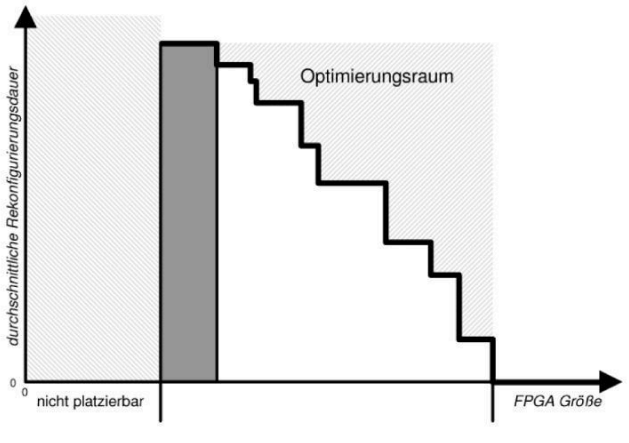
Wissenschaftliche Schriftenreihe
EINGEBETTETE, SELBSTORGANISIERENDE SYSTEME
Band 8

André Meisel

DESIGN FLOW FÜR IP BASIERTE, DYNAMISCH REKONFIGURIERBARE, EINGEBETTETE SYSTEME

Prof. Dr. Wolfram Hardt (Hrsg.)



TECHNISCHE UNIVERSITÄT
CHEMNITZ

Universitätsverlag Chemnitz

André Meisel

Design Flow für IP basierte, dynamisch rekonfigurierbare, eingebettete Systeme

Wissenschaftliche Schriftenreihe

EINGEBETTETE, SELBSTORGANISIERENDE SYSTEME

Band 8

Prof. Dr. Wolfram Hardt (Hrsg.)

André Meisel

DESIGN FLOW FÜR IP BASIERTE, DYNAMISCH REKONFIGURIERBARE, EINGEBETTETE SYSTEME

TECHNISCHE UNIVERSITÄT
CHEMNITZ

Universitätsverlag Chemnitz
2010

Impressum

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Angaben sind im Internet über http://dnb.d-nb.de abrufbar.

Zugl.: Chemnitz, Techn. Univ., Diss., 2010

Technische Universität Chemnitz/Universitätsbibliothek
Universitätsverlag Chemnitz
09107 Chemnitz
http://www.bibliothek.tu-chemnitz.de/UniVerlag/

Herstellung und Auslieferung
Verlagshaus Monsenstein und Vannerdat OHG
Am Hawerkamp 31
48155 Münster
http://www.mv-verlag.de

ISBN 978-3-941003-15-6

urn:nbn:de:bsz:ch1-201000890
URL: http://archiv.tu-chemnitz.de/pub/2010/0089

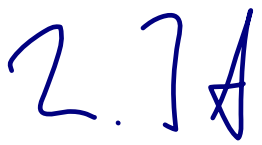
Vorwort zur Wissenschaftlichen Schriftenreihe „Eingebettete, Selbstorganisierende Systeme“

Der achte Band der wissenschaftlichen Schriftenreihe *Eingebettete, Selbstorganisierende Systeme* widmet sich der Synthese von partiell dynamisch rekonfigurierbaren, eingebetteten Systemen.

Mit der Möglichkeit Hardwareblöcke zur Laufzeit auf programmierbaren Bausteinen neu zu konfigurieren, lässt sich eine höhere Flexibilität im Vergleich zu einer Hardwarerealisierung in eingebettete Systeme integrieren. Gleichzeitig sind diese Systeme durch eine gesteigerte Performance gegenüber Software gekennzeichnet. Die Flexibilität kann ausgenutzt werden, um kleinere Schaltkreise bei gleichem Funktionalitätsumfang einzusetzen. Für die Integration von Rekonfigurierung sind zusätzliche Entwurfsschritte im Design Flow notwendig. Herr Meisel stellt hierfür in seiner Arbeit eine Entwurfsmethodik vor.

Schwerpunkte dieser Arbeit sind in der Partitionierung der Systemfunktionalitäten, der Platzierung von rekonfigurierbaren Modulen auf die Schaltkreisfläche und der Hardwaresteuereinheit zu sehen. Durch die Anwendung eines Speicherverwaltungskonzepts auf die dynamischere Rekonfigurierung konnte eine deutliche Reduzierung der benötigten Schaltkreisfläche erzielt werden. Die formale und algorithmische Vorgehensweise bei den einzelnen Entwurfsschritten sind Herrn Meisel sehr gut gelungen und bilden einen in sich geschlossenen Entwicklungsansatz.

Ich freue mich, Herrn Meisel für die Veröffentlichung der Ergebnisse seiner Arbeiten in dieser wissenschaftlichen Schriftenreihe gewonnen zu haben, und wünsche allen Lesern einen interessanten Einblick in die dynamische Rekonfigurierung von eingebetteten Systemen.



Prof. Dr. Wolfram Hardt
Professur Technische Informatik
Dekan der Fakultät für Informatik
Wissenschaftlicher Leiter Universitätsrechenzentrum
Technische Universität Chemnitz
Juni 2010

TECHNISCHE UNIVERSITÄT CHEMNITZ

Fakultät für Informatik

Design Flow für IP basierte, dynamisch rekonfigurierbare, eingebettete Systeme

Dissertation

zur Erlangung des akademischen Grades

Doktoringenieur
(Dr.-Ing.)

vorgelegt

der Fakultät für Informatik
der Technische Universität Chemnitz

von: Dipl.-Inf. André Meisel
geboren am: 01.09.1977
geboren in: Jena

Gutachter: Prof. Dr. rer. nat. habil. Wolfram Hardt
Prof. Dr.-Ing. Dietmar Fey

Chemnitz, den 19. Februar 2010

Danksagung

Für meine Doktorarbeit schulde ich sehr vielen Menschen einen herzlichen Dank. Besonders möchte ich mich bei meinem Doktorvater, Prof. Dr. Wolfram Hardt an der technischen Universität Chemnitz, bedanken. Durch ihn konnte ich als wissenschaftlicher Mitarbeiter an der Professur „Technische Informatik“ die Dissertation bearbeiten und erhielt reichhaltige Unterstützung und Wegweisung zum Gelingen der Arbeit. Ebenso möchte ich mich bei meinen Kollegen an der Professur für die gute Zusammenarbeit bedanken.

Für die Unterstützung bei der Durchführung des Prüfungsverfahrens möchte ich Herr Prof. Dr.-Ing. Dietmar Fey erwähnen und ihm meinen Dank für seine Mühe ausdrücken.

Weiterhin möchte ich meiner lieben Frau Evelyn für ihre Geduld und meinen beiden Kindern, Henry und Annelene, die ihren Vater, während der Promotion, öfter entbehren mussten, von ganzem Herzen danken. Meine Frau hatte während dieser Zeit mir immer wieder den Rücken freigehalten, weshalb ich ihr diese Arbeit widme. Nicht zuletzt gilt auch meinen Eltern, die mir ein Studium an der Universität in Jena ermöglichten und mich auch sonst vielseitig unterstützt haben, mein Dank.

Kurzfassung

Aktuelle, konfigurierbare Schaltkreise (FPGAs) bieten die technische Voraussetzung, um in Hardware realisierte Funktionalität zur Laufzeit auszutauschen. Die sogenannte dynamische Rekonfigurierung ist jedoch nicht in jedem Fall sinnvoll einzusetzen. Einen Vorteil bietet sie, wenn folgende Zielsetzungen umgesetzt werden müssen:

- Verwendung eines kleineren FPGAs bei gleichem Systemfunktionsumfang.
- Dynamische Funktionsmigration zur Verbesserung der Robustheit gegenüber partiellen Systemausfällen.
- Dynamische Funktionserneuerung und -erweiterung in nur aus der Ferne wartbaren Systemen.
- Bereitstellung von Selbstorganisationsmechanismen, wie z.B. die Möglichkeit Funktionen eigenständig anzufordern.

Kennzeichnend für solche Systeme ist, dass ein Teil ihrer Funktionalitäten nicht zu jedem Zeitpunkt aktiv Daten verarbeiten und daher nicht vorgehalten werden müssen. Eine Klasse von Systemen, die diesen Anforderungen in vielen Fällen entspricht, sind die eingebetteten Systeme, da sie als Automaten beschrieben werden können. Die Integration von Rekonfigurierung in diese Systeme erfordert jedoch zusätzlichen Aufwand im Design Flow.

In dieser Forschungsarbeit wird unter Berücksichtigung eines Rekonfigurierungskonzepts dieser Design Flow automatisiert und der Einsatz von kleineren FPGAs fokussiert. Für das Rekonfigurierungskonzept wurde das Overlaying Speicherverwaltungskonzept adaptiert, was eine Minimierung der durchschnittlichen Rekonfigurierungsdauer ermöglicht. Die entwickelten Methoden für die Partitionierung des Systems in funktionale Module, die Platzierung der Komponenten auf dem FPGA und der Steuerung des Rekonfigurierungsprozesses bieten, gegenüber anderen Ansätzen, einem Entwickler spezialisierte und damit optimierte Unterstützung bei der Integration von Rekonfigurierung in ein eingebettetes System.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

In vielen Bereichen des täglichen Lebens haben elektronische Geräte Einzug gehalten. Diese Durchdringung der Technik in jeden Lebensbereich und die immer größer werdende Zahl an Aufgaben und Anforderungen haben digitale Systeme etabliert. Die Flexibilität von digitalen Systemen wird durch eine einfache Anpassbarkeit von Software bewerkstelligt. Technische Verbesserungen, wie z.B. eine bessere Verarbeitungsgeschwindigkeit oder ein geringerer Energiebedarf, sind dagegen mehr auf Optimierungen in der Hardware zurückzuführen. Die rasante Entwicklung von Hardware ermöglicht nicht nur den gegebenen Anforderungen gerecht zu werden, sondern bieten auch die Möglichkeit neue Aufgaben und Funktionen in elektronische Systeme zu integrieren. Ebenso motiviert die Unterscheidung in notwendige und mögliche Funktionalität verschiedene Entwurfsansätze und Systemarchitekturen.

An der TU Chemnitz wurde im Rahmen des Forschungsthemas *Rekonfigurierbare Kommunikationssysteme* der Entwurf von dynamisch rekonfigurierbaren, eingebetteten Systemen untersucht. Hierzu wurde ein Design Flow entwickelt, der die Nutzung der Rekonfigurierung in eingebetteten Systemen mit nur geringem zusätzlichen Entwurfsaufwand für den Entwickler ermöglicht und die Flexibilität von Software in Hardware zur Verfügung stellt. In der vorliegenden Arbeit werden die Entwurfsschritte detailliert beschrieben.

Dieses Kapitel stellt die Motivation für den Design Flow vor und formuliert das Problem und die Aufgabenstellung. Im folgenden Abschnitt 1.1 werden, ausgehend vom Moor'schen Gesetz, die Wichtigkeit und der Entwicklungsbedarf von eingebetteten Systemen beleuchtet. Die Bedeutung der Dynamik im Bereich der Informationstechnik und deren Realisierungsmöglichkeiten sind Thema des Abschnitts 1.2. Auf verschiedene Arten dynamischer Rekonfigurierung und der Modellierung von dynamischen Eigenschaften im Spezifikationsgraph wird im Abschnitt 1.3 näher eingegangen. Im Abschnitt 1.4 wird das Ziel der Arbeit angegeben. Die weitere Strukturierung der Arbeit wird im Abschnitt 1.5 beschrieben.

1.1 Potenzial und Bedarf von eingebetteten Systemen

Die von Moore im Jahr 1965 veröffentlichte These [88], zur Entwicklung der Komplexität von integrierten Schaltkreisen (IC), lässt sich mit geringen Adaptierungen in verschiedenen Bereichen der Rechentechnik bis heute beobachten. Moore behauptete ursprünglich, dass die Komplexität beziehungsweise die Anzahl der Transistoren pro Fläche in einem IC sich schätzungsweise jedes Jahr durch technologische Verbesserungen verdoppeln würde. Es ging ihm ursprünglich nicht um die Rechenleistung, sondern primär um die Kostenminimierung bei der Herstellung von ICs. Zu damaliger Zeit wurden ICs hauptsächlich für das Militär produziert und entwickelt. Moore hatte jedoch die Vorstellung, dass ICs immer billiger und dadurch für jeden zugänglich werden würden. Dieser Punkt gewinnt in der heutigen Zeit wieder neu an Bedeutung. Während vor ca. 10 Jahren noch die Rechengeschwindigkeit und damit einherge-

hend die Taktfrequenz von CPUs (Central Processing Unit) im Mittelpunkt standen, beobachtet man heute mehr und mehr den Einzug von eingebetteten Systemen in fast jeden Bereich des täglichen Lebens.

Eingebettete Systeme sind hierbei informationsverarbeitende Systeme, wie digitale Steuerungen oder Rechner, die in ein meist größeren technischen Kontext integriert sind [80]. Durch die Integration in einen technischen Kontext ist ein informationsverarbeitendes System oft von dem Benutzer des Systems nicht als solches zu erkennen. Eingebettete Systeme werden daher auch mit „Invisible Computing“ bezeichnet.

Für diese aktuelle Entwicklung der Informationstechnik hatte der seinerzeit leitende Wissenschaftler am Xerox-Forschungszentrum im Silicon Valley, Mark Weiser, in seiner Veröffentlichung *The Computer for the 21st Century* [133] von 1991 den Begriff „Ubiquitous Computing“ geprägt. Weiser zeigte auf, dass der Rechner als sichtbares Gerät in den Hintergrund treten und durch neue, kleine, intelligente Gegenstände, die „Ubiquitous Computers“, in weiten Bereichen ersetzen wird. Seiner Meinung nach, ist „Ubiquitous Computing“ nicht die Möglichkeit den Rechner als Gerät überall, zum Beispiel an Strände oder Flughäfen, mitzunehmen. In der Industrie, überwiegend von IBM, wurde die Durchdringung von Informationstechnik aus einem anderen Blickwinkel betrachtet und daher mit „Pervasive Computing“ bezeichnet [81]. Der Fokus lag hier nicht so sehr auf neuen Technologien, als vielmehr auf neuen Anwendungsbereichen, wie zum Beispiel auf Geschäftsprozessen und allgemeinen Lebensbereichen. Ähnlich wie „Pervasive Computing“ in Verbindung mit IBM gebracht wird, ist die Firma Philips der Thematik der „Ambient Intelligence“ zuzuordnen [116]. Diese Entwicklungsrichtung von eingebetteten Systemen zielt mit dem Schwerpunkt der Interaktionen zwischen Geräten und Benutzern u.a. auf den Bereich der intelligenten Gebäudetechnik ab.

Für alle drei genannten Entwicklungstrends sind eingebettete Systeme der technische Ausgangspunkt und bilden einen Großteil der notwendigen technologischen Grundlage. Das Potenzial von eingebetteten Systemen wird ebenfalls deutlich, wenn man deren Einfluss auf Innovationen bei heutigen Kraftfahrzeugen betrachtet. Nur noch ein geringer Prozentsatz der Funktionen eines Autos werden heute noch nicht elektronisch gesteuert, geregelt oder überwacht [42, 130]. Hierbei ist zu beobachten, dass immer mehr neue Fahrerassistenz- und Sicherheitssysteme in das Auto integriert werden. Gleichzeitig treten diese Systeme immer weniger in das Bewusstsein der Fahrer und Beifahrer. Dies hat auch zur Folge, dass diese Systeme vom Kunden nicht extra bezahlt werden wollen. Es ist daher umso wichtiger, das Potenzial von eingebetteten Systemen zu untersuchen. Aus diesem Grund, legt die vorliegende Arbeit ihren Fokus auf eingebettete Systeme.

Die von Moore prognostizierte Verdopplung der Transitorenanzahl innerhalb eines Jahres, wurde 1975 von Moore in einer Rede vor der *Society of Photo-Optical Instrumentation Engineers* angepasst auf eine Verdopplung alle zwei Jahre [89]. Der Grund dafür war, dass sich die Entwicklung der Halbleitertechnik, durch steigende Komplexität der Systeme, verlangsamt hatte. Das Problem die Kapazitäten effizienter zu nutzen existiert bis heute und verdeutlicht den Bedarf an Verbesserungen und Optimierungen, besonders im Bereich der eingebetteten Systeme. Neben der optimalen Ausnutzung der technischen Möglichkeiten von eingebetteten Systemen für bestehende Probleme, ist auch die Entwicklung neuer Systeme mit neuen Funktionen und Aufgaben zu berücksichtigen. Nur durch innovative Entwicklungen ist eine Durchdringung von ICs in allen Lebensbereichen weiter voranzubringen. Aus diesem Bedarf heraus soll in dieser Arbeit der Entwurf von dynamisch veränderbaren, eingebetteten Systemen, als eine neue Systemarchitektur, betrachtet werden.

1.2 Dynamik und integrierte Schaltkreise

In verschiedenen Forschungsbereichen wird der Begriff Dynamik verwendet. Eine Kraft, die auf einen physikalischen Körper einwirkt, hat einen Einfluss auf die Bewegungsvorgänge des Körpers. Die Dynamik in der Physik ist die Lehre über die Wirkung dieser Kräfte. Das Wort Dynamik [griechisch] bedeutet Triebkraft und bezeichnet damit eine auf Veränderung gerichtete Kraft. Während in der Physik mehr die Ursache und damit die Kraft im Mittelpunkt steht, wird mit Dynamik in anderen Fachbereichen die Auswirkung beziehungsweise die Veränderung fokussiert. In der Musik wird beispielsweise das Verhältnis zwischen leisen und lauten Tönen und im Bereich der Versicherungen eine regelmäßige Erhöhung eines Beitrags mit dem Begriff Dynamik beschrieben. In der Informatik ist unter anderem die dynamische Speicherverwaltung zu nennen, bei der zur Laufzeit Speicherblöcke allokiert und wieder freigegeben werden. Allgemein bezeichnet Dynamik hier die Eigenschaft von Hard- oder Software auf geänderte Bedingungen zur Laufzeit zu reagieren. Die in einer Spezifikation beschriebenen Randbedingungen bestimmen den benötigten Dynamikbereich eines eingebetteten Systems. Die Hardware muss dann den Dynamikbereich abdecken, ist aber in den meisten Fällen nicht dynamisch variierbar. Erst die Software eines Systems gewährleistet die dynamische Steuerung, indem zu unterschiedlichen Zeitpunkten die verschiedenen Hardwarekomponenten aktiviert und abgeschaltet werden. Verhält sich ein System über den spezifizierten, notwendigen, Aufgaben erfüllenden Dynamikbereich hinaus deterministisch, handelt es sich um ein robustes System. Es ist robust gegenüber nicht spezifizierten oder nicht sicheren Anforderungen.

Die Flexibilität der eingebetteten Systeme basiert im Wesentlichen auf der eingesetzten Software. Software ist günstiger als Hardware zu produzieren und lässt sich leicht und dynamisch austauschen oder erneuern. Sie besitzt jedoch gegenüber der Hardware eine geringere Performanz und ist nur mit größerem Aufwand zu parallelisieren. In sicherheitskritischen Systemen mit hohen Echtzeitanforderungen sind daher geeignete Hardwarekomponenten für eingebettete Systeme unumgänglich. Die Flexibilität dieser Systeme wird oft nur durch Vorhaltung verschiedener Hardwarefunktionalitäten gewährleistet. Dies führt zu größeren, und damit teureren, Hardwarekomponenten, in denen nicht alle Funktionen gleichzeitig aktiv Daten verarbeiten. Die Unterscheidung in Soft- und Hardware führt zu zwei verschiedenen Ausführungsdimensionen. Software benötigt bei zunehmender Komplexität mehr Zeit zur Ausführung und wird daher als „Computing in Time“ bezeichnet. Im Gegensatz hierzu umschreibt „Computing in Space“ einen größeren Schaltkreis, wenn mehr Komplexität in Hardware realisiert werden muss.

Um die Entwicklungskosten von ASICs (Application Specific Integrated Circuit), insbesondere bei geringen Stückzahlen, zu senken, wurden programmierbare oder auch konfigurierbare Schaltkreise entwickelt. Diese programmierbaren Logik Bausteine (PLD) sind Schaltkreise, die keine vorgegebene, logische Funktion haben. Fertig produzierte PLDs erhalten ihre logischen Funktionen erst durch Programmierung. PLDs unterscheiden sich im Umfang der Funktionalität und der Art des Programmierens. Einfachere Logikbausteine, wie PROMs (Programmable Read Only Memory), PALs (Programmable Array Logic) oder PLAs (Programmable Logic Array), können nur zur Realisierung von kombinatorischer Logik verwendet werden und besitzen unter anderem keine Speicherzellen wie beispielsweise CPLDs (Complex Programmable Logic Device) oder FPGAs (Field Programmable Gate Array). Die meiste Flexibilität bei der Entwicklung von eingebetteten Systemen bieten FPGAs durch ihre programmierbaren Logikzellen, einem konfigurierbaren Verbindungsnetzwerk und speziellen Ein- und Ausgabeblöcken. Diese Flexibilität bildet die technische Grundlage der vorliegenden Arbeit.

Eine wichtige technische Möglichkeit von FPGAs, wie dem Virtex II Pro von Xilinx, ist darüber hinaus die dynamische Rekonfigurierung von Hardwarefunktionalität. Das Konzept der rekonfigurierbaren Rechensysteme stammt aus den 1960er Jahren, als Gerald Estrin in seiner Veröffentlichung „Organization of Computer Systems – The Fixed Plus Variable Structure Computer“ [30, 31] einen Rechner proklamierte, der aus einem Standardprozessor und mehreren konfigurierbaren Hardwarebausteinen besteht. Der Standardprozessor war in diesem Konzept dafür vorgesehen, das Verhalten der rekonfigurierbaren Hardware zu steuern. Heute sind die technischen Möglichkeiten gegeben, um dieses Konzept in einem FPGA zu realisieren. Die dynamisch rekonfigurierbaren FPGAs ermöglichen hierdurch eine weitere Klasse von Dynamik zu unterscheiden, bei der die Flexibilität von Software und die Performance von Hardware kombiniert werden. Folgende vier Anwendungsfälle sind für den Einsatz von Rekonfigurierung denkbar.

- Der Einsatz von kleineren FPGAs, um beispielsweise kostengünstigere eingebettete Systeme herstellen zu können, kann über Rekonfigurierung ermöglicht werden. In diesem Fall sind nur die zu einem Zeitpunkt benötigten Hardwarefunktionalitäten auf dem FPGA abgebildet. Werden zu einem anderen Zeitpunkt weitere Funktionalitäten benötigt, können diese auf externen Speichern vorgehalten und durch Rekonfigurierung im FPGA gegen nicht mehr benötigte Hardwaremodule ausgetauscht werden. Dieses Anwendungsgebiet wird in erster Linie in dieser Arbeit fokussiert.

- Ein weiteres Einsatzgebiet von Rekonfigurierung sind robuste Systeme. Fällt in solch einem System eine Hardwarekomponente aus, kann diese Funktionalität auf redundant vorhandene Hardwareressourcen durch Rekonfigurierung migriert werden. Auch das Nachladen von speziellen Funktionen, die bei Notsituationen Verwendung finden, kann zu einer besseren Robustheit des Systems beitragen.

- Systeme, die nur aus der Ferne gewartet werden können, sind ebenfalls gut geeignet, um Systemerweiterungen und Verbesserungen über die partielle Rekonfigurierung dynamisch zu realisieren.

- Als vierten Anwendungsfall seien, in diesem Zusammenhang, die selbstorganisierenden Systeme genannt. In diesem Fall sind Systeme mit einer komplexen Steuerung gemeint, die es ihnen erlaubt benötigte Hardwarefunktionalität dynamisch nachzuladen.

Im folgenden Kapitel 1.3 wird die Einordnung des Schwerpunkts dieser Arbeit, in den Forschungsbereich der dynamischen Rekonfigurierung, näher beleuchtet.

1.3 Dynamische Rekonfigurierung von Hardware

Durch die Konfigurierung eines FPGAs werden einzelne logische Funktionen zu größeren Hardwarekomponenten kombiniert und durch das Verbindungsnetzwerk verbunden. Rekonfigurierung bedeutet dagegen, das erneute oder wiederholte Einstellen einer Konfiguration. Die Rekonfigurierung kann zu unterschiedlichen Zeitpunkten, beim Start oder während das System läuft, stattfinden (siehe [16]).

Dynamische Rekonfigurierung eines FPGAs bezeichnet die partielle Veränderung der auf dem FPGA konfigurierten Hardwarefunktionalität, während Funktionen, die nicht von der Rekonfigurierung betroffen sind, weiterarbeiten können.

Zu dieser Definition ist noch anzumerken, dass die dynamische Rekonfigurierung immer eine partielle Rekonfigurierung darstellt, wenn die Steuerung der Rekonfigurierung einen Teil des FPGAs belegt. In diesem Fall handelt es sich um ein System mit Selbstrekonfigurierung. Auch bei der Verwendung integrierter Microcontroller zur Steuerung, werden Teile des Verbindungsnetzwerkes fest verschaltet und dürfen nicht durch die Selbstrekonfigurierung modifiziert werden. Nur im ursprünglichen Konzept von Gerald Estrin, können FPGAs vollständig zur Laufzeit rekonfiguriert werden, da die Steuerung in einem externen, nicht konfigurierbaren Standardprozessor implementiert ist.

Neben den Arten der Rekonfigurierung, vollständig vs. partiell, statisch vs. dynamisch, sind auch verschiedene Ebenen der Hardwarerekonfigurierung zu unterscheiden. In der Tabelle 1.1 sind die, auf verschiedenen Abstraktionsebenen, möglichen anpassbaren Elemente aufgezeigt.

Konfigurierbare Objekte / Abstraktionsebene	Kommunikation	Speicher	Datenverarbeitung
Gatter	Schalter Multiplexer	RAM-Organisation	CLB, parametrisierte IP-Blöcke
Analogelemente	Schalter	Energiespeicher	Operationsverst., Kapazitäten
Register / Transfer	Crossbar, Busse	Registerfeldgröße Cachearchitektur	Ausführungseinheiten
Instruktionssatz	Größe Adress / Datenbus	Register Speicherarchitektur	Sonderfunktionen Interrupts
Prozess / Systemarchitektur	Verbindungsnetzwerk	Puffergrößen	Anzahl und Typen von Prozessen, Tasks

Tabelle 1.1: Ebenen der Hardware Rekonfigurierbarkeit [102]

Ausgehend von heutigen FPGAs, bei welchen einzelne Gatter konfiguriert werden können, ist in der Tabelle die Gatterebene als erstes aufgelistet. Diese Rekonfigurierungsebene und besonders die Rekonfigurierung von IPs (Intellectual Property - geistiges Eigentum) als funktionale Hardwareblöcke, bilden das Hauptaugenmerk dieser Arbeit. Die Rekonfigurierung der Gatterebene führt natürlich auch zu einer Veränderung der Register-Transfer-Ebene und auch die Anpassung des Instruktionssatzes ist hierdurch möglich. Ein interessanter Forschungsbereich in dem, von der DFG[1] geförderten, Projekt „Rekonfigurierbare Rechensysteme“ [102], ist auch die Rekonfigurierung analoger Elemente, die aber, im hier vorliegenden Ansatz, nicht beleuchtet wird. Die dynamische Veränderung eines Systems kann auf drei verschiedenen Strukturen durchgeführt werden, auf der Kommunikationsstruktur, der Speicherstrukur und auf der Struktur für die Datenverarbeitung. Speicher und auch datenverarbeitende Komponenten werden mit der Kommunikationsstruktur verbunden, weshalb diese einen zentralen Punkt in der Tabelle 1.1 und in einem System einnimmt. Dieser enge Zusammenhang wird auch bei dem erweiterten Datenflussgraph, dem Problemgraph, berücksichtigt [106]. Der Problemgraph ist ein um Kommu-

[1] Deutsche Forschungsgemeinschaft - http://www.dfg.de

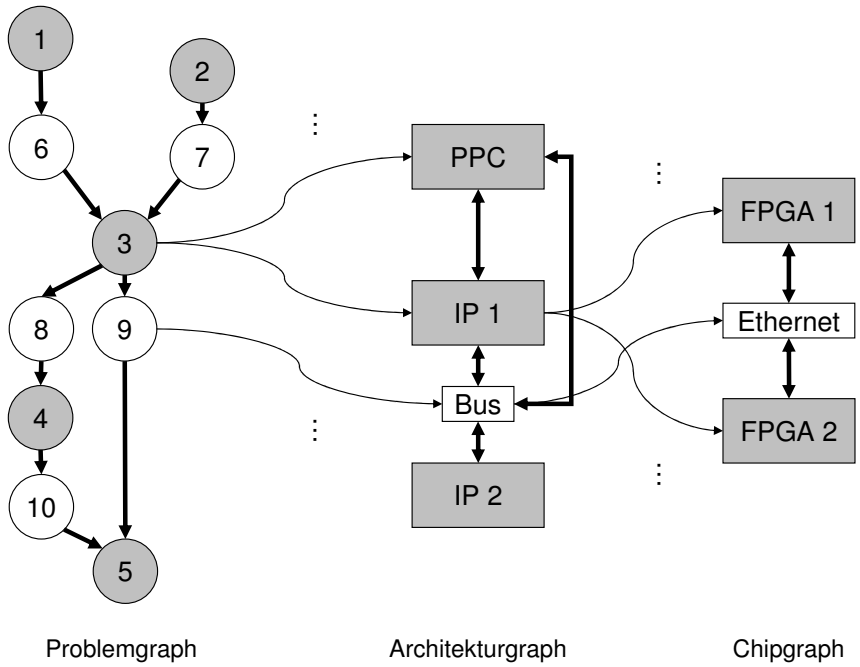


Abbildung 1.1: Spezifikationsgraph

nikationsknoten erweiterter Datenflussgraph und verdeutlicht, dass Datenfluss nur über entsprechende Kommunikationsmedien erfolgen kann. Ein Beispiel für einen Problemgraph ist in Bild 1.1 links zu sehen. Die weißen Knoten repräsentieren in diesem Beispiel die Kommunikationsknoten. Weitere Graphen der Abbildung 1.1 sind der Architekturgraph, eine Spezifikation der Zielarchitektur, und der Chipgraph, der eine weitere Ebene der Systemarchitektur beschreibt. Alle drei Graphen ergänzt um Abbildungskanten, die Synthesemöglichkeiten spezifizieren, bilden zusammen einen Spezifikationsgraph. Um dynamisches Verhalten eines Systems mit diesem Modell zu spezifizieren, sind nicht nur die räumlichen Abbildungen zu betrachten. Unter einer räumlichen Abbildung ist beispielsweise eine Abbildung eines funktionalen Knoten auf eine funktionale Ressource der Architektur und auch eine Abbildung einer Ressource auf einen Knoten im Chipgraph zu verstehen. Neben einer räumlichen, Hardwarefläche belegenden Abbildung kann auch eine zeitliche Abbildung definiert werden. Hierbei werden unterschiedliche Allokationen[2] und Bindungen[3] festgelegt, die dann dynamisch zu verschiedenen Zeitpunkten der Systemlaufzeit ausgetauscht und aktiviert werden. Bei der Festlegung der Abbildung von Problemgraphknoten auf den Architekturgraph wird entschieden, ob ein Problem in Hardware oder in Software gelöst wird. Die räumliche Abbildung vom Architekturgraph auf den Chipgraph dagegen, spezifiziert die technologische Umsetzung der Architektur. Die Flexibilität eines Systems kann also an zwei verschiedenen Stellen der Spezifikation beeinflusst werden. Einmal kann sie verbessert werden, indem mehr Funktionalität in Software realisiert wird und zum an-

[2]Eine Allokation ist Teilmenge des Spezifikationsgraphen ohne die Abbildungskanten und beschreibt alle notwendigen und verwendeten Knoten bzw. Kanten des Graphen.

[3]Eine Bindung beschreibt eine Menge von Abbildungskanten zwischen den Graphen des Spezifikationsgraphs

deren durch die Abbildung von Hardwarefunktionalität des Architekturgraphen auf dynamisch rekonfigurierbare FPGAs im Chipgraph.

Die Synthese eines dynamisch rekonfigurierbaren, eingebetteten Systems erfordert auf Grund der zeitlichen Veränderung der Hardwarefunktionalität, zusätzliche Designentscheidungen. Im Design Flow solcher Systeme ist die Modellierung der Dynamik und auch die zeitabhängige Partitionierung der Funktionalitäten zu berücksichtigen. Die vorliegende Arbeit geht dabei von einem einzelnen FPGA, als Zielplattform für ein rekonfigurierbares System, aus und es wird ohne Beschränkung der Allgemeinheit auf die Selbstrekonfigurierung fokussiert.

1.4 Ziel der Arbeit

Die Integration von Rekonfigurierung in eingebettete Systeme erfordert zusätzliche Entwurfsschritte gegenüber Systemen, die nicht dynamisch rekonfigurierbar sind. Das Ziel dieser Arbeit ist es, einen Design Flow vorzustellen, der die Integration unterstützt und Standardsyntheseschritte automatisiert. Der Design Flow findet Anwendung, um Systeme zu erstellen, die neben der Eigenschaft der dynamischen Rekonfigurierung folgende Eigenschaften besitzen:

- Die Systeme bestehen aus verschiedener Hardwarefunktionalität, die durch die Integration von IP Komponenten realisiert wird. Ziel ist ein modular aufgebautes System zu erstellen.
- Die eingesetzten IP Komponenten können mit unterschiedlichen Schnittstellen versehen sein. Die Kommunikation zwischen den IP Komponenten kann natürlich auch über einheitliche Schnittstellen und Busse realisiert werden.
- Die dynamische Rekonfigurierung und das Einbinden von redundant vorhandenen Hardwarekomponenten, führt bei diesen Systemen zu einem robusteren Verhalten gegenüber sich verändernden Umwelteinflüssen.
- Die Systeme sollen jeweils auf einem FPGA implementiert werden. Die vorgestellten Algorithmen sind jedoch nicht auf Ein-Chip-Lösungen beschränkt, sondern können auch für verteilte Systeme angewandt werden.

Die Konzeption und Entwicklung des Design Flows beinhaltet folgende Punkte:

Formalisierung: In Anlehnung an das Overlaying Konzept[4] der Programmiersprache Pascal [26], soll der Entwickler eines dynamisch rekonfigurierbaren Systems, durch die Zusammenfassung von Hardwarefunktionalität, die zu einem Zeitpunkt aktiv auf den FPGA funktionieren soll, die Menge an Konfigurationen beschreiben, die überlappt beziehungsweise rekonfiguriert werden können. Aus dieser Eingabe heraus muss eine formale Beschreibung des Systems generiert werden, damit weitere Entwurfsschritte automatisiert werden können. Die formale Beschreibung ist weiterhin notwendig, um das modellierte System optimal auf die zur Verfügung stehenden Hardwareressourcen abzubilden. Diese Optimierung ist über geeignete Funktionen, die die Platzierungs-, Kommunikations- und Rekonfigurierungskosten bestimmen, zu steuern.

[4] Auf das „Overlaying Konzept“ wird im Kapitel 2.4 näher eingegangen

Methoden: Ausgehend von der formalen Beschreibung eines rekonfigurierbaren Systems, sind die verschiedenen Syntheseschritte zu automatisieren. Die dafür benötigten Methoden umfassen die Aufteilung des beschriebenen Systems in austauschbare Module, die Umsetzung der Optimierungsschritte und die Generierung der Modulkommunikationskanäle. Durch das, auf Rekonfigurierung übertragende, Overlaying Konzept existierten keine geeigneten Methoden für die Automatisierung der Synthese und der Optimierung der rekonfigurierbaren Systeme. Das Konzept definiert einen abgeschlossenen Rahmen für die Methoden.

Werkzeug: Das Konzept und die entwickelten Methoden sollen in einem durchgängigen Entwicklerwerkzeug Anwendung finden. Mit diesem Werkzeug soll es einem Entwickler möglich sein,

- ein dynamisch rekonfigurierbares System zu definieren,
- Syntheseschritte zu aktivieren, optimieren und kontrollieren,
- und unter Verwendung von proprietären, zu den FPGAs gehörenden Synthesewerkzeugen dynamisch rekonfigurierbare Systeme zu generieren.

Eine Bewertung der Konzepte wird an verschieden Beispielen durchgeführt.

1.5 Struktur der Arbeit

Die im vorhergehenden Abschnitt 1.4 beschriebenen Ziele erfordern für die Realisierung technische Grundlagen. Die Grundlagen werden im folgenden Kapitel 2 beschrieben. In diesem Kapitel wird auf Eigenschaften von eingebetteten Systemen, den Aufbau von FPGAs und auf die Entwurfsschritte für eine FPGA Implementierung eingegangen. Kapitel 2 dient weiterhin der Vorstellung des, dieser Arbeit zu Grunde liegenden, Design Flows und des Overlaying Konzepts, das auf Rekonfigurierung adaptiert wird. Das Rekonfigurierungskonzept ist für die Verwendung von IPs geeignet, so dass Methoden und das mit XML beschriebene IPQ Format aus dem IPQ Projekt beleuchtet werden.

Bestehende Arbeiten zu den Problembereichen, Systempartitionierung, Modulplatzierung und Steuerung von Rekonfigurierungen, im Entwurf von rekonfigurierbaren Systemen werden im Kapitel 3 angesprochen.

Mit dem Kapitel 4 beginnt der Hauptteil dieser Arbeit. Dort wird eine automatisierte Partitionierung eines dynamisch rekonfigurierbaren Systems, die auf einem Konfigurationskonzept basiert und kritische Datenpfade berücksichtigt, vorgestellt. Bei diesem Verfahren kommt für die Bestimmung der Systemmodule eine Äquivalenzklassenbestimmung zum Einsatz. Dieses Kapitel wird abgeschlossen durch eine Betrachtung von Robustheitsaspekten in dynamisch rekonfigurierbaren Systemen.

Aufbauend auf diesem Partitionierungsverfahren wird im Kapitel 5 die Frage der dynamischen Platzierung von Modulen im FPGA beantwortet. Das entwickelte Verfahren besteht aus der Bestimmung der Overlaying-Bereiche und der Abbildung dieser Flächen auf den Schaltkreis. Dem Hauptnachteil der Rekonfigurierung, die Unterbrechung von einzelnen Teilen des Systems während einer Rekonfigurierung, wird mit einer Minimierung der durchschnittlichen Rekonfigurierungsdauer begegnet. Des Weiteren wird in diesem Kapitel die automatische Generierung der Kommunikationsverbindungen zwischen den Overlaying-Bereichen näher dargestellt.

Jede Rekonfigurierung besteht aus mehreren Schritten und muss daher gesteuert werden. Eine in Hardware realisierte Steuerung mit ihren notwendigen Schnittstellen und Datenflüssen wird im Kapitel 6 beschrieben. Ziel dieses Kapitels ist es, die Vorteile gegenüber einer prozessorgesteuerten Rekonfigurierung aufzuzeigen und die Selbstrekonfigurierung für eingebettete Systeme zu motivieren.

Im Kapitel 7 wird die im Rahmen dieser Arbeit entwickelte *System Design Gate* Software vorgestellt. Zusätzlich wird die Software zur Bestimmung von Modulgrößen im FPGA kurz erläutert und die automatische Generierung von Bus Makros betrachtet. Die Software dient der Erstellung einer Systembeschreibung in XML und der Ansteuerung der Design Flow Phasen.

Die Arbeit wird mit einer Zusammenfassung und einem Ausblick abgeschlossen.

1.6 Zusammenfassung

In diesem Kapitel wurde das Gebiet der dynamischen Rekonfigurierung von Hardwarefunktionalität für eingebettete Systeme motiviert. Neben der immer weiter wachsenden Durchdringung von eingebetteten Systemen, wurde auf die Bedeutung von Dynamik in der Physik und in der Informationstechnik eingegangen. Rekonfigurierungen in eingebetteten Systemen können verschiedene Strukturen und Abstraktionsebenen betreffen. Dieses Kapitel ist auf diese Rekonfigurierungsarten eingegangen, zeigte den Fokus dieser Arbeit auf und motivierte anhand des Spezifikationsgraphen die Entwicklung eines Design Flow für rekonfigurierbare Systeme. Die Übertragung des Overlaying Konzept auf Rekonfigurierung, die Automatisierung der Entwurfsschritte und die Minimierung der Rekonfigurierungskosten in der Entwurfsphase sind Schwerpunkte dieser Arbeit. Die Herausforderung an diesen Design Flow und die Ziele der Arbeit wurden in diesem Kapitel erläutert.

2 Grundlagen

Nach der einleitenden Motivation, dynamische Rekonfigurierungen in eingebetteten Systemen einzusetzen, werden im Folgenden grundlegende Informationen, auf denen die weiteren Ausführungen aufbauen, dargestellt. Nicht in jedem System ist es sinnvoll dynamische Rekonfigurierung einzusetzen. Daher wird in dieser Arbeit die Menge der möglichen Systeme auf eingebettete Systeme eingeschränkt, deren Eigenschaften in diesem Kapitel im Detail beschrieben werden. Ebenso werden die technischen Voraussetzungen, um dynamische Rekonfigurierungen realisieren zu können, vorgestellt und die Besonderheiten im Entwurfsablauf gegenüber Standard Design Flows erläutert. Eine weitere Fragestellung bildet die Auswahl eines geeigneten Konzeptes für den effizienten Einsatz von Rekonfigurierung in eingebetteten Systemen.

Im folgenden Abschnitt 2.1 wird auf die Eigenschaften von eingebetteten Systemen eingegangen. Die Zielhardware für dynamische Rekonfigurierung, die FPGAs, und der Entwurfsablauf für diese Systeme sind Thema der Abschnitte 2.2 und 2.3. Neben den technischen Gegebenheiten werden im Abschnitt 2.4 verschiedene Rekonfigurierungskonzepte vorgestellt. Der Abschnitt 2.5 vor der Zusammenfassung des Kapitels wird verwendet, um Möglichkeiten der Wiederverwendung von Hardwarebausteinen im Kontext von Rekonfigurierung zu beschreiben.

2.1 Eingebettete Systeme

Eingebettete Systeme werden in vielen Anwendungsbereichen eingesetzt. Im Automobilbereich übernehmen sie Aufgaben der Steuerung und der Fahrerassistenz. Neben Flugzeugen und Schiffen, sind auch Verarbeitungs- und Überwachungsprozesse in Industrieanlagen mit digitalen Steuersystemen ausgestattet. Im privaten Wohnraum sind in Waschmaschine, Kaffeeautomat, Mikrowelle und auch der Heizanlage eingebettete Systeme zu finden. Marwedel stellt in seinem Buch „Eingebettete Systeme“ [80] folgende Charakteristiken für diese Systeme dar.

Eingebettete Systeme sind durch die Integration in ihre physikalische Umwelt mit Sensoren und Aktoren ausgestattet. Da sie im Allgemeinen für den Anwender nicht sichtbar sind und bei sicherheitskritischen Anwendungen eingesetzt werden, müssen sie verlässlich sein. Unter verlässlich fasst Marwedel die Begriffe Zuverlässigkeit[1], Wartbarkeit[2], Verfügbarkeit[3], Sicherheit[4] und Integrität[5] zusammen. Ein weiterer wichtiger Aspekt von eingebetteten Systemen ist die Effizienz, bezüglich des Energieverbrauchs, der Codegröße, der Laufzeit, dem Gewicht und damit der Baugröße, sowie den Herstellungskosten. Die Effizienz von eingebetteten Systemen kann durch die Verwendung von dynamischer Rekonfigurierung deutlich gesteigert werden, im besonderen bei der Baugröße. Rekonfigurierung ermöglicht es bei gleichem Funktionsumfang

[1] ist die Wahrscheinlichkeit, dass ein System, nicht ausfällt

[2] ist die Wahrscheinlichkeit, dass ein ausgefallenes System innerhalb einer bestimmten Zeitspanne wieder repariert werden kann

[3] ist die Wahrscheinlichkeit, dass ein System korrekt arbeitet

[4] ist die Eigenschaft, der Umwelt keinen Schaden zuzufügen

[5] ist die Eigenschaft, vertrauliche Daten sicher zu verwalten

kleinere, kostengünstigere FPGAs einzusetzen. Voraussetzung hierfür ist ein System, das nicht alle Funktionalitäten gleichzeitig vorhalten muss.

Eingebettete Systeme sind meistens applikationsspezifisch, um die Systemverlässlichkeit besser zu garantieren und Ressourcen effizient auszunutzen. Als ein weiteres Merkmal ist die öfter benötigte Echtzeitfähigkeit der eingebetteten Systeme zu nennen. Beim Entwurf von echtzeitfähigen, dynamisch rekonfigurierbaren Systemen muss der Entwickler die Zeitanforderungen bei der Partitionierung des Systems berücksichtigen, weil durch das dynamische Aktualisieren der Hardwarefunktionalität Zeitverzögerungen auftreten können. Eingebettete Systeme sind typischerweise reaktive Systeme und können mit endlichen Automaten modelliert werden. Diese Eigenschaft ist eine wichtige Grundlage für den Einsatz von Rekonfigurierung und es wird daher im folgenden Absatz näher darauf gegangen. Eine reaktives System ist ein System, das in kontinuierlicher Interaktion mit seiner Umgebung steht und mit einer Geschwindigkeit arbeitet, die von seiner Umwelt vorgegeben wird [11]. Aus der Möglichkeit heraus, mit der Umwelt zu interagieren, sind die meisten eingebetteten Systeme hybrid aufgebaut. Es existieren sowohl analoge als auch digitale Komponenten. Zu den analogen Systemmodulen gehören Sensoren und Aktoren für die Interaktion mit der Umwelt bzw. dem Haltersystem. Die Funktionalität der eingebetteten Systeme wird zu einem großen Teil durch digitale Hardware- und Softwarekomponenten realisiert. Je nach Einsatzgebiet werden verschiedene Schaltkreistypen zur Datenverarbeitung eingesetzt. Zwei Schaltkreiseigenschaften, die in diesem Zusammenhang hervorzuheben sind, sind die Flexibilität und der Energieverbrauch. Gegenüber den ASICs, die meist für ihre speziellen Aufgaben einen optimalen Energieverbrauch haben, sind programmierbare Schaltkreise kennzeichnend für ihre Flexibilität. Die kostengünstige Flexibilität von FPGAs und die Möglichkeit diese Fähigkeit dynamisch einzusetzen sind Eigenschaften, die bei eingebetteten Systemen zur Minimierung der Größe und Kosten genutzt werden können. Dadurch ist es möglich Systeme mit neuer Funktionalität, ohne Austausch der Hardwarekomponenten und unter Umständen per Fernwartung, auszustatten.

Bei der Modellierung reaktiver Systeme mittels endlicher Automaten [55] umfasst das Eingabealphabet $\sum$ die Menge der vom eingebetteten System verarbeitbaren Sensorwerte. Analog hierzu setzt sich die Menge der Symbole, des Ausgabealphabets Ω, aus Steueranweisungen und Signalen zusammen. Eingebettete Systeme beeinflussen oder steuern ihren technischen Kontext und sind daher oft als Steuerautomaten, mit verschiedenen Systemzuständen Q, implementiert. Die Startkonfiguration dieser Systeme kann als Startzustand q_0 aufgefasst werden. In Abhängigkeit der Komplexität des eingebetteten Systems kann es ein oder mehrere Endzustände $f \in F$ geben. Die Zustandsübergangsfunktion δ beschreibt den Wechsel zwischen den verschiedenen Systemkonfigurationen (Formel (2.2)) und die Ausgabefunktion λ bildet in Abhängigkeit der Eingabe und des aktuellen Zustands auf das Ausgabealphabet ab (Formel (2.3)). Ein eingebettetes System, als Automat modelliert, kann daher als ein 7-Tupel aufgefasst werden (Formel (2.1)).

$$A = (Q, \sum, \Omega, \delta, \lambda, q_0, F) \tag{2.1}$$

$$\delta : Q \times \sum \rightarrow Q \tag{2.2}$$

$$\lambda : Q \times \sum \rightarrow \Omega \tag{2.3}$$

Rekonfigurierbare eingebettete Systeme lassen sich ebenfalls mit endlichen Automaten modellieren. Der Unterschied zu nicht rekonfigurierbaren Systemen besteht darin, dass ein Teil der Zustandsübergänge einen Austausch der Hardwarefunktionalität bedeuten können. Zur Unter-

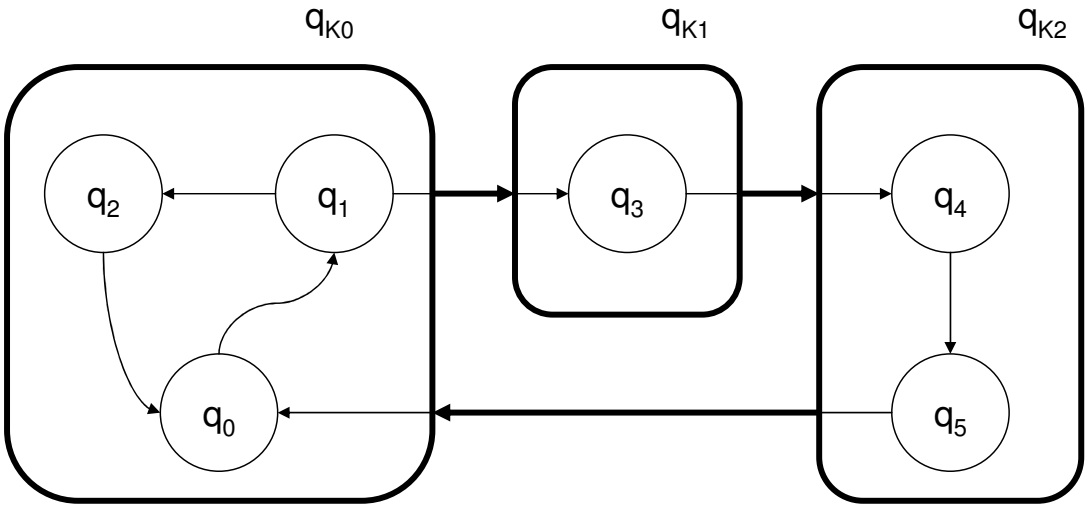


Abbildung 2.1: Modellierungsautomat für rekonfigurierbare, eingebettete Systeme

scheidung der verschiedenen Zustandsübergänge wird eine Hardwarekonfigurationsmenge Q_K definiert.

$$q_K \in Q_K : q_K \subseteq Q \tag{2.4}$$

Eine Hardwarekonfiguration q_K ist eine Teilmenge der Systemzustände Q (Formel (2.4)). Da der neue Automat $R = (Q_K, \sum, \Omega, \delta, \lambda, q_0, F)$ sich nur in der Zustandsmenge von Automat A unterscheidet, können beide Automaten A und R in einen Modellierungsautomat zusammengefasst werden. In der Abbildung 2.1 ist ein solcher Automat zu sehen. Die abgerundeten Rechtecke der Abbildung 2.1 stellen die verschiedenen Hardwarekonfigurationen q_{K0} bis q_{K2} dar und die Zustandsübergänge sind als Pfeile dargestellt. Fett gedruckte Pfeile symbolisieren eine Hardwarerekonfigurierung. Wenn mehrere Systemzustände in einer Hardwarekonfiguration enthalten und verbunden sind (Abbildung 2.1 - z.B. Rechteck q_{K0}), existieren Zustandsübergänge, die keine Rekonfigurierung der Hardwarefunktionalitäten zufolge haben.

Ergänzend zu dieser Einführung sei auf die Habilitationsschrift „Integration von Verzögerungszeit-Invarianz in den Entwurf eingebetteter Systeme“ [49] hingewiesen. Hardt führt hier eingebettete Systeme anhand ihrer Haupteigenschaften ein und gibt einen Vergleich zu General Purpose Computer (GPC).

2.2 Field Programmable Gate Arrays

Das Konzept der vorliegenden Arbeit basiert auf den Möglichkeiten heutiger FPGAs. FPGAs gehören zu der Gruppe der programmierbaren[6] Logikschaltkreise, deren endgültige Funktionalität erst durch einen Systementwickler festgelegt wird. Es können zwei Gruppen von programmierbaren Logikbausteinen unterschieden werden.

[6]Programmierung bedeutet in diesem Zusammenhang nicht, dass ein ablauffähiges Softwareprogramm, wie bei einem Mikroprozessor, geladen wird, sondern es werden komplexe Hardwarefunktionalitäten konfiguriert.

Die eine Gruppe umfasst Schaltkreise mit einer einfachen Schaltungskomplexität, zu denen beispielsweise die PROM und PAL Bausteine zählen. Für weiterführende Informationen zu diesen SPLDs (Simple Programmable Logic Device) wird auf die Referenzen [5, 12, 71] hingewiesen.

Als komplexe Logikbausteine werden hingegen die Vertreter der zweiten Gruppe, die CPLDs und FPGAs, bezeichnet. CPLDs bestehen aus vielen SPLDs, die über eine interne Schaltmatrix miteinander verbunden sind. Die I/O-Blöcke für die Ein- bzw. Ausgabepins sind über diese Matrix mit den SPLDs verbunden. Der homogene Aufbau der CPLDs ermöglicht, im Unterschied zu den FPGAs, eine exakte Bestimmung der Durchlaufzeiten, da die Geschwindigkeit des Systems nicht abhängig ist von der Größe der Schaltung. Weitere Eigenschaften von CPLDs und FPGAs sind in der Tabelle 2.1 gegenübergestellt.

Eigenschaft	CPLD	FPGA
Geschwindigkeit abhängig von der Schaltung	nein	ja
Art der Logikblöcke	UND/ODER-Matrix	feinkörnig
Stromverbrauch	hoch bis sehr hoch	gering bis mittel
Programmierung	EPROM, EEPROM, Flash	SRAM, Antifuse, Flash
erreichbare Ausnutzung	40% bis 60%	50% bis 95%
Geschwindigkeit	hoch	mittel ist hoch
Preis pro Gatter	mittel bis hoch	gering bis hoch

Tabelle 2.1: Eigenschaften von CPLDs und FPGAs [131]

Der prinzipielle Aufbau von FPGAs unterscheidet sich wesentlich von den CPLDs. An der Peripherie der FPGAs befinden sich die I/O Blöcke und im Zentrum existieren Logikblöcke und Verbindungsstrukturen. Die eigentliche Funktionalität wird bei FPGAs in dem konfigurierbaren Feld von Logikblöcken realisiert. Diese konfigurierbaren Logikblöcke (CLBs) sind, nicht wie bei den CPLDs aus UND/ODER-Matrizen, sondern aus mehreren Look Up Tables (LUT), Speicherzellen und Verknüpfungselementen aufgebaut. Der Aufbau der CLBs unterscheidet sich je nach Hersteller und Schaltkreisfamilie. Im Folgenden wird der Aufbau der CLBs des für diese Arbeit zur Verfügung stehenden FPGAs, Virtex II Pro der Firma Xilinx, beschrieben.

Der Aufbau eines CLBs der Virtex II Pro Familie ist in der Abbildung 2.2 dargestellt. Ein CLB besteht aus vier gleichartig aufgebauten Slices. Jeder Slice ist mit der benachbarten Switch Matrix verbunden, um die Ein- und Ausgänge an die globale Verbindungsstruktur anzuschließen. Weiterhin sind die Slices für eine schnelle Kommunikation direkt mit benachbarten CLBs verbunden. Zwischen den Slices existieren, zur Realisierung von Schieberegistern und Addierern, zusätzliche Verbindungen. Ein Slice ist aus zwei LUTs, zwei Speicherzellen (Flip Flop) sowie Verknüpfungselementen für arithmetische Funktionen aufgebaut. Mit einer LUT kann eine beliebige Funktion mit vier Eingangsvariablen realisiert werden. Hierfür stehen an jeder LUT vier Eingänge und ein Ausgang zur Verfügung. Außerdem kann eine LUT auch noch als 16x1 Speicher und als 16 Bit Schieberegister konfiguriert werden. Die Speicherelemente können als taktzustandsgesteuertes Latch oder als taktflankengesteuertes FlipFlop betrieben werden. Um Funktionen mit mehr als vier Variablen zu realisieren, können LUTs über Multiplexer verbunden werden. In jedem Slice existieren jeweils zwei solcher Multiplexer. Weitere Informationen

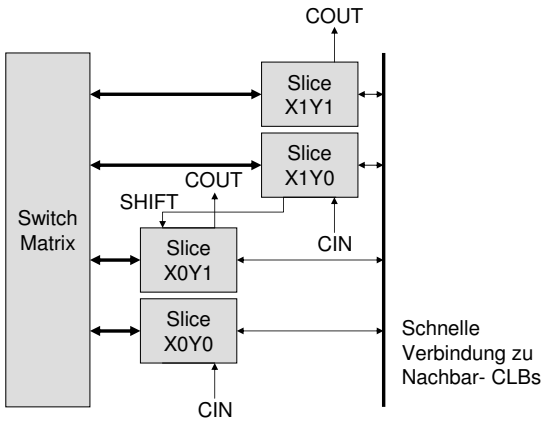


Abbildung 2.2: CLB der Virtex II Pro Familie

zum Virtex II Pro FPGA können dem Datenblatt [147] entnommen werden.

FPGAs und auch die CPLDs sind über verschiedenen Technologien zu konfigurieren. Prinzipiell ist zwischen reversiblen und irreversiblen Programmiertechnologien zu unterscheiden. Bei den reversiblen Technologien werden Speicherzellen genutzt, um eine Konfiguration abzulegen. Hierzu werden je nach Schaltkreis Flash-PROM, EPROM (Erasable Programmable Read Only Memory), EEPROM (Electrically Erasable Programmable Read Only Memory), SRAM (Static Random Access Memory) Zellen verwendet. Die Ausgänge der Speicherzellen werden zur Schaltung eines Durchgangsgatters (Pass Transistor) oder eines Multiplexers verwendet. Die LUTs sind ebenfalls aus Speicherzellen aufgebaut. In der Tabelle 2.1 ist noch die irreversible Programmiertechnik Antifuse aufgezählt. Bei diesen FPGAs wird während der Konfigurierung durch das Anlegen einer Programmierspannung eine isolierende Schicht durchgeschmolzen und es entsteht eine dauerhafte Verbindung. Der für diese Arbeit verwendete FPGA ist SRAM basiert und damit reversibel programmierbar. Diese Eigenschaft ist eine Grundvoraussetzung für die dynamische Rekonfigurierung.

SRAM basierte FPGAS werden unter anderem von Altera, Atmel, Silicon Blue und Xilinx hergestellt. Der Marktführer unter diesen FPGA Herstellern ist, laut eigener Aussage, Xilinx [4], der auch eine dynamische Rekonfigurierung für seine FPGAs unterstützt [140]. Im Jahr 1984 wurde von Ross Freeman, Bernie Vonderschmitt und Jim Barnett die Firma Xilinx gegründet [148]. Ein Jahr später wurde das, bis heute zum Einsatz kommende, neue Konzept von programmierbaren Bausteinen realisiert und der erste kommerzielle FPGA XC2064^{TM}[137] vorgestellt. Heutige FPGAs entwickeln sich immer mehr zu Systemen im Schaltkreis (SoC). So sind beispielsweise in der Virtex II Pro FPGA Familie bis zu zwei, von IBM entwickelte, PowerPC 405 Prozessorkerne [139] integriert. Neben Xilinx bieten auch weitere Firmen partiell, dynamisch, rekonfigurierbare FPGAs an. Eine Gegenüberstellung der verschiedenen Hersteller kann in [8] nachgelesen werden.

2.2.1 Konfigurierbarkeitsklassen

In Abhängigkeit der Programmiertechnologien und der technischen Umsetzung der Programmierschnittstellen, können verschiedene Klassen der Konfigurierbarkeit unterschieden werden.

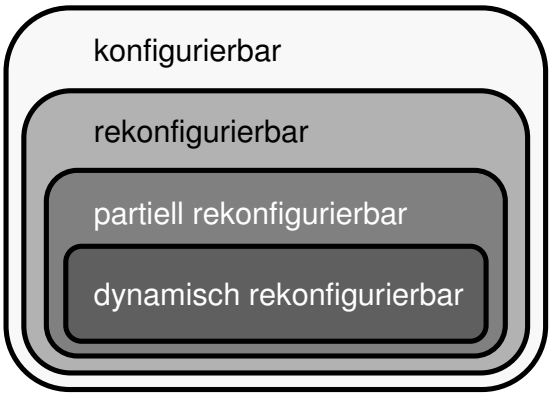


Abbildung 2.3: Klassen der Konfigurierbarkeit

In der Veröffentlichung von P. Lysaght und J. Dunlop [74] und auch im Buch von M. Wannemacher [131] werden vier Klassen unterschieden. In der Abbildung 2.3 sind diese Klassen aufgezählt und ihre Abhängigkeit untereinander grafisch dargestellt.

Konfigurierbar: Ein Kennzeichen von FPGAs, unabhängig welche Programmierertechnologie eingesetzt wird, ist gerade ihre Konfigurierbarkeit. Zu dieser Klasse gehören daher alle FPGAs, was durch das größte Rechteck in der Abbildung 2.3 aufgezeigt wird.

Rekonfigurierbar: Zu dieser Klasse gehören alle Schaltkreise, deren Konfiguration sich löschen lässt. Die Funktionalität der FPGAs kann beliebig oft neu festgelegt werden. FPGAs, die SRAM basiert sind, erfordern sogar bei jedem Neustart eine initiale Konfiguration, da beim Abschalten dieser Schaltkreise die aktuelle Konfiguration verloren geht.

Partiell rekonfigurierbar: Werden nur Teile des FPGAs gelöscht beziehungsweise neu programmiert, spricht man von partieller Rekonfigurierung. Diese Eigenschaft ist bei großen FPGAs sinnvoll, wo oft nur kleine Teile verändert werden sollen.

Dynamisch rekonfigurierbar: Wenn Hardwarefunktionalität eines FPGAs im laufenden Betrieb des Systems ausgetauscht werden soll, muss dieser dynamisch rekonfigurierbar sein. Mit der Abbildung ist diese Klasse von allen anderen eingeschlossen. Insbesondere ist jede dynamische Rekonfigurierung auch partiell, da Teile des FPGAs unverändert ihre Aktivitäten weiter ausführen und so das System am Laufen halten. Kennzeichnend für diese FPGAs ist außerdem, dass über die Programmierschnittstelle verschiedene Bereiche des FPGAs gezielt konfiguriert werden könnten.

