der Bundeswehr
Universität München

Fakultät für Informatik
Institut für Technische Informatik

Optimierung der Rechenleistung pro Fläche von Prozessorarchitekturen durch Rekonfiguration von Funktionseinheiten

Rainer Scholz

Dissertation

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

Promotionsausschuss:

Vorsitzender:	Prof. Dr. Lothar Schmitz
1. Gutachter:	Prof. Klaus Buchenrieder, Ph. D.
2. Gutachter:	Prof. Dr. Axel Lehmann

Tag der Prüfung: 17.10.2011

Neubiberg, Oktober 2011

Kurzfassung

Viele eingebettete Systeme, wie Smartphones, PDAs, MP3-Player und zahlreiche weitere, werden zur Miniaturisierung, Kostenreduktion und Steigerung der Robustheit zunehmend als System-on-a-Chip, also auf nur einem Stück Silizium, gefertigt. In solchen Systemen arbeiten sowohl Prozessoren und Speicher, wie auch mannigfaltige andere Peripherieeinheiten, welche spezialisierte Aufgaben des jeweiligen Einsatzgebietes des Systems übernehmen. Einige dieser Einheiten sind jedoch nicht durchgängig im Einsatz, wie beispielsweise ein GSM-Modulator bei Smartphones oder ein Hardware MPEG-Dekoder im PDA.
Aufgrund der benötigten Flexibilität und des einfacheren Entwurfsprozesses wird es zunehmend populärer, Systems-on-a-Chip mit Field Programmable Gate Arrays (FPGAs), frei programmierbaren Logikbausteinen, zu realisieren. Aktuelle Bausteine erlauben dynamische partielle Rekonfiguration. Sie können also Teile ihrer Logik ersetzen, während andere weiter in Betrieb bleiben. Die Ressourcen nicht aktiver Einheiten des Systems können somit dynamisch für andere Zwecke benutzt werden.
Diese Arbeit schlägt eine Prozessorarchitektur vor, deren Rechenleistung sich durch zeitlich variable Hinzunahme und Abgabe von zur Verfügung stehenden Ressourcen der programmierbaren Logik anpasst. Zusätzliche Ressourcen werden, um dies zu erreichen, durch zusätzliche Funktionseinheiten für den Prozessor belegt. Deren Einbindung in die Berechnungen wird durch parallel ausführbare, den Prinzipien des Explicitly Parallel Instruction Computings genügende Instruktionen erreicht. Werden die belegten Ressourcen des Prozessors an anderer Stelle wieder benötigt, werden schrittweise Funktionseinheiten abgetreten, bis ein Minimum an Rechenleistung des Prozessors erreicht ist.
Durch diesen Ansatz werden die zeitweise ungenutzten Ressourcen des Prozessors sinnvoll verwendet. Zudem bietet die vorgeschlagene Architektur die Fähigkeit, sich selbst an die auszuführenden Berechnungen anzupassen und sie somit schneller auszuführen.
Ziel dieser Arbeit ist es, eine solche Klasse neuer Prozessoren zu definieren, ihren möglichen Nutzen zu quantifizieren und ihre technische Umsetzbarkeit nachzuweisen.
Die mögliche Beschleunigung durch eine solche Architektur wird durch simulative Zuordnung von Befehlen potentieller Traces von Programmen auf Funktionseinheiten ermittelt. Die technische Machbarkeit des Ansatzes wird durch prototypische Implementierungen der kritischen Elemente der Architektur, vor allem im Bereich der partiellen Rekonfiguration von FPGAs, gezeigt.

Danksagung

Größter Dank gilt meinem Doktorvater Herrn Prof. Buchenrieder. Ohne seine Kreativität und sein Fachwissen, jedoch auch ohne seine menschliche Art, sein immer offenes Ohr und den *sanften Druck* wäre diese Arbeit nie zustande gekommen.
Zudem möchte ich mich bei meinem Zweitgutachter Herrn Prof. Axel Lehmann, der mir viele hilfreiche Anregungen zu meiner Arbeit gegeben hat, bedanken.
Für das hervorragende Arbeitsklima danke ich dem gesamten Institut für Technische Informatik der Universität der Bundeswehr München. Hervorheben möchte ich hierbei vor allem meine engen ehemaligen Kollegen Frau Heike Rolfs, Herrn Dr. Robert Fischer und Herrn Dr. Herbert Kleebauer.
Für weitere fachliche, mathematische und sprachliche Unterstützung danke ich Herrn Dr. Robert Eigner, Herrn Johann Schuster und Herrn Andreas Harlander.
Ganz herzlich möchte ich mich bei meinen Eltern dafür bedanken, dass sie mich zu jeder Zeit vorbehaltlos in meinem beruflichen und privaten Leben unterstützt haben.
Meiner Lebensgefährtin Tanja danke ich für ihr Verständnis, für ihre Gabe im richtigen Moment zu motivieren, für ihre Fürsorge und vor allem für ihre Liebe.

Der Druck dieser Arbeit wurde aus Haushaltsmitteln der Universität der Bundeswehr München gefördert.

Inhaltsverzeichnis

1 Einleitung und Motivation

An eingebettete Systeme, also solche Systeme, die in meist kleinen und speziellen Kontexten arbeiten, bestehen andere Anforderungen als bei Systemen, in denen Rechenleistung zur Lösung genereller Probleme benötigt wird. So sind hier Vorgaben von Baugröße, um in kleinen Geräten verbaut werden zu können, und Stromverbrauch, um auch im Akkubetrieb lange Laufzeiten zu ermöglichen, von sehr hoher Relevanz.
Mit neueren Geräten, in denen eingebettete Systeme arbeiten, erweitern sich diese Anforderungen jedoch in den letzten Jahren mehr und mehr. Vor allem bei mobilen Geräten, wie Mobiltelefonen und PDAs, steigt die Vielfalt an Anwendungen und Anwendungsgebieten. Neben der nötigen Flexibilität, neue Applikationen zu unterstützen, führt dies auch dazu, dass der Bedarf an Rechenleistung für Software auf solchen Systemen stetig wächst.
Durch die klassischen Optimierungsziele mobiler Systeme, also Baugröße und Stromverbrauch, ist es jedoch nicht ohne weiteres möglich, diese Rechenleistung, ähnlich wie bei Desktop Computern, durch aufwändigere Prozessoren mit schnellerem Takt zu erreichen.
Die Nutzung vorhandener Ressourcen des Systems zur Erhöhung der Rechenleistung kann ein Weg sein, den neuen Vorgaben gerecht zu werden.

1.1 Konzept dieser Arbeit

Diese Arbeit verfolgt den Ansatz, Rechenleistung für eingebettete Systeme aus vorhandenen Ressourcen, welche zu manchen Zeiten des Betriebs eines Gerätes ungenutzt sind, zu gewinnen.
Der Fokus liegt dabei auf Geräten mit vielfältigen Fähigkeiten, die alle auf einem Baustein integriert sind. Diese sogenannten Systems-on-a-Chip (SoC) werden häufig mit Hilfe programmierbarer Hardware, meist in Form von Field Programmable Gate Arrays (FPGAs), realisiert. Aktuelle FPGAs bieten die Möglichkeit, im Betrieb Teile der Logik durch Konfiguration zu ändern, während andere Teile unbeeinflusst weiterarbeiten. Dieser Vorgang der partiellen dynamischen Rekonfiguration erlaubt es, Logikmodule, welche zu einem Zeitpunkt nicht verwendet werden, so anzupassen, dass sie andere nützliche Aufgaben erledigen.
Um diese ungenutzten Logik-Ressourcen praktikabel zur Beschleunigung eines Prozessors zu verwenden, bedarf es mehrerer Konzepte. Ein normaler sequentiell arbeitender Prozessor kann ohne aufwändige Anpassung der Software nur sehr wenig von Zusatz-Hardware profitieren. Dazu kann nur eine Prozessorarchitektur im Stande sein, welche im Aufbau ihrer Programme

die exakte Organisation des Prozessors variabel lässt.
EPIC-Architekturen (Explicitly Parallel Instruction Computing) bieten diese Eigenschaft. EPIC-Compiler gruppieren Befehle, welche zeitgleich ausgeführt werden können, in sogenannte Befehlsbündel. Die Tatsache, dass die Einzelbefehle der Bündel bei geeigneten Compiler-Einstellungen auch sequentiell ausführbar sind, ohne die Korrektheit des Programms dadurch zu zerstören, ist der Ansatzpunkt des in dieser Arbeit vorgestellten Architekturkonzeptes.
Die hier aufgezeigte Klasse von Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren verwendet aktuell nicht genutzte Teile der zugrundeliegenden programmierbaren Logik, um dort zusätzliche Funktionseinheiten zu konfigurieren. Diese Funktionseinheiten sind in der Lage, Teilbefehle eines Befehlsbündels auszuführen und somit die gesamte Abarbeitung zu beschleunigen. Durch eine minimale Anzahl an Funktionseinheiten stellt sie sicher, dass Berechnungen auch ausgeführt werden können, wenn zu einem Zeitpunkt nur sehr wenige Logikressourcen verfügbar sind.

1.2 Aufbau der Arbeit

Abbildung 1.1 zeigt den Aufbau dieser Arbeit. Die roten gestrichelten Pfeile zeigen darin die Abfolge der Kapitel. Die schwarzen Pfeile geben den logischen Zusammenhang der Kapitel an.
Im folgenden Kapitel gibt diese Arbeit einen Überblick über rekonfigurierbare Hardware. Dabei betrachtet sie auch die Hardware-Synthese, also den Prozess zur Erstellung von realen Schaltungen aus einer Beschreibung der Hardware für rekonfigurierbare Bausteine. Spezielles Augenmerk wird zudem auf partielle Rekonfiguration und Prozessoren für FPGAs gelegt.
Kapitel 3 erklärt daraufhin die EPIC-Architektur und deren zugrundeliegende Konzepte. Sie grenzt sich durch ihre Methoden zur Ausnutzung extrahierbarer Parallelität auf Instruktionsebene und die Möglichkeit zur variablen Befehlsausführung gegen andere Architekturen ab. Als reale Implementierung einer EPIC-Architektur wird der Intel Itanium-Prozessor ausführlich beschrieben.
Im vierten Kapitel wird eine Übersicht über sowohl aktuelle als auch abgeschlossene Projekte gegeben, welche sich, wie diese Arbeit, mit Prozessoren beschäftigen, die ihre Eigenschaften ändern können. Besonders herausgehoben werden dabei Prozessoren, welche EPIC-Konzepte mit programmierbarer Logik verbinden.
Aufbauend auf diesen Grundlagen und Forschungsprojekten, führt Kapitel 5 das Konzept der Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren ein und erläutert dieses.
Das nachfolgende Kapitel legt den Grundstein zur Analyse des Verfahrens. Es leitet dazu formal die Schritte zur Berechnung der Laufzeit von Programmen auf der vorgeschlagen Prozessorarchitektur her.
Mit diesen Formalismen wird in Kapitel 7 der erzielbare Laufzeitgewinn durch Laufzeitrekonfigurierbare EPIC Soft Core Prozessoren anhand von Programm-Traces analysiert. Dabei

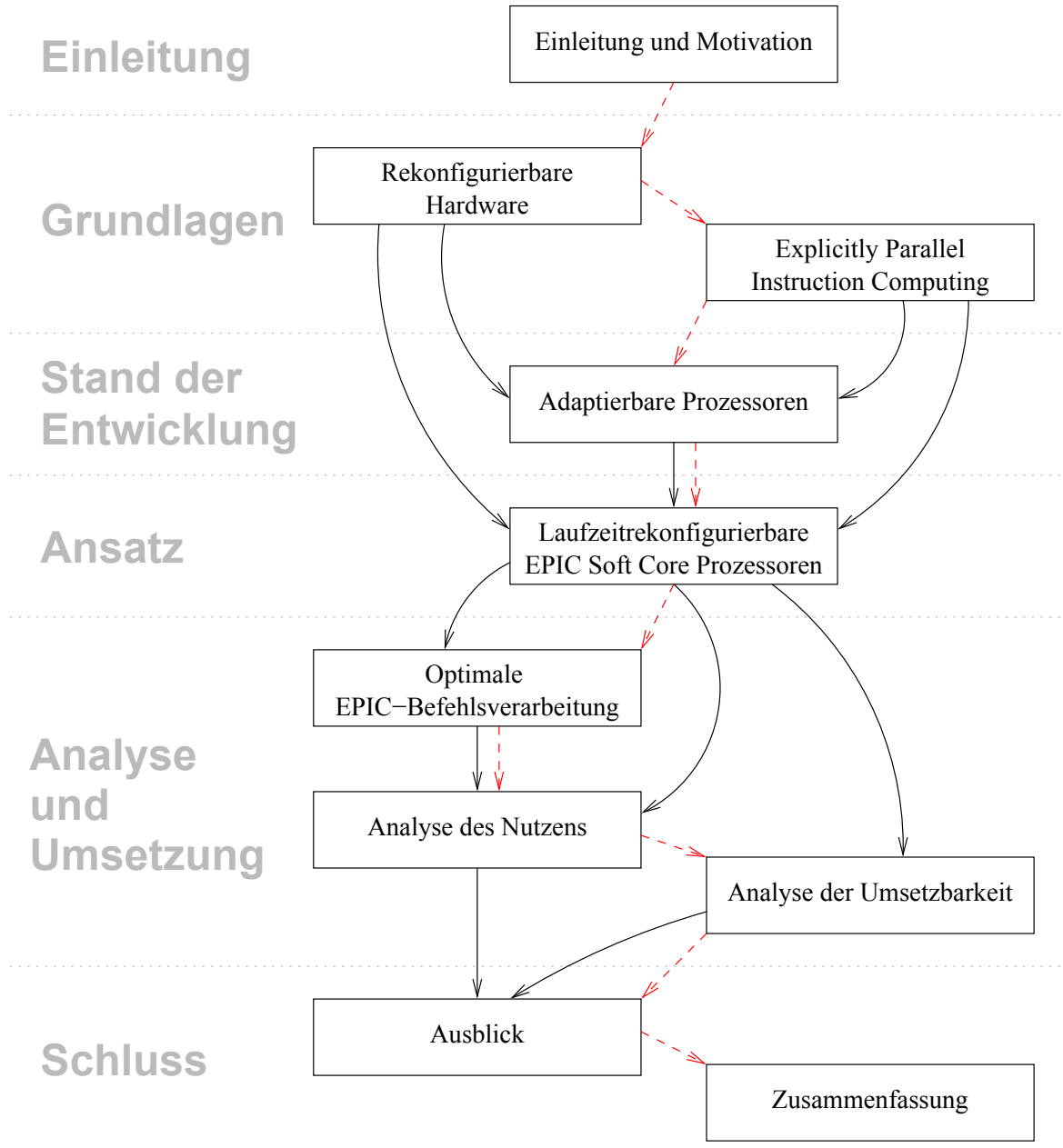


Abbildung 1.1: Aufbau der Arbeit.
Schwarzer Pfeil: Logischer Zusammenhang
Roter Pfeil: Abfolge in der Arbeit

wird sowohl der Nutzen durch die Hinzunahme von Funktionseinheiten als auch durch Funktionseinheitentausch zur Selbstoptimierung, also der Verbesserung des Gesamtdurchsatzes an Befehlen durch den Prozessor, quantifiziert.
Kapitel 8 greift die praktischen Probleme bei der Umsetzung von Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren auf und gibt Lösungsansätze für diese. Anhand prototypischer Implementierungen wird die Praktikabilität dieser Lösungen nachgewiesen.
Die Arbeit schließt mit einen Ausblick auf den Effekt, den zukünftige, verbesserte FPGAs und Compiler auf die vorgestellte Klasse von Prozessoren haben werden und gibt Ansatzpunkte für weitere Arbeiten. Zur Abrundung folgt eine Zusammenfassung der Arbeit und ihrer Ergebnisse.

1.3 Forschungsbeitrag

Diese Arbeit schlägt eine neue Prozessorarchitektur vor, welche die Konzepte von EPIC-Architekturen mit den Vorteilen der partiellen dynamischen Rekonfiguration von programmierbarer Logik verbindet. Sie zeigt zudem neue Methoden auf, solche Architekturen zu evaluieren und ihren Nutzen zu bestimmen. Dadurch gibt sie eine neue Möglichkeit, Platz und Rechenleistung in eingebetteten Systemen dynamisch gegeneinander zu tauschen und somit auf Aufgaben und vorhandene Ressourcen hin zu optimieren. Die vorgeschlagene Prozessorarchitektur ermöglichst somit erstmals, die zeitweise ungenutzten Ressourcen von auf FPGAs basierenden Systems-on-a-Chip für einen Zugewinn an Rechenleistung einzusetzen.
Zusätzlich werden neue Konzepte angegeben und nachgewiesen, welche in partiell dynamisch rekonfigurierenden Systemen auftretende Herausforderungen, wie beispielsweise Modulerstellung, Modulerkennung und Relokation von rekonfigurierbaren Modulen, lösen.

2 Rekonfigurierbare Hardware

Als rekonfigurierbare Hardware werden integrierte Schaltkreise (Integrated Circuits, ICs) bezeichnet, deren logische Funktionalität zum Fertigungszeitpunkt noch nicht feststeht und nachträglich durch entsprechende Verschaltung der Ressourcen festgelegt werden kann. Die Hardware ist also, im Gegensatz zu Logikbausteinen, deren Funktionalität bereits zum Produktionszeitpunkt unveränderbar ist, wie dies bei ASICs (Application Specific Integrated Circuit) der Fall ist, konfigurierbar. Es ist üblich, die Begriffe *programmieren* und *konfigurieren* synonym als Festlegen des Verhaltens eines rekonfigurierbaren Bausteins zu verwenden.
Vertreter rekonfigurierbarer Hardware sind Programmable Logic Devices (PLD) und Complex Programmable Logic Devices (CPLD). Eine Weiterentwicklung dieser Ansätze sind die größeren, flexibleren und nochmals komplexeren Field Programmable Gate Arrays (FPGAs). Es existieren auch andere Arten rekonfigurierbarer Hardware, wie beispielsweise Field Programmable Analog Arrays, FPAAs (z.B. in [77]) zur freien Konstruktion analoger Schaltkreise, die bisher jedoch kommerziell wenig erfolgreich sind.

2.1 Einfache Programmable Logic Devices

Für viele Anwendungen mit rekonfigurierbaren integrierten Schaltungen, wie als Verbindungslogik oder für einfache Funktionen, genügen schon relativ einfache Bausteine, wie PLDs oder CPLDs.

2.1.1 Programmable Logic Devices

PLDs sind programmierbare Bausteine, welche aus einer UND- und einer nachgeschalteten ODER-Struktur bestehen und somit sämtliche logischen Funktionen, welche in disjunktiver Form vorliegen, implementieren können. Solange die Größe des Bausteins ausreicht, kann jedes Schaltnetz mit einem PLD realisiert werden.
In Abbildung 2.1 ist eine UND-ODER-Struktur vereinfacht dargestellt. In solch einer Struktur können alle Eingangssignale, in negierter und nicht negierter Form, mit einem logischen UND verknüpft werden. Die resultierenden Terme können dann mit einem logischen ODER verknüpft und ausgegeben werden.
In der abgebildeten Struktur sind bestehende Verbindungen als ausgefüllter Kreis dargestellt. Die Struktur im Beispiel implementiert also die Funktion $O_1 = (I_1 \wedge I_2) \vee (\overline{I_1} \wedge \overline{I_2} \wedge I_3)$.

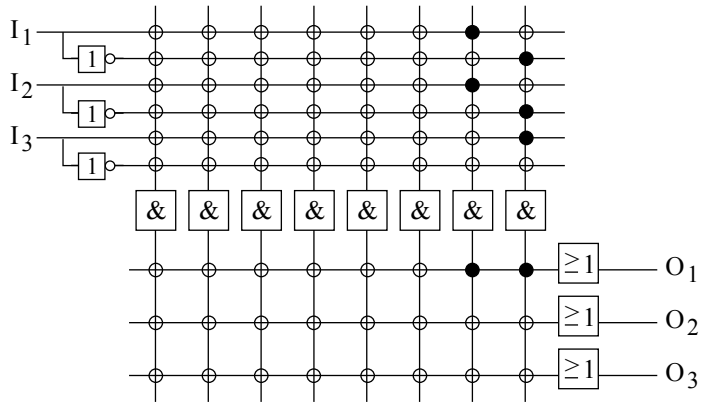


Abbildung 2.1: UND/ODER Struktur

Zu den PLDs zählen folgende Bausteine [34]:

- Ein *PROM (Programmable Read Only Memory)* ist ein PLD, bei dem nur die ODER-Struktur programmierbar ist. Die Verbindungen in der UND-Struktur sind festgelegt. Kann das PROM nach dem Programmieren wieder gelöscht werden, wird es EPROM (Erasable PROM) genannt. Elektrisch löschbare Ausführungen bezeichnet man als EEPROM (Electrically Erasable PROM).
- Im Gegensatz dazu kann bei einem *PAL (Programmable Array Logic)* nur die UND-Matrix programmiert werden. PAL-Bausteine können nur einmal programmiert werden. Dies geschieht durch Anti-Fuse Technologie, bei der durch Durchschmelzen einer Isolationsschicht Verbindungen irreversibel aufgetrennt werden. Aus einem voll verdrahteten Baustein entsteht dadurch die gewünschte Schaltung.
- Bei *GAL*-Bausteinen *(Generic Array Logic)* kann die UND-Matrix mehrfach programmiert werden, die ODER-Matrix jedoch nicht. GALs verfügen über FlipFlops, also 1-Bit-Speicher, und Rückführungsleitungen in die UND/ODER-Matrix an den Ausgabe-Pins, was auch die Realisierung einfacher Zustandsautomaten ermöglicht.
- Ein *PLA (Programmable Logic Array)* ist ein PLD, bei dem sowohl die Verbindungen der UND- als auch die der ODER-Struktur programmierbar sind.

2.1.2 Complex Programmable Logic Device

Complex Programmable Logic Devices, CPLDs, bestehen in der Regel aus mehreren Logik-Blöcken, einer programmierbaren Switch-Matrix, welche die Logikblöcke miteinander verbindet, und aus Input/Output Blöcken, welche auch über FlipFlops verfügen. Die Logikblöcke, die je nach Hersteller Generic Logic Block, Function Block oder Logic Array Block (LAB) heißen, sind aus programmierbaren Makrozellen aufgebaut. Typische CPLDs, wie der ispMACH von Lattice [76], der MAX von Altera [10] oder der CoolRunner II von Xilinx [133], ähneln

sich im grundlegenden Aufbau sehr stark. Die gemeinsame Grundarchitektur dieser Devices ist in Abbildung 2.2 gezeigt.

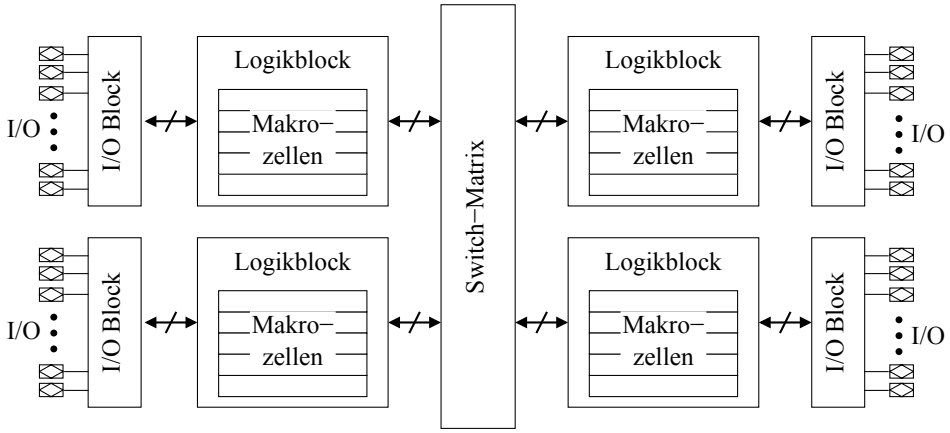


Abbildung 2.2: Aufbau eines CPLD

Durch den homogenen Aufbau können die Signallaufzeiten von Pin zu Pin in CPLDs exakt bestimmt werden. Die Konfiguration, also das Festlegen eines CPLDs auf eine spezielle Funktion, ist durch den Aufbau mittels EEPROM-Zellen, programmierbar, löschbar und nicht flüchtig, muss also nicht bei jedem Neustart des Gerätes neu geladen werden.
CPLDs lassen sich im Allgemeinen mit Dateien, die aus der Übersetzung von Spezifikationen in Hardware-Beschreibungssprachen (siehe Kapitel 2.3.1) erzeugt wurden, programmieren. Es existieren sogar sehr einfache Prozessoren, sogenannte Soft Core Prozessoren (siehe Kapitel 2.5.1), welche auf CPLDs abgebildet werden können. Ein Beispiel hierfür ist der 8-Bit-Prozessor PicoBlaze von Xilinx [141].
Da CPLDs ähnlich niedrige Preise, geringen Stromverbrauch und kleine Bauform wie PLDs haben, dabei jedoch mehr Ressourcen bieten, haben CPLDs die PLDS praktisch vom Markt verdrängt.

2.2 Field Programmable Gate Array

Noch mehr Kapazitäten als CPLDs werden von Field Programmable Gate Arrays, FPGAs, angeboten. Ihr komplexerer Aufbau bringt sowohl Vorteile, wie die Möglichkeit zur Erstellung großer Systeme, als auch Nachteile, wie den komplexeren Entwurfsprozess und die schwierigere Vorhersage von Signallaufzeiten, mit sich.

2.2.1 Einsatzgebiet

Ein Einsatzgebiet von FPGAs ist in Systemen, bei denen aufgrund der Aufgabe bereits absehbar ist, dass sie im Laufe ihrer Einsatzzeit an neue Aufgaben angepasst werden müssen.

Mögliche Gründe hierfür können beispielsweise zum Produktionszeitpunkt unfertig oder noch nicht definierte Standards sein, welche auf Dauer allerdings unterstützt werden sollen. Durch die Programmierbarkeit des FPGAs kann eine Anpassung jederzeit ohne physikalischen Eingriff in die Hardware geschehen. Auch Implementierungsfehler können durch Neuprogrammierung des FPGAs, ohne ihn aus dem System entfernen zu müssen, leicht behoben werden. Dieses Vorgehen wird als In-System Programming, ISP, bezeichnet.
Ein weiterer Bereich, in dem FPGAs weit verbreitet sind, ist Rapid Prototyping. Hierunter versteht man das frühe Erstellen von Prototypen von hochintegrierten Digitalen Schaltungen. Dieses Verfahren ermöglicht, Hardware-Systeme frühzeitig, kostengünstig und vor allem mit geringem Aufwand mehrmals zu testen. Somit können auch für kurze Produktdesignzyklen mehrere Iterationen des Hardwaredesigns durchgeführt werden. Durch diese Möglichkeiten zur einfacheren Validierung können die Entwicklungskosten von komplexen VLSI-Designs (Very Large Scale Integration, ICs mit einer hohen Anzahl an Transistoren) erheblich gesenkt und die Produkteinführungszeit, die Time-to-Market, verkürzt werden.
Ab einer gewissen Stückzahl an produzierten Systemen ist es günstiger, ASICs anstelle von FPGAs einzusetzen, da ASICs trotz hoher Entwurfskosten geringere Stückkosten verursachen. Bei aktuellen FPGAs ist jedoch der Trend zu beobachten, dass ihre Größe ständig steigt, während die Kosten sinken. Dadurch steigt die Stückzahl, bis zu welcher der Einsatz von FPGAs gegenüber dem von ASICs günstiger ist. Inzwischen sind FPGAs bei Kleinserien fast immer wirtschaftlicher. Durch ihre steigende Leistungsfähigkeit wächst die Zahl der Aufgaben, für welche sie einsetzbar sind.

2.2.2 Aufbau

Auch wenn sich die FPGAs der verschiedenen Hersteller (siehe Kapitel 2.2.4) in vielen Details unterscheiden, so ist ihnen allen gemein, dass sie aus Logikzellen, einem Verbindungsnetzwerk zwischen diesen Zellen und Ressourcen für Input/Output aufgebaut sind (vgl. Abbildung 2.3). Meist sind zusätzlich auch andere Komponenten, wie Block-RAM, verfügbar.

2.2.2.1 Logikzellen

Hauptbestandteil der Logikzellen aktueller FPGAs sind Lookup-Tables (LUT). Eine LUT gibt abhängig vom angelegten Eingangswert einen oder mehrere vorberechnete Werte aus. Da sie für eine Adresse, welche dem Eingangswert entspricht, einen Ausgangswert ausgibt, ist ihre Funktion mit der eines Speichers identisch.
LUTs werden nach der Anzahl an Eingangsvariablen bezeichnet, für die sie jede mögliche (binäre) Funktion abbilden können. So muss beispielsweise ein 4-Input LUT je mögliche Funktion auf vier Variablen, also 16 Stück, abbilden können. Sie entspricht also einem Speicher, welcher 16 einzeln adressierbare 1-Bit-Wörter speichern kann. Teilweise können sie auch direkt als RAM oder Schieberegister eingesetzt werden.

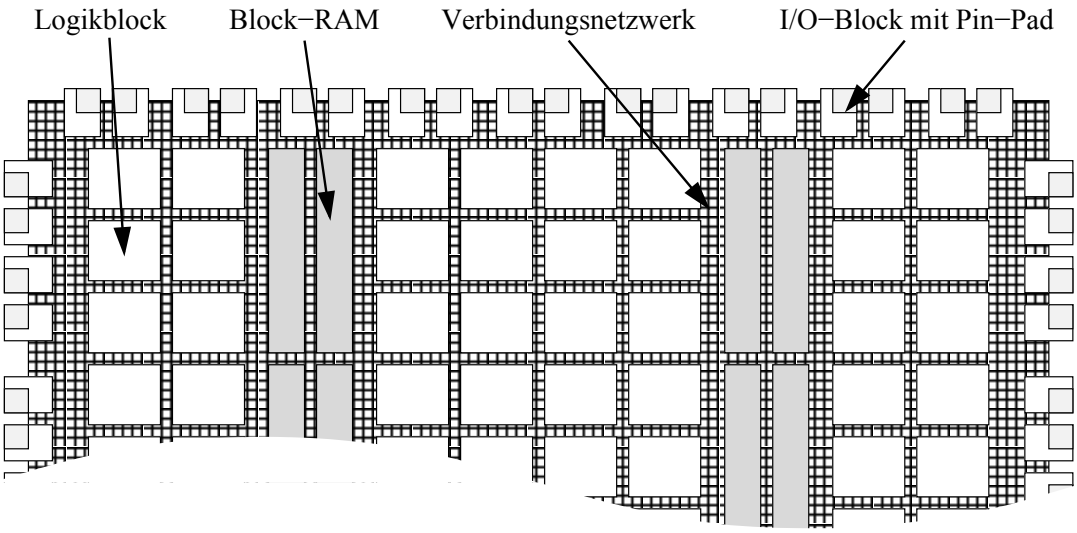


Abbildung 2.3: Aufbau eines FPGAs

Die aktuellen FPGAs der großen Hersteller sind aus 3-Input, 4-Input oder 6-Input LUTs aufgebaut. Durch die Implementierung in statischem RAM, SRAM, sind die LUTs programmierbar. Zudem sind in den Logikzellen auch stets FlipFlops die meist auch als Latch, also als FlipFlop mit Enable Eingang, eingesetzt werden können, zu finden. Abbildung 2.4 zeigt den typischen Aufbau einer einfachen Logikzelle.

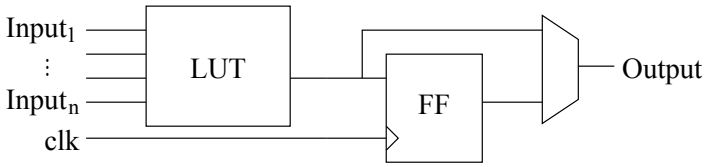


Abbildung 2.4: Aufbau einer einfachen Logikzelle eines FPGA

2.2.2.2 Verbindungsnetzwerk

Das Verbindungsnetzwerk verbindet die Logikblöcke, I/Os und anderen Komponenten des FPGAs miteinander. Es besteht aus vielen Verbindungen, den Interconnects, verschiedener Länge. Meist existieren direkte Verbindungen zwischen benachbarten Logikblöcken, lokale Verbindungen über wenige Logikblöcke hinweg und globale Verbindungen, welche auch weit entfernte Regionen des FPGAs miteinander vernetzen. Manche Interconnects sind gruppierbar und damit als interne Busse oder einzeln zur Verteilung von Clock-Signalen verwendbar.
Viele FPGAs verfügen zudem über Switch-Matrizen oder -Boxes, welche die Logikblöcke mit

dem Verbindungsnetz verknüpfen. Das Verbindungsnetzwerk belegt mit weitem Abstand den größten Anteil der Chip-Fläche. Abbildung 2.5 verdeutlicht dies am Beispiel eines Virtex-5 FPGAs.

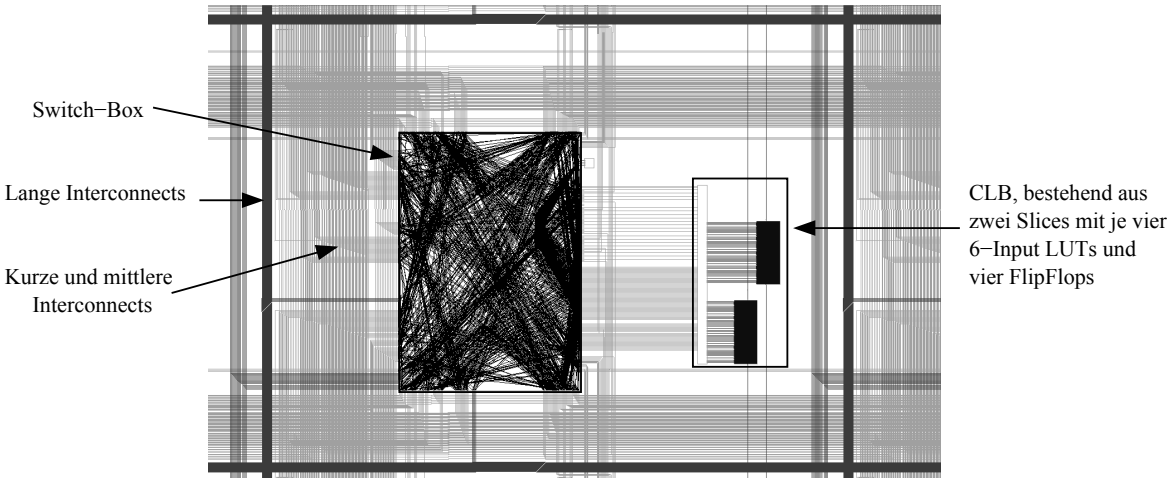


Abbildung 2.5: Logikzelle mit Verbindungsnetzwerk eines Virtex-5 FPGA

2.2.2.3 Input/Output

Allen FPGAs ist gemein, dass sie über Input/Output Blöcke (IOB) zur Kommunikation mit ihrer Umgebung verfügen. Diese sind mit einem oder mehreren Pins verbunden und bieten meist zusätzliche FlipFlops. Parameter wie Kommunikationsrichtung, Ausgangspannung etc. sind programmierbar. Es existieren FPGAs mit mehr als 1000 frei benutzbaren I/Os.

2.2.2.4 Sonstige Komponenten

Zusätzlich zu der Möglichkeit, die Logikzellen als RAM zu benutzen (Distributed RAM), haben die meisten FPGAs auch Block RAM. Dies sind Speicherbänke, die ausschließlich als RAM oder FIFO, also als First In - First Out Queue-Speicher, nutzbar sind. Große aktuelle FPGAs verfügen über bis zu 8 MByte Block RAM.
In der Regel sind auf FPGAs auch Komponenten zur Taktgenerierung, wie Phase Locked Loops (PLL), Delay Locked Loops (DLL), digitalen Frequenzsynthesizer (DFS) und Phasenschieber (PS) vorhanden.
Auch FPGAs mit mehreren Multiply-Accumulate-Units (MAC-Unit), welche vor allem für digitale Signalverarbeitung vorgesehen sind und deswegen auch als DSP Slice (Digital Signal Processor) oder DSP-Block bezeichnet werden, sind erhältlich.
Neben integrierten Prozessoren verschiedener Größenordnungen (Hard Core Prozessoren, sie-

he Kapitel 2.5.2) bieten manche Hersteller auch für vom Kunden gelieferte ASIC-Designs vorgesehene Bereiche auf dem FPGA an.

2.2.3 Größenangaben

Lange Zeit war es üblich, die Größe bzw. das Fassungsvermögen eines FPGAs in Gatteräquivalenten, der Zahl an NAND-Gattern mit zwei Eingängen, die nötig wären, um einen vollständig programmierten Baustein zu ersetzen, anzugeben. Dieses Maß ist jedoch vor allem beim Vergleich zwischen FPGAs verschiedener Hersteller extrem ungenau, da es stark von der Funktionalität, die implementiert werden soll, abhängt. Manche zusätzlichen Einheiten, wie Block RAMs oder DSP Slices, können bei einigen Designs erhebliche Vorteile im restlichen Ressourcenbedarf bringen, während andere Designs wieder von einer speziellen Verbindungsmatrix oder eingebetteten Prozessoren profitieren können. Auch das gewählte Synthesewerkzeug kann erheblichen Einfluss auf den Ressourcenbedarf eines Designs haben.
Inzwischen geben Hersteller der Bausteine die Vergleichszahl der Gatteräquivalente nur noch selten an. Stattdessen nennen sie meist nur noch die Anzahl der LUTs, die Menge des eingebetteten Speichers und die Anzahl der I/Os. Dies ermöglicht es immerhin, anhand des Platzverbrauches eines auf einem Baustein einer Familie implementierten Prototyps den entsprechend passenden derselben Familie zu ermitteln. Diese Richtwerte sind auch unter dem Aspekt eingängiger als das früher übliche Umrechnen von LUTs in Gatteräquivalente, dass eine 6-Input LUT sowohl zur Implementierung eines Inverters, welcher einem Gatter entspricht, als auch für komplexere Logikfunktionen, welche 20 Gattern entsprechen können, verwendet werden kann. Im maximalen Fall kann sie auch als 64-Bit RAM benutzt werden, welcher für Speicher und Adressdekoder insgesamt um die 100 Gatter benötigen würde.
Selbstverständlich versuchen die Hersteller trotzdem, die eigenen ICs als die besseren, größeren und effizienteren herauszustellen. Im Beispiel des Vergleiches des Virtex-5 mit dem Stratix III kommen die Hersteller Altera [6] und Xilinx [97] in ihren White Papers jeweils zu gegenteiligen Ergebnissen.
Ähnlich schwierig sind auch Aussagen über Geschwindigkeit, Logikdichte und sonstige technologische Fortschrittlichkeit zu treffen [88].

2.2.4 Hersteller von FPGAs

Den größten Anteil des Marktes für FPGAs teilen sich annähernd duopolistisch [60] die beiden Firmen Xilinx und Altera. Ihnen folgt der sich vor allem durch Flash-basierte, strahlungsresistente und low-power FPGAs absetzende Hersteller Actel und Lattice Semiconductor, welche zu den größten Herstellern von CPLDs gehören. Es gibt einige weitere Hersteller, wie Atmel und QuickLogic, welche auf dem Markt für FPGA jedoch eine untergeordnete Rolle spielen.

2.2.4.1 Xilinx, Inc.

Der marktbeherrschende Hersteller von FPGAs ist die Firma Xilinx. Ihre beiden großen Produktreihen sind die Hochleistungs-FPGAs der Virtex Serie und die günstigeren und stromsparenden Spartan FPGAs. Beide Reihen basieren auf SRAM. Die aktuellen Produkte (Stand Januar 2011) sind der Virtex-6 [143] und der Spartan-6 [142].
Deren Nachfolger werden die in 28-nm-Technologie gefertigten FPGAs der Xilinx Serie 7 sein [98]. Basierend auf einer einheitlichen Technologie sind die drei Familien Virtex-7, Kintex-7 und Artix-7 geplant. Diese werden sich voneinander in erster Linie in Größe, Leistung und Stromverbrauch unterscheiden.
Die aktuellen Devices von Xilinx werden in Tabelle 2.1 mit dem voraussichtlich größten Virtex-7 FPGA verglichen.

Tabelle 2.1: Vergleich aktueller und zukünftiger FPGAs von Xilinx (nach [130] und [139])

Features	Virtex-6	Virtex-5	Virtex-4	Spartan-6	Virtex-7
Logikzellen (max.)	760.000	330.000	200.000	150,000	1.955.000
User I/Os (max.)	1200	1200	960	576	1200
PLL	●	●	○	●	●
Block RAM (max.)	38 Mbits	18 Mbits	11 Mbits	4.8 Mbits	65Mbits
MAC-Einheiten	●	●	●	●	●
Eingebetteter Prozessor	○	PowerPC 440	PowerPC 405	○	○

Bis zum Virtex-4 waren die CLBs der Xilinx FPGAs aus vier Komponenten, den Slices, aufgebaut. Ein Slice bestand dabei aus zwei 4-Input LUTs und zwei FlipFlops. Seit den Virtex-5 Bausteinen bestehen die CLBs nur noch aus zwei Slices, welche jedoch aus jeweils vier 6-Input LUTs und vier FlipFlops aufgebaut sind [135]. Die Virtex-5 LUTs können zudem als 5-Input LUTs mit zwei Ausgängen benutzt werden. Die CLBs sind über Switch Matrizen mit Interconnects verschiedener Länge verbunden. Seit dem Virtex-6 sind in jedem CLB acht statt bisher vier FlipFlops integriert [138]. Abbildung 2.6 zeigt den groben Aufbau eines Virtex-6 CLBs.
Virtex FPGAs von Xilinx unterstützen seit vielen Jahren dynamisch partielle Rekonfiguration (siehe Kapitel 2.4).

2.2.4.2 Altera Corporation

Wie der Hauptkonkurrent Xilinx bietet auch Altera, als zweitgrößter Hersteller, sowohl High-End als auch low-cost FPGA Serien an, welche auf SRAM basieren. Erstere ist die Stratix Familie, mit dem Stratix V [12] als aktuellen Vertreter, letztere die Cyclone-Familie, deren neuestes Mitglied der Cyclone III [9] FPGA ist. Zwischen diesen beiden Familien siedelt Altera ihre ArriaGX [7] FPGAs an.

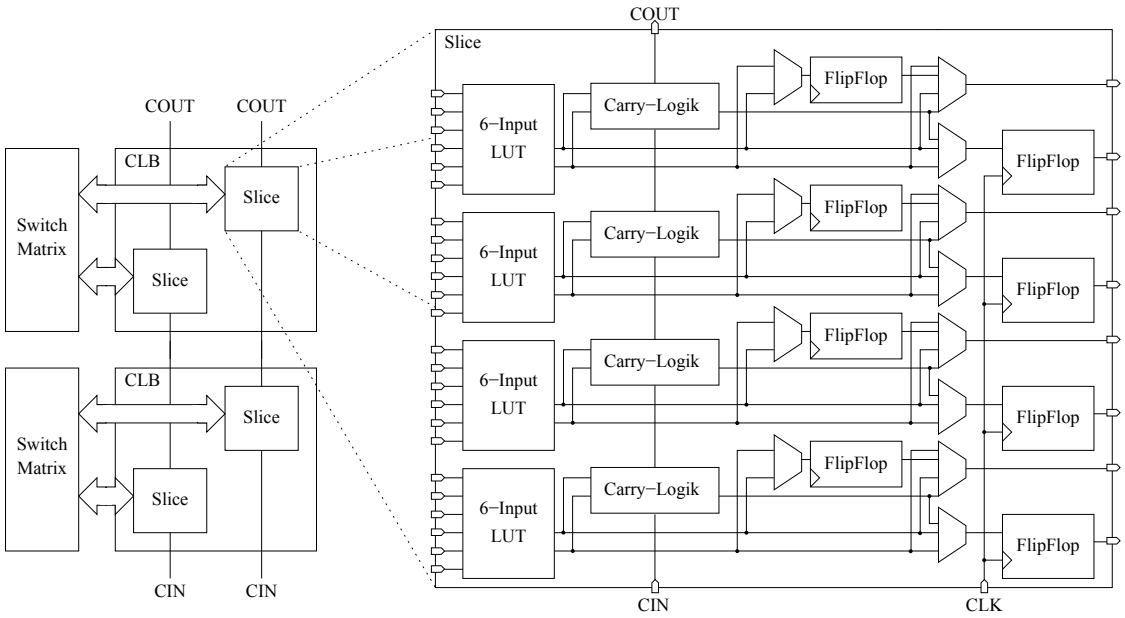


Abbildung 2.6: Stark vereinfachte CLBs eines Xilinx Virtex-6

Die Grundelemente des Aufbaus der Altera FPGAs sind ALMS, Adaptive Logic Modules. Vereinfacht sieht ein ALM aus wie ein halbes Virtex-6 Slice, hat also zwei 6-Input LUTs und vier FlipFlops. Da ein ALM nur acht Dateneingänge hat, sind die an die LUTs angelegten Werte jedoch voneinander abhängig. Zehn ALMs wiederum bilden einen Logical Array Block, LAB (siehe Abbildung 2.7). Die LABs sind untereinander durch Interconnect verschiedener Länge und Geschwindigkeit verbunden.

Seit Jahren im Angebot von Altera ist ihre spezielle Unterstützung zum Rapid Prototyping für ASICs. Diese sogenannte HardCopy-Technologie ermöglicht günstige ASICs zu fertigen, deren komplettes Verhalten und Bauform identisch mit dem von Stratix FPGAs ist. Ähnliche Konzepte sind beim Konkurrenten Xilinx erst seit kurzem mit der EasyPath Technologie [17] erhältlich. Seit dem Stratix V sind FPGAs von Altera wie die von Xilinx partiell rekonfigurierbar.

2.2.4.3 Actel Semiconductor Limited

Die Firma Actel zeichnet sich durch die Herstellung von FPGAs aus, die ihre Konfiguration nicht verlieren und somit nach dem Einschalten ohne Konfigurationsvorgang direkt benutzt werden können. Bei Bausteinen wie dem IGLOO und dem ProASIC [1] wird diese Eigenschaft durch mehrfach programmierbaren Flash-Speicher als Basis realisiert. Die erreichbare Logikdichte und Taktfrequenz ist bei diesen Produkten entschieden niedriger als bei den SRAM-basierten FPGAs der beiden Marktführer. Ebenso fertigt Actel FPGAs, deren Konfiguration

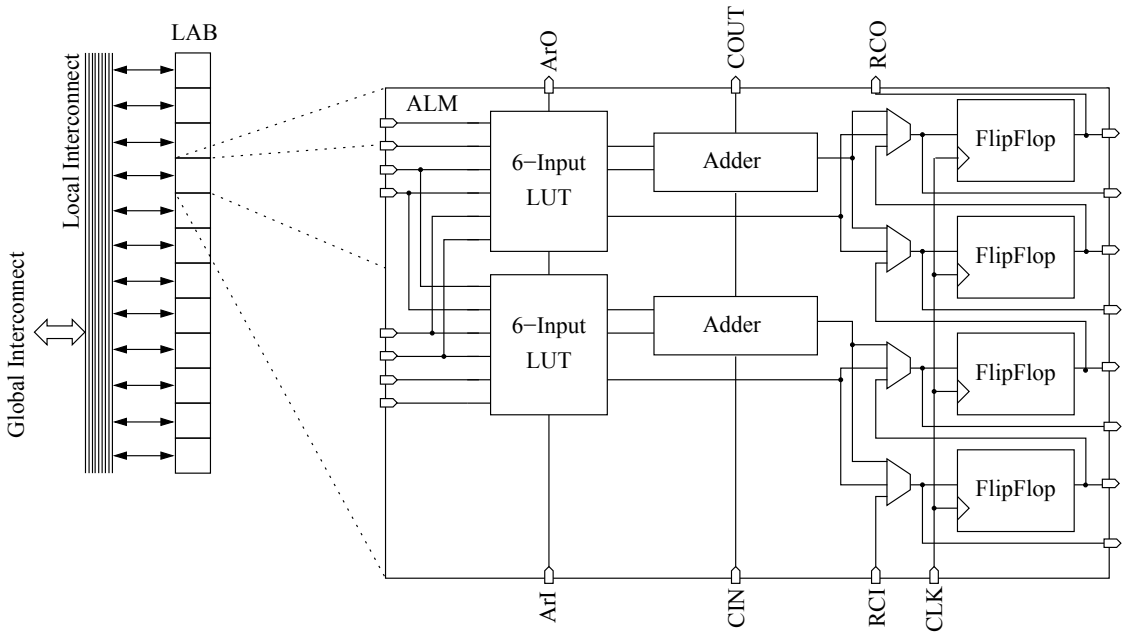


Abbildung 2.7: Stark vereinfachte LABs eines Altera Stratix V

in Anti-Fuse-Technik geschieht, wie den Axcelerator oder den SX-A. Actel bietet einige ihrer FPGAs auch als strahlungsresistente Variante für Anwendungen in Raumfahrt und Militär an. Einige System-on-a-Chip-Anwendungen werden durch die Actel Fusion Mixed-Signal FPGAs, welche über Analog/Digital-Wandler verfügen, erleichtert.
Die Flash-basierten FPGAs von Actel sind, wie in Abbildung 2.8 zu erkennen, sehr viel feingranularer aufgebaut als die der Konkurrenz. Die Logikzellen, hier VersaTile genannt [2], können als FlipFlop konfiguriert werden oder, wie eine 3-Input LUT, sämtliche Funktionen mit drei Eingängen realisieren.

2.2.4.4 Lattice Semiconductor Corporation

Lattice Semiconductor ist zwar in erster Linie Hersteller von CPLDs, fertigt jedoch auch FPGAs. Hier bieten sie sowohl den LatticeSC als High-Performance-Produkt, wie auch die low-cost LatticeEC-Reihe an, welche jeweils auf SRAM basieren. Es existieren Lattice FPGAs mit nicht flüchtiger Konfiguration auf Basis von Flash-Speicher und von EEPROM [74].
Die Logikzellen des LatticeSC werden Programmable Function Unit (PFU) genannt. Sie bestehen aus vier Slices, welche wiederum aus je zwei 4-Input LUTs und zwei FlipFlops aufgebaut sind. Neben den üblichen Features aktueller FPGAs verfügt der LatticeSC über ASIC Blocks, für die vorgefertigte IPs (Intellectual Property), wie Speicher oder PCI-Controller, oder auch IPs des Kunden schneller und platzsparender als in programmierbarer Logik arbeiten können.

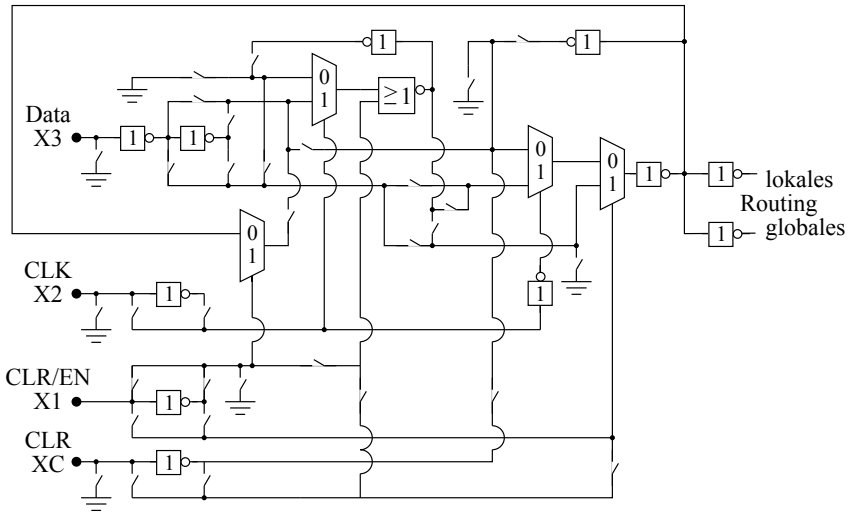


Abbildung 2.8: Logikzelle VersaTile von Atmel, Schalter sind Flash-programmierbar (nach [2])

2.2.4.5 Sonstige Hersteller von FPGAs

Neben den erwähnten Herstellern von FPGAs existieren noch viele weitere, die zwar einen sehr geringen Marktanteil haben, aber trotzdem teils kreative und ungewöhnliche Produkte bieten.
Die Firma Atmel, deren Hauptgeschäftsfeld die Herstellung von Microcontrollern ist, bietet mit dem AT40K [14] einen dynamisch partiell rekonfigurierbaren FPGA. Ihr FPSLIC [16] kombiniert einen AVR-Microcontroller mit einem AT40K auf einem Baustein.

2.3 Hardware-Synthese

Seit den 1980er Jahren ist es möglich, Hardware, ausgehend von einer Definition ihres Verhaltens, komplett automatisiert zu erstellen. Hierzu wird das Design in einer Hardware-Beschreibungssprache eingegeben und durch High-Level- und Logik-Synthese in Netzlisten, welche von der Zielhardware unabhängig sind, übersetzt.
Diese Netzlisten können dann durch die ebenso automatisierten Schritte Technology-Mapping und Place & Route in Bitströme zur Konfigurierung eines FPGAs umgesetzt werden.

2.3.1 Hardware-Beschreibungssprachen

Eine Hardware-Beschreibungssprache (Hardware Description Language, HDL) dient zur formalen Spezifikation von Hardware. Sie kann sowohl das Verhalten als auch die Struktur der Hardware beschreiben.

HDLs können Schaltungen auf mehreren Abstraktionsebenen beschreiben. Auf der abstrakten Systemebene beschreiben sie die Blöcke, aus welchen die Schaltung aufgebaut ist, als Black Boxes. Die Verbindungen zwischen den Blöcken werden ebenso angegeben. Das Verhalten dieser Blöcke wird dann auf der algorithmischen oder Register-Transfer-Ebene (Register Transfer Level, RTL) beschrieben. Algorithmische Beschreibung ähnelt durch Benutzung von Variablen und arithmetischen und logischen Operationen dabei den Programmiersprachen. Die Grundelemente der Beschreibung auf RTL sind Register, welche durch Signale miteinander verbunden sind [41]. Durch Simulation der HDL-Beschreibung kann Hardware vor der Herstellung getestet werden. HDLs basieren meist auf Programmiersprachen, welche durch Konstrukte zur Beschreibung von parallel auszuführenden Operationen, zeitlichem Verhalten, Hardware-Typen wie Signalen und Registern, erweitert sind.
Die bekanntesten HDLs sind die von der IEEE standardisierten Sprachen VHDL (VHSIC HDL, Very High Speed Integrated Circuit HDL) [53], welche auf der Programmiersprache ADA aufbaut, und Verilog [52], welche an C erinnert. Die meisten Konstrukte dieser beiden Sprachen sind synthetisierbar, lassen sich also automatisiert in Hardware umsetzen.
Eine beispielhafte Beschreibung eines 2:1-Multiplexers in VHDL sowie Auszüge der daraus synthetisierten Netzliste und des resultierenden Bitstreams sind in Abbildung 2.9 dargestellt.

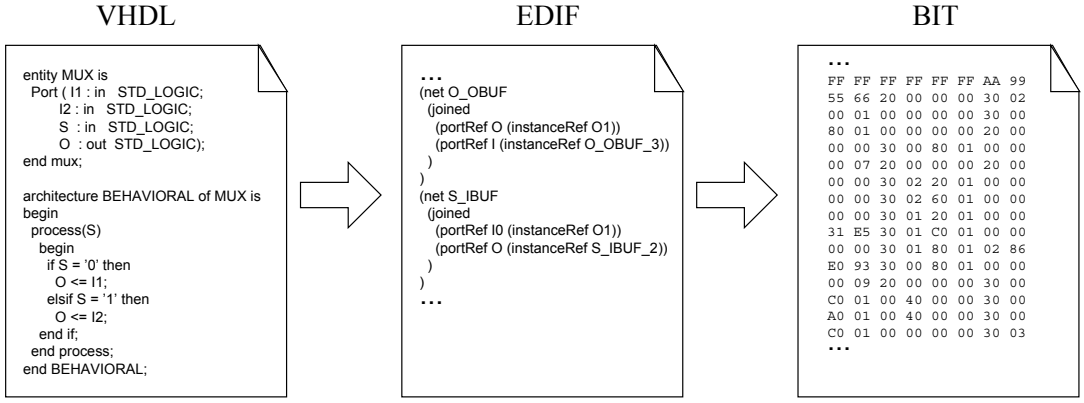


Abbildung 2.9: Übersetzungsprozess einer VHDL-Beschreibung am Beispiel eines 2:1-Multiplexers

2.3.2 High-Level- und Logik-Synthese

Bei der Synthese wird die meist in einer HDL oder durch CAD-Werkzeuge erstellte Grafiken beschriebene Schaltung in eine Netzliste übersetzt. Netzlisten enthalten die Beschreibung der Schaltung in textueller Form, deren Grundelemente Gatter und Speicher sowie die Verbindungen zwischen diesen sind. Netzlisten sind zumeist unabhängig von der Hardware, auf der

die Schaltung implementiert werden soll.
Wie in Abbildung 2.9 exemplarisch dargestellt, werden solche Netzlisten meist im Electronic Design Interchange Format (EDIF) erstellt. Grundsätzlich wird Synthese als Überführung einer Verhaltensbeschreibung in eine Strukturbeschreibung angesehen [87].

