Anwendungsorientierte FPGA Architekturen

Virtual Coarse Grained Reconfigurable Array für die Berechnung Arithmetischer Operation

Von der Fakultät Fakultät 1 – MINT – Mathematik, Informatik, Physik, Elektro- und Informationstechnik der Brandenburgischen Technischen Universität Cottbus–Senftenberg genehmigte Dissertation zur Erlangung des akademischen Grades eines Doktors der Ingenieurwissenschaften (Dr.-Ing.)

vorgelegt von

André Werner

geboren am 04.07.1986 in Greiz, Thüringen

Vorsitzende/r: Vertretungsprofessor Dr.-Ing. Marc Reichenbach

Gutachter/in: Prof. Dr.-Ing. habil. Michael Hübner

Gutachter/in: Prof. Dr.-Ing. Holger Blume

Tag der mündlichen Prüfung: 01.02.2023

DOI: 10.26127/BTUOpen-6248

A. Inhaltsverzeichnis

B. Abbildungsverzeichnis

C. Tabellenverzeichnis

D. Quellcodeverzeichnis

E. Abkürzungsverzeichnis

Abb. Abbildung

ALU Arithmetic Logical Unit

ARM Advanced RISC Machine

ASIC Application Specific Integrated Circuit

ASIP Application Specific Instruction Set Processor

BLE Basic Logic Element

blif Berkeley Logic Interchange Format

BRAM Block-Read-Access-Memory

CAD Computer-Aided Design

CB Connection Block

CCU Central Control Unit

CGRA Coarse Grained Reconfigurable Array

CISC Complex Instruction Set Computer

CLB Configurable-Logic-Block

CLI Command Line Interface

CPU Central Processing Unit

DCS Dynamic Circuit Specialization

DSP Digital Signal Processor

File engl. Datei

FPCA Field Programmable Crossbar Array

FPFA Field Programmable Function Array

FPGA Field Programmable Gate Array

GPP General Purpose Processor

JIT Just-In-Time

LSB Least Significant Bit

LUT Lookup-Table

MAC Multiply-And-Accumulate

M.I.T Massachusetts Institute of Technologie

MMU Memory Management Unit

MSB Most Significant Bit

MWTA Minimum Width Transistor Area

No. Numero (Nummer in Verbindung mit einer Zahl)

NoC Network-on-Chip

op Operant

PE Processing Element

PGM Portable Gray Map

PL Programmable Logic

PPC Partial Parameterized Configuration

PP&R Packing, Placement and Routing

PS Prozessorsystem

RISC Reduced Instruction Set Computer

RTTI Run Time Type Identification

SB Switch Block

SIMD Single Instruction Multiple Data

SISD Single Instruction Single Data

SoC System on Chips

SPM Scratchpad Memory

Tab. Tabelle

TC Template Configuration

TLM Transaction-Level-Modeling

VCGRA Virtual Coarse Grained Reconfigurable Array

VDR Virtual Dynamic Reconfigurable

VHDL Very High Speed Integrated Circuit Hardware Description Language

VLIW Very Long Instruction Word

VPR Versatile Place and Route

VTR Verilog-To-Routing

XML Extensible Markup Language

F. Inhaltsangabe

Sowohl in Anlagen mit Hochgeschwindigkeitsrechensystemen als auch in der Industrie in so genannten smarten Problemlösungen spielen Rechenbeschleuniger eine immer bedeutendere Rolle. Die Leistung der digitalen Rechner soll immer mehr gesteigert werden, während gleichzeitig der Energieverbrauch sinken soll. Diese wissenschaftliche Arbeit befasst mich mit der Untersuchung anwendungsorientierter FPGA-Architekturen, um die Lücke zwischen Performanz und Energieverbrauch zu verkleinern oder gar zu schließen.

In etablierten digitalen Lösungen wird versucht einen Algorithmus oder eine Applikation durch Programmiersprachen an ein bestehendes Hardwaresystem anzupassen. Es sind bereits unterschiedliche Hardwarelösungen für unterschiedliche *Usecases* verfügbar wie *General Purpose* Prozessoren oder Digitale Signal Prozessoren. Allerdings passen deren innere Strukturen oftmals nicht zum Algorithmusaufbau. Vielmehr greift die Arbeit die Idee Reiner Hartensteins [1] auf, eine Hardware an einen Algorithmus anzupassen. Diese Idee ist nicht neu, neu ist, dass durch Rekonfiguration in Echtzeit die Flexibilität und Anwendbarkeit dieser Architektur erhöht werden soll.

Diese Arbeit untersucht zunächst die optimale Granularität einer Architektur für einen Algorithmus. Anhand der Auswertung einer umfangreichen Versuchsreihe mit ca. 650 Simulationen und etwa 200 verschiedenen Architekturen wird versucht einen FPGA mit optimalen Eigenschaften für eine Beschleunigerhardware abzuleiten. Für die Untersuchung wird das Evaluationswerkzeugs *Verilog-To-Routing* genutzt, dessen Steuerung nebst dem Parsen der Ergebnisse mittels Python automatisiert wird.

Basierend auf den Ergebnissen dieser Granularitätsuntersuchung wird die Einführung einer parameterisierbaren CGRA-Architektur beschrieben, welche als *Overlay* auf einer beliebigen FPGA-Architektur aufgebracht werden kann. Zur Steigerung der Rekonfigurationsgeschwindigkeit wird versucht das eingeführte *Virtual Coarse Grained Reconfigurable Array* mittels der Anpassung an den *TLUT/TCON Toolflow* [2] einer Partneruniversität im Rahmen des gemeinsamen EXTRA-Projektes [3] zu optimieren. Ein *Usecase* in diesem Projekt war ein bildverarbeitender Algorithmus für die Segmentierung von Blutgefäßen in der Retina. Aus diesem Grund wird in der Arbeit anhand einer Faltung zur Kantenerkennung das Design des *Virtual Coarse Grained Reconfigurable Arrays* untersucht.

Die ersten Ergebnisse weisen auf Probleme beim Datentransfer zwischen einem Prozessorsystem und dem VCGRA als Beschleuniger hin, weshalb in zusätzliche

Module für eine Entspannung dieses Flaschenhalses investiert wird. Aus den Überlegungen entstehen so genannte *Pre-Fetcher* und Datenpuffer, eine *Memory Management Unit* und eine zentrale Steuerstelle, die *Central Control Unit,* welche das VCGRA zu einer Gesamtarchitektur ergänzen. Die Entwicklungsschritte und Evaluation dieser zusätzlichen Module erfolgt nun in SystemC, statt in VHDL, um den Zyklus von Design und Evaluation zu verkürzen. Die Ausführung eines Compilervorgangs ist um ein Vielfaches kürzer als die ständige Synthese neuer Designs, weshalb neben der neuen Architektur auch das Zukunftsziel einer Explorationsplattform entstand und am Ende der Dissertation beschrieben ist. Die Akzeptanz von *Coarse Grained Reconfigurable Arrays* mangelt nach Recherchen des Verfassers auch an Werkzeugen für Entwicklung und Evaluation für Entwickler auf diesen Plattformen.

Im Zusammenhang mit der *Central Control Unit* wird ein Maschinencode im Binärformat und ein passender Assemblerbefehlssatz entworfen, um das VCGRA autark zu steuern. Alle Werkzeuge und die Architektur stehen unter Open-Source-Lizenzen der wissenschaftlichen Community zur Verfügung.

G. Abstract[1]

Computing accelerators are playing an increasingly important role both in plants with high-speed computing systems as well as in industry in so-called smart problem solutions. The performance of digital computers is to be increased more and more, while at the same time the energy consumption is to be reduced. This scientific work deals with the investigation of application-oriented FPGA architectures in order to reduce or even close the gap between performance and energy consumption.

In established digital solutions, programming languages are used to try to adapt an algorithm or an application to an existing hardware system. There are already different hardware solutions available for different use cases like general purpose processors or digital signal processors. However, their internal structures often do not match the algorithm structure. Instead, this work takes up Reiner Hartenstein's [1] idea of adapting a hardware to an algorithm. This idea is not new, what is new is that real-time reconfiguration should increase the flexibility and applicability of this architecture.

This work first investigates in the optimal granularity of an architecture for an algorithm. Based on the evaluation of an extensive series of experiments with about 650 simulations and about 200 different architectures, an attempt is made to derive an FPGA with optimal characteristics for an accelerator hardware. For the investigation the evaluation tool Verilog-To-Routing is used, whose control is automated by means of Python in addition to the parsing of the results.

Based on the results of this granularity investigation, the introduction of a parameterizable CGRA architecture is described, which can be applied as an overlay on any FPGA architecture. To increase the reconfiguration speed, an attempt is made to optimize the introduced *Virtual Coarse Grained Reconfigurable Array* by means of adaptation to the TLUT/TCON toolflow [2] of a partner university in the context of the joint EXTRA project [3]. One usecase in this project was an image-processing algorithm for retinal blood vessel segmentation. For this reason, the work uses convolution for edge detection to investigate the design of the *Virtual Coarse Grained Reconfigurable Array*.

[1] Translated with www.DeepL.com/Translator (free version)

The first results indicate problems with the data transfer between a processor system and the VCGRA as an accelerator, so additional modules are invested to relax this bottleneck. From these considerations, so-called pre-fetchers and data buffers, a memory management unit and the central control unit, are created, which complement the VCGRA to form an overall architecture. The development steps and evaluation of these additional modules is now done in SystemC, instead of VHDL, to shorten the cycle of design and evaluation. The execution of a compiler process is many times shorter than the constant synthesis of new designs, which is why, in addition to the new architecture, the future goal of an exploration platform was also created and is described at the end of the dissertation. The acceptance of Coarse Grained Reconfigurable Arrays also lacks tools for development and evaluation for developers on these platforms, according to the author's research.

In the context of the central control unit, a machine code in binary format and a matching assembly instruction set are designed to control the VCGRA autonomously. All tools and the architecture are available under open source licenses to the scientific community.

1 Motivation

1.1 Die digitale Revolution

Ähnlich der industriellen Revolution im 19. Jahrhundert schreitet die digitale Revolution seit den 70iger Jahren des 20. Jahrhunderts mit wachsender Geschwindigkeit voran. Die Erfindung von Relais und Röhren durch Visionäre wie Konrad Zuse oder Alan Turing legten die Grundlagen für unser heutiges digitales Zeitalter. Mit der Erfindung von Transistoren 1948 durch Bardeen, Brattain und Shockley wurde die Entwicklung der digitalen Verarbeitungssysteme nochmals beschleunigt. Bereits 1950 erschien am M.I.T der erste Rechner basierend auf Transistoren statt Röhren. Die Entwicklung integrierter Schaltkreise durch Robert Noyce 1958 und des EPROM als nichtflüchtiges Speichermedium wurde bereits 1969 von IBM der erste Mikroprozessor vorgestellt, der 4004. Er war frei programmierbar, bestand aus einer 4-Bit ALU, 4-Bit Registern und konnte einen Arbeitsspeicher von vier Kilobyte ansprechen. In den 70iger Jahren folgten mehrere Weiterentwicklungen inklusive des sehr erfolgreichen 8086, den ersten 16-Bit-Mikroprozessor. Durch die gefallenen Kosten für die Herstellung von Rechnersystemen wurden sie auch für den kleinen Mann erschwinglich. Heute sind Strom und digitale Rechensysteme aus keinem Bereich unseres sozialen Lebens mehr wegzudenken, ob Medizin, Entertainment, industrielle Produktion oder Landwirtschaft. Längst nehmen Maschinen Menschen immer mehr komplexe Aufgaben ab oder vernetzen verschiedene Disziplinen der Produktion. Wo zur industriellen Revolution Maschinen die körperlichen Kräfte des Menschen steigerten, erreichen digitale Rechnersysteme dies heute mit unserer geistigen Leistungsfähigkeit und unserem Komfort. Mindestens seit der Einführung „smarter“ (intelligenter) Produktionsanlagen oder Stromverteilungsanlagen befindet sich in nahezu jedem elektrisch betriebenen Gerät eine digitale Rechenmaschine beliebiger Art.

Die Entwicklung neuer Rechnersysteme verlief in den vergangenen Jahren sehr gleichmäßig, wobei sich die Anzahl der Transistoren innerhalb der integrierten Schaltung alle 18 Monate verdoppelte. Diese Feststellung wurde 1965 von Gorden Moore erbracht und deswegen als Moore‘sches Gesetz bezeichnet. Heute befinden sich in modernen Rechnersystemen mehr als 1 Milliarde Transistoren und die Taktraten erreichen mehr als 3GHz. Durch physikalische Bedingungen sind aber den Bauformen und Taktraten Grenzen auferlegt. Mit steigender Frequenz steigt die Verlustleistung der integrierten Schaltungen und durch Reduktion der Größe der Transistoren bis auf Atomgröße der verwendeten Siliziumatome müssen andere

physikalische Phänomene beachtet werden als bisher. Durch die kleinen Fertigungstechnologien im Bereich von 14 Nanometer und kleiner wächst der Aufwand im Produktionsprozess. Schon kleinste Verunreinigungen können den Elektronenfluss negativ beeinflussen. Bei noch kleineren Fertigungsprozessen werden Photonen statt Elektronen für den Informationsfluss verwendet. Diese sind für Strahlungseinflüsse durch Röntgen oder Sonnenstrahlung empfindlicher. Mit dem Reduzieren der Fertigungsgröße sollen beispielsweise auf die gleiche Chipfläche mehr Recheneinheiten platziert werden, sodass diese in parallel mehr Aufgaben ausführen können. Die wachsende Komplexität der Rechenaufgaben und der damit verbundene Energieverbrauch bedingen die Entwickelung so genannter Quantenrechner neben den traditionellen digitalen Binärsystemen, welche nicht nur zwei, sondern vier konstante Zustände für die Darstellung von Informationen besitzen. Solche Quantenrechner sind zwar für den Anwender zu Hause noch Zukunftsmusik, dennoch werden in der Forschung bereits komplexe Algorithmen durch die Berechnung mittels Quantencomputern um ein Vielfaches beschleunigt. So hat zuletzt Google in 2019 mit seinem Quantenrechner „Sycamore“ ein hochkomplexes Problem der statistischen Quantenphysik in etwa 200 Sekunden gelöst, während ein klassischer Supercomputer 10.000 Jahre benötigt hätte [4]. Ich möchte mich zunächst mit den gängigen Architekturen in der digitalen, Bool'schen Rechentechnik beschäftigen und trotz der jüngsten Durchbrüche zunächst die Quantenrechner ausklammern.

1.2 Hardware-Software Design-Space

Zur Umsetzung eines Algorithmus oder einer Aufgabe stehen einem Entwickler mehrere Möglichkeiten zur Verfügung. Abb. 1 zeigt eine Übersicht etablierter Rechenarchitekturen nach Flexibilität, Performanz (Rechenleistung) und Leistungsaufnahme.

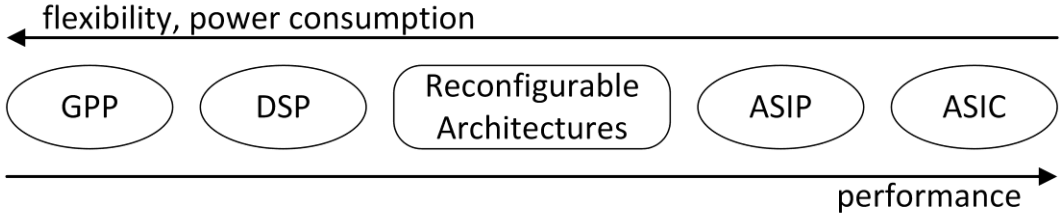

Abb. 1: Hardware-Software-Design-Space

„*General Purpose Processor*“en (GPP) bieten die flexibelste Umsetzung eines Algorithmus. Die Beschreibung der Aufgabe wird in Software in Form einer Programmiersprache ausgedrückt und in die Maschinensprache der GPP übersetzt. Moderne GPP enthalten verschiedene digitale Berechungseinheiten wie „*Floating*

Point Units“, Multipliziereinheiten, Sprungvorhersagen („*Branch Predigtion*“) oder Verarbeitungspipelines, um die Geschwindigkeit der Berechnung zu erhöhen. Superskalare Prozessoren mit „*Out-of-Order Execution“* lassen parallele Verarbeitung durch mehrere Verarbeitungseinheiten parallel zu. Doch aufgrund der erheblichen Komplexität ist eine Entwicklung von GPPs aufwendig und nur für hohe Stückzahlen rentabel. Man unterscheidet bei GPPs in RISC und CISC Architekturen. „*Complex Instruction Set Computer*“ (CISC) Architekturen besitzen einen umfangreichen Befehlssatz von bis zu mehreren 100 Befehlen. Die Länge der Befehle und der Aufwand ihrer Bearbeitung können sich unterscheiden. Die Anzahl und die Unterschiede an Befehlen führten zu großen Dekodiereinheiten, welche die Chips groß und damit teuer und verlustreicher machten. Wegen der langsameren Speicherzugriff konnten in der Vergangenheit mehrere Micro-Ops ausgeführt werden, doch durch die Fortschritte in der Anbindung von Speichermedien sowie die Einführung von Speicherhierarchien mit Level-Caches und Hauptspeicher verloren die CISC-Architekturen zusätzlich ihren Geschwindigkeitsvorteil gegenüber RISC Architekturen. Als Micro-Ops bezeichnet man den Befehlssatz eines CISC, in welchen die komplexeren Befehle jeweils zerlegt werden. Bei „*Reduced Instruction Set Computer*“ (RISC) Architekturen sprechen Befehle direkt Hardwareeinheiten des Prozessors an und werden nicht noch in Mirco-Ops des CISC Prozessors übersetzt. Das die Anzahl möglicher Befehle und vereinfacht die Dekodiereinheit. Aus Kompatibilitätsgründen werden CISC Befehle alter CISC Architekturen in RISC Befehle umgewandelt, um sie auf neuen Prozessorgenerationen ohne die alten CISC Befehle noch ausführen zu können. Daten werden vor der Verarbeitung zunächst in Verarbeitungsregister geladen, weshalb man häufig als Synonym von Load-Store-Architekturen spricht. Daher werden diese Architekturen mit einem vergrößerten Registersatz von Universalregistern ausgestattet, um Speicherzugriffe zu minimieren.

Digitale Signal Prozessoren (DSP) eignen sich besonders für datenflussorientierte Anwendungen, bei denen arithmetische Operationen gegenüber Datenflussanweisungen überwiegen. Vorwiegend wird als Prozessorarchitektur *Harvard* statt *von-Neumann* eingesetzt. Der Unterschied besteht hauptsächlich in der Adressierung und Anbindung des Instruktions- und Datenspeichers. Während in von-Neuman-Architekturen in einem gemeinsamen Speicherbereich sowohl Daten als auch Programmbefehle abgelegt werden, ist dies bei der Harvard-Architektur jeweils separiert. Damit verschwindet ein Nadelöhr in der Computerarchitektur, in der Daten und Programmbefehle sequenziell geladen werden müssen. Durch die getrennten Speicherbereiche in Harvard-Architekturen kann dies im Beispiel des DSPs parallel

geschehen und erzeugt damit einen Geschwindigkeitsvorteil. DPS enthalten spezielle Einheiten in Hardware, welche beispielsweise Multiplikationen für Fest- und Gleitkommazahlen, Adressierungen in Schleifen oder „*Multiply-And-Accumulate*“ (MAC)-Befehlssequenzen beschleunigen.

„*Application Specific Instruction Set Processors*“ (ASIP) sind Prozessoren, deren Architektur und Instruktionsbefehlssatz auf einen bestimmten Anwendungstyp optimiert sind. Dadurch werden Fläche und Verlustleistungen reduziert. Die Prozessoren sind im Vergleich zu „*Application Specific Integrated Circuits*“ (ASIC) immer noch programmierbar. Das Designen beider Architekturtypen ist jeweils teuer und nur für große Stückzahlen profitabel. Durch seine Programmierbarkeit ist der ASIP immer noch flexibler als ein ASIC, dennoch ist ein sinnvoller Einsatz auf bestimmte Applikationstypen beschränkt. An Performanz und Effizienz – im Sinne von Leistungsaufnahme zu Berechnungsgeschwindigkeit – für eben einen bestimmten Applikationstyp sind applikationsspezifische Architekturen unschlagbar. Dies wird erreicht durch Anpassungen bei der Speicheranbindung, angepasste Registergrößen, spezielle Funktionseinheiten in Hardware oder angepasste Verbindungsstrukturen innerhalb des Prozessors oder des integrierten Schaltkreises.

Zwischen diesen Extremen, den GPP Architekturen mit ihrer hohen Flexibilität für jeden Anwendungstyp und den ASIC als eine Sonderlösung für genau eine Anwendung befindet sich der Architekturtyp der programmierbaren Hardware, beispielsweise „*Field Programmable Gate Arrays*“ (FPGA), „*Field Programmable Crossbar Arrays*“ (FPCA) [5], „*Field Programmable Function Array*“ (FPFA) [6] oder „*Coarse Grained Reconfigurable Arrays*“ (CGRA). Im Gegensatz zu Prozessoren wie DSP oder GPP besitzen diese Chipdesigns keine festgelegte Architektur, sondern können, je nach Granularität (siehe Abschnitt 2.1.3), in ihrer Hardwarestruktur auf ein Problem angepasst werden. Damit wird ein Algorithmus also im Beispiel eines FPGAs nicht durch ein Softwareprogramm umgesetzt, sondern es wird eine Hardware beschrieben, die die vorhandenen Ressourcen eines FPGAs an die Applikation anpasst, sodass der Algorithmus direkt in Hardware umgesetzt werden kann. Der Vorteil: Die Umsetzung ist dabei nicht fest, sondern kann im Entwicklungsprozess immer wieder angepasst werden. Dies muss aber nicht durch einen teuren, aufwendigen Produktionsprozess passieren – beispielsweise wie bei einem ASIC –, sondern durch die Synthese eines so genannten Bitstromes, der den FPGA „programmiert“ oder besser konfiguriert. Die Anzahl möglicher Konfigurationen ist vom FPGA-Typ abhängig. Es haben sich heute aber verbreitet RAM-basierte FPGAs durchgesetzt, welche nahezu unbegrenzt beschreibbar sind. Sie werden mit einem Bootloader aus einem nichtflüchtigen

Speicher heraus bei einem Power-On konfiguriert, bevor sie mit der eigentlichen Verarbeitung beginnen. Im Vergleich zur Programmierung von Software ist die Beschreibung von Hardware mittels Hardwarebeschreibungssprachen wie VHDL oder Verilog komplex und die Synthese im Vergleich zu einem Kompiliervorgang sehr zeitaufwendig. Große Designs in VHDL oder Verilog können mehrere Stunden oder gar Tage benötigen, bis der entsprechende Bitstrom generiert werden konnte. Aus diesem Grund haben sich programmierbare Hardwarearchitekturen trotz gewisser Vorteile gegenüber Prozessoren noch nicht durchsetzen können. Das Prinzip des „*Reconfigurable Computing*“ ([7], [8], [9]), hat durch die komplexe Entwicklung vorwiegend im akademischen Bereich und in der Hochgeschwindigkeitsberechnung Aufmerksamkeit erfahren. Durch den steigenden Bedarf an Rechenleistung, sowie dem gleichzeitigen Interesse, die Leistungsaufnahme zu reduzieren, wird dieses Prinzip aber mehr und mehr an Interesse zurückgewinnen. Im Jahre 2015 beispielsweise fusionierte der größte Hersteller für General Purpose Prozessoren Intel, mit Altera, einen FPGA Hersteller. Im August 2019 stellte Intel beispielsweise den Intel Agilex vor [10]. Einen auf 10nm basierenden FPGA für *High Performance Computing* Anwendungen. Der Agilex ist Teil eines Frameworks, welches Softwareentwicklern die Beschleunigung von Algorithmen erleichtern soll [11]. Im Oktober 2020 zog AMD, der zweitgrößte Hersteller von GPP auf x86-Basis, nach und startete die Fusion mit Xilinx, ebenfalls ein FPGA Hersteller.

1.3 Zusammenfassung

Die steigende Komplexität von Software beispielsweise durch die Entwicklung künstlicher Intelligenz und die immer weiter fortschreitende digitale Revolution führen zu steigenden Anforderungen an die Effizienz der verarbeitenden Rechensysteme. Durch die steigenden Energiepreise und den Klimawandel werden Konzepte interessant, welche die Energieverbräuche unserer digitalen Rechnersysteme reduzieren, gleichzeitig aber die Verarbeitungsgeschwindigkeit der auszuführenden Algorithmen steigern. Der Parallelisierung sind nach Amdahl‘s Gesetz [12] Grenzen gesetzt und physikalische Einschränkungen in der Fertigung neuer Prozessorarchitekturen erfordern neue Konzepte, um durch die digitale Revolution unsere Welt (weiterhin) positiv zu entwickeln. Aus diesem Grund soll in dieser Arbeit ein weiterer heterogener Ansatz vorgestellt werden, der eine Applikation in Softwarekomponenten und Hardwarekomponenten, welche auf einem applikationsspezifischen Beschleuniger ausgeführt werden, aufteilt, um somit die Effizienz des Gesamtsystems zu steigern.

Doch die Integration von Software und die Hardwarebeschleunigung durch rekonfigurierbare Architekturen gestaltet sich immer noch als Herausforderung im Ingenieursalltag. Softwareentwickler verfolgen andere Denkmuster als Hardwareentwickler und umgekehrt. Diese Arbeit soll die Integration dieser beiden Welten vorantreiben durch eine datenflussorientierte Architektur, welche die Hardwareentwicklung für den Softwareentwickler abstrahiert, eine Hardwarebeschleunigung von Softwarekomponenten ermöglicht und dabei mittels schneller Rekonfiguration zur Laufzeit Hardwareressourcen schont.

Das vorgestellte Design der beschleunigenden Architektur soll dabei das Konzept der Laufzeitrekonfiguration möglichst ressourcenschonend implementieren. Dazu gehört eine schnelle Rekonfiguration sowie reduzierte Ressourcen für Berechnung und Speicherung entsprechender Hardwarekonfigurationen. Unter Laufzeitrekonfiguration versteht man, dass während der Ausführung eines Algorithmus die Hardwarekonfiguration ohne äußere Erkennbarkeit und ohne Datenverlust ausgetauscht werden kann.

Diese Arbeit untersucht also zunächst in Abschnitt 2.1.3 den Einfluss von Applikationen auf die Struktur (Bitbreite, Anzahl und Art der Logikeinheiten) von rekonfigurierbarer Hardware, um damit die Frage nach der optimalen Granularität eines Beschleunigerdesigns zu klären. Auf Grundlage der Resultate dieser Untersuchung wird ein rekonfigurierbares Design eines FPGA-*Overlays* das Virtual Coarse Grained Reconfigurable Array, kurz VCGRA, in Abschnitt 2.3 eingeführt. Es lässt sich in seiner Struktur an eine Applikation oder eine Applikationsklasse anpassen und ist rekonfigurierbar zur Laufzeit. Anhand der Evaluationsergebisse dieser ersten Variante des *Virtual Coarse Grained Reconfigurable Arrays* werden ab Abschnitt 3 zusätzliche Komponenten entwickelt, welche die Verarbeitung von Daten durch das VCGRA nochmals beschleunigen sollen. Da der Prozess der Entwicklung neuer Architekturkomponenten und die Evaluation deren Einfluss sehr zeitintensiv ist, wird parallel dazu eine Evaluationsplattform entwickelt, um den Einfluss neuer Systemkomponenten schneller zu evaluieren. Deshalb wird ebenfalls ab Abschnitt 4 die Entwicklung einer solchen Evaluationsplattform in SystemC vorangetrieben.

2 Konzept

2.1 Grundlegende Überlegungen

2.1.1 Der Algorithmus

Die Beschreibung des Algorithmenbegriffes ist überwiegend herausgearbeitet aus [13].

Im Allgemeinen beschreibt ein Algorithmus eine Methodik zur Lösung einer Aufgabe. Dabei wird die Lösung durch eine Sequenz von Schritten herbeigeführt. Die (korrekte) Abarbeitung dieser Schritte wird Prozess genannt, die verarbeitende Einheit dieser Schritte Prozessor. Um einen Prozess auszuführen, muss der Prozessor die Schritte des Algorithmus verstehen und verarbeiten können. Den Algorithmus so zu formulieren, dass der Prozessor die Anweisungen und Befehle versteht heißt Programmierung, das verwendete Werkzeug Programmiersprache. Mittels der Programmiersprache werden Algorithmenschritte in Anweisungen und Befehlen ausgedrückt. Bei der Formulierung von Algorithmen müssen drei wesentliche Merkmale beachtet werden:

- Berechenbarkeit: Die Berechenbarkeit beschäftigt sich mit dem Halteproblem eines Algorithmus, also ob für eine beliebige Eingabe von Daten der Algorithmus ein Ergebnis innerhalb einer endlichen Zeit finden kann.
- Komplexität: Bei der Komplexitätsanalyse sollen die notwendigen Ressourcen zum Lösen des Algorithmus ermittelt werden. Dabei handelt es sich beispielsweise um Zeitdauer, Speicher, Energiekosten oder Chipfläche.
- Korrektheit: Erzeugt der Algorithmus für (gültige) Eingaben auch (gültige) Ausgaben.

Die einfachste Form einer solchen Formulierung aus Sicht eines Prozessors wird als Maschinensprache bezeichnet. Dabei kann jede Anweisung des Algorithmus direkt als ein Verarbeitungsschritt des Prozessors abgearbeitet werden. Die Syntax und Semantik dieser Sprachen sind an die Fähigkeiten einer digitalen Maschine angelegt und somit oft nicht leicht lesbar bzw. interpretierbar durch einen Menschen. Auch sind die Verarbeitungsschritte einer Maschine auf kleine Teilaufgaben wie beispielsweise Multiplikationen oder Additionen und Vergleiche begrenzt, sodass die Komplexität der Programmierung auch für kleine Algorithmen schnell stark ansteigt. Aus diesem Grund haben sich verschiedenste höhere Programmiersprachen entwickelt, um die Lesbarkeit und Komplexität bei der Implementierung auch größerer Algorithmen zu erleichtern. Für die Ausführung auf einer

Verarbeitungseinheit muss der Code, also die Beschreibung des Algorithmus durch Anweisungen und Befehle, in die Maschinensprache übersetzt werden. Die Übersetzer werden beispielsweise als Compiler bzw. Assembler bezeichnet. Es haben sich in der Vergangenheit verschiedenste Programmiersprachen entwickelt und entwickeln sich heute noch. Dies liegt daran, dass sich neue Ideen und Technologien entwickeln, deren Nutzung ggf. mit neuen Programmiersprachen verbunden ist oder sein muss. Des Weiteren lassen sich Algorithmen unterschiedlich komplex in den unterschiedlichen Programmiersprachen beschreiben. Ein Algorithmus an sich ist aber unabhängig von der verwendeten Implementierungssprache. Er bildet die Grundlage für ein Programm, welches durch seine Ausführung auf einem Prozessor eine Lösung generiert. Somit bildet der Algorithmus die Grundlage für die Informationsverarbeitung.

2.1.2 Das Beschleunigerprinzip – Performanzsteigerung bei gleichzeitiger Reduktion der Leistungsaufnahme

Basierend auf den Vorüberlegungen zum Algorithmus lassen sich aus einem beliebigen Prozess in der Regel Codesequenzen ableiten, die folgenden Ansprüchen genügen:

I. Wiederkehrende Sequenzfolge, angewendet auf (unterschiedliche) Eingangsdaten
    a. Die Sequenzfolge enthält Einzelschritte möglichst gleichem oder ähnlichen Aufgabentyps (bspw. arithmetische Operationen oder Vergleichsoperationen)
II. Geringe Kopplung der zu errechnenden Ausgabedaten voneinander (Steigerung paralleler Verarbeitung)
III. Geringer Anteil an Kontrollflussanweisungen (bspw. Verzweigung, Schleifen, etc.)

Basierend auf diesen Eigenschaften lassen sich Prozessabschnitte als „*Kernel-Code*“ analysieren und auf eine zu beschleunigende Hardware, einem spezialisierten Prozessor, extrahieren. Diese Hardware passt sich dem Datenstrom der zu bearbeitenden Daten an, wodurch sich die Energieaufnahme reduzieren, der Datendurchsatz (Performanz) hingegen steigern lässt.

Grundlage für die im Folgenden beschriebene Architektur bilden datenstromorientierte Applikationen, welche von dieser Architektur selbstständig berechnet/ausgeführt werden können. Die Architektur wirkt dabei als Beschleuniger (engl. *Accelerator*) für den genannten Kernel-Code. Das Ziel des Beschleunigers ist es,

wiederkehrende Codesequenzen schneller und gleichzeitig energiesparender zu bearbeiten. Um die Flexibilität des Beschleunigers zu steigern, soll dieser während seiner Laufzeit konfigurierbar sein, um Ressourcen zu sparen. Die Rekonfiguration muss schnell und ohne Unterbrechnung der laufenden Berechnungen erfolgen. Der Grad der Flexibilität und der Aufwand für die Rekonfiguration während der Laufzeit wird beeinflusst durch die Granularität einer Architektur. Die Frage nach der optimalen Granularität soll daher 0im folgenden Abschnitt untersucht werden.

2.1.3 Die Frage nach der optimalen Granularität

Die Frage nach der Granularität einer programmierbaren Hardware beeinflusst die Komplexität für die Konfiguration der Architektur als auch die Ausführungsgeschwindigkeit einer Applikation auf eben dieser Hardware. Bezieht man die Möglichkeit einer Rekonfiguration der Hardware zur Laufzeit einer Applikation mit ein, beeinflusst die Granularität ebenfalls die benötigte Zeit und die benötigte Datenmenge an Rekonfigurationsdaten. Feingranulare Hardware-Architekturen wie FPGAs werden mit Bitströmen konfiguriert. Diese Bitströme entstehen durch die Hardwaresynthese der Beschreibungen von Architekturelementen in einer Hardwaredesignsprache wie VHDL oder Verilog. Ein Bit im Konfigurationsbitstrom beeinflusst dabei den Zustand eines Eintrages in einer „*Lookup-Table*“ (LUT) oder einer Verbindungsstelle (*Switch Block*) im Verbindungsnetzwerk. Im Gegensatz werden grobgranulare Architekturen (bspw. CGRA) aus Verarbeitungselementen aufgebaut, welche einen reduzierten Instruktionssatz besitzen und bereits auf Datenrepräsentationen größer einem Bit arbeiten. Durch die gröbere Struktur reduziert sich verglichen mit FPGAs die Länge des kritischen Pfades, was die Performanz eines CGRAs im Vergleich zum FPGA erhöht. Des Weiteren reduziert sich der Konfigurationsaufwand grobgranularer Architekturen, da ein Bit der Konfigurationsdaten nicht nur eine Verbindung oder einen Eintrag in einer Lookup-Table auf Bitlevel, sondern Verbindungen auf Bytelevel oder Funktionen einer Verarbeitungseinheit beeinflusst. Die Grenzen zwischen den Granularitätsstufen verlaufen dabei fließend, weshalb an dieser Stelle versucht wird mittels drei Definitionen den Granularitätsbegriff einzugrenzen.

2.1.3.1 Granularität gemessen am (Re-)Konfigurationsaufwand

Wie bereits im vorhergehenden Abschnitt angedeutet, beeinflusst die Granularität einer Hardware den Konfigurationsaufwand in Form von Energieverbrauch, Zeit und Datenmenge. Anhand des Einflusses eines Konfigurationsbits innerhalb eines so

genannten Konfigurationsbitstromes lassen sich feingranulare und grobgranulare Strukturen ableiten:

Beeinflusst ein Bit im Konfigurationsbitstrom die Zielarchitektur auf Bitebene, beispielweise durch den Eintrag in einer Logikzelle einer Lookup-Table, spricht man von einer feingranularen Architektur, wie beispielsweise einem FPGA. Eine Konfiguration auf Bitebene erbringt eine hohe Flexibilität, wird aber mit erhöhten Kosten in Form von Datenmenge und Zeitaufwand erkauft. FPGAs wurden daher in der Vergangenheit gerne als so genannte „*Glue Logic*“ verwendet, weil man binäre Signale mittels Konfiguration flexibel auf unterschiedliche Zielschnittstellen anpassen oder umleiten konnte.

Beeinflusst ein Konfigurationsbit das Verhalten einer Verarbeitungseinheit auf Datenebene, spricht man von grobgranularen Architekturen wie beispielsweise CGRAs. Bei der Datenebene handelt es sich im Allgemeinen um Datentypen, welche aus mehreren Bits bestehen wie einem Byte, einem Wort oder einem Doppelwort. Durch den reduzierten Instruktionssatz von grobgranularen Architekturen sinkt die Flexibilität im direkten Vergleich zu feingranularen Strukturen. Operationen auf Datentypen höherer Bitanzahl werden aber im Vergleich schneller ausgeführt als bei einem FPGA.

2.1.3.2 Granularität bestimmt durch die Bitbreite von Routing Ressourcen und Logikzellen

Diese Beschreibung von Granularität schließt sich an die Überlegungen aus Abschnitt 2.1.3.1 an und ist wie folgt beschrieben in [14]: „*The terminology of granularity describes the size of the bitwidth of the smallest addressable unit within the reconfigurable architecture.*“ Die Granularität einer Architektur wird also bestimmt durch die kleinste addressierbare Einheit der Architektur. Dabei wird sich vorwiegend auf die Eingangssignale von Logikzellen und die Vernetzungsressourcen bezogen. [14] beschreibt dabei folgende Einteilung nach Tab. 1.

Tab. 1: Granularität einer Architektur in Bezug auf die Bitbreite von Routing-Ressourcen und logischen Zellen

Nr.	Bitbreite	Klassifikation
1	1 … 3 Bit	feingranular (*fine-grained*)
2	4 … 8 Bit	mittelgranular(*middle-grained*)
3	9 … Bit	grobgranular (*coarse-grained*)

Im Unterschied zur ersten Definition wird die Bitbreite der Architektur zur Klassifikation betrachtet. Eine genauere Beschreibung der Unterschiede oder eine Abgrenzung zwischen den einzelnen Stufen wird in [14] nicht aufgeführt. Auch weist die Definition Schwächen auf, wenn die Architektur einen heterogenen Aufbau mit unterschiedlichen Datenbreiten aufweist.

2.1.3.3 Granularität bestimmt durch Eigenschaften eines FPGA

Diese Beschreibung einer Definition von Granularität wirkt auf den ersten Blick trivial, gilt der FPGA doch als feingranular. Für die ersten Modelle dieser Architektur trifft dies auch durchaus zu, doch hat sich der Aufbau moderner FPGAs zur Bearbeitung neuer Applikationen gewandelt. Moderne FPGAs besitzen neben den *Routing*-Ressourcen und Logikzellen Speicherzellen oder Blöcke zur digitalen Signalverarbeitung. Schematisch ist ein solcher Aufbau eines *„island-style“* FPGAs in Abb. 2 aufgezeigt.

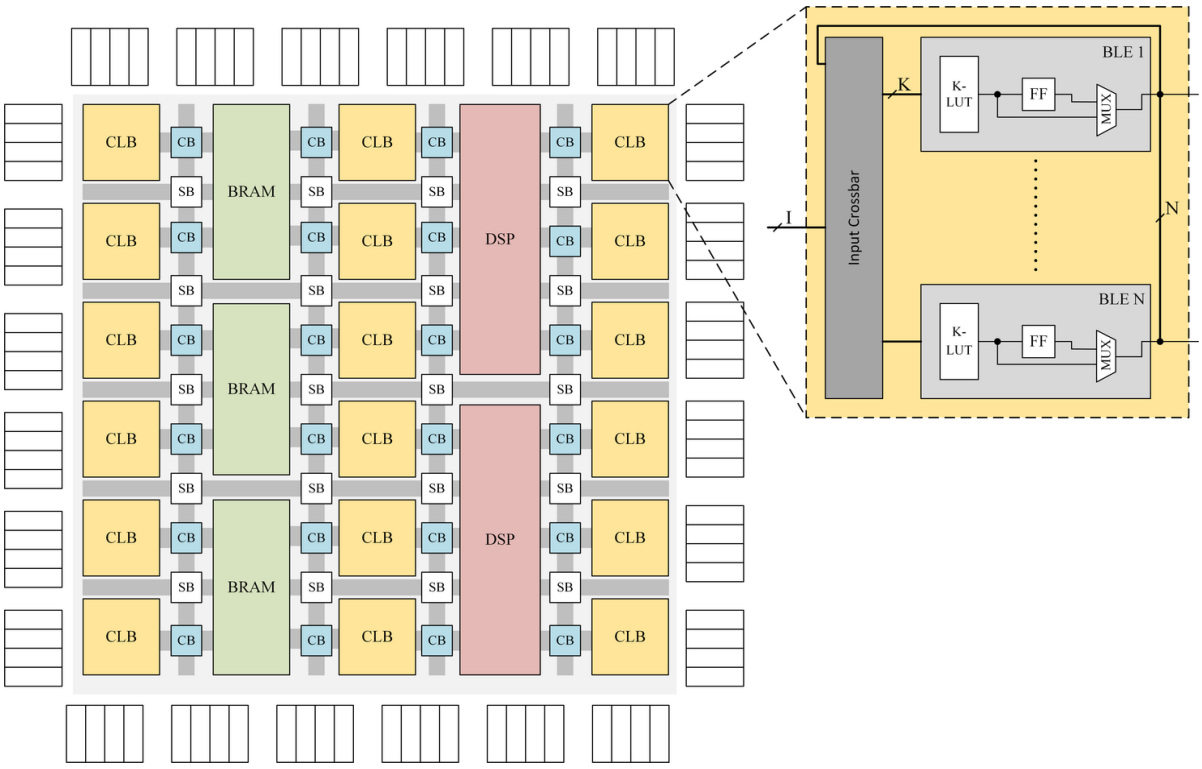

Abb. 2: Schematischer Aufbau eines modernen Island-Style FPGAs

So genannte Inseln von „*Configurable-Logic-Blocks*“ (CLB) oder „*Hardblocks*“ wie „*Blocked-RAM*“ (BRAM) oder digitale Signalverarbeitungsblöcke (DSP) sind von *Routing*-Ressourcen umgeben. Die *Routing* Ressourcen werden an Zweigstellen über „*Switch Blocks*“ (SB) verbunden. Die Anbindung der Logik-Inseln erfolgt über "*Connection Blocks*“ (CB). Eine CLB wiederum besteht aus mehreren „*Basic Logic Elements*“ (BLE), welche über einen Eingangsmultiplexer mit Eingangssignalen

verbunden werden. Eine BLE, in diesem Beispiel bestehend aus einer Lookup-Table, einem Ergebnis-Multiplexer und einem FlipFlop, bildet das kleinste Element der Architektur. Über den *2-1*-Multiplexer kann das aktuelle oder das um einen Takt verzögerte Ergebnis für die Weiterverarbeitung gewählt werden.

Durch die Struktur des FPGAs ergeben sich folgende Freiheitsgrade, welche die Granularität beeinflussen:

- Anzahl (Größe) der Lookup-Table einer BLE
- Anzahl der BLEs innerhalb einer CLB
- Art und Anzahl von Hardblocks (BRAM, DSP, etc.)
- Eigenschaften der Routing-Ressourcen:
    - Anzahl und Art der CB für den Anschluss einer CLB
    - Segment-Länge (Abstand zwischen zwei SB oder CB)
    - Verbindungstyp zwischen sich kreuzenden *Routing*-Ressourcen (*universal*, *subset*, *wilton*)
- Arrangieren der Blocktypen im FPGA-Aufbau (Wie sind die Spalten aufgebaut?).

Die CLBs und die *Routing*-Ressourcen bilden den feingranularen Teil des FPGAs, während die Hardblocks als grobgranular definiert werden können. In einer umfangreichen Untersuchung wurden die genannten Parameter verändert und die Auswirkung der Anpassung auf verschiedene Benchmarks untersucht. Ziel war es, eine Aussage über die Granularität einer Architektur zu gewinnen und wie sich Architektureigenschaften auf bestimmte Klassen von Applikationen auswirken. Die Untersuchung und ihre Ergebnisse sind im folgenden Abschnitt 2.1.3.4 genauer beschrieben.

2.1.3.4 Granularitätsuntersuchung einer FPGA-Architektur mit Verilog-To-Routing

2.1.3.4.1 Beschreibung der Toolchain

Die Untersuchung der Granularität einer FPGA-Architektur erfolgte mit „*Verilog-To-Routing*“ (VTR) [15], dem gebräuchlichsten quelloffenen Werkzeug für CAD („*Computer-Aided-Design*“) und FPGA Entwicklung. Mittels VTR lassen sich virtuelle FPGA-Architekturen erstellen und testen. Es besteht aus drei verschiedenen Werkzeugen:

- ODIN II: Elaboration und Synthese
- ABC: Logiksynthese und Technologiemapping auf Standardzellen

- VPR: „*Packing*“, „*Placement*“ und „*Routing*“ auf virtuelle FPGA-Architekturen mit Abschätzung von Timings, Leistungsaufnahme und Platzverbrauch.

Eine Übersicht zu den Abhängigkeiten der aufgeführten Werkzeuge ist in Abb. 3 dargestellt.

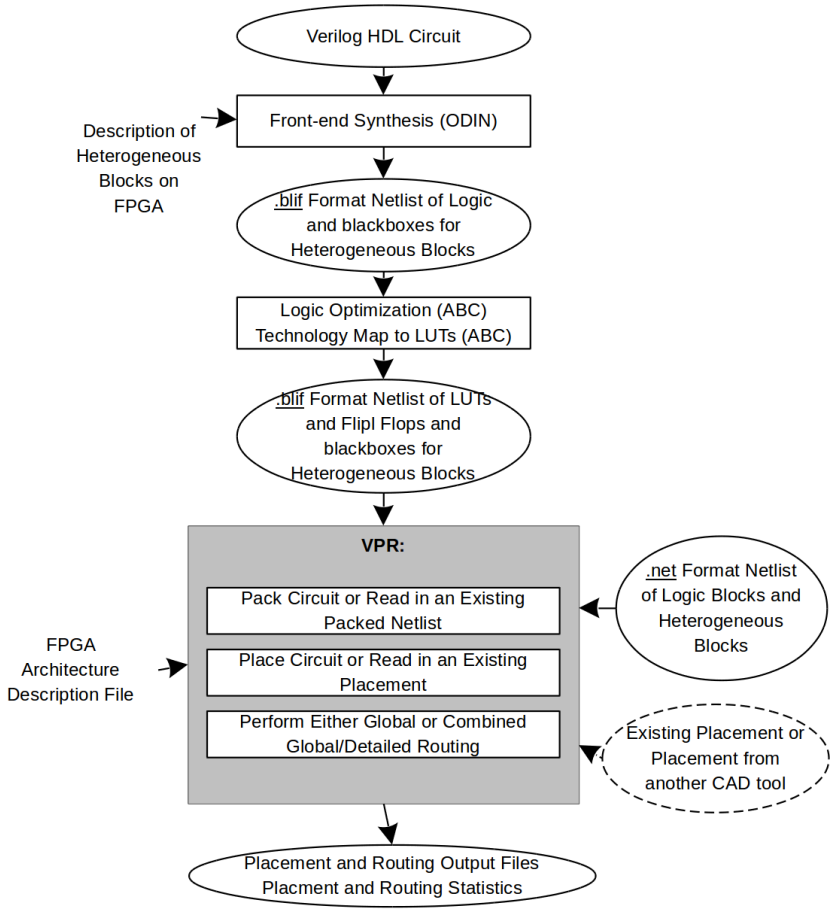

Abb. 3: Übersicht der VTR7.0 [15] Toolchain

Für die Konfiguration der Architektur dient eine Beschreibung in *XML*-Format, die Beschreibung der Applikation erfolgt durch Verilog. Mittels ODIN II als erste Station wird aus Verilog eine Netzliste erstellt, welche aus Standardzellen und anderen Hardblocks besteht. Diese Hardblocks sind modelliert als Blackboxes. Die Identifikation solcher Blackboxes wird erreicht mittels der Hardwarebeschreibung im *XML*-Format. Nativ werden Addierer, Multiplizierer und Speicherblöcke unterstützt. Die Netzliste im *blif*-Format (Berkeley Logic Interchange Format) wird an ABC übergeben. Dieses ignoriert die Hardblocks, führt aber eine Synthese und Optimierung der Standardzellen durch und erstellt eine aktualisierte Netzliste im *blif*-Format. Bei der Synthese werden die Standardzellen auf LUTs und FlipFlops übertragen. Dies geschieht aber ohne Einbeziehung der Eigenschaften der virtuellen FPGA-Architektur. Im letzten Schritt, dem VPR-Werkzeug, werden diese

Eigenschaften dann beachtet und das optimierte *blif*-File auf die vorgegebene Architektur platziert. Nach einer initialen Platzierung wird das Design mittels „*Simulated Annealing*[2]“ optimiert bevor das *Routing* die Blöcke des Designs verbindet. Am Ende werden Informationen zum Platzbedarf, Performanz und *Routing* sowie die Ausführungszeiten für das CAD dem Nutzer zur Verfügung gestellt.

Für die Untersuchung von FGPA-Eigenschaften auf Applikationen durch die Ausführung von mehr als 650 Benchmarks wurde ein Toolflow in Python entwickelt. Die Programmiersprache Python hat eine große Bandbreite an Bibliotheksfunktionen für den Zugriff auf Betriebssystemfunktionen, File-Parsing und zur Erstellung von Diagrammen. Die Struktur des Toolflows ist dargestellt in Abb. 4.

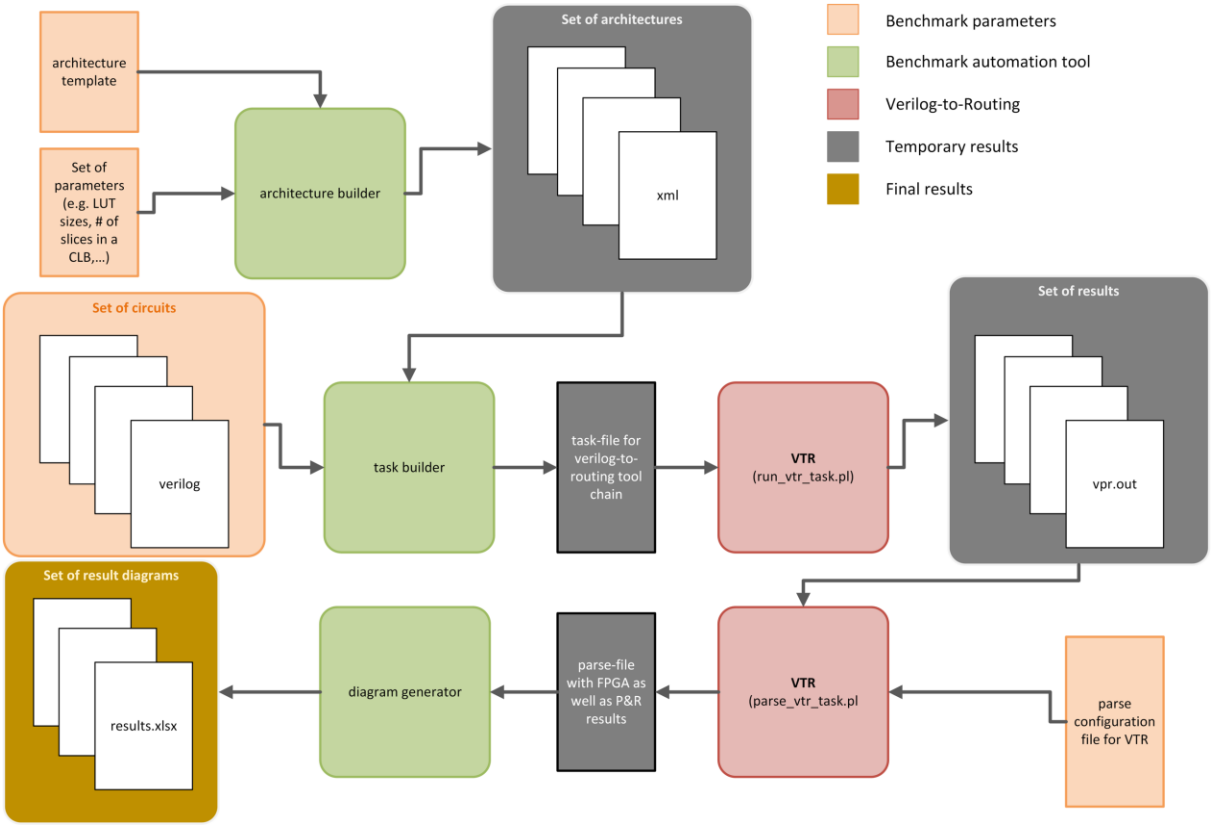

Abb. 4: Toolflow für das Benchmarking von FPGA-Architekturen

Die grünen Elemente des Diagramms zeigen Blöcke des auf Python-basierenden Toolflows. Im ersten Schritt, dem „*architecture builder*“, werden auf Basis eines Architektur*Templates* und einer Auswahl an Kommandozeilenoptionen verschiedene

[2] Heuristisches Approximationsverfahren zur Lösung von Optimierungsproblemen [65]

Architekturen generiert. Als Einstellungsmöglichkeiten werden folgende FPGA-Parameter zur Verfügung gestellt:

- LUT-Größe
- Anzahl der BLEs innerhalb einer CLB
- Art der Switch Blocks (*wilton*, *subset*)
- Verfügbarkeit von Hardblocks (Multiplizierer, Addierer, BRAM)

Die Art der „*Switching*“ Matrix beeinflusst die Flexibilität beim *Routing*, aber auch den notwendigen Platzbedarf und die Performanz. Je mehr Verbindungen innerhalb eines Switch Blocks vorhanden sind, desto größer seine Flexibilität und sein Platzbedarf.

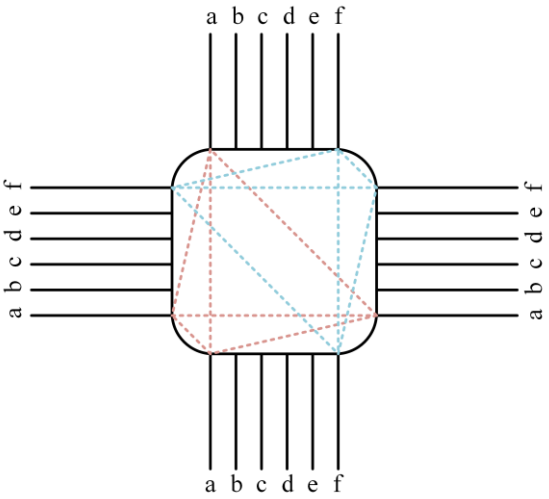

Abb. 5: SB-Typ „subset“

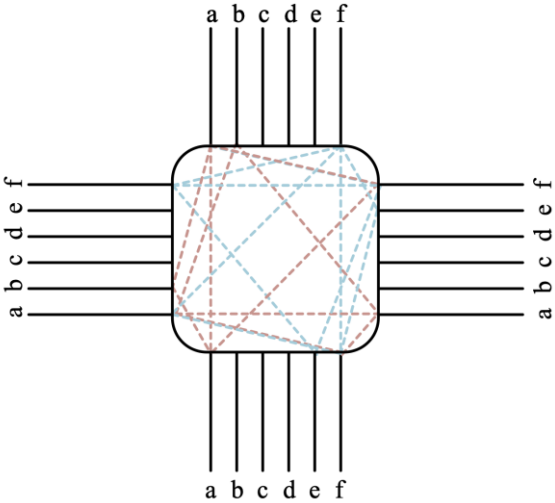

Abb. 6: SB-Typ „wilton“

VTR bietet noch einen weiteren Typ names „*universal*“ an. Dieser verbraucht aber zu viele Ressourcen und wird daher vernachlässigt. So berechnet sich die Anzahl notwendiger Transistoren mit jeweils 4 Eingangssignalen pro Seite überschlagsweise für „*universal*“ auf:

4 Eingänge pro Seite

x (4 Transistoren pro SRAM-Zelle + jeweils 1 Transistor pro Schalter)

x (12 + 8 + 4) Verbindungen (1)

= 480 Transistoren pro SB

Im Vergleich mit dem SB-Typ „*subset*“ verbraucht dieser unter gleichen Randbedingungen nur ca. 120 Transistoren. Nebenbei entstehen durch größere

Switch Blocks größere Latenzen für die Signallaufzeiten, was die Performanz der Architektur gesamtheitlich reduziert.

Die betrachteten SB-Typen sind in Abb. 5 und Abb. 6 vergleichend gegenübergestellt. Um die Verbindungsmöglichkeiten hervorzuheben, sind jeweils beispielhaft zwei Eingangssignale „*a*“ und „*f*“ hervorgehoben, „*a*“ in rot und „*f*“ in blau. In „*subset*“ werden immer Eingänge mit der gleichen Bezeichnung verbunden, während in „*wilton*“ die Verbindungen zu den benachbarten Seiten rotieren. In adjazenten (benachbarten) Kanälen gehören bei „*subset*“ alle Leitungen mit dem gleichen Label dem gleichen Signal an. Rotiert man die diagonalen Kanäle jedoch um einen Kanal wie beim SB-Typ „*wilton*“, steigt die Anzahl der Verbindungsmöglichkeiten. Wird beispielsweise das Signal auf dem vertikalen Kanal „*a*“ auf den horizontalen Kanal „*b*“ verbunden, ist die horizontale Verbindung gegenüber von „a“ nun frei und kann für eine weitere Verbindung genutzt werden. Somit können unter Umständen SB-Blöcke eingespart werden, was wiederum Signallaufzeiten und Platzbedarf reduziert und damit Performanz steigert bzw. Leistungsaufnahme reduziert.

Ebenfalls mittels Kommandozeilenparametern einstellbar sind die Benchmarks, welche für die Evaluation auf den verschiedenen Architekturen ausgeführt werden sollen. Bei diesen Applikationen handelt es sich um *Verilog*-Dateien. Für die Durchführung der Benchmarks wurden Beispielapplikationen von VTR genutzt:

- **bgm.v:** Monte Carlo Simulation für Preisentwicklung privater Derivate (Finanzmathematik)
- **LU8PEEng.v:** „*LU Decomposition*“ zur Lösung linearer Gleichungssysteme
- **stereovision2.v:** Akademischer Schaltplan für computerbasierte Bildverarbeitung

Im zweiten Schritt, dem „*task builder*“, wird ein Arbeitsplan für VTR erstellt. Die Anzahl der Aufgaben – der auszuführenden Simulationen – ergibt sich aus der Anzahl der Benchmarks und der Anzahl der vorher erstellten Hardware-Konfigurationen. Zusätzlich wird eine Konfiguration für das Auslesen der VTR-Ergebnisse mit angegeben („*parse configuration file*“). Diese enthält Angaben über die Formatierung und die Art der zu untersuchenden Ergebnisse. VTR unterstützt die parallele Simulation mehrerer Aufgaben, die zunächst vollständig bearbeitet werden. Das Aufbereiten der Ergebnisse findet nach dem Ablauf der Simulationen statt. Die so erhaltenen Simulations- und Synthesedaten werden weiterverarbeitet und in ein Format gebracht, welches sich erleichtert in Excel auswerten lässt. Dies geschieht im dritten Schritt, dem „*diagram generator*“.

Um den Entwicklungsraum einzuschränken, wurden die Parameter für das Benchmarking neben den drei genannten Applikationen und den zwei SB-Typen auf folgende Architekturparameter begrenzt:

- LUT-Größe zwischen 3 und 9: Die LUT-Größe basiert auf [16] und umfasst typische LUT-Größen kommerzieller FPGAs
- Anzahl der BLEs pro CB: Die BLE-Größe wird jeweils verdoppelt und startet bei 5
- Hardblocks: Die Hardblocks orientieren sich an den Speichergrößen und dem Vorhandensein von DSP-Blocks aktueller FPGA-Architekturen
    - 32KB „fracturable“ BRAM
    - 36Bit „fracturable“ Multiplizierer

Während initialer Versuche für die Granularitätsuntersuchung wurde festgestellt, dass VTR nicht in der Lage war CLBs mit mehr als 80 BLEs zu verarbeiten. Daher beschränkt sich die maximale Anzahl der BLEs auf 40. Für das Benchmarking wurden Beschreibungen der Hardblocks aus dem VTR-Repository genutzt. Multiplizierer und BRAM-Block lassen sich dabei in schmalere Blöcke zerlegen. Diese Eigenschaft wird in VTR als „*fracturable*“ bezeichnet.

2.1.3.4.2 Ergebnisse des Benchmarkings

Die folgenden Diagramme zeigen die Ergebnisse der Benchmarks. Zunächst wurden die Ergebnisse für jede Applikation jeweils auf ihren Maximalwert normalisiert:

$$val_{REF_j} = \frac{val_i}{\max\{val_0, \dots, val_i, \dots val_n\}} \quad i \in [0, n], j \in \mathbb{N} \tag{2}$$

Des Weiteren wurde für jeden Ergebniswert über alle Applikationen der Durchschnitt errechnet:

$$\overline{val_j} = \frac{1}{3} \sum_{k=1}^{3} val_{REF_j k} \quad j \in \mathbb{N} \tag{3}$$

Das bedeutet, es ergibt sich für jeden Architekturtyp ein Sammelergebnis zu einem Untersuchungsgegenstand, bestehend aus den arithmetischen Mittelwerten der normalisierten Ergebnisse je Applikation und Hardwarekonfiguration. Für die Übersichtlichkeit wurden die Ergebnisse für die LUT-Größen auf drei, sieben und neun reduziert, um den kleinsten Wert, den größten Wert und einen Wert in der Mitte gegenüberzustellen. Auch sind für die gewählten Eigenschaften alle Simulationen ohne Fehlerabbruch verwendet worden, sodass vollständige Ergebnisse vorliegen.

Überstieg die Anzahl der „*Routing-Tracks*“ in VPR tausend, war das Programm nicht in der Lage, das *Routing* abzuschließen und beendete das Programm mit einer Fehlermeldung. Aus diesem Grund konnten nicht alle Benchmarks abgeschlossen werden. Die Ergebnisse der beteiligten Architekturkonfigurationen dieser Benchmarks wurden von der Auswertung ausgeschlossen.

Als erstes sollen die Ergebnisse für Platzbedarf (engl. *Area*) und maximale Taktfrequenz (engl. *Frequency*) beleuchtet werden. Der notwendige Platzbedarf setzt sich zusammen aus dem Platz für CLBs, Hardblocks, falls diese verfügbar sind, und dem Platzbedarf für die *Routing*-Ressourcen:

$$area_{FPGA} = area_{CLB} + area_{Hardblocks} + area_{Routing} \quad (4)$$

Als Einheit für die Beschreibung von Area wird die MWTA (*Minimum Width Transistor Area*) eingesetzt. Sie beschreibt technologieunabhängig den kleinsten Platzbedarf eines Transistors inklusive des minimalen fertigungsbedingten Abstands um einen Transistor herum [16]. Der Platzbedarf eines CLBs wird bestimmt durch die Anzahl seiner BLEs und deren Platzbedarf sowie den *Routing*-Ressourcen innerhalb der CLB, um die BLEs bzw. den CLB mit den *Routing*-Ressourcen des FPGAs zu verbinden. Die Fläche einer BLE entsteht hauptsächlich durch die Größe seiner LUT, sowie durch die Größe des FlipFlops und des 2-1-Multiplexers (siehe Abb. 2). Die Größe eines Hardblocks ist in den VTR-Bibliothekselementen beschrieben, sodass für den FPGA die Anzahl der Hardblockelemente gezählt und mit den gegebenen Größen multipliziert wird. Für den Flächenbedarf der CLBs wird der Platzbedarf einer CLB ebenfalls mit der Anzahl aller CLBs auf dem Chip multipliziert. Für die Abschätzung des Flächenbedarfes einer CLB wurde eine Formel entworfen, da VTR die Fläche einer CLB nicht auflöst, womit Eigenschaften wie Anzahl der BLEs und die LUT-Größe die Ergebnisse nicht ausreichend beeinflussen. Es ergab sich ein rein linearer Zusammenhang zwischen der Anzahl der BLEs und der LUT-Größen zur Gesamtfläche der CLBs.

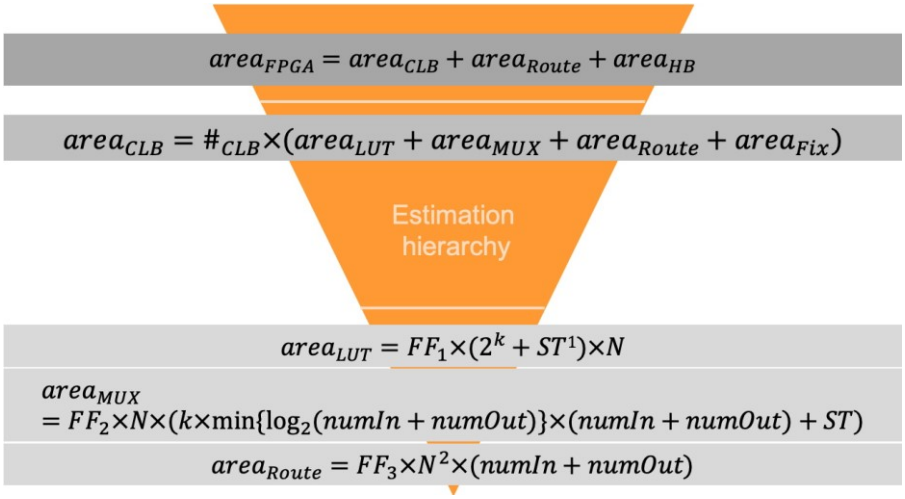

Abb. 7: Abschätzung zur Fläche eines FPGA

Abb. 7 verdeutlicht den verwendeten hierarchischen Ansatz für einen auf SRAM-basierenden FPGA. Das Kopfende bildet die bereits angegebene Summe aus Fläche für *Complex Logic Block*, Fläche für Hardblocks und Fläche für das externe *Routing*. Die Fläche für *Routing* und die Hardblocks können wie angesprochen direkt aus VPR entnommen werden. Für die *Routing*-Ressourcen wird die Anzahl der Transistoren abgeschätzt, die in den SBs und CBs verwendet werden, wie auch Transistoren für Track-Puffer[3]. Die Fläche einer CLB ergibt sich aus dem Platzbedarf für die LUTs, die internen *Routing*-Ressourcen, den Multiplexer für das interne *Routing* und einen Fixwert (ST), der beispielsweise das FlipFlop einer BLE enthält.

$$area_{CLB} = FF_1 \times area_{LUT} + FF_2 \times area_{Mux} + FF_3 \times area_{Route} + area_{Fix} \quad (5)$$

$$area_{LUT} = 2^K \times \#Transistors \times \#BLEs \quad (6)$$

$$area_{Mux} = K \times BLEs \times (\min\{log_2(\#Inputs + \#Outputs)\} \times (\#Inputs + \#Outputs) \times \#Transistor) \quad (7)$$

$$area_{Route} = (\#BLEs)^2 \times (\#Inputs + \#Outputs) \quad (8)$$

Die Fläche einer LUT wird abgeschätzt durch die Anzahl der Zellen zum Speichern der Informationen multipliziert mit der Anzahl der Transistoren ($\#Transistors$), welche eine SRAM-Zelle bilden. Die Gesamtfläche für alle LUTs ergibt sich dann durch die Multiplikation mit der Anzahl der BLEs ($\#BLEs$) innerhalb der CLB. Jedes BLE besitzt einen eigenen Multiplexer mit M Ausgängen, welche jeweils G-Addressleitungen

[3] FlipFlops innerhalb des Routings, um die Signallaufzeiten zu reduzieren und die Taktfrequenz des Designs zu erhöhen.

benötigen, wobei sich die Anzahl der Adressleitungen aus $G = \min\{log_2(\#Inputs + \#Outputs)\}$ ergibt. Die Anzahl der Eingänge eines Multiplexers ergibt sich aus der Anzahl der Eingänge einer CLB plus der Anzahl der BLEs innerhalb einer CLB, da jede BLE noch als Eingang rückgekoppelt werden kann. Die als „*full crossbar*“ bezeichnete Variante eines Eingangsmultiplexers kann als *Worst Case Scenario* angesehen werden, da er auf Grund der Flexibilität den größten Platzaufwand benötigt. Für das *Routing* wird als *Worst Case Scenario* von exklusiven Signalleitungen ausgegangen. Die Anzahl der Eingänge und Ausgänge sowie die Anzahl der BLEs beeinflussen die *Routing* Ressourcen linear bzw. quadratisch. Zur Steuerung der Approximation werden noch Faktoren (FF_1, FF_2, FF_3) eingeführt, welche jeweils die Anteile gleichverteilt beeinflussen. Zur Approximation wurden die Linearfaktoren und die Anzahl der Transistoren zur Bereitstellung einer SRAM-Zelle auf Eins gesetzt.

