionalen Test von integrierten Schaltkreisen und Systemen zu erwarten und sollte weiter untersucht werden.

Literaturverzeichnis

[1] 4DSP, “FM480 User Manual v1.1”, 4DSP Inc. 1757 Fox Run road, Reno, www.4dsp.com, 4DSP Inc. 2006.

[2] A. Ali,C. Bobda, S. Fekete, T. Haller, A. Linarth, M. Majer, J. Teich, J. van der Veen, “A Highly Flexible FPGA-Based Reconfigurable Platform”, Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Marriott at Napa Valley, California, USA, 319-320, April 17-20, 2005.

[3] Altera, “Dynamic Reconfiguration of Excalibur Devices with Linux”, Altera Application Note 278, 2003.

[4] Altera, “Nios II Processor”, Altera Product Brief, www.altera.com, 2007.

[5] Altera, “StratixII vs. Virtex-4 Density Comparison”, White Paper Altera, August 2005.

[6] G. Archer, “JIFFY - Ein FPGA-basierter Java Just-in-Time Compiler für eingebettete Anwendungen”, Dissertation, Lehrstuhl für Rechnertechnik und Rechnerorganisation, Technischen Universität München, 2003.

[7] ATMEL Corporation, “Atmel’s FPSLIC II Dynamically Reconfigurable SoC Supports Silicon-Sharing For Peripherals & Interfaces”, Data Sheet, Atmel 2005.

[8] ATMEL, “FPSLIC on-chip Partial Reconfiguration of the Embedded AT40K FPGA”, Application Note, Atmel, 2002.

[9] J. Babb, T. Russel, M. Dahl, S. Hanono, D. Hoki, A. Agarwal, “Logic Emulation with Virtual Wires”, IEEE Transactions on Computer Aided Design, 609 - 626, June 1997.

[10] L. G. Barbero, J. S. Thompson, “RAPID PROTOTYPING OF THE SPHERE DECODER FOR MIMO SYSTEMS”, Institute for Digital Communications University of Edinburgh, IEEE/EURASIP Conference on DSP Enabled Radio, London, 19-20th September, 2005.

[11] E. Barke,et. al, “Electronic Design Automation Entwurfsautomatisierung in der Mikroelektronik”, IMS, Institute of Microelectronic Systems. Leibnitz University of Hannover, Design Automation (http://edascript.ims.uni-hannover.de) Emulation, 2007.

[12] E. G. Barrantes, D. H. Ackley, S. Forrest, D. Stefanovic, “Randomized Instruction Set Emulation”, ACM Transactions on Information and System Security, Vol. 8, No. 1, February 2005.

[13] R. Beckert, M Boden, “An approach for design automation based upon the Xilinx-Modular-Design-Flow”, Workshop for Chip and System-design, DASS, Dresden 2005.

[14] R. Beckert, S. Altmann, “A verification methodology for programmable and reconfigurable processors”, Proceedings of Mentor Graphics User2User Conference, San José, 2007.

[15] R. Beckert, St. Rülke, W. Hardt, “Optimized FPGA-Based Hardware Emulation by Using Partial Runtime Reconfiguration (pRTR)”, Proceedings of the Work in Progress Session held in connection with SEAA 2006, the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications and DSD 2006, the 9th EUROMICRO Conference on Digital System Design, Cavtat, Dubrovnik, Croatia, 2006.

[16] R. Beckert, “A Novel Simulation Approach for Partial Run-Time Reconfigurable Hardware”, Poster in Workshop for Chip and Systemdesign, DASS, Dresden 2006.

[17] R. Beckert, “Prefetching Algorithmen für intelligente Vektor-DSP Caches”, Diplomarbeit am Vodafone-Stiftungslehrstuhl der Fakultät Elektrotechnik an der Technischen Universität Dresden, 2004.

[18] I. Benkermi, A. Benkhelifa, D. Chillet, S. Pillement, J.C. Prévotet, F. Verdier, “System-Level Modelling for Reconfigurable SoCs", 20th Conference on Design of Circuits and Integrated Systems (DCIS), Lisboa, Portugal, November, 2005.

[19] T. Blochmann, ”Simulation partiell rekonfigurierbarer Hardwarearchitekturen”, Diplomarbeit am Fraunhofer Institut für Integrierte Schaltungen - Institutsteil Entwurfsautomatisierung Dresden, in Zusammenarbeit und eingereicht im Fachbereich Elektrotechnik der Hochschule für Technik und Wirtschaft Dresden (FH), 2005.

[20] T. Blochmann, ”Design-Partitionierung für eine verteilte Hardware-Emulation auf einer dynamisch rekonfigurierbaren FPGA-Plattform”, Masterabreit am Fraunhofer Institut für Integrierte Schaltungen - Institutsteil Entwurfsautomatisierung Dresden, in Zusammenarbeit und eingereicht im Fachbereich Elektrotechnik der Hochschule für Technik und Wirtschaft Dresden (FH), 2006.

[21] C. Bobda, “A. Ahmadinia, “Dynamic Interconnection of Reconfigurable Modules on Reconfigurable Device”, in IEEE Design & Test of Computers, Special Issue, 2005.

[22] C. Bodba, “Introduction to Reconfigurable Computing: Architectures, Algorithms, and Applications”, ISBN: 1402060882, Springer Verlag 2007.

[23] C. Bobda, “Synthesis of dataflow graphs for reconfigurable systems using temporal partitioning and temporal placement”, Universität Paderborn, Dissertation, 2003.