2.3.3 Technology Mapping und Place & Route

Beim Technology Mapping werden die Schaltungen in Form von hardware-unabhängigen mehrstufigen Netzlisten in die Grundelemente des Bausteins, auf dem die Schaltung implementiert werden soll, übersetzt. Vorgaben zum Design, die sogenannten Constraints, wie beispielsweise über das Timing, werden bei diesem Prozess mit einbezogen [87]. Diese Grundelemente, in die bei FPGAs übersetzt wird, sind zumeist Lookup-Tables oder Standardzellen. Beim Schritt Place & Route werden dann die resultierenden Schaltungen auf die vorhandenen Ressourcen des Bausteins abgebildet. Dabei werden die Grundelemente platziert. Bei den meisten FPGAs bedeutet dies, dass die LUTs der Schaltung auf eine Position des FPGAs festgelegt werden. Mittels der vorhandenen Routing-Ressourcen werden sie miteinander verbunden. Das Problem, hierbei eine optimale Verteilung und Verdrahtung zu erreichen, ist NP-hart. Es wird durch heuristische Algorithmen, wie beispielsweise Simulated Annealing [125], gelöst.
In sogenannten Constraints werden für diesen Prozess Optimierungsziele vorgegeben. Diese sind meist der erreichbare Takt und die benötigte Fläche der Schaltung, umfassen jedoch auch genaue Ortsangaben für Module auf dem Chip und die Lage von I/O-Pins.

2.3.4 Bitstreams und Konfiguration

Das Resultat von Place & Route wird zur Implementierung auf einen FPGA in eine serielle binäre Beschreibung der fertigen Schaltung übersetzt. Hierzu sind Informationen über das Konfigurationsprotokoll, also das Verfahren, über das der FPGA programmiert wird, nötig. Mit dem sich daraus ergebenden sogenannten Bitstream kann die Schaltung dann auf den FPGA konfiguriert werden. Dabei wird der Bitstream direkt in die Speicherzellen, welche die Funktion des FPGAs festlegen, geladen.
Das Format der Bitstreams ist herstellerspezifisch und nicht freigelegt. Zum Schutz geistigen Eigentums können Bitstreams zusätzlich verschlüsselt werden [35].

2.4 Rekonfiguration von FPGAs

Damit die programmierte Schaltung auf einem FPGA arbeitet, muss sie auf ihn übertragen, also konfiguriert, werden. Man unterscheidet verschiedene Arten der Konfiguration.

2.4.1 Konfiguration von FPGA-Bausteinen

FPGAs, welche nur ein einziges Mal konfiguriert werden können, wie z.B. die auf Anti-Fuse basierenden FPGAs, sind nicht rekonfigurierbar. Im Gegensatz dazu existieren auch FPGAs, welche mehrmals konfiguriert werden können. Ihre Konfigurierbarkeit lässt sich, wie in Abbildung 2.10 gezeigt, in verschiedene Klassen einteilen.

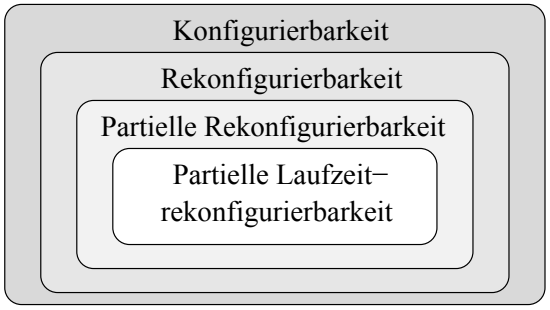


Abbildung 2.10: Übersicht über Klassen der Konfigurierbarkeit

Jeder programmierbare Logikbaustein ist konfigurierbar. Man nennt ihn rekonfigurierbar, wenn seine Konfiguration nach einmaliger Programmierung nochmals geändert werden kann. Zu dieser Klasse gehören alle SRAM-, Flash- und EEPROM-basierten FPGAs. Die Anzahl der Rekonfigurationen ist bei SRAM-FPGAs nicht beschränkt, EEPROM und Flash-Speicher sind zwar nur auf einige Tausend Änderungen ausgelegt, im realen Einsatz spielt dies allerdings sehr selten eine Rolle.
Wenn es der FPGA erlaubt, dass nur ein Teil seiner Konfiguration verändert wird, so gehört dieser zur Klasse der partiell rekonfigurierbaren FPGAs. Wenn zudem während des Vorgangs der partiellen Rekonfiguration die restliche Logik weiterarbeiten kann, nennt man den Vorgang partielle Laufzeitrekonfiguration, partial Runtime Reconfiguration (pRTR, teilweise auch nur RTR) oder Dynamisch Partielle Rekonfiguration (DPR).
Die Selbstrekonfigurierbarkeit kann zwar als Unterklasse der dynamisch partiellen Rekonfigurierbarkeit gesehen werden, die Abgrenzung ist jedoch in der Realität nur schwer möglich. Es sollte bei jedem partiell laufzeitrekonfigurierbaren System möglich sein, den Vorgang der Rekonfiguration auch selbst durchzuführen, sei es durch interne Zugänge zu den Konfigurationsdaten oder durch Verbindung einiger Ausgangsports mit den externen Konfigurations-Ports des Chips. Vorstellbar ist ein dynamisch partiell rekonfigurierbarer, aber nicht zeitgleich selbstrekonfigurierbarer Baustein nur dann, wenn seine Logik- oder I/O-Ressourcen nicht ausreichen, um das Konfigurationsprotokoll zu implementieren.

2.4.2 Konfigurationsprotokolle

Bitstreams können auf unterschiedliche Arten auf den FPGA übertragen werden. Die gängigste ist die standardisierte Konfiguration über Boundary Scan [51]. Dieses ist ein Verfahren der Joint Test Action Group (JTAG), welches ursprünglich nur zum Testen von Bausteinen verwendet wurde. Dabei werden durch eine meist mehrere Bausteine verbindende serielle Kette Daten in den FPGA geschoben. Abbildung 2.11 zeigt eine einfache solche Kette, die häufig

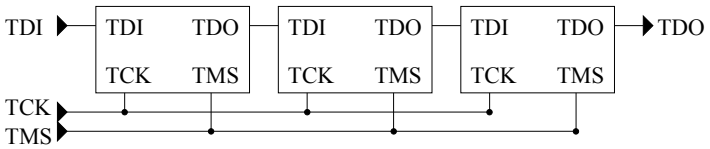


Abbildung 2.11: Einfache JTAG-Chain

auch als Daisy- oder JTAG-Chain bezeichnet wird. Boundary Scan wird meist für die Konfiguration des FPGAs über ein Programmierkabel genutzt.
Da die Daten dabei von einem externen Gerät in den zu programmierenden Baustein geschoben werden, nennt man diese Form der Konfiguration auch passive Konfiguration. Etabliert sind hier herstellerspezifische Protokolle, welche es erlauben, den FPGA passiv über ein paralleles Interface zu konfigurieren.
Die Konfiguration ist im Gegensatz dazu aktiv, wenn der FPGA die Daten selbständig aus einem Speicher, meist einem PROM, ausliest. Die Protokolle hierfür sind meist proprietär. Dabei ist es sowohl üblich, dass die Daten seriell als auch bis zu 32 Bits parallel übertragen werden können.
Die maximalen Taktraten zur Konfiguration eines FPGAs liegt bei diesen Protokollen stets im Bereich von etwa 50 MHz. Die maximale Übertragungsrate von Konfigurationsdaten auf den FPGA liegt somit bei paralleler Konfiguration bei etwa 200 MByte/s. Real muss von ca. 50% Overhead ausgegangen werden, was die reale Übertragungsraten auf ca. 100 MByte/s senkt.
Die Bitstreams zur kompletten Konfiguration der größten erhältlichen FPGAs haben von ca. 4,7 MByte (Altera Cyclone IV GX 150) bis ca. 11 MByte (Xilinx Virtex 6 LX760). Die schnellste komplette Konfiguration eines solchen Bausteins liegt ohne die Initialisierungszeit im Bereich von 50 bis 100 ms.
Die Konfiguration kleiner Bitstreams zur partiellen Rekonfiguration bietet hier für viele Anwendungsfälle die Möglichkeit, die Konfigurationszeit deutlich zu verringern.

2.4.3 Software Defined Radio als Beispielanwendung

Eine treibende Kraft hinter der Entwicklung partiell rekonfigurierbarer FPGAs ist derzeit ihre Einsatzmöglichkeit als digitale Komponente für weitgehend digitale Funkgeräte, den Software

Defined Radios (SDR).
Klassische Funkgeräte sind aus einer Vielzahl analoger Bauteile aufgebaut (siehe Abbildung 2.12). Soll ein Funkgerät mehrere Frequenzen, Modulationsarten und Verschlüsselungsverfahren unterstützen, muss die Hardware jeweils erweitert werden. Damit steigen sowohl Kosten und Stromverbrauch als auch die Größe des Funkgerätes an. Zudem ist Anpassung an neue Übertragungsstandards nicht oder nur mit sehr großem Aufwand möglich.

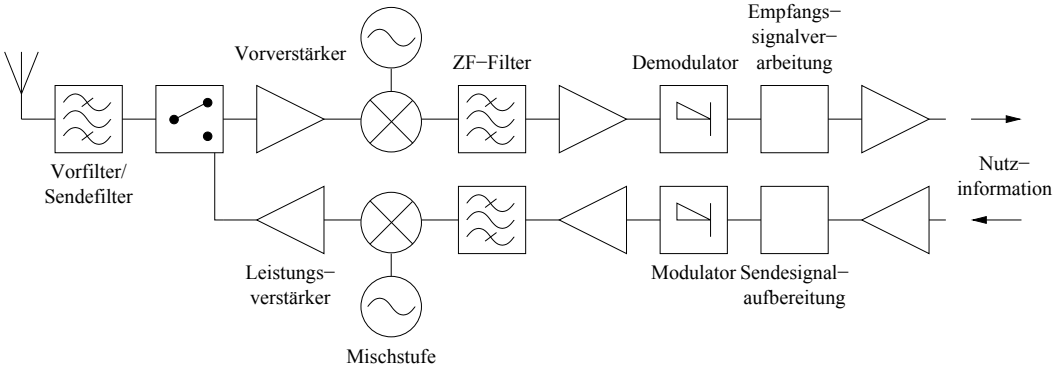


Abbildung 2.12: Aufbau eines klassischen Funkgerätes nach [103]

Das Konzept des idealen Software Radios (SR) schlägt ein Funkgerät vor, das außer Antenne, Leistungsverstärker, Mikrophon und Lautsprecher ausschließlich aus digitalen Komponenten besteht (siehe Abbildung 2.13). Dieses hat das Potenzial, alle erwähnten Nachteile analoger Funkgeräte zu beheben.

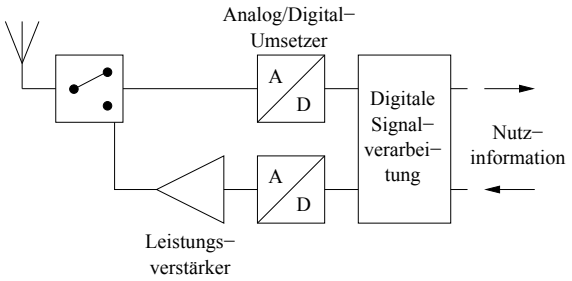


Abbildung 2.13: Aufbau eines idealen Software Radios nach [103]

Der in den sehr hohen Übertragungsfrequenzen, welche oft im GHz Bereich liegen, begründete Rechenaufwand verhindert die Umsetzung eines idealen Software Radios im VHF- und UHF-Bereich derzeit. Werden allerdings nur wenige zusätzliche analoge Komponenten, meist Mischer, zu den für das SR erlaubten eingesetzt, kann die nötige Rechenleistung in realistische Bereiche gesenkt werden.

Als digitale Hardware können in SDRs sowohl gewöhnliche Prozessoren (General Purpose Computer, GPP), DSPs, ASICs wie auch FPGAs eingesetzt werden. FPGAs ermöglichen hierbei durch Rekonfiguration die flexible Anpassung an neue Übertragungsstandards. Dies erhält die geforderte, oft jahrzehntelange Einsatzzeit der Funkgeräte.
Durch die voll parallele Durchführung der Berechnungen in Hardware erzielen sie vergleichsweise hohe Rechenleistung und können somit helfen, den Anteil an analoger Hardware im SDR weiter zu reduzieren.
Die Fähigkeit zur partiellen Laufzeitrekonfiguration schafft die Option, neue Übertragungsstandards zu konfigurieren, während andere in Gebrauch sind.
Für abhörsichere Kommunikation sind durch pRTR auch Verfahren denkbar, bei denen adaptiv zwischen Funkgeräten die Implementierung der Übertragungsstandards in Form verschlüsselter Bitstreams sicher über die Luftschnittstelle ausgetauscht wird. Diese Bitstreams werden dann erst durch bereits konfigurierte Kryptographiemodule im FPGA entschlüsselt und zur Selbstrekonfiguration benutzt, ohne jemals außerhalb des FPGAs im Klartext vorzuliegen.

2.4.4 Partiell laufzeitrekonfigurierbare FPGAs

Der erste partiell rekonfigurierbare FPGA war der von der Firma Algotronix produzierte CAL1024 [64]. Auf diesem aufbauend veröffentlichte die Firma Xilinx im Jahr 1996 mit dem XC6200 [131] den ersten kommerziellen FPGAs mit Unterstützung für pRTR. Seither sind alle FPGAs der Virtex-Familie von Xilinx partiell dynamisch rekonfigurierbar. Die Design-Werkzeuge zur Erstellung partieller Bitstreams waren jedoch wegen mangelnden Interesses seitens der Industrie weder ausgereift noch zuverlässig. Erst durch die Anwendung in Software Defined Radios (siehe Abschnitt 2.4.3) und die somit steigende Nachfrage wurden sie so weit verbessert, dass inzwischen der Einsatz von pRTR auch in kommerziellen Produkten empfohlen wird.
Wegen eben dieser Nachfrage ist auch in Alteras aktueller FPGA-Generation, der Stratix V Familie [12], erstmals die Möglichkeit, partiell zu rekonfigurieren, vorgesehen. Frühere Bausteine von Altera waren lediglich komplett rekonfigurierbar.
Die beiden partiell rekonfigurierbaren FPGA-Reihen, ORCA- [75] der Firma Lattice und AT40K-Familien [14] der Firma Atmel, sind beide nicht mehr erhältlich. FPGAs anderer Hersteller sind derzeit nicht partiell rekonfigurierbar.

2.5 Prozessoren auf FPGAs

Oft ist es in Systems-on-a-Chip effizienter, Aufgaben in Software anstatt in programmierbarer Hardware zu realisieren. Dies ist für solche Aufgaben der Fall, deren Ressourcenbedarf in Hardware zu hoch wäre, wie z.B. lange sequentielle Programmteile, oder für Programmteile, deren Performance-Anspruch nicht sonderlich hoch ist. Auch der Aufwand des Abbildens bestimmter Funktionalität in Software sowie der Portierung vorhandener Funktionalität ist

im Regelfall weitaus geringer.
Um das Auslagern von Aufgaben in Software zu ermöglichen, müssen Prozessoren in das System integriert werden, von denen die Software abgearbeitet wird.

2.5.1 Hard Core und Soft Core Prozessoren

Oben beschriebene integrierbare Prozessoren lassen sich in die beiden Kategorien Hard Core und Soft Core unterscheiden.
Hard Core Prozessoren sind vom Hersteller in die FPGA Struktur, das sogenannte FPGA Fabric, eingebrachte, festverdrahtete Prozessoren. Sie zeichnen sich durch hohe Kompaktheit und somit durch gute Leistung aus. Allerdings ist durch die feste Verdrahtung ihre Variabilität stark eingeschränkt. Portierungen von Designs mit Hard Core Prozessoren zwischen FPGA Fabrics verschiedener Hersteller gestalten sich durch die Vielzahl verschiedener Prozessoren schwierig.
Basiert der Prozessor auf den programmierbaren Logikzellen des FPGA, so wird dieser als Soft Core Prozessor bezeichnet. Soft Core Prozessoren werden entweder als synthetisierte Netzliste oder als synthetisierbarer Source-Code in einer Hardware-Beschreibungssprache ausgeliefert. Vor allem letztere bieten den Vorteil, dass sich der Prozessor auf eine spezielle Anwendung optimieren lässt. Durch Rekonfiguration sind solche Anpassungen auch zur Laufzeit denkbar. Der Aufwand für Portierungen von Designs mit Soft Core Prozessoren zwischen FPGAs verschiedener Hersteller ist für Prozessoren, die in HDL vorliegen, entsprechend gering. Sie können für das neue Zielsystem meist komplett automatisiert synthetisiert werden.
Hybrid-Prozessoren, bei denen Teile des Prozessors zur Effizienzsteigerung festverdrahtet sind und zur Steigerung der Flexibilität durch programmierbare Logik verbunden und gesteuert werden, sind ebenso denkbar. Solche Prozessoren werden bisher von keinem der großen FPGA-Hersteller angeboten. Die Soft Core Prozessoren der FPGA-Hersteller können durch ihre starke Bindung und Optimierung an die Hardware jedoch, auch wenn sie stets als Soft Core bezeichnet werden, als Hard-Soft Core Hybrid Prozessor angesehen werden.

2.5.2 Hard Core Prozessoren

Durch die feste Verdrahtung von Hard Core Prozessoren, werden diese nur direkt vom FPGA-Hersteller angeboten. Trotz deren hohen Leistungsfähigkeit bieten von den großen FPGA-Herstellern lediglich Xilinx und Atmel FPGAs mit Hard Core Prozessoren an.

2.5.2.1 PowerPC

Der wohl am weitesten verbreitete festverdrahtete Prozessor auf einem FPGA ist der PowerPC [50] von IBM. Er ist ein RISC-Prozessor, welcher der Harvard-Architektur entspricht, also getrennte Daten- und Adressbusse hat. Er ist seit der Virtex-II Pro-Familie [132] auf Xilinx FPGAs erhältlich. Bei diesen FPGAs sind bis zu zwei PowerPC 405 integriert, welche

mit bis zu 400 MHz betrieben werden können.
Bei Virtex-5 FPGAs sind bis zu zwei 32-Bit PowerPC 440 eingebettet [137]. Diese verfügen über interne Caches, Anbindungsmöglichkeiten für diverse Busse, Fließkommaeinheiten und Koprozessoren, die anwendungsspezifische Befehle implementieren. Sie können mit bis zu 550 MHz getaktet werden.
Die Konfiguration, Einbindung in komplette Designs und Software-Erstellung dieser Hard Core Prozessoren wird durch die ausführliche Tool-Unterstützung durch das Embedded Developement Kit, EDK [136] von Xilinx vereinfacht.
Virtex-6 FPGAs mit PowerPC Kernen sind nicht mehr erhältlich und auch für die Virtex-7-Familie sind keine solchen geplant.

2.5.2.2 AVR

Der FPSLIC Baustein [16] der Firma Atmel ist eine Kombination aus einem 8-Bit AVR Microcontroller und dem AT40K FPGA. Der Microcontroller ist ein 8-Bit RISC-Prozessor mit Harvard-Architektur. Seine einstufige Pipeline ist darauf ausgelegt eine Instruktion pro Taktzyklus zu berechnen. Die maximale Taktfrequenz liegt beim AVR des FPLSLIC bei 25 MHz.

2.5.3 Soft Core Prozessoren

Weiter verbreitet als die Implementierung von Hard Core Prozessoren ist die Unterstützung von Soft Core Prozessoren. Die aktuelle Entwicklung zeigt den Trend, dass keine Hard Core Prozessoren mehr implementiert werden. Stattdessen sind meist Einheiten zur Beschleunigung zeitkritischer Komponenten von Soft Core Prozessoren vorhanden.
Häufig sind Soft Core Prozessoren in HDLs implementiert und dadurch einfach und kostengünstig einsetzbar. Dies erklärt die große Anzahl an verfügbaren Soft Core Prozessoren unterschiedlicher Qualität und Leistungsfähigkeit.

2.5.3.1 OpenCores

OpenCores [95] ist die größte Community für den Entwurf und die Veröffentlichung von unter der GNU Lesser General Public License (LGPL) [39] stehenden Hardware-Designs. Unter den ca. 800 Projekten sind gut 100 frei verfügbare Prozessor-Designs zu finden, von denen ca. 30 fertiggestellt sind. Zu den interessantesten verfügbaren Prozessoren gehören der OpenRISC 1200 [72], der openMSP430 [43] und der AVR-Core [94].

OpenRISC
Der OpenRISC 1200 Prozessor ist der erste Prozessor nach der von OpenCores definierten OpenRISC 1000 Architektur [73]. Er wurde speziell für den Einsatz in FPGAs entwickelt. Der in Verilog geschriebene 32-Bit RISC-Prozessor hat eine Harvard-Architektur mit fünf

Pipeline-Stufen. Er unterstützt virtuellen Speicher.
Sein Kern ist eine CPU mit grundlegenden DSP-Fähigkeiten. Der Prozessor hat sowohl eine konfigurierbare Single Precision Floating Point Unit als auch eine Einheit für Hardware-basierte Multiplikation und Division.
Implementiert auf einem Xilinx Virtex-5 benötigt er in der Minimalkonfiguration ca. 7.000 LUTs, in der Standardkonfiguration ca. 15.000 LUTs.

openMSP430
Der openMSP430 ist ein in Verilog geschriebener Microcontroller, der mit dem MSP430 [118] von Texas Instruments kompatibel ist. Er unterstützt den vollen Instruktionssatz, alle Adressierungsarten und Stromsparmodi. Ein Hardware-Multiplizierer kann konfiguriert werden. Der 16-Bit-Controller ist für die Implementierung auf FPGAs optimiert.
Auf einem Virtex-V FPGA belegt er in minimaler Konfiguration ca. 1.200 LUTs, in maximaler Konfiguration ca. 1.800 LUTs.

AVR-Core
Der AVR-Core ist mit dem Atmel ATmega103 [15] voll kompatibel. Der VHDL-Quellcode dieses einfachen 8-Bit-Microcontrollers ist ebenfalls frei erhältlich. Es ist möglich, die Software-Entwicklungsumgebungen von Atmel für diesen Controller einzusetzen.

2.5.3.2 Leon

Der Leon3 [3] ist ein in VHDL geschriebener 32-bit Open Source Soft Core Prozessor von Aeroflex Gaisler. Er basiert auf der SPARC-Architektur (Scalable Processor ARChitecture) [129] und ist stark konfigurierbar. Als Teil der GRLIB [40] IP Bibliothek sind für den Leon3 Prozessor vielfältige Peripheriemodule erhältlich, mit welchen er über einen AMBA 2.0-Bus (Advanced Microcontroller Bus Architecture) [13] kommunizieren kann. Es existieren Linux-Portierungen für Leon-Prozessoren.
Seit Anfang 2010 ist auch der Leon4 Prozessor [4] erhältlich. Durch breitere interne Busse, eine modifizierte Pipeline und die Unterstützung für Level 2 Cache erzielt er höhere Rechenleistung als der Leon3. Auf einem Virtex-5 FPGA kann der Kern mit einer Taktfrequenz von 125 MHz betrieben werden und benötigt ca. 4.000 LUTs.
Sowohl der Leon3 als auch der Leon4 sind darauf ausgelegt, als synchrones oder asynchrones Multiprozessorsystem in eingebetteten Systemen zu arbeiten. Ihr Ziel ist es, damit höhere Rechenleistung bei niedrigeren Taktraten als Einzelprozessoren zu bieten. Beide Leon-Prozessoren stehen unter der GNU General Public License [38] und sind somit für nicht kommerziellen Einsatz kostenfrei nutzbar.

2.5.3.3 MicroBlaze und PicoBlaze

Der MicroBlaze [140] ist ein 32-Bit Soft Core Prozessor der Firma Xilinx. Er ist ein RISC-Prozessor mit Harvard-Architketur. Sein Quellcode ist nicht offengelegt. Die zugehörige Design-

Software Embedded Development Kit (EDK) ermöglicht jedoch weitreichende Konfiguration des Prozessors. Zudem bietet sie eine Umgebung zur Entwicklung von Software an. Linux-Portierungen für den MicroBlaze existieren. In einer flächenoptimierten Version benötigt er ca. 5100 Virtex-6 LUTs und läuft dann mit ca. 230 MHz.
Mit gut 100 LUTs hat der PicoBlaze-Prozessor [141] von Xilinx entschieden weniger Platzbedarf. Er ist ein in VHDL geschriebener 8-Bit RISC-Prozessor. Für Xilinx-Kunden ist er kostenfrei zu nutzen.

2.5.3.4 Nios

Das Konkurrenzprodukt zum MicroBlaze ist der Nios II [11] Soft Core Prozessor von Altera. Er ist ein 32-Bit-Prozessor mit vielen Konfigurationsmöglichkeiten, durch die es möglich ist, Platzverbrauch gegen Rechenleistung abzuwägen. Eine Softwareentwicklungsumgebung für den Nios II sowie eine Embedded Linux-Portierung ist verfügbar.
Ein Feature, welches den Nios-II Prozessor von seiner Konkurrenz abhebt, ist die Möglichkeit, rechenleistungsintensive Programmteile automatisiert in maßgeschneiderte Instruktionen auszulagern [8]. Diese können aus dem C-Quelltext automatisch erstellt, in Hardware-Beschreibung übersetzt und dann vom Prozessor ausgeführt werden. Die I/O dieser Module funktioniert durch Abbildung im Speicher des Prozessors.

2.6 Zusammenfassung und Ausblick

Dieses Kapitel bietet einen Überblick über programmierbare Hardware. Für viele Aufgaben reichen einfache PLDs, die jedoch durch den ebenso niedrigen Preis nach und nach von CPLDs ersetzt werden.
Für anspruchsvollere Aufgaben werden größere programmierbare Logikbausteine eingesetzt, die FPGAs. Das klassische Einsatzgebiet für FPGAs ist das Rapid Prototyping.
Durch die steigende Integrationsdichte bei der Chip-Produktion wachsen die Logikressourcen auf FPGAs, steigen die erreichbaren Taktraten und sinken die Preise stetig. Dies macht den Einsatz von FPGAs für immer größere Stückzahlen und immer mehr Einsatzgebiete rentabel.
Getrieben durch den Wunsch der Industrie nach FPGAs als Grundlage für Software Defined Radios, gibt es derzeit einen Aufschwung für die dynamisch partielle Rekonfiguration. Die großen Hersteller unterstützen sie bereits und vereinfachen ihren Einsatz durch Verbesserung ihrer Tool Chains stetig.
Derzeit lässt sich ein Trend weg vom Hard Core Prozessor hin zum Soft Core Prozessor auf FPGAs erkennen. Neben den stark beworbenen Soft Core Prozessoren der großen Hersteller existieren auch einige Open Source Prozessoren, deren Akzeptanz bei den Kunden ständig steigt.

3 Explicitly Parallel Instruction Computing

Die parallele Abarbeitung einzelner Befehle eines Programms ist eine der populärsten Möglichkeiten, Berechnungen zu beschleunigen. Um die vorhandene Parallelität in einem Programm ausschöpfen zu können, wurde seit Ende der 1980er Jahre das Explicitly Parallel Instruction Computing (EPIC) Paradigma eingeführt [89]. Um die Komplexität der Prozessoren gering zu halten, wird die Parallelität auf Instruktionsebene (Instruction-Level Parallelism, ILP) bereits vom Compiler gefunden. Die identifizierten nebenläufigen Befehle werden in flexible Befehlsgruppen gepackt.

3.1 Parallelität auf Instruktionsebene

Um ein vorhandenes Programm durch parallele Ausführung ohne Eingriff in den Code zu beschleunigen, sind Instruktionen so weit wie möglich gleichzeitig auszuführen. Um dabei die Korrektheit der Berechnung sicherzustellen, müssen potentiell nebenläufige Instruktionen, also solche Befehle, die sich wechselseitig nicht beeinflussen, identifiziert werden. Bei superskalaren Prozessoren geschieht dies im Prozessor, bei Very Long Instruction Word (VLIW) [36, 37] oder EPIC-Architekturen [107, 108] werden nebenläufige Instruktionen bereits vom Compiler bestimmt. Dieser Unterschied zwischen den Architekturen ist in Abbildung 3.1 verdeutlicht.

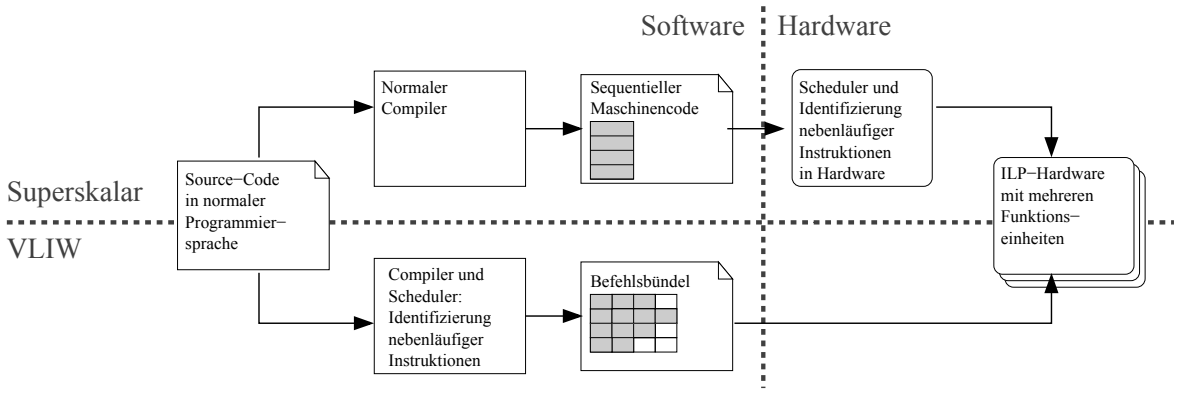


Abbildung 3.1: Superskalare und VLIW-Architektur

3.1.1 Abgrenzung zwischen EPIC, VLIW und superskalaren Prozessoren

In [114] wird vorgeschlagen, superskalare, EPIC- und VLIW-Architekturen anhand der Aufgabenteilung zwischen Compiler und Prozessor zu unterscheiden. Die drei Hauptaufgaben der ILP-Architektur sind dabei das Gruppieren der Befehle, die Zuteilung eines Befehls zur verarbeitenden Funktionseinheit und die Initiierung der Befehlsausführung.
Bei superskalaren Rechnern übernimmt die Hardware all diese Aufgaben. Im Gegensatz dazu werden sie bei einer streng klassischen VLIW-Architektur komplett vom Compiler übernommen. Führt der Compiler nur die Gruppierung der Befehle und der Prozessor sowohl Zuteilung als auch Initiierung durch, wird die Architektur als EPIC bezeichnet. Wie in [104] vorgeschlagen, werden die Architekturen, bei denen die Befehlsausführung von der Hardware initiiert wird, während die anderen genannten Aufgaben vom Compiler durchgeführt werden, Dynamic VLIW genannt. Diese Unterscheidung ist in Tabelle 3.1 dargestellt.

Tabelle 3.1: Unterscheidung Instruktions-Level-paralleler Architekturen (nach [114])

	Gruppierung der Befehle	Zuteilung zur FU	Initiierung der Ausführung
Superskalar	Hardware	Hardware	Hardware
EPIC	Compiler	Hardware	Hardware
Dyn. VLIW	Compiler	Compiler	Hardware
Klass. VLIW	Compiler	Compiler	Compiler

3.1.2 Superskalarität

In superskalaren Prozessoren wird der sequentielle Befehlsstrom zur Laufzeit im Prozessor analysiert. Nebenläufige Instruktionen, also solche, die nicht voneinander abhängig sind, werden dabei identifiziert und von mehreren Funktionseinheiten im Prozessor zeitgleich ausgeführt. Die Anzahl an gleichzeitig ausführbaren Befehlen hängt maßgeblich von der Menge der Befehle ab, welche in die Analyse auf Unabhängigkeit mit einbezogen werden.
Sollen zur Erhöhung der Parallelität mehr Funktionseinheiten auf dem Prozessor untergebracht und ausgelastet werden, müssen größere Mengen an Befehlen zeitgleich geladen, dekodiert und auf Abhängigkeiten hin untersucht werden. Je größer das Fenster an vorgeladenen und dekodierten Befehlen ist und je komplexer die Analysealgorithmen sind, desto höher ist auch der Bedarf an Hardware-Ressourcen.
Würden diese Ressourcen nicht benötigt, stünden sie für Caches und zusätzliche Verarbeitungseinheiten zur Verfügung oder könnten komplett eingespart werden. Somit benötigen superskalare Prozessoren meist mehr Chipfläche als andere, ähnlich parallel arbeitende Prozessoren.

Der Hauptvorteil von superskalaren Prozessoren ist, dass ihr Befehlssatz zu dem von sequentiellen Prozessoren voll kompatibel sein kann. Es ist dann nicht nötig, Programme neu zu kompilieren. Sie können direkt mit dem Geschwindigkeitsgewinn der parallelen Ausführung abgearbeitet werden.
Für superskalare Prozessoren können Compiler für sequentiellen Code benutzt werden. Diese sind nicht nur einfacher zu entwerfen, auch die Compile-Zeit ist wesentlich kürzer.
Klassische Beispiele für superskalare Prozessoren sind die MIPS R10000 [145] und die DEC Alpha-Architekturen[30]. Aufgrund der Kompatibilität kompilierter Programme zu früheren Prozessoren, sind nahezu alle modernen Prozessoren, wie Intels Core i7 Familie [55], superskalar.

3.1.3 Very Long Instruction Word

Bei Prozessoren, die auf VLIW-Technologie basieren, werden nebenläufig ausführbare Instruktionen bereits vom Compiler erkannt. Das kompilierte Programm besteht dann aus den namensgebenden, sehr langen Befehlswörtern. In diesen ist der Befehl für jede der parallel arbeitenden Funktionseinheiten des, in [36] als „dumm“ bezeichneten, Prozessors angegeben. Die Parallelität des Programms wird also statisch vom Compiler für eine bestimmte Zielarchitektur und deren Anzahl und Art an Funktionseinheiten vorgegeben.
VLIW-Maschinen benötigen keine Hardware zur Analyse des Befehlsstroms und sind somit sparsamer in Bezug auf die Chipfläche. Jedes lange Befehlswort enthält eine Instruktion für jede Einheit des Prozessors, wie in Abbildung 3.2 gezeigt. Wird eine Einheit im aktuellen

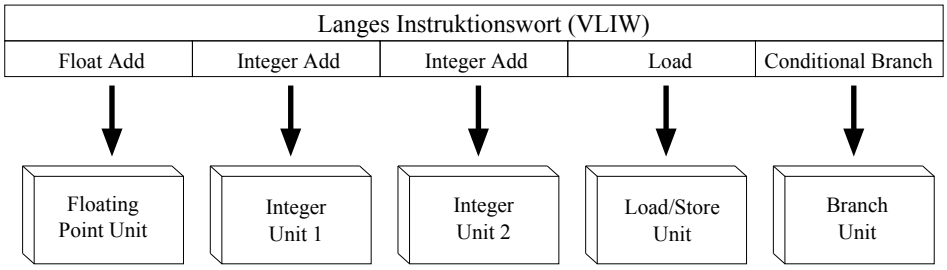


Abbildung 3.2: Beispiel eines Befehlswortes bei VLIW-Architekturen

Zyklus nicht benötigt, so wird die Stelle des Befehlswortes mit einer leeren Anweisung (No Operation, NOP) aufgefüllt. Somit sind die Anforderungen an die Speicherbandbreite hoch. Aufgrund der engen Verwobenheit zwischen Compiler und Prozessor sowie des exakt auf die Funktionseinheiten abgestimmten Befehlswortes müssen Programme für neue Prozessoren meist erneut kompiliert werden.
Eine der erfolgreichsten VLIW-Implementierungen ist die TMS320C6x-Familie von Texas Instruments, welche auf der VelociTI-Architektur [111] basiert und hauptsächlich in eingebetteten Systemen Einsatz findet. Transmetas Crusoe [65] und Efficeon [27] Prozessoren sind

ebenfalls VLIW-Prozessoren. Durch Code-Morphing-Strategien [28] erreichen sie Kompatibilität zu x86 Prozessoren.

3.2 Grundlagen der EPIC-Architektur

Die EPIC-Architektur basiert auf den Konzepten von VLIW und kann nicht scharf gegen diese abgegrenzt werden, da nach und nach viele EPIC-Konzepte in die VLIW-Definitionen aufgenommen wurden. Die EPIC-Idee stammt von Hewlett Packard und ist in [107] definiert. Basierend auf dieser Idee, entwickelte HP in einer Kooperation mit Intel die 64-Bit-Architektur IA-64 [49], welche der EPIC-Architektur mit wenigen zusätzlichen Konkretisierungen entspricht.
Wie bei VLIW werden bei EPIC nebenläufige Instruktionen bereits vom Compiler identifiziert. Die Architektur der Hardware vereinfacht es dem Compiler, einen eng und somit effizient gepackten Ausführungsplan (Plan of Execution, POE) zu erstellen. Von klassischen VLIW-Architekturen heben sich EPIC-Prozessoren vor allem durch das flexiblere Befehlsformat und die dadurch entstehende Flexibilität in der parallelen Verarbeitung ab. Durch die Unterstützung für spekulative Befehlsausführung und Predication erhöhen EPIC-Prozessoren den Grad der Instruktions-Level-Parallelität. Diese Anpassungen lasten vorhandene Funktionseinheiten besser aus, optimieren die nötige Speicherbandbreite und ermöglichen Kompatibilität zwischen Prozessoren verschiedener Generationen.

3.2.1 Flexible Gruppierung der Befehle

Bei EPIC-Prozessoren werden nebenläufige Befehle flexibel gruppiert. Im Gegensatz zu VLIW-Prozessoren ist es hier nicht nötig, dass jedes Bündel von Befehlen eine Instruktion für jede Funktionseinheit bereitstellt. Stattdessen ist ein Template, welches sich aus zusätzlichen Bits zusammensetzt, Bestandteil eines jeden langen Befehlsworts. In diesem ist angegeben, welcher Teilbefehl des langen Wortes von welcher Gruppe von Funktionseinheiten ausgeführt werden muss. Meist existieren Einheiten für Ganzzahl- und Fließkommaarithmetik sowie Verzweigungs- und Speicherverwaltungseinheiten.
Die Befehlsbündel werden bei EPIC-Prozessoren MultiOp genannt. Die in Art und Aufbau gewöhnlichen RISC (Reduced Instruction Set Computer) oder CISC (Complex Instruction Set Computer) Instruktionen entsprechen einzelnen Befehle einer MultiOp. Alle Instruktionen einer solchen MultiOp können immer gleichzeitig ausgeführt werden, ohne sich gegenseitig zu beeinflussen. Nach dem EPIC-Prinzip wird mit der Ausführung einer MultiOp erst dann begonnen, wenn die vorherige vollständig abgearbeitet wurde. Die mögliche Länge einer MultiOp ist nicht beschränkt.

3.2.2 Unterstützung für aggressive Parallelisierung

Um einen hohen Grad an Parallelität auf Instruktionsebene erreichen zu können, ist es Teil der EPIC-Definitionen, dass es der Aufbau des Prozessors dem Compiler erleichtert, Operationen als gleichzeitig ausführbar einzuplanen. Die dazu angewandten Techniken umfassen Spekulation, Predication, Rotation einer hohen Registerzahl und Software Pipelining.

3.2.2.1 Spekulation

Als Spekulation wird das Vorgehen bezeichnet, Befehle gleichzeitig auszuführen, auch wenn noch nicht sicher ist, ob sie sich gegenseitig beeinflussen. Der Grad der Instruktions-Level-Parallelität kann hierdurch erhöht werden. Man unterscheidet Kontroll- und Datenspekulation [36, S.163ff].

Kontrollspekulation
Unter Kontrollspekulation versteht man das Ausführen von Befehlen, bevor feststeht, dass sie ausgeführt werden müssen. Hierbei wird auf eine Verzweigung im Programm spekuliert, obwohl noch nicht sicher ist, dass das Programm sie wirklich nimmt. Somit werden die Befehle, die abhängig von ihr eintreten, ausgeführt.
Im Falle falscher Spekulation sind grundsätzlich Vorgehen notwendig, um die Effekte der fälschlich ausgeführten Befehle rückgängig zu machen. Dies kann entweder durch speziell erstellten Recovery Code oder durch vorherige Vermeidung direkter Einflüsse auf den späteren Programmablauf geschehen. Letzteres geschieht durch Verfahren, welche Ergebnisse als gültig oder ungültig markieren.
Falls genügend Verarbeitungseinheiten zur Verfügung stehen, können auch mehrere Zweige eines Programms gleichzeitig ausgeführt werden, um dann zu entscheiden, vom welchem die Ergebnisse weiter genutzt werden.

Datenspekulation
Unter Datenspekulation versteht man das Ausführen von Befehlen, bevor feststeht, dass sie zu diesem Zeitpunkt schon korrekt ausgeführt werden können. In der Praxis bedeutet dies meist, dass das Laden von Daten angestoßen wird, obwohl vor deren Verwendung noch Schreibebefehle ausgeführt werden könnten, welche die zu ladenden Daten verändern. Falls die Daten nicht verändert wurden, wird hierdurch die hohe Latenzzeit bei Speicherzugriffen eingespart. Das geladene Datum liegt ohne Wartezeit zum Zeitpunkt, an dem es verarbeitet werden soll, bereit.
Es besteht jedoch die Gefahr, dass die Daten nach dem Laden von anderen Instruktionen verändert werden oder Exceptions in der Programmausführung auftreten. Also müssen in Prozessoren, welche Datenspekulation betreiben, sowohl Verfahren zur Erkennung als auch zur Behebung dieser Probleme vorhanden sein, wie beispielsweise in [86] gezeigt ist.

Es ist möglich, manche Abhängigkeiten im Kontrollfluss zu Datenabhängigkeiten zu konvertieren, welche häufig leichter parallelisierbar sind [5].

3.2.2.2 Predication

Predication ist ein Verfahren, bei welchem durch das Schützen von Instruktionen mittels Prädikaten Kontrollflussabhängigkeiten in Datenabhängigkeiten überführt werden. Das Ergebnis solcher geschützter Befehle (Guarded Instructions [48]) wird erst dann für weitere Berechnungen freigeschaltet, wenn ein zugehöriges Prädikat einen bestimmten Wert hat. Dieses Prädikat kann zeitgleich berechnet werden und wird in einem Prädikatenregister gesetzt.
Da durch dieses Verfahren einige Sprünge überflüssig werden, werden die zusammenhängenden Programmblöcke länger. Dadurch können Einbußen durch falsche Spekulation (Misprediction Penalties) vermieden werden.
Folgendes einfache Beispiel zeigt ein Stück eines Assembler-Programms, welches mit und ohne Predication geschrieben wurde:

Assembler-Programm ohne Predication

```
        cmp R0, R1
        jne dest
        add R2, #1
        add R3, #1
dest:   add R0, #1
```

Assembler-Programm mit Predication

```
pred_cmp_eq R0, R1, p1
add R2, #1, <p1>
add R3, #1, <p1>
add R0, #1
```

Für den Fall, dass die beiden Register `R0` und `R1` ungleich sind, überspringt das Assembler-Programm einen Programmteil. Mit Predication wird für den Fall, dass die beiden Register gleich sind, das Prädikatenregister `p1` gesetzt. Die beiden folgenden Additionen sind Guarded Instructions und werden somit nur ausgeführt, falls das Prädikat `p1` gesetzt ist. Mit Predication kann das Programm ohne Sprünge ausgeführt werden.

3.2.2.3 Hohe Registerzahl

Um große Mengen paralleler Operationen zu erreichen, ist für EPIC-Architekturen grundsätzlich eine hohe Anzahl an Registern gefordert. Diese verhindert, dass Instruktionen, welche semantisch nicht voneinander abhängen, sequentiell ausgeführt werden, weil Register wieder benutzt werden müssen.
In Abbildung 3.3 ist ein einfaches Beispiel für die Beschränkung der ILP durch zu wenige freie Register dargestellt. In 3.3 a) stehen nur drei Register zur Verfügung, das Programmstück muss komplett sequentiell in vier Zyklen abgearbeitet werden. Allein ein zusätzliches Register (3.3 b)) ermöglicht hier die Ausführung des Codes in der halben Zeit.

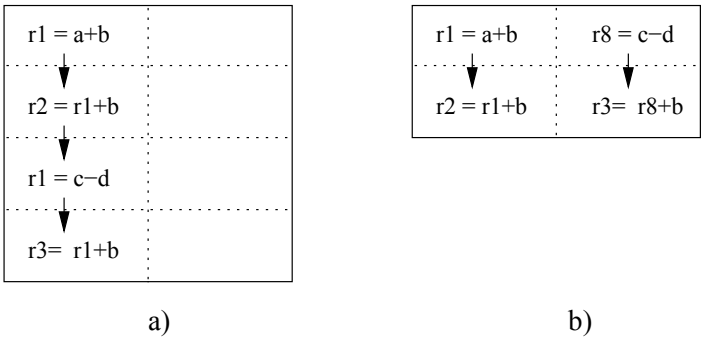


Abbildung 3.3: Einfluss der Registerzahl auf den Grad an ILP anhand eines kurzen Programmstückes: a) drei freie Register b) ein zusätzliches freies Register

3.2.2.4 Software Pipelining und Register Rotation

Loop Unrolling ist ein bekanntes Verfahren aus dem Compilerbau zur Beschleunigung von Programmen. Hierbei wird am Ende eines Schleifendurchlaufs der Aufwand für den Sprung an den Schleifenanfang (Schleifenprolog und -epilog) vermieden, indem der Schleifenkörper vervielfacht und somit mehrfach ausgeführt wird. Durch dieses Verfahren benötigen Programme jedoch mehr Speicher.
Register Roatation ist eine Fortführung dieses Konzeptes ohne die einhergehende Verlängerung des Programmcodes. Hierbei bietet die Hardware einen zusätzlichen Befehl, welcher einen Schleifenkörper mehrfach zeitgleich ausführt [29] und somit auch den Grad an ILP erhöht. Die benutzten Register des ursprünglichen Schleifenkörpers werden auf freie Register verlegt, was eine einfache Art des Register Renamings ist [113]. Die Anzahl der möglichen zeitgleich ablaufenden Schleifenkörper ist durch die Anzahl verfügbarer Register beschränkt. Sobald ein Schleifendurchlauf beendet ist, werden die dadurch wieder freien Register für den nächsten Durchlauf benutzt, sie werden also rotiert.
Register Rotation darf im Allgemeinen nur dann durchgeführt werden, wenn keine Datenabhängigkeiten zwischen den zeitgleich ausgeführten Schleifendurchläufen bestehen. Es ist nur dann sinnvoll, wenn Datenabhängigkeiten zwischen den einzelnen Befehlen eines Durchlaufes bestehen, da sonst bereits diese ohne zusätzlichen Aufwand zeitgleich ausgeführt werden können.
Abbildung 3.4 a) zeigt den Quell-Code einer einfachen Schleife. Mittels Register Rotation können mehrere Iterationen zeitgleich ausgeführt werden, wie in Abbildung 3.4 b) dargestellt. Damit die Schleifendurchläufe sich nicht ungewollt gegenseitig beeinflussen, werden die benutzten Register umbenannt. Zur Verdeutlichung hierfür werden im Beispiel den Registernamen zusätzliche Indizes mit der Nummer der Iteration der Schleife hinzugefügt. Bei dieser Form der Ausführung werden immer gleiche Befehle zur selben Zeit ausgeführt. Es werden also stets so viele Funktionseinheiten des entsprechenden Typs benötigt, wie Schleifenkör-

per zeitgleich ausgeführt werden. Software Pipelining [70] kann diesen potentiellen Engpass vermeiden, indem es die einzelnen Inkarnationen der Schleife zeitversetzt beginnt, wie in Abbildung 3.4 c) abgebildet. Dadurch werden zeitgleich stets unterschiedliche Operationen ausgeführt.
Geht man exemplarisch davon aus, dass die Operationen A_i und B_i von einem Funktionseinheitentyp I ausgeführt werden können, während C_i von ein Funktionseinheitentyp F ausgeführt werden muss, benötigt die Abarbeitungsfolge in 3.4 b) von jedem der beiden Typen je drei, während in c) zwei FUs vom Typ F und eine vom Typ I ausreicht.

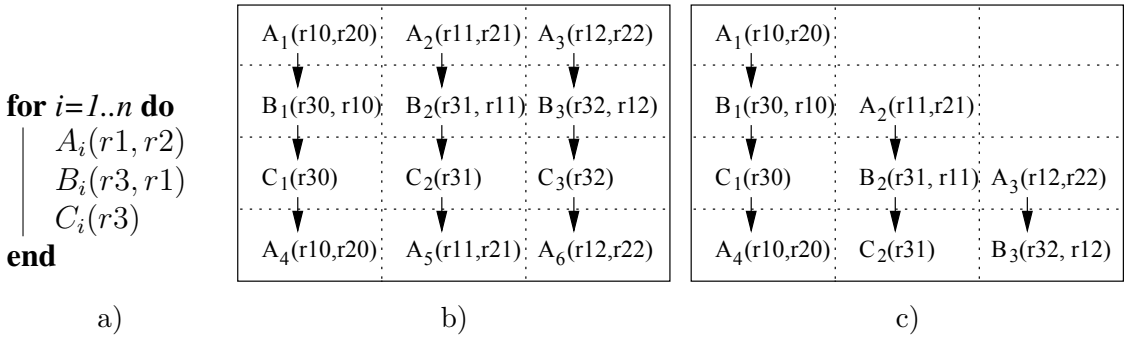


Abbildung 3.4: Beispiel für Register Renaming und Software Pipelining a) Source-Code einer einfachen Schleife b) Parallele Ausführung der Schleife mit Register Renaming c) Ausführung der Schleife mit Register Renaming und Software Pipelining

3.2.3 Besonderheiten bei der Ausführung von EPIC-Code

Bei EPIC-Prozessoren sind die Befehle, wie in Abschnitt 3.2.1 beschrieben, flexibel in Bündel gruppiert. Um dadurch eine entsprechende Beschleunigung bei der Ausführung des Codes zu erreichen, muss sichergestellt werden, dass möglichst so viele Befehle, wie passende Funktionseinheiten im Prozessor vorhanden sind, zur Verfügung stehen. Hierfür wird ein Instruction Cache mit Instruktionen befüllt. Die nebenläufigen Instruktionen eines Bündels werden an sogenannte Issue Slots verteilt, wobei jeder Issue Slot genau einer Funktionseinheit zugeordnet ist.
Im Regelfall sollte EPIC-Code so beschaffen sein, dass alle Befehle eines Bündels in einem Zyklus abgearbeitet werden können. Besitzt der Compiler Informationen über den Aufbau des Prozessors, so kann er dies bei statischen Prozessorarchitekturen sicherstellen und den Aufbau der Bündel dahingehend optimieren.
Jedoch widerspricht das EPIC-Paradigma auch nicht dem Vorgehen, ein Bündel nebenläufiger Instruktionen teilweise sequentiell abzuarbeiten. Hierzu muss lediglich vom Compiler sichergestellt werden, dass Vor- und Nachbereich der gebündelten Befehle disjunkt sind, ihre sequentielle Abarbeitung also nicht unsicher wird. Dieses Vorgehen ermöglicht es, dass un-

terschiedliche EPIC-Prozessoren untereinander kompatibel sein können. Selbst wenn sie eine unterschiedliche Anzahl an Funktionseinheiten haben, kann der für sie optimierte Code auch auf anderen EPIC-Prozessoren ausgeführt werden.

3.3 Kompilation für EPIC-Prozessoren

Bei EPIC-Architekturen sind Compiler und Prozessor enger miteinander verwoben. Um die Komplexität der Hardware geringer zu halten, wird der Plan of Execution bereits vom Compiler erstellt, der Prozessor muss Nebenläufigkeiten der Befehle nicht mehr identifizieren, um von der ILP Nutzen zu ziehen.
Um einen möglichst hohen Grad an ILP zu erzielen, müssen EPIC-Compiler die Methoden zur Parallelisierung von Programmen (Abschnitt 3.2.2) unterstützen. Es ist Aufgabe des EPIC-Compilers, Kontroll- und Datenspekulationsfähigkeiten des Prozessors so einzusetzen, dass der Prozessor ausgelastet bleibt. Um möglichst viele Befehle zeitgleich ausführen zu können, müssen die kritischen Pfade im Programmfluss so kurz wie möglich gehalten werden [62]. Hierfür ist zeitaufwändige Code-Analyse notwendig.
Auch wenn viele Probleme der Compilation für EPIC-Prozessoren gelöst sind, so bleibt sie trotzdem der wohl wichtigste Faktor für den langfristigen Markterfolg von EPIC-Architekturen. Etwas schwarz sieht hier Donald Knuth: „...the Itanium approach that was supposed to be so terrific – until it turned out that the wished-for compilers were basically impossible to write“[66].

3.4 Grenzen der Parallelität auf Instruktionsebene

In realen Systemen ist es nicht möglich, Programme beliebig zu parallelisieren. Sowohl die verarbeitende Hardware als auch Daten- und Kontrollflussabhängigkeiten im Programm beschränken dies.

3.4.1 Beschränkung durch Hardware

Parallelität auf Instruktionsebene ist durch die Hardware, auf der das Programm ausgeführt wird, beschränkt. Offensichtlich können pro Zeiteinheit nicht mehr Befehle ausgeführt werden, als freie Verarbeitungseinheiten zur Verfügung stehen. Die beschränkenden Faktoren sind hier die Anzahl der Funktionseinheiten und die Speicherbandbreite.
Technisch ist es wenig problematisch, diese Faktoren stark zu erhöhen. Dies kann für viele Anwendungsbreiche auch wirtschaftlich sein, wenn dadurch ein entsprechender Anstieg der Rechenleistung gewährleistet wird.

3.4.2 Beschränkung durch Abhängigkeiten

Schwieriger zu bestimmen ist die Beschränkung der Parallelität durch Abhängigkeiten im Programm. In Walls Studie von 1991 [128] wurden mehrere Benchmarks ausgeführt, um dann die Instruktionen möglichst optimal in maximal 64 parallel ausführbare zu zerlegen. Ihr Ergebnis ist, dass Parallelität auf Instruktionsebene selten den Faktor sieben übersteigt. Sie besagt jedoch auch, dass perfekte Vorhersage der Verzweigungen diesen Faktor für die meisten Benchmarks mindestens auf einen Faktor sechzehn bis hin zu 60 heben könnte. Dies entspricht auch in etwa den sehr frühen Ergebnissen aus [93], welche besagen, dass mit extrem hohem Aufwand bei der Kompilation ein Faktor 90 realistisch sein kann.
Mit den Grenzen der Parallelisierbarkeit speziell für EPIC-Architekturen beschäftigt sich [78]. Auch hierfür wird unter normalen Umständen der Faktor im Bereich sieben genannt. Jedoch werden auch Techniken analysiert, welche durch aufwändigere Kompilation den Grad der Parallelisierung auf ca. 30 heben.
In [71] wird der Kontrollfluss als das beschränkende Element der Parallelisierbarkeit auf Instruktionsebene identifiziert. Gleichzeitige Ausführung weiter entfernter Teile des Codes wird als Lösung vorgeschlagen.
Formal findet genau dies in [79] statt. Um die Anzahl parallel ausführbarer Befehle zu erhöhen und somit die vorhandene Hardware besser auszulasten, werden hier gleichzeitig ablaufende Ausführungsstränge (Thread-Level Parallelism, TLP) in Instruktions-Level-Parallelität konvertiert.

3.5 HPL-PD und Trimaran

Die Play-Doh-Architektur der Hewlett Packard Laboratories (HPL-PD) [61] ist eine parametrisierbare Prozessor Meta-Architektur zur Forschung an ILP-Architekturen. Auch wenn sie Superskalarität und VLIW unterstützt, ist ihr Hauptziel die Förderung der Forschung an EPIC-Architekturen.
Aus der HPL-PD, dem IMPACT-Project der University of Illinois und dem ReaCT-ILP-Projekt der New York University ging Trimaran, eine Kombination aus Compiler und Simulator für HPL-PD-Architekturen, hervor [22]. Der Quellcode von Trimaran ist frei zugänglich.
Die HPL-PD legt sowohl den unterstützten Befehlssatz der Maschine, aus dem ein Subset gewählt werden kann, wie auch die Art der Register fest. Sie lässt jedoch die Anzahl und Art der Funktionseinheiten, die Anzahl der Register, die Befehlslänge, sowie Ressourcenbedarf und Latenz der Instruktionen variabel. In Trimaran werden diese Parameter durch eine Maschinenbeschreibung in HMDES [44] festgelegt.
Der Aufbau des Trimaran Frameworks ist in Abbildung 3.5 dargestellt. Es besteht aus dem IMPACT-Modul, welches ein maschinenunabhängiges Compiler-Front-End ist und aus C-Code eine Zwischenrepräsentation erzeugt. Zusammen mit der Maschinenbeschreibung dient

diese als Eingabe für das Compiler-Back-End ELCOR. Dieses generiert Maschinenprogramme für EPIC-Prozessoren. Diese können als Eingabe für die ReaCT-ILP Simulation dienen, welche abhängig von der Maschinenbeschreibung Ausführungsstatistiken produziert. Die Ergebnisse können wiederum zur Optimierung der Zwischenrepräsentation genutzt werden.