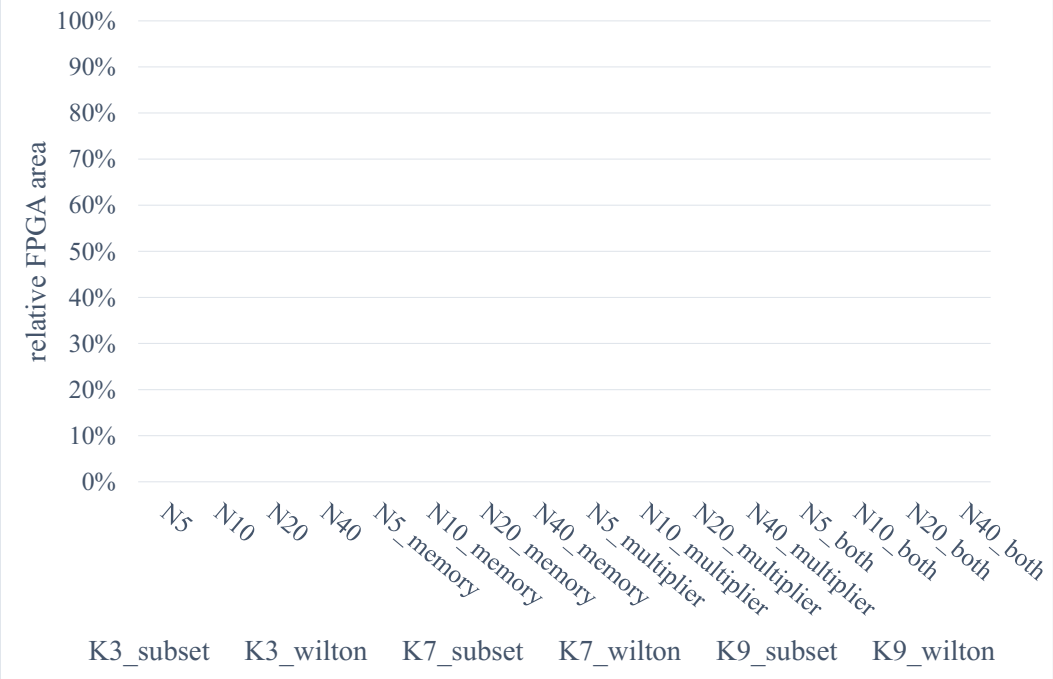

Abb. 8: Zusammenfassung der Ergebnisse für den Flächenbedarf

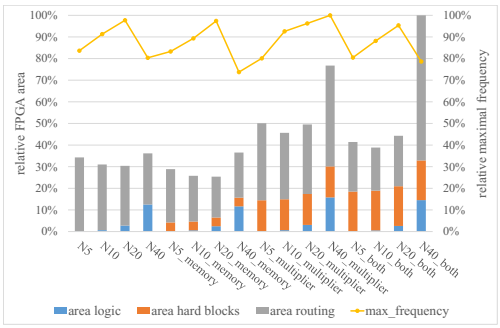

Abb. 9: Area und max. Freq.
K: 3, Switch Block Typ: subset

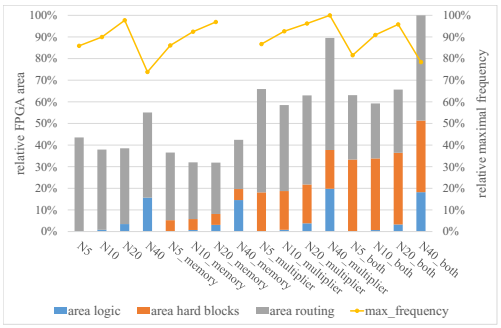

Abb. 10: Area und max. Freq.
K: 3, Switch Block Typ: wilton

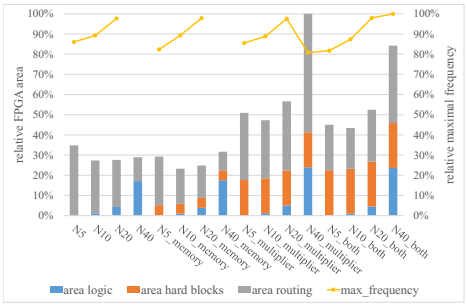

Abb. 11: Area und max. Freq.
K: 7, Switch Block Typ: subset

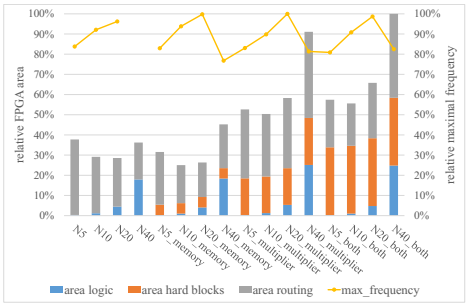

Abb. 12: Area und max. Freq.
K: 7, Switch Block Typ: wilton

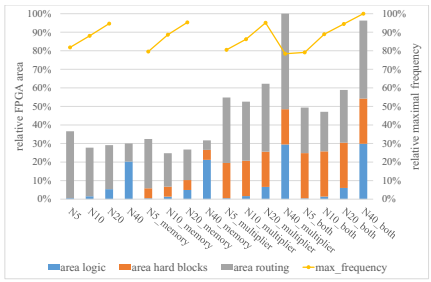

Abb. 13: Area und max. Freq.
K: 9, Switch Block Typ: subset

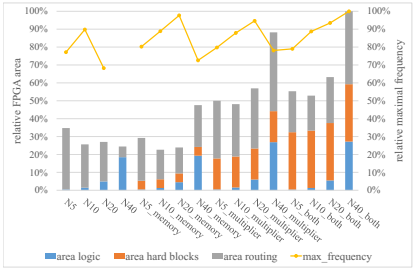

Abb. 14: Area und max. Freq.
K: 9, Switch Block Typ: wilton

In Abb. 8 sind zunächst die Gesamtflächen normiert auf den größten Wert aus allen Versuchsreihen gegenübergestellt. Es wurden für die LUT Größe K der kleinste Wert der Versuchsreihe, der größte Wert und ein Wert aus der Mitte gewählt. Zusätzlich wurden Hardblocks für Memory, Multiplizierer oder beides in das Design miteingefügt. Die Nummer „N“ steht jeweils für die Anzahl der BLEs innerhalb der

CLBs. *Wilton* und *Subset* beschreiben die Routing Möglichkeiten wie unter Abschnitt 2.1.3.4.1 eingeführt. Die Abbildungen Abb. 9 bis Abb. 14 lösen die Gesamtfläche noch einmal jeweils in seine einzelbestandteile auf und bilden darüber hinaus die Entwicklung der max. Frequenz ab. Alle Ergebnisse sind normiert auf das Maximum aller verfügbaren Messergebnisse und zeigen daher relative Zusammenhänge.

Für die untersuchten Applikationen benötigt nach Abb. 8 *K3_subset* mit beiden Hardblocks den meisten Platz. Schaut man sich dazu passend auch das Diagramm nach Abb. 9 an, sieht man, dass der notwendige Platzbedarf vom *Routing* Anteil dominiert wird. Durch die geringe LUT-Größe und die vielen BLEs werden anscheinend gleichzeitig viele CLBs benötigt um die Applikationen zu mappen und diese müssen durch viele *Routing*-Ressourcen verbunden werden. Als Folge wird wenig internes *Routing* in den CLBs verwendet.

Für die verwendeten Applikationen spielt nach den Ergebnissen Speicher eine entscheidende Rolle. Er reduziert den notwendigen Platzbedarf auf die minimalen Werte nach Abb. 8. Die Verwendung einer mittleren LUT-Größe zeigt hier die besten Ergebnisse wie auch in [16] beschrieben. Für eine große Anzahl an BLEs innerhalb einer CLB werden die Ergebnisse generell schlechter. Es ist aber nicht ganz klar, ob dies an den verwendeten Benchmark-Applikationen selbst oder an den Algorithmen für das PP&R liegt, dass bei so großen CLBs ggf. nicht so gut optimieren kann, wie bei kleinen und das somit häufig neue CLB-Blöcke einfügt. Das solche großen CLBs dem Programm Schwierigkeiten bereiten ist auch daraus erkennbar, dass die Designs mit „N“ = 40 häufig nicht vollständig geroutet werden konnten und Ergebnisse zur maximalen Ausführungsgeschwindigkeit fehlen.

Wie auch in [16] beschrieben dominiert den Flächenbedarf vorwiegend das *Routing*. Dies ist meist unabhängig von der Anzahl der BLEs und der LUT-Größe „K“. Die Verwendung von bis zu 20 BLEs pro CLB und die Verwendung von Hardblocks hat einen positiven Einfluss auf die Ausführungsgeschwindigkeit. So befinden sich die Maxima für mittlere und große LUT-Größen immer in FPGAs mit Hardwareblöcken beider Arten und einer Mindestzahl von 20 BLEs pro CLB. Nur für kleine LUT-Größen zeigen die Ergebnisse abweichende Maxima. Hier spielt nur der Multiplizierer eine Rolle. Gegebenenfalls sind hier durch die kleinen LUTs mehr Logikeinheiten durch kürzere Pfade verbunden, was den kritischen Pfad verkürzt und damit die Ausführungsgeschwindigkeit erhöhen kann.

Als nächstes werden die Ergebnisse für die Kanalbreite (engl. *channel width*) des FPGA ausgewertet. Sie wird bestimmt durch die Anzahl von Verbindungen (engl. *Tracks*),

welche parallel verlaufend zwischen den Blöcken eines FPGAs liegen und diese Blöcke miteinander verbinden. Je mehr Verbindungen parallel liegen, desto mehr Signale können parallel geleitet werden. Dies erhöht aber den notwendigen Flächebedarf.

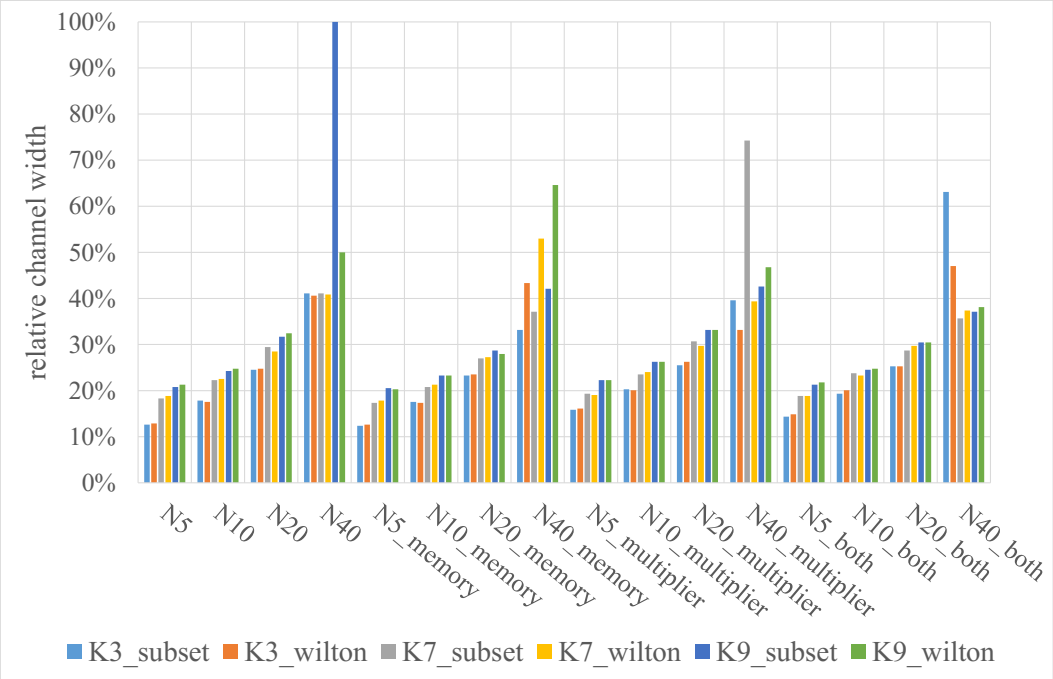

Abb. 15: Relative Kanalbreite in Abhängigkeit von den Eigenschaften der Architektur

Abb. 15 fasst die Ergebnisse der Versuchsreihe für den Einfluss von FPGA-Eigenschaften auf die Kanalbreite zusammen. Die Bezeichnung der Datenreihen laut Legende orientiert sich an der Benennung in den Ergebnissen zum Flächenbedarf. Die Ergebnisse sind wiederum auf das Maximum der Versuchsreihe normiert. Daher ist bei einer LUT-Größe von neun mit 40 BLEs innerhalb einer CLB die Kanalbreite am größten. Die Kanalbreite steigt mit der Anzahl der BLEs innerhalb einer CLB als auch mit der Größe einer LUT innerhalb einer BLE. Dies wird verursacht durch die notwendige Anzahl von Tracks, um die BLEs bzw. die Eingänge der LUT einer BLE an das *Routing*-Netzwerk anzuschließen. Eine große LUT-Größe und eine große Anzahl an BLEs innerhalb einer CLB hat generell einen schlechten Einfluss auf die Kanalbreite. Es sind immer Ausreißer erkennbar, egal ob mit Hardblocks oder ohne. Das Hinzufügen von zusätzlichen Hardblocks hat in den meisten Fällen nur einen kleinen positiven Einfluss auf die Kanalbreite und ist von der Art des Hardblocks abhängig.

Tab. 2: Beispiel Auswirkungen von Hardblocks auf die Kanalbreite (K 3; SB-Typ wilton; N 5)

lfd. Nr.	Multiplizierer	BRAM	Relative Kanalbreite
1			12,62%
2		X	12,38%
3	X		15,84%
4	X	X	14,36%

Der Auszug in Tab. 2 zeigt ein Beispiel für eine Architektur mit einer LUT-Größe von drei mit fünf BLEs innerhalb eines *Complex Logic Block* und „*wilton*“ als Switch Block Typ. Die Verwendung von Multiplizierern führt eher zu einer Zunahme der Kanalbreite. Dies liegt an der Bitbreite der Multiplizierer, die an die vorhandenen *Routing*-Ressourcen angeschlossen werden müssen. Der Einsatz von Speicher verringert die Anzahl von LUTs, die verwendet werden müssen um Informationen abzulegen. Daher müssen weniger Tracks parallel geführt werden, um die gespeicherten Informationen an anderer Stelle zu verwenden. Der Anschluss der BRAM-Zellen ist effizienter. Die Verwendung von Hardblocks bei der Spitzenreiterkonfiguration (K 9, N 40, SB-Typ *subset*) eines FPGAs hatte in den Versuchsreihen einen positiven Einfluss. Durch die eingeschränkten *Routing*-Möglichkeiten beim SB-Typ „*subset*“ – siehe Abschnitt 2.1.3.4.1 – kommt es bei mindestens einer der gewählten Applikationen zu hohen parallelen Datenübertragungen. Für diese Datenübertragungen müssen zusätzlichen Tracks bereitgestellt werden. Durch die Verwendung effizienter Hardblocks können diese Einflüsse laut der Ergebnisse reduziert werden. Generell führt die Verwendung von BRAM bei Architekturen mit einer CLB bestehend aus maximal 20 BLEs zu einer Reduktion der Kanalbreite um durchschnittlich 4,5%. Die Ergebnisse sind dabei unabhängig vom Switch Block Typ oder der LUT-Größe. Dies wird deutlich durch die annähernd gleich hohen Balken für einen der untersuchten *Complex Logic Block* Typen (bspw. K3/N5, blauer und orangner Balken). Daraus folgt, dass die Verwendung von „*wilton*“ als SB-Typ keinen negativen Einfluss auf die Kanalbreite hat, die Optionen für das *Routing* aber wie oben bereits beschrieben verbessert. Für Architekturen mit mehr als 20 BLEs innerhalb einer CLB ist die Auswahl des Switch Block Typs bzw. die Verwendung von Hardblocks signifikant für die errechnete Kanalbreite. Es ist nicht ganz klar, ob dieser Einfluss aus den gewählten Benchmarkapplikationen hervorgeht,

denn durch die geringe Anzahl beeinflussen Ausreißer das Durchschnittsergebnis bereits stark. Ebenfalls möglich ist, dass die Effizienz der Algorithmen von VPR bei zu großen CLB-Dimensionen keine aussagekräftigen Ergebnisse mehr liefert. Eine erweiterte Untersuchung mit mehr Applikationen war leider im Zuge dieser Dissertation nicht mehr durchführbar. Daher können die genannten Annahmen leider nicht vollständig geklärt werden.

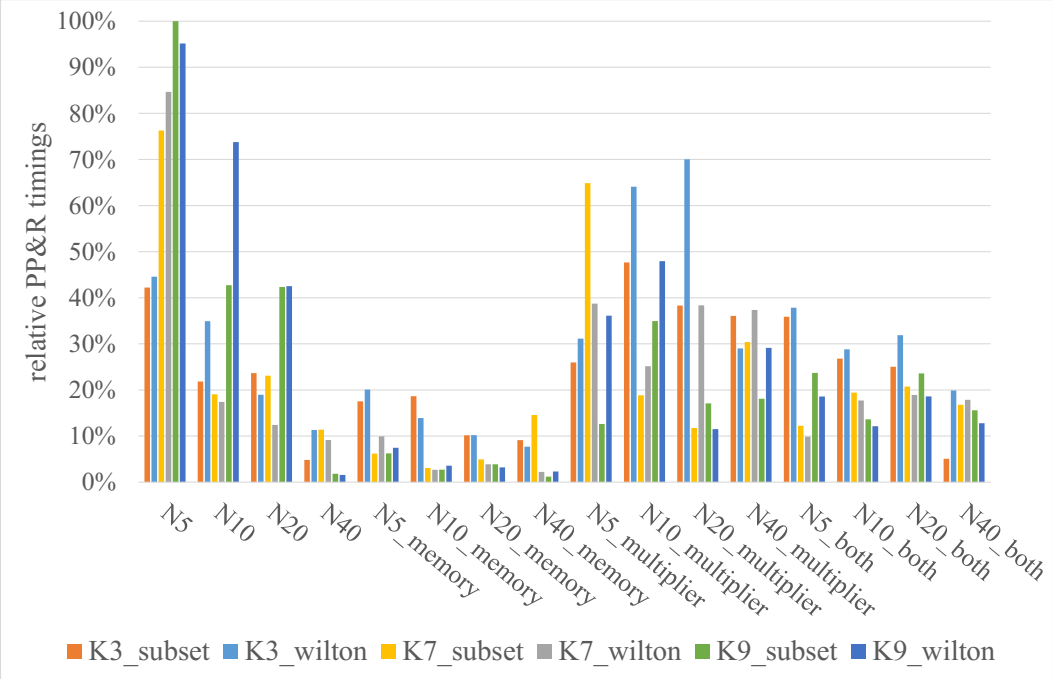

Abb. 16: Relative Ausführungsdauer von VPR in Bezug auf die maximale Ausführungsdauer

Als weitere Metrik wurde im Rahmen der Dissertation die Zeitdauer für „*Packing*“, „*Placement*“ und „*Routing*“ (PP&R) untersucht. In Abb. 16 ist jeweils die Gesamtzeit von VPR, welche sich in die Zeit für „*Packing*“, „*Placement*“ und „*Routing*“ aufteilt, für je eine Architekturvariante aufgezeigt. Das Maximum für diese Zeiten ergibt sich für große LUTs (K > 7) innerhalb einer kleinen Anzahl an *Basic Logic Elements* (N = 5) in einem CLB, unabhängig vom Switch Block Typ. Die Verwendung von Hardblocks für diese Architekturvarianten beeinflussen das Timing durchweg positiv. Dabei ist die Art des Hardblocks unerheblich. Für Architekturvarianten mit CLBs anderer Konfigurationen bestimmt die Art des Hardblocks, BRAM oder Multiplizierer, die Wirkung auf die Durchführungszeiten von VPR. Für die ausgeführten Benchmarks verursacht die Verwendung von BRAM die größte Verbesserung. Der Einsatz von Multiplizierern führt nicht generell zu einer Verbesserung. Wahrscheinlich ist, dass bei den verwendeten Benchmarks der Einfluss von BRAM stärker ausfällt als die Verwendung von Multiplizierern. So führt der zusätzliche Speicher zu einer

erheblichen Reduktion der *Complex Logic Blocks* und damit zu einer erheblichen Reduktion notwendiger Verbindungen. Für eine genauere Analyse müssten auch hier nochmals mehr Benchmarks ausgeführt werden. Die Abb. 17 bis Abb. 22 zeigen nochmals die Ergebnisse der Benchmarks aufgelöst nach den Eigenschaften der Architektur. Dabei werden hier die Durchlaufzeiten für VPR separiert nach „*Packing*“, „*Placement*“ und „*Routing*“ ausgeführt. Für feingranulare Strukturen verteilen sich die Verarbeitungsdauern annährend gleich, gemessen an der Gesamtzeit. Für Architekturen mit größeren LUT-Größen und CLBs domminiert sehr stark das *Routing*. Es nimmt mitunter mehr als 90% der Gesamtzeit ein. „*Packing*“ verzeichnet generell den geringsten Einfluss. Je grobgranularer die Struktur des Field Programmable Gate Arrays, also je mehr BLEs innerhalb einer CLB stecken, desto größer der Platzbedarf für die Architektur, aber desto geringer die Zeiten für das *Routing*. Dies liegt daran, dass sehr viel *Routing* bereits innerhalb der CLBs stattfinden kann. Durch die große Anzahl an BLEs innerhalb eines *Complex Logic Block* ist aber der Platzbedarf wesentlich größer, da die Multiplexer im Inneren einer CLB mehr Raum benötigen.

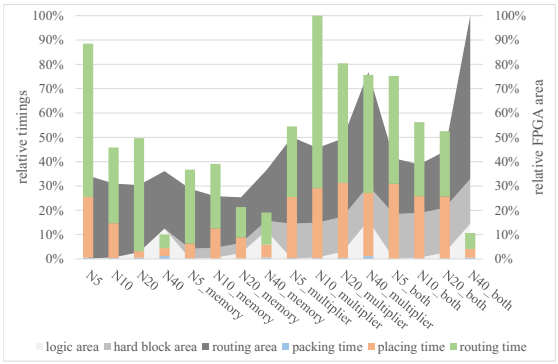

Abb. 17: Area und PP&R-Dauer
K: 3, Switch Block Typ: subset

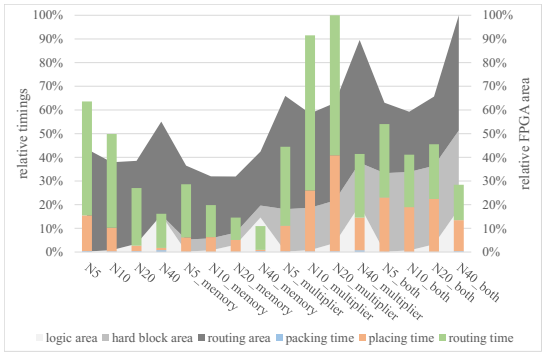

Abb. 18: Area und PP&R- Dauer
K: 3, Switch Block Typ: wilton

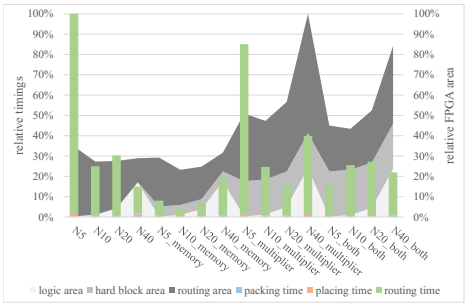

Abb. 19: Area und PP&R Dauer
K: 7, Switch Block Typ: subset

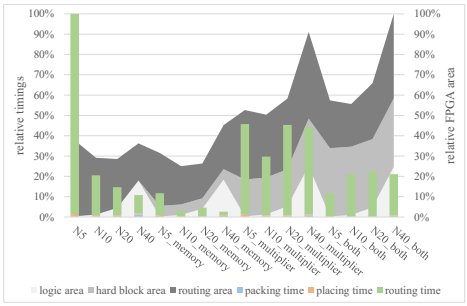

Abb. 20: Area und PP&R Dauer
K: 7, Switch Block Typ: wilton

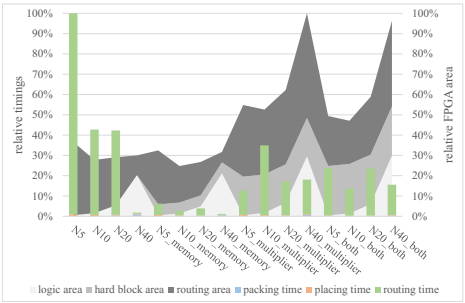

Abb. 21: Area und PP&R Dauer
K: 9, Switch Block Typ: subset

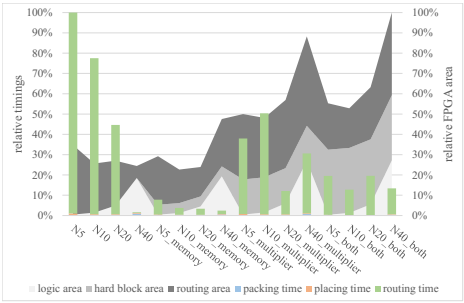

Abb. 22: Area und PP&R Dauer
K: 9, Switch Block Typ: wilton

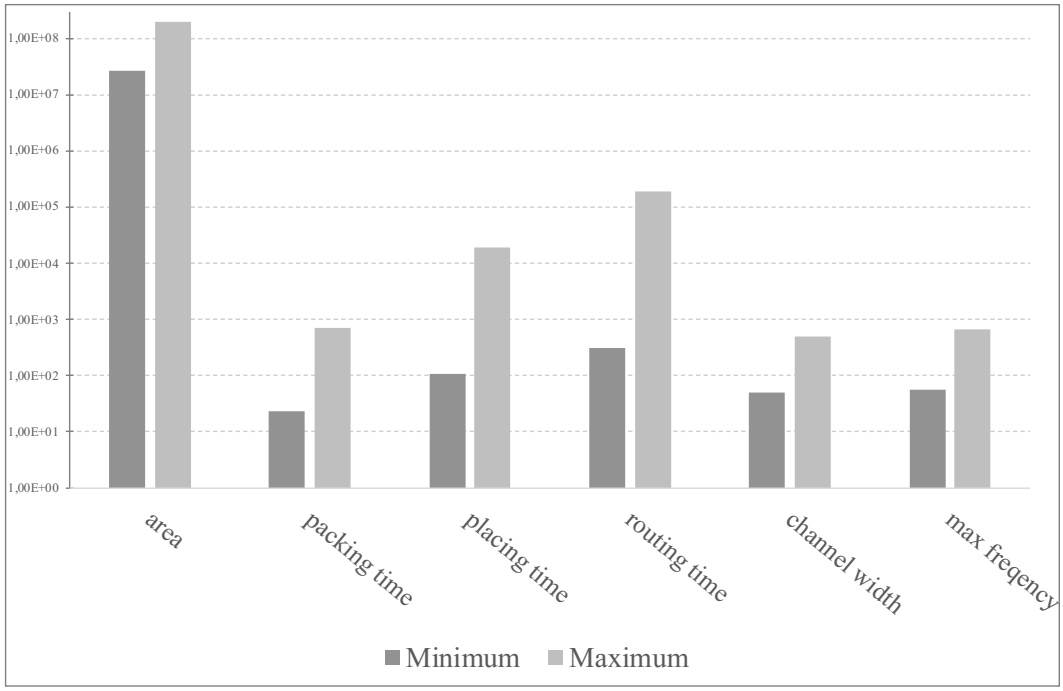

Abb. 23: Minimal- und Maximalwerte für alle Benchmark-Metriken

Zusammenfassend wurden im Rahmen dieser Untersuchung 650 Benchmarks mit mehr als 200 Architekturen durchgeführt. Den kleinsten Platzbedarf erreichte die Architektur mit einer LUT-Größe von fünf und 40 *Basic Logic Elements* innerhalb eines *Complex Logic Block*. Überraschend ist, dass weder der reduzierte Switch Block Typ „*wilton*“ noch zusätzliche Hardblocks eine Verbesserung des Platzbedarfes erreichten. Die Taktfrequenz kann um 12,5x erhöht werden, wenn größere CLBs, bestehend aus mehr als 20 BLEs, verwendet werden. Durch die Nutzung interner *Routing*-Ressourcen zwischen den BLEs einer CLB sinkt die Länge des kritischen Pfades und damit steigt die Performanz. Der positive Einfluss von Hardblocks auf die Taktfrequenz erscheint ebenfalls überraschenderweise gering. Es liegt die Vermutung nahe, dass die Auswahl der Benchmark-Applikationen nicht die nötige Variation beinhaltete, um Hardblocks entsprechend ausgiebig zu nutzen. Daher müsste im Rahmen weiterer Forschungsarbeiten die Anzahl der Benchmarks nochmals erhöht werden, um die Aussagekraft der Ergebnisse durch höhere Varianz in den Eigenschaften der Applikationen zu steigern. Das Mitteln der Ergebnisse sollte eine allgemeingültigere Aussage einer Hardwarekonfiguration auf beliebige Applikationen ermöglichen. Es verhindert aber auch, dass Vorteile für eine einzelne Applikation, wie beispielsweise die Verwendung von BRAM oder der SB-Typ „*wilton*“, markant werden. In Abb. 23 ist weiterhin erkennbar, dass die Metrik für die Kanalbreite (*channel width*) um 10 × schwanken kann zwischen kleinen *Complex Logic Block*, beispielweise K = 3 und N = 5, und großen CLBs mit beispielsweise K ⩾ 8 und N = 40. Dies liegt an dem höheren

externen Verdrahtungsaufwand außerhalb der *Complex Logic Blocks*, da nun weniger CLB-internes *Routing* stattfinden kann. Bei den Ausführungszeiten für PP&R dominiert der „*Routing*“ Prozess. Es ergeben sich für diese Metriken auch die größten Unterschiede zwischen Minimum und Maximum von 560x. Die längste Zeit benötigt eine Architektur mit einer LUT-Größe K = 8 und fünf *Basic Logic Elements* innerhalb einer CLB. Die Architektur verfügt sonst über keine weiteren Hardblocks. Die kürzeste Zeit benötigt eine Architektur mit K = 9 und N = 40 mit zusätzlichen BRAM-Blöcken. Dies ist ebenfalls begründet durch die geringere Anzahl an externem *Routing*, da sehr viele Verbindungen innerhalb der CLB stattfinden.

Die Resultate im Einzelnen zeigen keine optimale Granularität für alle betrachteten Metriken. Manche Konfigurationen sind besonders gut für den Platzbedarf geeignet, andere wiederum beeinflussen die Laufzeiten für PP&R besonders gut. Die Verwendung grobgranularer Blöcke steigert die Performanz der Architektur, aber auch den Platzbedarf. Reduziert man allerdings die Flexibilität der *Complex Logic Block* und spezialisiert diese Blöcke beispielsweise auf Teilgebiete der Informationsverarbeitung, wie die Lösung arithmetischer Aufgaben, kann der Platzbedarf dieser Blöcke reduziert werden. Aus diesem Grund wird eine grobgranulare Struktur in Form eines CGRAs angestrebt, welche in der Art der vorhandenen Operationen wie auch in der Bandbreite für die verarbeitenden Daten anpassbar ist. Des Weiteren, wie bereits weiter oben abgeleitet, soll durch Rekonfiguration während der Laufzeit die Chipgröße nochmals optimiert werden. Die Prozesselemente und Verbindungsstrukturen eines CGRAs, welche nicht wie bei einem FPGA durch dessen spaltenförmige Struktur wie nach Abb. 2 vorgegeben sind, sondern sich an den Datenstrom des Algorithmus anpassen lassen, können durch wenige Rekonfigurationsbits schnell umgestellt werden. Diese Reduktion des Konfigurationsaufwandes ist zwingend notwendig, da sonst die Performanzsteigerung und die Optimierung des Platzbedarfes durch den erhöhten Rekonfigurationsaufwand wieder reduziert oder gar vollständig aufgehoben werden.

2.1.3.5 Datenflussmodellierung mittels Graphen

Die angesprochenen arithmetischen Aufgaben lassen sich durch Datenflussgraphen darstellen. Ein einfaches Beispiel einer simplen Folge voneinander abhängiger arithmetischer Operationen soll diese Form der Darstellung näher vermitteln:

$$\begin{aligned} g &= a - b \\ f &= c + d \\ x &= g \times f \\ y &= max\{x, f\} \end{aligned} \tag{9}$$

Die Subtraktion und die Addition sind einfache Aufgaben einer Standard ALU. Sie benötigen wenig logische Zellen. Die Multiplikation ist bereits aufwändiger und es entstehen schnell Ergebnisse, deren Bitbreite nicht mehr in die Bitbreite der Operatoren passt. Deshalb werden zur Beschleunigung gerne Multiplikationseinheiten in Hardware zur ALU hinzugefügt. Das Maximum ist eine Funktion, welche sich per ALU durch eine Subtraktion und einer Überprüfung des „*negative*“ Statusbits prüfen lässt. Alle genannten Operationen bilden eine Grundlage für Computerprogramme. So werden Multiplikation und Addition in Bildverarbeitungsalgorithmen benutzt, welche auch als Anwendungen zur Evaluation der entstehenden Architektur Verwendung finden (siehe Abschnitt 2.3.6). Die Zwischenergebnisse werden wie durch die verwendeten Buchstaben verdeutlicht als Operanden für weitere Berechnungen verwendet, sodass sich ein Datenfluss ergibt.

Modelliert man die Operationen als Knoten und die Parameter als Kanten entsteht folgender Graph nach Abb. 24.

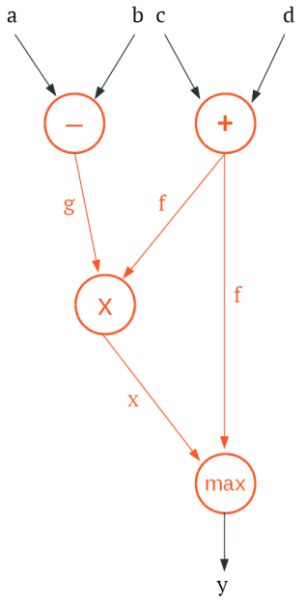

Abb. 24: Beispiel eines Datenflussdiagramms

Es lässt sich erkennen, dass sich aus den Abhängigkeiten der Gleichungen eine hierarchische Struktur für den Datenfluss von oben nach unten ergibt. Auch haben arithmetische Operationen den Vorteil, dass sie jeweils aus zwei Eingangsgrößen eine Ausgangsgröße bilden. Damit kann die Schnittstelle für ein Prozesselement, welche die Operationen in einem CGRA ausführen soll, vereinheitlicht werden. Damit können gleiche Prozesselemente an unterschiedlichen Stellen der Architektur auf jeweils die notwendige Operation konfiguriert werden.

2.1.3.6 Komplexitätsreduktion durch Mapping vs. Hardware-Synthese

Werden Applikationen auf programmierbarer Hardware implementiert, müssen durch Logiksynthese, Platzierung und *Routing* Signalwege hergestellt werden, um die entsprechenden Funktionalitäten abzubilden. Diese Aufgabe ist komplex und auch auf modernen Rechensystemen zeitaufwendig. Mehrere wissenschaftliche Arbeiten haben sich bereits mit der Beschleunigung von PP&R Algorithmen beschäftigt [17], [18], [19] und [20]. Trotz vieler verschiedener Ansätze sind die erzielten Beschleunigungen entweder zu gering, oder die Qualität der Ergebnisse für die Platzierung der Komponenten und deren Verbindung durch das Routing sind nicht zufriedenstellend.

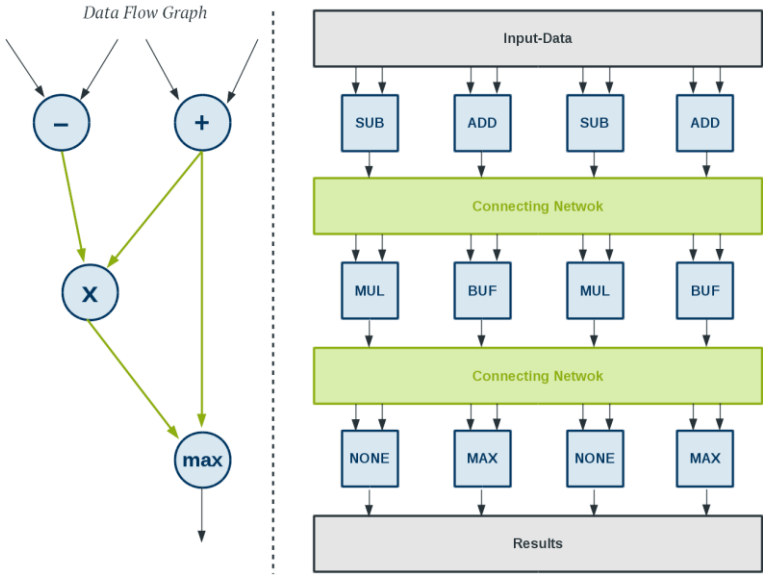

Abb. 25: Datenflussdiagramm, abgebildet auf einer CGRA-Architektur

Aus diesem Grund soll ein Ansatz verfolgt werden, bei dem die Repräsentation mittels Datenflussgraph direkt auf eine bestehende Architektur übertragen werden kann. Die Übertragung der Operationen auf Prozesselemente und die Herstellung der Verbindungen mittels Verbindungsnetzwerken wird innerhalb dieser Arbeit als „*Mapping*“ bezeichnet. Dieser Begriff wird in anderen Disziplinen synonym benutzt, wenn es die Abbildung eines Formates direkt in ein anderes Format beschreibt. In Abb. 25 ist zu erkennen, dass die Struktur eines hypothetischen CGRAs aus Prozesselementen besteht, welche die notwendigen Operationen implementieren. Dabei werden die Datenabhängigkeiten mittels Schichten dargestellt, die über Verbindungsnetzwerke verbunden sind. Ändert sich ein Datum innerhalb einer Schicht nicht, so kann dieser Status beispielsweise als Datenpuffer simuliert werden. Dies dient dazu zeitliche Abhängigkeiten wiederzugeben. Bietet die angenommene Architektur auch noch mehr verfügbare Ressourcen als unbedingt notwendig und sind die Daten für die Verarbeitung auch noch entkoppelt, können mehrere Instanzen auf der Architektur eine parallele Berechnung vollziehen und die Verarbeitung weiter beschleunigen. Andererseits muss, wenn die verfügbaren Ressourcen unzureichend sind, die Architektur in der Lage sein, durch schnelle Rekonfiguration den gesamten Algorithmus abzubilden. Für diese Aufgabe bedarf es zusätzlicher Hardwarekomponenten, welche das entsprechende Scheduling organisieren.

2.1.4 Virtuelle Architektur als „Overlay“ für bestehende FPGA-Architekturen

Die Entwicklung einer neuen Hardwarearchitektur mittels Hardwarebeschreibungssprachen wie VHDL oder Verilog, deren Simulation über entsprechende Designsoftware wie *Xilinx (AMD) Vivado®* oder Altera (Intel) Quartus®/Modelsim® und deren Produktion und Evaluation ist zeit- und geldintensiv. Entwicklungsteams von mehreren Personen benötigen Monate bis Jahre in einem iterativen Entwicklungsprozess. Daher lag der Fokus der vorliegenden Arbeit auf der Verwendung bestehender physikalischer Ressourcen, um damit den Entwicklungsprozess zu beschleunigen und Kosten zu senken.

Als Plattform für die Integration der Hardware ist ein FPGA verwendet worden. Komponenten der Architektur sowie das Gesamtdesign sind in VHDL beschrieben und mit Hilfe von *Xilinx Vivado®* auf verschiedene Zynq®-Plattformen synthetisiert. Dabei wird ein Multilevelansatz nach Abb. 26 verfolgt.

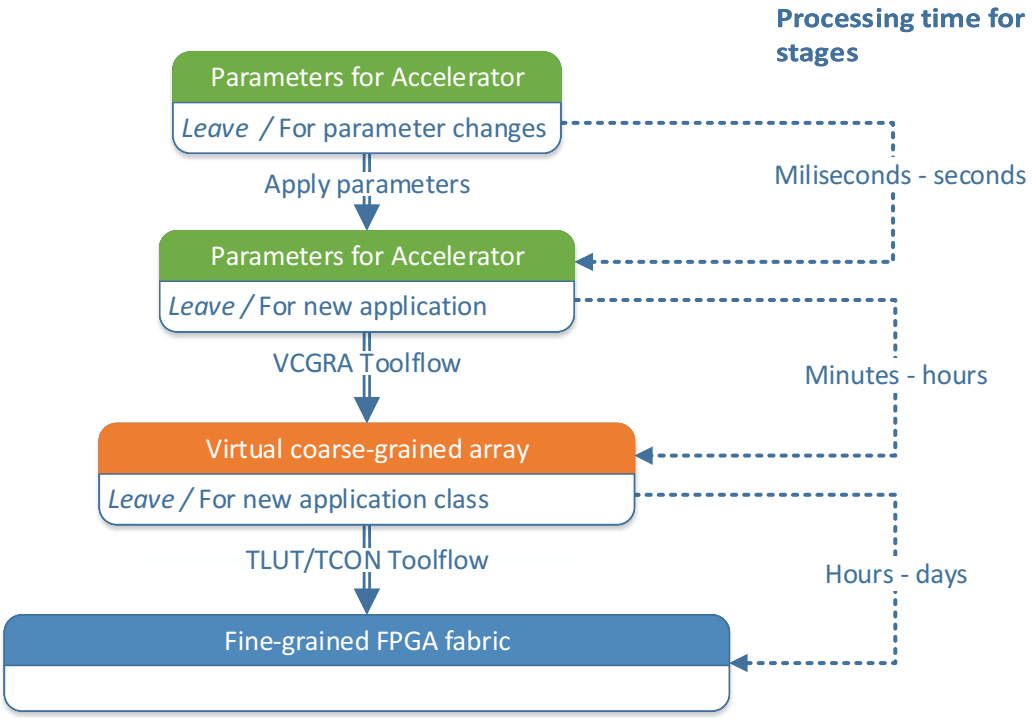

Abb. 26: Multilevelansatz für eine Beschleunigungsarchitektur und dessen Toolflow

Durch Abstraktion und Aufteilung der Hardwaregenerierung in einen Offline- (statisch) und einen Onlineteil (dynamisch; zur Laufzeit) soll die Dauer für die Laufzeitrekonfiguration des Beschleunigers reduziert werden. Basis der Implementierung bildet ein FPGA mit beliebigen Ressourcen. Über den Standard-Synthese-Prozess wird eine Instanz der noch beschriebenen Architektur als *Overlay* auf dem FPGA konfiguriert. Der entsprechende Bitstrom wird dabei nur einmal

erstellt, sodass der entsprechende Aufwand von Stunden oder Tagen im besten Fall nur einmalig auftritt. Der Bitstrom weist eine Besonderheit auf. Er soll mittels TLUT/TCON Toolflow erstellt werden. Dieser Toolflow reduziert seinerseits den Rekonfigurationsaufwand eines Bitstroms für Field Programmable Gate Arrays, wodurch erreicht werden soll, dass die Ressourcen des Field Programmable Gate Arrays noch optimaler ausgenutzt werden können. Details zum Toolflow werden im nachfolgenden Kapitel 2.2 näher erläutert. Die Generierung einer *Overlay*instanz soll automatisiert über ein entsprechendes Entwicklungswerkzeug erfolgen. Eine Instanz soll dabei für eine entsprechende Applikationsklasse generiert werden. Je spezieller die Instanz, desto ressourcenschonender ist die Implementierung, aber desto eingeschränkter ist seine Verwendung. Das Optimum für eine Implementierung soll durch eine entsprechende Analyse des Entwicklungsraumes und durch Simulationswerkzeuge erleichtert werden. Der „*VCGRA Toolflow*“, wie er in Abb. 26 genannt ist, erstellt eine entsprechende Instanz anhand von Parametern. Für zukünftige Entwicklungen wäre auch vorstellbar, dass die Eigenschaften einer Instanz aus dem zu beschleunigenden Kernelcode abgeleitet werden. VCGRA steht für „*Virtual Coarse Grained Reconfigurable Array*“ und beschreibt eine Architektur aus kleinen Funktions- oder Prozesselementen, welche über konfigurierbare Datenkanäle verbunden werden. Alle weiteren Entwicklungswerkzeuge adressieren eine Instanz eines solchen *Overlays*. Dieses besitzt seinerseits eine Konfiguration in Form eines Bitstromes. Da Bits dieses Bitstromes aber nicht Einträge einer Lookup-Table des Field Programmable Gate Arrays, sondern Funktionseinheiten oder *Routing*ressourcen der *Overlay*architektur adressieren, reduziert sich der Rekonfigurationsaufwand. Die Generation dieser Bitströme erfolgt automatisch mit Hilfe eines weiteren entsprechenden Entwicklungswerkzeuges. Dieses kann anhand der Eigenschaften einer *Overlay*instanz den Kernelcode in Cluster unterteilen und für diese Cluster Bitstromkonfigurationen erzeugen und ihren Ablauf planen. Veränderungen von Parametern führen beispielsweise zum Laden einer neuen Konfiguration für die *Overlay*architektur, womit sich beispielweise auch Lösungen für Kontrollanweisungen umsetzen lassen.

Die Erstellung einer Instanz eines *Overlays* sowie die Synthetisierung und Generierung eines FPGA Bitstromes sind statisch und werden „*Offline*“ ausgeführt. Die Komplexität ist zu gewaltig, als dass diese Operationen dynamisch und während der Laufzeit einer Anwendung ausgeführt werden könnten. Die Generierung erfolgt auf entsprechend leistungsstarken Entwicklungsrechnern oder Servern. Je nach Rechenleistung und verfügbarer Ressourcen kann die Generierung der Konfiguration

für ein *Overlay* auch „Online“, also direkt auf dem Zielsystem selbst erfolgen, zumindest aber das Laden neuer Konfigurationen anhand wechselnder Parameter.

Die Verwendung des Multilevelansatzes mit *Overlay*architekturen hat dabei folgende Vorteile:

- Die Adressierung einer *Overlay*instanz macht das Design unabhängig von der physikalischen FPGA-Ressource. Das Design kann auf eine beliebige Zielarchitektur synthetisiert werden.
- Die Erstellung von einzelnen Komponenten der *Overlay*architektur kann von Hardwaredesignern vorgenommen und in Form einer Bibliothek aus IP-Blöcken Softwareanwendern über ein „VCGRA-Toolflow-Werkzeug“ zur Verfügung gestellt werden. Dies abstrahiert Komplexität für den Anwendungsingenieur, da dieser die Details der IP-Blöcke nicht kennen muss.
- Erstellte Konfigurationen für eine *Overlay*instanz behalten ihre Gültigkeit, selbst wenn die physikalische Ressource FPGA ausgetauscht wird. Damit erhöht sich die Lebensdauer und Wiederverwendbarkeit von Implementierungen.
- Es werden Hardware-Software-Codesign Entwicklungsprozesse gefördert: Da der Anwendungsentwickler gegen eine virtuelle Architektur programmiert, kann der Hardwareingenieur die Komponente der Architektur verbessern und das Design bereits neu synthetisieren. Die Konfigurationen für die *Overlay*instanz des Anwendungsentwicklers bleiben bestenfalls erhalten, sodass eine erneute Generierung nicht notwendig wird.
- Die komplexen Berechnungen für die Bitstromgenerierung werden von Servern durchgeführt, sodass das Prinzip auch für eingebettete Prozessoren mit Hardwarebeschleunigern attraktiv wird. Als Firmware würde eine Instanz eines *Overlays* und die entsprechenden Bitströme zur Konfiguration programmiert. Kleinere Anpassungen oder das Laden entsprechender Bitströme könnten leicht, da nicht sonderlich komplex, von einem Mikroprozessor durchgeführt werden.

Das Konzept verfolgt dabei den Einsatz eines Hardwarebeschleunigers parallel zu einem Prozessor, welcher die vollständige Bearbeitung des Algorithmus steuert, im Gegensatz zu [1], bei dem eine gesamte Applikation auf einem rekonfigurierbaren Computersystem stattfindet. Es erleichtert das Design einer Hardwarearchitektur, da beispielsweise Kontrollflussanweisungen in Algorithmen nur eingeschränkt beachtet werden müssen. Damit spielt die Anbindung zwischen Prozessor und Beschleuniger

eine entscheidende Rolle. Entsprechende Konzepte für die Bereitstellung der Daten bzw. die asynchrone Bearbeitung von Daten von Beschleuniger und Prozessor wird in Kapitel 3 und 4 diskutiert. Im Folgenden werden zunächst die Kernelemente des VCGRA vorgestellt, doch zuvor wird eine Übersicht zum TLUT/TCON Toolflow gegeben, da dessen Einschränkungen/Anforderungen das Design der Komponenten des *Virtual Coarse Grained Reconfigurable Arrays* stark beeinflussen.

2.2 Übersicht über den TLUT/TCON Toolflow

TLUT und TCON stehen als Abkürzung jeweils für „*Tunable Lookup-Tables*“ bzw. „*Tunable Connections*“. Sie bilden die Grundlage für eine Technik, welche als „*Dynamic Circuit Specialization*“, kurz DCS, bezeichnet wird [21], [22]. In einem Algorithmus gibt es Parameter, welche sich im Vergleich zu Daten seltener ändern, aber die Verarbeitung des Algorithmus trotzdem beeinflussen. Ein solches Beispiel wären die Koeffizienten eines Filters oder einer Faltung. Der TLUT/TCON-Ansatz bezeichnet solche Faktoren als Parameter und weist diese als solche innerhalb der VHDL-Quelldateien aus (siehe auch Code 1). Die als Parameter markierten Eingangsdaten werden für ihr aktuelles Design als Konstanten angenommen, worauf sich das Hardwaredesign für diese Konstanten optimieren lässt. Ändern sich die als Parameter markierten Eingangssignale, wird das Design durch Rekonfiguration für die neuen Parameter optimiert. Konventionelle Rekonfigurationstechniken sind zu langsam bzw. zu unflexibel, und der Overhead für die Rekonfiguration würde den Performanzgewinn durch die optimierten Designs aufheben. Konventionelle Rekonfigurationstechniken benötigen entweder eine Bibliothek fertiger Bitströme für alle möglichen Parametervarianten oder es muss der gesamte PP&R Arbeitsablauf ausgeführt werden. Aus diesem Grund verfolgt der TLUT/TCON Toolflow einen zweistufigen Ansatz, um die Optimierung auf Parameteränderungen zu beschleunigen [2]. Dieser ist in Abb. 27 dargestellt und unterteilt sich in einen allgemeinen Teil („*Generic Stage*“) und einen Spezialisierungsteil („*Specialization Stage*“).

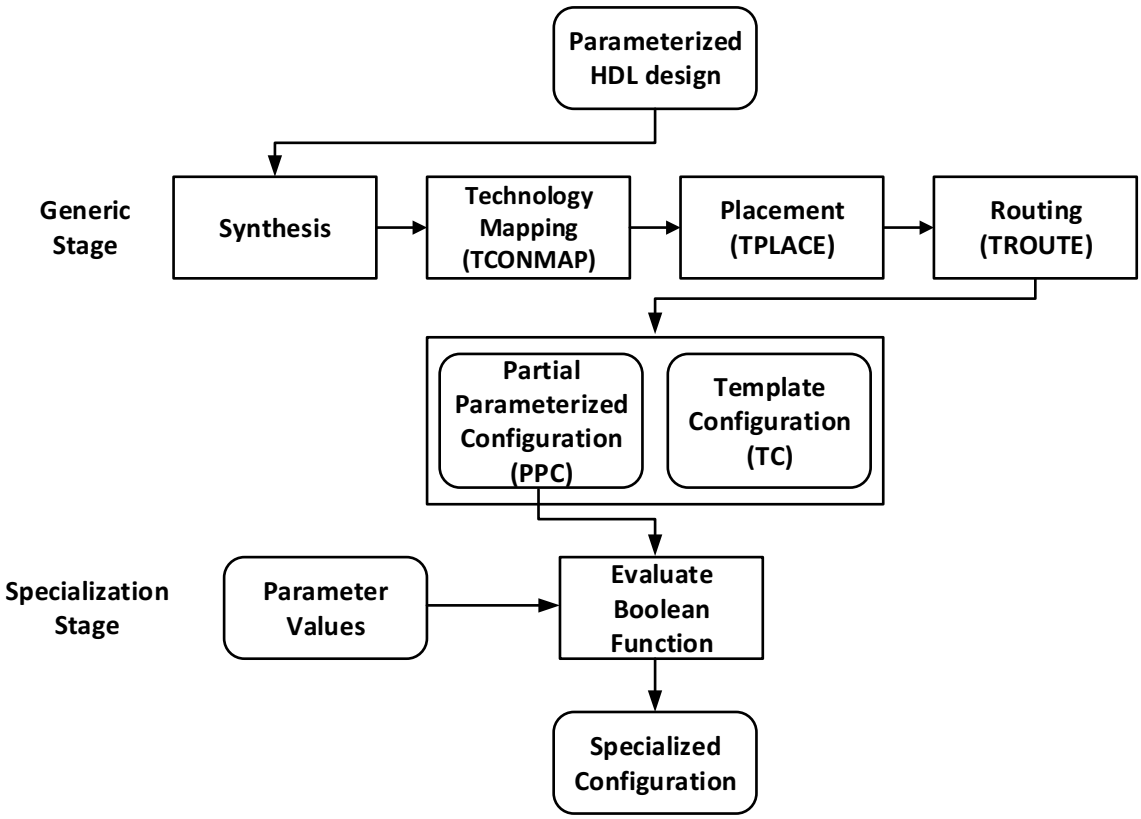

Abb. 27: Übersicht TLUT/TCON Toolflow [2]

Grundlage für den Toolflow bildet eine Designbeschreibung in VHDL, bei dem die als Parameter (siehe auch Code 1) verwendeten Eingangsdaten eines Designs innerhalb einer mit „`--PARAM`“ markierten Sektion in der VHDL-Quelldatei eingeschlossen stehen. Die Synthese unterscheidet sich nicht von einem konventionellen Prozess, es wird sich nur zusätzlich gemerkt, welche Eingangsdaten als Parameter markiert wurden. Für das Mapping auf eine Zieltechnologie wird eine eigene Software namens „*TCONMAP*“ verwendet. Im Unterschied zu konventionellen Mappern verwendet dieser Mapper TLUTs und TCONs als besondere Ziele für die Ergebnisse der Synthese. TLUTs und TCONs sind im Unterschied zu LUTs und Switch Blocks bzw. Connection Blocks virtuell. Es entspricht einer weiteren Abstraktionsschicht eines FPGA-Designs.

Ihre Konfiguration ist nicht fest, sondern durch Bool'sche Ausdrücke beschrieben, deren Lösungen abhängig sind von den als Parametern definierten Eingängen des Designs:

- TLUT: Die Einträge der LUT werden durch die Lösung der Bool'schen Gleichungen bestimmt.
- TCON: Die Punkt-zu-Punkt Verbindungen zwischen zwei Switch Blocks oder Connection Blocks werden durch die Lösung einer Bool'schen Gleichung getrennt oder geschlossen.

Auf Grund der speziell adressierten Hardwarekomponenten für die LUTs und die *Routing*ressourcen werden ebenfalls spezielle *Placer* und *Router* verwendet, sie heißen *TPLACE* und *TROUTE* [23]. Als Resultate der Verarbeitung entstehen zwei Designblöcke:

- *Template Configuration* (TC): Dieser Teil des Designbitstromes enthält unveränderbare Designelemente, welche nicht durch Parameteränderungen angepasst werden müssen.
- *Partial Parameterized Configuration* (PPC): Diese Konfiguration besteht aus Bool'schen Gleichungen, welche durch Parameteränderungen gelöst werden.

Durch Lösung der Bool'schen Gleichungen und durch einen *Merge* der entstehenden Bitströme wird für jede Parameterkonstellation jeweils ein eigener, optimierter Designbitstrom berechnet. Die Lösungen der Bool'schen Gleichungen sind dabei um ein Vielfaches schneller als die Generierung eines vollständigen, neuen partiellen Bitstromes zur Rekonfiguration.

Überträgt man das Prinzip der Parameter und Daten auf ein CGRA Design, so ändern sich die Konfiguration eines CGRA-Designs für die Verbindungen und Prozesselemente nur niederfrequent im Vergleich zu den Daten, welche durch das CGRA hindurchlaufen.

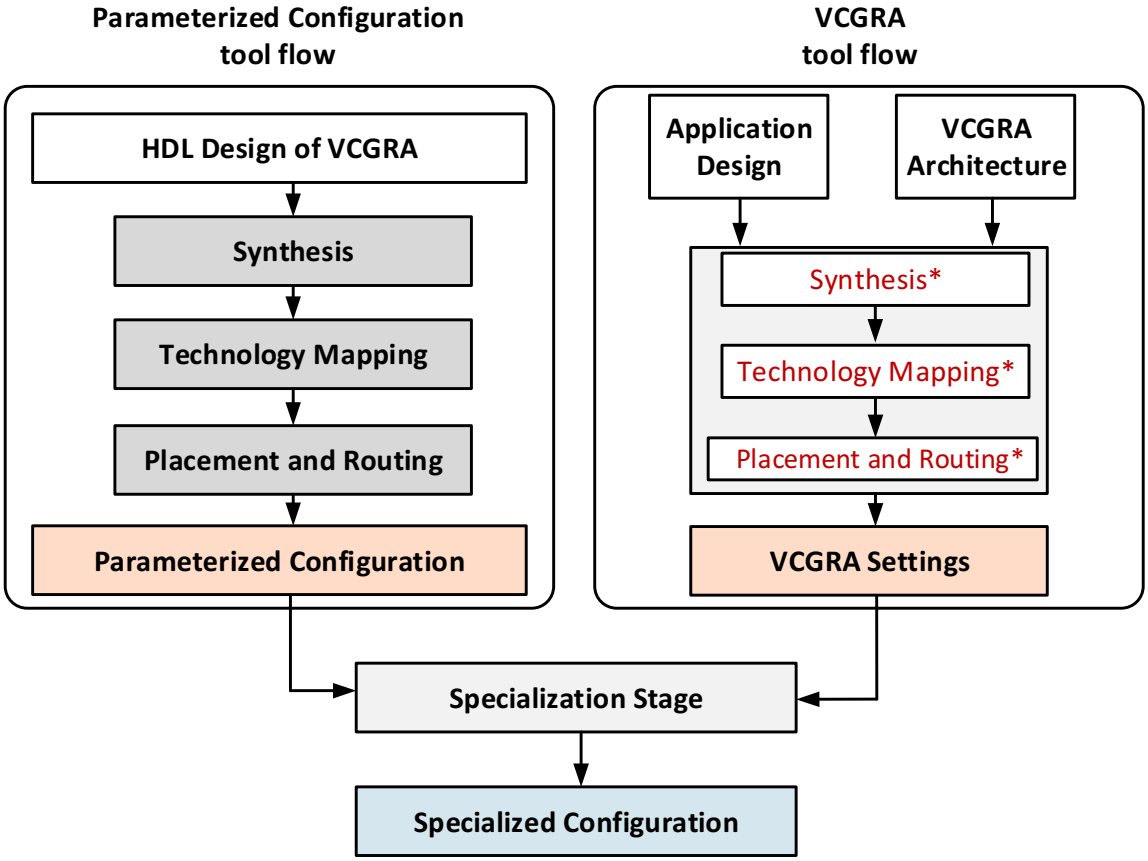

Abb. 28: Entwurfsvorschlag für ein VCGRA mit Laufzeitrekonfiguration und TLUT/TCON-Integration

Im Hardwaredesign des VCGRA werden daher die Ports für die Konfigurationen der Verbindungskanäle und der Prozesselemente als Parameter gekennzeichnet. Dies ist beispielhaft in Code 1 aufgeführt. Das VCGRA in diesem Codebeispiel ist generiert für eine Faltungsoperation, bei der jeweils vier Elemente der Faltung mit ihren Gewichten multipliziert werden können. Die Koeffizienten der Gewichte sind ebenfalls als Parameter markiert, da sie sich durch den Algorithmus bedingt auf mehrere Datenobjekte anwenden lassen. In Abb. 28 ist passend dazu ein erster Toolflow beschrieben, welcher die Technologien des TLUT/TCON Toolflows integriert. Die Beschreibung des *Virtual Coarse Grained Reconfigurable Arrays* mit seinem parametrisierbaren Verbindungs- und Prozesselementen wird zunächst, wie auf der linken Seite angedeutet, in einem parametrisierbaren Bitstrom übersetzt. Über die rechte Seite wird eine Applikation unter Beachtung der Eigenschaften des *Virtual Coarse Grained Reconfigurable Arrays* auf diesem platziert. Dadurch entstehen Parameter für die Konfigurationen der Prozesselemente und virtuellen Kanäle. Diese Parameter werden dann zur Spezialisierung des parametrisierten Bitstromes genutzt, um die Bool'schen Gleichungen zu lösen und eine neue Konfiguration für den FPGA zu erzeugen. Dabei wird versucht, die Konfigurationen und die Verbindungen jeweils

auf TLUTs bzw. TCONs zu implementieren, um damit das Design schlank und ressourcensparend auszuführen.

```
entity vcgra is
    port (
        pixIn_00  : in  signed(31 downto 0);
        pixIn_01  : in  signed(31 downto 0);
        pixIn_02  : in  signed(31 downto 0);
        pixIn_03  : in  signed(31 downto 0);
        --PARAM
        coeffIn_00  : in  signed(31 downto 0);
        coeffIn_01  : in  signed(31 downto 0);
        coeffIn_02  : in  signed(31 downto 0);
        coeffIn_03  : in  signed(31 downto 0);
        --PARAM
        pixOut_00  : out  signed(31 downto 0);
        pixOut_01  : out  signed(31 downto 0);
        pixOut_02  : out  signed(31 downto 0);
        pixOut_03  : out  signed(31 downto 0);
        ready : out std_logic;
        --PARAM
        confPE  : in  std_logic_vector(47 downto 0);
        confCh  : in  std_logic_vector(55 downto 0);
        confMa  : in  std_logic_vector(3 downto 0);
        --PARAM
        clk     : in  std_logic;
        rst     : in  std_logic;
        start   : in  std_logic
    );
end entity vcgra;
```

Code 1: VHDL-Beispiel – VCGRA Entity mit TCON/TLUT Parametern

2.2.1 Evaluationsbeispiel – Verwendung eines VCGRA zur Bildverarbeitung

In [24] wurde eine Faltungsoperation für eine Kantenerkennung („*Edge Detection*“) auf solch beschriebene Weise implementiert. Das VCGRA, dessen Basiskomponenten noch in Abschnitt 2.3 genauer beschrieben werden, wird in dieser wissenschaftlichen Veröffentlichung so gewählt, dass der gesamte Filter mit 3×3 Filterkoeffizienten auf diesem vollständig platziert werden kann. Als Software wird VTR [15] verwendet, um die konventionellen Ergebnisse nach Tab. 3 zu errechnen. Das konventionelle Design verwendet keine Rekonfigurationsmöglichkeiten zur Laufzeit. Für die parametrisierten Ergebnisse in Tab. 3 werden die bereits aufgezählten Werkzeuge *TCONMAP*, *TPLACE* und *TROUTE* benutzt [21], [22], [2], [23], [25].

Tab. 3: Ergebnisse einer VCGRA Umsetzung für eine Kantendetektion (Faltung)

Item	LUTs (TLUTs)	TCONs	Logic Depth Level	Wire Length	Minimal Channel Width
Virtual Channel (konventionell)	176 (0)	0	2	3186	7
Virtual Channel (parametrisiert)	32 (0)	72	1	782	4
PE (konventionell)	408 (0)	0	47	3832	8
PE (parametrisiert)	387 (32)	22	47	3769	8
VCGRA (konventionell)	17066 (0)	0	155	176200	14
VCGRA (parametrisiert)	16099 (976)	561	153	169560	12

Es wurden jeweils die Elemente des *Virtual Coarse Grained Reconfigurable Arrays* einzeln und das gesamte VCGRA durch die Werkzeuge auf eine virtuelle 4-LUT FPGA-Architektur umgesetzt. Die Verwendung von VTR [15] zur Bestimmung der konventionellen Bezugsgrößen ist auf solche virtuellen FPGA-Architekturen begrenzt. Auch die Verwendung von TLUTs und vor allem TCONs war zum Zeitpunkt der Veröffentlichung auf solche virtuellen Field Programmable Gate Arrays reduziert. Für die Anpassungen der Bitströme zur Rekonfiguration eines echten Field Programmable Gate Arrays während der Laufzeit durch Lösung der Bool'schen Gleichungen (siehe Abb. 27: Übersicht TLUT/TCON Toolflow) ist eine Kenntnis der korrespondierenden Konfigurationsbits für einen Switch Block oder einen Connection Block im FPGA ausschlaggebend. Das Wissen um diese Korrelation ist geschützt durch die FPGA Hersteller, weshalb die entsprechenden Korrelationen durch Reverseengineering ermittelt werden müssen. Das entsprechende Wissen über aktuelle FPGA-Architekturen ist daher noch lückenhaft oder fehlt gar vollständig.

Durch den Aufbau des virtuellen Kanales können knapp 70% der notwendigen Ressourcen über Switch Blocks und Connection Blocks (TCONs) der FPGA-Architektur statt durch LUTs implementiert werden. Zum konventionellen Design werden darüber hinaus 40% der Ressourcen eingespart. Die Ergebnisse in der Signalweite („*Wire*

Length“) und der minimalen Kanalweite („*Minimal Channel Width*“) sind ebenfalls vielversprechend mit jeweils 75% und 43% Einsparung im Vergleich zum konventionellen Design. Der Einfluss des TLUT/TCON Toolflows auf das Design eines Prozesselements ist geringer. Die insgesamt verwendeten Ressourcen sind nahezu identisch, der Anteil der verwendeten LUTs kann aber um 5% reduziert werden, da ein Teil der Logik auf die Switch Blocks und Connection Blocks augenscheinlich ausgelagert wird. Die Signalweite reduziert sich um 1,6%. Auch für das komplette Design zeigen sich insbesondere Vorteile für die Kanalbreite mit einer Reduzierung um 14% und der Signalweite mit einer Reduzierung um 3,7%. Die notwendigen Logikressourcen sind um etwa 2,5% gesenkt.

Ergebnisse von mehr als 40% Einsparpotential zeigten an dieser Stelle ein hohes Potential zur weiteren Evaluierung des VCGRA Designs. Die Möglichkeiten der schnellen Rekonfiguration während der Laufzeit eröffnen weitere Vorteile:

- Verwendung kleinerer Field Programmable Gate Arrays senkt die Kosten für ein Design.
- Kürzere *Time-To-Market*, da Standarddesigns weiterverwendet werden können (linke Seite in Abb. 28) und nur die Einflüsse der Parameteranpassungen neu berechnet werden müssen (rechte Seite in Abb. 28).

Die Ergebnisse dieser ersten Untersuchung basieren auf den im Folgenden beschriebenen Komponenten nach Abschnitt 2.3: Prozesselemente und virtuelle Kanäle.

2.2.2 Evaluationsbeispiel – “Superimposed In-Circuit Fault Tolerant Architecture”

Basierend auf der Idee und der Implementierung nach Abschnitt 2.2.1 wurde in [26], [27] das Design um einen weiteren Layer für die Minderung von Fehlern SRAM-basierter Field Programmable Gate Arrays durch eine mehrstufige Konfigurationskorrektur in Anwendungen für die Raumfahrttechnik[4] erweitert. Durch kosmische Strahlung kann es im Weltall zu ungewollten Bitflips kommen, sodass die Konfigurationen SRAM-basierter Field Programmable Gate Arrays nicht mehr mit ihrem Soll übereinstimmen und die Funktion fehlerhaft oder eingeschränkt verläuft. Eine Korrektur/Prüfung des Zustandes findet dabei in diesen Arbeiten nicht auf Gate-

[4] Dieser Abschnitt beschreibt dabei die Nutzung der vom Autor erstellten Architektur. Der Autor hatte dabei an den vorgestellten Erweiterungen und dessen Ergebnissen keinen direkten Anteil.

Ebene der FPGA-Ressourcen statt (*State-of-the-Art*), sondern auf Ebene des *Virtual Coarse Grained Reconfigurable Arrays*. Weiter wird beschrieben, dass die Korrekturen in einem dreistufigen Prozess stattfinden:

1. *Discrete Microscrubbing*: Ein Konfigurationsfehler („*Single-Event-Upset*“), wird lokal detektiert und nur der fehlerhafte Konfigurationsbitstrom wird aus nichtflüchtigem Speicher wieder hergestellt.
2. *Microscrubbing*: Sequenziell wird über das VCGRA hinweg die Konfiguration für jeden virtuellen Kanal und jedes Prozesselement aus dem nichtflüchtigen Speicher wieder hergestellt. Dies dient der Korrektur von „*Multi-Bit-Upsets*“.
3. *Parameterized Scrubbing*: Die Funktion des *Virtual Coarse Grained Reconfigurable Arrays* wird durch die TLUT/TCON Technik für eine neue Aufgabe rekonfiguriert. Dies geschieht durch die Evaluation Bool’scher Gleichungen anstatt einer neuen Synthese des Designs (siehe Abschnitt 2.2).

2.2.3 Evaluationsbeispiel – „Superimposed Debugging Architecture“ (SDA)

Ebenfalls basierend auf der Idee und Implementierung nach Abschnitt 2.2.1 und als Erweiterung der Architektur nach Abschnitt 2.2.2 wird in [27] eine VCGRA-basierte Debugging Architektur vorgestellt.[4] Debugging von Hardware auf FPGA-Ebene ist komplex und zeitaufwendig. Die zu untersuchenden Signale müssen vor der Synthese bestimmt werden. Ändert sich das Interesse an Signalen, muss eine neue Synthese des kompletten Designs stattfinden. Das kostet sehr viel Entwicklungszeit. Je mehr Signale untersucht oder aufgezeichnet werden sollen, desto mehr Fläche wird auch für den Debugging-Controller benötigt, was Ressourcen für das eigene Design bindet.

Die Architektur besteht ebenfalls wieder aus zwei Ebenen. Eine Ebene ist erneut das VCGRA bestehend aus virtuellen Kanälen und Prozesselementen. Die zweite Ebene fügt für die Ein- und Ausgänge der PEs und virtuellen Kanäle *Trace-Buffer* und Multiplexer hinzu. Diese Buffer speichern den Zustand der VCGRA-Komponenten von Interesse. Für das *In-Circuit Debugging* der Kanäle werden die Zustände der Ausgänge verbundener virtueller Kanäle und Eingänge der angeschlossenen PEs verglichen. Für das *In-Circuit Debugging* der Prozesselemente wird das Ergebnis einer PE mit dem Ergebnis eines Prozessors verglichen, welcher dieselbe Berechnung ausführt. Um für die *Trace-Buffer* Ressourcen zu sparen und während der Laufzeit die Elemente von Interesse wechseln zu können, ist diese zweite Ebene rekonfigurierbar mit Hilfe der TLUT/TCON Technik aufgebaut.

Neben der Evaluation durch eine Sobel-Operation ist in [27] das modulare Design der VCGRA-Architektur erweitert um Funktionalitäten eines Convolutional Neuronale Network (CNN).

2.3 Grundlegende Architektur

Die grundlegende Struktur des VCGRA orientiert sich am Aufbau eines Datenflussgrafen (siehe auch Abschnitt 2.1.3.5 und Abschnitt 2.1.3.6). Dabei bilden die Knoten des Grafen die Operationen der Prozesselemente ab, während die Kanten des Grafen mittels so genannter virtueller Kanäle dargestellt werden.