[24] C. Bobda, N. Steenbock, ”Singular Decomposition on Distributed Reconfigurable Systems”, in 12th IEEE International Workshop on Rapid System Prototyping Monterey, California, USA, 2001.

[25] C. Bobda, N. Steenbock, “A Rapid Prototyping Environment for Distributed Reconfigurable Systems“, Proceedings 13th IEEE International Workshop on Rapid System Prototyping, Darmstadt, Germany, 153 - 158, July 1-3, 2002.

[26] M. Bocchi, C. De Bartolomeis, C. Mucci, F. Campi, A. Lodi, M. Toma, R. Canegallo, R. Guerrieri, “A XiRisc-based SoC for embedded DSP applications”, Proceedings of the IEEE 2004 Volume , Issue , 3-6 Oct. 2004 Page(s): 595 - 598, Custom Integrated Circuits Conference, 2004.

[27] K. K. Bondalapati, “Modeling and Mapping for Dynamically Reconfigurable Hybrid Architectures”, Dissertation thesis, Faculty of the Graduate School University of Southern California, 2001.

[28] M. Boule, J.S. Chenard, Z. Tilic, “Adding Debug Enhancements to Assertion Checkers for Hardware Emulation and Silicon Debug and In-Field Diagnostics”, Proceedings of the 8th International Symposium on Quality Electronic Design, ISQED’07, 2007.

[29] P. Christie and D. Stroobandt, “The interpretation and application of Rent’s rule”, IEEE Trans. on VLSI Systems, Special Issue on System Level Interconnect Prediction, 8(6):639-648, December 2000.

[30] M. Cochand, G.abrio Caimi, “Diskrete Optimierung Teil 2: Matching maximaler Kardinalität”, Eidgenössische Technische Hochschule Zürich, ETH Zentrum, CLV B2 Clausiusstrasse 47 CH8092 Zürich, 2005.

[31] M. Danek, P. Honzik, J. Kadlec, R. Matousek, Z. Pohl “Reconfigurable System on a Programmable Chip Platform”, Institute of Information Theory and Automation, Dept. of Computer Science & Eng. and Centre of Applied Cybernetics, 2004.

[32] D. Dempster, M. Stuart, “Verification Methodology Manual - Techniques for Verifying HDL Designs, Second Edition”, ISBN 0-9538-4821-3, ©Teamwork International UK, 2001.

[33] T. Dewey, “IP Reuse for FPGA Design Rapidly Unravel Internal and Third-Party IP”, White Paper by Tom Dewey, Technical Marketing Engineer Mentor Graphics, October 2002.

[34] M. Dyer, C. Plessl, M. Platzner, “Partially Reconfigurable Cores for Xilinx Virtex”, Lecture Notes In Computer Science; Vol. 2438, Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications 292 - 301, ISBN: 3-540-44108-5, 2002.

[35] A. Ejnioui, N. Ranganathan, “Design Partitioning on Single-Chip Emulation Systems”, vlsid, p. 234, 13th International Conference on VLSI Design, 2000.

[36] C. S. Ferner, R. G. Babb, “Automatic choice of scheduling heuristics for parallel/distributed computing”, Scientific Programming archive, Volume 7 , Issue 1 (January 1999) pp. 47 - 65, 1999.

[37] C.M. Fiduccia, R.M. Mattheyses, A linear-time heuristic for improving network partitioins, on Proc. Design Automation Conf., 1982.

[38] T. Fuchs, ”Entwurf und Programmierung einer generischen Monitor- und Konfigurationsumgebung für einen Hardware-Emulator”, Diplomarbeit am Fraunhofer Institut für Integrierte Schaltungen - Institutsteil Entwurfsautomatisierung Dresden, in Zusammenarbeit und eingereicht am Institut für Informatik der Humbold-Universität zu Berlin, Juni 2007.

[39] D. Gajski, N. Dutt, A, Wu, S, Lin, “High- Level Synthesis, Introduction to Chip and System Design”, ISBN 0-7923-9194-2, ©Kluwer Academic Publisher, 1992.

[40] M. Garey, D. Johnson, “Computers and Intractibility - A Guide to the Theory of NP-Completeness”, W.H. Freeman & Company, 1979.

[41] C. Gremzow, “High-Level-Synthese aus flachen Kontroll-/Datenflussgraphen”, Fakultät IV - Elektrotechnik und Informatik der Technischen Universität Berlin, 2004.

[42] W. Hardt, “Integration von Verzögerungszeit-Invarianz in den Entwurf eingebetteter Systeme”, Habilitationsschrift Fachbereich Mathematik/Informatik Universität Paderborn, 2002.

[43] W. Hardt, A. Meisel, M. Visarius, “Automatische Rekonfiguration von Hardware-Schnittstellen”, in Proceedings of the German Workshop Dresdner Arbeitstagung Schaltungs- und Systementwurf, Dresden, Germany, April 2004.

[44] W. Hardt, “Hardware/Software-CoDesign I,II”, Vorlesungsveranstaltung Professur Technische Informatik an der Technischen Universität Chemnitz, 2008.

[45] W. Hardt, “HW/SW-CoDesign auf Basis von C-Programmen unter Performanz-Gesichtspunkten”, Dissertation Universität Paderborn, ISBN: 3-8265-2051-3, 1996.

[46] W. Hardt, B. Kleinjohann, “Flysig: Dataflow Oriented Delay Insensitive Processor for Rapid Prototyping of Signal Processing”, in 9th IEEE International Workshop on Rapid System Prototyping, IEEE Computer Society Press, 1999.