2.2.2 Konfigurationsverfahren

Die FPGA Schaltkreise lassen sich je nach Programmiertechnologie unterschiedlich konfigurieren. Im Abschnitt 2.2 wurde schon auf die Unterscheidung, reversibel und irreversibel eingegangen. Eine weitere Einteilung kann über den Ort der Programmierbarkeit vorgenommen werden. SRAM basierte FPGAs müssen im System programmierbar (ISP) sein, da bei jeder Unterbrechung der Stromversorgung die Konfiguration des FPGA gelöscht wird. Im Gegensatz dazu sind FPGAs mit Antifuse Technologie vor dem Einbau in ein System programmierbar.

Der für diese Arbeit eingesetzte FPGA Virtex II Pro besitzt mehrere Anschlüsse zur Konfigurierung. Allgemeine Einstellungen können über die Anschlüsse mit den Bezeichnungen

M[2:0] und PROG_B vorgenommen werden. Hierbei wird M[2:0] verwendet, um die Schnittstelle festzulegen, über die, nach dem Einschalten, der FPGA initial konfiguriert werden soll. Um den FPGA in den Zustand, wie nach dem Einschalten, zurückzusetzen, steht PROG_B zur Verfügung. Über die Signale INIT_B und DONE kann zum einen festgestellt werden, ob der interne Initialisierungsvorgang und zum anderen die Konfigurierung erfolgreich abgeschlossen wurde.

Mit der JTAG (Joint Test Action Group) Schnittstelle können nicht nur Konfigurationen geschrieben, sondern auch für Tests und Fehlersuche aus dem Virtex II Pro gelesen werden. Die JTAG Schnittstelle nach dem IEEE-Standard 1149.1 wird auch mit *Boundary Scan* [10] bezeichnet. Um mehrere FPGAs über eine Programmierschnittstelle zu konfigurieren, kann der JTAG Datenausgang TDO an den Dateneingang TDI eines weiteren FPGA angeschlossen und eine JTAG-Kette aufgebaut werden.

Als weitere Schnittstellen sind die serielle und die parallele Konfigurationsschnittstelle des Xilinx FPGA zu erwähnen. Diese lassen sich jeweils im Master oder Slave-Modus betreiben. Der Master-Modus ist dadurch gekennzeichnet, dass der Takt für die Schnittstelle vom FPGA erzeugt wird. Im Slave-Modus ist der Takt von außen vorzugeben. Ähnlich wie bei der JTAG Schnittstelle können im seriellen Programmiermodus mehrere FPGAs hintereinander geschaltet und nacheinander programmiert werden. Hierzu muss ein FPGA als Master und die weiteren als Slaves betrieben werden. Wenn die Konfiguration eines FPGAs abgeschlossen ist, reicht dieser die Daten einfach an den nachfolgenden weiter. Unter Verwendung eines Mikroprozessors kann ein FPGA auch im parallelen Modus programmiert werden. Der FPGA wird dazu an den Bus des Prozessors angeschlossen.

Eine Schnittstelle die im Zusammenhang dieser Arbeit im besonderen Verwendung findet, ist der Internal Configuration Access Port (ICAP). Die ICAP Schnittstelle kann von der im FPGA konfigurierten Logik direkt angesprochen werden. Mit dieser Möglichkeit ist die Voraussetzung für eine Selbstrekonfigurierung gegeben. Technisch bildet ICAP auf die parallele Konfigurationsschnittstelle ab.

Die Daten zur Konfiguration von FPGAs werden in so genannten Bitstreams gespeichert. Diese Dateien enthalten nicht nur die Daten für die zu konfigurierenden SRAM Zellen, sondern auch Steuerbefehle für die Steuerungslogik im FPGA. Dadurch lässt sich der Konfigurationsprozess beeinflussen. Diese Eigenschaft ist eine wichtige Grundlage für die dynamische Rekonfigurierung von FPGAs, da über diesen Mechanismus unter anderem gezielt einzelne Bereiche neu beschrieben werden können. FPGAs der Firma Xilinx sind technisch so aufgebaut, dass unterschiedliche, partielle Flächen programmiert bzw. konfiguriert werden können. Alle Schaltkreise der Virtex I und II Serie sind, durch die zu Schieberegister verbunden SRAM Zellen für die Programmierung, nur spaltenweise konfigurierbar. Im Gegensatz dazu bieten die Schaltkreise der Virtex 4 und Virtex 5 Reihe mehr Freiheiten bei Form und Größe der auszutauschenden Hardwarefunktionalitäten [145, 149]. Im folgenden Abschnitt 2.3 wird auf den Herstellungsprozess der Bitstreams näher eingegangen.

2.3 Entwurfsablauf von FPGA-Schaltungen

Bei der Entwicklung von Systemen kann man drei Schritte unterscheiden [38, 131].

- Spezifizierung des Systems
- Synthese des Systems

- Validierung des Systems

Im ersten Schritt werden die geforderte Leistungsdaten, der Funktionsumfang und auch technische Rahmenbedingungen spezifiziert. Die Spezifikation ist oft noch informell, in natürlicher Sprache beschrieben. Daraus ergibt sich das Problem, dass keine automatische Validierung möglich ist. Auch mit Unvollständigkeit oder nicht eindeutigen Beschreibungen ist in der Spezifikation zu rechnen. Das Ergebnis dieses Schrittes ist eine Systemspezifikation in Form eines Pflichten- oder Lastenhefts.

Aufbauend auf der Spezifikation, kann die Synthese, der eigentliche Entwurfsablauf (engl.: Design Flow), folgen. Bei der Synthese eines Systems unter Verwendung eines FPGAs werden Konfigurationsdaten, die mittels eines Bitstreams auf den FPGA geladen werden können, erzeugt. Hierbei werden automatisiert Verhaltensbeschreibungen in Hardwarestrukturbeschreibung umgesetzt. Diese Umsetzung wird im Allgemeinen als Synthese definiert [49] und setzt sich aus verschiedenen Entwurfsschritten[7] zusammen.

Während des gesamten Design Flows sind immer wieder die Ergebnisse der Entwurfsschritte zu validieren[8], verifizieren[9] und ist das Produkt zu testen. Dieser dritte Schritt ist daher nicht nur nach Fertigstellung des Systems durchzuführen.

Im Folgenden wird die *Implementierung des Systems*, der Design Flow, für in einem FPGA zu realisierende Hardware näher betrachtet. Für die Strukturierung des Design Flows existieren mehrere Ansätze, die über alle Abstraktionsebenen (siehe Tabelle 2.2)[10] hinweg Sichten oder Beschreibungsdomänen definieren. Diese Entwurfsstrukturierungen sind primär für den ASIC Entwurf gedacht, können aber mit kleinen Einschränkungen, hauptsächlich beim Schaltkreislayout, auf programmierbare Logik übertragen werden. Eine erste, anschauliche Darstel-

Ebene	Eigenschaften
System	Betrachtung der gesamten Schaltung bestehend aus Modulen, Komponenten und Modulzellen.
Algorithmen	Nebenläufige Algorithmen bilden den Fokus, die strukturell als verbundene Blöcke aufgelistet werden.
Register-Transfer (RT)	Unterscheidung zwischen Kontroll- und Datenpfad wobei eine Menge von Operationen durch RT-Komponenten (Register, Addierer, Multiplizierer, ...) realisiert werden.
Logik / Gatter	Das Netz von Logik-Gattern und Flipflops wird über boolesche Gleichungen modelliert.
Schaltkreis / Layout	Elektronische Grundbausteine und das nicht lineare Verhalten der Schaltung werden hier beschrieben.

Tabelle 2.2: Abstraktionsebenen im Hardware Design Flow [131]

lung der Abstraktionsebenen und Sichten wurde von Gajski und Kuhn mit dem Y-Diagramm

[7] Entwurfsschritte werden in dieser Arbeit auch mit Syntheseschritten bezeichnet

[8] Validierung ist die Überprüfung, ob das Gewünschte realisiert wurde

[9] Verifizierung ist die Überprüfung oder auch der Beweis, dass das Gewünschte korrekt (ohne Fehler) realisiert wurde

[10] In manchen Darstellungen [49, 79] wird noch eine Physikalische bzw. Elektrische Ebene unterschieden. Diese Ebene sowie die Schaltkreisebene sind nur für den Entwurf von FPGAs von Interesse und nicht für die im FPGA zu konfigurierende Funktionalität.

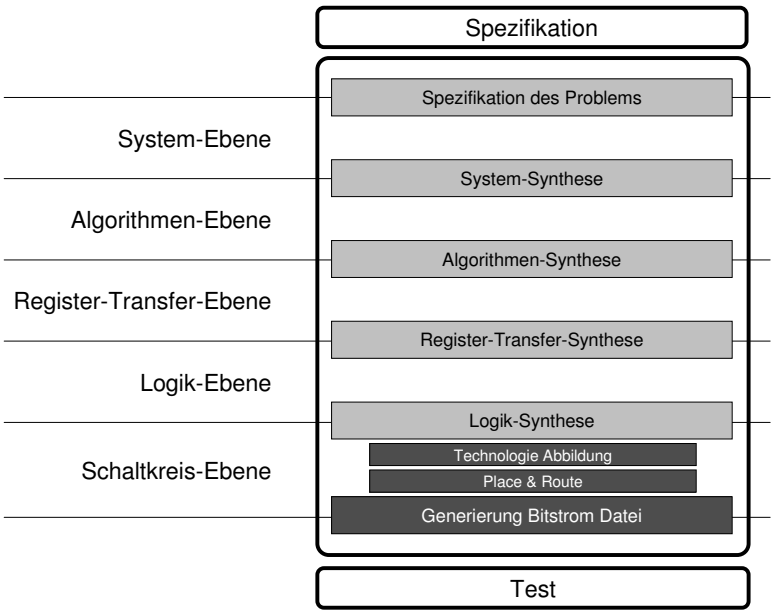


Abbildung 2.4: Entwurfsschritte für FPGA basierten Hardwareentwurf

veröffentlicht [37]. Sie unterscheiden die Verhaltens-, Struktur- und Geometrie-Sicht. Im Y-Diagramm entspricht der Wechsel vom Verhaltensmodell zur Struktur einem Syntheseschritt und in der umgekehrten Richtung einer Analyse. Elemente einer Abstraktionsebene setzen sich zusammen aus Komponenten der nächst niedrigeren Ebene. Das Modell wird daher, bei einem Top-Down Ansatz, immer weiter verfeinert bzw. implementiert. Aus der Struktur heraus lassen sich dann die Elemente der Geometrie-Sicht generieren. Diese Entwurfsstrukturierung wurde von Rammig um die Test-Sicht erweitert [95]. Unter Berücksichtigung, das Hardware nur mit Software funktioniert, entwickelte Teich die Doppel-Dach-Strukturierung, die Verhalten und Struktur für Hardware und Software berücksichtigt [106]. In [49] überträgt Hardt die Strukturierung von Entwurfssichten, Entwurfsschritten und Entwurfsebenen auf die verschiedenen Schwerpunkte bei der Entwicklung von eingebetteten Systemen und führt für die Schwerpunkte den Begriff Dimension ein.

Beim Top Down Entwurf wird, aufgrund der Komplexität der zu entwickelnden Hardwarefunktionalität, bei höheren Abstraktionsebenen, mit einer Hardwarebeschreibungssprache (HDL), begonnen zu implementieren. Im Bereich der programmierbaren Logik wird meist VHDL (Very High Speed Integrated Circuit Hardware Description Language) eingesetzt. Mit VHDL wird das Verhalten des Systems, unabhängig von der Zieltechnologie, parametrisierbar beschrieben. Die Abbildung der Funktionalität in eine Beschreibungssprache ermöglicht eine frühzeitige Simulation des Systems. VHDL wird nicht nur bei der Entwurfseingabe (engl.: Design Entry) sondern auch über verschiedene Abstraktionsebenen hinweg eingesetzt.

In Abbildung 2.4 sind die Syntheseschritte, beginnend mit der Entwurfseingabe (*Spezifikation des Problems*), aufgezeigt. Aus den Anforderungen der Spezifikation wird, z.B. mit Hilfe eines Datenflussgraphen, einem Problemgraphen [106], die benötigten funktionalen Aufgaben auf der Systemebene spezifiziert.

In einem ersten Verfeinerungsschritt (*Systemsynthese*) wird die Beschreibung der Systemebene in eine Verhaltensbeschreibung der Algorithmenebene abgebildet und das System in Untersysteme partitioniert. Diese Untersysteme werden dann meist vom Entwickler durch den Einsatz von fertigen Hardwareblöcken (siehe Abschnitt 2.5) oder durch Programmierung der Systemkomponenten realisiert. Synthesewerkzeuge setzen erst auf der Ebene der Algorithmen an. Auf höheren Abstraktionsebenen sind die Werkzeuge auf eine Assistenz des Entwicklers beschränkt und dienen lediglich der schnellen Bewertung von Systemvarianten. Sie werden auf der Systemebene zur Erkundung des Entwurfsraums (engl.: Design Space Exploration) eingesetzt.

Die nachfolgende *Algorithmensynthese* führt zu einer Beschreibung des Entwurfs aus Elementen der RT Ebene. In diesem Schritt werde erste Optimierungen durch die Synthesewerkzeuge vorgenommen. Die Hauptaufgaben der Algorithmensynthese sind die Festlegung der Art und Anzahl der benötigten Komponenten (engl.: Allocation), die Zeitplanung für die Operationen (engl.: Scheduling) und die Zuweisung der Komponenten auf vorhandene Ressourcen (engl.: Assignment).

In der *Register Transfer Synthese* werden die Daten- und Kontrollpfade der Algorithmenebene in entsprechende Boolesche Funktionen, Register und Verbindungsleitung transformiert. Die Kodierung der Automatenzustände und die Berechnung der Funktionen, die zur Ansteuerung von Registern benötigt werden, findet in diesem Schritt statt.

Mit der *Logiksynthese* werden letzte technologieunabhängige Abbildungen und Optimierungen durchgeführt. Hierzu werden das Flattening[11] und auch das Structuring[12] durchgeführt. Die dabei, auf Gatterebene, entstehende Netzliste wird nachfolgend auf die zur Verfügung stehende FPGA Technologie abgebildet. Bevor nun, der zur Konfigurierung eines FPGAs benötige, Bitstream erzeugt werden kann, muss noch das *Place* und *Route* durchgeführt werden, damit die einzelnen Funktionen im FPGA platziert und miteinander verbunden werden.

2.3.1 Synthese partiell rekonfigurierbarer Systeme

Bei der Synthese von dynamisch rekonfigurierbaren, eingebetteten Systemen sind zusätzliche Entwurfsschritte erforderlich, um die zeitliche Veränderung der Systemfunktionalität im FPGA zu ermöglichen. Für jedes auszutauschende Modul und der statischen Logik[13] sind die in Abschnitt 2.3 vorgestellten Design Flow Schritte durchzuführen. Zusätzlich sind die Platzierungen der Module vom Entwickler anzugeben und das System mit allen Modulen in einer ganzheitlichen Sicht zu synthetisieren. Diese Schritte haben Ähnlichkeit mit Design Projekten, wo mehrere Entwickler verschiedene Teile des Systems entwerfen. Xilinx hat diese Analogie in [140] ausgenutzt und ihren *Modular Design Flow* für dynamisch rekonfigurierbare Systeme adaptiert. Eine ausführliche Beschreibung der Xilinx Entwurfsansätze ist im *Development System Reference Guide* [142] zu finden. Ein wichtiger Unterschied zum Standard *Modular Design Flow* ist die Integration von festen Kommunikationskanälen[14] zwischen den Modulen und der statischen Logik. Der *Modular Design Flow* für *Module-Based Partial Reconfiguration* setzt sich aus folgenden drei Schritten zusammen [143].

- **Initial Budgeting:** In diesem Schritt werden Randbedingungen (engl.: Constraints) und

[11] Entfernung der Zwischenvariablen in Booleschen Ausdrücke und Auflösung von Klammern
[12] Ausklammerung von gemeinsamen Faktoren und Minimierung von Produktthermen
[13] Unter statischer Logik wird in diesem Zusammenhang im FPGA konfigurierte Logik bezeichnet, die über alle Rekonfigurierungen hinweg sich nicht ändert.
[14] Xilinx nennt diese, als Hard Makro realisierten Kommunikationskanäle, Bus Makro

der FPGA Belegungsplan für das Gesamtsystem festgelegt. Das Ergebnis bildet eine UCF Datei (User Constraint File) mit allen Platzierungs- und Zeitvorgaben. Voraussetzungen für diesen Schritt sind unter anderem, die Festlegung der Rekonfigurierungsflächen, eine Netzliste im Xilinx Dateiformat NGC des Top-Levels[15] mit den Instanzen der Module als Black-Boxes und die Platzierung der Bus Makros.

- **Implementing Active Modules:** Nach Fertigstellung des Initial Budgeting werden alle statischen und dynamischen Module des Systems separat in Abhängigkeit von der Top-Level Netzliste synthetisiert und die partiellen Bitstreams generiert. Dieser Schritt kann für alle Module parallel ausgeführt werden.
- **Finale Assembling:** Abschließend folgt aus der Top-Level Netzliste und einem oder mehreren synthetisierten Modulen die Generierung einer initialen, ausführbaren Konfiguration.

Wie in der Erläuterung zum *Initial Budgeting* Schritt erwähnt, müssen mehrere Designentscheidungen festgelegt werden, um den Modular Design Flow durchzuführen zu können. Diese teilen sich in drei Bereiche ein [16].

1. Partitionierung der Systemfunktionalität
   - Festlegung der globalen, statischen Logik
   - Bestimmung der dynamisch austauschbaren Module
2. Abbildung der Module auf den FPGA
   - Festlegung der Modulflächenanzahl
   - Platzierung der Rekonfigurierungsflächen im FPGA
3. Platzierung der Kommunikationskanäle
   - zwischen statischer und austauschbarer Logik
   - zwischen Peripherieanschlüssen und FPGA Logik
   - zwischen rekonfigurierbaren Modulen

Zusätzlich zu den Voraussetzungen für die Syntheseschritte ist auch eine Steuerung des dynamischen Verhaltens im laufenden Betrieb notwendig [87, 52]. Diese Steuerung der Rekonfigurierungen ist in Abhängigkeit der partitionierten Systemfunktionalität und den partitionierten FPGA Ressourcen zu konzipieren. Aus den unterschiedlichen Realisierungsvarianten, in Software oder als Hardwareblock, ergeben sich auch unterschiedliche Anpassungen des Systemdesigns. Zum einen muss eine Schnittstelle für die dynamische Rekonfigurierung (siehe Abschnitt 2.2.2) entweder von Software oder Hardware ansteuerbar sein. Diese Schnittstelle ist zusätzlich, zu der eigentlichen Systemfunktionalität und deren Schnittstellen, im Design zu berücksichtigen. Zum anderen benötigt, bei der Selbstrekonfigurierung, die Hardwarelösung eine Zugriffsmöglichkeit auf die im Speicher liegenden Module und weitere Logik zur Ablaufsteuerung der Rekonfigurierungen. Ein selbstrekonfigurierbares, eingebettetes System bedarf daher der Integration einer Rekonfigurierungssteuerung (engl.: Reconfiguration Control Unit / RCU), eines RCU Moduls.

[15] Diese Beschreibung wird auch als Basis Entwurf (engl.: Base Design) bezeichnet

2.3.2 Entwurfsschritte partiell selbstrekonfigurierbarer Systeme

Die allgemeinen Entwurfsschritte für ein im FPGA zu realisierendes System und die Berücksichtigung der zusätzlichen Syntheseschritte für die Modularisierung eines partiell dynamisch rekonfigurierbaren Systems führen zu dem dieser Arbeit zu Grunde liegenden Gesamt-Design-Flow [85, 86, 144]. In Abbildung 2.5 sind die Einzelschritte ersichtlich.

Da dynamische Rekonfigurierung im Kontext dieser Arbeit keine Systemanforderung ist, sondern eine Technologie zur Umsetzung der Spezifikation, unterscheidet sich diese nicht von anderen Spezifikationen für in FPGAs zu realisierende Systeme. Wenn die *Spezifikation* vorliegt kann mit der Implementierung des Systems begonnen werden. Im Schritt *Spezifikation des Problems* wird das Zusammenspiel von verschiedenen Funktionalitäten in einem Problemgraph oder auch einer HDL beschrieben und als Design Entry bereitgestellt. Die nachfolgende *System-Synthese* dient zur Partitionierung eines Systems in Untersysteme und zur Bestimmung der von den Untersystemen auszuführenden Algorithmen. Für die Realisierung der Untersysteme können entweder fertige Hardwareblöcke (siehe Abschnitt 2.5) Einsatz finden oder der Entwickler implementiert diese von Hand.

Die Einteilung des Systems in Untersysteme beinhaltet nicht die Aufteilung in statische und dynamische Module. Diese Aufteilung wird erst im folgenden Schritt *Partitionierung* durchgeführt. Prinzipiell kann in jeder Abstraktionsebene das System in austauschbare Module eingeteilt werden. In Kapitel 4 wird dieser Entwurfsschritt im Detail vorgestellt. Als nächstes werden die rekonfigurierbaren Module auf die FPGA Ressourcen abgebildet. Das Ergebnis dieses Schrittes ist die Voraussetzung für den Initial Budgeting Schritt, wo Größe und Lage der Modulflächen in der UCF Datei eingetragen werden. Detaillierte Informationen über diese *Platzierung* sind in Kapitel 5 aufgezeigt. Aus der Platzierung der Module heraus lässt sich die Lage der Kommunikationskanäle bestimmen und es stehen alle Informationen bereit, um eine Rekonfigurierungssteuerung (siehe Kapitel 6) zu entwickeln. Das Top-Level Design, in welchem die dynamischen und statischen Module sowie die RCU als Komponenten integriert sind, ist vor den *Module-Based Partial Reconfiguration* Schritten (vergleiche dick umrandete Kästchen in Abbildung 2.5) zu generieren.

Der *Module-Based Partial Reconfiguration* Design Flow umfasst das *Initial Budgeting*, zur Erzeugung der UCF Datei, die *Dynamic* und *Static Module Implementation*, um die Modulbitstreams zu generieren, sowie den abschließenden Schritt dem *Final Assemble*, zur Erstellung eines initialen Systems. Grau markiert wurden in der Abbildung 2.5 die allgemeinen Entwurfsschritte, die unabhängig vom Einsatz der Rekonfigurierung durchgeführt werden müssen. Die gestrichelten Entwurfsschritte kennzeichnen Schritte, die nur durchgeführt werden, wenn für ein Modul kein fertiger Hardwareblock, in Form einer Netzliste, wiederverwendet wird. Alle technologieabhängigen Syntheseschritte sind in dunkelgrauen Kästen mit weißer Schrift dargestellt.

2.4 Rekonfigurierungskonzepte

Um die technische Möglichkeit der dynamischen Rekonfigurierung von Hardwarefunktionalität effizient in ein System zu integrieren, existieren verschiedene Konzepte. Grundlegend helfen diese Konzepte die folgenden drei Fragen zu beantworten.

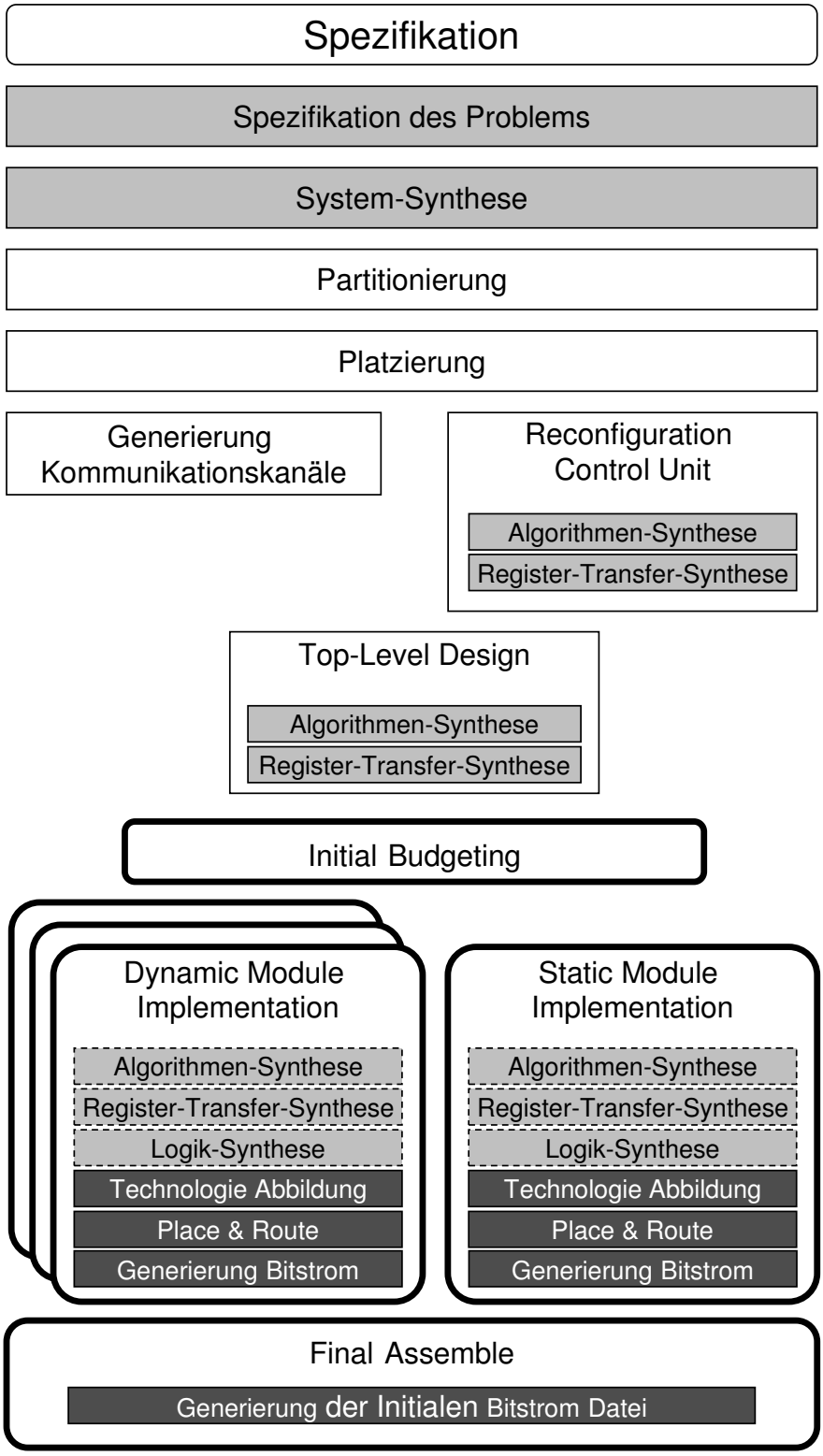


Abbildung 2.5: Design Flow für dynamisch rekonfigurierbare eingebettete Systeme

- Was soll rekonfiguriert werden?
- Wo im FPGA soll etwas rekonfiguriert werden?
- Wann findet die Rekonfigurierung statt?

Die erste Frage zielt auf die Bestimmung der Teile eines Systems hin, die durch Rekonfigurierung ausgetauscht werden sollen. Die Antwort auf diese Frage führt zu einer Partitionierung des Systems in einzelne Module. Neben den Funktionalitäten des Systems sind die Ressourcen, die von dem System genutzt werden, zu verwalten. Das Problem der Platzierung von Funktionalitäten auf der Hardwareressource wird von der zweiten Frage adressiert. Der dritte Aspekt betrifft die Zeitpunkte für Rekonfigurierungen. Als Ergebnis der zweiten und dritten Frage entsteht eine so genannte Zeitablaufsteuerung (eng. Scheduling).

Die hier vorgestellten Fragen lassen sich auch verallgemeinern und auf andere Schwerpunkte der Informatik anwenden. In diesem Zusammenhang ist die Analogie zur Speicherverwaltung interessant. Bei der Speicherverwaltung werden Programme in mehrere Module aufgeteilt, die nicht zur gleichen Zeit im Speicher vorhanden sein müssen. Die zur Verfügung stehende Ressource ist der Speicher, der für bestimmte Zeiträume von Programmmodulen genutzt werden kann. Aus dieser gegebenen Analogie heraus, ist es offensichtlich, dass bewährte Speicherverwaltungskonzepte für die dynamische Rekonfigurierung von Hardwarekomponenten genutzt werden sollten. Einschränkend können nicht alle Konzepte der Speicherverwaltung Anwendung finden, da bei Rekonfigurierung die Kommunikation zwischen Modulen berücksichtigt werden muss.

Das Konzept der **direkten Adressierung** findet in der Speicherverwaltung bei einfachen universellen Betriebssystemen Anwendung [6]. Meist wird der gesamte Speicher in einem Systembereich für das Betriebssystem und einem Bereich, in welchem das Programm einen zusammenhängenden Teil belegt, aufgeteilt. Eine besondere Eigenschaft dieses Konzepts ist, dass alle Adressen der geladenen Programme bereits im Objektcode als absolute Adresse vorliegen. Daher wird dieses Konzept nur bei einfachen, überschaubaren Systemen eingesetzt. Bei der Übertragung der direkten Adressierung auf die Rekonfigurierung ist nachteilig, dass der Systementwickler sich weitestgehend selbst um die Adressierung und damit die Partitionierung und Platzierung kümmern muss.

Ein weiteres Konzept das aus der Speicherverwaltung als Rekonfigurierungskonzept genutzt werden kann, ist die **Relocation**. Wenn Programme nur zusammenhängende Speicherbereiche nutzen dürfen und diese nicht direkt hintereinander angeordnet sind, kann es zu einer ungleichmäßigen Auslastung des Speichers kommen. Durch die Verschiebung von Programmen im Speicher kann eine bessere Ausnutzung der Ressourcen bewerkstelligt werden. Dadurch, dass FPGAs wie auch Arbeitsspeicher sehr regelmäßige Strukturen aufweisen, ist dieses Konzept gut auf Rekonfigurierung übertragbar. Diese Relocation wurde unter anderem von der DFG im Programm SPP1148 *Reconfigurable Computing Priority Program* [102] fokussiert und auch die Veröffentlichungen [22, 61, 60] basieren auf diesem Konzept. Dabei ist dieses Konzept in Verbindung mit einem On Chip Bus *RecoNet* realisiert. Bei dieser Methode wird ebenfalls nicht die Partitionierung betrachtet. Während man bei der Speicherverwaltung von einer eindimensionalen Verschiebung der Programme ausgehen kann, sind es bei den Hardwaremodulen zweidimensionale Flächen, die eine Hardwarefunktionalität realisieren und die verschoben werden sollen. Diese Verschiebung kann technisch realisiert werden, durch eine direkte Änderung der Positionsdaten im fertig synthetisierten Bitstream. Die Form der Hardwaremodule lässt sich

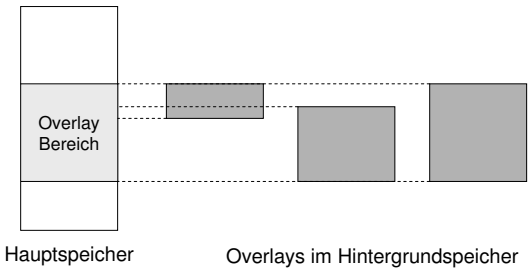


Abbildung 2.6: Overlay im Arbeitsspeicher

jedoch nur durch eine erneute Synthese ändern und auch die benötigten Kommunikationskanäle schränken die Flexibilität der Verschiebung ein.

Die **Overlay Technik** wurde entwickelt, um Programme ausführen zu können, die nicht komplett in den verfügbaren Speicher geladen werden können. Dieses Konzept basiert darauf, dass nicht alle Teile des Programms im Speicher vorgehalten werden müssen, sondern dynamisch nachgeladen werden, sobald sie benötigt werden. Für das Nachladen ist im Speicher ein Bereich festgelegt, der von den Programmsegmenten genutzt werden kann (siehe Abbildung 2.6). Befindet sich schon ein Segment in diesem Speicherbereich, wird es zwangsläufig durch neue Programmteile überschrieben (überlagert). In der Programmiersprache Pascal wurde diese Technik unterstützt [136]. Die Segmentierung eines Programms in verschiedene Teile, welche nicht gleichzeitig im Speicher vorgehalten werden müssen, ist von dem Programmierer zu bewerkstelligen, weil es keine praktikable Möglichkeit für den Compiler und den Linker gibt dies automatisch zu tun. Das Nachladen der einzelnen Segmente und auch die Festlegung des Overlay Bereichs wird von der Laufzeitumgebung gesteuert. Aus dem Overlaying Konzept heraus ergibt sich eine grundlegende Erkenntnis für den Einsatz von Rekonfigurierung in eingebetteten Systemen. Das Überlagern von mehreren Programmsegmenten ist nur sinnvoll, wenn nicht genügend Arbeitsspeicher vorhanden ist. Ebenso sollte Rekonfigurierung nur eingesetzt werden, wenn die gesamte Systemfunktionalität nicht auf dem FPGA platziert werden kann und das System nicht alle Hardwarefunktionalitäten gleichzeitig benötigt. Da diese Voraussetzungen bei vielen eingebetteten Systemen gegeben sind, wird in dieser Arbeit für das Rekonfigurierungskonzept die Overlay Technik adaptiert. Die Anpassung beinhaltet die Erweiterung des Overlaying auf mehrere Überlagerungsbereiche und die Berücksichtigung der Kommunikationsschnittstellen. Durch das Overlaying werden folgende drei Punkte, in Anlehnung an die oben genannten Fragen, festgelegt.

- Die zu rekonfigurierenden Module werden vom Entwickler definiert.
- Zur Platzierung wird der FPGA in vordefinierte Bereiche eingeteilt. Die Zuordnung der Module zu diesen Bereichen kann vor der Synthese statisch durchgeführt werden.
- Die Bestimmung der Rekonfigurierungszeitpunkte ist Datenfluss abhängig und wird zur Laufzeit vom System festgelegt.

Die Verwendung dieses Overlay Konzepts bewirkt eine zeitliche Trennung für die Festlegung des Scheduling in eine statische Platzierung (Ressourcenzuweisung) und in eine dynamische Zeitplanung (Schedul). Statisch bedeutet in diesem Zusammenhang, dass die Ressourcenzuwei-

sung beim Entwurf des Systems festgelegt wird. Der Zeitpunkt, wann diese Ressourcen genutzt wird, wird dagegen zur Laufzeit des Systems durch die zu verarbeitenden Daten bestimmt.

Im Gegensatz zu Programmsegmenten die im Arbeitsspeicher abgelegt werden, existieren zwischen Hardwaremodulen Kommunikationsschnittstellen. Durch diese Schnittstellen werden die einzelnen Module miteinander verbunden. Ein besonderes Merkmal dieser Kommunikationsschnittstellen ist, dass sie rekonfigurierbare und auch nicht rekonfigurierbare Module miteinander verbinden und daher unabhängig von der eigentlichen Hardware Rekonfigurierung sind. Daher erfordert die Problematik der Kommunikation in dynamisch rekonfigurierbaren Systemen, eine gesonderte Betrachtung. Die in diesem Bereich existierenden Forschungsarbeiten basieren auf den folgenden drei Konzepten.

- Eine Möglichkeit besteht darin, allen Modulen eine einheitliche, in bestimmten Fällen auch universelle Schnittstelle zur Verfügung zu stellen. Dieses Konzept ist analog zu den Mechanismen von *Plug and Play*. Bei USB (Universal Serial Bus) werden zum Beispiel einheitliche Stecker und Buchsen und ein einheitliches Kommunikationsprotokoll verwendet um verschiedene Hardware anzuschließen. Übertragen auf die dynamische Rekonfigurierung bedeutet dies, dass es eine einheitliche Schnittstelle für alle Module gibt. Die Kommunikation zwischen den Modulen ist dann auf einem Bus mit entsprechenden Protokoll abzubilden. Als Beispiele für Bussysteme die in dynamisch rekonfigurierbaren Systemen Einsatz finden, sei der DyNoC (Dynamic Network on Chip) [14, 77] und CuNoC (scalable Communication Unit based Network on Chip) [59] erwähnt. Durch die Homogenität der Schnittstelle kann solch ein Bus in verschiedenen Systemen ohne größere Anpassungen eingesetzt werden. Nachteilig ist jedoch, dass alle Module über diesen Bus kommunizieren und damit ein einheitliches Protokoll bereitstellen müssen. Im Falle einer Wiederverwendung von Hardwarefunktionalitäten in unterschiedlichen Systemen, kann eine Adaptation des Protokolls und der Schnittstelle dieses Moduls notwendig werden.

- Neben der homogenen Schnittstelle für rekonfigurierbare Module, können bei der Integration von Rekonfigurierung in ein System auch heterogene Schnittstellen zum Einsatz kommen. Dies erfordert jedoch Mehraufwand bei der Konzeption und Umsetzung der Kommunikationsverbindungen. Die verwendeten IPs können im Gegensatz dazu ohne Adaptierung eingesetzt werden, was sich in vielen Fällen positiv für die Effizienz des Systems auswirkt. Unter der Berücksichtigung, dass diese Verbindungen automatisch hergestellt werden können und dass Rekonfigurierung meist in kleine eingebettete Systeme integriert werden soll, wird dieses Konzept für die weiteren Ausführungen zu Grunde gelegt.

- Bezüglich der automatischen Erstellung von Kommunikationsverbindungen ist auch das Konzept der Interface Synthese zu erwähnen. Um nicht die einzelnen Hardwaremodule auf ein spezielles Protokoll anzupassen, kann ein Hardwareblock zwischen einem Modul und dem Kommunikationsbus zur Protokollkonvertierung eingesetzt werden. Stefan Ihmor hat in seiner Dissertation eine automatische Synthese solch eines Hardwareblocks, dem IFB (Interface Block), beschrieben [51]. Neben der eigentlichen Protokollkonvertierung ist hier im Besonderen die dynamische Anpassung auf neue Protokolltypen [57, 58, 87] von Interesse. Die Möglichkeit der Protokolladaptation ist auch bei der Kommunikation über Systemgrenzen hinweg, wenn z.B. simulierbare und in Hardwa-

re vorliegende Komponenten für Funktionstest miteinander kommunizieren sollen, von Vorteil [36].

2.5 Hardwaremodule

Mit immer größer werdenden Integrationsdichten von Schaltkreisen, lassen sich auch immer komplexere eingebettete Systeme realisieren. Der Nachteil dieser Entwicklung ist, dass durch das rasante Tempo der Entwicklung der Fertigungstechnologie die Designfähigkeit dieser Systeme nur mit verstärktem Aufwand gewährleistet werden kann. Die Problematik der Entwurfslücke (engl.: Design Gap) resultiert aus der langsameren Entwicklung von Entwurfsmethoden gegenüber den Fertigungstechnologien. Sind eingebettete Systeme nur mit sehr großem Zeit- und Personalaufwand entwickelbar, ist verstärkt mit Fehlern zu rechnen und die Systeme können nicht profitabel auf den Markt gebracht werden.

Um dem Design Gap entgegenzuwirken, ist die Wiederverwendung von verifizierten Hardwarebausteinen naheliegend. Prinzipiell bieten verschiedene Hardwarebeschreibungssprachen, wie VHDL, die Möglichkeit solche Bausteine (engl.: Building Block), Module oder Komponenten zu beschreiben. Im Gegensatz zu einem Building Block bezeichnet eine IP eine vorgefertigte und verifizierte Beschreibung eines Entwurfs, die als Baustein in den eigenen Entwurf mit möglichst wenig Zeitaufwand integriert werden kann. Diese Elemente können, wie in der

IP Typ	Eigenschaft
Soft-IP	synthedisierbare HDL
Firm-IP	HDL und Constraints
Hard-IP	verdrahtete, platzierte Netzliste

Tabelle 2.3: IP-Core Typen

Tabelle 2.3 aufgeführt, in unterschiedlichen Beschreibungsarten vorliegen. Soft-IPs lassen sich leicht in unterschiedlichen Chiptechnologien realisieren, da sie als synthetisierbares HDL vorliegen. Wenn die Soft-IPs mit Vorgaben (engl.: Constraints) für eine optimale Implementierung auf einer FPGA-Familie ausgeliefert werden, handelt es sich um Firm-IPs. Bei der dritten Art sind die IPs optimal, für eine FPGA Familie, komplett verdrahtet und platziert. Hard-IPs liegen in Form einer Netzliste vor. IPs werden von Halbleiterherstellern, EDA-Anbietern und auch von unabhängigen Anbietern angeboten, oder existieren firmeninternen (engl.: Design Re-Use). Auch wenn die Zahl der verfügbaren IP-Cores steigt, existieren Probleme bei der Suche, Archivierung, Weitergabe und Vermarktung dieser Bausteine, sowie beim Entwurf und der Integration.

- Für eine frühzeitige Einbindung von IP-Modulen, oder auch deren Spezifikation, werden Spezifikationsmethoden notwendig, die Wiederverwendungen berücksichtigen und unterstützen.

- Aus der Historie der einzelnen IP-Anbieter und IP-Käufer heraus ergeben sich Differenzen und Inkompatibilitäten bei den, für die Entwicklung, verwendeten Methoden.

- Die Suche von auf dem Markt verfügbaren IPs wird durch eine zeitaufwändige und oft auch unvollständige Informationsbeschaffung behindert. Um IPs von verschiedenen Herstellern in ein System zu integrieren, ist eine umfassende Dokumentation unumgänglich.
- Die Schwierigkeit der Bewertung von IPs wird deutlich, wenn garantiert werden muss, ob die sich aus dem Gesamtsystem und dessen Entwurfsmethodik ergebenden Anforderungen erfüllt sind.
- Eine Integration von IPs ist mit einem hohen Adaptionsaufwand verbunden, wenn die spezifizierten Anforderungen nicht vollständig gegeben sind.

Um diesen Problemen begegnen zu können, wurde das Projekt IPQ von verschiedenen IP-Herstellern, IP-Kunden und mehreren Universitäten durchgeführt. In diesem Projekt wurde ein Format (IPQ Format) zur Beschreibung von IP Komponenten sowie ein dazu passender Werkzeugsatz [117, 118, 123], mit aktuellen Webservice Technologien, entwickelt. Das Format beschreibt die IP Charakterisierung [100], beinhaltet den IP Content und ein IP Service Format [119] mit verschiedenen Taxonomien [121]. Aufbauend auf diesem IPQ Format lassen sich, über die entwickelten Werkzeuge [120], IP-Suchen spezifizieren [125, 122], Kaufprozesse einschließlich der Übermittlung abwickeln und die Archivierung von Hardwarebausteinen für eine spätere Wiederverwendung [124] realisieren. Um den modularen eingebetteten System Entwurf mit IPs zu unterstützen wird in dieser Arbeit auf das IPQ Format aufgesetzt. Das Format ist in einem XML (Extensible Markup Language) Schema beschrieben und kann daher leicht in andere Systeme integriert werden [126]. Um die Integration von IP auch bei dynamisch rekonfigurierbaren Systemen zu unterstützen, kommt bei der Umsetzung der in dieser Arbeit entwickelten Methoden XML zum Einsatz. Grundlegende Informationen zu XML können unter anderem in [42, 43, 9] nachgeschlagen werden.

2.6 Zusammenfassung

In diesem Kapitel wurden verschiedene Grundlagen vermittelt, auf denen die im Rahmen dieser Arbeit entwickelten Konzepte aufbauen. Im ersten Abschnitt dieses Kapitels wurden Eigenschaften der eingebetteten Systemen vorgestellt und in Bezug auf dynamische Rekonfigurierung näher beleuchtet. FPGAs als Grundvoraussetzung für die dynamische Rekonfigurierung wurden im zweiten Teil dieses Kapitels dargestellt. Neben dem Aufbau eines FPGAs wurden auch die Konfigurierbarkeitsklassen und mögliche Verfahren, um diese Schaltkreise zu konfigurieren, vorgestellt. Aufbauend auf die technischen Gegebenheiten eines FPGAs sind im Abschnitt 2.3 die Entwurfsschritte für eine FPGA Implementierung beschrieben worden. Speziell wurde auf die Voraussetzungen zur Erzeugung eines dynamisch rekonfigurierbaren Systems und die daraus resultierenden zusätzlichen Entwurfsschritte eingegangen. Der dieser Arbeit zu Grunde liegende Design Flow wurde im Unterabschnitt 2.3.2 vorgestellt. In 2.4 wurde auf verschiedene Rekonfigurierungskonzepte eingegangen und das Overlaying Konzept, das in dieser Arbeit Anwendung findet, vorgestellt. Die Möglichkeiten und die Problematik bei der Integration von Hardwaremodulen in ein eingebettetes System wurden in Abschnitt 2.5 präsentiert. Es wurde auf das Projekt IPQ, den darin entwickelten Methoden und das mit XML beschriebene IPQ Format eingegangen.

3 Stand der Technik

Der im Abschnitt 2.3.2 vorgestellte Design Flow zeigt mehrere Schritte auf, die als allgemeine Problemstellung im Zusammenhang mit dynamischer Rekonfigurierung aufgefasst werden können.

Im folgenden Abschnitt 3.1 wird auf die Grundvoraussetzung, der Partitionierung des Systems, für dynamische Rekonfigurierung eingegangen. Hier werden im Besonderen Techniken aus dem Bereich der verteilten Systeme vorgestellt und die Möglichkeiten der funktionalen Partitionierung aufgezeigt. Mit dem Problem der temporalen Abbildung von Modulen auf eine FPGA Fläche wird sich im Abschnitt 3.2 beschäftigt und bestehende Arbeiten zu dynamische und statische Platzierungen vorgestellt. Dass eine dynamische Rekonfigurierung einer Steuerung bedarf, wird im Abschnitt 3.3 anhand von speziellen Hardwareplattformen, gezeigt. Bestehende Konzepte für eine RCU und Implementierungsbeispiele werden hier näher beleuchtet. Ein viertes Problem, im Abschnitt 3.4, betrifft die Bereitstellung der Kommunikationsverbindungen, die als Bus Makros zu realisieren sind. Hier werden technische Möglichkeiten für die Integration eines Netzwerkes in dynamisch rekonfigurierbare Systeme aufgezeigt und auf die automatisierte Generierung von individuellen Bus Marko Verbindungen eingegangen.

3.1 Partitionierung von Systemen

Eine Entwurfspartitionierung ist nicht erst für den Einsatz von dynamischer Rekonfigurierung notwendig geworden. Schon im Bereich der Multi-FPGA Systeme finden strukturelle Systemaufteilungen Anwendung. Die Partitionierungsverfahren können in strukturelle und funktionale Ansätze eingeteilt werden.

Bei der strukturellen Partitionierung werden Methoden der Graphentheorie eingesetzt, um Netzlisten zu unterteilen. Dies ist möglich, da Netzlisten nach der Synthese homogene Strukturen aufweisen. Die Homogenität wird durch die Optimierung während der Synthese erreicht und bedeutet, dass alle hierarchischen Ebenen und funktionalen Zusammenhänge weitestgehend aufgelöst sind. Für Einzelheiten der Theorie und der Formalisierung des Problems der Graphenpartitionierung sei auf die Veröffentlichungen [17, 90] hingewiesen. Je nach Optimierungsziel für eine Systemaufteilung ergeben sich sehr große Lösungsräume[1], die nur mit geeigneten Heuristiken durchsucht werden können. Für den Einsatz von dynamischer Rekonfigurierung bei Hardware-Emulatoren wurden mehrere dieser Heuristiken in [8] vorgestellt und angewandt. Dort wird eine Einteilung der Heuristiken in folgende Klassen nach [50] unterschieden: *Group-Migration*, *metrische Allokationsmethoden* und *Simulated Annealing*. Die entwickelte Methodik in [8] fokussiert hierbei auf Methoden der *Group-Migration*. Als bekanntesten Vertreter dieser Klasse ist die Bi-Partitionierung von Kernighan und Lin zu nennen [62]. Eine verbesserte Technik mit gewichteter Partitionierung, um einer ungünstigen Auslastung von Multi-FPGA Syste-

[1] Problem der Lösbarkeit: Graphenpartitionierung liegt in NP (Non Deterministic Polynomial-Time Hart) [41]

men, die bei der ursprünglichen Bi-Partitionierung auftreten kann, entgegenzuwirken, wurde in [35] vorgestellt und durch das Zellreplikationsverfahren [56] weiterentwickelt. Ein genaueres Partitionierungsergebnis hinsichtlich gegebener Randbedingungen erreichen *metrische Allokationsmethoden* und auch *Simulated Annealing* Verfahren. Auf struktureller Ebene wurden solche Verfahren in [15, 16] angewandt. Aufbauend auf den ASAP/ALAP-List Scheduling Ansatz [92, 94, 19] zur Lösung des Partitionierungsproblems eines gegebenen Datenflussgraph, wird in dieser Arbeit ein erweiterter Ansatz des Verfahrens vorgestellt. Mit diesem erweitertem Verfahren konnte die Anzahl der physikalischen Konfigurationen gegenüber dem ursprünglichen Ansatz halbiert werden. Hierbei wurde eine gemeinsame Benutzung von Komponenten (Configuration Switching), bei aufeinander folgenden Partitionen, ausgenutzt. Des Weiteren wurde in [15] eine so genannte *Wire Length Model* vorgestellt, die auf Ebene des Datenflussgraphen, die Kommunikation zwischen den Graphknoten berücksichtigt. Ziel dieser Methode ist es, die Kommunikation zwischen verschiedenen Partitionen zu minimieren, indem man stark verbundene Komponenten des Graphen in eine Partition zusammenfasst. Hierzu wurde eine dreidimensionale Spektralplatzierung verwendet. Spektralmethoden wurden in der Vergangenheit öfter für Partitionierung und Platzierung eingesetzt [2, 3, 54, 20].

Eine strukturelle Partitionierung einer Netzliste erscheint für einen IP basierten Entwurf jedoch nicht sinnvoll zu sein, da gegebene Funktionsblöcke, um eine Netzliste zu erhalten, synthetisiert und hernach mit Methoden der Graphentheorie neu partitioniert werden müssten. Daher wird in der vorliegenden Arbeit auf eine funktionale Partitionierung gesetzt.

Die funktionale Partitionierung wird, im Gegensatz zur Strukturellen, nicht auf Gatter oder Netzlistenebene, sondern auf funktionaler beziehungsweise auf Systemebene durchgeführt [38]. In [69, 70] wird diese Partitionierung auch als Architekturpartitionierung bezeichnet. Auf dieser Ebene sind die für die Kostenfunktionen benötigten Designparameter, wie zum Beispiel die Anzahl der benötigten Gatter, noch nicht bekannt und sind daher mit geeigneten Methoden zu schätzen. Der Vorteil dieses Ansatzes ist, die Berücksichtigung funktionaler Zusammenhänge des Systems [115]. Dadurch werden Module generiert, die weitestgehend funktional unabhängig sind.

Einen direkten Vergleich zwischen struktureller und funktionaler Partitionierung ist in [112] zu finden. In dieser Veröffentlichung wird auch auf das Anwendungsfeld der funktionalen Partitionierung, dem Hardware/Software CoDesign, eingegangen. Frühere Forschungen fokussierten hierbei im Besonderen auf eine, bezüglich der Implementierungskosten und der Systemperformance, optimale Aufteilung der Systemkomponenten in Hardware oder Software [114, 110, 29]. Die Existenz einer funktionalen Beschreibung eines Systems zeigt, dass eine Automatisierung möglich ist, was beispielsweise in [70, 47, 67, 113] erforscht wurde. Auch in [8] wird, neben der strukturellen, ein Vorgehen für eine automatisierte, funktionale Partitionierung im Kontext von FPGA-basierten Hardware Emulatoren vorgestellt.

- Ein Ansatz zur Automatisierung basiert hierbei auf dem von Mentor Graphics favorisierten Entwurfsablauf für die Integrierung von IPs in FPGA Entwürfe [25]. Bei diesem werden Informationen zum hierarchischen Aufbau des Systems für die Partitionierung in Module aus Synthesewerkzeugen, wie z.B. dem HDL-Designer, extrahiert.

- Des weiteren werden Entwurfsmuster für reguläre Architekturen berücksichtigt. Über generische Beschreibungen können dadurch mehrfach instanziierte Module selektiert und zusammengefasst werden, so dass auf dem Emulator nur eine Instanz realisiert werden muss.

- Als drittes Verfahren wird in [8] die Spezifikation um Informationen bezüglich der Entwurfsarchitektur erweitert. Aus diesen Metadaten werden vom Partitionierungsverfahren definierte Muster für die Emulation extrahiert. Diese Vorgehensweise wurde angelehnt an ähnliche Verfahren im Bereich des IP-Entwurfs [53].

Der Nachteil dieser funktionalen Partitionierungsverfahren ist, dass zeitkritische Unterbrechung, hervorgerufen durch Rekonfigurierung des Emulators, immer noch auftreten können und über eine zusätzliche Steuerung abgefangen werden müssen. Ziel der Partitionierung in [8] sind möglichst parameteräquivalente Module, weshalb auch strukturelle und funktionale Methoden kombiniert werden. Im Kontext von eingebetteten Systemen ist jedoch die Vermeidung von kritischen Systemunterbrechungen durch dynamische Rekonfigurierung zu fokussieren. Dies bedeutet, dass eine funktionale Beschreibung nicht auf Ebene von arithmetischen und logischen Ausdrücken, wie bei dem Yorktown Silicon Compiler in [21] oder auch dem Bottom-Up Design in [82], partitioniert wird. Für eingebettete Systeme müssen daher komplexere Systemaufgaben als Funktionen angesehen werden.

3.2 Platzierung in dynamisch rekonfigurierbaren Systemen

Dynamisch rekonfigurierbare Systeme unterscheiden sich von herkömmlichen, durch eine zusätzlich zu berücksichtigende Dimension, der Zeit. Jedes Modul, das durch die Partitionierung des Systems bestimmt wurde, ist aus diesem Grund nicht nur auf der FPGA Fläche zu platzieren, sondern ist auf ein Zeitfenster abzubilden. In der Literatur wird daher auch der Begriff *Temporale Platzierung* verwendet [48, 1, 34].

Die Platzierung von Modulen eines rekonfigurierbaren Systems kann entweder statisch, für alle Module, oder dynamisch, während das System läuft, durchgeführt werden. Im Kontext von eingebetteten Systemen, bei welchen die funktionalen Anforderungen im Vorhinein festliegen und sich selten zur Laufzeit ändern, ist eine dynamische Platzierung eher nicht geeignet. Weitergehende Informationen zu Verfahren der dynamischen Platzierung können unter anderem in [16, 48, 72, 129, 64] nachgelesen werden.

Im Bereich der statischen Platzierung sind als einfachste Methoden die First-Fit und Best-Fit Ansätze zu nennen [7, 15, 16]. Das schnelle First-Fit Verfahren wählt die erstbeste freie Position im FPGA und platziert das aktuell betrachtete Modul auf diese Fläche. Der Nachteil gegenüber dem Best-Fit Ansatz ist, dass große freie Flächen mit kleinen Modulen belegt werden können und dadurch FPGA Ressourcen ungenutzt bleiben. Bei der Best-Fit Platzierung wird für ein zu platzierendes Modul die kleinste, ausreichende FPGA Fläche gesucht. Betrachtet man jedoch eine komplette Konfiguration, kann auch hier eine ineffiziente Belegung generiert werden. Aus diesem Grund wurde, unter Verwendung von Integer Linear Programming, ein *Packing* Ansatz entwickelt [107, 34]. Ziel dieses Ansatzes ist es, die Module unter Berücksichtigung eines Optimierungsziels, in den dreidimensionalen Ressourcenraum, FPGA Fläche und Zeit, zu packen. Mit einem geeigneten Optimierungsziel kann beispielsweise der kleinste Schaltkreis, auf welchen ein System funktionstüchtig abgebildet werden kann, bestimmt werden.

Kennzeichnend für diese statischen Platzierungsansätze ist, dass die Abarbeitungsreihenfolge der einzelnen Konfigurationen bekannt sein muss. In vielen Fällen müssen eingebettete Systeme jedoch auf Menschen oder andere Umwelteinflüsse reagieren können, wodurch eine feste Konfigurationsreihenfolge nicht garantiert werden kann. Aus diesem Grund wurde im Rahmen dieser

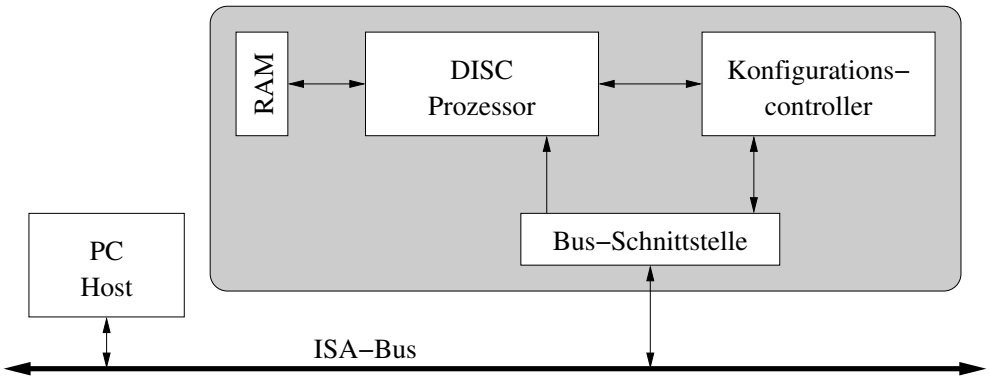


Abbildung 3.1: Struktur der DISC-Erweiterungskarte [134]

Arbeit ein Verfahren entwickelt, das für beliebige Konfigurationsreihenfolgen geeignet ist und die Idee des Best-Fit Ansatz ausnutzt, um passende Überlappungsbereiche zu bestimmen.

3.3 Steuerung von Rekonfigurierung

Unabhängig von den funktionalen Aufgaben eines rekonfigurierbaren Systems ist für die Konfigurationswechsel eine Steuerung notwendig. Die Realisierung der Steuerung ist stark von der Verwendungsart der konfigurierbaren Schaltkreise abhängig. Die Schaltkreise können beispielsweise als Coprozessor in ein System integriert werden oder auch einen zentralen Prozessor ersetzen, indem dessen Aufgaben im FPGA ausgeführt werden (siehe [74]). Bei selbständigen Systemen sind auch die Steueraufgaben von und im FPGA zu realisieren. Im Folgenden werden mehrere Systeme und deren RCU vorgestellt.

Dynamic Instruction Set Computer

Der in [134] vorgestellte *Dynamic Instruction Set Computer* (DISC) ist ein Prozessor, dessen Befehlssatz sich zur Laufzeit erweitern lässt. Als PC-Erweiterungskarte umfasst dieses System zwei FPGAs und einen Speicher für die auszuführenden Programme (siehe Abbildung 3.1). Ein FPGA wird für das Steuerwerk und der Abarbeitung des Programms verwendet. Der andere FPGA steuert die Rekonfigurierung und fordert die Bitstreams für fehlende Befehlseinheiten vom PC an.

Zu den Aufgaben einer RCU gehören hauptsächlich die Verwaltung der FPGA-Ressourcen, das Auslösen von Rekonfigurierungen und das Anfordern von benötigten Bitstreams. Im DISC wird Initial nur der auszuführende Programmcode in den Speicher geladen und das Steuerwerk mit häufig benötigten Standardinstruktionen in den Konfigurationscontroller FPGA konfiguriert. Trifft das Steuerwerk bei der Abarbeitung des Programms auf einen unbekannten Befehl, wird dieser beim PC angefordert und löst dadurch eine Rekonfigurierung aus. Vom PC wird daraufhin anhand der aktuellen Belegung des DISC Prozessor FPGAs eine Position für das Befehlsmodul bestimmt und der Bitstream geliefert. Sollten im FPGA keine Ressourcen mehr frei sein, werden die am seltensten verwendeten Befehlsmodule entfernt. Die Verwaltung der FPGA-Ressourcen und auch die Organisation der Bitstreams werden nicht in der DISC-Hardware realisiert, son-

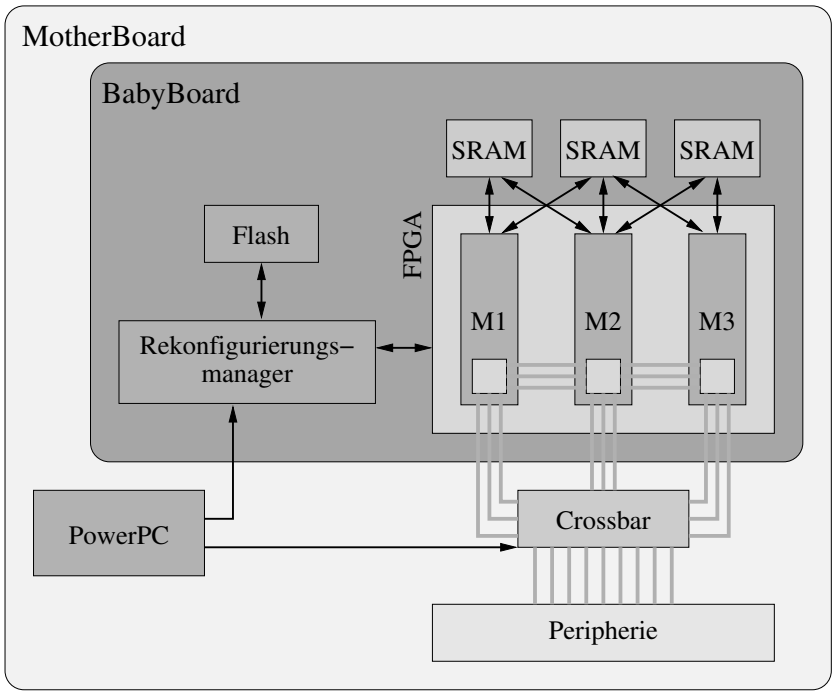


Abbildung 3.2: Architektur der Erlangen Slot Machine [78]

dern sind Aufgaben des Hostsystems. Dieser Softwareansatz im DISC ist für eingebettete Systeme ungeeignet, da bei diesen die zusätzlichen Systemkomponenten nicht immer zur Verfügung stehen.