2.3.1 Komponente Prozesselement

Das Prozesselement (*Processing Element*) in der Gesamtstruktur des VCGRA führt die Verarbeitung von Daten aus. Die Abb. 29 zeigt einen vereinfachten schematischen Aufbau dieser Komponente, reduziert auf wesentliche Bestandteile und entspricht nicht dem RTL-Design nach einer Logiksynthese. Vielmehr sind einige interne Elemente als eine Art Blackbox zusammengefasst, um die Darstellung nicht zu überladen. Wenn gewünscht, kann der Quellcode in VDHL mittels *Altera Modelsim®* oder *Xilinx Vivado®* synthetisiert und betrachtet werden. Der Quellcode verwendet keine herstellerspezifischen VHDL-Designelemente, sondern nur Grundelemente aus dem VHDL-Standard IEEE 1076 [28].

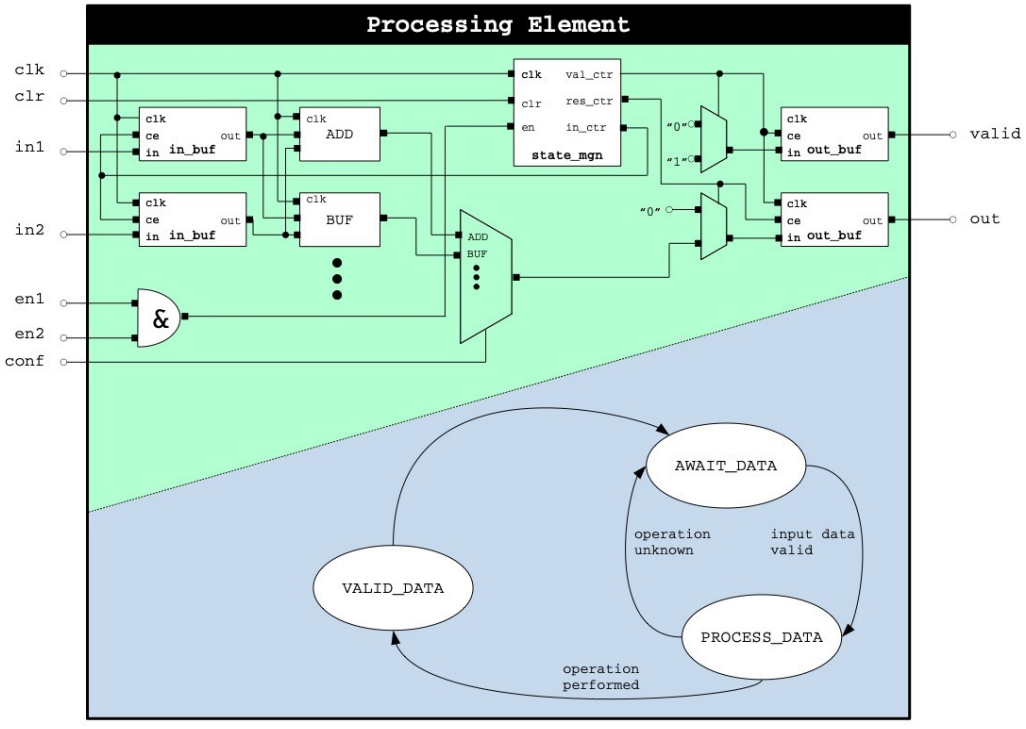

Abb. 29: Schematische Darstellung – Processing Element

Das Prozesselement besitzt sieben Eingangsports und zwei Ausgangsports:

- `in[1,2]`: Dies sind die Eingangsports für die Nutzdaten. Sie sind als Vektor definiert, wobei die Größe des Vektors als *Template*-Parameter bei der Instanziierung einer PE gesetzt werden kann. Damit lassen sich innerhalb eines *Virtual Coarse Grained Reconfigurable Arrays* auch Prozesselemente mit unterschiedlicher Bitbreite für die Eingangsdaten definieren. Dies soll die Flexibilität erhöhen und wenn notwendig, Chipfläche einsparen.
- `en[1,2]`: Das Prozesselement besitzt jeweils einen „*enable*“ Port korrespondierend zu jedem Input-Datenport (`in[1,2]`). Er dient dazu, die Daten an diesem Port als valide oder gültig zu markieren. Erst wenn beide `enable`-Ports mit „*High*“ innerhalb eines Zyklus gleichzeitig die Daten als valide markieren, wird die Bearbeitung fortgesetzt.
- `clr`: Mit diesem Eingang lässt sich der aktuelle Wert im Ausgangsregister mit Nullen überschreiben, um damit das Ergebnis einer Berechnung zurücksetzen. Das Signal muss für mindestens einen Taktzyklus anstehen. Die Auswertung ist „*active low*“. Bleibt das Signal auf „*active low*”, bleibt der Ausgang dauerhaft auf Null.
- `clk`: Es handelt sich bei diesem Eingang um den Takt. Die Verarbeitung der Prozesselemente findet immer bei einer positiven Taktflanke statt.
- `conf`: Dieser Port dient der Konfiguration der aktuellen Operation des Prozesselements. Er wird als Parameter für den TLUT/TCON Toolflow markiert, sodass das Design entsprechend auf die aktuelle Operation optimiert werden kann.
- `valid`: Das `valid`-Signal dient zur Synchronisation der Ausgangsdaten mit einem nachfolgenden Prozesselement. Es wird genutzt, um das `enable`-Signal am Eingang einer nachfolgenden Einheit zu triggern, um so die Verarbeitung dieser Einheit anzustoßen.
- `out`: Dies entspricht dem Ergebnis der Operation des Prozesselements. Es handelt sich auch hier um einen Bitvektor, dessen Bitlänge während der Instanziierung für ein Prozesselement eingestellt werden kann. Sie ist entsprechend als *Template*-Parameter im Design festgelegt.

Die Bitlänge der Eingänge zur Bitlänge des Ausgangs kann sich unterscheiden. Dies ist dann sinnvoll, wenn Operationen verwendet werden, welche größere Bitlänge im Ergebnis verursachen können – beispielsweise Multiplikation – oder bei denen kleinere Bitlängen ausreichen – zum Beispiel Vergleichsoperationen mit Bool’schen Ergebnis. Auch hier kann diese Flexibilität zur Einsparung von Chipfläche verwendet

werden. Die Bitlänge eines Designs kann durch den Designer einer Architektur vorgegeben werden oder lässt sich an den Eigenschaften der Applikation festsetzen. Während der Laufzeit lassen sich die Bitlängen nicht mehr beeinflussen. Es muss sich während der Designphase, also vor der Synthese des *Virtual Coarse Grained Reconfigurable Arrays*, darauf festgelegt werden.

Die Entity für dieses Prozesselement ist in Code 2 exemplarisch aufgeführt.

```
entity PE is
  generic (
    N : positive := 8;                          --bitwidth for input data type
    K : positive := 8                           --bitwidth for output data type
  );
  port (
    In1    : in  signed (N - 1 downto 0);        --operand one
    In2    : in  signed (N - 1 downto 0);        --operand two
    --PARAM
    Conf     : in  std_logic_vector(3 downto 0);  --operation of PE (--PARAM)
    --PARAM
    Clk      : in  std_logic;                     --PE clock
    clear_reg : in  std_logic;                    --clear accumulator register
    Enable     : in  std_logic_vector(1 downto 0); --mark inputs of PE as valid
    Res      : out signed (K - 1 downto 0);       --result of operation
    Valid    : out std_logic                      --mark current result as valid
  );
end entity PE
```

Code 2: VHDL-Entity eines Processing Element

Aus der VHDL-Beschreibung ist erkennbar, dass das Prozesselement derzeit für Ganzzahltypen konzipiert und dass die Konfiguration für die aktuelle Operation als Parameter für den TLUT/TCON Toolflow markiert ist. Die Länge des Konfigurationsvektors `conf` ist in diesem Designbeispiel auf die Anzahl der verfügbaren Operationen ausgelegt. Diese sind: `ADD`, `SUB`, `MUL`, `IDIV`, `GRE`, `EQL`, `MDL` und `BUF`. Die Operationen sind als VHDL-Package programmiert für einfache Wartbarkeit und Erweiterung und sind in Code 32 im Anhang I. Quellcodeanhänge aufgeführt.

Das Prozesselement arbeitet als *State Machine* mit drei Zuständen. Die Abhängigkeiten der Zustände sind in Abb. 29 ebenfalls schematisch abgebildet. Ein Prozesselement befindet sich überwiegend im Zustand „`AWAIT_DATA`“, um auf die Prozessdaten zu warten. Wenn die entsprechenden Daten über die `enable`-Ports zur weiteren Verarbeitung freigegeben wurden, verarbeitet das Prozesselement die Daten

im „PROCESS_DATA“-Zustand und gibt sie für einen Zyklus im „VALID_DATA“ Zustand „frei“. Das vorliegende Design hat den Nachteil, dass die verfügbaren Operationen innerhalb eines Taktzyklus beendet werden müssen. Dadurch werden komplexe Operationen in Hardware, beispielsweise eine Division, sehr groß, das heißt, sie benötigen sehr viel Chipfläche. Das Design gibt hier aber Möglichkeiten zur Verbesserung: Die Implementierung der Operationen im Package (siehe Code 32) kann entsprechend angepasst werden, um komplexere Operationen auf mehrere Taktzyklen zu verteilen. Die Umschaltung in den nächsten Zustand kann entsprechend gesteuert und damit auch verzögert werden. Durch das `valid`-Ausgangssignal und das `enable`-Signal des folgenden Prozesselements ist die weitere Verarbeitung von Daten synchronisiert. Entsprechende Designoptimierungen werden in Abschnitt 3 beschrieben.

2.3.2 Komponente virtuelle Kanäle

Die Verbindung der Prozesselemente findet mit Hilfe virtueller Kanäle (*Virtual Channel*) statt, denn virtuelle Kanäle verbinden jeweils zwei adjazente (benachbarte) Ebenen aus Prozesselementen. Dabei kann in dem Design nach Abb. 30 jede PE aus der Vorgängerebene mit jeder PE aus der Nachfolgerebene verbunden werden.

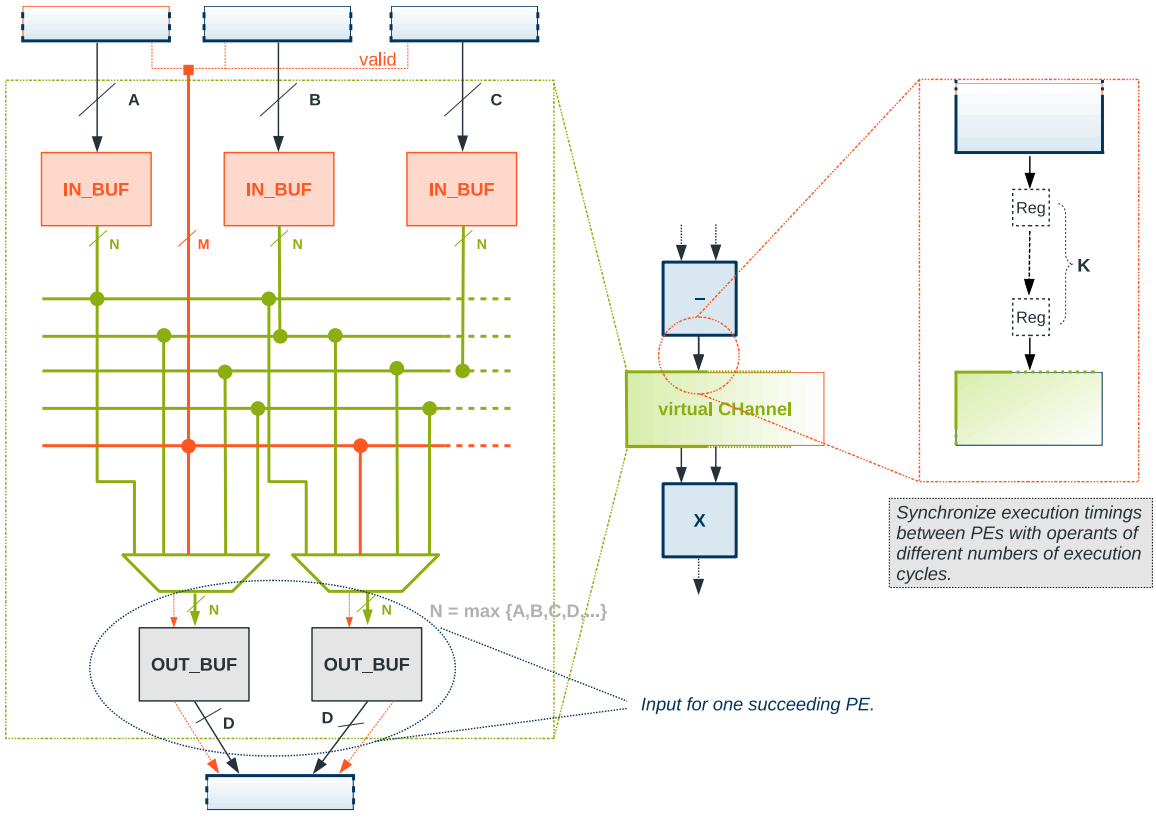

Abb. 30: Schematische Darstellung – Virtual Channel

Für eine hohe Flexibilität ist der Kanal als „*Full Crossbar Switch Matrix*“ gestaltet. Das bedeutet, dass jeder Eingang auf jeden Ausgang geschaltet werden kann. Die Anzahl der Eingänge eines Kanals und die Anzahl seiner Ausgänge ist abhängig von der Anzahl der Vorgänger- und Nachfolger-Prozesselemente eines Kanals. Die Anzahl der Eingänge entspricht der Anzahl der Vorgänger-PEs. Bei den Ausgängen ist die Anzahl doppelt so hoch wie die Anzahl der Nachfolger-PEs. Dies liegt am Design der Prozesselemente, welche jeweils zwei Eingangsgrößen erwarten. Es werden sowohl die Daten als auch die korrespondierenden `valid`-Signale jeweils gemeinsam auf den Ausgangspuffer bzw. einen Eingang eines Prozess Elements geführt.

Die `valid`-Signale sind nur ein Bit breit und werden deshalb gemeinsam in einem Puffer-Vektor gesammelt. Die Größe des Vektors korrespondiert mit der Anzahl der Dateneingänge. Als Index dient jeweils die Position des korrespondierenden Dateneingangs eines virtuellen Kanals. Die Daten werden einzeln in eigenen Puffern zwischengespeichert. Das hat den Grund, dass die Eingänge und Ausgänge jeweils für sich eine eigene Bitbreite für die Daten besitzen können. Die Bitbreite wird erneut bestimmt durch die Eigenschaften der am Kanal angeschlossenen Prozesselemente. Intern arbeitet der Kanal aber mit einer einheitlichen Bitbreite. Diese wird zur Designzeit des Kanals bestimmt und errechnet sich aus der höchsten Bitbreite aller am Kanal angeschlossenen Prozess Elemente:

$$N = \max\{A, B, C, D, \dots\} \mid A, B, C, D, \dots, N \in \mathbb{N}^{+} \tag{10}$$

Auch die Ausgangsdaten werden gepuffert, um die Bitbreite innerhalb des Kanals wieder an die Bitbreite für einen Eingang der folgenden PE anzupassen. In Abb. 30 ebenfalls angedeutet ist die Synchronisation für PE-Operationen mit unterschiedlicher Zyklenanzahl. Es können Register in die Pfade mit Operationen eingesetzt werden, die zur Umsetzung weniger Zyklen benötigen. In jedem zusätzlichen Takt einer komplexeren Operation wird das Datum jeweils von Register zu Register weiter gereicht. Diese Technik soll die Größe der PEs reduzieren – beispielsweise Vereinfachung einer Division durch Umsetzung in mehreren Taktzyklen – und das „*Pipelining*“ von Datenberechnungen ermöglichen. Unter *Pipelining* versteht man in diesem Fall, dass eine Ebene aus Prozesselementen die Berechnung neuer Daten beginnt, sobald die Daten an die folgende Ebene übergeben wurden. Im besten Fall kann somit in jedem Taktzyklus eine Berechnung pro Ebene erfolgen und wenn die Pipeline voll ist, kann sogar in jedem Taktzyklus ein Ergebnis ausgegeben werden.

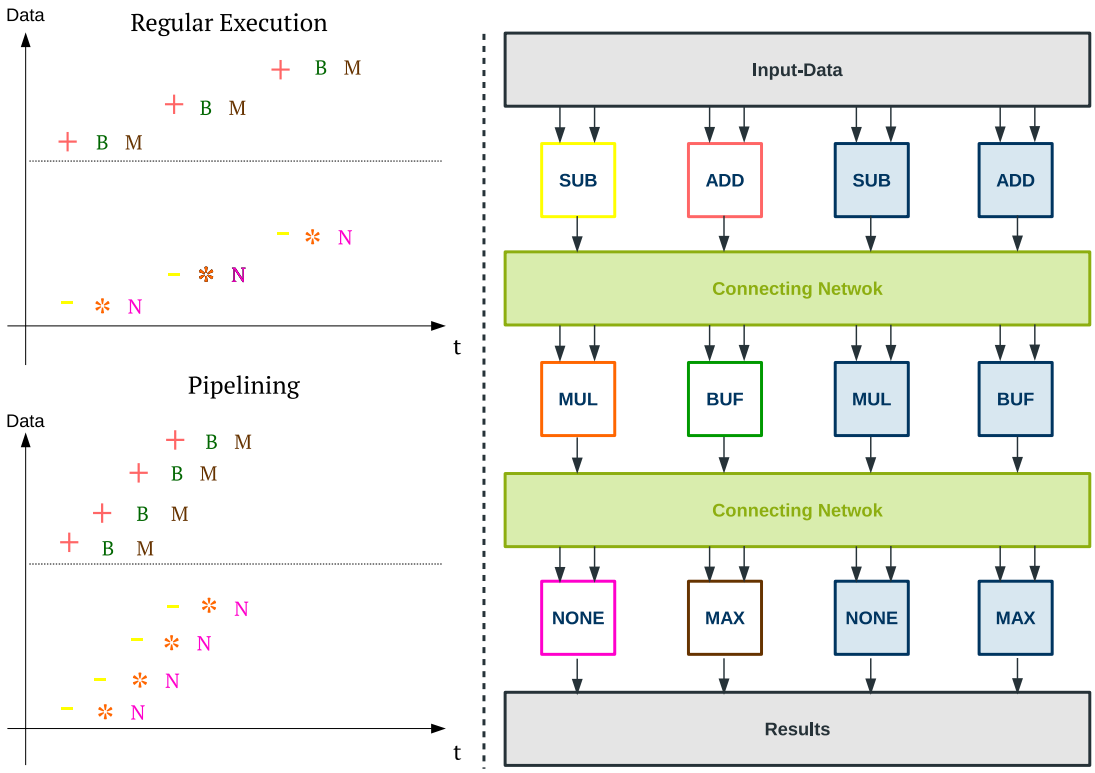

Abb. 31: Beispiel – Pipelining

Betrachtet man das Beispiel aus Abb. 25 und vergleicht eine reguläre Verarbeitung mit einer Verarbeitung mit *Pipelining*, so ergibt sich für diese für die ersten beiden Verarbeitungspfade eine Verteilung wie in Abb. 31. Da sich die betrachteten Datenpfade jeweils noch einmal wiederholen, wurden sie in der Darstellung nach Abb. 31 zur Übersichtlichkeit vernachlässigt. Neben dem CGRA als Blockschaltbild sind zwei Graphen angedeutet. Die Operationen in der Pipeline und parallel dazu im CGRA sind mit der korrespondierenden Farbe jeweils verknüpft. Auf der horizontalen Achse befindet sich die Zeit, auf der vertikalen Achse befindet sich jeweils ein Datenpaket, welches durch das CGRA verarbeitet werden soll. Bei der regulären Verarbeitung oben sieht man, dass die Zweige zwar parallel ihr Datum bearbeiten können, dass aber ein Datum durch das gesamte CGRA laufen muss, bevor die Verarbeitung eines kommenden Paketes angestoßen werden kann. Beim Pipelining wird, sobald die Operation in der aktuellen Ebene abgeschlossen ist, das nächste Datenpaket zur Verarbeitung in das CGRA gegeben. Dadurch verkürzt sich, wenn die Pipeline einmal voll ist, die Zeit, bis ein nächstes Ergebnis verfügbar ist, im besten Fall auf einen Taktzyklus. Für die Verarbeitung vieler Daten nach immer dem gleichen Verarbeitungsmuster – nach der Flynn'schen Klassifikation SIMD – wird erwartet, dass sich daraus Beschleunigungen ergeben. Bei zu vielen Kontextwechseln, ergo

einer zu häufigen Rekonfiguration, geht dieser Beschleunigungsvorteil verloren, da sich die Pipeline wieder erneut füllen muss.

Damit in jedem Taktzyklus ein neues Datum übernommen und der Verarbeitung zugeführt werden kann, muss ein Datum innerhalb des CGRAs auch in jedem Taktzyklus weitergereicht werden können. Für die zusätzlichen Register in den Datenpfaden des virtuellen Kanals ergibt sich daraus noch ein zu lösendes Problem bei diesem Konzept: Wie schaltet man die Register vorteilhaft zu- und wieder ab, wenn sich durch Rekonfiguration die Operation und damit die zeitlichen Bedingungen ändern? Auf entsprechende Verbesserungen des Grundkonzeptes wird ab Abschnitt 3 eingegangen. Abschließend ist auch für dieses Modul noch einmal die Entity eines virtuellen Kanals in Code 3 angegeben.

```
entity virtualChannel_4_2 is

  generic (
    chWidth     : positive := 16;   -- internal channel width for data transfer
    numberOfIns : positive := 4;    -- number of incoming connections into the
channel
    in0_bitwidth  : positive := 8;    -- bitwidth of incoming data connection
    in1_bitwidth  : positive := 8;    -- bitwidth of incoming data connection
    in2_bitwidth  : positive := 8;    -- bitwidth of incoming data connection
    in3_bitwidth  : positive := 8;    -- bitwidth of incoming data connection
    numberOfOuts  : positive := 2;    -- number of outgoing data connections
    out0_bitwidth : positive := 8;    -- bitwidth of outgoing data connection
    out1_bitwidth : positive := 8;    -- bitwidth of outgoing data connection
  );
  port (
    in0     : in  signed (in0_bitwidth - 1 downto 0);
    in1     : in  signed (in1_bitwidth - 1 downto 0);
    in2     : in  signed (in2_bitwidth - 1 downto 0);
    in3     : in  signed (in3_bitwidth - 1 downto 0);
    in4     : in  signed (in4_bitwidth - 1 downto 0);
    out0      : out signed (out0_bitwidth - 1 downto 0);
    out1      : out signed (out1_bitwidth - 1 downto 0);
    enables   : out std_logic_vector(2 * numberOfOuts - 1 downto 0);
    --PARAM
    Conf  : in std_logic_vector(7 downto 0);
    --PARAM
    rst   : in  std_logic;
    clk   : in  std_logic;
    valids  : in  std_logic_vector(numberOfIns - 1 downto 0)
  );
end entity virtualChannel_4_2;
```

Code 3: VHDL-Entity eines Virtual Channel

Auch dieses Modul ist sehr flexibel angelegt durch die vielen generischen Attribute, die für eine Kanalinstanziierung angegeben werden können. Bei dem Beispiel nach Code 3 handelt es sich um einen Kanal mit vier Eingängen und zwei Ausgängen. Die interne Datenbreite wird mittels der Variablen `chWidth` angegeben. Aus der gegebenen Anzahl der Eingänge (`numberOfIns`) und Ausgänge (`numberOfOuts`) wird jeweils die Größe der Vektoren bestimmt, welche die Synchronisationssignale für die Daten repräsentieren – am Eingang `valids` und am Ausgang `enables`. Über die übrigen Parameter kann für jeden Datenport die Bitbreite einzeln konfiguriert werden. Dies schafft ein Maximum an Flexibilität. Mit `rst` werden die internen Puffer zurückgesetzt. Die Konfiguration `conf`, welche die Multiplexer im Kanal bedient, ist wieder als Parameter für die TCON/TLUT Toolflow markiert um entsprechend optimiert werden zu können. Die Ergebnisse dieser Optimierung sind in Abschnitt 2.2 in Tab. 3 exemplarisch bereits aufgeführt.

2.3.3 Komponente Synchronization Unit

Das VCGRA kann von außen durch einen entsprechenden Port gestartet werden. Da es zunächst als passiver Beschleuniger konzipiert ist, welcher Daten aus einem Eingangsspeicherbereich verarbeitet und in einem Ergebnisspeicherbereich ablegt, benötigt es zur Synchronisation mit einem Prozessorsystem (PS) einen weiteren *Finish*-Port.

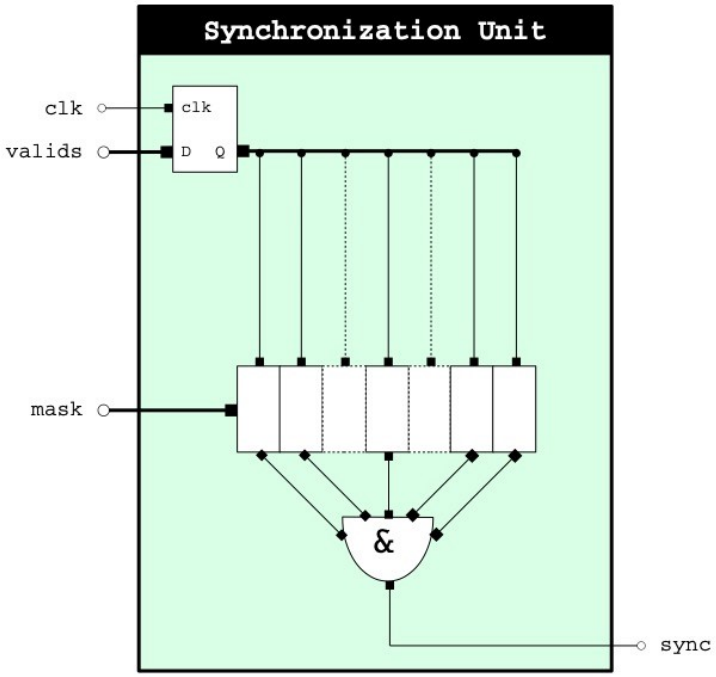

Abb. 32: Schematische Darstellung – Synchronization Unit

Zur Bedienung dieses Ports dient eine „*Synchronization Unit*“, welche im Folgenden näher erläutert wird. Das Prinzip ist in Abb. 32 schematisch dargestellt. Bei jeder steigenden Flanke werden die aktuellen `valid`-Signalzustände des letzten Levels von Prozesselementen in einem FlipFlop gespeichert. Im kommenden Zyklus werden die Zustände mit einer Maske (`mask`) zunächst verodert und anschließend dieses Zwischenergebnis verundet. Der Vektor der Bitmaske ist genauso groß wie die Anzahl der `valid`-Signale für das letzte Level von Prozesselementen. Eine PE, die in der letzten Reihe keine Verarbeitung in der aktuellen Konfiguration mehr durchführt, wird durch die Maske mit einer logischen Eins an seiner Position herausmaskiert. Denn werden Prozesselemente als `NONE` konfiguriert, haben sie keinen Einfluss auf ihre `valid`-Signale, weil sie keine keine Operation mit Daten ausführen. Wenn also alle Prozesselemente des letzten Levels ihre Bearbeitung beenden und ihr `valid`-Signal setzen, wird durch die Maske mit Platzhaltern die Verundung im Gesamtergebnis ebenfalls logisch Eins, wodurch ein Synchronisationssignal erzeugt wird. Dieses Signal kann von dem PS als Trigger verwendet werden, um die Ergebnisse abzuholen und eine neue Verarbeitung zu starten.

```
entity sync_ps is
  generic ( N : positive := 4);     -- set number of incoming valid signals
  port (
    valids  : in  std_logic_vector(0 to N - 1);
    clk   : in  std_logic;
    --PARAM
    mask  : in  std_logic_vector( N - 1 downto 0);
    --PARAM
    sync  : out   std_logic
  );
end entity sync_ps;
```

Code 4: VHDL-Entity einer Synchronization Unit

Auch dieses Modul des *Virtual Coarse Grained Reconfigurable Arrays* ist generisch geschrieben. Es kann als Instanz an den Aufbau des *Virtual Coarse Grained Reconfigurable Arrays* angepasst werden, sprich an die Anzahl der Prozesselemente in der letzten Ebene. Die Anpassung geschieht zur Designzeit des *Virtual Coarse Grained Reconfigurable Arrays*. Durch die Konfiguration mit der Maske kann nur auf die aktuelle Konfiguration einer PE im letzten Level reagiert werden. Die Anzahl der Eingänge (`valids`) ist während der Laufzeit nicht mehr konfigurierbar. Da sich die Konfiguration für die `mask` als weiterer Parameter für das VCGRA niederfrequent verändert wird diese für den TLUT/TCON Toolflow als Parameter markiert.

2.3.4 VCGRA Gesamtsystem

Das VCGRA setzt sich mindestens aus jeweils einer der genannten Komponenten zusammen. Die VHDL-*Entity* nach Code 5 entspricht dem VCGRA aus [24] mit fünf virtuellen Kanälen, 45 Prozesselementen und einer *Synchronization Unit*. In diesem Beispiel haben alle Prozesselement die gleiche Bitbreite ihrer Datenports. Aus diesem Grund sind zur Übersichtlichkeit die Ports des VCGRA für die Koeffizienten, die Dateneingänge sowie die Datenausgänge jeweils als *Array-Ports* definiert.

```
-- Type for the input data of a VCGRA
type DATA_ARRAY is array (natural range <>) of signed(15 downto 0);

entity vcgra is
  port (
      pixIn   : in  DATA_ARRAY (8 downto 0);
      coeffIn : in  DATA_ARRAY (8 downto 0);
      pixOut  : out DATA_ARRAY (8 downto 0);
      ready   : out std_logic;
      --PARAM
      confPE  : in  std_logic_vector(0 to 179);
      confCh  : in  std_logic_vector(0 to 449);
      confMa  : in  std_logic_vector(0 to 8);
      --PARAM
      clk     : in  std_logic;
      rst     : in  std_logic;
      start   : in  std_logic
  );
end entity vcgra;
```

Code 5: VHDL-Entity eines VCGRA

Die Konfigurationen für die Kanäle, die Prozesselemente und die Maske der *Synchronization Unit* sind jeweils als ein großer Bitstromport als Parameter für den TLUT/TCON Toolflow markiert. Die Bitströme werden beim Instanziieren des VCGRA jeweils auf ihre Komponenten aufgeteilt. Dabei gelten folgende Regeln:

- Beim Konfigurationsbitstrom für die Prozesselemente korrespondieren die Bits von links nach rechts mit den entsprechenden Prozesselementen von links oben nach rechts unten.
- Beim Konfigurationsbitstrom für die virtuellen Kanäle korrespondieren die Bits von links nach rechts mit den entsprechenden Kanälen von oben nach unten und innerhalb der Kanäle mit den Multiplexern von links nach rechts.
- Bei der Maske für die *Synchronization Unit* korrespondieren die Bits von links nach rechts mit den Prozesselementen des letzten Levels von links nach rechts.

In Abb. 33 ist diese Zuordnung noch einmal schematisch dargestellt. Für die Konfiguration der Prozesselemente ist gezeigt, wie die ersten Bits als Gruppe der ersten PE oben links in einem Hardwaredesign zugeordnet werden. Die folgende Bitgruppe erhält die nächste PE in der Reihe und so fort. Für den Kanal ist in einen solchen hineingezoomt. Aus dem Gesamtbitstrom für die Kanalkonfiguration werden die notwendigen Konfigurationsbits jeweils von links nach rechts einem Multiplexer zugeteilt.

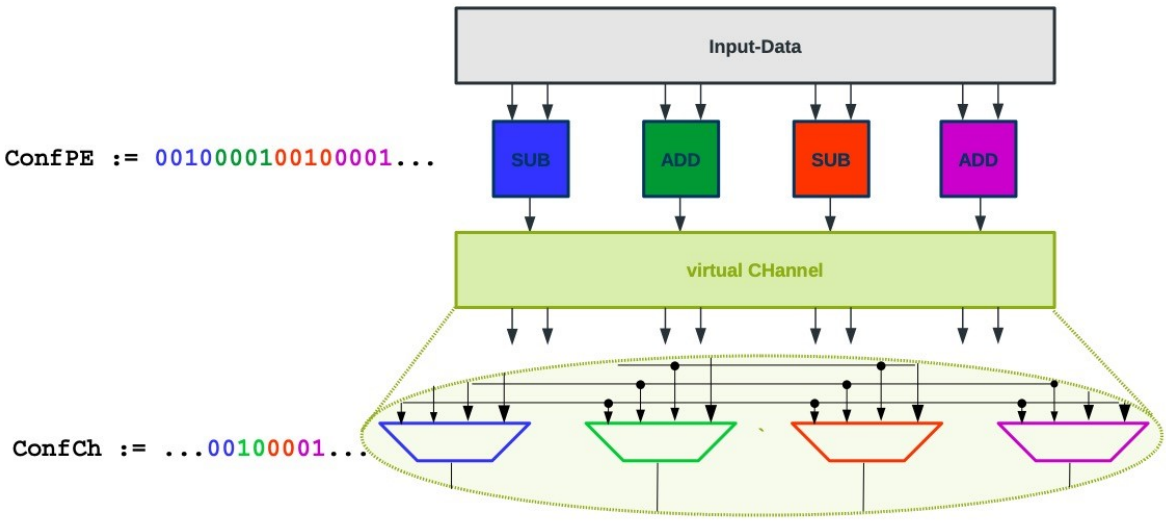

Abb. 33: Zuordnung der Bits aus einem Konfigurationsbitstrom zu den Komponenten eines VCGRAs

Die Länge der Bitströme für die Konfigurationen ist abhängig von der Anzahl der Verbindungsmöglichkeiten der virtuellen Kanäle und der Anzahl der Operationen der Prozesselemente sowie dem Aufbau des *Virtual Coarse Grained Reconfigurable Arrays* und der damit verbundenen Anzahl von Prozesselementen im letzten Level. Je optimierter ein Hardwaredesign für eine bestimmte Applikationsklasse während der Designphase erstellt ist, desto kleiner werden die Konfigurationen. Als Beispiel sei genannt, dass man die verfügbaren Operationen einer PE für eine Faltung beispielsweise auf `ADD`, `MUL` und `BUF` reduziert und das VCGRA so aufbaut, dass es mit dem Datenflussgraphen korrespondiert. Eine solche Gegenüberstellung ist in Abb. 34 und Abb. 35 aufgezeigt. Auf der linken Seite handelt es sich um ein universales *Grid* mit 9×5 Prozesselementen, auf der rechten Seite um ein für eine Faltung optimiertes Design. Ein solches Design entspricht einer Optimierung für eine bestimmte Algorithmenklasse, denn eine Faltung, beispielsweise auch für 2×2, wird ähnlich aussehen, nur nicht so viele Prozesselemente benutzen. Daher ist das Design abwärtskompatibel weiterverwendbar.

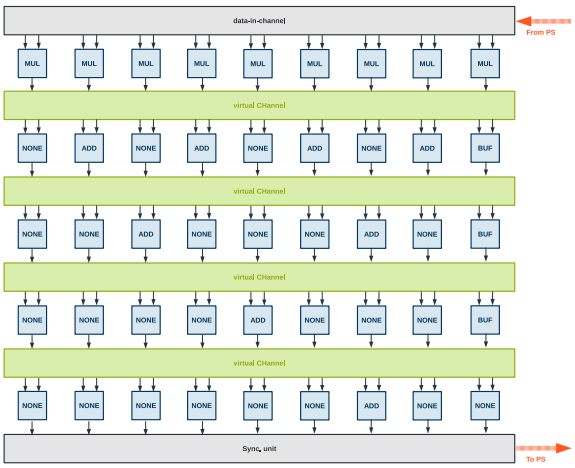

Abb. 34: VCGRA Raster 9 x 5

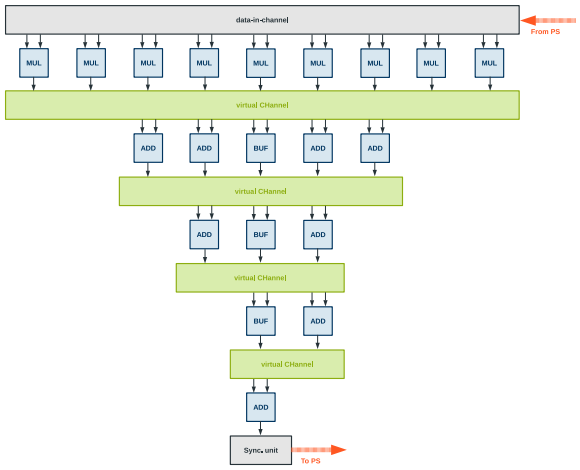

Abb. 35: VCGRA Raster optimiert für eine 3 x 3 Faltung

Schaut man sich die notwendige Bitstromlänge an, welche für je eine Konfiguration notwendig ist, erhält man folgende Rechnung:

$$
\begin{aligned}
pe_{bw} &= 4bit \\
ch_{bw} &= 5bit \\
mask_{bw} &= 9bit \\
bw_{ges} &= N_{pe} \times pe_{bw} + N_{ch} \times N_{mux} \times ch_{bw} + mask_{bw} \\
bw_{ges} &= 45 \times 4bit + 5 \times 18 \times 5bit + 9bit = 639bit
\end{aligned}
\tag{11}
$$

$$
\begin{aligned}
pe_{bw} &= 2bit \\
ch_{bw} &= 5bit \text{ abwärts bis } ch_{bw} = 1bit \\
mask_{bw} &= 1bit \\
bw_{ges} &= N_{PE} \times pe_{bw} + \sum_{i=0}^{4} N_{mux_i} \times ch_{bw_i} + mask_{bw} \\
bw_{ges} &= 20 \times 2bit + 169bit + 1bit = 210bit
\end{aligned}
\tag{12}
$$

- pe_{bw}: Bitbreite für die Konfiguration eines Prozesselements
- $ch_{bw_{[i]}}$: Bitbreite für die Konfiguration eines Multiplexers innerhalb eines virtuellen Kanals
- $mask_{bw}$: Bitbreite für die Maske der *Synchronization Unit*
- bw_{ges}: Notwendige Bitbreite für die gesamte Konfiguration des VCGRAs
- N_{pe}: Gesamtanzahl aller Prozesselemente
- N_{ch}: Gesamtanzahl aller Kanäle
- $N_{mux_{[i]}}$: Gesamtanzahl der Multiplexer eines Kanals

Das *Raster* auf der linken Seite ist homogen aufgebaut. Jede PE und jeder virtuelle Kanal verwendet die gleiche Anzahl an Konfigurationsbits. Auf der rechten Seite sind

die Operationen der Prozesselemente reduziert auf die Notwendigsten für diese Applikationsklasse. Die Summe der Konfigurationsbits für die virtuellen Kanäle ist hier abhängig von den jeweiligen Eigenschaften eines jeden virtuellen Kanals und durch die Summer über alle virtuellen Kanäle dargestellt. Das optimierte Design benötigt daher nur ca. 33% des generalisierten Designs an Bitstromlänge. Während der Entwurfsphase ist es daher sinnvoll, über den Zweck des VCGRA nachzudenken und ggf. ein optimiertes Design für eine bestimmte Applikationsklasse zu erarbeiten, um somit wertvolle Ressourcen und Rekonfigurationszeit zu sparen.

Die verbleibenden Ports an der VCGRA-*Entity* dienen dem Anschluss von Synchronisationsleitungen mit dem PS (`start`, `ready`), dem Takteingang (`clk`) und dem Rücksetzport (`rst`) für die Komponenten, welche einen solchen Port besitzen.

2.3.5 AXI4-Lite Wrapper für Kommunikation mit einem Prozessorsystem

Um die Ports des *Virtual Coarse Grained Reconfigurable Arrays* einfach mit dem Prozessorsystem zu verbinden, wurde das VCGRA in einen *AXI4-Lite* BUS-Wrapper eingebunden. Dieser Wrapper kommuniziert mittels des *AXI4*-Protokolls [29] (AMBA-BUS) von ARM mit dem PS und leitet dann die Daten an das VCGRA weiter.

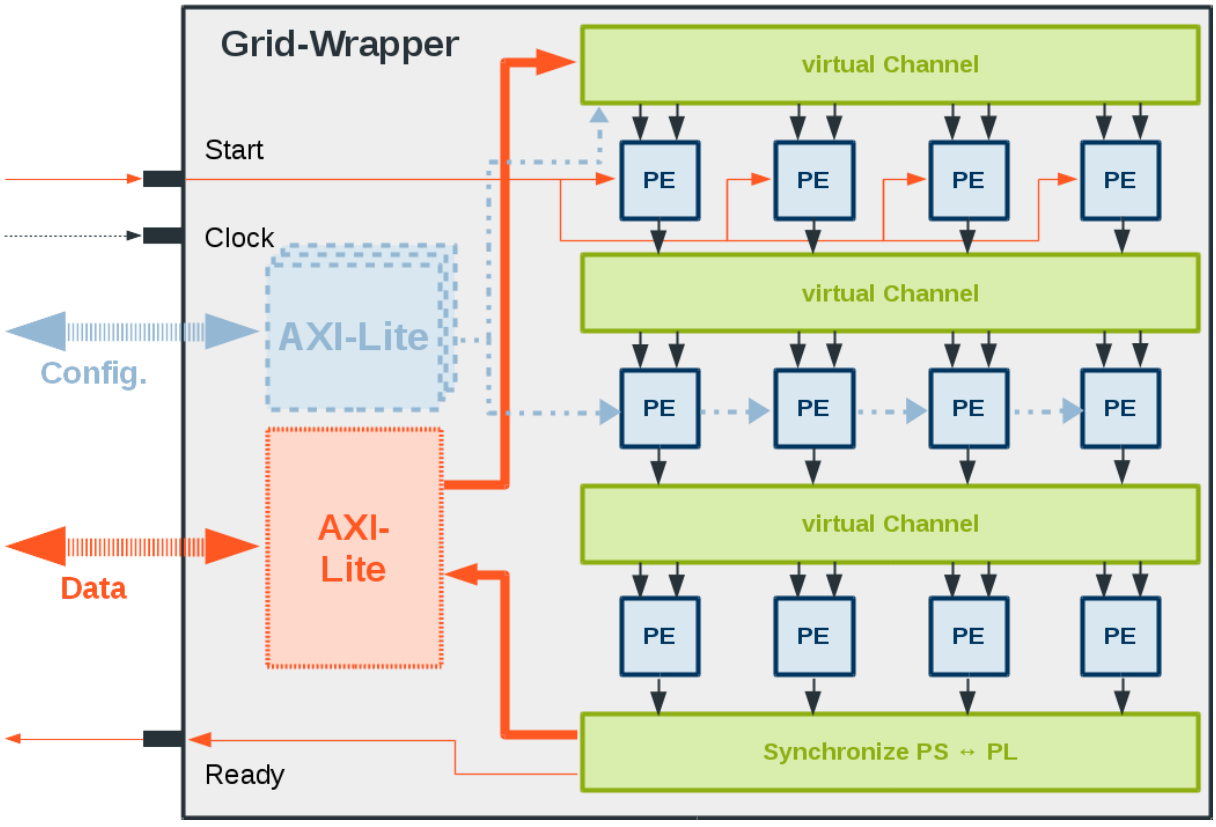

Abb. 36: Schematische Darstellung – VCGRA zusammen mit einem AXI4-Lite Wrapper

Der Aufbau des Wrappers ist schematisch in Abb. 36 angedeutet. Er besteht aus einer *AXI4-Lite* Schnittstelle für die drei Konfigurationsbitströme für die Prozesselemente,

die virtuellen Kanäle und der *Synchronization Unit* und einem *AXI4-Lite* Port für den Datenaustausch der Nutzdaten. Außerdem werden die *Flags* für die Synchronisation mit dem Prozessorsystem herausgeführt. Die Eigenschaften der *AXI4*-Schnittstellen sind abhängig von den Eigenschaften des *Virtual Coarse Grained Reconfigurable Arrays*. Je größer das VCGRA, desto mehr Puffer werden für das Speichern der Konfigurationsdaten benötigt. Das Gleiche gilt für die Nutzdaten: Je mehr Eingangs- und Ausgangsdaten, desto mehr Puffer werden im *AXI4*-Wrapper verlangt.

Die Schnittstellenvorlage von *Xilinx Vivado®* organisiert die Implementierung des *AXI4*-Protokolls. Bei der Konfiguration der Vorlage lassen sich Register definieren, welche vom Nutzer direkt durch Adressen im globalen Adressraum des Prozessorsystems erreichbar sind. Für das Mapping der Daten gelten folgende Festlegungen:

1. Jeder Datenport des VCGRA, egal ob Eingang oder Ausgang, wird auf ein eigenes Register gemapped. Daher ist die maximale Datenbreite eines Eingangs- oder Ausgangsdatums begrenzt auf 32-Bit. Dies entspricht der Bitbreite eines *Slave*-Registers, bereitgestellt von der *Vivado®*-Vorlage für die *AXI4-Lite* Schnittstelle.
2. Die *Slave*-Register in den *AXI4-Lite-Ports* für die Konfiguration werden aneinandergereiht, um dadurch die Gesamtlänge des Konfigurationsbitstromes zu erzeugen. Das erste Bit der Konfiguration befindet sich auf der Position des MSB (Bit 31; nullbasierte Nummerierung) des ersten *Slave*-Registers. Durch das Verknüpfen der Registerinhalte mit den folgenden *Slave*-Registern in steigender Reihenfolge wird der Gesamtkonfigurationsbitstrom hergestellt und an die Kanäle und Prozesselemente verteilt. Dabei werden bei einem unvollständigen letzten Register die Bits in Richtung LSB abgeschnitten, wenn diese für die Konfiguration nicht mehr benötigt werden. Die ausreichende Anzahl der *Slave*-Register muss beim Anlegen des *AXI4-Lite* Wrappers angegeben werden. Die Anzahl der Register errechnet sich nach:

$$\#_{Reg} = integer\left\{\frac{\left(\#_{ConfBitsTotal} + bw_{slvReg}\right) - 1}{bw_{slvReg}}\right\} \quad (13)$$

- $\#_{Reg}$: Anzahl der notwendigen *Slave*-Register des *AXI4-Lite* Wrappers
- $\#_{ConfBitsTotal}$: Gesamtanzahl der notwendigen Konfigurationsbits für die Konfiguration der virtuellen Kanäle, Prozesselemente und der *Synchronization Unit*
- bw_{slvReg}: Bitbreite eines *Slave*-Registers

Die Bitbreite der Register lässt sich bei diesem einfachen Kommunikationsbus nicht verändern. In der Regel werden von der Synthesesoftware ungenutzte Ressourcen wegoptimiert und beeinflussen daher den Flächenverbrauch nur gering. Darüber hinaus besetzt das *AXI4-Lite* Interface im Vergleich zum VCGRA nur wenig Platz auf dem Ziel-FPGA. Im Codeausschnitt Code 33 (Anhang Seite III) ist ein Reportausschnitt für ein VCGRA Design mit *AXI4-Lite* Wrapper angehängt. Aus Darstellungsgründen sind die Namen der Modulkomponenten abgekürzt, da es sonst zu ungewollten Zeilenumbrüchen kommt. Der Ausschnitt beschreibt den Platzverbrauch in der Anzahl von genutzten Logikzellen. Vom Gesamtverbrauch für die VCGRA-Implementierung werden lediglich 2,8% der Ressourcen für das *AXI4-Lite-Interface* benötigt. Es handelt sich dabei nur um den Verbrauch der letzten vier Einträge der Tabelle aus Code 33 bezogen auf den Gesamtverbrauch des *Virtual Coarse Grained Reconfigurable Array*s „`cgra_overlay_cgra_overlay_wrapper_0_0`“. Die *Flags* für die Synchronisation mit dem PS werden separat herausgeführt und in *Vivado®* über einen *AXI4-GPIO IP*-Block angesteuert. Damit lässt sich der Zustand der *Flags* vereinfacht vom Prozessorsystem über eine Adresse aus dem Adressraum auslesen oder setzen. Der Wrapper kann als *IP-Core* dann in einem *Vivado®*-Blockdesign verwendet und angesprochen werden.

2.3.6 Evaluation des IP-Cores

2.3.6.1 Applikation: Sobel Filter

Für die Evaluation des VCGRA Designs wird der Kernelcode für einen Sobel-Filter (Sobel-Operator) auf das VCGRA gemapped. Der Sobel-Filter gehört zu der Gruppe der Imagefilter für eine Kantenerkennung [30]. Er entspricht einer Filtermaske in der Größe 3×3 und wird mittels einer Faltung auf ein Eingangsbild angewendet. Ist der Filteraufsetzpunkt in der Mitte der Filtermaske definiert, ergibt sich folgende Berechnungsvorschrift:

$$G_x(x,y) = \sum_{i=1}^{3}\sum_{j=1}^{3} I(x + i - a, y + j - a) \cdot S_x(n + 1 - i, n + 1 - j)\Big|_{a=1} \quad (14)$$

- $G_x(x,y)$: Entspricht einem Ergebnispixel im korrespondierenden Gradientenbild zum Eingangsbild $I(x,y)$. Die Anwendungsrichtung beschreibt der Index an G_x. Die Anwendung kann beispielsweise senkrecht und waagerecht erfolgen. Die Filtermaske wird entsprechend der Anwendungsrichtung angepasst.
- $I(x,y)$: Dies ist das Eingangsbild für die Kantendetektion. Die Adressierung eines Bildpunktes im Eingangsbild erfolgt relativ zum Filteraufsetzpunkt a.
- $S_x(x,y)$: Dies ist die anzuwendende Filtermaske des Sobel-Operators. Die Indizierung der Filterparameter erfolgt absolut.

Die Filterkoeffizienten werden jeweils auf einen korrespondierenden Bildpunkt unterhalb der Filtermaske angewendet. Aus den gewichteten Bildpunkten wird dann eine Gesamtsumme gebildet und als Ergebnis auf dem korrespondierenden Gradientenbild gespeichert. Abb. 37 veranschaulicht nochmals die beschriebene Berechnung.

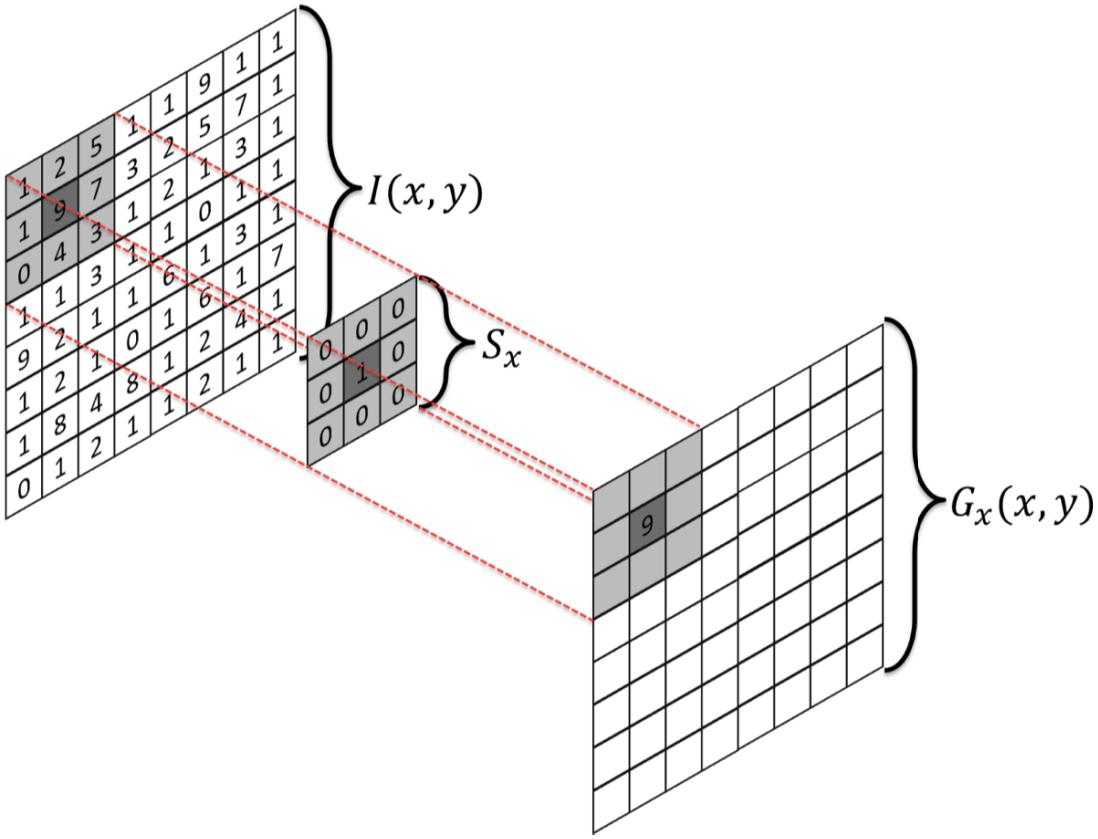

Abb. 37: Illustration einer Kantenerkennung, basierend auf einer Faltungsoperation

2.3.6.2 Kernelcode

Die Berechnungsformel nach obiger Gleichung kann in Form eines Algorithmus als geschachtelte Schleife über die Koordinaten des Filters und des Bildes ausgedrückt werden (siehe Code 6).

```
extern const int16_t sobel[3][3];
void convolute(const uint8_t* inputImage, uint8_t* gradientImage,
  const uint32_t sizeX, const uint32_t sizeY) {
  //temporary variables
  int16_t tempResult{0};
  uint32_t posX, posY;
  // run over all input pixels of input image
  for (uint32_t yIter = 1; yIter < sizeY - 1; ++yIter) {
    for(uint32_t xIter = 1; xIter < sizeX - 1; ++xIter) {
      tempResult = 0;
      posX = xIter - 1;
      posY = yIter - 1;
      //CODE
      for(uint8_t j = 0; j < 3; ++j)
        for(uint8_t i = 0; i < 3; ++i)
          tempResult +=
            ((int16_t) *(inputImage + (((posY + j) * sizeX) + (posX + i))))
            * sobel[2 - j][2 - i];
      //CODE
      if (tempResult < 0)
          tempResult *= -1;
      // save pixel in gradient image.
      *(gradientImage + ((yIter*sizeX)+xIter)) = (uint8_t)(tempResult/9);
    }
  }
  return;
}
```

Code 6: Kernel Code einer möglichen Implementierung des Faltungsalgorithmus

Die ersten beiden Schleifen laufen über das Ausgangsbild von oben links nach unten rechts. Das zweite Schleifenpaar läuft über alle Filterkoeffizienten und berechnet die gewichtete Summe. Am Ende des Algorithmus wird die gewichtete Summe als Betrag noch normiert. Der zu beschleunigende Anteil des Codes ist zwischen den Kommentaren `//CODE` eingerahmt. Die folgenden Betrachtungen beziehen sich auf dieses Codesegment.

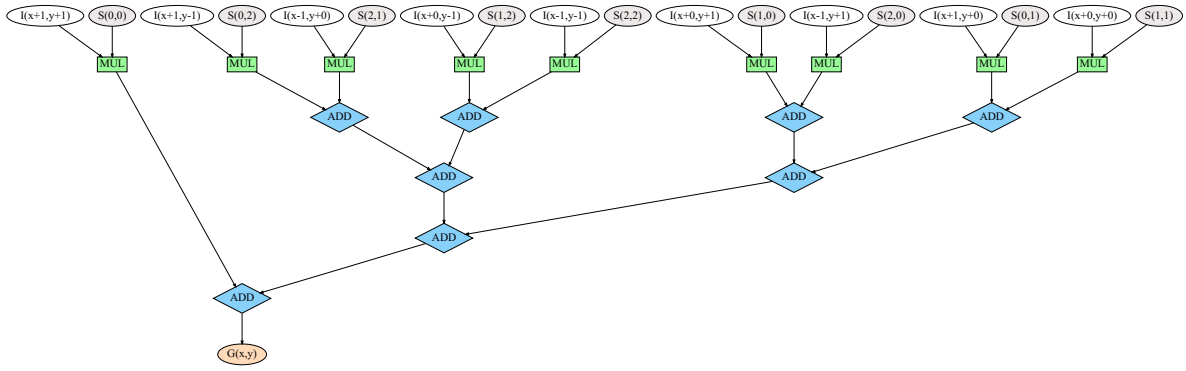

Abb. 38: Datenflussdiagramm zur Berechnung eines Ergebnispixels mit Hilfe einer Faltungsberechnung

Abb. 38 zeigt die Auflösung des Kernelcodes als Datenflussgraphen. Jedem Filterkoeffizienten *S* wird ein korrespondierender Pixelwert des Eingangsbildes *I* zugeordnet. Während die Filterkoeffizienten absolut adressiert werden, ist der korrespondierende Pixel aus dem Eingangsbild abhängig von der Position des Aufsetzpunktes des Filters auf diesem. Die Knoten stellen jeweils die Operationen eines Prozesselements dar. Die Normierung wurde aus dem Kernelcode herausgelassen, da eine Division in Hardware teuer ist und sehr viele Logikressourcen verbraucht. Jeder Knoten des Datenflussgraphen in Abb. 38 besitzt zwei Eingänge und einen Ausgang und passt daher zum Design des Prozesselements eines VCGRA.

2.3.6.3 Implementierung des VCGRA

Das Design der Implementierung ist stark anhängig von dessen Anforderungen. Zum einen sind die Eigenschaften der physikalischen Hardware (FPGA) ein limitierender Faktor. Wenn die notwendigen Ressourcen auf dem FPGA nicht zur Verfügung stehen, kann das entsprechende Design auch nicht umgesetzt werden. Zum anderen sind Designentscheidungen des Entwicklers ausschlaggebend:

- Laufen bspw. mehrere Applikationen parallel auf der Hardware in unterschiedlichen Regionen, sodass die Ressourcen eingeschränkt sind?
- Bevorzugt der Designer wenige Rekonfigurationen, sodass die *Cluster* auf dem VCGRA möglichst groß werden?
- Soll das Design sehr applikationsspezifisch sein oder handelt es sich um ein verallgemeinertes VCGRA Design mit zahlreicheren verfügbaren Operationen als die eigentliche Applikation benötigt?

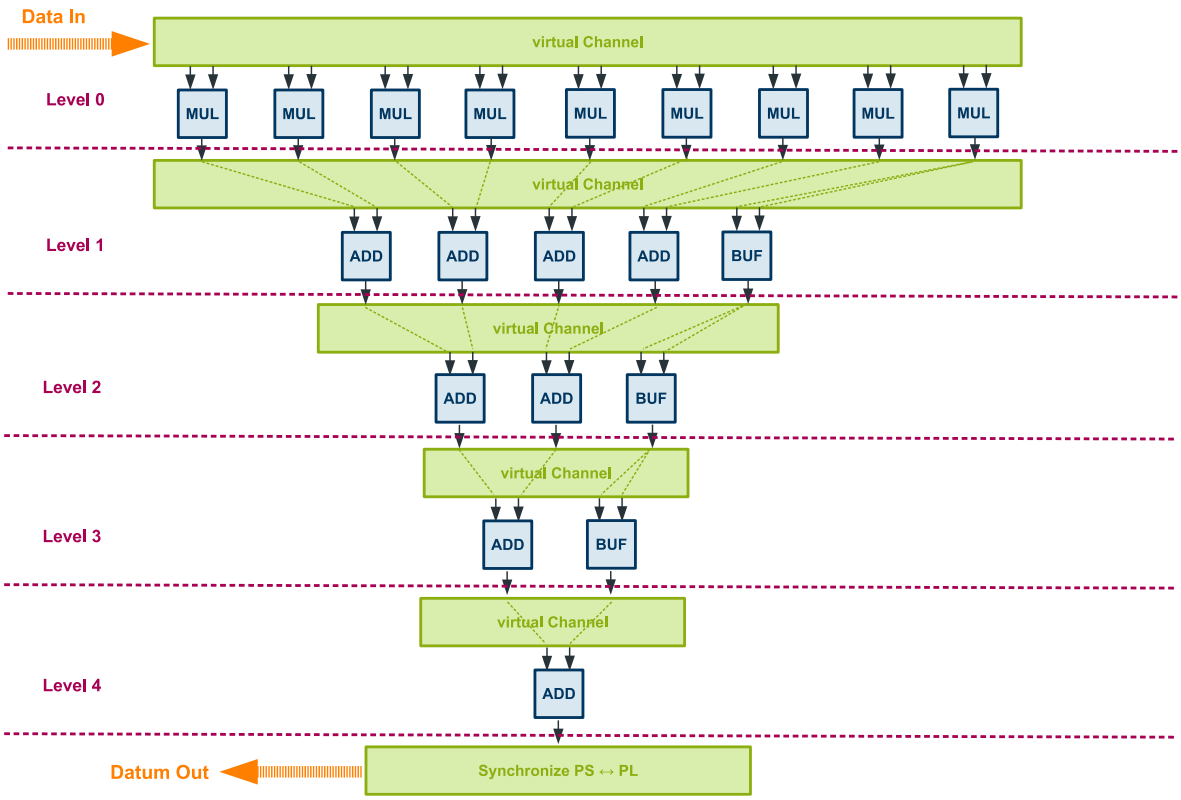

Abb. 39: Faltungsbeispiel – Cohesive Design

Mit der Annahme, es stünden genügend Ressourcen auf der Zielhardware für ein *Overlay* zur Verfügung, welches den gesamten Datenflussgraph des Kernelcodes umsetzen kann, entstünde ein Design ohne jegliche Rekonfiguration. Dieses Design wird im Folgenden als „*cohesive Design*“ bezeichnet. Der Datenflussgraph wird direkt durch die Zielarchitektur des *Virtual Coarse Grained Reconfigurable Arrays* abgebildet.

Es handelt sich bei diesem Design um eine stark spezialisierte Umsetzung der Applikation. Dadurch wird die Performanz hauptsächlich von der Bearbeitungsdauer der Prozesselemente dominiert. Die Kanäle beeinflussen die Performanz nur gering, da sie nur als Schaltnetz fungieren. Durch den Verzicht auf Rekonfiguration kann das Design schon während der Synthesephase durch eine statische Konfiguration vollständig optimiert werden. Deshalb entsteht ein Design mit der besten Performanz und dem geringsten Flächenverbrauch an Chipfläche. Aber es wird vollständig auf Flexibilität verzichtet. Der VCGRA fungiert ausschließlich als Abstraktionsebene zur physikalischen Zielhardware darunter, weshalb sich diese einfach austauschen lässt. Für das Beispiel entsteht ein VCGRA-*Shape* nach Abb. 39 mit 20 PEs, sechs virtuellen Kanälen und fünf Leveln. Aus dem Design ist erkennbar, dass mehr Prozesselemente im VCGRA notwendig sind als Knoten im Datenflussgraph vorhanden. Dies liegt an dem Design der Prozesselemente, welche jeweils auf zwei Eingangswerte festgelegt

sind. Daher muss ein gewichteter Pixel durch Pufferoperationen durch die jeweiligen Level geschleust werden, um am Ende das vollständige Ergebnis zu erhalten. Ein *Bypassing* von Zwischenwerten in ungenutzten virtuellen Kanälen ist in diesem Design nicht vorgesehen.

Als alternatives Design, das im Folgenden als „*clustered Design*“ bezeichnet ist, wird von Einschränkungen in der Chipfläche ausgegangen, sodass sich nur ein VCGRA implementieren lässt, welches aus drei Levels besteht mit jeweils vier Prozesselementen. Durch diese Einschränkungen ist es notwendig den Datenflussgraph in drei *Cluster* zu zerlegen, woraus sich maximal drei Rekonfigurationen pro Ausgabepixel ergeben. Der in *Cluster* zerlegte Datenflussgraph für die Faltung ist in Abb. 40 dargestellt.

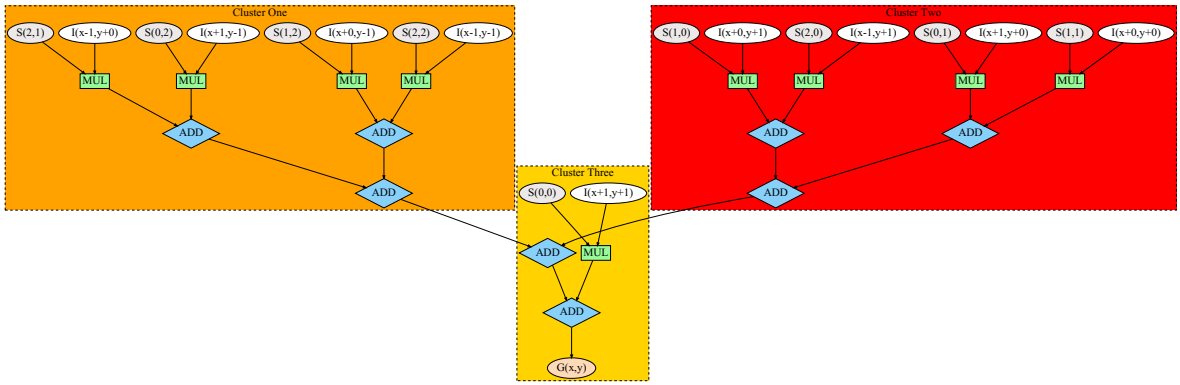

Abb. 40: Datenflussdiagramm – clustered *Design*

Zwei der drei *Cluster* weisen die gleiche Struktur auf, weshalb insgesamt nur zwei VCGRA-Konfigurationen benötigt werden. Die beiden Konfigurationen sind in Abb. 41 abgebildet: Links die Konfiguration für das rote und das orange *Cluster*, rechts die Konfiguration für das gelbe *Cluster* in der Mitte. Die den Datentransport zwischen dem Prozessorsystem und dem VCGRA anzeigenden Pfeile sind an den Farben ihrer korrespondierenden *Cluster* angelegt. Die temporären Zwischenergebnisse werden im Prozessorsystem gespeichert und im letzten Durchlauf wieder an das VCGRA übergeben, um das Gesamtergebnis zu berechnen. Prozesselemente die als `NONE` gekennzeichnet sind, haben keinen Einfluss auf die Berechnung der Ergebnisse.

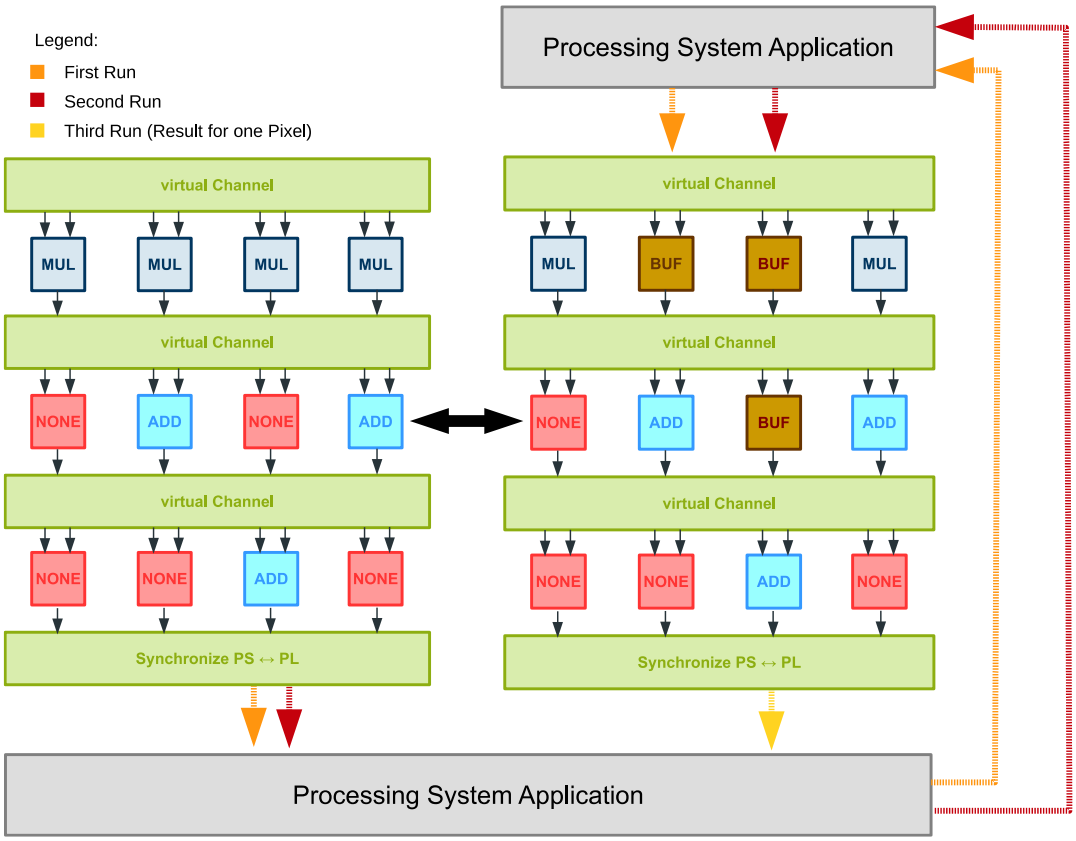

Abb. 41: Faltungsbeispiel – clustered *Design*

2.3.6.4 Evaluationsergebnisse

Beide Designs wurden in verschiedene Synthesewerkzeugen verarbeitet: *Vivado® 2017.1*, *ISE® 14.7* und der TLUT/TCON Toolflow [31]. *Vivado® 2017.1* ist der Nachfolger für *ISE® 14.7* und ist für die aktuellen FPGA Derivate von Xilinx *State-of-the-Art*. Es wird daher auch als Referenz für die Ergebnisse verwendet. ISE® 14.7 kam zum Einsatz, weil es in den wissenschaftlichen Veröffentlichungen nach [21], [2], [25] für die Evaluation des TLUT/TCON Toolflows auch verwendet wurde und somit als weitere Referenz gilt. Für den Vergleich wurden die einzelnen Komponenten wie das Prozesselement sowie die verschiedenen Implementierungen der virtuellen Kanäle als auch das gesamte VCGRA synthetisiert. Für die Verwendung der Xilinx Software *Vivado® 2017.1* und *ISE® 14.7* war die Angabe einer Zielarchitektur notwendig. Als eine solche kam für *Vivado® 2017.1* das *Zedboard* [32] zum Einsatz und für *ISE® 14.7* das *XUPV5-board* [33] mit einem *Virtex-V*. Beide Boards verwenden 6-LUT-Logikelemente. Auf Grund dieser Einstellung wurde auch für den TLUT/TCON Toolflow die LUT-Größe entsprechend auch sechs parametriert. Die Ergebnisse der Logiksynthesen sind in Tab. 4 aufgelistet.