[47] W. Hardt, F. Rammig, C. Böke, J. Stroop, A. Rettberg, G. Del Castillo, B. Kleinjohann, J. Teich, “IP-based System Design within the *Paradise* Design Environment”, in Journal of Systems Architecture - the Euromicro Jounal 2001.

[48] W. Hardt, W. Rosenstiel, “Prototyping of Tightly Coupled Hardware/Software-Systems”, in Kluwer Journal, Design Automation for Embedded Systems 2, Nr.1, 1997.

[49] R. W. Hartenstein, J. Becker, “A Two-level Co-Design Framework for Xputer-based data-driven reconfigurable Accelerators”, Proceedings of the 30th Hawaii International Conference on System Sciences, Advanced Technology Track, Volume 5, pp. 125, 1997.

[50] R. W. Hartenstein, J. Becker, M. Herz, U. Nageldinger, “Data Scheduling in Hardware/Software Co-Design for Field-programmable Accelerators, Proc. of 7th International Workshop on Field Programmable Logic, FPL’97, London, UK, September 1-3, 1997.

[51] R. Hartenstein, “Standort Deutschland: Wozu noch Mikrochips? Einführung in Methoden der Technischen Informatik”, Kapitel 20 Strukturierte Entwürfe, ITpress, 1964.

[52] S. Hauck, G. Borriello, C. Ebeling, ”Mesh Routing Topologies For Multi-FPGA Systems”, International Conference on Computer Design, 170-177, 1994.

[53] J. Haufe, C. Fritsch, M. Gulbins, V. Lück, P. Schwarz, “Real-Time Debugging of Digital Integrated Circuits”, Proc. Design, Automation and Test in Europe Conference, User Forum, Paris, 235-241, March 27-30, 2000.

[54] J. R. Hauser, J. Wawrzynek, “Garp: A MIPS Processor with a Reconfigurable Co-Processor” published in Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM ’97, April 16-18, pp. 24-33, 1997.

[55] U. Heinkel, “Interface Specification and Validation using Extended Timing Diagrams”, PhD Forum DAC, San Francisco, 1998.

[56] U. Heinkel, “Formale Spezifikation und Validierung digitaler Schaltungsbeschreibungen mit Zeitdiagrammen”, Dissertation, Friedrich-Alexander-Universität Erlangen-Nürnberg, 1999.

[57] J. Hennessy, D. Patterson, Computer Architecture A Quantitative Approach, Second Edition Page 296, Morgan Kaufmann Publishers, INC, 1996.

[58] D. Herrscher, “Emulation von Rechnernetzen zur Leistungsanalyse von verteilten Anwendungen und Netzprotokollen”, Dissertationsschrift, Fakultät Elektotechnik und Informationstechnik Universität Stuttgart, 2005.

[59] N. Horspool, P. Gorman, “The AISC Handbook”, ISBN 0-13-091558-0, ©Prentice Hall, Inc., 2001.

[60] H. Huber, “SoC for evolving standards: A new challenge for rapid prototyping using FPGAs?”, Mint Technology, Division of LSI Logic, 2006.

[61] J. Hwang, A. El Gamal, “Optimal Replication for Min Cut Partitioning”, Proc. IEEE International Conference on Computer-Aided Design, Santa Clara, California, November 1992.

[62] IBM, “Product Overview PowerPC 405 CPU Core”, IBM Product Brief, http://www-01.ibm.com, 2007.

[63] B.W. Kernighan, S.Lin, An efficient heuristic procedure partitioning graphs, Bell Syst. Tech. Journal. vol 49, 1970.

[64] S. Koh, O. Diessel, “COMMA: A Communications Methodology for Dynamic Module-based Reconfiguration of FPGAs”, School of Computer Science & Engineering The University of New South Wales & Embedded Real-Time, and Operating Systems (ERTOS) Program National ICT Australia Kensington Laboratory Sydney, Australia UNSW-CSE-TR-0603 February 2006, Dagstuhl Seminar Proceedings 06141 “Dynamically Reconfigurable Architectures”, 2006.

[65] M. Köster, “Analyse und Entwurf von Methoden zur Ressourcenverwaltung partiell rekonfigurierbarer Architekturen”, Dissertation Universität Paderborn, 2007.

[66] R. Kuznar, F. Brglez, B. Zajc, “Multi-way Netlist Partitioning into Heterogeneous FPGAs and Minimization of Total Device Costs and Interconnect”, 31th Design Automation Conference ACM/IEEE, pp 238-243, June 1994.

[67] J. Langer, U. Heinkel, V. Jerninic´ D. Müller, “Analyse von Corner Cases und funktionaler Abdeckung auf Basis von Entscheidungsdiagrammen”, Proc. Dresdner Arbeitsstagung Schaltkreis und Systementwurf DASS, 2007.

[68] E. Langnese, D. Thomas, “Architectural Partitioning for System Level Synthesis of Integrated Circuits”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 10, no 7, 1991.

[69] G. Lantzsch, A. Schneider, P. Schwarz, "Verteilte ereignisorientierte Simulation auf der Basis von High Level Architecture (HLA)", 14. ASIM-KI-Workshop "Multiagentensysteme und individuenbasierte Simulation", 121-127, Würzburg, 20.-21. März 2000.

[70] D. I. Lehn, "Framework for a Context-Switching Run-Time Reconfigurable System", Master Thesis, Faculty of the Virginia Polytechnic Institute and State University, Bradley Departement of Electrical and Computer Engineering, Blacksburg, Virginia, 2002.

[71] C. E. Leiserson, J. B. Saxe, "Retiming Synchonous Circuits", digital System Research Center, 130 Lytten Avenue, Porto Alto, 94301 California, August 20th, 1986.