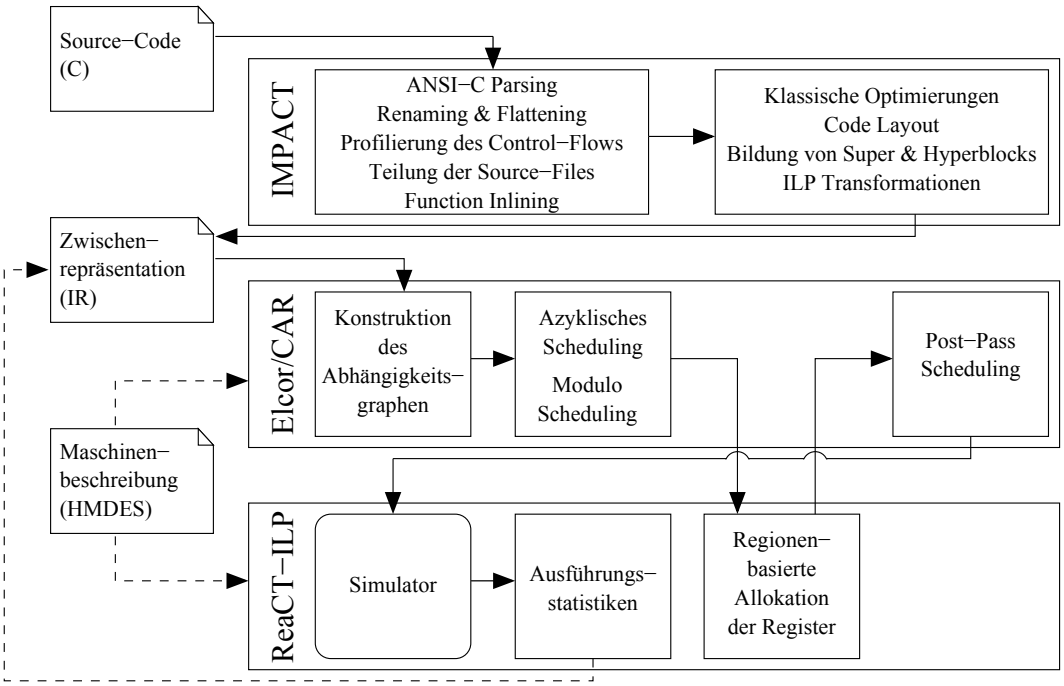


Abbildung 3.5: Aufbau des Trimaran Frameworks (nach [122])

Trimaran unterstützt Kontroll- und Datenspekulation sowie Predication und Register Rotation. Es kann dem Compiler die Kontrolle über die Speicherhierarchie geben und umfasst Mechanismen zur schnellen Vereinfachung von booleschen Ausdrücken. Zudem ermöglicht es die Anwendung von Methodiken wie gleichzeitiges Schreiben in Register sowie die Entfernung redundanter Load-Store-Operationen und Register-zu-Register-Schreibeoperationen in Schleifen.

3.6 Intel Itanium

Ein reales Beispiel für die Umsetzung von Prozessoren, die dem EPIC-Paradigma genügen, ist die auf der IA-64-Architektur [49] basierende Intel Itanium-Familie. IA-64 ist eine Umsetzung der implementierungsunabhängigen HPL-PD-Definitionen (vgl. Kapitel 3.5). Eingeführt wurden die ersten Itanium-Prozessoren, die auf dem Merced-Kern [112] basierten, im Jahre 2001. Deren Rechenleistung war jedoch aufgrund hoher Latenzzeiten der Caches enttäuschend. Diese Probleme wurden beim im Jahr 2002 erschienenen Itanium 2 [82] weitestgehend behoben.

Um Kompatibilität zu x86 Prozessoren zu gewährleisten, sind beim Itanium und Itanium 2 zusätzlich zu den IA-64-Instruktionen auch sämtliche IA-32-Befehle einsetzbar. Eine interne Dekodiereinheit passt diese an.
Ein grober Überblick über den Aufbau von Itanium-Prozessoren ist in Abbildung 3.6 gegeben. Eine genauere Erläuterung der Architektur folgt in den Abschnitten ab 3.6.3.

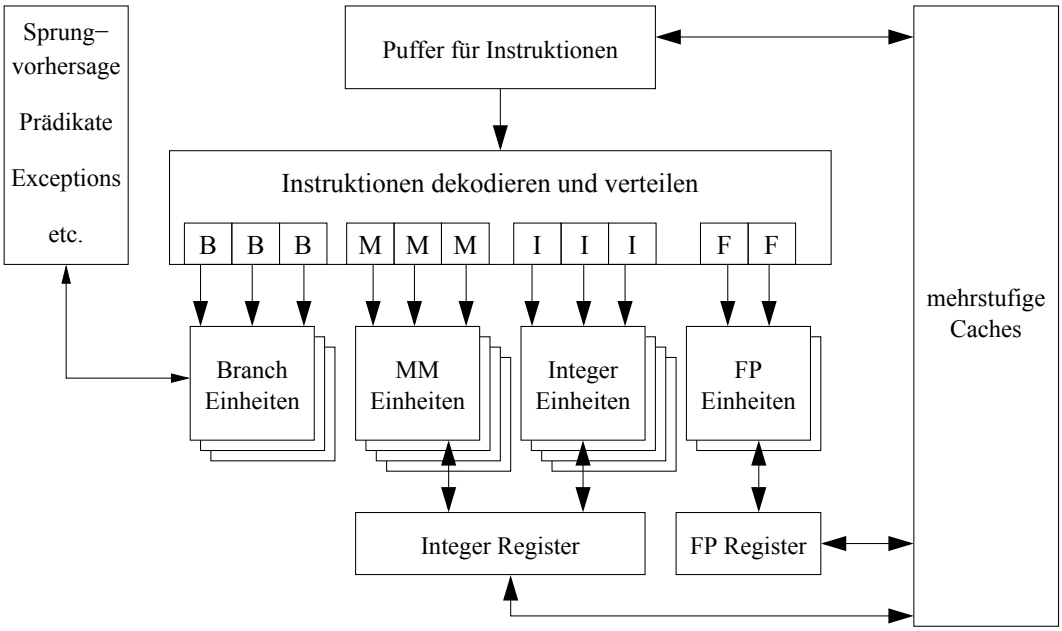


Abbildung 3.6: Stark vereinfachtes Blockdiagramm des Itanium-Prozessors

3.6.1 Mehrkern Itanium-Prozessoren

Seit 2006 werden Itanium 2-Prozessoren mit Dual Core angeboten [81]. Der als Itanium 9000 oder Montecito bezeichnete Prozessor wurde in 90-nm-Technologie gefertigt. Erstmals bietet er Hardware-gestütztes Dual-Threading an. Als erster Itanium wurde bei der 9000-Familie keine Hardware zur Ausführung von IA-32-Befehlen mehr implementiert.
Im Jahre 2007 folgte mit der Itanium 9100-Familie (auch Montvale genannt) ein Dual Core Prozessor, welcher sich nur geringfügig vom Vorgänger unterscheidet.
Seit Anfang 2010 sind Prozessoren der in 65-nm-Technolgie gefertigten Itanium 9300-Familie (Code-Name Tukwila) [115] erhältlich. Die vier Kerne des Prozessors sind über einen 12 Port Crossbar Router untereinander sowie mit den I/O Interfaces und zwei Speicher Controllern verbunden. Jeder der Kerne verfügt über etwa 1 MByte Level 2 Cache und 6 MByte Level 3 Cache. Ein vereinfachtes Blockdiagramm des Tukwila Prozessors ist in Abbildung 3.7 zu sehen. Sämtliche Multi Core Itanium-Prozessoren basieren auf dem Itanium 2 mit geringfügigen Änderungen.

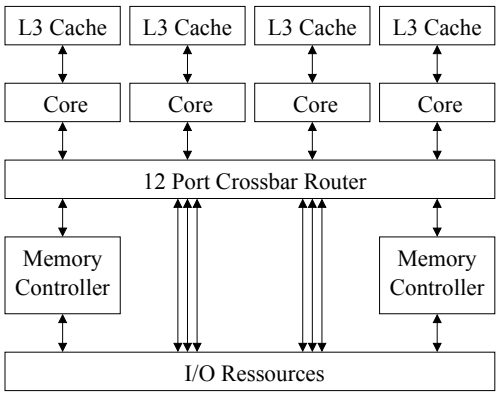


Abbildung 3.7: Blockdiagramm des Itanium 9300 Quad Core Prozessors

Die nächste Itanium-Generation (Code-Name Poulson) [102] wird voraussichtlich 2012 erscheinen. Sie soll in 32-nm-Technologie gefertigt werden, aus acht Cores mit verbessertem Multi-Threading bestehen und 50 MByte Caches bieten. Für 2014 ist bereits dessen Nachfolger, der Kittson, geplant.

3.6.2 Einsatzgebiet und Markt

Ursprünglich sollte der Itanium die x86 Prozessoren für Desktop-Rechner ablösen. Dies scheiterte jedoch bereits bei der ersten Itanium-Generation an mangelnder Software und zu langsamer x86-Emulation. Seither sind Itanium-Prozessoren in erster Linie für den Server-Betrieb konzipiert.
Bis heute basieren einige Supercomputer auf dem Itanium. Vor allem im Jahr 2004 waren diese aktuell, als zeitweise 84 der 500 leistungsfähigsten Supercomputer der Welt aus Itanium-Prozessoren aufgebaut waren [120].
Heute werden Itanium-Prozessoren in erster Linie für Zuverlässigkeit, Skalierbarkeit und Energieeffizienz im Serverbetrieb beworben [56].
Blieben die Verkaufszahlen des Itanium in den ersten Jahren weit hinter den Erwartungen zurück [25], soll laut [92] Intel inzwischen Gewinn mit den Prozessoren erwirtschaften. Auch wenn sich viele große Firmen, wie Microsoft oder Red Hat, aus dem Itanium-Geschäft zurückziehen, plant Intel, wie in Abschnitt 3.6.1 beschrieben, weitere Generationen zu fertigen.

3.6.3 Funktionseinheiten

Alle Prozessoren der Itanium-Familie haben vier verschiedene Typen von Funktionseinheiten:

- *Integer Units*: Einheiten zur Verarbeitung von ganzen Zahlen, auch für Schiebeoperationen, Festkommaoperationen

- *Memory Units*: Einheiten zur Kommunikation mit der Speicherhierarchie, zum Laden und Speichern und (wie Integer-Einheiten) auch für einfache arithmetische und logische Operationen, da der Datenpfad über eine ALU (arithmetisch-logische Einheit) führt, ursprünglich zum Inkrementieren der Adressen
- *Floating Point Units*: Einheiten zur Verarbeitung von Fließkommabefehlen
- *Branch Units*: Einheiten zur Durchführung von bedingten sowie unbedingten Sprüngen im Programm, für Unterprogrammaufrufe und Rücksprünge aus Unterprogrammen

Die ersten Itanium-Prozessoren hatten je vier Integer- und Memory-Einheiten, zwei Fließkommaeinheiten sowie drei Branch-Einheiten [112]. Beim Itanium 2 wurden zu diesen je zwei zusätzliche Integer- und Memory-Einheiten implementiert [82]. Um hohe Taktfrequenzen zu erlauben, sind die Funktionseinheiten in zehn Pipelinestufen aufgegliedert.

3.6.4 Befehlswort

Ein Befehlsbündel ist bei Itanium-Prozessoren 128 Bit lang. Es besteht aus drei 41 Bit langen Instruktionen sowie einem fünf Bit langem Template, vgl. Abbildung 3.8 [54]. Im Template wird sowohl die Stelle der Bündelgrenzen als auch die Funktionseinheiten, für welche die Instruktionen bestimmt sind, codiert.
Maximal zwei dieser Befehlsworte, also sechs Befehle, könne zeitgleich ausgeführt werden.

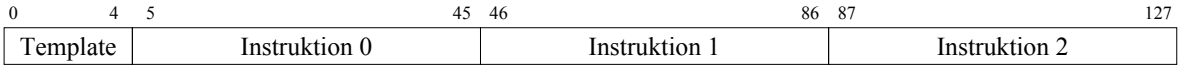


Abbildung 3.8: Aufbau des Befehlswortes von Itanium-Prozessoren

Bei den vier verschiedenen Funktionseinheiten und drei verschiedenen Stellen für Bündelgrenzen müssten mindestens $4^3 * 2^3 = 512 = 2^9$ verschiedene Templates existieren, um alle möglichen Befehls- und Grenzkombinationen zu kodieren. Zusätzlich gibt es auch noch einige Befehle, welche doppelte Länge, also 82 Bit, haben. Tatsächlich werden aus Gründen der Effizienz beim Itanium nur 24 unterschiedliche Templates unterstützt. Andere Befehlskombinationen müssen vom Compiler vermieden werden. Die Templates sind in Tabelle 3.2 dargestellt, wobei die Buchstaben die jeweilige Funktionseinheit für den Befehl und eine doppelte vertikale Linie eine Bündelgrenze darstellen. Dabei steht M für einen Memory-Befehl, I für einen Integer-Befehl, F für einen Floating-Point-Befehl und B für einen Branch-Befehl. Die mit L X angegebenen Long-Extended-Befehle haben doppelte Länge und können auf einer Branch- oder Integer-Einheit ausgeführt werden. Die nicht dargestellten Templates sind nicht belegt.

Tabelle 3.2: Templates für Itanium-Prozessoren

Template	Slot 0	Slot 1	Slot 2	Template	Slot 0	Slot 1	Slot 2	Template	Slot 0	Slot 1	Slot 2
00	M	I	I	0A	M	M	I	12	M	B	B
01	M	I	I ‖	0B	M	M	I ‖	13	M	B	B ‖
02	M	I ‖	I	0C	M	F	I	16	B	B	B
03	M	I ‖	I ‖	0D	M	F	I ‖	17	B	B	B ‖
04	M	L	X	0E	M	M	F	18	M	M	B
05	M	L	X ‖	0F	M	M	F ‖	19	M	M	B ‖
08	M	M	I	10	M	I	B	1C	M	F	B
09	M	M	I ‖	11	M	I	B ‖	1D	M	F	B ‖

3.6.5 Parallele Befehlsverarbeitung

Die Prozessoren der Itanium-Familie sind darauf ausgelegt, zwei Befehlsworte zeitgleich zu laden und auszuführen. Durch die beschränkte Anzahl der Verarbeitungseinheiten und Verbindungen zwischen den Komponenten des Prozessors sind jedoch nicht alle Kombinationen an Befehlen parallel ausführbar, selbst wenn keine Abhängigkeiten zwischen ihnen bestehen. Beim Schritt vom Itanium zum Itanium 2 wurden die möglichen Kombination erweitert, was die Rechenleistung annähernd verdoppelt. Die zeitgleich ausführbaren Operationen von Itanium und Itanium 2 sind in Tabelle 3.3 dargestellt [121].

Tabelle 3.3: Zeitgleich ausführbare Befehlskombinationen für Itanium- und Itanium 2-Prozessoren (nach [121])

	MII	MLI	MMI	MFI	MMF	MIB	MBB	BBB	MBB	MFB
MII	◑	○	◑	◑	◑	◑	●	●	◑	●
MLI	◑	◑	◑	●	◑	●	◑	●	◑	●
MMI	◑	◑	◑	◑	◑	◑	◑	●	◑	◑
MFI	◑	●	◑	●	◑	●	●	●	◑	●
MMF	◑	◑	◑	◑	◑	◑	◑	●	◑	◑
MIB	◑	●	◑	●	◑	●	●	○	◑	●
MBB	○	○	○	○	○	○	○	○	○	○
BBB	○	○	○	○	○	○	○	○	○	○
MBB	◑	◑	◑	◑	◑	◑	◑	○	◑	◑
MFB	●	●	◑	●	◑	●	●	○	◑	●

Kombination unterstützt bei
●: Itanium und Itanium 2 ◑: nur Itanium 2 ○: nicht unterstützt

3.6.6 Hardwareunterstützung zur Spekulation

Die Hardware des Itanium bietet sowohl Unterstützung für Kontroll- als auch für Datenspekulation.
Um Kontrollspekulation zu ermöglichen, können fälschlich genommene Abzweigungen mit ei-

nem NaT-Bit (Not-A-Thing) markiert werden, um die Ausführung von Compiler-generiertem Recovery Code anzustoßen.
Für Datenspekulation, also vorgezogenem Laden, wurde beim Itanium eine Advanced Load Address Table, ALAT, implementiert. In ihr werden spekulativ geladenen Daten markiert. Sollte eine der folgenden Operationen die Daten verändern, so wird auch die Markierung aus der ALAT gelöscht und die Daten müssen, sobald sie benötigt werden, erneut geladen werden.

3.7 Zusammenfassung und Ausblick

Bei der Beschleunigung der Programmausführungen von Prozessoren durch die Ausnutzung von Parallelität auf Instruktionsebene unterscheidet man verschiedene Architekturen anhand der Aufgabenteilung zwischen Compiler und Hardware. Einen Mittelweg zwischen den sehr Hardware-bestimmten superskalaren und den Compiler-orientierten VLIW-Prozessoren, gehen hierbei die Prozessoren, die auf der EPIC-Architektur basieren. Bei ihnen übernimmt der Compiler die flexible Gruppierung parallel ausführbarer Befehle, deren Zuteilung zur verarbeitenden Einheit ist Aufgabe der Hardware.
Sind der erreichbaren Parallelität auf Instruktionsebene auch Grenzen gesetzt, so bietet die EPIC-Architektur doch Ansätze, sie möglichst weit zu steigern. Zu diesen gehören Spekulation, Predication und Register Rotation. Um diese effizient einzusetzen, sind jedoch aufwändige Compiler nötig, deren Erstellung schwierig ist.
Die einzigen EPIC-Prozessoren, die bisher in großen Stückzahlen gefertigt wurden, sind die Server-Prozessoren der Intel Itanium-Familie. Auch wenn ihre Akzeptanz auf dem Markt nach wie vor hinter den Erwartungen zurückbleibt, sind trotzdem auf viele Jahre weiterhin neue Generationen geplant.
Es ist davon auszugehen, dass diese Tatsache die Entwicklung der Compiler-Technologie für EPIC-Prozessoren weiter vorantreibt. Mit besseren Compilern und der steigenden Integrationsdichte bei der Chip-Fertigung, besteht in Zukunft die Chance, dass EPIC-Prozessoren aus der Nische der Server-Prozessoren herauskommen und in anderen Gebieten Einsatz finden.

4 Adaptierbare Prozessoren

Um die Abarbeitung von Programmen durch Prozessoren zu beschleunigen, besteht die Möglichkeit, die massive Parallelität, welche die direkte Abarbeitung durch Spezial-Hardware bietet, mit üblichen, meist rein sequentiell arbeitenden Prozessoren zu verbinden. Größere Flexibilität erhalten solche speziellen Prozessoren durch den Einsatz von programmierbarer Hardware, sowohl nur für spezielle Komponenten als auch für die Implementierung des kompletten Prozessors als Soft Core. Diese können auf das zu bearbeitende Problem optimiert werden und lassen sich sogar während der Laufzeit des Programms anpassen.
Kern dieser Arbeit ist die Definition und Analyse einer Prozessorarchitektur, welche sich durch dynamische partielle Rekonfiguration an ihr umgebendes System anpasst. Hierzu nutzt sie Eigenschaften der EPIC-Architektur.
Deswegen gibt dieses Kapitel einen Überblick über den Stand der Entwicklung auf diesem Gebiet, indem es innovative adaptive Prozessoren aufzeigt. Besonderes Augenmerk wird dabei auf spezialisierte Prozessoren gelegt, welche sich, wie der in dieser Arbeit vorgeschlagene Ansatz, die Vorteile der EPIC-Architektur zunutze machen.

4.1 Einordnung adaptiver Prozessoren

Berechnungen können sowohl mit festverdrahteter Hardware als auch als Software, welche dann von einem Prozessor abgearbeitet wird, ausgeführt werden. Hardware-Lösungen führen die Berechnungen meist schneller und mit geringerem Energieverbrauch durch. Software-Implementierungen sind demgegenüber meist kostengünstiger, flexibler und schneller umzusetzen. Es existieren zudem einige Mischformen aus Prozessoren und festverdrahteten Bausteinen.
Noch sehr nah verwandt mit gewöhnlichen Prozessoren (General Purpose Processors, GPP) sind die anwendungsspezifischen Prozessoren (Application Specific Instruction Set Processor, ASIP). Bei diesen Prozessoren ist der Befehlssatz für eine bestimmte Problemklasse optimiert. Das kann zum einen bedeuten, dass sie spezielle Befehle haben, welche Teilprobleme effizienter lösen. Zum anderen wird ein Prozessor auch als ASIP bezeichnet, wenn Befehle, welche für die spezifische Problemklasse nicht erforderlich sind, aus Platz- und Kostengründen nicht Teil des Befehlssatzes (Instruction Set Architecture, ISA) sind.
Bei konfigurierbaren Prozessoren kann der Aufbau des Prozessors verändert werden, indem vorgefertigte Komponenten entsprechend verbunden und angepasst werden. Dies kann vor

der Fertigung eines Prozessors geschehen, aber auch erst vor der Konfiguration eines Prozessors auf einen konfigurierbaren Baustein. In zweitem Fall nennt man den Prozessor Soft Core Prozessor (vgl. Abschnitt 2.5.3). Anpassungen am Prozessor betreffen meist die ISA, die Wortbreite, die Anzahl der Pipelinestufen, Einheiten zur Peripherie- und Busanbindung oder auch die Anzahl an parallel arbeitenden Funktionseinheiten.
Einen weiteren Schritt in Richtung Verarbeitung von Programmen in Hardware gehen rekonfigurierbare Prozessoren. Bei dieser Klasse können die Eigenschaften des Prozessors während der Abarbeitung einer Aufgabe geändert werden. Der Grund für diese Anpassung kann beispielsweise die Optimierung auf das zu bearbeitende Problem oder eine veränderte Menge an Hardwareressourcen sein.
Geschieht die Veränderung selbständig durch den Prozessor, entweder durch spezielle Befehle der ISA, Nutzereingaben oder andere äußere und innere Einflüsse, so ist der Prozessor adaptiv.
Noch stärker konfigurierbar und somit flexibler ist die Verbindung von Prozessoren mit frei programmierbarer Hardware. Hiermit ist jedoch auch stark erhöhter Entwicklungsaufwand für Anwendungen verbunden.
Wird noch höhere Performanz benötigt, können Probleme auch direkt mit festverdrahteter, einfach oder auch partiell dynamisch konfigurierbarer Spezial-Hardware ohne Prozessor gelöst werden. Wegen des ungleich aufwändigeren Entwicklungs- und Testprozesses betragen dabei die Entwicklungskosten jedoch im Regelfall ein Vielfaches einer Lösung mit einem einfachen Prozessor.

4.2 Konfigurierbare Soft Core Prozessoren

Einer der Vorteile von Soft Core Prozessoren (vgl. Abschnitt 2.5.1) ist ihre Anpassbarkeit an die zu erledigende Aufgabe. Selbst wenn dies meist nicht zur Laufzeit möglich ist, so können sie trotzdem als adaptiv angesehen werden. Dieser Abschnitt betrachtet Soft Core Prozessoren, die ohne aufwändiges Eingreifen in die Hardware-Beschreibung an ihr Einsatzgebiet angepasst werden können.

4.2.1 MicroBlaze

Der 32-bit Soft Core Prozessor MicroBlaze der Firma Xilinx wurde in Kapitel 2.5.3.3 bereits beschrieben. Seine Hardware-Beschreibung ist nicht frei erhältlich.
Rekonfiguration zur Laufzeit ist beim MicroBlaze nicht vorgesehen. Allerdings besteht die Möglichkeit, seine Komponenten und seine Struktur vor der Synthese mittels eines Konfigurationsprogramms anzupassen. Somit kann seine Leistung auf spezielle Anwendungsklassen optimiert werden, ohne dabei unnötig Chipfläche für ungenutzte Komponenten zu verbrauchen.
So kann beispielsweise beim MicroBlaze ausgewählt werden, ob die Pipeline drei oder fünf

Stufen hat. Weiterhin können je nach Bedarf Fließkommaeinheiten, eine Memory Management Unit (MMU), Multiplizierer und Dividierer verschiedener Wortlängen, Barrel-Shifter zur schnellen Ausführung von Schiebeoperationen, sowie Caches durch Konfiguration zum Prozessor hinzugefügt werden.

4.2.2 Leon3

Der bereits in Kapitel 2.5.3.2 vorgestellte Leon3 Prozessor von Gaisler Research ist bei der Implementierung stark konfigurierbar. Laufzeitrekonfiguration ist beim Leon3 nicht vorgesehen. Durch den offengelegten Quellcode der Hardware-Beschreibung ist es möglich, Anpassungen des Prozessors direkt zu implementieren. Viele Eigenschaften des Prozessors lassen sich über eine grafische Oberfläche (siehe Abbildung 4.1) und entsprechende Konfigurations-Files festlegen. Zu den abänderbaren Eigenschaften gehören vor allem Peripherieanbindungen, die Konfiguration der Caches sowie Unterstützung für Fließkommaeinheiten, Multiplizierer und Dividierer.

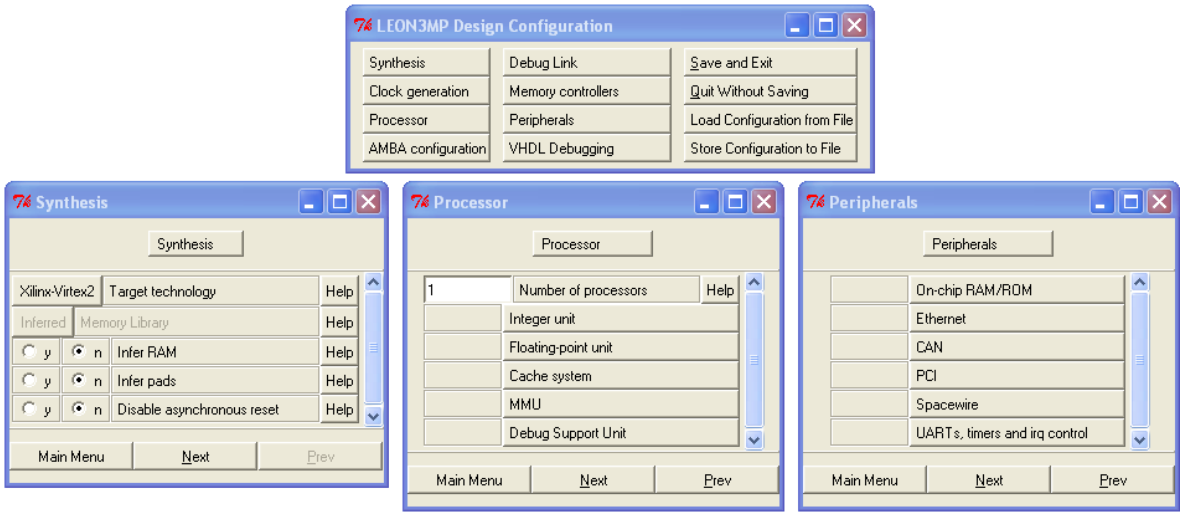


Abbildung 4.1: Grafische Oberfläche zur Konfiguration des Leon3 Prozessors

4.3 Prozessoren mit rekonfigurierbaren Koprozessoren

Die Koppelung eines Standard-GPPs mit einem rekonfigurierbaren Baustein, der als Koprozessor arbeitet, ist eine bereits seit längerer Zeit angewandte Möglichkeit, die Vorteile der parallelen Arbeitsweise von Berechnungen in Hardware zu nutzen. Die meisten Ansätze in diesem Bereich unterscheiden sich hauptsächlich in der Art der Kopplung zwischen Prozessor und rekonfigurierbarem Array. Trotz unterschiedlicher Programmierparadigmen ist für alle

Prozessoren mit rekonfigurierbarem Koprozessor der Aufwand für die Implementierung von effizienten Programmen verhältnismäßig hoch.

4.3.1 Garp

Ein früher Ansatz, welcher rekonfigurierbare Hardware als Koprozessor einsetzt, ist der Garp Prozessor [47] der University of California in Berkeley aus dem Jahr 1997. Auf einem Die sind ein Standard-MIPS-II-Prozessorkern und FPGA-Zellen, welche sich an den CLBs der Xilinx 4000er Serie orientieren, integriert. Der Befehlssatz des MIPS wurde beim Garp um Befehle zur Konfiguration der FPGA-Zellen sowie zum Datenaustausch zwischen Prozessor und rekonfigurierbarem Koprozessor erweitert.

4.3.2 Molen Reconfigurable Processors

Das Molen Processor Konzept [124] der TU Delft beschreibt Prozessoren, welche sich rekonfigurierbarer Hardware bedienen, um ihre Berechnungen zu beschleunigen. Es bezieht auch partielle Rekonfiguration, also den teilweisen Austausch der Berechnungseinheiten in der rekonfigurierbaren Hardware, mit ein.
Um die Rekonfiguration der Hardware durchzuführen, werden bei Molen Prozessoren spezielle zusätzliche Mikroinstruktionen, der sogenannte $\rho\mu$-Code, benutzt. Diese Instruktionen entstehen durch Übersetzung der Konfigurations-Files der programmierbaren Hardware.

4.3.3 Reconfigurable Processor Units

Eine verfügbare kommerzielle Familie rekonfigurierbarer Koprozessoren sind die Reconfigurable Processor Units (RPU) der Firma DRC [32]. Diese arbeiten als Koprozessoren für AMD Opteron Prozessoren. Durch die Möglichkeit, sie direkt in einen freien Prozessorsockel eines Multiprozessor-Mainboards einzusetzen, erreichen sie hohe Bandbreiten zu Speicher und Host-Prozessor. Die RPUs setzen gewöhnliche Xilinx Virtex-4 FPGAs und die zugehörigen Entwicklungswerkzeuge ein.
Seit 2007 werden RPUs von der Firma Cray optional für ihre $XT5_h$-Supercomputer angeboten [31].

4.4 Prozessoren mit konfigurierbaren Funktionseinheiten

Eine weitere schon lange genutzte Koppelung zwischen programmierbarer Hardware und Prozessoren, ist das Konzept, eine oder mehrere konfigurierte Funktionseinheiten zu benutzen. Dieses unterscheidet sich dadurch von dem oben genannten Ansatz, dass die rekonfigurierbare Logik Teil des Datenpfades des Prozessors ist. Sie ist somit keine beschleunigende Zusatzeinheit mehr, sondern integraler Bestandteil des Prozessors.

4.4.1 PRISC

Ein von der Harvard University und DEC im Jahr 1993 veröffentlichtes frühes Beispiel für eine Prozessorarchitektur mit konfigurierbaren Funktionseinheiten ist der PRISC (PRogrammable Instruction Set Computer) [101]. Durch Analysen des auszuführenden Programms zur Compile-Zeit werden hier programmierbare Funktionseinheiten (Programmable Funtional Units, PFU) aus einem vordefinierten Satz an FUs vor der Ausführung zum Datenpfad des Prozessors statisch hinzugefügt, um die Programmausführung zu optimieren.

4.4.2 Spyder

Der ebenfalls bereits 1993 von der École Polytechnique Fédérale de Lausanne vorgestellte Spyder (Reconfigurable Processor DEvelopment System) [57, 58] benutzt drei FPGAs, welche direkt als Funktionseinheiten in einem separaten Rechenwerk eines Host-Rechners eingebunden werden. Diese FPGAs können vom Host mit verschiedenen vorgefertigten Funktionen konfiguriert werden. Der Host ist ebenso für die Generierung des VLIW-Codes zuständig, welcher die Befehle für die konfigurierten FUs enthält. Abbildung 4.2 zeigt den Aufbau des Prozessors.

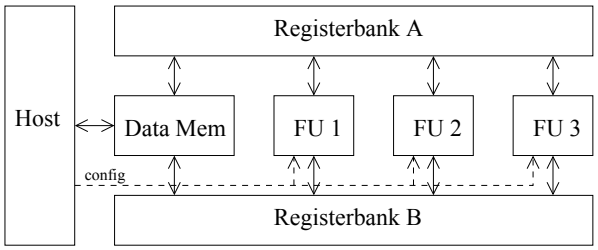


Abbildung 4.2: Stark vereinfachter Aufbau des Spyder-Prozessors

4.4.3 Chimaera

Der Chimaera-Prozessor [144] verbindet eine kleine rekonfigurierbare Funktionseinheit (Reconfigurable Functional Unit, RFU) mit einem superskalaren Prozessorkern. Die RFU ist dafür ausgelegt, die Ausgabe von Funktionen mit bis zu neun Eingaberegistern zu berechnen. Ein Performanzgewinn wird durch die enge Koppelung der programmierbaren Logikzellen und dem Prozessorkern erreicht. Das Identifizieren von Funktionen, welche auf der RFU berechnet werden sollen, geschieht bei der Chimaera automatisch während der Übersetzung des in C geschriebenen Eingabeprogramms.

4.5 Rekonfigurierbare Prozessorarchitekturen

Architekturen, welche sich programmierbarer Hardware bedienen, diese aber weder als Koprozessor noch ausschließlich als Funktionseinheit nutzen, werden hier als rekonfigurierbare Prozessorarchitekturen zusammengefasst.

4.5.1 Reconfigurable Architecture Workstation

Die vom MIT vorgestellten Raw-Maschinen (Reconfigurable Architecture Workstation) [126] bestehen aus mehreren einfachen, gleichartigen RISC-Prozessoren, die eng miteinander verbunden sind. Jeder dieser kleinen Prozessoren verfügt über eine Anzahl an rekonfigurierbaren Logikzellen, welche eine Erweiterung des Befehlssatzes ermöglichen. Das Finden von ILP wird dabei, ähnlich wie bei VLIW- und EPIC-Prozessoren, dem Compiler überlassen, allerdings werden nebenläufig ausführbare Programmstücke hier als getrennte Befehlsströme an die einzelnen RISC-Prozessoren übergeben.

4.5.2 ADRES

Die ADRES-Architektur (Architecture for Dynamically Reconfigurable Embedded Systems) [84] des Forschungszentrums IMEC aus Leuven ist ein Architektur-Template für rekonfigurierbare Prozessoren, welches auf VLIW-Prinzipien beruht. Die VLIW-Funktionseinheiten sind bei ADRES-Prozessoren direkt mit dem Befehlsregister verbunden und haben zusätzlich eine Verbindung zu einem Array aus grobgranular rekonfigurierbaren Zellen (Coarse-Grained Reconfigurable Array, CGRA). Der Aufbau eines ADRES-Prozessorkerns ist in Abbildung 4.3 dargestellt.
Der Einsatz von VLIW-Prinzipien soll den Flaschenhals des Kontrollflusses von Programmen verbreitern. Wie in vielen vergleichbaren Architekturen werden häufig ausgeführte Programmteile von der rekonfigurierbaren Hardware parallel verarbeitet. Die daraus resultierenden Operationsmodi des ADRES Prozessors werden als zwei *Sichten*, nämlich als VLIW-Prozessor Sicht und als Reconfigurable Array Sicht, bezeichnet [83].
Seit Oktober 2008 wird die ADRES-Architektur sowie die DRESC-Compiler Technologie von Toshiba lizenziert [33].

4.5.3 Dynamically Reconfigurable Processor

Der 2002 herausgegebene DRP (Dynamically Reconfigurable Processor) [90] der Firma NEC ist ein grobgranular rekonfigurierbarer Prozessor, der auch während der Programmverarbeitung seinen Datenpfad dynamisch und partiell anpassen, also Teile des Datenpfades ändern, kann. Zwischen den 16 Konfigurationen, die er gleichzeitig hält, kann innerhalb eines Taktzyklus umgeschaltet werden.
Er besteht aus einem Array von Processing Elements, welche wiederum je eine 8-bit ALU,

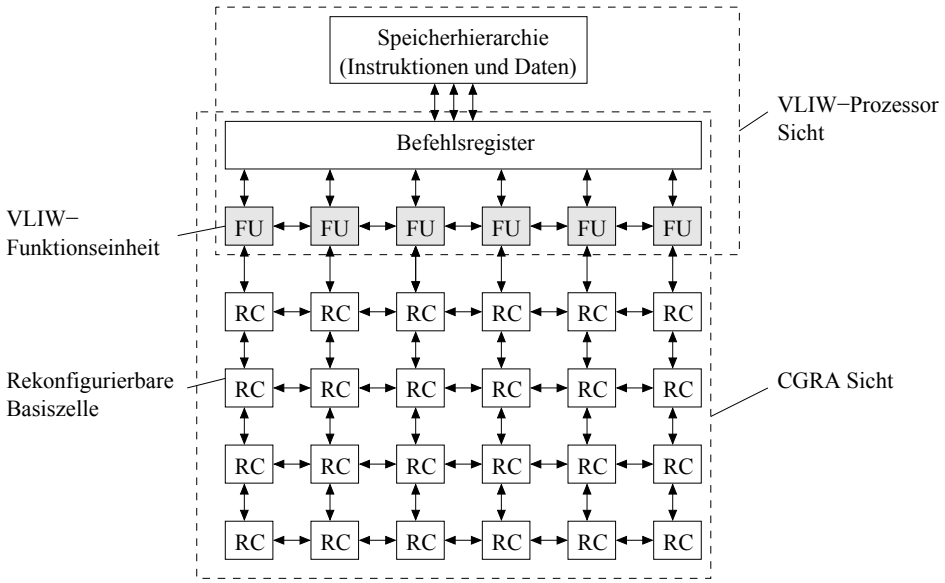


Abbildung 4.3: Aufbau eines ADRES Prozessorkerns (Abbildung angelehnt an [83])

eine DMU (Data Management Unit) für Schiebe- und Maskieroperationen, 16 Register sowie 8 FlipFlops enthalten. Zudem verfügen sie über Speicher, der die verschiedenen Konfigurationen enthält. Die Konfiguration eines Tiles aus 8x8 Processing Elements wird von einem State Transition Controller gesteuert. Zudem verfügen die Tiles über vertikalen und horizontalen Speicher. Die Anzahl solcher Tiles ist variabel. Der Prototyp des DRP, der in Abbildung 4.4 dargestellte DRP-1, enthält acht Tiles [63].

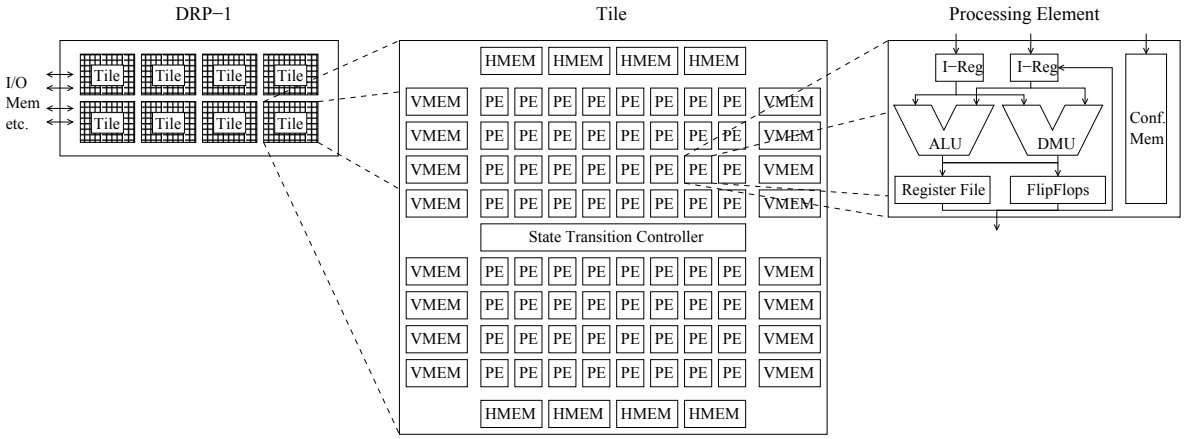


Abbildung 4.4: Aufbau eines DRP, seiner Tiles und der Processing Elements

4.5.4 Pact XPP

Der XPP von Pact [18] ist ein Prozessor, dessen grobgranulare Zellen dynamisch partiell rekonfigurierbar sind. Die konfigurierbaren Zellen, hier PAE (Processing Array Element) genannt, bestehen aus einer rekonfigurierbaren ALU und verschiedenen Registern. Diese PAE sind durch ein paketorientiertes Netzwerk verbunden.
Das Konzept des XPP basiert darauf, Datenflussgraphen direkt einzubetten. Die ALUs übernehmen die Operationen an den Knoten. Während der Abarbeitung eines Algorithmus findet keine Rekonfiguration statt. Auch wenn die Rekonfiguration durch die grobe Granularität schneller als bei anderen programmierbaren Architekturen durchgeführt werden kann, ist auch hier dieses Vorgehen nur für Programme, in denen Programmteile vielfach wiederholt werden, rentabel.
Deswegen verfügt der im Morpheus (siehe Kapitel 4.5.5) eingesetzte XPP-III [96] zusätzlich zu den Komponenten des XPP über Function-PAEs, welche dafür optimiert sind, längere sequentielle und verzweigungsreiche Programmteile zu verarbeiten. Sie führen zeitgleich bis zu acht Operationen aus, die durch ein VLIW-ähnliches Befehlswort übergeben werden. Sie unterstützen Predication (siehe Abschnitt 3.2.2.2) und Chaining, also die direkte Übernahme von Ausgaben einer Operation als Eingaben einer neuen Operation. Beim XPP-II wurden zudem RAM-PAEs eingeführt, welche über RAM und I/O-Komponenten verfügen. Die Anordnung der PAEs zueinander ist in Abbildung 4.5 dargestellt.

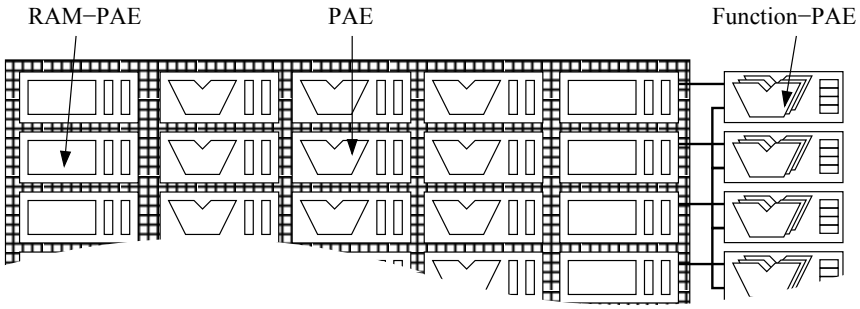


Abbildung 4.5: Grundstruktur eines Pact XPP-III

4.5.5 MORPHEUS

Das Resultat des EU-geförderten Projekts MORPHEUS (Multi-purpOse dynamically Reconfigurable Platform for intensive HEterogeneoUS processing) [119] ist eine Prozessorarchitektur, die einen Standard-Prozessor mit drei rekonfigurierbaren Verarbeitungseinheiten (Heterogeneous Reconfigurable Engines, HREs) verschiedener Granularität verbindet. Dadurch erreicht sie hohe Flexibilität und Rechenleistung.
Der in Abschnitt 4.5.4 genauer betrachtete, grobgranular rekonfigurierbare Pact XPP-III-

Kern dient zur schnellen Abarbeitung von rechenintensiven Algorithmen, die hauptsächlich deterministisch Daten verarbeiten. Der PiCoGa (Pipelined Configurable Gate Array) Core besteht aus rekonfigurierbaren 4-bit Lookup-Tabellen und 4-bit ALUs. Er zielt in erster Linie auf die Verarbeitung von automatisch kompiliertem Instruktions-Level parallelem Code ab. Die zusätzlichen eingebetteten FlexEOS FPGA-Zellen sind feingranular rekonfigurierbar. Sie dienen der freien Implementierung von beschleunigender Hardware.
Diese drei HREs sind durch ein Network-On-Chip (NoC) miteinander verbunden. Ein AMBA-Bus (Advanced Microcontroller Bus Architecture) verbindet dieses NoC mit einem ARM 9 GPP-Prozessor und externem Speicher. Ein weiterer AMBA-Bus verbindet die HREs mit Konfigurationsspeicher und einem Konfigurations-Controller, welcher die Rekonfigurationsvorgänge durchführt. Abbildung 4.6 zeigt die beschriebenen Komponenten und deren Verbindung.
Seit 2009 baut der Projektpartner STMicroelectronics erste Prototypen des MORPHEUS [24].

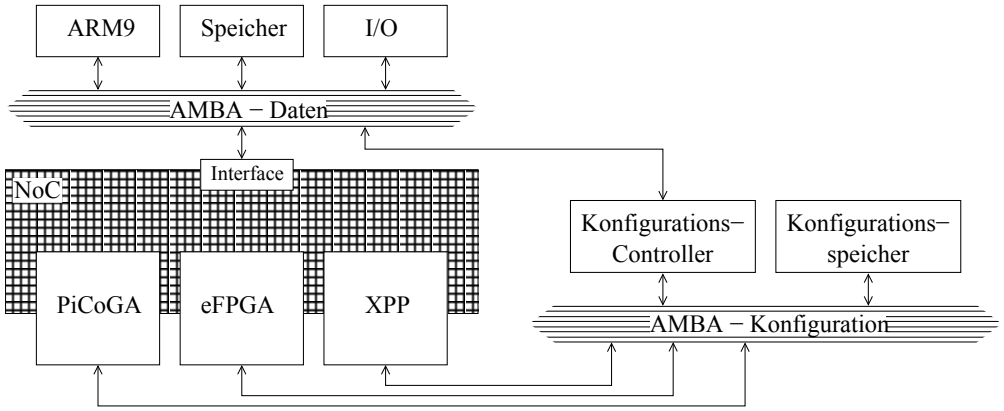


Abbildung 4.6: Verbindung zwischen den Hauptkomponenten der MORPHEUS-Architektur

4.5.6 Xputer

Das Xputer Paradigma der Universität Kaiserslautern [46] hebt sich stark von den anderen Ansätzen ab. Es beschreibt eine parallel arbeitende Rechenmaschine, deren Programmfluss datengetrieben ist. Durch ihren Einsatz von mehreren Daten-Sequenzern ist sie weder als eine von Neumann noch als eine Datenflussmaschine anzusehen. Die Funktionseinheiten von Xputern (rekonfigurierbare ALU, rALU) sind grundsätzlich rekonfigurierbar.
Die drei Hauptkomponenten eines Xputers sind ein zweidimensionaler Datenspeicher, ein Datensequenzer, der mehrere Adressgeneratoren (Generic Address Generator, GAG) enthält und eine aus mehreren Subnetzen bestehende rALU mit Scan Windows. Ein Scan Window ist dabei ein Registersatz, welcher zu jedem Zeitpunkt einen zweidimensionalen Ausschnitt

des Datenspeichers enthält. Abbildung 4.7 zeigt den Aufbau eines Xputers.

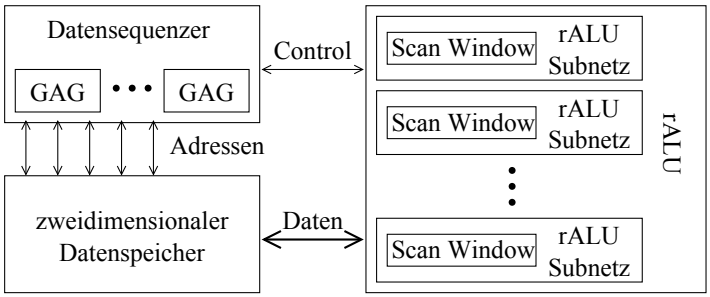


Abbildung 4.7: Hauptkomponenten eines Xputers

Die GAGs generieren eine Folge von Adressen. Die zugehörigen Daten werden in die Scan Windows der rALU geladen. Bewegt sich das Scan Window eines rALU-Subnetzes weiter, wird dessen zusammengesetzte Operation ausgeführt. Solch eine Operation kann aus mehreren Befehlen, Eingaben und Ausgaben bestehen und ist rekonfigurierbar.
Es existieren mehrere Prototypen des Xputers, welche alle Operatoren der Programmiersprache C unterstützen. Aus dem Ansatz einer schneller rekonfigurierbaren ALU [67] entstanden im Xputer-Umfeld zudem die Kress-Arrays, eine Familie grobgranularer rekonfigurierbarer Hardware-Bausteine.

4.5.7 Extensible Processing Platform

Mitte 2010 kündigte die Firma Xilinx die Extensible Processing Platform [26] an. Diese besteht aus einem ARM Dual-Cortex A9MPCore, der auf einem Chip sowohl mit festverdrahteter Peripherie und einem Speicher-Interface als auch mit programmierbarer Logik verbunden ist. Der 32-Bit Dual-Core Prozessor kann mit bis zu 800 MHz getaktet werden. Der grobe Aufbau des in 28-nm-Technologie gefertigten Bausteins ist in Abbildung 4.8 dargestellt.
Die Extensible Processing Platform unterscheidet sich nicht entscheidend von FPGAs mit Hard Core Prozessor. Durch ihren prozessororientierten Aufbau und den damit Software-basierten Entwicklungsprozess, kann sie jedoch als rekonfigurierbare Prozessorarchitektur angesehen werden.

4.6 Verbindung zwischen der EPIC-Architektur und programmierbarer Logik

Einige Ansätze beschäftigen sich, wie diese Arbeit, mit EPIC-ähnlichen Prozessoren, die davon profitieren, dass sie in programmierbarer Logik implementiert sind. Sie ziehen dabei aus der variablen Form der parallelen Befehlsausführung der EPIC-Architektur Nutzen.

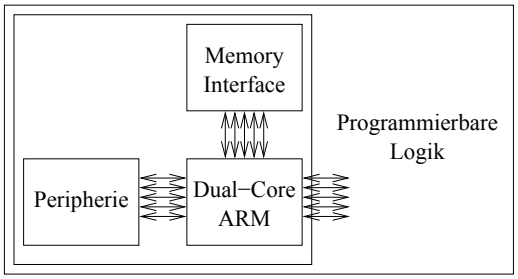


Abbildung 4.8: Stark vereinfachter Aufbau der Xilinx Extensible Processing Platform

4.6.1 Adaptive EPIC

Surendranath Talla stellt in seiner Dissertation [117] aus dem Jahre 2000 die Adaptiven EPIC-Prozessoren (AEPIC), eine Klasse von Prozessoren mit rekonfigurierbarer ISA, vor. Diese Prozessoren bestehen aus einer Anzahl von festverdrahteten Funktionseinheiten und aus adaptiven Funktionseinheiten (Configurable Functional Units, CFUs), die aufwändige Programmteile durch parallele Implementierung in Hardware effizient berechnen.
Diese adaptiven Einheiten werden zur Laufzeit dynamisch konfiguriert, wobei der Zeitpunkt ihrer Konfiguration statisch vom Compiler festgelegt wird. Zusätzliche Befehle im EPIC-Befehlssatz der Prozessorklasse lösen die Rekonfiguration der Einheiten aus, versorgen sie mit Daten und ermöglichen die Eingabe von Operanden und Ausgaben der Ergebnisse. Die adaptiven Funktionseinheiten greifen nicht direkt auf das Befehlsbündel zu, sondern werden von den festverdrahteten FUs mit Befehlen und Daten beliefert. Der vereinfachte Aufbau eine AEPIC-Prozessors ist in Abbildung 4.9 dargestellt.

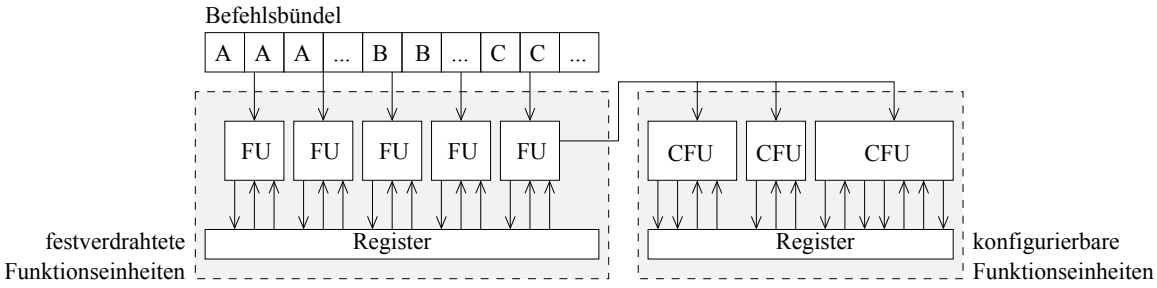


Abbildung 4.9: Verbindungen der Funktionseinheiten eines Adaptive EPIC-Prozessors

Um extrem schnelle Rekonfiguration zwischen verschiedenen adaptiven Funktionseinheiten zu ermöglichen, bauen Adaptive EPIC-Prozessoren auf MRLAs (Multicontext Reconfigurable Logic Arrays). Diese Form der programmierbaren Hardware ermöglicht, im Gegensatz zu gewöhnlichen FPGAs, durch zusätzliche Register das direkte Umschalten zwischen Konfigura-

tionen. Somit ist es möglich, zeitgleich mehrere Funktionalitäten für die Hardware vorzuhalten und zwischen diesen mit nur minimalem Zeitverzug umzuschalten.

4.6.2 SNAPP

Der SNAPP-Prozessor (Shared Networked Acceleration Parallel Processor) [19] der SNAPP Technologies Corporation ist ein Soft Core Prozessor, dem das Konzept der EPIC-Architektur zugrunde liegt. Er ist die Kommerzialisierung der REBUS-Architektur (Regulated Elements By Universal Scheme) und als Soft Core Prozessor erhältlich.
Er verbindet mehrere gleichartige Processing Elements, einfachen Kleinprozessoren, die synchron oder asynchron je einen Befehlsstrom abarbeiten. Die einzelnen Befehlsströme sind durch hinzugefügte Kontrollanweisungen verbunden. Die Anzahl der Processing Elements ist bei der Erstellung des Prozessors konfigurierbar, Veränderungen des Prozessors zur Laufzeit sind nicht vorgesehen.
Laut dem Hersteller existiert ein einfaches Verfahren, welches bereits kompilierte Maschinenprogramme in Programmblöcke zerteilt, welche dann den einzelnen Processing Elements zugeordnet werden.

4.6.3 Customisable EPIC

Eine mit dem SNAPP verwandte Idee ist der Ansatz des Customisable EPIC-Prozessors [23] vom Imperial College in London. Dieser arbeitet jedoch mit verschiedenartigen Funktionseinheiten. Zur Compile-Zeit der Software wird bei diesem Ansatz entschieden, welche Funktionseinheiten die Aufgabe so effizient wie möglich lösen können. Mit dieser Information wird dann ein möglichst optimaler EPIC-Prozessor erzeugt. Da dieser Ansatz keine Aussagen über dynamisches Verhalten trifft, ist das Konzept nicht auf rekonfigurierbare FPGAs beschränkt, sondern auch für ASIC-Entwicklungen geeignet.