Erlangen Slot Machine

Einen ähnlichen Aufbau wie der DISC ist auch bei der *Erlangen Slot Machine* (ESM) zu finden [78]. Die ESM ist ein dynamisch rekonfigurierbares System basierend auf FPGAs der Firma Xilinx und teilt sich in ein „BabyBoard“ und ein „MotherBoard“ ein. Das MotherBoard dient der Ansteuerung von Peripherieschnittstellen, wie IEEE1394, USB, Ethernet, PCMCIA sowie Video und Audio Ein- und Ausgänge. Es ist vergleichbar mit dem Hostsystem des DISC, da es ebenfalls die Verwaltung der FPGA-Ressourcen und der Bitstreams übernimmt.

Der schematische Aufbau der ESM ist in Abbildung 3.2 zu sehen. Auf dem BabyBoard befinden sich ein dynamisch rekonfigurierbarer FPGA mit zusätzlich angeschlossenem SRAM, ein Rekonfigurierungsmanager, bestehend aus einem weiteren FPGA und einem CPLD[2], und ein Flashspeicher zum Vorhalten der Bitstreams. Der dynamisch rekonfigurierbare FPGA, ein Virtex II, ist in eine feste Anzahl vertikaler Rekonfigurierungsbereiche (Slots) aufgeteilt. Module lassen sich relativ einfach in verschiedene Slots laden, da die Slots gleichartig aufgebaut sind. Im Rekonfigurierungsmanager ist das eigentliche Laden und Entladen der Module in Hardware implementiert [76] und besteht aus einem PicoBlaze Microcontroller, an den verschiedene Plug-Ins angeschlossen werden können. Die Plugins ermöglichen es, verschiedene Manipulationen an den Bitstreamdaten vorzunehmen, bevor diese an die SelectMAP-Schnittstelle des FPGA

[2]Der CPLD initialisiert nach dem Einschalten den FPGA im Rekonfigurierungsmanager.

übergeben werden.

Zur Rekonfigurierung ist die ESM auf den Prozessor auf dem MotherBoard angewiesen, da hier die Verwaltung der Ressourcen erfolgt und auch das Auslösen der Rekonfigurierungen. Damit ist diese Struktur für Systeme, die keinen separaten Prozessor besitzen, ebenfalls weniger gut geeignet. Auch die feste Aufteilung des FPGA kann sich für Systeme mit vielen kleinen Modulen negativ auswirken, da hierdurch eventuell die Ausnutzung nicht optimal ist.

Dynamic Circuit Switching

Der Dynamic Circuit Switching Ansatz in [73] ist geeignet, um das Verhalten dynamisch rekonfigurierbarer Systeme zu simulieren. Bei diesem Ansatz werden die Verbindungen zwischen einzelnen Schaltungsteilen, die eine bestimmte Aufgabe erledigen, gesteuert. In [75] wird das Dynamic Circuit Switching weiter ausgebaut. Schwerpunkt hier ist es, den zusätzlichen Aufwand für Rekonfigurierung, durch Ersetzen der fest vorgegebenen Steuerung mit einer automatisch Generierten, zu minimieren. Eine automatische Erzeugung hätte den Vorteil, dass sie zu dem jeweiligen Anwendungsfall genau passt, was mit einer allgemeinen Lösung schwer zu realisieren wäre. Es müssen natürlich von dem Entwickler die entsprechenden Informationen für die Generierung bereitgestellt werden. Eine automatische Generierung der RCU ist auch für eingebettete Systeme zu bevorzugen.

In der Veröffentlichung [83] wird ein mit diesem Ansatz modelliertes System vorgestellt, welches die Rekonfigurierung selbst steuert. Des Weiteren ist eine allgemeine Steuereinheit, welchem das Dynamic Circuit Switching zu Grunde liegt, in [97] beschrieben worden.

Weitere Steuerkonzepte

Viele dynamisch rekonfigurierbare Systeme werden per Software gesteuert und sind auf einen Anwendungsfall spezialisiert. In [18] wurden deshalb drei Anwendungsgebiete für Rekonfigurierung untersucht und Anforderungen für eine allgemeine Steuerung extrahiert. Die Autoren stellen ein erweiterbares Laufzeitsystem, RAGE, zur Verwaltung von dynamisch rekonfigurierbaren FPGAs vor, dass auf diesen Anforderungen aufbaut.

Des Weiteren wird in [105] eine Software gestützte Steuerung für die Konfigurationswechsel beschrieben. Mit diesem Ansatz wird eine Programmierschnittstelle bereitgestellt, über die der Entwickler auf Hardwarefunktionalität zugreifen kann, die durch Rekonfigurierung auf einem FPGA dynamisch bereitgestellt wird. Der vorgestellte Prototyp wurde für ein FPGA der Familie Xilinx Virtex II Pro entwickelt. Dadurch war es möglich, wie auch in [13] vorgestellt, die Steuerung in Software auf dem integrierten PowerPC zu realisieren.

Im Gegensatz zu diesen Ansätzen wurde in [32] ein generisch aufgebauter *Reconfigurable System Configuration Manager* vorgeschlagen. Diese RCU ist besonders für eingebettete Systeme interessant, da sie komplett in Hardware realisierbar ist. Der vorgeschlagene Aufbau der Steuerung besteht aus sechs verschiedenen Modulen (siehe Abbildung 3.3).

- *Configuration Memory*: In diesem Speicher sind alle Bitstreams hinterlegt.
- *Self-Configuration*: Mit diesem Modul wird der Rekonfigurierungsprozess gesteuert.
- *Configuration Interface*: Die FPGA Programmierschnittstelle wird hiermit angesprochen.

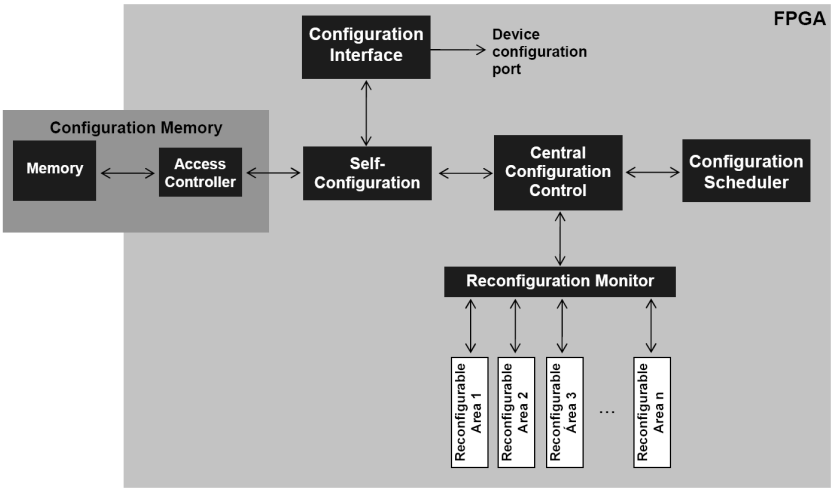


Abbildung 3.3: Aufbau des *Reconfigurable System Configuration Manager* [32]

- *Reconfiguration Monitor*: Aufgabe des Monitors ist es, Situationen zu ermitteln, in welchen eine Rekonfigurierung durchgeführt werden muss.
- *Central Configuration Control*: Diese zentrale Einheit ist für die Verwaltung aller Steuerabläufe verantwortlich.
- *Configuration Scheduler*: Verwaltet die verschiedenen Konfigurationen des Systems und bestimmt die zu ladenden Module.

Im Kapitel 6 wird die im Rahmen dieser Arbeit entwickelte RCU im Detail beschrieben. Mehrere der in [32] beschriebenen Module finden sich, aufgrund von analogen, technischen Gegebenheiten, auch in diesem Ansatz wieder. Durch das gewählte Rekonfigurierungskonzept für eingebettete Systeme ergeben sich jedoch auch Unterschiede, die sich im Besonderen in den Modulen, die vom jeweiligen System abhängig sind, wie die *Self-Configuration*, *Configuration Scheduler* und *Reconfiguration Monitor*, widerspiegeln.

3.4 Kommunikation in rekonfigurierbaren Systemen

Ein grundlegender Unterschied zur überlappenden Speicherverwaltung in Software, ist die Kommunikation zwischen den Overlaybereichen in Hardware. Auch unabhängig von einem bestimmten Rekonfigurierungskonzept oder auch bei nur einem Rekonfigurierungsbereich, müssen die Daten zu den Modulen übertragen und empfangen werden können. Erst durch die technische Möglichkeit, diese Kommunikation fest im FPGA und unabhängig von den Konfigurationswechseln bereitzustellen [140], ermöglichte es die dynamische Rekonfigurierung einzusetzen.

Wie im Abschnitt 2.4 aufgezeigt, können zwei Ansätze bei der Bus Makro Kommunikation unterschieden werden. Die eine Variante beinhaltet ein On Chip Bussystem (NoC - Network on Chip), das alle Module über eine homogene Schnittstelle anbindet. Dabei kann einfach ein

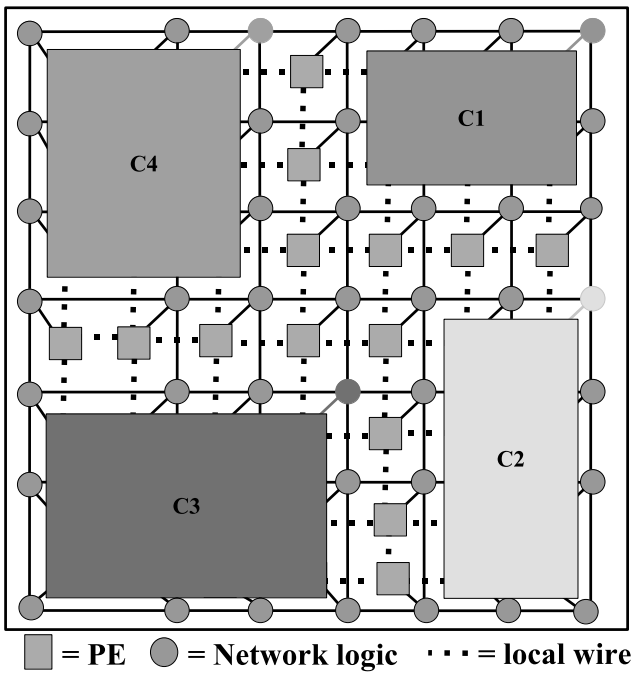


Abbildung 3.4: Aufbau des DyNoC [14]

festes Bus Makro für viele Systeme vorgegeben werden, über welches die Module kommunizieren. Alle Module müssen jedoch das Busprotokoll realisieren. Beispiele für solche NoC Architekturen sind unter anderen: SPIN [46], CLICHE [68], Torus [24] und BFT [91].

Mit einem festen Makro werden jedoch auch die Rekonfigurierungsflächen bezüglich Lage und Größe eingeschränkt. Ein Ansatz, um diesen Beschränkungen bei den Austauschflächen zu begegnen, ist in [14, 77] zu finden. In diesen Veröffentlichungen wird nicht nur die Architektur des *Dynamic Network on Chip* (DyNoC) vorgestellt, sondern auch eine Erweiterung des XY-Routings, um den Nachteilen des Netzwerkes zu begegnen. Das Kommunikationsnetzwerk besteht aus einem gleichmäßig über die FPGA Fläche verteiltem Netz von Kommunikationsknoten, an die jeweils ein Modul angeschlossen werden kann. Um die Fläche für die Module nicht an dieses Netzwerk zu binden, können Netzknoten entfernt und die frei gewordenen Ressourcen für das entsprechende Modul genutzt werden. In Abbildung 3.4 ist schematisch das DyNoC mit vier angeschlossenen Module dargestellt. Der größte Nachteil des DyNoC ist die Dimension der verschiedenen Kommunikationsknoten, mit welchen das Routing koordiniert wird. Aus diesem Grund wurde ein neuer Ansatz in [59] vorgestellt, mit welchem die Kommunikationsverbindungen zwischen den Modulen dynamisch auf dem Schaltkreis platziert werden können. Hierfür werden unabhängige Router eingesetzt, die so genannten Communication Units. Im Kontext von eingebetteten Systemen, ist jedoch ein Ansatz mit dynamisch anpassbaren und oftmals aufwändigen Bussystemen nicht immer zielführend. In [65] wird neben der Kommunikationsinfrastruktur der Schwerpunkt auf eine automatisierte, Werkzeug unterstützte Generierung gelegt. Dadurch können Netzwerke, aufbauend auf eine so genannte *Pin Virtualisation* und unabhängig von einer speziellen Hardware, in verschiedenen Systemen Anwendung finden.

Der zweite Ansatz für eine Bus Makro Kommunikation fokussiert auf einer individuellen heterogenen Struktur für jedes rekonfigurierbare System. Die Idee hierbei ist es, Module in verschiedenen eingebetteten Systemen zu verwenden, ohne die Schnittstelle auf ein neues Bussystem anpassen zu müssen. Dadurch können in den Systemen verschiedene Kommunikationsverbindungen, wie Busse und Punkt zu Punkt Verbindungen, auftreten. Die Kommunikationsart wird hierbei von den Modulen bestimmt. Der Vorteil dieses Ansatzes besteht im Wegfall der Routingknoten im Bus Makro. Es müssen lediglich die Verbindungsleitungen bereitgestellt werden und alle andere Kommunikationslogik ist in den Modulen zu implementieren.

Um individuelle Kommunikationsleitungen als Bus Makro zu erzeugen, kann der Xilinx FPGA Editor [140] genutzt werden. Aufgrund der Scriptfähigkeit des Werkzeugs ist eine automatisierte Erstellung möglich. Die Scriptsteuerung erwies sich, bei Test im Rahmen dieser Arbeit, jedoch als instabil. Ebenso passte Xilinx mit den Versionswechsel auch die Scriptsprache des Editors immer wieder an. Eine weitere Möglichkeit automatisch die Bus Makros zu generieren, kann über das Werkzeug *xdl* von Xilinx realisiert werden. Mit Hilfe der *Xilinx Description Language* (XDL) ist es möglich, die proprietären Binärformate *Native Circuit Description* und *Native Macro Circuit* in eine Textbeschreibung zu konvertieren. Da auch die umgekehrte Transformation erlaubt ist, ist dies eine Variante, eigene Bus Makros zu beschreiben. Der Entwurfsprozess ist dadurch gegenüber dem Entwickler transparent, da die Erstellung rein textuell erfolgt und weitestgehend versionsunabhängig ist. In [111] sind weiterhin Anleitungen und Tests zur automatisierten Generierung von Bus Makros untersucht worden. Die Tests zeigen, dass eine automatische Generierung möglich ist und für eingebettete Systeme angewandt werden kann. In Abschnitt 5.4 wird auf das in dieser Arbeit eingesetzte Verfahren zur Bus Makro Erzeugung kurz eingegangen.

3.5 Zusammenfassung

Die dynamische Rekonfigurierung ist Thema von vielen Forschungsarbeiten, was sich in den in diesem Kapitel zitierten Veröffentlichungen widerspiegelt. Bezüglich des in 2.3.2 vorgestellten Design Flows, sind hier Arbeiten zu den drei Problembereichen, Systempartitionierung, Modulplatzierung und Steuerung von Rekonfigurierungen, vorgestellt worden. Mit der Darstellung verschiedener Netzwerkimplementierungen auf einem Chip und der Generierungsmöglichkeiten von Bus Makros wurde dieses Kapitel abgeschlossen.

Für die Unterteilung eines Systems in Module wurden strukturelle Platzierungsmethoden vorgestellt, die auf niedrigeren Abstraktionsebenen, wie zum Beispiel der Gatterebene, Anwendung finden. Eine weitere Gruppe von Algorithmen bilden die funktionalen Partitionierungsansätze, die auch im Kontext dieser Arbeit Anwendung finden.

Die Methoden zur Platzierung der Module auf dem FPGA, können in dynamische und statische Ansätze unterteilt werden. Im Bereich von eingebetteten Systemen ist davon auszugehen, dass zur Laufzeit nur selten neue Module platziert werden müssen, so dass eine Berechnung der Lage der Module im FPGA zur Entwurfszeit automatisiert durchgeführt werden kann.

In den vorgestellten Arbeiten zur Steuerung der Rekonfigurierungen wurden Implementierungen in Software und Hardware unterschieden. Die Softwareansätze sind durch eine hohe Flexibilität gekennzeichnet, benötigen aber in den meisten Fällen zusätzliche Hardware, die in eingebetteten Systemen nicht immer zur Verfügung steht. In Hardware realisierte Steuerungen werden daher im weiteren fokussiert. Diese zeigen auch keine Nachteile, bezüglich einer automatisierten Generierung.

4 Partitionierung

Nach Vorstellung des Design Flows für dynamisch rekonfigurierbare, eingebettete Systeme und der notwendigen Voraussetzungen für die Synthesewerkzeuge im Kapitel 2.3, folgt in diesem Kapitel die Darstellung eines Konzepts und dessen Umsetzung zur automatisierten Partitionierung eines Systems für dynamische Rekonfigurierung.

Für die Partitionierung eines Systems wird eine Systembeschreibung benötigt. Diese Beschreibung und deren Erstellung wird im Abschnitt *Ausgangssituation* vorgestellt. Zur besseren Veranschaulichung des in diesem Abschnitt generierten Problem- und Architekturgraphen wird ein Beispielsystem eingeführt. Im Abschnitt 4.2 wird auf das Problem der Partitionierung eingegangen und ein Lösungsansatz, sowie eine automatisierte Modulbildung vorgestellt. Dieses Kapitel wird abgeschlossen durch eine Betrachtung von Robustheitsaspekten und einer Zusammenfassung.

4.1 Ausgangssituation

Aus einer gegebenen Spezifikationen heraus ist es die Aufgabe des Entwicklers zuerst die Funktionalitäten zu bestimmen[1], die das System realisieren. Es sind daher alle Anwendungsfälle (engl.: Use Cases), auf ihre Anforderungen hin, zu untersuchen. Diese Untersuchung führt zu einer Spezifikation von Funktionalitäten in Form eines Problemgraphen, die entweder implementiert werden müssen oder von früheren Projekten wieder verwendet werden können. Wie im Kapitel 2.5 aufgezeigt ist auch der Erwerb der Hardwaremodule bei IP Anbietern in Betracht zu ziehen. Der Problemgraph PG (Formel (4.1)) ist ein Datenflussgraph, bestehend aus einer Menge von Knoten $SE = \{se_1, ..., se_n\}$, den Systemelementen, und einer Menge von gerichteten Kanten $KV = SE \times SE$, die die Kommunikationsverbindungen repräsentieren[2].

$$PG = (SE, KV) \tag{4.1}$$

$$kv = (se_i, se_j, \alpha) \in KV : se_i \neq se_j \tag{4.2}$$

Ein Systemelement umfasst eine Spezifikation einer zu realisierenden Funktionalität. In diesem ersten Entwurfsschritt werden die von Teich [106] eingeführten Kommunikationsknoten nicht mit berücksichtigt, da die Struktur des Systems mit seinen Funktionen und nicht die technische Realisierung im Vordergrund steht.

> Als Beispiel für einen Problemgraphen soll ein Musik-Player betrachtet werden. Die Spezifikation dieses Players umfasst die Anforderung, verschieden kodierte Musikdateien und digitales Radio zu unterstützen. Folgende vier Formate seien spezifiziert:

[1] Diese Bestimmung der Funktionalitäten wurde im *Design Flow für dynamisch rekonfigurierbare eingebettete Systeme* (siehe Abbildung 2.5) mit *Spezifikation des Problems* bezeichnet

[2] α ist die Kardinalität einer Kommunikationsverbindung, die spätestens im unten definierten Architekturgraphen festgelegt wird

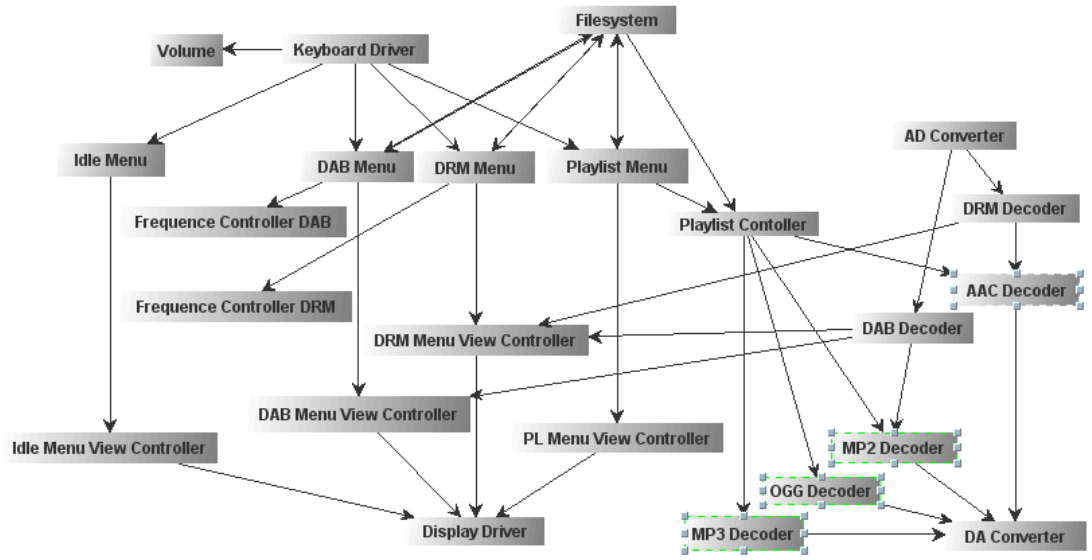


Abbildung 4.1: Problemgraph für einen Musik-Player mit Mp3 und digitalem Radio

- MP3 (MPEG 1, Audio Layer 3)
- Ogg Vorbis
- AAC (Advanced Audio Coding)
- MP2 (MPEG 1, Audio Layer 2)

Für das digitale Radio soll das auf MPEG 1 Layer 2 basierende DAB (Digital Audio Broadcasting) und DRM (Digital compressed Radio Mondiale) unterstützt werden. Bei DRM werden die Audiodaten mit dem ACC Codec komprimiert. Der resultierende Problemgraph ist in Abbildung 4.1 dargestellt und beinhaltet die vier, markierten Dekodierungsverfahren als Kernfunktionalitäten.[3] An den vier verschiedenen Systemelementen für die Menüs wird deutlich, dass der Problemgraph anhand der spezifizierten Anwendungsfälle erstellt wurde. Es gibt daher für den Fall, dass der Player im Ruhezustand ist das Menü *Idle Menu*. Analog sind für die beiden Radiovarianten und für das Abspielen von Musikdateien Menü-Systemelemente im Problemgraph enthalten. Das Systemelement *File System* wird zum Laden der Musikdateien und zum Lesen und Schreiben von Konfigurationsdateien der einzelnen Kodierungsverfahren verwendet. Nicht alle Elemente des Problemgraphen sind auf dem FPGA abzubilden. Systemelemente die auf externer Hardware realisiert werden müssen, sind für dieses Beispiel der *DA* und *AD Converter*. Um die Übersichtlichkeit des Problemgraphen zu erhalten, wurden nicht alle benötigten Systemelemente für das digitale Radio in den Graphen aufgenommen. Man kann jedoch davon ausgehen, dass unter einem Systemelement verschiedene Teilelemente zusammengefasst werden und somit ein hierarchischer Ansatz bei der Beschreibung des Systems vorliegt.

[3] Im Kontext dieser Arbeit stehen die Qualität und der Nutzen des Entwurfs nicht im Vordergrund.

Ist der Problemgraph PG erstellt, folgt die Abbildung der benötigten Funktionalitäten auf den Architekturgraphen AG. Hierzu sind, für die spezifizierten Systemelemente $se_i \in SE$, geeignete IPs zu suchen oder zu entwickeln. Der Unterschied zwischen einem Systemelement und einer IP besteht darin, dass neben den Metadaten, der Charakterisierung der Hardwarefunktionalität, auch der IP Content, eine konkrete VHDL Beschreibung, vorliegt. Existieren für die Systemelemente $se_i \in SE$ die IPs $ip_i \in IP = \{ip_1, ..., ip_n\}$ ist auch die Kardinalität α der Kommunikationsverbindungen $kv_i \in KV = \{kv_1, ..., kv_m\}$ festgelegt (siehe Gleichung 4.2). Aus dieser Definition heraus ergibt sich, dass sich der Architekturgraph AG, bis auf Morphologie, nicht von dem des Problemgraphen PG unterscheidet. Für die spätere Erstellung des Top Level Designs und der darin enthaltenen Portmaps, werden die Kanten des Architekturgraphen AG entsprechend der Hardwarebeschreibungen, mit Interfaces und Bitbreiten, genauer unterschieden.

Über ein an die Eigenschaften von IPs angepasstes, fallbasierten Schließen (engl.: case-based reasoning) [99, 132, 66], werden in der *System-Synthese* zu den spezifizierten SEs geeignete IPs gesucht. Aus der Menge der gefundenen IPs für ein Systemelement wählt der Entwickler das am besten Passende aus. Die IP-Suche und die Erstellung der Systembeschreibung sind unter anderem in [127] beschrieben. In dieser Veröffentlichung werden die Erstellung des Problemgraphen und die *System-Synthese* zusammengefasst als Systemkomposition bezeichnet. Das Ergebnis der *System-Synthese* ist eine in XML vorliegende Systembeschreibung, dem *System Format*. Sie enthält sowohl die Beschreibung der Suchanfragen, als auch die IP Charakterisierungen der vom Entwickler ausgewählten IPs. Ein Ausschnitt des XSD Schemas des IPQ Formats, welches sowohl für die IP Charakterisierung, als auch zur Beschreibung der Systemelemente genutzt wird, ist in Abbildung 4.2 angegeben. Bei der Beschreibung der Systemelemente, werden nur die für die Suche relevanten Eigenschaften einer Hardwarefunktionalität im IPQ Format spezifiziert. Das *System Format* beinhaltet neben den IP Beschreibungen weitere Systemeigenschaften (siehe in Abbildung 4.3 die Elemente Board, Chip, ...), die bei der IP Suche berücksichtigt werden können. Der Problemgraph wird im XML Schema über die beiden Elemente *System-Elements* und *Connections* im Teilbaum *AbstractDefinition*[4] abgebildet. Analog dazu ist der Architekturgraph AG mit seinen *IPs* unter dem Element *ConcreteDefinition*[5] zu finden.

Die Kommunikationsverbindungen der Graphen sind in den Unterbäumen *.../AbstractDefinition/Connections* und *.../ConcreteDefinition/Connections* des *System Formats* aufgelistet. Diese beiden Graphen bilden die Grundlage für den nächsten Entwurfsschritt, die *Partitionierung*.

4.2 Konfigurationskonzept und Modulbildung

Die Partitionierung des Systems in Module, die je nach Bedarf statisch sind, nachgeladen oder überschrieben werden sollen, bildet den ersten für Rekonfigurierung notwendigen Entwurfsschritt. Aus den unterschiedlichen Möglichkeiten ein System in Module zu unterteilen, können sich zusätzliche Aufwände für die Platzierung der Module auf dem FPGA, der Steuereinheit für die Rekonfigurierung und der Bereitstellung von festen Kommunikationsverbindungen für die Synthese ergeben. Um den Entwickler weitestgehend zu entlasten, sind die Möglichkeiten für

[4] Hier wurde nicht der Begriff *ProblemGraph* verwendet, da zusätzliche, abstrakte Definitionen diesem Element zugeordnet sind.

[5] Hier wurde ebenfalls nicht der Begriff *ArchitekturGraph* verwendet, da zusätzliche, konkrete Festlegungen diesem Element zugeordnet sind.

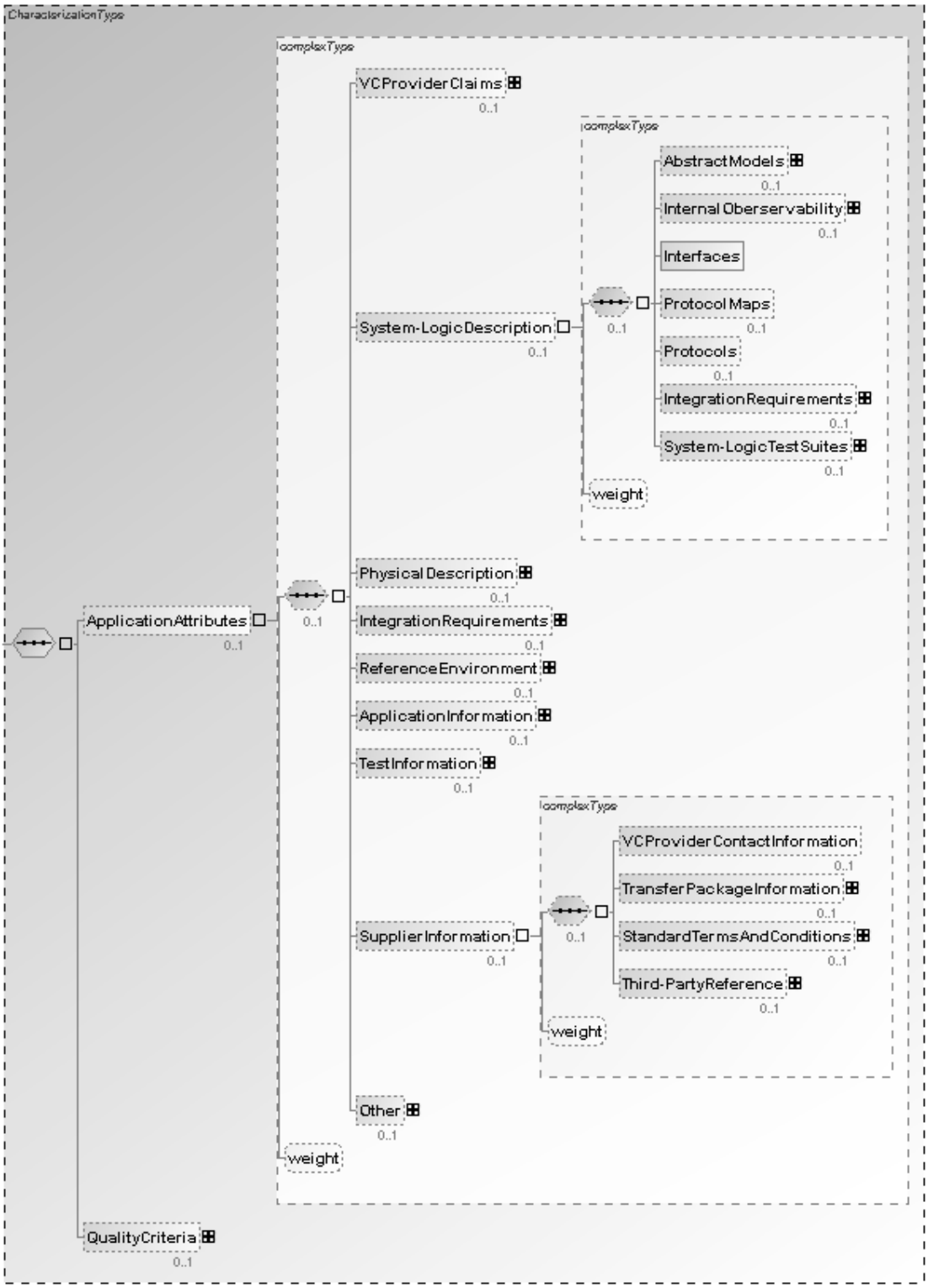


Abbildung 4.2: Ausschnitt des XML Schema für das IPQ Format

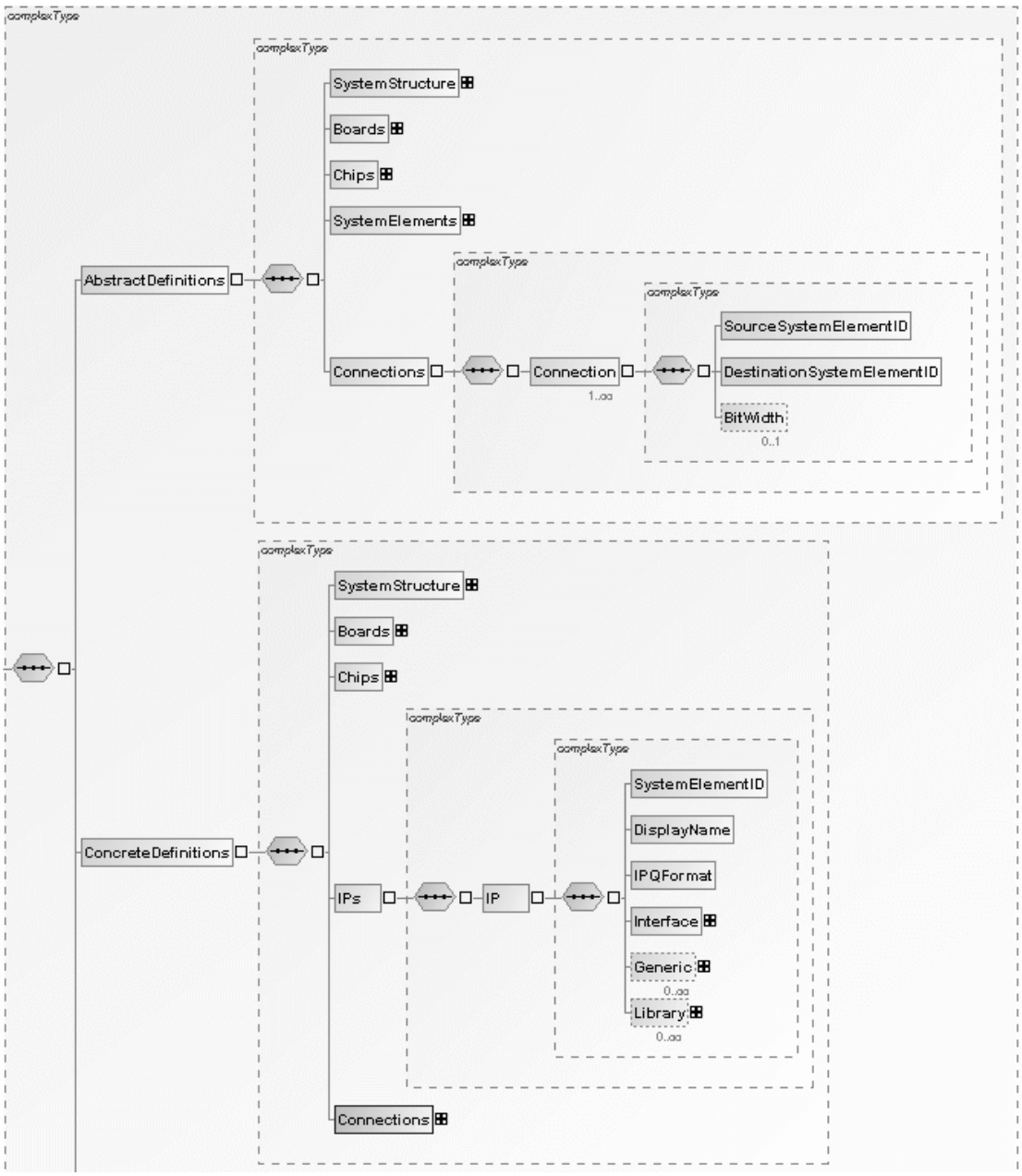


Abbildung 4.3: Ausschnitt des XML Schema für das *System Format*

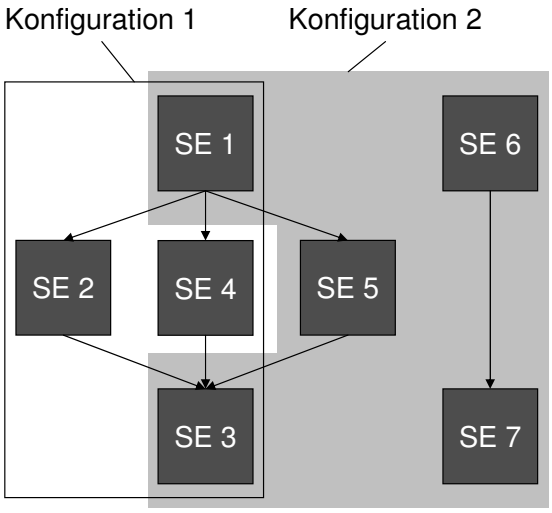


Abbildung 4.4: Problemgraph mit zwei Konfigurationen

eine Automatisierung der Entwurfsschritte und im Besonderen die Partitionierung, zu untersuchen.

Neben diesen Einflüssen im Design Flow, wirkt sich die Partitionierung auch auf den späteren Betrieb des eingebetteten, rekonfigurierbaren Systems aus. Ziel der Partitionierung muss es daher sein, unabhängig vom möglichen Mehraufwand im Design Flow, dass die spezifizierten Eigenschaften des Systems eingehalten werden können. Da die Rekonfigurierungen, im Kontext dieser Arbeit, nicht Bestandteil des eigentlichen Datenflusses des Systems sind, führen sie in den meisten Fällen zu Unterbrechungen im Systemablauf. Daraus ergibt sich, dass Rekonfigurierungen nur in vertretbaren Fällen, was mitunter selten bedeuten kann, durchgeführt werden sollten. Die Dauer der jeweiligen Rekonfigurierungen, die von der Größe der zu rekonfigurierenden Module abhängig ist, ist ebenfalls auf ein Minimum zu reduzieren. Für eine geeignete Partitionierung bedeutet dies, dass der Datenfluss eines Systems bei der Einteilung berücksichtigt werden muss.

Werden kritische Datenflusspfade[6] nicht durch Rekonfigurierungen unterbrochen, lassen sich die Systemanforderungen aus der Spezifikation mit bekannten Entwurfsmitteln leicht umsetzen. Um eine Unterbrechung zu vermeiden, sind die kritischen Pfade des Datenflussgraphen jeweils komplett über eine Konfiguration im FPGA bereitzustellen. Als kritisch können beispielsweise solche Datenflusspfade angesehen werden, die zu einem Anwendungsfall der Spezifikation gehören. Diese Definition bedeutet, dass eine Rekonfigurierung innerhalb eines solchen Anwendungsfalls als nicht vertretbar angesehen wird. Aus der Menge der Anwendungsfälle ergeben sich dann verschiedene Konfigurationen $konf_i \in Konf = \{konf_1, ..., konf_o\}$[7], die auf den FPGA abzubilden sind. Eine Konfiguration $konf_i \in Konf$ des Systems muss nicht nur einen Pfad des Datenflussgraphen umfassen (siehe Konfiguration 1 in Abbildung 4.4). Des Weiteren

[6] Ein Pfad P ist eine Folge von Kanten. $P_{PG} = (kv_1, ..., kv_z) : kv_i = (se_j, se_k) \wedge kv_{i+1} = (se_k, se_l)$ (Analog gilt diese Definition für den Architekturgraphen AG jedoch mit der Möglichkeit von Mehrfachkanten)

[7] Im Abschnitt 2.1 wurden eingebettete Systeme mittels Automaten beschrieben. Die in diesem Abschnitt definierte Menge Q_K kann mit der Menge $Konf$ verglichen werden.

können auch unabhängige Teile des Graphen, wie bei der Konfiguration 2, zu einer Konfiguration gehören. Die Kommunikationskanäle werden bei der Definition 4.3 von Konfigurationen außer Acht gelassen, da Konfigurationen sich aus einer Menge der Funktionalitäten (Systemelementen), die für einen Anwendungsfall benötigt werden, definieren. Eine Konfiguration $konf_i \in Konf$ besteht aus einer Teilmenge der Systemelemente $SE = \{se_1, ..., se_n\}$.

$$\forall konf_i \in Konf : konf_i \subseteq SE \tag{4.3}$$

Alle Systemelemente, die zu einer Konfiguration $konf_i$ gehören, sind, wenn diese Konfiguration $konf_i$ geladen ist, auf dem FPGA Baustein abgebildet. Da keine Einschränkungen bei der manuellen Definition von Konfigurationen existieren, kann das Ziel, kritische Datenflusspfade zu Konfigurationen zusammenzufassen, auch verfehlt werden. Hier sind daher vom Entwickler die spezifizierten Anforderungen zu berücksichtigen. Eine automatisierte Unterstützung bei der Erkennung von kritischen Pfaden ist nicht vorgesehen. Es würde ein sehr hohen Simulationsaufwand benötigt, um eine zuverlässige Erkennung bereitzustellen, wenn dies überhaupt in einer akzeptablen Zeit möglich ist.

Die Bestimmung der Konfigurationen $konf_i \in Konf$ kann prinzipiell auf allen Abstraktionsebenen des Design Flow durchgeführt werden. Empfehlenswert ist jedoch, nach der Erstellung des Problemgraphen oder auch des Architekturgraphen, die Konfigurationen festzulegen, da auf dieser Abstraktionsebene die verschiedenen Anwendungsfälle der Spezifikation, und damit die kritischen Pfade, für den Entwickler noch ersichtlich sind und der Datenflussgraph überschaubar ist.

Im Overlaying Konzepts von Pascal ist die Bestimmung der Programmmodule, die in den Arbeitsspeicher nach Bedarf geladen werden sollen, vom Programmierer manuell durchzuführen. Analog obliegt dem Systementwickler die Definition der Konfigurationen. Neben der Bestimmung der Konfigurationen aus der Spezifikation heraus, ist auch eine automatisierte Detektion der kritischen Pfade des Datenflussgraphen denkbar. Mit einer Überwachung des Systems während des Betriebs, kann festgestellt werden, welche Datenpfade bei welchem Anwendungsfall genutzt werden. Alle Teile des Systems, die bei einem Anwendungsfall nicht aktiv genutzt werden, können aus der Konfiguration für diesen Anwendungsfall entfernt werden. Die automatisierte Bestimmung der Konfigurationen kann entweder auf höheren Abstraktionsebenen durch Simulation oder, wenn entsprechende Bitstreams vorliegen, durch Emulation realisiert werden. Aufgrund des höheren Entwurfsaufwands, durch das Einbinden von Datenflussmonitoren und deren Auswertung nach einer Simulation oder Emulation, wird auf die automatisierte Konfigurationsbestimmung in dieser Arbeit nicht näher eingegangen.

> Für das oben angeführte Beispiel eines Musik-Players sind sieben Anwendungsfälle spezifiziert, was zu folgenden Konfigurationen bei der Partitionierung führt. Zwei dieser Konfigurationen umfassen die beiden, digitalen Radiofunktionen. Für das Abspielen von Musik Dateien ergeben sich, für die jeweiligen Kodierungsverfahren, vier Konfigurationen. Die siebte Konfiguration umfasst den Ruhezustand des Systems. In der Abbildung 4.5 sind, für das Beispiel des Musik Players, die Systemelemente die zur Konfiguration des Ruhestandes (engl.: Idle) gehören, markiert. Ein Überblick über alle Systemelemente des Beispiels und deren Zugehörigkeit zu Konfigurationen ist in der Tabelle 4.1 aufgelistet.

Mit der Einteilung des Systems in Konfigurationen, kann die Unterbrechung von kritischen Datenpfaden durch Rekonfigurierung unterbunden werden. Jedoch bedeutet jeder Konfigurati-

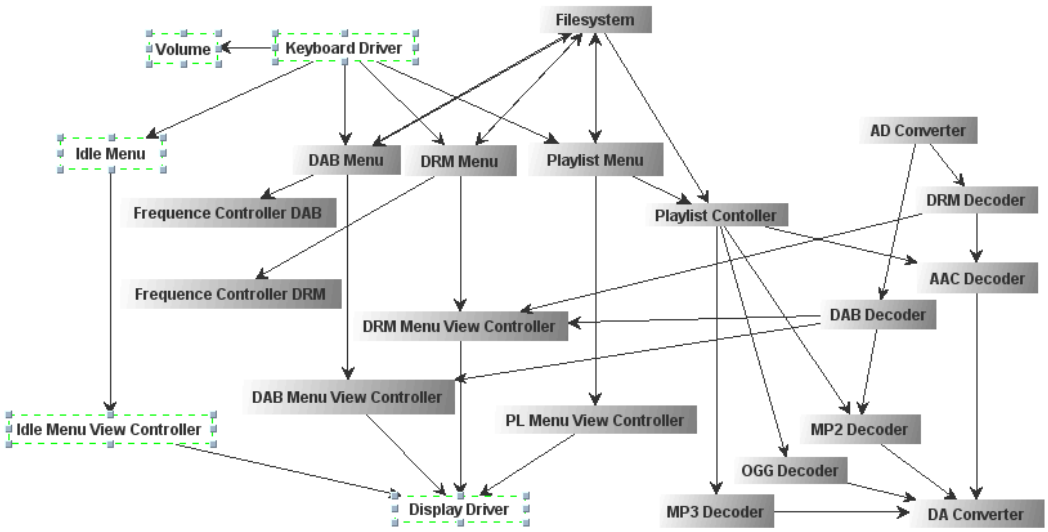


Abbildung 4.5: Idle Konfiguration des Musik Players

onswechsel eine komplette Neuprogrammierung des FPGAs. Diese zeitaufwändige Programmierung steht im Widerspruch zu der Forderung, dass die Rekonfigurierungsdauer minimal gehalten werden sollte. Um dieser Anforderung zu genügen, werden die Konfigurationen in Module unterteilt. Ein Konfigurationswechsel setzt sich dann aus der Rekonfigurierung von Modulen zusammen. Module die zu zwei Konfigurationen gehören, müssen bei einem Konfigurationswechsel zwischen diesen beiden Konfigurationen nicht rekonfiguriert werden. Aus der Definition von Konfigurationen (siehe Gleichung 4.3) wird deutlich, dass auch ein Modul $mod_i \in Mod = \{mod_1, ..., mod_p\}$ eine Teilmenge der Systemelemente SE ist (siehe Gleichung 4.4) und aus einem oder mehreren $se_i \in SE$ besteht.

$$\forall mod_i \in Mod : mod_i \subseteq SE \tag{4.4}$$

$$\bigcap Mod = \emptyset \tag{4.5}$$

$$\forall konf_i \in Konf : konf_i \subseteq Mod \tag{4.6}$$

$$\exists konf_a, konf_b \in Konf : konf_a \neq konf_b \wedge konf_a \cap konf_b = c : c \subseteq SE \wedge c \neq \emptyset \tag{4.7}$$

Eine weitere Eigenschaft der Module mod_i ist, dass sie zueinander disjunkt sind, was für die Konfigurationen $konf_i$ nicht gelten muss. Wenn es Systemelemente $se_i \in SE$ in der Schnittmenge zweier Konfigurationen gibt (siehe Gleichung 4.7), können diese, bei einer Rekonfigurierung von $konf_a$ nach $konf_b$ bzw. umgekehrt, auf dem FPGA belassen werden und auch weiter aktiv Daten verarbeiten. In der Gleichung 4.6 ist die Unterteilung von Konfigurationen in einzelne Module beschrieben. Die Konfigurationen wurden aus den Anwendungsfällen der Spezifikationen heraus definiert. Im Gegensatz dazu definieren sich die Module aus einer Äquivalenzklassenbildung über die Zugehörigkeit von Systemelementen zu Konfigurationen.

Die Äquivalenzklasse eines Systemelements ist die Menge aller Systemelemente, die zur gleichen Menge von Konfigurationen gehören. Zwei Systemelemente se_i und se_j gehören zu

einer Klasse, wenn über die Menge aller Konfigurationen die Gleichung 4.8 gilt.

$$\bigwedge_{konf \in Konf} (se_i \in konf \Leftrightarrow se_j \in konf) \tag{4.8}$$

Die Menge der Äquivalenzklassen aller Systemelemente ist die Menge aller Module.

Auch wenn die Konfigurationsbestimmung prinzipiell in allen Abstraktionsebenen möglich ist, so ist die Modulbildung über Systemelemente leichter zu handhaben als beispielsweise auf der RT Ebene, weil fertige IPs nicht mehr zur Bestimmung der kritischen Pfade analysiert werden müssen. Wenn eine IP als kompletter Baustein eingesetzt wird, bleiben die IP spezifischen Laufzeiten und Optimierungen weitestgehend erhalten. Die Einteilung des Systems in Module auf höheren Abstraktionsebenen, hat Einfluss auf die Optimierungsmöglichkeiten bei der Platzierung der Module im FPGA. Es ist möglich, dass größere Flächen im FPGA temporär ungenutzt bleiben. Dieser Nachteil wirkt sich jedoch nicht negativ auf die Systemperformanz aus und wird daher als zweitrangig eingestuft.

Die Kommunikationskanäle KV wurden bei der Konfigurationsbildung, wie auch bei der Unterteilung der Konfigurationen in Module nicht betrachtet, da die Partitionierung über die Funktionalität des Systems motiviert wurde. Die einzelnen Funktionalitäten $se \in SE$ sind jedoch im Problemgraph PG mit Kanten verbunden. Diese Kommunikationskanten werden durch die Konfigurationsbildung nicht verändert, sondern werden durch die Modularisierung des Systems in zwei verschiedene Typen unterteilt. Der eine Typ definiert Verbindungen kv zwischen Systemelementen se, die zu unterschiedlichen Modulen mod gehören. Diese Kanten $ModKV$ (siehe Gleichung 4.10) eines Modulgraphen MG (siehe Gleichung 4.9) werden im Verlauf des Design Flows als Bus Makros realisiert.

$$MG = (Mod, ModKV) \tag{4.9}$$

$$\begin{aligned} ModKV \subseteq KV : \\ \forall modkv = (mod_k, mod_l, \beta) \in ModKV : \\ \exists kv = (se_i, se_j, \alpha) \in KV : \\ se_i \in mod_k \wedge se_j \in mod_l \wedge mod_k \neq mod_l \end{aligned} \tag{4.10}$$

$$modkv \subseteq KV \tag{4.11}$$

$$\beta = \sum_{kv=(se_a,se_b,\alpha) \in modkv} \alpha \tag{4.12}$$

Der zweite Kantentyp bildet die Teilmenge $KV \setminus ModKV$[8], die nicht als Bus Makro realisiert werden muss. Es können mehrere Kanten kv in einer Modulgraphkante zusammengefasst werden (siehe Gleichung 4.11). Im Kontext des Architekturgraphen wird diese Mehrfachkante jedoch mit der Kardinalität β gewichtet (siehe Gleichung 4.12). Der beschriebene Modulgraph ist im nächsten Entwurfsschritt, der *Platzierung*, auf einen geeigneten FPGA abzubilden.

> Aus den in Abbildung 4.4 schematisch dargestellten Konfigurationen, ergeben sich drei Äquivalenzklassen für die Systemelemente 1 bis 7. In einer ersten Klasse, die

[8] $KV \setminus ModKV := \{x | (x \in KV) \wedge (x \notin ModKV)\}$

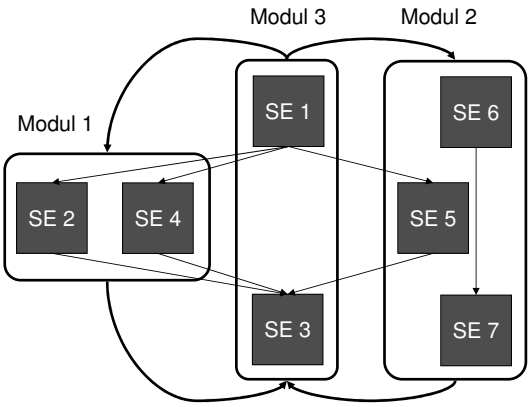


Abbildung 4.6: Beispiel für einen Modulgraphen

als *Modul 1* in Abbildung 4.6 bezeichnet ist, befinden sich alle Systemelemente, die nur für die *Konfiguration 1* benötigt werden. Im Beispiel sind dies die Elemente *SE 2* und *SE 4*. Analog dazu bilden die Systemelemente *SE 5*, *SE 6* und *SE 7* eine zweite Äquivalenzklasse. Alle weiteren Systemelemente, die für beide Konfigurationen benötigt werden, bilden das, aus der Äquivalenzklassenbildung resultierende, dritte Modul, *Modul 3*.

In diesem Beispiel ist auch die Zusammenfassung mehrerer Kanten kv zu einer Modulkante $modkv$ dargestellt. So setzt sich die Kante $modkv =$ (*Modul 3*, *Modul 1*, β) aus den beiden Kanten $kv =$ (*SE 1*, *SE 2*, α) und $kv =$ (*SE 1*, *SE 4*, α) zusammen.

Für das Beispiel des Musik-Players sind die durch Äquivalenzklassenbildung ermittelten Module in Tabelle 4.2 aufgelistet.

Aus dem vorgestellten Ansatz zur Partitionierung, resultieren folgende Eigenschaften für ein dynamisch rekonfigurierbares System:

Alle kritischen Datenpfade des Systems werden durch den Konfigurationsansatz komplett auf den FPGA abgebildet, so dass die kritischen Pfade eines Anwendungsfalls nicht durch Rekonfigurierungen unterbrochen werden. Aus dieser Konfigurationsbildung heraus ergibt sich, dass die benötigte FPGA-Größe durch die Konfiguration mit den meisten Ressourcenanforderungen festgelegt wird. Unter der Annahme, dass ein Anwendungsfall wiederholt ausgeführt wird, bevor zu einem weiteren Anwendungsfall gewechselt wird, gilt für diesen Ansatz, dass relativ selten rekonfiguriert werden muss. Eine Reduzierung der Rekonfigurierungsdauer gegenüber einer kompletten FPGA Rekonfigurierung konnte durch die Einteilung von Konfigurationen in Module erreicht werden. Durch die Äquivalenzklassenbildung über alle Systemelemente werden die Differenzen zwischen den einzelnen Konfigurationen bestimmt. Eine Rekonfigurierung des kompletten FPGA kann hierdurch umgangen werden. Bei einem Konfigurationswechsel werden nur neue, benötigte Systemelemente, zusammengefasst zu Modulen, in freie oder überschreibbare Flächen des FPGA geladen.

4.3 Robustheitsaspekte

Es ist von großer Bedeutung, dass künftige elektronische Systeme ein hohes Maß an Vertrauen in ihre Funktionalität erreichen. Besonders wegen der zunehmenden Durchdringung von eingebetteten Systemen [81] in alle Lebensbereiche müssen sie zuverlässig und robust sein.

Mit Robustheit kann im Allgemeinen die Fähigkeit eines Systems bezeichnet werden, bei veränderten Gegebenheiten keine spezifikationswidersprechende Reaktionen aufzuweisen. Veränderte Gegebenheiten können dabei durch neue Umweltbedingungen, partielle Systemausfälle oder Bedienfehler auftreten. Die resultierenden schlimmsten Fälle (engl.: worst case) erfordern, um die Robustheit zu gewährleisten, entweder geeignete Algorithmen mit akzeptabler Laufzeit oder auch Migrationsmöglichkeiten und Redundanzkonzepte innerhalb des eingebetteten Systems. Robustheit bezieht sich dabei auf eine oder mehrere veränderte Gegebenheiten. Ein System kann beispielsweise robust gegen starke Erschütterungen sein, was bei eingebetteten System in Flugzeugen gefordert wird. Im Folgenden wird auf den Einsatz von Rekonfigurierung zur Verbesserung des Maßes an Robustheit eingegangen.

Sind einzelne Funktionen oder Module in einem System fehlerhaft, bieten rekonfigurierbare Schaltkreise die Möglichkeit, diese Fehler zu eliminieren ohne den Chip austauschen zu müssen. Ein Entwickler muss, um ein Update durchzuführen, nur einen korrigierten Bitstream bereitstellen und auf den FPGA laden. Für das in dieser Arbeit zu Grunde liegende Rekonfigurierungskonzept sind hierbei folgende Punkte zu berücksichtigen. Unter Ausnutzung der partiellen Rekonfigurierung können Updates auf Module beschränkt werden. Dabei ist zu berücksichtigen, dass es statische Module gibt, die immer auf dem FPGA Chip geladen sein müssen. Eine Korrektur solcher Module ist durch eine neue Initialisierung des Systems realisierbar, was zwangsweise eine Systemunterbrechung bedeutet. Im Gegensatz zu den statischen Modulen lassen sich die dynamischen leicht austauschen. In diesem Fall muss nur das für die Rekonfigurierung im Speicher bereitgestellte, zu erneuernde Modul überschrieben werden. Die Aktualisierung des FPGAs erfolgt dann durch die Konfigurationswechsel im Ablaufprozess des Systems und führt zu keiner zusätzlichen Unterbrechung. Module, die dynamisch nachgeladen werden, sind nur dann leicht zu überschreiben, wenn das korrigierte Modul in die gleiche, zur Verfügung stehende FPGA Fläche passt und auch dieselben Schnittstellen verwendet wie das fehlerhafte Modul. Sind diese Randbedingungen nicht gegeben, muss nicht nur dieses Modul synthetisiert werden, sondern das gesamte System, um eine ausreichend große Modulfläche im FPGA bereitzustellen. Daraus ergibt sich, wie bei den statischen Modulen, dass das System komplett neu initialisiert werden muss. Dieser Robustheitsaspekt von dynamisch rekonfigurierbaren Systemen kann folgendermaßen zusammengefasst werden.

- Ein dynamisch rekonfigurierbares System, ist robust gegen zusätzliche Unterbrechungen, hervorgerufen durch Updates, die aufgrund von fehlerhaften Modulen durchgeführt werden müssen, wenn die fehlerhaften Module nicht statisch sind und deren FPGA Fläche und Kommunikationsverbindungen zu anderen Modulen beibehalten werden können.

Bei verschiedenen Einsatzbereichen von eingebetteten Systemen besteht die Forderung, dass diese gegen Fehler, hervorgerufen durch Ausfälle, robust sind. Um diese Robustheit, in Grenzen, bereitstellen zu können, werden Komponenten eines Systems, beispielsweise in der Raumfahrt, redundant vorgehalten [135]. Man unterscheidet hierbei zwischen statischer und dynamischer Redundanz [96]. Die statische Redundanz bedeutet, dass alle redundant vorhandenen Kompo-

nenten ständig aktiv sind. Im Gegensatz dazu wird bei der dynamischen Redundanz[9] die redundante Komponente erst nach dem Auftreten eines Fehlers aktiviert. Beide Typen der Redundanz lassen sich mit dem oben vorgestellte Konfigurationskonzept (siehe Abschnitt 4.2) modellieren und in dynamisch rekonfigurierbare Systeme integrieren. Im Fall der statischen Redundanz sind nur den jeweiligen Konfigurationen die redundanten Systemelemente hinzuzufügen.

Sind redundant vorhandene Systemelemente erst bei Auftreten eines Fehlers zu aktivieren, kann dies über zusätzliche Konfigurationen modelliert werden. Über Rekonfigurierung können dann die redundanten Funktionalitäten aktiviert werden. Durch die Rekonfigurierung muss ein Modul nicht unbedingt redundant vorhanden sein, sondern kann prinzipiell in andere Bereiche eines FPGAs oder auch auf andere Schaltkreise migrieren. Eine Beschreibung eines Rechenclusters, wo eine derartige Migration von Hardwarefunktionalität über Rekonfigurierung realisiert werden soll, findet sich [101]. Bei der Rekonfigurierung zur Aktivierung von Modulen sind, wenn das originale und das redundante Modul in unterschiedlichen Bereichen des FPGAs platziert sind, Multiplexer[10] notwendig, damit die Kommunikationskanäle umgeschaltet werden können. Auf die Multiplexer und deren Notwendigkeit in dynamisch rekonfigurierbaren Systemen wird in den Abschnitten 5.4 und 6.2.5 näher eingegangen.

- Die Rekonfigurierung kann zur Aktivierung von redundanten Modulen, im Fall der dynamischen Redundanz, eingesetzt werden. Mit dem Konfigurationskonzept können die verschiedenen Szenarien, bei denen redundante Funktionalitäten aktiv sind, beschrieben werden.

Analog zur dynamischen Redundanz, bei welcher funktionsäquivalente Konfigurationen mit redundanten Modulen definiert werden, können auch Konfigurationen definiert werden, die beispielsweise Fail Safe Funktionalität bereitstellen.

- Die Behandlung von Systemfehlern über zusätzliche Konfigurationen ist mit dem in dieser Arbeit zu Grunde liegenden Konzept für dynamisch rekonfigurierbare Systeme leicht zu realisieren und kann zur Robustheit gegenüber solchen Fehlern führen.

Bei einem eingebetteten System, kann es notwendig sein, um den Leistungsanforderungen gerecht zu werden, verschiedene Hardwarefunktionalitäten einzusetzen. Ein Beispiel, wo unterschiedliche Datenmengen und daraus sich ergebende Hardwareanforderungen auftreten, ist das Raytracing von 3-D Objekten. Weit verbreitet sind, für die Beschreibung einer 3-D Welt, Polygonnetze, die einen Kompromiss zwischen Speicherverbrauch und Darstellungsgeschwindigkeit bilden. Auch Voxelgitter[11] finden Anwendung bei Volumendarstellungen [23, 103, 45]. Voxelgitter zeichnen sich, je nach Detaillierungsgrad, durch einen sehr hohen Speicherbedarf aus. Das Konfigurationskonzept bietet hierfür, in Verbindung mit Rekonfigurierung, eine einfache Möglichkeit spezifizierte Anforderungen, für die Verarbeitung von 3-D Daten, effizient umzusetzen.

- Rekonfigurierbare Systeme können robust gegenüber unterschiedlichen Effizienzanforderungen realisiert werden.

[9] Dies wird auch mit Standby Verfahren bezeichnet.

[10] Ein Multiplexer ist eine Logikschaltung, die, über Steuerleitungen, aus mehreren Eingangsleitungen eine auswählt. Der Antagonist des Multiplexes wird mit dem Demultiplexer bezeichnet.

[11] Zusammengesetzt aus den Wörtern *Volumetric* und *Pixel*. Eine Menge von Punkten im 3-D Raum.

Die vorgestellten Möglichkeiten, um das Maß der Robustheit in eingebetteten Systemen zu erhöhen, zeigen auf, dass die Anforderungen an die Robustheit des Systems schon in den frühen Phasen eines Entwurfs zu berücksichtigen sind. Für verschiedene Aspekte der Robustheit ist der Einsatz von Rekonfigurierung vorteilhaft, um Systemausfälle oder Einschränkungen zu unterbinden. Der Abschnitt *Robustheitsaspekte* hat nicht den Anspruch vollständig zu sein und geht ebenso nicht auf eine Bewertung der vorgestellten Konzepte ein, da Robustheit nicht den Kernpunkt dieser Arbeit darstellt. Es wurden lediglich Einsatzmöglichkeiten für rekonfigurierbare, eingebettete Systeme motiviert.