Tab. 4: Logic Utilization für cohesive und clustered Design

Component	**Vivado® 2017.1**		**ISE® 14.7**		**TLUT**		**TLUT/TCON**		
	LUT	FF	LUT	FF	LUT	TLUT	LUT	TLUT	TCON
Logic Verwendung für das *cohesive* Design									
Proc. Elem.	61	27	74	27	67	14	72	8	12
vCh. 18/9	973	306	1171	288	151	703	144	0	2378
vCh. 9/5	321	162	361	152	72	180	72	0	640
vCh. 5/3	109	94	109	88	40	54	40	0	204
vCh. 3/2	37	60	37	56	24	36	24	0	63
vCh. 2/1	10	34	19	32	16	18	16	0	18
VCGRA	5117	1861	5941	1861	Nicht synthetisierbar				
VCGRA (static)	2528	1078	1341	1485	Nicht synthetisierbar				
Logic Verwendung für das *clustered* Design									
Proc. Elem.	61	27	74	27	67	14	72	8	12
vCh. 8/4	145	136	145	128	64	144	64	0	435
vCh. 4/4	73	104	73	36	32	72	32	0	224
VCGRA	2528	1076	2971	1078	2724	181	2801	181	1410

Bei der Implementierung der Prozesselemente liegen die Designs aus den kommerziellen Werkzeugen und die des TLUT/TCON Toolflows gleich auf. Am wenigsten Logikressourcen verbraucht das Design aus dem TLUT Toolflow mit 81 Logikelementen, gefolgt von *Vivado® 2017.1* mit 88 Logikelementen, Schlusslicht

bildet der *ISE® 14.7* mit 101 Logikelementen. Dies kann zum einen an der Zielarchitektur, welche für die Implementierung des *Virtual Coarse Grained Reconfigurable Arrays* architektonisch ungeeignet ist, als auch zum anderen an der Synthesesoftware selbst liegen, welche in der Synthese nicht die neuesten Optimierungsverfahren beinhaltet. Das Design der Prozesselemente enthält viele Puffer für die Eingangs- und Ausgangsdaten. Diese Puffer können nicht als geteilte Ressource verwendet werden. Die virtuellen Kanäle sind am ressourcenschonendsten im TLUT/TCON Toolflow implementiert, da sich die Multiplexer als Kernelemente der virtuellen Kanäle sehr gut als geteilte Ressource von diesem Toolflow optimieren lassen. Die LUTs für die virtuellen Kanäle im TLUT/TCON Toolflow werden ebenfalls nur als Puffer genutzt und lassen sich daher nicht teilen. Auch sie sind deshalb direkt auf Standard-LUTs gemapped. Die Ergebnisse für das VCGRA im *cohesive* Design lassen sich leider nicht vergleichen, da der Toolflow für TLUT/TCON Implementierungen mit einem Speicherbereichsfehler die Synthetisierung unterbrach. Dies scheint nicht ein Problem des Designs, sondern der Toolflow Software selbst zu sein. Eine Lösung der Urheber konnte leider auch nicht erreicht werden, da diese bereits ihre Dissertation beendet hatten und das Werkzeug seitdem nicht mehr gepflegt wird.

Durchschnittlich lassen sich durch die Verwendung von TLUTs und TCONs vor allem bei der Implementierung der Kanäle zwischen 9% und 75% der LUT-Ressourcen im Vergleich zu *Vivado® 2017.1* einsparen. Die Implementierung der Prozesselemente hingegen weist leider kein Optimierungspotential auf. TLUT und TLUT/TCON Design benötigen beim *clustered* Design jeweils gegenüber *Vivado® 2017.1* ca. 33% und gegenüber *ISE® 14.7* ca. 55% weniger Logikressourcen.

Interessant ist auch, dass das *clustered* Design in *Vivado® 2017.1* mit Rekonfigurationsmöglichkeit während der Laufzeit und das statisch-optimierte *cohesive* Design annähernd genauso viele Ressourcen benötigen. Das bedeutet im besten Fall, man gewinnt nicht nur einen Freiheitsgrad durch Rekonfiguration hinzu, bei dem man beispielsweise das *clustered* Design noch für andere Arbeitsaufgaben verwenden kann. Zusätzlich werden nicht mal mehr Ressourcen verwendet wie für das optimierte *cohesive* Design für genau eine spezielle Aufgabe. In diesem einen beobachteten Fall und diesem Synthesewerkzeug hat man einen Vorteil ohne Mehraufwand. Für das PP&R-Tool im *ISE® 14.2* zählt diese Aussage nicht mehr, da hier gegebenenfalls die Algorithmen oder der Aufbau der FPGA-Architektur den Aufwand für das *clustered* Design erhöhen. Die Anzahl der für die Untersuchung verwendeten Werkzeuge und Zielarchitekturen lässt hier keinen generellen Schluss

zu. Es kann sich auch in diesem einen Fall um Zufall handeln. Aus diesem Grund bietet es sich an in weiteren Untersuchungen auch Designtools von Altera (jetzt Intel) mit einzubeziehen.

Damit zeigt sich, dass durch die Rekonfigurationstechnologien enorm FPGA Chipfläche gespart werden kann. Es stellt sich aber die Frage welchen Einfluss diese Rekonfigurationen auf die Performanz der Applikation besitzen. Zur Untersuchung wird das Beispiel aus Abschnitt 2.3.6.2 auf eine echte Hardware synthetisiert. Leider konnte zu diesem Zweck nicht das optimierte Design aus dem TLUT/TCON Toolflow verwendet werden (siehe Abschnitt 2.2). Aus Gründen der Verfügbarkeit der Zielhardware wird das VHDL-Design des *Virtual Coarse Grained Reconfigurable Array* daher nur für das *Zedboard* synthetisiert.

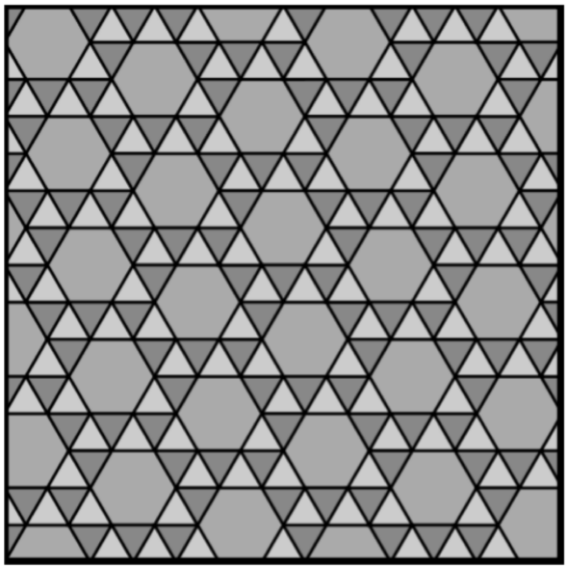

Abb. 42: Eingangsbild[5] für eine Kantendetektion

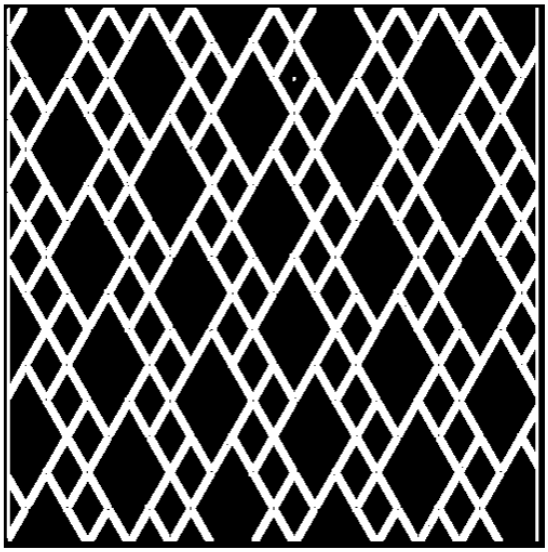

Abb. 43: Gradientenbild[5] für eine Kantendetektion

Zur Überprüfung der Funktionsfähigkeit des Algorithmus wurde das Muster aus Abb. 42 verwendet mit einer Größe von 500 × 500 Pixeln. Als Gradientenbild ergab sich das Muster nach Abb. 43 mit einer Größe von 498 × 498 Pixeln. Dies ist der Behandlung der Kanten des Eingabebildes geschuldet. Hier wird der Rand des Bildes weggelassen, indem beim Aufsetzen des Filters auf das Eingangsbild stets ein Abstand zu den Bildkanten eingehalten wird. Der Filter muss also in seiner Gesamtheit auf das Bild passen, ansonsten wird der Pixel unter dem Aufsetzpunkt nicht berechnet. Die Kanten zwischen den Mustern lassen sich klar erkennen durch den starken

[5] Bildformat PGM: https://en.wikipedia.org/wiki/Netpbm_format

Gradientenunterschied, aufgezeigt durch weiß und schwarz im Gradientenbild. Der starke Kontrast zwischen weiß und schwarz ist der Normierung geschadet, bei der alle Werte größer 0 auf 255 gesetzt werden.

Die Ergebnisse des Testlaufs sind in Abb. 44 visualisiert. Im Durchschnitt benötigt das Design 160µs, um einen Gradientenbildpixel zu berechnen und damit knapp 40s für die Berechnung des gesamten Gradientenbildes. Die Analyse der Ergebnisse zeigt, dass für die Rekonfiguration des *Virtual Coarse Grained Reconfigurable Arrays* nur jeweils ca. 6,5µs für die Konfiguration der Prozesselemente und der virtuellen Kanäle benötigt werden. Die Bereitstellung der Daten für das VCGRA über das *AXI4-Lite Interface* benötigt allerdings die Hälfte der Berechnungszeit. Diese Zeit verbringt das VCGRA derzeit nur mit Warten. Zusätzlich ist das *AXI4-Lite* Protokoll sehr langsam, da es keine Bursts oder Paketübertragungen unterstützt bzw. jedes Mal ein vollständiger Handshake zur Herstellung des Datenübertragungskanals und dem Senden der Daten erfolgen muss [29].

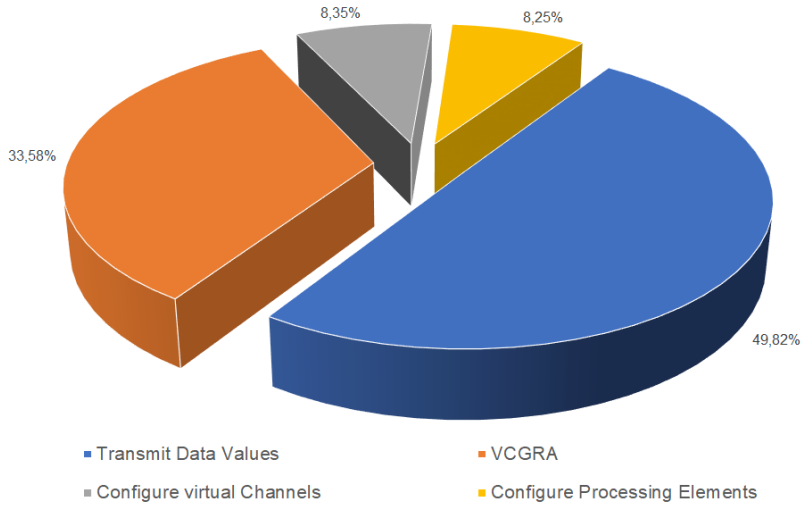

Abb. 44: Relative Ausführungszeiten eines VCGRA bei Verwendung des clustered Designs

Eine Schwachstelle in diesem Design ist derzeit das *Bottleneck der* Schnittstelle zwischen Prozessorsystem und VCGRA. Die Analyse des bestehenden VHDL-Codes hat aufgezeigt, dass eine weitere Einschränkung durch die Programmierung der Operationen einer PE selbst verursacht ist. Für arithmetische Operationen wie eine Division reduziert die Art der Implementierung in VHDL die Taktfrequenz des Gesamtsystems und macht eine PE unverhältnismäßig groß.

2.4 Zusammenfassung

Für die Entwicklung einer applikationsspezifischen, aber dennoch rekonfigurierbaren Architektur wurde zunächst der Algorithmusbegriff geklärt und die Eigenschaften von Algorithmen untersucht. Es wurde festgestellt, dass es in Algorithmen immer wieder Codesegmente gibt, deren Beschleunigung lohnt, da sich die Applikation im Verhältnis zur gesamten Ausführungszeit sehr lange in diesem „Kernelcode“ aufhält.

Anhand der gewonnenen Erkenntnisse stellt sich die Frage nach der optimalen Granularität, welche nicht vollständig beantwortet werden konnte. Vielmehr muss im besten Fall die Architektur in ihrer Bandbreite an die Applikation angepasst werden, um die besten Ergebnisse zu erhalten. Aus diesem Grund ist sich für die Entwicklung eines *FPGA-Overlays* als Architektur entschieden worden. Dieses *Overlay* abstrahiert die physikalische Architektur des Field Programmable Gate Arrays. Dadurch kann es flexibel auf unterschiedlichsten Zielplattformen implementiert und während der Designphase an die Eigenschaften der Applikation angepasst werden.

Für die Rekonfigurationen wurde die Architektur an die Designrichtlinien für DCS und dem TLUT/TCON Toolflow angepasst. Durch die Verwendung dieser Techniken soll das Design möglichst einfach und „in Echtzeit“ rekonfigurierbar werden. Das Design der Architektur gliedert sich also nun in zwei Prozesse, zunächst die Erstellung einer virtuellen Zielhardware, genannt Virtu*al Coarse Grained Reconfigurable Array*, und zum anderen einen dazugehörigen parametrisierten Konfigurationsbitstrom aus dem TLUT/TCON Toolflow. Als Konfigurationseinheiten wurden die Prozesselemente, die Verbindungseinheiten (virtuelle Kanäle) und die *Synchronization Unit* des *Virtual Coarse Grained Reconfigurable Arrays* festgelegt. Die Applikation selbst wird in einen Datenflussgraphen umgewandelt, um die Abhängigkeiten der Daten und die notwendigen Operationen zu evaluieren. Das vorgestellte Konzept der Architektur unterstützt arithmetische Operationen wie `ADD`, `SUB`, `MUL`, Integer-`DIV`, `GRE`, `EQL` und `BUF`. Anhand des Datenflussgraphen werden die Bool’schen Gleichungen aus den parametrisierten Bitströmen der Architektur gelöst um eine spezialisierte Variante des *Virtual Coarse Grained Reconfigurable Arrays* auf der Architektur zur erstellen (siehe Abschnitt 2.2).

Die Verwendung des TLUT/TCON Toolflows zeigt große Erfolge bei der Implementierung der virtuellen Kanäle, hier können bis zu 75% der Ressourcen im Vergleich zu aktuellen Synthesewerkzeugen kommerzieller FPGA Hersteller eingespart werden (siehe Abschnitt 2.3.6.4). Leider ist es dem Toolflow nicht möglich lauffähige Varianten des Designs auf echter Hardware für Laufzeitanalysen zu

generieren. Doch durch die Verwendung von VHDL ohne herstellerspezifische Erweiterungen einer virtuellen Overlayarchitektur kann das Design mit einem Synthesewerkzeug auf einen FPGA, bspw. *Zedboard,* programmiert und eine Laufzeitanalyse durchgeführt werden. Die Rekonfiguration der Architektur dauert nur wenige Microsekunden und nimmt weniger als 10% der Gesamtzeit zur Berechnung eines Ausgabewertes ein. Das Evaluationsbeispiel hat damit die generelle Funktionsweise des Konzeptes bewiesen allerdings mit folgenden Einschränkungen:

- Das Hardwaredesign weist in diesem Stadium Unschönheiten in der VHDL Programmierun auf, wodurch sich die Arbeitsfrequenz des Designs auf maximal 2MHz beschränkt.
- Der TLUT/TCON Toolflow kann nicht angewendet werden, da sich die Ergebnisse nicht auf eine echte FPGA-Architektur programmieren lassen.
- Das Design weist Schwächen auf bei der Bereitstellung der Daten. Das *AXI4-Lite Interface* zwischen Prozessor und Hardwarebeschleuniger ist ein *Bottleneck*, welches die Ausführungsgeschwindigkeit stark reduziert.
- Die Prozesselemente verbringen einen Großteil ihrer Zeit mit Warten auf Daten. Durch fehlendes *Pipelining* ist die Verarbeitungsgeschwindigkeit noch stark reduziert.

In den folgenden Abschnitten werden die Probleme der Architektur weiter analysiert und Verbesserungsmöglichkeiten herausgearbeitet.

3 Spezialisierung und Optimierung

3.1 Pre-Fetcher und Datenpuffer

Während der Elaboration der grundlegenden Architektur in Abschnitt 2.3 wurde bereits festgestellt, dass ein Großteil der Laufzeit für die Bewältigung von Datentransferaufgaben verwendet wird. Der Umfang oder die Masse der Daten lässt sich für eine Applikation nur eingeschränkt beeinflussen. Es können Datentypen verwendet werden, welche eine geringere Größe aufweisen wie beispielsweise `Float` statt `Double` Typen oder Integer-Werte mit geringerer Bitanzahl wie `Short` anstatt `Int` (`Word`). Dies beeinflusst aber die Genauigkeit der Berechnungen und schränkt die Einsatzfähigkeit der Architektur ein. Andererseits kann versucht werden, durch die Erhöhung der Übertragungsgeschwindigkeit der Daten die Geschwindigkeit der Architektur zur erhöhen. Durch diese Maßnahme wird zwar eine Optimierung der Gesamtverarbeitungsdauer erreicht, das VCGRA ist aber dennoch nicht optimal genutzt, da es weiterhin Zeit mit Warten verbringt, um alle Konfigurationen und Daten an seinen Eingängen zu erhalten. Des Weiteren steigt durch die Erhöhung der Taktfrequenz die Verlustleistung des integrierten Schaltkreises linear durch Zunahme der Umladungen der enthaltenen Kapazitäten des Schaltkreises.

Aus Sicht des Verfassers ist es sinnvoller, die Verarbeitungszeit des *Virtual Coarse Grained Reconfigurable Arrays* parallel zur Datenübertragung demnächst zu verarbeitender Daten und Konfigurationen zu nutzen. Dieser als *„Pre-Fetching"* bezeichnete Vorgang würde die Zeit aktiven Wartens extrem einschränken und optimieren. Während eines Verarbeitungszyklus[6] beziehungsweise der Berechnung neuer Daten können bereits neue Daten in vorgesehene Speicherbereiche, so genannte *„Pre-Fetcher"*, geladen werden. Sobald die Architektur fertig ist, kann im besten Fall, wenn während dieser Zeit alle notwendigen Daten zur nächsten Berechnung übertragen werden konnten, im Anschluss direkt die Verarbeitung fortgesetzt werden. Solche *Pre-Fetcher* sind für alle Eingänge des *Virtual Coarse Grained Reconfigurable Arrays* sinnvoll und anwendbar, bei denen es sich nicht um einfache Steuersignale mit 1Bit Breite handelt wie beispielsweise die Konfigurationen der Prozesselemente und virtuellen Kanäle als auch die Eingangsdaten am Kopf des VCGRA. Für den Ausgang des VCGRA bietet sich ein Datenpuffer an. Zum einen

[6] Als Verarbeitungszyklus bezeichnet der Verfasser die Zeit zwischen dem Starten der Berechnungen im VCGRA und dem Aktivieren des *Ready*-Flags zur Synchronisierung zwischen VCGRA und Prozessorsystem.

entspannt es die Abholung der verarbeiteten Daten durch das Prozessorsystem, denn auch dies kann nun parallel während der Datenverarbeitung des VCGRA erfolgen, sobald eine neue Bearbeitung angestoßen wurde. Zum anderen kann dieser Puffer als Zwischenspeicher genutzt werden, um temporäre Daten als *Feedback* wieder zur Verarbeitung an das VCGRA zurückzugeben. Es verhindert so das unnötige Kopieren der Daten vom VCGRA zum PS und zurück.

Die Größe einer *Pre-Fetcher* Zeile ist abhängig vom Design des VCGRA. So ergibt sich die Anzahl der Elemente einer Zeile für den Input-Data-*Pre-Fetcher* aus der Anzahl der Eingänge des *Virtual Coarse Grained Reconfigurable Arrays*. Gleiches gilt für den Ausgangsdatenpuffer. Die Größe einer Zeile eines Konfigurations-*Pre-Fetcher* ergibt sich aus der Größe des Konfigurationsbitstromes für virtuelle Kanäle und Prozesselemente. Erweitert man die Funktionalität der *Pre-Fetcher* noch um weitere Intelligenz ist es vorstellbar, diese als Daten- bzw. Konfigurations-*Caches* zu verwenden. Es würden dann zwischen Prozessorsystem und VCGRA nur noch solche Datenübertragungen notwendig, die neue Daten beinhalten. Bereits vorhandene Daten liegen lokal am VCGRA und können durch Umschaltung bzw. Auswahl einer entsprechenden Cache-Zeile direkt vom VCGRA verwendet werden. Dies sparte weitere Übertragungszeiten und Energie.

3.2 Shared Scratchpad Memory und Memory Management Unit

Um den Datenaustausch zwischen einem VCGRA und einem Prozessorsystem weiter zu beschleunigen bzw. der Architektur mehr Speicher für Zwischenergebnisse, Daten und Konfigurationen zur Verfügung zu stellen wird die Architektur um einen dedizierten Speicherbereich erweitert, der sowohl als „Notizblock“ (*Scratchpad*) für temporäre Ergebnisse des *Virtual Coarse Grained Reconfigurable Arrays* als auch als „*Shared Memory*“ für das Prozessorsystem und VCGRA fungiert. Die Idee dabei ist, die Architektur unabhängiger vom PS zu gestalten, also die Kontrolle des Prozessorsystems so weit wie möglich einzuschränken. Dem PS soll damit erleichtert werden, freiwerdende Rechenzeit für andere Aufgaben (engl. *Tasks*) zu nutzen. Die Anbindung von Seiten des PS kann beispielsweise über einen DMA stattfinden, gehören soll der Speicherbereich aber dem VCGRA, damit dieses möglichst verlustfrei an Zeit und Energie auf den Speicherbereich zugreifen kann. Als Nutzungsszenario stellt sich der Autor folgende Arbeitsweise zwischen Prozessorsystem und VCGRA so vor, dass das PS die Nutzdaten und die Konfigurationen im Speicherbereich auf bestimmten Adressbereichen über DMA Zugriff ablegt.

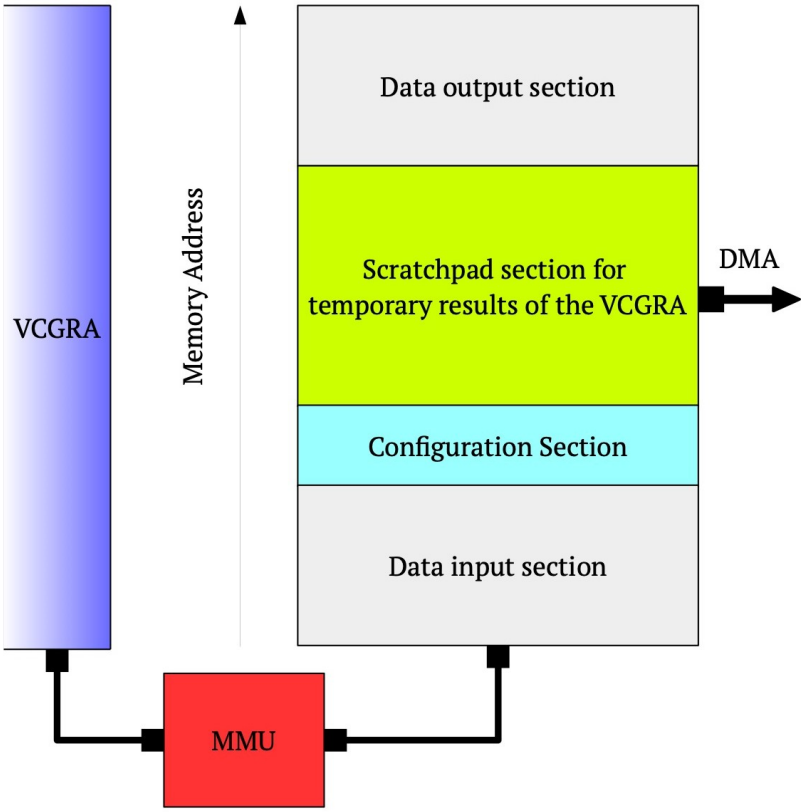

Abb. 45: Speicherdesign vom dedizierten Speicherbereichen eines VCGRA

Abb. 45 zeigt beispielhaft den Aufbau eines solchen Speicherbereiches. Es gibt gleich große Speicherbereiche für die Eingangsdaten und die Ausgangsdaten aus Sicht des VCGRA. Im Input-Datenbereich liegen die Daten vom Prozessorsystem, welche vom VCGRA verarbeitet werden sollen, im Output-Datenbereich können die Ergebnisse der Verarbeitung durch das PS abgeholt werden. Der Datenbereich für die Speicherung von Konfigurationen grenzt direkt an den Input-Daten-Speicherbereich an und kann kleiner sein als dieser. Es wird davon ausgegangen, dass sich die Anzahl der Konfigurationen und deren Größe im Vergleich zu den Eingangsdaten und den Ausgangsdaten beschränkt bzw. kleiner ist als diese. Des Weiteren besitzt der dedizierte Speicherbereich nach Abb. 45 ein Segment, welches nur für die VCGRA Architektur verwendbar ist und für die Ablage temporärer Daten verwendet wird.

Nach der Ablage der Eingangsdaten und der Konfigurationen wird das VCGRA gestartet, nachdem in einer Initialisierungsphase die Konfigurationen in die *Pre-Fetcher* geladen wurden und eine entsprechende Konfiguration am System anliegt. Die Daten werden verarbeitet und können beispielsweise am Ende der Verarbeitung vom Prozessorsystem im Ausgangsdatensegment abgerufen werden.

Der Zugriff auf die Datensegmente wird durch eine *Memory Management Unit* (MMU) geregelt. Sie ist in der Lage Nutzdaten elementweise/blockweise und Konfigurationen blockweise zwischen den *Pre-Fetcher* und dem Datenpuffer auszutauschen. Elementweise bedeutet, dass in jede Zeile des Eingangsdaten *Pre-Fetcher* oder des Ausgangsdatenpuffers Elemente geschrieben oder geladen werden können. Blockweise bedeutet, dass jeweils eine ganze Zeile eines *Pre-Fetcher* oder des Puffers ausgetauscht wird. Der Datenaustausch zwischen dem *Pre-Fetcher* und dem Speicherbereich wird dabei vollständig von der MMU geregelt. Der Datenaustausch findet mit *Write*- und *Acknowledge*-Signalen statt. Damit wird sichergestellt, dass der Datenaustausch vollständig beendet wurde, bevor der Datentransfer beendet oder mit dem nächsten Transfer eines Datums fortgesetzt wird.

3.3 Management Unit (Central Control Unit)

Durch die zuvor genannten Maßnahmen aus Abschnitt 3.1 und 3.2 soll eine Beschleunigung durch Parallelisierung von Verarbeitung und Datentransfer erreicht werden. Dennoch ist für die Steuerung des Gesamtablaufes noch viel Zugriff durch das Prozessorsystem notwendig. Neben dem Ansteuern des *Virtual Coarse Grained Reconfigurable Arrays*, um die Verarbeitung zu starten müssen derzeit zusätzlich alle Steuersignale für die *Memory Management Unit* durch das PS bereitgestellt werden. Dies entlastet das Prozessorsystem nur unwesentlich und schafft damit keinen Freiraum, um dort andere Aufgaben parallel zu verarbeiten.

Um das Prozessorsystem zu entlasten, soll das *Virtual Coarse Grained Reconfigurable Array* selbstständiger Arbeiten. Die Synchronisation zwischen VCGRA und Prozessorsystem findet nun über eine *Central Control Unit* (CCU) statt. Damit wird das PS nur noch unterbrochen, wenn die Daten vollständig vom Beschleuniger bearbeitet sind und nicht nach jedem Teilergebnis. Die „*Central Control Unit*“ erhält somit die weitgehende Kontrolle über alle Teilelemente – beispielsweise MMU, *Pre-Fetcher*, Puffer oder das VCGRA – der Gesamtarchitektur.

Die Kontrolle der CCU erfolgt durch Maschinenbefehle auf die im Folgenden in den Abschnitten 3.3.1ff noch genauer eingegangen wird. Die Instruktionen der Maschinenbefehle laden beispielsweise Daten in die *Pre-Fetcher* oder steuern den Start des *Virtual Coarse Grained Reconfigurable Arrays* und die Synchronisierung mit dem Prozessorsystem. Mit dieser Maßnahme wird eine vollständige Parallelisierung zwischen Prozessorsystem und VCGRA erreicht, denn nach dem Start der Verarbeitung durchläuft die *Central Control Unit* alle Maschinenbefehle und gibt Rückmeldung, wenn ein Fehler aufgetreten oder die Verarbeitung der Daten

vollständig abgeschlossen ist. Das Laden der Eingangsdaten, das Laden der Konfigurationen und deren Umschaltung im Programmablauf erfolgt vollständig autark durch die Gesamtarchitektur. Damit ist das Prozessorsystem vollständig entlastet und kann andere Tasks bearbeiten oder beispielsweise neue Daten für das VCGRA vorbereiten. Diese Art der Parallelisierung kann das *Bottleneck* „Datentransfer“ nahezu vollständig kompensieren, da das VCGRA theoretisch nur vor dem ersten Durchlauf auf Daten warten muss. Anschließend sind keine weiteren aufwendigen Kommunikationen zwischen Prozessorsystem und Beschleuniger mehr notwendig.

3.3.1 Aufbau und Funktion des Maschinencodes

Tab. 5: Maschinencode-Binärformat

31 30 … 17 16	13	12 … 7 6	5 … 1 0
Address	Cache Line	Place in Cache Line	Command Identifier

Die Architektur besitzt ein festes Maschinencode-Binärformat. Ein Befehlswort im Maschinencode umfasst immer 32 Bit. In diesen 32 Bit sind vier Informationen kodiert:

a) [16Bit] Startadresse eines Datums im *Shared Memory* zwischen PS und Gesamtarchitektur
b) [3Bit] *Line* eines *Pre-Fetcher* oder eines Puffers
c) [7Bit] *Place* innerhalb einer Line eines *Pre-Fetcher* oder eines Puffers
d) [6Bit] *Command-ID* welche die Funktion der Architektur steuert

Für die Adresse werden 16 Bit im Binärformat reserviert. Damit lassen sich 65536 Adressen unterscheiden. Abhängig von der Speicherstruktur – wieviel Byte Speicher können per Adresse angesprochen werden – lässt sich die Größe des *Shared Memory* einstellen. Bei der Verwendung von Adressierung einzelner Bytes zum Beispiel lassen sich über den Adressraum 64KiB[7] Speicher ansprechen, bei Verwendung von Wörtern mit 32 Bit Breite können schon 265KiB Speicher verwenden werden.

Es lassen sich bis zu acht *Line*s eines *Pre-Fetcher* oder eines Puffers unterscheiden. Da diese Einheiten nicht als *Caches* fungieren, sondern nur als temporäre Speicher

[7] KiB: Kibibyte =1024 Byte https://en.wikipedia.org/wiki/Kibibyte.

dienen, um die Verarbeitungszeit des *Virtual Coarse Grained Reconfigurable Arrays* nicht mit Warten auf Nutzdaten zu verschwenden, ist diese Anzahl an *Line*s zunächst ausreichend. Für viele Applikationen sind in Theorie bereits zwei *Line*s genügend: Eine Zeile enthält die gerade verarbeitenden Daten und die andere wird währenddessen mit neuen Daten gefüllt. Anschließend erfolgt der Wechsel und so fort.

Innerhalb einer *Line* lassen sich 0...126 = 127 *Place*s unterscheiden. Da *Place*s Eingangs- bzw. Ausgangswerte des Virtua*l Coarse Grained Reconfigurable Arrays* entsprechen, können somit 127 Ein- und Ausgänge an einem VCGRA angesprochen werden. Dies ist für die meisten Applikationen, selbst für bildverarbeitende Algorithmen, ausreichend. Ist als *Place* 127 angegeben wird dies als Übernahme einer gesamten *Line* interpretiert und es werden ab der Startadresse alle Werte hintereinander vom *Shared Memory* zum Ziel-*Pre-Fetcher* oder vom Datenpuffer zum *Shared Memory* übertragen.

Über die *Command-ID* können 64 Befehle für die Architektur codiert werden. In Tab. 6 sind die derzeit unterstützten Befehle aufgelistet. Es können neue Maschinencode-Kommandos hinzugefügt werden. Dafür dient unter anderem die Konfigurationsdatei des Assemblers, um die Struktur der Befehle zu beschreiben. Code 34 im Anhang zeigt einen Auszug aus der Konfiguration für die bestehenden Befehle. Die Optionen `LineSize`, `PlaceSize` und `OpCodeSize` beschreiben die Anzahl der Bits, welche für die Darstellung im Maschinencode-Binärformat verwendet werden und entsprechen der Bitanzahl wie in Tab. 5 beschrieben. Eine genauere Beschreibung der Assemblerkonfiguration erfolgt in Abschnitt Assembler Configuration File.

3.3.2 Aufbau der Assembler-Sprache

Es werden im Maschinencode fünf Arten von Kommandos nach der Anzahl der notwendigen Operanten unterschieden: Control, No-Operand, One-Operand, Two-Operand und Three-Operand. Sie sind in Tab. 6 gelistet und jeweils farblich gekennzeichnet. Nicht alle Zustände der *Central Control Unit* sind mittels Assemblerkommandos ansprechbar. Zu ihnen gehören beispielsweise die Control-Operationen der CCU selbst wie das Laden und Dekodieren des nächsten Befehls. Die für Nutzer verwendbaren Kommandos sind in Tab. 6 mit einem „X“ unter „*User-Access*“ markiert. Die anderen Zustände werden von der CCU während des Betriebs der Architektur verwendet. Neben der Umsetzung von Maschinenbefehlen werden auch zusätzlich arithmetische Operationen unterstützt. Eine Übersicht zu allen Assemblerbefehlen bindet sich in Tab. 7.

Tab. 6: Verfügbare Maschinencode-Befehle der Central Control Unit

CMD-ID (Hex)	Command	User-Access	Description
00	`NOOP`	X	Keine Operation durchführen
01	`ADAPT_PP`		*Program Pointer* anpassen
02	`FETCH`		Lade das nächste Kommando aus dem Instruktionsspeicher
03	`DECODE`		Dekodiere die aktuelle Instruktion
04	`WAIT_READY`	X	Pausiere, bis das *Ready*-Signal des *Virtual Coarse Grained Reconfigurable Array* auf High wechselt
05	`LOADD <Addr> <Line> <Place>`	X	Lade das Datum im *Shared Memory* unter der gegebenen Adresse in den *Data-Input Pre-Fetcher*
06	`LOADDA <Addr> <Line>`	X	Lade benachbarte Daten aus dem *Shared Memory* ab der gegebenen Adresse in den *Data-Input Pre-Fetcher*
07	`STORED <Addr> <Line> <Place>`	X	Speichere das Datum aus dem *Data-Output* Puffer an die gegebene Adresse im *Shared Memory*
08	`STOREDA <Addr> <Line>`	X	Speichere die gesamte Zeile des *Data-Output* Puffers ab der gegebenen Adresse in das *Shared Memory*

CMD-ID (Hex)	Command	User-Access	Description
09	`LOADPC <Addr> <Line>`	X	Lade die gesamte Konfiguration der Prozesselemente unter der gegebenen Adresse in den PE-Konfiguration *Pre-Fetcher*
0A	`LOADCC <Addr> <Line>`	X	Lade die gesamte Konfiguration der virtuellen Kanäle und der *Synchronization Unit* unter der gegebenen Adresse in den *Channel*-Konfiguration *Pre-Fetcher*
0B	`START`	X	Starte die Verarbeitung des *Virtual Coarse Grained Reconfigurable Arrays*
0C	`FINISH`	X	Markierung für das Ende der Maschinencodesequenz zur Verarbeitung in der *Central Control Unit*
0D	`WAIT_MMU`		Wartezustand auf die MMU, während diese den Datenzugriff auf dem *Shared Memory* und die *Pre-Fetchers*/Datenpuffer durchführt
0E	`CONT_MMU`		Fortsetzung der Verarbeitung nach dem Warten auf die MMU
0F	`SLT_DIC_LINE <Line>`	X	Auswahl der aktuellen Zeile für den *Data-Input Pre-Fetcher*; die

CMD-ID (Hex)	Command	User-Access	Description
			aktive Zeile wird im aktuellen VCGRA-Prozess verwendet
10	`SLT_DOC_LINE <Line>`	X	Auswahl der aktuellen Zeile für den *Data-Output* Puffer; die aktive Zeile wird im aktuellen VCGRA-Prozess verwendet
11	`SLT_PECC_LINE <Line>`	X	Wählt die aktuelle Zeile für die Konfiguration der Prozesselemente aus; die aktive Zeile wird im aktuellen VCGRA-Prozess verwendet
12	`SLT_CHCC_LINE <Line>`	X	Wählt die aktuelle Zeile für die Konfiguration der *Virtual Channels* und *Synchronization Unit* aus; die aktive Zeile wird im aktuellen VCGRA-Prozess verwendet

In Code 7 ist ein kurzer Assembler-Programmcodeausschnitt eingefügt, an welchem die Hauptfunktionalitäten und Randbedingungen erklärt werden.

```
#Constants
CONST const1 10
CONST const2 0x40

#Variables
VAR var1 0
VAR var2 0x567

#Loops and Arithmetic
VAR caddr coeff0
SLCT_DIC_LINE 1
VAR place 0
LOOP 0 4 1
    LOADD caddr 0 place
    ADDI place 2
    ADD caddr coeffSize
POOL

#VCGRA control
START
FINISH
WAIT_READY
```

Code 7: Beispiel – VCGRA Assembler-File

Kommentare werden mit „#" eingeleitet. Es sind derzeit nur Kommentare in einer separaten Zeile erlaubt, um damit den Parser in der Assemblersoftware einfacher zugestalten. Mehrzeilige Kommentare müssen mit jeweils einem eigenen „#" eingeführt werden. Schlüsselwörter werden immer in GROSSBUCHSTABEN erwartet. Ein Kommando wird immer zeilenweise interpretiert, das heißt, jeder Assemblerbefehl muss mit einem Zeilenumbruch abgeschlossen werden. Auf ein Befehls-Endezeichen, wie beispielsweise das „;" in VHDL oder C/C++, wurde bewusst verzichtet, da Assembler-Sprache sich näher an der Maschinensprache orientiert und daher komplexere Sprachkonstrukte nicht erlaubt. Auch wird versucht, jeden Maschinenbefehl aus Tab. 6 direkt in Maschinencode zu übersetzen, sodass eine Zeile Assemblerbefehl aus dem Code direkt interpretierbar ist.

Es sind sowohl Konstanten als auch Variablen zulässig. Konstanten werden mit dem Schlüsselwort „CONST", Variablen mit dem Schlüsselwort „VAR" eingeleitet. Es folgt der Bezeichner der Variablen oder Konstanten und der zugewiesene Wert. Alle Elemente werden durch Leerzeichen oder Tabs getrennt. Bezeichner müssen mit einem Buchstaben beginnen, können aber auch Zahlen enthalten. Sonderzeichen

werden nicht unterstützt genauso wie Leerzeichen. Variablen werden „*Case-Sensitive*" unterschieden. Das bedeutet, „`name`", „`NAME`" und „`Name`" referenzieren jeweils eine eigene Variable oder Konstante. Konstanten können nur einmalig innerhalb eines „Namensraums" definiert und anschließend referenziert werden. Wird eine Konstante nochmals innerhalb eines Namesraumes beschrieben, kommt es zu einem Übersetzungsfehler. Auf Namensräume wird in Abschnitt 3.3.3.1 „Softwaredesign" im Detail eingegangen. Die Werte für Variablen oder Konstanten können in dezimaler bzw. hexadezimaler Darstellung angegeben werden.

Die Assemblersprache unterstützt Schleifen mit fester Länge. Eine solche Schleife wird eingeführt mit dem Schlüsselwort „`LOOP`" und abgeschlossen mit dem Schlüsselwort „`POOL`". Beim Beginn einer Schleife müssen deren Startwert, Schrittweite und Endwert jeweils getrennt durch ein Leerzeichen mit angegeben werden. Derzeit unterstützt der Assembler den Zugriff auf die Laufvariable der Schleife *nicht*. Dies liegt daran, dass keine Variable für das Anlegen der Schleife definiert wird, welche den Zählwert enthält. Eine Systemvariable der Assemblersprache, beispielsweise ein „`$i`", kann auch nicht verwendet werden, da sie den Zugriff bei geschachtelten Schleifen nicht abbilden kann. Es handelt sich bei Assembler auch um eine maschinennahe Sprache, weshalb auf solche komplexeren Konstrukte verzichtet wird. Daher gilt: Alle Variablen, welche innerhalb der Schleife verändert werden, müssen vor dieser zunächst definiert sein. Dies gilt auch für geschachtelte Schleifen, welche ebenfalls vom Assembler unterstützt werden. Der Startwert, die Schrittweite und der Endwert müssen während der Übersetzungszeit feststehen. Eine Berechnung dynamischer Grenzen oder eine Abbruchbedingung innerhalb einer Schleife werden nicht unterstützt. Dies liegt an der Tatsache, dass die Architektur selbst noch keine Bedingungen und Verzweigungen selbstständig verarbeiten kann. Die Intelligenz liegt derzeit im Assembler selbst. So wird die Übersetzung einer Schleife hin zu Maschinencode dadurch umgesetzt, dass die Schleife ausgerollt wird und die Maschinenbefehle mit angepassten Parametern sequenziell in der passenden Reihenfolge erstellt werden. Aus diesem Grund müssen zur Übersetzungszeit alle Grenzen feststehen. Durch diese Maßnahme wird die Komplexität der Architektur reduziert, da diese nicht zusätzliche Arithmetik/Logik benötigt, um über Schleifenabbrüche oder Verzweigungen zu entscheiden. Andererseits schränkt es den Funktionsumfang der Architektur ein, weshalb eine Erweiterung in Zukunft sinnvoll erscheint, falls die Architektur gegebenenfalls nicht als Beschleuniger, sondern beispielsweise als eigenständige Verarbeitungseinheit agieren soll.

Die Assemblersprache unterstützt arithmetische Befehle wie Addition, Subtraktion oder Multiplikation. Die Berechnungen finden nicht auf der Architektur statt, sondern werden zur Übersetzungszeit durch den Assembler durchgeführt. Die verwendeten Befehle haben nichts mit der Applikation zu tun, welche auf dem VCGRA verarbeitet werden soll und welche sich an der graphischen Repräsentation nach Abschnitt 2.1.3.5 orientiert. Sie dienen dazu, beispielsweise Adressen im *Shared Memory* während der Übersetzungszeit zu berechnen. Die arithmetischen Operationen beschränken sich derzeit nur auf Ganzzahlen aus der Menge der natürlichen Zahlen. Für die Berechnungen der Adressen oder des Platzes (*Place*) innerhalb einer Zeile (*Line*) eines *Pre-Fetcher* ist dies ausreichend, da hier nur ganzzahlige Ergebnisse sinnvoll und gültig sind.

Neben den bereits beschrieben Assembler-Befehlen gibt es auch solche, die direkt in Maschinenbefehle nach Tab. 6 übersetzt werden. Zur Übersichtlichkeit sind in Tab. 7 noch einmal alle Assemblerbefehle und dessen Parameter aufgelistet.

Tab. 7: Übersicht – Assemblerbefehle

Lfd. No.	**Befehl**	**Beschreibung**
Arithmetische Befehle		
1	`ADDI <op1> <number>`	Addiert eine Ganzzahl auf eine Variable `<op1>`. Es sind auch negative Ganzzahlen zulässig.
2	`ADD <op1> <op2>`	Addiert `<op2>` zu `<op1>` und speichert das Ergebnis in `<op1>`. Für `<op2>` sind Konstanten oder Variablen zulässig. `<op1>` muss eine Variable sein.
3	`SUBI <op1> <number>`	Subtrahiert eine Ganzzahl von einer Variablen `<op1>`. Es sind auch negative Ganzzahlen zulässig.
4	`SUB <op1> <op2>`	Subrahiert `<op2>` von `<op1>` und speichert das Ergebnis in `<op1>`. Für `<op2>` sind Konstanten oder Variablen zulässig. `<op1>` muss eine Variable sein.

Lfd. No.	Befehl	Beschreibung
5	`MULI <op1> <number>`	Multipliziert Variable <op1> mit einer Ganzzahl. Es sind auch negative Ganzzahlen zulässig.
6	`MUL <op1> <op2>`	Multipliziert <op1> mit <op2> und speichert das Ergebnis in <op1>. Für <op2> sind Konstanten oder Variablen zulässig. <op1> muss eine Variable sein.
Schleifen		
7	`LOOP <start> <stop> <step>`	Startet einen Schleifenbereich für eine Schleife mit festen Grenzen und fester Schrittweite.
8	`POOL`	Schließt einen Schleifenbereich ab.
Maschinenbefehle		
9	`START`	Startet das VCGRA der Architektur.
10	`FINISH`	Signalisiert das Ende der Assemblersequenz. Hier wird dem aufrufenden Prozessorsystem mitgeteilt, dass die Verarbeitung beendet ist. Der „*Program Pointer*“ der Architektur wird auf den Anfang zurückgesetzt, um die Verarbeitung erneut zu starten.
11	`WAIT_READY`	Die CCU wird per Assembler dazu veranlasst, auf das Ende der Berechnung im VCGRA zu warten, bevor mit dem nächsten Befehl fortgesetzt wird. Solche Synchronisationen sind sinnvoll, wenn parallel zur Verarbeitung im VCGRA die

Lfd. No.	Befehl	Beschreibung
		Pre-Fetcher mit neuen Daten initialisiert werden.
12	`NOOP`	Leeroperation
13	`SLCT_DIC_LINE <line>`	Mit `<line>` wird die Zeile des *Data-Input Pre-Fetcher* gewählt, die als Eingang am VCGRA verwendet werden soll.
14	`SLCT_DOC_LINE <line>`	Mit `<line>` wird die Zeile des *Data-Output* Puffers gewählt, die als Ausgang am VCGRA verwendet werden soll.
15	`SLCT_PECC_LINE <line>`	Mit `<line>` wird die Zeile des PE-Konfiguration *Pre-Fetcher* gewählt, die als Eingang am VCGRA verwendet werden soll.
16	`SLCT_CHCC_LINE <line>`	Mit `<line>` wird die Zeile des *Channel*-Konfiguration *Pre-Fetcher* gewählt, die als Eingang am VCGRA verwendet werden soll.
17	`LOADDA <addr> <line>`	Lädt ab *Shared Memory* Adresse `<addr>` Daten für eine ganze Zeile in den *Data-Input Pre-Fetcher* in die mit `<line>` gewählte Zeile.
18	`LOADD <addr> <line> <place>`	Lädt ein Datum in Zeile <line> Datum `<place>` von Adresse `<addr>` im *Shared Memory* in den *Data-Input Pre-Fetcher*.
19	`STOREDA <addr> <line>`	Speichert eine Zeile des *Data-Output* Puffer ab Adresse `<addr>` in den *Shared Memory*. Mit `<line>` wird die Zeile des *Data-Output* Puffer gewählt, von welcher die Daten geladen werden sollen.

Lfd. No.	Befehl	Beschreibung
20	`STORED <addr> <line> <place>`	Speichert ein Datum in Zeile `<line>` Datum `<place>` an Adresse `<addr>` im *Shared Memory*.
21	`LOADPC <addr> <line>`	Lädt eine Konfiguration für die Prozesselemente des *Virtual Coarse Grained Reconfigurable Arrays* in den korrespondierenden *Pre-Fetcher* in Zeile `<line>`.
22	`LOADCC <addr> <line>`	Lädt eine Konfiguration für die virtuellen Kanäle des *Virtual Coarse Grained Reconfigurable Arrays* ab Adresse `<addr>` im *Shared Memory* in den korrespondierenden *Pre-Fetcher* in Zeile `<line>`.

Nur die Maschinenbefehle werden auch vom Assembler in interpretierbare Befehle der *Central Control Unit* der Architektur übersetzt. Die anderen Befehle werden verwendet, um im Assembler Berechnungen auszuführen, um beispielsweise Parameter für den Maschinencode zu berechnen.

Die gegebenen Befehle sind *keine* Repräsentation der zu beschleunigenden Applikation, sondern sie steuern nur das *Virtual Coarse Grained Reconfigurable Array* innerhalb der Architektur bzw. entsprechen Kommandos für die *Central Control Unit*. Aus diesem Grund ist für viele Anwendungen der gegebene Befehlssatz in Assembler bereits ausreichend, da sich Daten zwischen VCGRA und *Shared Memory* austauschen und Konfigurationen zu gegebenen Zeitpunkten automatisch an das VCGRA anlegen lassen. Als Erweiterung sind Abbruchbedingungen in Schleifen und Verzweigungen noch hilfreich, um die Ausführung des Programmcodes durch das VCGRA noch dynamischer und flexibler zu gestalten. Hierbei ist es für Verzweigungen vorstellbar, dass durch die *Pre-Fetcher* Konfigurationen und Daten für beide Zweige einer Verzweigung bereits vorgeladen werden und das Ergebnis der aktuellen Berechnung im VCGRA bestimmt, welche Konfigurationen oder welche Zeilen innerhalb der *Pre-Fetcher* an das VCGRA gelegt werden für die weitere Berechnung. Die CCU verfügt aber

noch nicht über die notwendige Intelligenz, um diese Auswertung durchzuführen, weshalb diese Funktionen noch nicht implementiert sind. Auch Abbruchbedingungen in Schleifen lassen sich auf Grund fehlender Vergleichsfähigkeiten der Ergebnisse durch die *Central Control Unit* ebenfalls noch nicht umsetzen. Diese beiden Programmier-Mechanismen stellen aber nützliche Funktionalitäten für zukünftige Erweiterungen dar.

Nachdem ausgiebig auf die Assemblersprache für die Architektur eingegangen worden ist, wird nun der Assembler selbst erläutert.

3.3.3 Der Assembler – Funktionsbeschreibung und Aufbau

3.3.3.1 Softwaredesign

Der CGRA Assembler ist in *C++* programmiert. Seine Hauptkomponenten und deren Abhängigkeiten sind in Abb. 46 dargestellt. Neben diesen Komponenten, auf die in den folgenden Abschnitten noch eingegangen wird, sind auch Komponenten aus der „*Boost*“ Bibliothek [34] verwendet:

- *Program Options*: Diese Bibliothek erlaubt die Programmierung eines „*Command Line Interfaces*“ (CLI) und dessen Parser. Sie wird verwendet, um die Kommandozeilenschnittstelle des CGRA-Assemblers zu realisieren.
- *File System*: Diese Bibliothek wird verwendet, um Dateizugriffe und Dateipfadüberprüfungen zu realisieren.
- *Property Tree*: Diese Bibliothek wird verwendet, um die Konfigurationsdatei des Assemblers – siehe auch Abschnitt 3.3.3.3 – zu parsen.
- *Format*: Diese Bibliothek wird verwendet, um Fomatstrings zu verwenden, damit der Assembler aus den gegebenen Parametern direkt den Maschinenbefehl im richtigen Hex-Format ausgibt.

Als Boost Version wurde „*1.66.0*“ verwendet, da diese in der Linux-Distribution des Entwicklungsrechners im Paketmanager verfügbar war. Durch eine kleine Anpassung im *CMakeLists.txt* File zur Boost Bibliothek sind aber auch höhere Versionen verwendbar.

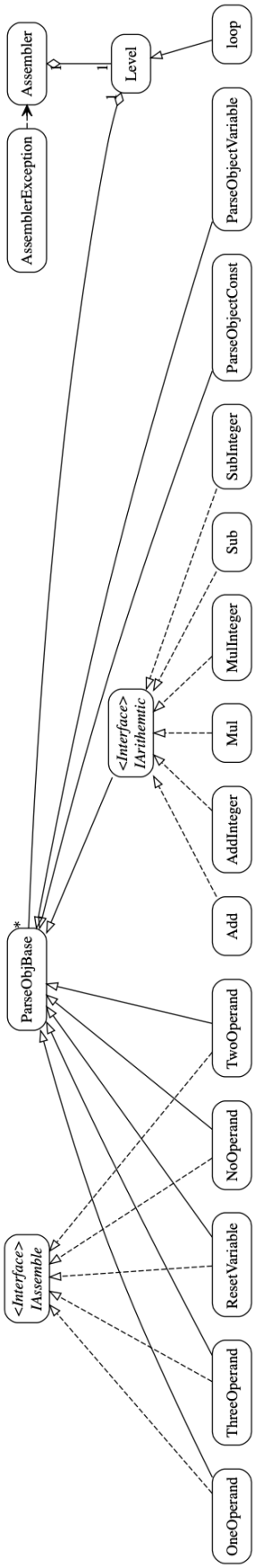

Abb. 46: Klassendiagramm – CGRA-Assembler-Programm

3.3.3.1.1 Level

Level bildet eine Basisklasse und wird zur hierarchischen Gliederung des Quellcodes genutzt. Wie in Abschnitt 3.3.3.1.7 beschrieben werden durch Levels auch Namensräume für Variablen und Konstanten erzeugt. Es gibt immer mindestens ein Level („*root*[8]"-Level). Ein Level kann beliebig viele „*Child-Level*" besitzen, aber jedes *Child-Level* besitzt genau ein „*Parent-Level*". Jedes Level wiederum besitzt eine Liste über die Parseobjekte innerhalb seiner Hierarchieebene. Als neue Level einer Hierarchie fungieren beispielweise Schleifen oder Entscheidungen („*if-then-else*"). Während des Parsevorgangs einer Datei wird also durch Einträge wie „`LOOP`" jeweils ein neues Level zur Hierarchie hinzugefügt. Parseobjekte, welche innerhalb dieser neuen Ebene verarbeitet werden, werden dieser Ebene zugeteilt und in einer Datenbank je Level abgelegt. Die Klasse Level enthält eine statische Klassenvariable „`activeLvl`". Diese enthält das Handle auf das gerade aktive Level. Über die statische Funktion „`getCurrentLevel`" kann das Handle erfragt werden. Wird im Assembler die Anweisung `POOL` erreicht, wird Klassenvariable „`activeLvl`" auf das *Parent-Level* des aktuellen Levels gesetzt. Damit wird in der Hierarchie eine Ebene zurückgegangen.

3.3.3.1.2 Loop

`LOOP` ist die bisher einzige abgeleitete Klasse von Level, welche aktuell im Assembler implementiert ist. Sie erweitert die Basisklasse Level um Variablen zur Speicherung des Startwerts, Endwerts und der Schrittweite sowie um Mechanismen, um das „*Loop-Unrolling*" durchzuführen. Weitere Einzelheiten sind bereits in Abschnitt 3.3.2 aufgeführt.

3.3.3.1.3 ParseObjectBase

Diese bildet die Basisklasse für alle interpretierbaren Elemente des Quellcodes. Der Quellcode einer Assemblerdatei wird Zeile für Zeile interpretiert. Jede Zeile kann ein bestimmtes Parseobjekt enthalten. Aus diesem Grund implementiert ein Parseobjekt über diese Basisklasse essentielle Eigenschaften auf die im Folgenden detailliert eingegangen wird:

- `Level`: Beim Erstellen wird in der Regel ein Parseobjekt immer genau einem Level zugeordnet. Ausnahme bilden Ganzzahlkonstanten, auf die noch separat

[8] Engl. Wurzel: Beschreibt in der Informationstechnik häufig den Startpunkt einer Hierarchie.

eingegangen wird. Das Handle zu einem Level wird verwendet bei der Suche nach Konstanten und Variablen. Dafür wird zunächst im zugehörigen Level nach den entsprechenden Operanten einer Anweisung gesucht. Sind diese nicht vorhanden, wird rekursiv die Levelhierarchie durchsucht, bis der entsprechende Operant gefunden wurde oder bis die Wurzel der Levelhierarchie erreicht ist. Kann der Operant nicht gefunden werden und handelt es sich nicht um eine Ganzzahlkonstante, wird ein Fehler beim Parsen zurückgegeben. Andernfalls erhält die Operation ein Handle auf den gefundenen Operanten. Aus diesem Grund müssen Variablen und Konstanten **vor** ihrer ersten Benutzung definiert werden. Eine Ausnahme bilden Ganzzahlkonstanten. Diese können auch nachträglich durch die Verwendung in einem Befehl angelegt werden. Mehr noch, sie werden, um Platz zu sparen möglichst nur einmalig angelegt und mehrfach referenziert. Das bedeutet, wenn innerhalb einer Levelhierarchie zwischen *Leaf*[9]-Level und *Root*-Level mehrfach eine Ganzzahlkonstante mit gleichem Wert verwendet wird, wird diese immer wieder referenziert. Derzeit funktioniert das nur innerhalb eines Zweiges einer Hierarchie, denn die Ganzzahlkonstanten sind nicht global über eine Datenbank angelegt, weshalb die rekursive Suche nur in einem Zweig funktioniert. Hier wäre ein Punkt, an dem die Softwarestruktur verbessert werden kann, um effektiver und ressourcenschonender zu arbeiten. Dafür sind Ganzzahlkonstanten in einer globalen Datenbank zu sammeln, um diese für den Übersetzungsvorgang nur einmalig zu erstellen. Ähnliche Konzepte sind bereits bekannt aus anderen objektorientierten Sprachen wie Python, in deren Objekte auch nicht neu erzeugt werden, sondern nur Referenzen auf diese angelegt werden um so Speicherbedarf zu reduzieren [35].

- `COMMANDCLASS`: Assembler Kommandos werden Klassen zugewiesen – Kommandoklasse. Es gibt wie oben beschrieben neben Variablen und Konstanten auch Klassen, welche nach der Anzahl der Parameter (siehe auch Tabelle Tab. 6) unterteilt werden. Über die Kommandoklasse wird nach dem Parsen die weitere Verarbeitung gesteuert oder es werden semantische Überprüfungen durchgeführt, wie beispielsweise der Versuch, auf einer Konstante eine Schreiboperation auszuführen. Wenn es sich dagegen bei der Kommandoklasse um einen interpretierbaren Befehl der *Central Control Unit*

[9] Engl. Blatt: Beschreibt in der Informationstechnik häufig den Endpunkt einer Hierarchie.

handelt, wird das Erzeugen des Maschinenbefehls veranlasst, während bei arithmetischen Operationen wiederum Berechnungen und Aktualisierungen der Variablen des Assemblerprogramms ausgeführt werden. Durch die Verwendung der Kommandoklasse wird beim Erzeugen des Parseobjekts die weitere Verwendung bereits genau definiert. Bei der Implementierung können dadurch auch statische *Typecasts* statt dynamische *Typecasts* verwendet werden, was den Aufwand für *„Run Time Type Identification“* (RTTI) spart.

- `Commandline`: Für die Debug-Ausgabe wird in jedem Parseobjekt die korrespondierende Zeile des Assemblerfiles gespeichert.
- `LineNumber`: Für die Debug-Ausgabe wird in jedem Parseobjekt die korrespondierende Zeilennummer des Assemblerfiles gespeichert.

Parseobjekte werden während des Einlesens des Assemblerfiles dynamisch erzeugt – Ablage der Daten auf dem Heap – und pro Level über einen Vektor der Basisklasse, welche Handles abspeichert, verwaltet. Aus diesem Grund ist die Klassifizierung über die Kommandoklasse essenziell, da sonst für die weitere Verarbeitung aus der Basisklasse nicht mehr auf das eigentliche Parseobjekt geschlossen werden kann.

3.3.3.1.4 IAssemble

Parseobjekte, welche Maschinencode für die *Central Control Unit* der VCGRA Architektur erzeugen sind von dieser Klasse abgeleitet und implementieren jeweils die drei folgenden Funktionen:

```
virtual std::string assemble(const boost::property_tree::ptree &ptreeA) = 0;
virtual uint32_t getMachineCodeId(void) const = 0;
virtual uint32_t setMachineCodeId(const uint32_t machineIdA) = 0;
```

Der Maschinencode eines Parseobjektes ist eindeutig einer Operation der *Central Control Unit* zugeordnet und kann in der Konfigurationsdatei (siehe auch Abschnitt 3.3.3.3) eingesehen und angepasst werden. Die Zuordnung **muss** zwingend zum Befehlssatz der CCU der Architektur passen. Über die *Get/Set* Methoden lassen sich diese Maschinencodeidentifier auslesen und ggf. anpassen. Die wichtigere Methode ist die „`assemble`“-Methode, welche den eigentlichen Maschinencode als String zurück liefert. Für die Formatierung des Strings im richtigen Format wird aus der Boost Bibliothek [34] der Formatstring verwendet:

```
boost::format m_fmtStr{"0x%1$04X%2$04X"};
```

Der Format-String erwartet zwei Elemente:

1. Die *Shared Memory* Adresse, welche in einen 16 Bit Hexwert ausgegeben wird. Führende Stellen werden ggf. mit Nullen aufgefüllt, wenn der Ganzzahlwert nicht die gesamten vier Nibble[10] benötigt.
2. 16 Bit, welche den Befehl als Maschinencode-ID, die Zeile und ggf. den Platz innerhalb eines *Pre-Fetcher* im Hexformat angeben (siehe auch Abschnitt 3.3.1). Die Parameter für die Beschreibung des Aufbaus des Maschinencodes sind im Konfigurationsfile 3.3.3.3 einstellbar.

Die notwendigen Informationen für einen Maschinencodebefehl werden durch Shiftoperationen an die richtigen Stellen geschoben und anschließend durch den Formatstring ausgegeben. Derzeit wird für die Verwendung in der SystemC-Simulation ein `std::vector` in C++ aus 32 Bit Ganzzahlelementen generiert, welcher als Abhängigkeit in die Main der Simulation „includiert“ wird und beim Erzeugen der *Central Control Unit* Instanz als Referenz mit übergeben wird:

```
auto toplevel = new cgra::TopLevel{"TopLevel", cgra::assembly.data(),
cgra::assembly.size()};
```

Das „`TopLevel`” beschreibt die gesamte Architektur und verbindet alle Teilkomponenten zur funktionsfähigen Simulation. Es wird in Abschnitt 4.5 noch genauer auf die Implementierung der Architektur eingegangen.

Der Maschinencode ist damit statisch in die Simulation gelinkt und wird nicht per Aufruf ausgetauscht. Für eine Flexibilisierung kann dies als Verbesserung noch getan werden. Vorstellbar ist ein Kommandozeilenparameter in der SystemC-Simulation, welche einen Pfad zu einer Datei enthält, welche ihrerseits die Maschinencodesequenz beinhaltet. Dieser Pfad kann für jeden Aufruf beliebig gesetzt werden.

3.3.3.1.5 IArithmetic

Dieses Interface ist speziell für arithmetische Operationen in der Assemblersprache vorgesehen. Der Assembler ruft bei einer arithmetischen Operation immer die Methode „`processOperation`“ auf. Diese ist von jeder abgeleiteten Klasse zu implementieren und im Interface „*IArithmetic*“ nur als abstrakte Funktion definiert.

[10] Umschreibt die Datenmenge von 4 Bit.

Arithmetische Operationen werden ebenfalls als Parseobjekte behandelt und in der Datenbank eines Levels abgelegt. Im Gegensatz zu anderen Parseobjekten erzeugen sie keinen Maschinencode, sondern werden vom Assembler direkt ausgewertet. Dies liegt am Fehlen entsprechender verarbeitender Hardware in der *Central Control Unit* und ist auch nicht Zweck dieser Architekturkomponente. Die CCU dient nur der Steuerung des *Virtual Coarse Grained Reconfigurable Arrays* und nicht der Verarbeitung von Nutzdaten.

3.3.3.1.6 ParseObjectConst/ParseObjectVariable

Diese beiden Klassen werden eingesetzt, um symbolische Variablen oder Konstanten im Assemblercode zu verwenden. Sie müssen vor ihrer Benutzung in einem Kommando zunächst angelegt und initialisiert werden. Dies geschieht mit den Schlüsselwörtern `VAR` und `CONST` im Assemblerquelltext. Konstanten erlauben nur einen einzigen schreibenden und beliebig viele lesende Zugriffe auf ihren Wert, während Variablen auch mehrfach geschrieben werden können. Eine Überprüfung des Zugriffs erfolgt über die Kommandoklasse gespeichert in der Basisklasse „`ParseObjBase`“ nach Abschnitt 3.3.3.1.3. Das Zuweisen eines neuen Wertes an eine Konstante führt zu einem Übersetzungsfehler während des Parsing. Beide Parseobjekte beinhalten neben ihrem Wert noch ihren Namen. Dieser Name wird verwendet bei der Suche nach Operanten für andere Assemblerkommandos. Der Vergleich bei der Suche nach der passenden Variablen erfolgt „*case-sensitiv*“.

3.3.3.1.7 ResetVariable

Diese Kommandoklasse wird verwendet, falls eine Variable im Verlauf des Assemblercodes erneut initialisiert wird. Eine erneute Initialisierung erfolgt, wenn der Variable nochmals über das Schlüsselwort `VAR` ein Wert zugewiesen wird.

```
VAR var 10      #Definition of a variable
...
VAR var 20      #Redefinition/Reset of a variable
```

Das Parseobjekt dieser Kommandoklasse speichert eine Referenz auf die Variable, die sie beeinflusst und eine Referenz auf den Wert, auf den die Variable gesetzt werden soll. Dies hat zwei Gründe: Zum einen ist der Assembler-Prozess in drei Teilprozesse aufgegliedert: „*Parsing*“, „*Assembling*“ und „*Writing*“. Durch diese Kommandoklasse wird für das Assembling der Zeitpunkt im Assemblercode markiert, an dem der Variable ein neuer Wert zugewiesen wird. Dadurch können die beiden Prozesse getrennt werden. Dies hat Vorteile, wenn beispielsweise für mehrere

Assemblercodedateien eine parallele Verarbeitung der Files erreicht werden soll. Es kann nach dem Parsen des ersten Files mit dem Anlegen der Datenbank der Parseobjekte für ein zweites File begonnen werden, während ein anderer *Thread* die erste Datenbank für das Assembling nutzt. Das Assembling speichert seine temporären Ergebnisse, welche in einem dritten Prozess wiederum unabhängig in eine Maschinencodedatei geschrieben werden.

Zum anderen kann der Rücksetzwert keine Konstante sein, sondern eine Variable, welche sich während der Interpretation des Quelltextes beispielsweise in einer Schleife durch Berechnung verändert hat oder regelmäßig verändert. Innerhalb einer Levelhierarchie können Konstanten nur einmal angelegt werden. Eine Levelierarchie ist somit vergleichbar mit einem Namensraum in C++. Auch hier können Konstanten gekapselt werden und sind dann für diesen Namensraum nur einmalig anzulegen. Eine erneute Definition einer Konstanten führt zu einem Compiler-Fehler. Der Zusammenhang soll im Folgenden anhand eines Beispiels in Code 8 illustriert werden.

```
1  # global namespace
2  CONST rootVar 1
3  # namespace One
4  LOOP 0 4 1
5    CONST loopVar 10
6    LOOP 0 3 1
7      CONST rootVar 2    # Redefinition of constant raises an error.
8      ...
9    POOL
10   CONST loopVar 20     # Redefinition of constant raises an error.
11   ...
12   LOOP
13     CONST loopVar 30   # Redefinition of constant raises an error.
14     ...
15   POOL
16 POOL
17 # end namespace One
18 # namespace Two
19 LOOP 0 4 1
20   CONST loopVar 20    # No error, because constant is defined only once in
namespace.
21   ...
22 POOL
23 # end namespace Two
24 # end global namespace
```

Code 8: Namensräume – Hierarchieebenen – Definition von Konstanten

Zur besseren Erklärung sind die Namensräume noch einmal als Kommentare miteingefügt. Konstanten, welche auf dem „*root*“-Level angelegt werden, besitzen globalen Charakter für das gesamte Assemblerfile und können daher nur einmal pro Datei angelegt werden. Daraus resultiert die Bezeichnung „`global`“. Ein `LOOP` ist ein neues Level, dementsprechend ein neuer Namesraum. Dabei wird eine neue Levelhierarchie mit dem ersten neuen Level ausgehend vom „*root*“-Level eröffnet. Man erkennt dies im Quellcode daran, dass jeweils die erste LOOP-Anweisung einen neuen Namensraum öffnet, was durch die Kommentare angezeigt wird. Der erste Übersetzungsfehler in *Zeile 7* tritt auf, weil versucht wird, die globale Konstante aus *Zeile 2* zu überschreiben. Der globale Namensraum besitzt auch im Namesraum „*One*“ Gültigkeit. Der zweite Fehler in *Zeile 10* tritt auf, weil `loopVar` in *Zeile 5* für diesen Namensraum bereits definiert wurde, Gleiches gilt für *Zeile 13*. Obwohl ein neues Level in der Hierarchie erstellt wird, entsteht **kein** neuer Namensraum. Diese Einschränkung verhindert *Shadowing* von Variablen und Konstanten und vereinfacht eine rekursive Suche bei der Verwendung der Werte. Damit ist die Konstante bereits in *Zeile 5* für diesen Namensraum festgelegt. Die Zuweisung eines neuen Wertes in *Zeile 20* ist zulässig, weil vom „*root*“-Level aus ein neuer Namesraum geöffnet wird.

3.3.3.1.8 Arithmetic Operations

Arithmetic Operations erwarten als ersten Operanden eine Variable als Type und als zweiten Operanden eine Variable oder eine symbolische Konstante. Die Verwendung von Ganzzahlen bei einer arithmetischen Operation ist durch extra Kommandos mit dem Zusatz „`I`“ implementiert (siehe Abschnitt 3.3.3.1.9). Es sind die Operationen „`ADD`“, „`SUB`“ und „`MUL`“ verfügbar und jeweils als eigene Klassen implementiert. Arithmetische Operationen werden nicht in der *Central Control Unit* verarbeitet, sondern dienen dem Assembler dazu, Adressen oder Orte in einer *Pre-Fetcher*-Zeile zu berechnen. Eine Interpretation der Befehle während der Übersetzung führt zu keinem Maschinencode für die Architektur, sondern aktualisiert/ändert Variablen des Assemblers. Jede arithmetische Operation implementiert das Interface „*IArithmetic*“ mit der Methode „`processOperation`“. Diese Methode führt die korrespondierende arithmetische Operation aus.