4.7 Zusammenfassung und Bewertung

Es existiert eine Vielzahl adaptierbarer Prozessoren. Um die Vorteile rekonfigurierbarer Logik auszuschöpfen, verwenden die Prozessoren diese meist als Koprozessoren oder Funktionseinheiten, welche ihre Fähigkeiten der aktuellen Aufgabe anpassen können. Einige Ansätze nutzen zudem die Fähigkeit zur Rekonfiguration als integralen Bestandteil ihrer Prozessorarchitektur. Manche dieser Prozessoren, wie die Reconfigurable Processor Units oder die ADRES-Architektur, haben dabei bereits den Weg aus dem rein akademischen Bereich in den industriellen Einsatz gefunden.
Adaptierbare Prozessoren, welche die Vorteile von EPIC-Architekturen mit denen rekonfigurierbarer Logik verbinden, sind demgegenüber noch rar. Sowohl der SNAPP-Prozessor als auch der Customisable EPIC-Prozessor ziehen Nutzen aus der flexiblen Befehlsgruppierung

von EPIC-Architekturen. Die sich durch partielle dynamische Rekonfiguration ergebenden Möglichkeiten werden jedoch in beiden Ansätzen nicht berücksichtigt.
Lediglich die Adaptive EPIC-Prozessorarchitektur verbindet die meisten Eigenschaften von EPIC-Prozessoren mit dynamischer Rekonfiguration. Die Generierung ihrer anwendungsspezifischen konfigurierbaren Funktionseinheiten ist jedoch nur sehr schwer zu automatisieren. Dies macht komplexe und aufwändige Compiler nötig. Sowohl die Probleme mit der Erstellung effizienter EPIC-Compiler als auch mit der schnellen und einfachen Entwicklung von spezialisierter Hardware sind aktuell noch nicht zufriedenstellend gelöst. Dies legt die Annahme nahe, dass ein möglicher Erfolg für Adaptive EPIC-Prozessoren noch in ferner Zukunft liegt.
Die bisher vorhandenen Ansätze so zu erweitern, dass sie dynamisch freie Ressourcen des Systems für die Beschleunigung ihrer Berechnungen einsetzen, ist wenig erfolgversprechend. Entweder ist ihr Architekturkonzept nicht auf Rekonfiguration zur Laufzeit anpassbar oder ihre Programme sind auf komplexe Spezial-Hardware angewiesen, um Berechnungen überhaupt durchführen zu können.
Ein Prozessor, welcher die Programmausführung, abhängig von den verfügbaren Hardware-Ressourcen, dynamisch beschleunigen kann und dabei ohne die komplexe Entwicklung für programmabhängige, dedizierte Spezial-Hardware auskommt, ist für viele Einsatzgebiete wünschenswert. Gerade in den immer weiter verbreiteten dynamischen Systems-on-a-Chip, kann ein solcher Prozessor immense Vorteile bieten.

5 Laufzeitrekonfigurierbare EPIC Soft Core Prozessoren

Aufbauend auf den Konzepten der EPIC-Architektur wird in dieser Arbeit eine Klasse von Prozessoren definiert, welche aus den Eigenschaften von rekonfigurierbaren Logikbausteinen Nutzen ziehen. Neuartig an diesen *Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren* ist, dass sie ermöglichen, ihre Berechnungsgeschwindigkeit durch dynamische Hinzunahme zusätzlicher Hardware-Ressourcen zu erhöhen, ohne dass dabei die Software angepasst werden muss. Stehen die zusätzlichen Ressourcen nicht mehr zur Verfügung, können die Berechnungen des Prozessors mit reduzierter Geschwindigkeit fortgesetzt werden.
Um diese Dynamik zu ermöglichen, dienen in erster Linie die flexibel gruppierten EPIC-Befehlsbündel. Ihre Einzelbefehle können entweder von wenigen Funktionseinheiten größtenteils sequentiell oder mit Hilfe zusätzlicher Funktionseinheiten komplett parallel verarbeitet werden.
Die hier vorgestellten Konzepte wurden zum Teil in frühem Stadium bereits in [110] veröffentlicht.

5.1 Einsatzgebiet

Die vorgestellte Klasse von Prozessoren basiert auf zur Laufzeit rekonfigurierbarer Logik. Der zusätzliche Aufwand, um Hardware rekonfigurierbar zu gestalten, wird immer auf Kosten der Größe des Fabrics, also der Chip-Fläche, und damit der Geschwindigkeit der implementierten Logik gehen. Es ist nicht das Ziel, bezogen auf Performanz mit herkömmlichen Höchstleistungsprozessoren zu konkurrieren. Vielmehr bietet der Ansatz dieser Arbeit eine performante Alternative zu herkömmlichen eingebetteten Soft Core Prozessoren.
Der Einsatzbereich, in welchem die vorgestellte Architektur reelle Vorteile gegenüber herkömmlichen Prozessoren bietet, liegt in Systemen, welche bereits auf FPGAs basieren. Am deutlichsten werden die Vorteile, wenn sich auch das restliche System dynamisch ändert und somit zu verschiedenen Zeiten verschiedenen Anspruch an Hardware-Ressourcen auf dem FPGA hat. Laufzeitrekonfigurierbare EPIC Soft Core Prozessoren können im Gegensatz zu anderen Prozessoren diesen Platz nutzen, um ihre Berechnungen zu beschleunigen.

5.1.1 System-on-a-Chip

Die Integration von Prozessoren und anderer Hardware auf einem FPGA-Baustein lässt sich in der Regel als System-on-a-Chip (SoC), also der Zusammenführung der meisten Komponenten eines eingebetteten Systems auf einem Baustein, ansehen. Diese Komponenten können verschiedenste Funktion haben. Üblich ist es, Prozessoren und Speicher sowie Komponenten zur Anbindung von Peripheriegeräten oder solche zur Beschleunigung spezieller Berechnungen zu integrieren.

5.1.2 Beispielsystem

Der mögliche Nutzen von Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren wird beim Einsatz in Smartphones sehr deutlich. Hier kann ein System-on-a-Chip für mannigfaltige Aufgaben vorbereitet sein, von den üblichen Aufgaben eines Mobiltelefons, wie der Sprach- und Datenübertragung über GSM oder UMTS, bis hin zu GPS Navigation oder Dekodierung und Wiedergabe von Musik und Videos.
Abbildung 5.1 a) zeigt den klassischen Aufbau eines solchen Systems. Die einzelnen Hardware-Komponenten sind hier auf separaten Bausteinen realisiert, welche durch Busse oder dedizierte Leitungen miteinander verbunden sind. In Abbildung 5.1 b) ist demgegenüber die Realisation derselben Funktionalität auf einem System-on-a-Chip gezeigt. Die Komponenten des Systems sind hier Intellectual Property-Blöcke (IP-Cores), also wiederbenutzbare Teile eines Hardware-Designs, auf nur einem Baustein.
Falls in dem Beispielsystem nicht alle Funktionalitäten zu jeder Zeit gleichzeitig benutzbar sein müssen, ist es denkbar, die nötige Hardware während des Betriebs an die Anforderungen dynamisch anzupassen. Benötigte Module werden auf Kosten anderer nachkonfiguriert, wie in Abbildung 5.1 c) dargestellt.
Somit können während des Betriebs des Systems Hardware-Ressourcen zeitweise unbenutzt sein. Diese Ressourcen in sinnvolle Rechenleistung umzusetzen ist Ziel der EPIC Soft Core Prozessoren.

5.2 Konzept

In Systemen, in denen zu unterschiedlichen Zeiten eine unterschiedliche Menge an Hardware-Ressourcen der programmierbaren Logik benutzt wird, können Laufzeitrekonfigurierbare EPIC Prozessoren die freien Ressourcen verwenden, um ihre Rechenleistung zu erhöhen. Zu Zeiten, in denen nur wenige Ressourcen zur Verfügung stehen, können sie trotzdem ohne Änderung der Software eine grundlegende Rechenleistung sicherstellen.
Dies geschieht durch Änderung der Anzahl und Art der Funktionseinheiten des Prozessors zur Laufzeit. Durch die Erhöhung und Verringerung deren Anzahl steigt bzw. sinkt der Anspruch des Prozessors an Hardware-Ressourcen, also die vom Prozessor eingenommene Fläche auf

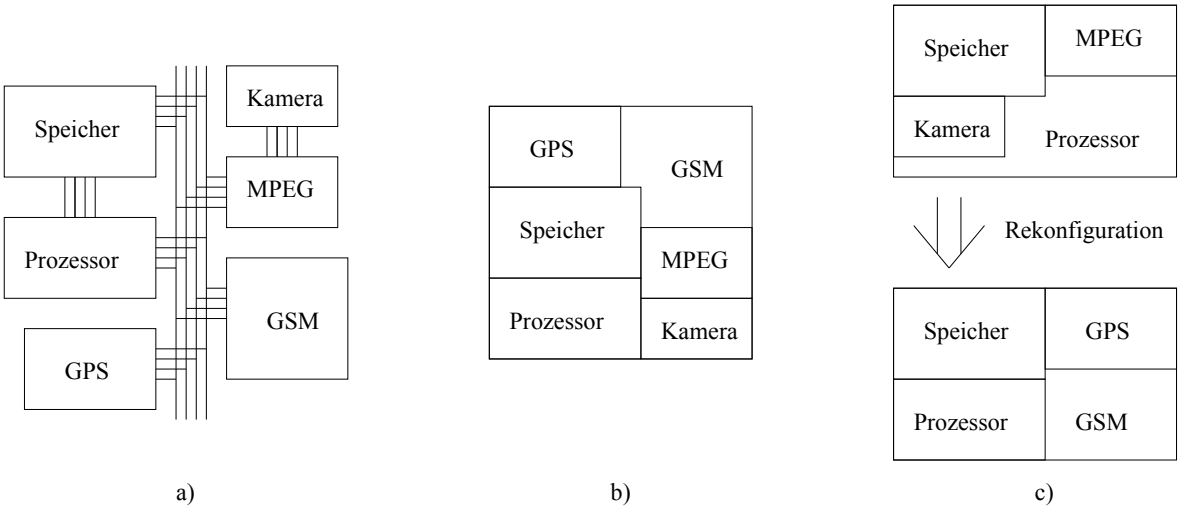


Abbildung 5.1: Realisierung eines exemplarischen Smartphones als a) klassisches System aus mehreren Komponenten, b) System-on-a-Chip und c) rekonfigurierendes SoC mit adaptivem Prozessor

dem programmierbaren Logikbaustein. Durch die Anpassung der Art der Funktionseinheiten kann eine Optimierung des Prozessors auf das zu lösende Problem erreicht werden.
Die zusätzlichen Funktionseinheiten werden benutzt, um die aus EPIC-Befehlsbündeln bestehende Software schneller abzuarbeiten. Eine reelle Beschleunigung der Abarbeitung wird hierbei dadurch möglich, dass sämtliche in einem solchen Bündel zusammengefassten Befehle nebenläufig abgearbeitet werden können. Es sinkt also die Verarbeitungszeit der einzelnen Bündel. Möglich wird diese Abarbeitung einer variablen Anzahl von Befehlen ohne Änderung der Software durch die flexible Gruppierung nebenläufig ausführbarer Operationen im EPIC-Befehlswort.
Ein solches auch als Befehlsbündel bezeichnetes Befehlswort besteht aus mehreren Befehlen unterschiedlicher Befehlsklassen. Jeder Befehl eines Bündels muss von einer zu seiner Befehlsklasse passenden Funktionseinheit des Prozessors verarbeitet werden. Zwischen den einzelnen Befehlen eines Befehlsbündels besteht kein kausaler Zusammenhang. Sie können also in beliebiger sequentieller Folge oder zeitgleich abgearbeitet werden. Ebenso ist es möglich, beliebige Kombinationen der Befehle eines Bündels zeitgleich abzuarbeiten, ohne die Korrektheit des Programms damit zu beeinflussen.
Der Hardware-Aufwand für dieses Vorgehen zur Skalierung des Prozessors ist im Vergleich zu superskalaren Prozessoren relativ gering, da es Aufgabe des Compilers ist, die Nebenläufigkeit der Befehle zu bestimmen und sie zu Befehlsbündeln zu gruppieren.
Für die zusätzliche Flexibilität mit variabler Anzahl an Funktionseinheiten ist es lediglich nötig, die Befehle eines Wortes auf die vorhandenen FUs zu verteilen. Hierbei ist ein Gree-

dy Algorithmus, welcher unbearbeitete Befehle zu beliebigen freien Funktionseinheiten vom passenden Typ zuteilt, ausreichend (s. Kapitel 5.2.3). Ist ein Befehl einer FU zugeordnet, so muss er als abgearbeitet markiert werden. Dies vermeidet mehrfache Verarbeitung, welche zu fehlerhaften Berechnungen und ineffizienter Ressourcen-Nutzung führen würde.

5.2.1 Aufbau

Der Aufbau eines Laufzeitrekonfigurierbaren EPIC Soft Core Prozessors ähnelt dem eines gewöhnlichen EPIC-Cores. Seine Funktionseinheiten sind jedoch in partiell rekonfigurierbarer Logik realisiert. Um die Variabilität des Prozessors zu gewährleisten, ist zudem eine Einheit für die zentrale Verwaltung der Ressourcen nötig. Diese speichert den Abarbeitungsstatus eines Befehlswortes im *Done-Flag-Register*.
Der schematische Aufbau eines solchen Prozessors ist in Abbildung 5.2 dargestellt.

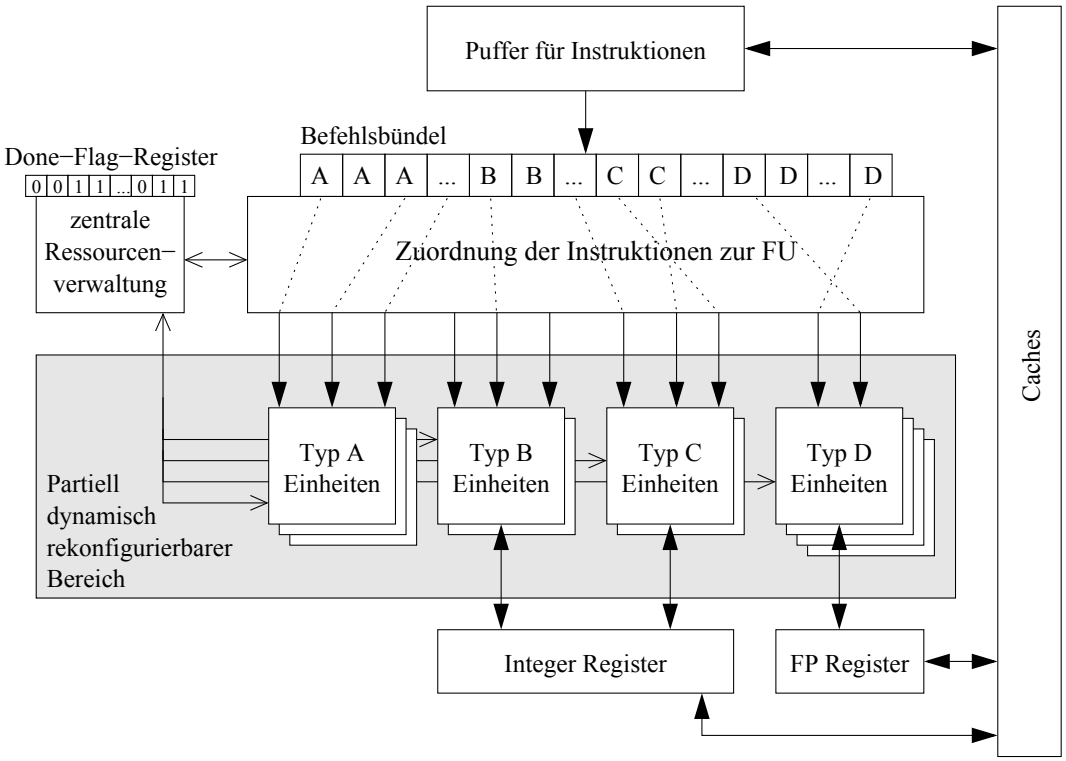


Abbildung 5.2: Abstrahierter Aufbau eines Laufzeitrekonfigurierbaren EPIC Soft Core Prozessors

5.2.1.1 Funktionseinheiten

Die Funktionseinheiten des Laufzeitrekonfigurierbaren EPIC Soft Core Prozessors können durch ihre Implementierung in rekonfigurierbarer Logik hinzugenommen, entfernt und ausge-

tauscht werden. Alle konfigurierten Funktionseinheiten geben ihren Status, welcher ihren Typ enthält, an eine Komponente zur zentralen Verwaltung der Ressourcen des Prozessors weiter. Über diese Status-Information melden sie sich bei der Ressourcen-Verwaltungseinheit als *konfiguriert*. Zudem wird im Status übertragen, ob die jeweilige Einheit gerade einen Befehl bearbeitet oder ob sie auf die Zuteilung eines neuen Befehls wartet.

5.2.1.2 Zentrale Ressourcen-Verwaltung

Abhängig von diesen Informationen ordnet die Ressourcen-Verwaltung Befehle aus dem Befehlsbündel zu den Funktionseinheiten zu. Sobald eine Funktionseinheit meldet, dass sie den ihr zugeteilten Befehl abgearbeitet hat, wird dies im Done-Flag-Register gespeichert. Darauf folgend teilt die Ressourcen-Verwaltung der Einheit den nächsten zu ihrem Typ passenden Befehl des Bündels zu. Sind alle Befehle eines Bündels verarbeitet, initialisiert die Einheit das Holen des nächsten Befehlsbündels.

5.2.1.3 Done-Flag-Register

Das Done-Flag-Register enthält den Abarbeitungsstatus des aktuellen Befehlsbündels. Für jeden Befehl des Bündels wird in das Register eine Eins an die Stelle des jeweiligen Befehls im Bündel gesetzt. Ist ein Befehl abgearbeitet, wird die zugehörige Eins gelöscht. Im Folgenden wird davon ausgegangen, dass sowohl die Länge des Done-Flag-Registers als auch die Länge des Registers, welches das aktuelle Befehlsbündel hält, groß genug ist, um das längste Bündel des auszuführenden Programms zu halten und zu verwalten.

5.2.2 Befehlszyklus

Der Befehlszyklus eines Laufzeitrekonfigurierbaren EPIC Soft Core Prozessor beginnt durch das Holen eines Befehlsbündels aus dem Speicher. Dieses wird dekodiert und die Klassen der Befehle, aus denen das Bündel aufgebaut ist, identifiziert.
Das Done-Flag-Register, welches den Verarbeitungszustand der Befehle des Bündels enthält, hat vor dem Zyklus den Startzustand, in welchem alle Flags auf *verarbeitet* stehen. Durch Umsetzen der Anzahl an Flags, welche der Anzahl an Befehlen im Befehlsbündel entspricht, wird es initialisiert.
Die zentrale Einheit zur Ressourcenverwaltung teilt dann so viele Befehle wie möglich an die vorhandenen Funktionseinheiten der entsprechenden Typen zu. Sie speichert im oben erwähnten Done-Flag-Register, welche der Befehle des Bündels bereits abgearbeitet wurden.
Die einzelnen Funktionseinheiten melden hierzu ihren Status an die Ressourcenverwaltung. Dieser Status gibt über Vorhandensein der Einheit und Verarbeitungsstatus des zugeteilten Befehls Auskunft.
Sobald eine Einheit ihren Befehl fertig abgearbeitet hat, wird ihr von der zentralen Verwaltungseinheit ein als noch nicht abgearbeitet markierter Befehl aus dem aktuellen Bündel

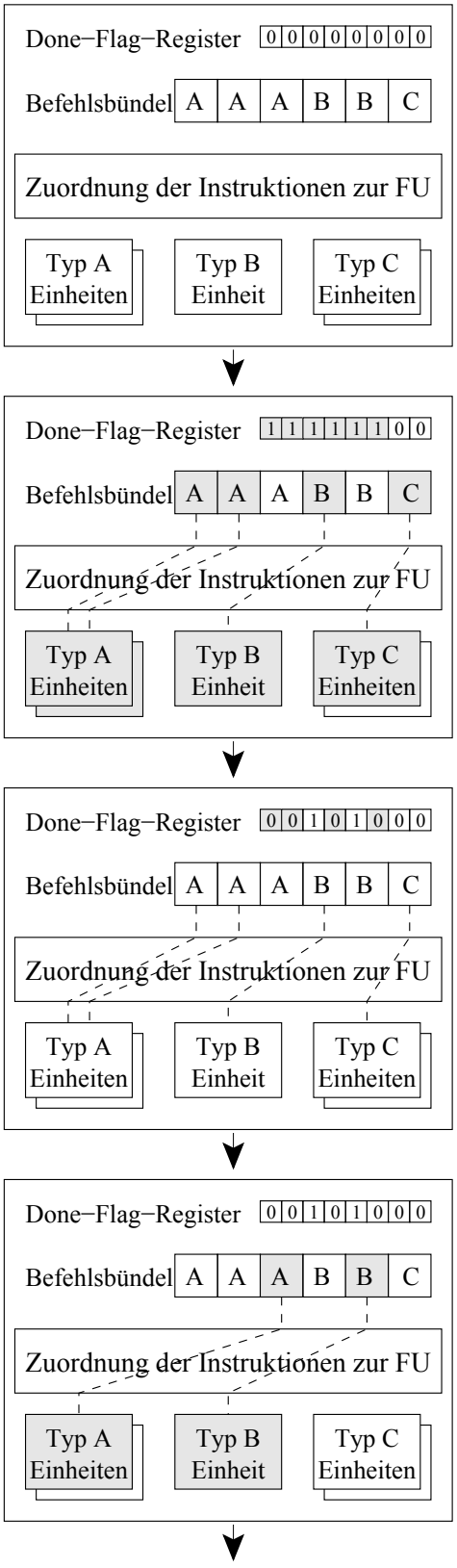
zugeordnet. Wurde ein Bündel komplett verarbeitet, so wird das nächste Bündel geladen und der Befehlszyklus beginnt von neuem.

5.2.3 Zuordnung der Befehle zur verarbeitenden Funktionseinheit

Laufzeitrekonfigurierbare EPIC Soft Core Prozessoren basieren darauf, dass ein aus Einzelbefehlen aufgebautes Befehlsbündel in einem Zyklus oder in mehreren Zyklen abgearbeitet wird. Stehen genügend Funktionseinheiten des richtigen Typs zur Verfügung, kann ein Bündel in einem Zyklus abgearbeitet werden. Sind nicht genügend passende FUs konfiguriert, so muss das Befehlswort über mehrere Zyklen hinweg ausgeführt werden.
Hierzu ist es nötig, die Befehle an die Funktionseinheiten zu verteilen. Die FUs eines Typs sind einheitlich aufgebaut und müssen sowohl Zugriff zum Befehlscache als auch zu den Registern haben. Bei einem Befehlssatz mit einheitlicher Abarbeitungszeit der Befehle genügt es, wenn ein Befehl nach dem anderen an eine beliebige Einheit des passenden Typs vergeben wird.
Haben die Befehle unterschiedliche Laufzeiten, so wird eine minimale Abarbeitungsdauer für einen Befehlstyp dadurch erzielt, dass zuerst die Befehle mit den längsten Laufzeiten an die passenden Verarbeitungseinheiten verteilt werden. Meldet eine Einheit, dass sie die Abarbeitung ihres aktuell zugeteilten Befehls abgeschlossen hat, wird ihr aus der Menge der verbleibenden passenden Befehle wiederum der mit der längsten Laufzeit zugeordnet. Dieses Vorgehen stellt eine insgesamt optimale, da möglichst kurze, Ausführung des kompletten Bündels sicher.
Wichtig bei der Zuordnung der Befehle ist es, zu markieren, welcher der Einzelbefehle bereits einer FU zugewiesen und von dieser abgearbeitet wurde, um später nur die fehlenden zu verarbeiten. Eine doppelte Verarbeitung würde häufig zu fehlerhaften Ergebnisse der Gesamtberechnung führen. Für diese Markierung ist ein Flag je Befehl des Bündels ausreichend. Diese Done-Flags werden als spezielles internes Register im Prozessor zusammengefasst.

5.2.4 Beispiel zur Arbeitsweise

Ein typisches Beispiel für die Arbeitsweise eines Laufzeitrekonfigurierbaren EPIC Soft Core Prozessor wird im Folgenden gegeben. Hier wird exemplarisch ein Befehlsbündel abgearbeitet, eine Rekonfiguration findet nicht statt. Die einzelnen Phasen der Abarbeitung sind nicht als Taktzyklen anzusehen und können sehr unterschiedlichen Zeitverbrauch haben.
Hier wird ein Prozessor angenommen, welcher aus drei verschiedenen Typen von Funktionseinheiten aufgebaut ist. Er besteht aus je zwei Einheiten des Typs A und C und einer Einheit vom Typ B. Zudem ist sein Done-Flag-Register und das aktuell geladene Befehlsbündel angegeben. Eine Zuordnungseinheit weist die Einzelbefehle den Funktionseinheiten zu. Aktive Komponenten sind grau hervorgehoben.

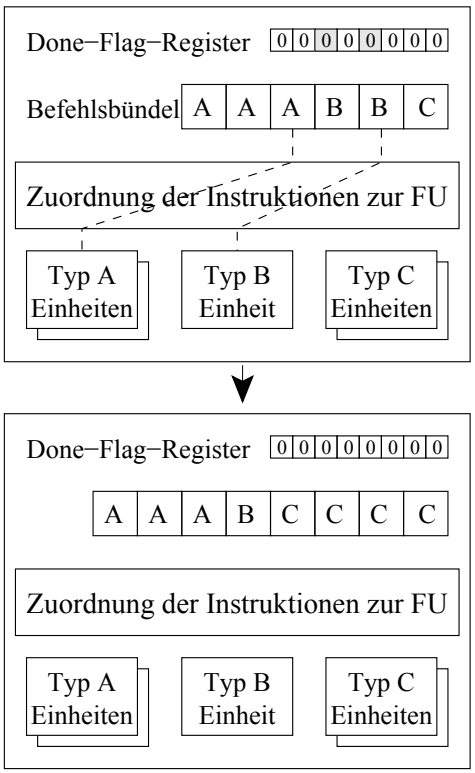


Zu Beginn der Befehlsverarbeitung wurde bereits ein Befehlsbündel geladen. Da das vorherige Bündel komplett verarbeitet wurde, stehen alle Flags des Done-Flag-Registers auf Null.

Im nächsten Schritt wird für jeden Befehl des Bündels eine Eins in diesem Register gesetzt. So viele Befehle wie möglich werden auf passende Funktionseinheiten verteilt.

Sobald die Funktionseinheiten rückmelden, dass sie ihren Befehl fertig abgearbeitet haben, werden die zu den Befehlen zugehörigen Flags zurückgesetzt.

Wie bereits zuvor werden darauffolgend wieder so viele Befehle wie möglich an die Funktionseinheiten verteilt.



Nach der Abarbeitung der verteilten Befehle werden wiederum die zugeordneten Flags des Done-Flag-Registers umgesetzt.

Sobald alle Flags des Registers wieder auf Null stehen und somit alle Befehle des Bündels verarbeitet wurden, wird das nächste Befehlswort geholt, um dann analog zum vorhergehenden abgearbeitet zu werden.

5.2.5 Alleinstellungsmerkmale

Neuartig an diesem Ansatz ist, dass der arbeitende Prozessor seine Rechenleistung zur Laufzeit anhängig von den verfügbaren Ressourcen skalieren kann. Durch Anwendung von EPIC-Befehlsbündeln ist zur Verarbeitung der Software mit variierender Anzahl von Funktionen keine erneute Kompilation der Software nötig. Die Technik der Partiellen Dynamischen Rekonfiguration (vgl. Kapitel 2.4) ermöglicht mit dem beschriebenen Ansatz die Anpassung des Ressourcenbedarfs des Prozessors ohne jegliches Anhalten der Programmausführung.

5.3 Einschränkungen des Ansatzes

Sowohl die Hardware als auch die Compiler schränken das maximale Potential Laufzeitrekonfigurierbarer EPIC Soft Core Prozessoren ein. Wie in den folgenden Unterkapiteln gezeigt wird, sind diese Einschränkungen jedoch relativ gering und sprechen nicht gegen den Einsatz der Prozessoren.

5.3.1 Beschränkung der möglichen Beschleunigung

Eine Grenze für die maximale Anzahl an erreichbarer Beschleunigung ist durch die mögliche Länge der Befehlsbündel gegeben, da vor der Abarbeitung eines neuen Bündels das vorherige komplett verarbeitet sein muss. Die Länge der Bündel wird vom Compiler festgelegt. Je stär-

ker dieser das Eingabeprogramm parallelisiert, desto höher ist die mögliche Beschleunigung für Laufzeitrekonfigurierbare EPIC Soft Core Prozessoren.
Eine weitere Größe, welche die Beschleunigung begrenzt, ist die Größe der zugrunde liegenden programmierbaren Logik und der daraus resultierenden maximalen Anzahl an konfigurierbaren Funktionseinheiten. Aktuelle FPGAs bieten ausreichend Platz für eine hohe Zahl an Funktionseinheiten, welche in erster Linie durch Bit-Breite und Funktionsumfang der FU festgelegt wird.
Eine weitere Begrenzung stellt die Anforderung an die Speicherbandbreite dar. Diese muss hoch genug sein, um eine möglichst hohe Auslastung der Funktionseinheiten sicherzustellen. Die Routing-Ressourcen von FPGAs bieten auch hier meist genug Potential, um die konfigurierbaren Einheiten adäquat an Speicher anzubinden.

5.3.2 Kompilierung für Laufzeitrekonfigurierbare EPIC Soft Cores

Auch wenn es grundsätzlich nicht nötig ist, die Anzahl der Funktionseinheiten eines EPIC-Prozessors zu kennen, um EPIC-Maschinencode zu erstellen, so ist es von Vorteil, dem Compiler die maximale Anzahl an Funktionseinheiten vorzugeben.
So führt beispielsweise, bei gleicher Anzahl an Bündeln und Befehlen, eine zu starke Variation in der Länge der Bündel generell zu schlechteren Ergebnissen als einheitlich lange Bündel. Dies liegt darin begründet, dass bei der Abarbeitung sehr kurzer Bündel viele Funktionseinheiten keinen Befehl ausführen, während sie bei sehr langen Befehlsbündeln mehrere Zyklen arbeiten.
Zudem fügt der Compiler bei höherer Anzahl an Funktionseinheiten zusätzliche Befehle für die spekulative Befehlsausführung ein. Weitere Verfahren zur Erhöhung der Parallelität, wie z.B. Software Pipelining, tragen ebenfalls dazu bei, die Länge der Programme zu erhöhen. Sie senken allerdings die Rechenzeit deutlich.
Tabelle 5.2 zeigt die Anzahl der ausgeführten Einzelbefehle der drei Benchmarks MM_INT, einer Matrizenmultiplikation mit Integer-Werten, CJPEG, der Codierung eines JPEG-Bildes, und MPEG2DEC, der Dekodierung eines MPEG2 Video. Der Compiler der Benchmarks wurde auf die angegebene Anzahl an Slots limitiert. Dabei wurden die Slots gleichmäßig über die vier Einheitentypen Branch (B), Float (F), Integer (I) und Memory (M) verteilt.
Wie in der Tabelle zu erkennen ist, müssen bei sehr hoher Slot-Anzahl bis zu etwa zehn Prozent mehr Befehle vom Prozessor ausgeführt werden. Diese sind die oben erwähnten zusätzliche Befehle zur Erhöhung der Parallelität. Im Worst Case, also der Kompilation für eine sehr hohe Anzahl an FUs und der Ausführung auf einem Prozessor mit minimaler FU-Zahl, kann man entsprechend davon ausgehen, dass die fehlende Zusatzinformation für den Compiler über die Anzahl an Funktionseinheiten zu zehn Prozent längerer Ausführungszeit führt.
Um eine hohe mögliche Beschleunigung für Laufzeitrekonfigurierbare Soft Core Prozessoren zu erreichen, wurde in Abschnitt 5.3.1 vorgeschlagen, das Programm so stark wie möglich zu parallelisieren. Die eben beschriebenen gewonnenen Erkenntnisse aus Tabelle 5.2 schränken

Tabelle 5.2: Anzahl der Einzelbefehle in einem EPIC-Programm in Abhängigkeit der dem Compiler vorgegebenen Prozessorkonfiguration nach [91]

Anzahl	Konfiguration	Benchmark		
Slots	(B, F, I, M)	MM_INT	CJPEG	MPEG2DEC
4	(1, 1, 1, 1)	88338	14921008	136581012
8	(2, 2, 2, 2)	88438	14921706	136610523
12	(3, 3, 3, 3)	91738	14941483	136616134
16	(4, 4, 4, 4)	91738	14941482	136616179
20	(5, 5, 5, 5)	101338	14941482	136616214
32	(8, 8, 8, 8)	101338	14941482	136616217
64	(16, 16, 16, 16)	101338	14941482	136616219
96	(24, 24, 24, 24)	101338	14941482	136616219

dies insofern ein, dass dies nur für Systeme sinnvoll ist, in welchen der Prozessor nur selten minimal konfiguriert ist.

5.4 Klassen von Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren

Wegen den verschiedenen Voraussetzungen des zugrundeliegenden programmierbaren Logikbausteins ist es sinnvoll, den Laufzeitrekonfigurierbaren EPIC Soft Core Prozessor an die Parameter des Bausteins anzupassen. Die Parameter, welche den Prozessor festlegen, sind:

- Die Dauer eines Rekonfigurationsschrittes beim Ersetzen einer Funktionseinheit,
- die Multi-Context-Fähigkeit des Bausteins, also die Möglichkeit, zwischen zuvor geladenen Konfigurationen umzuschalten,
- die durch die Bausteingröße beschränkte maximale Anzahl an Funktionseinheiten,
- der Ressourcenbedarf der einzelnen Funktionseinheit,
- die Anzahl an Instruktionsklassen, also an verschiedenen Typen von Befehlen und entsprechend verarbeitenden Funktionseinheiten und
- der zu unterstützende Befehlssatz mit dessen Granularität und der Ausführungsdauer der Einzelbefehle.

Diese Parameter sind entscheidend für den problemspezifisch bestmöglich geeigneten Prozessor und seinen Aufbau, aber auch für grundlegende Vorgehensweisen, wie z.B. die Strategie, welche den Zeitpunkt zur Ersetzung von Funktionseinheiten bestimmt. Bei sehr langen Rekonfigurationszeiten lohnt dabei eine Änderung der Zusammenstellung der Funktionseinheiten nur dann, wenn zusätzliche Ressourcen frei werden oder benutzte Ressourcen abgegeben werden müssen. Bei sehr kurzen Rekonfigurationszeiten oder Multi-Context-Bausteinen hingegen

verbessert für viele Programme eine Rekonfiguration den Gesamtdurchsatz an Befehlen durch den Prozessor beträchtlich. Die Rechenleistung des Systems steigt effektiv.

5.5 Erweiterungsmöglichkeiten des Ansatzes

Allen Prozessoren dieser Klasse ist gemein, dass sie auf rekonfigurierbarer Logik basieren. Diese erlaubt auch weitere Anpassungen des Prozessors an das zu bearbeitende Problem. Sie alle dienen der Beschleunigung der Ausführung von Programmen und der effizienteren Nutzung der programmierbaren Hardware.

5.5.1 Variable Registersätze

Eine Möglichkeit hierzu ist es, zu Zeiten, in denen eine große Menge an programmierbaren Hardware-Ressourcen zur Verfügung steht, zusätzliche Register für den Prozessor zu konfigurieren. Werden diese für Unterprogrammaufrufe genutzt und die vorher benutzten Register gehalten, beschleunigt dies den Prozessor, ohne dass eine Veränderung der Software nötig ist. Lediglich die Verdrahtung zu den Registern muss beim Sprung geändert werden. Wird zum Rücksprung die ursprüngliche Verdrahtung wiederhergestellt, entfällt der Zeitaufwand zum Sichern und Wiederherstellen der Registerinhalte. Diese Änderung der Verdrahtung kann dabei durch den Einsatz von Multiplexern so gelöst werden, dass sie ohne Zeitverzögerung stattfindet.
Im Gegensatz zur gewöhnlichen Erhöhung der Registerzahl, wie sie in aktuellen Prozessoren umgesetzt wird, bietet diese Erweiterung den Vorteil, dass zu Zeiten hohen Hardware-Aufwandes im kompletten System der Platz für die Register wieder für andere Funktionalität freigegeben werden kann.

5.5.2 Spezialisierte Funktionseinheiten

Wie bereits in verschiedenen Ansätzen für andere Prozessoren gezeigt (vergleiche Abschnitt 4.3), kann auch diese Klasse von Prozessoren durch Hinzunahme hochspezialisierter Funktionseinheiten profitieren. Mit diesen werden große Teile des Programms durch die massive Parallelität der Abarbeitung in Hardware in kürzester Zeit berechnet. Denkbar sind hier beispielsweise vektororientierte ALUs oder Multiply-Accumulate-Einheiten.
Ebenso können aber auch noch höher spezialisierte Einheiten hinzugefügt werden, welche durch Analysen aufgedeckte Hot-Spots, also voraussichtlich vielfach auszuführende Programmteile, durch deren Implementierung in Hardware schneller ausführen. Gerade die Ausführung häufig iterierender Schleifen wird durch speziell für diese implementierte Funktionseinheiten beschleunigt. Durch die Gleichartigkeit der Daten bei Anwendungen aus dem Multimedia-Bereich, ist dieses Vorgehen auch hier erfolgversprechend.

Eine Anpassung der Software und Compiler sowie ein entschieden höherer Implementierungsaufwand für sämtliche Programme sind für diese Erweiterung jedoch unumgänglich.

5.5.3 Tausch von Funktionseinheiten zwischen Prozessoren

Arbeiten mehrere Prozessoren in einem eingebetteten System zur selben Zeit unabhängig voneinander, so ist es möglich, dass diese während der Laufzeit ihre Funktionseinheiten untereinander tauschen. Benötigt also ein Prozessor zu einem Zeitpunkt seiner Berechnung eine FU nicht, so kann er diese freigeben. Ein anderer Prozessor des Systems kann diese dann einbinden und für seine eigenen Berechnungen verwenden.
Durch dieses Vorgehen wird die Chance, dass sich die Prozessoren während der Laufzeit auf ihre aktuelle Aufgabe optimieren können, erhöht. Da die FUs bereits vorhanden sind, entfallen die Konfigurationszeiten, was das Tauschen bereits für wenige Takte rentabel macht. Es wird sogar realistisch, dass anhand eines Lookahead-Buffers im Programm vorhergesehen werden kann, für welchen Zeitraum eine Einheit abgegeben werden kann, da sie nicht benötigt werden wird.
Eine zusätzliche Erweiterung dieses Ansatzes ist es, dass die Funktionseinheiten nicht vom Prozessor freigegeben werden, sondern, abhängig von den Prioritäten der gerade berechneten Prozesse, Funktionseinheiten von einer übergeordneten Einheit entzogen und umverteilt werden. Gerade in Realzeitsystemen kann es so ermöglicht werden, durch Optimierung der Gesamtleistung des Systems, mit kürzeren Deadlines zu arbeiten.

5.6 Zusammenfassung

Die vorgestellten Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren schaffen die Möglichkeit, ungenutzte Ressourcen der programmierbaren Hardware eines Systems-on-a-Chip für sinnvolle Rechenleistung zu verwenden. Durch die variable Abarbeitungsfolge der Befehle eines EPIC-Befehlsbündels können sie ihre Rechenleistung dynamisch an die vorhandenen Hardware-Ressourcen anpassen. Eine erneute Kompilation der Software ist nicht nötig.
Beschränkungen des Ansatzes, wie der zusätzliche Anspruch an Hardware und Befehlen, sind nicht relevant genug, um seine Einsatzmöglichkeiten grundlegend einzuschränken. Nutzen und Umsetzbarkeit zu analysieren ist Ziel der folgenden Kapitel.

6 Optimale EPIC-Befehlsverarbeitung

Die Ausführungszeit maximal parallelisierter EPIC-Befehlsbündel hängt in erster Linie von der Architektur der ausführenden Prozessoren ab. Herausforderungen wie Caching, Durchlaufzeiten einzelner Befehle, Pipelining etc. unterscheiden sich nicht von denen bei herkömmlichen Prozessoren und sind dort größtenteils gelöst.
Dementsprechend wird im folgenden Kapitel formal die Art der Funktionseinheiten für eine optimale, also zeitminimale, Ausführung eines EPIC-Befehls bestimmt, ein Merkmal, welches einzigartig für EPIC-Prozessoren ist. Die hierbei verwendeten Symbole sind auf Seite 149 im Verzeichnis verwendeter Symbole zusammengefasst.

6.1 Vorbedingungen

Um die optimale Verarbeitung von EPIC-Befehlsbündeln analytisch zu bestimmen, ist es nötig, sowohl die relevanten Eigenschaften des Programms als auch des Prozessors zu formalisieren.

6.1.1 Formalisierung des Aufbaus eines EPIC-Prozessors

Die folgende Analyse geht davon aus, dass ein Befehl eine Funktionseinheit in einer Zeiteinheit durchläuft. Durch Pipelining ist es in den meisten Fällen durchaus realistisch, diese Zeiteinheit einem Takt gleichzusetzen. Entsprechend werden diese Ausdrücke hier synonym verwendet. Vorerst wird davon ausgegangen, dass jede FU dieselbe Menge an Ressourcen auf dem Chip benötigt. Dieser Platz wird im Folgenden auch als *Slot* bezeichnet
Die Anzahl n an Arten von FUs F ist durch die Architektur vorherbestimmt. Eine Konfiguration C des Prozessors wird charakterisiert durch ihre Anzahlen a_i an Funktionseinheiten F_i der verschiedenen Arten.

$$C = (a_1, ..., a_n), \text{ mit } a_i = \#F_i \tag{6.1}$$

Die Anzahl an vorhandenen Slots wird mit S bezeichnet. Der Prozessor kann nie über mehr Funktionseinheiten verfügen, als Slots vorhanden sind.

$$S \geq \sum_{i=1}^{n} a_i \tag{6.2}$$

Die Anzahl der leeren, nicht verwendeten Slots wird als a_0 definiert. Es gilt:

$$a_0 = S - \sum_{i=1}^{n} a_i \;\Rightarrow\; S = \sum_{i=0}^{n} a_i \tag{6.3}$$

Die Menge aller Konfigurationen mit bis zu S Funktionseinheiten wird im Folgenden mit $\mathcal{C}_S$ bezeichnet.

6.1.2 Formalisierung des Aufbaus eines EPIC-Programms

Wie an Absatz 3.2.1 beschrieben besteht ein EPIC-Programm aus Befehlsbündeln, in denen nebenläufig ausführbare Befehle gruppiert sind. Ein aus k Einzelbefehlen β_i bestehendes Befehlsbündel B kann als k-Tupel definiert werden:

$$B = (\beta_1, ..., \beta_k) \tag{6.4}$$

Die Indizes der Befehle β_j, welche zum selben Typ F_i gehören, werden als Menge $\mathcal{B}_i$ zusammengefasst:

$$\mathcal{B}_i = \{\ j \in \{1, ..., k\} | \beta_j \text{ ist ausführbar von } F_i\ \} \tag{6.5}$$

Die Reihenfolge, in der einzelne Befehle ausgeführt werden, ist beliebig. Sie lassen sich je einer Funktionseinheit zuordnen, durch welche sie bearbeitet werden. Für die Laufzeit eines Programms ist es durch die angenommene einheitliche Durchlaufzeit also ausreichend, ein Befehlsbündel als n-Tupel aus der Anzahl b_i seiner Einzelbefehle eines Befehlstyps F_i anzusehen.

$$B = (b_1, ..., b_n), \quad \text{wobei } b_i = |\mathcal{B}_i| \tag{6.6}$$

Da ein Programm P aus einer Anzahl m von Befehlsbündeln besteht, kann es als Matrix aus den Befehlsanzahlen b_{ij} der einzelnen Bündel B_j dargestellt werden:

$$P = \begin{pmatrix} B_0 \\ \vdots \\ B_m \end{pmatrix} = \begin{pmatrix} b_{10} & \cdots & b_{n0} \\ \vdots & \ddots & \vdots \\ b_{1m} & \cdots & b_{nm} \end{pmatrix} \tag{6.7}$$

Jede Zeile der Matrix beschreibt hierbei ein Befehlsbündel aus nebenläufig ausführbaren Befehlen, jede Spalte die Abfolge der von einem Funktionseinheitentyp abzuarbeitenden Befehle. Ohne Beschränkung der Allgemeinheit wird davon ausgegangen, dass jedes Programm mindestens einen Befehl von jedem Befehlstyp hat, also keine der Spalten der Matrix komplett mit Nullen gefüllt ist.

6.2 Ausführungszeiten auf statischen EPIC-Prozessoren

Diese Notationen erlauben es, die Ausführungszeit eines Befehlsbündels sowie eines kompletten Programms formal zu definieren. Zunächst werden statische EPIC-Prozessoren, also solche, deren Funktionseinheiten sich nicht ändern können, betrachtet.

6.2.1 Ausführungsdauer eines Befehlsbündels

Die Befehle β eine Bündels, welche vom Funktionseinheitentyp F_i bearbeitet werden müssen, werden so lange den entsprechenden Funktionseinheiten zugeordnet, bis sie komplett abgearbeitet sind. Es ergibt sich also die Anzahl an Zeitschritten θ für die Verarbeitung als Quotient der Anzahl b_i von Befehlen durch die Anzahl a_i an entsprechenden Funktionseinheiten. Da vor der Abarbeitung aller Befehle auch nicht verwendete Funktionseinheiten auf die Fertigstellung der letzten Befehle warten und Zeitschritte grundsätzlich ganzzahlig sind, muss der Quotient aufgerundet werden.

$$\theta(F_i) = \left\lceil \frac{b_i}{a_i} \right\rceil \tag{6.8}$$

Wie beschrieben, müssen alle Befehle von allen Typen eines Befehlsbündels komplett abgearbeitet sein, bevor mit den Berechnungen für das nächste begonnen werden kann. Also entspricht die Abarbeitungsdauer t eines kompletten Bündels der Zeit, welche die Einheit, die bei diesem Bündel am längsten arbeitet, benötigt:

$$t_C(B) = \max\left(\{\theta(F_i) \mid i \in \{1, ..., n\}\}\right) \tag{6.9}$$

6.2.2 Ausführungsdauer eines Programms

Nachdem die Dauer für die Ausführung eines Bündels bestimmt wurde, ergibt sich die Ausführungsdauer T für ein komplettes Programm P mit der Konfiguration C. Da ein solches eine Ansammlung von unabhängigen Bündeln ist, können hierfür die Ausführungsdauern t der einzelnen Bündel addiert werden.

$$T_C(P) = \sum_{i=0}^{m} t_C(B_i) \tag{6.10}$$

6.3 Optimale statische Prozessorkonfiguration

Um ein Programm abarbeiten zu können, muss eine statische Konfiguration C zu jedem Befehlstypen der im Programm vorkommenden Befehle mindestens eine Funktionseinheit besitzen. Die restlichen vorhandenen Slots können sich beliebig auf alle Funktionseinheitentypen verteilen.

$$C = (a_1, ..., a_n), \quad \text{wobei } a_i \in \{1, 2, ..., S - n\} \tag{6.11}$$

Eine Konfiguration der Funktionseinheiten eines Prozessors ist dann optimal für ein bestimmtes Programm P, wenn keine andere Konfiguration mit der gleichen Anzahl an Slots eine schnellere Abarbeitung ermöglicht. Für eine optimale Konfiguration C_{opt} muss folgende Gleichung erfüllt sein:

$$T_{C_{opt}}(P) = \min\left(\{T_C(P) \mid C \in \mathcal{C}_S\}\right) \tag{6.12}$$

6.3.1 Optimale Konfiguration für die Verarbeitung eines Befehlsbündels

Für ein einzelnes Befehlsbündel gestaltet sich die Bestimmung der optimalen Konfiguration wenig aufwändig. Die Ausführungsgeschwindigkeit des Bündels wird, wie bereits in Gleichung 6.9 gezeigt, immer durch den am längsten ausgelasteten Funktionseinheitentyp begrenzt. Es ist also möglich, ausgehend von einer minimalen Konfiguration mit $a_1 = a_2 = ... = a_n = 1$, iterativ immer dem aktuell am meisten ausgelasteten Funktionseinheitentyp eine weitere Funktionseinheit hinzuzufügen. Die Auslastung eines Einheitentyps entspricht dabei dem Quotienten aus der Anzahl an Befehlen und Funktionseinheiten eines Typs. Sobald alle verfügbaren Slots vergeben sind, ist eine optimale Konfiguration für die Abarbeitung des Bündels gefunden. Dieses Vorgehen lässt sich mit einem Algorithmus mit der Laufzeit $\Theta((S-n)\cdot n)$ formal wie folgt beschreiben:

while *Slots remaining* **do**
  for $i = 1..n$ **do**
    $load\,[\,i\,] = B\,[\,i\,]\ /\ C\,[\,i\,]$
  end
  $index = (j \mid load\,[\,j\,] == max\,(load)\,)$
  $C\,[\,index\,]++$
  $Slots--$
end

Führt dieser Algorithmus auch stets zu einer optimalen Lösung, so ist er trotzdem nicht deterministisch. Gilt während des Aufbaus der Konfiguration für eine Zwischenkonfiguration C_{temp} die Gleichung $\theta(F_i) = \theta(F_j) = t_{C_{temp}}(B),\ i \neq j$, so bleibt die freie Wahl, ob a_i oder a_j in der Folgekonfiguration erhöht wird.

6.3.2 Optimale Konfiguration für die Verarbeitung eines Programms

Soll ein Programm, also eine größere Anzahl an Befehlsbündeln, ausgeführt werden, kann es passieren, dass die optimale Konfigurationen bei Erhöhung der Slot-Anzahl auch weniger Funktionseinheiten von einem Typ enthält als die vorhergehende. Diese Tatsache lässt sich an diesem einfachen Beispiel mit einem Programm aus zwei Bündeln auf einem Prozessor mit zwei Typen von Funktionseinheiten erkennen:

$$P = \begin{pmatrix} 1 & 2 \\ 3 & 1 \end{pmatrix}$$

$$\begin{aligned} S = 2: &\quad C_{opt} = (1,1), &\quad T_{C_{opt}}(P) = 5 \\ S = 3: &\quad C_{opt} = \mathbf{(1,2)}, &\quad T_{C_{opt}}(P) = 4 \\ S = 4: &\quad C_{opt} = \mathbf{(3,1)}, &\quad T_{C_{opt}}(P) = 3 \end{aligned}$$

Hier kann die optimale Konfiguration C_{opt} für $S = 4$ nicht mehr iterativ aufbauend auf der Konfiguration für $S = 3$ ermittelt werden. Somit kann der Algorithmus aus Abschnitt 6.3.1 nicht mehr angewendet werden. Die Optimierung der Konfiguration für ein Programm muss auf eine andere Weise geschehen.

6.3.3 Optimale Konfiguration als Optimierungsproblem

Es stellt sich somit ein Optimierungsproblem, bei dem der Lösungsraum Ω den möglichen Konfigurationen und die Fitnessfunktion f der Abbildung der Konfigurationen auf deren Ausführungszeiten für ein vorgegebenes Programm entspricht. Genauer kann das Problem als Suchproblem beschrieben werden, da eine optimale Konfiguration $C_{opt} \in \Omega$ gesucht wird.

$$\Omega = \{C \in \mathcal{C}_S \mid \forall\, i \in \{1, ..., n\} : a_i \geq 1\} \tag{6.13}$$

$$f(C_{opt}) = \min \{T_C(P) \,|\, C \in \Omega\} \tag{6.14}$$

Dadurch, dass in diesem Problem alle Variablen nur ganzzahlige Werte annehmen dürfen, liegt hier ein Problem der *Ganzzahligen Linearen Optimierung* vor. Solche Probleme sind NP-vollständig und somit nicht effizient zu lösen.

6.3.4 Lösbarkeit des Optimierungsproblems

Für solche NP-vollständigen Probleme erhält man stets eine optimale Lösung, wenn man die Brute-Force-Methode anwendet und die Lösung für alle möglichen Eingaben berechnet, sofern die Anzahl der möglichen Lösungen klein genug ist, um dies zuzulassen. Der Rechenaufwand hierfür ist jedoch in der Regel sehr groß. In diesem Problem entsprechen die möglichen Eingaben den Konfigurationen des Prozessors.
Die Funktionseinheiten einer Konfiguration haben keine Reihenfolge, Funktionseinheiten desselben Typs sind nicht unterscheidbar. Also entspricht die Menge aller möglicher Konfigurationen, und somit die Menge Ω, der Menge aller Kombinationen mit Repetition von S Funktionseinheiten aus n Funktionseinheitentypen. Dadurch, dass wie eben beschrieben jede Funktionseinheit mindestens einmal vorkommen muss, können nur $S - n$ frei gewählt werden. Die Mächtigkeit der Menge Ω, und somit die Anzahl der möglichen Lösungen, kann deswegen wie folgt berechnet werden:

$$|\Omega| = \left(\begin{array}{c} n + (S - n) - 1 \\ S - n \end{array} \right) = \left(\begin{array}{c} S - 1 \\ S - n \end{array} \right) \tag{6.15}$$

Die Anzahl der möglichen Lösungen wächst somit, wie auch in Abbildung 6.1 zu erkennen, sehr schnell. Existierende EPIC-Prozessoren und mit ihnen auch die passenden Compiler sind meist auf vier Typen von Funktionseinheiten ausgelegt. Für reale Beispiele von Konfigurationen kann also von $n = 4$ ausgegangen werden. Damit bleibt die Anzahl an Konfigurationen

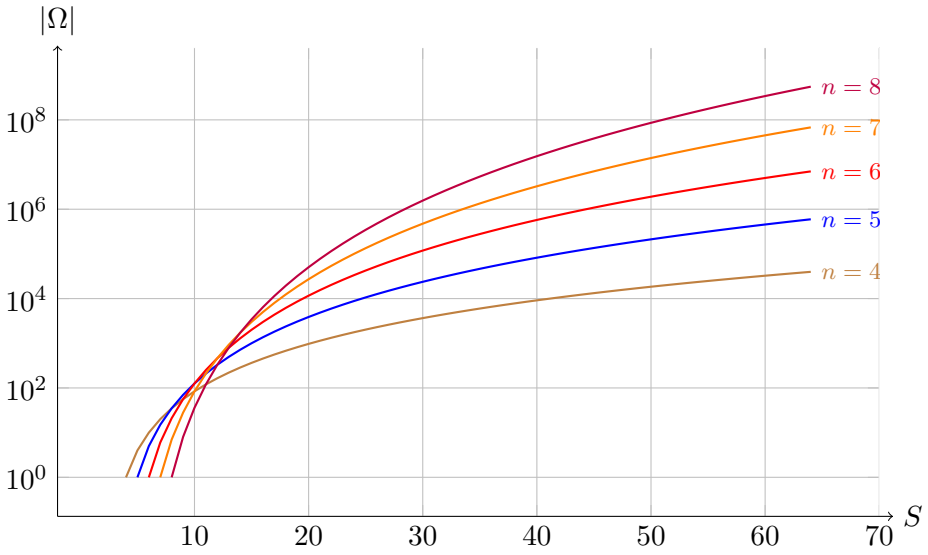


Abbildung 6.1: Mächtigkeit des Lösungsraums Ω

noch für eine aus technischer Sicht sehr hohe Anzahl an Slots $S > 64$ unter 10^5. Ist der konstante Aufwand für die Berechnung der Laufzeit einer Konfiguration gering, können für diese Eingabewerte optimale Lösungen in akzeptabler Berechnungszeit gefunden werden.