4.4 Zusammenfassung

In diesem Kapitel wurde das Konfigurationskonzept für die automatisierte Partitionierung eines dynamisch rekonfigurierbaren Systems vorgestellt. Die Partitionierung baut auf einer Systembeschreibung auf, die in den vorhergehenden Schritten des Design Flows erzeugt wird. In einem ersten Schritt werden aus der Spezifikation eines Systems die benötigten Funktionalitäten bestimmt und mit Hilfe eines Problemgraphen beschrieben. Ausgehend von diesem Graphen werden, zur Realisierung der Funktionalitäten, IPs gesucht und zu einem System kombiniert. Der resultierende Architekturgraph dient zur Beschreibung des aus IP Blöcken bestehenden Systems. Sowohl der Problemgraph, als auch der Architekturgraph, werden für die weiteren Entwurfsschritte, in XML gespeichert.

Im Hauptteil dieses Kapitels wurde ein Konfigurationskonzept vorgestellt, um kritische Datenpfade eines Systems nicht durch Rekonfigurierungen zu unterbrechen. Zur Reduzierung der Rekonfigurierungsdauer, die von der Größe der zu rekonfigurierenden Hardwarefunktionalitäten abhängig ist, wurde die Modulbildung durch Äquivalenzklassenbestimmung vorgestellt. Das Ergebnis der vorgestellten Partitionierung ist der Modulgraph, dessen Kanten die benötigten Bus Makros repräsentieren.

Dieses Kapitel wird abgeschlossen durch eine Betrachtung von Robustheitsaspekten in dynamisch rekonfigurierbaren Systemen. Dynamische Rekonfigurierung kann ausgenutzt werden, um das Maß an Robustheit zu steigern, wobei die vorgestellte Partitionierung keine nachteiligen Auswirkungen auf die Robustheit hat.

SE \ $Konf$	DAB	DRM	Idle	PL AAC	PL MP2	PL MP3	PL OGG
AAC Decoder		X		X			
AD Converter	*X*	*X*					
DA Converter	*X*	*X*		*X*	*X*	*X*	*X*
DAB Decoder	X						
DAB Menu	X						
DAB Menu View Controller	X						
Display Driver	X	X	X	X	X	X	X
DRM Decoder		X					
DRM Menu		X					
DRM Menu View Controller		X					
Filesystem	X	X		X	X	X	X
Frequence Controller DAB	X						
Frequence Controller DRM		X					
Idle Menu			X				
Idle Menu View Contoller			X				
Keyboard Driver	X	X	X	X	X	X	X
MP2 Decoder	X				X		
MP3 Decoder						X	
OGG Decoder							X
PL Menu View Controller				X	X	X	X
Playlist Controller				X	X	X	X
Playlist Menu				X	X	X	X
Volume	X	X	X	X	X	X	X

Tabelle 4.1: Systemelemente und Konfigurationen des Beispiel Musik-Players

Systemelemente	Module
AAC Decoder	Modul AAC
DAB Decoder DAB Menu DAB Menu View Controller Frequence Controller DAB	Modul DAB
DRM Decoder DRM Menu DRM Menu View Controller Frequence Controller DRM	Modul DRM
Filesystem	Modul Filesystem
Idle Menu Idle Menu View Contoller	Modul Idle
Display Driver Keyboard Driver Volume	Modul Keyboard-Display
MP2 Decoder	Modul MP2
MP3 Decoder	Modul MP3
OGG Decoder	Modul OGG
PL Menu View Controller Playlist Controller Playlist Menu	Modul Playlist
RCU	*Modul RCU*

Tabelle 4.2: Module des Beispiel Musik-Players

5 Platzierung

Mit dem Schritt der Partitionierung werden eingebettete Systeme in Module unterteilt. Diese Module sind, nach dem, in Abschnitt 2.4 vorgestellten, Overlaying Konzept für dynamisch rekonfigurierbare Systeme, für den weiteren Design Flow auf einen FPGA abzubilden. Dies erfordert eine Aufteilung der FPGA Fläche in Rekonfigurierungsbereiche, denen die verschiedenen Module zugeordnet werden. In diesem Kapitel wird ein Lösungsansatz für diese Anforderung vorgestellt und ein Ansatz für die Platzierung der Rekonfigurierungsbereiche im FPGA beschrieben.

In diesem Kapitel wird als erstes, im Abschnitt 5.1, das Platzierungsproblem näher beleuchtet und die Ziele des entwickelten Verfahrens vorgestellt (siehe nachfolgenden Abschnitt 5.2). Der Hauptabschnitt 5.3 dient zur Einführung von formalen Definitionen und der Vorstellung des Lösungsansatzes. Die Slotbestimmung und die Slotplatzierung des entwickelten Verfahrens werden in den zwei Unterabschnitten 5.3.1 und 5.3.2 im Detail beschrieben. Ebenso befindet sich in diesem Hauptteil der Arbeit eine Bewertung der Laufzeit der Algorithmen vorgenommen. Bevor das Kapitel mit einer Zusammenfassung abgeschlossen wird, wird im Abschnitt 5.4 auf Besonderheiten der Kommunikation zwischen Overlaying Bereiche hingewiesen.

5.1 Platzierungsproblem

Die Aufgabe des Syntheseschrittes *Platzierung* ist es, für jede Konfiguration die Module auf dem FPGA zu platzieren. Dabei sind jedoch mehrere Anforderungen zu berücksichtigen. Beispielsweise müssen Module, die gegeneinander ausgetauscht werden, innerhalb einer gemeinsamen FPGA Fläche platzierbar sein, um eine fehlerhafte Veränderung von nicht betroffenen Modulkonfigurationen zu vermeiden. Aus dieser Anforderung heraus ergibt sich, dass Bereiche eines FPGAs als Rekonfigurierungsflächen definiert werden sollten. Um nun rekonfigurierbare Module zu synthetisieren, müssen die Synthesewerkzeuge das jeweilige Modul in solch eine Fläche platzieren. Diese Begrenzungsrahmen für die Synthese werden auch mit *Bounding Box* bezeichnet [28, 141]. Da die für diese Arbeit verfügbaren Virtex II Pro FPGAs nur eine spaltenweise Rekonfigurierung[1] erlauben (siehe Abschnitt 2.2.2), werden im Folgenden diese FPGA Bereiche als *Slots* bezeichnet.

Durch die Definition von Slots wird das Problem der Platzierung von Modulen in zwei Schritte unterteilt. Der erste Schritt umfasst die Bestimmung der Slots, die mit geeigneten Methoden auf die zur Verfügung stehende FPGA Fläche zu platzieren sind. Die Slots werden dann, für die weitere Synthese der Module, in einer mit *Constraint File* bezeichneten Datei den Synthesewerkzeugen bereitgestellt. Im zweiten Schritt werden die Gatter der Module in die entsprechenden Slots durch Standardsynthesewerkzeuge abgebildet. Dieser zweite Schritt wird zu einem späteren Zeitpunkt des Design Flows, bei der Synthese der Bitstreams, durchgeführt und

[1] Die Rekonfigurierung Fläche umfasst die komplette Höhe eines FPGAs und kann, abgesehen von einer Mindestbreite, beliebig groß sein.

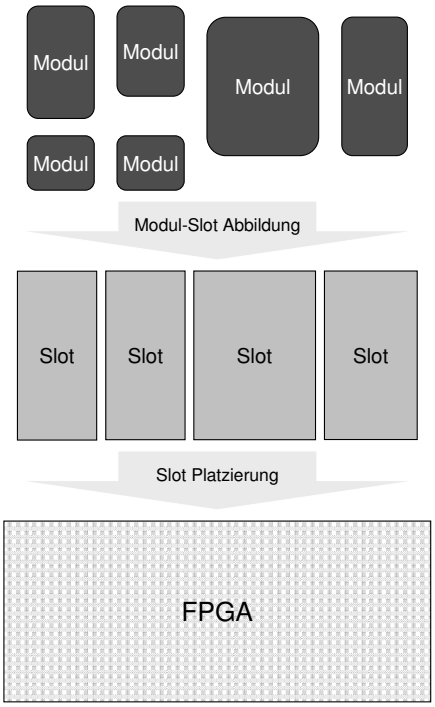


Abbildung 5.1: Schematische Darstellung des aufgeteilten Platzierungsproblems

ist daher nicht Inhalt des in diesem Kapitel vorgestellten Entwurfsschritts. Die hier vorgestellte Bestimmung der Slots ist jedoch abhängig von den abzubildenden Modulen. In diesem Zusammenhang wird festgelegt, welche Module auf welche Slots im späteren Design Flow abgebildet werden (siehe Abbildung 5.1). Die Bestimmung des Slots ist nicht allein für die Synthese der partiellen Bitstreams notwendig, sondern beeinflusst auch die automatisierte Generierung einer Steuereinheit für die Rekonfigurierungen und ist von essenzieller Bedeutung für die Platzierung der Bus Makros.

Um für die Module Rekonfigurierungsflächen zu definieren, sind folgende Probleme zu lösen:

- Wie viele Slots werden benötigt und wie groß müssen diese sein? Diese Fragestellung zielt unter anderem auf die Mindestgröße des FPGAs zu Realisierung eines dynamisch rekonfigurierbaren, eingebetteten Systems ab.

- Welche Module werden bei welcher Konfiguration in welche Slots abgebildet? Diese Information wird bei der Bitstream Generierung und auch für die Steuerung der Rekonfigurierung benötigt.

- Wo werden die Slots im FPGA platziert? Diese Frage setzt voraus, dass die Modul-Slotabbildung, Anzahl und Größe der Slots definiert sind. Die Lage der Slots im FPGA und die Modul-Slotabbildung sind für die Bus Makro Generierung notwendig.

5.2 Ziele der Platzierung

Um nicht nur *eine mögliche*, sondern *eine optimale* Lösung[2] für die im Abschnitt 5.1 vorgestellten Probleme zu finden, sind messbare Eigenschaften die durch die Slotplatzierung beeinflusst werden, zu definieren.

- Eine dieser Eigenschaften ist die **Rekonfigurierungsdauer**. Die Rekonfigurierungsdauer ist die Zeit, die von der Anforderung, eine neue Konfiguration zu laden, bis zur Rückmeldung der Steuereinheit, dass die Rekonfigurierung abgeschlossen ist, vergeht. Während dieser Zeit kann die alte Konfiguration in den meisten Fällen nicht mehr aktiv Daten verarbeiten, da Module dieser Konfiguration höchstwahrscheinlich rekonfiguriert werden. Und auch die neu zu ladende Konfiguration steht erst nach Abschluss der Rekonfigurierung zur Verfügung. Ziel sollte es daher seien, die Dauer dieser Systemunterbrechungen zu minimieren. Die Dauer der Rekonfigurierung ist direkt proportional zur Menge der zu übertragenden Rekonfigurierungsdaten, die wiederum abhängig ist von der Anzahl und Größe der Module. Die Minimierung der Systemunterbrechungen wurde auch schon bei den Einflüssen einer Partitionierung in Abschnitt 4.2 als ein wichtiges Entwurfsziel herausgestellt.

- Für miteinander kommunizierende Module, die Slots zugeordnet sind, die nicht nebeneinander im FPGA liegen, werden lange **Kommunikationskanäle** notwendig. FPGAs bieten hierfür, wie in der Abbildung 5.2 zu sehen ist, so genannte *Long Lines* [147]. Lange Kommunikationsverbindungen haben jedoch den Nachteil einer größeren Latenz der über diesen Kanal übertragenen Signale. Dies ist in vielen Fällen der Grund, für eine gedrosselte Taktfrequenz. Beim Entwurf eines dynamisch rekonfigurierbaren, eingebetteten Systems sollte daher darauf geachtet werden, die Slots so anzuordnen, dass möglichst wenig lange Kommunikationsverbindungen benötigt werden. In FPGAs existieren weniger *Long Lines* als kurze Verbindungen, wie die *Hex Lines*, *Douple Lines* und *Direct Connections*. Aus diesem Grund sollte ebenfalls die Menge der langen Verbindungen gering gehalten werden.

- Eine dritte Eigenschaft, die bei dem Entwurfsschritt *Platzierung* zu berücksichtigen ist, ist die Größe des externen **Bitstream Dateispeichers**, der so klein wie möglich gehalten werden sollte. In Bitstreams wird nicht nur die Funktionalität beschrieben, sondern auch die Position dieser Funktionalität im FPGA. Das bedeutet für die Module eines rekonfigurierbaren Systems, dass aus einem Modul, das auf zwei verschiedene Slots abgebildet werden soll, zwei verschiedene Bitstream Dateien resultieren.

Um die oben genannten drei Ziele zu erreichen, existieren unter anderem folgende Möglichkeiten. Module die in vielen oder häufig verwendeten Konfigurationen benötigt werden und daher oft Einsatz finden, sollten auf dem FPGA belassen werden. Dies setzt voraus, dass ein FPGA genügend Ressourcen bietet, um diese Module fest zu platzieren. Für eine kürzere, durchschnittliche Rekonfigurierungsdauer sollten nur selten benötigte Module ausgetauscht werden müssen. Positiv auf die Rekonfigurierungsdauer wirken sich auch kleine, auszutauschende Module aus. Daher sollten große Module permanent im FPGA zur Verfügung stehen, auch wenn

[2] Aufgrund der Größe des Lösungsraums, ist ein globales Optimum mit akzeptablem Aufwand schwer zu finden. Eine optimale Lösung kann daher auch ein lokales Optimum bedeuten.

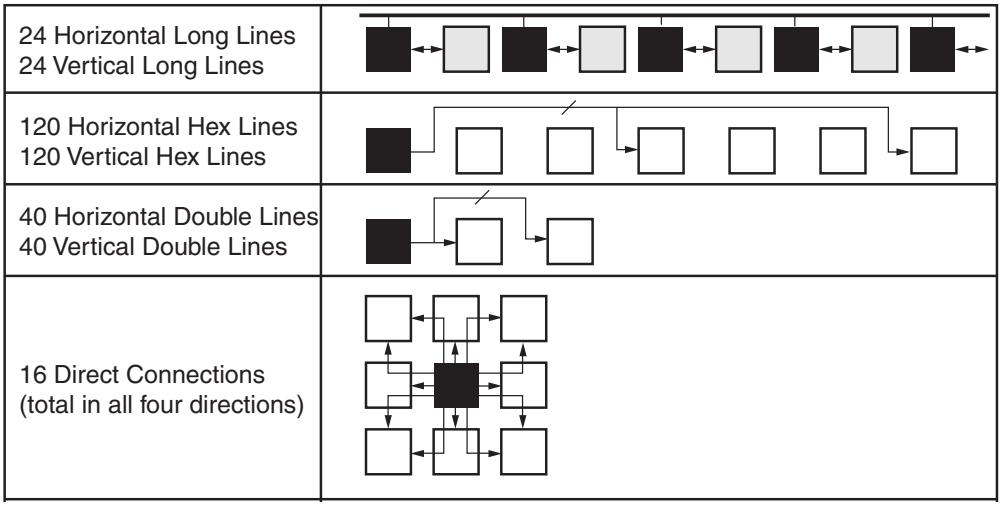


Abbildung 5.2: Kommunikationsverbindungen im Virtex II Pro [147]

sie nicht in allen Konfigurationen benötigt werden. Eine Möglichkeit den Speicher für die Bitstreams nicht unnötig zu belegen, ist die Abbildung der Module auf jeweils nur einen Slot. Die Anzahl und Länge der Kommunikationsverbindungen zwischen den Slots, lässt sich über die Anordnung der Slots im FPGA variieren. Existieren viele, als Bus Makro realisierte Slotverbindungen zwischen zwei Slots, sollten diese nebeneinander platziert werden.

5.3 Platzierungsverfahren

In diesem Abschnitt, wird das im Rahmen dieser Arbeit entwickelte Platzierungsverfahren vorgestellt. Bevor im Abschnitt 5.3.1 auf die Algorithmen des Verfahrens eingegangen wird, werden Definitionen, Modelle und die Voraussetzungen für das Verfahren vorgestellt.

Um eine Aussage treffen zu können, wie oft ein Modul im Verhältnis zu anderen Modulen benötigt wird, muss die Modul-Konfigurationszugehörigkeit betrachtet werden. Wird durch Rekonfigurierung häufig zu einer Teilmenge der Konfigurationen gewechselt, werden auch die Module dieser Teilmenge oft benötigt. Eine Möglichkeit Rekonfigurierungen formal zu modellieren, ist die homogene Markov Kette.

Markov Kette

Die homogene Markov Kette ist ein Modell zur Beschreibung von Zuständen und zufälligen Zustandsübergängen. Eine Einführung zur Markov Kette befindet sich in [93] und [44]. Eine homogene Markov Kette MK wird definiert über ein Tupel von drei Elementen (siehe Gleichung 5.1).

$$MK = (V, E, \vartheta) \tag{5.1}$$

$$\vartheta : E \rightarrow [0, 1] \tag{5.2}$$

Sie besteht aus einer Menge von Knoten V, den Zuständen, und einer Menge von gerichteten Übergangskanten E zwischen den Knoten. Die Funktion ϑ beschreibt die Wahrscheinlichkeit

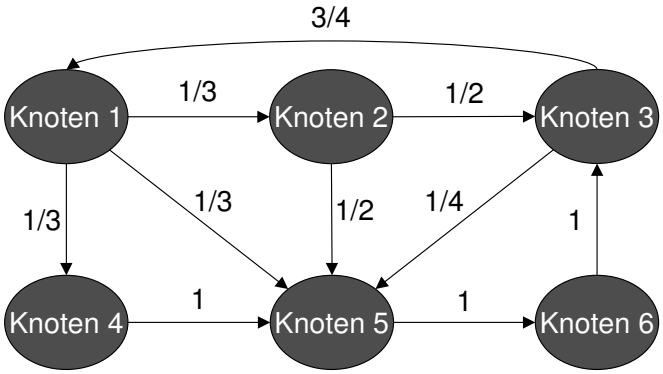


Abbildung 5.3: Beispiel für eine Markov Kette

von Zustandsübergängen (siehe Gleichung 5.2). Ein Beispiel für eine Markov Kette ist in Abbildung 5.3 zu sehen. Die Annotation an den Kanten der Abbildung 5.3 geben, die über die Funktion ϑ definierten Wahrscheinlichkeiten p für das Eintreten eines Zustandsübergangs, an.

Sind die Übergangswahrscheinlichkeiten einer Markov Kette unabhängig von der Zeit, so wird diese als *homogene* Markov Kette bezeichnet. Dies bedeutet, für eine Übergangswahrscheinlichkeit p vom Zustand i zum Zustand j gilt für alle Zeitpunkte t : $p_{i,j} = p_{i,j}(t)$. Neben der Homogenität, ist auch die in Gleichung 5.3 definierte Eigenschaft kennzeichnend für eine Markov Kette.

$$\sum_{j \in V} p_{ij} = 1 : \forall i \in V \tag{5.3}$$

Die Summe der Wahrscheinlichkeiten p aller möglichen Zustandsübergänge $e \in E$ von einem Knoten aus, ist 1. In einer Markov Kette gilt diese Aussage für alle Zustandsknoten $v \in V$. Unter der Voraussetzung, dass eine Markov Kette stark zusammenhängend ist, können, aus den Übergangswahrscheinlichkeiten heraus, die stationären Wahrscheinlichkeiten Π der Zustände $v \in V$ berechnet werden. Die Wahrscheinlichkeit eines Zustands sagt aus, wie wahrscheinlich es ist, nach beliebig vielen Zustandsübergängen, in diesem Zustand zu sein. Eine stark zusammenhängende Markov Kette, in der mindestens ein gerichteter Pfad von jedem Zustand $i \in V$ zu jedem anderen Zustand $j \in V$ existiert, nennt man ergodisch [44].

Die Berechnung der stationären Zustandswahrscheinlichkeiten Π erfolgt über das lineare Gleichungssystem 5.4.

$$\Pi_i = \sum_{j=1}^{n} p_{j,i} \cdot \Pi_j : \forall i = 1, ..., n-1 \tag{5.4}$$

$$\sum_{i=1}^{n} \Pi_i = 1 \tag{5.5}$$

Die stationäre Wahrscheinlichkeit Π_i eines Zustandes i, definiert sich aus der Summe der Wahrscheinlichkeiten aller Zustände j multipliziert mit den Übergangswahrscheinlichkeiten p_{ji} der Übergänge, die den Zustand i als Zielzustand haben. Um die lineare Unabhängigkeit des Gleichungssystems zu gewährleisten, wird die stationäre Wahrscheinlichkeit der Zustände nur für

die Zustände 1 bis $n-1$ definiert. Das Gleichungssystem lässt sich über die Gauss Elimination lösen, indem man die Bedingung 5.5 voraussetzt, die bei Markov Ketten per Definition gegeben ist.

Bildet man die Definition der homogenen Markov Kette auf Rekonfigurierungen ab, so sind die Zustandsknoten V mit der Menge der Konfigurationen $Konf$ gleichzusetzen. Ein Zustandsübergang $e \in E$ ist gleichzusetzen mit einer Rekonfigurierung $rekonf$. Eine Rekonfigurierung definiert sich aus der Ausgangskonfiguration $konf_A$ und der Zielkonfiguration $konf_Z$, die mit einer Wahrscheinlichkeit p auftritt (siehe Definition 5.7). Aus dieser Übertragung der Rekonfigurierung auf die Markov Kette, definiert sich der in Gleichung 5.6 beschriebene Konfigurationsgraph KG. Der Konfigurationsgraph besteht aus der Menge der Konfigurationen $Konf$ und der Menge der Rekonfigurierungen $Rekonf$.

$$KG = (Konf, Rekonf) \tag{5.6}$$

$$rekonf = (konf_A, konf_Z, p) : konf_A, konf_Z \in Konf \tag{5.7}$$

$$p = P(X_{k+1} = konf_Z | X_k = konf_A) \tag{5.8}$$

$$P_{abs}(rekonf = (konf_A, konf_Z, p)) = \Pi_{konf_A} \cdot p \tag{5.9}$$

Die Rekonfigurierungswahrscheinlichkeit p ist die Wahrscheinlichkeit, dass die Konfiguration $konf_Z$ zum Zeitpunkt $k+1$ auf dem FPGA geladen ist, unter der Bedingung, dass zum Zeitpunkt k die Konfiguration $konf_A$ geladen war. Mit der bedingten Wahrscheinlichkeit p lässt sich auch die absolute Wahrscheinlichkeit P_{abs} einer Rekonfigurierung definieren (siehe Gleichung 5.9). Die stationäre Wahrscheinlichkeit des Ausgangszustands multipliziert mit der Wahrscheinlichkeit der betrachteten Rekonfigurierung ergibt P_{abs}. Folgende zwei Fragen lassen sich mit der Modellierung der Rekonfigurierung als Markov Kette beantworten:

- Wie wahrscheinlich ist es, dass eine bestimmte Konfiguration aktiv ist?
- Welche Rekonfigurierung ist höchstwahrscheinlich, d.h. welche hat die größte absolute Wahrscheinlichkeit P_{abs}?

Die erste Frage lässt sich mittels des Gleichungssystems 5.4 beantworten und die zweite mit Berechnung der Gleichung 5.9. Voraussetzung für die Beantwortung der Fragen ist, dass die Bedingung 5.3 gilt und die Ergodizität für den Konfigurationsgraph KG gegeben ist. Theoretisch kann in der Markov Kette von einem Zustand zu sich selbst ein „Zustandsübergang“ auftreten. Da aber bei der Abbildung auf die Rekonfigurierungen keine Kosten für diese Konfigurationswechsel entstehen, werden diese Kanten, im Kontext dieser Arbeit, in der Menge $Rekonf$ des Konfigurationsgraphen KG ausgeblendet.

Die homogene Markov Kette bietet eine passende Möglichkeit die Rekonfigurierungen zu modellieren. Analog zur Definition von Konfigurationen sind auch die Rekonfigurierungswahrscheinlichkeiten vom Entwickler des eingebetteten Systems zu definieren. Das Modell der Markov Kette bietet eine große Flexibilität bezüglich der Übergangswahrscheinlichkeiten. So lassen sich feste Rekonfigurierungsabläufe (engl.: Schedule) mittels Übergangswahrscheinlichkeiten von 1 definieren (siehe Abbildung 5.4). Sind die Wahrscheinlichkeiten der Rekonfigurierungen während des Entwurfs noch nicht bekannt, kann auch eine Gleichverteilung der Rekonfigurierungen und Konfigurationen modelliert werden.

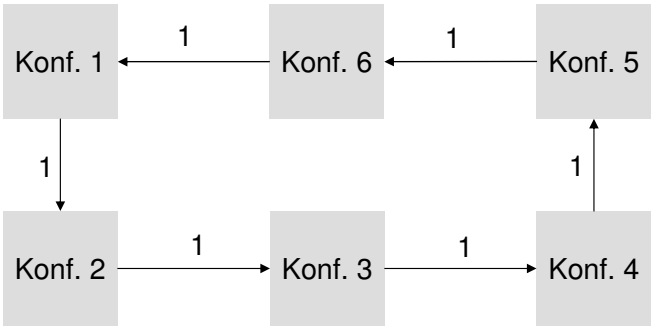


Abbildung 5.4: Beispiel für einen Konfigurationsgraph mit festem Schedule

Modulwahrscheinlichkeit

Welche Module mit hoher und welche mit niedriger Wahrscheinlichkeit im FPGA zur Verfügung stehen müssen, lässt sich aus der stationären Wahrscheinlichkeit von Konfigurationen bestimmen. Mit der Gleichung 5.10 wird die Modulwahrscheinlichkeit $\Gamma(mod)$ berechnet.

$$\Gamma(mod) = \sum_{konf \in Konf} \Pi_{konf} \cdot \begin{cases} 1 & mod \in konf \\ 0 & mod \notin konf \end{cases} \tag{5.10}$$

$\Gamma(mod)$ ist die Summe über die stationären Wahrscheinlichkeiten Π_{konf} aller Konfigurationen $konf \in Konf$, zu welchen das Modul mod benötigt wird. Wird ein Modul mit einer Wahrscheinlichkeit von 0 bestimmt, kann dieses aus dem System entfernt werden, da es in keiner Konfiguration enthalten ist.

FPGA Ressource und Slotgraph

Die Zielarchitektur für das rekonfigurierbare System ist ein FPGA. Um die Slots auf diese Architektur platzieren zu können, muss die FPGA Fläche formal beschrieben werden. Durch die regelmäßige FPGA Struktur können Spalten sp und Zeilen ze von CLBs unterschieden werden. In Gleichung 5.11 ist die Definition für ein FPGA mit Spalten und Zeilen angegeben.

$$FPGA = (sp, ze) : sp, ze \in \mathbb{N} \tag{5.11}$$

Auf einem FPGA wird eine Menge von Slots $Slot$ platziert. Ein Slot ist eine rechteckige Fläche. Die Position der linken, oberen Ecke wird durch die Parameter x_0 (CLB Spalte des FPGAs) und y_0 (CLB Zeile des FPGAs) bestimmt. Jeder Slot hat eine Höhe ho und Breite br, die in CLBs angegeben wird. Die durch einen Slot beschriebenen CLBs können in zwei verschiedene Arten unterteilt werden. Die, in den meisten Fällen, größere Menge $Logik$[3] wird für die Gatterabbildung von Modulen auf Slots verwendet. Der kleinere Teil, Menge Kom, ist für die Anschlussenden der Bus Makros und gegebenenfalls Multiplexern reserviert. Die

[3]Die Funktion $Logik(slot_i)$ gibt die Anzahl CLBs des Slots i an, die für Modullogik genutzt werden kann.

Gleichung 5.12 gibt die Definition eines Slots an.

$$slot = (x_0, y_0, ho, br, Logik, Kom) \tag{5.12}$$

$$Logik(slot) + Kom(slot) \leq ho(slot) \cdot br(slot) \tag{5.13}$$

$$\sum_{slot \in Slot} ho(slot) \cdot br(slot) \leq sp \cdot ze \tag{5.14}$$

Bei der Platzierung der Slots auf einem FPGA dürfen sich die Slots nicht überlappen. Die Summe der CLBs für die Module $Logik(slot)$ und für die Kommunikation $Kom(slot)$ muss kleiner gleich der durch $ho(slot)$ und $br(slot)$ beschriebenen Menge sein (siehe Gleichung 5.13). Ebenso sind die Slots nur auf dem FPGA platzierbar, wenn die Summe aller Slot Flächen kleiner gleich der FPGA Fläche ist (siehe Gleichung 5.14).

Für jede Konfiguration lässt sich nun eine Funktionen ϕ definieren, die Module auf die Menge der Slots abbildet.

$$\phi(konf, mod) : Konf \times Mod \rightarrow Slot \tag{5.15}$$

Die Funktion ϕ ist in Gleichung 5.15 beschrieben. Um ein System komplett auf einem FPGA abzubilden, muss für alle Module in jeder Konfigurationen die Funktion ϕ definiert sein (siehe Gleichung 5.16). Wie in Gleichung 5.17 angegeben, dürfen verschiedene Module in Verbindung mit einer Konfiguration nicht auf ein und denselben Slot abgebildet werden. Im Gegensatz dazu erlaubt die Funktion ϕ, dass zusätzlich zu den Modulen einer Konfiguration, Module auf freie Slots abgebildet werden können. Die Menge der zusätzlichen, auf Slots abgebildeten Module werden im Folgenden mit $konf^+$ bezeichnet (siehe Gleichung 5.19). Für alle Abbildungen muss gelten, dass die Module in den jeweiligen Slot passen (siehe Gleichung 5.18). In dieser Gleichung wird die Kommunikation vernachlässigt, da noch nicht alle Module bekannt sind, die auf einen Slot abgebildet werden. Die CLBs für die Kommunikation können erst abgeschätzt werden, wenn die maximale Anzahl der benötigten Slot-Verbindungen bekannt ist. Einem Konflikt wird über zusätzliche CLBs, die zu der Modulgröße als Puffer mit eingerechnet werden, entgegengewirkt.

$$\forall konf \in Konf \wedge \forall mod \in konf : \exists \phi(konf, mod) \tag{5.16}$$

$$\forall mod_a, mod_b \in Mod \wedge mod_a \neq mod_b : \phi(konf, mod_a) \neq \phi(konf, mod_b) \tag{5.17}$$

$$size_{mod} \leq Logik(\phi(konf, mod)) \tag{5.18}$$

$$konf^+ \subseteq Mod : \forall mod \in konf^+ \; \exists \phi(konf, mod) \wedge mod \notin konf \tag{5.19}$$

Für die Abbildung ϕ lässt sich auch eine Umkehrung ϕ^{-1} (siehe Gleichung 5.20) definieren.

$$\phi^{-1}(konf, slot) : Konf \times Slot \rightarrow Mod \tag{5.20}$$

Diese Funktion gibt in Abhängigkeit einer Konfiguration $konf$ an, ob ein Slot frei ist, beziehungsweise mit einem Modul, das zu dieser Konfiguration $konf$ oder zu $konf^+$ gehört, belegt ist.

Die Modulgröße $size_{mod}$ gemessen in Anzahl CLBs ist die Menge an CLBs die für die Implementierung benötigt wird. Einzelne IPs haben in ihrer Charakterisierung die Größe in CLBs angegeben. Wenn dies nicht der Fall ist, kann entweder eine einzelne IP oder ein komplettes Modul synthetisiert und die benötigten CLBs beispielsweise aus den Daten des *Synthesis Report* im Xilinx ISE Werkzeug bestimmt werden.

Ein statisches Modul ist für alle Konfigurationen auf dem FPGA zu platzieren. Mittels der Funktion ϕ kann ein Modul als statisch definiert werden (siehe Gleichung 5.21).

$$mod = \text{„}statisch\text{“} \leftrightarrow \forall konf_i, konf_j \in Konf : \phi(konf_i, mod) = \phi(konf_j, mod) \quad (5.21)$$

Bildet die Funktion ϕ ein Modul in allen Konfigurationen $konf$ auf den gleichen Slot ab, so ist das Modul „statisch“. Durch diese Definition wird auch die Menge der Slots in zwei Typen, statische Slots und dynamische Slots, unterteilt. Statische Slots sind Slots, auf die in allen Konfigurationen ein und dasselbe Modul abgebildet ist. Dynamische Slots hingegen sind Bereiche, auf die mehrere Module abgebildet werden. Die Platzierung der statischen Slots ist weitestgehend frei. Bei FPGAs die nur spaltenweise rekonfiguriert werden können, wie dem Virtex II Pro, sind die dynamischen Slots so zu platzieren, das komplette Spalten überdeckt werden.

Neben der Abbildung der Module auf Slots, sind auch die Kommunikationsverbindungen zwischen den Modulen auf Bus Makros abzubilden. Für diese Abbildung ist die Kardinalität β der Verbindungen zu berücksichtigen. Die Abbildungsfunktion θ in Gleichung 5.22 bildet von der Menge der Kommunikationsverbindungen KV auf die Menge der Bus Makros BM ab. Ein Bus Makro $bm \in BM$ implementiert mehrere, gerichtete Kanten zwischen zwei Slots (siehe Gleichung 5.23). Die Anzahl der Verbindungen wird über die Kardinalität γ angegeben.

$$\theta(konf, modkv) : Konf \times ModKV \rightarrow BM \quad (5.22)$$

$$bm = (slot_A, slot_Z, \gamma) \quad (5.23)$$

Die Anzahl der abgebildeten Verbindungen $modkv$ muss immer kleiner gleich der Anzahl der Bus Makros sein.

Die Menge der Slots $Slot$ und die Menge der Bus Makros BM definieren einen Slotgraphen SG, auf welchen der Modulgraph MG abzubilden ist. In Bild 5.5 ist eine solche Abbildung dargestellt. Die gestrichelten Kanten entsprechen der Funktion ϕ und die gepunkteten der Abbildung von $ModKV$ auf die Bus Makros BM.

Rekonfigurierungsdauer

Die Rekonfigurierungsdauer ist eine der zu minimierenden Kostenfaktoren. Für eine Rekonfigurierung werden mitunter mehrere Module ausgetauscht. Jedes Modul benötigt, in Abhängigkeit der Modul-Bitstream Größe, eine bestimmte Zeit um auf den FPGA geladen zu werden. Eine geeignete Abschätzung der Modulladezeit t_{mod} kann mit der linearen Abhängigkeit zur Größe der Module in Gleichungen 5.24 beschrieben werden.

$$t_{mod} = a \cdot size_{mod} + b \quad (5.24)$$

Die Modulladezeit kann sich von FPGA zu FPGA unterscheiden. Diese Tatsache wird mit unterschiedlichen Faktoren a berücksichtigt. Im Normalfall werden aber nur Module betrachtet, die auf einem FPGA geladen werden und daher ist eine relative Modulladezeit ausreichend. Bei einer relativen Modulladezeit, kann für den Faktor $a = 1$ angenommen werden. Die Konstante b verdeutlicht die Zeit, die von Anforderung an eine Rekonfigurierungssteuerung, ein Modul zu laden, vergeht, bis der Datenstrom an der Programmierschnittstelle anliegt. Diese Zeit ist unabhängig von der jeweiligen Modulgröße.

Die Größe eines Moduls ist die Größe der Bitstream Datei in kByte. Da zum Zeitpunkt der *Platzierung* diese noch nicht bekannt ist und auch die FPGA Ressource über CLB Spalten und

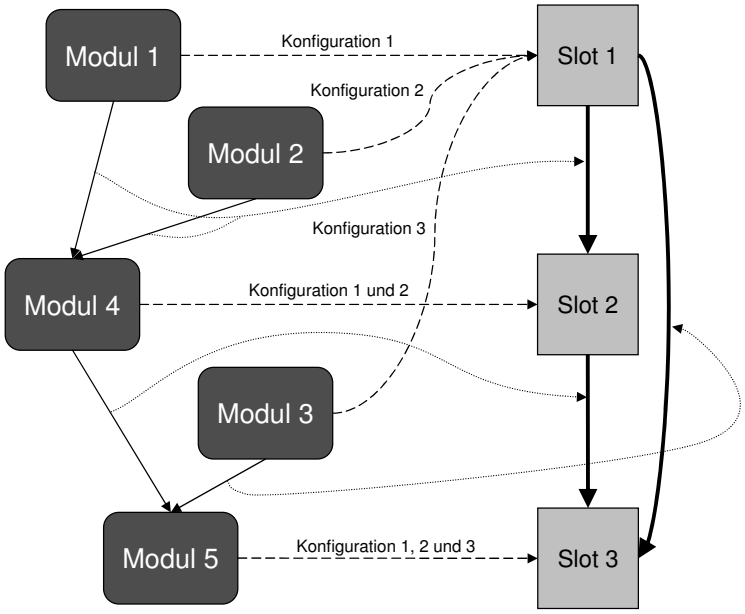


Abbildung 5.5: Beispiel für die Abbildung eines Modulgraphen auf einen Slotgraphen

Zeilen definiert ist, wird hier ebenfalls die Abschätzung, über die Anzahl an CLBs, die ein Modul belegen wird, vorgenommen.

Die Dauer einer Rekonfigurierung t_{rekonf} setzt sich aus allen Modulladezeiten t_{mod} der Module, die neu geladen werden müssen, zusammen.

$$t_{rekonf} = c + \sum_{mod \in konf_Z \vee konf_Z^+} t_{mod} \cdot \begin{cases} 1 & \phi(konf_A, mod) = \emptyset \\ 1 & \phi(konf_A, mod) \neq \phi(konf_Z, mod) \\ 0 & else \end{cases} \tag{5.25}$$

In der Gleichung 5.25 ist die Rekonfigurierungsdauer einer Rekonfigurierung angegeben. Die Konstante c ist gegenüber der Konstante b eine zusätzliche feste Zeit, die die Latenz der Steuereinheit zwischen den einzelnen Modulrekonfigurierungen darstellt. Es werden nur Modulladezeiten von Modulen aufaddiert, für die es keine Abbildung auf einen Slot in der Ausgangskonfiguration $konf_A$ gab, oder wenn das Modul in der Zielkonfiguration $konf_Z$ in einem anderen Slot als in der Ausgangskonfiguration abgebildet wird. Alle anderen Module der Zielkonfiguration und der zusätzlich abgebildeten Module $mod \in konf^+$ sind schon auf dem FPGA vorhanden und müssen nicht nachgeladen werden.

Die Modellierung der Rekonfigurierung als Markov Kette ermöglicht die Berechnung einer durchschnittliche Rekonfigurierungsdauer. Die durchschnittliche Rekonfigurierungsdauer T_{rekonf} ist die Summe über alle absoluten Rekonfigurierungswahrscheinlichkeiten P_{abs} multipliziert mit deren Rekonfigurierungsdauer (siehe Gleichung 5.26).

$$T_{rekonf} = \sum_{rekonf \in Rekonf} (\Pi_A \cdot p) \cdot t_{rekonf} \tag{5.26}$$

Eine Systemabbildung auf einem FPGA sollte insbesondere die durchschnittliche Rekonfigurierungsdauer minimieren. Dieses Optimierungsziel ist der Kernpunkt für die Algorithmen der

Slotbestimmung.

Eingabedaten der Slotbestimmung und Platzierung

Die Algorithmen für die Slotbestimmung und Slotplatzierung benötigen folgende Eingabedaten:

- Systembeschreibung mit Modulgraph MG und Konfigurationsgraph KG

  Für den Konfigurationsgraph sind die Übergangswahrscheinlichkeiten anzugeben und die Eigenschaften einer ergodischen, homogenen Markov Kette müssen eingehalten werden.

- Beschreibung der FPGA Größe als Zielarchitektur

- Funktionen zur Abschätzung von Systemeigenschaften: durchschnittliche Rekonfigurierungsdauer, benötigte CLBs für die Kommunikation und Bufferfläche $f(size_{mod}) = 1,05 \cdot size_{mod}$ für die Module

  Die zusätzlichen CLBs für die Module sind notwendig, damit die Synthesewerkzeuge optimalere Platzierungen erreichen können. Für eine Steuereinheit, der RCU, ist ein Bereich im FPGA zu reservieren. Da die RCU erst nach dem Entwurfsschritt der Platzierung generiert werden kann, ist eine geschätzte Menge an CLBs für die folgenden Algorithmen anzugeben.

  Am Beispiel des im Abschnitt eingeführten Musik-Players, werden kurz die benötigten Eingabedaten für die Algorithmen der Platzierung aufgelistet. Die Module des benötigten Modulgraphen MG und deren geschätzte Größe in CLBs, sowie die Systemelemente die zu den Modulen gehören, sind in Tabelle 5.1 aufgeführt. Ebenso ist die Fläche, die für die Rekonfigurierungssteuereinheit (RCU) auf dem FPGA zu reservieren ist, in der letzten Zeile der Tabelle angegeben. Die notwendigen Übergangswahrscheinlichkeiten des Konfigurationsgraphen KG sind als Annotationen an den Rekonfigurierungskanten in Abbildung 5.6 angegeben. Diese wurden exemplarisch festgelegt. Von allen Konfigurationen aus gibt es einen gerichteten Pfad zu jeder anderen Konfiguration, womit die geforderte Ergodizität für den Konfigurationsgraph gegeben ist. Ebenso kann man in der Abbildung 5.6 feststellen, dass die Summe der Wahrscheinlichkeiten aller Kanten die von einer Konfiguration ausgehen gleich 1 ist.

  Als Zielarchitektur für das Beispielsystem wurden drei verschiedene FPGAs der Firma Xilinx untersucht. Der Xilinx XC2V2000 [146], mit 48 CLB Spalten und 56 CLB Zeilen, ist groß genug, um das System mit Rekonfigurierung nach dem Overlaying Prinzip abbilden zu können. Aber die FPGA Fläche von 2688 CLBs ist zu klein, für eine Platzierung des kompletten Musik-Players. Damit wird Rekonfigurierung notwendig, denn um das System komplett platzieren zu können, werden ohne Berücksichtigung der CLBs für die Bus Makros, mindestens 3260 CLBs[4] benötigt. Weiterhin wurden die Algorithmen der *Platzierung* mit den FPGA Werten von XC2V1500 (48 x 40 CLBs) und XC2V3000 (64 x 56 CLBs), die ebenfalls zu klein sind für eine komplette Platzierung, getestet.

[4] Summe aus den Geschätzten Größen der Tabelle 5.1.

Systemelemente	Module	Größe
AAC Decoder	Modul AAC	400 CLBs
DAB Decoder DAB Menu DAB Menu View Controller Frequence Controller DAB	Modul DAB	610 CLBs
DRM Decoder DRM Menu DRM Menu View Controller Frequence Controller DRM	Modul DRM	600 CLBs
Filesystem	Modul Filesystem	130 CLBs
Idle Menu Idle Menu View Contoller	Modul Idle	150 CLBs
Display Driver Keyboard Driver Volume	Modul Keyboard-Display	150 CLBs
MP2 Decoder	Modul MP2	250 CLBs
MP3 Decoder	Modul MP3	300 CLBs
OGG Decoder	Modul OGG	320 CLBs
PL Menu View Controller Playlist Controller Playlist Menu	Modul Playlist	350 CLBs
RCU	*Modul RCU*	*80 CLBs*

Tabelle 5.1: Module des Beispiel Musik-Players

Zur Berechnung der durchschnittlichen Rekonfigurierungsdauer, wird die Modulladezeit, wie sie in Gleichung 5.27 angegeben ist, definiert. Diese Gleichung basiert auf der Annahme, dass die Rekonfigurierung über einen mit 33 MHz getakteten Bus, wie PCI (Peripheral Components Interconnection) oder CardBus [146], durchgeführt wird. Wie dem Datenblatt des FPGAs entnommen werden kann, entspricht ein CLB 128 Bit im Bitstream. Daraus ergibt sich eine Rekonfigurierungsdauer für 1 CLB: $1/(33MHz) \cdot 128 \approx 3.9\mu s$. Zusätzlich wird die Latenz, die die RCU benötigt, mit 20 μs abgeschätzt.

$$t_{mod} = (3.9 \cdot size_mod + 20)\mu s \tag{5.27}$$

Weiterhin wird für die Slotbestimmung eine Funktion zur Abschätzung der zusätzlich für Kommunikation verwendeten CLBs benötigt. Für ein Bus Makro mit einer Breite von 4 Bit wird je ein CLB an beiden Enden des Bus Makros benötigt. Daraus ergibt sich zur Bestimmung der CLBs $BMCLBs(\beta)$ für ein Kommunikationsverbindungsende von einem Modul aus mit der Bitbreite β die

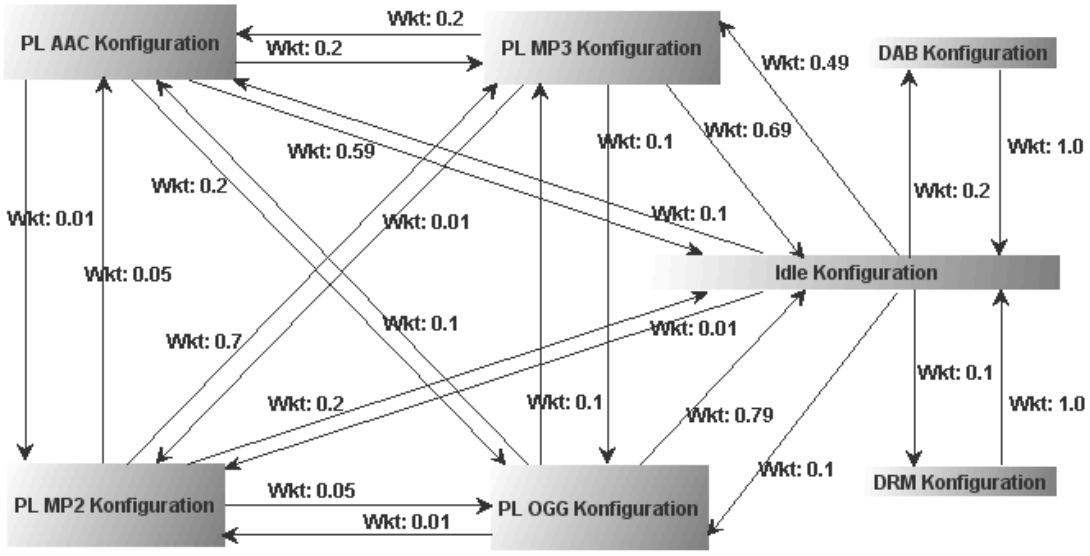


Abbildung 5.6: Konfigurationsgraph des Beispiel Musik-Players

Gleichung 5.28.

$$BMCLBs(\beta) = \left\lceil \frac{\beta}{4} \right\rceil \tag{5.28}$$

Da die Pufferfläche der Module hauptsächlich für die Synthese benötigt wird und das Verfahren zur Slotbestimmung und Platzierung analog mit kleineren bzw. größeren Modulen arbeitet, werden diese Pufferflächen hier nicht als Eingabedaten berücksichtigt. Für das Musik-Player Beispiel arbeitet das im folgenden Abschnitt beschriebene Verfahren daher direkt mit den in Tabelle 5.1 angegebenen Größen für die Module.

5.3.1 Slotbestimmung

Die im Rahmen dieser Arbeit entwickelte Methode der Platzierung teilt sich in zwei Schritte ein. In einem ersten Schritt werden, unter Berücksichtigung der durchschnittlichen Rekonfigurierungsdauer und in Anlehnung an das *Best-Fit Placement* [16], für alle Module passende Slots bestimmt.Im zweiten Teil der Methode werden diese Slots dann auf einem FPGA platziert.

Bei der Bestimmung des Slots können zur Optimierung der durchschnittlichen Rekonfigurierungsdauer, verschiedene Operationen durchgeführt werden. Durch die Optimierung werden die Module so platziert, dass sie selten ausgetauscht werden müssen. Eine wichtige Forderung an diese Operationen ist, dass die Abbildung von Modulen auf Slots konsistent gehalten wird. Dies setzt voraus, dass bevor die im Folgenden vorgestellten Operationen angewendet werden können, eine konsistente Abbildung vorliegen muss. Konsistent bedeutet, dass für jede Konfiguration und deren Module die Abbildung ϕ definiert ist. Mit drei dieser Operationen werden die Abbildungen ϕ variiert.

Modulaustausch Eine Operation ist der Modulaustausch. Bei dieser Operation werden in einer Konfiguration $konf_i$ die Abbildungen $\phi(konf_i, mod_j) = slot_m$ und $\phi(konf_i, mod_k) = slot_n$ so geändert, dass $\phi(konf_i, mod_j) = slot_n$ und $\phi(konf_i, mod_k) = slot_m$ ist. Damit tauschen die Module mod_j und mod_k den Slot. Dieser Modulaustausch kann, um die Konsistenz des Systems zu erhalten, nur durchgeführt werden, wenn $Logik(slot_n)$ und $Logik(slot_m)$ größer gleich $size_{mod_i}$ und $size_{mod_j}$ sind. Dies bedeutet, dass die Module in den jeweils anderen Slot passen. Die Konsistenz ist weiterhin gegeben, da sich die Anzahl der Funktionen $|\phi|$ durch diese Operation nicht ändert und die Abbildungen nur innerhalb einer Konfiguration getauscht werden. Für die Abbildungen ϕ der Tabelle 5.2 ist solch ein Austausch für die Module mod_3 und mod_4 in Konfiguration $konf_1$ durchgeführt worden. Das Ergebnis ist in Tabelle 5.3 zu sehen.

$Konf$ \ $Slot$	$slot_1$	$slot_2$	$slot_3$	$slot_4$	$slot_5$
$konf_1$	mod_1	mod_2	mod_3	mod_4	mod_5
$konf_2$	mod_1	mod_2	mod_4	mod_3	mod_6
$konf_3$	mod_1	mod_7	mod_8		mod_3
$konf_4$	mod_9	mod_2	mod_4	mod_5^+	mod_3
$konf_5$	mod_9		mod_8		mod_6

Tabelle 5.2: Belegungen von Slots in einem Beispielsystem

$Konf$ \ $Slot$	$slot_1$	$slot_2$	$slot_3$	$slot_4$	$slot_5$
$konf_1$	mod_1	mod_2	mod_4	mod_3	mod_5
$konf_2$	mod_1	mod_2	mod_4	mod_3	mod_6
$konf_3$	mod_1	mod_7	mod_8		mod_3
$konf_4$	mod_9	mod_2	mod_4	mod_5^+	mod_3
$konf_5$	mod_9		mod_8		mod_6

Tabelle 5.3: Belegungen von Slots nach Modulaustausch

Modulverschiebung Die Modulverschiebung ist eine weitere Operation zur Beeinflussung der Abbildungen ϕ. Hierbei wird ein neuer Slot einer Funktion $\phi(konf, mod)$ zugewiesen. Um diese Operation durchführen zu können, muss der neu zugewiesene Slot das Modul fassen können ($size_{mod} \leq Logik(slot_{neu})$). Ebenso ist es erforderlich, dass $\phi^{-1}(konf, slot_{neu}) = \emptyset$ oder $\phi^{-1}(konf, slot_{neu}) = mod \in konf^+$ gilt. Bei der zweiten Bedingung wird durch die Modulverschiebung die Kardinalität von ϕ um 1 reduziert, jedoch ist die Konsistenz weiterhin gegeben, da das Modul $mod \in konf^+$ für diese Konfiguration nicht benötigt wird (siehe Definition 5.19). In der Tabelle 5.3 können, unter der Annahme, dass die Forderung an die Größe des Slots $slot_4$ gegeben ist, das Modul mod_3 in Konfiguration $konf_3$ und $konf_4$ auf den Slot $slot_4$ verschoben werden. Das Modul mod_5 in Konfiguration $konf_4$ ist hierbei aus der Menge $konf^+$. In Tabelle 5.4 ist das Ergebnis der Modulverschiebungen abgebildet.

$Konf$ \ $Slot$	$slot_1$	$slot_2$	$slot_3$	$slot_4$	$slot_5$
$konf_1$	mod_1	mod_2	mod_4	mod_3	mod_5
$konf_2$	mod_1	mod_2	mod_4	mod_3	mod_6
$konf_3$	mod_1	mod_7	mod_8	mod_3	$\leftarrow$
$konf_4$	mod_9	mod_2	mod_4	mod_3	$\leftarrow$
$konf_5$	mod_9		mod_8		mod_6

Tabelle 5.4: Belegungen von Slots nach Modulverschiebung

Modulhinzufügung Die Modulhinzufügung ist eine dritte Operation, bei welcher Module die in einer Konfiguration nicht benötigt werden ($mod \in konf^+$) auf freie Slots $\phi^{-1}(konf, slot) = \emptyset$ abgebildet werden. Bei dieser neuen Abbildung ϕ muss ebenfalls gelten, dass das Modul in den freien Slot passt ($size_{mod} \leq Logik(slot_{frei})$). Das Ergebnis einer Modulhinzufügung, bei welcher das Modul mod_3 auf den Slot $slot_4$ in Konfiguration $konf_5$ abgebildet wurde, ist in Tabelle 5.5 zu sehen.

$Konf$ \ $Slot$	$slot_1$	$slot_2$	$slot_3$	$slot_4$	$slot_5$
$konf_1$	mod_1	mod_2	mod_4	mod_3	mod_5
$konf_2$	mod_1	mod_2	mod_4	mod_3	mod_6
$konf_3$	mod_1	mod_7	mod_8	mod_3	
$konf_4$	mod_9	mod_2	mod_4	mod_3	
$konf_5$	mod_9		mod_8	mod_3^+	mod_6

Tabelle 5.5: Belegungen von Slots nach Modulhinzufügung

Mit weiteren drei Operationen werden die Slotanzahl und deren Eigenschaften verändert.

Slothinzufügung Bei der Slothinzufügung wird die Menge der Slots um eins erhöht. Durch zusätzliche Slots lassen sich Module auf dem FPGA platzieren, die dann zur Menge $konf^+$ gehören. Diese Operation ist nur ausführbar, wenn auf dem FPGA Platz ist, um ein weiteres Modul platzieren zu können. Die FPGA Fläche abzüglich der Größe der Slots muss größer gleich der Modulgröße und der CLBs für ihre Kommunikationsanschlüssen sein (siehe Gleichung 5.29 und 5.30).

$$size_{frei} = sp \cdot ze - \sum_{slot \in Slot} ho(slot) \cdot br(slot) \tag{5.29}$$

$$size_{frei} \geq size_{mod} + Kom(slot_{neu}) \tag{5.30}$$

Slotverkleinerung Um für eine Slothinzufügung mehr Platz auf dem FPGA zu schaffen, kann eine Verkleinerung aller Slots auf eine Mindestgröße durch die Slotverkleinerung hilfreich sein. Die Mindestgröße eines Slots definiert sich aus dem größten Modul eines Slots (siehe Gleichung 5.31) und der Fläche für die Kommunikationsverbindungen $Kom(slot)$. Da die für

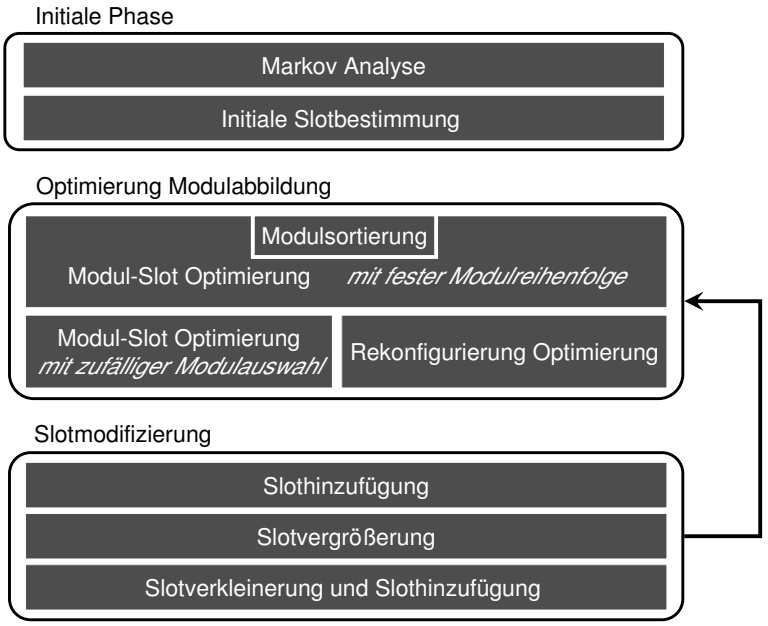


Abbildung 5.7: Schematische Darstellung der Slotbestimmung

diese Arbeit zu Verfügung stehenden FPGAs nur spaltenweise rekonfigurierbar sind, wird bei dieser Operation die Breite br der Slots minimiert (siehe Gleichung 5.32).

$$Logik(slot) = max\{size_{\phi^{-1}(konf_1,slot)}, ..., size_{\phi^{-1}(konf_n,slot)}\} : n = |Konf| \quad (5.31)$$

$$br(slot) = \left\lceil \frac{Logik(slot) + Kom(slot)}{ho} \right\rceil \quad (5.32)$$

Slotvergrößerung Die dritte Operation ist die Slotvergrößerung. Sie bewirkt das Gegenteil der Operation Slotverkleinerung. Durch eine Slotvergrößerung ergeben sich mehr Möglichkeiten für die Operationen Modulaustausch, Modulverschiebung und Modulhinzufügung. Alle statischen Slots werden von der Slotvergrößerung ausgeschlossen, da durch sie die durchschnittliche Rekonfigurierungsdauer nicht mehr beeinflusst wird. Die übrigen Slots werden nur soweit vergrößert, dass das nächstgrößere Modul gegenüber dem größten schon Platzierten abbildbar ist. Indem alle Slots nur minimal vergrößert werden ist die Wahrscheinlichkeit höher, dass alle nicht statischen Slots vergrößert werden können. Der Slot auf welchem das größte Modul abgebildet ist, wird ebenfalls von der Vergrößerung ausgeschlossen. Ist die Differenz zwischen alter und neuer Breite br eines Slots multipliziert mit der Höhe ho größer als die freie Fläche auf dem FPGA $size_{frei}$, kann die Slot Vergrößerung nicht durchgeführt werden.

Verfahren zur Slotbestimmung

Die vorgestellten Operationen werden nun in geeigneter Weise zur Minimierung der durchschnittlichen Rekonfigurierungsdauer angewendet. Dazu wurde ein Verfahren entwickelt, das schematisch in Abbildung 5.7 dargestellt ist. Das Verfahren teilt sich in drei Phasen ein. In der ersten Phase wird für alle Module eine initiale, konsistente Modul-Slot Abbildung bestimmt.

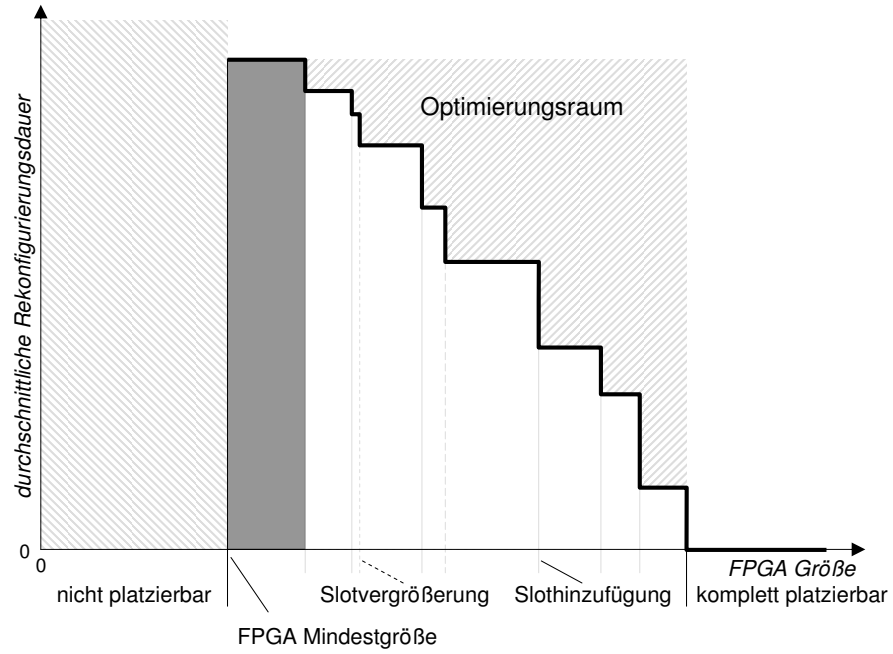


Abbildung 5.8: Zu erwartende Optimierungskurve für die Slotbestimmung

Diese Abbildungen werden dann, in der zweiten Phase, mittels der oben beschriebenen, ersten drei Operationen solange variiert, bis keine Verbesserung der durchschnittlichen Rekonfigurierungsdauer mehr ermittelt wird. Die abschließende dritte Phase *Slotmodifizierung* wird genutzt, um den Lösungsraum für die Abbildungen zu verändern. Dabei werden die Slotanzahl und -größe in Abhängigkeit der zur Verfügung stehenden FPGA Fläche geändert. Über die zweite Phase wird dann versucht die Rekonfigurierungsdauer neu zu optimieren. Können bessere Modul-Slot Abbildungen gefunden werden, wird die *Optimierung Modulabbildung* und *Slotmodifizierung* wiederholt, bis die Änderungen der Slots keine Verbesserung der durchschnittlichen Rekonfigurierungsdauer mehr bewirken. Dieses iterative Vorgehen ist in der Abbildung 5.7 durch den Pfeil angedeutet.