3.3.3.1.9 Arithmetic Integer Operations

Diese arithmetischen Operationen bilden eine spezielle Variante der bereits beschriebenen arithmetischen Operationen in Abschnitt 3.3.3.1.8. Sie erhalten den Zusatz „`I`“ im Assemblercode. Es ermöglicht eine Integerkonstante als zweiten Operanden direkt beim Aufruf des Kommandos zu erstellen. Andererseits wird bei

Verwendung dieser Kommandos auch zwingend eine Ganzzahl erwartet, andernfalls wird ein Syntaxfehler ausgegeben:

```
ERROR 1051: Syntax error line 51. Unknown variable.
```

Alle anderen Eigenschaften der Operationen sind vergleichbar zu denen, welche bereits in Abschnitt 3.3.3.1.8 aufgeführt sind.

3.3.3.2 Assembler Command Line Interface

Der Assembler besitzt ein Kommandozeilen basierte Aufrufschnittstelle:

```
cgra_assembler --file <filepath> --config <configfilepath> [--log <logfilepath>]
```

Zwingende Eingabe ist immer die Assemblerdatei, welche verarbeitet werden soll. Diese wird hinter der Option „`--file`“ angegeben. Details zum Assembler sind in Abschnitt 3.3.2 angegeben. Als Dateiendung wird zwingend *asm* erwartet. Ebenfalls eine notwendige Eingabe stellt die Konfigurationsdatei des Assemblers dar. Sie kann entfallen, wenn sich die Datei „*config.xml*“ mit einer gültigen Konfiguration im Aufrufordner der Applikation befindet. Andernfalls kann die Datei beliebig benannt sein und sich an einen beliebigen Ort im Dateisystem befinden. Auf den Aufbau der Konfigurationsdatei wird in Abschnitt 3.3.3.3 eingegangen. Zuletzt kann zusätzlich der Pfad zu einer Logdatei angegeben werden in den Informationen zum Übersetzungsvorgang festgehalten werden. Diese Informationen helfen bei der Fehlersuche. Ist keine Datei angegeben werden die Informationen auf „*standard out*“ ausgegeben.

3.3.3.3 Assembler Configuration File

Die Konfigurationsdatei ist im *XML*-Format abgelegt und enthält Abschnitte über das verwendete VCGRA, die Assembler-Sprache und allgemeine Konfigurationen. Sie wird immer zum Start der Applikation geladen, und ist notwendig für das Übersetzen des Assemblerfiles in Maschinencode, weil es mit den Eigenschaften des *Virtual Coarse Grained Reconfigurable Arrays* die Gültigkeit für beispielsweise Adressen im *Shared Memory* prüft oder die gültigen Kommandos und deren Maschinencode-Repräsentation beschreibt. Die Verwendung einer Konfigurationsdatei soll die Flexibilität zur Unterstützung der Zielhardware erhöhen und die Eingabe über die CLI vereinfachen. Gäbe es keine Konfigurationsdatei, müssten viele Eingaben fest im Programmcode stehen oder umständlich über die CLI eingegeben werden.

`General`

Die General-Sektion beschreibt derzeit nur den Speicherort und Name der Ausgabedatei. Diese muss derzeit als „hpp“ Datei gespeichert werden und entspricht einem C++ Header, welcher anschließend in den Kompilierprozess der SystemC-Simulation mit eingebunden wird.

`VCGRA_Property`

Diese Sektion beschreibt Eigenschaften des *Virtual Coarse Grained Reconfigurable Arrays* näher, welches den Assemblercode verarbeiten soll. Die Eigenschaften des *Virtual Coarse Grained Reconfigurable Arrays* sind deshalb mit in die Konfiguration eingeflossen, damit während der Erzeugung des Maschinencodes an verschiedenen Stellen geprüft werden kann, ob die verwendeten Parameter für den Maschinenbefehl auch gültig sind. Folgende Eigenschaften können über die Konfigurationsdatei eingestellt werden:

Tab. 8: Verfügbaren Eigenschaften im Assembler Configuration File

Lfd. No.	Eigenschaft	Beschreibung
1	`Available_Memory`	Verfügbarer Speicher in Bytes des *Shared Memory*. Die Adressierung im *Shared Memory* beginnt bei 0x0000.
2	`Num_Dic_Lines`	Anzahl an verfügbaren Zeilen im *Data-Input Pre-Fetcher*.
3	`Num_Dic_Places`	Anzahl der verfügbaren Plätze einer Zeile im *Data-Input Pre-Fetcher*.
4	`Num_Doc_Lines`	Anzahl an verfügbaren Zeilen im *Data-Output* Puffer.
5	`Num_Doc_Places`	Anzahl der verfügbaren Plätze einer Zeile im *Data-Output* Puffer.
6	`Num_PC_Lines`	Anzahl an verfügbaren Zeilen im *Data-Input Pre-Fetcher*.

Lfd. No.	Eigenschaft	Beschreibung
7	`Num_CC_Lines`	Anzahl an verfügbaren Zeilen im *Data-Input Pre-Fetcher*.

`Assembler_Property`

Hier werden die verfügbaren Assemblerbefehle nach Gruppen sortiert angelegt und deren Eigenschaften definiert. Zu Beginn wird über die drei Parameter „`LineSize`“, „`PlaceSize`“ und „`OpCodeSize`“ genauer spezifiziert, welche Anzahl von Bits im Maschinenbefehl verwendet wird, um die Anzahl der verfügbaren Zeilen und Plätze innerhalb eines *Pre-Fetcher* oder Puffers zu beschreiben und wie viele Bits für den *Command Identifier* der *Central Control Unit* zur Verfügung stehen. Diese Einstellungen müssen derzeit nicht angepasst werden, sind aber bereits mit aufgeführt, falls das Befehlswort der CCU angepasst werden soll. Sie werden dynamisch aus dieser Datei geladen und sind nicht fest im Quellcode des Assemblers hinterlegt. Für die Maschinenbefehle ist vorwiegend der *Command Identifier* für einen Befehl definiert, der vom Interpreter der *Central Control Unit* der Architektur ausgewertet wird. Die Gruppen, nach denen die Operationen in der Konfigurationsdatei sortiert sind, orientieren sich an den Befehlsgruppen nach Tab. 6. Der Codeausschnitt Code 34 im Anhang I. Quellcodeanhänge zeigt einen entsprechenden Auszug aus der Konfigurationsdatei. Eine Anpassung dieses Bereichs ist in der Regel nur notwendig, wenn der *Central Control Unit* der Architektur neue Befehle hinzugefügt werden oder wenn die *Command Identifier* der Befehle an Architekturänderungen angepasst werden müssen.

3.4 Zusammenfassung

Die beschriebenen Verbesserungen der Abschnitte 3.1 bis 3.3 beseitigen gezielt das *Bottleneck* Datenaustausch zwischen Prozessorsystem und VCGRA. Wie in Kapitel 5 gezeigt ist, tragen die angestrebten Verbesserungen wesentlich zu einer Beschleunigung des Evaluationsbeispiels aus Abschnitt 2.3.6 bei. Die zusätzlichen Erweiterungen beeinflussen allerdings die Größe der Architektur. Je größer das VCGRA – Anzahl der Ebenen, der Prozesselemente und der Datenein- bzw. Datenausgänge – desto größer auch der notwendige Platzbedarf für die zusätzlichen *Pre-Fetcher*, Datenpuffer und „*Dedicated Memory Regions*“.

Die Verwendung eines Assemblers als Programmierhilfe für den Anwender erhöht die Akzeptanz der Architektur. Das Design der Maschinenbefehle ist einfach, stabil und dennoch auf weitere Befehle erweiterbar. Die Intelligenz der Architektur steckt hierbei vor allem im Assembler. Die *Central Control Unit* ist noch nicht darauf ausgelegt Entscheidungen zu treffen und bestimmte Adressen im Programmcode anzuspringen. Schleifen beispielsweise benötigen feste Grenzen, die zur Compilezeit feststehen und werden anschließend ausgerollt. Weitere Erweiterungen in dieser Richtung konnten vom Autor nicht mehr umgesetzt werden, sind aber für die Nutzbarkeit und Akzeptanz der Architektur von Vorteil. Vor allem Sprünge für Entscheidungen oder Schleifenabbrüche würden den Maschinencode wesentlich verkürzen und auch dynamische Grenzen von Schleifen zulassen. Gegen diesen Aspekt spricht wiederum die Spezialisierung der Architektur. Je universeller die Architektur, desto komplexer ihr Aufbau und damit ihr notwendiger Platzbedarf. Die Möglichkeit Funktionen zu definieren und während der Programmausführung anzuspringen würde ebenfalls dazu beitragen, Programmcode übersichtlicher zu strukturieren und Maschinencode zu verkürzen – mehrfachen Code einsparen und durch Funktionen zentral abbilden. Für die Verwendung von Parametern innerhalb von Funktionsaufrufen oder Schleifenzählern, für die *Central Control Unit* beispielsweise, bräuchte diese als Erweiterung Arbeitsspeicher oder Registersätze. Als Beschleunigerarchitektur kann aber auf komplexere Strukturen vorerst verzichtet werden, da nur dedizierte Programmteile auszuführen sind und nicht eine gesamte Applikation. Diese wird weiterhin überwiegend durch ein Prozessorsystem verarbeitet.

Was durch die gegebenen Maßnahmen noch nicht erreicht wird, sind Beschleunigungen in der Verarbeitung des *Virtual Coarse Grained Reconfigurable Arrays* selbst. Es findet noch kein *Pipelining* zwischen Ebenen des *Virtual Coarse Grained Reconfigurable Arrays* statt und auch ein *Bypassing* von nicht verwendeten Ebenen oder ein *Feedback* von Zwischenergebnissen an den Kopf des *Virtual Coarse Grained Reconfigurable Arrays* ist in der Architektur noch nicht implementiert. Diese zusätzlichen Erweiterungen können die eigentliche Verarbeitung der Prozesselemente und den Datentransfer durch das VCGRA selbst stark beschleunigen. Es muss dann gleichzeitig erreicht werden, dass der Rest der Architektur und die Anbindung an das Prozessorsystem in der Lage sind, den Datenhunger der Architektur zu stillen, sodass keine Totzeiten entstehen, weil das VCGRA wieder auf Daten wartet. Es war dem Autor bis zum Beenden der Doktorarbeit nicht mehr möglich, diese Erweiterungen zu implementieren und zu testen.

4 Umsetzung der erweiterten Systemarchitektur

Die Verwendung von Field Programmable Gate Arrays und die Beschreibung der Architektur mit VHDL hat den Vorteil, dass man Konzepte auf einer programmierbaren Hardware zunächst testen und die Beschreibungen dann in einem ASIC-Design direkt wiederverwenden kann. Dennoch ist die Zeit für die Berechnung der Bitströme zum Programmieren der programmierbaren Hardware sehr lang im Vergleich zu einem Kompiliervorgang. Auch wächst die Zeit für die Generierung eines Bitstromes nicht linear mit der Komplexität des Designs und kleine Veränderungen im bestehenden Design erfordern häufig eine langwierige Regenerierung des kompletten Bitstromes für den FPGA. Dies schränkt eine schnelle Entwicklung während der Explorationsphase einer Architektur stark ein. Aus diesem Grund bietet sich die Verwendung von „*SystemC*“ [36] an. In diesem Kapitel wird nach einer kurzen Beschreibung von *SystemC*, dessen Simulationszyklus sowie die verwendeten Prozessmethoden auf die Umsetzung der Architektur im Detail eingegangen.

4.1 Register-Transfer-Level-Beschreibung mit SystemC

Es handelt sich bei *SystemC* nicht um eine eigene Programmiersprache, sondern um eine Klassenbibliothek in der Programmiersprache *C++*. Für die Beschreibung von Hardware auf Register-Transfer-Ebene (RT-Ebene) gibt es neben der bereits verwendeten Sprache *VHDL* auch noch *Verilog*. Sie sind heute für die Programmierung von Field Programmable Gate Arrays vorherrschend. Auf RT-Ebene werden Hardwarekomponenten durch Strukturbeschreibungen wie Signale und Ports verschaltet. Die Beschreibung der Datenverarbeitung findet mit Hilfe von höheren Programmiersprachenkonstrukten wie z.B. Entscheidungen oder Schleifen statt. Simuliert werden auf RT-Ebene Zeitpunkte, zu denen Register oder andere Komponenten Werte übernehmen bzw. ändern. Ein RT-Modell ist daher taktzyklengenau[11]. „Da man zunehmend dazu überging, Software und Hardware parallel zu entwickeln, kam der Wunsch auf, möglichst nur eine Sprache sowohl für die Hardware- als auch für die Softwareentwicklung benutzen zu können“ [37]. Als Programmiersprache für Betriebssysteme und hardwarenahe Programmierung wird auch heute noch vorwiegend C/C++ eingesetzt. Deshalb wurde eine Klassenbibliothek zur Hardwarebeschreibung in C++ entwickelt.

[11] Ein Modell auf dieser Ebene benötigt genauso viele Zyklen wie eine reale Hardware, welche das Modell beschreibt.

Der Fokus der Klassenbibliothek lag zunächst in der Beschreibung von Hardware auf ähnliche Weise wie in den bereits etablierten Hardwarebeschreibungssprachen. Aus diesem Grund ist auch eine Untermenge an *SystemC*-Beschreibungsmechanismen [38] erlaubt, die sich mit modernen Synthesewerkzeugen in passende Bitströme für Field Programmable Gate Arrays übersetzen lassen [39].

Die Implementierung des *Virtual Coarse Grained Reconfigurable Arrays* in *SystemC* in dieser Arbeit orientiert sich an der bereits bestehenden Implementierung in VHDL, um die Vergleichbarkeit mit den bestehenden Ergebnissen zu verbessern. Der entstandene Code ist trotz Modellierung auf Register-Transfer-Ebene aber nicht direkt synthetisierbar durch entsprechende Entwicklungswerkzeuge auf einen FPGA wie beispielsweise Xilinx Vivado® HLS. Dies liegt vor allem an der Verwendung von *Templates* für die eigenen Klassen zur Beschreibung der Architekturelemente, um die Evaluation von verschiedenen VCGRA Instanzen zu beschleunigen. Die Synthesewerkzeuge erlauben nur statische Konstrukte mit festen Größen zum Übersetzungszeitpunkt und können solche *Template*konstrukte nicht interpretieren. Das bedeutet, dass die *Template*-Parameter durch feste Werte im Code ersetzt werden müssen. Details zu weiteren Einschränkungen müssen dem verwendeten Synthesewerkzeug entnommen werden.

Die Komplexitätszunahme beim Entwurf von Hardware-Software-Lösungen und der Anzahl an Transistoren auf einem *SoC* („*System on Chip*“) machte eine Abstrahierung notwendig. Um das Jahr 2000 wurde daher eine Modellierungstechnik eingeführt, welche als „*Transaction-Level-Modeling*“, kurz *TLM* bezeichnet wird. Zur Steigerung der Produktivität und zur Schließung der Entwurfslücke sollten schnelle virtuelle Prototypen beschrieben werden können, die einen parallelen Entwurf von Hard- und Software ermöglichen. So können durch abstraktere Strukturen Architekturänderungen schneller durchgeführt und ihre Beziehungen zueinander bewertet werden. Aus diesen Gründen ist auch seit der Version des Standards von *SystemC* aus dem Jahre 2012 die TLM-Bibliothek ein fester Bestandteil. TLM unterstützt eine Trennung von Kommunikation und Datenverarbeitung beteiligter Teilsysteme. Sie beschreibt Systeme grober durch Verbindung von Modulen über Bussysteme. Dabei wird auf eine genaue Definition der Bussignale verzichtet und mit Hilfe von abstrakten Kanälen eine Verbindung hergestellt. Die Datenverarbeitung innerhalb verbundener Module erfolgt durch Softwarefunktionen ebenfalls in C/C++. Für eine Vergleichbarkeit mit der bestehenden Architektur in VHDL aus Abschnitt 2.3 wird auf die Modellierung auf *Transaction Level* verzichtet. Vielmehr werden die Beschreibungen auf der RT-Ebene genutzt, um möglichst nah an der VHDL

Implementierung zu bleiben und trotzdem den Evaluationszyklus zu beschleunigen. Als Zukunftsziel ist ein synthetisierbarer Code angestrebt, beispielsweise mittels eines Codegenerators, der aus den *Templates* statischen Code erzeugen kann.

Die Bestandteile für die Hardwarebeschreibung in *SystemC* bilden die folgenden Module:

- Ein Simulator, welcher in die Modelle mit einkompiliert wird. Damit ist keine eigene Simulationsumgebung notwendig.
- Die Modellierung paralleler Prozesse durch *Methods*, *Threads* und *Clocked-Threads*.
- Flexible Datentypen mit einstellbarer Bitbreite oder Mehrwertigkeit der Logik.
- *Ports* und *Sockets* zur Implementierung von Schnittstellen zur Verbindung von Modulen.
- *Signale* und spezielle Signaltypen wie Puffer, *Fifo*, *Mutex* und *Semaphore* für die Modellierung auf Register-Transfer-Ebene.
- Busprotokoll und Transaktionsobjekt für die *Transaction Level* Modellierung.

4.1.1 Beschreibung des SystemC Simulators

Den Kern von *SystemC* bildet der Simulator. Es handelt sich um eine diskrete, ereignisgesteuerte Simulation. Das bedeutet, dass nicht jeder Modellzeitpunkt, sondern nur Zeitpunkte in denen sich im Modell etwas ändert auch simuliert, sprich berechnet werden müssen. Dies reduziert die Simulationskomplexität. Man muss sich ebenfalls vergegenwärtigen, dass es nur einen Simulator für das Modell gibt. Parallele Prozesse können also nur sequenziell berechnet werden, sodass eine zeitliche Parallelität nur simuliert wird. Eine Simulation startet immer mit dem Aufbau des Modells in der *Elaborationsphase*. Die Konstruktoren der Module werden aufgerufen und die Signalverbindungen vorgenommen. Mit dem Aufruf der statischen Methode `sc_start()` werden in der *Initialisierungsphase* die Prozesse beim Simulator angemeldet und als „ausführbar“, markiert.

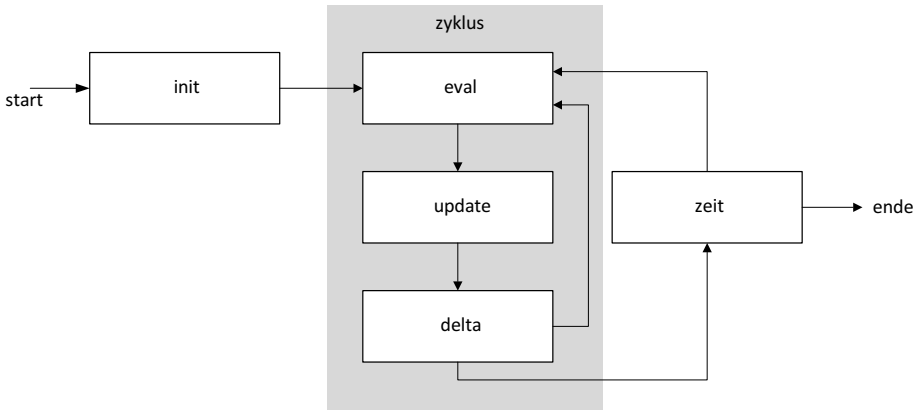

Abb. 47: Simulationszyklus in SystemC

Abb. 47 nach [37] auf Seite 135 Abb. 5.2: „Ablaufdiagramm für den *SystemC*-Schedular“ zeigt den Ablauf des Simulationsalgorithmus mit Request-Update-Mechanismus“. In der *Evaluatephase* (`eval`) werden alle als ausführbar markierten Prozesse in beliebiger Reihenfolge simuliert. Ein laufender Prozess kann unterbrochen werden oder er läuft bis zum Prozessende durch. Da es sich bei Prozessen um Methoden einer C++-Klasse handelt, bedeutet das Erreichen des Prozessendes ein Erreichen des Funktionsendes der Methode. Die Unterbrechung wird durch Aufruf einer globalen Funktion vom Prozess selbst gesteuert und kann nicht vom Simulator durchgeführt werden. Bei Verwendung sogenannter „primitiver Kanäle“ in der RT-Ebene werden Signaländerungen mit einem „*Update-Request*“ versehen. Bei einem Kontextwechsel wird dem gerade bearbeiteten Prozess dann schlussendlich die Markierung „ausführbar“ entfernt. Ein Kontextwechsel tritt ein, wenn das Prozessende erreicht oder die Methode zur Suspendierung des laufenden Prozesses in diesem aufgerufen wird. In der *Updatephase* (`update`) werden die Signalzuweisungen vorgenommen, welche in der *Evaluatephase* mit einem *Update-Request* versehen worden sind. Prozesse, die Wertänderungen dieser Signalzuweisungen beobachten, werden als ausführbar markiert und in der nächsten *Evaluatephase* berechnet.

Reaktionen zur selben Simulationszeit werden in der „*Delta-Notification-Phase*“ (`delta`) eingeleitet. Dies bedeutet, dass in dieser Phase kein weiterer Zeitvorsprung erfolgt. Diese Tatsache muss man sich bei der Modellierung bewusst machen. Zum einen gibt es eine Trennung zwischen Rechenzeit und Simulationszeit. Zum anderen gibt es eine Unterscheidung in der *Evaluatephase* von aktuellem Wert und neuem Wert. Letzterer wird gültig in der nächsten *Evaluatephase*, sodass in Prozessen innerhalb eines Delta-Zyklus immer mit Werten aus dem vorhergehenden Zyklus gearbeitet wird. Werden keine ausführbaren Prozesse mehr mit einer *Delta-Notification* gefunden, wird auf den nächsten Zeitpunkt vorangeschritten, an dem ein

Prozessereignis stattfindet. Diese „*Time-Notification-Phase*“ ist in der Abb. 47 als `zeit` bezeichnet. Ein Fortgang der Simulationszeit findet nur statt, wenn solche *Time-Notifications* implementiert sind. Es werden also alle Prozesse an dem Zeitpunkt berechnet, an dem sie ein Ereignis aufweisen, also für diesen Simulationszeitpunkt als ausführbar markiert sind. Die Simulation wird beendet, wenn:

- Eine vorher angegebene Simulationszeit abgelaufen ist: `sc_start(100.0, sc_core::SC_NS)`
- Die statische Methode `sc_stop()` aufgerufen wurde.
- Keine weiteren ausführbaren Prozesse, weder in der *Time-* noch in der *Delta-Notification*, gefunden werden können.

4.1.2 SystemC Module

Das Grundelement für die Beschreibung von Hardwarestrukturen in *SystemC* ist die Basisklasse `sc_core::sc_module`. Objekte, welche von dieser Basisklasse direkt bzw. indirekt erben, dürfen **nicht** während der Simulation dynamisch erzeugt werden. Außerdem muss es **genau einen** Konstruktorparameter vom Typ `sc_core::sc_module_name` geben. Soll ein Modul nicht nur Strukturbeschreibung sein, sondern auch Prozesse am Simulator anmelden können, muss das Makro `SC_HAS_PROCESS(myModuleClass);` in der Definition, vorzugsweise vor den Konstruktoren der Klasse, stehen. Im Konstruktor der eigenen Klasse können dann Methoden dieser Klasse, über die im kommenden Abschnitt beschriebenen Makros für *Method*s und *Thread*s zur Anmeldung vorgesehen werden. Der Nachteil der Verwendung dieser Makros liegt darin, dass die benutzten Methoden **keine** Rückgabewerte und Parameter besitzen dürfen. Soll dies der Fall sein, so muss man mit `sc_spawn` eine Anmeldung selbstständig durchführen. Auch geben die Makros keine sogenannten „*ProzessHandles*“ zurück.

4.1.2.1 Nebenläufigkeit der Prozesse

Zur Modellierung von Nebenläufigkeit stellt *SystemC* verschiedene Möglichkeiten zur Verfügung. Man unterscheidet zwischen dynamischen und statischen Prozessen, wobei sich diese nur im Zeitpunkt ihrer Anmeldung beim Simulator unterscheiden. In gleicher Weise unterscheidet man dynamische und statische Sensitivität für die Ereignissteuerung. Als Prozessarten werden *Method*, *Thread* und *Clocked-Thread* bereitgestellt. Da *Clocked-Thread*s den *Thread*s ähneln, werden diese nicht näher beschrieben. *Clocked-Thread*s sind nur sensitiv auf ein Signal, welches direkt beim Anmelden des *Thread*s beim Simulator festgelegt wird. Es handelt sich also direkt um einen weiteren Parameter für das Makro „`SC_CTHREAD`“.

4.1.2.2 Method

*Method*s werden immer vollständig, also bis zum Ende der schließenden Klammer (}) der Funktion durchlaufen und können **nicht** suspendiert werden. Daher dürfen sie nicht in Endlosschleifen verwendet werden, da ansonsten die Kontrolle nicht zum Simulator zurückkehrt. Die Anmeldung beim Schedular des Simulators erfolgt im Konstruktor des betreffenden Moduls mit folgendem Makro:

```
SC_METHOD(name_Memberfunction);
sensitive << signal << ...;
```

Code 9: Registrieren einer Modul-Methode im SystemC-Simulationsscheduler

Die nach dem Makro angegebene „*Sensivitätsliste*“ beschreibt die Signale des Moduls, welche zu einer Ausführung der *Method* führen. Die Ausführung des Prozesses lässt sich zusätzlich dynamisch durch die Methode `next_trigger` steuern, jedoch nicht unterbrechen. Es ist darüber hinaus möglich, *Method*s dynamisch während der Laufzeit beim Simulator anzumelden. *Method*s kommen vorwiegend in der Modellierung der Register-Transfer-Ebene zum Einsatz und werden daher für die Implementierung dieser Gesamtarchitektur und deren Komponenten verwendet. Zur Simulation der Gesamtarchitektur mit einer Testbench kommen *Thread*s zum Einsatz, weshalb auf diese im folgenden Abschnitt auch noch einmal kurz eingegangen wird.

4.1.2.3 Thread

Ein *Thread* wird nach der Anmeldung beim Simulator, entweder zu Beginn statisch in der Ebalorationsphase oder dynamisch während der Evaluationsphase der Simulation, nur ein einziges Mal gestartet. Allerdings lassen sich *Thread*s während der Ausführung durch `wait()` suspendieren. Eine Suspendierung bedeutet, dass der Simulator einen Kontextwechsel durchführt und der *Thread* so lange pausiert, bis ein Ereignis diesen wieder weiterführt. *Thread*s können statische Sensitivität über eine Sensitivitätsliste vergleichbar zu *Method*s aufweisen, es kann der `wait()`-Methode ebenfalls ein Ereignis, eine Ereignisliste oder eine Wartezeit übergeben werden.

Die Anmeldung mit statischer Sensitivität erfolgt analog zu *Method* mit:

```
SC_THREAD (name_Memberfunction);
sensitive << signal << ...;
```

Code 10: Registrieren einer Thread-Methode im SystemC-Simulationsscheduler

Im Unterschied zur *Method* muss der Simulator bei einem Kontextwechsel den Zustand des Prozesses abspeichern. Wenn das Ereignis zur Fortführung des *Thread*s eintritt, wird die Simulation vom letzten `wait()`-Aufruf bis zum nächsten oder bis zum Ende der geschlossenen Klammer (}) fortgesetzt. Ein einmal vollständig beendeter *Thread* kann während der Simulation nicht erneut gestartet werden. Wie bereits erwähnt finden *Thread*s in dieser Arbeit als Testbenches für Architekturkomponenten oder der Gesamtarchitektur Anwendung. Sie werden außerdem verstärkt in der *Transaction Level* Simulation verwendet.

4.2 Pre-Fetcher und Datenpuffer

In diesem Abschnitt wird auf die Implementierungsdetails der unter Abschnitt 3 eingeführten Verbesserungsvorschläge der Architektur eingegangen. Zur besseren Illustration sind Diagramme und Schemata verwendet, an denen die Elemente näher beschrieben werden. Diese Schemata dienen vorwiegend der Übersicht und haben keinen Anspruch auf Vollständigkeit. So sind ggf. unwesentliche Signale innerhalb einer Komponente zum Zwecke der Übersichtlichkeit weggelassen oder vereinfacht dargestellt. Sie dienen daher nicht als Blaupause für eine Nachimplementierung in einer anderen Hardwarebeschreibungssprache.

Es wird zunächst auf die Puffer für Daten und Konfigurationen eingegangen. Diese dienen wie in Abschnitt 3 beschrieben dem Zweck, die aktive Wartezeit auf Daten oder Konfigurationen des eingebetteten *Virtual Coarse Grained Reconfigurable Arrays* in der Architektur zu verkürzen oder gar zu vermeiden. Dafür werden Nutzdaten und Konfigurationen in den Puffern parallel zur Verarbeitung des *Virtual Coarse Grained Reconfigurable Arrays* vorgeladen. Anschließend wird auf die *Memory Management Unit* eingegangen, welche den Datenzugriff auf das *Shared Memory* zwischen Prozessorsystem und VCGRA entkoppelt, um damit das PS für andere Prozessschritte zu entlasten. Zuletzt folgt die Beschreibung der *Management Unit* oder *Central Control Unit*, welche das VCGRA noch weiter vom Prozessorsystem entkoppelt und autark Daten verarbeiten lässt.

4.2.1 Data-Input Pre-Fetcher

Diese Komponente ist als *Template*-Klasse erstellt. Die Verwendung eines *Template* hat den Vorteil, dass die Eigenschaften des Moduls für eine Simulation flexibel über *Template*-Parameter angepasst werden können, um so die Evaluation einer Architektur einfacher zu gestalten. Die Parameter des *Template* sind in der Datei „*Typedef.h*“ abstrahiert und einstellbar.

```
static constexpr uint16_t cDataValueBitwidth{16};
//!< \brief Number of bits for one data value
static constexpr uint16_t cNumberOfValuesPerCacheLine{8};
//!< \brief Number of accessible data values in a cache line
static constexpr uint16_t cNumberDataInCacheLines{2}
```

Code 11: Abstraktionsparameter für die Data-Input Pre-Fetcher Instanz

Die *Template*-Parameter konfigurieren den *Pre-Fetcher* in der Bitbreite für einen Wert innerhalb des *Pre-Fetcher*, der Anzahl der verfügbaren Zeilen (*Lines*) und der Anzahl der Plätze (*Places*) pro Zeile. Die Höchstgrenzen für die Anzahl der Zeilen und Plätze wird durch das Maschinenwort beschränkt wie in Abschnitt 3.3.1 beschrieben.

In Headerfile des *Data-Input Pre-Fetcher* ist mittels der Abstraktionsparameter die Instanz des *Pre-Fetcher* für eine Architektur als Typendefinition wie in Code 12 deklariert.

```
//Definition of input data pre-fetcher type
typedef DataInCache<cgra::cDataValueBitwidth,
    cgra::cNumberOfValuesPerCacheLine,
    cgra::cNumberDataInCacheLines
    > data_input_cache_type_t;
```

Code 12: Beschreibung Data-Input-Pre-Fetcher Instanz

Die Schnittstellen der Architekturkomponente sind in Abb. 48 aufgeführt. Die Komponente arbeitet taktgesteuert. Steuersignale und Daten müssen mindestens für einen Takt an den Eingangsports anstehen.

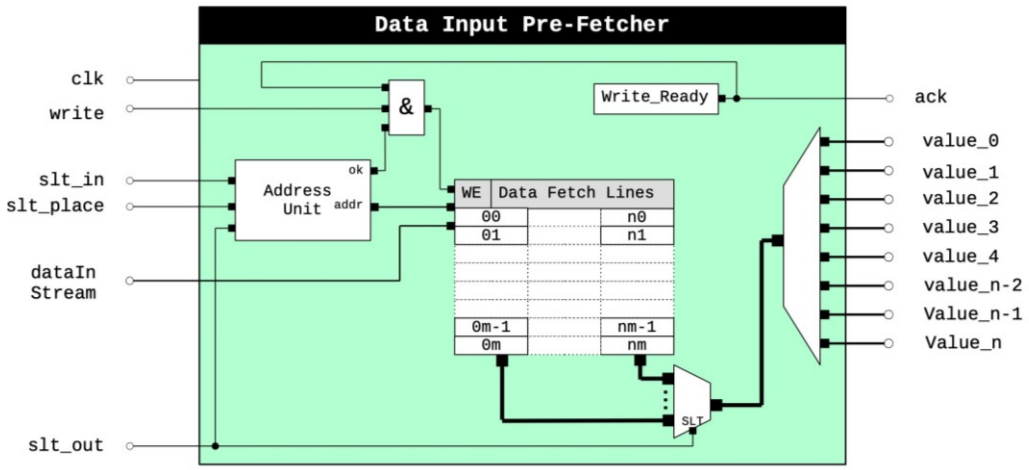

Abb. 48: Schematische Darstellung – Data-Input Pre-Fetcher

Zwei Hauptkomponenten bilden das Herzstück des *Data-Input Pre-Fetcher*: Der Datenspeicher implementiert als ein 2D Array und eine „*Address Unit*“. Die Daten sind als zweidimensionales Array abgelegt, weil über die Ports „`slt_in`“ und „`slt_place`“ eine genaue Adressierung innerhalb des Arrays stattfinden kann. Die *Address Unit* kontrolliert weiterhin, ob in die adressierte Zeile geschrieben werden darf. Dies ist nur dann zulässig, wenn die entsprechende Zeile nicht durch „`slt_out`“ gerade als die korrespondierende Konfiguration als Eingangsparameter an das VCGRA gelegt ist.

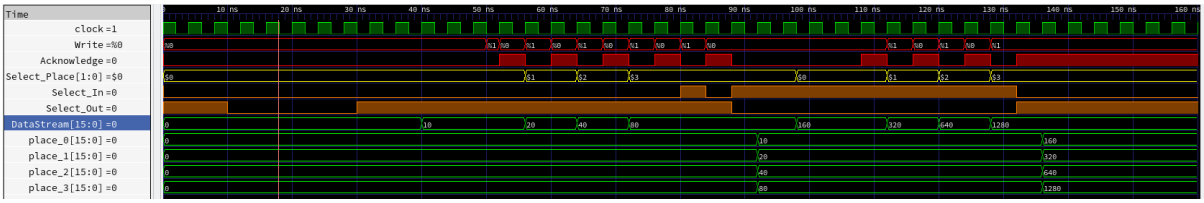

Abb. 49: Timingdiagramm (klein) – Data-Input Pre-Fetcher

Das Timingdiagramm für die Verwendung des *Data-Input Pre-Fetcher* ist in Abb. 49 und noch einmal größer in Abb. 72 im Anhang beigefügt. Zu den folgenden Zeitpunkten der Simulation ist jeweils das Verhalten beschrieben:

- 10ns: Es wird „`slt_out`“ umgeschaltet von *Line* Eins auf *Line* Null. Da beide *Lines* auf „0“ vorinitialisiert sind, ist an den Ausgängen für die Daten keine Änderung sichtbar.
- 20ns: Es wird eine *Line* außerhalb des gültigen Bereiches gewählt. Der Simulator gibt eine Warnung aus, es findet aber keine Änderung der Signale statt. Es lässt sich keine Signaländerung auf „`slt_out`“ erkennen.

- 30ns: „slt_out“ wird auf eine andere *Line* als „slt_in“ gestellt, damit Daten in „slt_in“ gespeichert werden können.
- 40ns: Es werden Nutzdaten an „dataInstream“ angelegt, es findet aber noch keine Datenübernahme statt, da „write“ noch keine steigende Flanke hatte.
- 50ns: Ein Schreiben wird durch eine steigende Flanke am Eingangsport „write“ getriggert. Dabei darf ein laufender Schreibzugriff nicht unterbrochen werden. Aus diesem Grund ist durch ein „acknowledge“ zum Aufrufer der Schreibbefehl synchronisiert. Dafür wird der „ack“-Port so lange auf „*High*“ gelassen, bis das „write“-Signal innerhalb eines Taktes wieder auf „*Low*“ fällt erkennbar durch das Abwechseln von „*High*“ und „*Low*“ des „write“ und „ack“-Signals. Es folgt ein Schreiben von vier Datenwörtern, wobei jeweils der *Place* innerhalb der *Line* angepasst wird. Dies zeigt die gelbe Kurve, die von Null auf Drei hinaufgezählt wird.
- 80ns: Es wird versucht auf die Daten zu schreiben, die gerade als Ausgang aktiv sind. Dies führt zu einer Warnung in der Simulation. Die Daten werden nicht verändert, aber der „write“-Befehl wird mit einen „ack“ bestätigt.
- 90ns: „slt_out“ und „slt_in“ wechseln, wodurch die zuvor übertragenen Daten aktiv an den korrespondierenden Ausgang gelegt werden. Die alte Zeile wird nun mit neuen Daten initialisiert.
- 140ns: Es erfolgt ein erneutes Umschalten der Zeilen und damit ein Aktualisieren der Datenausgänge des *Data-Input Pre-Fetcher*.

Aus dem Beispiel ergeben sich als wichtigste Eigenschaften: Eine Synchronisierung über „write“ und „ack“ erfolgt immer, selbst im Fehlerfall. Es werden dann auf der Konsole zusätzlich Warnungen während der Simulation generiert. Der *Pre-Fetcher* unterstützt nur ein wortweises Schreiben. Das bedeutet, die Bitbreite des „dataInStream“-Ports entspricht der Bitbreite eines Platzes innerhalb einer Zeile des *Pre-Fetcher*. Dies erleichtert die Konsistenz von Daten, da diese immer vollständig geschrieben werden und beschleunigt die Übertragung, da immer genau ein neuer Wert geschrieben wird und nicht nur ein Teil eines Wertes. „slt_out“ bestimmt, welche Zeile des *Pre-Fetcher* gerade an den Ausgang gelegt ist und stellt den dominierende Auswahlport im Vergleich mit dem „slt_in“-Port dar. In der Gesamtarchitektur korrespondieren die Bitbreiten von *Data-Input Pre-Fetcher* und VCGRA-Nutzeingangsdaten miteinander, da jeder Platz innerhalb des *Pre-Fetcher* mit einem Eingang des *Virtual Coarse Grained Reconfigurable Arrays* korrespondiert.

Das *SystemC*-Modul verfügt über zwei Methoden, welche das Speichern neuer Daten in einer Zeile und das Wählen der Zeile für die Ausgänge steuern. Außerdem ist es

möglich, eine Zeile des *Pre-Fetcher* auf der Konsole auszugeben oder den gesamten Status des Moduls zu „dumpen“.

4.2.2 Configuration Pre-Fetcher

Die Architektur besitzt zwei *Pre-Fetcher* für Konfigurationen des *Virtual Coarse Grained Reconfigurable Arrays*, einen für die Konfiguration der Prozesselemente und einen für die Konfigurationen der virtuellen Kanäle inklusive der *Synchronization Unit*. Diese Komponente ist ebenfalls als *Template*-Klasse erstellt, die Parameter des *Templates* sind in der Datei „*Typedef.h*“ abstrahiert und einstellbar.

```
//Properties for PE configuration cache
//--------------------------------------
static constexpr uint16_t cPeConfigBitWidth{48};
//!< \brief Number of bits for whole PE configuration of VCGRA
static constexpr uint16_t cNumberOfPeCacheLines{2};
//!< \brief Number of cache lines for PE configuration cache
static constexpr uint16_t
cSelectLineBitwidthPeConfCache{calc_bitwidth(cNumberOfPeCacheLines)};
//!< \brief Bitwidth to select available cache lines round-
up{log2(cNumberOfCacheLines)}
static constexpr uint16_t
cBitWidthOfSerialInterfacePeConfCache{cDataStreamBitWidthConfCaches};
//!< \brief Bitwidth for serial configuration input stream to configuration
cache

//Properties for virtual channel configuration cache
//-------------------------------------------------
static constexpr uint16_t cVChConfigBitWidth{64};
//!< \brief Number of bits for whole vCh configuration of VCGRA
static constexpr uint16_t cNumberOfVChCacheLines{2};
//!< \brief Number of cache lines for vCh configuration cache
static constexpr uint16_t
cSelectLineBitwidthVChConfCache{calc_bitwidth(cNumberOfVChCacheLines)};
//!< \brief Bitwidth to select available cache lines round-
up{log2(cNumberOfCacheLines)}
static constexpr uint16_t
cBitWidthOfSerialInterfaceVChConfCache{cDataStreamBitWidthConfCaches};
//!< \brief Bitwidth for serial configuration input stream to configuration
cache
```

Code 13: Abstraktionsparameter einer Configuration Pre-Fetcher Instanz

Die *Template*-Parameter konfigurieren einen *Configuration Pre-Fetcher* vergleichbar zum *Data-Input Pre-Fetcher*. Die Bitbreite für die Segmentübertragung kann unterschiedlich gewählt werden zwischen dem PE-Konfiguration *Pre-Fetcher* und dem *Pre-Fetcher* für Konfigurationen der virtuellen Kanäle, wenn die Quelle der

Konfiguration parallele Übertragung von Daten durch seperate Ports zulässt. Zur Reduktion von *Routing*-Ressourcen wird in der Gesamtarchitektur nur ein gemeinsamer Datenstrom-Port an der *Memory Management Unit* verwendet, weshalb für beide *Pre-Fetcher* die gleiche Bitbreite für ein Segment der Konfiguration verwendet wird. Des Weiteren arbeitet die *Central Control Unit* derzeit Befehle nicht parallel ab und auch die MMU kann Befehle nur sequenziell bearbeiten. Aus diesem Grund ist es nicht möglich, eine Konfiguration für ein Prozesselement und eine Konfiguration für einen virtuellen Kanal parallel zu laden. Daher ist ein gemeinsamer Datenstrom-Port an der *Memory Management Unit* ausreichend sowie nur eine gemeinsame Bitbreite auswählbar.

Im Headerfile des *Configuration Pre-Fetcher* ist mittels der Abstraktionsparameter je eine Instanz des *Pre-Fetcher* als Typendefinition wie in Code 14 deklariert.

```
//Definitions of configuration pre-fetcher types
//-----------------------------------------------
typedef ConfigurationCache<sc_dt::sc_lv<cgra::cPeConfigBitWidth>,
    cgra::cSelectLineBitwidthPeConfCache,
    cgra::cNumberOfPeCacheLines,cgra::cSelectLineBitwidthPeConfCache>
pe_config_cache_type_t;
//!< \brief Type definition for Processing_Element configuration cache
typedef ConfigurationCache<sc_dt::sc_lv<cgra::cVChConfigBitWidth>,
    cgra::cSelectLineBitwidthVChConfCache,
    cgra::cNumberOfVChCacheLines,
    cgra::cBitWidthOfSerialInterfaceVChConfCache> ch_config_cache_type_t;
//!< \brief Type definition for VirtualChannel configuration cache
```

Code 14: Beschreibung der Configuration Pre-Fetcher Instanzen

Der Aufbau eines *Configuration Pre-Fetcher* ist in Abb. 50 schematisiert. Er ist vielfach vergleichbar mit dem Aufbau des *Data-Input Pre-Fetcher*. Im Gegensatz zu diesem speichert eine Zeile des *Configuration Pre-Fetcher* aber immer eine gesamte Konfiguration. Aus diesem Grund können auch keine einzelnen Plätze zum Schreiben von Werten angesprochen werden, sondern es wird immer eine gesamte Zeile ausgetauscht. Durch „`slt_out`“ wird stets eine gesamte Konfigurationszeile an die Komponenten des *Virtual Coarse Grained Reconfigurable Arrays* der Gesamtarchitektur geschaltet. Über Port „`slt_in`“ wird die Zeile gewählt, auf die eine neue Konfiguration gespeichert werden soll. Dabei dominiert Port „`slt_out`“ vor Port „`slt_in`“, was bedeutet, dass nur eine Zeile verändert werden kann, die nicht gerade am Ausgang als aktuelle Konfiguration in Verwendung ist. Damit soll verhindert werden, dass eine Konfiguration während der Verarbeitung im VCGRA verändert wird und zu unkontrolliertem Verhalten führt.

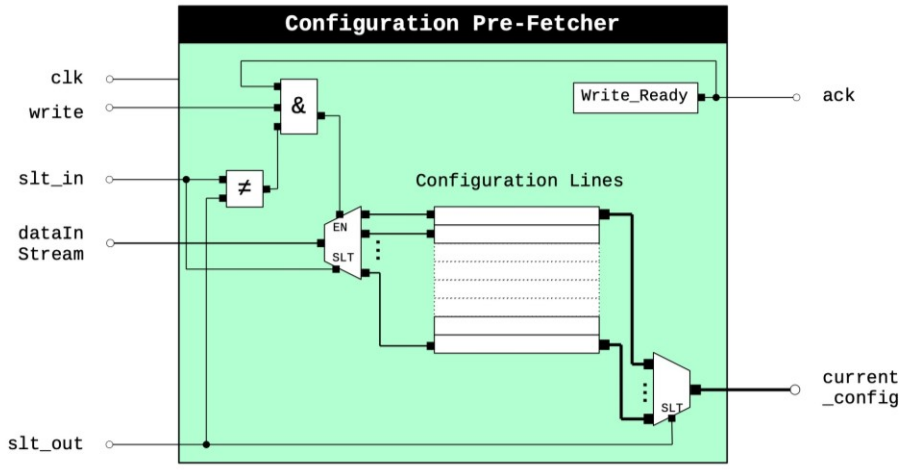

Abb. 50: Schematische Darstellung – Configuration Pre-Fetcher

Der Port „`dataInStream`“ kann eine beliebige Größe haben. Es werden die Konfigurationsdaten im *Pre-Fetcher* aus den einzelnen Segmentabschnitten wieder zu einer Gesamtkonfiguration zusammengesetzt. Dabei kann der *Pre-Fetcher* den Datenstrom nicht auf Korrektheit überprüfen. Die aktuelle Konfiguration in der gewählten Zeile wird rotiert um die Anzahl der Bits entsprechend der Bitbreite von „`dataInStream`“ und das neue Datensegment wird an das Ende der bereits Empfangenen Teilkonfiguration angehängt. Daraus ergeben sich zwei Abhängigkeiten:

- Eine Zeile des *Pre-Fetcher* **muss immer** ein ganzzahliges Vielfaches der Bitbreite des „`dataInStream`“-Ports sein.
- Die Konfiguration **muss immer** mit „*Big-Endianess*“ übertragen werden. Das heißt, eine Übertragung beginnt immer mit der Konfiguration für die am weitesten oben links liegende Komponente (Prozesselement oder Multiplexer im virtuellen Kanal).
- Passt eine Konfiguration nicht in ein Vielfaches der Bitbreite von „`dataInStream`“, wird mit Nullen aufgefüllt. Die angehängten Nullen werden bei der Verteilung der Konfigurationsbits auf die Komponenten des *Virtual Coarse Grained Reconfigurable Arrays* nicht weiter beachtet.

Eine Beschreibung der Funktionsweise anhand eines Timingdiagramms wird im Zusammenhang mit der *Memory Management Unit* angegeben. Diese ist im Dokument unter Abschnitt 4.3 aufzufinden.

4.2.3 Data-Output Puffer

Diese Komponente ist als *Template*-Klasse erstellt, woraus sich ebenfalls für dieses Modul die genannten Vorteile aus Abschnitt 4.2.1 ergeben. Die Parameter des *Template* sind in der Datei „*Typedef.h*“ abstrahiert und einstellbar.

```
static constexpr uint16_t cDataValueBitwidth{16};
//!< \brief Number of bits for one data value
static constexpr uint16_t cNumberOfValuesPerCacheLine{8};
//!< \brief Number of accessible data values in a cache line
static constexpr uint16_t cNumberDataOutCacheLines{2}
```

Code 15: Abstraktionsparameter für eine Data-Output Puffer Instanz

Die *Template*-Parameter konfigurieren den Puffer in der Bitbreite für einen Wert innerhalb des Puffers, der Anzahl der verfügbaren Zeilen (*Lines*) und der Anzahl der Plätze (*Places*) pro Zeile. Die Höchstgrenzen für die Anzahl der Zeilen und Plätze wird durch das Maschinenwort beschränkt wie in Abschnitt 3.3.1 beschrieben.

Im Headerfile des *Data-Output* Puffer ist mittels der Abstraktionsparameter die Instanz des Puffers als Typendefinition wie in Code 16 deklariert.

```
//Definition of output data buffer type
typedef DataOutCache<cgra::cDataValueBitwidth,
    cgra::cNumberOfValuesPerCacheLine,
    cgra::cNumberDataOutCacheLines
    > data_output_cache_type_t;
```

Code 16: Beschreibung einer Data-Output Puffer Instanz

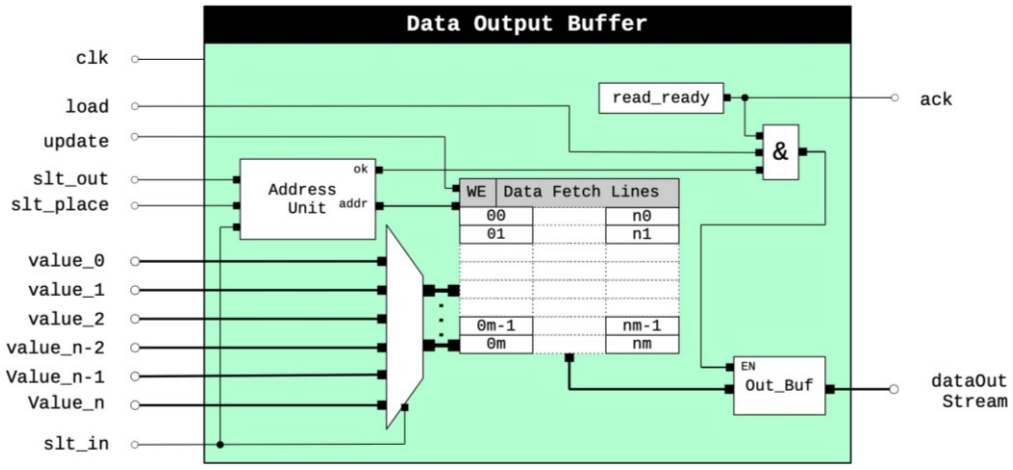

Abb. 51: Schematische Darstellung – Data-Output Puffer

Der grundsächliche Aufbau des Puffers ist in Abb. 51 dargestellt. Er gleicht ebenfalls dem Aufbau des *Data-Input Pre-Fetcher*, allerdings werden mehrere Inputs in einem zweidimensionalen Datenspeicher abgelegt. Jeder Eingang korrespondiert mit jeweils einem Platz innerhalb einer Zeile. Der Eingangsport „`update`“ steuert die Übernahme aller Zustände an Port „`value_0`“ bis „`value_n`“ in der gerade selektierten Zeile durch den Wert an Port „`slt_in`“. Die Adressierung innerhalb des Arrays erfolgt mittels Port „`slt_out`“ und Port „`slt_place`“. Die *Address Unit* kontrolliert, ob die adressierte Zeile gelesen werden darf. Dies ist nur dann zulässig, wenn die entsprechende Zeile nicht durch Port „`slt_in`“ gerade als Eingangsparameter an das VCGRA gelegt ist und ein Update dieser Zeile abläuft.

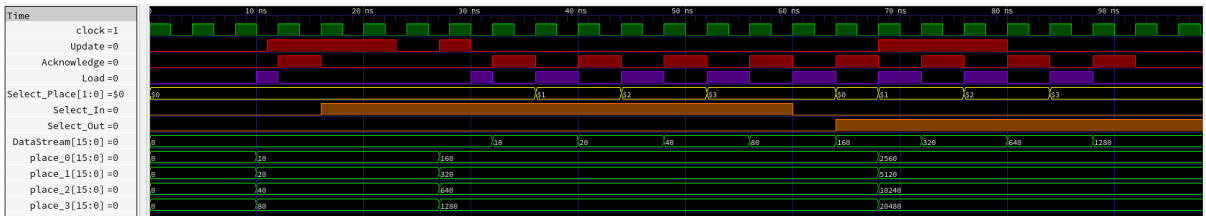

Abb. 52: Timingdiagramm (klein) – Data-Output Puffer

Das Timingdiagramm für die Verwendung des *Data-Output* Puffers ist in Abb. 51 und noch einmal größer in Abb. 73 im Anhang dargestellt. Zu den folgenden Zeitpunkten der Simulation ist jeweils das Verhalten beschrieben:

- 10ns: Es wird der Datentransport über ein externes Triggersignal an Port „`load`“ angefordert.
- 11ns: Es wird extern ein Update (Port „`update`“) der Pufferzeile angefordert. Dieses Update ist höher priorisiert als der Datentransport, sodass der Datentransport unterbrochen und das Update der Pufferzeile vollständig

beendet wird. Das Acknowledge bleibt aktiv für einen Zyklus, um den Aufrufer für den Datentransport zu synchronisieren. Man kann ebenfalls erkennen, dass ein Update asynchron stattfinden kann, denn 11ns liegt außerhalb des Taktes von 2ns. Die Werte, die an Port „value_0“ („*place_0*“) bis Port „value_n“ („*place_3*“) anstehen, werden in Zeile Null gespeichert.

- 16ns: Es soll ein Wechsel der Pufferzeile für eingehende Daten stattfinden. Da das „update“-Signal aber noch aktiv ist, findet der Wechsel nicht statt. Es wird eine Warnung ausgegeben, denn ein Wechsel einer Pufferzeile kann nur stattfinden, wenn gerade kein Update oder ein Datentransfer in Bearbeitung ist.
- 23ns: Das Update der Werte im Puffer ist beendet. Die Pufferzeile für eingehende Werte wird auf Eins gewechselt.
- 27ns: Es wird extern, asynchron ein neues Update von Eingangswerten getriggert. Dabei werden diesmal die Werte die an Port „value_0“ („*place_0*“) bis Port „value_n“ („*place_3*“) anstehen auf Zeile Eins geschrieben. Zeile Null bleibt unverändert.
- 30ns: Es wird der Datentransport über ein externes Triggersignal an Port „load“ angefordert. Nacheinander werden die Werte nun über Port „dataOutStream“ („*dataStream*“) an den Empfänger übertragen. Mit Ackknowledge wird der Versand jedes Wertes beim Empfänger angekündigt. Welcher Wert angefragt wird durch einen Trigger an Port „load“, wird mittels dem Wert an Port „slt_place“ („*Select_Place*“) ebenfalls vom Empfänger gesteuert.
- 60ns: Es sind alle Werte des Puffers aus Zeile Null übertragen. Nun wird ein „Swap“ (Wechsel) der Pufferzeilen durchgeführt. Es wird der Datentransport über ein externes Triggersignal an Port „load“ erneut angefordert. Nacheinander werden die Werte jetzt über den Port „dataOutStream“ („*dataStream*“) an den Empfänger übertragen. Mit Ackknowledge wird der Versand jedes Wertes beim Empfänger angekündigt. Welcher Wert durch einen Trigger an Port „load“ angefragt wird, wird durch den Wert an Port „slt_place“ („*Select_Place*“) ebenfalls vom Empfänger gesteuert.
- 68ns: Parallel zur laufenden Übertragung findet ein Trigger für das Update der Werte der Eingangszeile des Puffers statt. Da diese gerade nicht für eine Datenübertragung genutzt wird, werden die Eingangsdaten in die Zeile übernommen, sodass die Daten nicht verloren gehen.

Das Beispieldiagramm zeigt die Vorteile des *Data-Output* Puffers. Datentransfers lassen sich zeitlich entspannen, da trotz laufender Übertragung neue Daten in eine Pufferzeile geschrieben werden können. Daneben findet die Übernahme der Daten in den Puffer asynchron statt. Damit können Daten schon übernommen werden, bevor synchron ein neuer Datentransfer zum Empfänger angeregt wird oder ein Wechsel der Pufferzeile stattfindet. Ein Update der Pufferdaten einer Zeile dominiert immer das Verhalten des Puffers. So wird eine laufende Datenübertragung gegebenenfalls unterbrochen, wenn ein externes Update angetriggert wurde, damit keine Nutzdaten aus dem VCGRA verloren gehen.

Eine Synchronisierung per Acknowledge erfolgt immer, selbst im Fehlerfall. Es werden dann während der Simulation auf der Konsole zusätzliche Warnungen generiert. Der Puffer unterstützt nur ein wortweises Lesen. Das bedeutet, die Bitbreite des „`dataOutStream`“-Ports entspricht der Bitbreite eines Platzes innerhalb einer Zeile des Puffers. Dies erleichtert die Konsistenz von Daten, da diese immer vollständig geschrieben werden und beschleunigt die Übertragung, da immer genau ein neuer Wert geschrieben wird und nicht nur ein Teil eines Wertes. Port „`slt_out`“ bestimmt, welche Zeile des Puffers gerade an den Ausgang gelegt ist und ist der dominierende Auswahlport im Vergleich mit dem „`slt_in`“-Port. In der Gesamtarchitektur korrespondieren die Bitbreiten von *Data-Output* Puffer und VCGRA-Nutzausgangsdaten miteinander, da jeder Platz innerhalb des Puffers mit einem Ausgang des *Virtual Coarse Grained Reconfigurable Arrays* korrespondiert.

Das *SystemC*-Modul verfügt über drei Methoden, welche das Lesen/Aktualisieren neuer Daten in einer Zeile und das Wählen der Zeile für den Eingang neuer Daten aus dem VCGRA steuern. Außerdem ist es möglich, eine Zeile des Puffers auf der Konsole auszugeben oder den gesamten Status des Moduls zu „dumpen“.

4.3 Memory Management Unit

Die *Memory Management Unit* unterstützt dem Zugriff auf den „*Shared Memory*“, welcher zum Datenaustausch zwischen Prozessorsystem und VCGRA-Architektur dient. Des Weiteren verteilt es die Nutzdaten und Konfigurationen auf die *Pre-Fetcher* und lädt die Ergebnisse aus dem *Data-Output* Puffer. Die MMU arbeitet getaktet nach einer State Machine wie sie in Abb. 53 dargestellt ist.

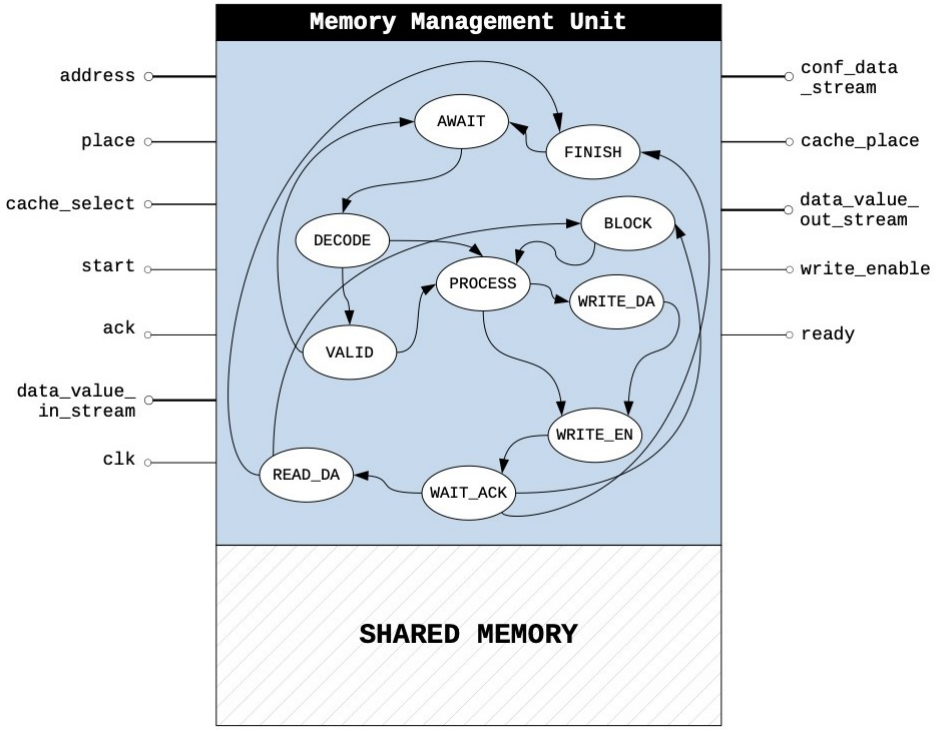

Abb. 53: Schematische Darstellung – Memory Management Unit

4.3.1 Konfiguration

Dieses Modul ist nicht als *Template* konzipiert, dennoch ist seine Konfiguration über die Datei „*Typedef.h*“ für den Anwender abstrahiert. Die Konfiguration der *Memory Management Unit* passend zu den verwendeten *Pre-Fetcher*n und Puffern wird dem Konstruktor als „*Initializer List*“ übergeben. Die Konfiguration der Initializer List ist in Code 17 abgebildet.

In der Liste werden zwölf Einträge erwartet, jeweils drei Einträge pro *Pre-Fetcher* beziehungsweise Puffer. Die Reihenfolge der Einträge ist passend zu den Komponenten *Data-Input Pre-Fetcher*, *Data-Output* Puffer, PE-Konfiguration *Pre-Fetcher* und *Channel*-Konfiguration *Pre-Fetcher* angeordnet. Die Einträge für jede Komponente sind unterteilt in:

- Länge einer Zeile einer Komponente (*Pre-Fetcher*, Puffer) in der Anzahl der Bytes,
- Anzahl der Zeilen einer Komponente und
- Bitbreite für ein Datum beispielsweise einen Wert oder einer Konfiguration für die Prozesselemente oder die virtuellen Kanäle.

Für die Länge einer Zeile einer Komponente wurde nicht die Anzahl der Plätze, sondern die Anzahl der Bytes verwendet, da es Plätze für die *Configuration Pre-Fetcher* nicht gibt. Hier entspricht eine Zeile einer Konfiguration.

```
//MMU initializer list
static constexpr std::initializer_list<uint16_t> cCacheFeatures{
    static_cast<uint16_t>(cgra::calc_numOfBytes(cgra::cDataValueBitwidth * 2
* cgra::cPeLevels.front())), cgra::cNumberDataInCacheLines,
cgra::cDataValueBitwidth,
    static_cast<uint16_t>(cgra::calc_numOfBytes(cgra::cDataValueBitwidth *
cgra::cPeLevels.back())), cgra::cNumberDataOutCacheLines,
cgra::cDataValueBitwidth,
    static_cast<uint16_t>(cgra::calc_numOfBytes(cgra::cPeConfigBitWidth)),
cgra::cNumberOfPeCacheLines, cgra::cPeConfigBitWidth,
    static_cast<uint16_t>(cgra::calc_numOfBytes(cgra::cVChConfigBitWidth)),
cgra::cNumberOfVChCacheLines, cgra::cVChConfigBitWidth
};
```

Code 17: MMU-Initialisierungsliste zum Einstellen von Pre-Fetcher- und Puffereigenschaften

Über den letzten Eintrag der Bitbreite kann aber wieder auf die Anzahl der Plätze rückgeschlossen werden, da die Anzahl der Plätze mit der Anzahl der Ein- bzw. Ausgänge eines VCGRA korrespondiert und damit ein Platz der Bitbreite eines Datums entspricht. Die *Memory Management Unit* entscheidet selbst, ob sie in Plätze umrechnen muss oder nicht.

Für den *Data-Input Pre-Fetcher* und den *Data-Output* Puffer wird jeweils die Anzahl der Plätze anhand der VCGRA-Konfiguration für die Prozesselemente abgeleitet. Der erste Eintrag der PE-Konfiguration „`cPeLevels`“ entspricht der Anzahl der Prozesselemente am Eingang des *Virtual Coarse Grained Reconfigurable Arrays*, der letzte Eintrag der Anzahl der Prozesselemente am Ausgang. Da eine PE zwei Eingänge besitzt, ergibt sich die Anzahl der Plätze im *Data-Input Pre-Fetcher* aus der doppelten Anzahl der Prozesselemente am Eingang. Ein Eintrag in „`cPeLevels`“ entspricht der Anzahl der Prozesselemente in der korrespondierenden Zeile (Level) des *Virtual Coarse Grained Reconfigurable Arrays*. Der Index des Eintrags in aufsteigender Reihenfolge gehört zum passenden Level von Eingang zum Ausgang im VCGRA.

```
//MMU Configuration Parameter
static constexpr uint16_t cDataStreamBitWidthConfCaches{8};
static constexpr uint16_t cMemorySize{UINT16_MAX};
static constexpr uint16_t cDataValueBitwidth{cPeDataBitwidth};
static constexpr uint16_t
cMaxNumberOfValuesPerCacheLine{static_cast<uint16_t>(
    (2*cgra::cPeLevels.front()) >= cgra::cPeLevels.back() ? 2 *
cgra::cPeLevels.front() : cPeLevels.back())};
static constexpr uint16_t cNumberDataInCacheLines{2};
static constexpr uint16_t cNumberDataOutCacheLines{2};
```

Code 18: Abstraktionsparameter einer Memory Management Unit Instanz

Die weiteren Abstraktionsparameter sind in Code 18 aufgelistet:

- `cDataStreamBitWidthConfCaches`: Die Anbindung der *Pre-Fetcher* für die Konfiguration der Prozesselemente und virtuellen Kanäle ist geteilt. Beide nutzen zur Übertragung diese Bitbreite für die Datenströme.
- `cMemorySize`: Beschreibt die Größe des adressierbaren Speichers des *Shared Memory*. Der Adressbereich orientiert sich zum einem an dem Maschinenwort (siehe auch Abschnitt 3.3.1), der sechzehn Bit Speicher adressieren lässt und der Anzahl der Bytes, die je Adresse auf einmal adressiert werden. In der Grundkonfiguration entspricht dies je ein Byte, weshalb sich 65536 Byte Speicher gleich `UINT16_MAX` verwenden lassen.
- `cDataValueBitwidth`: Dies ist eine Verallgemeinerung für die Datenbreite der Prozesselemente innerhalb des *Virtual Coarse Grained Reconfigurable Arrays*. Sein Design lässt es zu, jeder PE darin eine eigene Bitbreite ihrer Datenports zu geben. Ohne Autogenerator der VCGRA Instanz und der restlichen Architektur erschwert dies zunehmend den Entwurf. Aus diesem Grund ist die Bitbreite aller Prozesselemente gleich. Außerdem wird die gesamte Architektur statisch in C++ allokiert, es werden also keine expliziten Aufrufe von `new` und die Verwaltung über *Pointer* verwendet. Dafür werden unter anderem Arrays verwendet, weshalb die „Datentypen“ der Elemente wie beispielsweise der Prozesselemente identisch sein müssen.
- `cMaxNumbersOfValuesPerCacheLine`: Das Signal zur Auswahl des Platzes in einer Zeile eines *Pre-Fetcher* oder Puffers ist geteilt zwischen *Data-Input Pre-Fetcher* und *Data-Output* Puffer. Daher wird an dieser Stelle geschaut, welcher der beiden Komponenten mehr Plätze besitzt und adaptiert damit die Anzahl der adressierbaren Plätze.

- `cNumberOfData[Out,In]CacheLines`: Anzahl der Lines in *Data-Output* Puffer und *Data-Input Pre-Fetcher*.

4.3.2 Schnittstelle zum Prozessorsystem

Die *Memory Management Unit* bietet Schnittstellen an, um beliebige Daten in das *Shared Memory* zu schreiben, davon zu lesen oder für Debug-Zwecke zu dumpen. Die entsprechenden Funktionen sind in Code 19 aufgeführt.

```
//MMU Shared Memory Interface for Prozessor Systems
template<typename T>
  bool write_shared_memory(const uint32_t startAddrA, T* startDataA, uint32_t
numOfValuesA = 1);
  template<typename T>
  bool read_shared_memory(const uint32_t startAddrA, T* startDataA, uint32_t
numOfValuesA = 1) const;
template<typename T>
  void dump_memory(const uint32_t startAddrA, const uint32_t endAddrA,
  sc_dt::sc_numrep formatA = sc_dt::SC_NOBASE, bool showBaseA = false,
  std::ostream& os = std::cout) const;
```

Code 19: Abstrakte Systemschnittstelle des Prozessorsystems für den Zugriff auf den gemeinsamen Speicher

Die Funktionen sind als *Template*-Funktionen implementiert, um beliebige Datentypen in das *Shared Memory* zu schreiben oder von diesem zu lesen. Beim Schreiben von Daten muss der Pointer „`startDataA`“ auf den Quelldatenbereich, beim Lesen auf den Zielspeicherbereich zeigen. Von der Startadresse „`startAddrA`“ wird jeweils die Anzahl der Werte „`numOfValuesA`“ geschrieben oder gelesen. Dabei wird über den *Template*-Parameter „`T`“ entschieden, um wie viele Bytes der Daten-Pointer verschoben wird. Um welchen Wert sich die Adresse im *Shared Memory* verändert, hängt davon ab, wie viele Byte über eine Adresse abgefragt werden. Wird beispielsweise über eine Adresse ein Byte im *Shared Memory* angesprochen und es werden Integer-Daten geschrieben, „`T`“ ist ein „`int`“, dann wird mit jedem Schreiben die Adresse im *Shared Memory* um vier erhöht. Handelt es sich um zwei Byte pro Adresse, wird die Adresse immer um zwei erhöht und verbirgt sich hinter einer Adresse eine vier-Byte Speicher-„Zelle“, dann wird die Adresse nur um jeweils eins erhöht. Die Flexibilität in der Adressierung der Speicherzellen vergrößert den Evaluationsraum um zu untersuchen, welchen Einfluss angepasste Speicherzellengrößen auf die Performanz des Gesamtsystems haben.

Die Dump-Funktion der *Memory Management Unit* gibt standardmäßig auf *Standard-Outstream* aus, kann aber auch in eine beliebige Datei umgeleitet werden. Des

Weiteren bietet die Funktion Parameter zur Konfiguration der Formatierung an. So lassen sich Inhalte des *Shared Memory* als Dezimalzahlen, Hexadezimalzahlen oder Binärzahlen ausgeben.