[72] A. Maili, C. Steger, R. Weiss, R. Quigley, D. Dalton, "Reducing the Communication Bottleneck via On-Chip Cosimulation of Gate-Level HDL and C-Models on a Hardware Accelerator", isvlsi, pp. 290-291, IEEE Computer Society Annual Symposium on VLSI, (ISVLSI'05), 2005.

[73] A. Maili, C. Steger, R. Weiss, D. Dalton, "An Architecture to Enable On-Chip-Cosimulation of IP-Models with the APPLES Gate-Level Accelerator", Telecommunications and Mobile Computing Graz, Austria, March 8-9, 2005.

[74] G. Martin, L. Scheffer, L. Lavagno, "Electronic Design Automation for Integrated Circuits Handbook - Two Volume Set", ISBN 9780849330964, ©2005.

[75] A. Meisel, A. Draeger, S. Schneider, W. Hardt, "Design Flow for Reconfiguration based on the Overlaying Concept", Chemnitzer Informatik-Berichte, Fakultät für Informatik, TU-Chemnitz, ISSN 0947-5125, Januar 2008.

[76] A. Meisel, M. Visarius, W. Hardt, S. Ihmor, "Self-Reconfiguration of Communication Interfaces", in Proceedings of the 15th IEEE International Workshop on Rapid System Prototyping, pp. 144-150, Geneva, Switzerland, ISBN: 0-7695-2159-2, June 2004.

[77] M. Ng, M. Paettie, "Using a Microprocessor to Configure Xilinx FPGAs via Slave Serial or SelectMAP Mode", Xilinx Application Note, 2002.

[78] U. Ober, "Partitionierung und Laufzeitoptimierung komplexer Anwendungen für FPGA-basierte Rapid-Prototyping-Board-Architekturen", Dissertation, Fachbereich 18 der Technischen Universität Darmstadt, ISBN 3-925178-23-6, 1998.

[79] OpenCores, "Reed-Solomon-Decoder 31,19,6", www.opencores.com, April 2007.

[80] P. Peichen, L. Chih-Chang, "A new retiming-based technology mapping algorithm for LUT-based FPGAs", International Symposium on Field Programmable Gate Arrays, Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays, Monterey, California, United States Pages: 35 - 42, ISBN:0-89791-978-5, 1998.

[81] Z. Pohl, R. Matousek, J. Kadlec, M. Danek, "AN001 Runtime Partial Reconfiguration for Atmel AT94K", Department of Signal Processing, Institute of Information Theory and Automation, Academy of Sciences of Czech Repulick, Atmel Application Note rev1.0, 2005.

[82] W. Qin, B. Hu Fudan, “A Technique to Exploit Memory Locality for Fast Instruction Set Simulation”, ASIC 2005. ASICON 2005. 6th International Conference On Volume 2, Issue , 24-27 Oct. 2005.

[83] F. Rogin, E. Fehlauer, “FSM-based Rule Specification Aiming at a Generic Code Analysis Library”, In Proceedings of WIP-Session at EUROMICRO’04, Rennes (France), 2004.

[84] T. Romanek, “Test & Diagnose”, PG Auto Lab, Seminarwochende 21.-23. Oktober 2007.

[85] Saros Technology, “HDL Import as Text and Graphics with HDL Designer”, © Saros Technology Ltd., Application Note Saros Support Team, 2002.

[86] G. Saucier, D. Brasen, J.P. Hiol, “Partitioning with Cone Structures”, International Conference on Computer Aided Design, pp 236-239, 1993.

[87] P. Schlegel, “Simulator-Emulator-Interface EMULINK zur Simulationsbeschleunigung”, elektronikindustrie 7-2000, “http://www.elektronik-industrie.de”, 2000.

[88] S. Schmitt, “Integrierte Simulation und Emulation eingebetteter Hardware/Software-Systeme”, Dissertation der Fakultät für Informations- und Kognitionswissenschaften der Ebert-Karls-Universität Tübingen, ISBN 3-86537-511-1, 2005.

[89] J. Schönherr, B. Straube, “Formale Verifikation auf höheren Entwurfsebenen mittels symbolischer Traversierung unendlicher Automaten”, Workshop GI/ITG/GME Methoden des Entwurfs und der Verifikation digitaler Schaltungen und Systeme & Beschreibungssprachen und Modellierung von Schaltungen und Systemen, Frankfurt, 28. Februar - 1. März 2000.

[90] P. Schwarz, C. Claus, K. Einwich, U. Knöchel, K. Matz, “Hybride Simulation nachrichtentechnischer Systeme”, 12. Symposium Simulationstechnik, 67-74, Zürich 15.-18. September 1998.

[91] R. Sedaghat, “Fault Emulation: Reconfigurable Hardware-Based Fault Simulation Using Logic Emulation Systems with Optimized Mapping”, Dissertation, Fachbereich Elektrotechnik und Informationstechnik der Universität Hannover, 1999.

[92] R. Siripokarpirom, F. Mayer-Lindenberg, “Hardware-Assisted Simulation and Evaluation of IP Cores” 16th IEEE/IFIP International Workshop on Rapid System Prototyping, RSP, 2004.

[93] A. A. Khan, C. L. McCreary, and Y. Gong, ”A Numerical Comparative Analysis of Partitioning Heuristics for Scheduling Task Graphs on Multiprocessors”, Technical Report CSE93-12. Dept. of Computer Science and Engineering, Auburn University, 1993.