In der Abbildung 5.8 sind die zu erwartenden Auswirkungen auf die durchschnittliche Rekonfigurierungsdauer, im Verhältnis zur FPGA Fläche, durch die drei Phasen skizziert. Ist die FPGA Größe kleiner als eine Mindestgröße lässt sich keine durchschnittliche Rekonfigurierungsdauer bestimmen, da in diesem Fall mindestens eine Konfiguration nicht komplett, beziehungsweise konsistent, platzierbar ist. In der Abbildung 5.8 markiert der schraffierte Bereich (*nicht platzierbar*) diesen Fall. Die Mindestgröße wird durch die *Initiale Phase* bestimmt. Ab dieser FPGA Größe lässt sich auch die durchschnittliche Rekonfigurierungsdauer berechnen, die als fett markierte Stufenfunktion dargestellt ist. Die durchschnittliche Rekonfigurierungsdauer bleibt konstant, bis die Operation *Slothinzufügung* das erste Mal erfolgreich durchgeführt werden kann. Dieser Bereich ist in der Abbildung 5.8 dunkelgrau markiert. Mit zunehmender FPGA Größe können wiederholt sowohl die Operation *Slothinzufügung* als auch die *Slotvergrößerung* zur Minimierung der durchschnittlichen Rekonfigurierungsdauer beitragen. Die grau markierten FPGA Größen deuten diese Stellen an. Die Anwendung der Operationen wird zu einer Stufenfunktion im Optimierungsraum führen. Ziel der Slotbestimmung insgesamt ist es, durch die vor-

gestellten Operationen, die durchschnittliche Rekonfigurierungsdauer zu minimieren bzw. die Kurve soweit wie möglich zu „drücken“. Da hierbei ein sehr großer Lösungsraum durchsucht werden muss, kann das globale Optimum, angedeutet durch die Stufenfunktion, nicht garantiert werden. Es wird jedoch eine durchschnittliche Rekonfigurierungsdauer erwartet, die unterhalb der Oberkante des schraffierten Bereichs des Optimierungsraums ist. Voraussetzung hierfür ist eine FPGA Größe, die größer ist als der dunkelgrau markierte Bereich in der Abbildung, so dass mindestens ein zusätzlicher Slot platziert werden kann. Bei FPGAs die groß genug sind, um das komplette System zu platzieren (siehe in Abbildung 5.8 FPGA Größe *komplett platzierbar*), existiert trivialer Weise keine Optimierungsmöglichkeit, da keine Rekonfigurierungen notwendig werden. Der genaue Verlauf der Kurve wird vom jeweiligen System abhängig sein.

Initiale Phase Nach Bereitstellung der Eingabedaten für die Platzierung, wird als erstes eine *Markov Analyse* durchgeführt. Mit dieser Analyse werden aus den Übergangswahrscheinlichkeiten des Konfigurationsgraphen die stationären Wahrscheinlichkeiten der Konfigurationen Π_{konf} berechnet. Hierzu wird das Gleichungssystem 5.4 und 5.5 mittels der Gauss Eliminierung gelöst. Über die Gleichung 5.10 werden danach aus den Konfigurationswahrscheinlichkeiten die Modulwahrscheinlichkeiten $\Gamma(mod)$ bestimmt.

> Aus den Beispieleingabedaten des Musik-Players errechnet die Markov Analyse die in Tabelle 5.6 angegebenen stationären Wahrscheinlichkeiten für Konfigurationen. Mit diesen Werten ergeben sich für die Module des Beispielsystems die Wahrscheinlichkeiten der Tabelle 5.7. Wie zu erwarten ist die Wahrscheinlichkeit der Module die statisch sind, das Keyboard-Display und das RCU Modul, gleich 1.

$Konf$	Π_{konf}
DAB	0.085
DRM	0.043
Idle	0.426
PL AAC	0.100
PL MP2	0.016
PL MP3	0.241
PL OGG	0.088

Tabelle 5.6: Stationäre Zustandswahrscheinlichkeiten der Konfigurationen des Beispiel Musik Players

Die erste Phase wird abgeschlossen durch eine initiale Bestimmung der Slotanzahl und einer ersten Abbildung der Module auf diese Slots. Die Bestimmung der Slots ist im Algorithmus 1 beschrieben. Im Algorithmus wird nur die Fläche für die Modullogik berücksichtigt. Die zusätzlichen CLBs für die Bus Makro Anschlüsse sind vorerst nur bei Anwendung der Operationen Slotverkleinerung und Slothinzufügung relevant. Die initiale Lösung ist dadurch gekennzeichnet, dass nur so viele Slots $slot$ in der Menge $Slot$ vorhanden sind, wie es maximal Module mod in einer Konfiguration $konf$ gibt. Dies bedeutet, wenn ein Slot aus der Menge $Slot$ entfernt wird, kann mindestens eine Konfiguration nicht mehr auf den FPGA abgebildet werden. Analog zur Anzahl der Slots, ist auch die Größe $Logik(slot)$ der Slots nicht größer

Algorithmus 1: Initiale Slotbestimmung

Input : Menge der Konfigurationen $Konf$
Output : Menge der Slots $Slot$

$Slot \leftarrow \emptyset$;
$Slot_{markiert} \leftarrow \emptyset$;
foreach $konf \in Konf$ **do**
    sortiere alle Module $mod \in konf$ der Größe nach absteigend;
    verschiebe alle $slot \in Slot_{makiert}$ nach $Slot$;
    foreach $mod \in konf$ **do**
        $slot_{groeßer} \leftarrow$ kleinster Slot $slot \in Slot : Logik(slot) \geq size_{mod}$;
        $slot_{kleiner} \leftarrow$ größter Slot $slot \in Slot : Logik(slot) \leq size_{mod}$;
        if $slot_{groeßer} \notin Slot \wedge slot_{kleiner} \notin Slot$ **then**
            $Slot_{markiert} \leftarrow Slot_{markiert} \cup slot_{neu} : Logik(slot_{neu}) = size_{mod}$;
            $\phi(konf, mod) = slot_{neu}$;
        else if $slot_{groeßer} \notin Slot$ **then**
            verschiebe $slot_{kleiner}$ nach $Slot_{markiert}$;
            $Logik(slot_{kleiner}) = size_{mod}$;
            $\phi(konf, mod) = slot_{kleiner}$;
        else
            verschiebe $slot_{groeßer}$ nach $Slot_{markiert}$;
            $\phi(konf, mod) = slot_{groeßer}$;
        end
    end
    verschiebe alle $slot \in Slot_{makiert}$ nach $Slot$;
end

Mod	$\Gamma(mod)$
Modul AAC	0.143
Modul DAB	0.085
Modul DRM	0.043
Modul Filesystem	0.574
Modul Idle	0.426
Modul Keyboard-Display	1
Modul MP2	0.102
Modul MP3	0.241
Modul OGG	0.088
Modul Playlist	0.446
Modul RCU	1

Tabelle 5.7: Stationäre Zustandswahrscheinlichkeiten der Module des Beispiel Musik Players

als das größte Modul, welches diesem Slot zugewiesen ist. Mit dieser initialen Slotmenge $Slot$ ist auch der kleinste FPGA definiert, auf welchem das dynamisch rekonfigurierbare System läuft. Die FPGA Fläche $sp \cdot ze$ wird hierbei durch die Summe über alle Slothöhen ho mal die Slotbreiten br bestimmt.

Das Ergebnis der initialen Slotbestimmung und die Anzahl an $Logik$ CLBs der einzelnen Slots für den Beispiel Musik-Player sind in der Tabelle 5.8 angegeben. Es wurden fünf Slots bestimmt, die der Größe nach sortiert sind, da die abzubildenden Module der Größe nach platziert wurden (siehe Algorithmus 1). Platziert man die Slots auf die FPGA Fläche unter Berücksichtigung, dass nur komplette Spalten rekonfigurierbar sind, werden 1512 CLBs bzw. 27 CLB Spalten des XC2V2000 belegt. Bei Verwendung der bei den Eingabedaten spezifizierten Funktionen 5.27, ergibt sich für diese Slot Abbildungen eine durchschnittliche Rekonfigurierungsdauer von ca. 2.86 ms.

$Slot$ / $Konf$	$slot_1$	$slot_2$	$slot_3$	$slot_4$	$slot_5$
DAB	DAB	MP2	Keyb.-Disp.	Filesystem	RCU
DRM	DRM	AAC	Keyb.-Disp.	Filesystem	RCU
Idle	Keyb.-Disp.	Idle	RCU	-	-
PL AAC	Playlist	AAC	Keyb.-Disp.	Filesystem	RCU
PL MP2	MP2	Playlist	Keyb.-Disp.	Filesystem	RCU
PL MP3	MP3	Playlist	Keyb.-Disp.	Filesystem	RCU
PL OGG	OGG	Playlist	Keyb.-Disp.	Filesystem	RCU
$Logik(slot)$	610 CLB	400 CLB	150 CLB	130 CLB	80 CLB

Tabelle 5.8: Ergebnis der initialen Slotbestimmung für das Musik-Player Beispiel

Optimierung Modulabbildung In der zweiten Phase wird die initiale Lösung optimiert. Optimierung bedeutet in dieser Phase, dass die durchschnittliche Rekonfigurierungsdauer T_{rekonf} gegenüber der ersten Phase minimiert wird. Diese Phase ist durch die Annahme gekennzeichnet, dass Module, die eine hohe Modulwahrscheinlichkeit haben und zusätzlich eine hohe Rekonfigurierungsdauer aufweisen, einen stärkeren Einfluss bei der Optimierung haben als Module, die selten benötigt werden und eine kurze Rekonfigurierungsdauer haben. Mit dieser Annahme lässt sich eine Rangfolge für die Module aufstellen (siehe Gleichung 5.33).

$$Rang(mod) = \Gamma(mod) \cdot t_{mod} \tag{5.33}$$

Die Rangfolge wird durch die Methode *Modulsortierung* bestimmt. Entsprechend dieser wird, in dieser zweiten Phase, als erstes eine *Modul-Slot Optimierung* durchgeführt.

Modul-Slot Optimierung Der *Modul-Slot Optimierungs* Algorithmus versucht so oft wie möglich ein Modul mod genau auf einen, passenden Slot $slot$ abzubilden ($\phi(konf, mod) = slot$). Dadurch wird bewirkt, dass dieses Modul mod auch nur einmal im externen Modulspeicher vorgehalten werden muss. Kann bei allen Konfigurationen dieses Modul auf einen Slot abgebildet werden, wird das Modul statisch. Zudem reduziert sich die durchschnittliche Rekonfigurierungsdauer, da in mehreren und im Fall eines statischen Slots, in allen Konfigurationen das Modul schon geladen ist. Bei diesem Algorithmus kommen die Operationen *Modulaustausch*, *Modulverschiebung* und *Modulhinzufügung* zum Einsatz. Das Ergebnis einer *Modul-Slot Optimierung* ist in Tabelle 5.5 ebenfalls zu erkennen. Das Modul mod_3 wurde dort auf den Slot $slot_4$ für alle Konfigurationen $konf_1, ..., konf_5$ abgebildet, wodurch $slot_4$ statisch wird. Nicht in allen Fällen wird ein Modul statisch, wie bei dem Modul mod_5 in Slot $slot_5$ der Tabelle 5.5 ersichtlich ist. Um das Modul mod_5 durch eine Modulverschiebung und -hinzufügung auf Slot $slot_5$ statisch abzubilden, müsste insbesondere in der Konfiguration $Konf_2$ das Modul mod_6 verschoben werden. Dies ist jedoch nicht möglich, da in dieser Konfiguration kein weiterer freier Slot vorhanden ist. Wenn auch das Modul mod_5 in den Konfigurationen, $konf_3$, $konf_4$ und $konf_5$ zusätzlich auf den Slot $Slot_5$ abgebildet werden kann, so ist es doch für die Konfiguration $konf_2$ nicht möglich.

Der *Modul-Slot Optimierungs* Algorithmus startet in der Phase *Optimierung Modulabbildung* mit einem nicht statischen Modul mod, das den höchsten Rang hat und einem nicht statischen Slot, der am besten zu Modul mod passt. Der passende Slot für ein Modul kann über die Slotqualität $q(slot, mod)$, die in Gleichung 5.34 definiert ist, bestimmt werden.

$$q(slot, mod) = \sum_{konf \in Konf} \Pi_{konf} \cdot \begin{cases} 1 & \phi^{-1}(konf, slot) = mod \\ 0 & else \end{cases} \tag{5.34}$$

Die Slotqualität eines Slots $slot$ bezüglich eines Moduls mod ist umso höher, je öfter[5] das Modul auf $slot$ abgebildet ist. Wird ein Modul in keiner Konfiguration auf dem Slot $slot$ abgebildet, so ist die Slotqualität für diese Slot-Modul Abbildung gleich 0. Wenn die *Modul-Slot Optimierung* für das Modul mit dem höchsten Rang abgeschlossen ist, wird die Abbildung des nächst geringeren Moduls in der Rangfolge durch den Algorithmus optimiert, bis alle Modulabbildungen betrachtet wurden.

Wie oben schon erwähnt, kann kein optimales Ergebnis durch die *Modul-Slot Optimierung* garantiert werden. Um ein besseres Ergebnis im Lösungsraum zu finden, können Verfahren

[5] In mehreren Konfigurationen

angewandt werden, die zufällig das Ergebnis variieren. Im Fall der *Modul-Slot Optimierung* könnte *Hill Climbing* eingesetzt werden. Problematisch ist dieses Verfahren bei lokalen Minima bzw. Maxima (siehe [98]). Eine Erweiterung des Hill Climbing ist das *Simulated Annealing* [63]. Dieses Verfahren ist an den Abkühlungsprozess von erhitzten Metallen oder Gläsern angelehnt. Nach dem Erhitzen des Materials sorgt eine langsame Abkühlung dafür, dass die Atome ausreichend Zeit haben, sich in regelmäßige und damit stabile Strukturen anzuordnen. Die Temperaturfunktion des Verfahrens entspricht der Wahrscheinlichkeit, mit der sich ein Zwischenergebnis der Optimierung, verschlechtern darf. Dadurch ist es möglich auch ein lokales Optimum wieder zu verlassen. Ein ähnlicher Ansatz ist das *Threshold Accepting* (Schwellenakzeptanz), welches zuerst in der Veröffentlichung [27] präsentiert wurde und hier für die Slotbestimmung eingesetzt wird. Der Vorteil gegenüber dem *Simulated Annealing* ist, dass der Lösungsraum, im Hinblick auf die Laufzeit, schneller durchsucht wird. Ebenso deuten Experimente der Erfinder darauf hin, dass dieses Verfahren unter vergleichbaren Bedingungen häufig qualitativ bessere Lösung liefert als *Simulated Annealing*.

Das *Threshold Accepting* wird nun verwendet, um zufällig ein nicht statisches Modul und ein nicht statischen Slot auszuwählen, die dann, mittels des *Modul-Slot Optimierungs* Algorithmus, aufeinander abgebildet werden. Führt diese Optimierung zu einer besseren, durchschnittlichen Rekonfigurierungsdauer T_{rekonf}, wird das Ergebnis, die neue Referenzlösung und die *Modul-Slot Optimierung* mit einem neuen Modul und einem neuen Slot erneut ausgeführt. Das Verfahren wird spätestens nach einer vorher festgelegten Anzahl von Probierschritten beendet. Als geeignete Anzahl an Probierschritten l hat sich die doppelte Menge der möglichen, dynamischen Module-Slot Paare erwiesen. Wird durch den *Modul-Slot Optimierungs* Algorithmus eine schlechtere Lösung als die Referenzlösung erzeugt, wird mit dieser weitergearbeitet, falls die Verschlechterung unterhalb eines Schwellwertes liegt. Im Laufe des Verfahrens wird dieser Schwellwert sukzessive auf 0 reduziert. Bei der, im Rahmen dieser Arbeit, implementierten Variante wurde der Anfangsschwellwert auf $\frac{2 \cdot T_{rekonf}}{l}$ gesetzt. Je nach Größe des zu entwerfenden, eingebetteten Systems kann eine andere Anzahl an Probierschritten und eine Variation des Schwellwertes sich als sinnvoll erweisen. An dieser Stelle soll nicht näher darauf eingegangen werden, da sich durch die Anpassung dieser beiden Werte nur geringe Veränderung der Laufzeit des Verfahrens ergeben haben.

Falls die Anzahl der nicht statischen Module multipliziert mit der Anzahl der nicht statischen Slots größer ist als die Anzahl der Rekonfigurierungsmöglichkeiten, wird nicht die *Modul-Slot Optimierung* durchgeführt, sondern die *Rekonfigurierung Optimierung* (siehe Abbildung 5.7).

Rekonfigurierung Optimierung Der Unterschied zur *Modul-Slot Optimierung* ist, dass nicht alle Konfigurationen betrachtet werden, sondern nur die Rekonfigurierung zwischen zwei Konfigurationen. Aus der Menge der Module die zu diesen beiden Konfigurationen gehören, wird in diesem Verfahren zufällig eins ausgewählt und nach Möglichkeit in beiden Konfigurationen auf den Slot mit der höchsten Qualität $q(slot, mod)$ abgebildet. Ein Beispiel für das Ergebnis einer *Rekonfigurierung Optimierung* ist in Tabelle 5.3 zu sehen. Dort wurden die Konfiguration $konf_1$ und $konf_2$ betrachtet und das Modul mod_3 zur Optimierung ausgewählt. Durch einen Modulaustausch konnte die Rekonfigurierungsdauer zwischen den Konfigurationen $konf_1$ und $konf_2$ reduziert werden, da nur noch die Module mod_5 beziehungsweise mod_6 geladen werden müssen.

Ist die Bedingung zur Ausführung der *Rekonfigurierung Optimierung* erfüllt, wird über das *Threshold Accepting* Verfahren zufällig eine der Rekonfigurierungen $rekonf$ zur Optimierung bestimmt.

Die *Optimierung Modulabbildung* bewirkt für das Musik-Player Beispiel eine Verkürzung der durchschnittlichen Rekonfigurierungsdauer von 2,86 ms auf ca. 1,47 ms[6]. Dies ist eine Reduktion der durchschnittlichen Rekonfigurierungsdauer auf 51,4%. Der erste Schritt *Modul-Slot Optimierung* mit fester Modulreihenfolge bewirkte hierbei eine Verbesserung von 1,125 ms. Dies wurde erreicht, in dem unter anderem, die Module *Keyboard-Display*, *Filesystem* und *RCU* statisch auf die Slots $slot_3$, $slot_4$ und $slot_5$ abgebildet wurden. In Tabelle 5.9 sind alle Module markiert, deren Abbildung gegenüber der initialen Slotbestimmung verändert wurde. Ausgehend von diesem Zwischenergebnis ergab, nach 32 Itera-

$Konf$ \ $Slot$	$slot_1$	$slot_2$	$slot_3$	$slot_4$	$slot_5$
DAB	DAB	MP2	Keyb.-Disp.	Filesystem	RCU
DRM	DRM	AAC	Keyb.-Disp.	Filesystem	RCU
Idle	Idle	-	Keyb.-Disp.	Filesystem$^+$	RCU
PL AAC	AAC	Playlist	Keyb.-Disp.	Filesystem	RCU
PL MP2	Playlist	MP2	Keyb.-Disp.	Filesystem	RCU
PL MP3	MP3	Playlist	Keyb.-Disp.	Filesystem	RCU
PL OGG	OGG	Playlist	Keyb.-Disp.	Filesystem	RCU

Tabelle 5.9: Ergebnis der *Modul-Slot Optimierung* mit fester Modulreihenfolge für das Musik-Player Beispiel

tionen, die *Modul-Slot Optimierung* mit zufälliger Auswahl der Module über das *Threshold Accepting* Verfahren eine weitere Minimierung der Rekonfigurierungsdauer T_{rekonf} von 0,271 ms. Es wurde nicht die *Rekonfigurierung Optimierung* durchgeführt, da die Anzahl aller dynamischen Slots (2 Stück) multipliziert mit allen dynamischen Modulen (8) kleiner war als die 24 möglichen Rekonfigurierungen des Musik-Player Beispiels. Das Ergebnis der *Modulen-Slot Optimierung* mit *Threshold Accepting* ist in Tabelle 5.10 zu sehen.

Slotmodifizierung Bei der *Optimierung der Modulabbildung*, der zweiten Phase des Slotbestimmungsverfahren, wurden nur die in der Initialen Phase erzeugten Slots verwendet. In den meisten Fällen stehen jedoch FPGAs zur Verfügung, die mehr Platz bieten als von dem Ergebnissystem der Initialen Phase benötigt wird. Diese freie Fläche $size_{frei}$ wird in der Phase *Slotmodifizierung* für zusätzliche Slots und Vergrößerung der bestehenden verwendet (siehe Abbildung 5.7).

In einem ersten Schritt wird die Slotmenge $Slot$ um so viele Slots wie möglich durch die Operation *Slothinzufügung* erweitert. Hierzu wird als erstes der freie Platz $size_{frei}$ im FPGA berechnet und die Liste der nicht statischen Module der Größe nach absteigend sortiert[7]. Danach erfolgt die eigentliche Slot Hinzufügung. Für jedes nicht statische Modul, angefangen vom

[6] Alle berechneten Werte dieses Beispiels basieren auf der Gleichung 5.27.

[7] Im Fall der Slot Hinzufügung ist es nicht notwendig die Modulwahrscheinlichkeit zu berücksichtigen, da sich nur an der Größe eines Moduls entscheidet, ob dieses auf einen Slot abbildbar ist. Kleine Module können auch auf große Slots abgebildet werden.

$Slot$ / $Konf$	$slot_1$	$slot_2$	$slot_3$	$slot_4$	$slot_5$
DAB	DAB	MP2	Keyb.-Disp.	Filesystem	RCU
DRM	DRM	AAC	Keyb.-Disp.	Filesystem	RCU
Idle	Playlist$^+$	Idle	Keyb.-Disp.	Filesystem$^+$	RCU
PL AAC	Playlist	AAC	Keyb.-Disp.	Filesystem	RCU
PL MP2	Playlist	MP2	Keyb.-Disp.	Filesystem	RCU
PL MP3	Playlist	MP3	Keyb.-Disp.	Filesystem	RCU
PL OGG	Playlist	OGG	Keyb.-Disp.	Filesystem	RCU

Tabelle 5.10: Ergebnis der *Modulen-Slot Optimierung* mit *Threshold Accepting* für das Musik-Player Beispiel

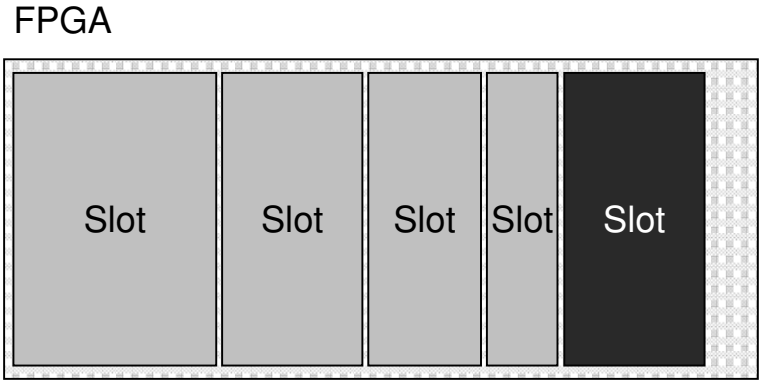


Abbildung 5.9: Schematische Darstellung der Slotbestimmung

größten Modul, wird geprüft, ob das Modul auf die freie FPGA Fläche platziert werden kann. Ist der freie Platz ausreichend für das aktuell betrachtete Modul, wird ein neuer Slot, der genau so groß ist, dass auf ihm dieses Modul abgebildet werden kann, der Slotmenge $Slot$ hinzugefügt und der restliche freie Platz $size_{frei}$ neu berechnet. Im Falle, dass der Platz nicht ausreichend ist, wird das nächst kleinere Modul betrachtet. Ist die *Slothinzufügung* abgeschlossen und wurde mindestens ein Slot hinzugefügt, wird die Phase *Optimierung Modulabbildung* wiederholt. In Abbildung 5.9 ist schematisch ein mögliches Ergebnis der *Slothinzufügung* dargestellt.

Der zweite Schritt der *Slotmodifizierung* ist die *Slotvergrößerung*, die ausgeführt wird, wenn keine Slots hinzugefügt werden konnten. Durch die Slotvergrößerungen ergeben sich mehr Variationsmöglichkeiten für die Modulabbildungen. Um diese Möglichkeiten für die Optimierung der durchschnittlichen Rekonfigurierungsdauer auszunutzen, wird, wie schon nach dem ersten Schritt, die Phase *Optimierung Modulabbildung* wiederholt. Die Details der Operation *Slotvergrößerung* sind im vorhergehenden Abschnitt 5.3.1 beschrieben.

Nach der *Slotvergrößerung* und der anschließenden *Optimierung Modulabbildung* kann weder ein neuer Slot eingefügt noch ein Slot vergrößert werden. Es ist jedoch möglich, dass durch die *Optimierung Modulabbildung* kleine Module auf große Slots abgebildet wurden. Diese

Fragmentierung wird nun im dritten Schritt der *Slotmodifizierung* ausgenutzt, um weitere Slots einzufügen. Mit der oben beschriebenen Operation *Slotverkleinerung* wird versucht, genügend freien Platz im FPGA zu schaffen, damit danach die *Slothinzufügung* im ersten Schritt der *Slotmodifizierung* ausgeführt werden kann. Konnte mindestens ein Slot der Menge der Slots hinzugefügt werden, wird wie bei den vorhergehenden Schritten die zweite Phase der Slotbestimmung wiederholt. Die Slotbestimmung wird beendet, wenn keine der drei Schritte der Slotmodifizierung erfolgreich war.

Durch die Slotmodifizierung werden, im Beispiel des Musik-Players, der Menge der Slots $Slot$ drei weitere, der Größe 350 CLBs, 300 CLBs und 150 CLBs, hinzugefügt. Die nach der Slothinzufügung ausgeführte *Optimierung Modulabbildung* führte zu einer neuen durchschnittlichen Rekonfigurierungsdauer von 0.641 ms. Hierbei wurden die Module *Modul Idle*, *Modul MP3* und *Modul Playlist* auf die drei neuen Slots statisch abgebildet.

Nachdem keine weiteren Slots hinzugefügt werden konnten, sind nach einer ersten Ausführung der Slotvergrößerung drei Slots und nach einer weiteren Iteration nochmals ein Slot vergrößert worden. Eine Verbesserung der durchschnittlichen Rekonfigurierungsdauer, wurde durch die *Optimierung Modulabbildung* jedoch nicht erreicht. Ebenso konnte durch die Slotverkleinerung nicht genügend Platz auf dem FPGA, für eine weitere Slothinzufügung, bereitgestellt werden, womit das Verfahren der Slotbestimmung terminiert.

Für den FPGA XC2V2000 ermittelte die Slotbestimmung 8 Slots mit der in Tabelle 5.11 angegebenen Modul-Slot Abbildungen und eine durchschnittliche Rekonfigurierungsdauer von ca. 0.64 ms. Mit dem etwas größeren FPGA XC2V3000

$Slot$ / $Konf$	$slot_1$	$slot_2$	$slot_3$	$slot_4$	$slot_5$	$slot_6$	$slot_7$	$slot_8$
DAB	DAB	MP2	PL^{+a}	MP3^{+}	KD^{b}	Idle^{+}	FS^{c}	RCU
DRM	DRM	AAC	PL^{+}	MP3^{+}	KD	Idle^{+}	FS	RCU
Idle	DAB^{+}	-	PL^{+}	MP3^{+}	KD	Idle	FS^{+}	RCU
PL AAC	DAB^{+}	AAC	PL	MP3^{+}	KD	Idle^{+}	FS	RCU
PL MP2	DAB^{+}	MP2	PL	MP3^{+}	KD	Idle^{+}	FS	RCU
PL MP3	DAB^{+}	-	PL	MP3	KD	Idle^{+}	FS	RCU
PL OGG	DAB^{+}	OGG	PL	MP3^{+}	KD	Idle^{+}	FS	RCU

[a] Modul Playlist
[b] Modul Keyboard-Display
[c] Modul Filesystem

Tabelle 5.11: Ergebnis des Slotbestimmungsverfahren für das Musik-Player Beispiel

können 10 Slots platziert werden, wodurch sich die Rekonfigurierungsdauer T_{rekonf} auf 0.22 ms reduziert. Bei dem XC2C1500 berechnete das Verfahren T_{rekonf} = 1,097 ms und bestimmte 6 Slots. Bei diesen berechneten Werten ist anzumerken, dass sie nicht das Optimum darstellen müssen. Wird das Verfahren zur Slotbestimmung mehrmals hintereinander ausgeführt, sind aufgrund des

Threshold Accepting Verfahrens, verschiedene Ergebnisse möglich. Im Beispiel des FPGA XC2V2000 kann mitunter das Modul AAC, statt des Moduls MP3, statisch auf einem Slot abgebildet sein. Aus diesen unterschiedlichen Abbildungsvarianten resultieren dann differierende Werte für die durchschnittliche Rekonfigurierungsdauer.

5.3.2 Slotplatzierung

Nach der Bestimmung, wie viele Slots platziert werden können und welche Module auf diese abzubilden sind, werden nun die Slots auf dem FPGA angeordnet. Ziel der Slotplatzierung ist eine vollständige Beschreibung der Slots $Slot$, so dass sie als Bounding Boxes im Constraint File für den weiteren Syntheseprozess eingetragen werden können. Des Weiteren wird, für eine automatische Generierung, in diesem Schritt die Anzahl, Länge und ungefähre Lage im FPGA der Bus Makros BM festgelegt. Die beiden Mengen $Slot$ und BM definieren zusammen einen Slotgraphen SG. Die Definition dieser Graphen wurde im Abschnitt 5.3 vorgestellt. Ein Beispielslotgraph ist auf der rechten Seite der Abbildung 5.5 zu sehen.

Die in diesem Abschnitt vorgestellten Schritte für die Platzierung der Slots auf einem FPGA sind für Schaltkreise ausgelegt, die spaltenweise rekonfigurierbar sind (siehe Abschnitt 2.2.2). Dies bedeutet jedoch keine Einschränkung für FPGAs, die diese Einschränkung nicht aufweisen, da sie ebenfalls spaltenweise rekonfiguriert werden können. Nicht auf spaltenweise Rekonfigurierung beschränke FPGAs, bieten unter Umständen bessere Voraussetzungen für die Modulsynthese, durch die frei definierbaren Slotrechtecke[8] und die Bus Makro Optimierung durch mehr Slotplatzierungsmöglichkeiten.

Kantenbestimmung des Slotgraphen

Die Kanten BM des Slotgraphen SG können erst nach Abschluss der *Slotbestimmung* und damit nach der Festlegung der Modul-Slot Abbildungen ϕ, ermittelt werden. Analog zu der Abbildung ϕ von Modulen einer Konfiguration auf Slots, sind die Kanten $ModKV$ der Module für die jeweilige Konfiguration $konf$ auf die Bus Makros BM abzubilden. Diese Kommunikationsabbildung θ wurde in Gleichung 5.22 definiert. Während bei der Abbildungsfunktion ϕ Module nur auf einen Slot abgebildet werden können, wenn $Logik(slot) \geq size_{mod}$ gilt, so muss bei θ die Kardinalität β der Kommunikationsverbindungen $ModKV$ kleiner gleich der Bitbreite γ des jeweiligen Bus Makros sein. Die Schritte zur Bestimmung der Bus Makros BM und der Abbildungen θ sind im Algorithmus 2 gegeben. Aus der Zeile drei des Algorithmus ist zu erkennen, dass nur Module, die zur aktuellen Konfiguration $konf$ gehören ($mod \in konf$), betrachtet werden. Dies ist korrekt, da das Konfigurationskonzept die Funktionalitäten beschreibt, die auf einem FPGA im jeweiligen Systemzustand funktionstüchtig abgebildet sein müssen. Zusätzliche Module müssen daher auch nicht mit Bus Makros verbunden sein. Wurde der Algorithmus ausgeführt, ist die Anzahl der benötigten Bus Makros bestimmt.

Der Slotgraph für das Musik-Player Beispiel ist in Abbildung 5.10 zu sehen. Die Annotationen der gerichteten Kanten zwischen den Slots, geben die Kardinalität

[8] Module die auf schmale Spalten abgebildet werden müssen, können unter Umständen nicht synthetisiert werden, wenn entfernt liegende, benötigte Ressourcen der Spalte nicht erreicht werden können. Insbesondere wenn lokal alle Kommunikationsmöglichkeiten belegt sind und CLB Spalten rechts beziehungsweise links des Slots genutzt werden müssten.

Algorithmus 2: Kantenbestimmung des Slotgraphen

Input : Menge der Konfigurationen $Konf$
Input : Modulgraph $MG = (Mod, ModKV)$
Input : Menge der Slots $Slot$
Output : Menge der Bus Makros BM

```
1  BM ← ∅;
2  foreach konf ∈ Konf do
3      foreach modkv = (mod_a, mod_b, β) : mod_a, mod_b ∈ konf ∧ mod_a ≠ mod_b
       do
4          slot_A ← φ^{-1}(konf, mod_a);
5          slot_Z ← φ^{-1}(konf, mod_b);
6          if bm = (slot_A, slot_Z, γ) ∈ BM then
7              γ(bm) = max{γ(bm), β(modkv)} ;
8          else
9              BM ← BM ∪ bm;
10             γ(bm) = β(modkv) ;
11         end
12         θ(konf, modkv) = bm;
13     end
14 end
```

γ der Bus Makros an. In diesem Beispiel sind keine Kanten von, beziehungsweise zur RCU hin, dargestellt. Dies ist aber nicht in jedem eingebetteten System der Fall, da auch Steuerleitungen, zum Beispiel für einen Modulreset, über Busmakros mit dynamischen Modulen verbunden werden müssen. Für die jeweiligen Slots sind die Module angegeben, die durch die *Slotbestimmung* auf diese abgebildet sind.

Slotanordnung

Nach Bestimmung der Bus Makro Anzahl wird im Folgenden die Slotanordnung definiert. Mit den konkreten Positionen der Slots im FPGA sind implizit auch Länge und ungefähre Lage der Bus Makros festgelegt. Eine triviale Variante die Slots zu platzieren, stellt sicherlich die zufällige Aneinanderreihung im FPGA dar. Dies bedeutet, dass man die Slots, wie in Abbildung 5.9 beispielhaft zu sehen ist, nebeneinander positioniert. Der dunkelgrau markierte Slot kann als der nächste anzuordnende, zufällig ausgewählte Slot, angesehen werden. In vielen Fällen wird sicherlich diese Art der Platzierung ein synthetisierbares System liefern.

In manchen Fällen müssen jedoch, wie auch schon im Abschnitt 5.2 angedeutet, zwei Punkte beachtet werden.

- Es gibt mehr kurze Kommunikationsverbindungen als *Long Lines* in einem FPGA. Werden zwischen zwei Slots viele Kommunikationsverbindungen benötigt, was einem Bus Makro mit einer hohen Kardinalität γ entspricht, sollten diese möglichst dicht nebeneinander auf dem FPGA platziert sein, um nicht in das Problem zu geraten, keine langen Verbindungen mehr für die Bus Makros zur Verfügung zu haben.

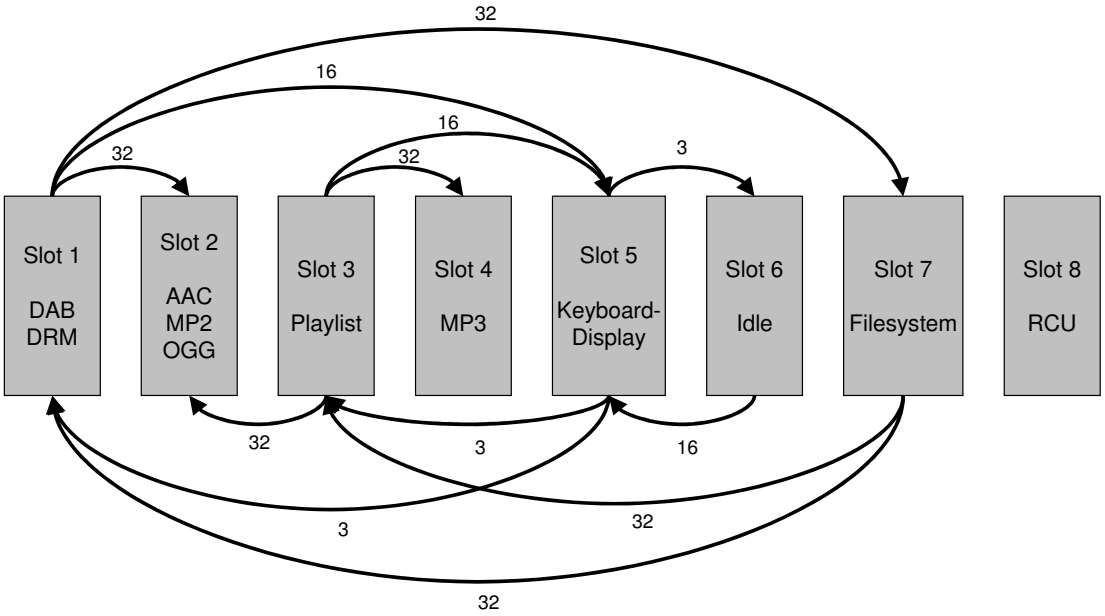


Abbildung 5.10: Schematische Darstellung des Slotgraph für das Musik-Player Beispiel

- Lange Kommunikationsverbindungen führen zu geringeren Taktraten im eingebetteten System. Um dieser Eigenschaft von Schaltkreisen Rechnung zu tragen, sind Slots, die mit Bus Makros verbunden sind, in nächster Nachbarschaft anzuordnen.

Für das zweite Problem ist jedoch anzumerken, dass eine einzelne Kommunikationsverbindung über eine größere Strecke im FPGA ausreicht, die Taktrate negativ zu beeinflussen. Es wird also nur bei wenigen, ganz speziellen Systemen möglich sein, alle langen Verbindungen zu eliminieren. Daher wird von einer kompletten Nichtverwendung der Long Lines im Folgenden abgesehen. Nur wenn jede Konfiguration eine unidirektionale Verarbeitungskette beinhaltet, besteht eine hohe Wahrscheinlichkeit, dass keine Long Lines verwendet werden müssen.

Die unterschiedliche Anzahl an Kommunikationsverbindungstypen, die im ersten Punkt angesprochen wurde, kann bei einer ungünstigen Slotanordnung zum Problem werden. Ziel muss es daher sein, aufgrund einer geeigneten Platzierung der Slots, wenig lange und viele kurze Bus Makros zu benötigen. Formal kann dieses Problem als eine Minimierung der Busmakrogesamtlänge BML definiert werden. Da die exakte Länge eines Bus Makros noch nicht bestimmt werden kann, wird an dieser Stelle das Längenmaß der Sloteinheit se eingeführt. Für ein Bus Makro, das benachbarte Slots verbindet, bedeutet dies eine Länge von $1se$. Befindet sich ein Slot zwischen diesen, mit einer Bus Makro Kante verbundenen Slots, ist die Länge gleich $2se$. Um die Gesamtlänge der Bus Makros zu bestimmen, muss eine Nachbarschaftsbeziehung zwischen den Slots existieren. Für diese Nachbarschafts Beziehungen kann eine Abbildung $Platz(slot)$ auf die natürlichen Zahlen genutzt werden (siehe Gleichungen 5.35). Die Busmakrogesamtlänge lässt sich dann, wie in Gleichungen 5.36 angegeben, definieren. Hierbei werden die einzelnen

Bus Makro Längen mit ihrer Bandbreite multipliziert und aufsummiert.

$$platz(slot) : Slot \rightarrow \mathbb{N} \tag{5.35}$$

$$BML = \sum_{bm=(slot_A, slot_Z, \gamma) \in BM} |platz(slot_A) - platz(slot_Z)| \cdot \gamma(bm) \tag{5.36}$$

Die Bestimmung der minimalen Bus Makro Länge erweist sich bei größer Slotanzahl n als aufwändig, da $n!$ Anordnungsmöglichkeiten verglichen werden müssen. Eine einfachere Möglichkeit ist dagegen, zufällig zwei Slots auszuwählen und deren Positionen in der Nachbarschaft zu vertauschen. Nach jedem Slotaustausch wird die Busmakrogesamtlänge neu berechnet und falls ein besserer Wert erreicht wurde, kann die neue Anordnung als Ausgangspunkt für weitere Optimierungen genutzt werden. Nach einer begrenzten Anzahl von Iterationen terminiert dieses Verfahren.

Mittels eines Greedy Algorithmus kann ebenfalls eine gute[9] Busmakrogesamtlänge erzielt werden. Bei diesem Algorithmus 3 wird als erstes der Slot in eine Slotliste $SlotList$ eingefügt, der die kleinste Valenz $deg(slot)$[10] besitzt. Als Nächstes wird das Element hinzugefügt, dass die aktuelle Busmakrogesamtlänge, die Valenz des ersten Slots $deg(SlotList[1])$[11], am geringsten erhöht. Für einen geeigneten nächsten Slot, kommen alle, die mit den vorhergehenden Slots der Liste durch Bus Makros verbunden sind und der mit der zweitgeringsten Valenz nicht verbundene Slot, in Frage. Im aufgezeigten Algorithmus werden jedoch alle nicht zur Liste $SlotList$ gehörenden Slots jeweils einer Testliste hinzugefügt und die entsprechende neue Busmakrogesamtlänge berechnet. Für die Berechnung der Busmakrogesamtlänge (siehe Algorithmus 4) werden die Positionen $platz(slot)$ der Slots die nicht zu $SlotList$ gehören auf die nächste freie Stelle der Liste gesetzt (siehe Zeile 2 in Algorithmus 4).

Dass in diesem Algorithmus verwendete Längenmaß se kann für eine noch bessere Verkürzung der Busmakrogesamtlänge präzisiert werden. Statt der Position der Slots in der Liste, kann bei der Berechnung der Faktoren (siehe Algorithmus 4 Zeile 7) die Breite der Slots berücksichtigt werden. Daraus würde sich ein genaueres Längenmaß in CLB Einheiten ergeben. Dadurch könnte berücksichtigt werden, wenn ein Bus Makro, das über einen großen Slot hinweg Slots verbindet, länger ist als ein Bus Makro, mit welchem mehrere schmale Slots überbrückt werden. Mit diesem neuen Längenmaß ändert sich die Struktur des Algorithmus jedoch nicht und wird daher im Folgenden nicht näher betrachtet. Die Anwendung des Greedy Algorithmus wird mit hoher Wahrscheinlichkeit keine Eliminierung der *Long Lines* zur Folge haben. Ebenso kann das Verfahren kein optimales Ergebnis garantieren, da theoretisch teilweise schlechtere Zwischenlösungen zu einem besseren Gesamtergebnis führen könnten. Dieses gilt auch für die Auswahl des ersten Slots in der Liste. Um ein eingebettetes System auf einem FPGA abbilden zu können, muss jedoch nicht die optimale Lösung für die Slotanordnung gefunden werden.

> Der Algorithmus 3 wurde auf das Musik-Player Beispiel angewandt. Die Busmakrogesamtlänge BML, des in Abbildung 5.10 dargestellten Slotgraphen beträgt $741se$ (Sloteinheiten). Das Ergebnis nach Anwendung des Greedy Verfahrens ist in Abbildung 5.11 zu sehen. Diese Anordnung der Slots bewirkt eine Verkürzung der Busmakrogesamtlänge auf $389se$. Als Startknoten wurde der Slot 8 ($slot_8$)

[9] Es wird nicht garantiert, dass die beste Lösung gefunden wird.

[10] Anzahl an eingehenden und ausgehenden Bus Makros eines Slots multipliziert mit den entsprechenden Kardinalitäten γ.

[11] Es sei angenommen, dass die Listen nicht mit dem Index 0 beginnen.

Algorithmus 3: Slotanordnung mit Greedy

Input : Slotgraph $SG = (Slot, BM)$
Output : Liste von Slots $SlotList$

```
1  SlotList ← leere Liste;
   /* Bestimmung des ersten Slot in der Liste          */
2  deg_min ← deg(slot ∈ Slot);
3  foreach slot ∈ Slot do
4      if deg_min > deg(slot) then
5          deg_min ← deg(slot);
6          slot_start ← slot;
7      end
8  end
9  SlotList ← add(slot_start);
   /* Bestimmung des nächsten Slots                    */
10 while Slot \ SlotList ≠ ∅ do
11     Δ_BML ← 0;
12     foreach slot ∈ Slot \ SlotList do
13         SlotListTest ← SlotList;
14         SlotListTest ← add(slot);
15         if Δ_BML = 0 then
16             Δ_BML ← BML(SlotListTest);
17             slot_next ← slot;
18         else if Δ_BML > BML(SlotListTest) then
19             Δ_BML ← BML(SlotListTest);
20             slot_next ← slot;
21         end
22     end
23     SlotList ← add(slot_next);
24 end
```

Algorithmus 4: Bestimmung der aktuellen Busmakrogesamtlänge BML

Input : Slotgraph $SG = (Slot, BM)$
Input : Liste von Slots $SlotList$
Output : $BML(SlotList)$ in se

```
1 foreach slot ∈ Slot \ SlotList do
2 |   platz(slot) ← length(SlotList) + 1
3 end
4 BML ← 0;
5 foreach bm = (slot_A, slot_Z, γ) ∈ BM do
6 |   if slot_A ∈ SlotList ∨ slot_B ∈ SlotList then
7 |   |   BML ← BML + (γ · |platz(slot_A) − platz(slot_Z)|);
8 |   end
9 end
```

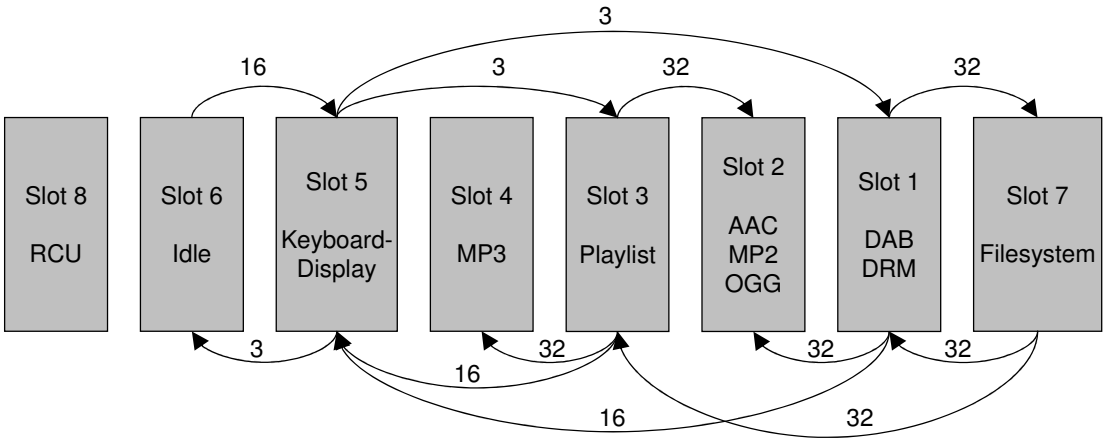


Abbildung 5.11: Mit Greedy definierte Anordnung der Slots für das Musik-Player Beispiel

ausgewählt, da dessen Valenz $deg(slot_8)$ gleich $0se$ ist. Den geringsten Zuwachs für die Busmakrogesamtlänge bewirkte als nächstes der Slot 6 mit einer Valenz von $deg(slot_6) = 19se$. Alle weiteren Zwischenwerte der BML sind: $57se$, $127se$, $210se$, $293se$ und $389se$. Die Busmakrogesamtlänge ändert sich beim letzten Schritt, dem Hinzufügen des Slots 7 zur Liste, nicht mehr, da keine neuen Bus Makros hinzukommen und bei der Berechnung von BML alle Kanten zu nicht platzierten Slots bis zum Ende der Liste verbunden werden.

Slotkoordinaten

Der letzte Schritt in der Platzierung ist die Extraktion der Slotkoordinaten. Wie in der Definition 5.12 angegeben, besteht ein Slot aus der linken oberen Ecke und einer entsprechenden Höhe und Breite. Nach der Anordnung der Slots in einer Liste (siehe Algorithmus 3) können die linken oberen Ecken der Slots bestimmt werden. Für den ersten Slot in der Liste ist $x_0 = 1$ und $y_0 = 1$

zu setzen[12]. Die nächste Position für x_0 errechnet sich aus der Position x_0 des Vorgängers und dessen Breite br (siehe Gleichung 5.37).

$$slot_{n+1}(x_0) = slot_n(x_0) + br_n \tag{5.37}$$

Bei FPGAs die nur spaltenweise rekonfiguriert werden können, ist der Wert y_0 für alle Slots gleich 0. Für das Beispiel des Musik-Players ist in Abbildung 5.11 die schematische Anordnung der Slotpositionen angegeben. Sind alle Slotkoordinaten bestimmt, ist der Syntheseschritt der Platzierung abgeschlossen.

Sowohl für die Slot Anordnung wie auch für die Extraktion der Koordinaten sei noch angemerkt, dass die Slots in dynamische und statische unterschieden werden können. Statische Slots, die ein Modul während der ganzen Laufzeit des Systems beinhalten und daher nicht rekonfiguriert werden, müssen auch nicht die komplette FPGA Höhe belegen. Man könnte versuchen mehrere statische Slots in einem gemeinsamen Bereich frei anzuordnen oder die Module mehrerer statischer Slots in einen großen Slot zusammenzufassen. Mit einer freien Anordnung mehrerer Slots in einem größeren Bereich könnte die Busmakrogesamtlänge weiter minimiert werden. Die erwartete Verbesserung der Busmakrogesamtlänge ist minimal gegenüber der oben beschriebenen Aneinanderreihung im FPGA. Diese Variabilität der Slot Anordnung wurde daher im Kontext dieser Arbeit nicht weiter untersucht. Wenn mehrere Module aus statischen Slots auf einen gemeinsamen größeren Slot abgebildet werden, könnten sich Vorteile bei der Synthese zeigen. In diesem Fall werden die Optimierungsmöglichkeiten der Synthese besser ausgenutzt. Prinzipiell ist dann auch eine Einsparung von Bus Makros möglich.

> Würde man zum Beispiel die statischen Module MP3 und Playlist des Musik-Players auf einen Slot abbilden, könnten die Bus Makros zwischen den beiden Slots, die diese Module beinhalten, entfernt werden (siehe Slot 3 und 4 in Abbildung 5.11).

Nachteilig ist die Zusammenfassung von Modulen in einem Slot, für die in Abschnitt 4.3 beschriebenen Robustheitsaspekten. Module die redundant an verschiedenen Stellen auf dem FPGA platziert sein sollen, würden durch diesen Ansatz zusammengefasst werden. Da diese Änderungen der Slotanordnungen, durch das zu Grunde liegende Overlaying Konzept, jedoch keinen, beziehungsweise nur marginalen Einfluss auf das Verhalten des Systems haben, wurde von einer Evaluierung abgesehen.

5.3.3 Ergebnisse

Durch die Platzierung wurde ein vollständiger Slotgraph erzeugt. Alle diese generierten Daten wurden in der Implementierung auf XML Strukturen abgebildet, um leicht von den nachfolgenden Entwurfsschritte genutzt zu werden. Der Einsatz von XML erwies sich auch als vorteilhaft bei der Bewertung der entwickelten Methoden.

Für Tests wurden verschiedene zufällige Systeme generiert und mit den entwickelten Methoden optimiert. Diese Systeme bestanden aus m verschiedenen Modulen mit zufälliger Größe. Dazu wurden für jedes dieser Systeme n Konfigurationen definiert die aus drei bis acht Modulen bestanden. Die beiden Parameter m und n nahmen Werte zwischen 10 und 30, in allen Kombinationen an. Für jede dieser Kombinationen wurden mehrere, zufällige Varianten berechnet, um

[12] Diese Angabe mündet dann in der *Bounding Box* Definition : `AREA_GROUP <name> RANGE=CLB_Rm1Cn1:CLB_rm2Cn2;`

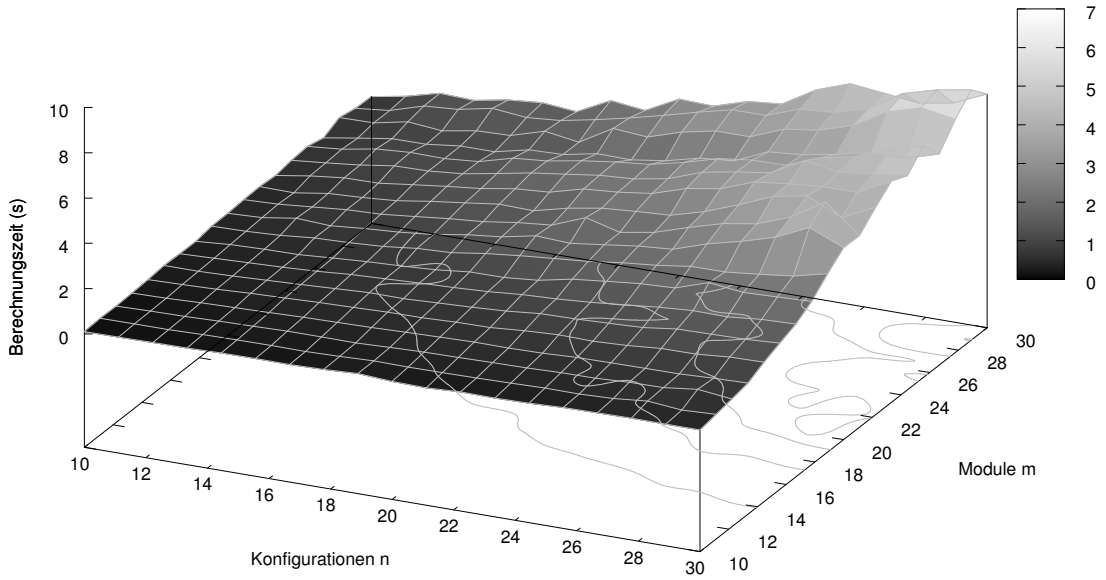


Abbildung 5.12: Berechnungszeiten für die Slotgraphbestimmung

den Durchschnitt der Berechnungszeit zu bestimmen. Für die Übergangswahrscheinlichkeiten zwischen den einzelnen Konfigurationen wurde eine Gleichverteilung angenommen. Die Anzahl der Rekonfigurierungen wurde ebenfalls variiert. In Abbildung 5.12 ist eine Grafik, mit den benötigten Zeiten für die Berechnung der Slotgraphen, dargestellt. Die Messungen haben ergeben, dass bei größeren Systemen, mit vielen Modulen und vielen Konfigurationen, die Berechnungszeit in den zweistelligen Sekundenbereich hineinreichen kann. Insgesamt wurden 94.211 Tests mit 7660 verschiedenen Systemen durchgeführt. Im Durchschnitt über allen Tests lag die Berechnungsdauer bei ca. 1,2 Sekunden. Alle Tests wurden auf einem Rechner mit einem 2 GHz AMD Athlon XP Prozessor und 640 MB RAM ausgeführt.

Neben den Berechnungszeiten für die Slotgraphenbestimmung wurde auch die durchschnittliche Rekonfigurierungsdauer analysiert. Hierfür wurde für das Musik-Player Beispiel die Slotbestimmung bei verschiedenen FPGA Größen durchgeführt. Als Eingabe wurden für die Tests FPGAs mit 46 CLB Zeilen und 1 bis 100 CLBs Spalten festgelegt. Für jede FPGA Größe wurde 1000 mal die Bestimmung der Slots durchgeführt, um zufällige Abweichungen der durchschnittliche Rekonfigurierungsdauer zu vermeiden. Diese Abweichungen sind zu erwarten, da bei dem Verfahren heuristische Optimierungsalgorithmen eingesetzt werden. Aus demselben Grund kann auch das globale Minimum der Rekonfigurierungsdauer nicht garantiert werden. Die Abbildung 5.13 zeigt die Ergebnisse der Slotbestimmung, die durchschnittliche Rekonfigurierungsdauer in Abhängigkeit der FPGA CLB Spalten, auf. Es wurden nur die kleinsten berechneten Werte pro FPGA Größe abgebildet. Ist der FPGA kleiner als 46*35 CLBs wird von der Slotbestimmung ermittelt, dass der Musik-Player nicht platziert werden kann. Für diese Fälle ist in der Abbildung 5.13 auch keine Rekonfigurierungsdauer angegeben. Betrachtet man die Modulgrößen ohne die CLBs für die Kommunikation,

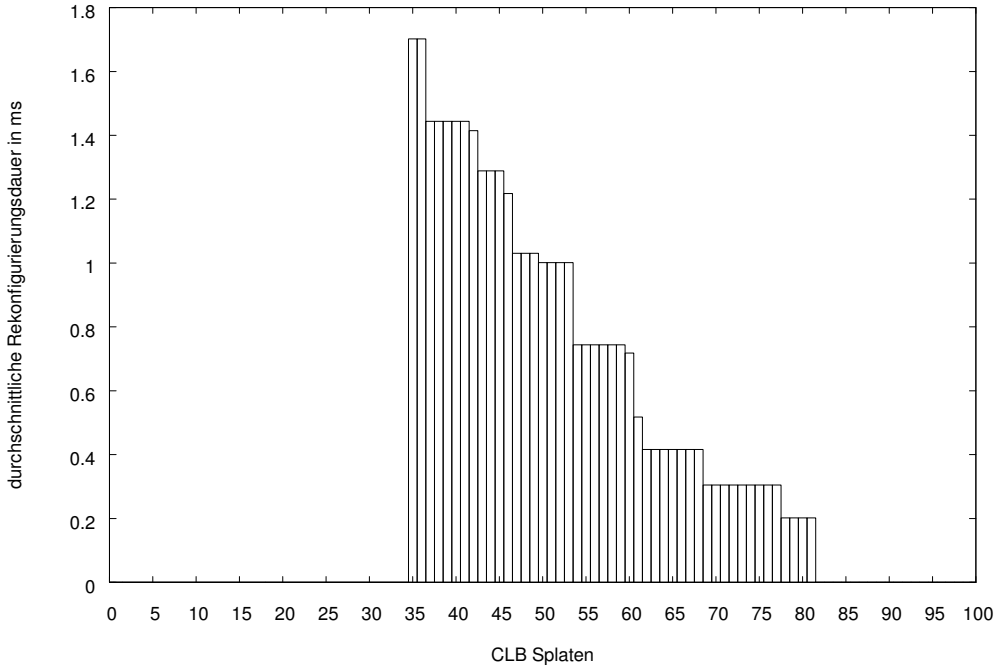


Abbildung 5.13: Ergebnisse der Slotbestimmung für das Musik-Player Beispiel

würden 30 Spalten ausreichen, da die „größte“ Konfiguration *DRM* 1360 CLBs benötigt. Wenn der FPGA genau 1360 CLBs groß wäre, könnte das Beispielsystem nicht platziert werden, auch wenn die Kommunikationsverbindungen nicht berücksichtigt würden. Der Grund dafür ist die Abbildung der Module *DAB* und *DRM* auf den gleichen Slot (vergleiche Tabelle 5.8).

Ab einer FPGA Größe von 1610 CLBs[13] können alle Konfigurationen des Musik-Players platziert werden. Die durchschnittliche Rekonfigurierungsdauer beträgt hierbei ca. 1,7 ms.Mit wachsender Größe des FPGAs nimmt auch die durchschnittliche Rekonfigurierungsdauer ab, wenn weitere Slots platziert werden können oder die Slotvergrößerung eine bessere Modul Slot Abbildung ermöglicht. Bei einer Größe des FPGA von 46*82 CLBs kann der komplette Musik-Player auf dem FPGA platziert werden, was eine durchschnittliche Rekonfigurierungsdauer von 0 ms bedeutet.

Die empirisch berechneten Beisieldaten des Musik-Players bestätigen, dass, wie im Abschnitt 5.3.1 prognostiziert, die durchschnittliche Rekonfigurierungsdauer einer Stufenfunktion entspricht. Unter Verwendung eines Standard FPGA XC2V2000 mit 48 CLB Spalten und 56 CLB Zeilen ermittelt das vorgestellte Verfahren für das Musik-Player Beispiel eine Verbesserung der durchschnittliche Rekonfigurierungsdauer von 62 % gegenüber den 1.7 ms bei der FPGA Mindestgröße. Aus der Abbildung 5.13 lässt sich des Weiteren der Vorteil von dynamischer Rekonfigurierung bezüglich der FPGA Größe erkennen. Das vorgestellte Rekonfi-

[13] $46 * 35 = 1610$

gurierungskonzept führt bei diesem Beispielsystem zu einer Einsparung der FPGA Fläche von 57 %.

5.4 Kommunikation

Mit dem Overlaying Ansatz und der freien Wahl an Kommunikationsmöglichkeiten für die Module ergeben sich Punkte, die sowohl bei der Generierung der Steuerungseinheit *Reconfiguration Control Unit* als auch beim *Top-Level Design*, berücksichtigt werden müssen. Kommunizieren Funktionalitäten mit mehreren verschiedenen Teilnehmern eines Systems, wird in den meisten Fällen ein Bussystem eingesetzt. Kann jedoch garantiert werden, dass nur eine bidirektionale Kommunikation in einem, durch zwei Rekonfigurierungen abgegrenzten, Zeitraum stattfindet, sind mehrere Eigenschaften, wie zum Beispiel die Adressverwaltung der Kommunikationsteilnehmer, von Bussen überflüssig. Stattdessen kann die Kommunikation über eine Punkt zu Punkt Verbindung realisiert werden. Dies ist genau dann der Fall, wenn mehrere Kommunikationsverbindungen $modkv$, die zu einem Modul mod_k hinführen oder davon ausgehen, jeweils nur bei einer Konfiguration genutzt werden. Zum Beispiel könnte ein Modul mod_k statisch sein und mit dynamischen Modulen mod_i und mod_j kommunizieren. Durch Rekonfigurierung wird dann zum nächsten dynamischen Kommunikationspartner gewechselt. In einem solchen Fall kann ein benötigtes Bus Makro für verschiedene Kanten des Modulgraphen genutzt werden. Wenn für unterschiedliche Konfigurationen $konf_a$ und $konf_b$ die Beziehung 5.38 gilt, muss zwischen verschiedenen Bus Makros mit Multiplexer umgeschaltet werden.

$$\begin{aligned} MUX \leftarrow \phi^{-1}(konf_a, mod_i) \neq \phi^{-1}(konf_b, mod_j) : konf_a \neq konf_b \wedge mod_i \neq mod_j \wedge \\ (\exists\, modkv_m = (mod_i, mod_k, \beta), modkv_n = (mod_j, mod_k, \beta) \vee \\ \exists\, modkv_m = (mod_k, mod_i, \beta), modkv_n = (mod_k, mod_j, \beta)) \end{aligned} \quad (5.38)$$

Die Gleichung 5.38 sagt aus, dass ein Multiplexer MUX benötigt wird, wenn die Module mod_i und mod_j, zweier unterschiedlicher Konfigurationen, die mit demselben Modul mod_k kommunizieren auf unterschiedliche Slots abgebildet sind. Die Notwendigkeit von Multiplexern lässt sich sehr gut an dem Musik-Player Beispiel zeigen. Wie in der Abbildung 4.1 zu sehen, ist fließen Daten von zwei verschiedenen Systemelementen, *Playlist Controller* und *DAB Decoder*, zum *MP2 Decoder*. Jede dieser beiden Verbindungen wird in genau einer Konfiguration genutzt.