4.3.3 Beschreibung der Zustandsmaschine der MMU

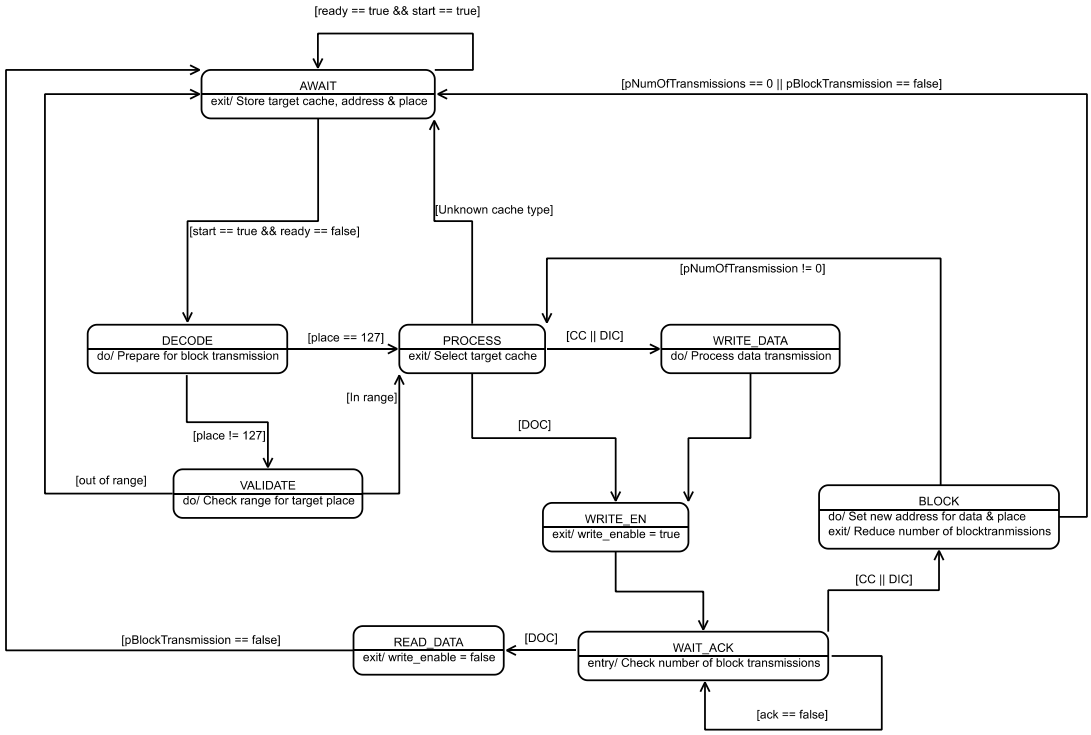

Abb. 54: Memory Management Unit – Zustandsautomat

Die *Memory Management Unit* arbeitet getaktet nach einem Zustandsautomat wie er in Abb. 54 dargestellt ist:

- `AWAIT`: Kennzeichnet den Initialzustand nach Systemstart und den Grundzustand, wenn die MMU kein Signal zur Verarbeitung empfangen hat. Die Verarbeitung wird generell durch ein positives Signal an „`start`“ eingeleitet. Das positive Signal muss für mindestens einen Taktzyklus anstehen. Beim Start einer neuen Verarbeitung werden die Signalzustände an „`place`“, „`address`“ and „`cache_select`“ gepuffert. Es kann daher schon eine neue Verarbeitung vorbereitet werden, indem beispielsweise eine neue Adresse am „`address`“-Port von außen eingestellt wird. Dies spart Zeit durch Parallelisierung.
- `DECODE`: In diesem Zustand werden die Daten an den gepufferten Ports ausgewertet. Es wird anhand des gewählten Ziels durch „`cache_select`“ die Übertragungsbitbreite passend zum *Outstream* bestimmt –

„data_value_out_stream“ oder „conf_data_stream“ – und die Anzahl der „*Block Transmissions*“ berechnet. Eine *Block Transmission* ist eine vollständige Übertragung eines gesamten Datums zwischen *Memory Management Unit* und Zielkomponente. Sie kann nicht unterbrochen werden. Wird beispielsweise als Bitbreite für einen Wert des *Virtual Coarse Grained Reconfigurable Arrays* 32 Bit verwendet, die Bitbreite des „data_value_out_stream“ beträgt aber nur 8 Bit, dann sind vier Übertragungen als *Block Transmission* notwendig, um das gesamte Datum zu übertragen. Bei Konfigurationen werden *Block Transmission* häufiger verwendet als bei Werten zur Verarbeitung für das VCGRA, da diese Konfigurationen wesentlich zahlreichere Bits beinhalten als ein Ganzzahlwert. Es wäre von der notwendigen Bitbreite unverhältnismäßig, alle Bits parallel zu übertragen, da diese Verbindungen gegebenenfalls mehrere 100 Bit an Breite aufweisen müssten.

- VALIDATE: In diesem Schritt wird kontrolliert, ob der angegebene Platz innerhalb der Zielkomponente gültig ist. Wenn nicht, wird die Datenübertragung zwischen *Memory Management Unit* und Zielkomponente abgebrochen, über das „ready“-Signal der Aufrufer des Speicherzugriffs per die MMU synchronisiert und eine zusätzliche Fehlermeldung ausgegeben. Das „ready“-Signal bleibt bestehen, bis das „start“-Signal innerhalb eines Taktzyklus auf Null ist.
- PROCESS: Die Datenübertragung zwischen *Memory Management Unit* und Ziel wird vorbereitet. Dazu werden der gewählte Platz innerhalb der Zeile einer Zielkomponente gesetzt, sowie anhand des Zieles das weitere Vorgehen bestimmt. Beim Lesen des *Data-Output* Puffers wird das Lesen neuer Daten bei diesem getriggert, bei den *Pre-Fetcher* werden zunächst die Nutzdaten an den korrespondierenden *Outstream* gelegt.
- WRITE_DATA: Die Nutzdaten werden anhand der Anzahl der *Block Transmission* in Teilsegmente zerlegt und an den *Outstream* gelegt.
- WRITE_EN: Das „write_enable“-Signal wird an die Zielkomponente gelegt. Dies führt entweder zu einem Schreiben der Nutzdaten an einem *Outstream* in den Ziel-*Pre-Fetcher* oder zu einem Start des Auslesens von Nutzdaten im *Data-Output* Puffer.
- WAIT_ACK: Eine Zielkomponente bestätigt einen Zugriff stets mit einem positiven „ack“-Signal. Die MMU pausiert die Verarbeitung, bis sie das entsprechende Signal erhält. Sind noch Übertragungen innerhalb einer *Block*

Transmission übrig, wird mit einer weiteren Übertragung fortgesetzt. Andernfalls wird der Aufrufer über das Ende der Verarbeitung der *Memory Management Unit* informiert.

- `READ_DATA`: Dieser Zustand ist exklusiv für das Lesen von Daten aus dem *Data-Output* Puffer. Die Daten werden an die Zieladresse im *Shared Memory* geschrieben.
- `BLOCK`: Überprüft und berechnet die Anzahl verbleibender Übertragungen einer *Block Transmission*.
- `FINISH`: Es wird das „`ready`"-Signal auf *High* gezogen, um den Aufrufer eine beendete Übertragung anzuzeigen. Die MMU ist bereit für eine neue Übertragung, wenn das „`start`"-Signal für einen Zyklus Null ist und dann wieder auf *High* wechselt. Die *Memory Management Unit* wartet in diesem Zustand, bis der entsprechende Flankenwechsel an „`start`" nach *Low* für einen Taktzyklus stattgefunden hat.

Die Anzahl der Zyklen für eine Übertragung ist abhängig von der Größe des Datums, welches übertragen werden soll und der Bitbreite zur Übertragung über die Streamingports. Für jede Übertragung werden mindestens acht Taktzyklen benötigt, um alle notwendigen Zustände des Zustandsautomaten zu durchlaufen.

4.3.4 Timingdiagramme zur Funktionsweise der MMU

Im Folgenden sind für alle *Pre-Fetcher* sowie den *Data-Output* Puffer Timingdiagramme dargestellt. Die Diagramme wurden in einer gemeinsamen Simulation erstellt. Der schematische Aufbau der Simulation ist in Abb. 55 dargestellt.

Die Signale am rechten Bildrand sind in einer vollständigen Architektur mit einem VCGRA verbunden, Signale am linken Bildrand mit der *Central Control Unit*. Eine *Testbench* simulierte ein VCGRA für die Verbindung der Datensignale und der Signale für die Konfigurationen der Prozesselemente und der virtuellen Kanäle. Des Weiteren beaufschlagt die *Testbench* die Steuersignale der *Memory Management Unit* wie beispielsweise des „`slt_in`", „`slt_out`" und des „`cache select`" Signales. Diese Signale werden nicht von der *Memory Management Unit* selbst gesteuert, sondern zum Beispiel über die *Central Control Unit*. Der Grund ist, dass in einer zukünftigen Erweiterung der Gesamtarchitektur die *Pre-Fetcher* und Puffer als *Caches* fungieren sollen. Die *Central Control Unit* übernimmt in diesem Szenario die Aufgabe, die Inhalte der *Caches* zu überwachen und deren Nutzung zu steuern. Die Wahl fällt dabei auf die CCU, weil diese als erste Instanz den Assemblercode interpretiert und damit über die Vorgänge des Algorithmus am ehesten Bescheid weiß.

Die Synchronisierungssignale der Kommunikation zwischen der MMU und einer Pufferkomponente sind geteilt und werden über Multiplexer umgeschaltet. Diese Multiplexer werden vom „`cache_select`“-Signal der *Central Control Unit* gesteuert. Das spart Leitungsaufwand im Vergleich zu separaten Signalen und ist ausreichend, da die *Memory Management Unit* Zugriffe auf Speicherkomponenten nur sequenziell und nicht parallel durchführen kann.

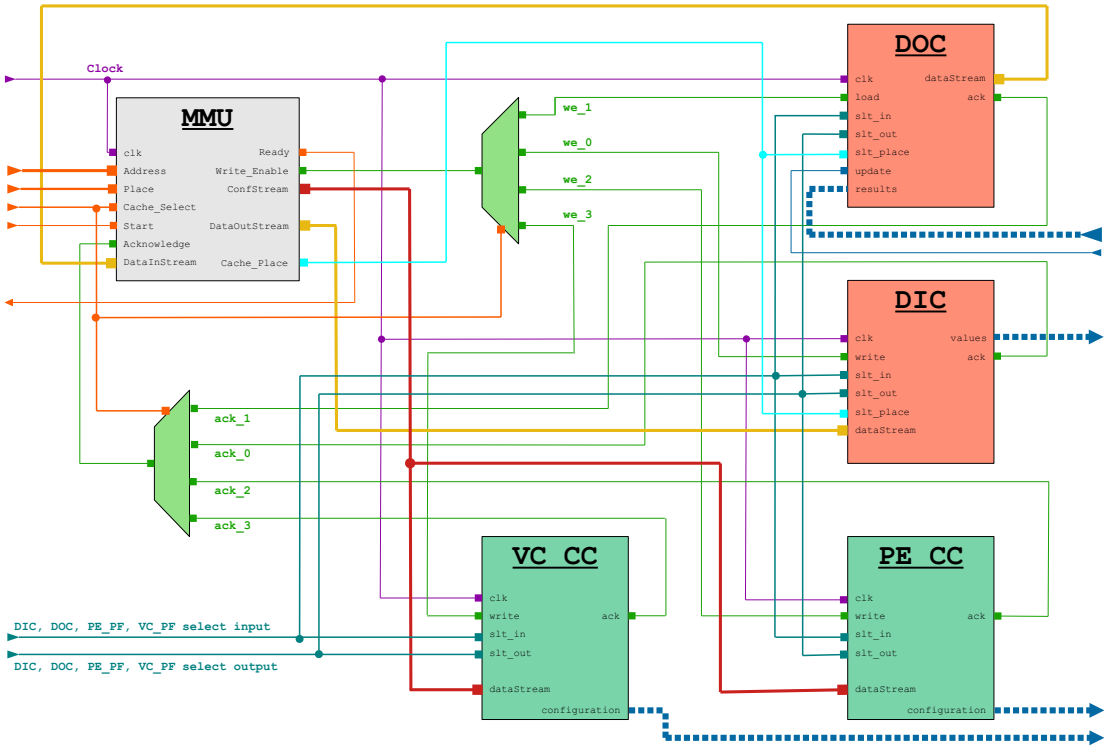

Abb. 55: Schematische Darstellung – Funktionstest der MMU- und Pufferkomponenten

Durch die gemeinsame Simulation sind die Ergebnisse der nicht-zugehörigen Komponenten ausgeblendet. Die Architektur wird mit acht Eingangswerten und Ausgangswerten mit jeweils sechzehn Bit Bitbreite angenommen. Dementsprechend wird die Bitbreite für „*data-streams*“ „`DataInStream`“ und „`DataOutStream`“ ebenfalls auf sechzehn Bit gesetzt. Die Bitbreiten für „`Address`“ und „`Place`“ ergeben sich aus den Bitbreiten aus dem Maschinencode-Binärformat aus Abschnitt 3.3.1, die Bitbreite für „`Cache_Place`“ wurde zur Erstellungszeit der Architektur errechnet und resultiert aus der Anzahl der zu adressierenden Plätze in *Data-Input Pre-Fetcher* und *Data-Output* Puffer. In diesem Fall sind es jeweils acht Plätze, weshalb drei Bit für deren Adressierung notwendig sind. Die Bitbreite des „`ConfStream`“ ist

auf acht Bit Breite selbst festgesetzt. „`Cache_Select`“ ist drei[12] Bit breit, um die vier Architekturkomponenten:

- `DIC`: *Data-Input Pre-Fetcher*
- `DOC`: *Data-Output* Puffer
- `PE_CC`: *Processing Element Configuration Pre-Fetcher*
- `VC_CC`: *Virtual Channel Configuration Pre-Fetcher*

zu unterscheiden.

4.3.4.1 Data-Input Pre-Fetcher

In der Simulation nach Abb. 56 (siehe auch Abb. 74 im Anhang für eine größere Darstellung) wurde jeweils das Übertragen einzelner Werte sowie ein Blocktransfer simuliert. In die Zeile null des *Data-Input Pre-Fetcher* („`Cache_Select == DIC`“) werden zunächst alle acht Werte in separaten Übertragungen geladen, während nach einem Wechsel in Zeile eins direkt eine *Blocktransmission* angestoßen wird. Die Übertragung eines Datums an den *Pre-Fetcher* benötigt vier Taktzyklen, zu sehen an der steigenden Flanke von „`Write_Enable`“ und der fallenden Flanke von „`Acknowledge`“ für jede Übertragung.

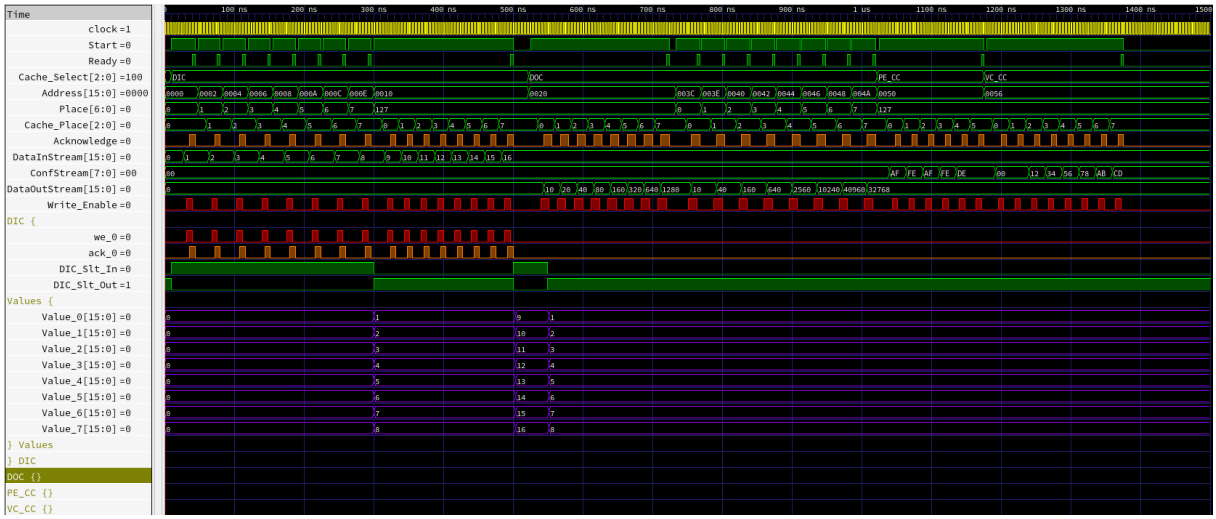

Abb. 56: Timingdiagramm (klein) – MMU - Data-Input Pre-Fetcher

Jede neue Übertragung eines Wertes wird durch ein „`Start`“-„`Ready`“-Signalpaar abgegrenzt. Jedes Paar ist für die Einzelübertragung der Werte acht Taktzyklen lang. Mit jeder Übertragung wird die Adresse und der Zielpatz im *Data-Input Pre-Fetcher* gesetzt. Die Adresse erhöht sich jeweils um zwei Byte auf Grund der Bitbreite eines

[12] Es ist auch eine Bitbreite von 2 Bit ausreichend.

Datums von sechzehn Bit, wobei festgelegt ist, dass in diesem Test eine Adresse ein Byte Speicher im *Shared Memory* adressiert.

Beim Wechsel auf die Zeile null wird eine *Blocktransmission* durch den Zielplatz „127“ angestoßen. Die übergebene Adresse wird als Startadresse für acht Werte im Speicher interpretiert. Wie zu erkennen ist, ist kein weiterer Aufruf notwendig, um die Werte einer Zeile sequentiell aus dem Speicher ab einer bestimmten Adresse zu laden. Die Anpassung der Adresse für den Speicherzugriff und der Platz in der Zielzeile des *Pre-Fetcher* werden von der *Memory Management Unit* selbstständig errechnet und gesetzt. Bei einem Blocktransfer reduziert sich die Anzahl der Taktzyklen für eine Übertragung auf sechs, da der FINISH- und DECODE -Zustand nur einmal bzw. der VALIDATE-Zustand überhaupt nicht durchlaufen werden (siehe Abb. 54). Dafür wird der zusätzliche BLOCK-Zustand durchlaufen, weshalb insgesamt nur zwei Taktzyklen Gewinn je Übertragung erreicht werden. Eine *Blocktransmission* kann nicht unterbrochen werden, es ist daher vom Anwender zu prüfen, ob die MMU für in diesem Fall 50 Taktzyklen beschäftigt sein darf. 48 Taktzyklen werden für die Übertragung verwendet, das entspricht acht mal sechs Taktzyklen plus zwei Taktzyklen für „DECODE“ und „FINISH“. Zu den Zeitpunkten 300ns und 500ns werden die übertragenen Werte jeweils an die Ausgänge des *Data-Input Pre-Fetcher* gelegt. Dies geschieht durch den Wechsel von „DIC_Slt_In“ und „DIC_Slt_Out“.

4.3.4.2 Data-Output Buffer

In der Simulation nach Abb. 57 (siehe auch Abb. 75 im Anhang für eine größere Darstellung) wurde das Übertragen einzelner Werte sowie ein Blocktransfer simuliert. In die Zeile null des *Data-Output* Puffer („Cache_Select == DOC“) werden zunächst alle acht Werte durch den externen Trigger „Update“ gepuffert und anschließend als *Blocktransmission* in das *Shared Memory* übertragen. Während der Übertragung wird ein weiterer externer Trigger an „Update“ verwendet, um neue Daten im Puffer zwischenzuspeichern. Nach der Übertragung als *Blocktransmission* erfolgt die Übertragung der zweiten Werte in Zeile eins als Einzeltransfers.

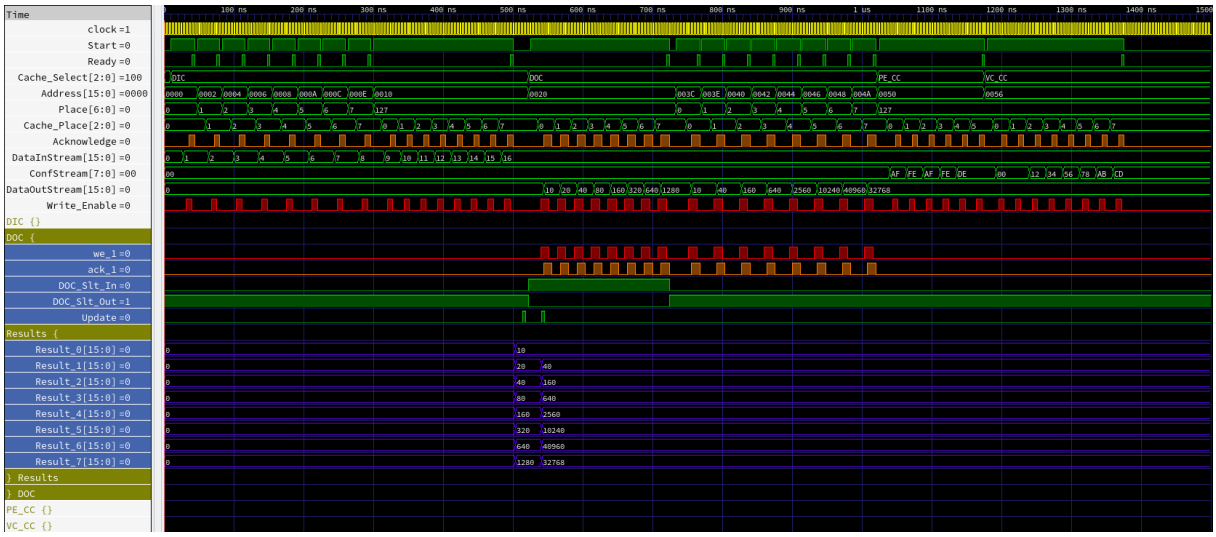

Abb. 57: Timingdiagramm (klein) – MMU - Data-Output Buffer

Über den „`DataOutStream`“ ist erkennbar, dass die Resultwerte des ersten Triggers nacheinander übertragen werden. Des Weiteren ist erkennbar, dass die Trigger für „`Write_Enable`“ und „`Acknowledge`“ explizit für eine Komponente getriggert sind. Die Signale sind zwar geteilt in Verwendung (siehe Abb. 55), jedoch wird über „`Cache_Select`“ das Signal über einen Multiplexer jeweils an die richtige Komponente hin- und dessen Signale auch wieder zurückgeführt. Aus diesem Grund sind die Signalwechsel für den *Data-Input Pre-Fetcher* nicht beim *Data-Output* Puffer angekommen, was durch das ständige Nullsignal über den Zeitraum „`Cache_Select == DIC`“ sichtbar wird.

Die Signale für „`Write_Enable`“ und „`Acknowledge`“ wechseln sich auf Grund des Synchronisierungsmechanismus, beschrieben in Abschnitt 4.2.3, ab, ihre Aktivzeit ist aber erkennbar höher. Dies liegt daran, dass die *Memory Management Unit* eine Abfrage eines Wertes beim *Data-Output* Puffer zunächst anfordert, dann der Wert vom Puffer zunächst bereitgestellt werden muss, anschließend über „`Acknowledge`“ das Datum als valide markiert wird und erst dann von der MMU übernommen werden kann. Das Bereitstellen der Daten und das Tiggern von „`Acknowledge`“ dauert einen Taktzyklus beim *Data-Output* Puffer. Der anschließende Wechsel der Zustände im Zustandsautomaten der *Memory Management Unit* vom Warten auf den „`Acknowledge`“-Trigger, der Datenübernahme und der Synchronisation durch einen Reset des „`Write_Enable`“-Signals dauern jeweils einen weiteren Taktzyklus. Insgesamt werden aber für eine Übertragung eines Wertes ebenfalls wieder sechs Takte benötigt, wenn der *Data-Output* Puffer nicht beschäftigt ist und damit die Synchronisation innerhalb eines Taktzyklus abgeschlossen werden kann. Die *Blocktransmission* dauert in diesem Fall erneut 50 Taktzyklen, beginnt zum Zeitpunkt

524ns und ist abgeschlossen zum Zeitpunkt 724ns – die Periodendauer der *Clock* ist auf 4ns gestellt. 48 Taktzyklen werden für die Übertragung verwendet, das entspricht acht mal sechs Taktzyklen, plus zwei Taktzyklen für „`DECODE`“ und „`FINISH`“. Beim Start einer *Blocktransmission*, angestoßen durch den Zielplatz „`127`“, wird die Startadresse zur Ablage der Daten mit angegeben. Die *Memory Management Unit* passt die nachfolgenden Adressen wieder selbstständig an anhand der Bitbreite des zu speichernden Wertes.

Im Timingdiagramm Abb. 57 ist sehr gut die Parallelisierung der Datenübertragung zwischen *Shared Memory* und *Data-Output* Puffer erkennbar. Trotz der laufenden Übertragung der Daten, welche in Zeile null gespeichert sind, können die Daten in Zeile eins übernommen und anschließend übertragen werden. Werden die Daten nicht als Block übertragen, muss jeweils die Zieladresse des Wertes mit an die MMU übertragen werden. Aus diesem Grund ist wie beim *Data-Input Pre-Fetcher* jeweils eine neue Zieladresse pro Übertragung angezeigt, ebenso wie ein neuer Zielplatz in der Zielzeile.

4.3.4.3 Configuration Pre-Fetcher

Die Übertragungen der *Configuration Pre-Fetcher* unterscheiden sich nicht stark von dem des *Data-Input Pre-Fetcher*. Im Gegensatz zu diesem werden keine Einzelwertübertragungen unterstützt. Die Anzahl der Übertragungen einer *Blocktransmission* ist abhängig von der Größe des Konfigurationbitstrom und der Bitbreite des „`ConfStream`“. Die Anzahl der Taktzyklen für die Übertragung eines Konfigurationsteils ist gleich dem des *Data-Input Pre-Fetcher*.

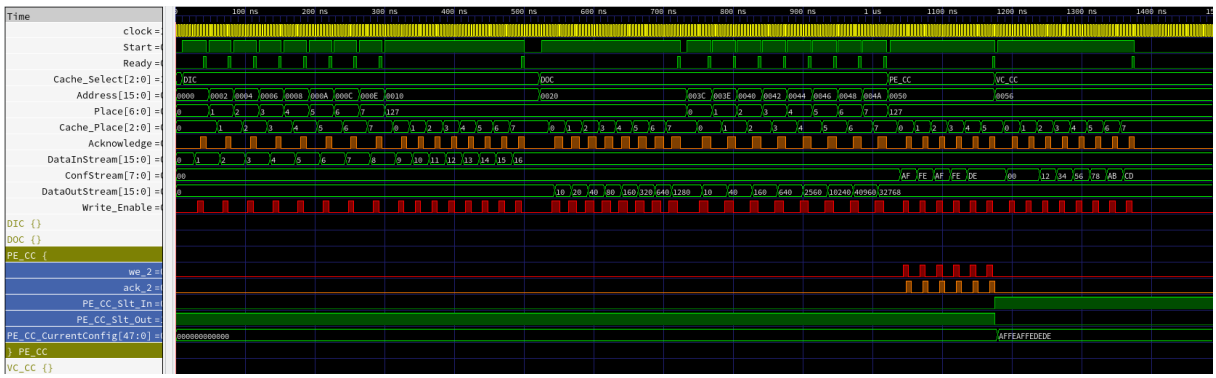

Abb. 58: Timingdiagramm (klein) – MMU – PE Configuration Pre-Fetcher

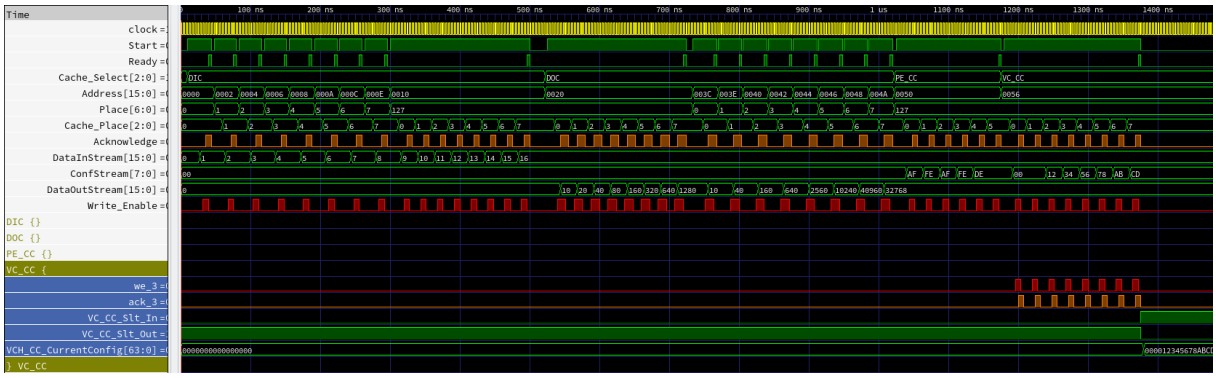

Abb. 59: Timingdiagramm (klein) – MMU – Virtual Channel Configuration Pre-Fetcher

In Abb. 58 (Abb. 76 in groß im Anhang) und Abb. 59 (Abb. 77 in groß im Anhang) sind jeweils die Timingdiagramme für die *Configuration Pre-Fetcher* der Konfigurationen für die Prozesselemente und die virtuellen Kanäle inklusive *Synchronization Unit* abgebildet. Da die Konfiguration für die virtuellen Kanäle länger ist, benötigt die Gesamtübertragung mehr Zeit. Der Bitstrom für die Prozesselemente ist in 36 Taktzyklen plus jeweils einen Taktzyklus für „`DECODE`“ und „`FINISH`“ übertragen. Das ergibt eine Übertragungszeit von 152ns bei einer *Clock*-Periode von 4ns. Für die Konfiguration der virtuellen Kanäle und der *Synchronization Unit* werden 50 Taktzyklen inklusive jeweils einen Taktzyklus für „`DECODE`“ und „`FINISH`“ benötigt, was einer Zeit von 200ns entspricht bei gleicher Periodendauer der *Clock*.

4.4 Management Unit (Central Control Unit)

Die *Management Unit* oder *Central Control Unit* übernimmt die Steuerung der *Memory Management Unit* und des *Virtual Coarse Grained Reconfigurable Arrays*. Sie interpretiert den Maschinencode, welcher aus einem Assembler passend zur Architektur generiert werden kann (siehe Abschnitt 3.3.3). In Abb. 60 ist die CCU schematisch abgebildet. Sie besteht im Wesentlichen aus drei Komponenten, der *Module State Machine*, der *Process State Machine* und dem *Command Interpreter*.

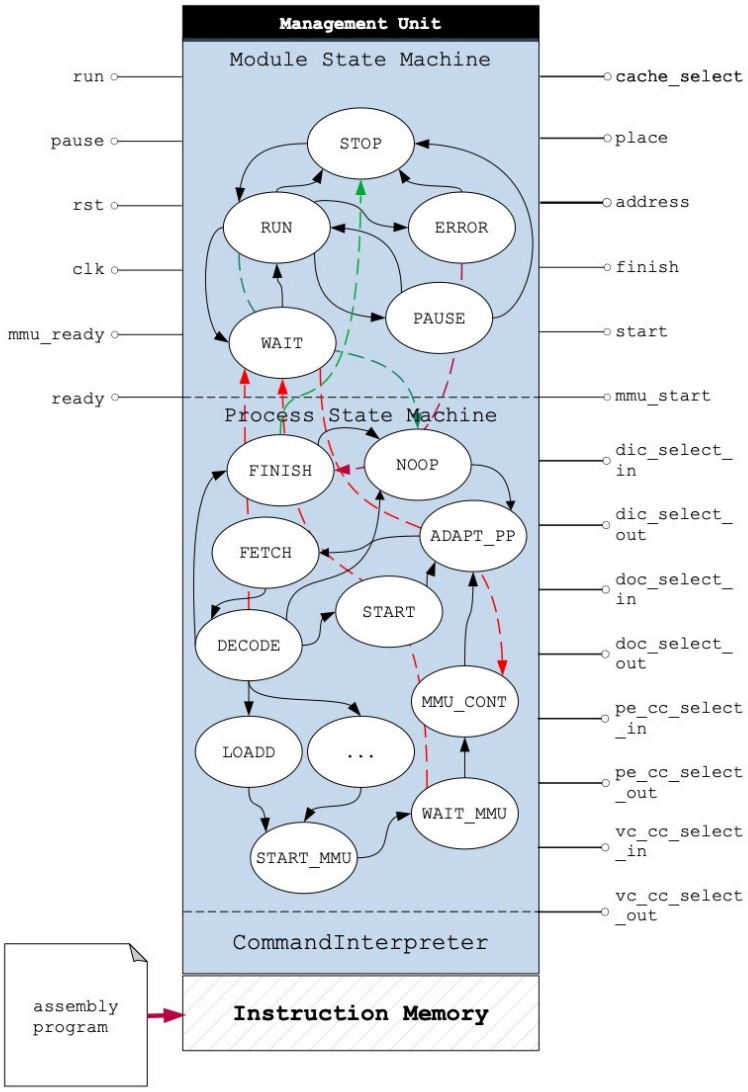

Abb. 60: Schematische Darstellung – Central Control Unit

4.4.1 Module State Machine

Die *Module State Machine* steuert das Verhalten der CCU. Die *Central Control Unit* befindet sich standardmäßig im „STOP"-Zustand. Sie muss von außen über ein *High*-aktives Signal am „run"-Port in den „RUN"-Zustand überführt werden. Es reicht hierbei ein positiver Flankenwechsel. Im „RUN"-Zustand ist die *Process State Machine* aktiv, welche die Verarbeitung der Kommandos aus dem *Command Interpreter* vornimmt. Sie beginnt die Verarbeitung im „NOOP"-Zustand und fährt fort mit dem Laden und Dekodieren des ersten Befehls. Die Beziehung zwischen den beiden Zustandsautomaten ist durch die gestrichelten Linien dargestellt. Die grüne, gestrichelte Linie von „RUN" nach „NOOP" veranschaulicht beispielsweise diese Beziehung. Die Verarbeitung kann von außen pausiert werden. Dies geschieht,

solange das Signal an „pause“ auf *High* gesetzt bleibt. Die Verarbeitung wird fortgesetzt, sobald das Signal an „pause“ wieder *Low* ist. Tritt ein Fehler auf, wird eine Fehlermeldung ausgegeben und der aktuelle Zustand „*gedumpt*“. Außerdem wird das Ende der Verarbeitung über den „finish“-Port angezeigt. Die lila, gestrichelte Linie von „ERROR“ nach „FINISH“ veranschaulicht diese Beziehung. Die *Central Control Unit* befindet sich anschließend im „STOP“-Zustand und kann nur durch einen Neustart, wie oben beschrieben, die Verarbeitung ab Programmanfang erneut beginnen. Die grüne, gestrichelte Linie von „FINISH“ nach „STOP“ veranschaulicht wiederum diese Beziehung.

Neben dem „RUN“ ist der „WAIT“-Zustand der *Module State Machine* noch mit der *Process State Machine* verschränkt. Wird eine Aktion der *Memory Management Unit* durch ein Kommando erzeugt oder gar durch das Kommando „WAIT_READY“ ein Warten auf das Beenden der Verarbeitung des *Virtual Coarse Grained Reconfigurable Arrays* verlangt, befindet sich die *Central Control Unit* im „WAIT“-Zustand. Es findet keine weitere Interpretation von Befehlen statt. Es wird auf eine positive Flanke am „mmu_ready“ bzw. „ready“-Port gewartet. Ersterer wird durch die *Memory Management Unit*, Zweiterer durch das VCGRA gesteuert. Werden während der Verarbeitung im VCGRA Daten über die MMU an die *Pre-Fetcher* und den Puffer übertragen und meldet das VCGRA parallel schon ein „*ready*“ an dem korrespondierenden Port, wird dieses Event zwischengespeichert. Wird im Assemblerquellcode dann die Stelle „WAIT_READY“ erreicht und der entsprechende Trigger kam bereits, wird direkt mit der Verarbeitung in der *Central Control Unit* fortgesetzt. Andernfalls wird auf den entsprechenden Trigger aus dem VCGRA gewartet. Über den „rst“-Port kann zu jeder Zeit ein asynchroner Reset ausgelöst werden. Dadurch wird der „*Program Pointer*“ an den Anfang und der Modulzustand auf „STOP“ gesetzt. Das Programm kann dann neu gestartet werden durch eine positive Flanke am „run“-Port. Die Kontrolle des Moduls durch das Schalten der Steuerports kann asynchron erfolgen, da diese von extern erfolgen und damit die Gesamtarchitektur nicht taktsynchron mit dem Prozessorsystem ausgeführt werden muss. Die Verarbeitung der *Process State Machine* erfolgt zyklisch getaktet durch den „clk“-Port.

4.4.2 Process State Machine

Wie bereits erwähnt sei an dieser Stelle nochmal explizit darauf verwiesen, dass die *Process State Machine* nur im *Module State Machine* Zustand „RUN“ ausgeführt wird. Im „NOOP“-Zustand wird für einen Taktzyklus keine Operation ausgeführt. Im

„FETCH“-Zustand wird der nächste Befehl aus dem *Instruction Memory* des Assembler Programmes (siehe auch Abschnitt 3.3.3) geladen und dem *Command Interpreter* (siehe Abschnitt 4.4.3) zugeführt. Anschließend folgt im „DECODE“-Zustand die Interpretation des Befehls. Aus dem Ergebnis des „DECODE“-Zustandes lassen sich fünf Hauptaufgaben unterscheiden:

- Speicherzugriff über die *Memory Management Unit*: >7 Zyklen
- Auswählen einer Konfiguration oder eines Datensatzes an einem der *Pre-Fetcher* oder des *Data-Output* Puffer: 4 Zyklen
- Starten der Verarbeitung des VCGRA: 5 Zyklen
- Beenden des aktuellen Assemblerprogrammes und Synchronisation mit dem Prozessorsystem: 4 Zyklen
- Leeroperation („NOOP“) ausführen: 4 Zyklen

Beim Speicherzugriff werden die *Load-Store*-Befehle des Assemblers verarbeitet. Dazu werden im „DECODE“-Zustand die Adresse, der *Place* und die *Line* an die Signalports der *Memory Management Unit* gelegt und die Multiplexer für die Synchronisierungssignale – *Write/Load* und *Acknowledge* Signale, siehe auch Abschnitt 4.2 und Abschnitt 4.3 – eingestellt. Nach der Vorbereitung schaltet die *Central Control Unit* im „MMU_START“-Zustand die MMU aktiv. Hierbei wird das „start“-Signal der MMU getriggert, worauf diese die Daten an den vorher vorbereiteten Ports übernimmt und diese bearbeitet. Anschließend schaltet sich die CCU im „WAIT“-Zustand „*idle*“ und wartet durch einen externen Interrupt von der MMU darauf, die Verarbeitung fortzusetzen. Im „MMU_CONT“-Zustand wird nach dem Aufwachen durch den externen Trigger die Kommunikation zur *Memory Management Unit* wieder zurückgesetzt. Zuletzt wird über die Anpassung des *Program Pointers* im „ADAPT_PP“-Zustand der *Fetch* für den nächsten Befehl vorbereitet.

Die *Load-Store*-Befehle des Assemblers sind in Abb. 60 exemplarisch mit „LOADD“ und „. . .“ angedeutet. Die Vorbereitung für den Datenaustausch der MMU mit einer *Pre-Fetcher* oder Puffer Komponente dauert sieben Taktzyklen. Für die aktive Zeit der MMU wartet die *Central Control Unit*. Die Wartezeit ist abhängig von der Datenübertragung der MMU zur Zielkomponente, beispielsweise Menge der zu Übertragenen Daten, Bitbreite der Übertragungskanäle zwischen *Memory Management Unit* und Ziel oder ob Einzel- bzw. Blockübertragungen stattfinden. Aus diesem Grund werden mindestens sieben Zyklen benötigt, tendenziell eher mehr.Das Auswählen eines Parameters oder Datensatzes dauert vier Taktzyklen. Anhand des *Command Identifier* des Maschinencode-Binärformats (siehe Abschnitt 3.3.1) wird die

entsprechende Zielkomponente gewählt und in die aktive Zeile umgeschaltet. Die Befehle verändern den Zustand der „`_select_out`“-Ports der MMU. Die „`_select_in`“-Ports werden dagegen durch die *Load-Store*-Befehle adressiert. Ebenfalls vier Taktzyklen benötigen die Befehle für eine Leeroperation und *Finish*. Der „*Finish*“-Befehl setzt das Signal am „`finish`“-Port für einen Taktzyklus auf *High*, um das Prozessorsystem auf den Abschluss des Programmes hinzuweisen. Der Befehl zum Starten des *Virtual Coarse Grained Reconfigurable Arrays* benötigt fünf Zyklen und beeinflusst das Signal am „`start`“-Port der *Central Control Unit*.

4.4.3 Command Interpreter

Der *Command Interpreter* führt eine Aufteilung des Maschinencode-Binärformats in seine Bestandteile durch, beschrieben in Abschnitt 3.3.1. Die einzelnen Informationen werden in der CCU in separaten Puffern zwischengespeichert und zur weiteren Verarbeitung genutzt. Aus diesem Grund wäre es an dieser Stelle auch möglich, bereits ein *Pre-Fetching* von Kommandos zu implementieren, um somit das *Fetching* und *Decoding* parallel zur weiteren Verarbeitung der Befehle durchzuführen. Der Zustandsautomat der *Central Control Unit* ist dafür aber noch nicht vorgesehen, da er beispielsweise im „`WAIT`“-Zustand vollständig suspendiert wird und weil der „`FETCH`“- und „`DECODE`“-Prozesszustand ein Teil des gesamten Zustandsautomaten ist. Es sind daher weitreichende Umbaumaßnahmen notwendig, die ggf. zu einem späteren Zeitpunkt zur weiteren Geschwindigkeitssteigerung durchgeführt werden könnten.

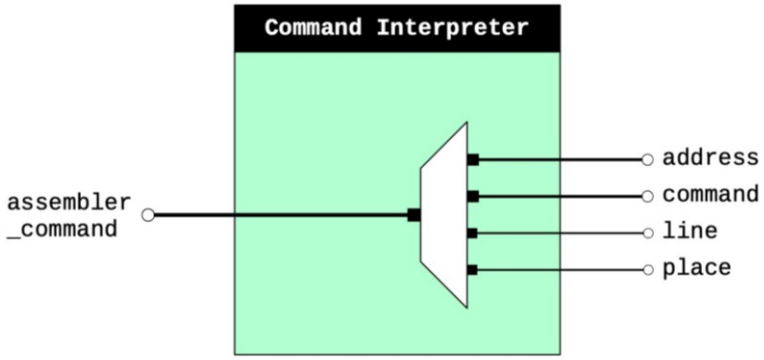

Abb. 61: Schematische Darstellung – Command Interpreter

4.5 Übersicht der Gesamtarchitektur

4.5.1 VCGRA Implementierung in SystemC

Im Folgenden wird nun in Kürze auf die Umsetzung der einzelnen Komponenten des *Virtual Coarse Grained Reconfigurable Arrays* eingegangen, welche bereits in VHDL umgesetzt wurden. Für eine gute Vergleichbarkeit in der SystemC-Simulation mit der Umsetzung in VHDL wird bei der Implementierung auf zusätzliche Neuheiten verzichtet.

4.5.1.1 Processing Element

Das Prozesselement ist eine reine *Template*implementierung. Durch die drei *Template*-Parameter (siehe Code 20) werden die Bitbreiten für die Eingangsdaten, die Ausgangsdaten und die Bitbreite für den Konfigurationseingang (siehe auch Abb. 29) konfiguriert.

```
//! \tparam N Bitwidth of incoming data
//! \tparam K Bitwidth of outgoing data
//! \tparam L Bitwidth of configuration data
/*****************************************************************/
template <uint16_t N = 8, uint16_t K = 8, uint16_t L = 4>
class Processing_Element;
```

Code 20: SystemC-Klasse – Processing Element

Die Bitbreite des Konfigurationseinganges ist abhängig von der Anzahl der Operationen, welche ein Prozesselement ausführen kann. Jedes Prozesselement erhält bei seiner Konstruktion eine eindeutige ID. Die Verarbeitung findet synchron, taktgesteuert mit steigender Flanke am Port „`clk`“ statt. Auch in der Simulation arbeitet die PE nach dem Zustandsautomat wie in Abb. 29. Dieser läuft in einem als `SC_METHOD` definierten Prozess. Es werden die identischen arithmetischen Operationen für Ganzzahlen unterstützt wie für die VHDL-Implementierung.

Die *Template*-Parameter zur Konfiguration der Komponenten des *Virtual Coarse Grained Reconfigurable Arrays* sind wieder in der Datei „*Typedef.h*“ abstrahiert. In Code 21 sind die Parameter zur Konfiguration der Prozesselemente aufgeführt. Durch das Design des *Virtual Coarse Grained Reconfigurable Arrays* und der statischen Allokationen der Module sowie die Verwaltung der Prozesselemente in einem gemeinsamen Array ist die Flexibilität der Prozesselemente eingeschränkt. So sind die Bitweiten für Eingangs- und Ausgangsdaten identisch und durch `cPeDataBitwidth` festlegbar. `cPeConfigLvSize` beschreibt die Bitweite für die Konfiguration der

verfügbaren Operationen, sprich die Bitweite des „conf“-Ports. Da alle Prozesselemente in einem gemeinsamen Array verwaltet werden, müssen alle Instanzen vom Type `ProcessElement` identisch sein und besitzen damit auch alle die gleichen verfügbaren Operationen.

```
// Typedef.h
static constexpr uint32_t cPeDataBitwidth{16};
static constexpr uint32_t cPeConfigLvSize{4};

// vcgra.h
typedef cgra::Processing_Element<cgra::cPeDataBitwidth,
                cgra::cPeDataBitwidth, cgra::cPeConfigLvSize> pe_type_t;
sc_core::sc_vector<pe_type_t> m_pe_instances{"VCGRA_PEs"};
```

Code 21: Abstraktionsparameter für eine Prozesselement Instanz

4.5.1.2 Virtual Channel

Ein virtueller Kanal (engl. Virtual Channel) ist als reine *Template*implementierung programmiert. Der interne Aufbau entspricht dem Aufbau nach Abb. 30. Die *Template*-Klasse besitzt zur Konfiguration sechs Parameter. Diese beschreiben die Anzahl und Bitbreite der Eingänge des Kanals, die Anzahl und Bitbreite der Ausgänge des Kanals sowie die interne Bitbreite der Signalleitungen innerhalb des virtuellen Kanals. Zusätzlich muss für eine Instanz eines virtuellen Kanals die Anzahl der internen Multiplexerinstanzen und deren Bitbreite für die Selektion eines Ausganges spezifiziert werden. Die notwendige Bitbreite für den Konfigurationseingang des virtuellen Kanals wird errechnet aus den gegebenen Parametern für die internen Multiplexer. Die interne Bitbreite der Signale des virtuellen Kanals muss größer oder gleich dem größten Wert der beiden Parameterwerte für die Bitbreiten der Eingänge und Ausgänge gesetzt werden.

```
//! \tparam N Bitwidth of incoming data
//! \tparam K Bitwidth of outgoing data
//! \tparam L Bitwidth of configuration data
/*****************************************************************/
template <uint16_t N = 8, uint16_t K = 8, uint16_t L = 4>
class Processing_Element;
```

Code 22: SystemC-Klasse – Virtual Channel

Das Modul besitzt drei Prozesse, welche als `SC_METHOD` implementiert sind. Zwei dieser Prozesse verlaufen synchron, taktgesteuert mit steigender Flanke am Port „`clk`“. Diese übernehmen die Eingangsdaten an den Eingangspuffern und setzen die

Ausgangsdaten an den Ausgangspuffern. Die Umstellung der aktuellen Konfiguration erfolgt asynchron, direkt mit dem Ändern des Signals am Port „`conf`“. Die Verarbeitung der Eingangsdaten und dem Ausgangsdaten dauert zwei Taktzyklen.

Die *Template*-Parameter zur Konfiguration der Komponenten des *Virtual Coarse Grained Reconfigurable Arrays* sind wieder in der Datei „*Typedef.h*“ abstrahiert. In Code 23 sind die Parameter zur Konfiguration der virtuellen Kanäle aufgeführt. Es wird im aktuellen Design nur in zwei Implementierungen unterschieden: Den Eingangskanal des *Virtual Coarse Grained Reconfigurable Arrays* (*Input Channel*) und den verbleibenden Kanälen der weiteren Ebenen. Wie bei den Prozesselementen ist die Anzahl der Konfigurationen stark vom Design abhängig. Im Beispieldesign aus Abschnitt 2.3.6.3 besteht eine Ebene jeweils aus vier Prozesselementen. Aus diesem Grund sind die Konfigurationen für die virtuellen Kanäle der verbleibenden Ebenen gleich und die virtuellen Kanäle können im C++-Quellcode zu einem Array zusammengefasst werden, da sie alle mit Ausnahme des Eingangskanals den gleichen Typen besitzen.

In dem gegebenen Codebeispiel Code 23 unterscheiden sich die beiden Implementierungen durch ihre Anzahl an Eingängen und Ausgängen und durch die Bitbreite des „`select`“-Ports, verursacht durch die unterschiedliche Anzahl an Eingängen der beiden Kanalvarianten. Da das VCGRA-Design nur eine Bitbreite für Daten unterstützt (`cPeDataBitwidth`), um die Prozesselemente in einem Array zu verwalten, ist dieser Parameter auch verwendet für alle Datenbitbreiten der Kanäle. Diese Einschränkungen sind derzeit getroffen, um den Programmieraufwand zu reduzieren. Zum Erstellen der Verbindungen von Signalen und Ports wird zur Vereinfachung des Quellcodes mit Schleifen über Arrays iteriert. Bei einer zukünftigen Automatisierung der Architekturgenerierung können die Elemente und deren Verbindungen auch durch einzelne Quellcodezeilen instanziiert bzw. vorgenommen werden. Dann können die derzeitigen Einschränkungen bezüglich der Datentypen der Module und deren Verwaltung in Arrays auch aufgeweicht werden.

```
//Typedef.h
static constexpr uint32_t cInputChannel_NumOfInputs{8};
static constexpr uint32_t cInputChannel_InputBitwidth{cPeDataBitwidth};
static constexpr uint32_t cInputChannel_NumOfOutputs{8};
static constexpr uint32_t cInputChannel_OutputBitwidth{cPeDataBitwidth};
static constexpr uint32_t cInputChannel_MuxScltBitwidth{3};
static constexpr uint32_t cInputChannel_InternalBitwidth{cPeDataBitwidth};
static constexpr uint32_t cChannel_NumOfInputs{4};
static constexpr uint32_t cChannel_InputBitwidth{cPeDataBitwidth};
static constexpr uint32_t cChannel_NumOfOutputs{8};
static constexpr uint32_t cChannel_OutputBitwidth{cPeDataBitwidth};
static constexpr uint32_t cChannel_MuxScltBitwidth{2};
static constexpr uint32_t cChannel_InternalBitwidth{cPeDataBitwidth};

// vcgra.h
typedef cgra::VirtualChannel<
        cgra::cInputChannel_NumOfInputs,
        cgra::cInputChannel_InputBitwidth,
        cgra::cInputChannel_NumOfOutputs,
        cgra::cInputChannel_OutputBitwidth,
        cgra::cInputChannel_MuxScltBitwidth,
        cgra::cInputChannel_InternalBitwidth> input_channel_type_t;
typedef cgra::VirtualChannel<
        cgra::cChannel_NumOfInputs,
        cgra::cChannel_InputBitwidth,
        cgra::cChannel_NumOfOutputs,
        cgra::cChannel_OutputBitwidth,
        cgra::cChannel_MuxScltBitwidth,
        cgra::cChannel_InternalBitwidth> channel_type_t;
input_channel_type_t m_input_channel{"Input_Channel"};
sc_core::sc_vector<channel_type_t> m_channel_instances{"VCGRA_Channels"};
```

Code 23: Abstraktionsparameter für eine Virtual Channel Instanz

4.5.1.3 Synchronization Unit

Dieses Modul ist ebenfalls vollständig als *Template* implementiert. Es besitzt zwei *Template*-Parameter: Mit dem ersten Parameter wird der Signaltyp für ein „`valid`“-Signal bestimmt. Dies kann zum Beispiel ein Standard-`Bool` oder ein `sc_dt::sc_logic`-Typ sein. Mit dem zweiten Parameter wird die Anzahl der „`valid`“-Signale, welche am Ende verarbeitet werden müssen, bestimmt. Das Modul besitzt einen Prozess, welcher als `SC_METHOD` implementiert ist. Dieser Prozess verläuft synchron, taktgesteuert mit steigender Flanke am Port „`clk`“. Er führt eine sequentielle AND-Operation (*„reduced-AND“*) auf alle Eingangssignale aus, um das

Gesamtergebnis zu ermitteln. Nur wenn alle Ergebnisse, gegebenenfalls durch Verwendung der Maske („mask“-Port nach Abb. 32), logisch wahr sind, wird für einen Zyklus das Ergebnis am „sync“-Port *High*, um den Signalempfänger zu benachrichtigen. Die Maske für den Vergleich wird bei jedem Zyklus neu vom „mask“-Port eingelesen und nicht lokal im Modul gepuffert.

```
//! \tparam T  Valid signal type of PEs
//! \tparam N  Number inputs
/*****************************************************************************/
template<typename T, uint32_t N>
class Synchronizer;
```

Code 24: SystemC-Klasse – Synchronizer (Synchronization Unit)

4.5.2 Zusätzliche Teilkomponenten

Diese Komponenten sind Teil der Gesamtarchitektur und teilweise Einschränkungen der Beschreibungsmöglichkeiten von *SystemC* geschuldet. So ist es beispielsweise nicht möglich einzelne Bits aus einem Gesamtbitstrom durch *Slicing*[13] beim *Binding* der Architektur auszuwählen. Dies muss innerhalb eines Modules durch einen simulierbaren Prozess erfolgen. Im Folgenden werden die notwendigen Teilkomponenten und deren Funktionsweise kurz beschrieben.

4.5.2.1 Slicer

Dieses Modul ist als *Template* geschrieben. Es wird genutzt, um einen Konfigurationsbitstrom, beispielsweise den für die Prozesselemente, in mehrere **gleich große** Teile zu zerlegen. Die Schaltung arbeitet als einfaches Schaltwerk und führt ein *Slicing* des Eingabebitvektors „config_input“ aus. Er hat somit keinen Einfluss auf die Verarbeitungsgeschwindigkeit. In Code 25 ist eine Dummy-Implementierung im direkten Vergleich mit einer VHDL-Implementierung dargestellt.

In der Gesamtarchitektur wird diese Komponente genutzt, um die Konfiguration für die Prozesselemente in Teilstücke aufzuteilen und anschließend diesen zuzuteilen. Die Prozesselemente haben auf Grund der Verwaltung in der SystemC-Simulation als Array alle die gleichen Operationen und demnach auch die gleiche Anzahl an

[13] *Slicing*: engl. “In Scheiben schneiden.” Beschreibt in der Informationstechnik einen Prozess, bei dem Teile eines Datenvektors/-arrays als Block aus diesem herausgeschnitten werden.

Konfigurationbits. Werden in einer Variante des *Virtual Coarse Grained Reconfigurable Arrays* Prozesselemente mit unterschiedlichen Operationen verwendet, sodass ein einfaches *Slicing* nicht mehr genügt, ist das Modul aus Abschnitt 4.5.2.2 zu verwenden.

```
/*VHDL Implementation*/
signal configuration: std_logic_vector(15 downto 0);
signal firstpart: std_logic_vector(7 downto 0);
signal secondpart: std_logic_vector(7 downto 0);

firstpart <= configuration(15 downto 8);
secondpart <= configuration(7 downto 0);

/*SystemC Implementation*/
#include <systemc>
#include <cstdint>
#include "Slicer.h"

constexpr uint8_t config_bw = 16;
constexpr uint8_t num_parts = 2;
constexpr uint8_t part_bw = 8;

sc_core::sc_signal<sc_dt::sc_lv<config_bw>> configuration("configuration");
sc_core::sc_signal<sc_dt::sc_lv<part_bw>> firstpart("firstpart");
sc_core::sc_signal<sc_dt::sc_lv<part_bw>> secondpart("secondpart");
cgra::Slicer<sc_dt::sc_lv<config_bw>, num_parts, config_bw>
mySlicer("mySlicer");

mySlicer.config_input.bind(configuration);
mySlicer.config_parts[0].bind(firstpart);
mySlicer.config_parts[1].bind(secondpart);
```

Code 25: VHDL – Bit-Slicing Beispiel – Implementierung in SystemC

4.5.2.2 Selector

Der *Selector* ist ebenfalls als *Template* programmiert. Er dient dazu, eine Bitsequenz aus einer Eingangsbitsequenz zu entnehmen, sprich zu selektieren. Das Modul kann verschieden eingesetzt werden: Zum einen kann damit eine Bitsequenz aus einem Bitstrom herausgenommen werden, zum anderen kann auch ein Bereich eines Bitstroms in mehrere Bitsequenzen zerlegt werden. Im Gegensatz zum *Slicer* wird hier also nicht der gesamte Bitstrom am Eingang in gleiche Teile zerlegt, sondern zuvor kann der Bereich selektiert werden. Auch für dieses Element ist in Code 26 ein Beispielcode in VHDL und seine Umsetzung in *SystemC* angegeben.

Der *Selector* bekommt bei seiner Instanziierung neben der Anzahl und Größe eines Bitstrom-Ausschnitts sowie der Gesamtausschnittgröße auch noch das Startbit für den Ausschnitt übergeben. Im Beispiel nach Code 26 wären dies:

- Anzahl der Parts: `num_parts`
- Bitbreite pro Part: `part_bw`
- Größe des Bitstromausschnitts, der zerlegt werden soll: `range_bw`
- Startadresse des ersten Bits: `start_bit` (nullbasierte Adressierung)

Das *Template* berechnet sich aus diesen Werten selbst die weiteren Adressen der einzelnen Parts und zerlegt den Bitstrom entsprechend den Angaben ab dem Startbit.

```
/*VHDL Implementation*/
signal configuration: std_logic_vector(63 downto 0);
signal firstpart: std_logic_vector(7 downto 0);
signal secondpart: std_logic_vector(7 downto 0);

firstpart <= configuration(25 downto 18);
secondpart <= configuration(33 downto 26);

/*SystemC Implementation*/
#include <systemc>
#include <cstdint>
#include "Selector.h"

constexpr uint8_t config_bw = 64;
constexpr uint8_t num_parts = 2;
constexpr uint8_t part_bw = 8;
constexpr uint8_t range_bw = 16;
constexpr uint8_t start_bit = 18;

sc_core::sc_signal<sc_dt::sc_lv<config_bw>> configuration("configuration");
sc_core::sc_signal<sc_dt::sc_lv<part_bw>> firstpart("firstpart");
sc_core::sc_signal<sc_dt::sc_lv<part_bw>> secondpart("secondpart");
cgra::Selector<sc_dt::sc_lv<config_bw>, num_parts, part_bw, range_bw>
mySelector("mySelector", start_bit, config_bw);

mySelector.config_input.bind(configuration);
mySelector.config_parts[0].bind(firstpart);
mySelector.config_parts[1].bind(secondpart);
```

Code 26: VHDL – Bit-Slicing Beispiel – Selector Implementierung in SystemC

Möchte man nur einen Bereich aus einem Bitstrom ausschneiden, legt man nur einen Ausgangspart an und gibt als Bitbreite pro Part und Ausschnitt die Größe des gewünschten Ausschnitts an. Ab der Startbitadresse wird dann der gesamte Bereich als ein Bitstrom ausgeschnitten und zurückgegeben. Durch mehrere Instanzen des

*Selector*s mit dem gleichen Eingangsbitstrom lassen sich damit auch überlappende Selektionen realisieren. Die Schaltung arbeitet als einfaches Schaltwerk, das bedeutet keine zyklischen Verzögerungen bei einer Schalthandlung und keine Zustandsspeicherung.

4.5.2.3 General (1-to-N) Demultiplexer

Diese *Template*komponente arbeitet wie ein Standard Demultiplexer-Schaltwerk. Das bedeutet, hier gibt es keine zyklischen Verzögerungen bei einer Schalthandlung und keine Zustandsspeicherung. Die Schaltung verteilt ein Eingangssignal „`input`“ an beliebig viele Ausgangssignale „`outputs`“. Zur Auswahl des richtigen Ausgangs wird der „`select`“-Port verwendet.

4.5.2.4 General (N-to-1) Multiplexer

Diese *Template*komponente arbeitet wie ein Standard Multiplexer-Schaltwerk. Das bedeutet, auch hier gibt es keine zyklischen Verzögerungen bei einer Schalthandlung und keine Zustandsspeicherung. Die Schaltung fasst mehrere Eingangssignale „`inputs`“ als ein Ausgangssignal „`output`“ zusammen. Zur Auswahl des richtigen Ausgangs wird der „`select`“-Port verwendet.

4.5.3 Beschreibung der Gesamtarchitektur

Alle aufgeführten Komponenten werden zu einer Gesamtarchitektur zusammengesetzt. In Abb. 62 ist diese Gesamtarchitektur einmal schematisch abgebildet.

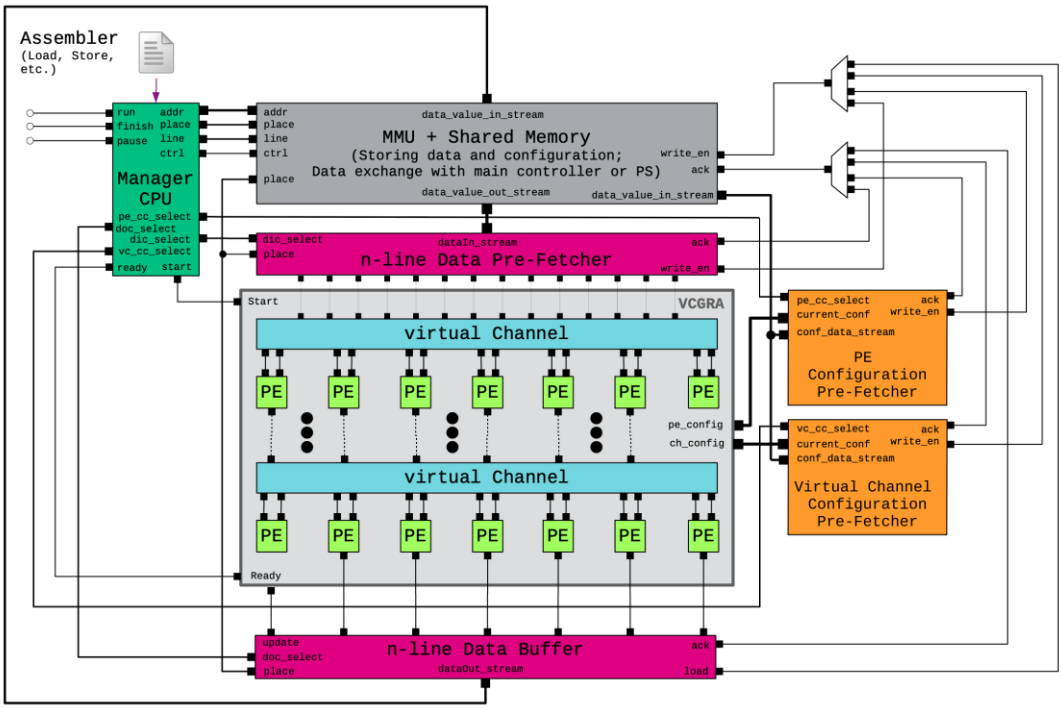

Abb. 62: Übersicht zur erweiterten VCGRA-Architektur

Das Zentrum der Architektur bildet das VCGRA aus den drei Komponenten Prozesselemente, virtuelle Kanäle und *Synchronization Unit*. Direkt mit dem VCGRA verbunden sind die *Pre-Fetcher* für die Eingangsdaten und die Konfigurationen für die drei Grundelemente des *Virtual Coarse Grained Reconfigurable Arrays*. Am Ausgang des *Virtual Coarse Grained Reconfigurable Arrays* ist der *Data-Output* Puffer angekoppelt. Die *Pre-Fetcher* und Puffer erhalten ihre Daten im Zusammenspiel mit der *Memory Management Unit*. Das Verhalten der *Memory Management Unit* und die Steuerung des *Virtual Coarse Grained Reconfigurable Arrays* wird über die *Central Control Unit* (in Abb. 62 Manager CPU) organisiert und gesteuert. Die Funktionsweise der *Central Control Unit* wiederum wird über das Assemblerprogramm im Machinencode-Boinärformat organisiert. Das Starten, Pausieren oder Stoppen der Gesamtarchitektur erfolgt durch die entsprechenden Portanbindungen „`run`“, „`finish`“ und „`pause`“ an der CCU und wird mit einem externen Prozessorsystem verbunden und gesteuert.

Die Verteilung der Konfigurationsbitströme an die einzelnen Komponenten ist in SystemC durch die zusätzlichen Komponenten *Slicer* und *Selector* gelöst, deren Einsatz in Abb. 63 abgebildet ist. Auf der rechten Seite, wo sich der *Pre-Fetcher* für die Prozesselemente befindet, wird der *Slicer* („`m_pe_config_slicer`“) eingesetzt, da die Prozesselemente wie bereits mehrfach erwähnt zur einfacheren Programmierung ohne Codegenerator als Array vom gleichen Datentypen verwaltet werden.

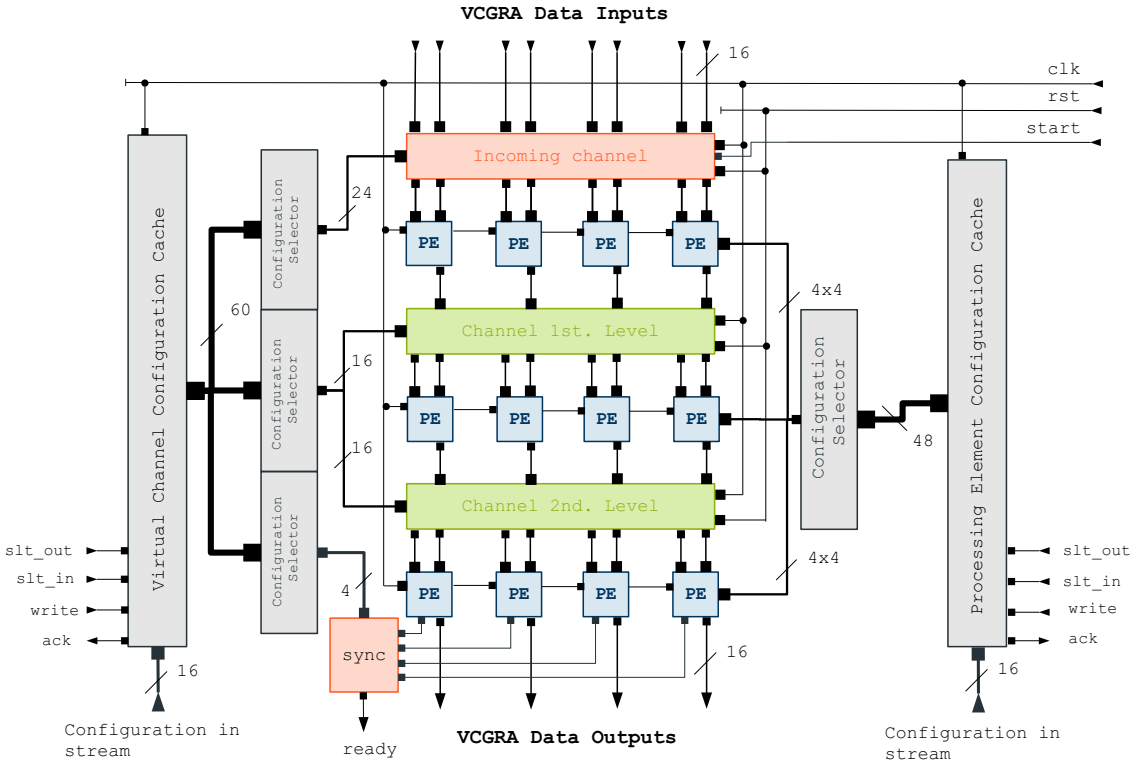

Abb. 63: VCGRA und dessen Konfigurationsinfrastruktur

Code 28 zeigt den Aufbau des *Virtual Coarse Grained Reconfigurable Arrays* aus den Komponenten nach Abb. 63. Zu erkennen ist die Verwendung des *Vectors* für die Prozesselemente „`m_pe_instances`“. Der Datentyp „`pe_type_t`“ wird bestimmt durch die Konfigurationen nach Code 21. Die Verwendung des SystemC-*Vectors* anstatt des Standard-*Vectors* bringt den Vorteil, die Elemente des *Vectors* mit einem eigenen so genannten „*Creator*“ zu erzeugen. Dieser *Creator* ist ein *Functor*, der den Konstruktor der Elemente mit beliebigen Parametern ausführen kann. Die Funktionalität wird verwendet, um den Prozesselementen jeweils eine eigene Identifikationsnummer mitzugeben.

```
struct pe_creator
{
public:
    pe_creator(uint32_t pe_id) : m_pe_id(pe_id) {};
    pe_type_t* operator()(const char* name, size_t size)
    {
        return new pe_type_t(name, m_pe_id++);
    }

    uint32_t m_pe_id;
};
```

Code 27: Creator für einen Vector von Prozesselementen

Die Identifikation wird vorwiegend verwendet, um bei der Ausgabe der Simulation die Prozesselemente eindeutig zu unterscheiden. Falls die einfache Signalbindung zum Datentransport zwischen den Komponenten den Ansprüchen an die Architektur nicht mehr genügen, kann diese ID auch zur Adressierung einer Nachricht in einem Bussystem genutzt werden, da sichergestellt ist, dass die Identifikation bei dieser Methode nur einmalig vergeben wird.

```
    //VCGRA component instances
    sc_core::sc_vector<pe_type_t> m_pe_instances{"VCGRA_PEs"};
    // Array of PE instances of the current VCGRA
    input_channel_type_t m_input_channel{"Input_Channel"};
    // VirtualChannel instance of the first level
    sc_core::sc_vector<channel_type_t> m_channel_instances{"VCGRA_Channels"};
    // Array of VirtualChannel instances of the current VCGRA
    synchronizer_type_t m_sync{"VCGRA_Sync"};
    // Synchronizer for VCGRA ready signal generation.
    slicer_type_t m_pe_config_slicer{"PE_config_Slicer"};
    // Distributes PE configuration bitstream to PE instances
    in_ch_config_selector_type_t m_input_channel_selector{
        "In_Channel_Selector",
        0,
        cgra::cVChConfigBitWidth};
    // Selector to distribute VirtualChannel configuration to input channel.
    ch_config_selector_type_t m_channel_selector{
        "Channel_Selector",
        24,
        cgra::cVChConfigBitWidth};
    // Selector to distribute VirtualChannel configurations to general
channels.
    sync_selector_type_t m_sync_selector{
        "Sync_Selector",
        72,
        cgra::cVChConfigBitWidth};
    // Selector to distribute synchronization mask to general Synchronizer.
```

Code 28: VCGRA Instanz in SystemC

Da sich der virtuelle Kanal am Eingang des *Virtual Coarse Grained Reconfigurable Arrays* von den anderen virtuellen Kanälen unterscheidet, ist dieser als eigener Type instanziiert. Die verbleibenden virtuellen Kanäle werden durch die homogene Struktur des *Virtual Coarse Grained Reconfigurable Arrays* wieder gemeinsam in einem *Vector* instanziiert und verwaltet. Die *Selectoren* erhalten jeweils als Eingang die Konfigurationen für die virtuellen Kanäle. Allerdings unterscheiden sie sich selbst in der Anzahl ihrer Ausgänge, der Bitbreite der Ausgänge und dem Startbit, ab welchem die Selektion der Konfiguration beginnt. Aus diesem Grund gibt es drei verschiedene Instanzen: Einen für den Eingangskanal, einen für die weiteren „regulären“ virtuellen Kanäle und einen für die „*Synchronization Unit*“, deren Maske an das Ende der Konfiguration für die virtuellen Kanäle angehängt wird.