6.4 Berechnung der Laufzeit bei dynamischer Rekonfiguration

Zur Selbstoptimierung kann, während eines Laufes eines wie in Gleichung 6.7 definierten Programms, die Konfiguration des Prozessors geändert werden. Wegen der Dauer einer Rekonfiguration wird im Folgenden von der Änderung von nur einer Funktionseinheit zu einem Zeitpunkt ausgegangen.
Die Konfiguration C entspricht einer Startkonfiguration des Prozessors und wird als gegeben angesehen. Um den Prozessor zu optimieren, und somit eine Funktionseinheit gegen eine andere auszutauschen, muss zuerst eine FU entfernt werden. Wird eine Funktionseinheit vom Typ F_j entfernt, wird die dadurch entstehende Konfiguration $C_{\square j}$ genannt. Während der von der Hardware vorgegebenen Rekonfigurationszeit $\mathcal{T}$, welche benötigt wird, um die neue Funktionseinheit vom Typ F_i zu konfigurieren, kann der Prozessor mit dieser Konfiguration $C_{\square j}$ weiterarbeiten. Danach wird der restliche Programmteil dann von der resultierenden Konfiguration C_{ij} abgearbeitet.
Der Fall $i = j$ würde bedeuten, dass eine sinnlose Rekonfiguration ohne Änderung der Konfiguration oder keine Rekonfiguration, wie in Abschnitt 6.3 behandelt, stattfindet. Dieser Fall wird entsprechend hier ausgeschlossen.

$$C = (a_1, ..., a_n), \quad C_{\square j} = (a'_1, ..., a'_n), \quad C_{ij} = (a''_1, ..., a''_n), \quad i \neq j \tag{6.16}$$

Da sich die beiden Konfigurationen C und C_{ij} nur in der Anzahl zweier Funktionseinheitentypen um jeweils eins unterscheiden gilt:

$$\begin{aligned} a'_l &= \begin{cases} a_l & \text{falls } l \neq j \\ a_l - 1 & \text{falls } l = j \end{cases} \\ a''_l &= \begin{cases} a_l & \text{falls } l \neq j \\ a_l - 1 & \text{falls } l = j \\ a_l + 1 & \text{falls } l = i \end{cases} \end{aligned} \tag{6.17}$$

Wird die Konfiguration nach der Abarbeitung von k Bündeln geändert, werden mit dieser Anfangskonfiguration C die Bündel B_0 bis B_k abgearbeitet. $T_C(B_0, ..., B_k)$ entspricht der bis dahin vergangenen Zeit. Die Zeit $T_{C_{\square j}}(B_{k+1}, ..., B_l)$ ist die Rekonfigurationszeit $\mathcal{T}$.

$$\mathcal{T} = T_{C_{\square j}}(B_{k+1}, ..., B_l) \tag{6.18}$$

Die Bündel B_{k+1} bis B_l sind die Bündel, welche während dieser Zeit mit der verminderten Konfiguration $C_{\square j}$ abgearbeitet werden können. Der restliche Teil B_{l+1} bis B_m des Programms wird dann von der optimierten Konfiguration C_{ij} in der Zeit $T_{C_{ij}}(B_{l+1}, ..., B_m)$ ausgeführt.

$$P = \left(\begin{array}{c} B_0 \\ \vdots \\ B_{ka} \\ B_{ka+1} \\ \vdots \\ B_{ke} \\ B_{ke+1} \\ \vdots \\ B_m \end{array} \right) \quad \begin{array}{l} \Big\} \text{ wird abgearbeitet mit } C \\ \Big\} \text{ wird abgearbeitet mit } C_{\square j} \\ \Big\} \text{ wird abgearbeitet } \textit{mit } C_{ij} \end{array} \quad \begin{array}{c} \Big\updownarrow \; T_C(B_0, ..., B_k) \\ \Big\updownarrow \; \mathcal{T} \\ \Big\updownarrow \; T_{C_{ij}}(B_{l+1}, ..., B_m) \end{array}$$

Die Laufzeit $T_{C \to C_{ij}}$ des Programms P entspricht somit der Summe aus den Laufzeiten des Programmteils, welcher mit Konfiguration C_1 ausgeführt wird, desjenigen, welcher mit Konfiguration C_2 ausgeführt wird, und der der Rekonfigurationszeit.

$$T_{C \to C_{ij}}(P) = T_C(B_0 ... B_k) \; + \; \mathcal{T} \; + \; T_{C_{ij}}(B_{l+1}, ..., B_m) \tag{6.19}$$

Da B_k ein gewähltes Bündel ist, nach dessen Ausführung die Rekonfiguration beginnt, lässt sich die Laufzeit $T_C(B_0 ... B_k)$ mit den Gleichungen 6.9 und 6.10 berechnen. Mit der konstanten Rekonfigurationszeit $\mathcal{T}$ ist durch die Addition der Laufzeiten der Bündel ab B_{k+1} der Index l bestimmbar, indem die Aufsummierung abgebrochen wird, sobald die Summe $\mathcal{T}$ erreicht oder übersteigt. Folgender Algorithmus erfüllt dies:

$time = 0$
$l = k$
while $time < \mathcal{T}$ **do**
  $l++$
  $time = time + t_{C_{\square j}}(B_l)$
end

Mit dem Index l kann dann $T_{C_{ij}}(B_{l+1}, ..., B_m)$ wieder mit den Gleichungen 6.9 und 6.10 berechnet werden.

6.4.1 Ideale Änderung der Konfiguration ohne Kosten

In dem idealen Fall, dass keine Kosten für die Rekonfiguration anfallen, also $\mathcal{T} = 0$ gilt, vereinfacht sich die Berechnung der Laufzeit des Programms bei der Änderung einer Funktionseinheit. Beim Einsatz von Multi-Context FPGAs kann dieser Fall realistisch sein.
Direkt aus der nicht vorhandenen Rekonfigurationszeit folgt, dass kein Bündel des Programms mit $C_{\square j}$ abgearbeitet wird. Dementsprechend gilt $k = l$. Die komplette Laufzeit für ein Programm berechnet sich also ohne Kosten mit folgender Gleichung:

$$T_{C \to C_{ij}}(P) = T_C(B_0...B_k) \; + \; T_{C_{ij}}(B_{k+1}, ..., B_m) \tag{6.20}$$

Die zwei Summanden lassen sich hier beide mit den Gleichungen 6.9 und 6.10 berechnen.

6.4.2 Rentabilität der Rekonfiguration

Solch ein Rekonfigurationsvorgang ist rentabel, wenn die komplette Ausführung des Programms mit der Rekonfiguration trotz der Kosten schneller abläuft als ohne die Rekonfiguration. Entsprechend muss, damit eine Rekonfiguration der Funktionseinheiten einen Nutzen ergibt, folgende Ungleichung erfüllt sein:

$$T_C(P) > T_{C \to C_{ij}}(P) \tag{6.21}$$

Da der erste Programmteil stets mit der Konfiguration C ausgeführt wird, beschreibt folgende leichter auszuwertende Ungleichung ebenso die Rentabilität:

$$T_C(B_{k+1}...B_m) > T_{C_{\square j}}(B_{k+1}...B_l) + T_{C_{ij}}(B_{l+1}...B_m) \tag{6.22}$$

6.4.3 Optimale Rekonfiguration

Ein Rekonfigurationsschritt ist dann als optimal zu sehen, wenn durch ihn die Berechnung des Programms minimale Laufzeit benötigt. Somit muss die Rekonfiguration, um optimal zu sein, auf jeden Fall rentabel (vgl. Abschnitt 6.4.2) sein. Zudem darf kein anderer Rekonfigurationsschritt zu diesem Zeitpunkt eine schnellere Ausführung des Programms ermöglichen.

Als variable Größen wird hier der Typ F_j der entfernten sowie der Typ F_i der hinzugefügten Funktionseinheit angenommen. Feste Größen sind demgegenüber die Startkonfiguration sowie der Zeitpunkt der Rekonfiguration. Im realen Fall entspricht die Dauer eines Rekonfigurationsschrittes $\mathcal{T}$ in der Regel dem Quotienten aus der Größe der Konfigurationsdaten für eine Funktionseinheit und der Rekonfigurationsbandbreite. Sie ist somit eine technologieabhängige Größe der Hardware. Bei angenommener gleicher Größe für jeden Funktionseinheitentyp ist auch sie somit konstant.
Mit diesen Annahmen lässt sich die Berechnungszeit eines Programms mit optimaler Rekonfiguration wie folgt bestimmen:

$$T_{C \xrightarrow{opt} C_{ij}}(P) = \min \{T_C(B_0...B_k) + T_{C_{\square j}}(B_{k+1}...B_l) + T_{C_{ij}}(B_{l+1}...B_m) \mid i,j \in \{1,...,n\}\} \tag{6.23}$$

Um eine nach dieser Voraussetzung optimale Rekonfiguration zu bestimmen, ist es somit im Normalfall nötig, für alle n möglichen Funktionseinheitentypen die Anzahl der berechneten Bündel während der Rekonfiguration zu bestimmen. Ist bereits in der Ausgangskonfiguration C ein Funktionseinheitentyp bereits nur einmal vorhanden, senkt sich diese Anzahl entsprechend. Zudem muss die Laufzeit für jede der $n-1$ möglichen Folgekonfigurationen ermittelt werden. Um eine optimale Rekonfiguration zu finden, ist es somit nötig, $\mathcal{O}(n^2)$ mal die Ausführungszeit $T_{C \to C_{ij}}(P)$ zu berechnen.

6.5 Zusammenfassung

Die Formalisierung von Konfiguration und Programm eines Laufzeitrekonfigurierbaren EPIC Soft Core Prozessors ermöglicht die Bestimmung der Laufzeiten von Programmen. Darauf aufbauend ist die Konfiguration, welche ein Programm in minimaler Zeit abarbeitet, berechenbar. Dies ist jedoch mit hohem Rechenaufwand verbunden.
Die Laufzeit eines Programms, während der ein Optimierungsschritt durch Rekonfiguration stattfindet, ist analog bestimmbar. Ein solcher Schritt ist rentabel, wenn er die Ausführung, auch unter Berücksichtigung seiner Kosten, beschleunigt. Als optimal kann er dann angesehen werden, wenn zu dem Zeitpunkt keine andere Rekonfiguration eine kürzere Programmlaufzeit ermöglichen würde.
Mit den Ergebnissen dieses Kapitels wird im Folgenden der Nutzen von Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren anhand geeigneter Eingabewerte bestimmt.

7 Analyse des Nutzens

Um die Effizienz der vorgestellten Prozessorklasse zu bestimmen, bietet es sich an, die Laufzeiten wie im vorherigen Kapitel beschrieben zu berechnen. Geschieht dies mit geeigneten Eingabedaten, können durch deren Variation die Vor- und Nachteile der Architektur in Abhängigkeit von ihrem Umfeld bestimmt werden. Der Nutzen der partiellen Rekonfiguration zur Änderung der Anzahl an Funktionseinheiten sowie zur Selbstoptimierung steht dabei als herausragende Eigenschaft der Architektur im Fokus.

7.1 Eingabedaten

Die Eingabedaten für die Effizienzberechnungen repräsentieren die Eingaben für den Prozessor, also das Programm. Sie sind auf die für die Laufzeit relevanten Informationen reduziert.

7.1.1 Programm-Traces

Würde ein nicht weiterverarbeitetes Programm als Eingabe benutzt, müsste die komplette Ablaufsteuerung des Prozessors, ähnlich einer virtuellen Maschine, nachgebildet werden. Da dies die Berechnungen stark verkomplizieren und somit verlangsamen würde, bietet es sich stattdessen an, Programm-Traces zu benutzen.
Ein Programm-Trace besteht aus den Befehlen, die ein Prozessor bei der Abarbeitung eines Programms tatsächlich ausführt. Diese Befehle erhalten im Trace auch ihre tatsächliche Reihenfolge. Der Trace unterscheidet sich vom Maschinenprogramm in erster Linie dadurch, dass Verzweigungen im Programm bereits aufgelöst sind. Sprungbefehle sind, da sie vom Prozessor ausgewertet werden müssen, Bestandteil von Traces. Traces enthalten jedoch keine Schleifen. Folgendes Beispiel zeigt den Unterschied anhand eines kurzen Programmstücks:

Assembler-Programm

```
      mov R0, #0
      mov R1, #2
loop: add R0, #1
      cmp R0, R1
      jne loop
      add R1, #1
```

Programm-Trace

```
mov R0, #0
mov R1, #2
add R0, #1
cmp R0, R1
jne loop
add R0, #1
cmp R0, R1
jne loop
add R1, #1
```

Solche linearen Traces entstehen durch Ausführung des tatsächlichen Programms auf einem entsprechend instrumentierten Prozessor oder einer bestehenden virtuellen Maschine. Sie ermöglichen es somit, exakte Zeitberechnungen durchzuführen.

7.1.2 Erzeugung von Programm-Traces

Die eingesetzten Traces wurden mit der in der Bachelorarbeit [91] definierten Vorgehensweise erstellt. Hierzu werden C-Programme mittels des Trimaran-Frameworks (siehe Kapitel 3.5) kompiliert und simuliert.
Sowohl für den Compiler als auch für die Simulation wird eine große Maschine mit jeweils 16 Funktionseinheiten von jedem der vier Typen Integer, Floating Point, Memory und Branch zugrunde gelegt. Zudem werden große Registersätze benutzt. Dadurch werden Traces erzeugt, welche die vom Compiler maximal extrahierbare Instruktions-Level-Parallelität erreichen.

7.1.3 Benutzte Benchmarks

Die Traces werden aus Benchmarks, also Test-Programmen, generiert. Diese Benchmarks liegen in C-Code vor. Für weitere Berechnungen wird ein großer Teil der *simple*-Benchmarks des Trimaran-Frameworks eingesetzt. Zudem werden die Olden-Benchmarks [20, 123], welche für Parallelisierung optimiert sind, benutzt. Wegen ihrer Größe kommt auch teilweise die CRC-Berechnung aus den Netbench-Benchmarks [85] zum Einsatz.
Die verwendet Benchmarks sind in den Tabellen 7.1, 7.2 und 7.3 erklärt.

7.1.4 Aufbau der Eingabedaten

In den erstellten Traces sind die Befehlsbündel durch Zeilenumbrüche voneinander getrennt. Jede Zeile enthält genau ein Bündel. Ein solches besteht aus den durch zwei Semikolons getrennten Einzelbefehlen ohne Argumente. In Klammern ist jedem Befehl der verarbeitende Funktionseinheitentyp zugeordnet. Der Typ wird durch einen Buchstaben codiert. Wie zuvor entspricht dabei `M` einem Memory-Befehl, `I` einem Integer-Befehl, `F` einem Floating-Point-Befehl und `B` einem Befehl für eine Branch-Einheit. Eine beispielhafte Zeile eines Traces könnte also wie folgt aussehen:

```
LG_W_C1_C1(M) ;; PBR(B) ;; PBR(B) ;; MOVE(I) ;; ADD_W(I) ;; BSAVEG(M) ;;
```

7.1.5 Statistische Eingabedaten

Zum Vergleich mit den Benchmarks werden zudem statistisch generierte Eingabedaten benutzt. Sowohl gleichverteilte als auch normalverteilte Traces kommen dabei zum Einsatz. Sie dienen dazu, das Verhalten des Prozessors mit realen Traces zu approximieren. Sie ermöglichen somit, auch erste Aussagen über Konfigurationen mit einer höheren Anzahl an Funktionseinheitentypen sowie höherer Instruktion-Level-Parallelität zu treffen.

Tabelle 7.1: Benutzte *simple*-Benchmarks des Trimaran-Frameworks

Benchmark	Beschreibung
bmm	Blockweise Matrizenmultiplikation mit Integer-Werten
dag	If-Then-Else-Struktur in einer For-Schleife
fft	Schnelle Fourier-Transformation
fir	FIR-Filter mit Float-Werten
fir_int	FIR-Filter mit Integer-Werten
mm	Matrizenmultiplikation mit Float-Werten
mm_double	Matrizenmultiplikation mit Double-Werten
mm_dyn	Matrizenmultiplikation mit Integer-Werten mit dynamischer Allokation des Speichers
mm_int	Matrizenmultiplikation mit Integer-Werten
nested	Modulo Scheduling mit geschachtelten Schleifen
paraffins	Berechnet Anzahl der Paraffin-Isomere
sqrt	Quadratwurzel mit dem Newton'schen Näherungsverfahren
strcpy	Kopiert ein Array aus Buchstaben
switch_test	Switch-Case-Anweisungen in einer For-Schleife
type_test	Testet Structs und Unions
wave	Einfache 2D-Wellenfront-Berechnung

Tabelle 7.2: Benutzte Olden-Benchmarks

Benchmark	Beschreibung
olden_bisort	Sortieren von Zahlen mit Bitonic Sort
olden_em3d	Ausbreitung einer Welle in einem 3D-Objekt
olden_health	Columbian Health Care Simulation
olden_mst	Minimaler Spannbaum eines Graphen
olden_perimeter	Umfang von Regionen in Bildern
olden_treeadd	Rekursive Summe von Werten eines B-Baums
olden_tsp	Problem des Handlungsreisenden

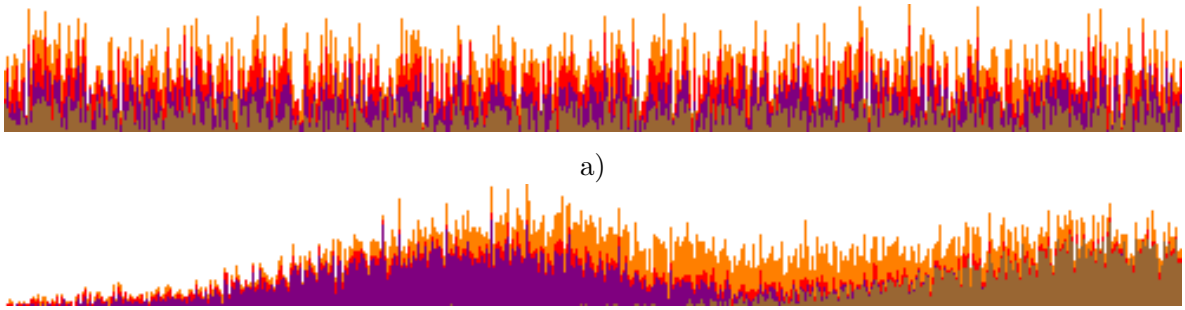
Tabelle 7.3: Benutzte Benchmarks aus der Netbench-Suite

Benchmark	Beschreibung
crc	Berechnet den CRC-Wert

Da diese Daten kein reales Programm repräsentieren, genügt es, sie so zu generieren, dass ein Befehlsbündel ausschließlich aus den verarbeitenden Funktionseinheiten besteht. Eine Zeile eines solchen randomisierten Traces könnte also so aussehen:

```
(M);;(M);;(B);;(B);;(I);;(I);;
```

Für gleichverteilte Traces wird die Anzahl an benutzten Einheiten eines Typs in jedem Bündel erneut zufällig festgelegt. Die maximale Anzahl an Einheiten eines Typs wird dabei durch eine Variable vorgegeben.
Für Traces mit normalverteilter Nutzung von Funktionseinheiten wird sowohl die Anzahl an sich überlagernden Normalverteilungen, deren Position im Trace als auch die Standardabweichung pro Funktionseinheitentyp per Zufall bestimmt. Der Grad an Parallelität wird hier durch einen mittleren Wert von zeitgleich ausführbaren Funktionen in einem Parameter angegeben.
Abbildung 7.1 visualisiert so erstellte randomisierte Eingabedaten. Dabei stellt jede der 500 Spalten des Diagramms ein Bündel dar. Die Farben entsprechen den unterschiedlichen Typen von Funktionseinheiten. Die maximal Anzahl bei den gleichverteilten Daten sowie die mittlere Anzahl bei den normalverteilten Daten wurden hier beispielhaft auf je 16 gesetzt.



a)

b)

Abbildung 7.1: Statistische Eingabedaten: a) Gleichverteilte Daten b) Normalverteilte Daten

7.2 Berechnung der Abarbeitungsdauer

Die Abarbeitungsdauer wird, wie in Kapitel 6.2 beschrieben, für die oben genannten Benchmarks berechnet. Um abhängig von der Anzahl an Slots die minimale Ausführungszeit zu bestimmen, werden die Laufzeiten aller Konfigurationen geprüft.

7.2.1 Reduzierung der Laufzeit der Berechnung einer Konfiguration

Wie in Abschnitt 6.3.4 dargestellt, ist es für die Berechnung optimaler Konfigurationen essentiell, den Aufwand zur Bewertung einer Konfiguration so weit wie möglich zu senken. Hierzu werden folgende drei Methodiken eingesetzt:

7.2.1.1 Numerische Darstellung der Eingabedaten

Wie in Abschnitt 6.1 beschrieben, wird sowohl das Programm als auch die Konfiguration so knapp wie möglich durch eine Menge ganzzahliger Werte repräsentiert. Im Gegensatz zu symbolischen Darstellungsformen können diese ohne große Umformungen direkt zur Berechnung der Laufzeiten benutzt werden.
Ein Bündel des Traces kann entsprechend zuerst textuell auf seinen Funktionseinheitenbedarf und dann auf die numerische Darstellung der benötigten Einheiten reduziert werden. Es resultiert ein Vektor zur Beschreibung des Bündels. Die Zuordnung der Position der Zahl im Vektor zum Funktionseinheitentyp ist nur zur Ausgabe der fertig berechneten Konfiguration nötig.
Das Beispiel aus Absatz 7.1.4 wird somit wie folgt reduziert:

Im Trace:	`LG_W_C1_C1(M);;PBR(B);;PBR(B);;MOVE(I);;ADD_W(I);;BSAVEG(M);;`	
Als Text:	`(B)(B)(I)(I)(M)(M)`	
Als Vektor:	`(2, 0, 2, 2)`	mit der Zuordnung (B, F, I, M)

7.2.1.2 Kumulation von Bündeln mit gleichem Ressourcenbedarf

Des Weiteren werden Bündel, welche die gleichen Funktionseinheiten benutzen, zusammengefasst und mit einem Multiplikator versehen. Somit ist es ausreichend, die für ihre Abarbeitung benötigte Zeit einmal pro Konfiguration zu berechnen, um sie dann mit dem Zeilenmultiplikator zu multiplizieren. Weil für die Dauer der Abarbeitung die Reihenfolge der Bündel keine Rolle spielt, bleibt das Ergebnis hierdurch korrekt.
Reale Eingaben bestehen meist aus kompilierten und simulierten Programm-Traces, also Abläufen von tatsächlich auszuführenden Befehlen. Allein die Tatsache, dass dabei aus einem Programm mit wenigen hundert Zeilen ein Programm-Trace mit mehreren Millionen ausgeführten Bündeln entstehen kann, legt nahe, dass dieses Vorgehen großen Nutzen bringt. Die Reduktion des Aufwandes hierdurch ist tatsächlich für die meisten realen Eingaben im Bereich mehrerer Größenordnungen. Tabelle 7.4 zeigt das Ergebnis der Kumulation für die benutzten Benchmarks, also die Anzahl der Zeilen der Programmmatrix, welche dann zur Berechnung der Laufzeit eingesetzt wird. Zudem zeigt sie die Zeit, welche einmalig pro Eingabe-Trace für diese Berechnung, der Kumulation, benötigt wird. Testrechner war hierbei ein mit 3 GHz getakteter Intel Pentium 4 mit 2 GByte RAM.

Tabelle 7.4: Reduktion der Programmmatrix durch Kumulation der Bündel

Benchmark	Bündel im Trace	Zeilen der Matrix	Dauer der Kumulation	Benchmark	Bündel im Trace	Zeilen der Matrix	Dauer der Kumulation
bmm	23.112	33	0.55 s	olden_health	417.874	77	7.58 s
dag	3.556	11	0.06 s	olden_mst	985.260	38	18.25 s
fft	1.321.921	61	27.72 s	olden_perimeter	1.165.213	74	24.33 s
fir	190.847	33	3.58 s	olden_treeadd	25.750	36	0.63 s
fir_int	54.303	32	1.23 s	olden_tsp	2.726	72	0.03 s
mm	25.065	29	0.63 s	paraffins	94.810	36	1.66 s
mm_double	24.667	33	0.61 s	sqrt	1.321	19	0.03 s
mm_dyn	19.351	32	0.63 s	strcpy	6.162	14	0.14 s
mm_int	17.381	28	0.59 s	switch_test	14.498	16	0.28 s
nested	15.167	24	0.28 s	type_test	5.277	30	0.11 s
olden_bisort	4.233	62	0.09 s	wave	5.906	23	0.13 s
olden_em3d	8.540	36	0.20 s	crc	13.401.599	23	248.36 s

7.2.1.3 Frühzeitiges Verwerfen nicht optimaler Konfigurationen

Weitere Laufzeitersparnis lässt sich erreichen, indem die aus der Kumulation resultierenden Zeilen nach ihrem Multiplikator sortiert und die Laufzeit der Bündel mit großem Multiplikator so früh wie möglich berechnet werden. Die akkumulierte Laufzeit des Programms steigt hierbei grundsätzlich anfangs sehr viel schneller. Erreicht sie während des Vorgangs die Laufzeit der bisher besten gefundenen Konfiguration, kann sie direkt verworfen und die nächste mögliche Konfiguration getestet werden.

7.2.1.4 Ergebnis

Durch diese drei Optimierungen wird die Laufzeit für die Berechnung trotz Anwendung der Exhaustionsmethode, also der erschöpfenden Berechnung mit allen möglichen Konfigurationen, für die benutzten Benchmarks in einen annehmbaren Bereich gesenkt.
Die Berechnungsdauer für die Laufzeit wird in Tabelle 7.5 angegeben. Die Dauer ist hierbei die Berechnungszeit für alle Kombinationen an Funktionseinheiten sowie für alle Anzahlen an Slots bis zum Finden einer Konfiguration, welche alle Bündel des Programms in je einem Zyklus abarbeitet.
Die Werte sind ohne die Dauer zur Kumulation, wie in Abschnitt 7.2.1.2 beschrieben, wiederum auf einem mit 3 GHz getakteten Intel Pentium 4 mit 2 GByte RAM gemessen.

Sehr viel länger dauern diese Berechnungen für Programm-Traces, welche weniger stark kumuliert werden können. Dies gilt beispielsweise für Traces, die aus Source-Code mit wenigen Schleifendurchläufen erstellt wurden oder für statistische Eingabedaten.

Tabelle 7.5: Berechnungsdauer für alle Anzahlen an Slots ohne Dauer der Kumulation

Benchmark	Dauer	Benchmark	Dauer	Benchmark	Dauer
bmm	0.75 s	mm_int	0.16 s	olden_tsp	5.60 s
dag	0.04 s	nested	0.25 s	paraffins	0.38 s
fft	16.80 s	olden_bisort	0.74 s	sqrt	0.06 s
fir	0.26 s	olden_em3d	0.49 s	strcpy	0.02 s
fir_int	0.22 s	olden_health	7.33 s	switch_test	0.10 s
mm	0.81 s	olden_mst	0.41 s	type_test	22.71 s
mm_double	0.26 s	olden_perimeter	1.93 s	wave	0.06 s
mm_dyn	0.34 s	olden_treeadd	0.26 s	crc	0.08 s

7.3 Abschätzung der Leistungsfähigkeit

Das im vorherigen Abschnitt beschriebene Verfahren liefert die Dauer der Abarbeitung der Benchmarks in Abhängigkeit von der Anzahl der verfügbaren Slots. Diese Werte gelten für die Verarbeitung mit einer optimalen Konfiguration an Funktionseinheiten, welche entsprechend ebenso berechnet wird. Hieraus lassen sich folgende Schlussfolgerungen ziehen.

7.3.1 Relative Beschleunigung durch Hinzunahme von Funktionseinheiten

Die Abbildungen 7.2 und 7.3 zeigen die Abarbeitungsdauer ausgewählter Benchmarks auf der Zielarchitektur. Die entsprechenden Graphen für alle Benchmarks sind in Anhang A.3 zu finden.
Die Dauer für S Slots $T_S(P)$ ist als Speedup $Speedup_S$ für den jeweiligen Benchmark dargestellt. Im Gegensatz zum üblichen Speedup-Begriff wird hier die Dauer allerdings ins Verhältnis zur Abarbeitungsdauer $T_{min-slots}(P)$ auf einem Prozessor mit minimaler Anzahl an Funktionseinheiten und nicht zur Dauer der rein sequentiellen Abarbeitung gesetzt. Es gilt also:

$$Speedup_S = \frac{T_{min-slots}(P)}{T_S(P)} \tag{7.1}$$

Der Referenzwert ist hier dementsprechend die Zeit, welche der Zielprozessor für die Berechnung des Benchmarks mit je einer Funktionseinheit der benutzten Typen benötigt. Sie erhält somit für jeden Benchmark den Wert eins. Ob dieser Wert hier der Slot-Anzahl drei oder vier zugeordnet wird, hängt davon ab, ob drei oder vier Funktionseinheitentypen im Trace benötigt werden. Am häufigste wird ein Funktionseinheitentyp dann nicht benötigt, wenn der ausgeführten Benchmark ausschließlich ganzzahlige Berechnungen durchführt und somit keine Fließkommaeinheit benötigt.
Zur besseren Lesbarkeit sind die diskreten Ergebnisse der Abschätzungen in den folgenden Abschnitten als kontinuierliche Graphen dargestellt.

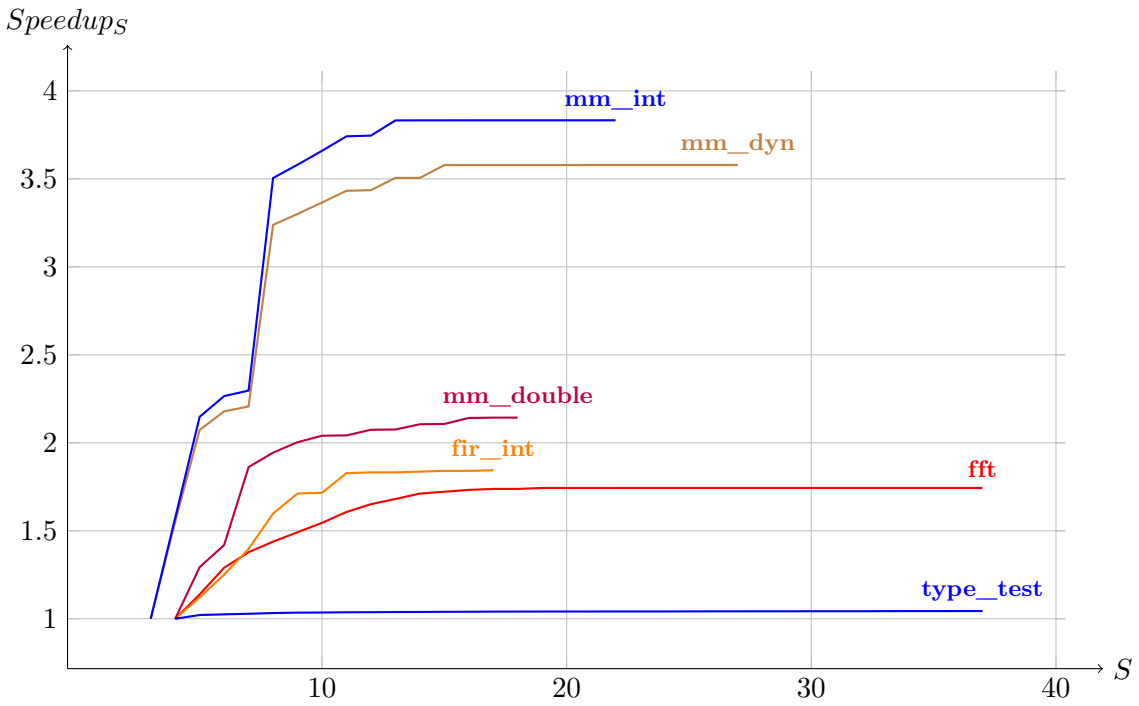


Abbildung 7.2: Speedup bei erhöhter Anzahl an Slots für die Trimaran-Simple-Benchmarks

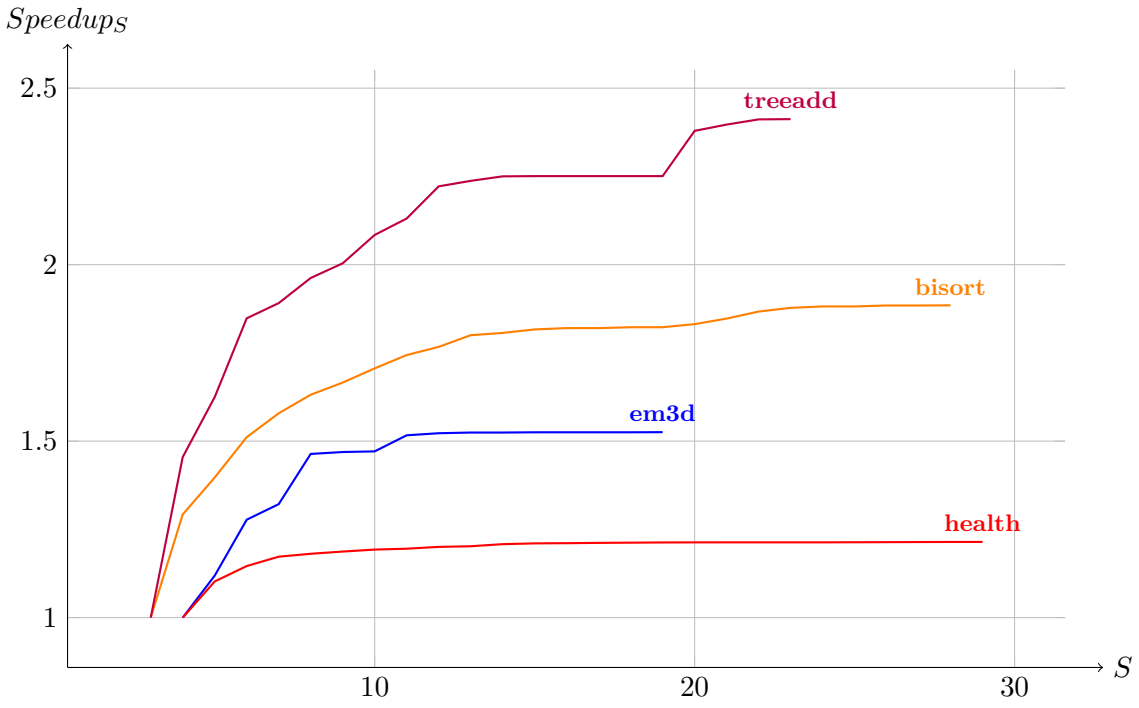


Abbildung 7.3: Speedup bei erhöhter Anzahl an Slots für die Olden-Benchmarks

7.3.1.1 Interpretation der Ergebnisgraphen

An den Graphen lässt sich erkennen, dass, wie zu erwarten war, die Steigerung der Verarbeitungsleistung stark vom verwendeten Benchmark abhängt. Dies ist durch die unterschiedliche Parallelisierbarkeit der Testprogramme durch den Compiler zu erklären. Programme mit wenigen identifizierten nebenläufigen Befehlen und somit geringer Bündelgröße profitieren entsprechend wenig von der Hinzunahme von Funktionseinheiten.
Bei den gut parallelisierbaren Benchmarks, wie beispielsweise den Matrizenmultiplikations-Benchmarks, werden jedoch bereits durch das Hinzunehmen nur weniger Funktionseinheiten beträchtliche Steigerungen der Verarbeitungsleistung erzielt.

Größere Sprünge im Speedup sind meist durch häufig auftretende identische Zeilen im Trace zu erklären. Diese entstehen üblicherweise durch Schleifen im Programm. Eine einzige zusätzliche Funktionseinheit kann es hier in manchen Fällen ermöglichen, dass sich die Verarbeitungszeit dieser Zeile halbiert. Diesen Fall veranschaulicht folgendes Beispiel:

Beispielhaftes häufiges Bündel	$B = (1, 1, 2, 2)$	
Verarbeitungsdauer mit fünf Slots	Zwei Zyklen	$C = (1, 1, 2, 1)$
Verarbeitungsdauer mit sechs Slots	Eins Zyklus	$C = (1, 1, 2, 2)$

Das im Beispiel angegebene häufige Bündel besteht aus sechs Einzelbefehlen. Jede Konfiguration mit fünf Slots benötigt also mindestens zwei Zyklen, um es abzuarbeiten. Stehen sechs Slots zur Verfügung, kann der Prozessor so konfiguriert werden, dass das Bündel nach einem Zyklus fertig abgearbeitet ist.

7.3.1.2 Rentable Anzahl an Slots

Wird also das in dieser Arbeit vorgeschlagene Verfahren mit dem genutzten Paar aus Compiler und Benchmarks angewandt, sind Prozessoren mit variabler Funktionseinheitenanzahl zwischen vier und ca. fünfzehn Slots sinnvoll. Eine größere Anzahl an Slots ermöglicht keinen signifikanten Leistungsgewinn mehr.

7.3.1.3 Maximaler Leistungsgewinn

Bei den Matrix-Multiplikations-Benchmarks mm_dyn und mm_int erreicht das Verfahren bereits durch Verdoppelung der Anzahl an Einheiten mehr als die doppelte, bei Verdreifachung sogar über die dreifache Leistung.
Diese Werte erscheinen auf den ersten Blick als fast schon unrealistisch. Das Beispiel aus Abschnitt 7.3.1.1, welches eine mögliche Halbierung der Rechenzeit durch Hinzunahme nur einer Einheit zeigt, erklärt jedoch dieses Verhalten.

7.3.1.4 Vergleich mit statistischen Eingabewerten

Ist die Beschleunigung des Prozessors durch Hinzunahme von Funktionseinheiten, wenn deren Anzahl gering ist, auch beträchtlich, so lässt sie für hohe Anzahlen stark nach. Theoretisch ist eine erheblich höhere Rechenleistungssteigerung denkbar, als durch die benutzten Traces erzielt wird. Vor allem für Prozessoren mit einer Funktionseinheitenzahl über 15 wird durch die beschränkte Parallelisierung der Benchmarks nur wenig Nutzen aus zusätzlichen Hardware-Ressourcen gewonnen.
Um trotzdem Aussagen über den möglichen Leistungsgewinn bei hohem Parallelisierungsgrad der Software treffen zu können, werden statistische Werte als Eingaben benutzt. Deren Aufbau ist in Abschnitt 7.1.5 beschrieben. Der Speedup für fünf ausgewählte Testläufe ist in Abbildung 7.4 dargestellt. Weitere Ergebnisse hierzu sind in Anhang A.2 und A.3 zu finden.

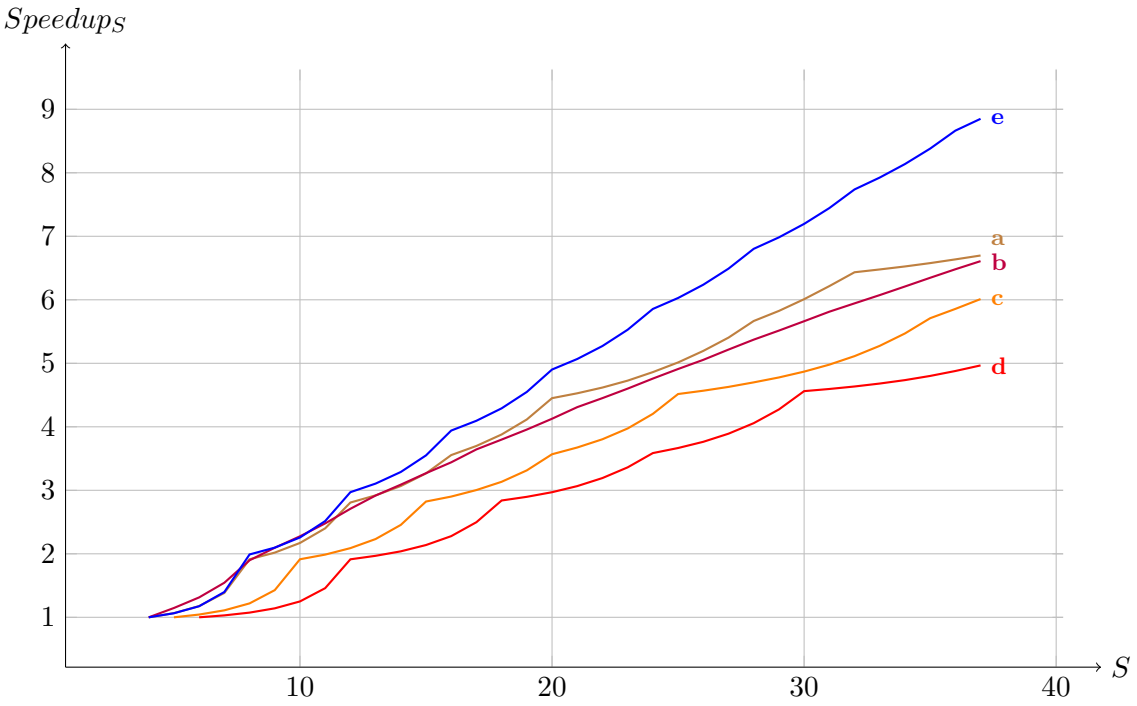


Abbildung 7.4: Speedup mit randomisierten Eingabedaten
a: 4 FU-Typen, max. 16 Befehle pro Typ parallel, gleichverteilt
b: 4 FU-Typen, max. 16 Befehle pro Typ parallel, normalverteilt
c: 5 FU-Typen, max. 16 Befehle pro Typ parallel, gleichverteilt
d: 6 FU-Typen, max. 16 Befehle pro Typ parallel, gleichverteilt
e: 4 FU-Typen, max. 128 Befehle pro Typ parallel, gleichverteilt

Wie zu erwarten war, lässt sich die Grenze des Speedups durch höhere Parallelität der Ein-

gaben hinausschieben. Am Vergleich zwischen den Graphen a und e aus Abbildung 7.4 lässt sich erkennen, dass stärker parallelisierte Programme auch bei gleicher Slot-Anzahl mehr von der Hinzunahme von Slots profitieren. Die Unterschiede zwischen normal- und gleichverteilten Eingabewerten unterscheiden sich dabei recht wenig.
Der Speedup bei Architekturen sinkt mit der Anzahl an Funktionseinheitentypen. Dies liegt jedoch am höheren Referenzwert der Beschleunigung, also der minimalen Anzahl an nötigen Funktionseinheiten. Die reale Ausführzeit eines Programms leidet nicht unter der Erhöhung der Typenzahl.
Auffällig ist zudem, dass sich bei gleichverteilten Eingabewerten ein leichter Anstieg im Speedup beobachten lässt, wenn die Slot-Anzahl sich einem ganzzahligen Vielfachen der Anzahl an FU-Typen nähert. Bei normalverteilten Eingaben ist dieser Effekt schwächer, bei den realen Benchmarks nicht mehr zu beobachten.

7.3.2 Abhängigkeit optimaler Konfigurationen vom Benchmark

Eine Anpassung des Prozessors an die zu erfüllende Aufgabe lohnt nur dann, wenn sich die optimalen Konfigurationen für die verschiedenen Benchmarks stark genug voneinander unterscheiden. Auch hierzu werden die in Abschnitt 7.1.3 beschriebenen Simple- und Olden-Benchmarks benutzt. Exemplarisch werden die Konfigurationen mit acht und zwölf Slots betrachtet, da in diesem Bereich bereits ein hoher Grad an Instruktions-Level-Parallelität ausgenutzt wird und trotzdem für die meisten Benchmarks noch ein hoher Speedup zu beobachten ist. Tabelle 7.6 zeigt die optimalen Konfigurationen für die entsprechenden Benchmarks.

7.4 Nutzen der Selbstoptimierung

Wie in Kapitel 5.2 beschrieben, haben Laufzeitrekonfigurierbare EPIC Soft Core Prozessoren das Potenzial, sich auf das auszuführende Problem zu optimieren, indem sie die Art ihrer Funktionseinheiten zur Laufzeit ändern.
Offensichtlich kann ein solches Ändern der Funktionseinheiten und somit der Prozessorkonfiguration nur dann einen Zugewinn an Rechenleistung ermöglichen, wenn sich zu einem Zeitpunkt der Programmausführung der Bedarf an Funktionseinheiten relevant ändert. Es ist anzunehmen, dass die größte Änderung des Funktionseinheitenbedarfs dann auftritt, wenn zwei Programme sequentiell ausgeführt werden und komplett unterschiedliche Berechnungen durchführen.
Bei der hier eingesetzten Methodik entspricht die sequentielle Berechnung der Laufzeit zweier Traces einem solchen Programmwechsel.

Tabelle 7.6: Optimale Konfigurationen in Abhängigkeit von Benchmark und Slot-Anzahl

Benchmark	Optimales C mit 8 Slots				Optimales C mit 12 Slots			
	B	F	I	M	B	F	I	M
bmm	1	1	4	2	2	1	6	3
dag	1		4	3	1		8	3
fft	1	2	3	2	2	2	4	4
fir	2	1	2	3	3	1	3	5
fir_int	2	1	2	3	2	1	5	4
mm	1	1	4	2	3	1	4	4
mm_double	1	1	3	3	1	1	6	4
mm_dyn	1		5	2	3		6	3
mm_int	1		5	2	2		7	3
nested	2	1	3	2	2	1	5	4
olden_bisort	1		3	4	2		3	7
olden_em3d	1	1	3	3	2	1	6	3
olden_health	2	1	2	3	2	2	2	6
olden_mst	2		3	3	3		3	6
olden_perimeter	2		2	4	2		4	6
olden_treeadd	2		2	4	2		2	8
olden_tsp	1	2	1	4	2	2	2	6
paraffins	2		4	2	2		7	3
sqrt	1	1	2	4	2	1	4	5
strcpy	1		4	3	2		5	5
switch_test	3		2	3	6		1	5
type_test	2	1	2	3	3	1	3	5
wave	1	1	3	3	2	1	5	4

7.4.1 Potential zur Optimierung zwischen zwei Benchmarks

Zur Auswahl zweier Traces, anhand welcher der Nutzen der Selbstoptimierung quantifiziert werden kann, wird ein Maß, das Potential P, definiert. Es gibt an, wie viel Prozent der Zyklen durch Änderung der Prozessorkonfiguration zwischen zwei sequentiell ausgeführten Traces maximal eingespart werden können. Somit beschreibt es die maximal erreichbare Optimierung.
Um dieses Maß zu bestimmen, wird die Laufzeit T der statischen Ausführung beider sequentieller Benchmarks $BM1$ und $BM2$ mit der optimalen Konfiguration C_{BM1} des zuerst ausgeführten Benchmarks berechnet. Von dieser Zeit wird die Laufzeit, welche beide Benchmarks mit den jeweils optimalen Konfigurationen C_{BM1} und C_{BM2} benötigen würden, subtrahiert. Es ergibt sich die maximale Einsparung an Zyklen, welche bei Rekonfiguration beliebig vieler Funktionseinheiten des Prozessors erreicht werden kann. Um die prozentual erreichbare Verbesserung darzustellen, wird diese Differenz mit der Laufzeit der statischen Berechnung ins Verhältnis gesetzt.

$$\begin{aligned} P &= \frac{\left(T_{C_{BM1}}(BM1) + T_{C_{BM1}}(BM2)\right) - \left(T_{C_{BM1}}(BM1) + T_{C_{BM2}}(BM2)\right)}{T_{C_{BM1}}(BM1) + T_{C_{BM1}}(BM2)} = \\ &= \frac{T_{C_{BM1}}(BM2) - T_{C_{BM2}}(BM2)}{T_{C_{BM1}}(BM1) + T_{C_{BM1}}(BM2)} \end{aligned} \tag{7.2}$$

Die Potentiale zwischen den in den Tabellen 7.1 und 7.2 aufgelisteten Benchmarks sind in Anhang B zu finden. Es sind sowohl die absolut einsparbaren als auch die prozentualen Zyklen der Benchmarkfolgen angegeben.

7.4.2 Maximaler realer Nutzen der Selbstoptimierung

Um den Nutzen der Selbstoptimierung zu bestimmen, muss die Dauer, welche die Rekonfiguration einer Funktionseinheit benötigt, mit in die Berechnung der Laufzeit einbezogen werden. Da die Funktionseinheiten von Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren partiell rekonfiguriert werden, arbeitet der Prozessor für die Dauer der Rekonfiguration mit den verbleibenden Funktionseinheiten weiter. Somit ist die Rechenleistung während der Rekonfiguration reduziert.
Eine Anpassung der Prozessorkonfiguration kann ab dem Zeitpunkt als nützlich angesehen werden, ab dem die während der Rekonfiguration verlorenen Zyklen durch die schnellere Berechnung mit der optimierten Konfiguration wieder eingespart wurden.
Im Folgenden wird dieser Nutzen für ausgewählte Benchmarks quantifiziert.

7.4.2.1 Anzahl der veränderlichen Slots

Betrachtet wird hier nur die Änderung einer Funktionseinheit. Da die Rekonfiguration auf realer Hardware immer sequentiell durchgeführt wird, ist es nur möglich, eine FU zu einem Zeitpunkt zu ändern. Sollen mehrere Einheiten geändert werden, entspricht dies also der Änderung von einer Einheit nach der anderen. Das Verfahren ist somit für den Austausch mehrerer FUs identisch, lediglich die Benchmarks müssen zu den Rekonfigurationszeitpunkten zerteilt werden.

7.4.2.2 Auswahl der Sequenz an Benchmarks

Anhand der Daten aus Abschnitt 7.4.1 ist für die Berechnung des Nutzens der Selbstoptimierung ein Paar von Benchmarks zu wählen, bei welchen ein Geschwindigkeitsgewinn durch Änderung der Funktionseinheitenzahl theoretisch möglich ist. Folgende Paare werden im Folgenden dementsprechend genauer untersucht:

- fir → fft, als Benchmark-Folge mit sehr hohem absoluten Potential zur Selbstoptimierung
- mm_double → fir_int, als Benchmark-Folge mit durchschnittlichem Potential zur Selbstoptimierung
- olden_tsp → bmm, als Benchmark-Folge mit sehr hohem prozentualen Potential zur Selbstoptimierung
- sqrt → olden_tsp, als Benchmark-Folge, bei der das Potential von der Reihenfolge der Ausführung der beiden Benchmarks annähernd unabhängig ist.

7.4.2.3 Berechnung des maximalen Nutzens

Um den Nutzen der Selbstoptimierung zu quantifizieren, wird die Laufzeit des ersten Benchmark-Traces wie in Abschnitt 7.2 berechnet. Die hieraus resultierende, für den ersten Trace optimale Konfiguration, dient als Grundlage zur Berechnung der Laufzeit des zweiten Traces.
Zur Bestimmung des maximalen Nutzens wird die Laufzeit des zweiten Traces für den Austausch jedes Funktionseinheitentyps durch jeden anderen Funktionseinheitentyp bestimmt. Die minimale Laufzeit dieses exhaustiven Verfahrens entspricht somit der optimalen Rekonfiguration einer Funktionseinheit.
Die Berechnung der Laufzeit eines Tausches der Funktionseinheiten geschieht dabei durch Ausführung des Programms für die Dauer der Rekonfiguration mit der um eine FU reduzierten Konfiguration. Die Laufzeit des danach noch nicht ausgeführten Teils des Traces wird dann mit einer wieder um eine Funktionseinheit eines anderen Typs erweiterten Konfiguration bestimmt. Dieses Verfahren entspricht der Berechnung von Gleichung 6.19 aus Abschnitt 6.4.1.
Für die ausgewählten Traces sind die Ergebnisse dieser Berechnungen in Abbildung 7.5 dargestellt. Die abgebildeten Graphen zeigen jeweils die Dauer der Ausführung der Benchmarks

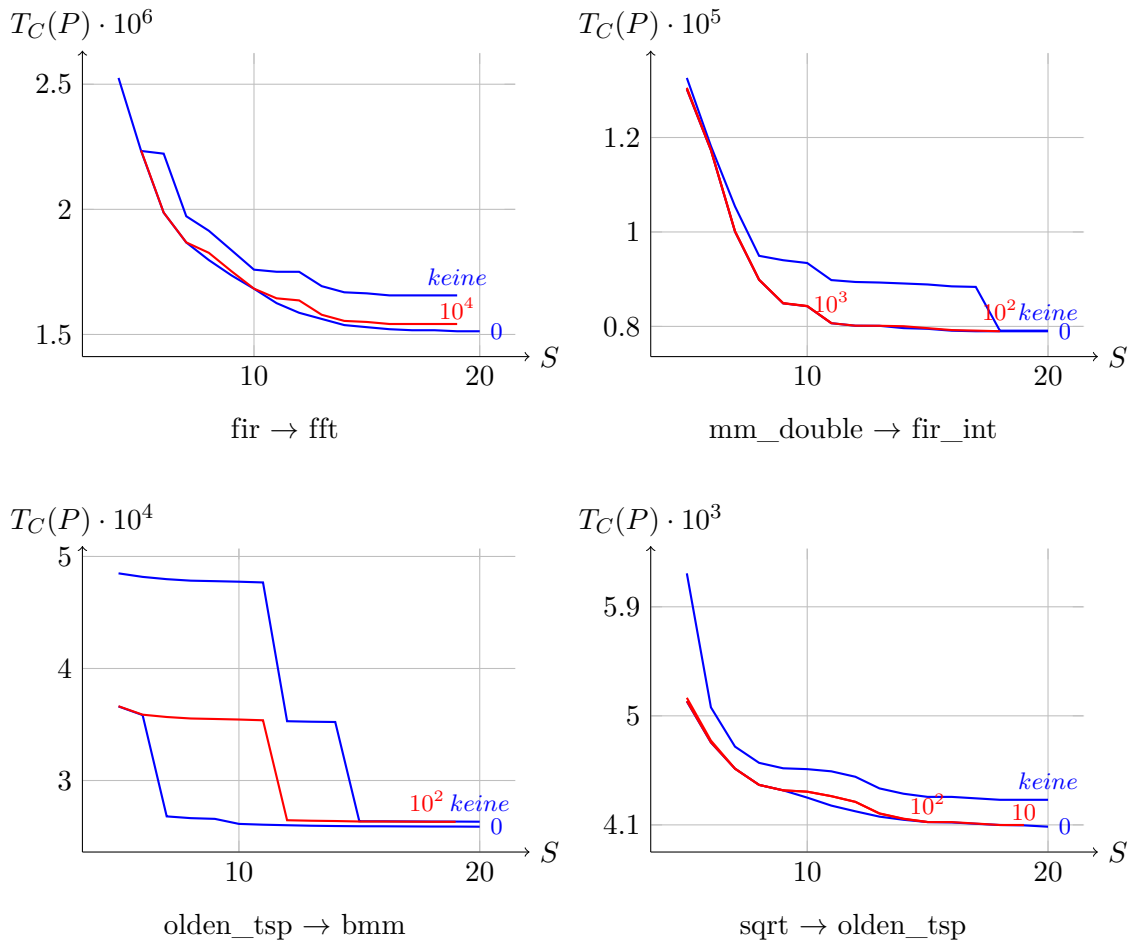


Abbildung 7.5: Rekonfiguration einer Funktionseinheit mit variabler Slot-Anzahl und Rekonfigurationsdauer

ohne Rekonfiguration mit der für den ersten Trace optimalen Konfiguration (Beschriftung: *keine*), mit Rekonfiguration einer Einheit ohne Kosten (Beschriftung: 0) und mit Kosten (Beschriftung: Höhe der Kosten in der Form 10^x). Die diskreten Ergebnisse der Berechnung sind wiederum als kontinuierliche Graphen dargestellt.
Bei der Berechnung der Daten wurden diese Kosten, also die Rekonfigurationszeit, in Zehnerpotenzen gewählt. Sobald diese Zeit länger ist als die Laufzeit des zweiten Benchmarks mit der Ursprungs-Konfiguration, kann eine Rekonfiguration nicht mehr sinnvoll sein. Aus diesem Grunde wurden in diesen Beispielen keine höheren Rekonfigurationskosten angenommen.
Es lässt sich erkennen, dass sich für die gewählten Folgen an Traces, trotz Rekonfigurationszeiten von bis zu zehntausend Zyklen, Ergebnisse erzielen lassen, welche sehr nahe an der minimal erreichbaren Laufzeit bei Rekonfiguration einer Funktionseinheit liegen. Wie zu erwarten war, lässt sich der Prozessor ab der Slot-Anzahl, mit der bereits mit der Ursprungskonfiguration alle Bündel in einem Zyklus abgearbeitet werden, nicht mehr optimieren.
Folgende Faktoren sind entsprechend für einen rentablen Optimierungsschritt nötig:

- Hohes Potential zur Optimierung
- Geringe Rekonfigurationszeit
- Lange Restlaufzeit des Programms, viel länger als Rekonfigurationszeit
- Geschickte Wahl der Funktionseinheiten zum Tausch

7.4.2.4 Berechnung der tatsächlichen Kosten einer Rekonfiguration

Da der Prozessor während einer Rekonfiguration weiterarbeitet, sind die Kosten einer Rekonfiguration nicht identisch mit der Rekonfigurationszeit. Die realen Kosten entsprechen der Anzahl an Zyklen, welche der Prozessor zur Berechnung benötigt hätte, wenn die Rekonfiguration keinen Zeitbedarf hätte.
Diese Kosten lassen sich aus den in Abschnitt 7.4.2.3 berechneten Laufzeiten extrahieren, indem der Zeitbedarf des Programmablaufs mit Rekonfigurationskosten von dem ohne Kosten subtrahiert wird.
Die realen Kosten der Rekonfiguration einer Funktionseinheit sind für die entsprechenden Benchmarks in Abhängigkeit von Slot-Anzahl und Rekonfigurationsdauer in Abbildung 7.6 dargestellt. In diesen Graphen sind die realen Kosten der Rekonfiguration in Zyklen bei variabler Slot-Anzahl aufgetragen. Die Beschriftung der Kurven entspricht der angenommenen Rekonfigurationsdauer.
Diese Grafik zeigt, dass für die gewählten Folgen von Traces die realen Kosten meist knapp unter 10% der Rekonfigurationsdauer liegen. Die Anzahl der Slots spielt hierbei nur eine untergeordnete Rolle. Ist also zu erwarten, dass ein Selbstoptimierungsschritt des Prozessors einen Zeitgewinn von mehr als 10% der Rekonfigurationszeit bringt, so kann dieser typischerweise als rentabel angenommen werden.