Eine wichtige Voraussetzung, um die benötigten Multiplexer zu bestimmen, ist die Definition der Modul-Slot Abbildungen ϕ. Anhand der gegebenen Definition kann die Bestimmung der Multiplexer leicht automatisiert werden. Das Umschalten der Multiplexer wird, im späteren System, durch die RCU gesteuert. Die eigentliche Einbindung der Multiplexer erfolgt dann automatisch im Entwurfsschritt der *Top-Level Design* Generierung (siehe Abbildung 2.5). Von der technischen Realisierung her existiert keine Notwendigkeit die Multiplexer als Hard-Makro bereitzustellen.

Sollte es notwendig sein die Anzahl der Bus Makros weiter zu reduzieren, könnte bei der *Platzierung* die Anzahl der benötigten Multiplexer berücksichtigt werden. Eine Minimierung der Multiplexeranzahl hat jedoch eine Steigerung der durchschnittlichen Rekonfigurierungsdauer zur Folge, da zusätzliche statische Slots verhindert werden könnten. Aus diesem Grund wurde diese Minimierung im Kontext dieser Arbeit nicht weiter untersucht.

Um die Robustheit von eingebetteten, dynamisch rekonfigurierbaren Systemen zu steigern, kann es sinnvoll sein, redundant vorhandene Module oder auch Systemelemente, auf verschiede-

ne Slots abzubilden (siehe Abschnitt 4.3). Dies kann mit dem vorgestellten Design Flow leicht realisiert werden, indem ein Entwickler von vornherein Multiplexer in Form eines speziellen Systemelements definiert und in den Problemgraph integriert. Anhand dieser speziellen Systemelemente kann automatisch eine zusätzliche Konfiguration ermittelt werden, die bewirkt, dass die mit dem Multiplexer verbundenen Systemelemente gleichzeitig und damit auf verschiedenen Slots im FPGA platziert werden. Daraus ergibt sich quasi eine Umkehrung der obigen Definition 5.38.

Neben der automatischen Bestimmung von Multiplexern, können auch die Bus Makros automatisch generiert werden. Da die Bus Makros eine feste Vorgabe für die Synthese sind, können die Anschlüsse der Bus Makros relativ frei innerhalb eines Slots platziert sein. Für jede Kommunikationsverbindung zwischen Slots wird eine Bus Makro Komponente, die bei allen zu synthetisierenden Modulen benötigt wird, erzeugt. Jede Kommunikationsverbindung besteht aus Anfangs- und Endpunkt, die einem konkreten Element im FPGA entsprechen. In dem implementierten Verfahren wird daher eine Beschreibung[14] des hierarchischen Aufbaus vom Ziel FPGA mit all seinen Low-Level Komponenten eingelesen und alle potentiellen Anfangs- und Endpunkte herausgefiltert[15]. Für jeden Slot wird dann die benötigte Menge von Anschlüssen bestimmt und in einem weiteren Schritt mit den Anschlüssen der anderen Slots verbunden.

In diesem Abschnitt wurde die Notwendigkeit von Multiplexern und die Ausnutzung von Multiplexer für Robustheitsaspekte sowie die automatische Generierung von Bus Makros kurz vorgestellt.

5.5 Zusammenfassung

In diesem Kapitel wurden Konzepte zur Bestimmung der Overlaying Bereiche, den Slots, die für nachfolgenden Entwurfsschritte, *Generierung Kommunikationskanäle*, *Top Level Design* Erstellung und *Module Implementation*, benötigt werden, im Detail beleuchtet. Die drei Hauptprobleme der Platzierung, wie viele Slots werden benötigt, welche Module werden auf welche Slots abgebildet und wo werden die Slots im FPGA platziert, wurden in Abschnitt 5.1 dargestellt. Mit dem vorgestellten Verfahren für die Platzierung wurden drei Ziele verfolgt. Das erste Ziel beinhaltet die Minimierung der durchschnittlichen Rekonfigurierungsdauer durch eine geeignete Abbildung von Modulen auf Slots. Bei diesem Ziel ist die Menge der Rekonfigurierungsdaten zu berücksichtigen. Eine optimale Platzierung sollte weiterhin gekennzeichnet sein durch wenige lange und mehr kurze Kommunikationsverbindungen. Dieses Optimierungsziel wird im Besonderen durch eine geeignete Slotplatzierung erreicht. Die Minimierung des externen Bitstream Dateispeicher stellt das dritte Ziel der Platzierung dar. Im entwickelten Verfahren werden daher große Module, je nach Ressourcenverfügbarkeit des FPGAs, auf statische oder zu mindestens nicht auf mehrere verschiedene Slots abgebildet.

Der Hauptteil dieses Kapitels bilden die formale Definitionen und die Beschreibung des Platzierungsverfahrens (siehe Abschnitt 5.3). Für die Slotbestimmung 5.3.1 und die Berechnung der durchschnittlichen Rekonfigurierungsdauer ist die Kernidee die Modellierung der Rekonfigurierungen als Markov Kette. Die Abbildung der Module auf Slots, wird neben einer heuristischen Modul-Slot Optimierung auch durch eine angepasste Variante des Threshold Accepting

[14] Es handelt sich hierbei um XDL Resource Report Dateien (xdlrc). Diese Dateien werden vom Xilinx XDL-Tool im Report-Modus erzeugt.

[15] Dieser Prozess bildet die eigentliche technische Herausforderung bei der *Bus Makro Generierung*, da die FPGA Beschreibungen, je nach FPGA Typ, sehr groß sein können

variiert, um den sehr großen Lösungsraum besser zu durchsuchen. Dieser Lösungsraum wird zusätzlich durch verschiedene Slotmodifizierungen verändert, um die durchschnittliche Rekonfigurierungsdauer weiter zu verbessern. Nach der Bestimmung der Slots, sind diese auf den FPGA abzubilden, was in dieser Arbeit mittels eines Greedy Algorithmus realisiert wird. Die Platzierung ist, bis auf die optionale Angabe von Übergangswahrscheinlichkeiten, vollständig automatisiert, wodurch ein Systementwickler entlastet wird.

Des Weiteren wurden in diesem Kapitel die benötigten Zeiten für die Berechnung des Slotgraphen bestimmt und die automatische Generierung der Kommunikationsverbindungen zwischen Slots näher dargestellt.

6 Rekonfigurierungssteuereinheit

Neben der Entwicklung der Funktionalitäten und deren Synthese für ein eingebettetes System, erfordert der Einsatz von dynamischer Rekonfigurierung zusätzlich eine Steuerung zur Konfigurationswechseldurchführung. Dieses Kapitel dient zur Vorstellung eines Konzeptes für eine Rekonfigurierungssteuereinheit (RCU). Kernpunkte sind hierbei nicht nur die Verwaltung der Bitstreams und der Aufbau dieser Einheit, sondern auch deren automatische Generierung. In dem folgenden Abschnitt 6.1 wird die Problematik der Rekonfigurierungsdurchführung näher betrachtet. Dabei wird sowohl der technische Kontext als auch das zu Grunde liegende Rekonfigurierungskonzept berücksichtigt und die Steuerungsaufgaben herausgestellt. Für diese Aufgaben werden im Abschnitt 6.2 die Konzepte der resultierenden RCU Komponenten vorgestellt. Dieser Abschnitt dient auch der Beschreibung des Zusammenspiels der RCU Teile und des Rekonfigurierungsablaufes. Mit dem Abschnitt 6.3 wird das RCU Konzept auf seinen Platzbedarf hin untersucht und das Verhältnis zu FPGA Größe dargestellt. Bevor die Zusammenfassung dieses Kapitels folgt, wird noch auf Eigenschaften der Software zur automatisierten Generierung der Steuereinheit im Abschnitt 6.4 eingegangen.

6.1 Anforderungen an eine Steuerung

Ein dynamisch rekonfigurierbares, eingebettetes System besteht aus drei notwendigen Teilen. Als technische Voraussetzung ist ein FPGA notwendig, der in mehrere Overlaying Bereiche, die Slots, eingeteilt ist. Dieser FPGA muss des Weiteren mit einem externen Speicher für die Module verbunden sein. Der dritte Teil umfasst die eigentliche Hardwarefunktionalität, die das eingebettete System bereitstellen soll. Aufgrund dieser Architektur und der Möglichkeit, dynamisch Konfigurationen zu ändern, ist eine Rekonfigurierungssteuereinheit notwendig.

Unabhängig vom zu Grunde liegenden Rekonfigurierungskonzept, sind bei der Durchführung einer Rekonfigurierung die Bitstreamdaten aus einem externen Speicher abzurufen und an der Konfigurierungsschnittstelle bereitzustellen.

Ein weiterer Punkt ist die Bestimmung, welche Module neu zu laden sind. Hierfür ist es notwendig, die Zustände der Slots zu verwalten. Eine Steuereinheit muss von jedem Slot wissen, ob er belegt oder frei ist und wenn er belegt ist, ob er für die Zielkonfiguration rekonfiguriert werden muss. Diese Verwaltung der Slots muss so frei gestaltet sein, dass alle Konfigurationswechselmöglichkeiten ohne Anpassung der RCU umgesetzt werden können. Im Abschnitt 5.4 wurde gezeigt, dass die Kommunikationsverbindungen zwischen den Slots, entsprechend der jeweiligen Konfiguration, unter Umständen, über Multiplexer mit den Modulen verbunden werden. Diese Multiplexer sind nur bei bestimmten Rekonfigurierungen umzuschalten und sind daher von der Steuereinheit zu konfigurieren.

Eine weitere Problematik die sich bei einer Rekonfigurierung ergibt, ist die Vermeidung von Datenverlust. Ist bei einer aktiven Konfiguration die Abarbeitung der aktuellen Daten noch nicht abgeschlossen, könnten durch eine verfrühte Rekonfigurierung diese Daten überschrieben werden und Systemzustände (Registerinhalte) verloren gehen. Alle auszutauschenden Module

müssen vor der Rekonfigurierung in einem „sicheren“ Zustand sein. In engem Zusammenhang mit diesem Problem steht auch die Frage: Wann und von wem wird eine Rekonfigurierung ausgelöst? Die Aktivierungen der Rekonfigurierungen können entweder zu Zeit des Entwurfs fest, mit genauen Zeitpunkten, geplant sein oder sind abhängig von Umwelt- und Systemereignissen, die nicht vorher bestimmbar sind. Eine feste Rekonfigurierungsabfolge könnte mit in die Steuereinheit integriert werden. Bei allen anderen Fällen würde der Auslöser für eine Rekonfigurierung vom umgebenden System kommen. Das System, in welchem das dynamisch Konfigurierbare eingebettet ist, kann einen Konfigurationswechsel anfordern, aufgrund von inneren Systemzuständen und Sensordaten oder indem es eine Systemnutzereingabe weiterleitet. Theoretisch kann auch ein spezielles Modul zur Auslösung der Rekonfigurierungen in das eingebettete System integriert werden, jedoch kann dieses nur schwer automatisch generiert werden und es ist für solch ein Modul fast unmöglich zu garantieren, dass alle zu rekonfigurierenden Module in einem „sicheren“ Zustand sind. Im Folgenden wird daher nur der automatisch generierbare Teil der Steuereinheit in der RCU zusammengefasst. Die zu generierende RCU soll keine Rekonfigurierungen selbst auslösen können und benötigt daher eine Schnittstelle zum restlichen System, um die Parameter für eine neue Zielkonfiguration zu lesen. Mit dieser Schnittstelle wird die RCU vom restlichen System abgegrenzt und ist unabhängig von einem speziellen Scheduling für die Konfigurationen.

Bei der automatischen Erstellung einer RCU muss berücksichtigt werden, dass sie systemabhängige Komponenten beinhaltet. Der aktuelle Zustand des Systems ist von der Anzahl der Slots und der abgebildeten Module abhängig und damit von System zu System verschieden. Im Gegensatz hierzu ist die Ansteuerung der ICAP Schnittstelle für alle Systeme gleich. Wie im Abschnitt 2.2.2 schon erwähnt, muss für die Verwendung der ICAP Schnittstelle die dynamische Rekonfigurierung aus dem FPGA heraus angesteuert werden. In diesem Fall ist eine in den FPGA integrierte, in Hardware realisierte RCU hilfreich. Eine Softwarelösung könnte ebenfalls in Frage kommen, wobei dann die von außen erreichbare JTAG Schnittstelle besser geeignet ist. Nachteilig bei der Softwarelösung wirken sich der zusätzliche Steuerschaltkreis neben dem FPGA und unter Umständen weitere belegte FPGA-Pins, die für Ansteuerung der Multiplexer benötigt werden, aus. Die gleichen Nachteile treten auch bei einer Hardwarelösung, die auf einem zweiten FPGA implementiert ist, auf. Alle Bemerkungen zur RCU in den vorangegangen Kapiteln zielten daher auf eine integrierte Hardwarelösung im dynamisch rekonfigurierbaren FPGA hin. Diese Entscheidung wurde getroffen, da zur Laufzeit der RCU keine komplexen Entscheidungen mehr getroffen werden müssen und die Entscheidung, welche Module bei einem Konfigurationswechsel ausgetauscht werden müssen, möglichst schnell erfolgen soll. Ziel muss es sein, dass die RCU möglichst wenig Platz im FPGA belegt und nicht mehr Zeit als unbedingt nötig in Anspruch nimmt, damit das restliche System nicht behindert wird.

Die Rekonfigurierungssteuereinheit muss automatisch generierbar sein, um den Entwickler bei der Verwendung von Rekonfigurierung im Entwurf zu entlasten. Durch die Anwendung der Rekonfigurierung in eingebetteten Systemen nach dem Overlaying Konzept kann diese Anforderungen erfüllt werden, weil sich die Anzahl der zu verwaltenden Slots nach der Optimierung nicht ändert und im Normalfall die Menge der Module und Multiplexer konstant bleibt.

6.2 Konzept und Aufbau der Steuereinheit

Aus den in Abschnitt 6.1 genannten Anforderungen ergibt sich die in Abbildung 6.1 dargestellte RCU. Die grau hinterlegten Rechtecke der Abbildung 6.1 stellen die in diesem Abschnitt

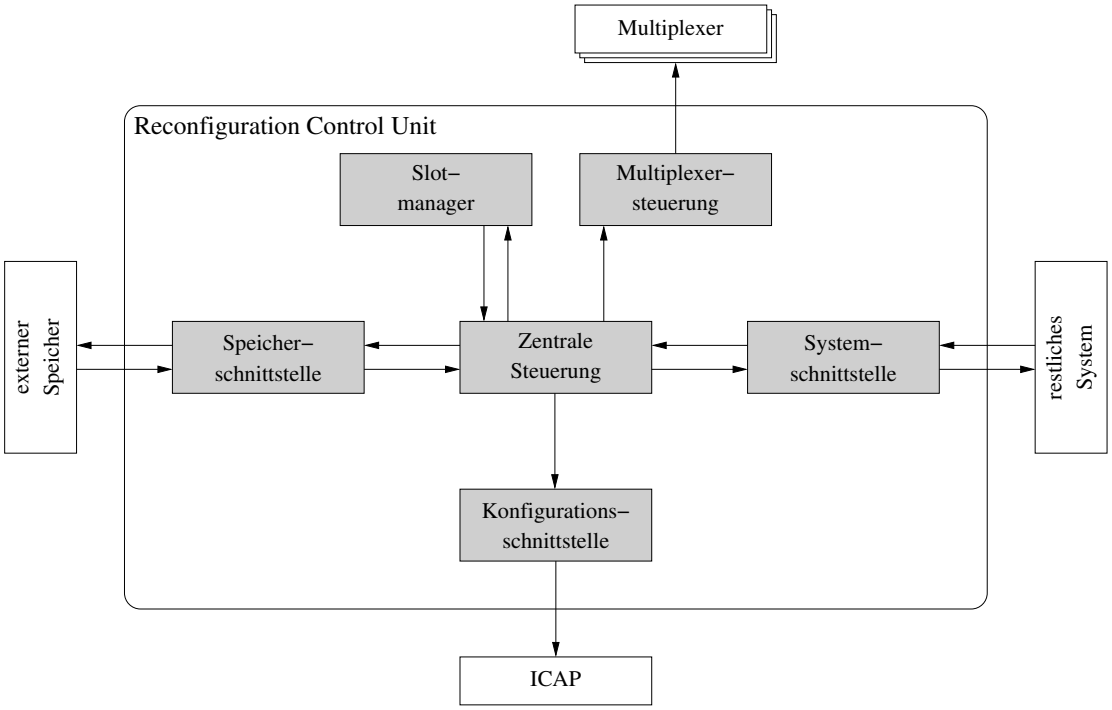


Abbildung 6.1: Zusammenhang der RCU Komponenten

beschriebenen Teilaufgaben der RCU dar. Alle Komponenten die nicht zur RCU gehören, die Multiplexer, der externe Speicher, die ICAP Schnittstelle und das restliche System, sind als weiße Rechtecke abgebildet. Die Schnittstellen der RCU und auch der zu erwartende, interne Datenfluss sind mit Pfeilen zwischen den Komponenten symbolisiert. Da einzelne Teile der RCU vom konkreten System abhängen, in dem die RCU eingesetzt wird, bietet es sich an, die RCU, wie im Abschnitt 2.3.2 beschrieben, nach der *Platzierung* automatisch zu erzeugen. In den folgenden Abschnitten werden die Teilaufgaben der RCU näher beschrieben.

6.2.1 Systemschnittstelle

Das umliegende System muss der RCU mitteilen, wann diese eine Rekonfigurierung durchführen soll. Dazu ist die von der RCU benötigte Zielkonfiguration $konf_Z$ durch das System mitzuteilen. Die Erstellung dieser Schnittstelle erfordert daher eine Abbildung der Konfigurationsbezeichnungen auf die natürlichen Zahlen, die dann binär kodiert an die RCU übertragen werden (siehe Gleichung 6.1).

$$konfID(konf) = Konf \rightarrow \mathbb{N} \tag{6.1}$$

Für die Regelung des weiteren Ablaufs einer Rekonfigurierung werden an diese Schnittstelle Steuersignale angelegt bzw. bereitgestellt. Das System signalisiert der RCU über ein „Start“–Signal, dass die Rekonfigurierung beginnen soll. Hat die RCU die Rekonfigurierung dann abgeschlossen, teilt sie dies dem System über ein weiteres Signal mit.

Die Systemschnittstelle ist von der eigentlichen Rekonfigurierungssteuerung getrennt, um eine Anpassung an verschiedene Systemumgebungen zu erleichtern. In der aktuellen Imple-

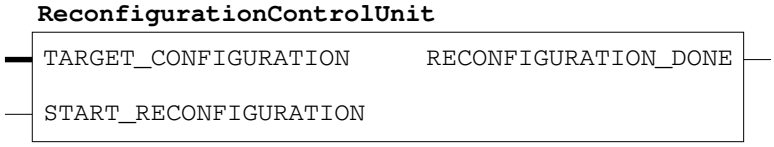


Abbildung 6.2: RCU aus Sicht des umgebenden Systems

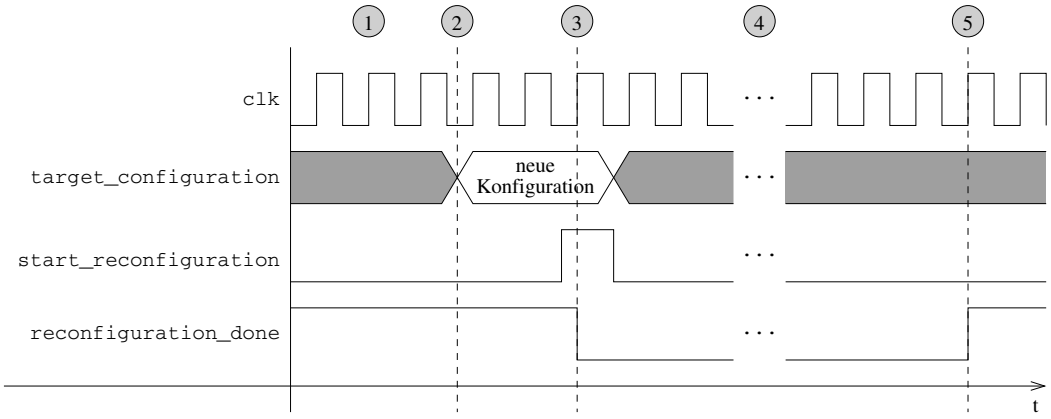


Abbildung 6.3: Signalverlauf für einen Konfigurationswechsel

mentierung der Systemschnittstelle sind die Leitungen von der RCU direkt nach außen geführt und das äußere System spricht die RCU direkt an.

Aus Sicht des restlichen Systems sieht die RCU damit wie in Abbildung 6.2 dargestellt aus. Über das Signal `TARGET_CONFIGURATION` wird die Zielkonfiguration angegeben und mit dem Signal `START_RECONFIGURATION` wird der Ablauf gestartet. Das Ausgangssignal `RECONFIGURATION_DONE` zeigt an, dass die Rekonfigurierung abgeschlossen ist.

Das Protokoll zur Durchführung eines Konfigurationswechsels ist in Abbildung 6.3 dargestellt. In der Abbildung sind die folgenden Schritte einer Rekonfigurierung markiert:

1. Der vorherige Konfigurationswechsel wurde abgeschlossen. Die RCU ist für einen neuen Konfigurationswechsel bereit.

2. Die Identifikationsnummer der neuen Konfiguration wird an den Eingang `target_configuration` angelegt.

3. Um die Rekonfigurierung zu starten, wird am Eingang `start_reconfiguration` ein High-Pegel angelegt.

4. Die RCU führt die Rekonfigurierung des FPGA durch.

5. Die Rekonfigurierung des FPGA ist abgeschlossen. Dies wird durch einen High-Pegel am Ausgang `reconfiguration_done` angezeigt.

Diese Schnittstelle kann gegebenenfalls leicht angepasst werden.

6.2.2 Verwaltung der Slotzustände

Ein Hauptkennzeichen der RCU ist die Verwaltung des Systemzustands, damit sichergestellt werden kann, dass in jeder Konfiguration des Systems, die richtigen Module in den Slots des FPGA geladen sind. Hierzu muss sowohl die Menge der Bitstreams $Bitstream$ als auch die Menge Slots $Slot$ bekannt sein.

Bitstreams

Zum Laden eines Moduls in den FPGA wird ein Bitstream benötigt, mit welchem die physischen Elemente des FPGA programmiert werden. In einem Bitstream ist auch die Position eines Moduls im FPGA festgelegt. Deshalb wird hier für jede Slot-Modul Kombination ein separater Bitstream benötigt, da eine dynamische Bitstreammodifizierung zur Verschiebung der Module innerhalb des FPGA im Overlaying Kontext nicht vorgesehen ist. Ein Bitstream $bitstream = (mod, slot)$ bezeichnet daher die Bindung eines Moduls an einen Slot (siehe Gleichung 6.2). Diese Bitstreams werden von der RCU an die Konfigurationsschnittstelle des FPGA übergeben.

Ein konkretes Modul kann in mehrere Slots platziert werden, was zu mehreren verschiedenen Bitstreams führt. Deshalb ist es nicht ausreichend, nur die in einer Konfiguration $konf$ benötigten Module $mod \in konf$ zu betrachten. Über die Funktion[1] ϕ^{-1} kann zu jedem Slot $slot_i$ für eine bestimmte Konfiguration $konf$ der entsprechende Bitstream $(\phi^{-1}(konf, slot_i), slot_i)$ ermittelt werden. Bildet die Funktionen ϕ^{-1} auf die leere Menge ab, so existiert auch kein Bitstream (siehe Gleichung 6.3). Bei der Durchführung einer Rekonfigurierung $rekonf = (konf_A, konf_Z, p)$ wird für jeden Slot $slot$ der entsprechende Bitstream $(\phi^{-1}(konf_Z, slot), slot)$ bestimmt. Es sind jedoch nur die Bitstreams neu zu laden, die noch nicht in der Ausgangskonfiguration $konf_A$ auf dem FPGA existierten.

$$Bitstream = Mod \times Slot \tag{6.2}$$

$$\forall\, \phi^{-1}(konf, slot) \neq \emptyset \Leftrightarrow \exists\, bitstream = (mod, slot) : \phi^{-1}(konf, slot) = mod \tag{6.3}$$

Slotzustände

Wie in dem vorhergehenden Absatz erwähnt müssen nicht immer alle angeforderten Bitstreams zum FPGA übertragen werden. Eine Rekonfigurierung eines Slots ist nicht notwendig, wenn sich der Bitstream in der Ausgangskonfiguration $konf_A$ nicht von dem in der Zielkonfiguration $konf_Z$ unterscheidet. $(\phi^{-1}(konf_A, slot), slot) = (\phi^{-1}(konf_Z, slot), slot)$.

- Das bedeutet entweder, dass auf diesem Slot ein Modul abgebildet ist, das von beiden Konfigurationen verwendet wird,
- oder das abgebildete Modul gehört zu $konf_A^+$ und wurde bei einer früheren Konfiguration in den betrachteten Slot $slot$ des FPGAs geladen.

Um diese Eigenschaft für eine minimale Rekonfigurierungsdauer auszunutzen, ist es notwendig, den aktuell im Slot befindlichen Bitstream zu identifizieren. Für jeden zu verwaltenden Slot existiert daher ein Register, das die BitstreamID des aktuell im Slot geladenen Bitstreams aufnimmt. Weiterhin sind nur die Slots zu betrachten, die Module aus der Menge $konf_Z$ und nicht

[1] siehe Gleichung 5.20 im Abschnitt 5.3

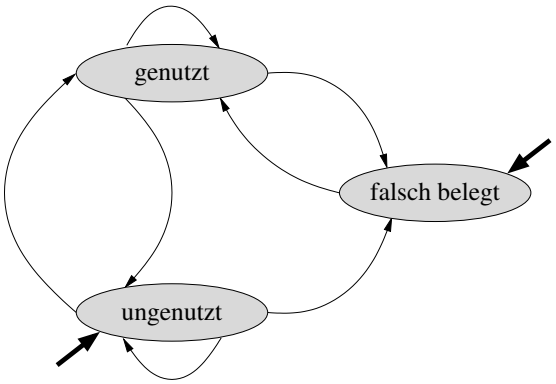


Abbildung 6.4: Zustandsgraph eines Slots

aus $konf_Z^+$ enthalten werden. Dadurch wird die Menge der Slots bezüglich einer Konfiguration unterschieden in benötigte und nicht benötigte Slots. Alle Slots die rekonfiguriert werden müssen stellen eine Teilmenge der benötigten Slots in Konfiguration $konf_Z$ dar. Die zu rekonfigurierenden Slots werden bei einer Rekonfigurierungsanfrage kurzzeitig in einen Wartezustand versetzt, um der RCU anzuzeigen, dass ein neuer Bitstream geladen werden muss. Die Unterscheidung der verschiedenen Slotzustände zu unterschiedlichen Zeitpunkten, kann mittels eines Automaten modelliert werden. Ein entsprechender Automat für den Zustand eines Slots ist in der Abbildung 6.4 dargestellt. In der Umsetzung der RCU ist für jeden Slot dieser systemunabhängige Automat zu erzeugen.

Der Anfangszustand (siehe fette Pfeile in Abbildung 6.4) des Automaten hängt davon ab, ob in der Anfangskonfiguration $konf_{init}$ ein Bitstream im betrachteten Slot benötigt wird. Wenn kein Bitstream geladen werden muss, ist der Anfangszustand *ungenutzt*, sonst *falsch belegt*. Wobei der Zustand *falsch belegt* nach Abschluss der initialen Konfigurierung zu *genutzt* wechselt. Im Folgenden werden die Übergangsbedingungen zwischen den einzelnen Zuständen beschrieben.

Wenn die benötigten Bitstreams in der aktuellen und nächsten Konfiguration für einen Slot gleich sind, bleibt der Slot im *genutzt* Zustand. $((\phi^{-1}(konf_A, slot), slot) = (\phi^{-1}(konf_Z, slot), slot))$. Ein erneutes Laden des Bitstreams wird in diesem Zustand nicht durchgeführt, da dies unnötig ist. Wenn in der nächsten Konfiguration kein Bitstream benötigt wird, geht der betrachtete Slot vom Zustand *genutzt* in den Zustand *ungenutzt* über $(\phi^{-1}(konf_Z, slot) = \emptyset)$. Der bis dahin im Slot geladene Bitstream wird nicht überschrieben oder gelöscht. Für diesen Slot wird aber weiterhin der geladene Bitstream gespeichert und das geladene Modul gehört dann zur Menge $konf_Z^+$. $(\phi^{-1}(konf_A, slot) \in)$ Der dritte ausgehende Pfeil vom Zustand *genutzt* zeigt den Wechsel zum Zustand *falsch belegt* an, wenn sich die Bitstreams in der aktuellen und nächsten Konfiguration unterscheiden. $((\phi^{-1}(konf_A, slot), slot) \neq (\phi^{-1}(konf_Z, slot), slot) \wedge \phi^{-1}(konf_Z, slot) \neq \emptyset)$

Ein Slot im Zustand *ungenutzt* kann bei einem Konfigurationswechsel weiterhin ungenutzt bleiben, in den *genutzt* oder in den *falsch belegt* Zustand wechseln. Dieser Slotzustand *ungenutzt* kann auftreten, wenn kein Bitstream geladen wurde, da keine Abbildung eines Moduls auf diesen Slot in der initialen Konfiguration $konf_{init}$ existiert $(\phi^{-1}(konf_{init}, slot) = \emptyset)$. Ist bei einer nächsten Konfiguration ebenfalls keine Modulabbildung vorhanden, bleibt der Slot im Zustand *ungenutzt*. Ein bis dahin geladener Bitstream bleibt erhalten und das geladene Mo-

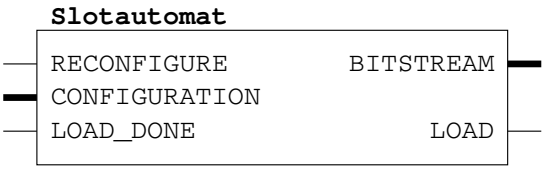


Abbildung 6.5: Schnittstelle zu einzelnem Slot

dul wird der Menge $konf_Z^+$ zugeordnet. Wird in einer nächsten Konfiguration der geladene Bitstream des Slots benötigt, geht der Slot direkt in den *genutzt* Zustand ohne Rekonfigurierung über $((\phi^{-1}(konf_A, slot) \in konf_A^+, slot) = (\phi^{-1}(konf_Z, slot), slot))$. Der Übergang von *ungenutzt* zu *falsch belegt* tritt auf, wenn ein anderer als der bereits geladene oder erstmals ein Bitstream benötigt wird $((\phi^{-1}(konf_A, slot), slot) \neq (\phi^{-1}(konf_Z, slot), slot) \vee \phi^{-1}(konf_A, slot) = \emptyset \wedge \phi^{-1}(konf_Z, slot) \neq \emptyset)$.

Der Zustand *falsch belegt* zeigt an, dass ein neuer Bitstream geladen werden muss. Solange das Laden des neuen Bitstreams noch andauert, verbleibt der Slot in diesem Zustand. Dieser Zustand geht in den *genutzt* Zustand über, sobald das Laden des Bitstreams $\phi^{-1}(konf_Z, slot)$ beendet ist.

Die Schnittstelle des implementierten Automaten, die für alle Slots gleich aufgebaut ist, ist in Abbildung 6.5 zu sehen. Zur Bestimmung des benötigten Bitstreams muss die neue Konfiguration am Eingang `CONFIGURATION` angelegt werden. Das Signal `RECONFIGURE` löst dann die eigentliche Slotabfrage aus. Falls für die neue Konfiguration ein Bitstream geladen werden muss, wird dies über die Ausgänge `BITSTREAM` und `LOAD` signalisiert. Am Eingang `LOAD_DONE` wird dem Automaten mitgeteilt, dass der Bitstream vollständig geladen wurde.

Ein Signalverlauf bei einem Konfigurationswechsel ist beispielhaft in Abbildung 6.6 dargestellt. Dabei laufen die folgenden, in der Darstellung markierten Schritte, ab.

1. Anfrage, in *Konfiguration 1* zu wechseln. In dieser Konfiguration wird für den Slot der *Bitstream 1* benötigt. Der Slot geht in den Zustand *falsch belegt* über und zeigt über das Signal `load` an, dass ein Bitstream geladen werden muss.

2. Das Laden des Bitstreams wurde abgeschlossen. Der Slot geht nun in den Zustand *genutzt* über und nimmt die Anfrage zurück (`load` schaltet auf 0).

3. Anfrage, in *Konfiguration 2* zu wechseln. In dieser Konfiguration wird der Slot in diesem Beispiel nicht benutzt. Der Slot geht in den Zustand *ungenutzt* über. Der im Slot geladene Bitstream bleibt erhalten.

4. Anfrage, in *Konfiguration 3* zu wechseln. Der noch im Slot geladene Bitstream kann nicht wiederverwendet werden, da in dieser Konfiguration *Bitstream 2* benötigt wird. Der Slot geht in den Zustand *falsch belegt* über und zeigt über das Signal `load` an, dass ein Bitstream geladen werden muss.

5. Das Laden des Bitstreams wurde abgeschlossen. Der Slot geht nun in den Zustand *genutzt* über und schaltet das Signal `load` auf 0 um.

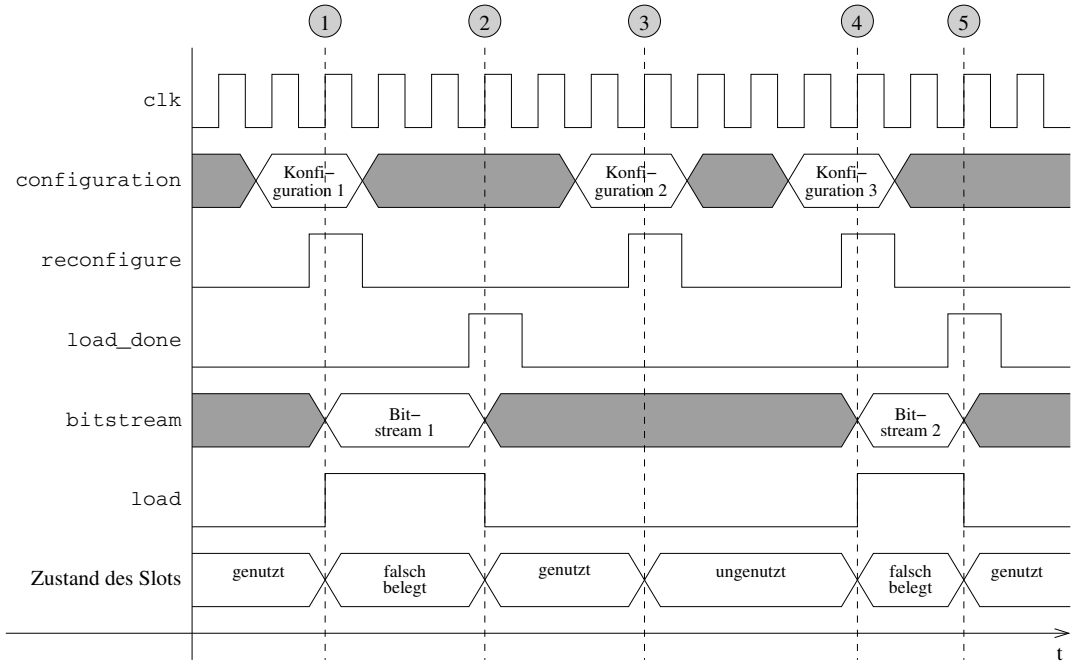


Abbildung 6.6: Signalverlauf bei einem Konfigurationswechsel an einem Slotautomaten

Abfrageeinheit

Mit der Abfrageeinheit werden die parallel existierenden Slotautomaten der Reihe nach abgefragt und die angeforderten Bitstreams sequentiell weitergeleitet, da die zu ladenden Bitstreams nur sequenziell an der Programmierschnittstelle angelegt werden können. Um die Abfrageeinheit automatisch generieren zu können, ist die Anzahl der Slots als Eingabe notwendig. Ihre Schnittstelle ist in Abbildung 6.7 dargestellt. Über die Signale `SLOT_n_BITSTREAM` und `SLOT_n_LOAD` werden die Anfragen der Slots entgegengenommen und über `SLOT_n_LOAD_DONE` wird den Slots das Laden der Bitstreams bestätigt. Die Signale `NEXT_BS`, `BS_REQUEST` und `BS_DONE` bilden die Schnittstelle zum umgebenden System und werden daher im folgenden Abschnitt *Slotmanager* beschrieben.

Nach einer Anfrage zu einem Konfigurationswechsel läuft zwischen der Abfrageeinheit und den Slots der in Abbildung 6.8 dargestellte Signalverlauf ab. Es werden der Reihe nach alle Anfragen der Slots über das Signal `BS_REQUEST` ausgegeben. Die folgenden Punkte sind im Signalverlauf markiert.

1. Abfrage nach der Anforderung des ersten Slots. Das Signal `bs_done` geht auf 0.

2. Abfrage nach der Anforderung des nächsten Slots. Da Slot 2 keinen Bitstream angefordert hat, erscheint am Ausgang `bs_request` *Bitstream 0*. Das Laden des Bitstreams für Slot 1 wird bestätigt.

3. Da Slot 2 keine Anforderung hatte, wird auch keine Bestätigung gesetzt. Die Anforderung von Slot 3 erscheint an `bs_request`.

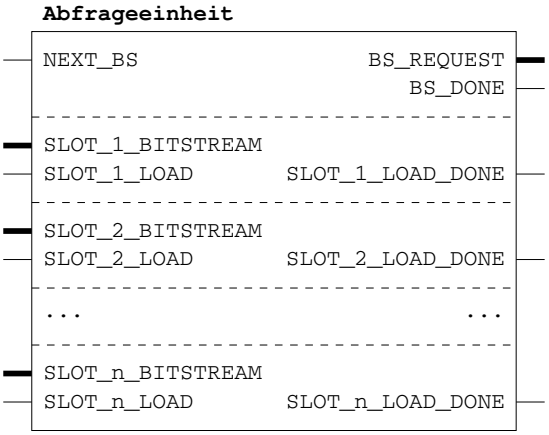


Abbildung 6.7: Schnittstelle der Abfrageeinheit

4. Die Anforderung des letzten Slots erscheint an `bs_request`.

5. Erneute Abfrage nach der Anforderung. Das Laden des Bitstreams für den letzten Slot wird bestätigt und `bs_done` geht wieder auf 1, da alle Slots abgefragt wurden.

Slotmanager

Der Slotmanager stellt gegenüber der restlichen Steuerung eine von der Slotanzahl unabhängige Schnittstelle bereit und fasst die Automaten für die einzelnen Slots und die Abfrageeinheit zusammen. Die Schnittstelle des Slotmanagers ist in Abbildung 6.9 dargestellt. Über Eingangssignale `TARGET_CONFIGURATION` und `RECONFIGURE` wird die Rekonfigurierungsanfrage an den Slotmanager weitergeleitet. Die von den Slots angeforderten Bitstreams werden der Reihe nach über `BS_REQUEST` ausgegeben und über das Signal `NEXT_BS` weitergeschalten. Bei diesem Vorgang werden stets alle Slots abgefragt. Wird von einem Slot kein Bitstream angefordert, so wird statt einer Anfrage die BitstreamID in der Implementierung eine 0 ausgegeben. Wurden alle Anfragen der Slots abgeholt, wird dies über das Signal `BS_DONE` angezeigt.

Der Aufbau des Slotmanagers ist in Abbildung 6.10 dargestellt. Neben den einzelnen Slotautomaten ist auch die oben beschriebene Abfrageeinheit integriert, die nacheinander die angeforderten Bitstreams an der Schnittstelle `NEXT_BS` ausgibt.

Bei einem Konfigurationswechsel laufen die in Abbildung 6.11 markierten Schritte ab[2].

1. Anlegen der Zielkonfiguration an `target_configuration` und Auslösen der Rekonfigurierungsanfrage durch eine 1 am Eingang `reconfigure`.

2. Abfrage der von den Slots angeforderten Bitstreams. Am Ausgang `bs_request` wird mit jeder aktiven Taktflanke, bei der `next_bs` den Wert 1 annimmt, die Anfrage des nächsten Slots ausgegeben.

3. Wenn alle Slots abgearbeitet wurden, wechselt der Ausgang `bs_done` auf 1.

[2]Statt der angeforderten BitstreamIDs ist in der Abbildung beim Signal `bs_request` der anfordernde Slot angegeben.

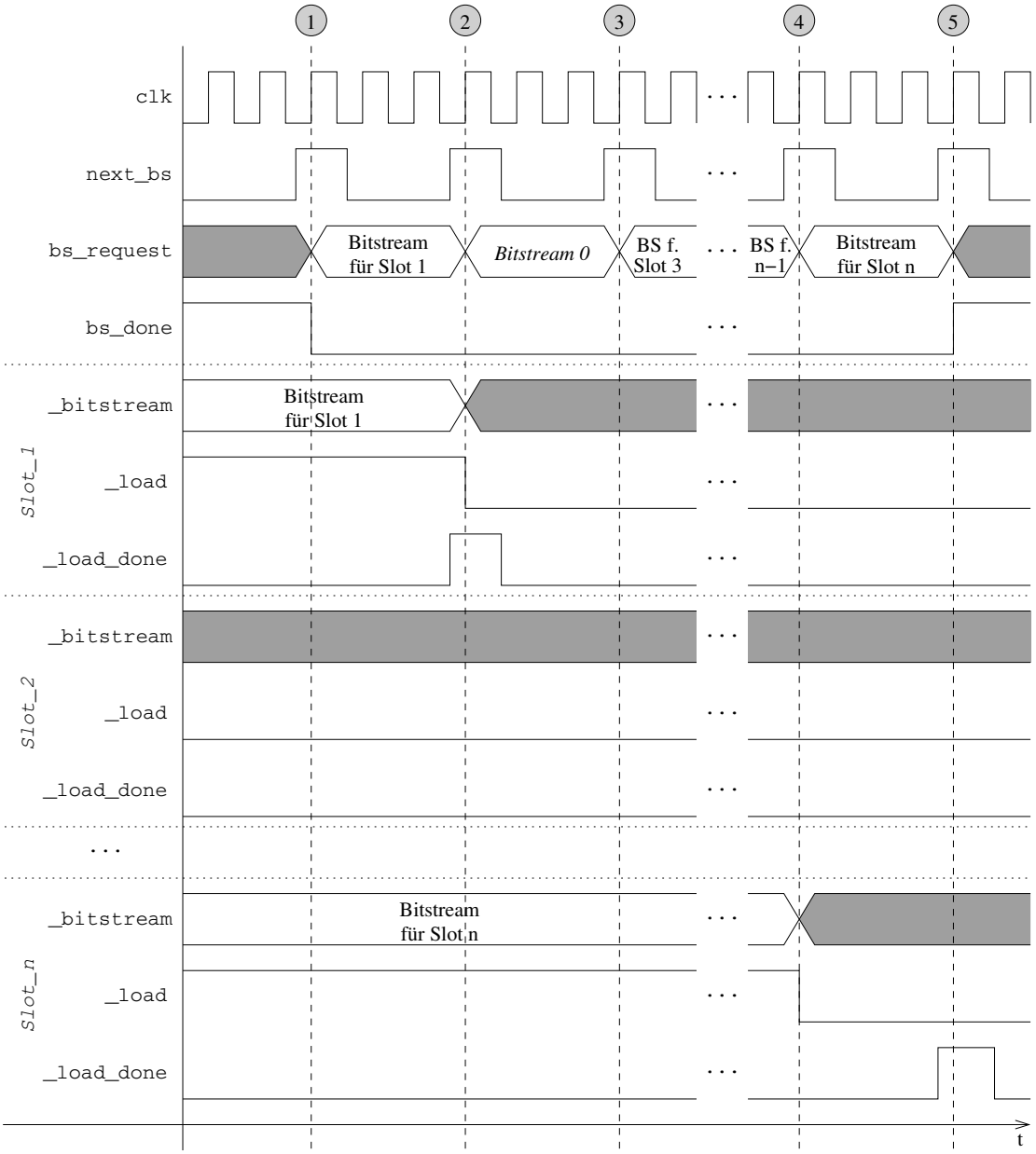


Abbildung 6.8: Signalverlauf zwischen Slots und Abfrageeinheit

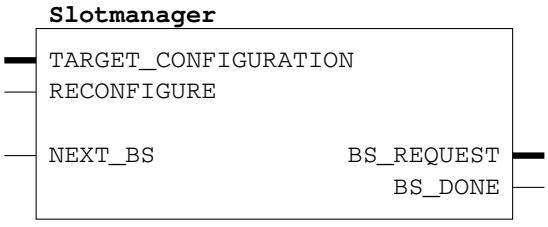


Abbildung 6.9: Schnittstelle des Slotmanagers

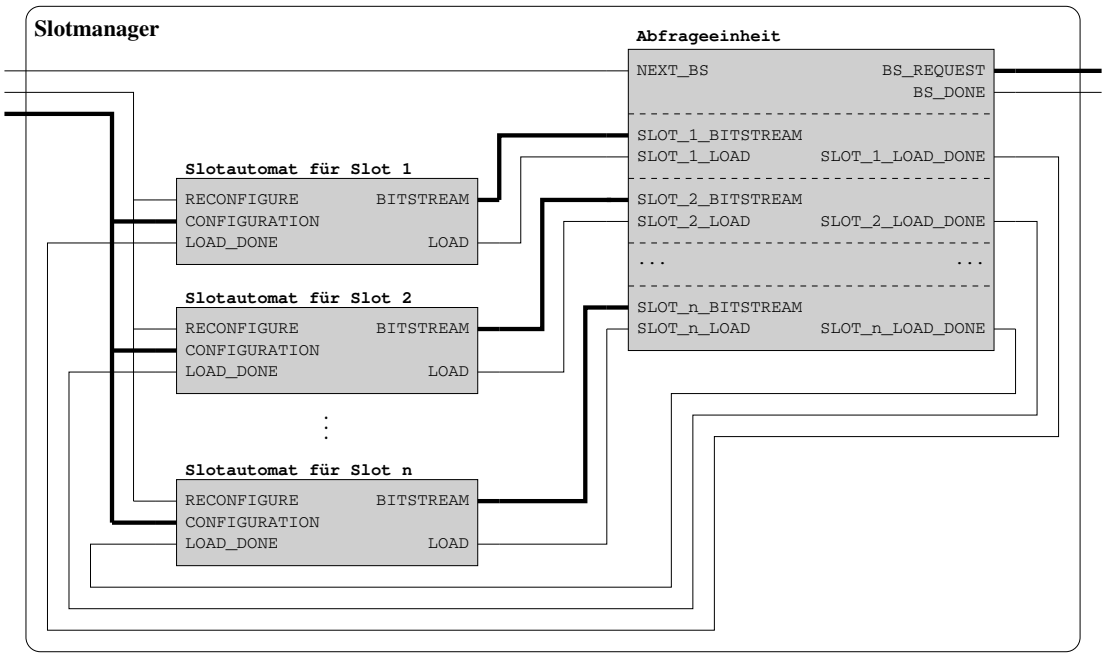


Abbildung 6.10: Blockbild Slotmanager

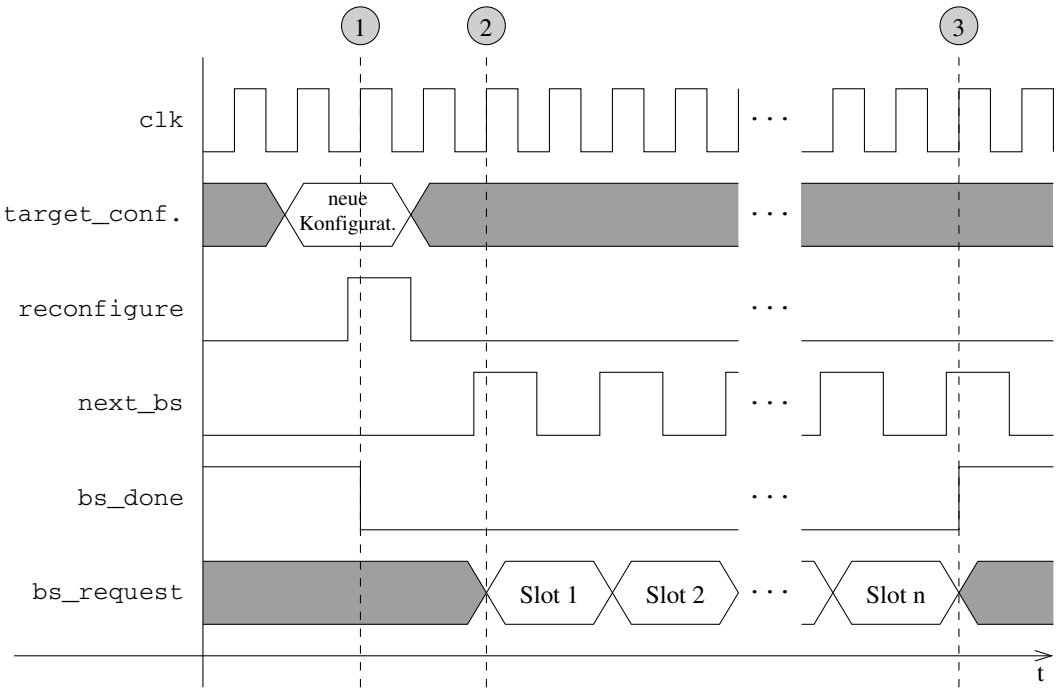


Abbildung 6.11: Signalverlauf bei Konfigurationswechsel an der Schnittstelle des Slotmanager

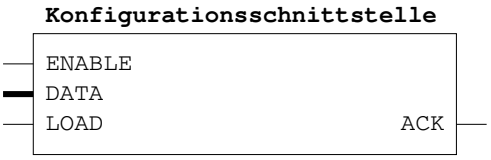


Abbildung 6.12: Abstrakte Konfigurationsschnittstelle

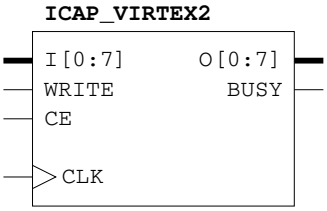


Abbildung 6.13: ICAP Schnittstelle

6.2.3 Abstrakte Konfigurationsschnittstelle

Für eine in Hardware realisierte RCU werden die Bitstreams zur Konfiguration des FPGAs an die ICAP Schnittstelle übergeben. Da sich die Konfigurationsschnittstelle von FPGA zu FPGA unterscheiden kann und neben den reinen Konfigurationsdaten der Bitstreams eventuell noch weitere Steuerinformationen an die Schnittstelle übergeben werden müssen, wird die FPGA spezifische ICAP Schnittstelle durch eine abstrakte Konfigurationsschnittstelle von der RCU abgetrennt. Die abstrakte Schnittstelle kapselt die ICAP Schnittstelle des FPGA. Dadurch kann, unabhängig vom tatsächlichen Konfigurationsverfahren, eine einheitliche Schnittstelle zur Steuerung innerhalb der RCU bereitgestellt werden.

Aus der Sicht der RCU besteht eine Rekonfigurierung aus der Übergabe der entsprechenden Bitstreamdaten an die abstrakte Konfigurationsschnittstelle. Die Ansteuerung der Schnittstelle am FPGA wird von der abstrakten Schnittstelle durchgeführt. Dazu gehört die Aktivierung der Schnittstelle bevor ein Bitstream geladen wird und das Deaktivieren der Schnittstelle nachdem der Bitstream vollständig übertragen wurde. Weiterhin ist eventuell noch die Datenbreite der Daten aus dem Speicher an die Datenbreite der Schnittstelle am FPGA anzupassen.

Die abstrakte Konfigurationsschnittstelle ist in Abbildung 6.12 dargestellt. Zum Laden eines Bitstreams muss die Konfigurationsschnittstelle nur aktiviert und die Bitstreamdaten im Speicher bereitgestellt werden. Der eigentliche Datentransfer wird dann von der Konfigurationsschnittstelle gesteuert. Eine Anpassung der eventuell verschiedenen Bitbreiten der Schnittstelle des FPGA und des Speichers kann hier ebenfalls erfolgen. Ein Beispielsignalverlauf für diese Schnittstelle wird im Abschnitt 6.2.6 in Zusammenhang mit weiteren Komponenten der RCU beschrieben.

ICAP-Schnittstelle

Die in Abbildung 6.13 dargestellte ICAP Schnittstelle dient zur Übertragung der Bitstreamdaten an den FPGA. Die ICAP Schnittstelle ist direkt in der abstrakten Konfigurationsschnittstelle der RCU eingebunden und wird von dieser gesteuert. Die Bitstreamdaten werden dem FPGA byte-

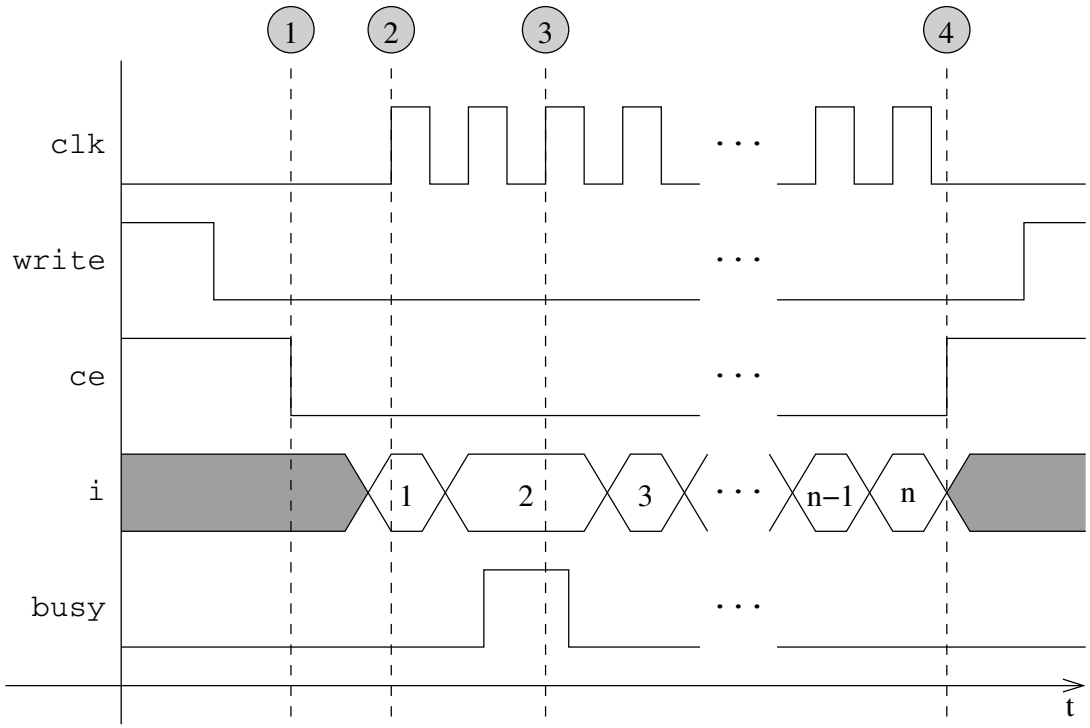


Abbildung 6.14: Signalverlauf an der ICAP Schnittstelle

weise übergeben. Beim Laden eines Bitstreams während einer Rekonfigurierung ergibt sich am ICAP der in Abbildung 6.14 dargestellte Signalverlauf, mit folgenden vier markierten Punkten.

1. Aktivieren der Schnittstelle durch Anlegen einer 0 an `write` und anschließendem Anlegen einer 0 an `ce`.
2. Anlegen eines Bytes des Bitstreams und Erzeugen eines Taktimpulses.
3. Wenn die ICAP-Schnittstelle das Signal `busy` auf 1 setzt, müssen die Daten an `i` noch für einen weiteren Takt gehalten werden.
4. Nachdem alle Bytes übertragen wurden, erfolgt das Deaktivieren der ICAP-Schnittstelle. Zuerst wird `ce` wieder auf 1 gesetzt und danach `write`.

6.2.4 Speicherschnittstelle

In den meisten Fällen ist der im FPGA integrierte Speicher nicht groß genug, um alle Bitstreams des Systems abzulegen. Oft wird der interne Speicher auch für die verschiedenen Register der Systemfunktionalitäten oder auch für zu verarbeitende Daten benötigt. Deshalb ist es notwendig, die Bitstreams in einem externen Speicher abzulegen und abzurufen.

Analog zur Kapselung der ICAP Schnittstelle wird auch die Speicherschnittstelle von der RCU getrennt, damit verschiedene Speicherarten leicht adaptiert werden können, ohne die Funktionalität der RCU zu beeinflussen. Zu den restlichen Komponenten der RCU kann dann immer die gleiche Speicherschnittstelle verwendet werden.

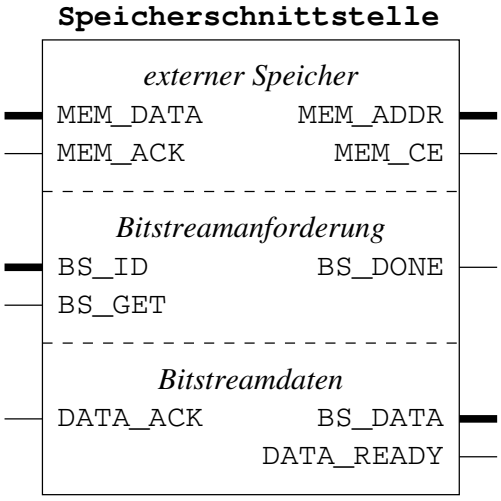


Abbildung 6.15: Schnittstelle der RCU Speicherschnittstellenkomponente

Um die Bitstreams zur Laufzeit im Speicher zu lokalisieren, muss bei einem einfachen Speicherkonzept, die Größe und die Anfangsadresse der jeweiligen Bitstreamdatei der Schnittstelle bekannt sein. Da die Bitstreams während der Erzeugung der RCU noch nicht vorliegen, ist auch deren Größe noch unbekannt. Deshalb ist es nicht möglich, die Anfangsadressen und Größen der Bitstreams fest in der RCU zu speichern. Um dieses Problem zu lösen, müssen diese Verwaltungsinformationen mit im Speicher bereitgestellt und von der Speicherschnittstelle ausgewertet werden. Der Vorteil hierbei ist, dass die Schnittstellenkomponente unabhängig vom jeweiligen System ist.

Die Schnittstelle zum externen Speicher ist nicht speziell auf einen bestimmten Speichertyp ausgerichtet, sondern so weit wie möglich allgemein gehalten. Der Anschluss eines bestimmten Speichers erfolgt über eine Anpassung der Speicherschnittstellenkomponente der RCU an die physische Speicherschnittstelle. Dadurch lassen sich verschiedene Speichertypen verwenden. Der Anschluss an einen Bus oder die Übertragung über ein serielles Protokoll sind durch dieses Konzept nicht beschränkt. Die Schnittstelle der RCU Speicherschnittstellenkomponente ist in Abbildung 6.15 zu sehen und in Abbildung 6.16 ist der Signalverlauf beim Zugriff auf diese Komponente dargestellt. Folgende Schritte laufen bei einem Speicherzugriff ab (siehe Markierungen in Abbildung 6.16).

1. Anlegen der Adresse an `mem_addr` und Aktivieren des Speichers durch eine 1 an `mem_ce`.

2. Die vom Speicher angeforderten Daten liegen an `mem_data` an. Dies wird durch eine 1 an `mem_ack` angezeigt.

Bei einer Anforderung eines Bitstreams durch die RCU liest die abstrakte Speicherschnittstelle zunächst die Größe und die Adresse des ersten Bytes des Bitstreams aus einer Tabelle aus (siehe folgenden Abschnitt). Danach stehen der Konfigurationsschnittstelle schrittweise die Bitstreamdaten zur Verfügung. Die Datenübertragung zur Konfigurationsschnittstelle ist in Abbildung 6.22 dargestellt und wird in Zusammenhang mit weiteren Komponenten der RCU in Abschnitt 6.2.6 erläutert.

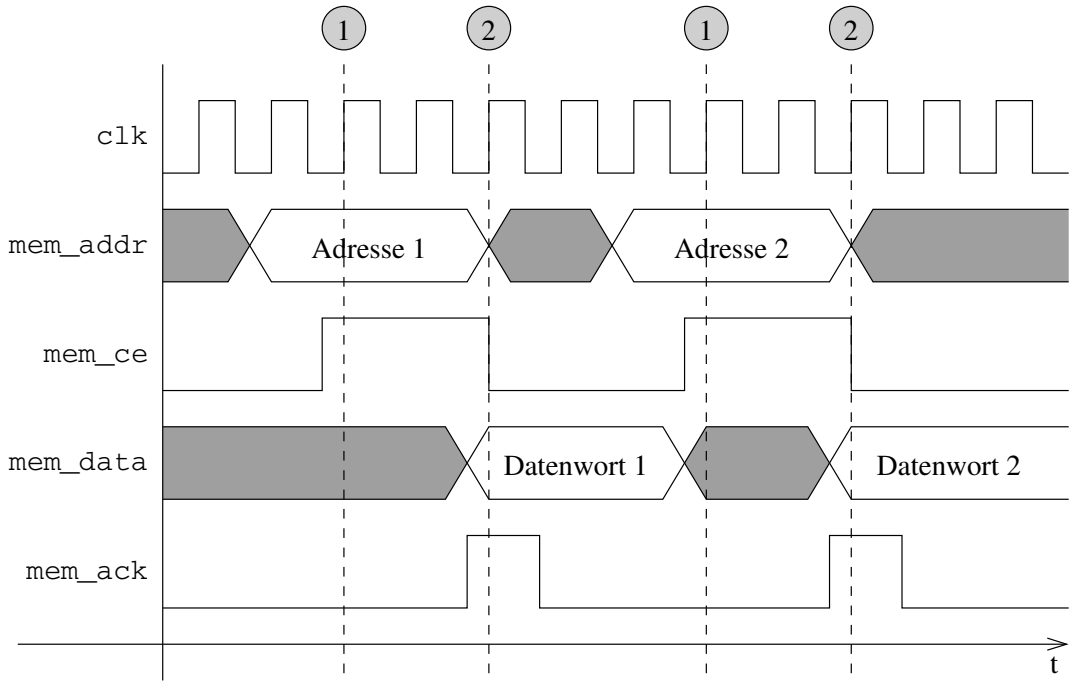


Abbildung 6.16: Zugriff auf den externen Speicher

Speicherlayout

Die benötigte Größeninformation zu einem Bitstream und die Startadresse im Speicher, kann in Tabellenform zusammengefasst werden. Dies ist beispielhaft in Tabelle 6.1 dargestellt. Wird jedem Bitstream eine eindeutige Nummer zugeordnet, können über diese *BitstreamID* die notwendigen Informationen zum Laden des Bitstreams aus dem Speicher nachgeschlagen werden. Eine solche Tabelle lässt sich einfach am Anfang des von der RCU verwendeten Speichers ablegen.