[94] A. A. Khan, C. L. McCreary, and M. S. Jones, “A comparison of multiprocessor scheduling heuristics” In Proceedings of the 23rd International Conference on Parallel Processing, Aug. 1994.

[95] J. Stohmann, K. Harbich, M. Olbrich, E. Barke “An Optimized Design Flow for Fast FPGA-Based Rapid Prototyping”, Field-Programmable Logic and Applications From FPGAs to Computing Paradigm, ISBN 3-540-64948-4, 79-88, 1998.

[96] Synplicity, “ASIC Prototyping Using Off-the-Shelf FPGA Boards: How to Save Months of Verification Time and Tens of Thousands of Dollars”, White Paper Synplicity Inc. January 2006.

[97] J. Teich, “Emulation und Rapid Prototyping”, Lecture of Hardware Software-Co-Design, Department of Computer Science-12, University of Erlangen-Nuremberg, SS-2006.

[98] Temento, “DiaLite Instrumentation Leading Edge Edition Rev4.5”, Temento Systems S.A. France P/N 4100S030100, ©2005.

[99] R. Tessier, J. Babb, M. Dahl, S. Hanono, A. Agarwal, “The Virtual Wire Emulation System: A Gate-Efficient ASIC Prototyping Environment”, FPGA ‘94, Berkeley, "citeseer.ist.psu.edu/article/tessier94virtual.html", 1994.

[100] Triscend A7S “Triscend A7S Configurable System-on-Chip Platform, Product Description, August 2002.

[101] Triscend, “Triscend DieCast ASIC Programm”, technical report, 2002.

[102] M. Treydte, A. Meisel, M. Visarius, “Bus Macro Generierung für rekonfigurierbare Hardwarekomponenten auf VHDL Ebene”, in Proceedings of the German Workshop Dresdner Arbeitstagung Schaltungs- und Systementwurf, pp. 59-63, Dresden, Germany, April 2005.

[103] TU-Wien, Vorlesungsskript, “Quelle: http://qse.ifs.tuwien.ac.at/courses /skriptum/download/09P_Funkt_wid_20040204.pdf”, Technische Universität Wien Institut für Softwaretechnik und interaktive Systeme Information und Software Engineering Quality Software Engineering, 2004.

[104] Projekt URANOS, “Analysemethoden für den Entwurf anwendungsrobuster nanoelektronischer Systeme - AP 3.1.2 On-Chip Monitoring”, www.uranos-projekt.de, 2005-2008.

[105] F. Vahid, T. Le, Y. Hsu, A Comparison of Functional and Structural Partitioning, Proceedings of the 9th International Symposium on System Synthesis, page 121, 1996.

[106] F. Vahid, T. Dm Le, H. Yu-Chin, “Functional Partitioning Improvements Over Structural Partitioning for Packaging Constraints and Synthesis-Tool Performance”, ACM Transactions on Design Automation of Electronic Systems vol. 3 no. 2 ©1997, 1998 by the Association for Computing Machinery.

[107] J. Varghese, M. Butts, J. Batcheller, “An Efficient Logic Emulation System”, IEEE Transactions on very large scale integration (VLSI) systems Bd. 1, 1993.

[108] M. Visarius, A. Meisel, M. Scheithauer, W. Hardt, “Dynamic Reconfiguration of IP based Systems”, in Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping, pp. 70-76, Montreal, Canada, ISBN: 0-7695-2361-7, June 2006.

[109] H. Walder, M. Platzner, “A Runtime Environment for Reconfigurable Hardware Operating Systems”, Proceedings of the 14th International Conference, FPL 2004, Field-Programmable Logic and Applications, Antwerb, Belgium, August/September, pp. 831-835, 2004.

[110] P. Wältermann, H. Schütte, K. Diekstall, “Hardware-in-the-Loop-Test verteilter Kfz- Elektroniksysteme”, dSPACE GmbH, dSPACE News, 2004.

[111] R. Weber, “Eine Methodik für die formale Anforderungsspezifikation verteilter Systeme”, Dissertationsschrift, Fakultät Mathematik und Informatik Technische Universität München.

[112] Xilinx, “MicroBlaze - The Industry’s Most Flexible Embedded Processing Solution”, Xilinx Product Brief, http://www.xilinx.com, 2007.

[113] Xilinx, “Virtex4 Family Overview DS112”, Xilinx Preliminary Product Specification v2.0, http://www.xilinx.com, 2007.

[114] Xilinx, “Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet DS083”, Xilinx Product Specification v4.6, http://www.xilinx.com, 2007.

[115] Xilinx, “Virtex4 Configuration Guide”, Xilinx Product Specification v1.4, 2006.

[116] Xilinx, “ChipScope Pro 9.1i Capabilities”, Xilinx Product Specification, (http://www.xilinx.com/ise/optional_prod/cspro.htm), 2007.

[117] Xilinx, “RocketIO Transceiver User Guide” Xilinx Application Node UG024 v3.0, 2007.

[118] H. Yang, D. F. Wong, “Efficient Network Flow Based Min-Cut Balanced Partitioning”, IEEE International Conference on Computer Aided Design, pp 50-55, November 1994.

[119] Z. A. Ye, A. Moshovos, S. Hauck and P. Banerjee, “Chimaera: A High-Performance Architecture with a tightlycoupled Reconfigurable Unit”, in Proc. 27th Intl. Symposium on Computer Architecture, Jun. 2000.

[120] Z. A. Ye, A. Moshvos, S. Hauck, N. Shenoy, P. Banerjee, “CHIMAERA: Integrating a Reconfigurable Unit into a High-Performance, Dynamically-Scheduled Superscalar Processor”, Faculty of Electrical and Computer Engineering Northwestern University, Faculty of Electrical Engineering, University of Washington, 2000.