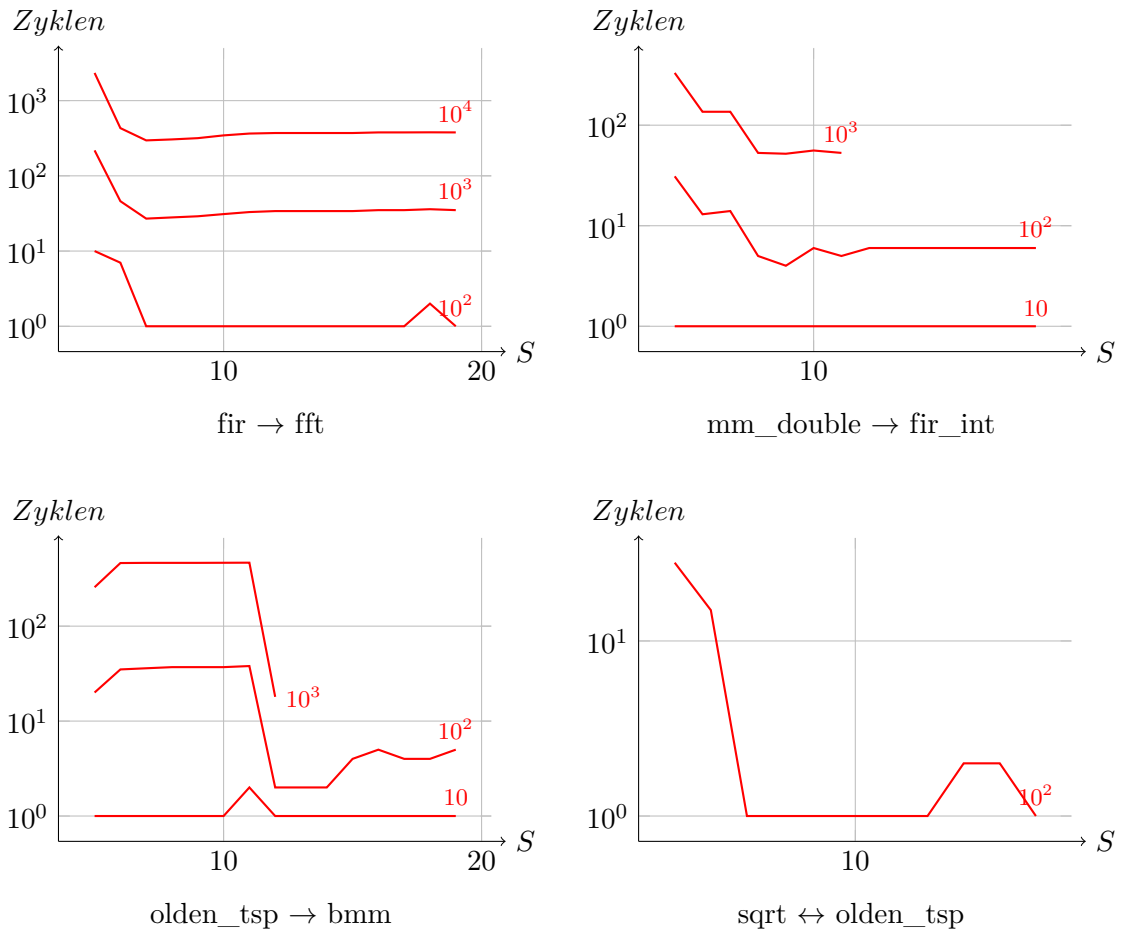


Abbildung 7.6: Tatsächliche Kosten der Rekonfiguration einer Funktionseinheit mit variabler Slot-Anzahl und Rekonfigurationsdauer

7.5 Abschätzung für reale programmierbare Hardware

Die maximale Rekonfigurationsdauer bestimmt den Nutzen der Selbstoptimierung von Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren. Im realen Einsatz hängt diese Dauer sowohl von der Rekonfigurationsbandbreite der zugrundeliegenden Hardware als auch der Größe der Rekonfigurationsdaten für eine FU ab. Die Rekonfigurationsbandbreite entspricht dem Rekonfigurationstakt multipliziert mit der Breite des Konfigurations-Ports:

$$\text{Rekonfbandbreite} = \text{Rekonftakt} \cdot \text{PortBreite}$$

Um die Dauer einer Rekonfiguration zu ermitteln, wird die Größe des Bitstreams zur Programmierung einer Funktionseinheit durch diese Bandbreite dividiert. Im realen Fall existiert ein Protokoll-Overhead bei der Übertragung der Dateien und ein Overhead zur Initialisierung der Rekonfiguration.

$$\text{Rekonfdauer} = \frac{\text{BitstreamGröße} \cdot \text{Overhead}}{\text{Rekonfbandbreite}} + \text{Initialisierungszeit}$$

Diese für die Berechnung der Rekonfigurationsdauer relevanten Werte sind aus mehreren Gründen sehr variabel. Der unterstützte Takt zur Rekonfiguration kann, abhängig von Hardware, Protokoll und Implementierung, um etwa den Faktor 10^2 schwanken. Ähnliches gilt für die Port-Breite.
Zudem ist die Bestimmung der Größe eines Bitstreams von vielen Faktoren abhängig. Diese sind in erster Linie die Bit-Breite der Architektur sowie der Funktionsumfang der einzelnen Funktionseinheit. Auch die Optimierung, sowohl bei der Implementierung als auch bei der Synthese, der Funktionseinheit auf Platz oder Geschwindigkeit hat erheblichen Einfluss auf die Größe des Bitstreams. Somit ist auch hier von einer Schwankung um einen Faktor von ca. 10^2 auszugehen.
Um dennoch eine tendenzielle Abschätzung zu ermöglichen, werden folgende Werte angenommen. Diese befinden sind für aktuelle FPGAs in der realistischen Größenordnung:

Rekonfigurationstakt	66 MHz
Rekonfigurationsbandbreite	32 Bit
Bitstream-Größe	100 kByte
Overhead	2
Initialisierungszeit	Mikrosekundenbereich

Mit diesen Werten ergibt sich eine Rekonfigurationsdauer im Bereich von 0,4 Millisekunden. Ein ebenso realistischer Takt des in rekonfigurierbarer Logik implementierten Prozessors liegt bei etwa 200 MHz. Die Rekonfigurationsdauer liegt dementsprechend im Bereich von ca. 10^5 Zyklen. Ob eine Selbstoptimierung rentabel ist, liegt dann wiederum am Potential zur Selbstoptimierung sowie an der Dauer, für welche das Programm mit der veränderten Konfiguration laufen wird.

7.6 Zusammenfassung

Als Eingabedaten für die Abschätzung der Leistungsfähigkeit von Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren dienen Programm-Traces, welche durch die Kompilierung und Simulation von Benchmarks erzeugt werden. Zudem wird das Verhaltens des Prozessors für andere Eingabedaten anhand randomisierter Programm-Traces analysiert.
Um optimal kurze Laufzeiten des Prozessors zu bestimmen, müssen alle möglichen Konfigurationen getestet werden. Dies wird ermöglicht, indem der Aufwand der Analyse durch mehrere Methoden, wie beispielsweise der Kumulation von Bündeln gleichen Ressourcen-Bedarfs, so weit wie möglich gesenkt wird.
Die Analyse des Verhaltens des Prozessors ist stark vom genutzten Compiler und Benchmark abhängig. Meist ist die Hinzunahme von ungenutzten Ressourcen jedoch sehr sinnvoll.
Die Leistung des Prozessors kann mit den genutzten Traces um annähernd den Faktor vier im Vergleich zur Ausführung auf einem Prozessor mit minimaler Konfiguration gesteigert werden. Ab einem Bereich von 15 bis 20 Funktionseinheiten lässt der Nutzen jedoch rapide nach. Der Vergleich mit randomisierten Traces zeigt, dass dies ausschließlich im Eingabeprogramm begründet liegt.
Ob die Optimierung eines Prozessors auf ein auszuführendes Programm sinnvoll ist, hängt von vielen Faktoren ab, wie beispielsweise von der Dauer einer Rekonfiguration. Bei den genutzten Benchmarks ist durch die Rekonfiguration von nur einer Einheit zur Optimierung bereits ein erheblicher Performance-Gewinn, also eine effektiver Steigerung des Gesamtdurchsatzes des Prozesors, möglich.
Dadurch, dass ein Prozessor während der Rekonfiguration mit der verminderten Konfiguration weiterarbeiten kann, entsprechen die realen Kosten einer Rekonfiguration typischerweise nur ca. 10% der Rekonfigurationszeit. Diese Zeit ist allerdings für reale Bausteine sehr stark variierend. Als grober Richtwert kann von ca. 10^5 Prozessor-Zyklen für eine Rekonfiguration ausgegangen werden.
Die Analyse in diesem Kapitel hat ergeben, dass für geeignete Programme und geeignete Hardware der Einsatz von Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren einen großen Gewinn an Performance und Variabilität bringen kann. Um diesen Einsatz zu ermöglichen, werden im folgenden Kapitel die größten technischen Herausforderungen bei der Implementierung solcher Prozessoren analysiert.

8 Analyse der Umsetzbarkeit

Um die Hard- und Software von Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren zu realisieren, sind einige nicht triviale Hindernisse zu überwinden. Im folgenden Kapitel werden diese aufgezeigt und mögliche Lösungen gegeben. Um die tatsächliche Lösbarkeit der Probleme nachzuweisen, werden zusätzlich prototypische Implementierungen vorgestellt, in welchen einzelne der vorgeschlagenen Lösungswege angewandt werden.

8.1 Überblick über die Herausforderungen

Zusätzlich zu den Aufgaben, welche nicht von der Prozessorklasse abhängen, bestehen speziell für die Umsetzung von EPIC-Prozessoren zusätzliche Schwierigkeiten. Im Vergleich wiederum zu klassischen EPIC-Prozessoren entstehen bei der Implementierung von Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren nochmals Herausforderungen, welche gelöst werden müssen. Ein Überblick über diese ist in Abbildung 8.1 gegeben. Dort sind bereits die speziellen Anforderungen für Laufzeitrekonfigurierbare EPIC Soft Core Prozessoren auf der rechten Seite zusammengefasst und hervorgehoben. Die Anforderungen an klassische EPIC-Prozessoren demgegenüber sind auf der linken Seite der Abbildung zu finden. Die nötigen Konzepte zur Implementierung sind in der Abbildung grob in die drei Bereiche Hardware, Organisation und Architektur des Systems untergliedert, wobei der Abstraktionsgrad von unten nach oben ansteigt.

8.1.1 Herausforderungen klassischer EPIC-Prozessoren

Die Anforderungen an hochintegrierte Halbleitertechnologie bezüglich Entwurf und Herstellung von EPIC-Prozessoren sind mit denen anderer Prozessorklassen noch sehr verwandt. Durch die nicht nötige Parallelisierung des Programm-Codes zur Laufzeit, können bei EPIC-Prozessoren größere Caches eingesetzt werden, deren Umsetzung, Anbindung und Organisation neue Möglichkeiten, jedoch auch Herausforderungen aufwerfen.
Die Art, die Anzahl und der Aufbau der Funktionseinheiten und Funktionseinheitentypen müssen festgelegt werden. Da die EPIC-Architektur grundsätzlich Prediction und Predication unterstützt, ist dies sowohl in der Organisation als auch bei der Architektur des Prozessors zu berücksichtigen. Wie bereits in den vorhergehenden Kapiteln erwähnt, übernimmt der Compiler bei EPIC-Prozessoren weit mehr Aufgaben, als dies bei anderen Prozessorklassen der Fall ist. Entsprechend sind spezielle Compiler-Technologien, welche den EPIC-Befehlssatz

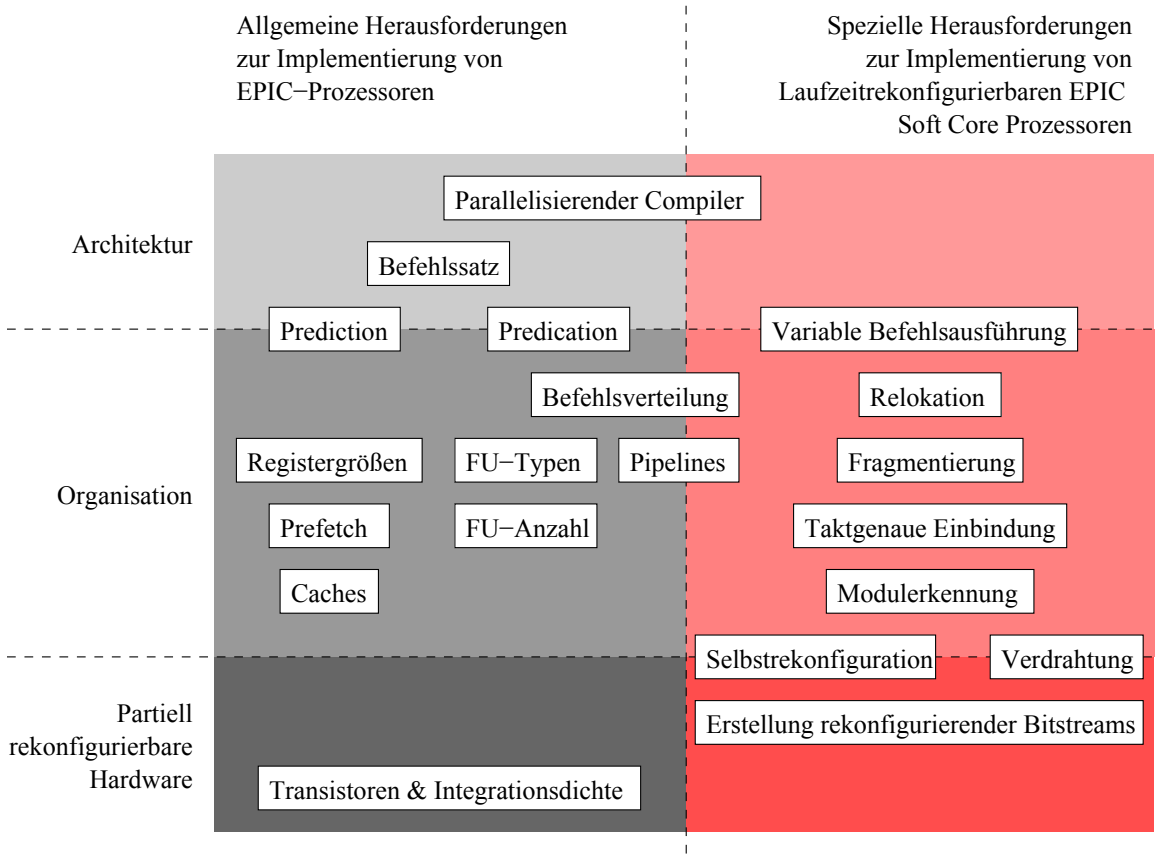


Abbildung 8.1: Herausforderungen bei der Implementierung von statischen EPIC-Prozessoren und Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren

unterstützen, erforderlich.
Zusätzliche Anforderungen zu denen der allgemeinen EPIC-Prozessoren, stellt, durch ihre Fähigkeit zur Rekonfiguration, die in dieser Arbeit beschriebene Prozessorklasse an die Pipelines der Funktionseinheiten und die Befehlsverteilung. Zudem ist es sinnvoll, den eingesetzten Compiler auf eine variable Anzahl an Funktionseinheiten zu optimieren.

8.1.2 Neue Herausforderungen Laufzeitrekonfigurierbarer EPIC Soft Core Prozessoren

Neben diesen Herausforderungen bestehen weitere zu beantwortende Fragen, welche sich ausschließlich für Laufzeitrekonfigurierbare EPIC Soft Core Prozessoren stellen. Zu ihnen gehört die Erstellung der rekonfigurierenden Bitstreams für die Funktionseinheiten. Diese Module müssen ohne Einwirkung von außen konfiguriert, erkannt, taktgenau eingebunden und jederzeit entfernt werden können. In manchen Fällen kann Modultausch zu Fragmentierung der programmierbaren Logik führen. Um das System variabel zu halten, sollte die Möglichkeit geboten werden, den Ort, an dem ein Modul arbeitet, zu wechseln. An der Grenze zwischen Architektur und Organisation der Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren, ist die Herausforderung der variablen Ausführung der Befehlsbündel anzusiedeln.
Da für statische EPIC-Prozessoren sämtliche Herausforderungen bereits mehr oder weniger effizient gelöst und ihre Umsetzbarkeit durch die Implementierung der Itanium-Prozessoren (siehe Abschnitt 3.6) bereits nachgewiesen wurde, befassen sich die nachfolgenden Kapitel nur mit den zusätzlichen Problemen, welche bei der Implementierung von Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren auftreten.
Die Anforderungen an die weniger abstrakten Ebenen folgen aus den Anforderungen der höheren Ebenen. Dementsprechend werden in Abschnitt 8.2 zuerst die architekturellen Herausforderungen, dann in Abschnitt 8.3 die der Organisation und in Abschnitt 8.4 die der rekonfigurierenden Hardware behandelt. Die einzelnen Herausforderungen werden dort im Detail erläutert, die Umsetzbarkeit ihrer Lösungen jeweils durch prototypische Implementierungen nachgewiesen.

8.2 Architektur

Die hauptsächlichen Herausforderungen der Architektur bei der Implementierung Laufzeitrekonfigurierbarer EPIC Soft Core Prozessoren liegen im Bereich des Compilers, welcher trotz variabler Funktionseinheitenanzahl möglichst optimalen Code liefern soll, sowie bei der variablen Ausführung der Befehlsbündel, deren exakte Abarbeitungsdauer vor der Laufzeit nicht feststeht. Beides wird im Folgenden analysiert. Entsprechende Beispielimplementierungen werden dazu herangezogen. Die Architekturebene ist im Bildausschnitt 8.2 dargestellt.

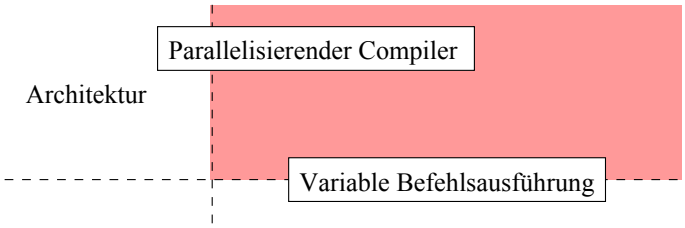


Abbildung 8.2: Herausforderungen der Architektur Laufzeitrekonfigurierbarer EPIC Soft Core Prozessoren, Ausschnitt aus Abbildung 8.1

8.2.1 EPIC-Code-Generierung

Die Generierung von parallelem Maschinen-Code durch einen Compiler ist eine Aufgabe, welche zu weiten Teilen auch für statische EPIC-Prozessoren besteht. Diese sind jedoch nach wie vor weit davon entfernt, ein wirklich optimales Kompilierungsergebnis zu erzielen (siehe Kapitel 3.3).
Die meisten existierenden EPIC-Compiler zielen auf Intels Itanium-Prozessoren ab. Deswegen optimieren sie die Code-Erstellung auf diese Architektur und auf deren Eigenschaften wie der starren FU- und Registeranzahl.
Mit dem Trimaran-Framework (Kapitel 3.5) existiert jedoch ein Compiler, welcher flexibel genug ist, um Maschinen-Code für Laufzeitrekonfigurierbare EPIC Soft Core Prozessoren zu erzeugen. Schöpft dieser Code zwar die Möglichkeiten bei weitem nicht voll aus, so genügt er doch, um die Vorteile der Prozessoren aufzuzeigen und ihre Einsatzfähigkeit nachzuweisen.
Bereits in Kapitel 7.1.2 wurde das Trimaran-Framework eingesetzt, um EPIC-Befehlsbündel-Traces zu erzeugen. Die Eingabe, welche hier für den Simulator zur Trace-Erstellung diente, ist grundsätzlich bereits für Laufzeitrekonfigurierbare EPIC Soft Core Prozessoren geeignet.

8.2.2 Variable Befehlsausführung

Die wichtigste Eigenart von Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren ist die Fähigkeit, Befehlsbündel mit unterschiedlichen Funktionseinheiten abzuarbeiten. Um dies zu ermöglichen, müssen die Befehle den passenden vorhandenen Verarbeitungseinheiten dynamisch zugeordnet werden. Bevor die Konfiguration des Prozessors bekannt ist, kann keine genaue Aussage über die Abarbeitungsdauer eines einzelnen Bündels getroffen werden.
Soll sich der Prozessor auf seine Aufgabe optimieren, ist es zudem nötig, zu erkennen, welche Funktionseinheit nachkonfiguriert werden soll. Dies kann zum einen durch zur Compile-Zeit erstellte Zusatzinformationen, welche dem Prozessor zur Verfügung gestellt werden, geschehen. Diese sind durch Berechnungs- und Analyseverfahren, wie sie in Kapitel 7 vorgestellt wurden, denkbar.
Eine weitere Möglichkeit ist es, auf geeignete Heuristiken zum Tausch der Funktionseinheiten

zurückzugreifen. Hier können beispielsweise Ersetzungsstrategien zur Anwendung kommen, welche denen von Caches ähneln. Je nach angewandter Strategie ist jeweils zusätzliche Hardware zur Bestimmung der zu entfernenden wie auch der nachzukonfigurierenden FU nötig.

8.2.3 Prototyp: Befehlsverteilung und Ausführungsstatistiken

Um die Möglichkeit zu schaffen, einen Prozessor während der Laufzeit auf seine Aufgabe zu optimieren, wird in [99] ein Hardware-IP erstellt, welches EPIC-Befehlsbündel aus einem Speicher ausliest, um diese dann an Funktionseinheiten weiter zu verteilen. Dabei werden die Befehle nach dem Typ der Funktionseinheit, auf welcher sie ausgeführt werden, klassifiziert. Ein Zähler pro Befehlstyp protokolliert die Ausführung mit.
Durch vom Zeitpunkt der Ausführung der Befehle abhängige Gewichtung der Zähler wird somit im IP eine Statistik darüber aufgebaut, wie häufig eine Art von Funktionseinheit in der letzten Zeit benutzt wurde.
Anhand dieser Statistik lässt sich eine Most-Recently-Used-Ersetzungsstrategie für die Funktionseinheiten umsetzen. Bei dieser werden wenig ausgelastete FUs durch solche ersetzt, deren zugehörige Befehle im vorherigen Programm häufig aufgetreten sind. Geht man davon aus, dass die Befehle der einzelnen Typen des noch auszuführenden Programms ähnlich verteilt sind wie im bisher ausgeführten Programm, so lässt sich mit dieser Strategie eine Optimierung des Prozessors erreichen. Dabei hängt die Kalibrierung der Gewichte für die Statistik maßgeblich von der Rekonfigurationsdauer einer zugehörigen Funktionseinheit ab. Ist diese kurz, so sollen alte Befehle schneller an Relevanz für die Statistik verlieren als bei langer Rekonfigurationsdauer, da dann eine häufigere Änderung der Funktionseinheiten eine optimalere Ausführung verspricht.

8.3 Organisation

Die meisten Herausforderungen bei der Implementierung von Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren sind im Bereich der Organisation des Prozessors zu lösen. Die hier betrachteten Probleme sind im Bildausschnitt 8.3 zusammengefasst. Geeignete prototypische Implementierungen werden wiederum zum Nachweis der Praktikabilität der gegebenen Lösungsansätze herangezogen.

8.3.1 Aktivierung und Deaktivierung von Modulen

Soll ein partiell neu konfiguriertes Modul zusammen mit anderen Modulen eine Aufgabe erfüllen, so ist es nötig, dass dieses, sobald es vollständig konfiguriert ist, erkannt und mit in die Berechnung aufgenommen wird. Ebenso dürfen die aktiven Berechnungen eines Moduls, welches entfernt werden soll, keinen Fehler in der Programmausführung erzeugen.
Ist das System selbstrekonfigurierend, verfügt es über den aktuellen Konfigurationsstatus und

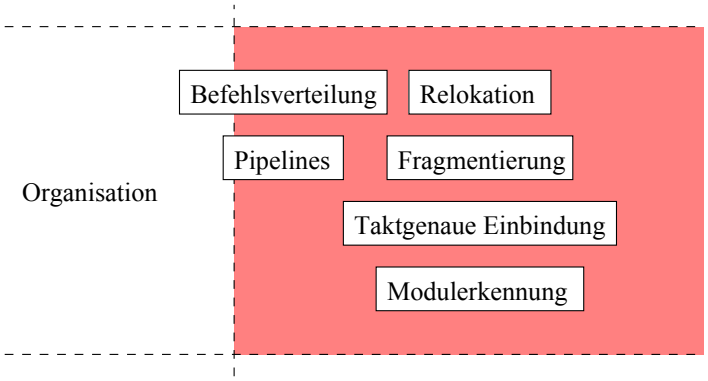


Abbildung 8.3: Herausforderungen an die Organisation Laufzeitrekonfigurierbarer EPIC Soft Core Prozessoren, Ausschnitt aus Abbildung 8.1

kann diesen nutzen, um den Zeitpunkt zum Einbinden des Moduls in die Berechnung sowie den Modultyp zu bestimmen.
Wird die Rekonfiguration von außen durch einen externen Host gesteuert, sind hierzu weitere Mechanismen nötig. Das Erkennen des Modultyps ist durch die Übermittlung einer ID vom Modul an seine Umgebung mit wenig Aufwand realisierbar. Nicht anwendbar ist die naheliegende Lösung, dass PR-Module selbst Informationen über ihren Konfigurationsstatus übermitteln, da sie während des Konfigurationsvorgangs eventuell nur teilweise funktionsfähig sind und somit trotz korrekt übermittelter ID fehlerhafte oder unvollständige Berechnungen durchführen könnten.
Um zu erkennen, dass die Konfiguration des Moduls abgeschlossen ist, gibt es mehrere Möglichkeiten:

8.3.1.1 Überwachung des Konfigurationsprotokolls

Die Konfiguration eines FPGAs geschieht über interne Ports oder externe Pins. Beide können von einem auf dem FPGA implementierten Modul überwacht werden. Durch Parsen des Konfigurationsprotokolls kann dieses feststellen, ob gerade eine Rekonfiguration initiiert wird, im Gange ist oder abgeschlossen wird. Somit kann sie erkennen, welches Modul gerade konfiguriert oder entfernt werden soll.
Der Ressourcenbedarf für dieses Modul ist jedoch relativ hoch. Zudem ist das Konfigurationsprotokoll für viele Bausteine nicht offengelegt, was die Schwierigkeit dieses Verfahrens zusätzlich erhöht.

8.3.1.2 Durchführung eines Built In Self Tests

Eine weitere Möglichkeit ist es, die Module mit einem Built In Self Test (BIST) zu versehen, der so aufgebaut ist, dass dieser erst nach der vollständigen Konfiguration des Moduls

eine eindeutige Signatur ausgibt. Diese Methode hat den Nachteil, dass sich die Entfernung von Modulen nur sehr schwer erkennen lässt. Zudem benötigt ein solcher Test zusätzliche Ressourcen auf dem FPGA. Dies führt zu größeren Bitstreams und somit auch zu längeren Konfigurationszeiten. Falls das Modul während des Selbsttests seine Arbeit noch nicht voll aufnehmen kann, verzögert dieser zusätzlich den Zeitpunkt bis zum Einsatz.

8.3.1.3 Implementierung von Partiell Rekonfigurierbaren Semaphoren

Eine dritte Möglichkeit ist es, zusätzlich zu jedem definierten partiell rekonfigurierbaren Bereich auf dem FPGA einen zusätzlichen minimal kleinen rekonfigurierbaren Bereich zu definieren, welcher ausschließlich den Status des großen Bereiches anzeigt. Da der kleine Bereich, ähnlich wie ein Semaphor, das zugehörige große Modul sperrt und wieder freigibt, wird er als Partiell Rekonfigurierbarer Semaphor (PARS) bezeichnet.
Sobald ein Modul fertig konfiguriert ist, wird sein PARS gesetzt, indem auch er konfiguriert wird. Soll ein Modul entfernt werden, wird zuerst der PARS gelöscht. Durch die stets sequentielle Konfiguration von FPGAs ist sichergestellt, dass, sobald ein PARS konfiguriert wurde, der Konfigurationsvorgang des vorhergehenden Moduls abgeschlossen ist. Für den Fall der Modulentfernung ist garantiert, dass das PR-Modul bis zur Entfernung der PARS voll konfiguriert ist.
Eine einfache Einheit im System kann sämtliche PARS überwachen und enthält somit zu jeder Zeit die Information über den Konfigurationsstatus des Systems.
Eine Möglichkeit, solche PARS umzusetzen, ist es, durch sie eine ID des zugehörigen Moduls zu senden und somit sowohl den Modultypen als auch dessen Konfigurationsstatus anzuzeigen.
In Abbildung 8.4 ist ein Beispiel für die Verwendung von PARS in einem einfachen System mit nur einem rekonfigurierbaren Modul und einem PARS aufgezeigt. Abbildung 8.4a) verdeutlicht das Vorgehen zum Hinzufügen eines Moduls. Hierzu wird zuerst das Modul konfiguriert. Daraufhin erfolgt die Konfiguration der PARS. Sobald dies abgeschlossen ist, wird von der restlichen Logik anhand der PARS der Abschluss des Konfigurationsvorgangs und der Modultyp erkannt und das neue Modul wird in Betrieb genommen.
In Abbildung 8.4b) wird ein Modul entfernt. Hierzu wird zuerst der PARS gelöscht, woraufhin durch deren fehlende Ausgabe erkannt wird, dass das Modul entfernt werden wird. Also wird das Modul aus den Berechnungen ausgeschlossen und die Fläche kann für andere Funktionen freigegeben werden.
In diesem Beispiel werden die beiden hintereinander auf denselben Ressourcen konfigurierten PR-Module für verschiedene Kontexte benutzt. Sie tragen also zur Berechnung unterschiedlicher Aufgaben bei. Denkbar ist beispielsweise, dass sie als Funktionseinheiten zweier unabhängig arbeitender Prozessoren dienen.

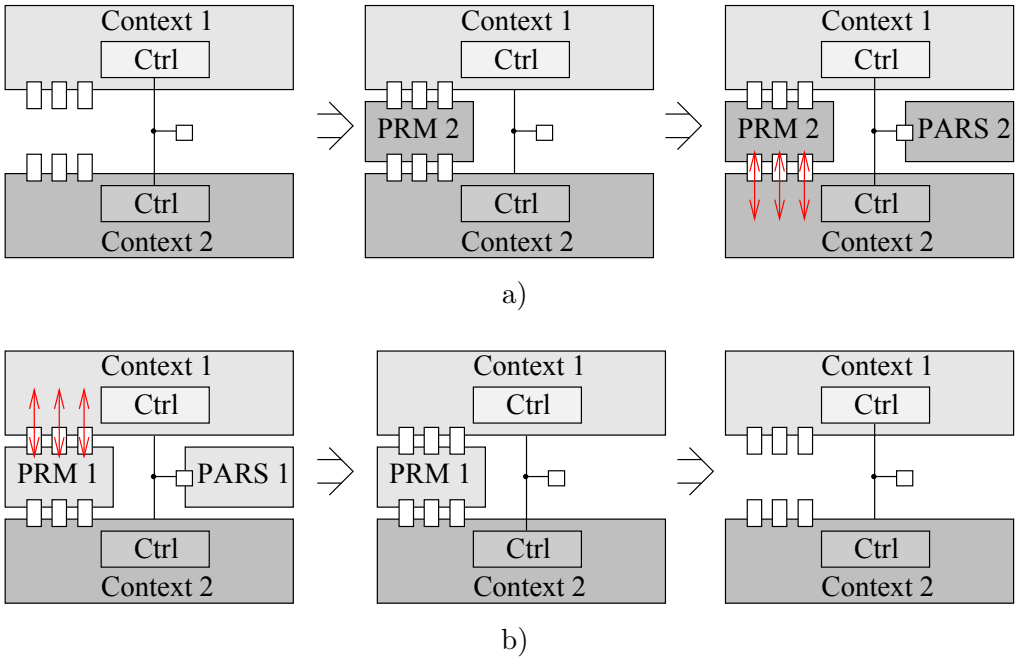


Abbildung 8.4: Einsatz von PARS zur PR-Modulerkennung: a) Hinzufügen eines Moduls b) Entfernen eines Moduls

8.3.2 Taktgenaue Einbindung von Modulen

Um die Korrektheit von Berechnungen sicherzustellen, muss die Einbindung sowie die Entfernung von Funktionseinheiten zu einem auf den Takt genau definierten Zeitpunkt erfolgen. Ab diesem Zeitpunkt muss sichergestellt werden, dass ab dem nächsten Takt Befehle an die Einheit übergeben werden können und sie diese korrekt ausführt. Geschieht dies nicht, kann es passieren, dass Befehle nicht oder doppelt ausgeführt werden. Beides würde meist fehlerhafte Ergebnisse zur Folge haben.
Bei der Einbindung von Modulen kann dies von einer übergeordneten Einheit noch sehr einfach umgesetzt werden. Sobald festgestellt wird, dass die Einheit voll konfiguriert und funktionsfähig ist, kann ein Befehl an sie erteilt werden, welcher dann zuverlässig ausgeführt wird.
Schwieriger gestaltet sich dies beim Entfernen von Modulen, da sich ein zu entfernendes Modul eventuell noch in einer Berechnung befindet, deren Ergebnisse noch nicht feststehen. Zwei Ansätze können dieses Problem lösen:
Zum einen ist es möglich, der zentralen Ressourcen-Verwaltung einige Takte vor der tatsächlichen Entfernung der Funktionseinheit mitzuteilen, dass dies geschehen wird. In dieser Zeit werden der betroffenen Funktionseinheit keine neuen Befehle mehr zugewiesen. Der Zeitraum von der Vorankündigung bis zur Löschung muss lange genug sein, um die Verarbeitung der aktuellen Befehle abzuschließen.

Eine zweite Möglichkeit ist es, den Prozessor darauf auszulegen, dass alle Befehle zurückgerollt werden können. Die zentrale Ressourcen-Verwaltung kann somit den noch nicht endgültig abgearbeiteten Befehl einer anderen Einheit erneut zuweisen. Zudem muss sie sicherstellen, dass der nicht vollständig abgearbeitete Befehl der entfernten Einheit keine Spuren, wie beispielsweise geänderte gemeinsam genutzte Register, hinterlässt.

8.3.3 Pipelining in Funktionseinheiten während der Rekonfiguration

Um die Verarbeitung von einem Befehl pro Taktzyklus und Funktionseinheit zu erreichen ist es gängige Praxis, Pipelining anzuwenden. Dabei wird ein einzelner Befehl in mehreren physikalisch getrennten, aufeinanderfolgenden Verarbeitungsschritten, den Pipelinestufen, ausgeführt. Die Verarbeitungszeit der Stufen beträgt hierbei idealerweise einen Takt. Jede Stufe beginnt dabei mit der Verarbeitung des nächsten Befehls, sobald sie mit der Ausführung ihres aktuellen Befehls fertig ist und die Zwischenergebnisse an die nächste Stufe weitergeleitet hat. Somit sind bei diesem Vorgehen zu jedem Zeitpunkt mehrere Befehle zeitgleich in der Verarbeitung. Die auftretenden Probleme gleichen denen bei der taktgenauen Einbindung von Modulen aus dem vorhergehenden Abschnitt 8.3.2.
Soll gemäß dem Ansatz der Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren eine Funktionseinheit neu zum Prozessor hinzukonfiguriert werden, füllt sich ab dem ersten Befehl, der ihr zur Abarbeitung zugewiesen wird, ihre Pipeline. Eine besondere Behandlung ist in diesem Fall nicht nötig.
Anders gestaltet sich dies für den Fall, dass eine Funktionseinheit vom Prozessor entfernt werden soll. Stellt es sich hier auch etwas komplexer dar, als bei Funktionseinheiten ohne Pipelining, so entsprechen die Lösungsansätze doch genau denen aus Abschnitt 8.3.2. Wie beschrieben, kann die Korrektheit der Berechnungen dadurch sichergestellt werden, dass die begonnenen Befehle, welche in der Pipeline der Einheit sind, noch vor der endgültigen Entfernung des Moduls vollständig abgearbeitet werden.
Es ist ebenso denkbar, die Funktionseinheiten so aufzubauen, dass sie ausschließlich in der letzten Stufe der Pipeline Änderungen am Speicher und an gemeinsam genutzten Registern vornehmen. Bei diesem Ansatz stellt sich jedoch dann das zusätzliche Problem, dass sich in der Pipeline Befehle verschiedener Bündel befinden können. Ist dies der Fall, ist zusätzlicher Aufwand nötig, um die noch nicht vollständig abgearbeiteten Befehle an andere Funktionseinheiten zu verteilen. Bündel mit betroffenen Befehlen müssen bis zur endgültigen Abarbeitung vorgehalten oder erneut geladen werden, was mit zusätzlichem Aufwand an Hardware verbunden ist.

8.3.4 Fragmentierung

Wie auch bei Speichern, die von mehreren Kontexten benutzt werden, so stellt sich auch bei der Rekonfiguration von Hardware das Problem der Fragmentierung, sobald verschiedene

Hardware-Module stark unterschiedliche Mengen an Hardware-Ressourcen benötigen.
Die Fragmentierung der Hardware entsteht hierbei dadurch, dass für eine längere Zeit unterschiedlich große Module so auf den Chip an Stellen konfiguriert werden, an denen genügend freier Platz verfügbar ist. Dies kann dazu führen, dass einzelne Teile eines Moduls unvorteilhaft über den FPGA verteilt werden, da die freien Ressourcen zum Zeitpunkt des Konfigurierens weit über den Chip verteilt sind. Dies resultiert in großen Leitungslängen, den damit einhergehenden langen Signallaufzeiten und einer daraus folgenden niedrigen Performance des Systems.
Diese Verteilung freier Ressourcen kann somit dazu führen, dass durch schlechtes Zeitverhalten oder mangelnde Routingkapazitäten ein Modul trotz augenscheinlich genügender Ressourcen nicht mehr konfiguriert werden kann.
Ein einfaches Beispiel für die Fragmentierung programmierbarer Hardware ist in Abbildung 8.5 gegeben. Vom dynamisch rekonfigurierbaren System aus 8.5 a) wird im ersten Schritt das Modul A entfernt (siehe Abbildung 8.5 b). Im nächsten Schritt zu c) wird dann das Modul B entfernt und ein Modul C hinzugefügt. Soll nun Modul A wieder konfiguriert werden, muss entweder Modul C verschoben werden, was einer Defragmentierung entspräche. Alternativ könnte auch, wie in d) dargestellt, Modul A in zwei Teile aufgeteilt, also fragmentiert, konfiguriert werden.

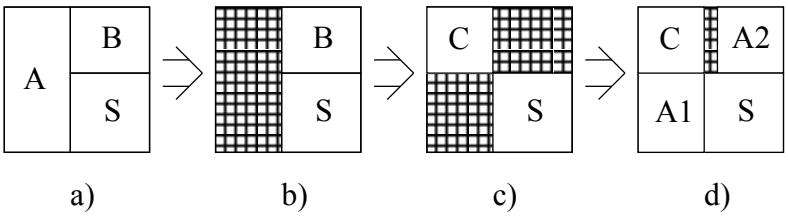


Abbildung 8.5: Beispiel für die Fragmentierung programmierbarer Hardware

8.3.4.1 Fragmentierung in aktuellen rekonfigurierbaren FPGAs

In realen FPGAs spielt Fragmentierung derzeit noch keine große Rolle, da eine einfache Relokation von Modulen in diesen Bausteinen nicht möglich ist. Durch die Diversität im Aufbau der Chips, das nicht freigegebene Format der Bitstreams zur Programmierung sowie die kompletten Tool-Unterstützung ist es nötig, mögliche Konfigurationen, in denen ein System arbeiten kann, bereits vor dessen Betrieb zu planen. Nicht nur die Funktionalität, sondern auch der genaue Ort der Modulschnittstellen müssen bereits vor dem Betrieb feststehen, um eine korrekte Interaktion untereinander und mit der Umgebung sicherzustellen.

8.3.4.2 Fragmentierung in zukünftigen Bausteinen

Auf zukünftig denkbaren FPGAs, auf denen wie in Speichern Inhalte zur Laufzeit nach Belieben konfiguriert und verschoben werden können, wird es langfristig möglich sein, die von Speichern bekannten Methoden zur Fragmentierungsvermeidung und Defragmentierung anzuwenden. Damit Bausteine dies leisten können, müssen ihre Strukturen sehr viel gleichmäßiger werden. Dadurch kann erneutes Platzieren und Verdrahten eines Moduls, um es an einem anderen Ort zu betreiben oder sogar um es automatisiert in auf dem Chip verteilbare Teilkomponenten aufzuspalten, unnötig werden.
Durch den zweidimensionalen Aufbau von FPGAs wird es wohl nötig sein, anfangs auch komplexere Defragmentierungsalgorithmen anzuwenden, als dies bei den meist linearen Speichern der Fall ist. Falls auch der logische Aufbau zukünftiger programmierbarer Bausteine linear gestaltet ist, wird der Einsatz der einfachen Standard-Algorithmen möglich.
Sind genügend Kapazitäten vorhanden, um ein Modul komplett an einen anderen Ort zu verschieben und es erst dann weiter zu betreiben, kann eine Defragmentierung im Hintergrund geschehen. Das Verschieben entspricht einer einfachen Relokation (s. Kapitel 8.3.5).
Anders als bei Speichern können hier Defragmentierungsvorgänge dazu führen, dass Module kurzzeitig nicht verfügbar sind. Dies rührt daher, dass ein Modul nur dann korrekt arbeiten kann, wenn alle seine Komponenten nicht nur vorhanden, sondern auch korrekt verbunden sind. Durch die grundsätzliche Beschränkung der Routingressourcen wird dies bei der Verschiebung von Teilkomponenten nicht jederzeit sichergestellt werden können.

8.3.4.3 Umgehung von Fragmentierungsproblemen für Laufzeitrekonfigurierbare EPIC Soft Cores

Die durch Fragmentierung auftretenden Probleme in Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren werden dadurch umgangen, dass die rekonfigurierbaren Bereiche verhältnismäßig grobgranular sind, also um Größenordnungen mehr Ressourcen als eine Logikzelle des FPGAs beanspruchen. Jede Funktionseinheit, sowie jede Hardware anderer Kontexte, benutzt somit einen dieser großen sogenannten Slots, als zusammenhängende, vordefinierte Bereiche des Chips. Entspricht, wie für diesen Ansatz bisher vorgesehen, die Slot-Größe genau dem Ressourcen-Bedarf der größten Funktionseinheit, tritt zumindest im Prozessor keine Fragmentierung der programmierbaren Hardware auf.
Dieses Vorgehen geht auf Kosten der Ressourcen. Innerhalb eines Slots können hier häufig größere Bereiche ungenutzt bleiben. Allerdings hat es, zusätzlich zur Lösung der Fragmentierungsproblematik, den Vorteil, dass für aktuelle Implementierungswerkzeuge, das Floorplanning, also die Anordnung der Module auf dem Baustein, realisierbar bleibt. Zudem gibt erhöhter Platz den Synthesewerkzeugen die Chance, stärkere Optimierung des Designs bezüglich der Laufzeit durchzuführen, was zudem die mögliche Taktung erhöht.
Für aufwändigere, größere Module ist es denkbar, auch mehr als einen Slot zu verwenden.

Durch die vorhergehende feste Planung der Module auf Slots tritt hier, so lange diese Slot-Anzahl eines Moduls in einem kleinen Bereich liegt, nur geringe Fragmentierung auf. Wird die Slot-Größe jedoch zu klein gewählt, resultieren daraus wiederum viele Module, welche sich über mehrere Slots erstrecken, und die Fragmentierungsproblematik kommt auch hier zu tragen.

8.3.5 Relokation von Modulen

Um frei werdende Hardwareressourcen zu nutzen, muss es ermöglicht werden, ein Modul an unterschiedlichen Orten im FPGA-Fabric zu betreiben.
Ein Weg, dies zu realisieren, ist es, den benötigten Bitstrom so zu manipulieren, dass er die Programmierung des FPGA an einer anderen als der vorhergeplanten Stelle ändert. Im Regelfall geschieht dies durch Änderung der Adressen der angesprochenen programmierbaren Logikzellen sowie des verbundenen Routings. Dieser Weg zur Relokation eines Moduls ist allerdings nur möglich, wenn das Format und der Aufbau des Bitstreams bekannt sind. Derzeit ist es bei den Hardware-Herstellern nicht üblich, das Format der Files offenzulegen.
Eine weitere Möglichkeit zur Relokation eines Moduls ist es, das Modul für sämtliche möglichen Stellen des Systems, an denen es betrieben werden kann, zu synthetisieren. Dieses Vorgehen ist relativ speicherintensiv, da für jede mögliche Stelle ein Bitstream vorgehalten werden muss. Der Grund, dieses Verfahren anzuwenden, ist, dass es vom Format des Bitstreams losgelöst ist und somit eine gewisse Unabhängigkeit von der Hardware und deren Hersteller schafft. Ebenso ist mit diesem Verfahren während der operativen Phase des Systems wenig Zeit-Overhead für Berechnungen nötig, da lediglich bestimmt werden muss, welcher Bitstream verwendet werden soll. Adress- und Routing-Berechnungen entfallen.
Mit entsprechenden Kenntnisse der Hardware wäre es möglich, dass die Relokation von Modulen gänzlich ohne zusätzliche Bitstreams ausgeführt werden kann. Dazu könnte die Programmierung ausgelesen werden, um diese daraufhin an einer anderen Stelle wieder einzuspielen.
Mit höherem Aufwand an Hardware wäre es auch denkbar, dass der programmierbare Hardware-Baustein über eine zusätzliche Abstraktionsschicht verfügt, welche den Ort eines Moduls verschattet. Das Problem der Relokation von Modulen würde sich in solcher Hardware für Anwender-Designs nicht mehr stellen. Es wäre nur noch für den Entwurf des programmierbaren Bausteins relevant.

8.3.6 Prototyp: Implementierung von Conways Game of Life

Zum Nachweis der praktischen Anwendbarkeit von PARS zur Modulerkennung und der Konzepte zur Relokation von Modulen wurde eine prototypische Implementierung von Conways Game of Life erstellt. Zudem zeigt sie Vorgehen zum Zurückrollen der Effekte vorheriger Berechnungsschritte, wie in Abschnitt 8.3.2 vorgeschlagen.

8.3.6.1 Der zelluläre Game of Life-Automat

Das Game of Life [127] ist ein von John Horton Conway im Jahr 1970 veröffentlichter zweidimensionaler zellulärer Automat. Es besteht aus einem Spielfeld, in dem Zellen belegt oder frei sind. Oft wird für den Status einer Zelle auch das Wort *lebend* anstatt belegt und *tot* anstatt frei benutzt. Das zu berechnende Problem wird durch eine passende Initialisierung des Spielfeldes mit belegten und freien Zellen dargestellt.
Eine Runde des Spiels, auch Zyklus genannt, besteht darin, dass alle Zellen des Automaten abhängig von ihrem Zustand vor der Runde entscheiden, ob sie lebendig oder tot bleiben oder aber ihren Zustand wechseln. Dies entscheiden sie aufgrund der benachbarten Zellen. Die acht unmittelbar benachbarten Zellen werden zur Bestimmung des Folgezustandes einer Zelle herangezogen.
Die Regeln von Conways ursprünglichem Game of Life werden als 23/3 bezeichnet. Dies bedeutet, dass eine lebendige Zelle, welche zwei oder drei Nachbarn hat, im nächsten Zyklus erhalten bleibt und eine tote Zelle mit genau drei Nachbarn in der nächsten Runde neu geboren wird. Zellen mit weniger als zwei oder mehr als drei Nachbarn sterben im nächsten Zyklus.
Da der Status einer Zelle nur von dessen Nachbarn im vorherigen Zyklus des Spiels abhängt, lässt sich das Berechnen eines solchen Zyklus einfach parallelisieren. Alle Zellen können, sofern sie das Wissen über ihre Nachbarn haben, auch zeitgleich ausgewertet werden. 1974 wurde nachgewiesen, dass ein Game of Life-System mit genügend großem Spielfeld Turing-vollständig ist [42], also alle berechenbaren Probleme lösen kann.
Für den Demonstrator wurde Conways Game of Life wegen seiner eben beschriebenen einfachen massiven Parallelisierbarkeit und der ansprechenden, wenn auch für das System wenig relevanten, Turing-Vollständigkeit gewählt.

8.3.6.2 Logischer Aufbau

Abbildung 8.6 zeigt die Module des Demonstrators. Diese Umfassen zwei Funktionseinheiten zur Berechnung der Folgezustände von mehreren Zellen, zwei partiell Rekonfigurierbare Semaphoren (vgl. Kapitel 8.3.1.3) und zwei RAMs in der Größe des Spielfeldes, die wiederum jeweils zweigeteilt sind. Zudem benötigt der Prototyp eine Hardwareeinheit, welche für die Ablaufsteuerung zuständig ist.
Beim implementierten Prototypen ist das Spielfeld 160x300 Zellen groß. Beide Speicher haben jeweils eine Speicherkapazität von 6000 Byte, die beiden Funktionseinheiten berechnen die Folgezustände von jeweils 80 Zellen in einem Taktzyklus und dadurch gemeinsam jeweils eine komplette Spalte des Spielfeldes.

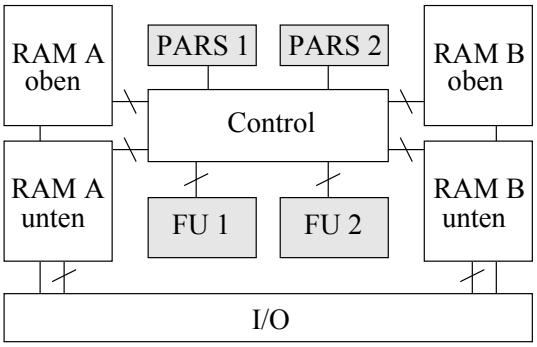


Abbildung 8.6: Module des Game of Life-Prototyps

8.3.6.3 Arbeitsweise

Der Game of Life-Prototyp besitzt zwei getrennte Speicher, welche jeweils groß genug sind, das komplette Spielfeld zu halten. Zur Berechnung eines Lebenszyklusses wird der Inhalt des einen Speichers über eine Anzahl an Funktionseinheiten in den anderen Speicher bewegt. Im nächsten Zyklus tauschen die beiden Speicher ihre Aufgaben und die Inhalte des vorherigen Zielspeichers wechseln über die Funktionseinheiten zurück in den vorherigen Quellspeicher.
Die Funktionseinheiten bestehen dabei aus vielen einfachen und gleichartigen Schaltungen, welche jeweils für eine aktuell zu bearbeitende Zelle anhand deren Nachbarn den Folgezustand bestimmen. Im implementierten Prototyp bilden jeweils 80 solcher Schaltungen eine Funktionseinheit.
Über einen Zähler wird bestimmt, wie viele Lebenszyklen berechnet werden sollen und wie oft somit das Spielfeld über die Funktionseinheiten von einem Speicher in den anderen transferiert wird.

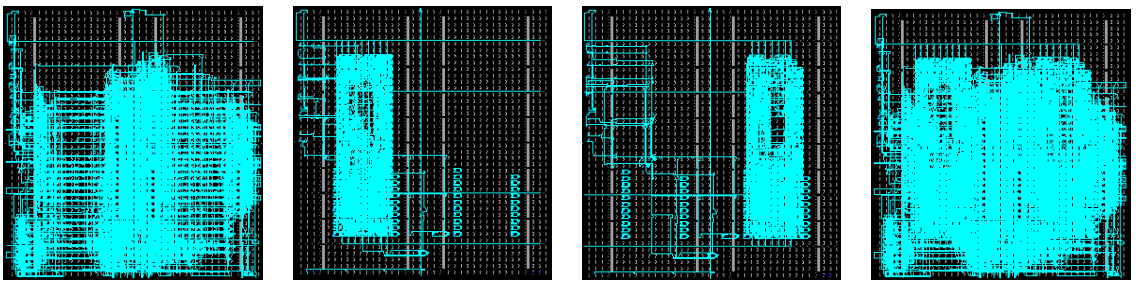
8.3.6.4 Partielle Rekonfiguration

Der Prototyp ist darauf ausgelegt, sowohl mit einer als auch mit zwei Funktionseinheiten, welche jeweils eine halbe Spalte des Spielfeldes in einem Taktzyklus berechnen, arbeiten zu können. Durch die Konfiguration der Funktionseinheiten mittels eines PARS erkennt die Kontrolleinheit, wie viele Funktionseinheiten vorhanden sind, und bindet diese entsprechend in die Berechnung ein. Wird der PARS entfernt, schließt die Kontrolleinheit die FU wieder aus. Dies kann auch zur Laufzeit während der Abarbeitung einer Anzahl an Zyklen geschehen. Der Platz auf dem Chip kann, falls vorhanden, für das Nachkonfigurieren einer Einheit verwendet werden. Wird der Platz für andere Aufgaben im System benötigt, kann auch zu jedem Zeitpunkt der Berechnung eine der Funktionseinheiten wieder entfernt werden ohne dabei die Korrektheit des Ergebnisses zu beeinflussen.
Hierzu überwacht die Kontrolleinheit die PARS. Dadurch kann sie zu jedem Zeitpunkt feststel-

len, ob sich die Anzahl der Funktionseinheiten verändert. Wird eine zusätzliche Funktionseinheit erkannt, so wird der nächste Lebenszyklus unter Verwendung beider Funktionseinheiten berechnet.
Für den Fall, dass während einer aktiven Berechnung eine Funktionseinheit entfernt wird, werden deren Ergebnisse des aktuellen Lebenszyklusses verworfen, um ein korrektes Ergebnis sicherzustellen. Diese Berechnungen werden dann von der verbleibenden Funktionseinheit erneut durchgeführt. Für die folgenden Zyklen werden dementsprechend beide Hälften sequentiell von der einen Einheit berechnet. Dieses Vorgehen benötigt folglich die doppelte Laufzeit, beeinflusst jedoch die Qualität des Ergebnisses nicht.

8.3.6.5 Hardware-Synthese

Der VHDL-Quell-Code für den Prototypen ist streng modular aufgebaut. Er besteht aus einem Top-Level-Modul, welches sowohl die Module für die statische Logik, wie die Kontrolleinheit und I/O-Module, als auch die rekonfigurierbaren Module instanziiert. Letztere sind zwei Funktionseinheiten, welche, wie beschrieben, jeweils aus 80 kleinen Einheiten zur Berechnung des Folgezustandes einer Zelle bestehen, und die zugehörigen PARS.
Die Synthese und Erstellung der auf den FPGA konfigurierbaren Netzlisten erfolgt mittels des Tools ReconfGenerator (siehe Abschnitt 8.4.2), welches auf Xilinx-Programme zurückgreift. Daraus resultieren Netzlisten für die statische sowie für die komplette Logik. Letztere vereinfacht die Initialisierung des Systems. Ebenso werden bei diesem Vorgehen automatisch partielle Netzlisten für die rekonfigurierbaren Module und leere Netzlisten, um diese wieder zu entfernen, erstellt. Diese Resultate sind grafisch in Abbildung 8.7 dargestellt.



a) b) c) d)

Abbildung 8.7: Netzlisten des Game of Life-Prototyps: a) Statische Module, b) linkes rekonfigurierbares Modul, c) rechtes rekonfigurierbares Modul, d) komplette Netzliste mit zusammengefügten statischen und rekonfigurierbaren Modulen

8.3.6.6 Größe der Netzlisten

Der absolute Logikbedarf der einzelnen Module, ihr Anteil am Design und an einem mittelgroßen FPGA von Xilinx (dem Virtex-II xc2v1000), sowie die Größe der Bitstreams sind in Tabelle 8.1 aufgetragen.
Erkennbar ist, dass der Bitstream einer FU wesentlich kleiner ist als der eines kompletten Designs. Da die Konfigurationszeit linear mit der Größe des Bitstreams wächst, sind also partielle Änderungen des Designs allgemein effizienter als eine komplette Rekonfiguration. Dies ist bereits ohne die Fähigkeit, während des Konfigurationsvorgangs weiterzuarbeiten, der Fall. Des Weiteren lässt sich an den Werten ablesen, dass weit über die Hälfte des Designs aus rekonfigurierbaren Modulen besteht. Es lässt also großen Spielraum für andere Anwendungen, welche zusätzlichen Platz für ihre Berechnungen benötigen. Auf dem eingesetzten FPGA stehen für solche Anwendungen auch noch annähernd die Hälfte der Logikressourcen exklusiv zur Verfügung.
Diese Daten bestätigen zudem, dass der Platzbedarf eines PARS nicht ins Gewicht fällt.

Tabelle 8.1: Größe der Bitstreams und Logiknutzung

	Größe des Bitstreams	Anzahl an Slices	Anteil am kompletten Design	Nutzung des xc2v1000
Komplett	499 kByte	2.728	100%	53%
Statisch	n.a.	1.164	43%	22%
Eine FU	83 kByte	772	28%	15%
Ein PARS	20 kByte	10	<1 %	<1%

8.3.6.7 Versuchsaufbau

Der Prototyp des zellulären Automaten ist auf einem Celoxica RC200 Board [21] mit einem Xilinx Virtex-II xc2v1000 realisiert. Dieses Board ist zur Konfiguration mittels eines Xilinx Platform Cable USB mit dem Host-PC verbunden. Zusätzlich ist es mit einem Parallelkabel (nach IEEE 1284), auf welchem ein einfaches proprietäres Protokoll benutzt wird, am Host angeschlossen. Die initiale sowie die partielle Rekonfiguration des FPGAs wird vom Host-PC aus mittels des Tools iMPACT von Xilinx durchgeführt. Eine Perl-Anwendung auf dem Host dient sowohl zum Austausch der Startkonfiguration des Automaten und der Anzahl zu berechnender Zyklen über das Parallelkabel als auch zum Aufnehmen, Konvertieren und Anzeigen der fertig berechneten Konfiguration. Ebenso misst es die Zeit, welche der Prototyp benötigt, um die Berechnungen durchzuführen. Dieser Aufbau ist in Abbildung 8.8 dargestellt.