BitstreamID	Startadresse	Größe
1	0x00010000	0x1234
2	0x00011234	0x4000
3	0x00015234	0x1000
...	...	...

Tabelle 6.1: Verwaltungsinformationen für die Bitstreams

Da der Speicher unter Umständen nicht ausschließlich von der RCU genutzt wird, sollte der verwendete Speicherbereich flexibel sein. Das bedeutet, der verwendete Speicherblock muss nicht am Anfang des Speichers liegen. Somit ist es möglich, dass der restliche Speicher von anderen Teilen des Systems genutzt wird. Alle Informationen in der Verwaltungstabelle werden dann relativ zum Anfang des von der RCU genutzten Speicherbereichs betrachtet (siehe Abbildung 6.17).

Abbildung 6.17: Speicherlayout

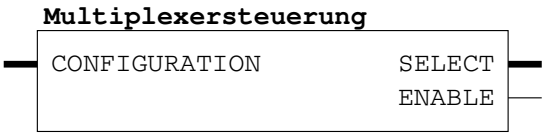


Abbildung 6.18: Schnittstelle der Multiplexersteuerung

6.2.5 Steuerung der Multiplexer

Die in Abschnitt 6.1 geforderte Steuerung der Multiplexer wird unabhängig vom Rekonfigurierungsprozess mit von der RCU bewerkstelligt. Diese Steuerung setzt voraus, dass alle anzusteuernden Multiplexer MUX mit ihren Steueranschlüssen $(select, enable)$ bekannt sind. Zusätzlich ist aus den in Abschnitt 5.4 vorgestellten Bedingungen die Abbildung τ_{mux} : $Konf \rightarrow (select, enable)$ zu definieren. Die Bestimmung der Multiplexer wird nach Abschluss der Platzierung durchgeführt.

Die Steuerinformation für einen Multiplexer besteht aus zwei Teilen. Das $select$-Signal bestimmt, welcher Eingang des Multiplexers auf den Ausgang durchgeschaltet wird ($select \in \mathbb{N}$). Damit lassen sich die Kommunikationsverbindungen entsprechend umschalten. Um Verbindungen auch trennen zu können, kann mit dem $enable$-Signal auch der Ausgang des Multiplexers von den Eingängen abgekoppelt werden ($enable \in \{true, false\}$). Für jeden Multiplexer sind die Steuersignale $(select, enable)$ über die Funktion τ_{mux} von der RCU bereitzustellen.

In der Umsetzung der RCU wird für jeden zu steuernden Multiplexer eine Komponente erzeugt, die aus der aktuellen Konfiguration die Ansteuerung für einen Multiplexer berechnet. Die Schnittstelle der Multiplexersteuerung ist in Abbildung 6.18 dargestellt. Durch kombinatorische Logik wird die Berechnung der Multiplexeransteuerung umgesetzt. Daher ist es notwendig, die bei einem Konfigurationswechsel angeforderte Konfiguration in einem Register zwischenzuspeichern. Der Ausgang dieses Registers ist mit dem `CONFIGURATION` Eingang der Multiplexeransteuerung verbunden.

6.2.6 Zentrale Steuerung

Die Zusammenschließung aller vorgestellten Komponenten der RCU wird durch eine *zentrale Steuerung* realisiert. Unter Einbeziehung dieser zentralen Einheit, ergibt sich das in Abbildung

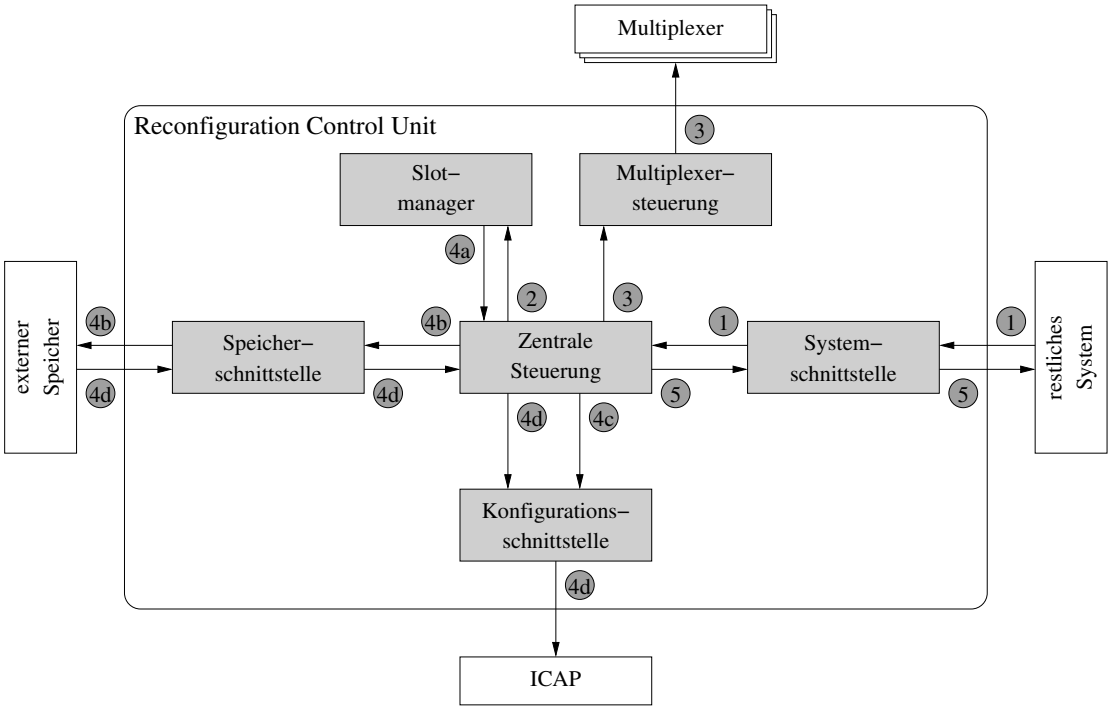


Abbildung 6.19: Interner Ablauf in der RCU bei einer Rekonfigurierung

6.19 skizzierte Gesamtbild der RCU. Zur Veranschaulichung der Abläufe zwischen den Komponenten der RCU sind an den Kanten der Abbildung 6.19 die nacheinander ablaufenden Schritte markiert.

1. Anfrage des umgebenden *Systems*, eine Rekonfigurierung durchzuführen.
2. Diese Anfrage wird an den *Slotmanager* weitergeleitet. Die Slots berechnen die in der neuen Konfiguration von ihnen benötigten Bitstreams.
3. Weiterleitung der Anfrage an die *Multiplexersteuerung*. Hier werden die neuen Steuersignale für die Multiplexer berechnet.
4. Für alle Slots werden folgende Schritte nacheinander abgearbeitet.
   a) Anforderung eines benötigten Bitstreams vom Slotmanager.
   b) Bitstream beim *Speicher* anfordern.
   c) Die *Konfigurationsschnittstelle* aktivieren.
   d) Bitstream vom *Speicher* zur *Konfigurationsschnittstelle* übertragen.
5. Mitteilung an das *System*, dass die Rekonfigurierung abgeschlossen ist.

Die Steuerung dieses Ablaufs ist die Aufgabe der *zentrale Steuerung*. In dieser muss der in Abbildung 6.20 aufgezeigte Ablaufplan als Automat realisiert werden. Der Automat wird über die Signale der restlichen Komponenten der RCU gesteuert. Die sich daraus ergebende Schnittstelle ist in Abbildung 6.21 zu sehen. Zur Kommunikation mit der Systemschnittstelle werden zwei

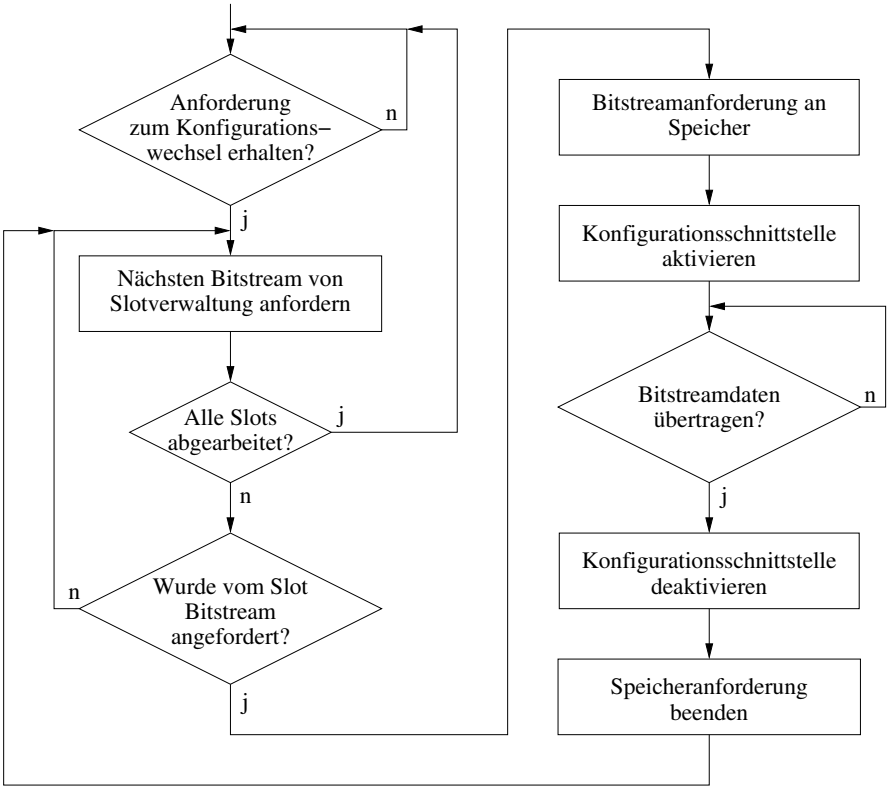


Abbildung 6.20: Ablauf der zentralen Steuerung

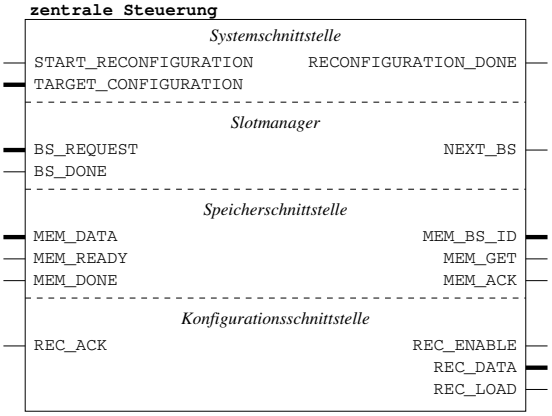


Abbildung 6.21: Schnittstelle der zentralen Steuerung der RCU

Signale, START_RECONFIGURATION und RECONFIGURATION_DONE benötigt, die anzeigen ob ein Konfigurationswechsel durchgeführt werden soll (1)[3] bzw. dass der Konfigurationswechsel abgeschlossen ist (5). Wenn eine Rekonfigurierung durchgeführt werden soll (2), ist START_RECONFIGURATION der zentralen Steuerung und auch der Eingang RECONFIGURE des Slotmanager gleich 1. Die daraus sich stellenden Anfragen der Slots werden über den Eingang BS_REQUEST der zentralen Steuerung mitgeteilt. Mit dem Signal NEXT_BS wird die nächste Anfrage abgearbeitet. Da die Bitstreams nur nacheinander geladen werden können, werden auch die Anfragen der Slots sequentiell bearbeitet. Der Slotmanager teilt der zentralen Steuerung außerdem über das Signal BS_DONE mit, ob alle Anfragen der Slots abgearbeitet wurden (4a). Mit diesem Konzept ist die Schnittstelle zwischen dem Slotmanager und der zentralen Steuerung unabhängig von der Anzahl der Slots.

Zur Steuerung der Speicherschnittstelle wird ein Signal zur Anforderung eines neuen Bitstreams durch die zentrale Steuerung am Ausgang MEM_GET erzeugt. Der Speicher signalisiert mit BS_DONE der Steuerung (Eingang MEM_DONE), dass der Bitstream vollständig übertragen wurde (4b). Für die Übertragung der Bitstreamdaten zur Konfigurationsschnittstelle existieren zwei weitere Signale, MEM_READY und MEM_ACK, die anzeigen wenn ein neues Datenwort aus dem Speicher zur Verfügung steht bzw. den Empfang des Datenwortes bestätigen (4d). Die Aktivierung der Konfigurationsschnittstelle wird über das Steuersignal REC_ENABLE gesteuert (4c).

Mit Vorstellung der zentralen Steuerung, kann auch der Zusammenhang mit weiteren Komponenten der RCU beschrieben werden. So ergibt sich beim Laden eines Bitstreams zwischen der Speicherschnittstelle, der Konfigurationsschnittstelle und der zentralen Steuerung, der in Abbildung 6.22 dargestellte Signalverlauf, mit folgenden markierten Schritten:

1. Anforderung eines Bitstreams bei der Speicherschnittstelle durch Anlegen einer 1 an bs_get und der BitstreamID an bs_id.
2. Die Konfigurationsschnittstelle wird durch eine 1 an enable aktiviert.
3. Ein Datenwort der Bitstreamdaten steht zur Verfügung. Dies wird durch eine 1 an data_ready signalisiert. Die zentrale Steuerung setzt daraufhin über das Signal REC_LOAD den load-Eingang der Konfigurationsschnittstelle ebenfalls auf 1. Die Bitstreamdaten werden an die Konfigurationsschnittstelle weitergeleitet.
4. Die Konfigurationsschnittstelle zeigt durch eine 1 an ack an, dass die Daten an den FPGA übertragen wurden. Dieses Signal wird an die Speicherschnittstelle als data_ack weitergeleitet.
5. Die Daten des Bitstreams wurden vollständig übertragen. Die Speicherschnittstelle setzt das Signal bs_done auf 1.
6. Das Laden des Bitstreams wird durch das Deaktivieren der Konfigurationsschnittstelle abgeschlossen.

Signalleitungen der RCU

Die in den vorangegangen Abschnitten beschriebenen Schnittstellen der RCU Komponenten und der zentrale Steuerung werden, wie in Abbildung 6.23 aufgezeigt, zu einem RCU Gesamt-

[3] Die Nummern in den Klammern beziehen sich auf den in Abbildung 6.19 dargestellten Ablauf.

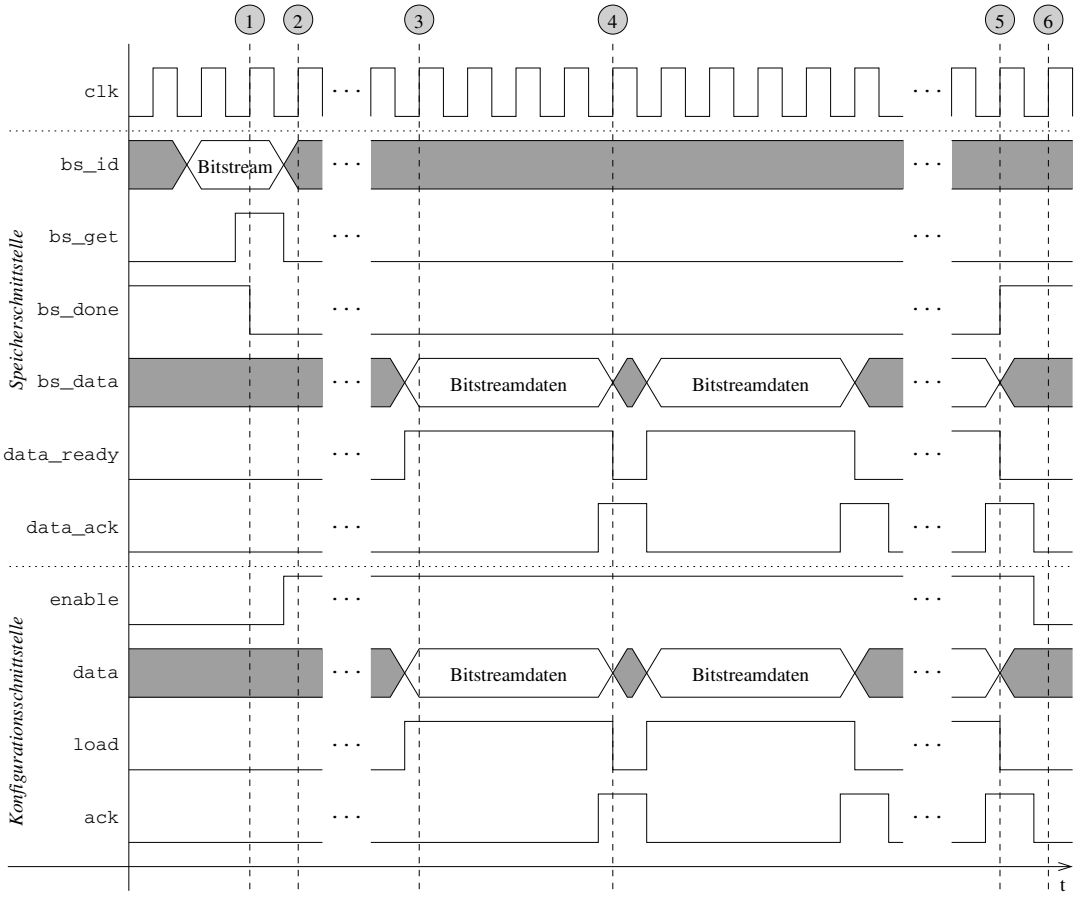


Abbildung 6.22: Signalverlauf beim Laden eines Bitstreams

system miteinander verbunden. Der modulare Aufbau ermöglicht eine einfache Anpassung an neue FPGAs.

6.3 Platzbedarf im FPGA

Eine in Hardware realisierte RCU ist nur insoweit sinnvoll, wenn zum Beispiel ein Mikroprozessor mehr Platz im FPGA einnimmt als die oben vorgestellte automatisch generierbare Steuerung. Ein im FPGA abgebildeter Prozessor benötigt relativ viel Platz (MicroBlaze: ca. 1.000 Slices) und stellt unter Umständen Funktionen bereit, die für die Steuerung der Rekonfigurierungen nicht benötigt werden. Daher wird im Folgenden der Platzbedarf des RCU Konzeptes näher betrachtet.

Der benötigte Platz auf dem FPGA hängt von der Komplexität der einzelnen RCU Komponenten ab. Ein Teil der Komponenten sorgt für eine gewisse Mindestgröße, da diese systemunabhängig sind und in allen Steuerungen enthalten sind. Zu diesen Komponenten zählen die *Konfigurationsschnittstelle*, die *Speicherschnittstelle* und die *zentrale Steuerung*. Die Größen dieser Komponenten, bei einer Abbildung auf einen Virtex II Pro FPGA (XC2VP30-6FF896), wurden über die Synthese des VHDL-Codes ermittelt und sind in Tabelle 6.2 aufgelistet. Die

Komponente	LUTs	Flip Flops	Slices
Konfigurationsschnittstelle	31	21	17
Speicherschnittstelle	268	172	168
zentrale Steuerung	10	7	6
gesamt	309	200	191

Tabelle 6.2: Größe der systemunabhängigen RCU-Komponenten

Anzahl der Konfigurationen, Slots, Module beziehungsweise Bitstreams und Multiplexer eines eingebetteten, dynamisch rekonfigurierbaren Systems beeinflussen die Größen der restlichen Komponenten der RCU. Daher lässt sich die Anzahl der von diesen Komponenten benötigten LUTs nur schwer abschätzen. Um diesem Problem zu begegnen, werden im Folgenden die verwendeten Flip Flops und damit die minimal benötigten Slices betrachtet. Ein Slice beinhaltet je zwei Flip Flops und LUTs (siehe Abschnitt 2.2). Da die Slices ohnehin für die Speicherzellen benötigt werden, können die LUTs für einen Teil der Überführungslogik der Automaten verwendeten werden. Die Erfahrung hat gezeigt, dass für die Logik eines Automaten etwa die selbe Menge an Slices wie Flip Flops benötigt werden.

Im Folgenden sind die Abschätzungen für die benötigten Flip Flops aufgezeigt. Für das Register $KonfReg$ zur Speicherung der aktuellen Konfiguration werden, entsprechend der Anzahl Konfigurationen, die in Gleichung 6.4 beschreibende Menge $FF_{KonfReg}$ an Flip Flops benötigt. Der Slotmanager besteht aus einem Automaten und einem Register für jeden Slot. Für das Bitstreamregister werden daher die in Gleichung 6.5 angegebene Anzahl Flip Flops notwendig. Falls die Zuordnung der Module zu den Slots noch nicht bekannt ist, lässt sich die Anzahl der Bitstreams durch die Gleichung 6.6 nach oben abschätzen, da jedes Modul in jedem Slot platziert werden kann und sich in einem Slot zu einem bestimmten Zeitpunkt nur ein Modul befinden kann. Der Slotautomat kann drei Zustände annehmen, daher werden für jeden

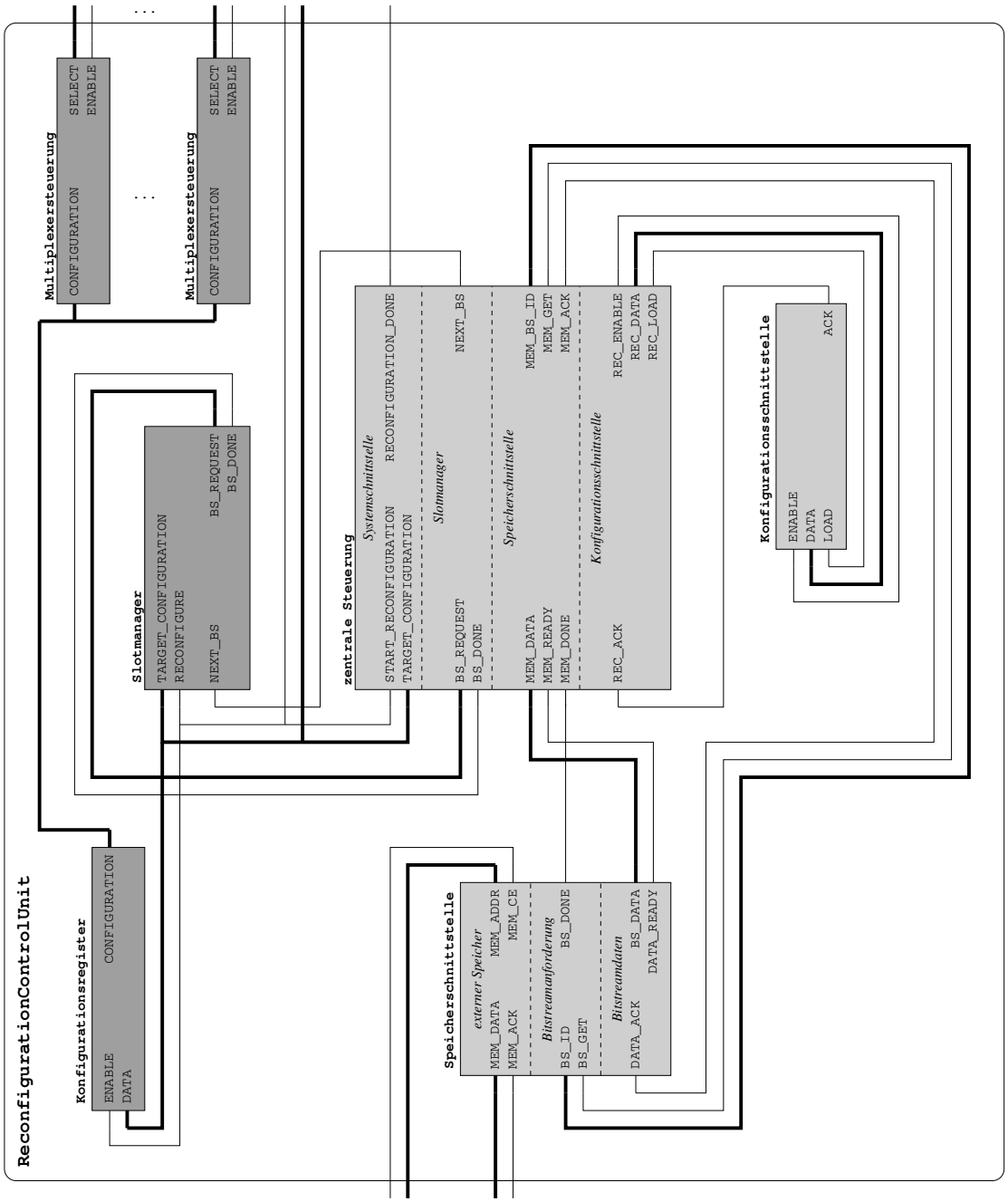


Abbildung 6.23: Verbindungen der RCU Komponenten

Slot noch zwei Flip Flops gebraucht (siehe Gleichung 6.7). Für den Automaten der Abfrageeinheit im Slotmanager existieren zwei Zustände, die zur Ausgabe der angeforderte Bitstream und als Wartezustand benötigt werden. Daraus resultieren für die Abfrageeinheit die in Gleichung 6.8 angegebene Menge an Flip Flops $FF_{Abfrage}$ und insgesamt für den Slotmanager die in Gleichung 6.9 Flip Flop Anzahl $FF_{Slotmanager}$.

$$FF_{KonfReg} = \lceil \log_2(\#Konf) \rceil \tag{6.4}$$

$$FF_{BitstreamReg} = \lceil \log_2(\#Bitstream) \rceil \tag{6.5}$$

$$\#Bitstream \leq \min\{\#Slot \cdot \#Mod, \#Slot \cdot \#Konf\} \tag{6.6}$$

$$FF_{Slot} = 2 \tag{6.7}$$

$$FF_{Abfrage} = \lceil \log_2(\#Slot + 1) \rceil \tag{6.8}$$

$$FF_{Slotmanager} = \#Slot \cdot (FF_{BitstreamReg} + FF_{Slot}) + FF_{Abfrage} \tag{6.9}$$

Neben der Abschätzung des Platzbedarfs für die systemabhängigen Komponenten wurden, zur Ermittlung des realen Platzbedarfs der RCU, verschiedene Systembeschreibungen mit unterschiedlichen Slotanzahlen und Bitstreams erzeugt. Die Anzahl der Slots variiert dabei zwischen 2 und 128. Um eine realistische Belegung der Slots zu erhalten, wurde die Anzahl der Bitstreams im System über einen Bitstreamfaktor k an die Anzahl der Slots gekoppelt. Damit wird vermieden, dass bei wenigen Slots und vielen Bitstreams sehr viele Konfigurationen entstehen und umgekehrt würde bei wenigen Bitstreams und vielen Slots nur eine Konfiguration mit vielen leeren Slots entstehen.

Die Konfigurationen des Systems ergeben sich aus dem Umstand, dass alle Bitstreams auf die Slots verteilt werden müssen. Es wurden zwei Varianten zur Belegung der Slots untersucht. Zum einen wurde den Slots in *jeder* Konfiguration ein Bitstream zugewiesen, wodurch die Anzahl der Konfigurationen dem Bitstreamfaktor k entspricht. In der zweiten Variante wurde einem Slot mit einer gewissen Wahrscheinlichkeit ein Bitstream zugewiesen. Dadurch kann ein Slot in einer bestimmten Konfiguration auch keinen Bitstream enthalten und es ergeben sich mehr als k Konfigurationen, um alle Bitstreams zu verteilen.

Die erstellten Testsysteme wurden mit Multiplexer für jeden Slot, die von der RCU gesteuert werden, ausgestattet. Dies stellt die Obergrenze für die Anzahl der Multiplexer dar. In diesen Systemen kann über die Multiplexer von jedem Slot zu jedem anderen Slot kommuniziert werden (siehe Abbildung 6.24). Die Steuereingänge der Multiplexer wurden in jeder Konfiguration zufällig belegt. Für jede Kombination der Parameter *Slotanzahl* und *Bitstreamfaktor* wurden drei Systembeschreibungen erzeugt und jedes dieser Systeme dann mit dem entwickelten RCU Generator die VHDL-Beschreibung erstellt und mit XST Version 9.1 für den Virtex II Pro FPGA XC2PV30 synthetisiert. Als Maß für den Platzbedarf der RCU, wurde die Anzahl der benutzten FPGA-Slices angegeben. Um Schwankungen durch die zufällige Belegung der Slots auszugleichen, wurde jeweils der Median der belegten Slices der drei Systeme mit gleichen Parametern als Wert in die Tabelle 6.3 und in die Abbildungen 6.25 und 6.26 eingetragen.

Bei den Systemvarianten, die nicht in allen Konfigurationen belegte Slots aufweisen, belegt, in den meisten Fällen, die RCU mehr Fläche im FPGA als bei den Varianten, bei denen die Slots immer belegt sind. Dies ist dadurch zu erklären, dass in diesem Fall mehr Konfigurationen entstehen und dadurch in den Slotautomaten mehr Logikfunktionen benötigt werden, um den jeweils benötigten Bitstream zu berechnen. Weiterhin ist zu beobachten, dass mit steigender Slotanzahl und damit steigender Anzahl an Bitstreams die Abweichung zwischen den tatsächlichen und den abgeschätzten Werten größer wird. Dies ist ebenfalls auf den erhöhten Aufwand für die

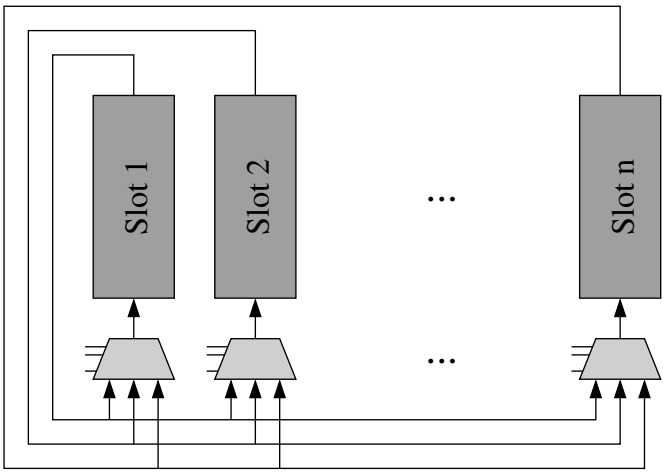


Abbildung 6.24: Multiplexer der Testsysteme

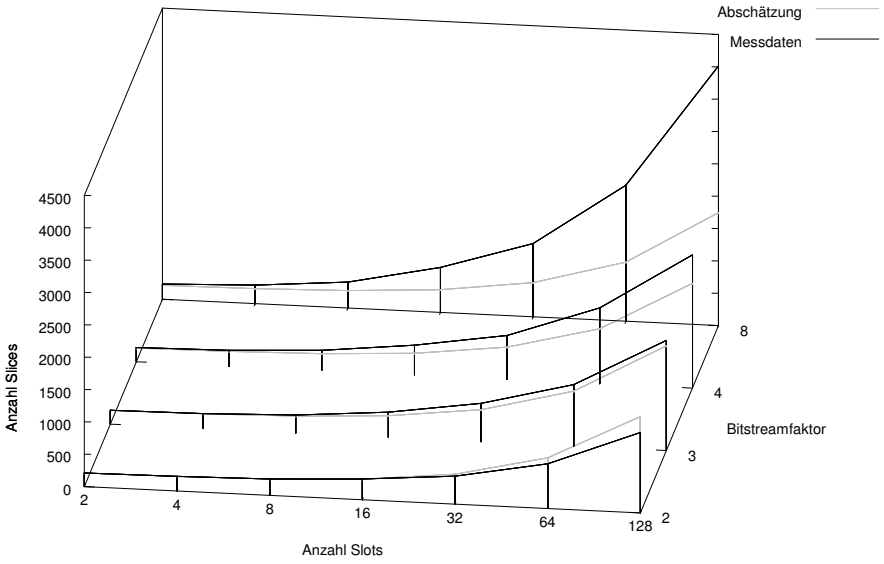


Abbildung 6.25: Platzbedarf der RCU bei teilweise nicht belegten Slots der Testsysteme

Bitstreamfaktor, Variante	Slots						
	2	4	8	16	32	64	128
2, alle Slots belegt	212	228	260	321	414	666	1.159
2, freie Slots	211	226	253	321	433	692	1.244
2, Abschätzung	211	224	253	318	463	784	1.489
3, alle Slots belegt	213	232	270	361	510	835	1.494
3, freie Slots	216	233	278	388	593	951	1.700
3, Abschätzung	214	229	262	335	496	849	1.618
4, alle Slots belegt	219	242	287	400	557	941	1.695
4, freie Slots	222	245	312	460	678	1.172	2.059
4, Abschätzung	214	229	262	335	496	849	1.618
8, alle Slots belegt	223	253	309	461	706	1.204	2.240
8, freie Slots	236	288	403	697	1.134	2.102	4.014
8, Abschätzung	217	234	271	352	529	914	1.747

Tabelle 6.3: Platzbedarf der RCU in Slices

Berechnung der angeforderten Bitstreams in den Slotautomaten zurückzuführen. Deutlich ist dieser Effekt in der Variante mit freien Slots bei einem Bitstreamfaktor von 8 und 128 Slots sichtbar. Hier ist der tatsächliche Platzbedarf mehr als doppelt so groß wie die Abschätzung.

Für eine kleinere Anzahl von Slots und Bitstreams bieten die abgeschätzten Werte aber einen guten Anhaltspunkt für den Platzbedarf der RCU. Eine Partitionierung des FPGA in mehr als 32 Slots ist ohnehin nicht sinnvoll, da dann in den einzelnen Slots nicht genügend Platz für die gewünschte Funktionalität verbleibt. Der in dieser Arbeit verwendete FPGA XC2VP30 besitzt 13.696 Slices. Würde dieser FPGA in 128 Slots aufgeteilt, so blieben für die einzelnen Slots im Mittel nur 107 Slices übrig, was 0,625 CLB Spalten entsprechen würde. Eine spaltenweise Rekonfigurierung wäre in diesem Fall nicht realisierbar. Realistische Slotanzahlen liegen, bei einer Größe von 80x46 CLBs des zur Verfügung stehenden FPGAs, im Bereich bis 32. Damit verbleiben pro Slot etwa 400 Slices und die RCU belegt dann ihrerseits etwa 500 bis 600 Slices, was ca. 5% der Fläche des XC2VP30 FPGA entspricht. Für die Systeme mit wenigen Slots wird weniger FPGA Fläche von der vorgestellten RCU belegt als bei einer Steuerung über einen eingebetteten Prozessor, die allein im Fall eines MicroBlaze [138] bereits ca. 1.000 Slices belegt. Der Anforderung aus dem Abschnitt 6.1, nach einem möglichst geringen Platzbedarf, wird durch den Platzvorteil der entworfenen RCU weitestgehend entsprochen.

6.4 Generierung der RCU

Die beschriebene Steuereinheit besteht aus systemabhängigen und -unabhängigen Komponenten. Daraus ergibt sich die Notwendigkeit, dass für jedes neu zu erstellende, eingebettete System auch eine neue RCU zu generieren ist. Um diesen Prozess dem Entwickler weitestgehend abzunehmen, wird im Folgenden auf die im Rahmen dieser Arbeit entwickelte RCU Generierungssoftware eingegangen.

Die Software zur Erzeugung der RCU wurde in JAVA [104] implementiert und führt drei

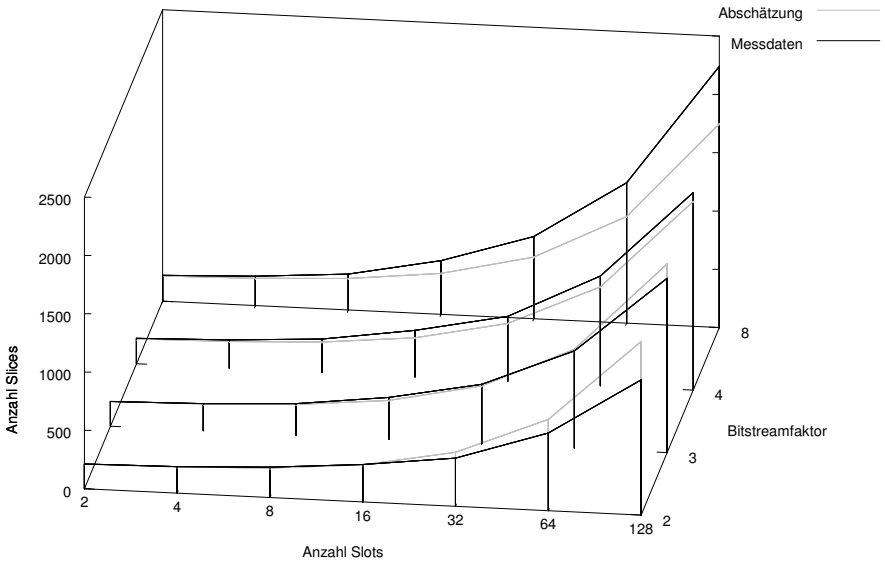


Abbildung 6.26: Platzbedarf der RCU bei Belegung aller Slots in allen Konfigurationen

Aufgaben hintereinander aus. In einem ersten Schritt sind die Eingabedaten des eingebetteten, rekonfigurierbaren Systems einzulesen und die für den Slotmanager notwendigen Informationen zu berechnen. Als zweites werden danach, in Abhängigkeit dieser Informationen, die Komponenten der RCU erzeugt. Die abschließende Aufgabe der Software ist es, wenn die Bitstreams vorliegen, das Speicherabbild für die Speicherverwaltung zu erzeugen.

Einlesen der Systembeschreibung

Wie schon in den Abschnitten 4.1 und 5.3.3 erwähnt, wurden die Beschreibung des Systems und die generierten Informationen für die Rekonfigurierung während des Design Flows in einer XML Datei gespeichert. Diese Systembeschreibung dient als Eingabe bei der Software zur Generierung der RCU. Zum Einlesen der Daten wird der SAX-Parser [84] eingesetzt. Die Daten werden zur weiteren Verarbeitung auf eine interne Objektstruktur abgebildet. Nachdem die Daten eingelesen wurden, werden aus der Zuordnung der Module zu Slots in Abhängigkeit der Konfigurationen die Menge der Bitstreams bestimmt.

Erzeugung der Hardwarebeschreibung

Für die Generierung der RCU Komponenten wurden Klassen entwickelt, die alle von allgemeinen Klassen abgeleitet sind und die Fähigkeit besitzen VHDL-Quellcode zu erzeugen. Da eine VHDL-Beschreibung hierarchisch strukturiert sein kann, d. h. eine Komponente besteht ihrerseits wieder aus Unterkomponenten, wurde die Erzeugung der Hardwarekomponenten an diese Eigenschaft angelehnt. Mit der Klasse *RCUTopLevel* wird die VHDL Generierung gesteuert. Diese Klasse erzeugt eine Beschreibung, in welcher alle Komponenten der RCU als Untermodule integriert und miteinander verbunden sind. Hierfür benötigt die Klasse die Systembeschreibung mit der Bitstreammenge, die Liste mit den Multiplexeransteuerungen, die initiale Konfiguration und den Offset der Bitstreams im Speicher. Mit Hilfe dieser Informationen,

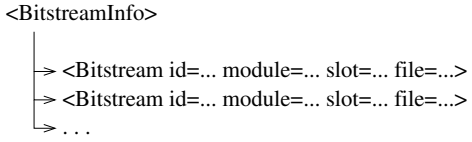


Abbildung 6.27: Struktur der Bitstreambeschreibung

werden dann über Unterklassen die Komponenten für den Slotmanager, die zentrale Steuerung, die Speicherschnittstelle, die Konfigurationsschnittstelle und die Steuerung der Multiplexer erzeugt.

Erzeugung des Speicherabbildes

Da die Größen der Bitstream bei der Erzeugung der RCU noch nicht bekannt sind, wird eine XML-Datei erzeugt, in der alle benötigten Bitstreams aufgeführt sind. Die Struktur der Datei ist in Abbildung 6.27 dargestellt. Hier ist zu sehen, das für jeden Bitstream vermerkt ist, zu welcher Modul-Slot-Kombination dieser gehört. Die Datei wird bei der späteren Erzeugung des Systems ausgewertet. Nach der Erstellung der Bitstreams müssen die Dateinamen der Dateien, die die Bitstreams enthalten, in die `file`-Attribute der Bitstreams eingetragen werden, um die, in Abschnitt 6.2.4 beschriebene Speicherstruktur zu erzeugen.

6.5 Zusammenfassung

Jede Rekonfigurierung besteht aus mehreren Schritten und muss daher gesteuert werden. Unter Berücksichtigung des Rekonfigurierungskonzeptes ergeben sich klare Anforderungen an eine Steuereinheit. Die in Abschnitt 6.1 vorgestellten Eigenschaften beziehen sich im Besonderen auf eingebettete Systeme, wodurch eine Selbstrekonfigurierung fokussiert wird. Im Vergleich zu einer prozessorgesteuerten Rekonfigurierung hat sich das vorgestellte RCU Konzept als günstiger, in Bezug auf den Platzbedarf im FPGA, herausgestellt. Die in Hardware realisierte Steuerung erfordert auch keinen zusätzlichen Aufwand beim Entwurf, da eine automatisierte Generierung entwickelt werden konnte. Durch die Verwendung des Overlaying Konzepts bei abgeschlossenen eingebetteten Systemen wurde die Entscheidung getroffen, die Multiplexer über die RCU zu steuern. Die RCU wurde darüber hinaus modular aufgebaut, damit technologieabhängige Komponenten, wie z.B. die Speicherschnittstelle, leicht ausgetauscht werden können.

Eine besonders hervorzuhebende Komponente der Steuerung ist der Slotmanager. Durch diesen wird der aktuelle Zustand des Systems verwaltet und unnötige Rekonfigurierungen von einzelnen Modulen unterdrückt. Um die Rekonfigurierungsdauer zu minimieren ist es sinnvoll, nur die Module nachzuladen, die in der Ausgangskonfiguration noch nicht auf dem FPGA vorhanden waren. Mit dem Slot Manager wird deshalb jeder Slot überwacht und benötigte Bitstreams ermittelt. Wird ein Bitstream und der entsprechende Slot in der aktuellen Konfiguration nicht benötigt, das bedeutet er gehört zu Menge $konf^+$, kann dieser Bitstream in einer späteren Konfiguration ohne nochmalige Rekonfigurierung benutzt werden. Dies erfordert natürlich eine Speicherung des aktuell geladenen Bitstreams im Slotmanager.

Die entwickelte Rekonfigurierungssteuerung wurde für mehrere Systeme automatisch generiert. Dabei wurden im Schnitt für Systeme mit bis zu 32 Slots 400 - 600 Slices benötigt, was

auf dem Xilinx FPGA XC2VP30 ca. 5% der Fläche bedeutet.

7 Implementierung der Methodik

In diesem Kapitel wird ein Überblick über die Designentscheidungen der entwickelten Software gegeben. Der in Abschnitt 2.3.2 vorgestellte Design Flow für dynamisch eingebettete Systeme zeigt mehrere, speziell für dynamische Rekonfigurierung notwendige Schritte auf, die mittels Software automatisiert wurden. Das Zusammenspiel diese Softwarekomponenten und auch die entwickelte System Design Gate (SDG) Oberfläche werden im Folgenden vorgestellt. Im folgenden Abschnitt 7.1 wird die Implementierung einer grafischen Oberfläche zur Eingabe und Spezifikation von dynamisch rekonfigurierbaren Systemen motiviert. Des Weiteren werden Randbedingungen und vorhandene Werkzeuge, die ebenfalls eingesetzt werden sollen, dargestellt.

Der Hauptteil dieses Kapitels bildet der Abschnitt 7.2, in welcher die SDG Software vorgestellt wird. Neben dem Aufbau der grafischen Oberfläche und deren Funktionen, wird auch auf das verwendete Softwarekonzept eingegangen.

7.1 Ausgangssituation und Ziel der Software

Ein wichtiger Punkt, bei der Integration von Rekonfigurierung in eingebettete Systeme, stellt die Entlastung des Entwicklers beim Entwurf eines solchen Systems dar. Mit der Automatisierung der zusätzlichen Entwurfsschritte, kann nicht nur Entwicklungszeit gespart, sondern es können auch potentielle Fehlerquellen ausgeschlossen werden. Daher wurde ein Werkzeug erstellt, das System Design Gate, mit welchem der in dieser Arbeit vorgestellte Design Flow gesteuert werden kann.

Um ein eingebettetes System auf höherer Abstraktionsebene automatisch in Module zu unterteilen, ist es notwendig, dass für die Partitionierungsalgorithmen die entsprechende Systembeschreibung digital vorliegt. Aufgabe der SDG Software ist es daher, dem Entwickler eine Unterstützung bei der Erstellung des Problemgraphen anzubieten. Dies beinhaltet im Besonderen die Möglichkeit der Definition von Systemelementen und Kommunikationsverbindungen. Für die Beschreibung der Systemelemente wird hierbei auf das IPQ Format zurückgegriffen, um die Suche nach IPs mit den vorhandenen Werkzeugen des IPQ Projekts durchführen zu können. Neben der Eingabe der Systemelemente ist, für eine dynamische Rekonfigurierung, noch die Definition von Konfigurationen essenziell. Für diese Aufgaben ist eine entsprechende grafische Oberfläche (GUI) notwendig.

Die Steuerung der verschiedenen Entwurfsschritte ist ein weiterer Teil im Konzept der Software. Hierbei spielt nicht nur das Auslösen der verschiedene Algorithmen eine Rolle, sondern auch die Darstellung der Zwischenergebnisse. Dadurch kann ein Entwickler, zu einem frühen Zeitpunkt der Systementwicklung, Entwurfsergebnisse analysieren und modifizieren.

Mit der SDG Software sind im Endergebnis alle Daten, für eine Erstellung der rekonfigurierbaren Bitstreams mit Synthesewerkzeugen der Firma Xilinx, bereitzustellen. Diese Werkzeuge können dann, unabhängig von der SDG Software, über einfache Skriptbefehle angesteuert werden. Die SDG Software dient nicht der VHDL Bearbeitung.

7.2 System Design Gate

Durch die Entscheidung, einen Design Flow für *IP basierte*, eingebettete Systeme zu entwickeln und die IPQ Werkzeuge für die Unterstützung der Suche von Systemfunktionalitäten zu verwenden, wurden Eigenschaften der Softwareplattform CAMP (Common Application Module Platform) bei der Erstellung des Softwarekonzepts berücksichtigt.

CAMP

CAMP wurde im Rahmen des IP Projekts entwickelt, um die verschiedenen Werkzeuge, zum Beispiel für die Bearbeitung von IP Beschreibungen in XML oder auch der IP Suche, zu integrieren. Die Plattform ist eine in Java entwickelte Software, in welche Softwaremodule über ein Plug-In Konzept eingebunden und angesteuert beziehungsweise ausgeführt werden können. Hierfür werden verschiedene Basisfunktionen bereitgestellt:

- grundlegende grafische Oberflächenstruktur
- Verwaltung von GUI-Events
- Funktionalitäten zum Anlegen, Öffnen, Speichern und Schließen von Dateien
- umfangreiche Konfigurationsmöglichkeiten für die integrierten Module und deren Abhängigkeiten
- Schnittstelle für die Bereitstellung von Diensten zwischen verschiedenen Softwaremodulen

Die Softwareplattform wurde an der UNI-Paderborn und der TU-Chemnitz[1] entwickelt [128] und ist nach dem Model-View-Controller [40, 39] (MVC) Konzept aufgebaut.

Ein Modul, das für die SDG Software benutzt wurde, ist die XML-API [128]. Die XML-API wurde ebenfalls im Rahmen des IPQ Projekts entwickelt. Sie fasst über eine Softwareschicht mit einer abgegrenzten Schnittstelle, gegen die programmiert wird, mehrere APIs zusammen. Hierzu gehören Xerces [109] zum Parsen, Xalan [108] für XPath[2] Funktionalität und die Verarbeitung der DOM[3] Struktur und die XSD-API zur XML Schemaverarbeitung. Über die angebotene Schnittstelle können XML Instanzen eingelesen, bearbeitet und traversiert werden.

Im Kontext der dynamischen Rekonfigurierung wird weiterhin das Modul XML-Editor zur Spezifizierung der Systemelemente eingesetzt, um die im IPQ Projekt entwickelten Methoden zur Suche von IPs direkt einsetzten zu können. Mit dem XML-Editor können XML-Schemas hinterlegt und eine entsprechende XML-Instanz durch „Drag & Drop“ erstellt werden [33]. Der Editor bietet hierfür grundlegende Operationen zur Bearbeitung von XML-Dokumenten.

[1] Unter anderem wurde an der TU-Chemnitz die CAMP-Implementierung im Rahmen des Projektes CompA [42] auf die Java Version 1.6 portiert.

[2] Xpath ist eine vom W3-Konsortium entwickelte Abfragesprache, um Teile eines XML-Dokumentes zu adressieren.

[3] DOM bedeutet Document Object Model und ist eine Spezifikation einer Schnittstelle für den Zugriff auf HTML- oder XML-Dokumente.

SDG

Die SDG Software ist so entworfen worden, dass es leicht in CAMP als ein Modul integrierbar ist. Da eine Bindung an CAMP nicht notwendig ist, um die XML-API zu nutzen, ist die SDG Software, zur besseren Analyse im Debug Modus, auch als eigenständige Software lauffähig. Analog zu CAMP wurde bei SDG ebenfalls das Model-View-Controller Konzept angewandt.

Für das Datenmodel war es wichtig, dass verschiedene Module die Daten bearbeiten oder nutzen können. Im Besonderen sollten die Werkzeuge des IPQ Projekts verwendet werden können, um Systemelementsuchanfragen zu beschreiben und durchzuführen. Dadurch wurde, wie auch schon in Kapitel 4 und 5 gezeigt, für das Model des MVC auf XML gesetzt. Ein Ausschnitt des XML Schemas für den Problem- und Architekturgraphen wurde in Abbildung 4.3 (siehe Abschnitt 4.1) vorgestellt. Das Datenmodel beinhaltet alle für die dynamische Rekonfigurierung erzeugten Daten des Systempartitionierungsprozesses, die Modul- und Konfigurationsbeschreibungen, und auch der Platzierung, den Slotgraph und die Multiplexerdefinitionen. Der entsprechende Ausschnitt des XML Schemas ist in Abbildung 7.1 zu sehen. In dem Zweig *Slotgraph* dieses XML Schemas ist die eindeutige Modul-Slot Abbildung in einer Konfiguration bis zu den Blättern des Zweiges dargestellt.

Da die Erstellung und Bearbeitung eines Systems über den XML-Editor nicht übersichtlich ist, wurde eine Oberfläche für die Erstellung von Problemgraphen entworfen. Mit dieser Oberfläche können unter anderem die Systemelemente und Kommunikationsverbindungen des Problemgraphen spezifiziert und in einer entsprechenden XML Datei abgespeichert werden. Ein Screenshot der GUI ist in Abbildung 7.2 zu sehen. Die GUI unterteilt sich in mehrere Bereiche:

- Im oberen Bereich des Fensters ist das Menü. Hier werden die Standard Funktionen wie Öffnen und Speichern von Dateien bereitgestellt und auch Steuerungsfunktionen für den Design Flow.
- Direkt darunter ist eine Karteikartenauswahl, um zwischen verschiedenen Sichten, auf das entworfene System, zu wechseln. In Abhängigkeit der jeweiligen Sicht werden auch die drei unteren Bereiche angepasst.
- Der Hauptbereich in der Mitte bildet die Arbeitsfläche, auf der die Graphen dargestellt werden.
- Links sind mehrere Werkzeuge zur Erzeugung, zum Löschen und auch zum Kopieren von Systemelementen bzw. Kanten des Problemgraphen auswählbar.
- Informationen zum System, wie die Menge der Konfigurationen oder auch Eigenschaften eines ausgewählten Elements, sind im rechten Informationsbereich dargestellt.

In der Abbildung 7.2 ist auf der Arbeitsfläche ein Problemgraph bestehend aus 13 Systemelementen zu sehen. Mittels eines Doppelklicks auf ein Systemelement öffnet sich ein weiteres Fenster, über das Eigenschaften des Elements, wie zum Beispiel der Anzeigenamen[4] bearbeitet werden können. Dieses Fenster dient auch zur Zuordnung einer entsprechenden IP zu dem markierten Systemelement, und dadurch der Definition des Architekturgraphen. Für das abgebildete Beispielsystem wurden vier Konfigurationen definiert. Im Informationsbereich wurde die zweite Konfiguration ausgewählt, was eine Markierung der zugehörigen Systemelemente auf der Arbeitsfläche bewirkte.

[4]Die Anzeigenamen der Systemelemente sind hier IP 1 bis IP 13

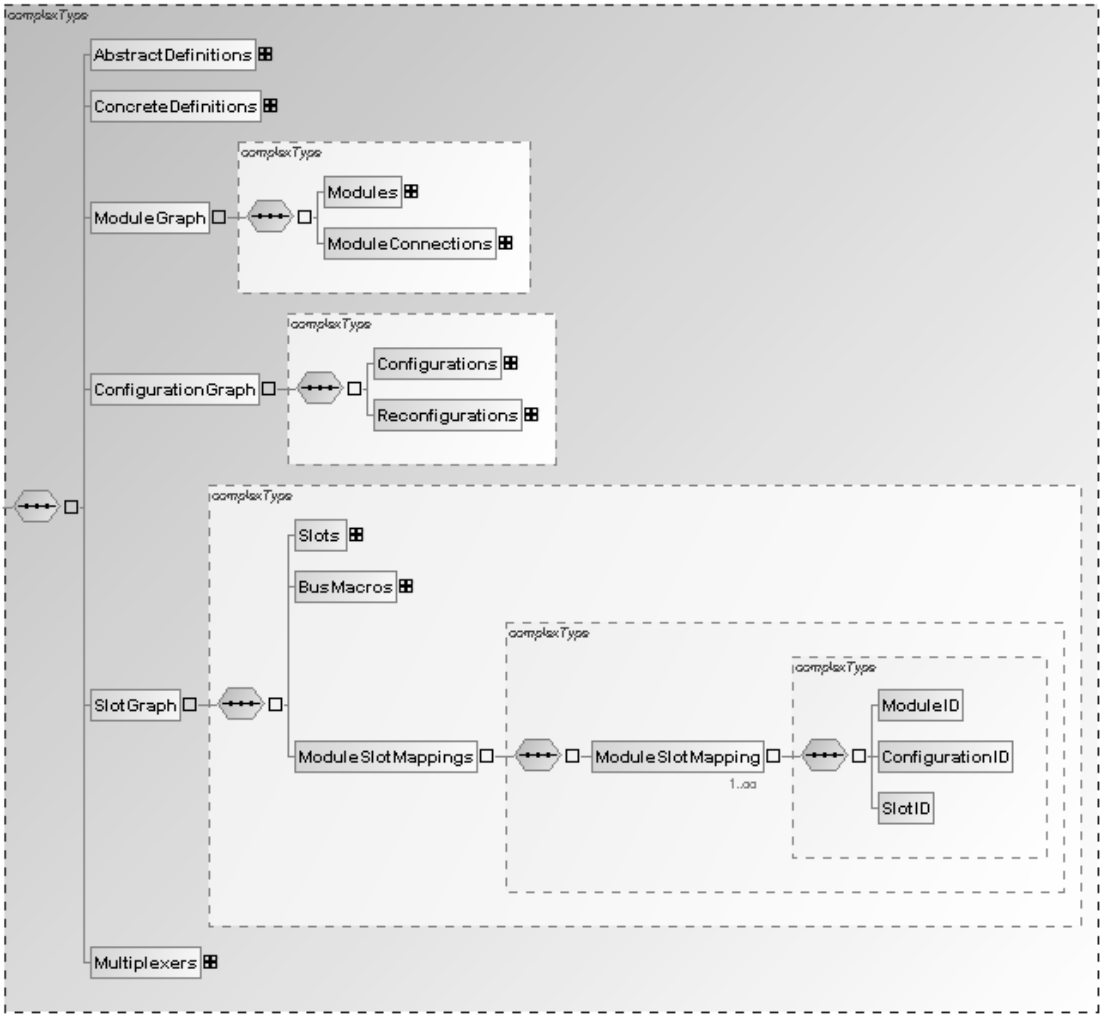


Abbildung 7.1: Ausschnitt des XML Schema für notwendige Rekonfigurierungsdaten

Die Erstellung und Verwaltung der Konfigurationen kann entweder über das Menü oder in der zweiten Karteikarte „Rekonfiguration“ erfolgen. Wählt man diese Karteikarte aus, erhält man die in Abbildung 7.3 dargestellte Bedienoberfläche. Analog zu den Systemelementen können auch die Konfigurationen und auch die Konfigurationsübergänge durch ein mit Doppelklick aufgerufenes Fenster bearbeitet werden. Da die Eingabe der Übergangswahrscheinlichkeiten erst erfolgen kann wenn eine Kante existiert, welche dann über den Doppelklick bearbeitet werden kann, wurde, zur Erleichterung der Definition der Wahrscheinlichkeiten, eine tabellarische Eingabe bereitgestellt (siehe Abbildung 7.3).

Wird über den Menüpunkt Modulgraph die Systempartitionierung ausgelöst, wird nach Abschluss dieses Schrittes die Karteikarte Module erzeugt. Die einzelnen Elemente des Graphen lassen sich in gleicher Weise analysieren und bearbeiten wie auch bei den anderen Sichten auf das System. Nach Erzeugung der Module kann auch die „Optimierung“ durchgeführt werden. Die Optimierung umfasst alle Schritte der Platzierung und gibt eine tabellarische Übersicht über die Anzahl der Slots und der darauf abgebildeten Module dem Entwickler als Information aus.

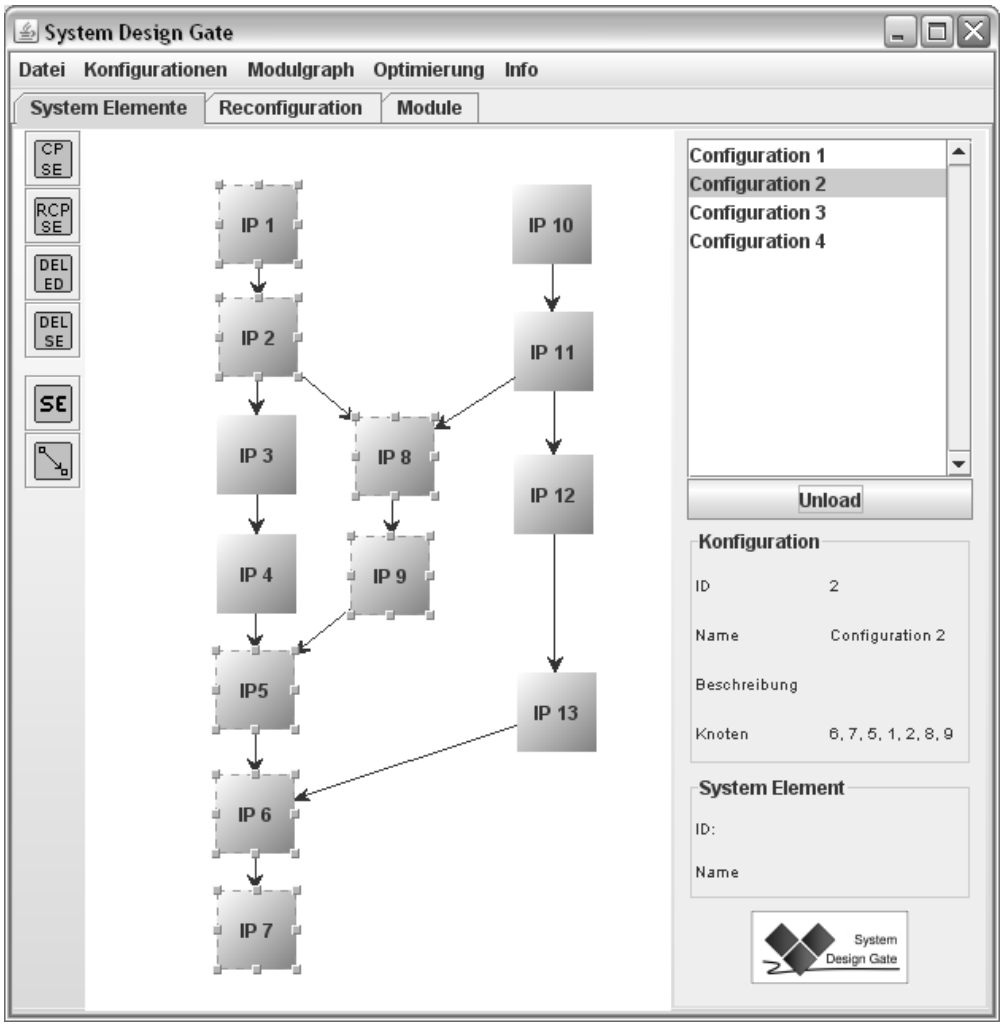


Abbildung 7.2: SDG Oberfläche mit Systemelementen und Kommunikationsverbindungen

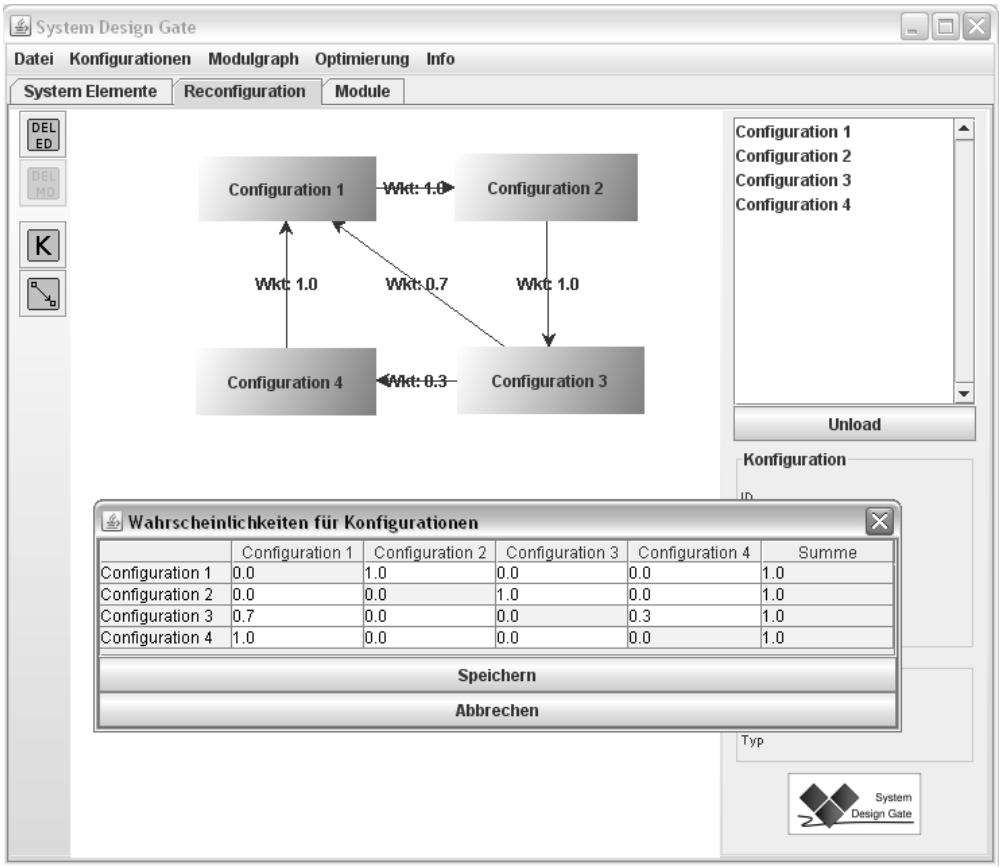


Abbildung 7.3: SDG Oberfläche mit Konfigurationsgraph

Die entwickelte Software diente lediglich der Evaluierung der entworfenen Methoden und beinhaltet daher nicht alle Softwaremodule zur Erstellung eines dynamisch rekonfigurierbaren Systems. So wurde die Generierung der RCU nicht in die SDG Software integriert, da für die Generierung keine Benutzerinteraktion beziehungsweise Visualisierung notwendig ist.