8 Anhang

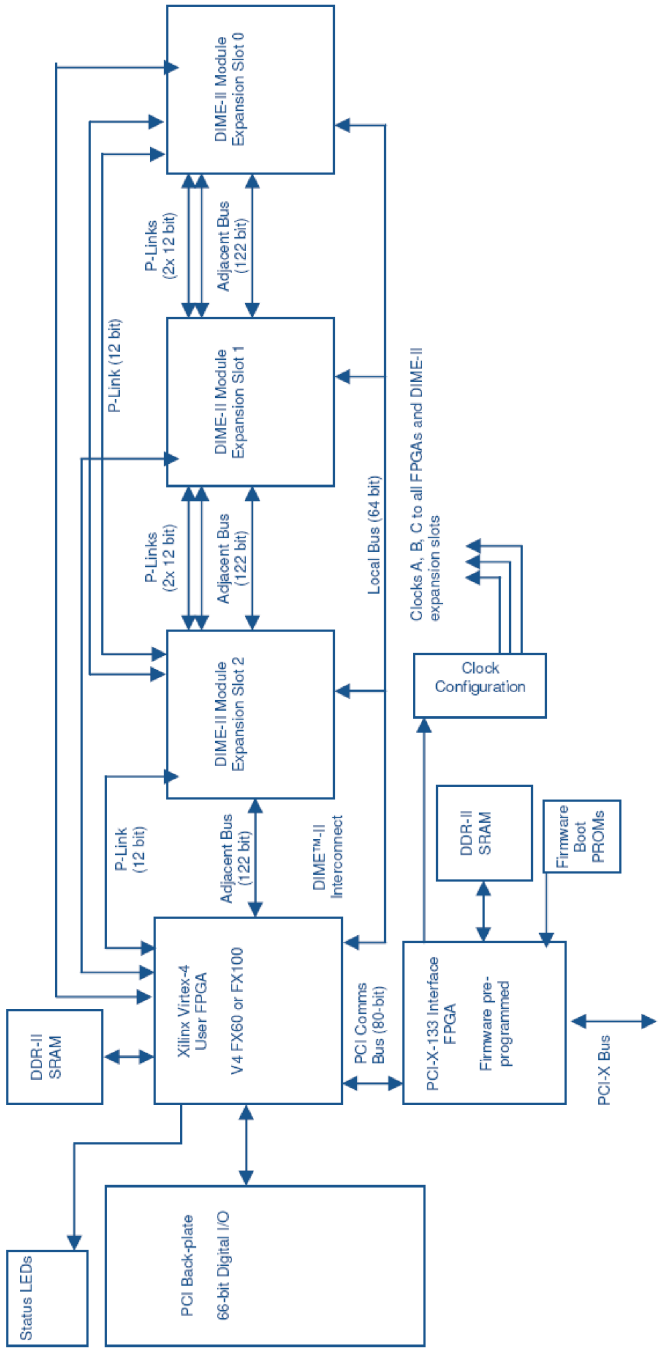


Abbildung 8.1: Blockschaltbild *Nallatech* HW-Emulator (BenNUEY PCI)