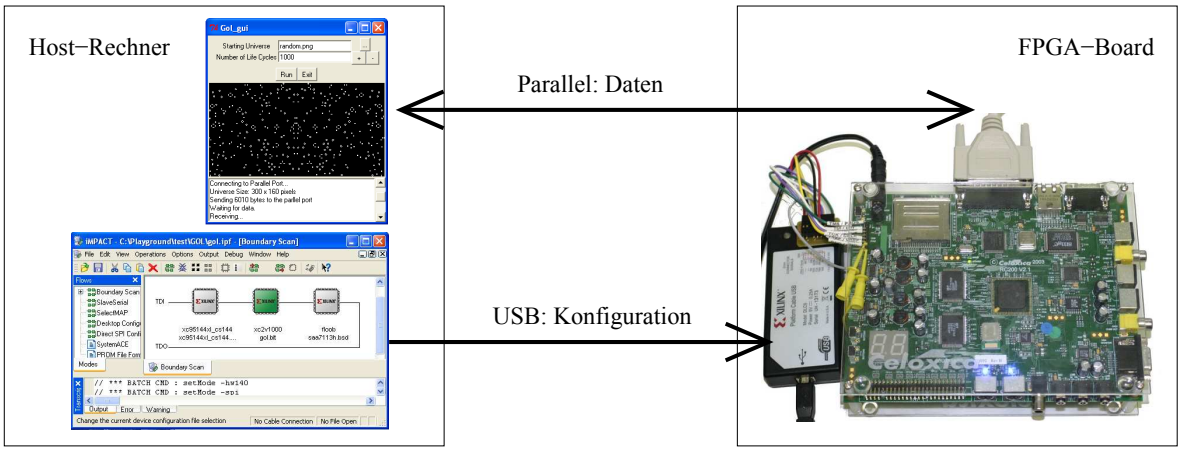


Abbildung 8.8: Versuchsaufbau des Game of Life-Prototyps

8.3.6.8 Performanzmessung im Betrieb

Die initiale Konfiguration wird zum Betrieb des Game of Life-Prototyps auf den FPGA konfiguriert. Auf dem Host-Rechner wird der Startzustand aus einer monochromen PNG-Grafik errechnet. Das resultierende Bitfeld und die Anzahl der zu berechnenden Zyklen wird, wie im vorherigen Abschnitt beschrieben, durch ein einfaches Protokoll über den Parallel-Port an das Board übertragen.
Sobald die Daten komplett übertragen wurden, wird im FPGA das Spielfeld für die angegebene Anzahl an Lebenszyklen über die Funktionseinheiten zwischen den beiden Speichern hin und her geschoben. Nach Beendigung wird das Spielfeld zurück an den Host gesendet, dort wieder in eine Grafik umgewandelt und angezeigt.
Auf die vom Host getriebene Rekonfiguration der Funktionseinheiten reagiert das System selbständig ohne Qualitätsverlust. Um die Korrektheit des Ergebnisses zu überprüfen, wurde das resultierende Spielfeld für verschiedene, teils konstruierte, teils zufällige Eingaben mit dem der Game of Life-Implementierung LIFE32[1] verglichen.
Um die Rechenzeit möglichst von der Kommunikationszeit zu trennen, beginnt das Host-Programm sofort im Anschluss an die Übertragung des letzten Bits über den Parallel-Port mit dem Zählen der Zeit und stoppt diese, sobald das erste Bit der Übermittlung des Ergebnisses ankommt.
Zur Überprüfung der Performanz des Systems wurde ein zufälliges Spielfeld der vorgegebenen Größe von 160x300 Zellen für 5 Millionen Lebenszyklen berechnet. Die Versuchsszenarien umfassen einen Lauf auf zwei Funktionseinheiten, einen Lauf auf einer Funktionseinheit sowie zwei Szenarien, in denen die Anzahl der Funktionseinheiten zur Laufzeit umkonfiguriert wird. Die in Tabelle 8.2 angegebenen Messwerte entsprechen hierbei den Erwartungen. Der

[1] erhältlich auf `http://psoup.math.wisc.edu/Life32.html`, Stand: Mai 2011

Geschwindigkeitsgewinn wächst bei diesem Problem linear mit der Anzahl an Funktionseinheiten.

Tabelle 8.2: Messwerte für fünf Millionen Lebenszyklen bei 25.175 MHz.

Konfigurierte FUs	Benötigte Zeit	Zellen pro Sekunde
2 FUs	60,7s	$3{,}95*10^9$
1 FU	121,5s	$1{,}97*10^9$
1 FU für ca. 40 s, dann 2 FUs	82,5s	$2{,}90*10^9$
2 FU für ca. 40s, dann 1 FU	78,3s	$3{,}06*10^9$

8.3.6.9 Erkenntnisse

Die Implementierung von Conways Game Of Life in rekonfigurierbarer Hardware liefert mehrere Erkenntnisse über Erstellung und Betrieb rekonfigurierbarer Systeme und ermöglicht zudem Messungen der Rekonfigurationsdauer.
Zum einen bestätigt der Prototyp den Ansatz, PARS als Hilfsmittel zur Erkennung von fertig konfigurierten Modulen und zur Ankündigung des Entfernens von Modulen einzusetzen. Durch ihren gemessen geringen Platzbedarf von ca. zehn Slices eines Virtex 2 FPGAs ist ihr Ressourcenverbrauch bereits bei kleinen Designs zu vernachlässigen. Die Dauer ihrer Konfiguration ist durch die geringe Größe ihrer Programmierungs-Bitstreams von nur ca. 20 kByte im Vergleich zur Konfigurationszeit einer FU verhältnismäßig klein.
Eine weitere wichtige Erkenntnis aus dem Prototyp ist, dass die Relokation von partiell rekonfigurierbaren Modulen ohne Einbußen in der Genauigkeit der Berechnung durchführbar ist. Dazu muss das Design jedoch entsprechend vorbereitet sein, um auf das Hinzufügen und Entfernen von Modulen zu reagieren.
Ist das Problem, wie das Game of Life, dazu geeignet, parallel abgearbeitet zu werden, können sich mehrere Funktionseinheiten in dessen Berechnung unterstützen. Sie müssen jedoch nicht beide vorhanden sein, um ein korrektes Ergebnis zu liefern.
Der Prototyp gibt zudem die Möglichkeit, die Rekonfigurationszeiten von partiellen Modulen zu messen. Die in Tabelle 8.3 aufgelisteten Messergebnisse zeigen jedoch, dass bei der Hostgetriebenen Konfiguration über JTAG diese Zeiten, um ein kleines Modul zu konfigurieren, noch im hohen Millisekundenbereich liegen. Berechnet man die Übertragungsgeschwindigkeit der Nutzdaten, also der Bitstreams aus Tabelle 8.1, so lässt sich erkennen, dass auf diese grob die Hälfte der tatsächlichen Übertragungsrate entfällt. Der Rest der Zeit ist dem Protokoll-Overhead zuzuschreiben.
Dadurch, dass die Konfigurationszeit der extrem kleinen PARS fast unabhängig von der Übertragungsrate ist, lässt sich erkennen, dass ein nicht zu vernachlässigender Teil der Zeit für den Verbindungsaufbau zwischen Host und FPGA benötigt werden. Diese Annahme wird dadurch bestätigt, dass die Konfigurationszeit bei einer Übertragungsrate von 6 MBaud sogar geringer

Tabelle 8.3: Messungen der Konfigurationszeiten unter Verwendung von iMPACT

	6 MBaud	12 MBaud	24 MBaud
Komplett	1.171 ms	747 ms	453 ms
Eine FU	210 ms	130 ms	85 ms
Ein PARS	69 ms	72 ms	72 ms

ist als bei höheren Raten. Dies sollte daher rühren, dass es bei der Nutzung der standardmäßigen Übertragungsrate zur Konfiguration des FPGAs, welche eben bei 6 MBaud liegt, nicht mehr nötig ist, die Übertragungsgeschwindigkeit über das Protokoll auszuhandeln, und somit die komplett nötige Kommunikation reduziert wird.
Zudem dient der Game of Life-Prototyp zum Nachweis der Funktion einiger der für Laufzeitrekonfigurierbare EPIC Soft Core Prozessoren nötigen Konzepte im Bereich der programmierbaren Hardware. Zu diesen zählen die Erstellung rekonfigurierender Bitstreams und die komplexe Verdrahtung rekonfigurierbarer Module mit dem Speicher. Diese Eigenschaften werden in den Kapiteln 8.4.2 und 8.4.5 erläutert.

8.3.7 Befehlsverteilung

Die Einzelbefehle eines Befehlsbündels müssen den unterschiedlichen Funktionseinheiten zugeordnet werden. Diese Anforderung besteht so auch bei klassischen EPIC-Prozessoren. Bei der in dieser Arbeit vorgestellten Klasse von Prozessoren besteht der Unterschied in der variablen Anzahl an Funktionseinheiten und den somit zusätzlich nötigen Verbindungen zu diesen. Sind ausreichend Routing-Ressourcen vorhanden, so können die Leitungen zu den Funktionseinheiten statisch aufgebaut werden. Falls nicht, ist es nötig, diese mit der Konfiguration einer Einheit ebenfalls zu konfigurieren.
Bei Funktionseinheiten, welche mehrere Takte für die Verarbeitung eines Befehls benötigen, können auch Switches zwischen die Funktionseinheiten und den Befehlsspeicher geschaltet werden. Dies senkt den Aufwand an Ressourcen und kann in manchen Fällen die Rechenleistung des Systems nur wenig verringern. Es besteht die Möglichkeit, auch diese Switches rekonfigurierbar zu implementieren.

8.3.8 Prototyp: Zusammenarbeit mehrerer Soft Core Prozessoren auf einem Befehlsstrom

Um die Befehlsverteilung von Bündeln in der Art und Weise eines EPIC-Prozessors zu untersuchen, befasst sich die im Rahmen dieser Forschungstätigkeit angefertigte Diplomarbeit [100] damit, mehrere einfache, jedoch komplette Soft Core Prozessoren so zu verbinden, dass sie gemeinsam einen einfachen Befehlsstrom abarbeiten. Dieser ist der EPIC-Architektur entsprechend aus Befehlsbündeln aufgebaut.

Die unabhängigen Befehle eines Befehlsbündels werden hier von je einem eigenen Prozessor abgearbeitet. Eine übergeordnete Einheit trägt dafür Sorge, dass die Teilbefehle korrekt an die Prozessoren verteilt werden und Befehlsbündel immer komplett abgearbeitet werden. Sie enthält auch einen gemeinsamen Program-Counter, dessen Verwaltung aus den einzelnen Prozessoren abgezogen wurde. Eine zusätzliche Aufgabe dieser Einheit ist die Verwaltung gemeinsam genutzter Ressourcen wie Speicher oder I/O-Ports. Der Aufbau dieses Ansatzes ist in Abbildung 8.9 dargestellt.

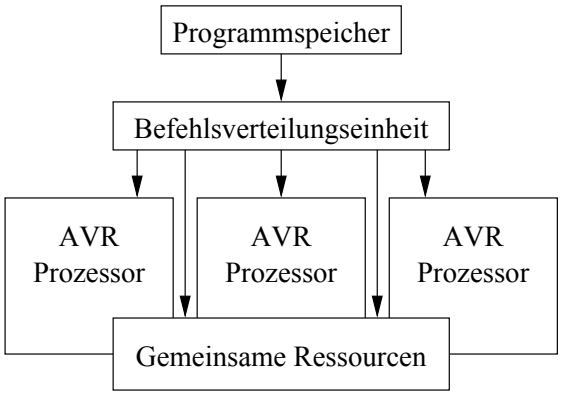


Abbildung 8.9: Aufbau eines prototypischen Multi-Prozessors zur Abarbeitung von EPIC-Befehlsbündel

Der erstellte Prototyp verbindet die in Abschnitt 2.5.3.1 vorgestellten AVR-Core-Prozessoren miteinander. Diese einfachen 8-Bit-Microcontroller sind frei als VHDL-Quellcode erhältlich [94].
Diese Arbeit bestätigt, dass es möglich ist, einen EPIC-Befehlsstrom an mehrere Verarbeitungseinheiten, in diesem Fall einfache Prozessoren, zu verteilen und das Programm dabei korrekt abzuarbeiten.

8.4 Rekonfigurierbare Hardware

Der Organisation eines Prozessors mit der in dieser Arbeit vorgestellten Architektur liegt rekonfigurierbare Hardware zugrunde. Um solch einen Prozessor zu implementieren, müssen für diese Hardware einige nicht triviale Anforderungen erfüllt werden. Diese sind in Abbildung 8.10 zusammengefasst und werden in den folgenden Abschnitten behandelt. Wie zuvor sind auch hier passende Beispielimplementierungen beschrieben.

8.4.1 Erstellung rekonfigurierender Bitstreams

Um verschiedene Funktionseinheiten hintereinander auf derselben Hardware zu betreiben, ist es nötig, für jede Einheit mindestens einen Bitstrom, welcher die Konfigurationsdaten für

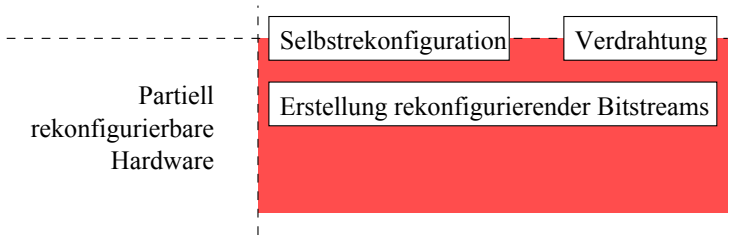


Abbildung 8.10: Herausforderungen bei der Nutzung der Hardware Laufzeitrekonfigurierbarer EPIC Soft Core Prozessoren, Ausschnitt aus Abbildung 8.1

die Hardware enthält, zu erzeugen. Soll eine Einheit an verschiedenen Stellen der Hardware benutzt werden, also wie in Abschnitt 8.3.5 beschrieben relokiert werden, ist es eventuell nötig, den entsprechenden Bitstrom für jedes Modul an jeder möglichen Stelle der Hardware zu generieren. Um die Erstellung der somit maximal $\mathcal{O}(n^2)$ Bitströme zu realisieren, ist ein möglichst einfacher und automatisierbarer Build-Prozess für diese in der Praxis unumgänglich.

8.4.2 Prototyp: ReconfGenerator

Das vorveröffentlichte Werkzeug ReconfGenerator [109] wurde im Rahmen dieser Forschungsarbeit entwickelt, um die automatisierte Synthese und Bitstromgenerierung für Hardware-Designs mit einer großen Anzahl rekonfigurierbarer Module mit verschiedenen Implementierungen zu vereinfachen. Es unterstützt die rekonfigurierbaren FPGAs der Firma Xilinx und baut dazu auf deren Early Access Partial Reconfiguration Flow [134] auf, erweitert und automatisiert diesen. Synthese und Implementierung erfolgen durch automatisch vom Tool erzeugte Skripte, welche dann Synthese- und Implementierungs-Tools mit den richtigen Eingabedaten und Parametern aufrufen.
Im Gegensatz zum Tool-Flow von Xilinx mit dem Tool PlanAhead [59], erzeugt der Generator voll automatisiert alle Bitstreams, sowohl die kompletten, statischen als auch partiellen. Viele Überprüfungen helfen dem Benutzer, übliche Probleme beim Design partiell rekonfigurierbarer Systeme zu umgehen. Dazu gehört vor allem die Platzierung von Modulen und Schnittstellen.
Als Eingabe benötigt das Tool lediglich die HDL-Dateien der einzelnen Module, die Constraints, also die Vorgaben zur Anordnung der partiell rekonfigurierbaren Bereiche sowie der Lage von Pins, und die festgelegten Schnittstellendefinitionen, die sogenannten Busmacros. Hieraus erzeugt es ohne Interaktion des Benutzers alle Bitströme zur Konfiguration des Zielsystems. Dieser im Gegensatz zu dem anderer Werkzeuge stark vereinfachte Ablauf ist in Abbildung 8.11 dargestellt.
Zur einfachen Bedienung und Projekterstellung verfügt das Tool über eine grafische Oberfläche, die sämtliche Funktionen unterstützt. Diese ist in Abbildung 8.12 dargestellt. Die

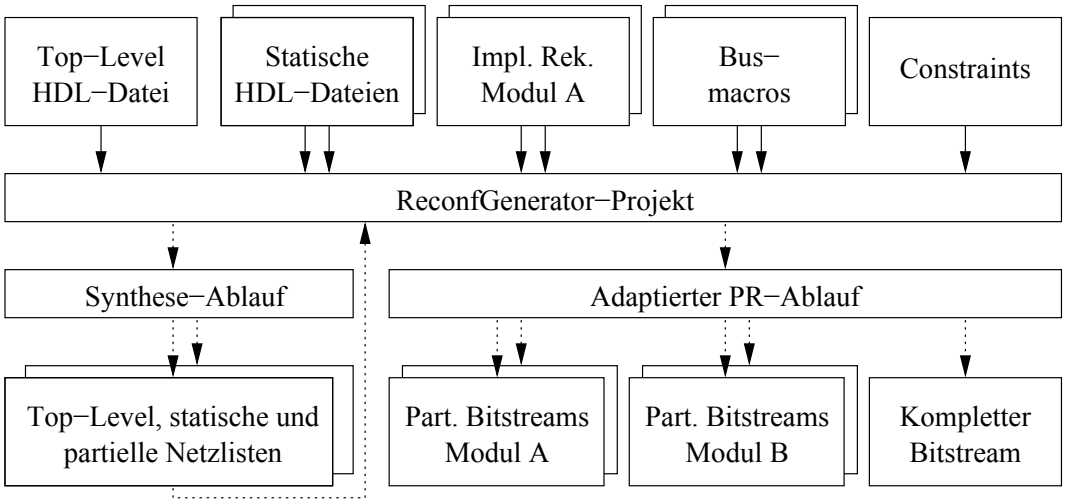


Abbildung 8.11: Interner Ablauf des ReconfGenerators bei der Erstellung der Bitstreams

erstellten Projekte können direkt aus der Oberfläche heraus, jedoch auch per Kommandozeile oder vollautomatisch durch Skripte abgearbeitet werden.
Der ReconfGenerator wurde in einer Diplomarbeit [69] um die Unterstützungen für Submodule von rekonfigurierbaren Modulen, bei der Schnittstellenfestlegung zwischen den Modulen, beim Einsatz aktueller FPGAs sowie für weitere Funktionen, die den Komfort bei der Benutzung erhöhen, erweitert. Ebenso wurden in dieser Arbeit umfangreiche Tests mit verschiedenen Designs durchgeführt und somit die Funktionalität und Praktikabilität des Ansatzes nachgewiesen.
Der komplexe Build-Prozess des Game of Life-Prototyps aus Abschnitt 8.3.6 wurde ebenfalls mit dem ReconfGenerator erstellt. In diesem Prozess werden für vier rekonfigurierbare Module, welche teilweise wieder aus Submodulen aufgebaut sind, sowie für ein ebenso hierarchisch aufgebautes statisches Design alle Bitstreams zur Konfiguration und Entfernung der partiell rekonfigurierbaren Module sowie des kompletten Designs erstellt.
Inzwischen sind viele der Ansätze des ReconfGenerators auch in der Tool-Suite von Xilinx vorhanden. Einige Vorteile des Tools, wie die schnelle und einfache maßgeschneiderte Projekterstellung und der offene Source-Code bleiben jedoch weiterhin erhalten.

8.4.3 Selbstrekonfiguration

Um eine dynamische Anpassung des Prozessors zur Laufzeit zu ermöglichen, ist es nötig, dass das System eine Möglichkeit bietet, selbst Einfluss auf die eigene Konfiguration zu nehmen. Abhängig davon, welche Möglichkeiten das Fabric hierzu bietet, sind unterschiedliche Vorgehensweisen zu dieser Selbstrekonfiguration anzuwenden.

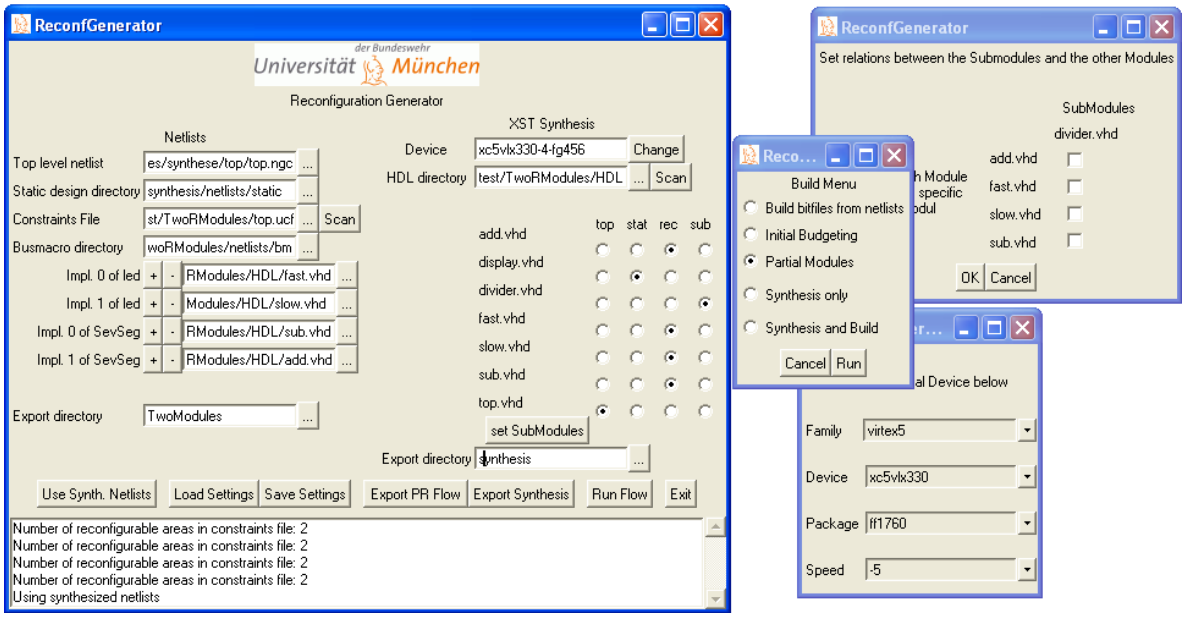


Abbildung 8.12: Oberfläche des ReconfGenerators

8.4.3.1 Selbstrekonfiguration durch internen Konfigurations-Port

Manche FPGAs sehen bereits die Mechanismen zur partiellen Selbstrekonfiguration vor. Hierzu verfügen sie über interne Ports, also spezielle Sites auf dem Silizium-Die, über welche die Konfigurationsdaten geändert und teilweise auch ausgelesen werden können. Bei aktuellen Bausteinen wird mit diesem Port meist parallel kommuniziert, was eine verhältnismäßig hohe Bandbreite ermöglicht.
Bei den aktuell am ehesten für partielle Rekonfiguration geeigneten FPGAs der Firma Xilinx wird dieser Port ICAP (Internal Configuration Access Port) genannt. Dieser 32-Bit breite Port sollte maximal mit 100 MHz angesprochen werden, hat also eine maximale Datenübertragungsrate von 3,2 GBit/s.
Eine Möglichkeit, das Protokoll, mit welchem über diese Ports kommuniziert wird, umzusetzen, ist durch spezielle Hardware-IPs. Diese implementieren in der Regel eine Zustandsmaschine. Durch den spezialisierten und parallelen Aufbau dieses Ansatzes erreicht er hohe Kommunikations- und Rekonfigurationsbandbreiten.
Ebenso kann der Port durch geeignete Interfaces an einen Controller angeschlossen und das Protokoll in Software implementiert werden. Es besteht sowohl die Möglichkeit, einen dedizierten Controller, welcher ausschließlich für die Rekonfiguration genutzt wird, oder den Laufzeitrekonfigurierbaren EPIC Soft Core Prozessor, welcher geändert werden soll, einzusetzen. Diese Design-Entscheidung ist abhängig von den verfügbaren Ressourcen des Systems.
Die Software-Lösung wird meist nicht so performant wie die implementierten Zustandsauto-

maten sein, da die Taktfrequenz eines Soft Core Prozessors meist um maximal den Faktor zwei höher als der Rekonfigurationstakt ist. Somit müsste der Prozessor, um die volle Rekonfigurationsbandbreite auszuschöpfen, in jedem zweiten Takt Daten an den Konfigurationsport senden.

8.4.3.2 Selbstrekonfiguration durch Rückschleifen der Konfigurations-Pins

Eine Möglichkeit, Selbstrekonfiguration durchzuführen, ist es, die externen Konfigurations-Pins mit frei programmierbaren Ausgangs-Pins des Chips außerhalb des Gehäuses zu verbinden. Das Konfigurationsprotokoll muss dann durch die Logik des FPGAs, als Hardware IP-Core oder als Software durch einen Prozessor, umgesetzt werden.
Obwohl dieser Ansatz allgemein anwendbar ist, ergeben sich Nachteile. Zum einen werden zusätzliche Eingangs-Pins benötigt. Da die Konfigurations-Pins nur einmal vorhanden sind, sind zudem externe Multiplexer nötig, falls der FPGA durch eine externe Quelle initialisiert werden soll. Da die Ausgangspannung des FPGAs nicht zwingend der für die Eingänge nötigen entspricht, kann es auch sein, dass Spannungswandler in die externe Verbindung eingebaut werden müssen. Beides erhöht die Größe des Systems. Bei aktuell verfügbaren FPGAs ist oftmals die Bandbreite der externen Konfigurations-Ports geringer als die der internen. Eine partielle Selbstrekonfiguration über den externen Port dauert in diesem Fall länger.

8.4.3.3 Host-getriebene Konfiguration

Eine ebenso universelle Form der Selbstrekonfiguration eines Systems ist es, ein zusätzliches externes Host-System zu verwenden. Dieses System wird mit den externen Konfigurations-Pins des FPGAs verbunden und kommuniziert mit diesem über die üblichen Protokolle. Um dieses Verfahren als Selbstrekonfiguration bezeichnen zu können, muss zusätzlich ein Kommunikationskanal vom FPGA zum Host vorhanden sein, über welchen die Rekonfiguration angestoßen wird.
Der Vorteil dieses Ansatzes ist, dass auch die Initialkonfiguration des FPGAs so bewerkstelligt werden kann. Er erhöht jedoch, wie das Vorgehen mit Rückschleifen der Konfigurations-Pins, die minimale Bauform des Systems und erreicht nicht die kürzestmöglichen Konfigurationszeiten. Durch den erhöhten Hardware-Aufwand eines Host-Systems ist dieser Ansatz nur dann sinnvoll, wenn bereits ein externer Prozessor im System vorhanden ist, dessen Ressourcen ausreichend hoch sind, die Rekonfiguration zusätzlich zu übernehmen. Für die meisten eingebetteten Systeme kommt die Host-getriebene Rekonfiguration somit nicht in Frage.

8.4.4 Prototypen: Selbstrekonfiguration

Die im vorherigen Abschnitt beschriebenen Techniken zur Selbstrekonfiguration von FPGAs werden im Folgenden anhand mehrerer Prototypen untersucht. Dabei kommen unterschiedliche Methoden und Bausteine zum Einsatz.

8.4.4.1 Vergleich externer und interner partieller Rekonfiguration

Die in Kooperation mit der Firma Rohde und Schwarz durchgeführte Arbeit [106] untersucht Ansätze externer und interner partieller Rekonfiguration. Der Forschungsbeitrag ist speziell auf den Anwendungsbereich Software Defined Radio (vergleiche Kapitel 2.4.3) zugeschnitten. Hierbei wurde ein einfaches rekonfigurierbares Design für einen Xilinx Virtex II Pro FPGA implementiert. Dieses kann für Messungen Host-getrieben konfiguriert werden.
Zudem wurde dieses Design um eine Konfigurationsmöglichkeit direkt über den internen Konfigurationsport, den ICAP, des FPGAs erweitert. Als Prozessorelement arbeitet ein PicoBlaze Soft Core Prozessor (siehe Abschnitt 2.5.3.3), welcher auf dem FPGA konfiguriert wird und die Kommunikation mit dem ICAP übernimmt.
Die Messungen der Rekonfigurationszeiten ergab, dass für Virtex II FPGAs die interne Rekonfiguration um etwa eine Zehnerpotenz schneller ist als die aktuell schnellstmögliche externe Konfiguration. Dabei erreichte die Rekonfiguration mittels des PicoBlaze fast die Geschwindigkeit der schnellsten Rekonfigurationsmöglichkeit, welche durch interne Steuerung des ICAP über eine Finite State Machine erzielt werden kann.

8.4.4.2 Rekonfiguration aktueller FPGAs über ICAP

In der Arbeit [105] werden Rekonfigurationseigenschaften eines Virtex 5 FPGAs untersucht. Hierzu dient ein einfaches rekonfigurierbares Design, welches um einen MicroBlaze-Prozessor (siehe Abschnitt 2.5.3.3) zur Ansteuerung des ICAP-Ports erweitert ist. Bei diesem Beispiel werden die Bitstreams zur Rekonfiguration von einer angeschlossenen CompactFlash-Karte eingelesen. Der Anstoß zur Selbstrekonfiguration wird dem FPGA durch einen Host-PC gegeben, welcher über die serielle Schnittstelle verbunden ist. Die Rekonfiguration erfolgt dann jedoch komplett unabhängig von diesem Host.
Zum Test des Werkzeugs ReconfGenerator aus Abschnitt 8.4.2 wurde das rekonfigurierbare Design in dieser Arbeit damit erstellt.
Durch die 32 Datenleitungen des ICAP von Virtex 5 FPGAs werden, im Gegensatz zu dem in Abschnitt 8.4.4.1 vorgestellten Virtex II Pro Design, bei dem nur acht Datenleitungen zum ICAP führen, erheblich kürzere Rekonfigurationszeiten erzielt. Hierzu trägt auch die mit 50 MHz, im Vergleich zu den 33 MHz beim ICAP des Virtex II Pro, höhere maximal erreichbare Taktfrequenz des ICAP beim Virtex 5 bei.

8.4.5 Verdrahtung

Wie bei der Synthese von Hardware-Designs für FPGAs üblich, so stellt auch für Laufzeitrekonfigurierbare EPIC Soft Core Prozessoren die effiziente Verdrahtung ein Optimierungsziel dar. Es ist eine Eigenheit dieser Prozessorklasse, dass mit der Zahl an Funktionseinheiten auch die Zahl an benötigten Zugängen zum Befehlsregister steigt. Zusätzlich dazu müssen einige globale Leitungen, beispielsweise für die Taktung, auch während der Rekonfiguration

anderer Verbindungen unversehrt bleiben.
Durch die weitgehend automatisierten Vorgänge beim Verdrahten besteht hier die hauptsächliche Herausforderung bei der Implementierung von Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren darin, so wenig Routing-Ressourcen wie möglich zu verwenden, ohne dabei an Effizienz zu verlieren.
Sehr komplexe Verdrahtung in einem rekonfigurierbaren System zeigt der Game of Life-Prototyp aus Abschnitt 8.3.6.

8.4.6 Prototyp: Synthese komplexer Soft Core Prozessoren

In der ihm Rahmen dieses Projektes angefertigten Studienarbeit [68] wird der Weg zur Synthese des Leon3 beleuchtet. Hierzu wird die Konfiguration des Prozessors so weit abgeändert, dass dieser auf einem Xilinx Virtex-II xc2v1000 passt. Nach Anbindung an den Speicher und passender Verdrahtung der Pins liegt ein funktionsfähiges System vor.
Als relevantes Ergebnis der Arbeit ist zu sehen, dass ein verhältnismäßig komplexer 32-Bit-Prozessor auf einem relativ kleinen FPGA arbeiten kann. Sowohl Logik als auch Routing-Ressourcen sind ausreichend.
Zudem wird in dieser Arbeit die Abhängigkeit der Größe des Designs vom Synthese-Tool bestätigt. Während der Prozessor mit den Tools von Xilinx zu groß für den eingesetzten FPGA wird, so genügen die Ressourcen in derselben Konfiguration bei der Synthese mittels des Tools Synplify Pro [116] von Synopsys.

8.5 Zusammenfassung

In diesem Kapitel wurden die Herausforderungen, welche bei der Implementierung von Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren zu bewältigen sind, identifiziert und analysiert.
Die praktische Anwendbarkeit der Lösungsansätze wurde durch prototypische Implementierungen nachgewiesen. Eine Zuordnung der Prototypen zu den Herausforderungen ist in Tabelle 8.4 gegeben. Somit wurde für jedes der eingangs identifizierten Probleme nicht nur ein theoretischer Lösungsansatz angegeben, sondern jeweils auch ein praktisches Beispiel erfolgreich umgesetzt.
Der Implementierung von Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren stehen also keine Hindernisse im Wege. Mit genügend Zeit und Aufwand ist ihre Umsetzung realisierbar. Vorschläge für weitere zukünftige Arbeiten und ein Ausblick folgen im nächsten Kapitel.

Tabelle 8.4: Zuordnung der Prototypen zu den Herausforderungen bei der Implementierung von Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren

Prototyp	**Herausforderung**
Trace-Erzeugung (Abschnitt 7.1.2)	EPIC-Code-Erstellung mit parallelisierendem Compiler
Befehlsverteilung und Ausführungsstatistiken (Abschnitt 8.2.3)	Variable Befehlsausführung
Game of Life (Abschnitt 8.3.6)	Relokation Vermeidung der Fragmentierung Pipelining, Rückgängigmachen von Befehlen Taktgenaue Einbindung von PR-Modulen Modulerkennung durch PARS Verdrahtung
Mehrere Prozessoren auf einem Befehlsstrom (Abschnitt 8.3.8)	Befehlsverteilung
ReconfGenerator (Abschnitt 8.4.2)	Erstellung rekonfigurierender Bitstreams
Vergleich externer und interner partieller Rekonfiguration (Abschnitt 8.4.4.1)	Selbstrekonfiguration
Rekonfiguration aktueller FPGAs über ICAP (Abschnitt 8.4.4.2)	Selbstrekonfiguration
Synthese komplexer Soft Core Prozessoren (Abschnitt 8.4.6)	Verdrahtung

9 Ausblick

Die letzten Kapitel haben sowohl den Nutzen als auch die Umsetzbarkeit von Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren nachgewiesen. Zukünftige FPGA-Bausteine und EPIC-Compiler mit verbesserten Eigenschaften öffnen zudem weitere Chancen für die Effizienz der vorgeschlagenen Prozessorklasse.
Bis zum realen Einsatz dieses recht weit gefassten Konzepts sind noch einige weitere Arbeiten nötig. Deren Ansatzpunkte werden ebenfalls im Folgenden aufgezeigt.

9.1 Nutzen aus zukünftigen Entwicklungen

Wie in den Kapiteln 2 und 3 dargelegt, sind sowohl für FPGAs als auch für EPIC-Prozessoren in Zukunft weitere Fortschritte in der Leistungsfähigkeit zu erwarten. Von den Entwicklungen in beiden Bereichen können auch Laufzeitrekonfigurierbare EPIC Soft Core Prozessoren profitieren.

9.1.1 Verbesserte FPGAs

FPGAs werden vielseitig weiterentwickelt. Dabei geht der Trend, bedingt durch die steigende Integrationsdichte, derzeit in Richtung Vergrößerung der Menge an programmierbaren Hardware-Ressourcen und Erhöhung der erreichbaren Taktfrequenzen. Durch den Bedarf an rekonfigurierbaren Software Defined Radios werden auch die Rekonfigurationsbandbreiten ständig erhöht und der Prozess zur Erstellung partiell rekonfigurierender Bitstreams ständig vereinfacht.
Laufzeitrekonfigurierbare EPIC Soft Core Prozessoren bieten durch ihre inhärente Skalierbarkeit einen Weg, die gesteigerten Hardware-Ressourcen zukünftiger FPGAs effizient in Rechenleistung umzusetzen. Höhere Taktfrequenzen resultieren dabei zusätzlich direkt in gesteigerter Rechenleistung. Wird er auch immer vorhanden sein, sinkt durch beides der Rechenleistungsunterschied zwischen Hard Core Prozessoren und Soft Core Prozessoren. Dies hebt wiederum die Anzahl der Systeme, in welchen der Einsatz der in dieser Arbeit vorgestellten Prozessoren sinnvoll ist.
Durch die einfacher werdende Erstellung partiell dynamisch rekonfigurierbarer Systeme ist davon auszugehen, dass deren Anzahl steigen wird. Der sinkende Aufwand zur Erstellung solcher Systeme wird sie auch für kleinere, weniger kostenintensive Projekte einsatzfähig machen. Diese Prognose steigert wiederum die Zahl an Systemen, welche von den hier vorgestellten

Prozessoren profitieren können.
Höhere Rekonfigurationsbandbreiten senken die Dauer einer Rekonfiguration. Dadurch wird für zukünftige rekonfigurierbare Systeme die dynamische Nutzung der Hardware rentabler. Diese Dynamik des Systems schafft für Laufzeitrekonfigurierbare EPIC Soft Core Prozessoren mehr temporär ungenutzte Ressourcen. Zudem steigt durch die sinkenden Rekonfigurationszeiten der Nutzen der Selbstoptimierung.

Um rekonfigurierbare Systeme zukünftig noch besser von der FPGA-Hardware zu unterstützen, wären einige derzeit nicht umgesetzte Verbesserungen denkbar. Zum einen könnte die Programmierung des Bausteins in mehreren übereinander liegenden Speicherschichten aufgebaut sein, wie bereits von den Firmen Tabula [45] und Tierlogic [80] vorgeschlagen. Jede Logik-Ressource hätte dadurch zeitgleich verschiedene Programmierungen, zwischen welchen ausgewählt werden kann. Dadurch könnte zwischen mehreren Konfigurationen des FPGAs zur Laufzeit umgeschaltet werden. Die Rekonfigurationszeit würde auf wenige Takte sinken. Allerdings wären die Kosten an Hardware hierfür wohl enorm.
Des Weiteren wäre es denkbar, die Struktur des FPGAs, ähnlich dem von RAM-Bausteinen, einheitlich zu gestalten. Dies würde die Möglichkeit bieten, die Zellen linear mit gleichartigen Adressen zu programmieren. Die Relokation eines Moduls wäre dann einfach durch Verschieben der Speicherinhalte auf dem Baustein oder durch Änderung der Adressen im Bitstream möglich.

9.1.2 Verbesserte Compiler

Wie bereits nachgewiesen, spielt der Compiler für die Effizienz von EPIC-Prozessoren eine entscheidende Rolle. Er muss aus dem Quellcode des Programms genügend hohe Instruktions-Level-Parallelität extrahieren, um die Funktionseinheiten des Prozessors so weit wie möglich auszulasten. Durch die Fortführung des Itanium-Projektes ist davon auszugehen, dass auch bei der Entwicklung der Compiler weitere Fortschritte gemacht werden.
Erreichen zukünftige Compiler einen höheren Grad an Parallelität der Programme, so kann sich auch der Speedup für die in dieser Arbeit eingeführten Prozessoren linear mit der Parallelität erhöhen. Ihre Parallelisierungstechniken sollten auch für Prozessoren mit mehr Funktionseinheiten als bei aktuellen Itanium-Prozessoren greifen. Somit wird sich auch die maximal rentable Größe Laufzeitrekonfigurierbarer EPIC Soft Core Prozessoren erhöhen.

9.2 Ansatzpunkte für weitere Arbeiten

Da diese Arbeit größtenteils konzeptuelle Untersuchungen über Laufzeitrekonfigurierbare EPIC Soft Core Prozessoren durchführt, schafft sie wesentliche Ansatzpunkte für weitere Arbeiten. Diese können auf den Grundlagen dieser Arbeit in erster Linie detailliertere Erkenntnisse über Teilaspekte des Konzepts sowie die technische Umsetzung liefern.

9.2.1 Länge und Rekonfiguration des Befehls- und Done-Flag-Registers

In dieser Arbeit wurde stets von einem genügend langen Befehlsregister, welches Bündellängen beliebiger Größe fassen kann, ausgegangen. Ebenso wurde für das Done-Flag-Register stets die passende Länge angenommen.
Eine interessante Fragestellung hierzu ist es, welche Länge hier optimal ist, um einen möglichst hohen Nutzen bei möglichst geringem Ressourcen-Aufwand zu erzielen. Diese optimale Länge steht in Abhängigkeit zu der Anzahl von Funktionseinheiten und der maximalen Bündellänge im Maschinenprogramm.
Dank der zugrundeliegenden rekonfigurierbaren Hardware wäre es denkbar, Done-Flags sowie die Länge des Befehlsregisters je nach der aktuellen Prozessorkonfiguration und des gerade ausgeführten Programms partiell dynamisch zu konfigurieren

9.2.2 Art und Anzahl der Typen von Funktionseinheiten

Auch die Art und die Anzahl der Typen von Funktionseinheiten wurden bisher in dieser Arbeit nicht festgelegt. Funktionseinheiten können sehr feingranular sein und nur wenige Befehle unterstützen, was zu geringerer Größe und zu mehr verschiedenen Typen führt. Ebenso können sie auch sehr mächtig sein und sehr viele Befehle unterstützen. In diesem Fall wären nur wenige Typen notwendig, welche jedoch pro Einheit höheren Ressourcen-Bedarf hätten.
Zudem wurde die Wortlänge des Prozessors für die durchgeführten Analysen außer Acht gelassen. Ob Laufzeitrekonfigurierbare EPIC Soft Core Prozessoren als 8-, 16-, 32- oder gar 64-Bit Prozessoren umgesetzt werden ist für die grundlegenden Eigenschaften und prozentuale Beschleunigungsmöglichkeit nicht relevant. Dies hat jedoch gravierenden Einfluss auf die absolute Rechenleistung und den Ressourcenbedarf des Prozessors.
Entscheidungen über die Funktionseinheitentypen und deren Wortbreite werden in erster Linie von der Anwendung sowie den vorhandenen Ressourcen abhängen. Eine weiterführende Arbeit könnte eine Entscheidungsmethodik zugrunde legen.

9.2.3 Strategien zur Ersetzung von Funktionseinheiten

In Kapitel 7 wurde betrachtet, welchen Nutzen die Konfiguration einer Funktionseinheit vom optimalen Typ hat. Zudem wurde analysiert, welchen Gewinn der Tausch einer Funktionseinheit des einen Typs gegen eine eines anderen Typs erreichen kann. Dabei wurden jeweils exhaustiv alle Typen getestet, um den optimalen zu bestimmen.
Dies ist aufgrund des Rechenaufwandes zur Laufzeit des Prozessors generell nicht durchführbar. Auch zur Compile-Zeit ist es nicht trivial, da hier weder der Rekonfigurationszeitpunkt noch die Konfiguration des Prozessors bekannt ist. Vor dem realen Einsatz von Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren ist es somit sinnvoll, zu analysieren, welche Ersetzungsstrategien für Funktionseinheiten die besten Ergebnisse liefern. Diese können durch den Compiler, durch spezielle Hardware oder durch eine Kombination beider realisiert werden.

Diese Arbeit liefert mit der Laufzeitberechnung anhand von Traces aus Kapitel 7 bereits eine Methodik, welche dazu genutzt werden kann, Strategien zur Auswahl der Funktionseinheit zu evaluieren. In Abschnitt 8.2.3 wurde zudem eine mögliche Strategie, nämlich Most-Recently-Used, prototypisch umgesetzt.

9.2.4 Umsetzung von Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren

Diese Arbeit liefert das Konzept sowie den Nachweis der Effizienz und der Umsetzbarkeit von Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren. Aufbauend auf sowohl diese als auch auf die weiteren vorgeschlagenen Forschungsthemen, wären weitere Analysen zur tatsächlichen Implementierung der Prozessoren nötig. Zusätzlich zu den bereits erwähnten Kernthemen sollten sie vor allem die Entscheidungen über eingesetzte Hardware, Entwurfsprozesse, Synthese-Werkzeuge sowie Compiler umfassen.

9.2.5 Senkung der Verlustleistung

Ein weiterführender Ansatz, welcher auf dem Konzept dieser Arbeit basiert, ist es, die Verlustleistung des Prozessors durch Einsatz von EPIC-Befehlen zu senken. Hierzu ist es denkbar, zu Zeiten geringer Rechenlast, Funktionseinheiten abzuschalten. Unterstützt die Hardware dieses Abschalten, verbrauchen die Einheiten in dieser Zeit keine Energie. Der Prozessor kann trotzdem mit reduzierter Rechenleistung weiterarbeiten und eine Grundperformanz aufrechterhalten. Dies entspricht dem Betrieb von Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren zu Zeiten, in welchen nur wenige Hardware-Ressourcen für zusätzliche Funktionseinheiten zur Verfügung stehen.
Auch dieser Ansatz kann anhand der in Kapitel 7 eingeführten Methodik bezüglich der Rechenleistung analysiert werden. Für die einsparbare Verlustleistung müssten jedoch neue Werkzeuge entwickelt und exemplarisch evaluiert werden.
Dieser Ansatz zum Senken der Verlustleistung ist unabhängig von der Implementierung des Prozessors in programmierbarer Hardware und könnte ebenso für Hard Core Prozessoren eingesetzt werden.

10 Zusammenfassung

In dieser Arbeit wurden Laufzeitrekonfigurierbare EPIC Soft Core Prozessoren zur Optimierung der Rechenleistung pro Fläche von Prozessorarchitekturen durch Rekonfiguration von Funktionseinheiten vorgestellt. Diese nutzen dynamisch freie Ressourcen von laufzeitrekonfigurierbaren Systemen, um effektiv ihre Rechenleistung zu erhöhen.
Zur Abrundung der Arbeit folgt eine Zusammenfassung. Abschließend werden die gewonnenen Erkenntnisse und gesammelten Ergebnisse resümiert.

10.1 Zusammenfassung der Arbeit

In Kapitel 2 wurde mit einem Überblick über rekonfigurierbare Hardware eines der dieser Arbeit zugrundeliegenden Konzepte erläutert. An der Analyse der aktuellen sowie der angekündigten FPGA-Bausteine der großen Hersteller ließ sich ein Trend in Richtung steigender Logik-Ressourcen und verbesserter Unterstützung partieller dynamischer Rekonfiguration erkennen.
Kapitel 3 diskutierte den Ansatz der EPIC-Prozessorarchitektur. Hierzu wurden nach einer Abgrenzung der EPIC-Architektur gegen andere Instruktions-Level-parallele Prozessorarchitekturen ihre grundlegenden Eigenschaften aufgezeigt. Im Speziellen wurden dabei die Methoden zur Parallelisierung sowie die Konzepte der flexiblen Befehlsgruppierung betrachtet. Darauf folgend wurde das Trimaran-Framework zur EPIC-konformen Kompilation und Simulation beschrieben. Am realen Beispiel des Intel Itanium-Prozessors ließ sich die Praktikabilität des Ansatzes, trotz der Schwierigkeiten mit passenden Compilern, erkennen. Die angekündigten zukünftigen Itanium-Prozessoren versprechen langfristige Investitionen der Industrie in die EPIC-Konzepte.
In Kapitel 4 wurden abgeschlossene und laufende Forschungsprojekte, welche partielle Laufzeitrekonfiguration für Prozessoren nutzen, vorgestellt und damit der Stand der Entwicklung aufgezeigt. Besonderes Augenmerk wurde dabei auf solche Ansätze gelegt, welche Teile der EPIC-Architektur verwenden. Diese nutzen jedoch entweder die Vorteile der partiellen Rekonfiguration nur wenig aus oder verlangen sehr komplexe Compiler- und Synthesewerkzeuge.
Aufbauend auf den vorhergehenden, stellte Kapitel 5 mit dem Konzept der Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren den Kern dieser Arbeit vor. Diese Prozessorklasse profitiert von der flexiblen Art der Verarbeitung von EPIC-Befehlsbündeln, um dynamisch konfigurierte Funktionseinheiten mit in die Berechnungen einzubeziehen. Dadurch wird er-

reicht, dass freie Ressourcen der programmierbaren Hardware für sinnvolle Rechenleistung genutzt werden. Das Konzept erlaubt zudem eine eigenständige Optimierung des Prozessors auf das zu berechnende Programm, indem es Funktionseinheiten verschiedener Typen gegeneinander tauscht.
Diese Prozessorklasse wurde in Kapitel 6 formalisiert. Die Berechnung der in Abhängigkeit von der Prozessorkonfiguration minimalen Laufzeit eines Programms auf einem statischen EPIC-Prozessor wurde dabei als Suchproblem identifiziert. Dessen Lösbarkeit wurde aufgezeigt und das Vorgehen zur Berechnung der Laufzeit eines Programms auf einem EPIC-Prozessor mit Rekonfiguration der Funktionseinheiten wurde formal dargelegt.
Kapitel 7 nutzte die Formalismen des vorangegangenen Kapitels, um den Nutzen von Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren zu quantifizieren. Um den Berechnungsaufwand gering zu halten, dienten Programm-Traces als Eingaben. Es wurde sowohl der relative Rechenleistungsgewinn bei Hinzunahme von Funktionseinheiten als auch der Nutzen der Selbstoptimierung bestimmt.
Da, wie in Kapitel 7 dargelegt, Laufzeitrekonfigurierbare EPIC Soft Core Prozessoren unter gewissen Voraussetzungen beträchtliche Rechenleistungssteigerungen erzielen können, beschäftigt sich Kapitel 8 mit deren Umsetzung. Hierzu wurden die hauptsächlichen Herausforderungen bei der Umsetzung solcher Prozessoren bestimmt. Jene, die nur bei dieser Klasse auftreten und nicht auch bei klassischen EPIC-Prozessoren auftauchen, wurden genauer analysiert. Für alle von ihnen wurden mögliche Lösungsansätze gegeben. Die Anwendbarkeit der Lösungsansätze wurde durch Prototypen belegt.
Kapitel 9 gab einen Ausblick auf den Effekt weiterer Entwicklungen im Bereich FPGAs und Compiler auf Laufzeitrekonfigurierbare EPIC Soft Core Prozessoren. Zudem wurden Ansatzpunkte für weitere Arbeiten in diesem Bereich gegeben.

10.2 Zusammenfassung der Ergebnisse

Die wichtigsten Ergebnisse und Erkenntnisse dieser Arbeit können wie folgt zusammengefasst werden:

- Die Prozessorklasse der Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren wird definiert und ihre Funktionsweise erläutert.
- Durch eine Formalisierung der Prozessorklasse wird es ermöglicht, die Programmlaufzeiten mit und ohne Rekonfiguration von Funktionseinheiten zu berechnen.
- Die Berechnung einer optimalen Konfiguration ist ein NP-hartes Suchproblem. Dieses ist jedoch bei klein gehaltenem Aufwand für die Berechnung der Laufzeit einer Konfiguration für eine realistische Anzahl an Funktionseinheitentypen, von ca. vier bis acht, und Slots, im Bereich von ca. 50, in annehmbarer Zeit berechenbar.
- Die Leistungsfähigkeit des Ansatzes ist essentiell vom Compiler abhängig. Je stärker dieser parallelisieren kann, umso besser können vorhandene Ressourcen genutzt werden.

- Aktuelle Compiler liefern für geeignete Benchmarks Programmcode, dessen Ausführung sich auf Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren mit vielen Ressourcen um etwa den Faktor vier, im Vergleich zur Ausführung auf minimaler Slot-Anzahl, beschleunigen lässt.
- Die sinnvolle maximale Slot-Anzahl bei der Verwendung aktueller Compiler liegt in etwa bei 16. Bei mehr Slots ist der Gewinn an Rechenleistung nur noch sehr gering.
- Das definierte Potential zur Selbstoptimierung gibt an, welchen maximalen Nutzen ein Selbstoptimierungsschritt eines Laufzeitrekonfigurierbaren EPIC Soft Core Prozessors haben kann.
- Bei genügend hohem Potential zur Selbstoptimierung wird bereits bei der Rekonfiguration einer Funktionseinheit eine beträchtliche, nahe am Optimum liegende Laufzeit erreicht.
- Die realen Kosten der Rekonfiguration einer Funktionseinheit, also die durch den Rekonfigurationsschritt verlorene Zeit, eines Laufzeitrekonfigurierbaren EPIC Soft Core Prozessors sind nur schwach von der Anzahl der Slots abhängig. Die verlorene Zeit beträgt ca. 10% der Rekonfigurationszeit.
- Die identifizierten Herausforderungen, die den Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren eigen sind, können gelöst werden. Die Umsetzbarkeit der Lösungen ist durch Prototypen nachgewiesen.
- Das entwickelte Verfahren Partiell Rekonfigurierbare Semaphoren, PARS, einzusetzen dient zur Identifizierung von Rekonfigurierbaren Modulen. Die Anwendbarkeit ist durch prototypische Implementierungen belegt.
- Weitere vorgeschlagene und belegte Konzepte zur Umsetzung umfassen die Vermeidung von Fragmentierung, die taktgenaue Einbindung von partiell rekonfigurierbaren Modulen sowie die Relokation von Modulen.
- Das entwickelte Tool ReconfGenerator dient zur Erstellung von rekonfigurierenden Bitstreams. Es automatisiert den Ablauf der Erstellung aller rekonfigurierender Bitstreams.
- Eine Methodik zur Messung von Rekonfigurationszeiten lässt auf Protokoll-Overheads schließen.

Diese Ergebnisse sind durch die Analyse des Konzepts der Laufzeitrekonfigurierbaren EPIC Soft Core Prozessoren entstanden. Einige von ihnen beschreiben jedoch Techniken, welche häufig auftretende Fragestellungen beim Aufbau von vielen dynamisch partiell rekonfigurierbaren Systemen lösen. Durch die steigende Leistungsfähigkeit und die Verbesserungen im Entwurfsprozess solcher Systeme wird der Anwendungsbereich der hier vorgestellten Lösungen stetig wachsen. Sie können dazu beitragen den Erfolg der dynamischen partiellen Rekonfiguration von FPGAs dauerhaft fortzuführen.

Anhang A

Ergebnisgraphen der Effizienztests

Um die Lesbarkeit der folgenden Graphen zu verbessern, sind die diskreten Werte der Berechnungen als kontinuierliche Kurven dargestellt.