In Code 29 werden die Instanzen der Komponenten der Gesamtarchitektur aufgeführt. Die Komponenten sind in gleicher Weise in Abb. 62 aufzufinden.

```
//Components
cgra::VCGRA vcgra{"vcgra"};
//!< VCGRA instance within the architecture
cgra::data_input_cache_type_t data_in_cache{"data_in_prefetcher"};
//!< VCGRA data value input cache.
cgra::data_output_cache_type_t data_out_cache{"data_out_buffer"};
//!< VCGRA data value output cache.
cgra::pe_config_cache_type_t pe_confCache{"pe_confic_prefetcher",
cgra::cPeConfigBitWidth};
//!< ConfigurationCache instance for ProcessingElement configuration.
cgra::ch_config_cache_type_t ch_confCache{"ch_confic_prefetcher",
cgra::cVChConfigBitWidth};
//!< ConfigurationCache instance for VirtualChannel and Synchronizer
configuration.
cgra::MMU mmu{"mmu", cgra::cCacheFeatures};
//!< MMU instance within the architecture.
cgra::ManagementUnit mu;
//!< ManagementUnit instance within the architecture.
cgra::General_DeMux<cgra::cache_write_type_t, 4, 3> we_dmux{"we_dmux"};
//!< Demultiplexer to contribute MMU write enable signal to all available
caches.
cgra::General_Mux<cgra::cache_ack_type_t, 4, 3> ack_mux{"ack_mux"};
//!< Multiplexer to collect acknowledge signals from all available caches.
```

Code 29: SystemC-Komponenten für die erweiterte Architektur

Die *Achknowledge*- und *Write-Enable*-Signale der *Memory Management Unit* werden über einen Multiplexer bzw. Demultiplexer geführt. Dadurch können Ports an der MMU gespart werden. Durch die *Central Processing Unit* werden die beiden Komponenten entsprechend umgeschaltet, um die Signale des gerade adressierten *Pre-Fetcher* oder Puffer zu verwenden. Aus diesem Grund ist ein paralleler Zugriff auf mehrere *Pre-Fetcher* derzeit nicht möglich.

4.5.4 Verbesserungskonzepte

Die Struktur des *Virtual Coarse Grained Reconfigurable Arrays* ist sehr statisch. Es wird derzeit nicht das volle Potential der Architektur und deren Flexibilität ausgenutzt. Die Verwendung von Arrays und Vektoren für die Verwaltung der Komponenten erfordert durch das Typensystem von C++ gleiche Eigenschaften für alle Elemente des Daten-Containers. Aus diesem Grund wäre die Erstellung eines Codegenerators vorteilhaft, der alle Komponenten mit ihren Eigenschaften automatisch einzeln instanziieren kann. Dadurch könnten beispielsweise Prozesselemente mit unterschiedlichen Eigenschaften oder auch inhomogene Strukturen in der Architektur selbst durch

Verwendung unterschiedlicher virtueller Kanäle erzeugt werden. Von Hand ist diese Arbeit allerdings für umfangreichere Strukturen unnötig mühsam.

Die *Pre-Fetcher* ermöglichen es, parallel zu einer Verarbeitung im VCGRA, Konfigurationen oder Daten zwischen *Memory Management Unit* und *Pre-Fetcher* zu übertragen, allerdings verhindert sie nicht eine überflüssige mehrfache Übertragung, bspw. wenn die richtige Konfiguration bereits geladen wurde. Ein solches *Caching* würde die Autonomie der Architektur und deren Einsatzmöglichkeiten erhöhen. Dies würde aber eine Erweiterung der Intelligenz der *Central Control Unit* verlangen, denn es müssten entsprechende *Cache-Tables* geführt werden, welche bereits geladene Inhalte erkennen.

Um den Übertragungsaufwand weiter zu reduzieren wäre es sinnvoll, das Auswechseln von Bitsegmenten einer Konfiguration zu erlauben. Derzeit können nur vollständige Konfigurationen zwischen MMU und *Pre-Fetcher* ausgetauscht und übertragen werden. Bei umfangreichen Architekturen können dies gleich mehrere hundert Bit sein. Ist die Übertragungsbitbreite zwischen *Memory Management Unit* und *Pre-Fetcher* schmal gewählt, um damit Chip-Fläche einzusparen, kann die Übertragung der Konfiguration wieder zu einem Flaschenhals werden. Zur Unterstützung dieser Funktionalität müsste der Assembler entsprechend um unterstützende Befehle erweitert werden.

Die *Memory Management Unit* selbst und deren verwaltete Speicherregion sollte auf parallele Speicherzugriffe erweitert werden. Daraus ergeben sich dedizierte Regionen im *Shared Memory* für Konfigurationen, Eingangsdaten, Ausgangsdaten und temporäre Daten während der Verarbeitung (siehe Abb. 45). Das Interface der MMU müsste erweitert werden um Ports für die parallele Synchronisation der Speicherzugriffe der aufgezählten Regionen.

5 Evaluation der erweiterten Architektur

Die Evaluation der Architektur bzw. deren Entwicklung über einzelne Entwicklungsstufen erfolgt mit Hilfe von McPAT [40] in drei Schritten. Zunächst wird eine Abschätzung in McPAT erstellt, die nur das VCGRA – die Prozesselemente, die virtuellen Kanäle und die *Synchronization Unit* – enthält. Diese wird verglichen mit der Implementierung auf einem FPGA in VHDL (siehe Abschnitt 2.3.6) und dient als Basis für die weiteren Abschätzungen in McPAT. Anschließend wird die Architektur um zwei weitere Entwicklungsschritte erweitert bis zu ihrer vollständigen Implementierung nach Abschnitt 4.5 und die Veränderungen im Vergleich zur Basisimplementierung analysiert. Zuletzt erfolgt eine Einordnung der eigenen Architektur im Vergleich zu anderen Architekturen der wissenschaftlichen Community.

5.1 Anpassungen an McPAT zur Abschätzung der VCGRA-Architektur

McPAT ist ein Werkzeug zur Simulation von Multicore bzw. Manycore Prozessoren. Die Einstellungen werden über XML-Dateien vorgenommen, auf denen statische wie auch dynamische Eigenschaften der Architektur beschrieben werden. Zu den statischen Eigenschaften der Architektur gehören beispielsweise das Vorhandensein und die Größe der Architekturkomponenten wie auch deren Verbindung per Adress- und Datenbus. Zu den dynamischen Eigenschaften gehören beispielsweise aktive und inaktive Zyklen im Gesamtprozess, Speicherzugriffe, *Cache*zugriffe oder auch welche Komponenten eines *Cores* aktiv sind.

Die Architektur, die McPAT abbildet, entspricht nicht direkt den Voraussetzungen zur Simulation der VCGRA-Architektur. Daher beschreibt dieser Abschnitt Annahmen und Anpassungen an McPAT, um die Abschätzung mit diesem Tooling zu ermöglichen. Die Bestrebungen, andere Werkzeuge, wie PowerSC [41] oder Powersim [42] zu verwenden, scheitern zum einen an einer fehlenden Abschätzung der Chipfläche durch diese Werkzeuge und zum anderen basieren sie auf einer veralteten SystemC-Version 2.2.0, welche nicht direkt kompatibel zur verwendeten Version 2.3.x ist. Auch TLM POWER3 [43] kann nicht verwendet werden, weil es sich zusätzlich nicht auf Register-Transfer-Ebene, sondern auf eine *Transaction Level* Simulation reduziert. Das Modell der VCGRA-Architektur ist aber nicht auf TLM-Basis erstellt.

Für die Abbildung der VCGRA-Architekturkomponenten werden in McPAT die folgenden verfügbaren Elemente genutzt: *Cores*, *L3-Cache*, *NoCs* und *Memory Controller*.

5.1.1 Cores

Die *Cores* werden verwendet, um die Prozesselemente, die *Synchronization Unit* und die *Central Control Unit* abzubilden. Die statischen Eigenschaften der *Cores* für die Prozesselemente werden stark reduziert. Es werden alle zusätzlichen Elemente wie *Floating Point Units*, *Hardware Mulitplier*, *Branch Prediction Unit* oder *Data/Instruction Caches* ausgeschaltet. Der Prozessor arbeitet nur *in-order und das Pipelining* ist ausgeschaltet. Die Verarbeitungseinheit wird auf genau eine ALU für Ganzzahloperationen reduziert, so wie es auch für die Prozesselemente designed ist.

Der *Core* für die *Synchronization Unit* ist auf die gleiche Weise parametriert, da seine Aufgabe – Verarbeitung von Eingangsdaten mit Bool'chen Operationen – als ALU-Operation am besten abgebildet wird. Die *Central Control Unit* unterscheidet sich nicht grundsätzlich in ihrem Aufbau von den anderen *Cores*. Es wird ein Puffer für *Decoding* und *Fetching* eingebaut, da die CCU diese Stufen der Verarbeitung in ihrem Zustandsautomaten besitzt. Die verfügbaren Parameter des McPAT Toolings lassen keine weiteren Möglichkeiten für eine bessere Abgrenzung zu, obgleich die Aufgabe der *Central Control Unit* sich doch stark von der eines Prozesselementes unterscheidet und auch als Modul in SystemC ist der interne Aufbau im Vergleich zum Prozesselement wesentlich umfangreicher.

5.1.2 NoC (Network-on-Chip)

Die *Network-on-Chip* werden eingesetzt, um die virtuellen Kanäle der Architektur zu simulieren. Dafür werden zunächst zwei Ansätze verfolgt und mit der Implementierung auf dem FPGA mittels VHDL verglichen.

Zunächst wird das Prinzip von Micro-Routern untersucht. Dazu wird jedem Eingang eines Prozesselementes ein eigenes *Network-on-Chip* zugeteilt. Jedes NoC hat dabei genau einen Ausgang, aber *N*-Eingänge. Die Anzahl *N* ist dabei abhängig von der Anzahl der Prozesselemente des vorhergehenden Levels. Jedes *Network-on-Chip* hat eine Verbindung zu den Micro-Routern des gleichen Levels durch horizontale Knoten (*horizontal Nodes*) und jeweils zu seinem direkten Vorgänger und Nachfolger über vertikale Knoten (*vertical Nodes*). Durch die vertikalen und horizontalen Verbindungen zu diesen Elementen ergeben sich die *N*-Eingänge und der eine Ausgang.

Stellt man sich die Struktur der Prozesselemente des *Virtual Coarse Grained Reconfigurable Arrays* als Raster vor, so besitzt ein NoC eine direkte Verbindung zu den Prozesselementen in der gleichen Spalte über die vertikalen Knoten. Die Verbindung zu den anderen Prozesselementen außerhalb der eigenen Spalte erfolgt

über die horizontale Verbindung zu den Micro-Routern (siehe auch Abb. 36). Bei diesem Ansatz liegt der Anteil der Chipfläche für die virtuellen Kanäle in der Approximation durch die *Network-on-Chip* mit McPAT bei ca. 25% der Gesamtfläche. Zum Vergleich, der Anteil der Chipfläche für die virtuellen Kanäle auf dem FPGA an der Gesamtfläche lag bei nur 3%. Während ein virtueller Kanal im VCGRA aus Multiplexern modelliert ist, besitzen NoCs doch ein wesentlich komplexeres Design. Die Summe der *Network-on-Chip* für diesen Ansatz ergibt sich vereinfacht angenommen zu:

$$\sum Noc = \sum \left(inputs + \sum_{lvl=1}^{n} PEs \right) \tag{15}$$

Durch die große Anzahl an NoCs mit der höheren Komplexität im Vergleich zu einem einfachen Multiplexer, welche wiederum direkten Einfluss auf die benötigte Chipfläche pro NoC hat, wird ein zu großer Flächenbedarf approximiert. Aus diesem Grund wird von diesem Prinzip Abstand genommen und ein neuer Ansatz verfolgt.

Bei diesem Ansatz entspricht ein *Network-on-Chip* einem virtuellen Kanal in einer Ebene des *Virtual Coarse Grained Reconfigurable Arrays*. Ein NoC hat so viele Eingangsports wie es Prozesselemente vor dem virtuellen Kanal gibt und einen Ausgangsport. Die Anzahl der vertikalen Knoten ist die Summe der vorhergehenden und nachfolgenden Prozesselemente eines virtuellen Kanals. Für den virtuellen Kanal am Eingang des *Virtual Coarse Grained Reconfigurable Arrays* werden jeweils zwei Eingangswerte als „*virtuelles Prozesselement*“ zusammengefasst. Allerdings wird jeder Eingangswert in das VCGRA als eigener Inputport in der Konfiguration mit aufgenommen. Auf horizontaler Ebene gibt es keine Verbindungen zu anderen Knoten. Da aber für McPAT der Wert nicht Null sein darf, ist er auf „1“ gesetzt.

Bei der Simulation in McPAT ergeben sich gerundet auf drei Nachkommastellen für die Gesamtfläche des *Virtual Coarse Grained Reconfigurable Arrays* 0.634mm^2 und für die *Network-on-Chip* 0,039mm^2. Das Verhältnis der NoC-Fläche zur Gesamtfläche beträgt damit ca. 6%. Der Fehler zur Implementierung auf einem FPGA hat sich damit um 76% reduziert. Für eine gezielte Optimierung ist es notwendig, durch weitere Implementierungen auf einem FPGA und passender Konfigurationen für McPAT zahlreiche Stichproben zu generieren, um daraus einen Korrekturfaktor für die Fläche ergänzend abzuleiten. Anhand dieser einen Stichprobe ist die Festlegung eines solchen Faktors im Rahmen der Dissertation nicht möglich, für eine grobe Abschätzung ist diese Abstraktion dennoch ausreichend.

5.1.3 L3-Caches

Mit Hilfe der *Caches* werden die *Pre-Fetcher* und Puffer abstrahiert. Für die Kapazität dieser Komponenten im Bereich einiger Bytes ist McPAT nicht ausgelegt. Daher wird die Gesamtgröße („*capacity*“) des *L3-Caches* auf die Summe aller Kapazitäten aller *Pre-Fetcher* und Puffer in der Gesamtarchitektur gesetzt:

$$\text{round\_up}\left(\text{L3}_{\text{cap.}} = \sum_{\text{Pre-F.}} size\right)_{\min(\log_2)} \tag{16}$$

Dabei wird jeweils auf den nächsten Wert aufgerundet, der ein Ganzzahlergebnis zum Logarithmus zur Basis 2 aufweist. Werden zum Beispiel rechnerisch 88 Byte an Speicher für die Summe der *Pre-Fetcher* benötigt, würde die simulierte Kapazität in McPAT auf 128 Byte (gleich 2^7) gesetzt. Grund ist, dass konstruktiv in der Informationstechnik verfügbare Speichergrößen auf Potenzen zur Basis Zwei (binäres System) beruhen. Das verursacht zwar eine Überabschätzung in der Simulation, orientiert sich aber näher an der Realität der verfügbaren Hardwaretechnologie.

5.2 Abschätzung der Dynamiken der Architekturkomponenten

Für die Abschätzung der Performanz wird über SystemC ein zyklenakkurates Modell erstellt. Über die Taktrate der Simulation kann die Gesamtdauer der Verarbeitung abgeschätzt und über Diagramme kann zu jedem Zeitpunkt der Simulation der Zustand auf den Signalleitungen der Komponenten erfasst werden. Aus diesen Daten lässt die Performanz der Gesamtarchitektur und deren Komponenten ableiten.

Für die Berechnungen des Energieverbrauchs benötigt McPAT Statistiken aus der SystemC-Simulation der VCGRA-Architektur, dafür werden in die Architektur Zähler für Zyklen eingebaut sowie für lesende oder schreibende Speicherzugriffe. Für die verarbeitenden Komponenten des McPAT Toolings, *Cores*, *Network-on-Chip* und *Memory Controller* sind „*busy*“- und „*idle*“-Zähler implementiert. Diese werden entsprechend in die Zustandsmaschinen der Komponenten in der SystemC VCGRA-Architektur integriert und liefern nach der Simulation in SystemC die entsprechenden Werte für die Simulation in McPAT. Ein Beispiel einer solchen Implementierung zeigt Code 30 für die *Synchronization Unit*.

```
#ifdef MCPAT
    /* A synchronizer always updates its input and output buffer states.
     * Thus the component is always busy and has no idle state.
     */
    ++m_totalCycles;
    ++m_busyCycles;
#endif
```

Code 30: McPAT-Statistiken von der SystemC-Architektur

Bei den *Pre-Fetcher*n und Puffern sind Zähler für *Cache-Hits* und *Cache-Misses* in Verwendung. Als *Hit* wird jeweils ein Lesen aus einem *Pre-Fetcher* oder Puffer gewertet, als *Miss* ein Schreiben in einen *solchen*. Für diese Komponenten sind die Zähler in einer eigenen Klasse gekapselt, da der grundsätzliche Aufbau von ihnen sich stark ähnelt und damit die Funktionalität für alle gleich ist. Auch hier ist in Code 31 exemplarisch ein Code-*Snipped* für den *Data-Input Pre-Fetcher* dargestellt.

```
class DataInCache : public sc_core::sc_module
#ifdef MCPAT
  , protected cgra::McPatCacheAccessCounter
#endif
{
  void storeValueInCacheLine()
  {
#ifdef MCPAT
    ++this->m_writeAccessCounter;
#endif
    ...
  }
  void switchCacheLine()
  {
#ifdef MCPAT
    ++this->m_readAccessCounter;
#endif
    ...
  }
  ...
}
```

Code 31: McPAT-Statistiken von der SystemC-Architektur

Alle Komponenten der VCGRA-Architektur mit McPAT Statistiken schreiben ihre aufgenommenen Zählwerte nach der Simulation in eine Datei separiert nach ihrem

Namen und der Art des Zählers. Diese Statistiken müssen dann derzeit noch von Hand für die Komponenten in McPAT eingetragen werden. Ein Auszug ist exemplarisch im Anhang beigefügt (Beispiel: Code 35: McPAT Dynamik Statistik Ausgabe einer SystemC-Simulation).

5.3 Übersicht zu den Implementierungen der Entwicklungsstufen

Für eine Vergleichbarkeit der Ausführungsgeschwindigkeit und der Chipgröße wurde das gleiche Beispiel verwendet wie für die Implementierung auf dem FPGA in VHDL. Für die Simulation wurde die Bildgröße des Eingangsbildes reduziert auf 64x64 Pixel. Dies reduziert die Dauer der Simulation und die Größe des Maschinencodes, welcher durch das Ausrollen der Schleifen im Algorithmus durch den Assembler entsteht. Da dieser Maschinencode Teil des Quellcodes für die Simulation ist, beschleunigt dies wiederum die Übersetzung der Gesamtarchitektur durch den C/C++ Compiler.

Es wird eine Kantendetektion in X-Richtung und in Y-Richtung ausgeführt und diese Ergebnisse überlagert in Form eines Gradientenbildes als Gesamtergebnis zurückgegeben (siehe Abb. 64). Die beiden Bildachsen X und Y stehen orthogonal zueinander (in einem Winkel von 90˚) und bilden die horizontale und vertikale Achse. Das Berechnen der Magnituden der beiden Filteroperationen an einem Bildpunkt des Ergebnisses wird nicht von der VCGRA-Architektur berechnet, weil die Wurzeloperation nicht in Hardware als Operation einer PE implementiert ist.

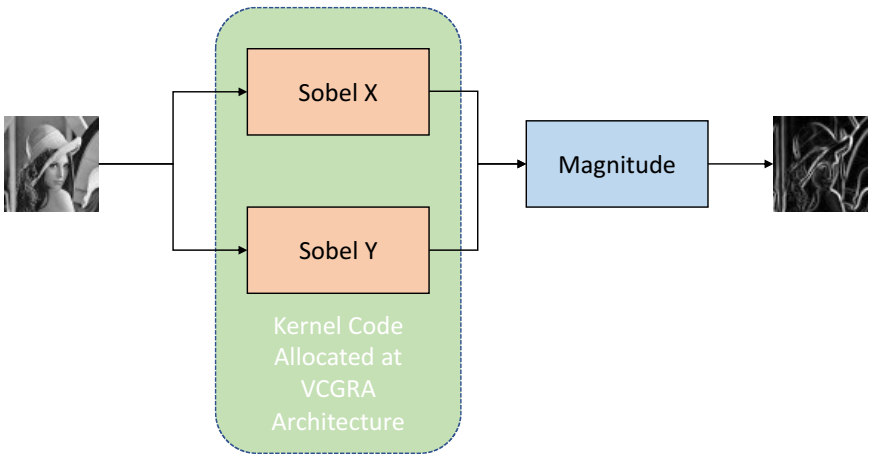

Abb. 64: Übersicht zum Evaluationsbeispiel

Das VCGRA übernimmt dabei die Aufgabe, den Gradienten für jedes Pixel zu berechnen. Das Gesamtergebnis (Magnitude) wird immer von einem angeschlossenen

(simulierten) Prozessorsystem erstellt. Eingangsbild und Ergebnis sind in Abb. 65 und Abb. 66 abgebildet.

Abb. 65: Lena Eingangsbild

Abb. 66: Lena Gradientenbild

Um die Vergleichbarkeit mit den Ergebnissen aus der Implementierung in VHDL für einen FPGA zu verbessern, muss der Einfluss des Prozessorsystems auf dem FPGA abstrahiert werden. Das PS benötigt für die Bereitstellung der Daten für das VCGRA und das Umschalten der Konfigurationen ebenfalls Zeit, welche die Gesamtverarbeitungsdauer des Beispiels beeinflussen. Daher sind Zeiten für einen Informationstransport zwischen Prozessorsystem und der VCGRA-Architektur abgeschätzt aus den Ergebnissen der Implementierung des VHDL-Designs.

Die Zeiten für eine Rekonfiguration lassen sich direkt aus diesen Ergebnissen ablesen und betragen:

$$\text{SwitchPeConfigDelay} = 6.4\mu\text{s} \tag{17}$$

$$\text{SwitchChConfigDelay} = 6.5\mu\text{s} \tag{18}$$

Für die Übertragungsdauer zwischen Prozessorsystem und VCGRA-Architektur wird aus den Ergebnissen der Implementierung in VHDL ein Mittelwert für die Übertragung von einem Byte ausgerechnet. Diese Näherung unterscheidet nicht zwischen der Dauer, welche das Prozessorsystem mit der Ausführung von Befehlen verbringt und der Signalübertragung per Bussystem. Es werden in der Implementierung in VHDL keine anderen Prozesse parallel auf dem PS ausgeführt. Eine Beeinflussung der Übertragungsdauer durch parallele Tasks auf dem Prozessorsystem erscheint daher als unwahrscheinlich. Für diese erste Abschätzung der Ausführungszeit ist diese Approximation somit ausreichend.

Die Verzögerung für die Übertragung eines Bytes wird anschließend in der SystemC-Simulation multipliziert mit der Anzahl der Bytes, die tatsächlich ausgetauscht werden. Bei der gewählten Taktgeschwindigkeit der Implementierung des VHDL-Designs von 5MHz dauert die Übertragung eines Bytes:

$$160\mu s \text{ pro Pixel Berechnungszeit} \tag{19}$$

$$\text{ca. } 50\% \text{ Berechnungszeit} = \text{Datentransfer} \rightarrow 80\mu s \tag{20}$$

$$128 \text{ Byte Konfigurationsdaten in } 80\mu s \rightarrow 625ns/Byte \tag{21}$$

Die 160µs nach Equation (19) ergeben sich als Durchschnitt aus der Gesamtberechnungsdauer des Gradienentenbildes (40s) nach Abschnitt 2.3.6, geteilt durch die Anzahl der verarbeiteten Pixel (498×498). Aus den Ergebnissen nach Abschnitt 2.3.6.4 werden knapp 50% der Zeit zur Berechnung eines Pixelwertes für die Datentransfers benötigt, aus der sich die Angabe nach Equation (20) ergibt. Die Anzahl der zu übertragenen Bytes im VHDL-Design nach Equation (21) für die Berechnung eines Bildpunktes im Gradientenbild ermitteln sich aus den Konfigurationsdaten für die Prozesselemente, virtuellen Kanäle und der Maske der *Synchronization Unit* sowie der Menge an Eingangs- und Ausgangsdaten des *Virtual Coarse Grained Reconfigurable Arrays* wie folgt:

- Maske der *Synchronization Unit*: 4 Byte
- Konfigurationen der Prozesselemente: $2 \times 2 \times 4$ Byte
- Konfigurationen der virtuellen Kanäle: $2 \times 2 \times 4$ Byte
- Pixelwerte des Eingangsbildes: 9×4 Byte
- Filterkoeffizienten des Sobel-Operators: 9×4 Byte
- Temporäre Zwischenergebnisse: 4×4 Byte
- Endergebnis: 4 Byte

Gesamt: 128 Byte

Auf Grund des *AXI4Lite*-Protokolls im Wrapper nach Abschnitt 2.3.5 werden jeweils Vielfache von vier Byte übertragen. Dies wird auch für alle Daten in der SystemC-Simulation so beibehalten, um die Vergleichbarkeit zu verbessern. Die temporären Zwischenergebnisse in der obigen Auflistung werden jeweils zweimal übertragen: Zunächst müssen die Teilergebnisse vom VCGRA an das Prozessorsystem übertragen und dort gespeichert, anschließend zur Berechnung des Endergebnisses wieder zurück an das VCGRA übertragen werden.

Die berechnete Übertragungsdauer für ein Byte zwischen Prozessorsystem und VCGRA ist natürlich eine grobe Schätzung und abhängig von der Taktfrequenz des gesamten Konstrukts aus Prozessorsystem und Bussystem. Für diese erste grobe Schätzung ist es dennoch zulässig, die Taktfrequenz über einen linearen Faktor an andere Ausführungsfrequenzen in der Simulation anzupassen:

$$\text{delay}_{F\text{req.}} = 625\text{ns} \times \frac{5[MHz]}{X[MHz]} \tag{22}$$

5.3.1 VCGRA

Diese erste Entwicklungsstufe entspricht der Nachimplementierung des Designs in VHDL aus Abschnitt 2.3.6.3. Die SystemC-Simulation ist schematisch in Abb. 67 dargestellt. Die VCGRA-*Testbench* approximiert das Verhalten des Prozessorsystems. Für die verbesserte Vergleichbarkeit beinhaltet sie ebenfalls die Verzögerungszeiten aus den Annahmen des einleitenden Abschnitts 5.3.

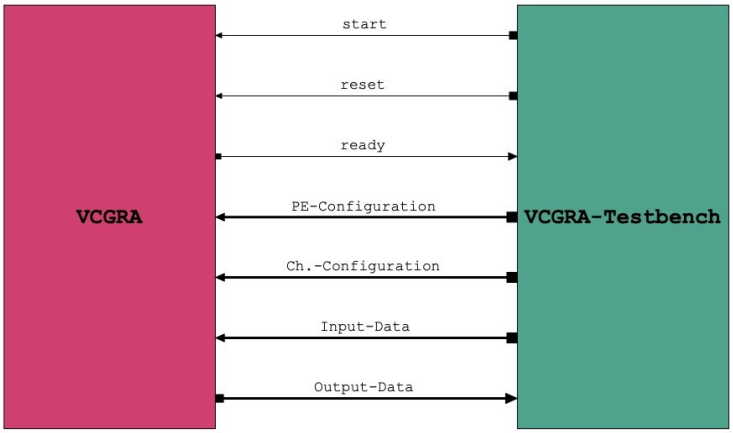

Abb. 67: Evaluation - Simulation nur mit einem VCGRA

Der folgende Ablauf wird für jedes Datenpixel des Gradientenbildes jeweils zweimal durchgeführt:

- Übertragung der ersten Konfiguration für die Prozesselemente und virtuellen Kanäle an das VCGRA
- Sequenzielle Übertragung von vier Pixelwerten und den korrespondierenden Gewichtsfaktoren des Sobel-Filters an das VCGRA
- Berechnung des ersten Zwischenergebnisses und Übertragung an das Prozessorsystem (*Testbench*)
- Sequenzielle Übertragung von vier Pixelwerten und den korrespondierenden Gewichtsfaktoren des Sobel-Filters an das VCGRA
- Berechnung des zweiten Zwischenergebnisses und Übertragung an das Prozessorsystem (*Testbench*)
- Übertragung der zweiten Konfiguration für die Prozesselemente und virtuellen Kanäle an das VCGRA

- Sequenzielle Übertragung von einem Pixelwert und dessen korrespondierenden Gewichtsfaktor sowie der beiden Zwischenergebnisse des Sobel-Filters an das VCGRA
- Berechnung des Gradientenpixels für die entsprechende Filterrichtung und Übertragung an das Prozessorsystem (*Testbench*)

Die häufigen Datenübertragungen dominieren die Gesamtverarbeitungsdauer der SystemC-Simulation und entspricht dem bekannten Verhalten aus der Implementierung in VHDL (siehe auch 2.3.6).

5.3.2 VCGRA und Pre-Fetcher für die PE- und Channel-Konfigurationen

Als erste Stufe der Erweiterung sind *Pre-Fetcher* für die Konfigurationen der Prozesselemente und der virtuellen Kanäle an das VCGRA angeschlossen worden. Der Aufbau der neuen Simulation in SystemC bestehend aus den *Pre-Fetcher*n, demVCGRA und der passenden *Testbench* ist in Abb. 68 dargestellt.

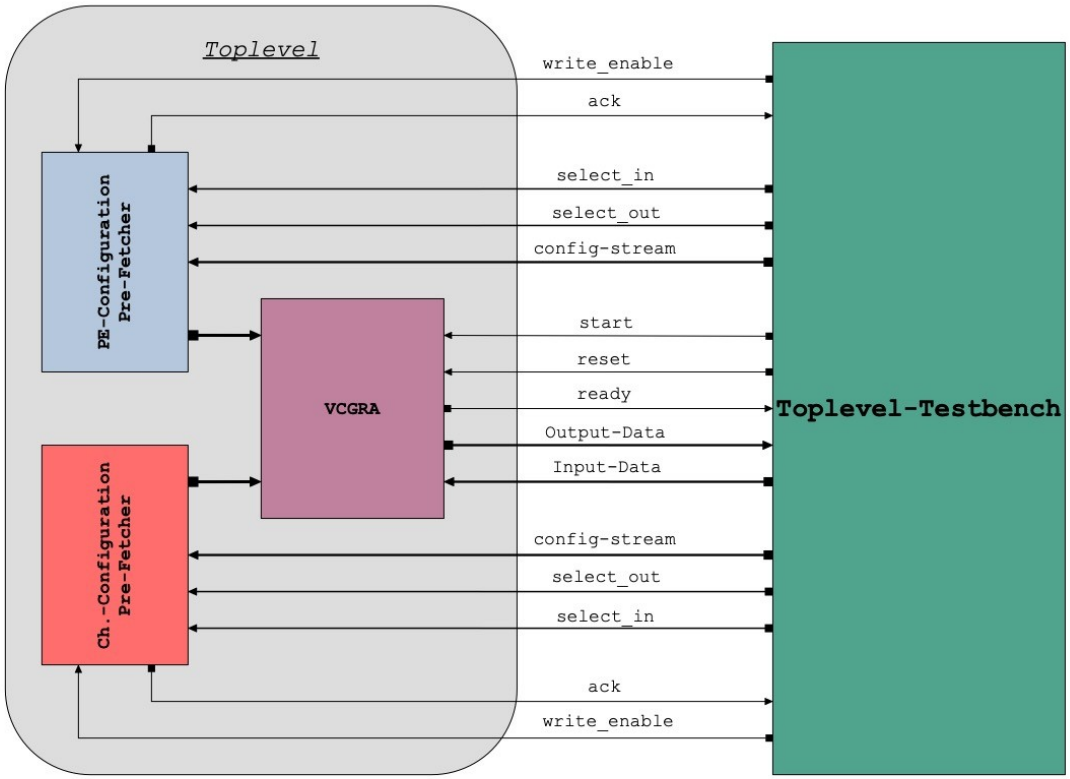

Abb. 68: Evaluation - Simulation mit VCGRA und den Configuration Pre-Fetchern

Augenscheinlich nimmt die Komplexität des Aufbaus zu, denn die Anzahl der Verbindungen zwischen *Testbench* und Architektur hat sich nach Abb. 68 mehr als verdoppelt. Die Kanalbreite (Anzahl der Signalleitungen) zwischen der *Testbench* und dem *Toplevel* hat sich aber reduziert, da die Konfigurationen zwischen *Testbench* und

Toplevel sequenziell übertragen werden können, sodass die Bitbreite für den `config-stream` jeweils reduziert werden kann. Des Weiteren erlauben es die *Pre-Fetcher* während der Laufzeit eine ungenutzte *Line* mit einer neuen Konfiguration zu überschreiben. Während also bereits mit einer Konfiguration für die virtuellen Kanäle und die Prozesselemente gearbeitet wird, kann parallel, sequenziell in mehreren Teilübertragungen, eine weitere Konfiguration vorgeladen werden. Dies spart Verarbeitungszeit der Architektur, die Übertragungszeit jedes Datums im Vergleich zu 5.4.1 bleibt aber konstant, da sich ja die Menge an Daten nicht verändert. . Die Bitbreite der `config-stream` Verbindung bzw. die Anzahl der *Lines* eines *Pre-Fetcher*s und damit verbunden die Bitbreite der `select`-Verbindungen ist ein Designparameter und kann beliebig für eine Applikation ausgewählt werden. Im Beispiel sind zwei Konfigurationen notwendig mit einer maximalen Länge von zehn Byte. Damit ergibt sich für die `select`-Verbindungen eine Bitbreite von Eins, um zwischen den beiden Konfigurationen jeweils umzuschalten. Als Bitbreite für die Übertragung der Konfigurationen zu den *Pre-Fetcher*n wird 8 Bit ausgewählt. Zum einen, um den Einfluss der parallelen Übertragung einer weiteren Konfiguration während der Ausführung zu visualisieren, zum anderen, um sich realen Applikationen mehr anzunähern, da diese deutlich größere Konfigurationen für ein VCGRA aufweisen und daher auch mehr Übertragungen mit üblichen Bitbreiten wie 32, 64 oder 128 Bit notwendig sein können.

Die *Pre-Fetcher* befinden sich auf dem Chip (*Toplevel*) zusammen mit dem VCGRA und können über die `select`-Verbindungen innerhalb eines Taktes umgeschaltet werden. Im Vergleich zum Aufbau nach Abschnitt 0 können somit knapp 99% der Übertragungen für Konfigurationen eingespart werden. Zunächst finden einmalig zu Beginn die folgenden Schritte statt:

- Übertragung der ersten Konfiguration für die Prozesselemente und virtuellen Kanäle an den ersten Slot des jeweiligen *Pre-Fetcher* des *Toplevel*s
- Übertragung der zweiten Konfiguration für die Prozesselemente und virtuellen Kanäle an den zweiten Slot des jeweiligen *Pre-Fetcher* des *Toplevel*s

Anschließend ergibt sich für jede Berechnung eines Pixels des Gradientenbildes nur noch folgender Ablauf jeweils zweimal:

- Auswahl der ersten Konfiguration für die Prozesselemente und virtuellen Kanäle
- Sequenzielle Übertragung von vier Pixelwerten und den korrespondierenden Gewichtsfaktoren des Sobel-Filters an das VCGRA

- Berechnung des ersten Zwischenergebnisses und Übertragung an das Prozessorsystem (*Testbench*)
- Sequenzielle Übertragung von vier Pixelwerten und den korrespondierenden Gewichtsfaktoren des Sobel-Filters an das VCGRA
- Berechnung des zweiten Zwischenergebnisses und Übertragung an das Prozessorsystem (*Testbench*)
- Auswahl der zweiten Konfiguration für die Prozesselemente und virtuellen Kanäle
- Sequenzielle Übertragung von einem Pixelwert und dessen korrespondierenden Gewichtsfaktor sowie der beiden Zwischenergebnisse des Sobel-Filters an das VCGRA
- Berechnung des Gradientenpixels für die entsprechende Filterrichtung und Übertragung an das Prozessorsystem (*Testbench*)

5.3.3 Gesamtarchitektur inklusive Memory Management Unit und Central Control Unit

Die Simulation der Gesamtarchitektur in SystemC ist in Abb. 62 bereits dargestellt. Eine *Testbench* steuert die Anschlüsse für `run`, `finish` und `pause`. Der Datenaustausch findet über das *Shared Memory* statt. Noch vor der eigentlichen Berechnung des Gradientenbildes werden alle notwendigen Daten in das *Shared Memory* geladen, auf welches das Prozessorsystem und die VCGRA-Architektur zugreifen können. Diese Vorgänge werden bei der Berechnung des Gradientenbildes nicht mit betrachtet, da der Zugriff aus der Sicht des Prozessorsystems auf das *Shared Memory* nicht modelliert wurde. Die Anbindung des Prozessorsystems an das *Shared Memory* ist noch nicht definiert. Möglich wäre eine dedizierte Hardware oder ein Zugriff per DMA.

Die volle Gesamtarchitektur arbeitet vollständig autark mit den vorgespeicherten Daten im *Shared Memory* und dem passenden Assemblerprogramm für die *Central Control Unit*. Aus diesem Grund reduziert sich der Ablauf in der *Testbench* auf:

- Laden der Sobel-Parameter in das *Shared Memory*
- Laden des Eingangsbildes in das *Shared Memory*
- Start der Verarbeitung durch *High*-Signal am `run`-Port
- Warten auf eine steigende Flanke am `finish`-Port („Interrupt“)
- Rücklesen der Ergebnisdaten (Gradientenbild) aus dem *Shared Memory*

Auch dieser Ablauf wird jeweils für eine Filterrichtung ausgeführt. Anschließend werden die Teilergebnisse vom Prozessorsystem zu einem Gesamtergebnis zusammengefügt. Das Verteilen der Daten, das Laden und das Wechseln der

Konfigurationen sowie das Abspeichern der Ergebnisse zurück in das *Shared Memory* findet vollständig ohne Einwirkung des Prozessorsystems statt, welches parallel eine andere Tätigkeit durchführen kann und ggf. über einem Interrupt am `finish`-Port die weitere Verarbeitung der Ergebnisse aus dem VCGRA fortsetzt.

5.4 Ergebnisse der Simulationen und Bewertung

In diesem Abschnitt werden die Ergebnisse für jeweils einen Entwicklungsschritt der Architektur aufgeführt und bewertet. Ein Vergleich zwischen den Architekturen erfolgt in Abschnitt 5.5. Es werden Ergebnisse zur Verarbeitungszeit (Performanz), Chipfläche und Verlustleistung zusammengefasst.

5.4.1 VCGRA

Zunächst werden in diesem Abschnitt die Ergebnisse für die Simulation des *Virtual Coarse Grained Reconfigurable Arrays* ohne weitere Anpassungen ausgewertet. Die Ergebnisse dieser Simulation bilden die Basis für den Vergleich zwischen den zusätzlichen Erweiterungen nach Abschnitt 5.3.2 und 5.3.3. Für die Bewertung ist es das Ziel, möglichst genau an die Ergebnisse des VHDL-Designs zu gelangen oder Unterschiede zu diesem Design zu erklären.

Tab. 9: Simulationsergebnisse für das VCGRA

Execution Time	Overall	908,7ms
	Sobel in X-Direction	454,3ms
	Sobel in Y Direction	454,4ms
Area	Total	0,634mm^2
	Area for one PE	0,035mm^2
	Area Synchronization Unit	0,035mm^2
	Area Input Virtual Channel	0,016mm^2
	Area General Virtual Channel (Virtual channels	0,008mm^2

	at over levels of the architecture)	
Power Consumption	Total Peak Power	0,342W
Calculation Time	One Pixel Result Value	115,9µs
	First Temporary	48,5µs
	Second Temporary	26,6µs
	Result Value	40,8µs
VCGRA Execution Time	First Temporary	4,350µs
	Second Temporary	4,300µs
	Result Value	4,350µs
	Median	4,333µs
	Total per Pixel	13,0µs
	Relative Execution Time per Pixel	11,21%

Mit 115,9µs ist die Verarbeitungsgeschwindigkeit zur Berechnung eines Pixels um ca. 31% schneller als die Referenzimplementierung in VHDL. Der Unterschied ergibt sich aus der unterschiedlichen Verarbeitungsgeschwindigkeit der Daten im VCGRA und durch die fehlende Simulation der Verarbeitungsdauer im Prozessorsystem, mit dem das VCGRA verbunden ist. Während in der Referenzimplementierung in VHDL, das VCGRA ca. 53,7µs (siehe *Abb. 44:*) der Verarbeitungsdauer für einen Pixel (ca. 160µs) benötigt, sind es in der Simulation nur 13,0µs. Daraus ergibt sich eine Differenz in der Verarbeitungszeit von 40,7µs, welche die Gesamtdifferenz in der Performanz mehr als 92% dominiert. Das ist folgendermaßen zu erklären: Der Zustandsautomat in der SystemC-Simulation und in der VHDL-Implementierung sind gleich. Im „PROCESS_DATA“-Zustand werden die Berechnungen nach der aktuellen Konfiguration vorgenommen. Da alle verfügbaren Operationen auch im VHDL-Design in der gleichen Zeit bearbeitet sein sollen, bestimmt die aufwendigste arithmetische Operation den kritischen Pfad. Die Divisionsberechnung ist innerhalb eines Taktes

auf der VHDL-Implementierung nicht ausführbar, in der Simulation schon. Da über die Konfiguration einer PE ein Multiplexer zwischen den Ergebnissen der verfügbaren Operationen umschaltet, aber immer alle Berechnungen aller arithmetischen Operationen ausgeführt werden, domminiert der kritische Pfad der Division, obwohl diese gegebenenfalls nicht genutzt wird. Dies führt zu der großen Differenz der Simulationen im Vergleich zur VHDL-Implementierung. Um die Performanzabschätzung zu verbessern, müsste sowohl im VDHL-Design die Division verbessert, als auch in der Simulation in SystemC eine hardwarenähere Implementierung programmiert werden. Für einen Funktionscheck und eine grobe Abschätzung ist die aktuelle Version zunächst aber ausreichend.

Die restliche Differenz zwischen den Gesamtverarbeitungszeiten des VHDL-Designs und der SystemC-Simulation ist die fehlende Betrachtung für die Berechnungen auf dem Prozessorsystem selbst, beispielsweise um die richtigen Pixelkoordinaten zu bestimmen, welche zu der aktuellen Filterposition im Bild passen. Dies ist unabhängig von der Implementierung des *Virtual Coarse Grained Reconfigurable Arrays* und wird daher vernachlässigt.

5.4.2 VCGRA und Pre-Fetcher für die PE- und Channel-Konfigurationen

Die Implementierung erhält als erste Verbesserung die *Pre-Fetcher* für die Konfigurationsdaten der Prozesselemente und der virtuellen Kanäle inklusive Maske der *Synchronization Unit*. Tab. 10 enthält die Übersicht der simulierten und abgeschätzten Messwerte.

Tab. 10: Simulationsergebnisse des VCGRA mit den Configuration Pre-Fetchern

Execution Time	Overall	536,7ms
	Sobel in X-Direction	268,3ms
	Sobel in Y Direction	268,3ms
Area	Total	0,650mm^2
	Area for one PE	0,035mm^2
	Area Synchronization Unit	0,035mm^2

	Area Input Virtual Channel	0,016mm²
	Area General Virtual Channel (Virtual channels at over levels of the architecture)	0,008mm²
	Area Pre-Fetchers (Sum of all used Pre-Fetchers)	0,016mm²
Power Consumption	Total Peak Power	0,345W
Calculation Time	One Pixel Result Value	67,5µs
	First Temporary	24,3µs
	Second Temporary	26,6µs
	Result Value	16,6µs
VCGRA Execution Time	First Temporary	4,200µs
	Second Temporary	4,300µs
	Result Value	4,300µs
	Median	4,266µs
	Total per Pixel	12,8µs
	Relative Execution Time per Pixel	18,96%
Time to Configure	Load PE Configuration to Pre-Fetcher	16,0µs
	Load Virtual Channel Configuration to Pre-Fetcher	20,0µs

	Total Configuration Time	36,0µs

Die relative zusätzliche Fläche für die *Pre-Fetcher* der Architektur bezogen auf die gesamte notwendige Chipfläche beträgt nur 2,4%. Die notwendige Zeit für das Laden einer neuen Konfiguration in einen *Pre-Fetcher* ist abhängig von der Anzahl der zu übertragenen Bytes und der gewählten Bitbreite für die Verbindung zwischen Prozessorsystem und *Pre-Fetcher*. Im simulierten Beispiel ist dies acht Bit, ergo ein Byte. Daher wird die gesamte Konfiguration byteweise übertragen. Laut Equation (22) benötigt die Verbindung zwischen Prozessorsystem und *Pre-Fetcher* ca. 625ns pro Byte bei einem Systemtakt von 5MHz. Die Anzahl der Bytes für die Konfigurationen des *Virtual Coarse Grained Reconfigurable Arrays*, welche zwischen dem Prozessorsystem und den *Pre-Fetcher*n ausgetauscht werden müssen, sind jeweils zweimal 18 Byte –acht Byte für die PE-Konfiguration und zehn Byte für die Konfiguration der virtuellen Kanäle und der Maske der *Synchronization Unit*. Die Übertragung der gesamten Konfiguration findet zu Beginn einmalig statt und beträgt gerade einmal 36µs. Sie ist daher gegenüber der Gesamtberechnungsdauer von 536,7ms vernachlässigbar klein. Der Aufwand für den Wechsel von der einen Konfiguration in die andere ist in dieser SystemC-Simulation ebenfalls vernachlässigbar, da dieser innerhalb eines Taktzyklus stattfindet.

5.4.3 Gesamtarchitektur inklusive Memory Management Unit und Central Control Unit

Da bei dieser Simulation alle notwendigen Daten zunächst in das *Shared Memory* geladen werden, finden während der Berechnung keine weiteren Datenübertragungen zwischen Prozessorsystem und der Gesamtarchitektur statt. Allerdings ist die Anbindung an ein solches *Shared Memory* bzgl. des Timings und der Signallaufzeiten nicht vollständig modelliert. Aus diesem Grund ist in der Ausführungszeit kein Aufwand für eine Datenübertragung in das *Shared Memory* oder für die Signallaufzeiten zwischen MMU und *Pre-Fetcher* oder Puffer enthalten.

Tab. 11: Simulationsergebnisse der gesamten Architektur

Execution Time	Overall	532,2ms
	Sobel in X-Direction	266,1ms
	Sobel in Y Direction	266,1ms
Area	Total	0,722mm^2

	Area for one PE	0,035mm²
	Area Synchronization Unit	0,035mm²
	Area Input Virtual Channel	0,016mm²
	Area General Virtual Channel (Virtual channels at over levels of the architecture)	0,008mm²
	Area Pre-Fetchers (Sum of all used Pre-Fetchers)	0,016mm²
	Area Central Control Unit	0,035mm²
	Area Memory Control Unit	0,037mm²
Power Consumption	Total Peak Power	0,371W
Calculation Time	One Pixel Result Value	66,2µs
	First Temporary	41,2µs
	Second Temporary	12,0µs
	Result Value	13,0µs
VCGRA Execution Time	First Temporary	4,2µs
	Second Temporary	4,2µs
	Result Value	4,2µs
	Median	4,2µs
	Total per Pixel	12,6µs

	Relative Execution Time per Pixel	19,1%
Time to Configure	Load PE Configuration to Pre-Fetcher	24,4µs
	Load Virtual Channel Configuration to Pre-Fetcher	26,8µs
	Total Configuration Time	51,2µs
	Change Configuration during Run-Time	1,0µs

Die relative Chipfläche für die *Central Control Unit* und die *Memory Mangement Unit* beträgt jeweils 4,8% und 5,1%, der Anteil für die *Pre-Fetcher* und Puffer der Konfigurationsdaten und Nutzdaten insgesamt 2,2% der gesamten Chipfläche. Für die Erweiterung des *Virtual Coarse Grained Reconfigurable Arrays* hin zu einem autarken System werden in diesem Beispiel grob $^1/_{10}$ der Chipfläche benötigt. Nicht Teil der abgeschätzten Chipfläche sind Aufwände für das *Shared Memory* und das *Instruction Memory* für den Maschinencode der *Central Control Unit* der Gesamtarchitektur.

Für den Wechsel einer Konfiguration vergeht ca. 1µs. Diese Zeit benötigt die *Central Control Unit*, um den Befehl zu dekodieren und die *Pre-Fetcher* für die Konfigurationen umzusteuern. Der eigentliche Wechsel zum Anlegen einer neuen Konfiguration an das VCGRA innerhalb eines *Pre-Fetcher*s findet in einem Taktzyklus statt. Die notwendige Zeit für das Laden der Konfigurationen einmalig zu Beginn der Simulation wird beeinflusst von der Bitbreite der Signalleitung zwischen *Memory Management Unit* und dem *Pre-Fetcher*. Sie ist in diesem Beispiel auf acht Bit eingestellt, da die Länge der Konfigurationen mit 144 Bits übersichtlich ist. Für größere Architekturen mit mehr Konfigurationsbits kann die Bitbreite zwischen *Pre-Fetcher* und *Memory Management Unit* erweitert werden. In der Dauer von 51,2µs sind die Befehlsverarbeitung der *Central Control Unit*, die Kommunikation zwischen *Memory Management Unit* und *Central Control Unit*, die Befehlsverarbeitung der MMU selbst und die Kommunikationen zwischen MMU und *Pre-Fetcher* oder Puffer berücksichtigt.

Die Verarbeitungsgeschwindigkeit (*Execution Time* nach Tab. 11) lässt sich noch weiter optimieren, wenn statt zwei *Lines* für den *Data-Input Pre-Fetcher*, drei *Lines*

verwendet würden. Dann ließen sich alle Koeffizienten für die Faltungsoperation des Sobel-Filters bereits vorladen und es müssten jeweils nur noch die notwendigen Daten aus dem *Shared Memory* geladen werden. Das ist möglich, weil sich die *Places* in den *Lines* über die *Assembler*befehle direkt adressieren lassen, weshalb nur die *Places* aktualisiert werden müssen, deren Daten sich verändern.

In der Berechnung des „*First Temporary*“ für die „*Calculation Time*“ ist der Datentransfer von der *Memory Management Unit* in den *Data-Input Pre-Fetcher* mit enthalten. Nach dem Start der Berechnung werden weitere Daten parallel zur aktuellen Berechnung vorgeladen.

5.5 Vergleich der Ergebnisse

5.5.1 Ergebnisvergleiche der Entwicklungsstufen

Für einen Vergleich der Entwicklungsstufen werden die drei Hauptmetriken *Area*, *Power Consumption* und *Calculation Time* verglichen. Durch die Verbesserungen in den einzelnen Entwicklungsstufen soll sich jeweils die *Calculation Time* verringern, während die Mehraufwände für *Power Consumption* und *Area* nur wenig ansteigen. Abb. 69 zeigt eine Übersicht zu den Metriken mit jeweils einem Balken für jede Entwicklungsstufe.

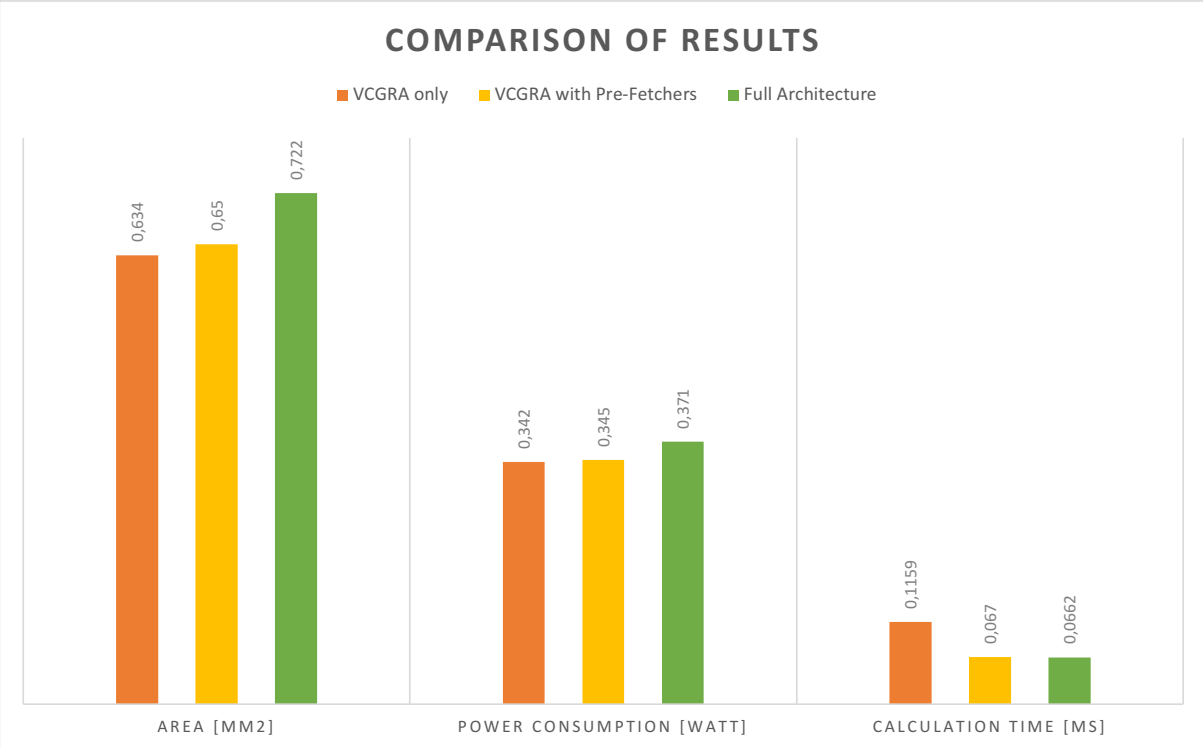

Abb. 69: Vergleich der Ergebnisse

Die Implementierung „*VCGRA only*“ orientiert sich an der Implementierung auf einem FPGA (siehe Abschnitt 2.3.6) und wird als Basis verwendet.

Die größte Verbesserung im Vergleich zu den entstehenden Mehraufwänden in *Area* und *Power Consumption* findet mit der Implementierung der *Pre-Fetcher* für die Konfiguration der virtuellen Kanäle und Prozesselemente statt. Im Vergleich zur Basisimplementierung ergibt sich eine Verbesserung in der Performanz (*Calculation Time*) um 42,2% bei einer gesteigerten Energieaufnahme von 0,8% und einer Fläçhenvergrößerung von 2,5%. Allerdings ist vor allem die geringe Flächenvergrößerung stark abhängig vom Design der *Pre-Fetcher*. Wird beispielsweise die Anzahl der *Lines* erhöht, steigt auch proportional der Flächenbedarf. Das genaue Verhältnis lässt sich auf Grund der groben Abschätzungsmöglichkeiten per McPAT

leider nicht herausstellen. Dennoch ist mindestens ein linearer Zusammenhang zwischen der Anzahl der *Lines* und dem Zuwachs des Flächenbedarfs zu erwarten.

Die Erweiterung der Architektur durch eine *Central Control Unit* und eine *Memory Management Unit* führt gegenüber der Basisimplementierung ohne *Pre-Fetcher* zu einer Verbesserung von ca. 43% bei einer gesteigerten Energieaufnahme von 13,8% und einer Flächenvergrößerung von 8,4%. Es gelten hierbei die gleichen Überlegungen zur *Area*, wie bereits bei der Erweiterung mit *Pre-Fetcher*n beschrieben. *Memory Management Unit* und *Central Control Unit* sind in ihrem Aufbau statisch und schwanken nur in der Bitbreite ihrer Anbindungen/Ports. Indes wird der Einfluss der *Pre-Fetcher* auf die Chipfläche durch *Data-Input Pre-Fetcher* und *Data-Output* Puffer noch einmal verstärkt. Zusätzlich zu den Konfigurationen können nun auch die Anzahl der *Lines* für diese zusätzlichen Komponenten auf die jeweilige Anforderung abgestimmt werden.

Die Verbesserung gegenüber der Ausbaustufe mit *Pre-Fetcher*n für die Konfiguration ist mit ca. 1,2% marginal. Der Unterschied fällt hier so gering aus, weil …

- … die Auswertung der Assemblerbefehle in der *Central Control Unit* ebenfalls Zeit benötigt, in der keine Verarbeitung durch das VCGRA stattfinden kann.
- … die Verarbeitung durch die *Memory Management Unit* Zeit benötigt, um die Daten für die Verarbeitung durch das VCGRA bereit zu stellen.
- … die Parallelisierung der Applikation nicht vollständig erfolgen kann. Die Verarbeitung der Daten durch das VCGRA erfolgt schneller als die Bereitstellung neuer Daten. Daher verbringt das VCGRA der Architektur ebenfalls Zeit mit Warten.
- … die Architektur nicht 100%ig optimiert ist für die Applikation. Die Anzahl der Prozesselemente bzw. die Bitbreite der Datenverbindungen könnte entsprechend angepasst werden.

Durch die Verwendung eines vollständig autarken Systems erhöht sich allerdings der Parallelisierungsgrad. So kann ein Prozessorsystem die Daten über das *Shared Memory* bereitstellen, die Verarbeitung anstoßen und per Interrupt auf die Fertigstellung warten. Parallel kann es andere Aufgaben wahrnehmen oder sich stromsparend schlafen legen. Vorstellbar wäre daher in diesem Zusammenhang auch ein Szenario, bei dem mehrerer solcher selbstständigen Minisysteme jeweils einzelne Teilaufgaben verarbeiten. Das Prozessorsystem kümmert sich um die Verteilung dieser Aufgaben und wartet per Interrupt auf deren Fertigstellung.

Ferner wäre vorstellbar, dass das Prozessorsystem den entsprechend passenden Assembler zur Bearbeitung der Teilaufgaben selbst ableitet und erstellt. Daten und Assembler werden dann einem verfügbaren Minisystem zur Verfügung gestellt, in dem die Daten in entsprechende Bereiche des *Shared Memory* kopiert werden und die Ausführung angestoßen wird. Solche Architekturen könnten als Konkurrenz zu DSPs oder Vektorarchitekturen fungieren, bei denen es vorwiegend zur Verarbeitung von SIMD-Aufgaben kommt.

Eine weitere Idee in diesem Zusammenhang wären selbst lernende „Neuronale-Netzwerk-Prozessoren“. Die Prozesselemente und die virtuellen Kanäle können die Verbindungsstruktur eines neuronalen Netzwerkes modellieren. Während das Prozessorsystem neue Daten aus der echten Welt aufnimmt, kann parallel die Verarbeitung über ein neuronales Netzwerk auf der VCGRA-Architektur erfolgen. Nach der Berechnung werden die Daten verglichen. Anhand des Fehlers zwischen Messdaten und Vorhersage des neuronalen Netzwerkes wird die Konfiguration für die VCGRA-Architektur optimiert, übertragen und eine neue Berechnung angestoßen. Daraus erwächst ein selbstlernendes, sich optimierendes KI-System.

5.5.2 Einordnung der Ergebnisse im Vergleich zu anderen Architekturen

Ein quantitativer Vergleich mit bestehenden Architekturen und Evaluationsplattformen gestaltet sich schwierig, da für die Implementierung anderer Architekturen und CGRAs Hardwarebeschreibungssprachen in VHDL und Verilog verwendet werden. Diese ermöglichen die Implementierung des Designs auf einen FPGA oder einem ASIC und erzeugen Laufzeitergebnisse an einer echten Plattform. Nichtsdestotrotz bietet SystemC einen hohen Aussagegrad ungeachtet der Abstraktion in eine höhere Programmiersprache, verwendet doch der Simulator eine zyklenakkurate, parallele Berechnung der Signale. Des Weiteren arbeiten führende Hersteller für FPGA- und ASIC- Synthese an der Übersetzung von SystemC, bzw. eines Subsets des Sprachumfangs in ein lauffähiges Hardwaredesign. Aus diesem Grund findet ein Vergleich auf qualitativer Ebene statt. Anhand ähnlicher Architekturen und FPGA-*Overlays* werden Vor- und Nachteile der eigenen beschriebenen VCGRA-Architektur herausgearbeitet.

5.5.2.1 QUKU: A Two Level Reconfigurable Architecture

QUKU ist ein FPGA-*Overlay* bestehend aus einer PE-Matrix gekoppelt an einen *Microblaze*-Softprozessor [44], [45], [46]. Das CGRA arbeitet dabei als Beschleuniger für den *Microblaze*. Zusätzlich zum CGRA enthält die Architektur noch ein *Configuration Manager Module* und ein *Address Manager Module*. Ein Prozesselement

besteht aus Speicher, einer *Functional Unit* und jeweils einem lokalen Adress- und Konfigurationscontroller. Die globalen Module für Speicheradressen und PE-Konfiguration dienen zum Laden der entsprechenden Informationen in die lokalen Controller einer jeweiligen PE. Die Kommunikation zwischen den Prozesselementen im CGRA findet mittels FIFO-Puffer statt nach dem Prinzip eines *DataFlow Process Networks* [47]. Das Design erlaubt zwei Rekonfigurationsstufen: Austausch eines Komplettdesigns mittels FPGA-Rekonfiguration; Austausch der Funktionalität durch Rekonfiguration der Prozesselemente.

Die Grundidee ähnelt der in dieser Thesis beschriebenen Gesamtarchitektur aus Prozessorsystem mit angeschlossenem Beschleuniger. Allerdings erscheint die vom Autor beschriebene VCGRA-Architektur unabhängiger, denn sie benötigt für die Bearbeitung der Nutzdaten über die gesamte Verarbeitungsdauer hinweg keinen Einfluss des angeschlossenen Prozessorsystems mehr. Das Laden der Konfigurationen geschieht automatisch durch die *Central Control Unit*, der Austausch einer PE-Konfiguration findet in der VCGRA-Architektur innerhalb eines Zyklus statt.

Grundsätzlich unterscheiden sich bei beiden Systemen der Funktionsumfang einer PE und die Netzwerkstruktur zur Verbindung dieser. Während QUKU im Laufe der Designphase aus den zu bearbeitenden Applikationen oder Kernels ein Superset erzeugt und daraus eine Konfiguration ableitet, wird ein Datenflussgraph auf ein VCGRA *gemapped*. Laut [46] ist die Anzahl der Konfigurationen hier begrenzt auf vier. Das VCGRA Design begrenzt diese nicht. Es lassen sich beliebig viele Konfigurationen im *Shared Memory* der Architektur hinterlegen und über die *Central Control Unit* und die *Memory Management Unit* in den *Pre-Fetcher*n vorladen. Die Anzahl der Konfigurationen ist nur begrenzt durch die Größe des *Shared Memory*. Eine PE speichert ihre aktuelle Konfiguration nicht selbst. Die *Pre-Fetcher* dienen als CGRA-weiter Konfigurationsspeicher, weshalb die gesamte VCGRA-Konfiguration inklusive virtueller Kanäle und PE-Funktion innerhalb eines *Clock-Cycle* synchron rekonfiguriert werden kann. Man erkauft sich das durch einen erhöhten Aufwand an Signalverbindungen, weshalb das VCGRA-Design für größere Implementierungen schlechter skaliert. Im Vergleich der Verbindungstopologie ist die VCGRA-Architektur durch ihren gerichteten Datenfluss eingeschränkter als das QUKU-Design. Allerdings reicht diese Art der Verbindungsstruktur für das Mapping gerichteter Datenflussgraphen aus. In der Designphase sind der Topologie des *Virtual Coarse Grained Reconfigurable Arrays* keine Einschränkungen gesetzt. Es kann in jedem Level der Architektur eine beliebige Anzahl an homogenen Prozesselementen

gesetzt werden. Aus den Beschreibungen für QUKU geht dieser Freiheitsgrad aufgrund der Netzwerktopologie nicht hervor.

5.5.2.2 [S]CREMA: A [Scalable] Coarse-Grain Reconfigurable Array with Mapping Adaptiveness

Die Forschungsgruppe um J. Nurmi hat in [48] ein CGRA-*Overlay Template* in VHDL vorgestellt: CREMA (*Coarse-grain REconfigurable array with Mapping Adaptiveness*). Das Ziel der Entwicklung war es, über ein parameterisierbares *Template* ein CGRA an eine bestimmte Applikation anzupassen. Dafür wurden neben der Architektur auch entsprechende grafische Entwicklungswerkzeuge bereitgestellt. CREMA wurde zusammen mit einen *General Purpose* RISC-Prozessor als Gesamtarchitektur vorgestellt. Der Prozessor kontrolliert das CGRA durch die Bereitstellung von Konfigurationen, hier genannt *Patterns*. Im Polling werden Konfigurationen und Daten in CREMA bereitgestellt und die Ergebnisse erwartet. Die Struktur des *Coarse Grained Reconfigurable Arrays* ist festgelegt auf eine 4×8 Matrix. Die Prozesselemente in der Matrix unterstützten diverse arithmetische Operationen für *Floating Point*- und Integer-Datentypen. Werden nur Ganzzahlen verwendet, kann auch die Datenbreite variiert werden. Andernfalls wird eine feste Datenbreite von 32Bit verwendet und es kann während der Laufzeit zwischen *Floating Point* und Integer umgeschaltet werden. Die festgelegte Grundstruktur auf eine Matrix von 4×8 wurde später erweitert auf zusätzliche mögliche Skalierungen [49]. Die Architektur erhielt den Prefix „S“ für *Scalable* in SCREMA. Die *Patterns* werden jeweils in einem eigenen Konfigurationsspeicher für eine PE abgelegt. Dieser kann bis zu vier *Patterns* abspeichern. Es kann zwischen *Patterns* innerhalb eines *Clock-Cycle* gewechselt werden. Um neue *Patterns* abzulegen, werden diese durch das CGRA ges*treamed*. Die Datenströme besitzen die Adresse der Ziel-PE, für die die Konfiguration bestimmt ist. Des Weiteren enthält der Datenstrom Details über die Matrixkonfiguration der Eingänge und die PE-Operation. Mehrere *Patterns* für die Prozesselemente formen den *Context* der Applikation; ergo den Algorithmus oder Kernel. Während der Designphase können somit verschiedene *Patterns* durch Wahl der Operation einer PE und den Eingangsdaten an den Multiplexern der PE-Eingänge festgelegt werden. Das *Template* bietet dabei verschiedene Routing Topologien an: *Nearest-Neighbor*, *Interleaved* und *Global*. Das Design wird umso kleiner, je weniger Konfigurationen an den Eingängen einer PE notwendig sind.

Das VCGRA basiert sowohl in der grundlegenden Struktur in VHDL nach Abschnitt 2.3, als auch im SystemC-Design aus einer Reihe von *Templates*. Diese sind ebenfalls

konfigurierbar in ihrer Bitbreite, CGRA-Struktur und PE-Operationen. Einzig der Ganzzahldatentyp ist derzeit ausgiebig getestet. Ebenfalls wird in eine Designphase und eine Laufzeitphase mit Rekonfiguration unterschieden. In der Designphase werden in der VCGRA-Architektur das Design und der Funktionsumfang des *Virtual Coarse Grained Reconfigurable Arrays* festgelegt als auch die Konfigurationen für die Prozesselemente und virtuellen Kanäle erstellt. Zusätzlich wird der Assembler für die *Central Control Unit* formuliert und in seinen Maschinencode übersetzt. Denn im Gegensatz zu CREMA muss die VCGRA-Architektur nicht durch Polling verwaltet werden, sondern arbeitet eigenständig den gesamten Kernel ab, inklusive Konfigurationswechsel sowie Laden und Speichern von Daten. Auch die VCGRA-Architektur unterstützt das Speichern von mehreren Konfigurationen in einem *Pre-Fetcher* für einen schnellen Wechsel einer CGRA-Konfiguration. Im Unterschied zu CREMA werden diese aber zentral verwaltet und über das VCGRA gleichzeitig verteilt. Das Laden neuer Konfigurationen in einen *Pre-Fetcher* kann während der Laufzeit des *Virtual Coarse Grained Reconfigurable Arrays* erfolgen und wird gesteuert durch die *Central Control Unit* und die *Memory Management Unit*. Die Konfigurationen werden ge*streamed*, die Bitbreite dafür ist einstellbar. Das Umschalten zwischen geladenen Konfigurationen erfolgt ebenfalls innerhalb eines *Clock-Cycle*. Außerdem ist die Konfiguration zwischen PE und virtuellem Kanal getrennt. Damit muss beim Umschalten von Eingängen in eine PE nicht die PE-Konfiguration, sondern nur die Konfiguration der virtuellen Kanäle angepasst werden.

Das VCGRA-Design kann im Gegensatz zum CREMA Design vollständig selbstständig arbeiten, wenn die entsprechenden Daten über das *Shared Memory* bereitgestellt sind. Einmal durch einen Startimpuls getriggert, kann das Prozessorsystem anderen Tätigkeiten nachgehen und wird per Interrupt über das Ende der Verarbeitung benachrichtigt. Dies erhöht den Parallelisierungsgrad des Gesamtsystems aus VCGRA-Architektur und Prozessorsystem. Im Vergleich der Verbindungstopologie ist die VCGRA-Architektur durch ihren gerichteten Datenfluss eingeschränkter als das CREMA-Design. Es werden keine globalen Kommunikationen über ein Level hinaus im VCGRA zugelassen. Allerdings reicht diese Art der Verbindungsstruktur für das Mapping gerichteter Datenflussgraphen aus.