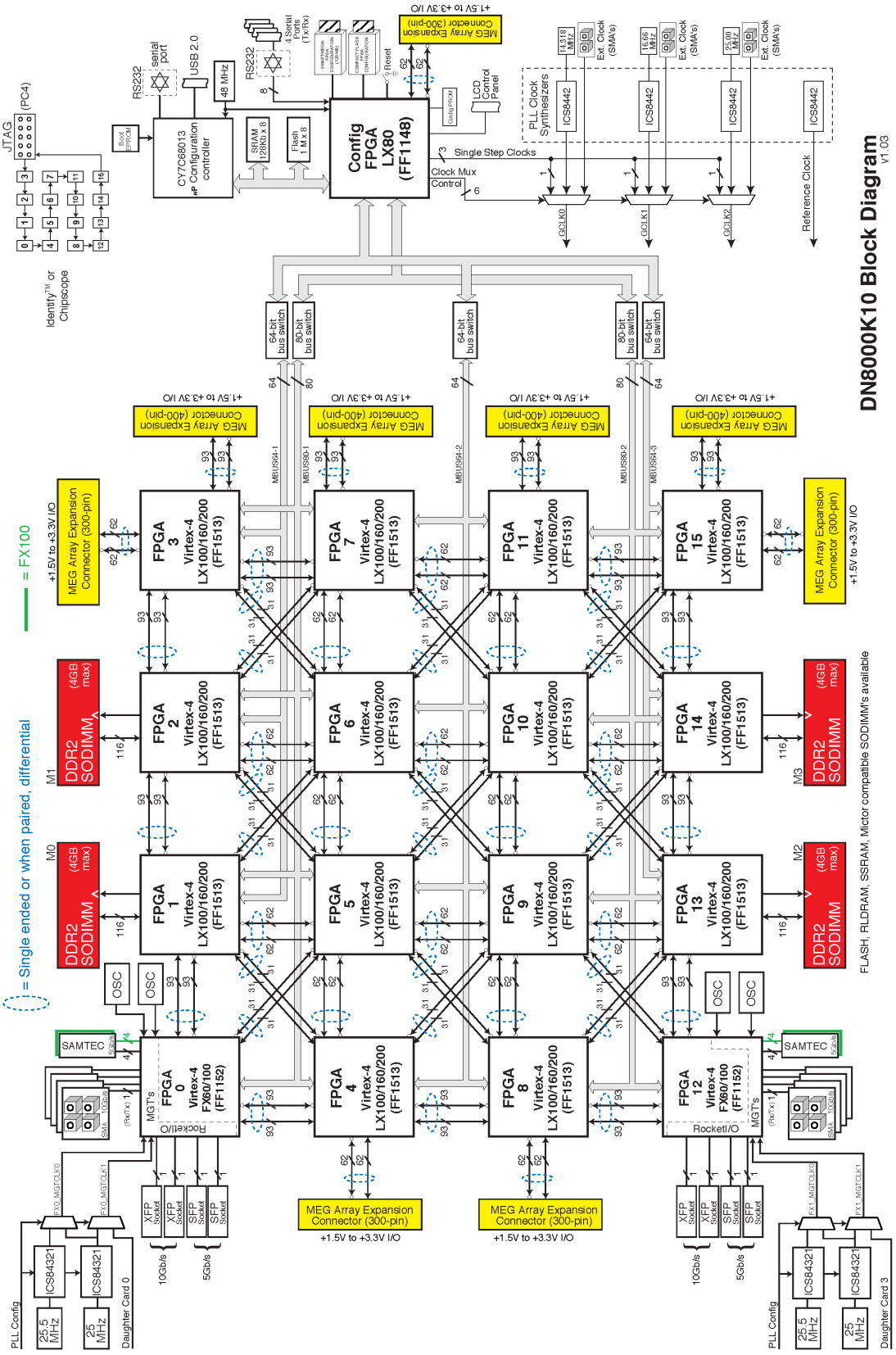


Abbildung 8.2: Blockschaltbild *TheDiniGroup* HW-Emulator (DN8000K10)

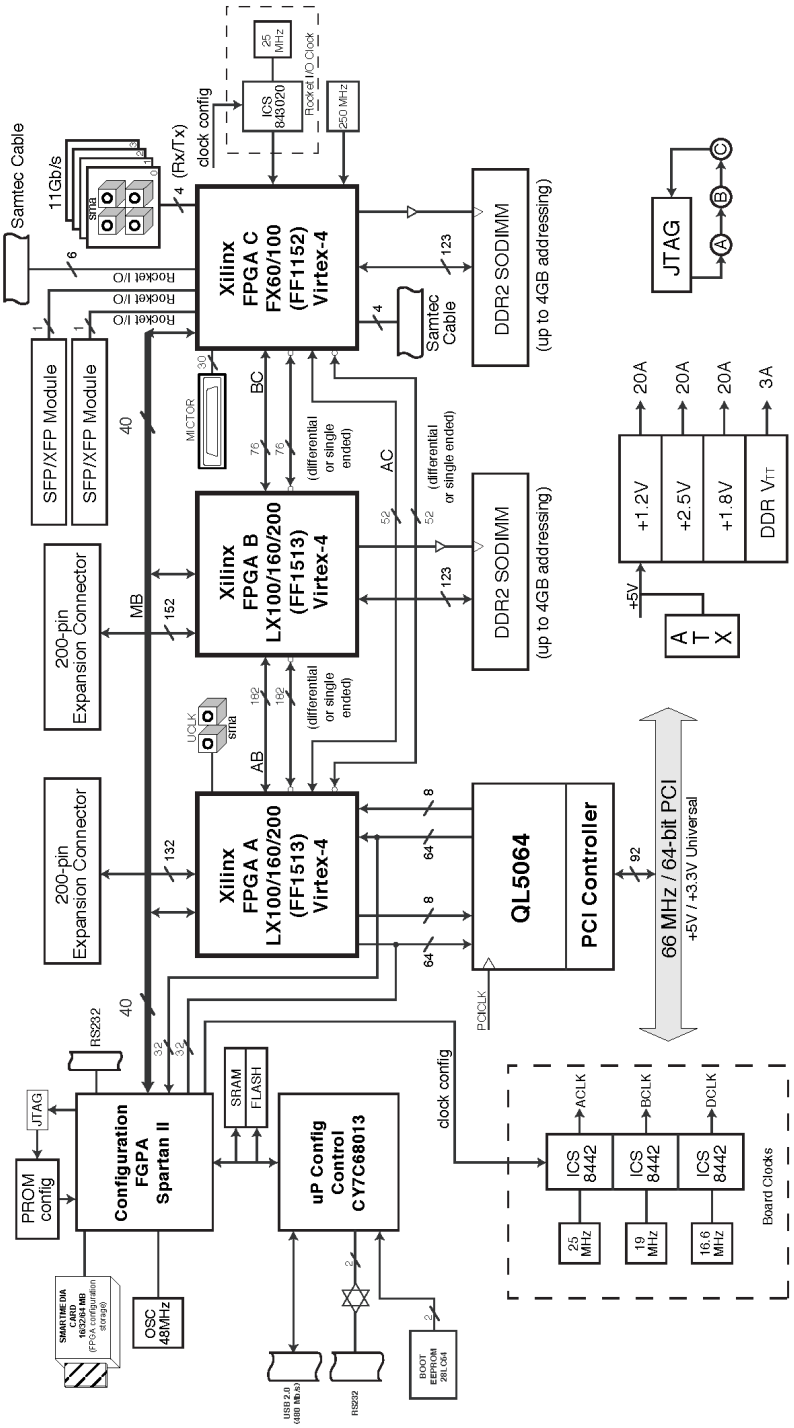
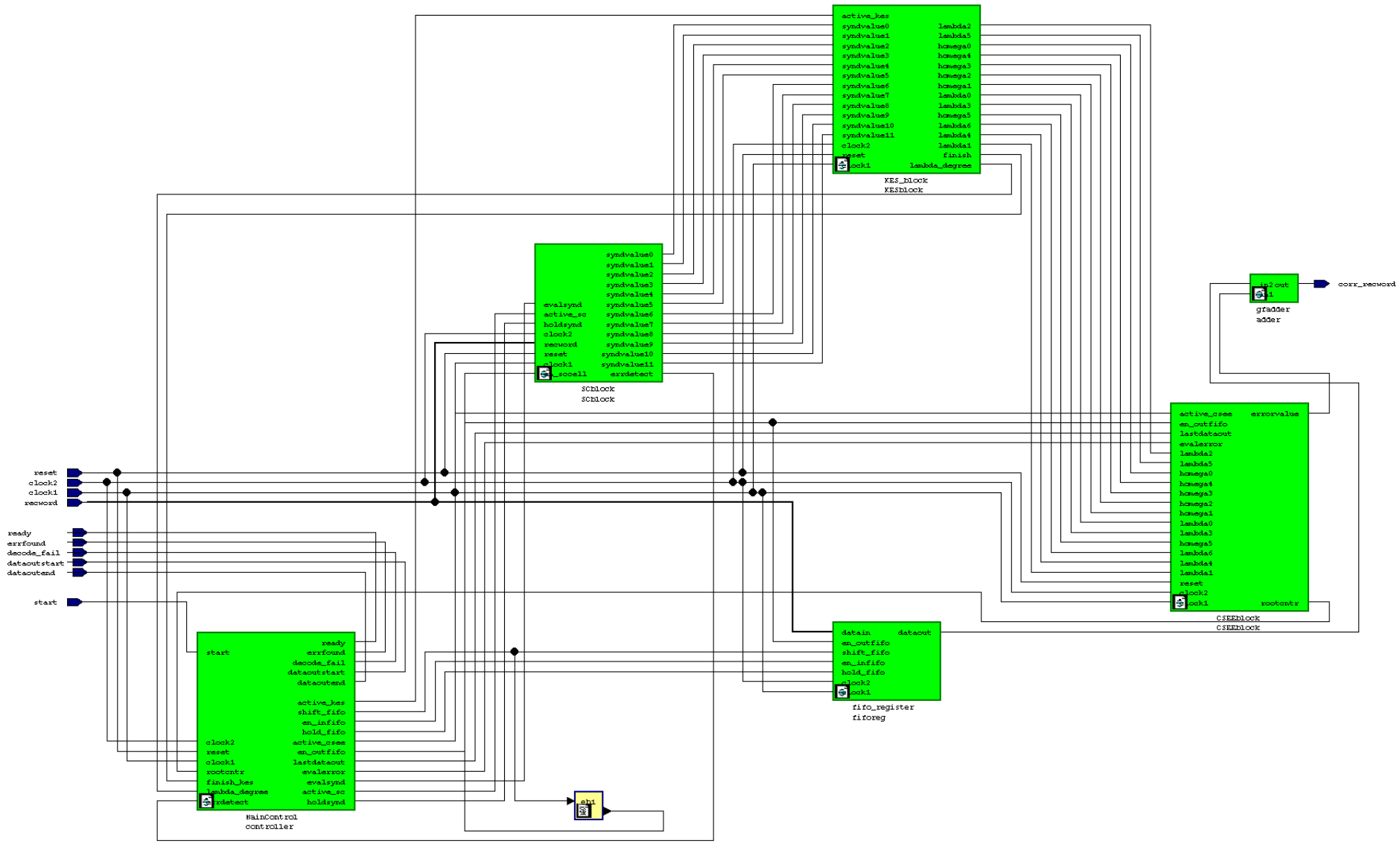