Weitere Implementierungen

An dieser Stelle seien noch zwei weitere Entwicklungen erwähnt, die im Rahmen dieser Arbeit implementiert wurden.

Mit der einen Entwicklung werden die IP beziehungsweise Modulgrößen bestimmt, wenn diese noch nicht in der IP Beschreibung enthalten ist. Die Software erzeugt hierfür als erstes ein VHDL Top Level für jedes Modul, das die VHDL Dateien der zugehörigen Systemelemente als Komponenten integriert. In einem nächsten Schritt wird unabhängig von dieser Software die Modulsynthese angestoßen und die Anzahl der benötigten CLBs im FPGA aus den Daten des *Synthesis Report* im Xilinx ISE Werkzeug bestimmt. Für die Modulsynthese sind neben den VHDL Beschreibungen auch Eigenschaften der Zielarchitektur anzugeben. Hierzu zählen insbesondere der FPGA Typ, aber auch Randbedingungen wie *Bounding Boxen*. Durch eine schrittweise Verkleinerung der Bounding Boxen und anschließender Modulsynthese ist die minimale Slotgröße für ein bestimmtes Modul ermittelbar. Die ermittelten Informationen über die Modulgrößen fließen dann in die Algorithmen der Platzierung ein. Zusätzlich zu diesen Schwerpunkt wird die Software auch zur Generierung der Top Levels mit eingebundenem Bus Makro für die einzelnen Bitstreams eingesetzt.

Die zweite Entwicklung dient der automatischen Generierung der Bus Makros. Ziel dieser Software ist es, automatisch aus den Daten des Slotgraphen die benötigten Bus Makro Verbindungen zu generieren. Dazu nutzt diese die XDL Resource Report Dateien, die vom XDL-Tool von Xilinx im Report-Modus erzeugt werden, als FPGA Beschreibung. Die Dateien beschreiben den hierarchischen Aufbau des FPGAs mit all seinen low-level Komponenten, wie Tiles, Sites oder auch Pins, und allen seinen Verdrahtungen. Für jede Kommunikationsverbindung zwischen zwei Slots wird nun eine passende Verdrahtung in der FPGA Beschreibung gesucht und als Hardmakro gespeichert. Dabei sind die Slotpositionen zu berücksichtigen und geeignete Routen zu bestimmen. Alle Leitungen ergeben dann ein Bus Makro, das in VHDL als Komponente eingebunden werden kann.

7.3 Zusammenfassung

In diesem Kapitel wurde die im Rahmen dieser Arbeit entwickelte SDG Software vorgestellt. Zusätzlich wurde die Software zur Bestimmung von Modulgrößen im FPGA kurz erläutert und Details zur automatischen Generierung von Bus Makros angegeben.

Die SDG Software dient der Erstellung einer Systembeschreibung in XML und der Ansteuerung der Design Flow Phasen. Über die grafische Oberfläche lässt sich der Problemgraph aus Systemelementen und Kommunikationsverbindungen konstruieren. Die einzelnen Elemente können mit Attributen versehen und einer entsprechenden Konfiguration zugeordnet werden. Für die Verwaltung der Konfigurationen ist ebenfalls eine GUI, zur übersichtlichen Darstellung der Rekonfigurierungsmöglichkeiten, implementiert worden. Ein weiteres wichtiges Merkmal der Software ist die Ansteuerung der Partitionierung und der Platzierung. Die daraus resultierenden Ergebnisse werden ebenfalls in einer XML Datei gespeichert und dem Nutzer zur

Evaluierung angezeigt.

Das entwickelte Werkzeug wurde so entwickelt, dass es leicht in die Softwareumgebung CAMP integriert werden kann. Die verschiedenen Algorithmen des Design Flows können unabhängig ausgeführt werden und wurden mit der aus dem SDG stammenden Systembeschreibung getestet.

8 Zusammenfassung und Ausblick

8.1 Zusammenfassung

Mit der Möglichkeit der dynamischen Rekonfigurierung können Systeme bereitgestellt werden, deren Funktionalitäten eine Flexibilität, wie in Software programmierte Methoden, und eine Performance, wie in Hardware realisierte Funktionalitäten, aufweisen. Eine spezielle Klasse dieser Systeme, die besonders, aufgrund ihrer Beschreibungsart als Automaten, für den Einsatz von Konfigurationswechsel zur Laufzeit geeignet erscheinen, sind die eingebetteten Systeme. In der vorliegenden Arbeit wurde daher der Fokus auf diesen Anwendungsbereich gelegt. Bei eingebetteten Systemen kann die dynamische Rekonfigurierung zu folgenden Verbesserungen beitragen:

- Verwendung kleinerer FPGA Schaltkreise bei gleichem Funktionalitätsumfang.
- Verbesserte Robustheit gegenüber partiellen Systemausfällen durch Funktionsmigration.
- Einfachere Realisierung von Fernwartung zur Funktionserneuerung und -erweiterung.
- Möglichkeit Funktionen dynamisch anzufordern, im Sinne von Selbstorganisation.

Unabhängig von der gewünschten Verbesserung eines eingebetteten Systems, sind für die Integration von Rekonfigurierung zusätzliche Entwurfsschritte im Design Flow notwendig. Thema dieser Arbeit war es daher, diesen Design Flow vorzustellen und entsprechende Methoden für die rekonfigurierungsspezifischen Schritte zu entwickeln.

Neben den Syntheseschritten, die auch bei einem statischen System notwendig sind, haben sich drei Problembereiche herauskristallisiert. Der erste Bereich beinhaltet die Systemaufteilung in verschiedene austauschbare Module. Darauf aufbauend ist die Frage zu klären, auf welche Bereiche des FPGAs die Module sinnvoll abgebildet werden können. Hierbei ist auch die zeitliche Veränderung der Abbildung durch Rekonfigurierungen zu berücksichtigen. Mit dem dritten Problembereich wird der Fokus auf die Ablaufsteuerung einer Rekonfigurierung gelenkt.

8.1.1 Partitionierung

Um eine vergleichsweise effizient zu realisierende Partitionierung des Systems in rekonfigurierbare Module zu erhalten, wurde das Overlaying Verfahren aus dem Bereich der Speicherverwaltung für dynamische Rekonfigurierung adaptiert. Mit diesem Verfahren werden zwei Punkte für die Systemaufteilung essenziell.

- Aktuell nicht benötigte Funktionen werden auch nicht im Speicher beziehungsweise auf dem FPGA gehalten und
- verschiedene Module nutzen dieselben Hardwareressourcen.

Ausgehend von diesen Anforderungen hat sich herausgestellt, dass strukturelle Partitionierungsansätze auf unteren Entwurfsebenen nicht geeignet sind. Der Vorteil des, in dieser Arbeit vorgestellten, Verfahrens zu Partitionierung ist, dass der Entwickler sich nicht darum kümmern muss, wie das System aufgeteilt wird, sondern nur festgelegt, was funktional zusammenhängend auf dem FPGA geladen sein soll. Die zusammenhängenden Komponenten, bzw. IPs, definieren dann die einzelnen Konfigurationen des Systems. Aus diesen Konfigurationen lassen sich, über eine Äquivalenzklassenbildung, die rekonfigurierbaren Module und die benötigten Bus Makros automatisch bestimmen. Dieser Ansatz führte zu einer signifikanten Reduzierung der benötigten FPGA Fläche. Für das vorgestellte Beispielsystem konnte die benötigte FPGA Fläche um 57 % gegenüber des komplett platzierten Systems reduziert werden. Durch die Verwendung von XML können verschiedene, bestehende Werkzeuge, zum Beispiel aus dem IPQ Projekt, beim Entwurf eingesetzt werden.

8.1.2 Platzierung

Von den beiden Ansätzen, der dynamische Platzierung und statische Platzierung, ist für eingebettete Systeme der statische Ansatz geeignet. Hier kann wesentlich mehr Aufwand für eine effektivere Auslastung des FPGAs getrieben werden. Die Aufgabe der Platzierung war hierbei, für die Generierung der Kommunikationskanäle, des Top Level Designs und der weiteren Syntheseschritte die Overlaying Bereiche, die sogenannten Slots, zu bestimmen.

Das vorgestellte Verfahren löst hierbei die folgenden drei Probleme:

- Wie viele Slots werden benötigt?
- Welche Module werden auf welche Slots abgebildet?
- Wo werden die Slots im FPGA platziert?

Für die Beantwortung der Fragen erwies sich das für Rekonfigurierung adaptierte Overlaying Konzept als vorteilhaft, so dass die durchschnittliche Rekonfigurierungsdauer minimiert, die Menge der benötigten, langen Kommunikationsverbindungen reduziert und die erforderliche Größe für den externen Bitstreamdatenspeicher minimal gehalten werden konnte.

Das automatisierte Platzierungsverfahren kann, durch die Angabe von Übergangswahrscheinlichkeiten, den individuellen Anforderungen eines eingebetteten Systems angepasst werden. Die Modellierung der Rekonfigurierungen als Markov Kette wurde für die Slotbestimmung und die Berechnung der durchschnittlichen Rekonfigurierungsdauer verwendet.

Die Fokussierung auf eingebettete Systeme ermöglicht es weiterhin, zur Minimierung der Rekonfigurierungsdauer, Module im FPGA vorzuhalten, die in einer aktuellen Konfiguration nicht benötigt werden. Dadurch wird die Rekonfigurierungsdauer weiter minimiert und der externe Bitstream Speicher kann gegebenenfalls kleiner dimensioniert werden. Bei der Platzierung des Beispiel Musik-Players auf einen XC2V2000 FPGA konnte mittels des vorgestellten Verfahrens die durchschnittliche Rekonfigurierungsdauer um 62 % reduziert werden.

Der Vorteil des vorgestellten Verfahrens, der sich aus dem Overlaying Konzept ergibt, ist, dass nicht mehr die Module auf den FPGA platziert werden müssen sondern deren Slots. Dies wird in dieser Arbeit mittels eines Greedy Algorithmus, unter Berücksichtigung der Busmakrogesamtlänge, realisiert.

8.1.3 Steuerung

Die Herausforderung des dritten Problembereichs lag in dem Entwurf einer ressourcensparenden Steuerung für die Rekonfigurierungen. Es können zwei Realisierungsformen für die RCU unterschieden werden:

- Steuerung in Software. Der Nachteil einer Softwaresteuerung ist die zusätzliche Hardware, die entweder neben dem FPGA vorhanden sein oder im FPGA als IP integriert sein muss. Als Vorteil ist die Flexibilität zu nennen.

- Steuerung in Hardware. Diese Steuerung kann ebenfalls außerhalb des zu rekonfigurierenden FPGA realisiert sein. Aufgrund der partiellen Rekonfigurierung ist aber auch eine Selbstrekonfigurierung möglich. Dadurch können unnötige Funktionalitäten eines Prozessors, wie er bei einer Steuerung in Software notwendig wäre, weggelassen werden.

Der vorgestellte Aufbau der RCU hat sich, gegenüber einer auf den FPGA abgebildeten Microprozessorlösung, als günstiger, in Bezug auf den Platzbedarf im FPGA, herausgestellt. Dieser Ansatz lässt sich auch automatisch generieren und erfordert dadurch auch keinen zusätzlichen Aufwand beim Systementwurf. Die RCU wurde modular konzipiert, damit technologieabhängige Komponenten, wie z.B. die Speicherschnittstelle, leicht ausgetauscht werden können.

8.2 Ausblick

Die dynamische Rekonfigurierung von Hardware Funktionalität kann nur dann für ein System vorteilhaft eingesetzt werden, wenn eine konkrete Systemanwendung berücksichtigt wird. In der vorliegenden Arbeit handelte es sich hierbei um eingebettete Systeme, deren Funktionalität durch Rekonfigurierung auf kleineren FPGAs lauffähig gemacht wurde. Zusätzlich zur Festlegung, welches Ziel mit Rekonfigurierung erreicht werden soll, wurde des Weiteren ein Rekonfigurierungskonzept bestimmt. In weiteren Arbeiten ist eine Übertragung dieser Herangehensweise auf andere Systeme denkbar. Beispielsweise könnten Hardwareupdates durch Rekonfigurierung in PC Erweiterungskarten erleichtert werden. Für die Organisation der auszutauschenden Module in diesen Karten, muss ein geeignetes Rekonfigurierungskonzept zu Grunde liegen. Dafür ist eine Anlehnung an Speicherverwaltungskonzepte zu evaluieren.

Die vorgestellten Verfahren und Methoden für die einzelnen Syntheseschritte können in weiteren Untersuchungen den Optimierungszielen angepasst werden. Je nach Entwicklungsstand eines eingebetteten Systems kann auch die Definition von Konfigurationen, über Simulation und Monitoring, automatisiert werden.

Eine Erweiterung des Platzierungsverfahrens, gegenüber dem vorgestellten Ansatz für Spalten-Rekonfigurierung, ist eine Ausnutzung des zusätzlichen Freiheitsgrads für die Rekonfigurierungsflächenform. Zu erwarten ist eine bessere Slotplatzierung, bei welcher die Anzahl der langen Bus Makros weiter reduziert werden kann. Ob sich hierdurch eine Verbesserung in der Performance des eingebetteten Systems erzielen lässt, wird systemabhängig sein. Die vorgestellten Heuristiken für die automatisierte Platzierung der Slots stellten nur eine Möglichkeit dar und zeigen die Anwendbarkeit. Mit weiteren Heuristiken, die nicht nur die Größe und Nutzungshäufigkeit der Module berücksichtigen, könnte hier ebenfalls die Effizienz des eingebetteten Systems optimiert werden.

Grundlegend für alle Erweiterungen und Optimierungen des vorgestellten Design Flows ist zu berücksichtigen, ob der Integrationsaufwand für die Rekonfigurierung in einem akzeptablen Verhältnis zu den Systemverbesserungen steht.

Literaturverzeichnis

[1] AHMADIFAR, Hamid R. ; MEHDIPOUR, Farhad ; ZAMANI, Morteza S. ; SEDIGHI, Mehdi ; MURAKAMI, Kazuaki: An incremental temporal partitioning method for real-time reconfigurable systems. In: *EHAC'06: Proceedings of the 5th WSEAS International Conference on Electronics, Hardware, Wireless and Optical Communications*. Stevens Point, Wisconsin, USA : World Scientific and Engineering Academy and Society (WSEAS), 2006. – ISBN 960–8457–41–6, S. 88–93

[2] ALPERT, C. J. ; KAHNG, A. B.: Multi-way partitioning via spacefilling curves and dynamic programming. In: *DAC '94: Proceedings of the 31st annual Design Automation Conference*. New York, NY, USA : ACM, 1994. – ISBN 0–89791–653–0, S. 652–657

[3] ALPERT, Charles J. ; YAO, So-Zen: Spectral partitioning: the more eigenvectors, the better. In: *DAC '95: Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*. New York, NY, USA : ACM, 1995. – ISBN 0–89791–725–1, S. 195–200

[4] AND TORZA, ANTHONY, Xilinx Inc.: *Using FPGA Technology to Solve the Challenges of Implementing High-End Networking Equipment: Adding a 100 GbE MAC to Existing Telecom Equipment*, September 2008

[5] AUER, Adolf: *Programmierbare Logik-IC*. 2. Aufl. Hüthig, 1994. – ISBN 978–3778522769

[6] BAUMGARTEN, Uwe ; SIEGERT, Hans-Jürgen: *Betriebssysteme: Eine Einführung*. 6. Oldenbourg Wissenschaftsverlag, 2006. – ISBN 3486582119

[7] BAYS, Carter: A comparison of next-fit, first-fit, and best-fit. In: *Commun. ACM* 20 (1977), Nr. 3, S. 191–192. – ISSN 0001–0782

[8] BECKERT, René: *Wissenschaftliche Schriftenreihe Eingebettete, Selbstorganisierende Systeme*. Bd. 7: *Untersuchung zur Kostenoptimierung für Hardware-Emulatoren durch Anwendung von Methoden der partiellen Laufzeitrekonfigurierung*. Bergstr.70, 01069 Dresden, Germany : TUDpress, 2008. – ISBN 978–3–940046–94–9

[9] BEHME, Henning ; MINTERT, Stefan: *XML in der Praxis*. 2nd. Munich, Germany : Addison Wesley, 2000. – ISBN 3827316367

[10] BENNETTS, R. G. ; OSSEYRAN, A.: IEEE standard 1149.1-1990 on boundary scan: history, literature survey, and current status. In: *J. Electron. Test.* 2 (1991), Nr. 1, S. 11–25. – ISSN 0923–8174

[11] BERGÉ, Jean-Michel ; LEVIA, Oz ; ROUILLARD, Jacques: *High-Level System Modeling*. Springer US, 1995. – ISBN 978–0–7923–9632–1

[12] BIEHL, Günter: Overview of Complex Array-Based PLDs. In: *Selected papers from the Second International Workshop on Field-Programmable Logic and Applications, Field-Programmable Gate Arrays: Architectures and Tools for Rapid Prototyping*. London, UK : Springer-Verlag, 1993. – ISBN 3–540–57091–8, S. 1–10

[13] BLODGET, B. ; JAMES-ROXBY, P. ; KELLE, E. ; MCMILLAN, S. ; SUNDARARAJAN, P.: *A selfreconfiguring platform*. `citeseer.ist.psu.edu/article/blodget03selfreconfiguring.html`. Version: 2003

[14] BOBDA, C. ; AHMADINIA, A. ; MAJER, M. ; TEICH, J. ; FEKETE, S. ; VEEN, J. van d.: DyNoC: A dynamic infrastructure for communication in dynamically reconfugurable devices. In: *International Conference on Field Programmable Logic and Applications* (2005), S. 153–158. ISBN 0–7803–9362–7

[15] BOBDA, Christophe: *Synthesis of Dataflow Graphs for Reconfigurable Systems using Temporal Partitioning and Temporal Placement*, Universität Paderborn, Heinz Nixdorf Institut, Entwurf Paralleler Systeme, Dissertation, 2003

[16] BOBDA, Christophe: *Introduction to Reconfigurable Computing*. 1. Springer Verlag, 2007. – ISBN 978–1–4020–6088–5

[17] BOBDA, Christophe ; STEENBOCK, Nils: A Rapid Prototyping Environment for Distributed Reconfigurable Systems. In: *Proceedings of the 13th IEEE International Workshop on Rapid System Prototyping (RSP'02)*. Darmstadt, Germany : IEEE Computer Society, July 2002. – ISBN 0–7695–1703–X, S. 153

[18] BURNS, J. ; DONLIN, A. ; HOGG, J. ; SINGH, S. ; DE WIT, M.: A dynamic reconfiguration run-time system. In: *FCCM '97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*. Washington, DC, USA : IEEE Computer Society, 1997. – ISBN 0–8186–8159–4, S. 66

[19] CARDOSO, ao M. P. Jo ; NETO, Horácio C.: An Enhanced Static-List Scheduling Algorithm for Temporal Partitioning onto RPUs. In: *VLSI '99: Proceedings of the IFIP TC10/WG10.5 Tenth International Conference on Very Large Scale Integration*. Deventer, The Netherlands, The Netherlands : Kluwer, B.V., 2000. – ISBN 0–7923–7731–1, S. 485–496

[20] CHAN, Pak ; SCHLAG, Martine ; ZIEN, Jason: SPECTRAL-BASED MULTI-WAY FPGA PARTITIONING. Santa Cruz, CA, USA : University of California at Santa Cruz, 1994. – Forschungsbericht

[21] COMPOSANO, Raul: Design process model in the Yorktown Silicon Compiler. In: *DAC '88: Proceedings of the 25th ACM/IEEE Design Automation Conference*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1988. – ISBN 0–8186–8864–5, S. 489–494

[22] CORBETTA, S. ; FERRANDI, F. ; MORANDI, M. ; NOVATI, M. ; SANTAMBROGIO, M. D. ; SCIUTO, D.: Two Novel Approaches to Online Partial Bitstream Relocation in a Dynamically Reconfigurable System. In: *ISVLSI '07: Proceedings of the IEEE Computer Society Annual Symposium on VLSI*. Washington, DC, USA : IEEE Computer Society, 2007. – ISBN 0–7695–2896–1, S. 457–458

[23] CRASSIN, Cyril ; NEYRET, Fabrice ; LEFEBVRE, Sylvain ; EISEMANN, Elmar: "GigaVoxels : Ray-Guided Streaming for Efficient and Detailed Voxel Rendering. In: *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* ACM, ACM Press, Feb 2009

[24] DALLY, W.J. ; TOWLES, B.: Route packets, not wires: on-chip interconnection networks. In: *Design Automation Conference, 2001. Proceedings*, 2001. – ISSN 0738–100X, S. 684–689

[25] DEWEY, Tom: IP Reuse for FPGA Design: Rapidly Unravel Internal and Third-Party IP. Technical Marketing Engineer Mentor Graphics, 2002. – Forschungsbericht

[26] DORFMAN, Len ; NEUBAUER, Marc J.: *Turbo Pascal Memory Management Techniques*. Windcrest Har Dsk edition, 1993. – ISBN 978–0830640591

[27] DUECK, Gunter ; SCHEUER, Tobias: Threshold accepting: a general purpose optimization algorithm appearing superior to simulated annealing. In: *J. Comput. Phys.* 90 (1990), Nr. 1, S. 161–175. – ISSN 0021–9991

[28] DYER, Matthias ; PLESSL, Christian ; PLATZNER, Marco: Partially Reconfigurable Cores for Xilinx Virtex. In: *In: Field Programmable Logic and Applications (FPL2002*, Springer, 2002, S. 292–301

[29] ELES, Petru ; PENG, Zebo ; DOBOLI, Alexa: VHDL system-level specification and partitioning in a hardware/software co-synthesis environment. In: *CODES '94: Proceedings of the 3rd international workshop on Hardware/software co-design*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1994. – ISBN 0–8186–6315–4, S. 49–55

[30] ESTRIN, Gerald: Organization of Computer Systems-The Fixed Plus Variable Structure Computer. In: *Proc. Western Joint Computer Conference*. New York, 1960, S. 33–40

[31] ESTRIN, Gerald: Reconfigurable computer origins: the UCLA fixed-plus-variable (F+V) structure computer. In: *IEEE Ann. Hist. Comput.* 24 (2002), Nr. 4, S. 3–9. – ISSN 1058–6180

[32] EWERSON CARVALHO, Fernando Moraes Ney C. Frederico Möller M. Frederico Möller: Design Frameworks and Configuration Controllers for Dynamic and Partial Reconfiguration / Pontifícia Universidada Católica do Rio Grande do Sul. 2004. – Forschungsbericht

[33] FAUST, Matthias ; HOVEN, Philipp: *Modellierung und automatische Auswertung von Regeln zur Beschreibung von Abhängigkeiten in XML-Dokumenten*, Universität Paderborn, Diplomarbeit, Januar 2004

[34] FEKETE, S. ; KÖHLER, E. ; TEICH, J.: Optimal FPGA module placement with temporal precedence constraints. In: *DATE '01: Proceedings of the conference on Design, automation and test in Europe*. Piscataway, NJ, USA : IEEE Press, 2001. – ISBN 0–7695–0993–2, S. 658–667

[35] FIDUCCIA, C. M. ; MATTHEYSES, R. M.: A linear-time heuristic for improving network partitions. In: *DAC '82: Proceedings of the 19th Design Automation Conference*. Piscataway, NJ, USA : IEEE Press, 1982. – ISBN 0–89791–020–6, S. 175–181

[36] FLADE, Marcel: Automatische Adaption von Hardware-Acceleratoren für Verhaltenssimulation . In: HARDT, Wolfram (Hrsg.) ; IHMOR, Stefan (Hrsg.) ; FLADE, Marcel (Hrsg.): *Rekonfigurierbare Schnittstellen* Bd. 1. Bergstr.70, 01069 Dresden, Germany, November , S. 196 – 303

[37] GAJSKI, D. D. ; KUHN, R. H.: New VLSI Tools. In: *Computer* 16 (1983), Nr. 12, S. 11–14. – ISSN 0018–9162

[38] GAJSKI, Daniel D. ; DUTT, Nikil D. ; WU, Allen C.-H. ; LIN, Steve Y.-L.: *High-Level Synthesis Introduction to Chip and System Design*. Berlin, Heidelberg, Deutschland : Springer Verlag. – ISBN 978–0–7923–9194–4

[39] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; JOHN, Vlissides: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. 1. Aufl. Bonn : Addison-Wesley, 1996. – ISBN 3–89319–950–0. – Design Patterns, 1995, Deutsche Übersetzung von Dirk Riehle

[40] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. Toronto, Ontario. Canada : Addisson-Wesley, 1995

[41] GAREY, Michael R. ; JOHNSON, David S.: *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA : W. H. Freeman & Co., 1990. – ISBN 0716710455

[42] GLOCKNER, Matthias: *Wissenschaftliche Schriftenreihe Eingebettete, Selbstorganisierende Systeme*. Bd. 6: *Methoden zur Analyse von Rückwärtskompatibilität von Steuergeräten*. Bergstr.70, 01069 Dresden, Germany : TUDpress, 2008. – ISBN 978–3–940046–60–4

[43] GOLDFARB, Charles F. ; PRESCOD, Paul: *XML Handbook™*. 5th. Upper Saddle River, New Jersey : Prentice-Hall, 2003 (The Charles F. Goldfarb Definitive XML Series). – ISBN 0–13–049765–7

[44] GRINSTEAD, Charles M. ; SNELL, Laurie J.: *Grinstead and Snell's Introduction to Probability*. Version dated 4 July 2006. American Mathematical Society, 2006

[45] GROSS, Markus: Are Points the Better Graphics Primitives? In: *Computer Graphics Forum* Bd. 20, The Eurographics Association and Blackwell Publishers, 2001

[46] GUERRIER, P. ; GREINER, A.: A Generic Architecture for On-Chip Packet-Switched Interconnections. In: *Design, Automation and Test in Europe Conference and Exhibition* 0 (2000), S. 250. – ISSN 1530–1591

[47] GUPTA, R. ; DE MICHELI, G.: Partitioning of functional models of synchronous digital systems. In: *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, 1990. – ISBN 0–8186–2055–2, S. 216–219

[48] HANDA, Manish: *Online placement and scheduling algorithms and methodologies for reconfigurable computing systems*. Cincinnati, OH, USA, Diss., 2004. – Chair-Vemuri, Ranga

[49] HARDT, Wolfram: *Integration von Verzögerungszeit-Invarianz in den Entwurf eingebetteter Systeme*. Aachen, Deutschland : Shaker Verlag. – ISBN 3–8265–8251–9

[50] HARDT, Wolfram: *HW/SW-Codesign auf Basis von C-Programmen unter Performanz-Gesichtspunkten*. Aachen : Shaker Verlag GmbH, 1996. – ISBN 3–8265–2051–3

[51] HARDT, Wolfram ; IHMOR, Stefan: *Wissenschaftliche Schriftenreihe:Eingebettete,selbstorganisierende Systeme*. Bd. 2: *Schnittstellensynthese*. Bergstr.70, 01069 Dresden, Germany : TUDpress, 2006

[52] HARDT, Wolfram ; MEISEL, André ; VISARIUS, Markus: Automatische Rekonfigurierung von Hardware-Schnittstellen. In: *Proceedings of the German Workshop on Intellectual Property Prinzipien*. Dresden, Germany, April 2004

[53] HARDT, Wolfram ; RAMMIG, Franz ; BÖKE, Carsten ; STROOP, Joachim ; RETTBERG, Achim ; DEL CASTILLO, G. ; KLEINJOHANN, Bernd ; TEICH, Jürgen: IP-based System Design within the PARADISE Design Environment. In: *Journal of Systems Architecture – the Euromicro Journal* (2001)

[54] HENDRICKSON, Bruce ; LELAND, Robert: An improved spectral graph partitioning algorithm for mapping parallel computations. In: *SIAM J. Sci. Comput.* 16 (1995), Nr. 2, S. 452–469. – ISSN 1064–8275

[55] HOPCROFT, John E. ; MOTWANI, Rajeev ; ULLMANN, Jeffrey D.: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. 2., überarbeitete Aufl. München : Pearson Studium, 2002. – ISBN 3–8273–7020–5

[56] HWANG, J. ; EL GAMAL, A.: Optimal replication for min-cut partitioning. In: *ICCAD '92: 1992 IEEE/ACM international conference proceedings on Computer-aided design*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1992. – ISBN 0–8186–3010–8, S. 432–435

[57] IHMOR, Stefan ; FLADE, Marcel ; HARDT, Wolfram: *Wissenschaftliche Schriftenreihe:Eingebettete,selbstorganisierende Systeme*. Bd. 1: *Rekonfigurierbare Schnittstellen*. Bergstr.70, 01069 Dresden, Germany : TUDpress, 2005

[58] IHMOR, Stefan ; HARDT, Wolfram: Runtime Reconfigurable Interfaces - The RTR-IFB Approach. In: *International Journal of Embedded Systems (IJES)* 1 (2005), Nr. 5/6/2005

[59] JOVANOVIĆ, S. ; TANOUGAST, C. ; BOBDA, C. ; WEBER, S.: CuNoC: A dynamic scalable communication structure for dynamically reconfigurable FPGAs. In: *Microprocess. Microsyst.* 33 (2009), Nr. 1, S. 24–36. – ISSN 0141–9331

[60] KALTE, H. ; LEE, G. ; PORRMANN, M. ; RUCKERT, U.: REPLICA: A Bitstream Manipulation Filter for Module Relocation in Partial Reconfigurable Systems. In: *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 3*. Washington, DC, USA : IEEE Computer Society, 2005. – ISBN 0–7695–2312–9, S. 151.2

[61] KALTE, Heiko ; PORRMANN, Mario: REPLICA2Pro: task relocation by bitstream manipulation in virtex-II/Pro FPGAs. In: *CF '06: Proceedings of the 3rd conference on Computing frontiers*. New York, NY, USA : ACM, 2006. – ISBN 1–59593–302–6, S. 403–412

[62] KERNIGHAN, B. W. ; LIN, S.: An Efficient Heuristic Procedure for Partitioning Graphs. In: *The Bell system technical journal* 49 (1970), Nr. 1, S. 291–307

[63] KIRKPATRICK, S. ; GELATT, C. D. ; VECCHI, M. P.: Optimization by Simulated Annealing. In: *Science* 220 (1983), May, Nr. 4598, 671–680. http://dx.doi.org/10.1126/science.220.4598.671. – DOI 10.1126/science.220.4598.671

[64] KOESTER, Markus ; PORRMANN, Mario ; RUCKERT, Ulrich: Placement-Oriented Modeling of Partially Reconfigurable Architectures. In: *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 3*. Washington, DC, USA : IEEE Computer Society, 2005. – ISBN 0–7695–2312–9, S. 164.2

[65] KOH, Shannon ; DIESSEL, Oliver: COMMA: A Communications Methodology for Dynamic Module-based Reconfiguration of FPGAs. In: KARL, Wolfgang (Hrsg.) ; BECKER, Jürgen (Hrsg.) ; GROSSPIETSCH, Karl-Erwin (Hrsg.) ; HOCHBERGER, Christian (Hrsg.) ; MAEHLE, Erik (Hrsg.): *ARCS Workshops* Bd. 81. Frankfurt am Main, Germany : GI, März 2006 (LNI). – ISBN 3–88579–175–7, S. 173–182

[66] KOLODNER, Janet: *Case-Based Reasoning*. San Mateo, California : Morgan Kaufmann Publishers In, 1993 (Morgan Kaufmann Series in Representation & Reasoning.). – ISBN 978–1–5586–0237–3

[67] KÜKÇAKAR, Kayhan ; PARKER, Alice C.: CHOP: A constraint-driven system-level partitioner. In: *DAC '91: Proceedings of the 28th ACM/IEEE Design Automation Conference*. New York, NY, USA : ACM, 1991. – ISBN 0–89791–395–7, S. 514–519

[68] KUMAR, Shashi ; JANTSCH, Axel ; MILLBERG, Mikael ; ÖBERG, Johny ; SOININEN, Juha-Pekka ; FORSELL, Martti ; TIENSYRJÄ, Kari ; HEMANI, Ahmed: A Network on Chip Architecture and Design Methodology. In: *VLSI, IEEE Computer Society Annual Symposium on* 0 (2002), S. 0117. http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/ISVLSI.2002.1016885. – DOI http://doi.ieeecomputersociety.org/10.1109/ISVLSI.2002.1016885. ISBN 0–7695–1486–3

[69] LAGNESE, E. D. ; THOMAS, D. E.: Architectural partitioning for system level design. In: *DAC '89: Proceedings of the 26th ACM/IEEE Design Automation Conference*. New York, NY, USA : ACM, 1989. – ISBN 0–89791–310–8, S. 62–67

[70] LAGNESE, E.D. ; THOMAS, D.E.: Architectural partitioning for system level synthesis of integrated circuits. In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 10 (1991), Jul, Nr. 7, S. 847–860. – ISSN 0278–0070

[71] LALA, Parag K.: *PLD: digital system design using programmable logic devices*. Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 1990. – ISBN 0–13–215088–3

[72] LEONARD, Brian ; YOUNG, Jeff ; SASS, Ron: Online placement infrastructure to support run-time reconfiguration. In: *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*. New York, NY, USA : ACM, 2004. – ISBN 1–58113–829–6, S. 256–256

[73] LYSAGHT, P. ; STOCKWOOD, J.: A Simulation Tool for Dynamically Reconfigurable Field Programmable Gate Arrays. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 4 (1996), Nr. 3, S. 381–390

[74] LYSAGHT, Patrick ; DUNLOP, John: Dynamic reconfiguration of FPGAs. In: *Selected papers from the Oxford 1993 international workshop on field programmable logic and applications on More FPGAs*. Oxford, UK, UK : Abingdon EE&CS Books, 1994. – ISBN 0–9518453–1–4, S. 82–94

[75] LYSAGHT, Patrick ; MCGREGOR, Gordon ; M, Gordon ; STOCKWOOD, Jonathan: Configuration Controller Synthesis for Dynamically Reconfigurable Systems. In: *In IEE Colloquium on Hardware–Software Cosynthesis*, 1996, S. IEE.

[76] MAJER, Mateusz ; AHMADINIA, Ali ; BOBDA, Christophe ; JÜRGEN: A Flexible Reconfiguration Manager for the Erlangen Slot Machine. In: *Dynamically Reconfigurable Systems Workshop*. Frankfurt (Main), Germany : Springer, March 2006, S. 183–194

[77] MAJER, Mateusz ; BOBDA, Christophe ; AHMADINIA, Ali ; TEICH, Jurgen: Packet Routing in Dynamically Changing Networks on Chip. In: *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 3*. Washington, DC, USA : IEEE Computer Society, 2005. – ISBN 0–7695–2312–9, S. 154.2

[78] MAJER, Mateusz ; TEICH, Jürgen ; AHMADINIA, Ali ; BOBDA, Christophe: The Erlangen Slot Machine: A Dynamically Reconfigurable FPGA-based Computer. In: *J. VLSI Signal Process. Syst.* 47 (2007), Nr. 1, S. 15–31. – ISSN 0922–5773

[79] MARWEDEL, P.: *Embedded System Design*. Secaucus, NJ, USA, 2006. – ISBN 1402076908

[80] MARWEDEL, Peter: *Eingebettete Systeme*. Springer-Verlag Berlin Heidelberg, 2007 (eXamen.press). – ISBN 978–3–540–34048–5

[81] MATTERN, Friedemann: Ubiquitous Computing: Schlaue Alltagsgegenstände – Die Vision von der Informatisierung des Alltags. In: *Bulletin SEV/VSE* (2004), September, Nr. 19, S. 9–13

[82] MCFARLAND, M.C. ; KOWALSKI, T.J.: Incorporating bottom-up design into hardware synthesis. In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 9 (1990), Sep, Nr. 9, S. 938–950. – ISSN 0278–0070

[83] MCGREGOR, Gordon ; LYSAGHT, Patrick: Self Controlling Dynamic Reconfiguration: A Case Study. In: *FPL*, 1999, S. 144–154

[84] MEGGINSON, David: *SAX Homepage*. `http://www.saxproject.org/`,

[85] MEISEL, André ; DRAEGER, Alexander ; SCHNEIDER, Sven ; HARDT, Wolfram: Design Flow for Reconfiguration based on the Overlaying Concept. In: *Proceedings of the 19th IEEE/IFIP International Symposium on Rapid System Prototyping*. Monterey, CA, USA, June 2008. – ISBN 978–0–7695–3180–9, S. 89 – 95

[86] MEISEL, André ; SCHNEIDER, Sven ; HARDT, Wolfram: Design-Flow für rekonfigurierbare eingebettete Systeme in Anlehnung an das Overlaying-Konzept. In: *Dresdner Arbeitstagung Schaltungs- und Systementwurf* Fraunhofer-Institut für Integrierte Schaltungen, 2008. – ISBN 3–9810287–2–4, S. 49 – 54

[87] MEISEL, André ; VISARIUS, Markus ; HARDT, Wolfram ; IHMOR, Stefan: Self-Reconfiguration of Communication Interfaces. In: *Proceedings of the 15th IEEE International Workshop on Rapid System Prototyping*. Geneva, Switzerland : IEEE Computer Society, June 2004. – ISBN 0–7695–2159–2, S. 144 – 150

[88] MOORE, G. E.: Cramming More Components onto Integrated Circuits. In: *Electronics* 38 (1965), April, Nr. 8, S. 114–117

[89] MOORE, G. E.: Progress in Digital Integrated Electronics. In: *Technical Digest of International Electron Devices Meeting* Bd. 21. Washington, D.C., Dezember 1975, S. 11–13

[90] OBER, Ulrike: *Partitionierung und Laufzeitoptimierung komplexer Anwendungen für FPGA-basierte Rapid-Prototyping-Board-Architekturen*. Bd. XIV. Darmstadt : Technische Univ. Darmstadt, 1999. – ISBN 3–925178–23–6

[91] PANDE, P.P. ; GRECU, C. ; IVANOV, A. ; SALEH, R.: Design of a switch for network on chip applications. In: *Circuits and Systems, 2003. ISCAS '03. Proceedings of the 2003 International Symposium on* Bd. 5, 2003, S. V–217–V–220 vol.5

[92] PANDEY, Awartika ; VEMURI, Ranga: Combined Temporal Partitioning and Scheduling for Reconfigurable Architectures. In: *In SPIE Conference on Con gurable Computing: Technology and Applications*, 1999

[93] PFLUG, Georg: *Stochastische Modelle in der Informatik*. Stuttgart : Teubner, 1986 (Leitfaeden und Monographien der Informatik). – ISBN 3–519–02259–1

[94] PURNA, Karthikeya M. G. ; BHATIA, Dinesh: Temporal Partitioning and Scheduling Data Flow Graphs for Reconfigurable Computers. In: *IEEE Transactions on Computers* 48 (1999), Nr. 6, S. 579–590. – ISSN 0018–9340

[95] RAMMIG, Franz J.: *Systematischer Entwurf digitaler Systeme*. Teubner Verlag, 1989

[96] RECHENBERG, Peter (Hrsg.) ; POMBERGER, Gustav (Hrsg.): *Informatik-Handbuch*. 2., aktualisierte und erweiterte. 1999. – ISBN 3–446–19601–3

[97] ROBINSON, David ; LYSAGHT, Patrick: Modelling and Synthesis of Configuration Controllers for Dynamically Reconfigurable Logic Systems using the DCS CAD Framework. In: LYSAGHT, Patrick (Hrsg.) ; IRVINE, James (Hrsg.) ; HARTENSTEIN, Reiner W. (Hrsg.): *Field-Programmable Logic and Applications*, Springer-Verlag, Berlin, 1999, S. 41–50

[98] RUSSELL, Stuart J. ; NORVIG, Peter: *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003 http://portal.acm.org/citation.cfm?id=773294. – ISBN 0137903952

[99] SASIKUMAR, M.: Case Based Reasoning, by Janet Kolodner and Morgan Kaufmann,. In: *User Modeling and User-Adapted Interaction* 8 (1998), Nr. 1-2, S. 157–160. http://dx.doi.org/http://dx.doi.org/10.1023/A:1008252324096. – DOI http://dx.doi.org/10.1023/A:1008252324096. – ISSN 0924–1868

[100] SCHAAF, Martin ; VISARIUS, Markus ; BERGMANN, Ralph ; MAXIMINI, Rainer ; SPINELLI, Marco ; LESSMANN, Johannes ; HARDT, Wolfram ; IHMOR, Stefan ; THRONICKE, Wolfgang ; FRANZ, Jasmin ; TAUTZ, Carsten ; TRAPHÖNER, Ralph: IPCHL - A Description Language for Semantic IP Characterization. In: MERMET, Jean P. (Hrsg.) ; VILLAR, Eugenio (Hrsg.): *Best of FDL'02, System Specification & Design Languages*, 2002

[101] SCHNEIDER, Sven ; MEISEL, André ; HARDT, Wolfram: Communication-aware Hierarchical Online-Placement inHeterogeneous Reconfigurable Systems. In: *Proceedings of the 20th IEEE/IFIP International Symposium on Rapid System Prototyping*. Paris, Frankreich, June 2009. – ISBN 978–0–7695–3690–3, S. 61 – 67

[102] SCHWERPUNKTPROGRAMM 1148, DFG: *Rekonfigurierbare Rechensysteme*. http://www12.informatik.uni-erlangen.de/spprr/. Version: September 2008

[103] SRAMEK, Milos ; KAUFMAN, Arie: Fast Ray-Tracing of Rectilinear Volume Data Using Distance Transforms. In: *IEEE Transactions on Visualization and Computer Graphics* 6 (2000), Nr. 3, S. 236–252. http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/2945.879785. – DOI http://doi.ieeecomputersociety.org/10.1109/2945.879785. – ISSN 1077–2626

[104] SUN MICROSYSTEMS, INC.: *Developer Resources for Java Technology*. http://java.sun.com/,

[105] TAN, H. ; DEMARA, R. F.: A Device-Controlled Dynamic Configuration Framework Supporting Heterogeneous Resource Management. In: *in Proceedings of Engineering of Reconfigurable System and Algorithm (ERSA05), Las Vegas*, 2005, S. 27–30

[106] TEICH, J.: *Digitale Hardware/Software-Systeme: Synthese und Optimierung*. Heidelberg, New York, Tokio : Springer-Lehrbuch, 1997

[107] TEICH, Jürgen ; FEKETE, Sándor P. ; SCHEPERS, Jörg: Optimization of Dynamic Hardware Reconfigurations. In: *J. Supercomput.* 19 (2001), Nr. 1, S. 57–75. http://dx.doi.org/http://dx.doi.org/10.1023/A:1011188411132. – DOI http://dx.doi.org/10.1023/A:1011188411132. – ISSN 0920–8542

[108] THE APACHE SOFTWARE FOUNDATION: *The Apache Xalan Projekt*. http://xalan.apache.org/,

[109] THE APACHE SOFTWARE FOUNDATION: *Xerces2 Java Parser Readme*. http://xerces.apache.org/xerces2-j/,

[110] THOMAS, Donald E. ; ADAMS, Jay K. ; SCHMIT, Herman: A Model and Methodology for Hardware-Software Codesign. In: *IEEE Des. Test* 10 (1993), Nr. 3, S. 6–15. – ISSN 0740–7475

[111] THORVINGER, Jens: *Dynamic Partial Reconfiguration of an FPGA for Computational Hardware Support*, Lund Institute of Technology, Diplomarbeit, June 2004

[112] VAHID ; LE ; HSU: A Comparison of Functional and Structural Partitioning. In: *ISSS '96: Proceedings of the 9th international symposium on System synthesis*. Washington, DC, USA : IEEE Computer Society, 1996. – ISBN 0–8186–7563–2, S. 121

[113] VAHID, F. ; GAJSKI, D. D.: Specification partitioning for system design. In: *DAC '92: Proceedings of the 29th ACM/IEEE Design Automation Conference*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1992. – ISBN 0–89791–516–X, S. 219–224

[114] VAHID, Frank ; LE, Thuy D.: Towards a Model for Hardware and Software Functional Partitioning. In: *CODES '96: Proceedings of the 4th International Workshop on Hardware/Software Co-Design*. Washington, DC, USA : IEEE Computer Society, 1996. – ISBN 0–8186–7243–9, S. 116

[115] VAHID, Frank ; LE, Thuy D. ; HSU, Yu-Chin: Functional partitioning improvements over structural partitioning for packaging constraints and synthesis: tool performance. In: *ACM Trans. Des. Autom. Electron. Syst.* 3 (1998), Nr. 2, S. 181–208. `http://dx.doi.org/http://doi.acm.org/10.1145/290833.290841`. – DOI http://doi.acm.org/10.1145/290833.290841. – ISSN 1084–4309

[116] VERHAEGH, Wim ; AARTS, Emile ; KORST, Jan: *Algorithms in Ambient Intelligence (Philps Research Book Series, "2)*. Norwell, MA, USA : Kluwer Academic Publishers, 2003. – ISBN 140201757X

[117] VISARIUS, Markus ; HARDT, Wolfram: Integration Infrastructure for IPQ Format compliant IP based Design Tools. In: *German Project-Workshop: IP-Qualifikation für effizientes Systemdesign*, 2003

[118] VISARIUS, Markus ; HARDT, Wolfram: IPQ Toolbox based on the IPQ Format. In: *Demo Presentations of the German Project-Workshop: IP-Qualifikation für effizientes Systemdesign*, 2003

[119] VISARIUS, Markus ; HARDT, Wolfram: The IPQ Format – An Approach to Support IP based Design. In: *Proc. of the 7. GI/ITG/GMM-Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, Shaker Verlag, Februar 2004, S. 106 – 115

[120] VISARIUS, Markus ; HARDT, Wolfram: An IPQ Format based Toolbox to Support IP based Design. In: *it - Information Technology* (2005), April, Nr. 2/2005, S. 98 – 106

[121] VISARIUS, Markus ; HARDT, Wolfram ; BERGMANN, Ralph ; VOLLRATH, Ivo ; SCHAAF, Martin ; VÖRG, Andreas ; MARTÍNEZ MADRID, Natividad ; SEEPOLD, Ralf ; RADETZKI, Martin ; THRONICKE, Wolfgang ; RÜLKE, Steffen ; TAUTZ, Carsten ;

TRAPHÖNER, Ralph ; JERINIC, Vasco ; MÜLLER, Dietmar ; SIEGMUND, Robert: Requirements on the IP Qualifying Format / University of Paderborn, Informatik- und Prozesslabor. Warburger Strasse 100, 33098 Paderborn, Germany, August 2001 (TR-IPL-2001-01). – Technical Report

[122] VISARIUS, Markus ; LESSMANN, Johannes ; HARDT, Wolfram: Tool Demonstration on IPQ Format based Retrieval. In: *Proc. of the International Workshop of the MEDEA+ Project A-511 ToolIP*, 2002

[123] VISARIUS, Markus ; LESSMANN, Johannes ; HARDT, Wolfram: IPQ Format based Tool-Chain. In: *Demo Presentations of the German Workshop on Entwurfsplattformen komplexer angewandter Systeme und Schaltungen (EkompaSS)*, 2003

[124] VISARIUS, Markus ; LESSMANN, Johannes ; HARDT, Wolfram ; KELSO, Frank ; THRONICKE, Wolfgang: An XML Format based Integration Infrastructure for IP based Design. In: *Proceedings of the 16th Symposium on Integrated Circuits and Systems Design (SBCCI)*, IEEE Computer Society, September 2003, S. 119 – 124

[125] VISARIUS, Markus ; LESSMANN, Johannes ; IHMOR, Stefan ; HARDT, Wolfram: IPQ Format based Retrieval. In: *MEDEA+ Design Automation Conference - Demonstration and Poster Exhibition*, 2002

[126] VISARIUS, Markus ; LESSMANN, Johannes ; KELSO, Frank ; HARDT, Wolfram: Generic Integration Infrastructure for IP based Design Processes and Tools with a Unified XML Format. In: *Integration - the VLSI journal* 37 (2004), September, Nr. 4, S. 289 – 321

[127] VISARIUS, Markus ; MEISEL, André ; SCHEITHAUER, Markus ; HARDT, Wolfram: Dynamic Reconfiguration of IP based Systems. In: *Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping*. Montreal, Canada : IEEE Computer Society, June 2005. – ISBN 0–7695–2361–7, S. 70 – 76

[128] VISARIUS, Markus ; SCHOLAND, Andreas: CAMP: Konzeption und Implementierung einer Java-basierten Integrationsplattform für Software Module / University of Paderborn, Informatik- und Prozesslabor. Warburger Strasse 100, 33098 Paderborn, Germany, August 2003 (TR-IPL-2003-03). – Technical Report

[129] WALDER, Herbert ; STEIGER, Christoph ; PLATZNER, Marco: Fast Online Task Placement on FPGAs: Free Space Partitioning and 2D-Hashing. In: *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*. Washington, DC, USA : IEEE Computer Society, 2003. – ISBN 0–7695–1926–1, S. 178.2

[130] WALLENTOWITZ, Henning (Hrsg.) ; REIF, Konrad (Hrsg.): *Handbuch Kraftfahrzeugelektronik*. Wiesbaden : Vieweg Verlag, 2006. – ISBN 978–3528–03971–4

[131] WANNEMACHER, Markus: *Das FPGA-Kochbuch*. Internatinal Thomson Publishing Co., 1998. – ISBN 3–8266–2712–1

[132] WATSON, Ian D.: An Introduction to Case-Based Reasoning. In: *Proceedings of the First United Kingdom Workshop on Progress in Case-Based Reasoning*. London, UK : Springer-Verlag, 1995. – ISBN 3–540–60654–8, S. 3–16

[133] WEISER, Mark: The Computer for the 21st Century. In: *Scientific American* (1991), Februar. http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html

[134] WIRTHLIN, Michael J. ; HUTCHINGS, Brad L.: DISC: The dynamic instruction set computer. In: *in Field Programmable Gate Arrays for Fast Board Development and Reconfigurable Computing*, 1995, S. 92–103

[135] WITTMANN, Klaus ; LEY, Wilfried ; HALLMANN, Willi ; WITTMANN, Wilfried; Hallmann W. Klaus; Ley L. Klaus; Ley (Hrsg.): *Handbuch der Raumfahrttechnik*. Carl Hanser Verlag, 2008. – 816 S. – ISBN 978–3–446–41185–2

[136] WOOD, Steve: *Using Turbo Pascal version 5*. Berkeley, CA, USA : Osborne/McGraw-Hill, 1989. – ISBN 0–07–881496–0

[137] XILINX INC.: *XC2064/XC2018 Logic Cell Array*, 2003. http://www.datasheetarchive.com/XC2064-datasheet.html

[138] XILINX INC.: *MicroBlaze Processor Reference Guide*, 2004. http://www.xilinx.com/support/documentation/sw_manuals/edk63i_mb_ref_guide.pdf

[139] XILINX INC.: *PowerPC^{TM}405 Processor Block Reference Guide*, August 2004. http://www.xilinx.com/ise/embedded/ppc405block_ref_guide.pdf

[140] XILINX, INC.: *XAPP290: Two Flows for Partial Reconfiguration: Module Based or Difference Based*, September 2004

[141] XILINX INC.: *Constraint Guide*. ISE 8.1i, 2005. http://www.xilinx.com/

[142] XILINX INC.: *Development System Reference Guide*. http://www.xilinx.com/itp/xilinx7/books/data/docs/dev/dev0001_1.html. Version: 2005

[143] XILINX INC.: *Running the Standard Modular Design Flow*. http://www.xilinx.com/itp/xilinx7/books/data/docs/dev/dev0030_7.html. Version: 2005

[144] XILINX INC.: *Early Access Partial Reconfiguration User Guide*. http://www12.informatik.uni-erlangen.de/esmwiki/images/f/f3/Pr_flow.pdf. Version: 2006

[145] XILINX INC.: *Virtex-4 Family Overview: Product Spezifikation DS 100*. v3.0, September 2007. http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf

[146] XILINX INC.: *Virtex-II 1.5V Platform FPGAs: Complete Data Sheet*, November 2007. http://www.xilinx.com/support/documentation/data_sheets/ds031.pdf

[147] XILINX INC.: *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, November 2007. http://www.xilinx.com/support/documentation/data_sheets/ds083.pdf

[148] XILINX INC.: *Our History*. http://www.xilinx.com/company/history.htm. Version: 2008

[149] XILINX INC.: *Virtex-5 Family Overview: Product Spezifikation DS 112*. v5.0, February 2009. http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf

Abkürzungsverzeichnis

α ..... Kardinalität von Kommunikationsverbindungen im Problem- bzw. Architekturgraph.

β ..... Kardinalität von Kommunikationsverbindungen im Modulgraph.

δ ..... Übergangsfunktion eines Automaten.

γ ..... Kardinalität von Bus Makros.

$\Gamma(mod)$ ... Funktionen zur Berechnung der Modulwahrscheinlichkeit.

λ ..... Ausgabefunktion eines Automaten.

Ω ..... Ausgabealphabet eines Automaten.

ϕ ..... Abbildungsfunktion von Modul auf Slot.

ϕ^{-1} ..... Abbildungsfunktion von Slot auf Modul.

Π ..... Zustandswahrscheinlichkeit.

$\sum$ ..... Eingabealphabet eines Automaten.

θ ..... Abbildungsfunktion von Modulkommunikationsverbindungen auf Bus Makros.

$Bitstream$ Menge der Bitstreams.

BM ..... Menge der Bus Makros.

BML ..... Busmakrogesamtlänge.

br ..... Breite eines Slots in CLBs.

F ..... Endzustandsmenge eines Automaten.

ho ..... Höhe eines Slots in CLBs.

k ..... Bitstreamfaktor .

KG ..... Konfigurationsgraph.

Kom ..... CLB Menge, die für die Abbildung der Kommunikation in einem Slot zur Verfügung steht.

$Konf$ ..... Menge der Konfigurationen.

$konf^+$ .... Menge der Module die nicht für die Konfiguration $konf$ benötigt, aber bei dieser Konfiguration mitgeladen werden.

KV ..... Menge der Kommunikationsverbindungen.

l ..... Anzahl der Probierschritte des *Threshold Accepting* Verfahrens.

$Logik$ ..... CLB Menge, die für die Abbildung der Module in einem Slot zur Verfügung steht.

MG ..... Modulgraph.

MK ..... Markov Kette.

Mod ..... Menge der Module.

$ModKV$ .. Menge der Modulkanten.

MUX .... Menge der automatisch integrierten Multiplexer.

p ..... bedingte Wahrscheinlichkeit einer Rekonfigurierung.

P_{abs} ..... absolute Wahrscheinlichkeit einer Rekonfigurierung.

PG ..... Problemgraph.

Q ..... Systemzustandsmenge eines Automaten.

q ..... Slotqualität.

q_0 ..... Startzustand eines Automaten.

Q_K ..... Menge der Hardwarekonfigurationen im Automatenmodell.

$Rang(mod)$ Funktion zur Berechnung einer Ordnungsrelation auf Modulen.

$Rekonf$ ... Menge der möglichen Rekonfigurierungen.

SE ..... Menge der Systemelemente.

se ..... Längenmaß Sloteinheit.

SG ..... Slotgraph.

$size_{frei}$ ... Größe der freien Fläche im FPGA in CLBs.

$size_{mod}$ ... Größe eines Moduls in CLBs.

$Slot$ ..... Slotmenge.

sp ..... Anzahl an CLB Spalten im FPGA.

t_{mod} ..... Modulladezeit.

T_{rekonf} .... durchschnittliche Rekonfigurierungsdauer.

t_{rekonf} .... Dauer einer Rekonfigurierung.

ze ..... Anzahl an CLB Zeilen im FPGA.

AAC ..... Advanced Audio Coding.

AG ..... Architekturgraph.

API ..... Application Programming Interface.

ASIC ..... Application Specific Integrated Circuit.

CAMP .... Common Application Module Platform.

CLB ..... Configurable Logic Block.

CPLD ..... Complex Programmable Logic Device.

CPU ..... Central Processing Unit.

CuNoC .... scalable Communication Unit based Network on Chip.

DAB ..... Digital Audio Broadcasting.

DISC ..... Dynamic Instruction Set Computer.

DRM ..... Digital compressed Radio Mondiale.

DyNoC .... Dynamic Network on Chip.

EDA ..... Electronic Design Automation.

EEPROM .. Electrically Erasable Programmable Read Only Memory.

EPROM ... Erasable Programmable Read Only Memory.

ESM ..... Erlangen Slot Machine.

FPGA ..... Field Programmable Gate Array.

GPC ..... General Purpose Computer.

GUI ..... Graphical User Interface.

HDL ..... Hardware Description Language.

IC ..... Integrated Circuit - integrierter Schaltkreis.

ICAP ..... Internal Configuration Access Port.

IFB ..... Interface Block.

IP ..... Intellectual Property - geistiges Eigentum.

ISP ..... In System Programmable.

JTAG ..... Joint Test Action Group.

LUT ..... Look Up Table.

MP2 ..... MPEG 1, Audio Layer 2.

MP3 ..... MPEG 1, Audio Layer 3.

MVC ..... Softwarearchitekturkonzept Model-View-Controller.

NGC ..... Native Generic Circuit.

NoC ..... Network on Chip.

PAL ..... Programmable Array Logic.

PCI ..... Peripheral Components Interconnection.

PLA ..... Programmable Logic Array.

PLD ..... Programmable Logic Device.

PROM .... Programmable Read Only Memory.

RCU ..... Reconfiguration Control Unit.

SDG ..... System Design Gate.

SoC ..... System on Chip.

SPLD ..... Simple Programmable Logic Device.

SRAM .... Static Random Access Memory.

TDI ..... Test Data Input.

TDO ..... Test Data Output.

UCF ..... User Constraint File.

USB ..... Universal Serial Bus.

VHDL .... Very High Speed Integrated Circuit Hardware Description Language.

Voxelgitter . Volumetric Pixel.

XDL ..... Xilinx Description Language.

XML ..... Extensible Markup Language.

XSD ..... XML-Schema-Definition.

Thesen

1. Dynamische Rekonfigurierung ist geeignet für eingebettete Systeme.
2. Durch dynamische Rekonfigurierung kann eine Reduzierung des Bedarfs an FPGA Ressourcen erreicht werden.
3. Die benötigten Designentscheidungen, um den Modular Design Flow durchzuführen, sind in Abhängigkeit eines Rekonfigurierungskonzeptes, dem Overlaying Konzept, automatisierbar.
4. Unter der Voraussetzung eines IP basierten Systems und der Definition von Konfigurationen, ist eine automatische Partitionierung in rekonfigurierbare Module durch Äquivalenzklassenbildung realisierbar.
5. Durch die Definition von Konfigurationen können Rekonfigurierungen, die einen kritischen Datenpfad im Problemgraph unterbrechen würden, vermieden werden.
6. Die Konfigurationskonzept basierende Partitionierung steht nicht im Widerspruch zu Robustheitsaspekten des Systems.
7. Mit zunehmenden FPGA Ressourcen kann die durchschnittliche Rekonfigurierungsdauer reduziert werden.
8. Die durchschnittliche Rekonfigurierungsdauer eines eingebetteten Systems wird am stärksten durch Module beeinflusst, die eine hohe Wahrscheinlichkeit und Modulrekonfigurierungsdauer aufweisen.
9. Die zur Verfügung stehenden Kommunikationsressourcen können besser, entsprechend der gegebenen Anzahl an kurzen und langen Kommunikationsverbindungen, ausgenutzt werden, wenn die Gesamtlänge der Bus Makros minimiert wird.
10. Eine in Hardware realisierte Rekonfigurierungssteuerung ist automatisch generierbar und bei realistischen eingebetteten Systemgrößen platzsparender als ein auf einem FPGA abgebildeter Prozessor.