5.5.2.3 Virtual Dynamically Reconfigurable Overlay

Capalija und weitere Authoren verwenden in diesem Usecase ein CGRA-*Overlay* als Ziel für einen JIT-Compiler [50]. Dabei geht es um die Beschleunigung des Kernelcodes einer Applikation in einem *General Purpose* Softprozessor. Die CGRA-

Struktur besteht aus einer Reihe von VDR-Units zusammengefasst in einem VDR-Block und VDR-Switches, welche die Blöcke verbinden. Dies ähnelt den Levels in der VCGRA-Architektur. Jedes Level besteht dabei aus einer Reihe von Prozesselementen. Ein Level ist mit seinem Vorgänger und Nachfolger über einen virtuellen Kanal verbunden. Die VDR-Units bilden dabei auch die Operationen innerhalb des *Coarse Grained Reconfigurable Arrays* ab. In beiden Architekturen können Elemente des gleichen Blocks oder Levels nicht miteinander kommunizieren. Es findet ein Datentransport nur zwischen unterschiedlichen Blocks oder Leveln statt. Die Kommunikation im VCGRA-Design ist im Vergleich zum VDR-Design durch den gerichteten Datenfluss in eine Richtung aber eingeschränkter.

Während der Ausführung von Befehlen im Softcore werden diese in der VDR-Architektur überwacht, um so genannte *Hot-Region*s, also Codesegmente, zu finden, die häufig ausgeführt werden. Diese werden dann auf das *Overlay* gemapped, indem für die CGRA-Struktur eine entsprechende Konfiguration geladen wird. Dies ist derzeit nicht möglich für das VCGRA-Design. Es benötigt alle Informationen für den Maschinencode der CCU und die passende Konfiguration der Komponenten zur Ausführung des Kernelcodes. Da sich die Zeit für die Rekonfiguration des *Virtual Coarse Grained Reconfigurable Arrays* im Bereich weniger Mikrosekunden befindet, wäre das Design gegebenenfalls eine Alternative Zielarchitektur für den Beschleuniger der *Hot-Regions*. Wiederkehrender Code der *Hot-Regions* könnte durch abgespeicherte Konfigurationen in den *Pre-Fetcher*n schnell erneut geladen werden. Allerdings würde dies bedeuten, dass es sich vorzugsweise um eine wiederkehrende SIMD-Aufgabe handelt. Andernfalls ist der Aufwand für die Bereitstellung der Daten, das Erstellen der Konfigurationen für das VCGRA und das Erstellen des passenden Assemblerprogramms für die CCU zu aufwendig.

5.5.2.4 ADRES: Architecture for Dynamically Reconfigurable Embedded Systems

Die Entwickler von DRESC (*Dynamically Reconfigurable Embedded System Compiler*) verfolgten bei der Implementierung ihres Compilers und der Zielarchitektur einen Top-Down-Ansatz [51]. Sie entwarfen zuerst den Compiler und anschließend die passende CGRA-Architektur ADRES. Es handelt sich um ein *Template* eines *Coarse Grained Reconfigurable Arrays* bestehend aus *Functional Units*, *Register Files* und *Routing Resources*. Die Matrix des *Coarse Grained Reconfigurable Arrays* weist in der obersten Ebene eine Besonderheit auf; sie wirkt nach außen wie ein VLIW-Prozessor. DRESC verteilt den Kernelcode einer Applikation auf das CGRA und beschleunigt verbleibenden Code durch die Ausführung im VLIW-Prozessor. Der Datenaustausch

zwischen VLIW-Prozessor und CGRA findet über *Shared Memory* statt. Die Architektur ist für Fixpoint-Operationen optimiert. Die *Functional Units* und *Register Files* sind homogen verteilt. Die *Register Files* dienen als lokale Datenspeicher oder als Zwischenspeicher in langen *Routing*pfaden, um dort die Verzögerung im kritischen Pfad zu reduzieren. Die Netzwerktopologie ist sehr flexibel und besteht aus Bussen, Leitungen und Multiplexern. Es wird zwischen einem Datennetzwerk unterschieden und einem Prädikatnetzwerk (ein Bit). Die Netzwerktopologien der beiden Netzwerke können sich dabei unterscheiden. *Functional Units* können in verschiedenen Topologien als Router verwendet werden. Das Prädikatnetzwerk wird verwendet, um Kontrollfluss aus Schleifen zu elimieren und bestimmt, ob eine Operation in einer *Functional Unit* ausgeführt wird. Die Funktion einer *Functional Unit* wird in einem *Configuration RAM* lokal gespeichert; ein Wechsel einer Konfiguration kann innerhalb eines *Clock-Cycle* erfolgen. Ist dieser nicht groß genug, können weitere *Contexts* aus normalem Speicher nachgeladen werden. Der Funktionsumfang einer *Functional Unit* entspricht der eines kleinen RISC-Prozessors.

Die VCGRA-Architektur und deren Framework weisen im Vergleich zu dieser Architektur noch Defizite auf. Kontrollfluss kann derzeit noch nicht verarbeitet werden, weshalb der Assembler der *Central Control Unit* für das *Loop-Unrolling* Schleifen mit festen Grenzen benötigt. In zukünftigen Erweiterungen soll durch das Laden von Konfigurationen in die *Pre-Fetcher* und dem Laden der richtigen Konfiguration durch die CCU Kontrollfluss modellierbar sein. Es gibt aber noch keine Rückmeldung des *Virtual Coarse Grained Reconfigurable Arrays* auf das Ergebnis einer Kontrollflussanweisung und dessen Auswertung in der *Central Control Unit*, weshalb dies im aktuellen Entwicklungsstand dieser Thesis nicht möglich ist. Im VCGRA-Framework müsste der CCU-Maschinencode und die Erstellung der passenden PE-Konfiguration und Konfiguration der virtuellen Kanäle aus dem Kernelcode automatisiert aufeinander abgestimmt werden. Die VCGRA-Architektur hat gegenüber ADRES den Vorteil, dass es Konfigurationen durch einen eigenen Konfigurationskanal parallel zur Verarbeitung des *Coarse Grained Reconfigurable Arrays* vorladen kann. Aus dem Verständnis des Autors lässt das ADRES-*Template* auch nur eine rechteckige CGRA-Struktur zu, während im VCGRA-Design die Topologie des *Overlay*s auf den Datenflussgraphen der Applikation in der Designphase angepasst werden kann. Dies spart Ressourcen bei der Implementierung für eine bestimmte Applikation(sklasse).

5.5.2.5 HyCUBE

Die HyCUBE-Architektur verfolgt den Ansatz, über ein intelligentes Verbindungsnetzwerk einen Beschleunigungsvorteil gegenüber anderen Architekturen zu erreichen [52]. *Functional Units*, so genannte *Tiles*, werden in einem 2D-*Mesh*-Netzwerk angeordnet. Durch Konfiguration des Netzwerkes können Daten über mehrere *Tiles* hinweg innerhalb eines *Clock-Cycle* ausgetauscht werden. Ein passend dazu entwickelter Compiler ermittelt statisch zur Compilierzeit die notwendigen Konfigurationen für das Netzwerk; die Konfigurationen werden für jedes *Tile* in einem eigenen *Configuration Memory* abgelegt. Jede *Functional Unit* verfügt über eine ALU, einen lokalen Speicher und einen *Crossbar-Switch* für das Data-*Routing*. Spezielle *Tiles* mit Speicherzugriff besitzen zusätzlich eine *Loard-Store-Unit*. Durch Konfiguration der *Crossbar-Switches* können Daten per *Bypass* unabhängig vom *Clock-Cycle* weitergeleitet werden. Damit lassen sich Datentransfers über mehrere *Functional Units* innerhalb eines *Clock-Cycle* bereitstellen.

Diese Art von Kommunikation ist im VCGRA nicht möglich. Ein *Bypassing* von Prozesselementen eines Levels ist nicht möglich. Daten, die innerhalb eines Levels nicht verwendet werden, werden für die synchrone Datenverarbeitung innerhalb einer PE dieses Levels gepuffert. Der Einbau dieser Puffer anhand des Datenflussgraphen erfolgt durch das zugehörige Framework automatisch. Im Vergleich zum HyCUBE wird aber mehr Energie benötigt, weil die PE eine Pufferoperation ausführt und nicht übergangen werden kann. Es erleichtert aber derzeit das Mapping des Datenflussgraphen auf das VCGRA und die Sychronisation der Verabeitung innerhalb des *Virtual Coarse Grained Reconfigurable Arrays*. Ein *Bypassing* wäre durch Erweiterung der virtuellen Kanäle möglich, da die Verarbeitung einer PE durch `valid`-Signale passend zum Eingangsdatum in ihrer Verarbeitung synchronisiert wird. Allerdings würde es im jetzigen Design die Größe der virtuellen Kanäle nochmals stark erweitern. Es müsste jeweils von Level zu Level jeder Eingang eines virtuellen Kanals auf jeden Eingang des folgenden virtuellen Kanals ge*mapped* werden können. Die Vorteile wären ein geringerer Energieverbrauch, weil die Pufferoperationen wegfallen, sowie eine erhöhte Arbeitsgeschwindigkeit, weil nicht jedes Level des *Virtual Coarse Grained Reconfigurable Arrays* eine Verarbeitung von Daten durchführen muss. Eine Evaluation wäre in einer zukünftigen Erweiterung der Architektur durchaus sinnvoll.

5.5.2.6 UltraSynth

Die Autoren von UltraSynth stellen in [53] ein CGRA als Rechenbeschleuniger in einer regelungstechnischen Umgebung vor. Dabei handelt es sich sowohl um einen vollständig automatisierten *Toolflow* als auch um ein *Overlay* für einen Xilinx Zynq® System-on-Chip. Der Toolflow ist in der Lage, eine regelungstechnische Aufgabe oder Teile daraus auf das CGRA und den im Zynq® enthaltenen ARM-Core zu verteilen. Der Anteil der auf dem Beschleuniger ausgeführten Applikation ist einstellbar. Für weitere Details zum Toolflow kann auf verschiedene wissenschaftliche Veröffentlichungen zurückgegriffen werden: [53], [54], denn es soll an dieser Stelle die Zielarchitektur analysiert und mit der vorgestellten VCGRA-Architektur verglichen werden.

Sensoren und Aktoren werden direkt an den Beschleuniger angebunden. Dabei dienen Puffer zur Synchronisation der zyklischen Verarbeitung der Daten. Die Einhaltung der Zykluszeiten wird überwacht und ein Fehler an den Cortex A9 gemeldet. Die Architektur besteht aus einem Netzwerk aus Prozesselementen, verschiedenen Puffern für Logs, Parameter, Sensordaten und Ergebnissen für den Host-Prozessor sowie Stellgrößen für Aktoren. Des Weiteren ist ein *Cycle-Counter* enthalten, der die jeweilige Konfiguration des *Coarse Grained Reconfigurable Arrays* steuert und das CGRA mit dem Host-Prozessor und den Sensordaten synchronisiert. Die Anbindung der einzelnen Elemente findet über einen AXI-Bus statt. Als weiteres Feature unterstützt die Architektur Kontrollfluss und verschachtelte Schleifen via spekulativen Berechnungen und „*predicated stores*“ – umgangssprachlich Speicher vorausgesagter Ergebnisse. Durch eine interne Auswertelogik werden anhand der Ergebnisse der Kontrollflussanweisung über den „*Current Context Counter*“ die Ergebnisse der entsprechenden PE ausgewählt. Jede PE besitzt neben einer ALU, einem *Register File* und einem Multiplexer für dessen Inputs auch einen Konfigurationsspeicher. Dieser ist per AXI-Adressbereich in den Speicherbereich des Host-Prozessors gemapped, um dessen Anbindung zu erleichtern.

Wie auch bei ADRES hat die Architektur von UltaSynth der VCGRA-Architektur die Verarbeitung von Kontrollflussanweisungen voraus, während die VCGRA-Architektur noch auf datenflussbasierte Verarbeitung konzipiert ist. Daher gelten die gleichen Bewertungsergebnisse wie für ADRES. Die fehlende Modellierung von Kontrollfluss spielt für viele Anwendungen aus dem Bereichen Audio, Video und Bildverarbeitung sowie künstliche Intelligenz und künstliche neuronale Netzwerke nur eine

untergeordnete Rolle. In diesen Bereichen kann die VCGRA-Architektur durch das *Pre-Fetching* während der eigentlichen Datenverarbeitung punkten.

5.5.2.7 DySER: Dynamically Specializing Execution Resource

Bei DySER handelt es sich um die Integration eines *Coarse Grained Reconfigurable Arrays* in die Ausführungsphase einer Pipeline eines General Purpose Prozessors [55]. Die Architektur verfolgt zwei eigentlich gegensätzliche Ziele: Parallelisierung von Datenverarbeitung und Funktionsspezialisierung. Während bei der Parallelisierung der Datenverarbeitung eher homogene Strukturen benötigt werden, sind für die Funktionsspezialisierung eher heterogene Strukturen erforderlich; homogene Strukturen wenden immer gleiche Instruktionen auf eine Masse von Daten an (SIMD), heterogene Strukturen wenden eine bestimmte Funktion auf wenige Daten an (SISD). Das Innere der DySER-Architektur besteht aus einem konfigurierbaren, umschaltbaren Netzwerk, welches eine Reihe von heterogenen Funktionseinheiten verbindet. Jede Funktionseinheit ist dabei mit jeweils vier benachbarten Netzwerkeinheiten verbunden. Für die Funktionsspezialisierung werden über einen angepassten Compiler der Datenpfad innerhalb des Netzwerkes und die Funktion der Funktionseinheit als Konfiguration abgelegt. Eine neue Konfiguration kann innerhalb von 64 Zyklen ausgetauscht werden. Dafür werden vorher alle Funktionseinheiten per Signal benachrichtig, ihre weitere Verarbeitung nach Abschluss der aktuellen Verarbeitung zu pausieren. Für die Datenumsetzung können innerhalb des *Coarse Grained Reconfigurable Arrays* verschiedene Aufgaben per Vektoralisierungstechniken parallel verarbeitet werden. Die Funktionseinheiten synchronisieren ihre Prozesse über „*valid-credit*“-Signale. Der Datenaustausch mit dem GPP findet über Eingangs- und Ausgangspuffer statt.

Durch die Verwendung eines Compiler-Frameworks, um entsprechende Konfigurationen für die DySER-Architektur innerhalb der Pipeline zu erstellen, ist die Konfiguration bereits vor der Umsetzung von Daten bekannt. Aus diesem Grund bietet sich auch die Verwendung der VCGRA-Architektur an dieser Stelle als Beschleuniger an. Innerhalb der Pipeline wird kein großes CGRA-Netzwerk benötigt. Damit ist der Bedarf an *Routing*ressourcen für die zentrale Verwaltung der Konfigurationen eingeschränkt. Es handelt sich um eine datenflussorientierte Verarbeitung, weshalb die Top-Down-Struktur des *Virtual Coarse Grained Reconfigurable Arrays* passend ist. Die Prozesselemente und die Struktur der virtuellen Kanäle lassen ebenfalls eine vektorbasierte Umwandlung von Daten zu; eine Beschränkung erfolgt über die Größe des gewählten *Virtual Coarse Grained Reconfigurable Arrays*. Da die statische Struktur

des Programmes bekannt ist, lassen sich der Maschinencode der *Central Control Unit* und die Konfigurationen für das VCGRA bereits parallel zur Verarbeitung in der Pipeline durch die VCGRA-Architektur benutzen. Während also die Pipeline läuft und in der aktuellen Ausführungsphase die Daten verrechnet werden, lädt die VCGRA-Architektur die neuen Konfigurationen und Daten bereits parallel selbständig vor, sodass eine neue Konfiguration innerhalb eines *Clock-Cycles* eingestellt werden kann. Damit lassen sich auch häufige Kontextwechsel innerhalb der Ausführungsphase leicht realisieren, was die notwendige Größe des *Coarse Grained Reconfigurable Arrays* reduziert.

Es wäre ebenfalls denkbar, die VCGRA-Struktur in eine OOO-Architektur einzubauen. Der *Data-Input Pre-Fetcher* und *Data-Output* Puffer dienen als Ablage für die Daten und Ergebnisse der Verarbeitung. Die *Pre-Fetcher* für die Konfiguration des *Virtual Coarse Grained Reconfigurable Arrays* enthalten die entsprechenden Konfigurationen zur Verarbeitung der Daten. Der Maschinencode für die *Central Control Unit* wird in einem FIFO abgelegt und damit sequenziell angewendet. Jede OOO-Einheit hätte innerhalb der *Pre-Fetcher* ein dediziertes Datensegment, um den Steuerungsaufwand für die CCU zu reduzieren. Durch die schnelleren Wechsel der Konfigurationen des *Coarse Grained Reconfigurable Arrays* könnte die DySER-Architektur weiter profitieren.

6 Zusammenfassung und Ausblick

Im Zuge der digitalen Revolution und der wachsenden Nachfrage nach Rechenleistung bei gleichzeitig gestiegenen Anforderungen an den Energieverbrauch spielen neuartige Verarbeitungstechnologien eine wichtige Rolle. Die Untersuchungen aus Abschnitt 2.1 haben gezeigt, dass grobgranulare Architekturen für arithmetische und logische Operationen Vorteile haben. *Coarse Grained Reconfigurable Arrays* gehören zu solchen grobgranularen Architekturen und sind bereits seit Anfang der neunziger Jahre [56] bekannt. Trotz ihrer Performanzvorteile handelt es sich bei dieser Rechnerarchitektur eher um einen Außenseiter durch den Mangel an Compiler- und Designunterstützung. CGRA-Designs sind häufig auch nicht an die Applikation anpassbar. Durch die Verwendung eines rekonfigurierbaren *Overlays* für eine programmierbare Hardware wie ein FPGA lassen sich hardwareunabhängig Designs entwickeln und testen, die entweder auf eine spezielle Anwendung/Anwendungsklasse angepasst sind oder einen eher generischen Ansatz verfolgen. Die Rekonfiguration spielt bei der Verwendung von Field Programmable Gate Arrays schon länger eine Rolle. Dennoch ist der Einsatz aufgrund mangelnden Supports in den Entwicklungsumgebungen der FPGA-Hersteller und Einschränkungen bei der Anwendung dieser Technik begrenzt.

Die wissenschaftlichen Untersuchungen im Rahmen dieser Arbeit verfolgen zwei Ziele: Entwickeln einer Hardwarestruktur mit *Just-In-Time*-Rekonfigurationsmöglichkeiten und Bereitstellen einer Toolchain, welche das Design, die Evaluation und die Simulation von CGRA-Architekturen erleichtert. Die Toolchain für das automatische Design einer VCGRA-Architekturinstanz und die Ableitung der Konfiguration aus einer graphischen Repräsentation einer Applikation sowie die Kopplung an ein Linux-System sind in Teilen in den wissenschaftlichen Mitveröffentlichungen [24], [31], [57], [58] beschrieben und nicht Teil dieser Dissertation. Dieses Dokument beinhaltet vielmehr die Entwicklungsphasen der *Overlayarchitektur* des *Virtual Coarse Grained Reconfigurable Arrays*.

Warum eine *Overlayarchitektur*? Die Performanz und die Flexibilität von Rechnerarchitekturen sind zwei gegensätzliche Eigenschaften, welche durch die ASIC-Designs in Sachen Performanz und von *General Purpose* Prozessoren in Sachen Flexibilität eingegrenzt werden. Konfigurierbare *Overlays* bilden eine Möglichkeit, diese Gegensätzlichkeit auszugleichen. So lässt sich das Design eines *Overlays* zum einen unabhängig von der Hardwareplattform, welche das *Overlay* instanziiert, entwickeln und für eine Applikation spezialisieren. Zum anderen bieten *Overlays*

durch ein parameterisierbares Design einen hohen Grad an Flexibilität in der Designphase. Zusätzlich können sie Möglichkeiten wie eine *Just-In-Time*-Rekonfiguration ermöglichen, obwohl die Hardwareplattform diese Möglichkeiten selbst gar nicht bietet.

Aus diesem Grund ist in den Abschnitten 2.2 und 2.3 ein *Overlay* für ein CGRA beschrieben. Es dient als Beschleuniger für den Kernelcode eines Algorithmus und ist an ein Prozessorsystem gekoppelt. Das VCGRA ist in VDHL designed ohne herstellerspezifische IP-Blocks, um die Unabhängigkeit gegenüber der instanziierenden Plattform zu ermöglichen. Die Architektur enthält nur die Grundkomponenten aus virtuellem Kanal, Prozesselementen und *Synchronization Unit*. Die Elemente des *Virtual Coarse Grained Reconfigurable Arrays* sind in Ebenen (*Levels*) angeordnet. Das Design erlaubt eine Rekonfiguration der virtuellen Kanäle, um die Prozesselemente adjazenter Ebenen zu verbinden wie auch die Operation der Prozesselemente selbst. Die Anbindung an ein Prozessorsystem erfolgt über *Memory Mapping* mit einem *AXI4Lite*-Interface auf einem Xilinx Zynq® System-On-Chip. Die Ergebnisse aus Abschnitt 2.3.6.4 zeigen das Potenzial der Architektur, weisen allerdings auch die Achillesverse des Designs auf. Das Bereitstellen der Daten und der erforderlichen Konfiguration des *Virtual Coarse Grained Reconfigurable Arrays* nehmen 50% der Rechenzeit ein. Um den Konfigurationsaufwand zu reduzieren, wurde im Rahmen des EU-Projektes EXTRA [3] in Kooperation mit der Hochschule in Ghent an der Integration des TLUT/TCON Toolflow, vorgestellt in Abschnitt 2.2, gearbeitet. Der Konfigurationsaufwand wird dahingehend reduziert, dass statt einer kompletten Synthese nur Bool'sche Gleichungen anhand von Inputparametern gelöst werden müssen, um einen Konfigurationsbitstrom zu vervollständigen. Leider hat der Toolflow grenzen bei der Umsetzbarkeit auf echte Hardware. Er kann nur virtuelle Field Programmable Gate Arrays als Zielplattformen adressieren. Er löst auch nicht das Problem der Rekonfiguration auf Bitebene für Field Programmable Gate Arrays, weshalb lange Konfigurationsbitströme notwendig sind.

Warum eine weitere Entwicklung und Evaluation auf SystemC-Ebene? Für die Evaluation weiterer Elemente zur Beschleunigung der Rekonfiguration waren die Zyklen aus Designänderung, Synthese, Messung und Messwerteauswertung sehr lange. Aus diesem Grund sollte eine schnellere Möglichkeit her auf Basis von zyklengenauer Simulation. SystemC bietet dafür die richtige Grundlage. Führende Hersteller von FPGA-Entwicklungswerkzeugen werben auch mit der Synthese eines Subsets des SystemC-Standards, weshalb der Autor beim Wechsel von einer Integration in Hardware aus dem SystemC-Design ausging. Leider war es nicht mehr

möglich im Rahmen der Dissertation das SystemC-Modell auf Hardware zu synthetisieren. Für die Evaluation der Entwicklungsphasen des Designs können die Stunden für eine Synthese hin zu wenigen Minuten für eine Kompilierung reduziert werden, weshalb es den Nutzern dieser Architektur erlaubt ist, mehr Zeit für die Auswertung ihrer Messergebnisse als mit Warten auf das Syntheseergebnis zu verbringen. Die Architektur ist über Github [59] und GitLab [60] öffentlich zugänglich. Somit muss für Evaluation und Erweiterungen der Architektur weder auf teure Hardware noch auf kommerzielle Entwicklungs- und Synthesewerkzeuge für VHDL zurückgegriffen werden. Das Design in SystemC ist ebenfalls parameterisierbar, beispielsweise in Verarbeitungsbitbreite, Operationen der Prozesselemente oder Anzahl der Levels.

Um die Ladezeiten für Konfigurationsdaten und Verarbeitungsdaten zu beschleunigen wird das VCGRA mit *Pre-Fetcher*n erweitert für die Konfigurationsdaten der virtuellen Kanäle und der Prozesselemente. Basierend auf der Größe der *Pre-Fetcher* können verschiedene Konfigurationen vorgeladen und innerhalb eines Zyklus kann zwischen Konfigurationen gewechselt werden. Die Konfigurationen sind dabei zentral gespeichert in den *Pre-Fetcher*n und werden auch zentral gesteuert im gesamten VCGRA synchron umgestellt. Die simulierte Verbesserung der Performanz beträgt nach Abschnitt 5.5.1 im Vergleich zum VCGRA ohne *Pre-Fetcher* ca. 42%. Dennoch muss das Prozessorsystem noch viele Aufgaben selbst wahrnehmen, wie beispielsweise das Bereitstellen der Nutzdaten und kann deshalb keine anderen Tasks verarbeiten. Daher wurde das bestehende Design nochmals um vier Grundeinheiten erweitert: einen *Data-Input Pre-Fetcher, einen Data-Output* Puffer, eine *Central Control Unit* und eine *Memory Management Unit* (siehe Abb. 62). Durch die Einführung der *Central Control Unit* wurde zusätzlich ein Assembler und ein Maschinencode entwickelt, um das VCGRA unabhängig und autark als Co-Prozessor zu verwenden, der mit dem gekoppelten Prozessorsystem Daten per *Shared Memory* austauscht.

Als *Usecases* für diese Architektur sind besonders SIMD-Applikationen geeignet. Das Prozessorsystem stellt Daten und den passenden Assembler für das VCGRA bereit, welches dann die Daten vollständig bearbeitet und per Interrupt dem Prozessorsystem die Fertigstellung der Ergebnisse mitteilt. Das Prozessorsystem ist währenddessen vollständig frei für eine andere Task zur Bearbeitung. Nach bestem Wissen des Autors sind die Verwendung einer MMU und einer CCU als Steuerungen für ein *Coarse Grained Reconfigurable Array* neuartig. Der Performanzgewinn blieb leider hinter den Erwartungen des Autors zurück, denn die zusätzlichen Erweiterungen brachten nur

einen Gewinn von ca. 1,2% im direkten Vergleich mit der *Pre-Fetcher*-Variante. Durch die Eigenständigkeit des Systems wird aber insgesamt wieder Rechenleistung für eine andere Berechnung im gekoppelten Prozessorsystem frei.

Der Einsatz der Architektur könnte aus diesen Gründen beispielsweise in *High Performance Computing* (HPC) oder in *Edge Devices* bei smarten Automatisierungslösungen liegen. Bei HPC-Anwendungen könnte ein Hauptprozessor Daten jeweils einem Cluster solcher VCGRA Co-Prozessoren bereitstellen und deren Berechnung anstoßen. Während der Berechnung einer Cluster-Instanz (eines *Virtual Coarse Grained Reconfigurable Arrays*) wird eine andere präpariert und ebenfalls gestartet und so weiter. Die einzelnen Instanzen des Clusters melden das Ende ihrer Berechnungen per Interrupt an den Hauptprozessor, der auf freiwerdende Ressourcen neue Aufgaben verteilen kann. In der *Edge*, beispielsweise in der Industrie 4.0, werden Messdaten vorverarbeitet, bevor sie in die Cloud geschickt werden. Diese Vorverarbeitung kann durch das VCGRA selbstständig auf bestehenden Daten ausgeführt werden, währenddessen bereits neue Daten von dem Prozessorsystem gesammelt werden. Im Vergleich zu einem Multicore oder einer GPU kann das Design des *Virtual Coarse Grained Reconfigurable Arrays* auf die Aufgabe optimiert werden und spart somit Energie im direkten Vergleich mit diesen Architekturen.

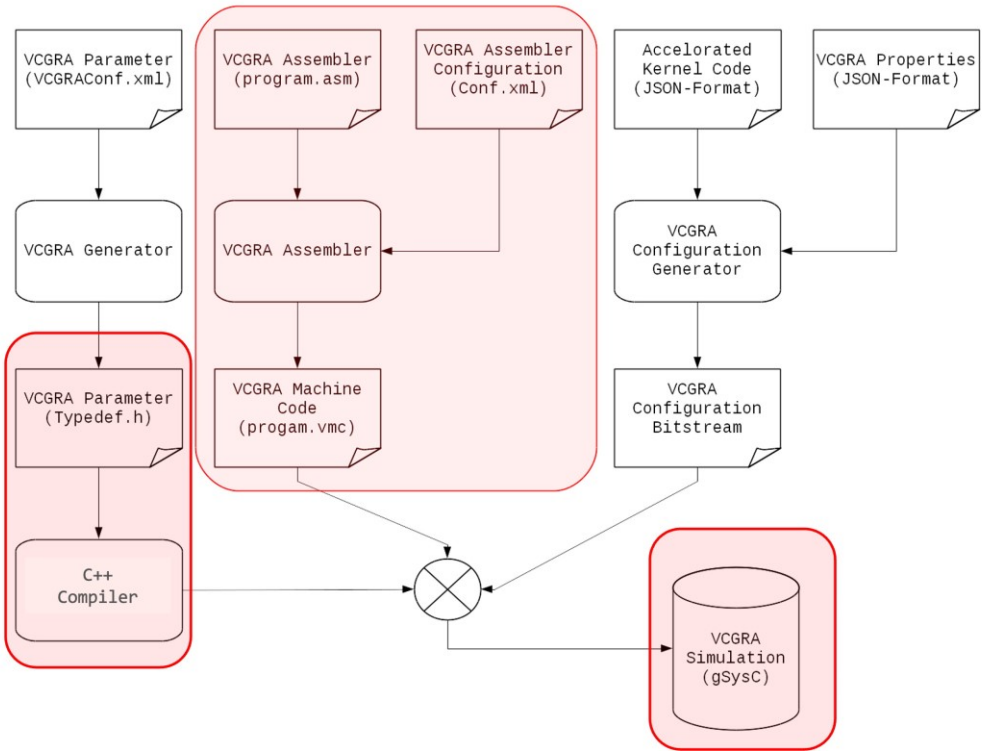

Abb. 70: Explorationsplattform

Abb. 70 zeigt eine Übersicht einer zukünftigen CGRA Explorationsplattform. Die mit rot markierten Teile der Abbildung sind im Rahmen der Dissertation umgesetzt. Einige Teile, wie der „*VCGRA Generator*“ und der „*VCGRA Configuration Generator*“, sind im Rahmen einer anderen Thesis entstanden und in den wissenschaftlichen Arbeiten [24], [31], [57], [58] beschrieben. Sie adressieren aber vorwiegend die VCGRA-Architektur ohne die Erweiterungen mit *Pre-Fetcher*n, *Central Control Unit* und *Memory Management Unit*. Hier ist geplant, die auf Python basierten Generatoren entsprechend auf die neuen Parametriermöglichkeiten zu erweitern. Die Parameter sollen zunächst alle innerhalb der „*Typedef.h*“-Datei gekapselt sein, um die automatische Erstellung möglichst einfach zu halten. Durch das strenge Typensystem von C++ und der Verwaltung der Komponenten mittels Vektoren und Arrays sind leider durch diesen Ansatz heterogene VCGRA Implementierungen nicht möglich. Die Implementierung der PE gibt dies aber generell her. Über ein Compileflag lässt sich eine Visualisierung über die *gSysC*-Bibliothek [61] anschalten. Diese basierte auf einer Qt3-Implementierung und SystemC 2.0 und wurde im Rahmen dieser Dissertation auf Qt5 und SystemC 3.x aktualisiert. Es lässt sich damit zyklengenau die Simulation steuern, um sich dabei den Zustand der Signale anzusehen.

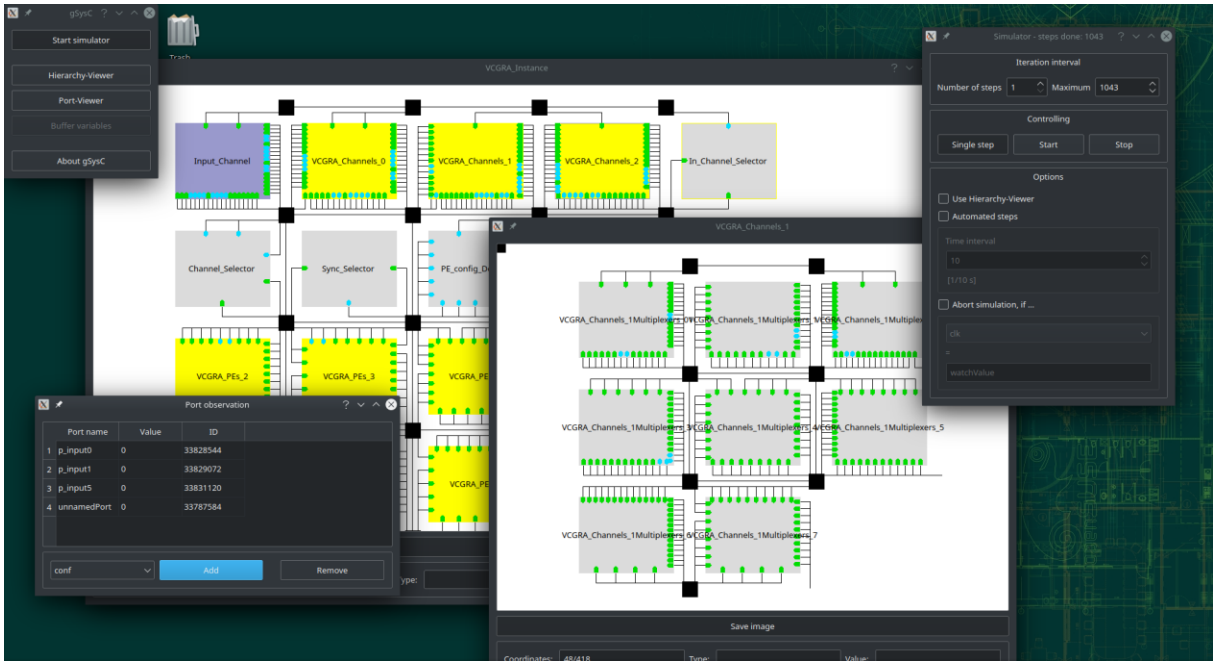

Abb. 71: gSysC Visualisierung eines Virtual Coarse Grained Reconfigurable Arrays

In Abb. 71 ist eine solche Simulation exemplarisch abgebildet. Über einen „*Hierarchy-Viewer*“ lässt sich durch das Design navigieren. Im „*Port-Viewer*“ lassen sich Signale und deren Zustände tracken. Über die Simulationssteuerung lässt sich der Simulationsablauf steuern.

Im Vergleich zu anderen CGRA-Explorationsplattfomen wie CGRA-ME [62] kann durch die Visualisierung und die Simulation per SystemC das Intervall für die Evaluation beschleunigt werden. Es werden keine weiteren kommerziellen Synthesewerkzeuge für die Funktionsüberprüfung der CGRA-Implementierung benötigt. Gleichzeitig können zu bestimmten Zeitpunkten die Zustände von Signalen überprüft werden, was eine Art Debugging im System ermöglicht. Die Visualisierung und Debuggingmöglichkeiten durch das gSysC-Framework sollen zukünftig noch besser integriert werden. Durch die Portierung des alten Programmcodes treten gelegentlich Probleme in der Steuerung der Simulation auf.

Da alle Tools und Frameworks Open-Source-Lizenzen unterliegen, soll die Visualisierungsplattform und die Architektur selbst für eine wissenschaftliche Community zur Nutzung und Erweiterung zur Verfügung gestellt werden [63], [64] und [59], [60].

I. Quellcodeanhänge

```
package pe_functions is

  --add: In1 + In2
  --returns sum; carry, negative and overflow marked in status reg
   procedure add(
    signal In1 : in signed;
    signal In2 : in signed;
    --variable N : in positive;
    signal result : out signed;
    signal status : inout std_logic_vector);

  --subtract: In1 - In2
  --returns difference, negative and underflow marked in status reg
   procedure subtract(
    signal In1 : in signed;
    signal In2 : in signed;
    --variable N : in positive;
    signal result : out signed;
    signal status : inout std_logic_vector);

  --multiply: In1 * In2
  --returns product, negative, overflow and underflow marked in status reg
   procedure multiply(
    signal In1 : in signed;
    signal In2 : in signed;
    --variable N : in positive;
    signal result : out signed;
    signal status : out std_logic_vector);

  --integer division: In1 / In2
  --returns quotient, negative, underflow, overflow marked in status reg
   procedure divide(
    signal In1 : in signed;
    signal In2 : in signed;
    --variable N : in positive;
    signal result : out signed;
    signal status : out std_logic_vector);
```

```
  --greater: In1 > In2
  --returns greater value, result of comparison is marked in status reg
   procedure greater(
    signal In1 : in signed;
    signal In2 : in signed;
    --variable N : in positive;
    signal result : out signed;
    signal status : out std_logic_vector);

  --equal: In1 == In2
  --returns In1, if equal and 0 if not, result of comparison is marked in status
reg
  procedure equal(
    signal In1 : in signed;
    signal In2 : in signed;
    --variable N : in positive;
    signal result : out signed;
    signal status : inout std_logic_vector);

  --modulo: In1 modulo In2
  --returns rest of a modulo operation, negative, zero and divided_by_zero marked
in status reg
   procedure modulo(
    signal In1 : in signed;
    signal In2 : in signed;
    --variable N : in positive;
    signal result : out signed;
    signal status : out std_logic_vector);

  --buffering: buffering input data at In1
  --buffers the signal at input one for one cycle
   procedure buffering(
    signal In1 : in signed;
    signal In2 : in signed;
    --variable N : in positive;
    signal result : out signed;
    signal status : out std_logic_vector);

end package pe_functions;
```

Code 32: VHDL-Package –Operationen des Prozesselements

```
Report Instance Areas:
+------+----------------------+---------------------------------------+------+
|      |Instance              |Module                                 |Cells |
+------+----------------------+---------------------------------------+------+
|1     |top                   |                                       | 50789|
|2     |cgra_overlay_i        |cgra_overlay                           | 50789|
|3     |cgra_overlay_wrapper_0|cgra_overlay_cgra_overlay_wrapper_0_0  | 48824|
|4     |U0                    |cgra_overlay_wrapper_v1_0              | 48092|
|5     |cgra                  |vcgra                                  | 46730|
|6     |PE_0_0                |PE                                     |  3465|
|7     |PE_0_1                |PE_5                                   |  3465|
|8     |PE_0_2                |PE_6                                   |  3465|
|9     |PE_0_3                |PE_7                                   |  3473|
|10    |PE_1_0                |PE_8                                   |  3463|
|11    |PE_1_1                |PE_9                                   |  3463|
|12    |PE_1_2                |PE_10                                  |  3463|
|13    |PE_1_3                |PE_11                                  |  3471|
|14    |PE_2_0                |PE_12                                  |  3463|
|15    |PE_2_1                |PE_13                                  |  3463|
|16    |PE_2_2                |PE_14                                  |  3463|
|17    |PE_2_3                |PE_15                                  |  3463|
|18    |sync                  |sync_ps                                |     1|
|19    |vCH_0                 |virtualChannel_8_4                     |  1281|
|20    |vCH_1                 |virtualChannel_4_4                     |   652|
|21    |vCH_2                 |virtualChannel_4_4_16                  |   652|
|22    |[..]ch_config_inst    |[..]ch_config                          |   229|
|23    |[..]data_inst         |[..]data                               |   469|
|24    |[..]pe_config_inst    |[..]pe_config                          |   433|
|25    |[..]sync_config_inst  |[..] sync_config                       |   231|
[...]
```

Code 33: Reportauszug Vivado® –Chipflächenverbrauch in Anzahl der Zellen

```
<Assembler_Property>
  <LineSize>3</LineSize>
  <PlaceSize>7</PlaceSize>
  <OpCodeSize>6</OpCodeSize>

  <NoOperator>
      <Operator>
          <Name>NOOP</Name>
          <MachineId>0</MachineId>
      </Operator>
      <Operator>
          <Name>START</Name>
          <MachineId>11</MachineId>
      </Operator>
```

```
        <Operator>
            <Name>FINISH</Name>
            <MachineId>12</MachineId>
        </Operator>
        <Operator>
            <Name>WAIT_READY</Name>
            <MachineId>4</MachineId>
        </Operator>
    </NoOperator>
    <OneOperator>
        <Operator>
            <Name>SLCT_DIC_LINE</Name>
            <MachineId>15</MachineId>
        </Operator>
        <Operator>
            <Name>SLCT_DOC_LINE</Name>
            <MachineId>16</MachineId>
        </Operator>
        <Operator>
            <Name>SLCT_PECC_LINE</Name>
            <MachineId>17</MachineId>
        </Operator>
        <Operator>
            <Name>SLCT_CHCC_LINE</Name>
            <MachineId>18</MachineId>
        </Operator>
    </OneOperator>
    <TwoOperator>
        <Operator>
            <Name>LOADDA</Name>
            <MachineId>6</MachineId>
        </Operator>
        <Operator>
            <Name>STOREDA</Name>
            <MachineId>8</MachineId>
        </Operator>
        <Operator>
            <Name>LOADPC</Name>
            <MachineId>9</MachineId>
        </Operator>
        <Operator>
            <Name>LOADCC</Name>
            <MachineId>10</MachineId>
        </Operator>
    </TwoOperator>
```

```
  <ThreeOperator>
      <Operator>
          <Name>LOADD</Name>
          <MachineId>5</MachineId>
      </Operator>
      <Operator>
          <Name>STORED</Name>
          <MachineId>7</MachineId>
      </Operator>
  </ThreeOperator>
</Assembler_Property>
```

Code 34: Assembler Configuration File – Konfiguration der verfügbaren Assembler Commands

```
VCGRA_Instance.VCGRA_PEs_0    Processing Element  ID:   0
total cycles: 3920880
idle cycles: 3874752
busy cycles: 46128

…

VCGRA_Instance.VCGRA_PEs_15   Processing Element  ID:  15
total cycles: 3920880
idle cycles: 3920880
busy cycles: 0

VCGRA_Instance.Input_Channel  Virtual Channel
total cycles: 3920881
idle cycles: 0
busy cycles: 3920881

…

VCGRA_Instance.VCGRA_Channels_2 Virtual Channel
total cycles: 3920881
idle cycles: 0
busy cycles: 3920881

VCGRA_Instance.VCGRA_Sync   Synchronizer
total cycles: 3920881
idle cycles: 0
busy cycles: 3920881
```

Code 35: McPAT Dynamik Statistik Ausgabe einer SystemC-Simulation

II. Bildanhänge

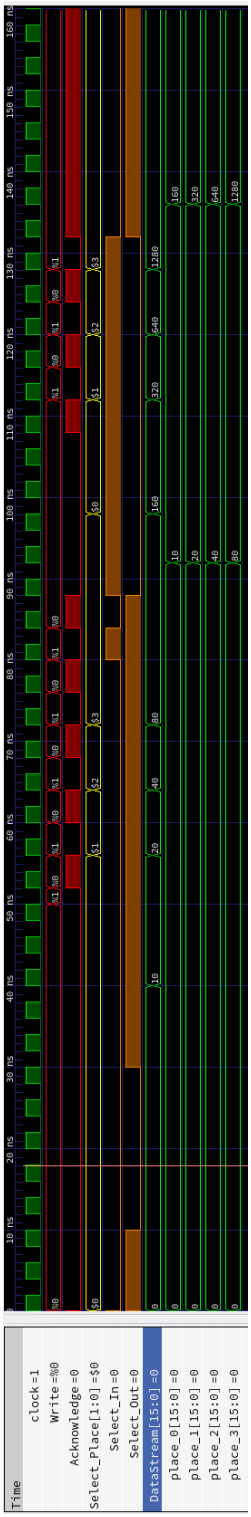

Abb. 72: Timingdiagramm – Data-Input Pre-Fetcher

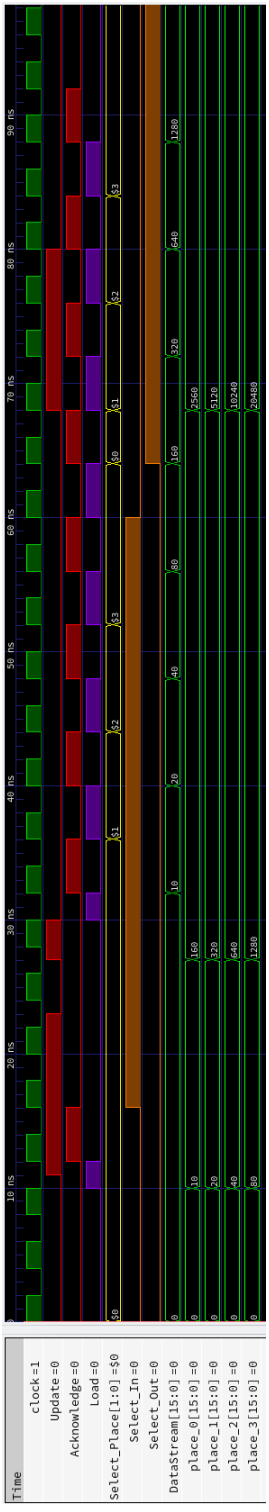

Abb. 73: Timingdiagramm – Data-Output Puffer

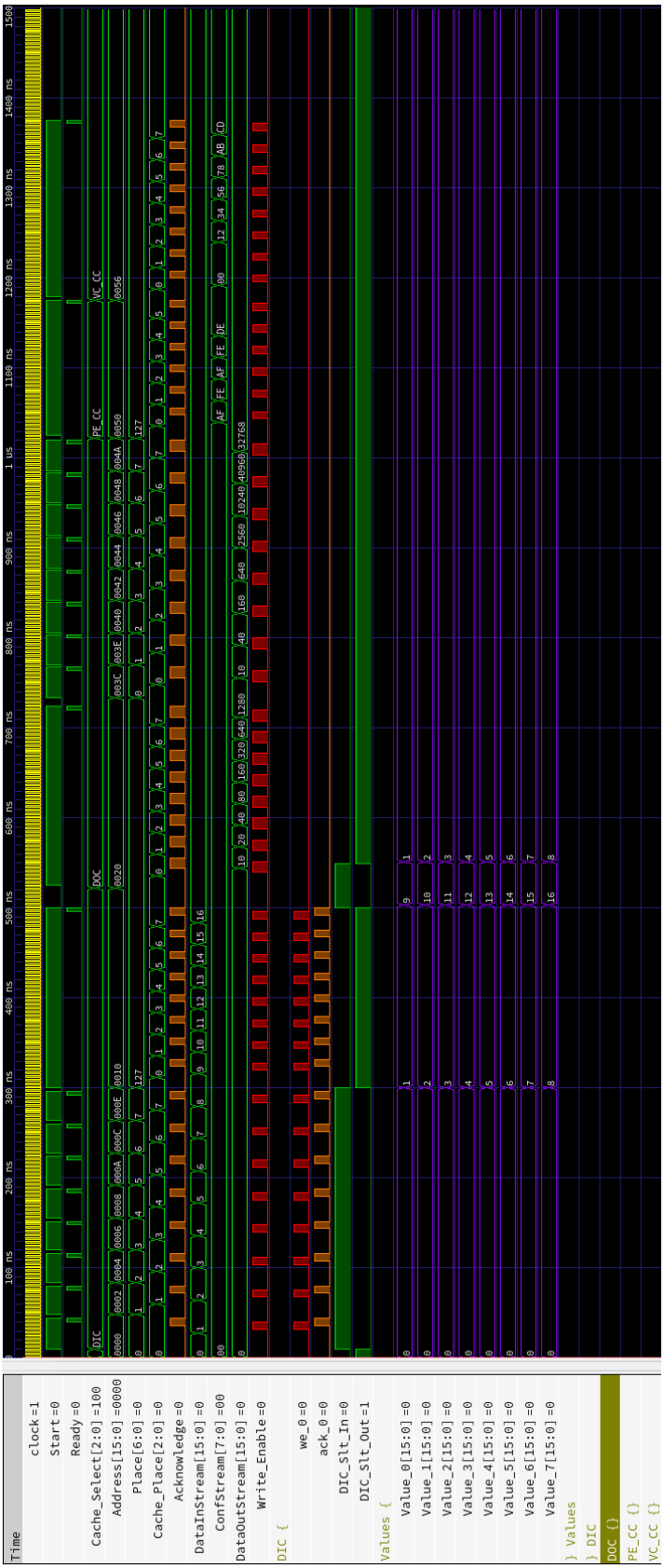

Abb. 74: Timingdiagramm – MMU - Data-Input Pre-Fetcher

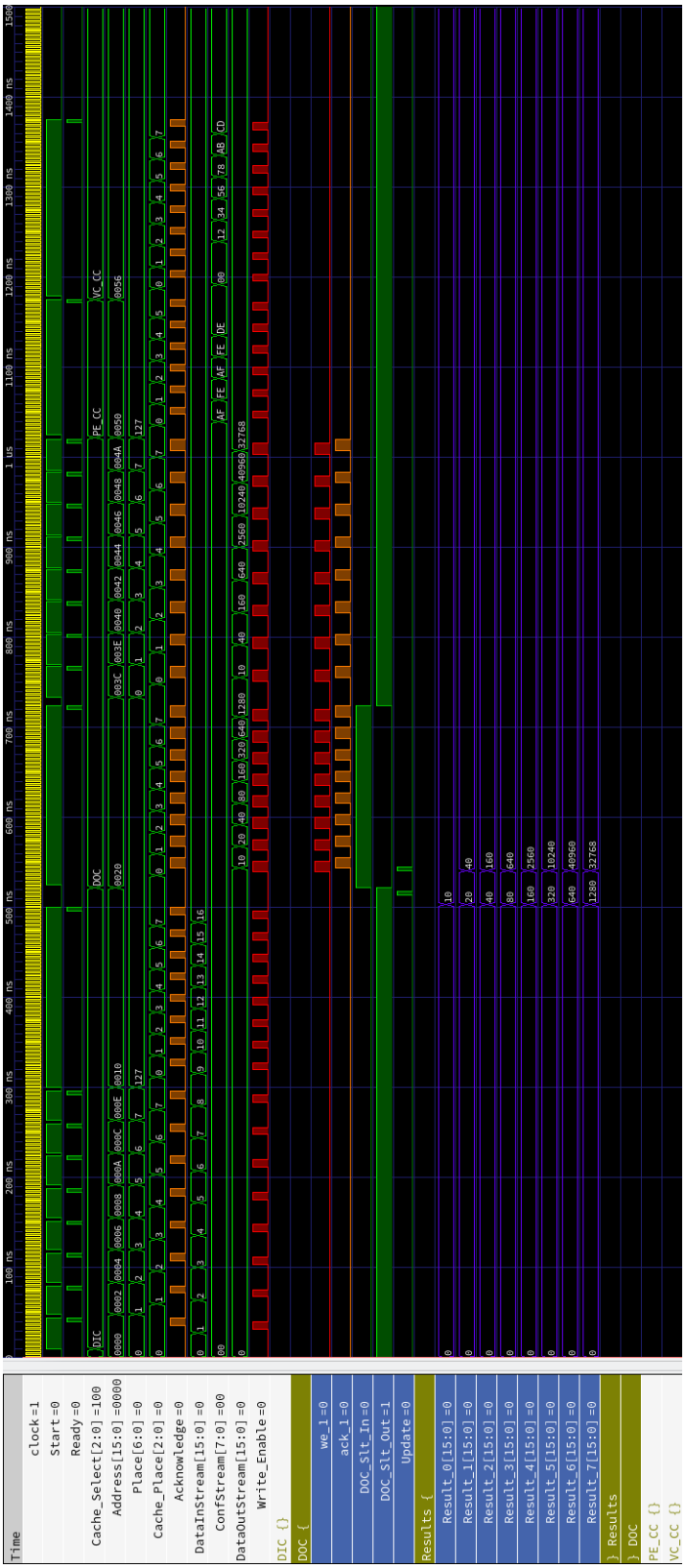

Abb. 75: Timingdiagramm – MMU - Data-Output Puffer

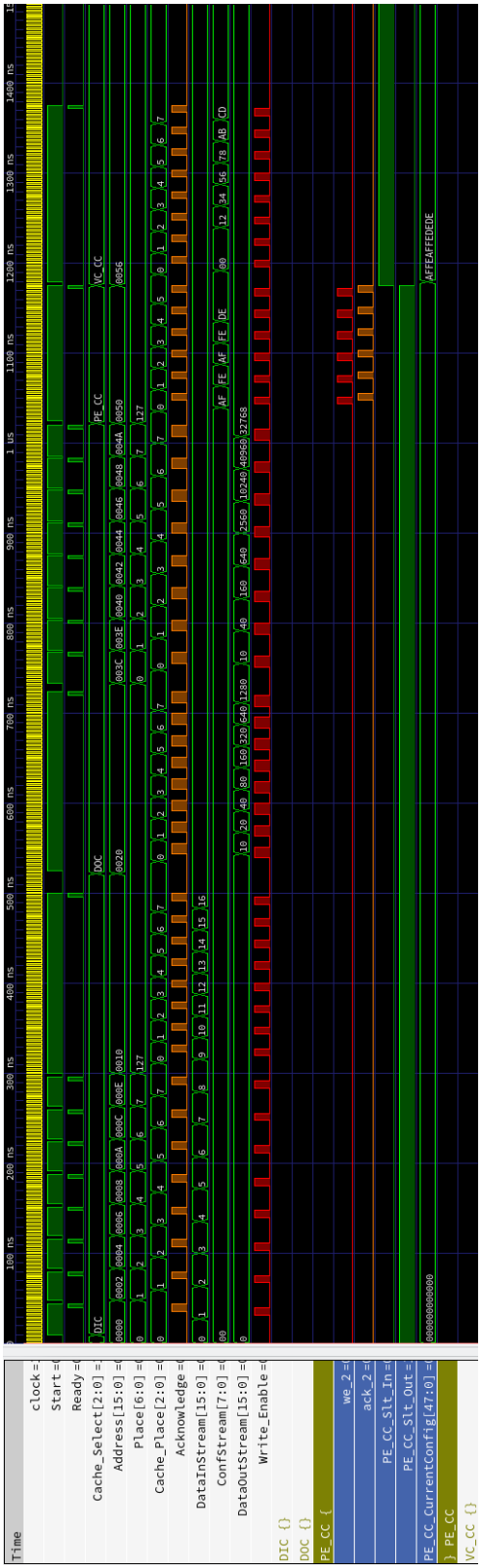

Abb. 76: Timingdiagramm – MMU – PE Config. Pre-Fetcher

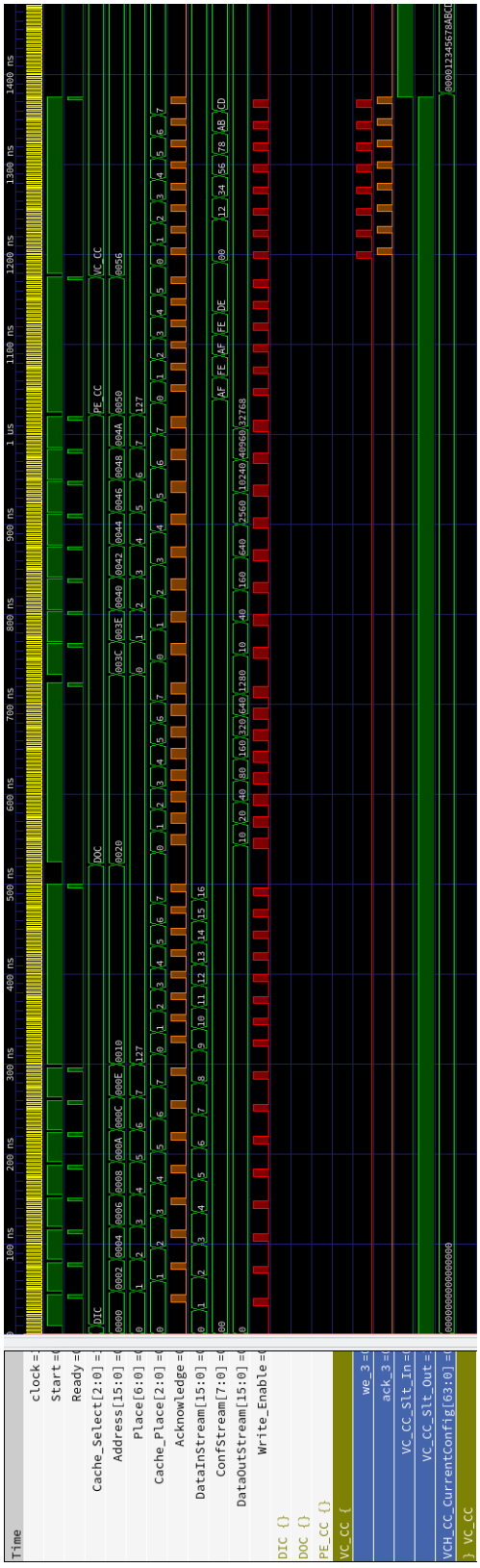

Abb. 77: Timingdiagramm – MMU - VC Config. Pre-Fetcher

III.Referenzliste

[1] R. W. Hartenstein, A. G. Hirschbiel und M. Weber, „XPUTERS: Very High Throughput By Innovative Computing Principles," in *Jerusalem Conference on Information Technology*, Jerusalem, 1990.

[2] K. Heyse, B. Al Farisi, K. Bruneel und D. Stroobandt, „TCONMAP: Technology Mapping for Parameterised FPGA Configurations," *ACM Transactions on Design Automation of Electronic Systems,* Bd. 20, Nr. 4, p. 48:1–48:27, September 2015.

[3] „Exploiting eXascale Technology with Reconfigurable Architectures," 2015. [Online]. Available: https://www.extrahpc.eu. [Zugriff am 24 10 2021].

[4] F. Arute, K. Arya und R. Babbush, „Quantum supremacy using a programmable superconducting processor," *Nature,* Bd. 574, p. 505–510, 2019.

[5] M. A. a. J. Y. a. S. J. H. a. D. C. a. Z. Z. a. L. W. D. Zidan, „Field-Programmable Crossbar Array (FPCA) for Reconfigurable Computing," *IEEE Transactions on Multi-Scale Computing Systems,* Bd. 4, Nr. 4, pp. 698-710, 2018.

[6] J. Smit, M. Stekelenburg, C. E. Klaassen, M. Mullender, G. Smit und P. Havinga, „Low Cost & Fast Turnaround: Reconfigurable Graph-Based Execution Units," in *7th Behavioral Design Methodologies for Digital Systems workshop*, Enschede, 1998.

[7] K. Campton und S. Hauck, „Reconfigurable Computing: A Survey of Systems and Software," *ACM Computing Surveys,* Bd. 34, Nr. 2, pp. 171-210, 2002.

[8] T. J. Todman, G. A. Constantinides, S. J. Wilton, O. Mencer, W. Luk und P. Cheung, „Reconfigurable computing: architectures and design methods," *IEE Proc.-Comput. Digit. Tech.,* Bd. 152, Nr. 2, 2005.

[9] R. Tessier, K. Pocek und A. DeHon, „Reconfigurable Computing Architectures," *Proceedings of the IEEE,* Bd. 103, Nr. 3, 2015.

[10] Intel, „Intel Ships First 10nm Agilex FPGAs," Intel, August 2019. [Online]. Available: https://newsroom.intel.com/news/intel-ships-first-10nm-agilex-fpgas/#gs.e63ygp. [Zugriff am 03 November 2019].

[11] M. S. Won, „Intel Agilex FPGAs and SOCs," Intel, 2019. [Online]. Available: https://plan.seek.intel.com/psg_WW_psgcom3_LPCD_EN_2019_AgilexArchitectureWP. [Zugriff am 3 November 2019].

[12] G. M. Amdahl, „Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, Atlantic City, New Jersey, 1967.

[13] L. Goldschlager und A. Lister, Informatik - Eine moderne Einführung, Bd. 3, München Wien: Carl Hanser Verlag, 1990.

[14] N. S. Voros und M. Hübner, „MORPHEUS: A Heterogeneous Dynamically Reconfigurable Platform for Designing Highly Complex Embedded Systems," in *ACM Transactions on Embedded Computing Systems*, 2019.

[15] J. Lue, „VTR7.0: Next Generation Architecture and CAD Systems for FPGAs," in *Lecture Notes in Computer Science*, 2014.

[16] E. Ahmed und J. Rose, „The effect of LUT and Cluster size on deep-submicron FPGA performance and density," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* Bd. 12, Nr. 3, p. 288–298, March 2004.

[17] M. Halder, A. Nayak, A. Choudhary und P. Banerjee, „Parallel Algorithms for FPGA Placement," in *10th. Great Lakes Symposium on VLSI*, New York, 2000.

[18] M. Lukowiak und B. Cody, „FPGA Based Accelerator for Simulated Annealing with Greedy Perturbations," in *14th International Conference on Mixed Design of Integrated Circuits and Systems*, Ciechocinek, 2007.

[19] H. Sidiropoulos, K. Siozios, P. Figuli, D. Soudris und M. Hübner, „OnSupporting Efficient Partial Reconfiguration with Just-In-Time Compilation," in *26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, Shanghai, 2012.

[20] M. An, J. G. Steffan und V. Betz, „Speeding Up FPGA Placement: Parallel Algorithms and Methods," in *22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, Massachusetts, 2014.

[21] K. Bruneel und D. Stroobandt, „Automatic generation of run-time parameterizable configurations," in *International Conference on Field Programmable Logic and Applications*, Heidelberg, 2008.

[22] K. Bruneel, W. Heirman und D. Stroobandt, „Dynamic Data Folding with Parameterizable Configurations," *ACM Transactions on Design Automation of Electronic Systems,* Bd. 16, Nr. 4, 2011.

[23] E. Vansteenkiste, B. Al Farisi, K. Bruneel und D. Stroobandt, „TPaR: Place and Route Tools for the Dynamic Reconfiguration of the FPGA's Interconnect Network," *EEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* Bd. 33, p. 370–383, March 2014.

[24] A. Kulkarni, A. Werner, F. Fricke, D. Stroobandt und M. Hübner, „Pixie: A heterogeneous Virtual Coarse-Grained Reconfigurable Array for high performance image processing applications," in *3rd International Workshop on Overlay Architectures for FPGAs*, Monterey, 2017.

[25] K. Bruneel, F. Abouelella und D. Stroobandt, „Automatically mapping applications to a self-reconfiguring platform," in *Proceedings of Design, Automation and Test in Europe*, Nice, 2009.

[26] A. Kourfali, A. Kulkarni und D. Stroobandt, „SICTA: A Superimposed In-Circuit Fault Tolerant Architecture for SRAM-based FPGAs," in *23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, Greece, 2017.

[27] A. Kourfali, M. D. Codinachs und D. Stroobandt, „Superimposed In-Circuit Fault Mitigation for Dynamically Reconfigurable FPGAs," in *17th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, Geneva, Switzerland, 2017.

[28] IEEE, *IEEE 1076 - IEEE Standard VHDL Language Reference Manual,* IEEE, 2009.

[29] Xilinx, „UG761 - AXI Reference Guide," Xilinx, 2011.

[30] A. Erhardt, „Hochpassfilter," in *Einführung in die Digitale Bildverarbeitung*, Wiesbaden, Vieweg+Teubner, 2008, pp. 154-162.

[31] F. Fricke, A. Werner, K. Shahin und M. Hübner, „CGRA Tool Flow for Fast Run-Time Reconfiguration," in *International Symposium on Applied Reconfigurable Computing*, Darmstadt, 2018.

[32] AVNET, „ZedBoard," 2019. [Online]. Available: http://zedboard.org/product/zedboard. [Zugriff am 22 December 2019].

[33] Xilinx Inc., „Digilent XUPV5 Board," 2019. [Online]. Available: https://www.xilinx.com/support/university/boards-portfolio/xup-boards/DigilentXUPV5Board.html. [Zugriff am 22 December 2019].

[34] R. Rivera, „Welcome to Boost.org!," 2007. [Online]. Available: https://www.boost.org. [Zugriff am 06th. June 2020].

[35] Python Software Foundation, „Data Model," Python Software Foundation, 2001-2020. [Online]. Available: https://docs.python.org/3/reference/datamodel.html#objects-values-and-types. [Zugriff am 12 06 2020].

[36] „IEEE Standard for Standard SystemC Language Reference Manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005),* pp. 1-638, 2012.

[37] F. Kesel, Modellierung von digitalen Systemen mit SystemC: Von der RTL-zurTransaction-Level-Modellierung., Oldenbourg Wissenschaftsverlag, 2012.

[38] Accellera Systems Initiative Inc., „SystemC Synthesizable Subset Version 1.4.7," Accellera Systems Initiative, 2016.

[39] Xilinx Inc., „4X C/C++/SystemC to RTL," Xilinx Inc., 2020. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/prod-advantage/rtl-synthesize.html. [Zugriff am 1st. June 2020].

[40] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen und N. P. Jouppi, „McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, New York, USA, 2009.

[41] F. Klein, R. Azevedo, L. Santos und G. Araujo, „SystemC-Based Power Evaluation with PowerSC," in *Electronic System Level Design*, Springer, Dordrecht, 2011, pp. 129-144.

[42] M. Giammarini, S. Orcioni und M. Conti, „Powersim: Power Estimation with SystemC," in *Solutions on Embedded Systems*, Bd. 81, Springer, Dordrecht, 2011, pp. 285-300.

[43] D. Greaves und M. Yasin, „TLM POWER3: Power Estimation Methodology for SystemC TLM 2.0," in *Models, Methods, and Tools for Complex Chip Design*, Springer, Cham., 2014, pp. 53-68.

[44] S. Shukla, N. W. Bergmann und J. Becker, „QUKU: a two-level reconfigurable architecture," in *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06)*, Karlsruhe, 2006.

[45] S. Shukla, N. W. Bergmann und J. Becker, „QUKU: A FPGA Based Flexible Coarse Grain Architecture Design Paradigm using Process Networks," in *IEEE International Parallel and Distributed Processing Symposium*, Long Beach, CA, USA, 2007.

[46] S. Shukla, N. W. Bergmann und J. Becker, „QUKU: A fast run time reconfigurable platform for image edge detection," in *Lecture Notes in Computer Science*, Bd. 3985, Delft, Springer, 2006, pp. 93-98.

[47] E. A. Lee und T. M. Parks, „Dataflow process networks," in *Proceedings of the IEEE*, Bd. 83, IEEE, 1995, pp. 773-801.

[48] F. Garzia, W. Hussain und J. Nurmi, „CREMA: A coarse-grain reconfigurable array with mapping adaptiveness," in *International Conference on Field Programmable Logic and Applications*, Prague, Czech Republic, 2009.

[49] W. Hussain, T. Ahonen und J. Nurmi, „Effects of scaling a coarse-grain reconfigurable array on power and energy consumption," in *International Symposium on System on Chip (SoC)*, Tampere, Finland, 2012.

[50] D. Capalija und T. S. Abdelrahman, „Towards Synthesis-Free JIT Compilation to Commodity FPGAs," in *IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, Salt Lake City, UT, USA, 2011.

[51] B. Mei, M. Berekovic und J.-Y. Mignolet, „ADRES & DRESC: Architecture and Compiler for Coarse-Grain Reconfigurable Processors," in *Fine- and Coarse-Grain Reconfigurable Computing*, Springer, 2007, pp. 255-297.

[52] M. Karunaratne, A. Kulkarni Mohite, T. Mitra und L.-S. Pe, „HyCUBE: A CGRA with reconfigurable single-cycle multi-hop interconnect," in *54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, Austin, TX, USA, 2017.

[53] D. Wolf, T. Rauschke, C. Hochberger, A. Engel und A. Koch, „UltraSynth: Integration of a CGRA into a Control Engineering Environment," in *Applied Reconfigurable Computing*, Darmstadt, Springer, 2019, pp. 247-261.

[54] D. Wolf, A. Engel, T. Ruschka, A. Koch und C. Hochberger, „UltraSynth: Insights of a CGRA Integration into a Control Engineering Environment," *Journal of Signal Processing Systems,* 2021.

[55] V. Govindaraju, C.-H. Ho, T. Nowatzk, J. Chhugani, N. Satish, K. Sankaralingam und C. Kim, „DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing," *IEEE Mirco,* Bd. 32, Nr. 5, pp. 38-51, 2012.

[56] A. Podopas, K. Sano und M. S., „A Survey on Coarse-Grained Reconfigurable Architectures From a Performance Perspective," *IEEE Access,* Bd. 8, 2020.

[57] F. Fricke, W. A. und H. M., „Tool flow for automatic generation of architectures and test-cases to enable the evaluation of CGRAs in the context of HPC applications," in *Conference on Design and Architectures for Signal and Image Processing (DASIP)*, Dresden, Germany, 2017.

[58] F. F., W. A., S. K., W. F. und H. M., „Automatic Tool-Flow for Mapping Applications to an Application-Specific CGRA Architecture," in *EEE*

International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Rio de Janeiro, Brazil, 2019.

[59] A. Werner, „CGRA SystemC," github, 2016. [Online]. Available: https://github.com/werneazc/cgra-systemc.git. [Zugriff am 01 04 2022].

[60] A. Werner, „CGRA-SystemC," Gitlab, 2015. [Online]. Available: https://gitlab.com/werneazc/cgra-systemc.git. [Zugriff am 01 04 2022].

[61] C. Eibl, C. Albrecht und R. Hagenau, „gSysC: A graphical front end for SystemC," in *In European Conference on Modelling and Simulation*, Riga, Latvia, 2005.

[62] S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi und J. Anderson, „CGRA-ME: A unified framework for CGRA modelling and exploration," in *IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Seattle, WA, USA, 2017.

[63] A. Werner, „CGRA-Assembler," 2021. [Online]. Available: https://github.com/werneazc/cgra_assembler. [Zugriff am 2021].

[64] A. Werner, „gsysc," 2021. [Online]. Available: https://github.com/werneazc/gsysc. [Zugriff am 2021].

[65] S. Kirkpatrick, C. D. Gelatt Jr. und M. P. Vecchi, „Optimization by Simulated Annealing," *Science,* Bd. 220, Nr. 4598, pp. 671-680, 1983.

[66] F. de Dinechin und B. Pasca, „Designing Custom Arithmetic Data Paths with FloPoCo," in *Design Test of Computers*, 2011.

[67] A. Kourfali und D. Stroobandt, „Superimposed In-Circuit Debugging for Self-Healing FPGA Overlays," in *19th Latin-American Test Symposium (LATS)*, Sao Paulo, Brazil, 2018.