Abbildung 8.3: Blockschaltbild *TheDiniGroup* HW-Emulator (DN8000K10PCI)


active_kes
syndvalue0
syndvalue1
syndvalue2
syndvalue3
syndvalue4
syndvalue5
syndvalue6
syndvalue7
syndvalue8
syndvalue9
syndvalue10
syndvalue11
clock2
reset
clock1
lambda2
lambda5
homega0
homega4
homega3
homega2
homega1
lambda0
lambda3
homega5
lambda6
lambda4
lambda1
finish
lambda_degree
KES_block
KESblock
syndvalue0
syndvalue1
syndvalue2
syndvalue3
syndvalue4
evalsynd
active_sc
holdsynd
clock2
recword
reset
clock1
syndvalue5
syndvalue6
syndvalue7
syndvalue8
syndvalue9
syndvalue10
syndvalue11
errdetect
SCblock
SCblock
in2out
i1
gfadder
adder
corr_recword
active_csee
en_outfifo
lastdataout
evalerror
lambda2
lambda5
homega0
homega4
homega3
homega2
homega1
lambda0
lambda3
homega5
lambda6
lambda4
lambda1
reset
clock2
clock1
errorvalue
rootcntr
CSEEDlock
CSEEDlock
reset
clock2
clock1
recword
ready
errfound
decode_fail
dataoutstart
dataoutend
start
datain
dataout
en_outfifo
shift_fifo
en_infifo
hold_fifo
clock2
clock1
fifo_register
fiforeg
start
ready
errfound
decode_fail
dataoutstart
dataoutend
active_kes
shift_fifo
en_infifo
hold_fifo
active_csee
en_outfifo
lastdataout
evalerror
evalsynd
active_sc
holdsynd
clock2
reset
clock1
rootcntr
finish_kes
lambda_degree
errdetect
MainControl
controller
ehi
i1