A.1 Speedup für reale Benchmark-Traces

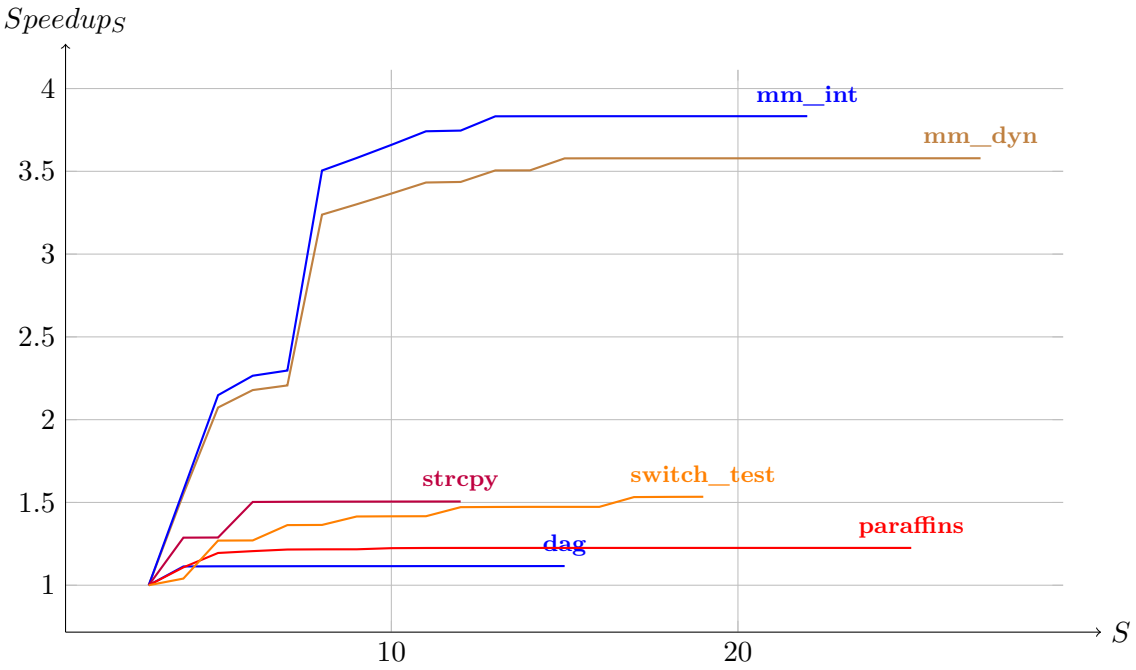


Abbildung A.1: Speedup bei erhöhter Anzahl an Slots für die Trimaran-Simple-Benchmarks, Teil 1

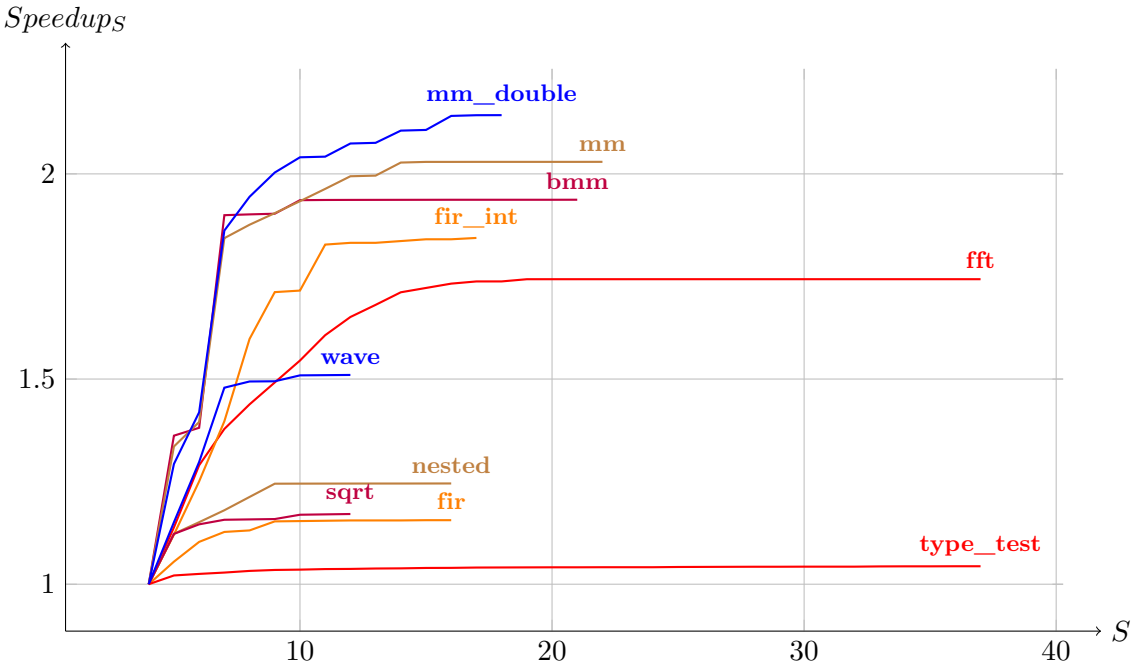


Abbildung A.2: Speedup bei erhöhter Anzahl an Slots für die Trimaran-Simple-Benchmarks, Teil 2

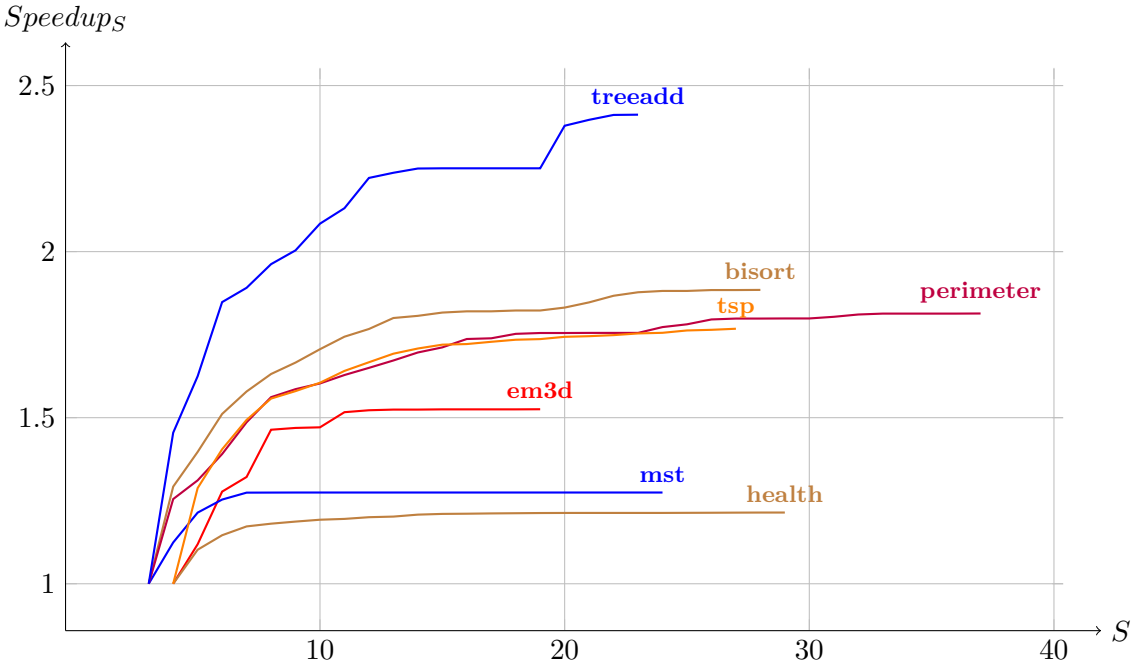


Abbildung A.3: Speedup bei erhöhter Anzahl an Slots für die Olden-Benchmarks

A.2 Speedup für gleichverteilte Eingaben

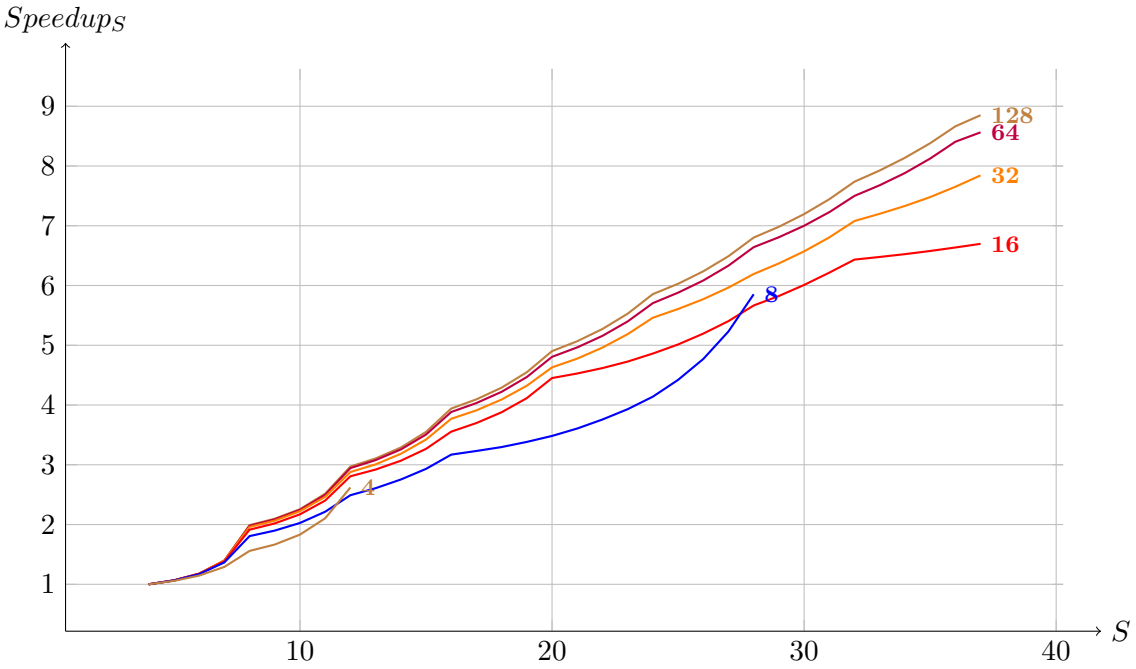


Abbildung A.4: Speedup bei erhöhter Anzahl an Slots für gleichverteilte zufällige Eingabedaten

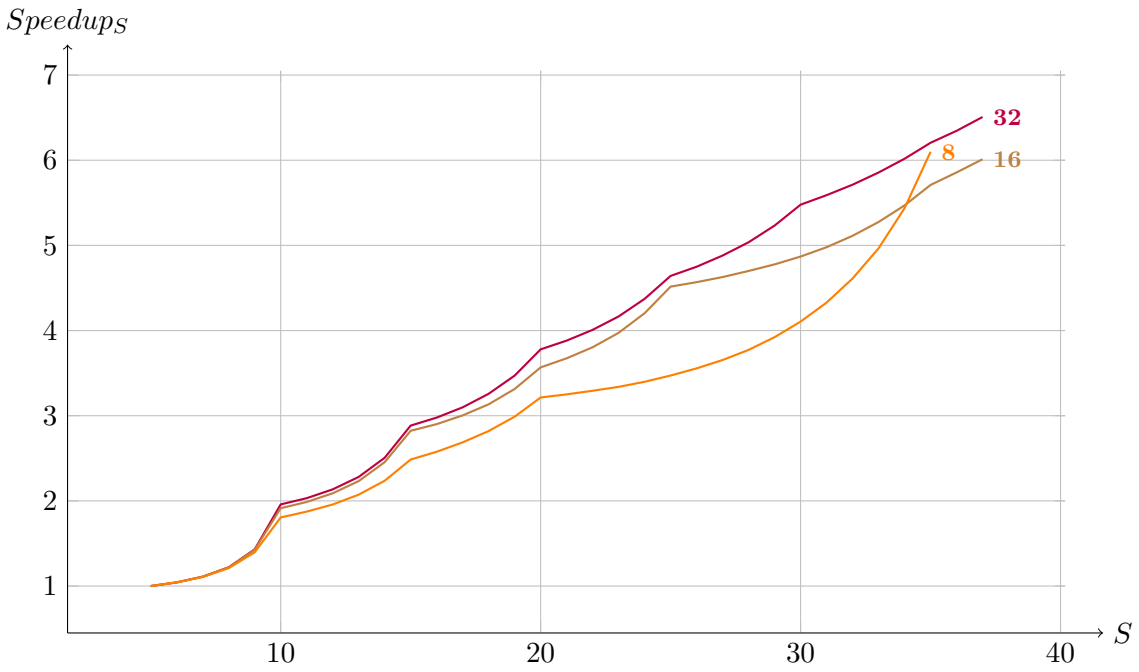


Abbildung A.5: Speedup bei gleichverteilten Eingaben für fünf Funktionseinheiten

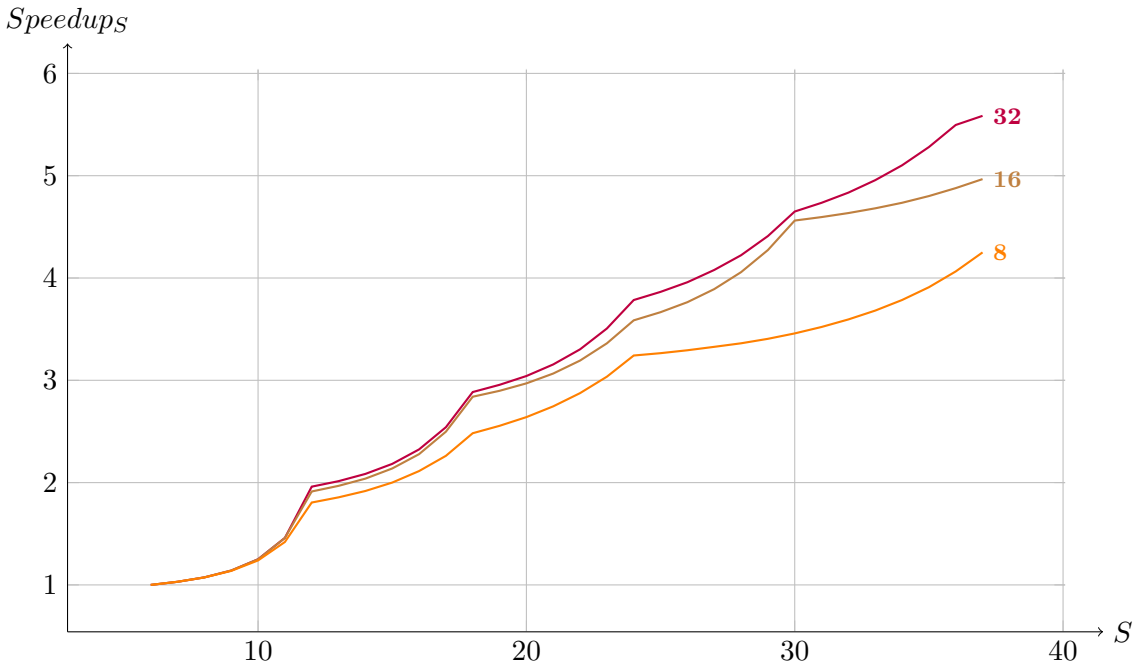


Abbildung A.6: Speedup bei gleichverteilten Eingaben für sechs Funktionseinheiten

A.3 Speedup für normalverteilte Eingaben

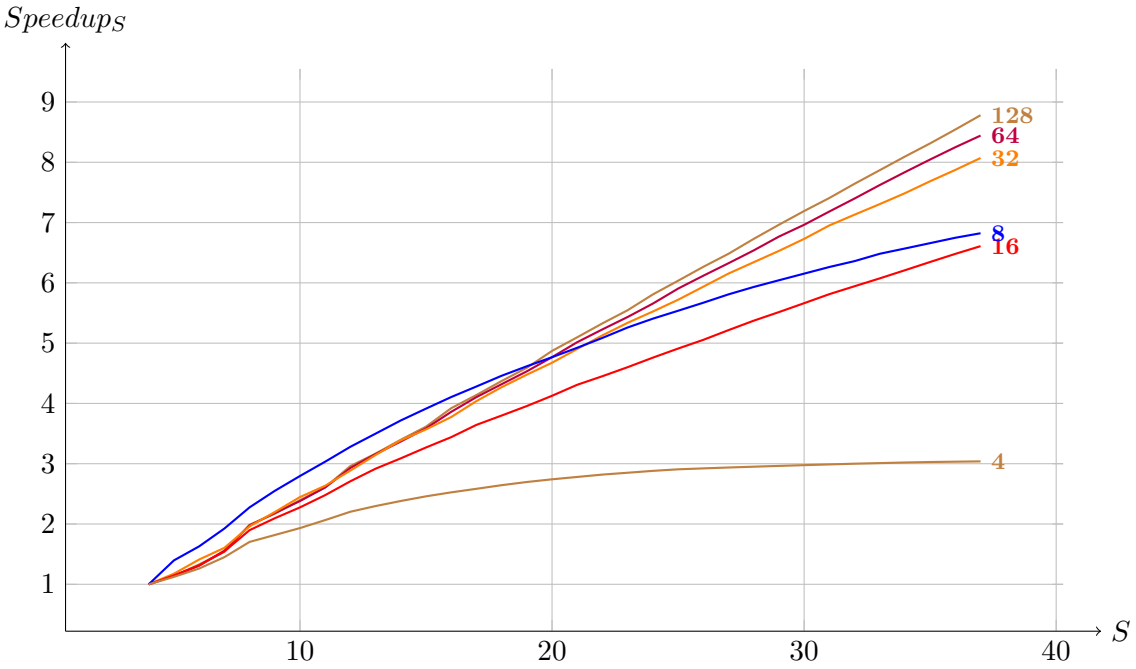


Abbildung A.7: Speedup bei erhöhter Anzahl an Slots für normalverteilte zufällige Eingabedaten

Anhang B

Potential zur Selbstoptimierung

B.1 Selbstoptimierungspotential bei sechs verfügbaren Slots

BM 1 / BM 2	bmm	fft	fir	fir_int	mm	mm_ double	nested	olden_ em3d	olden_ health	olden_ tsp	sqrt	type_ test	wave
bmm	0	0	12339	12335	0	0	0	0	0	12335	0	12335	0
	0,0%	0,0%	**5,0%**	**9,9%**	0,0%	0,0%	0,0%	0,0%	0,0%	**25,6%**	0,0%	**24,6%**	0,0%
fft	0	0	235932	196579	0	0	0	0	0	196579	0	196579	0
	0,0%	0,0%	**10,6%**	**9,5%**	0,0%	0,0%	0,0%	0,0%	0,0%	**9,9%**	0,0%	**9,9%**	0,0%
fir	4849	4849	0	8492	4849	4849	4849	4849	4849	8492	4849	8492	4849
	2,0%	0,2%	0,0%	**2,9%**	**2,0%**	**2,0%**	**2,2%**	**2,3%**	0,7%	**4,0%**	**2,4%**	**4,0%**	**2,3%**
fir_int	868	868	9262	0	868	868	868	868	868	0	868	0	868
	0,8%	0,0%	**3,2%**	0,0%	0,7%	0,7%	0,9%	**1,0%**	0,2%	0,0%	**1,1%**	0,0%	**1,0%**
mm	0	0	12783	12382	0	0	0	0	0	12382	0	12382	0
	0,0%	0,0%	**5,1%**	**9,6%**	0,0%	0,0%	0,0%	0,0%	0,0%	**23,7%**	0,0%	**22,8%**	0,0%
mm_ double	0	0	11982	11181	0	0	0	0	0	11181	0	11181	0
	0,0%	0,0%	**4,8%**	**8,7%**	0,0%	0,0%	0,0%	0,0%	0,0%	**21,6%**	0,0%	**20,8%**	0,0%
nested	0	0	2039	1663	0	0	0	0	0	1663	0	1663	0
	0,0%	0,0%	0,9%	**1,7%**	0,0%	0,0%	0,0%	0,0%	0,0%	**7,7%**	0,0%	**7,1%**	0,0%
olden_ em3d	0	0	2263	2227	0	0	0	0	0	2227	0	2227	0
	0,0%	0,0%	**1,1%**	**2,4%**	0,0%	0,0%	0,0%	0,0%	0,0%	**14,0%**	0,0%	**12,5%**	0,0%
olden_ health	0	0	15291	10165	0	0	0	0	0	10165	0	10165	0
	0,0%	0,0%	**2,3%**	**1,9%**	0,0%	0,0%	0,0%	0,0%	0,0%	**2,2%**	0,0%	**2,2%**	0,0%
olden_ tsp	290	290	295	0	290	290	290	290	290	0	290	0	290
	0,8%	0,0%	0,1%	0,0%	0,7%	0,7%	**1,4%**	**2,1%**	0,1%	0,0%	**5,7%**	0,0%	**2,7%**
sqrt	0	0	169	156	0	0	0	0	0	156	0	156	0
	0,0%	0,0%	0,1%	0,2%	0,0%	0,0%	0,0%	0,0%	0,0%	**3,2%**	0,0%	**2,3%**	0,0%
type_ test	4	4	13	0	4	4	4	4	4	0	4	0	4
	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%	0,1%	0,0%	0,0%
wave	0	0	1169	1170	0	0	0	0	0	1170	0	1170	0
	0,0%	0,0%	0,6%	**1,3%**	0,0%	0,0%	0,0%	0,0%	0,0%	**10,2%**	0,0%	**8,7%**	0,0%

B.2 Selbstoptimierungspotential bei zehn verfügbaren Slots

BM 1 / BM 2	**bmm**	**fft**	**fir**	**fir_int**	**mm**	**mm_ double**	**nested**	**olden_ em3d**	**olden_ health**	**olden_ tsp**	**sqrt**	**type_ test**	**wave**
bmm	0	438	436	436	414	414	436	437	9278	21620	413	436	395
	0,0%	0,0%	0,2%	0,5%	0,8%	0,8%	**1,1%**	**1,3%**	**2,0%**	**45,3%**	**1,7%**	**1,5%**	**1,3%**
fft	164011	0	76221	76221	57358	57358	76221	154076	35259	290852	96741	76221	127090
	9,8%	0,0%	**4,3%**	**4,7%**	**3,6%**	**3,6%**	**4,8%**	**9,3%**	**1,8%**	**16,3%**	**6,1%**	**4,8%**	**7,8%**
fir	13179	9407	0	0	9404	9404	0	128	3722	17028	125	0	9530
	5,8%	0,6%	0,0%	0,0%	**4,1%**	**4,2%**	0,0%	0,1%	0,6%	**8,1%**	0,1%	0,0%	**4,6%**
fir_int	14945	9262	0	0	9146	9146	0	128	4176	21445	12	0	5683
	15,5%	0,6%	0,0%	0,0%	**9,7%**	**9,8%**	0,0%	0,2%	0,9%	**25,9%**	0,0%	0,0%	**8,1%**
mm	384	481	79	79	0	0	79	481	8939	22124	0	79	382
	0,8%	0,0%	0,0%	0,1%	0,0%	0,0%	0,2%	**1,3%**	**1,9%**	**43,0%**	0,0%	0,2%	**1,2%**
mm_ double	1584	480	478	478	0	0	478	1280	9339	22125	800	478	782
	3,1%	0,0%	0,2%	0,6%	0,0%	0,0%	**1,2%**	**3,6%**	2,0%	**43,4%**	**2,9%**	**1,5%**	**2,4%**
nested	834	428	0	0	428	428	0	1	804	2899	2	0	430
	2,1%	0,0%	0,0%	0,0%	**1,0%**	**1,0%**	0,0%	0,0%	0,2%	**13,8%**	0,0%	0,0%	**2,0%**
olden_ em3d	692	44	11	11	44	44	11	0	1251	3565	11	11	34
	2,1%	0,0%	0,0%	0,0%	0,1%	0,1%	0,0%	0,0%	0,3%	**23,1%**	0,1%	0,1%	0,2%
olden_ health	17208	3104	2091	2091	5053	5053	2091	3334	0	20754	4018	2091	7179
	3,7%	0,2%	0,3%	0,4%	**1,1%**	**1,1%**	0,5%	0,8%	0,0%	**4,6%**	0,9%	0,5%	**1,6%**
olden_ tsp	699	52	218	218	255	255	218	349	32	0	352	218	388
	2,6%	0,0%	0,1%	0,4%	0,9%	0,9%	**1,2%**	**2,9%**	0,0%	0,0%	**7,5%**	**2,6%**	**4,2%**
sqrt	26	12	11	11	12	12	11	12	12	182	0	11	13
	0,1%	0,0%	0,0%	0,0%	0,0%	0,0%	0,1%	0,1%	0,0%	**4,0%**	0,0%	0,2%	0,2%
type_ test	52	21	0	0	20	20	0	5	4	35	11	0	31
	0,2%	0,0%	0,0%	0,0%	0,1%	0,1%	0,0%	0,0%	0,0%	0,4%	0,2%	0,0%	0,3%
wave	841	57	55	55	57	57	55	57	116	2130	57	55	0
	2,8%	0,0%	0,0%	0,1%	0,2%	0,2%	0,3%	0,4%	0,0%	**19,3%**	0,8%	0,5%	0,0%

B.3 Selbstoptimierungspotential bei vierzehn verfügbaren Slots

BM 1 / BM 2	bmm	fft	fir	fir_int	mm	mm_ double	nested	olden_ em3d	olden_ health	olden_ tsp	sqrt	type_ test	wave
bmm	0	402	400	400	0	422	445	0	420	9287	420	445	400
	0,0%	0,0%	0,2%	0,5%	0,0%	0,9%	**1,1%**	0,0%	0,1%	**26,4%**	**1,7%**	**1,5%**	**1,4%**
fft	126925	0	131002	122890	135161	185931	204776	135161	32766	163832	147453	204776	122890
	8,5%	0,0%	**7,9%**	**8,1%**	**9,0%**	**11,9%**	**13,1%**	**9,1%**	**1,8%**	**10,8%**	**9,9%**	**13,2%**	**8,3%**
fir	31	256	0	127	31	9536	4	31	129	3854	129	4	127
	0,0%	0,0%	0,0%	0,1%	0,0%	**4,2%**	0,0%	0,0%	0,0%	**2,0%**	0,1%	0,0%	0,1%
fir_int	128	129	0	0	129	12853	3579	129	3591	7883	3591	3579	0
	0,2%	0,0%	0,0%	0,0%	0,2%	**13,9%**	**4,9%**	0,2%	0,8%	**12,1%**	**6,0%**	**5,6%**	0,0%
mm	400	800	400	799	0	1221	1300	0	819	10160	819	1300	799
	0,8%	0,1%	0,2%	1,0%	0,0%	**2,4%**	**3,1%**	0,0%	0,2%	**26,7%**	**3,0%**	**4,1%**	**2,5%**
mm_ double	378	778	778	778	378	0	1278	378	798	9739	398	1278	778
	0,8%	0,1%	0,4%	**1,0%**	0,7%	0,0%	**3,1%**	**1,1%**	0,2%	**25,9%**	**1,5%**	**4,0%**	**2,4%**
nested	1	1	0	1	0	429	0	0	3	805	1	0	1
	0,0%	0,0%	0,0%	0,0%	0,0%	**1,1%**	0,0%	0,0%	0,0%	**4,3%**	0,0%	0,0%	0,0%
olden_ em3d	7	311	296	307	0	350	306	0	317	1557	317	306	307
	0,0%	0,0%	0,1%	0,5%	0,0%	**1,0%**	**1,3%**	0,0%	0,1%	**12,0%**	**3,1%**	**2,2%**	**2,1%**
olden_ health	5589	4789	4800	2281	6366	5169	3828	6366	0	2299	2081	3828	2281
	1,2%	0,3%	0,8%	0,5%	**1,4%**	**1,1%**	0,9%	**1,5%**	0,0%	0,5%	0,5%	0,9%	0,5%
olden_ tsp	355	194	352	259	396	255	212	396	51	0	215	212	259
	1,4%	0,0%	0,2%	0,4%	**1,4%**	0,9%	**1,2%**	**3,4%**	0,0%	0,0%	**4,9%**	**2,5%**	**2,9%**
sqrt	0	1	0	0	1	13	12	1	0	13	0	12	0
	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%	0,1%	0,0%	0,0%	0,3%	0,0%	0,2%	0,0%
type_ test	12	14	5	7	7	22	0	7	8	6	6	0	7
	0,0%	0,0%	0,0%	0,0%	0,0%	0,1%	0,0%	0,1%	0,0%	0,1%	0,1%	0,0%	0,1%
wave	0	0	0	0	0	61	59	0	59	120	59	59	0
	0,0%	0,0%	0,0%	0,0%	0,0%	0,2%	0,3%	0,0%	0,0%	**1,4%**	0,8%	0,5%	0,0%

Abkürzungsverzeichnis

ALAT	Advanced Load Address Table
ALM	Adaptive Logic Modules
ALU	Arithmetic-Logical Unit
AMBA	Advanced Microcontroller Bus Architecture
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction Set Processor
BIST	Built In Self Test
BM	Benchmark
CAD	Computer Aided Design
CGRA	Coarse-Grained Reconfigurable Array
CISC	Complex Instruction Set Computer
CLB	Configurable Logic Block
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DFS	digitaler Frequenzsynthesizer
DLL	Delay Locked Loop
DMU	Data Management Unit
DPR	Dynamisch Partielle Rekonfiguration
DSP	Digital Signal Processor
EDIF	Electronic Design Interchange Format
EEPROM	Electrically Erasable Programmable Read-Only Memory
EPIC	Explicitly Parallel Instruction Computing
EPROM	Erasable Programmable Read-Only Memory
FF	FlipFlop
FIFO	First In First Out
FPAA	Field Programmable Analog Array
FPGA	Field Programmable Gate Array
FU	Funtional Unit
GAG	Generic Address Generator
GAL	Generic Array Logic
GNU	GNU's Not Unix

GPL	GNU General Public License
GPP	General Purpose Processors
GPS	Global Positioning System
GSM	Global System for Mobile Communications
HDL	Hardware Description Language
HRE	Heterogeneous Reconfigurable Engines
I/O	Input / Output
IA-32	Intel 32-Bit Architektur
IA-64	Intel 64-Bit Architektur
IC	Integrated Circuit
ICAP	Internal Configuration Access Port
ID	Identifikator
IDE	Integrated Development Environment
ILP	Instruction Level Parallelism
IOB	Input/Output Block
IP	Intellectual Property
ISA	Instruction Set Architecture
ISP	In-System Programming
JPEG	Joint Photographic Experts Group
JTAG	Joint Test Action Group
LAB	Logic Array Block
LED	Light Emitting Diode
LPGL	GNU Lesser General Public License
LUT	Lookup Table
MAC	Multiply-Accumulate
MMU	Memory Management Unit
MPEG	Moving Picture Experts Group
MRLA	Multicontext Reconfigurable Logic Array
NaT	Not a Thing
NoC	Network-On-Chip
NOP	No Operation
NP	nichtdeterministisch polynomiell
PAE	Processing Array Element
PAL	Programmable Array Logic
PARS	Partiell Rekonfigurierbarer Semaphor
PCI	Peripheral Component Interconnect
PDA	Personal Digital Assistant
PFU	Programmable Functional Unit
PLA	Programmable Logic Array

PLD	Programmable Logic Device
PLL	Phase Locked Loop
PNG	Portable Network Graphics
POE	Plan of Execution
PROM	Programmable Read-Only Memory
pRTR	partial Runtime Rekonfiguration
PR	Partielle Rekonfiguration
PS	Phasenschieber
rALU	rekonfigurierbare ALU
RAM	Random Access Memory
RFU	Reconfigurable Functional Unit
RISC	Reduced Instruction Set Computer
RPU	Reconfigurable Processor Units
RTL	Register Transfer Level
RTR	Runtime Rekonfiguration
SDR	Software Defined Radio
SoC	System-on-a-Chip
SPARC	Scalable Processor ARChitecture
SR	Software Radio
SRAM	Static RAM
TLP	Thread-Level Parallelism
UHF	Ultra High Frequency
UMTS	Universal Mobile Telecommunication System
VHDL	VHSIC HDL
VHF	Very High Frequency
VHSIC	Very High Speed Integrated Circuit
VLIW	Very Long Instruction Word
VLSI	Very Large Scale Integration

Verzeichnis verwendeter Symbole

a_0	Anzahl nicht verwendeter Slots
a_i	Anzahl an Funktionseinheiten F_i einer Konfiguration C
a'_i	Anzahl an Funktionseinheiten F_i einer Konfiguration $C_{\square j}$
a''_i	Anzahl an Funktionseinheiten F_i einer Konfiguration C_{ij}
B	Ein Befehlsbündel
B_i	i-tes Befehlsbündel eines Programms
b_i	Anzahl der Befehle eines Bündels, welche von F_i ausgeführt werden können
b_{ij}	Anzahl der Befehle des Bündels B_j, welche von F_i ausgeführt werden können
$\mathcal{B}_i$	Menge der Indizes der Befehle eines Bündels, welche von F_i ausgeführt werden können
β_i	Einzelbefehl eines Befehlsbündels
C	Eine Konfiguration des Prozessors
$C_{\square j}$	Konfiguration C um eine Funktionseinheit vom Typ F_j reduziert
C_{ij}	Konfiguration C um eine Funktionseinheit vom Typ F_j reduziert und um eine vom Typ F_i erweitert
C_{opt}	Eine optimale Konfiguration des Prozessors
$\mathcal{C}_S$	Menge aller möglichen Konfigurationen mit maximal S Slots
F_i	Funktionseinheit vom Typ i
$f(C)$	Fitnessfunktion eines Suchproblems
Ω	Lösungsraum eines Suchproblems
P	Ein Programm
S	Anzahl der Slots
$T_C(P)$	Ausführzeit eines Programms P mit Konfiguration C
$t_C(B)$	Ausführzeit eines Bündels B mit Konfiguration C
$\mathcal{T}$	Hardwareabhängige Rekonfigurationszeit
$\theta(F_i)$	Ausführzeit der Befehle eines Bündels, welche von F_i ausgeführt werden können

Literaturverzeichnis

[1] Actel Corporation: *Actel: Devices*. Webseite. `http://actel.com/products/devices.aspx`, Stand: Mai 2011.

[2] Actel Corporation: *FPGA Array Architecture in Low-Power Flash Devices*, Juni 2008.

[3] Aeroflex Gaisler: *LEON3 Multiprocessing CPU Core*, April 2010.

[4] Aeroflex Gaisler: *LEON4 32-Bit Processor Core*, Januar 2010.

[5] Allen, J. R., Ken Kennedy, Carrie Porterfield und Joe Warren: *Conversion of Control Dependence to Data Dependence*. In: *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Seiten 177–189. ACM, 1983.

[6] Altera Corporation: *Stratix III FPGAs vs. Xilinx Virtex-5 Devices: Architecture and Performance Comparison*. White paper, Oktober 2007.

[7] Altera Corporation: *Arria GX Device Handbook, Volume 1*, Mai 2008.

[8] Altera Corporation: *Nios II C2H Compiler User Guide*, November 2009.

[9] Altera Corporation: *Cyclone IV Device Handbook, Volume 1*, Dezember 2010.

[10] Altera Corporation: *MAX V Device Handbook*, Dezember 2010.

[11] Altera Corporation: *Nios II Processor Reference Handbook*, Dezember 2010.

[12] Altera Corporation: *Stratix V Device Handbook, Volume 1*, Dezember 2010.

[13] ARM Limited.: *AMBA Specification (Rev 2.0)*, Mai 1999.

[14] Atmel Corporation: *5K – 50K Gates Coprocessor FPGA with FreeRAM*, Juli 2006.

[15] Atmel Corporation: *8-bit AVR Microcontroller with 128K Bytes In-System Programmable Flash: ATmega103, ATmega103L*, Februar 2007.

[16] Atmel Corporation: *FPSLIC AT94KAL Series Field Programmable System Level Integrated Circuit*, Januar 2008.

[17] Bapat, Shekhar: *EasyPath-6 Technology: Fast, Simple, Risk-Free FPGA Cost Reduction.* White paper, Xilinx Inc, November 2009.

[18] Baumgarte, Volker, G. Ehlers, Frank May, Armin Nückel, Martin Vorbach und Markus Weinhardt: *PACT XPP—A Self-Reconfigurable Data Processing Architecture.* The Journal of Supercomputing, 26(2):167–184, 2003.

[19] Berkovich, Efaim und Semyon Berkovich: *SNAPP Technologies Corporation's Processing Architecture - A Combinatorial Architecture for Instruction-Level Parallelism.* SNAPP Technologies Corporation, September 2005.

[20] Carlisle, Martin C. und Anne Rogers: *Software Caching and Computation Migration in Olden.* Technischer Bericht TR-483-95, Princeton University, 1995.

[21] Celoxica Limited: *Platform Developers Kit, RC200/203 Manual*, 2005.

[22] Chakrapani, Lakshmi N., John Gyllenhaal, Wen-mei W. Hwu, Scott A. Mahlke, Krishna V. Palem und Rodric M. Rabbah: *Trimaran: An Infrastructure for Research in Instruction-Level Parallelism.* In: *LCPC: Languages and Compilers for High Performance Computing*, Band 3602 der Reihe *Lecture Notes in Computer Science*, Seiten 32–41. Springer, 2005.

[23] Chu, W. W. S., Robert Dimond, S. Perrott, Shay Ping Seng und Wayne Luk: *Customisable EPIC Processor: Architecture and Tools.* In: *Conference on Design, Automation and Test in Europe*, Seiten 236–241, Washington, DC, USA, 2004. IEEE Computer Society.

[24] Clarke, Peter: *ST rolls Morpheus reconfigurable processor.* Programmable Logic Design Line, April 2009. `http://www.pldesignline.com/217100139`, Stand: Mai 2011.

[25] Cnet News: *Charts: Mining Itanium*, Dezember 2005. `http://news.cnet.com/2300-1006_3-5873647.html`, Stand: Mai 2011.

[26] DeHaven, Keith: *Extensible Processing Platform Ideal Solution for a Wide Range of Embedded Systems.* White paper, Xilinx Inc, April 2010.

[27] Dehnert, James C.: *Transmeta Crusoe and Efficeon: Embedded VLIW as a CISC Implementation.* SCOPES Invited Talk, 2003.

[28] Dehnert, James C., Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber und Jim Mattson: *The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges.* In: *CGO '03: Proceedings of the International Symposium on Code Generation and Optimization*, Seiten 15–24. IEEE, 2003.

[29] Dehnert, James C., Peter Y.-T. Hsu und Joseph P. Bratt: *Overlapped Loop Support in the Cydra 5.* In: *ASPLOS-III: Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, Seiten 26–38. ACM, 1989.

[30] Digital Equipment Corporation: *Digital Semiconductor Alpha 21064 and Alpha 21064A Microprocessors – Hardware Reference Manual*, Juni 1996.

[31] DRC Computer Corporation: *Cray Integrates DRC's Reconfigurable Coprocessor in Next Generation Hybrid Computing Platform*, November 2007.

[32] DRC Computer Corporation: *DRC Reconfigurable Processor Unit RPU110 Family*, 2007.

[33] EETimes Europe: *Toshiba licenses processor architecture, multiprocessor tools from IMEC.* Webseite, Oktober 2008. `http://www.eetimes.com/electronics-news/4193528/Toshiba-licenses-processor-architecture-multiprocessor-tools-from-IMEC`, Stand: Mai 2011.

[34] Ellwein, Christian: *Programmierbare Logik mit GAL und CPLD.* Oldenbourg Wissenschaftsverlag, Oktober 1998.

[35] Erickson, Charles R., Danesh Tavana und Victor A. Holen: *Encryption of Configuration Stream.* United States Patent 6,212,639, April 2001.

[36] Fisher, Joseph A., John R. Ellis, John C. Ruttenberg und Alexandru Nicolau: *Parallel Processing: A Smart Compiler and a Dumb Machine.* SIGPLAN Not., 19(6):37–47, 1984.

[37] Fisher, Joseph A., Paolo Faraboschi und Cliff Young: *Embedded Computing. A VLIW Approach to Architecture, Compilers and Tools.* Morgan Kaufmann, Januar 2005.

[38] Free Software Foundation, Inc.: *GNU General Public License, Version 3.* Webseite, Juni 2007. `http://www.gnu.org/licenses/gpl.html`, Stand: Mai 2011.

[39] Free Software Foundation, Inc.: *GNU Lesser General Public License, Version 3.* Webseite, Juli 2007. `http://www.gnu.org/licenses/lgpl.html`, Stand: Mai 2011.

[40] Gaisler, Jiri, Edvin Catovic, Marko Isomäki, Kristoffer Glembo und Sandi Habinc: *GRLIB IP Core User's Manual, Version 1.0.20.* Gaisler Research, Februar 2009.

[41] Gajski, Daniel D.: *Principles of Digital Design.* Prentice Hall, 1997.

[42] Gardner, Martin: *Mathematical Games: The fantastic combinations of John Conway's new solitaire game "life"*. Scientific American, 223:120–123, Oktober 1970.

[43] Girard, Olivier: *openMSP430 an MSP430 clone....*, August 2010.

[44] Gyllenhaal, John C., Wen-mei W. Hwu und B. Ramakrishna Rau: *HMDES Version 2.0 Specification*. IMPACT-96-03, University of Illinois, Februar 1996.

[45] Halfhill, Tom R.: *Tabula's Time Machine – Rapidly Reconfigurable Chips Will Challenge Conventional FPGAs*. Microprocessor Report, März 2010.

[46] Hartenstein, Reiner, Alexander Hirschbiel, Karin Schmidt und Michael Weber: *A Novel Paradigm of Parallel Computation and its Use to Implement Simple High Performance Hardware*. Future Generation Computer Systems, 7(2–3):181–198, 1992.

[47] Hauser, John R. und John Wawrzynek: *Garp: A MIPS Processor with a Reconfigurable Coprocessor*. In: *5th Annual IEEE Symposium on FPGAs for Custom Computing Machines*, Seiten 12–21, April 1997.

[48] Hsu, Peter Y. T. und Edward S. Davidson: *Highly Concurrent Scalar Processing*. In: *ISCA '86: Proceedings of the 13th Annual International Symposium on Computer architecture*, Seiten 386–395. IEEE, 1986.

[49] Huck, Jerry, Dale Morris, Jonathan Ross, Allan Knies, Hans Mulder und Rumi Zahir: *Introducing the IA-64 architecture*. IEEE Micro, 20(5):12–23, September/Oktober 2000.

[50] IBM Corporation: *Book E: Enhanced PowerPC Architecture*, Mai 2002.

[51] IEEE Computer Society: *IEEE Standard Test Access Port and Boundary-Scan Architecture*. IEEE Std 1149.1-2001(R2008) (Revision of IEEE Std 1149.1-1990), Juni 2001.

[52] IEEE Computer Society: *IEEE Standard Verilog Hardware Description Language*. IEEE Std 1364-2001, September 2001.

[53] IEEE Computer Society: *1076 IEEE Standard VHDL Language Reference Manual*. IEEE Std 1076-2002 (Revision of IEEE Std 1076, 2002 Edn), Mai 2002.

[54] Intel Corporation: *Intel Itanium Architecture Software Developer's Manual Volume 3: Instruction Set Reference*, Januar 2006.

[55] Intel Corporation: *Intel Core i7-900 Desktop Processor Extreme Edition Series and Intel Core i7-900 Desktop Processor Series – Datasheet*, Februar 2010.

[56] Intel Corporation: *Product Brief: Intel Itanium Processor 9300 Series*, 2010.

[57] Iseli, Cristian und Eduardo Sanchez: *Beyond Superscalar Using FPGAs.* In: *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, Seiten 486–490, Oktober 1993.

[58] Iseli, Cristian und Eduardo Sanchez: *Spyder: A Reconfigurable VLIW Processor using FPGAs.* In: *IEEE Workshop on FPGAs for Custom Computing Machines*, Seiten 17–24, April 1993.

[59] Jackson, Brian: *Partial Reconfiguration Design with PlanAhead.* Xilinx Inc, März 2008. Version 2.1.

[60] Jhonsa, Eric: *FPGAs, ASICs, and the Xilinx-Altera Duopoly.* The Digital Pathfinder FastNotes, 1(2), März 2004.

[61] Kathail, Vinod, Michael S. Schlansker und B. Ramakrishna Rau: *HPL-PD Architecture Specification: Version 1.1.* Hewlett-Packard Company, Februar 2000.

[62] Kathail, Vinod, Michael S. Schlansker und B. Ramakrishna Rau: *Compiling for EPIC Architectures.* Proceedings of the IEEE, 89(11):S. 1676–1693, 2001.

[63] Katoh, Kentaroh und Hideo Ito: *Built-In Self-Test for PEs of Coarse Grained Dynamically Reconfigurable Devices.* In: *European Test Symposium*, Seiten 69–74. IEEE Computer Society, Mai 2006.

[64] Kean, Thomas A.: *Configurable Logic : A Dynamically Programmable Cellular Architecture and its VLSI Implementation.* Doktorarbeit, University of Edinburgh, College of Science and Engineering, Juli 1989.

[65] Klaiber, Alexander: *The Technology Behind Crusoe Processors.* Transmeta Corporation, Januar 2000.

[66] Knuth, Donald E. und Andrew Binstock: *Interview with Donald Knuth.* InformIT, April 2008. `http://www.informit.com/articles/article.aspx?p=1193856`, Stand: Mai 2011.

[67] Kress, Rainer: *A Fast Reconfigurable ALU for Xputers.* Doktorarbeit, Universität Kaiserslautern, Fachbereich Informatik, Januar 1996.

[68] Kugler, Matthias Dietmar: *Synthetisierung eines Soft Core Prozessors.* Studienarbeit, Universität der Bundeswehr München, Fakultät für Informatik, Juni 2006.

[69] Kugler, Matthias Dietmar: *Erweiterung und Anpassung eines Tools zur Bitstromerstellung für partiell rekonfigurierbare FPGAs.* Diplomarbeit, Universität der Bundeswehr München, Fakultät für Informatik, Dezember 2007.

[70] Lam, Monica: *Software Pipelining: An Effective Scheduling Technique for VLIW Machines*. SIGPLAN Not., 23(7):318–328, 1988.

[71] Lam, Monica S. und Robert P. Wilson: *Limits of Control Flow on Parallelism*. In: *Proceedings of the 19th Annual International Symposium on Computer Architecture*, Seiten 46–57. ACM, 1992.

[72] Lampret, Damjan: *OpenRISC 1200 IP Core Specification*, September 2001.

[73] Lampret, Damjan et al.: *OpenRISC 1000 Architecture Manual*, Juli 2004.

[74] Lattice Semiconductor Corporation: *Lattice: FPGA Devices*. Webseite. `http://www.latticesemi.com/products/fpga/`, Stand: Mai 2011.

[75] Lattice Semiconductor Corporation: *ORCA Series 4 FPGAs: Datasheet*, Mai 2006.

[76] Lattice Semiconductor Corporation: *ispMACH 4000V/B/C/Z Family*, November 2007.

[77] Lee, Edward K. F. und P. Glenn Gulak: *A CMOS Field-Programmable Analog Array*. IEEE Journal of Solid-State Circuits, 26(12):1860–1867, Dezember 1991.

[78] Lee, Hsien-Hsin, Youfeng Wu und Gary Tyson: *Quantifying Instruction-Level Parallelism Limits on an EPIC Architecture*. In: *ISPASS: International Symposium on Performance Analysis of Systems and Software*, Seiten 21–27. IEEE, 2000.

[79] Lo, Jack L., Joel S. Emer, Henry M. Levy, Rebecca L. Stamm, Dean M. Tullsen und S. J. Eggers: *Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading*. ACM Transactions on Computer Systems, 15(3):322–354, 1997.

[80] Madurawe, Raminda Udaya: *Three Dimensional Integrated Circuits*. United States Patent 7,656,192, Februar 2010.

[81] McNairy, Cameron und Rohit Bhatia: *Montecito: A Dual-Core, Dual-Thread Itanium Processor*. IEEE Micro, 25(2):10–20, März/April 2005.

[82] McNairy, Cameron und Don Soltis: *Itanium 2 Processor Microarchitecture*. IEEE Micro, 23(2):44–55, März/April 2003.

[83] Mei, Bingfeng, Andy Lambrechts, Jean-Yves Mignolet, Diederik Verkest und Rudy Lauwereins: *Architecture Exploration for a Reconfigurable Architecture Template*. IEEE Design & Test of Computers, 22(2):90–101, March–April 2005.

[84] Mei, Bingfeng, Serge Vernalde, Diederik Verkest, Hugo De Man und Rudy Lauwereins: *ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix.* In: *13th International Conference on Field-Programmable Logic and Applications*, Band 2778 der Reihe *Lecture Notes in Computer Science*, Seiten 61–70. Springer, 2003.

[85] Memik, Gokhan, William H. Mangione-Smith und Wendong Hu: *NetBench: A Benchmarking Suite for Network Processors.* In: *International Conference on Computer Aided Design*, Seiten 39–42. IEEE Computer Society, 2001.

[86] Menon, Vijay S., Brian Murphy, Ali-reza Adl-tabatabai und Tatiana Shpeisman: *Method and Apparatus for Hardware Data Speculation to Support Memory Optimizations.* United States Patent Application 20,050,055,516, März 2005.

[87] Michel, Petra, Ulrich Lauther und Peter Duzy: *The Synthesis Approach to Digital System Design.* Kluwer Academic Publishers, 1992.

[88] Morris, Kevin: *Who's Winning? Early Returns in the FPGA Races.* FPGA and Structured ASIC Journal, November 2008. `http://www.eejournal.com/archives/articles/20081111_winning/`, Stand: Mai 2011.

[89] Morrison, Gordon E., Christopher B. Brooks und Frederick G. Gluck: *Parallel Processing Method and Apparatus for Increasing Processing Throughout by Parallel Processing Low Level Instructions Having Natural Concurrencies.* United States Patent 4,847,755, Juli 1989.

[90] Motomura, Masato: *A Dynamically Reconfigurable Processor Architecture.* In: *Microprocessor Forum*, Oktober 2002.

[91] Naprienko, Alexander: *Erstellung und Analyse von EPIC Programm-Traces.* Bachelorarbeit, Universität der Bundeswehr München, Fakultät für Informatik, Januar 2010.

[92] New York Times: Bits Blog: *A Decade Later, Intel's Itanium Chip Makes a Profit*, November 2009. `http://bits.blogs.nytimes.com/2009/11/17/a-decade-later-intels-itanium-chip-makes-a-profit/`, Stand: Mai 2011.

[93] Nicolau, Alexandru und Joseph A. Fisher: *Measuring the Parallelism Available for Very Long Instruction Word Architectures.* IEEE Trans. Comput., 33(11):968–976, 1984.

[94] OpenCores.org: *AVR Core :: Overview.* Webseite. `http://opencores.org/project,avr_core`, Stand: Mai 2011.

[95] OpenCores.org: *OpenCores*. Webseite. `http://opencores.org`, Stand: Mai 2011.

[96] PACT XPP Technologies: *XPP-III Processor Overview*, Juli 2006. White Paper.

[97] Percey, Andrew: *Advantages of the Virtex-5 FPGA 6-Input LUT Architecture*. White paper, Xilinx Inc, Dezember 2007.

[98] Przybus, Brent: *Xilinx Redefines Power, Performance, and Design Productivity with Three New 28 nm FPGA Families: Virtex-7, Kintex-7, and Artix-7 Devices*. White paper, Xilinx Inc, Juni 2010.

[99] Rave, Christopher Hans: *Befehls-Scheduler für EPIC-Prozessoren*. Studienarbeit, Universität der Bundeswehr München, Fakultät für Informatik, März 2006.

[100] Rave, Christopher Hans: *Entwurf eines modularen EPIC-Soft-Core-Prozessors*. Diplomarbeit, Universität der Bundeswehr München, Fakultät für Informatik, Februar 2007.

[101] Razdan, Rahul und Michael D. Smith: *A High-Performance Microarchitecture with Hardware-Programmable Functional Units*. In: *27th International Symposium on Microarchitecture*, Seiten 172–180, November 1994.

[102] Riedlinger, Reid, Rohit Bhatia, Larry Biro, Bill Bowhill, Eric Fetzer, Paul Gronowski und Tom Grutkowski: *A 32nm 3.1 Billion Transistor 12-Wide-Issue Itanium Processor for Mission-Critical Servers*. In: *2011 IEEE International Solid-State Circuits Conference*, Seiten 84–86. IEEE, Februar 2011.

[103] Rohde & Schwarz: *Software-basierte Funkgeräte - Überblick und Hardware-Aspekte*. Neues von Rohde&Schwarz, Heft 182, 2004.

[104] Rudd, Kevin W.: *VLIW Processors: Efficiently Exploiting Instruction Level Parallelism*. Doktorarbeit, Stanford University, Department of Electrical Engineering, Dezember 1999.

[105] Sauer, Johannes: *Partielle Selbst-Rekonfiguration von Virtex-5 FPGAs*. Diplomarbeit, Universität der Bundeswehr München, Fakultät für Informatik, Oktober 2008.

[106] Scheller, Patrick: *Dynamisch partielle Rekonfiguration für Software Defined Radios*. Diplomarbeit, Universität der Bundeswehr München, Fakultät für Informatik, Januar 2007.

[107] Schlansker, Michael S. und B. Ramakrishna Rau: *EPIC: An Architecture for Instruction-Level Parallel Processors*. Hewlett-Packard Company, Februar 2000.

[108] Schlansker, Michael S. und B. Ramakrishna Rau: *EPIC: Explicitly Parallel Instruction Computing*. IEEE Computer, 33(2):37–45, 2000.

[109] Scholz, Rainer: *Adapting and Automating XILINX's Partial Reconfiguration Flow for Multiple Module Implementations.* In: *Reconfigurable Computing: Architectures, Tools and Applications*, Band 4419 der Reihe *Lecture Notes in Computer Science*, Seiten 122–129. Springer, 2007.

[110] Scholz, Rainer und Klaus Buchenrieder: *Self Reconfiguring EPIC Soft Core Processors.* In: *Reconfigurable Computing: Architectures and Applications*, Band 3985 der Reihe *Lecture Notes in Computer Science*, Seiten 182–186. Springer, 2006.

[111] Seshan, Nat: *High VelociTI Processing.* IEEE Signal Processing Magazine, 15(2):86–101, 117, März 1998.

[112] Sharangpani, Harsh und Ken Arora: *Itanium Processor Microarchitecture.* IEEE Micro, 20(5):24–43, September/Oktober 2000.

[113] Sima, Dezsö: *The Design Space of Register Renaming Techniques.* IEEE Micro, 20(5):70–83, September/Oktober 2000.

[114] Smotherman, Mark: *Understanding EPIC Architectures and Implementations.* In: *The 40th Annual Southeast ACM Conference.* ACM, April 2002.

[115] Stackhouse, Blaine, Sal Bhimji, Chris Bostak, Dave Bradley, Brian Cherkauer, Jayen Desai, Erin Francom, Mike Gowan, Paul Gronowski, Dan Krueger, Charles Morganti und Steve Troyer: *A 65 nm 2-Billion Transistor Quad-Core Itanium Processor.* IEEE Journal of Solid-State Circuits, 44(1):18–31, Januar 2009.

[116] Synopsys, Inc.: *Synplify Pro The FPGA Synthesis Solution Of Choice – Datasheet*, 2010.

[117] Talla, Surendranath: *Adaptive Explicitly Parallel Instruction Computing.* Doktorarbeit, New York University, 2000.

[118] Texas Instruments, Incorporated: *MSP430x1xx Family User's Guide*, 2006. SLAU049F.

[119] Thoma, Florian, Matthias Kühnle, Philippe Bonnot, Elena Moscu Panainte, Koen Bertels, Sebastian Goller, Axel Schneider, Stéphane Guyetant, Eberhard Schüler, Klaus D. Müller-Glaser und Jürgen Becker: *MORPHEUS: Heterogeneous Reconfigurable Computing.* In: *International Conference on Field Programmable Logic and Applications*, Seiten 409–414, August 2007.

[120] TOP500.Org: *TOP500 Supercomputing Sites.* Webseite. `http://www.top500.org/`, Stand: Mai 2011.

[121] Triebel, Walt und Joe Bissell: *Scaling Itanium Architecture for Higher Performance*, Oktober 2008. `http://software.intel.com/en-us/articles/scaling-itaniumr-architecture-for-higher-performance`, Stand: Mai 2011.

[122] Trimaran Consortium, The: *An Overview of the Trimaran Compiler Infrastructure*, November 1999. `http://www.trimaran.org/documentation.shtml`, Stand: Mai 2011.

[123] Truong, Dan: *Olden benchmarks*. Webseite, 1999. `http://www.irisa.fr/caps/people/truong/M2COct99/Benchmarks/Olden/`, Stand: Mai 2011.

[124] Vassiliadis, Stamatis, Stephen Wong und Sorin Coţofană: *The MOLEN ρμ-coded Processor*. In: *11th International Conference on Field-Programmable Logic and Applications*, Band 2147 der Reihe *Lecture Notes in Computer Science*, Seiten 275–285. Springer, August 2001.

[125] Vecchi, Mario P. und Scott Kirkpatrick: *Global Wiring by Simulated Annealing*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2(4):215 – 222, Oktober 1983.

[126] Waingold, Elliot, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe und Anant Agarwal: *Baring It All to software: Raw machines*. IEEE Computer, 30(9):86–93, September 1997.

[127] Wainwright, Robert T.: *Life is universal!* In: *Proceedings of the 7th Conference on Winter Simulation*, Seiten 449–459. ACM, 1974.

[128] Wall, David W.: *Limits of Instruction-Level Parallelism*. In: *ASPLOS-IV: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Seiten 176–188. ACM, 1991.

[129] Weaver, David L. und Tom Germond: *The SPARC Architecture Manual - Version 9*. SPARC International, Inc., 1994.

[130] Xilinx Inc: *Xilinx: Silicon Devices*. Webseite. `http://www.xilinx.com/products/devices.htm`, Stand: Mai 2011.

[131] Xilinx Inc: *XC6200 Field Programmable Gate Arrays – Advance Product Specification*, Juni 1996.

[132] Xilinx Inc: *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, Juni 2004. XILINX DS083.

[133] Xilinx Inc: *CoolRunner-II CPLD Family*, September 2008. XILINX DS090.

[134] Xilinx Inc: *Early Access Partial Reconfiguration User Guide*, September 2008. XILINX UG208 (v1.2).

[135] Xilinx Inc: *Virtex-5 FPGA User Guide*, September 2008. XILINX UG190.

[136] Xilinx Inc: *EDK Concepts, Tools, and Techniques: A Hands-On Guide to Effective Embedded System Design*, Oktober 2009. XILINX UG200.

[137] Xilinx Inc: *Embedded Processor Block in Virtex-5 FPGAs*, Juni 2009. XILINX UG683.

[138] Xilinx Inc: *Virtex-6 FPGA Configurable Logic Block User Guide*, September 2009. XILINX UG364.

[139] Xilinx Inc: *7 Series FPGAs Overview – Advance Product Specification*, November 2010. XILINX DS180.

[140] Xilinx Inc: *MicroBlaze Processor Reference Guide, Embedded Development Kit EDK 12.2*, Juli 2010. XILINX UG081.

[141] Xilinx Inc: *PicoBlaze 8-bit Embedded Microcontroller User Guide for Spartan-3, Spartan-6, Virtex-5, and Virtex-6 FPGAs*, Januar 2010. XILINX UG129.

[142] Xilinx Inc: *Spartan-6 Family Overview*, November 2010. XILINX DS160.

[143] Xilinx Inc: *Virtex-6 Family Overview*, Januar 2010. XILINX DS150.

[144] Ye, Zhi Alex, Andreas Moshovos, Scott Hauck und Prithviraj Banerjee: *CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit.* In: *27th International Symposium on Computer Architecture*, Seiten 225–235, 2000.

[145] Yeager, Kenneth C.: *The Mips R10000 Superscalar Microprocessor.* IEEE Micro, 16(2):28–41, April 1996.

Tabellenverzeichnis

Abbildungsverzeichnis